الجمهورية العربية السورية

جامعة دمشق
كلية الهندسة المعلوماتية

المعهد العالي
للعلوم التطبيقية والتكنولوجيا

لغات البرمجة

الدكتور عمّار جوخدار الدكتورة غيداء ربداوي الدكتورة ندى غنيم

منشورات جامعة دمشق


وقل رب زدني علما
جامعة دمشق
Damascus University

بسم الله الرحمن الرحيم

مقدمة الكتاب

يتناول هذا الكتاب لغتي البرمجة C++ وJava معاً، لأنهما اللغتان الأكثر استعمالاً في التطبيقات العملية ومخابر الأبحاث، ولتوفر تقييس عالمي معروف لهما. تتميز هاتان اللغتان بكونهما غرضيتي التوجه، وهما بذلك تتبعان الاتجاه الغرضي التوجه الذي فرض نفسه في السنوات العشر الأخيرة في تحليل وتصميم البرمجيات.

بدأنا في هذا الكتاب بعرض لغة C++ كونها اللغة الأكثر شيوعاً واستعمالاً في العالم في المجال الصناعي والبحثي على حد سواء. يتوفر على الوِب العديد من البرامج المصدرية، بلغة C++، لخوارزميات هامة جداً في مجال الخوارزميات الهندسية والبيانية وغيرها. ويتوفر أيضاً مع لغة C++ مكتبة STL (Standard Template Library) لا تتوفر في غيرها من اللغات، وهي مكتبة خوارزميات، وهذا يجعلها مفيدة في تدريس الخوارزميات وتسهيل العمل البرمجي. يضاف إلى ذلك أن C++ هي مدخل هام جداً لتعلم لغة Java بسهولة، إذ اشتقت جميع بُناها الأساسية من لغة C++، وبذلك يكون تعلم Java بعد C++ أسهل للقارئ.

يتناول **الجزء الأول** من الكتاب ثلاثة مواضيع أساسية هي: لغة C++ باعتبارها تحسين للغة C ، وأساليب تعريف أنماط جديدة في C++، وأخيراً مفاهيم البرمجة الغرضية التوجه.

Damascus University

يتناول الجزء الثاني من الكتاب لغة Java، ونفترض هنا أن القارئ قد أتقن مفاهيم البرمجة الغرضية التوجه من خلال قراءته للجزء الأول، واستوعب تعليمات لغة C++ التي ورثتها Java، ولذلك نعرض للقارئ في الفصل الأول الفروق البسيطة بين التركيب النحوي للغة Java ولغة C++. في حين تتناول الفصول الأخرى من هذا الجزء السمات القياسية في لغة Java التي لا تتوفر في لغة C++ القياسية مثل برمجات جافا (java applets) والبيانيات (graphics) والبرمجة المقودة بالأحداث (event driven programming) وتعدد المسالك (multithreading).

نرجو أن تعمّ الفائدة من هذا الكتاب، والله ولي التوفيق.

د. عمّار جوخدار د. غيداء ربداوي د. ندى غنيم

دمشق 2000

الجزء الأول

C++

وقل رب زدني علما
جامعة دمشق
Damascus University


وقل رب زدني علما
جامعة دمشق
Damascus University

فهرس الجزء الأول

الفصل 1 : عموميات 2

1-1 مدخل 2

1-2 لمحة تاريخية 3

الفصل 2 C++ : تحسين للغة C 4

2-1 تذكرة بلغة C وبالمِقيَس ANSI 4

2-2 التعليقات 4

2-3 الثوابت 5

2-4 التصريح عن المتحولات وتعريفها 9

2-5 نمذجة التوابع 10

2-6 التحميل الزائد للتوابع 12

2-7 المعاملات الافتراضية 13

2-8 التوابع "السطرية" "INLINE" 15

2-9 تمرير المحدّدات بالمرجع 16

2-10 الأنماط المركبة 20

2-11 تحويل الأنماط 22

2-12 مؤثّر إبطال المدى 24

2-13 الحجز الديناميكي للذاكرة 25

2-14 التوابع العمومية 34

2-15 قنوات الدخل والخرج INPUT AND OUTPUT STREAMS 38

2-16 الكلمات المحجوزة 55

الفصل 3 : تعريف الأنماط الجديدة 57

3-1 مفهوم النمط المجرّد 58

3-2 التنفيذ (التحقيق) التقليدي لنمط مجرد في C++ 59

3-3 مفهوم الصف 63

3-3-1 المبادئ الأساسية 63
3-3-2 البوانى والهوادم 69
3-3-3 الصفوف المتداخلة 71
3-3-4 الصفوف ذات المعطيات الأعضاء من نمط صف 73
3-3-5 الصفوف العمومية generic classes 81
3-3-6 زيادة تحميل المؤثرات 88
3-4 تحويل الأنماط 124
3-5 مدى معيّنات الهوية 128
3-6 الثوابت غير الثابتة 131
3-7 فضاء الأسماء 132

الفصل 4 : البرمجة الغرضية التوجه في C++ 134

4-1 المفاهيم الأساسية 135
4-1-1 الغرض Object 136
4-1-2 الصف Class 137
4-1-3 الوراثة inheritance 137
4-1-4 تعددية الأشكال Polymorphism 141
4-1-5 الربط الديناميكي Dynamic binding 143
4-2 مِيْزات البرمجة الغرضية التوجه فيما يتعلق بتصميم البرمجيات 144
4-3 المفاهيم الغرضية التوجه في C++ 145
4-3-1 الوراثة البسيطة في C++ 145
4-3-2 تعددية الأشكال والربط الديناميكي 158
4-4 تنفيذُ صندوقِ أدواتٍ غرضي التوجه 162
4-4-1 الصفوف المجردة abstract classes 162
4-4-2 اللوائح الخطية المتباينة heterogeneous linear list 164
4-4-3 تركيبٌ أم وراثة 180
4-4-4 الشجرة الثنائية المتباينة 184
4-5 الوراثة المتعددة MULTIPLE INHERITANCE 201
4-6 إدارة الاستثناءات EXCEPTION MANAGEMENT 214

7-4 مداومة الأغراض (OBJECT PERSISTENCE) في C++ 225
8-4 المكتبة المِقيَس للقوالب STANDARD TEMPLATE LIBRARY 226

وقل رب زدني علما
جامعة دمشق
Damascus University


وقل رب زدني علما
جامعة دمشق
Damascus University

مقدمة

يتوجه هذا الكتاب إلى المبرمجين بلغة C الراغبين في تعلم لغة C++ انطلاقاً من معارفهم بلغة C، دون الغوص في الكتب المرجعية وأدلة استخدام C++ التي تزخر ببحر من المعلومات.

حاولنا في هذا الجزء من الكتاب أن ندخل المفاهيم التي تتميز بها لغة C++ تدريجياً، مع تزويد القارئ بعمات الأمثلة البرمجية. ينقسم محتوى هذا الجزء إلى ثلاثة أقسام :

- C++ : تحسين للغة C.
- تعريف الأنماط الجديدة.
- البرمجة الغرضية التوجه بـ C++ .

الفصل 1 : عموميات

1- 1 مدخل

يسمح هذا الكتاب للقارئ المتمكن من لغة البرمجة C بالمرور إلى لغة C++ و هي اللغة التالية لـ C. و C++ كما يدل اسمها هي لغة C التي طبق عليها معامل الإضافة "++" المعرف في لغة C القياسية. وقد حرص مؤلف اللغة على هذه التسمية بدلاً من أن يسميها D أو غير ذلك، ليدل على أن لغة C++ ليست إلا استمراراً لـ C وامتداداً لها نحو الأفضل.

C++ هي لغة برمجة حديثة، لذا، فهي متأثرة بمعظم الإضافات التي ظهرت في نظريات البرمجة. وهي من ثَم لغة غنية تعطي المبرمج إمكانات هائلة. و يُنصح بعدم محاولة تعلم هذه اللغة بتجريب جميع المفاهيم دفعةً واحدة. يربط الكثير من الناس بين البرمجة الغرضية التوجه ولغة C++، و هذا خطأ. إذ يمكننا البرمجة بـ C++ دون أي ذكر للأغراض، على عكس بعض اللغات الغرضية التوجه الصرفة، مثل إيفيل EIFFEL. وبهذا فإن C++ لغة هجينة بما لذلك من حسنات وسيئات. وقد تبين مع الوقت وجود أخطاء فادحة في لغة C، وهذا مما يجعل البرمجة فيها أمراً صعباً، ولاسيما إذا كان المبرمج مبتدئاً. وفي الحقيقة، فإن C هي لغة مجمَّع مطوَّرة، ومن ثَم، فهي تحتوي على عثرات يقع فيها العديد من المبرمجين. أما في لغة C++ ، فإن المترجم يقوم بالكثير من عمليات التحكم والتحقق، و لا سيما فيما يتعلق بالتوافق بين الأنماط، و تمرير الموسطات، ... الخ. وهكذا، يشعر المبرمج المعتاد لغةَ باسكال Pascal براحةٍ عند البرمجة بـ C++ إذ يشعر بأنه يتلقى الدعم من المترجم، على خلاف ما كانت عليه البرمجة بلغة C بالنسبة إليه.

تحتوي لغة C++ على تحسينات عديدة أُدخِلَت على لغة C، وهذا مما يجعلها تلتقي في الكثير من النواحي مع لغات كباسكال وأدا Ada. وقد كان للغة C++ أثر واضحٌ على تقييس

لغة C أي C بالتقييس ANSI وهو ما نسميه ANSI C.

1– 2 لمحة تاريخية

طور بيارن ستروستراب Bjarne Stroustrup لغة البرمجة C++ في مختبرات AT&T Bell في بداية الثمانينيات. وكما يقول مبتكر اللغة، فإن C++ تأخذ الكثير من C، إذ احتُفظ بلغة C كمجموعة جزئية من C++ للتسهيلات التي تقدمها في المستويات القريبة من الآلة، وهذا مما يسمح ببرمجة نظم التشغيل. وهكذا، فقد قام ستروستراب عام 1979 بتطوير لغة "C المزوّدة بالصفوف". وقد أعيد تشكيل هذه اللغة ما بين 1984/1983 فنتجت لغة C++ التي خضعت لبعض التحسينات الطفيفة فأصبحت C++ النسخة 1.2 عام 1987. انتشرت هذه اللغة انتشاراً واسعاً وأصبحت متوفرة على جميع بيئات التطوير، ثم طوّرت اللغة من النسخة 1.2 إلى النسخ 2.0 و 2.1 و 3.0.

في عام 1990 نشر ستروستراب مؤلفه "ARM" (Annotated C++ Reference Manual) أي مرجع لغة C++ الذي يعتبر الأساس في تقييس اللغة.

نلخص ماسبق بقولنا إن C++ هي تطوّر للغة C في ثلاثة مسارات أساسية جديدة وهي:

- احتواؤها على تحسينات واضحة مقارنة بلغة C التقليدية.
- دعمها لبناء واستكمال أنماط جديدة.
- احتواؤها على تقنيات البرمجة الغرضية التوجّه.

لابد أن يراعي تعلُّم لغة C++ هذا الترتيب. وسيشمل هذا الجزء من الكتاب هذه المحاور الثلاثة.

الفصل 2 : C++ تحسين للغة C

2– 1 تذكرة بلغة C وبالمِقيَس ANSI

قام Dennis Ritchie بتطوير لغة C عام 1972 في مختبرات Bell. ولغة C هي لغة برمجة عامة تتضمن بنى التحكم المألوفة (التكرارية، والاختيارية ، والتوابع والإجرائيات المعرفة من قبل المستثمر، ...) إضافة إلى احتوائها على مجموعة قوية من المؤثرات. وتعتبر ناقلية البرامج المكتوبة بلغة C من الميزات الهامة لهذه اللغة. وهنا لابد من الإشارة إلى وجود نسختين من لغة C وهما:

- لغة Kerninghan & Ritchie C
- لغة C ANSI

فرضت لغة Kerninghan & Ritchie C نفسها على الساحة عام 1987 مع ظهور كتاب المؤلفين حول اللغة "The C programming language"، وبقيت هذه النسخة متفرّدة على الساحة حتى عام 1989 حين ظهرت لغة C المُقَيَّسة أو ما يسمى C ANSI وفق المواصفة رقم ANSI X3.159 1984.

ويعتبر الانتقال من C إلى C++ أسهل على مستخدم لغة C ANSI مقارنة بمستخدم لغة C Kerninghan & Ritchie .

2– 2 التعليقات

سنبدأ بأبسط المستجدات وأكثرها ثانوية، وهي طريقة إدخال تعليقات وملاحظات في برنامج. يجري ذلك باستعمال علامة "//". يمكننا في برنامج بلغة C++ استعمال المحددات "/* */" الموجودة في لغة C والتي تسمح بكتابة تعليقات على عدة سطور متتالية.

يسمح المحدد // بتحديد تعليق يمتد على سطر وحيد، ويهمل المترجم كل ما يوجد على يمين المحدد // وعلى نفس سطره.

```
/ * This is a comment
in TWO Lines */
//  This is a comment in 1 Line
/* This type of  comment could include a comment
beginning by // and a comment */

/*  but no  /*  nested  */  */
```

وبسبب عدم إمكان وضع عدة مستويات متداخلة (nested) من التعليقات باستعمال /* */، فمن الأفضل استعمال المحدد // للتعليق على مجموعة تعليمات. والسبب في هذا التفضيل هو أنه قد يرغب المبرمج أثناء الاختبارات في تعليق (إبطال تنفيذ) مجموعة من التعليمات مؤقتاً. ويفضل في هذه الحالة تحديد هذه المجموعة من التعليمات باستعمال /* */ ، ولا يكون ذلك ممكناً إذا كانت هذه التعليمات تحتوي أصلاً على تعليقات محددة بـ /* */ . يؤدي هذا الأمر أحياناً إلى عدم التوافق بين C و C++.

```
x = a  //* we are doing a division */  b
```

يفهم مترجم C++ هذه التعليمة على أنها إسناد للقيمة a في المتحول x، على حين يولد مترجم C لها الرماز المكافئ للتعليمة:

```
x = a / b
```

يهدف هذا المثال إلى إظهار الصعوبات التي تطرأ عند توسيع لغة برمجة ما وإغنائها، مع الرغبة في الحفاظ على صحة ما كان موجوداً سابقاً.

2– 3 الثوابت

ألِف المبرمجون بلغة C استعمال المعالج المبدئي (الأولي) Preprocessor. وقد ثبت أن استعمال هذا المعالج يعتبر سبباً لأخطاء يصعب الكشف عنها. صمّمت لغة C++ بحيث يقتصر عمل المعالج المبدئي على بعض الحالات البسيطة كالترجمة الشرطية، وإحضار الملفات

لدمجها. وبهذا، مُنع استخدام المعالج المبدئي لتعريف الثوابت في برنامج. وتستعمل كلمة const لهذا الغرض.

تستخدم كلمة const لتعريف المترجم بأن الغرض الموصّف لا يمكن تعديله طَوال حيـــاة الغرض. من ثَم، فإن المترجم يقوم بجميع عمليات التحكم والمراقبة اللازمة، للتحقق من عدم تغيير الغرض المعني. يفيد هذا الأمر في زيادة قابلية قراءة البرنامج، وتسهيل صيانته. إضافة إلى ذلك، يمكن أن يكون الترميز الذي يولّده المترجم أكثر فعالية، لأن المترجم يعرف أن قيمـــة الغرض لن تتأثر طوال حياة البرنامج.

أمثلة:

```
const   int   i=7;
const   double  pi = 3.14159265353;
const    char  titre [ ] = "copyright xy";
i++ ;          // error!        خطأ يكشفه المترجم
titre[3] = 'a' ; // error!      خطأ يكشفه المترجم
```

يجب في C++ عند تعريف "متحول" على أنه ثابت، أن يعطي قيمة ابتدائية:

```
const   int  i;          // خطأ، يجب إعطاء قيمة
```

ويمكن استعمال const عند استعمال المؤشرات. في هذه الحالة يجب الانتبـــاه إلى الأثـــر المطلوب حدوثه إن كنا نريد أن يكون المؤشر ثابتاً، أو إذا كان المراد أن تكون القيمة المشار إليها ثابتة أو أن يكون كلاهما ثابتاً.

أمثلة:

```
const double   a= 8.15 ;          // 8.15 القيمة المشار إليها ثابتة
double  b= 3.14 ;
double* const  c = & b ;                    // المؤشر ثابت
double*  const d = & a ;          // a خطأ يمكن أن يقوم d بتغيير
```

لو كان السطر الأخير مسموحاً به، لأمكننا تعديل قيمة الثابت a بتعديل القيمة المشـــار

إليها بالمؤشر d ذلك أن تغيير محتوى الذاكرة المشار إليها بـــd غير ممنوع. في هذه الحالـــة d هي المحمية فقط.

```
b = 6.28 ;                          // 6.28 يصبح كذلك * c
* c = 123.0;                        // 123.0 يصبح كذلك * C
const double* const e = & b ;       // لا يتغير
b = 9.123 ;                         // 9.123 يصبح كذلك *e
*e = 123.0 ;          // خطأ: القيمة المشار إليها بـــ e ثابتة
e = C ;               // خطأ: المؤشر e ثابت
```

يلاحظ القارئ أن استعمال الكلمة const أقوى بكثير من استعمال كلمـــة define مـــع المعالج المبدئي. فالمترجم هنا يقوم بعدد من عمليات التحكم والمراقبة، ويكشف عن أخطـــاء دقيقة يصعب كشفها لاحقاً.

من الضروري ملاحظة الفرق بين تصرف التعليمة const في برنامج C++ وفي برنامج بـــ C ANSI. ففي C++ تهدف const إلى تجنب استعمال define# التي يعالجها المعالج المبدئي، أي إن التصريح عن متحول ثابت لا يقتضي بالضرورة حجز ذاكرة. أمـــا في C ANSI فـــإن المتحول المصرح عنه كثابت يعامل كمتحول له قيمة لا يمكن تغييرها، ولذا هناك عملية حجز للذاكرة، وهذا يؤدي إلى أن الثابت نفسه لا يمكن أن يظهر في نسقين modules مختلفين مـــن نفس الوحدة التنفيذية. يجب إذن ألا يظهر الثابت أبداً في ملفات الترويسات في CANSI. أما في C++، فيشجع المبرمج على فعل ذلك لأن تعريف الثابت ينطبق على الملف الذي يُعـــرّف ضمنه.

مثال:

لدينا برنامج مؤلف من ملفين. هذا البرنامج صحيح بلغة C ANSI لكنه يعطي خطـــأ في C++ لأن PI غير معرَّف في "prog3a.cpp".

```
#include <stdlib.h>
const double PI= 3.141592654;
```

```
void calcule_1(const double);
int main() {
  calcule_1(PI);
  return EXIT_SUCCESS;
}
```

prog3.cpp

```
#include <stdio.h>
extern const double PI;
void calcule_1(const double r) {
  printf("\n%g",PI*r*r);
}
```

prog3a.cpp

لكي يعمل هذا البرنامج في C++ يجب تعريف الثابت في كـــلا الملفـــين أي اســـتبدال التصريح:

```
const double PI= 3.141592654;
```

بالتصريح التالي:

```
extern const double PI;
```

يمكن أن نقوم بما هو أفضل من ذلك ألا وهو وضع التصريح في ملف ترويسة.

هنالك مثال آخر تقليدي يبين الفرق في التصرف بين C++ و C ANSI وهو التالي:

```
int  main ( ) {
   const int max = 50 ;
   char chaine [max] ;
   //...
   return EXIT_SUCCESS;
}
```

إن ترجمة هذا البرنامج في مترجم لـــ C ANSI يعطي خطأ لأن C لا يقبـــــل اســـتعمال متحول const في وضع يتوقع فيه أن يرى تعبيراً ثابتاً، على حين البرنامج صحيح في C++.

هناك فرق آخر بين اللغتين، وهو أنه يجب في C++ أن يقوم المبرمج بإعطاء قيمة ابتدائيـــة للثابت عند تعريفه، أما في C ، فليس هذا ضرورياً، إذ كل ثابت لم يُعط قيمة ابتدائية يـأخذ القيمة 0 ضمنياً.

2- 4 التصريح عن المتحولات وتعريفها

داخل كتلة في C، يجب أن تكون تعليمات التصريح سابقةً لجميع التعليمات التي تحتوي تعابير، أي تكون قبل التعليمة التنفيذية الأولى. لم يعد هذا الشرط قائماً في C++. وتعطي هذه الخاصة للمبرمج ميْزة تعريف المتحول أقرب ما يمكن من الموضع الذي سيستخدمه فيه للمرة الأولى. يسمح هذا الإمكان عندها، على سبيل المثال، بإدراج تعريف متحول التحكم في التعليمة التكرارية for.

```
#include <stdio.h>
#include <stdlib.h>

int main() {
  for (int i = 1;i <= 10; i++) {
    int square = i*i;
    printf("%d\t%d\t",i,i*i);
    int cube = square * i;  // not  acceptable in C
    printf("%d\n",cube);
  }
  printf("%d",i);   // error : i doesn't exist here !!

return EXIT_SUCCESS;
}
```

ويصبح من الممكن التصريح في C++ عن متحول في التعبير الموجود ضمن الشروط if و switch وwhile و do while. لكن في هذه الحالة يجب إعطاء قيمة ابتدائية للمتحول. على سبيل المثال:

```
if (int  i = read-number ( ) ) {
// i is defined here with a value # 0
}
else  {
//  i  is defined here with a value = 0
}
// i  is not defined here !
```

لا يسمح باستعمال تعليمة تصريح عن متحول في أحد العناصر التابعة لبنية اختيار يُتوقع عدم المرور عليها أثناء التنفيذ. يمكن القيام بذلك شريطة أن تكون هذه التعليمة موجودة داخل كتلة داخلية (ضمن { }). مثال على ذلك

```
int ch = 2;
 switch (ch) {
        case 1 :
                int e = 4;
                printf("%d",e);
        default : printf("%d",ch);
 }
```

تحتوي هذه المجموعة من التعليمات على خطأ، لأنه إذا نُفّذت التعليمة الافتراضية (أي default) فإننا سنتجاوز إذن تعليمة التصريح عن e مع إعطائها قيمة ابتدائية.

إن التعديل التالي يجعل مجموعة التعليمات هذه صحيحة:

```
int ch = 2;
 switch (ch) {
        case 1 : {
                int e = 4;
                printf("%d",e); }
        default : printf("%d",c);
 }
```

2- 5 نمذجة التوابع

كما في لغة C، التوابع هي اللبنات التي يبني بها المبرمج برنامجه في C++. يتألف البرنامج في C++ من تابع واحد على الأقل اسمه *main*.

وقد اتّخذت لجنة التقييس قراراً يقضي بأن يعيد التابع *main* القيمة 0، وهذا مما يختصر لنا كتابة ذلك صراحة، وهو ما سنفعله من الآن وصاعداً.

في C ANSI يجب على المبرمج التصريح عن أي تابع قبل استعماله أول مرة. يعرّف التصريح عن التابع المترجمَ باسم التابع، ونمط القيمة المعادة، وعدد وأنماط مُحَدِّداته. يسمى هذا التصريح عن التابع باسم نموذج التابع *function prototype*. يتألف توقيع التابع *signature* (وهو ما نسميه باللغة المتداولة ترويسة التابع) من اسم التابع، ونمط مُحَدِّداته. في C ANSI، إذا لم نُصرِّح عن التابع قبل استعماله، يفترض المترجم أنه يعيد عدداً صحيحاً.

في C++ يجب أن يُعرَّف كل تابع أو يصرّح عنه قبل استعماله أول مرة.

نورد فيما يلي مثالاً يبين عدم التوافق بين C و C++:

```
# incude < stdio.h >
int main ( )
{
printf ("The square root of  2 equals: %8.6/n",sqrt(2)) ;
}
```

تجري ترجمة هذا البرنامج في C ANSI على وجهٍ صحيح. ولكن عند التنفيذ يعطي البرنامج النتيجة الخاطئة التالية:

```
The square root of  2 equals:  0.00000
```

في C++ تؤدي ترجمة البرنامج السابق إلى ظهور خطأ، إذ يشتكي المترجم من غياب نموذج التابع *sqrt*.

هناك مصدر آخر للأخطاء، وهو عدم الإفصاح عن قائمة المُوسِطات الصورية في النموذج. يعني هذا في C، أن التابع ليس له محدّدات أو له محدّد واحد أو أكثر من أنماط غير محددة، من ثَم يكون التحقق من صلاحية الأنماط فيما بين الموسطات الصورية والفعلية أمراً مستحيلاً.

مثال:

```
#include <stdio.h>

extern int Min();

int main() {
  printf("%i", Min(4,2));
}
```

prog 6.cpp

```
int Min(int a, int b) {
return (a<b) ? a : b;
}
```

prog 6a.cpp

تجري ترجمة هذا البرنامج دون خطأ في C وتكون نتيجة تنفيذه صحيحة.

لو ترجمنا البرنامج نفسه في C++، لأشار إلى خطأ، لأنه لا يجد تعريفاً للتابع *Min()* يتوافق مع التوقيع في الملف "Prog 6.cpp"، ذلك أن التوقيع الموجود فيه يكافئ في الحقيقة extern int Min (void).

يعني غياب الموسطات الصورية في C++ أن التابع لا يقبل موسطات. وهذا الفرق هام جداً بين C و C++ في هذه النقطة.

نلاحظ كذلك في C++ أن التابع الذي يُعرَّف على أنه يعيد قيمة، يجب أن يعيد قيمة. وبتعبير أوضح، فإنه لا يسمح باستعمال التعليمة *return* في جسم التابع دون تحديد القيمة المعادة.

2- 6 التحميل الزائد للتوابع

يسمح في البرنامج نفسه في C++ أن يجري تعريف تابعين أو أكثر لهما الاسم نفسه. يفترض هذا ضمناً أن هذه التوابع تختلف فيما يتعلق بقائمة موسطاتها الصورية. وتقع على عاتق المترجم مهمة تحديد التابع الواجب استدعاؤه بناءً على الموسطات الفعلية المستخدمة عند استدعاء التابع.

يسمح هذا الإمكان بتسهيل قراءة البرامج، إذ لا يطالَب المبرمج في كل مرة باختراع اسم جديد لتابع يقوم بعمل مطابق لتابع آخر، لكن على معطيات من نمط مختلف. إن إمكان زيادة تحميل التوابع يسمح بإعطاء صفة تعددية الأشكال للعمل، يحدد الشكل حسَب نمط الموسطات وعددها عند الاستدعاء.

```
#include <stdio.h>

void display(int i) {
  printf("integer %d\n",i);
}
void display(double i) {
  printf("double %g\n",i);
}
int main() {
  display(7);
```

```
  display(3.5);
  return 0;
}
```

يعطي تنفيذ البرنامج:

```
integer 7
double 3.5
```

يتبين لنا هنا أن المترجم قام في كلتا الحالتين، بالاختيار الصحيح للتابع الواجب استدعاؤه. ولكن يجب الانتباه طبعاً إلى عدم التسبب في حالات التباس لا يستطيع فيها المترجم الاختيار بدلاً عن المبرمج.

مثال:

```
#include <stdio.h>
void display(int i) {
  printf("integer %d\n",i);
}
void display(double i) {
  printf("double %g\n",i);
}
int main() {
  long int i = 1;
  display(7);
  display(3.5);
  dispplay(i);  // Error ... compilation is ambiguous !!
  return 0;
}
```

في هذا المثال، يقع المترجم في حيرة، إذ أنه لا يعرف: أيحوّل المتحول i ذو القيمة 1 والنمط long ضمنياً إلى int أم إلى double؟ علماً بأن كلتا الحالتين ممكنتان. يجب إذن استعمال زيادة التحميل مع الحذر من الوقوع في مثل هذه الحالات.

2- 7 المعاملات الافتراضية

عندما يكتب المبرمج توابعَ تستخدم موسطات، يمكن أن يكون لبعض هذه الموسطات قيمٌ اعتيادية. في هذه الحالة، يمكن تعريف تابع بمحدّدات ذات قيم افتراضية. يسرّع هذا الأمـــر تنفيذ استدعاء التابع لأن هذه المحدّدات لا تحتاج إلى أن تُنسخ. لهذا السبب، جعلت المحدّدات

التي تأخذ قيماً افتراضية على اليمين في توقيع التابع. على سبيل المثال:

```
void init (int a , int b  = 0) ;     // 2nd argument = 0 by default
```

في هذه الحالة، init (3) و init (2,4) هي أشكال صحيحة لاستدعاء التابع المصرّح عنه.

إن المثال التالي خاطئ لأن المحدّدات الافتراضية يجب أن تكون متوالية اعتباراً من أقصى اليمين نحو اليسار.

```
void incorrect (int a =3, int b, int c = 0) ; // error , b has no value
```

إذا جرى التصريح عن تابع مع محدّدات افتراضية، فلا يصح عند تعريف التابع تحديد هذه القيم الافتراضية مرة أخرى.

```
void init (int a = 0) ;  // declaration of init
void init (int a = 0) {/* body   */ } //definition of init  ERROR


void init (int a = 0) ; // declaration of init
void init (int a ) { / * body * / }  OK
```

بالطبع، يسمح بالتعريف الزائد للتابع ذي المحدّدات الافتراضية. في هذه الحالة يجب التنبه إلى عدم تحديد القيم الافتراضية مرة أخرى عند إعادة التعريف.

```
#include <stdio.h>
void init(int,int=3);
void init(int a,int b) { printf("%d,%d\n",a,b); }

void init(int=7,int);  // overdefinition

int main() {
  init(2,1); // displays 2,1
  init(4);   // displays 4,3
  init();    // displays 7,3
  return 0;
}
```

عند التعريف الزائد للتابع *init ()* هنا لم يكن يسمح بإعادة تعريف القيمة الافتراضية للمحدّد الثاني. نلاحظ هنا أنه يجب ألا يذكر اسم المحدّد الذي يأخذ قيمة افتراضية.

2– 8 التوابع "السطرية" "Inline"

اعتاد المبرمجون بلغة C استعمالَ المعالج المبدئي لتعريف الماكرو التي يُعرَف تفصيلها قبل الترجمة مباشرة. إن المساوئ الناتجة عن استخدام الماكرو عديدة. ذلك أنه قد يكون من الصعب أحياناً كتابة الماكرو على وجهٍ صحيح، ولا يمكن كشف الأخطاء فيها إلا بصعوبة باستعمال أدوات التطوير التقليدية، ولا يستطيع المترجم أيضاً أن يقوم بأي تحكم للتحقق من صلاحية أنماط المحدّدات، إضافةً إلى عدم انسجام الماكرو مع بقية البرنامج، فالبرنامج لا يستطيع أن يمرر الماكرو كمحدّد، ....

لنأخذ مثالاً تقليدياً على الماكرو، وهو الماكرو الذي يحسب العدد الأكبر بين عددين صحيحين. في المثال التالي سنقارن الماكرو بالتوابع السطرية التي تعرف بالكلمة المحجوزة *.inline*

يبين هذا المثال صعوبات الاستعمال الصحيح للماكرو والناتجة عن الاستبدالات النصية التي يقوم بها المعالج المبدئي. من الواضح أن الماكرو يعمل في الحالات البسيطة، ولكن يجب على المبرمج أن يرهق نفسه ليعْرف الحالة التي هو بصددها: أهي بسيطة أم لا؟

```
#include <stdio.h>

#define max(x,y) (x > y) ? x : y

inline int Max(int x, int y) { return (x > y ? x : y); }

int main() {
  int a = 4;
  int b = 7;
  printf("%d,%d\n",a,b);          // displays 4,7
  printf("%d\n",max(a++,b++));    // displays 8  !!
  printf("%d,%d\n",a,b);          // displays 5,9 !!

  a = 4; b = 7;
  printf("%d,%d\n",a,b);          // displays 4,7
  printf("%d\n",Max(a++,b++));    // displays 7  !!
  printf("%d,%d\n",a,b);          // displays 5,8 !!

  return 0;
}
```

عندما يعرف المبرمج تابعاً سطرياً، فإنه يرغب في الابتعاد عن قيام المترجم عند كل استدعاء للتابع بإجراء استدعاء تقليدي له، بل يريد منه أن يضع في هذا الموضع الرماز الموافق لهذا التابع. يفيد تعريف التابع سطرياً في زيادة سرعة تنفيذ البرنامج، وذلك بتجنب العمليات المعتادة المجراة عند استدعاء التابع وانتهائه. ولكن تعريف تابع على أنه سطري ***inline*** قد يوقع في خطر زيادة حجم الرماز المولد، ومن ثَم عدم الحصول على هذه النتيجة المرغوبة.

في هذه الحالة يفضل انتقاء التوابع السطرية بحيث تكون "صغيرة" مكونة من تعليمة واحدة أو بضع تعليمات.

يصنّف هذا الإمكان ضمن مجموعة التحسينات التي تروق مبرمجي لغة C. فإذا كان المبرمج مزوّداً ببرنامج لقياس الأداء، وأظهرت النتائج ضرورة تسريع تابع صغير، يمكن عندها إضافة كلمة *inline* في بداية سطر تعريفه دون حاجة إلى تغيير أي شيءٍ آخر في البرنامج.

2-9 تمرير المحدّدات بالمرجع

في C، يوجد نمط وحيد لتمرير الموسطات، وهو التمرير بالقيمة ***by value***. إن هذه الطريقة هي أكثر الطرق أماناً لتمرير محدّدات لتابع دون أن يتمكن من تعديلها. في الواقع، عند استدعاء التابع، تحجز المساحة اللازمة من الذاكرة لمحدّدات التابع آلياً في مكدّس التنفيذ. ومن ثَم تنسخ قيم الموسطات الفعلية في هذه المساحة المخصصة للمحدّدات على المكدّس. وهكذا فإن التابع لا يعمل أبداً مباشرة مع الموسطات الفعلية، بل على نسخة منها. وهذا ما يفسّر كون التابع لا يستطيع تغيير قيم الموسطات الفعلية لدى التوابع المستدعية لهذا التابع.

غير أن هناك حالات يجب أن يقوم فيها التابع بتعديل موسط أو أكثر. وهذا يحصل على وجهٍ تقليدي في تابع يعيد عدة قيم، أو يحتاج إلى محدّدات من حجم كبير (مثل التسجيلات،...).

سنرى في المثال التالي تابعاً يبادل بين محتوى متحولين. إن البرنامج التالي خاطئ لأن القيم التي يجري تبديلها هي فقط القيم الموجودة على مكدّس التنفيذ، وليست الموسطات الفعلية.

```
#include <stdio.h>

void swap(int x, int y) {
  int tmp = x; x = y; y = tmp;
}

int main() {
  int a = 2;
  int b = 3;
  printf("\na=%i, b=%i",a,b);
  swap(a,b);
  printf("\na=%i, b=%i",a,b);
}
```

يعطي تنفيذ هذا البرنامج النتيجة التالية:

```
a=2 b=3
a=2 b=3
```

في مثل هذه الحالة يمكن للمبرمج بلغة C أن يقوم بالتبديل باستخدام المؤشرات. فهو يمرر للتابع swap() مؤشرين إلى المتحولين المراد تبديلهما (المؤشر هو عنوان المتحول). عند الاستدعاء يجب على المبرمج التنبه إلى إرسال عناوين المتحولات المعنية بالتبديل،لا قيمها!

```
#include <stdio.h>

void swap(int* x, int* y) {
  int tmp = x; x = y; y = tmp;
}

int main() {
  int a = 2;
  int b = 3;
  printf("\na=%i, b=%i",a,b);
  swap(&a,&b);
  printf("\na=%i, b=%i",a,b);
}
```

يعطي تنفيذ هذا البرنامج النتيجة التالية:

```
a=2 b=3
a=3 b=2
```

وهي نتيجة صحيحة. هذه إذن طريقة للتخلص من المشكلة، ولكنها ليست أنيقة.

تتيح C++ للمبرمج خياراً بين طريقتين لتمرير الموسطات، وهما: التمرير بالقيمة *by value*

– وهو النمط الافتراضي، والتمرير بالمرجع *by reference*. وهكذا، يصبح لدى C++ أنماط التمرير نفسها الموجودة في Pascal أو Cobol-85.

إذا ما جرى تمرير وسيط بالمرجع أثناء استدعاء التابع، فهذا يعني في الواقع أن عنوان الموسط الفعلي هو الذي يمرّر للتابع. و من ثَم فإن التابع يتعامل فعلياً مع الموسط الفعلي! يشير المبرمج إلى نمط التمرير أثناء كتابته لتوقيع التابع، إذ يُتبع نمطَ الموسط الذي يريد تطبيق آلية التمرير بالمرجع عليه، الرمزَ "&". وداخل جسم التابع يجري التعامل مع الموسطات الصورية كمتحولات عادية. عند الاستدعاء، لا يتميز الموسط الفعلي الممرّر بالمرجع عن موسط ممرّر بالقيمة.

```
#include <stdio.h>

void swap(int& x, int& y) {
  int tmp = x; x = y; y = tmp;
}

int main() {
  int a = 2;
  int b = 3;
  printf("\na=%i, b=%i",a,b);
  swap(a,b);
  printf("\na=%i, b=%i",a,b);
}
```

يعطي تنفيذ هذا البرنامج النتيجة التالية:

```
a=2 b=3
a=3 b=2
```

إن اسم الموسط الصوري الموافق للمحدّد الفعلي الممرّر بالمرجع هو إذن "اسم مستعار" للمحدّد الفعلي. وهكذا، ففي هذا المثال ، x هو اسم ثان لـ a و y هو اسم ثان لـ b، وهذا يعني أن x و y يعنيان a و b أينما وردا في التابع ( )swap.

عند استدعاء تابع، قد يكون المحدّد الفعلي الممرّر بالقيمة تعبيراً، على حين في حالة التمرير بالمرجع لا يمكن أن يكون المعامل إلا موصلاً لمتحول. في البرنامج السابق لا معنى لاستدعاء التابع بالشكل التالي: swap (2*a ,b).

يستعمل التمرير بالمرجع غالباً للحصول على المزيد من الفاعلية، وذلك عندما نريد تمرير محدّدات "ضخمة الحجم" لتابع. في هذه الحالة إذا كان التابع لا يغيّر المحدّد الممرّر بالمرجع، فمن الأفضل الإشارة بذلك إلى المترجم وذلك بإعلامه أن المحدّد هو ثابت. يزيد هذا الأمر من سهولة القراءة، ويُعلِم قارئ البرنامج بوضوح عن الهدف المنشود.

```
#include <stdio.h>

struct Person {
  char name[30];
  char address[40];
  int age;
};

void init(Person& p) {
  printf("\nEnter the name : "); gets(p.name);
  printf("Enter the address : "); gets(p.address);
  printf("Enter  the age : "); scanf("%d",&p.age); getchar();
}
void display(const Person& p) {
  printf("\n%s ,  %i years old , lies in %s",p.nom,p.age,p.adresse);
}
int main() {
  Person john, smith;

  init(john); init(smith);
  display(john); display (smith);
}
```

في الإجرائية ( ) init يمرّر المحدّد بالمرجع، إذ تغير الإجرائية قيمة المحدّد. على حين تستعمل الإجرائية ()display التمرير بالمرجع ولكن بغية تحسين سرعة التنفيذ فقط، ولهذا السبب يعرّف المحدّد كثابت. وهكذا تتسبب كل محاولة يقوم بها المبرمج لتغيير قيمة المحدّد داخل الإجرائية بحدوث خطأ أثناء الترجمة.

توجد ملاحظة أخرى في هذا البرنامج. ففي السطر الذي يعرف المتحولين John و Smith يلاحظ القارئ عدم وجود الكلمة المحجوزة struct كما في C. إن تعريف البنية هو إذن تعريف لنمط جديد. وسنصطلح على أن جميع أسماء الأنماط التي سنعرفها في برامجنا تبدأ بحرف كبير.

2- 10 الأنماط المركبة

إلى جانب الأنماط المعرَّفة سابقاً، يمكن للمبرمج تعريف أنماط جديدة، تماماً كما في C، وذلك بتعريف سجلات أو بُنى أو أنماط معدودة أو اجتماع أنماط. ولكن في C++، على العكس من C تكون معيِّنات الهوية وأسماء البنى والاجتماع والأنماط المعدودة في فضاء الأسماء نفسه.

```
struct Employee {
  long int matricule;
  char* name;
  char* address;
};

int main() {
  Employee john, smith;
// ...
}
```

في هذا المثال، نرى أن تعريف البنية Employee هو تعريف لنمط جديد قام به المبرمج، ويمكنه إذن تعريف متحولات من هذا النمط بتحديد اسم النمط بكل بساطة. لم نعد بحاجة إلى ذكر أننا بصدد struct في كل مرة. نشير هنا إلى أنه لأسباب تتعلق بالتوافقية، تبقى رموزC مقبولة في C++.

إن ما ذكرناه آنفاً من أجل struct هو أيضاً صحيح من أجل union. الجديد في C++ هو إمكان تعريف union مغفل الاسم (anonymous)، أي إنه لا يعرّف نمطاً جديداً ولا حقلاً جديداً أو متحولاً. هذا الإمكان مفيد في حالة تعريف union داخل سجل مثلاً.

```
#include <stdio.h>

struct Employee {
  long  matricule;
  char* name;
  char* address;
  char marital_status;          // coded
  union {
    char date_divorce[6];       // if divorced, date of divorce
    char date_death_conjoint[6]; // if widower, date of death of conjoint
    struct {                    // if married,
```

```
    char* name_conjoint;      // name of conjoint
    char date_marriage[6];    // date of  marriage
  } m;
 };
};

int main() {
 Employee john, smith;
 //...
 switch (john. marital_status) {
 case 'M' :
  printf("is married with  %s the %s",paul.m. name_conjoint, paul.m.date_marriage);
  break;
 case 'S' :
  printf("divorced from  %s",paul.date_divorce);
  break;
 case 'D' :
  printf("conjoint dead  the %s",paul.date_death_conjoint);
  break;
//...
 }
 //...
}
```

يوضّح هذا المثال أيضاً أن union فقط يمكن أن تكون مغفلة الاســـم، دون عـــن struct. وعندما يشمل مجال التعامل مع union المغفلة الاسم كاملَ الملف، يجب التصريح عنها بأنهـــا static.

وكما في حالة التسجيلات العادية والتسجيلات ذات الاحتمالات المختلفة، فإن التصريح عن نمط معدود يسمح بتعريف نوع جديد.

```
enum Boolean { FALSE, TRUE };
int main() {
 Boolean found = FALSE;
 found = 0;          // incorrect !
 found = TRUE;
// ...
}
```

في C++ كما في CANSI، يجب أن تكون القيم المعدودة غير مكرّرة. وبشكل ضمـــني، يكون ترتيب القيمة الأولى في العناصر هو 0 ، ويكون ترتيب القيم التالية هو 1 مضافاً إليـــه

ترتيب العنصر السابق. في هذه الحالة، يكون ترتيب TRUE هو 1.

يُمنع في C++ إسناد عدد صحيح إلى متحول من النمط *enum*، كما يمنع أن تُسند إليه قيمة من نمط معدود آخر، في حين هذه العمليات جائزة في C ANSI. نؤكد من جديد أن لغة C++ هي أكثر صرامة على مستوى التحقق من الأنماط. وهذا لا يعني أن C++ تقدم إمكانات أقل من CANSI، إذ يمكن للمبرمج أيضاً أن يُسند عدداً صحيحاً إلى متحول من النمط *enum*، ولكن يجب عليه أن يبين رغبته بوضوح للمترجم بإجراء تحويل للأنماط.

سنستغل الفرصة هنا لإدخال النمط الجديد المعرَّف سابقاً *bool*. أُدخل هذا النمط، بوجهٍ خاصٍ لوضع نهاية لفوضى تعاريف هذا النمط الأساسي. في الحقيقة، لا يمكن الاستغناء عن استعمال النمط المنطقي في كل تطبيق ولا في كل مكتبة. ولما كان يوجد عشرات الطرائق المختلفة لإنفاذ مثل هذا النمط، فإن هذا يؤدي إلى العديد من المشاكل، خاصةً إذا استعملنا مكتبات برامج قد كتبها آخرون غيرنا. لا تحتوي بعض المترجمات هذا النمط الجديد. من أجل ذلك، نعرف النمط المنطقي مع إعطاء تسميات ومعيّنات هوية مطابقة لما هو موجود في اللغة المِقيَس.

```
enum bool { false, true } ;
```

bool و false و true هي كلمات ثلاث جديدة محجوزة. تعيد مؤثرات المقارنة نتيجة منطقية من النمط bool. يمكن لمتحول bool أن يُرقّى نحو int ويُحوّل int ضمنياً إلى bool، إذا كان ضرورياً. وهكذا، يمكن تحويل قيمة مختلفة عن 0 ضمنياً إلى true ويمكن لـ true أن تتحوّل ضمنياً إلى 1. الأمر نفسه يسري على 0 و false.

2-11 تحويل الأنماط

لمّا كان التحكم بأنماط المعطيات ومراقبتها هي في C++ أكثر صرامةً منها في C، فإنه يجب على المبرمج، في بعض الظروف، أن يقوم بتحويلات صريحة للنمط، إضافةً إلى التحويلات الضمنية للأنماط التي يقوم بها المترجم آلياً.

يمكن استخدام نفس الرموز المستخدمة في C، ولكن C++ تقدم بالإضافة إلى ذلك رمزاً

تابعياً. وفيما يلي الرمزان المتوفران:

(type) expression كما في C

type (expression) الترميز التابعي، الصالح في C++

يظهر المثال التالي المجموعة الكاملة لمحارف الحاسوب على الشاشة، بإجراء تحويل ضمني للأنماط (لا داعي له) في التنسيق المسموح به في C++.

```
#include <stdio.h>

int main() {
   for (int i = 0; i <= 255; i++) printf("\n%i --> %c",i, char(i));
}
```

تجدر الإشارة هنا إلى أن الرمز التابعي لا يطبّق إلاّ في حالة الأنماط البسيطة مثل long، int، short، char، ... وأسماء لأنماط موصوفة.

ولما كان لغة C++ لغة شديدة التنميط، فإنه يجب على المبرمج أن يلجأ أحياناً إلى هذا المفهوم. يكون التحويل الصريح للنمط مفيداً بوجه خاص عند العمل مع مؤشرات عمومية من النمط *void، أو عند استخدام المفاهيم الغرضية التوجه التي سنراها لاحقاً. إن التحويل الصريح للنمط وسيلة متاحة للمبرمج، تمكّنه من الوصول إلى غاياته، مع أنه يعمل مع مترجم يقوم بمراقبة صحة الأنماط مراقبة صارمة.

أما فيما يخص تحويلات النمط الضمنية، فمن المفيد أن نذكّر أنها تجري آلياً بواسطة المترجم. فعندما نسند قيمة إلى متحول، فإن القيمة تتحول إلى نمط المتحول إذا كان ذلك ممكناً. وبنفس الطريقة، فإنه عند استدعاء تابع، يجري تحويل المحدّد الفعلي إلى نمط المحدّد الصوري إذا كان ذلك ممكناً بالطبع. يصح هذا التحويل خاصة في حالة الأنماط الرقمية المعرّفة مقدّماً. سنرى لاحقاً أنه يمكن للمبرمج تعريف أنماط جديدة وبناء توابع التحويل الضرورية.

في التعابير الحسابية يعتمد التحويل الضمني للنمط الذي يقوم به المترجم، على تحويل القيم التي يجري تداولها إلى النمط "الأكبر" المستخدم في التعبير. يوضّح المثال التالي هذا الإجراء:

```
int i = 10;
i = i* 1.12 ;
```

بعد تنفيذ هاتين التعليمتين، i تساوي 11 لا 10 لأن قيمة i على يمين رمز مؤثر الإسناد في التعليمة الثانية تتحول إلى double. تعطي عملية الجداء 11.2، ثم تقطع هذه النتيجة لتصبح عدداً صحيحاً int.

إن تغيير نمط أحد المعطيات هو من أكثر العمليات حساسية، وهو غالباً مصدر لأخطاء يصعب كثيراً كشفها. يمكن للمبرمج، مع الترميزين اللذين رأيناهما آنفاً، أن يكبّر قيمة عددية أو أن يحد منها، وأن يحذف توصيفات const، وأن يعيد تفسير قيمة على وجهٍ مختلفٍ تماماً (عبر المرور من نمط إلى آخر مختلف تماماً).

2- 12 مؤثّر إبطال المدى

يسمح مؤثّر إبطال المدى "::" بالنفاذ إلى معيّن هوية عمومي جرى تعريفه خارج أي تابع، ثم جرت مقاطعة مداه من قبل معيّن هوية مطابق في كتلة داخلية.

يسمح معيّن الهوية المسبوق بالمؤثّر "::" بالنفاذ إلى معيّن الهوية العمومي. وفيما يلي مثالٌ على ذلك:

```
#include <stdio.h>

int i = 3;    // i here is  external for main, so global

int main() {
  int i = 6;  // i here is local in main
  printf("\ni = %i",i);
  printf("\ni = %i",::i);
  {   // local  bloc in main
    int x = i;
    int i = 5;  // this i is local in the local  bloc in main
    printf("\n  i = %i",i);
    printf("\n  i = %i",::i);
    printf("\n  x = %i",x);
  }
  printf("\ni = %i",i);
  printf("\ni = %i",::i);
  return 0;
```

```
}
```

عند التنفيذ نحصل على:

```
i=6
i=3
  i=5
  i=3
  x=6
i=6
i=3
```

من وجهة نظر منهجية، لا ينصح باستعمال المؤثر "::" . ويدل استعماله على أسلوب سيئ في البرمجة. في الحقيقة سيتبين لنا الأهمية الكبيرة لهذا المؤثر في إطار آخر سنراه لاحقاً.

2- 13 الحجز الديناميكي للذاكرة

يكوِّن الحجز الديناميكي للذاكرة جزءاً أساسياً من لغة C++. وتقدّم اللغة مؤثرين هما *new* و *delete* لاستبدال التابعين ( ) *malloc* و ( ) *free* الموجودين في مكتبة لغة C.

يجب أن يتبع المؤثر الأحادي *new* بمعيِّن للنمط. ويتسبب هذا المؤثر عند التنفيذ بحجز كتلة متصلة من الذاكرة من حجم كاف يسمح باحتواء قيمة من النمط المحدّد. يُعاد عنوان هذه الكتلة من الذاكرة كنتيجة للتعبير *new*. ولما كان يمكن أن يفشل طلب الحجز هذا، فقد وُضعت آلية لمعالجة الأخطاء. وهكذا يستطيع المبرمج أن يعرّف ما يسمّى بـ *new_handler* وهو تابع يقوم بإدارة الأخطاء، ويتدخل عندما لا يستطيع المؤثّر *new* أن يجد كتلة من الحجم المطلوب.

وقد جرى تغيير المعامل *new* بحيث يؤدي عند فشله إلى إقلاع استثناء (exception) - وهو مفهوم سنراه بالتفصيل في الفصل القادم- من النمط *xalloc* . في السابق، وقبل أن يعمّم مفهوم معالجة الاستثناءات في C++ كان المؤثّر new يعيد القيمة 0 إذا فشل طلب الحجز . وللمحافظة على هذا التصرّف، يستطيع المبرمج تعريف new_handler عنوانه 0 (كما سنراه لاحقاً).

أمّا المؤثّر الأحادي *delete*، فإنه يأخذ كمحدّد مؤشّراً، وتكون قيمة المؤشر 0 أو يكون

ناتجاً عن استدعاء سابق للمؤثّر *new*. تُحرّر كتلة الذاكرة المشار إليها بهذا المؤشّر، أي تُعاد إلى النظام لإعادة استخدامها. إن إجراء عملية delete على مؤشّر معدوم ليس بذي نتيجة. هذا ويجب ألاّ يستخدم المؤثّر *delete* إلاّ في تحرير ذاكرة أسندت ديناميكياً بالمؤثّر *new*.

وهكذا، لا يوجد في C++ آلية للاسترداد الآلي للذاكرة، بل يجب على المبرمج القيام بهذه المهمة المزعجة، ولاسيما في حالة بنى المعطيات المعقّدة. هذا وتعتبر الإدارة السيئة للذاكرة الديناميكية مصدراً هاماً لأخطاء يصعب كشفها وتحديدها، إذ لا تظهر إلا بعد مدة طويلة من الاستخدام، ولا تتضمن أيّة إشارة تساعد المبرمج على حلّها.

وفيما يلي مثال على استعمال *new* و *delete*:

```
int *  i = new  int.
*i = 2;
delete i ;
```

يوضّح المثال التالي كيفية إدارة سلسلة من الأعداد الصحيحة، علماً بأن عناصر السلسلة تُنشأ ديناميكياً أثناء التنفيذ.

إن عناصر السلسلة هي بنى من النمط *Item* تتألّف من حقل اسمه *next* يؤشّر إلى العنصر التالي، من العدد الصحيح *val*. ومن ثَم، فالسلسلة هي مؤشّر إلى عنصر من النمط *Item*.

تُعرّف على السلسلة عمليات الإدراج في رأسها، وإظهار محتواها، وحذف عنصر منها، إضافةً إلى تابعين منطقيين أحدهما يبين إذا ما كانت السلسلة فارغة، والآخر إذا ما كان عنصر ما موجوداً ضمن عناصرها. إضافةً إلى هذه العمليات التقليدية، يجب أن نقوم ببناء عمليتين خاصّتين أولاهما هي البناء الصحيح للسلسلة، وثانيتهما هدمها وإزالتها.

إن بناء السلسلة هي عملية ضرورية بسبب خصوصية تمثيل السلسلة في الذاكرة. كما أن عملية الهدم ضرورية جداً إذا ما كانت السلاسل تُنشأ محلياً خلال عمليات استدعاء التوابع، لأنها تضمن الاستعادة الكاملة للذاكرة المحجوزة للسلسلة في نهاية تنفيذ التابع.

```
typedef int Type;
struct Item {
  Type val;
  Item* next;
```

```
};
typedef Item* List;
enum bool { false = 0, true };

void init(List&);              // initialize the  list
void insert(List&, Type);      // insert one element in the head of the list
void display(List);            // displays the list
void destruct(List&);          // destructs the list
void eliminate(List&, Type);   // eliminates x from the  list
bool is_empty(List);
bool exist(List, Type);
```

list.h

تستعمل بعض هذه التوابع مفهوم التمرير. وفيما يلي برنامج يستعمل النمط List:

```
#include <stdio.h>
#include "list.h"

int main() {
  List list;
  Type i;

  init(list);
  printf("\nEnter your integers (0 = stop) : ");
  scanf("%i",&i);
  while (i) {
    insert(list,i);
    scanf("%i",&i);
  }

  /* display the list */
  if (!is_empty(list))
    display(list);
  else
    printf("\nEmpty list!");

  /* find integers in the  list */
  printf("\nSearch in the list, enter an integer, 0 for end ");
  scanf("%i",&i);
  while (i) {
    if (exist(list,i))
          printf("%i is in the list",i);
    else
          printf("%i is not in the list",i);
    printf("\nSearch in the list, enter an integer, 0 for end ");
    scanf("%i",&i);
```

```
};

/* delete elements from the list */
printf("\nDeletion from list, enter an integer 0 for end ");
scanf("%i",&i);
while (i) {
  eliminate(list,i);
  display(list);
  printf("\nDeletion from list, enter an integer 0 for end ");
  scanf("%i",&i);
}

/* destruction of the list */
printf("\nDestruction of the  list");
destruct(list);
if (!is_empty(list))
  display(list);
else
  printf("\nEmpty list !");
 return 0;
}
```

يعطي تنفيذ هذا البرنامج ما يلي :

```
Enter your integers (0 = stop) :  1 2 3 4 0
(4 3 2 1)
Search in the list, enter an integer, 0 for end  2
2  is in the list
Search in the list, enter an integer, 0 for end 5
is not in the list
Search in the list, enter an integer, 0 for end 0

Deletion from list, enter an integer 0 for end 6
The element 6 is not in the list!
(4 3 2 1)
Deletion from list, enter an integer 0 for end 3
(4 2 1 )
Deletion from list, enter an integer 0 for end 2
(4 1)
Deletion from list, enter an integer 0 for end 0

Destruction of the  list
Empty list !
```

إن تعريف التوابع السابقة لا يبدي صعوبة. فالتابع (*insert* ينشئ عنصراً جديداً ويملؤه بالعدد الصحيح الموجود في المحدّد، وبعنوان العنصر الأول في السلسلة الحالية، ويصبح هذا

العنصر الجديد هو رأس السلسلة. تجدر الإشارة هنا إلى أن هذه الآلية لا تعمل ما لم تكـــن السلسلة قد هُيّئت سلفاً بالتابع *init*( ).

```
#include <stdio.h>
#include "list.h"

void init(List &l) { l = 0; };

bool is_empty(List l) { return l == 0 ? true : false; }

void inset(List &l, Type x) {
  Item* p = new Item;
  p->val = x;
  p->next = l;
  l = p;
}

bool exist(List l, Type x) {
  if (l == 0) return false;
  Item* p = l;
  bool found = false;
  while ((p) && (!found))
    if (p->val == x)
                    found = true;
    else
      p = p->next;
  return found;
}

void display(List l) {
  printf("( ");
  for (Item* p = l; p; p = p->next)
    printf("%i ",p->val);
  printf(")");
}

void destruct(List &l) {
List p = l;
List tmp = l;
  while (tmp) {
    tmp = p->next;
    delete p;
    p = tmp;
  }
  l = 0;
```

```
}

void eliminate(List &l, Type x){
 if (l == 0) {
  printf("\nEmpty list!");
  return;
 }
 // search for the element to delete
 List p = l;
 List pred = 0;
 bool found = false;
 while ((p) && (!found))
  if (p->val == x)
    found = true;
  else {
    pred = p;
    p = p->next;
   }
 if (!found) {
  printf("The element %i  is not in the list!\ n",x);
  return;
 }
 else { // delete the found element pointed by p
  if (pred) { // the element to be deleted has  a predecessor
    (pred->next) = (p->next);
    delete p;
   }
   else { the element to be deleted is the first  of the list
    l = (p->next);
    delete p;
   }
 }
}
```

عند حذف عنصر من السلسلة، يجب الانتباه للعنصر الذي يسبقه لربطه بالعنصر الـــذي يليه للحفاظ على ترابط السلسلة.

يوضّح المثال السابق استعمال المؤثّر *new*. يوجد في الحقيقة مؤثّر آخر قريب منه هو *new []*. يسمح هذا الأخير بحجز قائمة من العناصر ديناميكياً، وفي هذه الحالة يســتعمل المؤثّــر *delete []* لتحريرها.

ملاحظات:

- يجب عدم خلط استعمال المؤثّرات new و [ ] new بالتوابع malloc , free ,... . ويجب عدم استعمال أي تابع من C لتحرير الذاكرة المحجوزة ديناميكياً باستعمال new أو new [ ]. وبالعكس، يجب عدم استخدام مؤثّرات C++ وهي delete أو [ ] delete لتمرير الذاكرة المحجوزة ديناميكياً باستخدام توابع من C.
- إن الذاكرة المحجوزة ديناميكياً بـ new يجب أن تمرر بـ delete، وتلك المحجوزة بـ new[] تحرر بـ [ ] delete. هذا وإن عدم مراعاة هذه الإرشادات يؤدي إلى خطأ فادح عند التنفيذ، رغم أن المترجم لا ينبّه المبرمج على ذلك أثناء البرمجة.

في المثال السابق، لم نتحقق من نجاح المؤثّر new في حجز الذاكرة. وإذا لم يتمكن البرنامج من الحصول على الذاكرة التي يحتاج إليها، فسيحصل خطأ ويتوقف التنفيذ. لمعالجة هذه الحالة يمكن للمبرمج القيام بعدة آليات نذكر منها:

- التلقي "اليدوي" للخطأ
- إمكان استعمال assert
- تعريف مدير للمؤثّر new يسمى new_handler
- الإدارة العامة لأخطاء التنفيذ (وهي ما يسمى بالاستثناءات) من قبل آلية throw, try, catch.

إن التلقي اليدوي لهذا النوع من الأخطاء بسيط ولكنه ممل ويسبب تكرار الترميز. وأهم من ذلك أن هذه الطريقة من المعالجة لا تتلاءم مع تحقيق المكتبات القابلة لإعادة الاستعمال. فمن يبني مكتبة، لا يعرف سلفاً كيف يرغب المستثمر النهائي في التصرّف حيال مثل هذا الخطأ. والحل الأبسط هنا هو في إنهاء تنفيذ البرنامج فوراً وإعطاء الرسالة المناسبة.

```
#include <stdio.h>
#include <stdlib.h>
#include <new.h>
```

```
int main() {
 struct Geant {
      char p[10000];
 };
 Geant* p;
 int i=0;
 set_new_handler(0);
 do {
      p = new Geant;
      printf("Success of allocation %i time\n",i++);
      if (p == 0) {
           fprintf(stderr,"Failure of memory allocation for %s !\n","p");
           exit(EXIT_FAILURE);
      }
 } while (p);
}
```

يوضح هذا المثال كيف يمكن معالجة خطأ ناتج عن الفشل في حجز الذاكرة. نلاحظ هنا أن أول استعمال للمؤثّر new يجب أن يكون مسبوقاً باستدعاء set_new_handler المعرّف في new.h وهو الذي يضع مديراً خاصاً. وقد وضعنا قيمة محدّد الاستدعاء 0 وهذا ما يشير للمترجم إلى أنه عند فشل حجز الذاكرة أثناء استعمال new يجب أن يعيد القيمة 0.

يمكن اعتماد طريقة أخرى، تقوم على الإيقاف المفاجئ للبرنامج وذلك باستعمال الماكرو assert الموجودة في C ANSI.

أثناء بناء البرنامج، يضع المبرمج الماكرو assert في مواضع استراتيجية. تؤكد هذه الماكرو حالة القيم الحرجة للبرنامج في مواضع وجودها. يجري التحقق من التوكيد أثناء التنفيذ، وإذا لم يكن صحيحاً، يتوقف تنفيذ البرنامج مع إعطاء إشارة للمستخدم تحتوي نص التوكيد. وفي حال صحة التوكيد يستمر التنفيذ.

إن هذه الآلية قوية لأنها أكثر عمومية وتصح في حالات أخرى غير الإدارة الديناميكية للذاكرة.

```
#include <stdio.h>
#include <assert.h>
#include <new.h>

int main() {
  struct Geant {
```

```
  int p[20000];
};

Geant* p;
int i=0;
set_new_handler(0);
do {
      p = new Geant;
      assert(p != 0);
      printf("Success of allocation %i time\n",i++);
} while (p);
}
```

يُخفّض التحقق من صحة القيم سرعة التنفيذ، ولكن استعمال هذه الآلية يزيد سرعة تحقيق البرامج *الصحيحة*، إذ يفرض على المبرمج أن يحدد معنى المتحولات وأن يقبل قيام المترجم بتوليد عمليات تحقق حولها. بعد انتهاء اختبار البرنامج وتصحيحه، يمكن إبطال تفعيل هذه العمليات بتعريف الماكرو NDEBUG قبل ضم الملف assert.h وبذلك تبقى هذه العمليات غير مفعّلة في البرنامج المصدري ،ويمكن إعادة تفعيلها عند الصيانة.

توفّر C++ وسيلة لتعريف مدير أخطاء خاص مرتبط حصراً بالمؤثر new. يسمح التابع المِقيَس set_new_handler ( ) بتحديد عنوان التابع الذي يجب استدعاؤه آلياً عند استحالة تحقيق طلب حجز الذاكرة. وعلى وجه افتراضي، يقوم التابع المرتبط بهذا المدير بإقلاع استثناء ولا يعود. إن التابع المستعمل كمعامل للتابع set_new_handler ( ) يجب أن يكون تابعاً دون محدّدات، ويعيد void.

```
#include <stdlib.h>
#include <stdio.h>
#include <new.h>

struct Geant {
  int p[20000];
};

void no_more_memory()
{
  printf("No more free memory !");
  exit(EXIT_FAILURE);
}
```

```
int main() {
 set_new_handler(no_more_memory); // defined in <new.h>

 Geant* p;
 int i=0;
 do {
        p = new Geant;
        printf("Success of allocation %i time\n",i++);
 } while (p);
}
```

في هذا المثال جعلت إدارة الأخطاء بسيطة. ولكن في التطبيقات التي هي أكثر حساسية يجب أن تعالج الأخطاء بطريقة أكثر دقة.

يسمح الإمكان الذي تقدمه C++ فيما يتعلق ببناء إدارة الأخطاء المتعلقة بحجز الذاكرة شخصياً، إضافةً إلى إمكان إعادة تعريف عمليات new و delete، بتحقيق الإدارة الديناميكية للذاكرة الخاصة بالمسألة المراد حلها. ويمكن إذن أن نسند إلى set_new_handler دور جمع فتات الذاكرة وفق خوارزمية يقترحها المبرمج.

2-14 التوابع العمومية

اعتاد المبرمجون بلغة C استعمالَ المعالج المبدئي لتنفيذ برامجهم. ونحن لا ننكر الفوائد الناتجة عن ذلك، ولكن في الوقت نفسه لا نُغفِل الأخطار التي قد تنتج عن ذلك، والتي أشرنا إليها في حديثنا عن التوابع السطرية "inline". لقد تعلّم معمّمو لغة C++ الكثير من المشاكل الموجودة في لغة C ومن ثَم فهم ينصحون وبشدة بالتقليل ما أمكن من استعمال المعالج المبدئي.

يرى المدافعون عن المعالج المبدئي أنه لا يمكن الاستغناء عنه في حالات الترجمة الشرطية، و احتواء الملفات. وهذا صحيح ويجب أن يقتصر الأمر على هاتين الحالتين.

ولكن في C هنالك حالات أخرى يستعمل فيها المعالج المبدئي. فمثلاً يمكن أن نكتب تعاريف لتوابع عمومية، ثم يقوم المعالج المبدئي بإجراء الاستبدالات النصية المناسبة. والتوابع العمومية (generic functions) هي توابع مستقلة عن نمط المحدّدات. ولكن كان بناء هذه التوابع

مبرّراً في لغة C، إنه في C++ يعتبر لاغياً. وبسبب التنميط الشديد للغة C++، وللحفاظ على هذا الإمكان، زوّدت C++ بآلية خاصة قوية جداً لبناء توابع عمومية مع الحفاظ على ميّزة مراقبة الأنماط. هذه الآلية تعتمد على مفهوم القالب *template*.

لصنع تابع عمومي، يجب أن تُسبق ترويسة التابع بالكلمة المحجوزة *template*، يليها قائمة الموسطات محاطة بالرمزين < و >. يجب أن يكون كل موسط في هذه القائمة من الشكل "*class* identifier" أو أن يكون سلسلةً من المحارف، أو اسمَ تابع أو تعبيراً ثابتاً. في الواقع أثناء الحصول على مَثَل محدّد من هذا التابع العمومي (أي أثناء تخصيصه بأنماط محددة)، يجب أن يكون لكل المحدّدات "غير العمومية" قيمٌ ثابتة معروفة في لحظة الترجمة.

في حالة "*class* identifier"، يترك اختيار معيّن الهوية identifier للمبرمج. وعند وجود أكثر من موسط تُفصَل الموسطات بفاصلة. يمثّل معيّن الهوية نمطاً يجب تحديده عند الترجمة إلى نمط حقيقي. لهذا السبب يجب أن يكون الرماز المصدري للتابع العمومي متوفراً في لحظة الترجمة، ومن ثَم يجب أن يكون محتوىً أو موجوداً في الملف الذي يحتاج إلى تخصيص هذا التابع العمومي.

لنعد إلى المثال الوارد في فقرة التوابع السطرية. إن التابع max المذكور في المثال والمعرّف على مستوى المعالج المبدئي مستقلٌ عن النمط الفعلي للمحدّدات. يتيح لنا هذا التعريف خلطَ أنماط متعددة عند استدعاء التابع مع ما يحتويه هذا الخلط من أخطار. في المثال التالي، نبين إمكان وقوع هذا الأمر إذا ما جرى استدعاء max مع معاملين أحدهما صحيح والآخر من نمط محرفي. باستخدام آلية التوابع المعرّفة كقوالب، تُجرى مراقبة دقيقة للنمط. وبالطبع هذا لا يمنع من خلط الأنماط، شريطة أن يحتاط المبرمج لهذا بتصريحه عن هذه النية،باستعمال آلية تحويل الأنماط (type cast).

```
#include <stdio.h>

#define max(x,y) (((x) > (y)) ? (x) : (y))

template <class Type>
inline Type Max(Type x, Type y) {
```

```
 return (x > y ? x : y);
}

int main() {
 int a = 4;
 int b = 6;
 char c = '7';
 printf("%d\n",max(a,b));        // displays 6
 printf("%d\n",max(a,c));        // displays 55  !!

 double d,e;
 d = 12.3; e= 4.56;
 printf("%g\n",Max(d,e));        // displays 12.3
 printf("%d\n",Max(a,b));        // displays 6

 printf("%d\n",Max(a,int(c)));   // displays 55  !!
 printf("%d\n",Max(a,c));        // error !! a and c must have the same type
 printf("%d\n",Max(a,d));        // error !! a and d c must have the same type

return 0;
}
```

وكما في حالة التوابع العادية، يمكن إعادة تعريف التوابع العمومية. ثم إنه يمكن لتابع عمومي أن يقبل أنماطاً عمومية للمحدّدات وأخرى مثبّتة. يبين المثال التالي هذه الإمكان. يقبل التابع العمومي max ( ) التوقيعين: (Type table[ ], int) , (Type, Type). يعيد التابع الأول المحدّد الأكبر بين المحدّدين، على حين يعيد الثاني القيمة العظمى المحتواة في جدول عدد عناصره الفعلية يعطى بالمحدّد الثاني. يقوم التابع الثاني في أثناء عمله باستدعاء التابع max (Type x, Type y) وهذا ليس استدعاءً عودياً!

```
#include <stdio.h>

template <class Type>
inline Type max(Type x, Type y) { return (x > y ? x : y); }

template <class Type>
Type max(Type table[], int size) {
  Type maximum = table[0];
  for (int i = 1; i < size; i++)
    maximum = max(maximum,table[i]); // non recursive call !
  return maximum;
}
```

```
int tab_int[] = {2, 4, 1, 5, 7, 4, 3, 9, 3, 4};
char tab_char[] = {'g', 'd', 'z', 'a', 'q', 'c'};

int main() {
  printf("Maximum : %i\n",max(tab_int, int(sizeof(tab_int)/sizeof(int))));
  printf("Maximum : %c\n",max(tab_char, int(sizeof(tab_char)/sizeof(char))));
  return 0;
}
```

في التوابع العمومية، يجبّ أن تظهر جميع الموسطات العمومية في قائمة محدّدات التـــابع. ومن ثم، إنه لا يمكن كتابة تابع *template* يقبل محدّداً عمومياً، ويكون نمط النتيجة المعـــادة عمومياً ومختلفاً عن الأول، إذ إن جميع الأنماط الواردة في قائمة الموسطات العمومية يجـب أن تظهر فعلياً في سلسلة محدّدات التابع العمومي. لحل هذه المشكلة يمكن تعريف نمط النتيجـــة المعادة كموسط تخيلي.

نعرّف في البرنامج الوارد في المثال التالي تابعاً يقبل نمطين عموميين. هذا البرنامج يقـــوم بنسخ متحول نسميه المصدر، في متحول آخر نسميه المستقر، وذلك بنسخه ثُمانيّـــة تلـــو الثُمانية. ولما كان حجم المحرف في C++ هو 1 بالتعريف، فإن كلاً مـــن محـــدّدي المصـــدر والمستقر سيُحول إلى مؤشر إلى محرف. يُعطى المؤشر الأول عنوان المتحول المصدر كقيمـــة ابتدائية، ويعطي الثاني عنوان المتحول المستقر كقيمة ابتدائية. يسمح ذلك بالوصول إلى كـــل الثُمانيات في المتحولات ومن ثَم يسمح بإجراء النسخ.

```
template <class T1, class T2> void copy(T1 source, T2& dest) {
  unsigned int max = sizeof(source);
  if (max > sizeof(dest)) max = sizeof(dest);
  char* source_alias = (char*)&source;
  char* dest_alias = (char*)&dest;
  for(int i = 0;i < max; i++) dest_alias[i] = source_alias[i];
}

struct X {
  int x1;
  double x2;
} xx = { 123, 3.1415 } ;

struct Y {
  int y1;
} yy = { 987 };
```

```
int main() {
  copy(yy,xx);    // xx = { 987, 3.1415 } yy = { 987 }
  xx.x1 = 1;
  copy(xx,yy);    // xx = { 1, 3.1415 }  yy = { 1 }
  return 0;
}
```

2– 15 قنوات الدخل والخرج input and output streams

قمنا حتى الآن بتحقيق جميع برامج C++ باستعمال توابع الدخل والخرج التي تقدمها مكتبة C. ولكن C++ تقدم لحسن الحظ بديلاً أنيقاً جداً عن stdio.h من خلال مفهوم قنوات الثُمانيّات.

تمثّل القناة تدفق المعطيات بين البرنامج وجهاز دخل أو خرج خارجي (كالشاشة، الطابعة،....) أو بين البرنامج وملف ما. ضمنياً، ولأسباب تتعلق بزيادة الفعالية، يرتبط بكل قناة دخل/خرج ذاكرة مؤقتة (buffer) تنتقل المعطيات عبرها.

تعرّف مكتبة iostream.h التي توجد مع كل نظام C++ نمطين من الدخل والخرج يسمى الأول istream (input stream) (أي قناة الدخل) والثاني ostream, (output stream) (أي قناة الخرج).

تُنفّذ عملية الإخراج من قبل المبرمج باستعمال مؤثر الإدراج "<<" إذ يدرج قيمة في قناة الخرج.

أمّا عملية الإدخال فهي تنفّذ باستعمال مؤثر الاستخراج ">>". أي أننا نستخرج قيمة من قناة الدخل.

وهنالك ثلاثة متحولات قَبْلِيَّةُ التعريف من النمط "قناة" وهي:

- cin، متحول من النمط istream مرتبط بملف الدخل المِقيَس. وهو يُكافىء stdin في C
- cout، متحول من النمط ostream مرتبط بملف الخرج المِقيَس. وهو يكافىء stdout في C

- Cerr متحول من النمط ostream مرتبط بملف الأخطاء المِقيَس. وهو يكـــافىء stderr في C وهو مثله لا ترتبط به ذاكرة مؤقتة.

لنبدأ بعرض المثال التقليدي "Hello, world":

```
#include <iostream.h>

int main() {
  cout << "Hello, world\n";
}
```

يتسبب هذا البرنامج في إدراج سلسلة المحارف "Hello,world" في قناة الخرج المِقيَـــــس، وهي عادة شاشة الحاسوب. وقد يتساءل المبرمج بلغةC عن فائدة إدخال وســـيلة جديـــدة للقيام بما كان التابع ( ) Printf يقوم به. يوضّح المثال التالي فوائد هذه الوسيلة الجديدة.

```
#include <iostream.h>

int main() {
  int i = 123;
  char* s = "Streams make life easy!";
  double d = 67.435687;

  cout << "This is a mixing :\nfirst of all an integer : " << i;
  cout << " a double : " << d << "\nand a string : " << s << endl;
  cout << "Enter an integer : ";
  cin >> i;
  cout << "The integer equals : " << i << endl;
  return 0;
}
```

يعطي تنفيذ البرنامج ما يلي:

```
This is a mixing:
first of all an integer : 123 a double : 67.4357
and a string : Streams make life easy!
Enter an integer : 56
The integer equals : 56
```

يمكن كما رأينا في المثال أن نستعمل تعليمة إدراج وحيدة بدمج عدة أنماط من المعطيـات لإدراجها. فمؤثرات الإدراج والاستخراج تستعمل مع الأنماط المعرّفة سابقاً وهي long , int , short , double , float , char , char*. يجري تقييم تعليمة الإدراج والاستخراج من اليسار إلى

اليمين. وفي حالة الشك يفضّل استعمال الأقواس.

يبين المثال السابق استعمال المؤثر *endl* (handler) الذي يتسبب في إدراج المحرّف "\n" ثم يكتب الذاكرة المساعدة المرتبطة بملف الخرج المقيّس ويفرّغها.

إن استعمال "<<", ">>" أسهل بكثير من استعمال التوابع printf ( ) و scanf ( ) إذ لم يعد المبرمج مضطّراً إلى لاهتمام بأنماط المعطيات التي يعالجها. فالنظام ذكي بما فيه الكفاية ليستطيع تطبيق المؤثّر المناسب للنمط. سنرى في الفصل التالي، كيف تقدم هذه الطريقة فوائد إضافية، بمعنى أن المبرمج يستطيع تعريف المؤثرات << و >> على الوجه الذي يناسبه. ثم إن هذا النظام يقدم المزيد من الحماية لأن المبرمج لا يكون مضطراً إلى إعلام النظام بنمط المعطيات الذي تجري معالجته، ومن ثَم، يقل احتمال الوقوع في الخطأ.

إضافةً إلى المؤثر *endl*، تقدم C++ مؤثرات أخرى من خلال المكتبة iostream.h. فهنالك dec, flush, hex, oct, endl,...

تتسبب *flush* في كتابة الذاكرة المؤقتة المرتبطة بالقناة، ومن ثَم بمسحها وإفراغها. قد يكون هذا الأمر ضرورياً في عمليات التحصيل، إذ يجب أن يظهر النص الذي يوضح المطلوب قبل أن يقوم المؤثر بإدخال القيمة المطلوبة. فإن لم يستعمل flush، فلا نضمن هذا الأمر.

```
#include <iostream.h>
int main() {
  int numbe;
  cout << "Enter a number between 0 and 12 " << flush;

  cin >> nombre;
}
```

أما *endl* فهو لا يقوم بأكثر من إدخال "\n" قبل القيام بـ *flush*.

Hex يجعل النظام ينتقل للنمط الست عشري أثناء استخراج وإدراج الأعداد الصحيحة. يبقى هذا النظام فعّالاً إلى أن يطلب المبرمج تغييره. وهناك مؤثرات أخرى مشابهة هي *oct*، *dec* غير أن الأول يعتمد الأساس الثماني، والثاني يعتمد الأساس العشري. في الحالة العامة

الافتراضية يعمل النظام في النمط العشري.

```
#include <iostream.h>

int main() {
  int n;
  cout << "Enter an integer number " << flush;
  cin >> n;  // equivalent to cin >> dec >> n;
  cout << "This is the  number in hexadecimal : " << hex << n << endl;
  cout << " This is the  number in octal : " << oct << n <   endl;

  cout << (++n) << endl; // we are still in octal mode!

  cout << " This is the  number in hexadecimal : " << hex << n << endl;
  // now we move to decimal mode
  cout << " This is the  number in decimal : " << dec << n << endl;

  cout << " Enter an integer number " << flush;
  cin >> hex >> n;  // input in en hexadecimal mode
  cout << " This is the  number in hexadecimal: " << hex << n << endl;
  cout << " This is the  number in decimal: " << dec << n << endl;

  cout << " Enter an integer number " << flush;
  cin >> n;  // still in hexadecimal mode
 cout << "" This is the  number in hexadecimal: " << hex << n << endl;
  cout << "" This is the  number in decimal: " << dec << n << endl;

  return 0;
}
```

يعطي تنفيذ البرنامج ما يلي:

```
Enter an integer number 45
This is the  number in hexadecimal : 2d
This is the  number in octal : 55
56
This is the  number in hexadecimal : 2
This is the  number in decimal : 46
Enter an integer number AB
This is the  number in hexadecimal: ab
This is the  number in decimal: 171
Enter an integer number D2
This is the  number in hexadecimal: d2
This is the  number in decimal: 210
```

يسمح المؤثر ws (white space) أثناء القراءة من القناة باستخراج واستبعاد جميع المحارف

"الفارغة" أي محارف الفراغ والجدولة والقفز إلى سطر جديد والقفز إلى صفحة جديـــدة، وذلك اعتباراً من الموقع الحالي وحتى أول محرف غير فارغ.

وإضافةً إلى مؤثرات التحويل هذه، يوجد مجموعة مؤثرات أخرى تقبل محـــدّداً. هـــذه المؤثرات ذات المحدّد معرّفة في المكتبة iomanip.h. ومنها على سبيل المثال:

...setfill(char) , setprecision(int), setw(int)

يسمح setw(int) بتحديد حجم الحقل المراد إدراج القيمة ضمنه. وفي هذه الحالة يكـــون التنسيق إلى اليمين وهو أمر عملي في حالة تمثيل الأعداد.

```
#include <iostream.h>
#include <iomanip.h>
int main() {
 int n = 64;
 cout << "In hexadecimal : " << hex << n << endl;
 cout << " In octal : " << oct << n << endl;
 cout << " In decimal : " << dec << n << endl;
 // The same display right justified
 cout << setw(20) << "Hexadecimal : " << hex << setw(6) << n << endl;
 cout << setw(20) << "Octal : " << oct << setw(6) << n << endl;
 cout << setw(20) << "Decimal : " << dec << setw(6) << n << endl;

 return 0;
}
```

يعطي التنفيذ النتيجة التالية:

```
In hexadecimal :  40
 In octal : 100
In decimal : 64
        Hexadecimal :    40
              Octal :   100
    Decimal :    64
```

ثم إن هذا المؤثر يبدو مفيداً جداً عند تحصيل القيم. يقوم البرنامج التالي بإظهار النـــص المدخل على شكل كتل طولها LL-1 ينتهي كل منها بالمحرف "\0".

```
#include <iostream.h>
#include <iomanip.h>
const int LL = 10;  // maximum size of a line
int main() {
```

```
 char line[LL];
 while (cin >> setw(LL) >> line) cout << line << endl;
 return 0;
}
```

يعطي التنفيذ ما يلي:

```
abcdefghijklmnopqrstuvwxyz
abcdefghi
jklmnopqr
stuvwxyz
^Z[1]
```

في الحالة الافتراضية، يجري إظهار أعداد الفاصلة العائمة على 6 أرقام معنويـــة. يمكـــن تعديل هذه الدقة باستعمال المعالج setprecision (int).

```
#include <iostream.>
#include <iomanip.h>

const double pi = 3.141592654;

int main() {
 cout << pi << endl;                  // 6 digit by default
 cout << setprecision(9) << pi << endl; // 9 digit
 cout << pi/2.0 << endl;              // we are still on  9 digit
 cout << setprecision(2) << pi << endl; // 2 digit
 return 0;
}
```

وعند التنفيذ نجد:

```
3.14159
3.14159265
1.57079633
3.1
```

المؤثر الأخير الذي سنعرضه هنا هو setfill(char). يستعمل هذا المؤثر عادةً مرتبطاً بـــــ setw(int)، ويملأ المساحة غير المستعملة بالمحرف المحدَّد.

```
#include <iostream.h>
#include <iomanip.h>
```

[1] ^Z هو المحرف الذي ينهي ملف الدخل في الأجهزة التي تعمل بـــ Ms_Dos.

```
int main() {
 int n = 64;
 cout << "In hexadecimal : " << hex << n << endl;
 cout << "In octal : " << oct << n << endl;
 cout << "In decimal : " << dec << n << endl;
 // The same display right justified
 cout << setw(20) << "Octal : " << oct << setw(6) << n << endl;
 cout << setw(20) << "Decimal : " << dec << setw(6) << n << endl;

 cout << setw(20) << "In hexadecimal : ";
 cout << hex << setfill('.') << setw(6) << n << endl;
 cout << setfill(' ') << setw(20) << "In octal : ";
 cout << oct << setfill('.') << setw(6) << n << endl;
 cout << setfill(' ') << setw(20) << "In decimal : ";
 cout << dec << setfill('.') << setw(6) << n << endl;

 return 0;
}
```

يعطي التنفيذ النتيجة التالية:

```
In hexadecimal :  40
 In octal : 100
In decimal : 64
          In hexadecimal : ....40
                  In  Octal : ...100
     In Decimal  : ....64
```

يمكن للقارئ المهتم أن يعود لدليل المستخدم للمترجم C++ الذي يعمل عليه، للاطـــلاع على القائمة الشاملة للمؤثرات الموجودة.

في الفصل التالي، سنرى كيف يقوم المبرمج بتعريف مؤثراته الخاصة التي يمكن دمجـــها في المكتبات الموجودة.

إن المؤثرات "<<","<<" معرفة سابقاً لجميع الأنماط البسيطة وللنمط *char. تكون قيمـــة المؤثر "<<" هي True ما دامت نهاية الملف لم تظهر بعد. وسنستفيد من هـــذه الخاصيـــة في البرنامج التالي الذي ينسخ محتوى ملف الدخل المِقيَس في ملف الخرج المِقيَس.

```
#include <iostream.h>
int main() {
  char c;
  while (cin >> c) cout << c;
  return 0;
```

```
}
```

يعطي تنفيذ هذا البرنامج:

```
abc  defg  h^Z
abcdefgh
```

السطر الأول هو السطر المدخل عن طريق لوحة المفاتيح. أما السطر الثاني فهو السطر الناتج عن التنفيذ. يبين هذا المثال كيف أن الفواصل في C++ لا تعامل معاملة المحارف من قبل المؤثر "<<"، بل تستعمل لفصل قيمتين متتاليتين في ملف الدخل المِقيَس.

أما المثال التالي، فهو ينسخ ملف الدخل المِقيَس بحذافيره على ملف الخرج المِقيَس مستخدماً تابعاً خاصاً بالنمط istream وهو get(char&).

```
#include <iostream.h>
int main() {
  char c;
  while (cin.get(c)) cout << c;
  return 0;
}
```

يعطي تنفيذ البرنامج:

```
abc  defg  h
abc  defg  h
^Z
```

get(char&) هو تابع يستخرج محرفاً من ملف الدخل المِقيَس، ويعيده من خلال المحدّد الممرّر بالمرجع من نمط char. إن استعمال التابع get( ) غير مألوف لمبرمج بلغة C، ولكنه أمر مميز للبرمجة الغرضية التوجه كما سنرى في الفصول القادمة. يعالج get( ) الفواصل كمحارف عادية. وعند بلوغ نهاية الملف يُعيد التابع القيمة EOF. يعيد التابع get( ) القناة التي استدعي التابع معها، وهذه القناة تستعمل فيما بعد كشرط في الحلقة. في الحقيقة، إذا ما حصل أي شيء غير عادي، فإن القيمة المعادة تكون بحيث يصبح الشرط خاطئاً. وهنالك تابع معاكس هو put(char) يكتب المحرف الممرّر كمحدّد في القناة.

لمعرفة المزيد من التوابع التي تعالج القنوات، يستطيع القارئ العودة لدليل الاستخدام الخاص بمترجم C++ الذي يستعمله. وهنالك توابع مفيدة جداً مثل putback( ), get( ),

... ، tellg( ), seekg( ), peek(), getline( )

يسمح التابع getline(char* buf, int limit, char delim) باستخراج مجموعة محارف من القناة عددها يساوي العدد المعطى في المعامل الثاني، ما لم نبلغ نهاية الملف أو نصل إلى المحرف delim. إن المحرف delim هو في الحالة العامة المحرف "\n" ما لم ينص صراحة على غيره. وفي الحقيقة فإن العدد الأعظم للمحارف المستخلص هو limit-1 لأن آخر محرف مفيد في الذاكرة المؤقتة هو المحرف "\0" إجبارياً. توضع المحارف المستخرجة في الذاكرة المؤقتة المشار إليها بالمعامل الأول. أما المحرف الفاصل delim فيستخرج من القناة ولكنه لا ينسخ على الذاكرة. يسمح هذا التابع بقراءة ملف نصي بسهولة سطراً تلو السطر.

```
#include <iostream.h>
const int MAX_LINE = 128;
int main() {
  char line[MAX_LINE];

  while (!cin.eof ()) {
                cin.getline(line,MAX_LINE, '\n');
                cout << line << endl;
  }
  return 0;
}
```

يمكن للمبرمج أن يختبر حالة القناة باستعمال التوابع المخصصة لهذا، وهي: bad(), fail(), eof(), good()...

تعيد هذه التوابع قيم بتّات الحالة وهي badbit, failbit, eofbit المتعلّقة داخلياً بالقناة.

- eof( ): يعيد القيمة true إذا بلغنا نهاية الملف في القناة، وإلا فهو يعيد false.
- bad( ): يعيد القيمة true إذا فشلت عملية على القناة. وعموماً، لا يمكن استعادة المعطيات الموجودة عندها في القناة. وهي أسوأ حالة يمكن أن تطرأ.
- fail( ): يعيد القيمة true إذا فشلت عملية على القناة ولكن بعكس الحالة bad، يمكن استرداد المعطيات.

- good( ): لا يعيد true إلا في الحالة التي تكون فيها جميع التوابع ( )eof,( )fail,( )bad كلها خاطئة false.

هذا وإن العملية التي تجري على القناة والتي لا تكون حالتها ( )good ليس لهـــا تأثـــير. ويمكن إعادة بتّات الحالة إلى الصفر باستعمال التابع ( )clear.

لندرس الآن بعض التوابع المفيدة:

- التابع ( )get: يعرّف بنفس توقيع التابع ( )getline أي get(char* buf, int limit, char delim). ولكن تصرفه يختلف قليلاً عن getline، إذ باستعمال ( )get لا نستخرج الفاصل delim بعكس ( )getline.
- التابع putback(char): يسمح بإعادة وضع آخر محرف سبقت قراءته مـــن القنـــاة باستعمال التابع ( )get في القناة. هذا المحرف المعاد هو المحرف الذي سبقت قراءتـــه وإلا فالنتيجة تكون غير معرّفة.
- التابع( )peek: يعطي المحرف الموجود في القناة والذي ستجري قراءته في الاســتدعاء التالي للعملية ()get ولكن هذا التابع لا يستخرج المحرف من القناة.

قبل المتابعة، لا بد من الإشارة إلى أنه يرتبط بكل قناة مؤشران داخليان يشيران إلى موضع نسبي ضمن القناة، يستخدم في عملية الاستخراج أو الإدراج التالية.

- التابع( )seekg: هو تابع يسمح بوضع المؤشر الداخلي للاستخراج "get" للقناة علـى المحرف الذي يُعطى موضعه كمحدّد. افتراضياً، يكون هذا الموضع اعتباراً مـــن بدايـــة القناة.
- التابع( )seekp: عمله مطابق لعمل التابع ( )seekg ولكن في مؤشر الإدراج "put".
- التابعان ( )tellp, ( )tellg يعطيان الموضع الحالي للمؤشر "put", "get" على الـــترتيب ضمن القناة.

يجب ألا يغيب عن بالنا أن C++ تسمح باستعمال مكتبات CANSI مثل ctype.h الـــتي

تعرف توابع مفيدة جداً للتعامل مع المحارف الصادرة عن قناة أو الموجهة لقناة وهي:

```
int isalnum (char)
int isalpha (char)
int iscntrl (char)
int isdigit (char)
int isgraph (char)
int islower (char)
int isprint (char)
int ispunct (char)
int iscntrl( ),isspace( )
int isspace (char)
int isupper (char)
int isxdigit (char)
int  tolower (char)
int toupper (char)
```

من غير المنصوح به أن تستعمل المكتبتان iostream.h , stdio.h إحداهما مع الأخرى لأنه لا يوجد بينهما أي ارتباط.

يسمح نظام البرمجة C++ بربط الملفات بقنوات الدخل والخرج. يمتلك المبرمج ثلاثة أنواع قَبْلِيَّة التعريف في fstream.h وهي ofstream, ifstream, fstream. لا يمكن فتح ملف من النمط ifstream إلا للقراءة، ولا يمكن فتح ملف من النمط ofstream إلا للكتابة، أما ملف من النمط fstream فيمكن فتحه للقراءة أو الكتابة.

ويمكن تطبيق جميع التوابع المذكورة آنفاً التي تتعامل مع القنوات، على القنوات التي ترتبط بها ملفات. ويضاف إلى هذه التوابع، تابعان تقليديان يستخدمان في إدارة الملفات هما: close(), open(). يسمح التابع open( ) بفتح ملف، وربطه بقناة مصرح عنها سلفاً. يعتبر اسم الملف الخارجي محدّداً يعطى للتابع على شكل سلسلة محارف.

يقوم التابع close( ) بإغلاق الملف المرتبط بالقناة، ويقطع الصلة بين الملف والقناة. لا يقبل هذا التابع أية محدّدات.

ضمنياً، يغلق الملف المرتبط بالقناة آلياً عند انتهاء وجود المتحول الدال على القناة. ويمكن الدلالة على الاسم الخارجي للملف- أي الاسم الذي يُعرف به الملف من قبل نظام الاستثمار- عند تعريف المتحول الدال على القناة.

يبين البرنامج التالي بعضاً من الإمكانات المذكورة آنفاً، إذ يُنشأ الملف النصي أولاً. فإذا كان هناك ملف بنفس الاسم، فإن محتواه سيضيع ويُفقد. يقوم المستثمر بإدخال الكلمات، وفي نهاية عملية الإدخال، يُغلق الملف. يعاد فتح الملف بعد ذلك في نمط القراءة، وتُقرأ الكلمات الموجودة في الملف اعتباراً من بدايته وحتى نهايته. وبسبب لوجود فواصل تفصل الكلمات، يجب استخراج الفاصل الذي يتلو كل كلمة بعد استخراجها.

```
#include <fstream.h>
#include <stdlib.h>

const int MAX_LINE = 80;

int main() {
  ofstream outputFile("test.txt");
  cout << "Enter a word ; ^Z to exit" << endl;
  char word1[MAX_LINE];
  while (cin >> word1) {
    c << word1<< endl;
    cout << " Enter a word ; ^Z to exit " << endl;
  }
 outputFile.close();

  ifstream inputFile("test.txt");
  if (!inputFile) {
     cerr << "Cannot open the file TEST.TXT" << endl;
     exit(EXIT_FAILURE);
  }
  while (!inputFile.eof()) {
      inputFile >> word1;
      inputFile.get();
    cout << word1<< endl;
  }
   inputFile.close();
  return 0;
}
```

يمكن الاستغناء عن التعليمة close( ) في هذا المثال إذ تُغلق الملفات المرتبطة بالقنوات آلياً في نهاية البرنامج.

يقوم البرنامج التالي بإحصاء عدد المحارف الموجودة في الملف المنشأ سابقاً. نلاحظ هنا أن الفواصل لا تحسب لأننا باستعمال معامل الاستخراج نقوم باستخراج الفراغات وحذفها.

```
#include <fstream.h>
#include <stdlib.h>

int main() {
 ifstream outputFile ("test.txt");
 if (!outputFile) {
   cerr << " Cannot open the file 'TEST.TXT'" << endl;
   exit(EXIT_FAILURE);
 }
 char car;
 int nbr_car = 0;

 while (outputFile >> car) nbr_car++;

  outputFile.close();
 cout << "TEST.TXT includes " << nbr_car << " characters " << endl;
 return 0;
}
```

نلاحظ هنا الشكل المختصر للتعبير في الحلقة while للدلالة على أن عملية الزيادة يجب أن تستمر ما دامت عملية استخراج محرف من الملف ناجحة.

لإنهاء هذا الفصل الذي يُعنى بإدارة الملف، سنورد مثالاً يبين كيفية بناء ملفات نصية مؤلفة من تسجيلات بحجم ثابت وذات نفاذ مباشر.

إن الملف المراد بناؤه هو ملف موظفين. وقد تعمّدنا تبسيط بنية التسجيلة تسهيلاً لفهم البرنامج.

تواجهنا هنا صعوبة في البرنامج وهي ضرورة جعل التسجيلات جميعها ذات حجم ثابت، وضرورة كون الحقول الداخلية للتسجيلة ذات طول ثابت بحيث نتمكن من الوصول المباشر إليها.

في الواقع، ولأن التسهيلات المقترحة للإدخال والإخراج عبر القنوات تُعنى حصراً بإجراء تحويلٍ للأغراض ذات الأنماط إلى سلاسل من المحارف والعكس بالعكس، فإن مجرد كتابة عدد صحيح في القناة يعتبر مشكلة إذ إن حجم المكتوب يتعلق بالقيمة الجبرية للعدد. ففي النظم التقليدية يخزّن العدد الصحيح بشكله الثنائي ضمن حجم ثابت هو لمانيّتان مثلاً. ولكن

هذا الأمر ليس صحيحاً في حالتنا لأن الأعداد تخزن على شكل محارف. يجب إذن أن نتدارك الأمر بحيث يكون للحقل الحجم ذاته في جميع تسجيلات الملف.

```
#include <fstream.h>
#include <iomanip.h>
#include <stdlib.h>

enum bool {false=0, true};
const char TAB = '\t';

const short SIZE_MATR = 10;
const short SIZE _NAME = 20;
const short SIZE _SALARY = 10;

struct Employee {
  unsigned int matricule;
  char nom[SIZE _NAME];
  long unsigned int salary;
};
const int SIZE _REC = SIZE _MATR+( SIZE _NAME+1)+ SIZE _SALARY
                      *2+ sizeof(TAB)+sizeof('\n');
```

لا تُظهر البنية التقليدية Employee الحاجة إلى استعمال فواصل. يجب على حجم التسجيلة أن يأخذ بعين الاعتبار إذن الفواصل الموضوعة بين الحقول المختلفة (محرف الجدولة) وكذلك محرف نهاية السطر الذي نضعه في نهاية التسجيلة لفصلها عن التسجيلة التالية لها.

يهتم التابعان writeEmployee( ), readEmployee( ) بأن يكون للحقول الطول المطلوب، وبأن تكون الحقول مفصولة بمحارف جدولة، وبأن تنتهي التسجيلة بمحرف نهاية السطر.

```
void writeEmployee(ofstream& outputFile, const Employee& e) {
 outputFile << setw(SIZE_MATR) << e.matricule;
 outputFile << TAB;
 outputFile << setw(SIZE _NAME) << e.name;
 outputFile << TAB;
 outputFile << setw(SIZE _SALARY) << e.salary << endl;
}

void readEmployee(ifstream& inputFile, Employee& empl) {
 inputFile >> empl.matricule;
 inputFile >> ws;
 inputFile >> empl.name;
 inputFile >> ws;
```

```
inputFile >> empl.salary;
inputFile >> ws;   // read end of line
}
```

في هذين التابعين يمرّر المحدّد البنية بالمرجع. هذا الأمر بديهي في حالة القراءة، لأن التابع يعيد التسجيلة المقروءة عبر هذا المحدّد. أما في حالة تابع الكتابة، فيستعمل التمرير بالمرجع لأسباب تتعلق بالفعالية، ولكننا في الوقت نفسه نضع الحماية اللازمة خوفاً من حصول تجاوز، وذلك باستعمال الكلمة المحجوزة const التي تبين مقصدنا هذا بوضوح. يجري إظهار الموظف مباشرة.

```
void displayEmployee(const Employee& e) {
 cout << e.matricule << TAB << e.name << TAB << e.salary << endl;
}
```

في هذا البرنامج الصغير، نعرض كيفية معالجة عملية بحث تسلسلي عن موظف بواسطة الرقم الذاتي المعطى له، وكذلك الوصول المباشر إلى إحدى تسجيلات الملف بواسطة رقم التسجيل الذي يبدأ بـ 1.

يعيد لنا تابع البحث عن الرقم الذاتي القيمة true إذا نجح البحث، أو false إذا فشل. لهذا التابع ثلاثة محدّدات: القناة التي يجري البحث ضمنها، والرقم الذاتي الجاري البحث عنه، والبنية التي ستحتوي التسجيلة في حالة النجاح.

```
bool findEmployee(ifstream& inputFile,
                  unsigned int matricule,
                  Employee& empl) {

 /* sequential research for the matricule */

 inputFile.clear();        // clear file indicators
 inputFile.seekg(0);       // positionning on the beginning of the file
 bool found = false;
 while ((!inputFile.eof()) & (!found) ) {
  inputFile >> empl.matricule;
  if (empl.matricule == matricule)  found = true;
   inputFile >> ws;
   inputFile >> empl.name;
   inputFile >> ws;
   inputFile >> empl.salary;
   inputFile >> ws;
```

```
  }
  return found;
}
```

يوضع مؤشر القراءة الداخلي في بداية القناة، وتوضع مؤشرات الحالة في حالة الصفر قبل بدء البحث. عند إيجاد الرقم المطلوب تتوقف الحلقة بسبب تغير قيمة المتحول found. من الواضح أن المحدّد empl يحتوي على المعطيات، هذه المعطيات لن تكون معرَّفة في حالة إعادة التابع للنتيجة false.

```
bool accesEmployee(ifstream& inputFile,
                   unsigned int number,
                   Employee& empl) {

 /* direct access to the file records */

 inputFile.clear();   // clear file indicators
 inputFile.seekg((number-1)*SIZE_REC);
 if (inputFile.peek()==EOF) return false;
 inputFile >> empl.matricule;
 inputFile >> ws;
 inputFile >> empl.name;
 inputFile >> ws;
 inputFile >> empl.salary;
 inputFile >> ws;
 return true;
}
```

يكفي لتنفيذ النفاذ المباشر لتسجيلة بواسطة رقم نسبي، بالنسبة إلى بداية الملف، وعلماً بأن التسجيلة الأولى تحمل رقم 1، يكفي أن يجري حساب الانزياح بالمحارف بالنسبة إلى بداية القناة، وذلك بضرب حجم التسجيلة في رقم التسجيلة المراد الوصول إليها. يجب الانتباه طبعاً إلى عدم تجاوز نهاية الملف. وبالطبع يُفترض أن المستثمر يدخل عدداً موجباً.

للتبسيط، سنملأ الملف بسبع تسجيلات نصرح عنها كمتحولات ساكنة.

```
Employee empl1 = { 7, "Nino", 100000L };
Employee empl2 = { 2, "Laurent", 70000L };
Employee empl3 = { 3, "Julian", 50000L };
Employee empl4 = { 12, "Anna", 90000L };
Employee empl5 = { 30, "Lili", 110000L };
Employee empl6 = { 35, "Dominique", 40000L };
Employee empl7 = { 39, "Lina", 0L };
```

```
int main() {

 /* file creation */
 ofstream outputFile("employee.dat");
 writeEmployee(outputFile, empl1); writeEmployee (outputFile, empl2);
writeEmployee (outputFile, empl3); writeEmployee (outputFile, empl4);
writeEmployee (outputFile, empl5); writeEmployee (outputFile, empl6);
writeEmployee (outputFile, empl7);
outputFile.close();

 ifstream inputFile("employee.dat");
 if (!inputFile) {
   cerr << "Cannot open EMPLOYEE.DAT" << endl;
   exit(EXIT_FAILURE);
 }

 /* sequential file reading */
 Employee empl;
 while (inputFile.peek() != EOF) {
  readEmployee(inputFile, empl);
  displayEmployee(empl);
 }

 /* sequential search in the file */
 unsigned int matr;
 cout << "Enter a matricule, ^Z to exit " << flush;
 while (cin >> matr) {
  if (findEmployee (inputFile, matr,empl))
    displayEmployee(empl);
  else
    cout << matr << " not in  EMPLOYEE.DAT" << endl;
  cout << " Enter a matricule, ^Z to exit " << flush;
 }

 /* direct access in the file */
 cin.clear(); // clear the precedent character end of file
 cout << "Record number  ? ^Z to exit " << flush;
 unsigned int num;
 while (cin >> num) {
  if (accesEmployee(inputFile, num,empl))
    displayEmployee(empl);
  else
    cout << num << " not in EMPLOYEE.DAT" << endl;
  cout << " Record number  ? ^Z to exit " << flush;
 }
```

```
 inputFile.close();
  return 0;
}
```

يعطي تنفيذ البرنامج ما يلي:

```
7       Nino            100000
2       Laurent         70000
3       Julian          50000
12      Anna            90000
30      Lili            110000
35      Dominique       40000
39      Lina            0

Enter a matricule, ^Z to exit  5
5 not in  EMPLOYEE.DAT
Enter a matricule, ^Z to exit 30
30      Lili            110000
Enter a matricule, ^Z to exit ^Z
Record number  ? ^Z to exit 6
35      Dominique       40000
Record number  ? ^Z to exit 1
7       Nino            100000
Record number  ? ^Z to exit 8
8 not in  EMPLOYEE.DAT
Record number  ? ^Z to exit 0
0 not in  EMPLOYEE.DAT
Record number  ? ^Z to exit  ^Z
```

إن الشروح المقدمة في الكتاب حول مكتبات الدخل والخرج الخاصة بـــ C++ ليســـــت شاملة. ويمكن للقارئ المهتم العودة إلى دليل الاستخدام الخاص بنظام C++ الذي يستعمله.

2– 6 1 الكلمات المحجوزة

تعرف C++ عدداً من الكلمات المحجوزة، إضافة إلى الكلمـــات المحجـــوزة في C ANSI وتظهر في الجدول اللاحق بحروف مائلة.

and	*and_eq*	asm	auto	*bit_and*	*bit_or*
bool	break	case	*catch*	char	*class*
compl	const	*const_cast*	continue	default	*delete*
do	double	*dynamic_cast*	else	enum	*explicit*

extern	*false*	float	for	*friend*	goto
if	*inline*	int	long	*mutable*	*namespace*
new	*not*	*not_eq*	*operator*	*or*	*or_eq*
private	*protected*	*public*	register	*reinterpret_cast*	
return	short	signed	sizeof	static	*static_cast*
struct	switch	*template*	*this*	*throw*	*true*
try	typedef	*typeid*	*typename*	union	unsigned
virtual	void	volatile	*wchar_t*		while *xor*
xor_eq					

Damascus University

الفصل 3 : تعريف الأنماط الجديدة

درسنا في الفصل السابق بعض الإمكانات الجديدة التي تقدمها لغة C++ للمبرمجين بلغة C. تظهر فوائد هذه الإمكانات مباشرةً، لا سيما على مستوى الأمن أثناء البرمجة (التحقق من الأنماط)، والتجانس في إدارة الدخل والخرج، والتخفيف من استعمال المعالج المبدئي قدر الإمكان.

ومع أن هذه التوسعات على درجة كبيرة من الأهمية، فإنها ليست سوى جزءٍ صغيرٍ من قوة التعبير في لغة C++. سندرس في هذا الفصل الوسائل التي تقدمها C++ في مجال بناء أنماط جديدة.

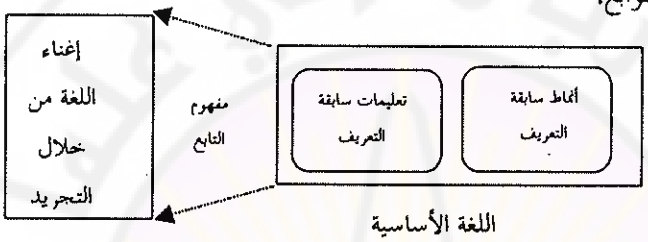
في لغة C، يملك المبرمج بعض الأدوات البدائية التي تسمح بتعريف أنماط جديدة إضافةً إلى الأنماط السابقة التعريف، وهي النمط الصحيح والنمط المحرفي والنمط الحقيقي. يمكن للمبرمج أن يشكّل أنماطاً جديدة باستعمال المؤشرات والجداول والبنى والاجتماع والأنماط المعدودة، إضافة إلى إمكان typedef، وبذلك يقرّب المفاهيم التي يتعامل معها في برنامجه من المفاهيم الحقيقية التي يجب عليه نمذجتها. ولكن مما يؤسف له أن المترجم في C لا يدعم المبرمج أثناء التطوير وهذا ما قد يؤدي إلى الوقوع في العديد من الأخطاء.

تقدم لغة C++ للمبرمج وسائل تعريف أنماط جديدة، مع إمكان دمج هذه الأنماط على وجهٍ أفضل في مجموعة الأنماط المعرفة سابقاً، يرافق ذلك كله قيام المترجم بجميع عمليات التحكم والمراقبة.

إن بناء أنماط جديدة قابلة للتكامل التام مع ما هو موجود في اللغة، هو خطوةٌ نحو بناء برمجيات أفضل جودةً، ونحو قابلية إعادة استعمال الترميز الموجود، باستغلال أنماط المعطيات التي يبنيها المبرمج أثناء عمله.

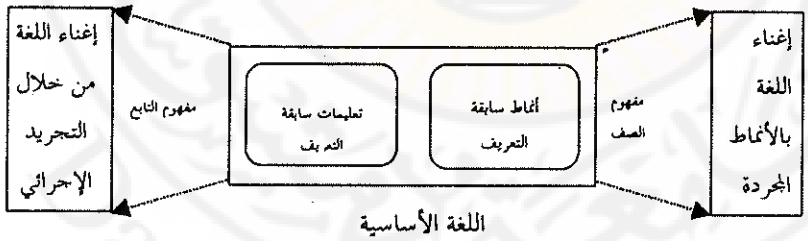
3– 1 مفهوم النمط المجرّد

تسمح غالبية لغات البرمجة الإجرائية بالقيام بما يسمى *التجريد الإجرائي procedural abstraction*، بمعنى أنها تعطي المبرمج إمكان بناء عمليات جديدة إضافة إلى العمليات السابقة التعريف في اللغة. في C وفي C++، نستطيع بلوغ هذا الشكل من أشكال التجريد باستعمال مفهوم التوابع.



الشكل 3–1: في C يمكن للمبرمج "توسيع" مجموعة التعليمات من خلال مفهوم التابع

غير أنه من الضروري أن ندرك أن التجريد الإجرائي لا يشمل إلا جانباً واحداً من جوانب السيطرة على التطوير البرمجي. في الحقيقة، من المفيد غالباً أن نتمكن من زيادة قوة تعبير اللغة ببناء *أنماط معطيات مجردة abstract data types* جديدة. فعندما نحدّد نمط معطيات مجرد، فإننا نعمد إلى تحديد *فضاء* للقيم أو بمجموعةٍ من الأغراض (وهي معلومات منظمة بطريقة خاصة) إضافة إلى العمليات المسموح القيام بها على هذه القيم أو الأغراض.



الشكل 3–2: في C++ يستطيع المبرمج أيضاً إغناء اللغة ببناء أنماط معطيات جديدة

وكما سنرى لاحقاً، فإن المفهوم الذي يسمح لنا بتنفيذ أنماط المعطيات المجردة في C++ هو مفهوم *الصف class*. ويعود السبب في تسمية هذه الأنماط بالأنماط المجردة إلى أن

مستعملها لا يعرف بالضرورة تمثيلها الداخلي. وهذه ميّزة هامة تفيد أثناء كتابة البرامج التي تستعمل هذه الأنماط. فعندما يتعامل المبرمج مع الأنماط السابقة التعريف في لغة البرمجة، فإنه يستعمل وضوحاً أنماطاً مجردة. ذلك أنه عندما يعطي المبرمج نمطاً ما لمتحول، يصبح للمتحول بنية داخلية معرّفة جيداً، وتكون العمليات المسموح بها على هذه البنية مشروحة في دليل اللغة (int, char...) دون الإشارة عموماً إلى تمثيلها الداخلي. وهنالك عدة أسباب تدعم العمل مع أنماط المعطيات منها:

- يعكس البرنامج العمل الذهني للمبرمج على وجهٍ أفضل، عندما نسمح لهذا الأخير بالعمل في مستوى إدراكي أعلى.
- يعزّز تكرار استعمال الأنماط الوقايةَ من الأخطاء، إذ يعلن المبرمج نواياه بصراحة، بتحديد عدد القيم والعمليات.
- يسهّل استعمال أنماط المعطيات صيانة البرامج.

يتيح استعمال الأنماط المجرّدة للمعطيات التفكير بعمق أكبر بوظائف النمط بدلاً من بنية المعطيات الواجب تنفيذها لتحقيق هذا النمط المجرد.

3– 2 التنفيذ (التحقيق) التقليدي لنمط مجرد في C++

من أكثر الأمثلة شيوعاً وسهولة في الأنماط المجردة مثال المكدّس stack .

عند تعريف نمط مجرد يجب تحديد طبيعة المعطيات التي يعالجها النمط، والعمليات المسموح القيام بها ومعناها.

نقترح هنا تنفيذ النمط Stack حيث Stack هو مكدّس من الأعداد الصحيحة، والعمليات المسموح بها على المكدّس هي التالية:

init: ( ) → stack
destruct: (stack) →
is_empty: (stack) → boolean
push : (int, stack) → stack

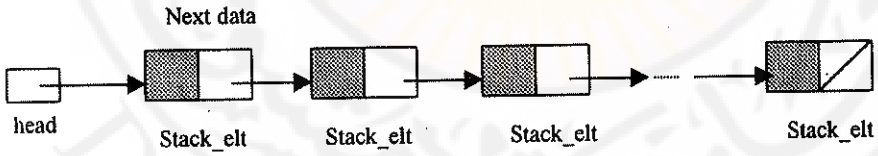
pop: (stack) → stack

top: (stack) → int

إن معنى هذه الرموز بسيط:

- *init* هي عملية لا تحتاج إلى محدّدات، وهي تعيد غرضاً من نمط *stack* أعطي القيم الابتدائية الصحيحة.
- *destruct*: هي عملية تهدم المحدّد المعطى لها من النمط *stack*.
- *is_empty*: هي قضية تقبل مكدّساً كمحدّد، وتعيد نتيجة هي صحٌ إذا كان المكدّس فارغاً وخطأٌ عند العكس.
- *push*: هي عملية تضع العنصر الصحيح الوارد في المحدّد على قمة المكدّس. والناتج هنا هو المكدّس المعدَّل.
- *pop*: هي عملية تنزع العنصر الموجود في قمة المكدّس. يُنزع هذا العنصر كلياً، ونتيجة العملية هي المكدّس الممرَّر في المحدّد بعد تعديله.
- *top*: هي عملية تعيد العنصر الموجود في قمة المكدّس دون أن تنزعه من مكانه.

يمكن تحقيق هذا المفهوم بطريقة تقليدية "طريقةC" باستعمال مفهوم التسجيلة. فتكون التسجيلة في مثالنا سلسلةً بسيطة الترابط.



الشكل 3-3: تحقيق النمط المجرد Stack على شكل سلسلة مترابطة

وفيما يلي ملف الترويسة الذي يتضمن التصريح عن نمط المعطيات مكدّس أعداد صحيحة.

```
enum bool {false= 0, true} ;
```

```
struct Stack_elt {

  int data;       // data to push

  Stack_elt* next; // pointer to the next element

};

struct Stack {

  Stack_elt* head;  // the stack is implemented as a list

};


inline void init(Stack& p) { p.head = 0;}
void destruct(Stack&);
inline bool is_empty(const Stack& p) {return (p.head == 0 ? true : false);}
inline int top(const Stack& p) {return (p.head->data);}
void push(Stack&, int);
void pop(Stack&);
```

stack1.h

جرى تعريف التوابع البسيطة ( )init و ( )is_empty و ( )top على شكل توابع ســـطرية.

يقوم التابع destruct بتحرير الذاكرة المحجوزة لكل عنصر من عناصر المكدّس باستعمال المؤثر delete بالتتالي على كل عنصر Stack_elt من المكدّس.

```
#include <stdlib.h>
#include <assert.h>

#include "stack1.h"

void destruct(Stack& p) {
  Stack_elt* tmp = p.head;
  while (tmp) {
    p.head = tmp->next;
    delete tmp;
    tmp = p.head;
  }
}
```

```
void push(Stack& p, int i) {
  Stack_elt* p_e = new Stack_elt;
  assert(p_e != 0);
  p_e->data = i;
  p_e->next = p.head;
  p.head = p_e;
}

void pop(Stack& p) {
  Stack_elt* p_e = p.head;
  p.head = p.head->next;
  delete p_e;
}
```

تمتلك الإجرائية ( )push محدّدين أحدهما عدد صحيح، والثاني يمثل المكدّس الذي يجب تكديس العدد الصحيح في قمته. يُمرّر المكدّس بالمرجع لأن هذا التابع يعدل المكدّس.

إن استعمال النمط المجرد stack سهل جداً.

```
#include <iostream.h>

#include <new.h>
#include "stack1.h"
int main () {
  set_new_handler(0);

  Stack p1,p2;
  init(p1); init(p2);        // stacks  initialization
  for(int i=0; i < 5; i++) push(p1,i);
  while (!is_empty(p1)) {
    cout << top(p1) << endl;
    push(p2,top(p1));
    pop(p1);
  }
  destruct(p1); destruct(p2);   // destruction of stacks
}
```

في هذا المثال، صُرِّح عن المكدّسين p2 و p1، ثم أُعطيا قيمة ابتدائية بصراحة باستعمال ( )init. هذه العملية أساسية، ويؤدي عدم القيام بها إلى أخطاء عند التنفيذ. توضع القيم التالية في المكدّس p1 حسب ترتيبها التالي: 4,3,2,1,0، ثم يكرّر إظهار العنصر الموجود في قمة المكدّس ونزعه، وقبل أن ينزع أي عنصر، يجري تكديسه في p2. في آخر البرنامج يُهدم المكدّسان صراحةً.

قد يستغرب القارئ أن نقوم بهدم P1 مع أن عناصره قد نزعت تماماً، ويعتقد أن هـــذه العملية لا داعي لها. من منظور تعريف النمط المجرد، لا يعتبر هذا صحيحاً. ففي الحقيقة نحن نعلم أنه في هذه الحالة بالتحديد، لا داعي لهذه العملية لأننا نعرف البنية الداخلية للمكـدّس. أما المستخدم النهائي فلا يفترض فيه أن يعرف هذه البنية. من ثَمَّ، يكون من المنطقـــي أن يجري استدعاء هادم النمط المجرد كلما دعت الحاجة إلى ذلك.

ولكن ما الفائدة من وجود عملية هدم لغرض؟ تكمن فائدة هذه العملية في كون بنـــاء الغرض يؤدي إلى تغييرات في حالة البرنامج (حجز ذواكر،....) ويجب أن تختفـــي هـــذه التغييرات عند انتهاء حياة هذا الغرض أي عند نهاية تنفيذ كتلة معينة. إن هدف الهـــادم إذن هو الإزالة المنطقية (وأحياناً المادّية) للغرض. في أكثر الحالات شيوعاً، تهدف هذه العملية إلى استعادة مساحة الذاكرة التي سبق تخصيصها للغرض وإعادتها إلى النظام.

إن الحل المقدّم آنفاً صحيح ويعمل جيداً، غير أن له سلبيّةً هامة جداً. فبنيـــة المعطيـــات المستخدمة لإنفاذ النمط المجرد، مرئية يستطيع المستخدم النهائي للنمط stack أن يصل إليـــها. ولا يوجد أي عائق يحول دون بلوغ حقول البنية المستخدمة مباشـــرةً، عـــدا الانضبـــاط الشخصي للمبرمج، وهذا ما يقيد أي تطوير مستقبلي لهذا النمط المجرد.

تلافياً لهذه النواقص في لغة C تقدم لغة C++ مفهوماً جديداً هو مفهوم *الصف class*.

3– 3 مفهوم الصف

3– 3– 1 *المبادئ الأساسية*

الصف هو نمط يتضمن وصفاً موحّداً

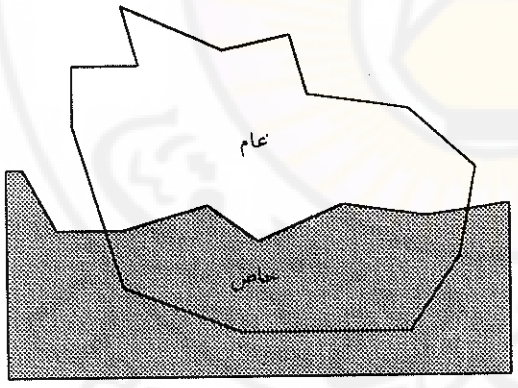
- لجمع من المعطيات التي ترتبط فيما بينها وتكون غالباً من أنماط مختلفـــة (وتســـمى "المعطيات الأعضاء").
- لجمع من الطرائق (وتسمى "التوابع الأعضاء" التي تسمح بالتعامل مـــع المعطيـــات

الأعضاء.

يسمح التصريح عن صف ما بتعريف نمط جديد تماماً، كما في حالة البنى وغيرها من الأنماط الجديدة التي يعرفها المبرمج. إن مفهوم الصف هو بطريقة ما تعميمٌ لمفهوم البنية struct بمعنى أن الصف يحتوي على حقول معطيات مضافاً إليها العمليات التي تتعامل مع هذه الحقول. وبفضل مفهوم الصف، يجري ضم المعطيات والطرائق (وهي العمليات التي تتعامل مع المعطيات) في بنية وحيدة.

إن الصف هو البنية الأساسية في لغات البرمجة الغرضية التوجه. وهو يسمح بتنفيذ نمط معطيات مجرد. يشجع مفهوم الصف على البرمجة بالمجتزآت ما دام سيجزي تنفيذ كل صف ضمن مجتزأ.

إن مفهوم الصف يذهب إلى ما هو أكثر من ذلك في فكرة البرمجة بالمجتزآت، حين يعطي المبرمج إمكان تحديد الأجزاء المرئية والأجزاء غير المرئية من النمط المجرد كما يظهر في الشكل التالي:



الشكل 3–4: تنفيذ النمط المجرد باستعمال مفهوم الصف

بإخفاء تفاصيل التنفيذ، يجب أن يقوم المبرمج بوضع عدد كافٍ من العمليات العامة التي تتعامل مع المعطيات الأعضاء، ليتمكن المستثمر من العمل بفعّالية.

يجب أن يتوفر عموماً أربعة أنواع مختلفة من التوابع الأعضاء العامة:

- البواني *Constructors*
- الهادم *Destructor*
- النوافذ *Accessors*
- المعدّلات *Modificators*

إن التوابع الأعضاء البواني تفيد في بناء عيّنات من الصف (أو مايسمى بـ*الأغراض*) وإعطائها قيماً ابتدائية على وجهٍ صحيح. على حين يسمح الهادم بهدم الغرض في نهاية حياته. أما التوابع النوافذ فهي تسمح بالنفاذ إلى المعلومات الخاصة المحتواة في الغرض (وهي المعطيات الأعضاء)، في حين تسمح المعدّلات بتغيير حالة المعطيات الداخلية في الغرض.

لتثبيت ما سبق، سنعيد تنفيذ النمط المجرد Stack اعتماداً على مفهوم الصف.

في C++ يجري تعريف الصف باستعمال الكلمة المحجوزة *class* متبوعة باسم الصف وبزوج من الأقواس { }. يتبع هذا الزوج فاصلة منقوطة أو قائمة من المتحولات. في الجزء المحصور بين القوسين يحدد المبرمج المعطيات الأعضاء والتوابع الأعضاء. في هذا المستوى يجري تنفيذ أو عدم تنفيذ إخفاء المعلومات. أي يجري هنا تحديد أي الحقول والتوابع ستكون خاصة وأيها ستكون عامة.

```
enum bool {false = 0, true};

struct Stack_elt {
  int data;       // data to push
  Stack_elt* next; // pointer to the next element
};


class Stack {
public :
  Stack();          // constructor
  ~Stack();         // destructor
```

```
  inline bool is_empty() const;
  inline int top() const;
  void push(int i);
  void pop();
private :
  Stack_elt* head;    // The stack is implemented as a list
};

inline bool Stack::is_empty() const {
  return (head == 0 ? true : false);
}

inline int Stack::top() const {
  return (head->data);
}
```

stack2.h

داخل تعريف كل صف، تكون ضمناً جميع الأعضاء (المعطيات والتوابع) خاصة، ما لم يصرّح خلاف ذلك. في حال رغبتنا في جعل المعطيات و/أو التوابع قابلة للنفاذ من الخلوج، يجب أن نضعها في قسم مسبوق بالكلمة المحجوزة public التي تتبع بالرمز " : ". وهكذا يمكن النفاذ إلى جميع الحقول والتوابع المصرّح عنها في القسم العام من قبل جميع مستخدمي الصف. هناك كلمة أخرى محجوزة وهي private للدلالة على أن المعطيات والتوابع الأعضاء التي تظهر في هذا القسم لا يمكن النفاذ إليها إلا من خلال التوابع الأعضاء في الصف نفسه. من ثَمّ لا يحق للمستخدم النهائي للصف أن ينفذ إلى المعطيات المعرّفة في القسم الخاص.

بوجهٍ عام، إذا لم ترد كلمة private أو public، فهذا يعني أن الحقول ستعامل كما لو كانت معرّفة في القسم الخاص.

في تعريف الصف الواحد، يمكن أن ترد الكلمات private و public عدة مرات وبأيّ ترتيب. ويمتد كل قسم من الكلمة المحجوزة حتى الكلمة المحجوزة التالية أو حتى نهاية الصف.

من وجهة نظر منهجية يفضّل أثناء تعريف الصف أن نُظهِر أولاً التوابع والمعطيات الأعضاء العامّة، ثم نُسرد المعطيات والتوابع الأعضاء الخاصة.

يجري التصريح عن المعطيات الأعضاء بالطريقة ذاتها التي يصرح بها عن المتحولات، عدا

أن المعطيات الأعضاء لا يمكن إعطاؤها قيماً ابتدائية. كما يمكن في صف ما وليكن اسمه X مثلاً أن يكون أحد المعطيات الأعضاء مؤشراً إلى غرض من النمط X (كما في لغة باسكال):

```
Class  X {
private:
    X*   p_x ;
 } ;
```

يمكن أن يحوي الصف تصريحات و/أو تعريفات لتوابع. ويجري التصريح عن هذه التوابع وتعريفها وفقاً للقواعد المتّبعة في الفصل السابق.

أمّا جسم التوابع الأعضاء، فيعرّف عادة تعريفاً منفصلاً في ملف منفصل. تستطيع التوابع الأعضاء النفاذ إلى جميع المعطيات في الصف سواءً أكانت خاصة أم لا. إذا عُرّف تابع داخل تعريف الصف، يعتبر هذا التابع سطرياً (inline) تلقائياً، ولو لم تذكر كلمة inline صراحةً قبل تعريفه! يمكن أيضاً تعريف تابع عضو كتابع سطري عندما يُعرّف خارج الصف.

إذا عُرّف تابع عضو خارج الصف، يجب أن يُسبَق معرّف التابع باسم الصف الذي يُصرّح عن التابع. ويستعمل المؤثر " :: " لهذه الغاية.

```
#include <stdlib.h>
#include <assert.h>
#include "stack2.h"

Stack::Stack() {
  head = 0;
}

Stack::~Stack()
{

Stack_elt* tmp = head;

  while (tmp) {
```

```
head = tmp->next;
  delete tmp;
  tmp = head;

}

}

void Stack::push(int i) {

 Stack_elt* p_e = new Stack_elt;
 assert(p_e != 0);
 p_e->data = i;
 p_e->next = head;
 head = p_e;
}


void Stack::pop() {

 Stack_elt* p_e = head;
 head = head->next;
 delete p_e;
}
```

يفيد استخدام الكلمة المحجوزة inline داخل الصف في تسهيل قراءة الترميز. ويجب أن تُعرّف التوابع الأعضاء المعرَّفة inline في الملف الذي يصرّح عنها.

يُسمح بتعريف تابع عضو كثابت. في مثالنا، يوافق ذلك حالة التابعين is_empty( ) و top( ). يعني هذا أن التابع المعني لا يمكنه تغيير قيم المعطيات الأعضاء في الصف، ولا إعادة مرجع غير ثابت أو مؤشر غير ثابت لإحدى المعطيات الأعضاء[2]. يعزّز هذا النوع من

[2] سنرى استثناء لهذه القاعدة في الفقرة 3. 6.

التصريحات المراقَبة التي يقوم بها المترجم، ويوفّر برمجةً أضمن، وذلك بزيادة وسائل تنفيذ البرمجة الدفاعية.

فيما يتعلق بالمستخدم النهائي للصف، يجري الوصول إلى الأعضاء (المعطيات والتوابع) تماماً مثل الوصول إلى الحقول في بنية. يجب مع ذلك أن نتذكر أن الوصول غير مسموح به إلاّ إلى الأعضاء العامة للصف. إذن، يكون التابع العضو مطبّقاً دائماً ضمنياً على غرض مخصص من الصف.

```
#include <iostream.h>
#include <new.h>
#include "stack2.h"
int main() {
  set_new_handler(0);
  Stack p1,p2;

  for(int i=0; i < 5; i++) p1.push(i);
  while (!p1.is_empty()) {
    cout << p1.top() << endl;
    p2.push(p1.top());
    p1.pop();
  }
}
```

هذا البرنامج هو نفسه البرنامج الوارد في الفقرة 3. 2 من هذا الفصل ، ولكنه هنا يستعمل مفهوم الصف.

p1 وp2 غرضان مخصّصان من الصف Stack المعرَّف في stack2.h. وعليه، فإن جميع الحقول والطرائق الموضوعة في الجزء العام من الصف قابلة للنفاذ.

3- 3- 2 *البواني والهوادم*

لابد أن يكون القارئ اليقظ قد لاحظ أنه في مثال استعمال الصف Stack لم تعد مؤثرات ابتداء وهدم الغرضينp1 وp2 ظاهرة صراحةً.

في الحقيقة، يُنفّذ المترجم هذه العمليات بطريقة ضمنية وآلية. عندما نأخذ غرضاً من صف، يُستدعى *الباني* المرتبط بهذا الصف، وهذا ما يضمن إعطاء قيمة ابتدائية صحيحة

للغرض.

يمكن تمييز باني الغرض عن غيره من التوابع بكونه يحمل اسم الصف نفسه. في مثالنا هذا، الباني هو التابع ( ) Stack الذي ليس له موسطات. لا يقبل الباني أي نمط عودة (ولا حتى من النوع void)، ولا يُسمح بالعمل مع عنوان الباني. يمكن أن يتوفر عدة بوانٍ في الصف، وبالطبع، يختلف توقيع هذه البواني، كما يُسمح بتعريف بوانٍ تقبل قيماً افتراضية لمحدّداتها. والباني كما رأينا سابقاً، يستدعيه آلياً نظام البرمجة C++ عندما يُنشأ متحول من نمط الصف، أو عندما ينشئ النظام ضمنياً متحولات موقتة من نمط الصف، أو عندما ننشئ غرضاً من الصف ديناميكياً باستعمال المؤثرnew.

نطلق اسم *الباني الافتراضي default constructor* في الصف على الباني الذي لا يتضمن أية محدّدات، أو الباني الذي تُعطى قيم افتراضية لجميع محدّداته.

إذا لم يحدّد المبرمج في تعريفه للصف أي بانٍ، يولّد المترجم بانياً افتراضياً. تكمن فائدة هذه العملية في كون الباني الافتراضي يقوم بانتظام باستدعاء جميع بواني المعطيات الأعضاء إن وجدت. ويعتبر هذا الأمر مفيداً عندما تكون المعطيات الأعضاء في الصف من نمط صفوف أخرى.

أما هادم الغرض، فهو يحمل اسم الصف ذاته، مسبوقاً بالرمز "~". وكما في البواني، لا يقبل الهادم نمط عودة. إن الهادم لا يقبل محدّدات، ومن ثَمّ لا يمكن زيادة تحميله.

إن دور الهادم هو أن يقوم بجميع أعمال الصيانة اللازمة بعد أن ينتهي "وجود" الغرض في ذاكرة الحاسوب. وهذا العمل يهدف تحديداً إلى استرداد الذاكرة المحجوزة ديناميكياً للغرض الذي سوف يختفي.

عند نهاية تنفيذ كتلة، يستدعي نظام البرمجةC++ هادم الغرض آلياً، وذلك لكل متحول من نمط صف معرّف في هذه الكتلة، أو خلال استعمال المؤثر *delete*. كما يستدعي النظامُ الهادمَ للتخلص من جميع الأغراض الموقتة التي أنشأها دون علم المبرمج. ويسمح باستدعاء هادم الغرض صراحةً (في حالات نادرة) وذلك على نقيض البواني.

إن استدعاء البواني والهوادم يجري آلياً في C++. ولهذا النمط من التشغيل فائدة كبيرة، إذ يتلافى نسيان المبرمج الساهي لهاتين العمليتين الحيويتين. أما سلبية هذه الطريقة فهي أن العديد من الأشياء الهامة تجري آلياً دون برمجة صريحة، وهذا ما قد يفاجئ المبرمج غير المتنبه ويشعره بالدهشة عند تنفيذ البرنامج خطوة خطوة.

تتحلّى البنى struct في C++ بالصفات التي رأيناها في الفصل السابق، أي إنها الصفات ذاتها الموجودة في لغة C. إضافةً إلى ذلك، يمكن أن تستعمل البنية كصف- في هذه الحالة نستعمل طبعاً الكلمة *struct* بدلاً من *class*. غير أن هناك فرقاً جوهرياً بينهما، وهو أنه في البنية تكون جميع الأعضاء (توابع ومعطيات) عامةً public ضمنياً، وذلك للحفاظ على التجانس مع C.

من وجهة النظر المنهجية، لا ينصح باستعمال مفهوم البنية لتنفيذ صف. ويفضل استعمال البنى في الحالات التقليدية، واستعمال مفهوم الصف عندما نريد تنفيذ نمط معطيات مجـــرد وذلك للحؤول دون الوقوع في الالتباس.

3-3-3 الصفوف المتداخلة

يبين المثال السابق تنفيذ النمط المجرد stack باستعمال مفهوم الصف. وقد استعنّا فيه ببنيـة مساعِدة هي Stack_elt لتحقيق المكدّس على شكل سلسلة بسيطة الترابط.

ومع أن هذه السلسلة لا يمكن الاستغناء عنها، فإنها تحمل سلبية كونها قابلة للنفاذ من قبل المبرمج بمجرد أن يقوم باحتواء ملف الترويسة. لا يعتبر هذا الأمر صحّياً من وجهـــة نظـــر منهجية، إذ إن الهدف من إخفاء المعلومات هو إخفاء جميع التفاصيل المتعلقة بتنفيـــذ بنيـــة المعطيات المجردة.

يمكن حل هذه المسألة بتعريف صف هوStack_elt وتضمين هـــذا الصـــف Stack_elt في الجزء الخاص من الصف stack (يسمى هذا بتداخل الصفوف).

```
enum bool {false = 0, true};

class Stack {
public :
```

```
  Stack();          // constructor
  ~Stack();         // destructor
  inline bool is_empty() const;
  inline int top() const;
  void push(int i);
  void pop();
private :
  class Stack_elt {
  public :
    int data;       // data to push
    Stack_elt* next; // pointer on the next element
  };
  Stack_elt* head;     // The is emplemented in the form of a list
};

inline bool Stack::is_empty() const {
  return (head == 0 ? true : false);
}

inline int Stack::top() const {
  return (head->data);
}
```

stack3.h

يسمح بتداخل الصفوف والبنى والأنماط المعدودة، ...إلخ ضمن صفوف أخرى أو بنى أخرى. وبهذه الكيفية لا ترى هذه الأسماء مباشرة إلا ضمن الصف الذي يحتويها، ومن ثمّ، فهي لا "تلوّث" فضاء الأسماء الشمولي.

وعلى عكس ما يمكن أن نتصوره حدسياً، فإن الصف المتضمِّن لا يملك حقوقاً مميّزة على الصف المضمَّن داخله. لذا، قمنا في هذه الحالة بالتصريح عن صف متداخل في الجزء الخاص من الصف المتضمِّن. جرى التصريح عن جميع أعضاء الصف الداخلي على أنها عامة بحيث يتمكن الصف المتضمِّن من النفاذ إليها. وبالمثل، فالتوابع الأعضاء في صف مضمَّن ليس لها حق النفاذ إلى المعطيات الأعضاء في الصف المتضمِّن إلّا إذا كانت هذه المعطيات الأعضاء موجودة في فقرة عامة.

عند العمل بهذه الطريقة، لن يتمكن المستخدم النهائي من إنشاء أغراض اعتباراً من الصف المضمَّن. غير أنه يمكنه فعل ذلك باستعمال مؤثر إبطال المدى وبتوضيح "مسار النفاذ"

إلى الصف المضمَّن. وعندما نعرّف توابع أعضاء جرى التصريح عنها في صف مضمَّن، يجب أن نصف اسم التابع وصفاً كاملاً.

لو جرى التصريح عن الصف الداخلي في فقرة عامة من الصف المتضمِّـــن، لاســـتطاع المستخدم النهائي النفاذ إليه، إن علم بوجوده. بناءً على ماسبق، فإن التصريح التالي:

```
Stack::Stack_elt* p_pelt ;
```

غير قانوني لأن Stack_elt صرّح عنها في الفقرة الخاصة من الصف Stack.

في الحقيقة، يمتلك المبرمج بديلاً عن استعمال الصفوف المتداخلة، وسنرى ذلك مـــن خلال نسخة جديدة من النمط المجرّد Stack.

3- 3- ٤ الصفوف ذات المعطيات الأعضاء من نمط صف

في هذه النسخة الجديدة، نبين كيف يمكن أن يتضمن صف مـــا حقـــولاً كمعطيـــات أعضاء، هي من نمط صف مصرّح عنه سابقاً.

لمّا كنا نستعمل سلسلة بسيطة الترابط لتنفيذ النمط Stack، فإنه يحتمل أن يكون لدينا نمط مجرد موجود سابقاً يسمى List. والعمليات المتاحة على هذا النمط هي:

```
init: ( ) → List
destruct: (List) →
is_empty: (List) → boolean
exist: (List, int) → boolean
insert : (int, List) → List
eliminate: (List, int) → List
eliminate_fr: (List) → List
first: (List) → int
```

إن ( )init و ( )destruct هما على الترتيب الباني والهادم للنمط المجرد List. أما ( )is_empty و ( )exist فهما قضيتان: الأولى تتحقق من كون القائمة المعطاة فارغة أو لا، والثانية تتحقق من كون العدد الصحيح المعطى في المحدّد موجوداً في القائمة أو لا. نذكر هنا بأننـــا نـــأخذ الحالة المبسطة لمكدّس (قائمة) من الأعداد الصحيحة. التابع ( )insert يدرج العدد الصحيـــح المعطى في المحدّد في رأس القائمة المعطاة، والناتج هو قائمة جديدة. أما ( )eliminate فهو تابع

يحذف العنصر الأول الذي يجده في القائمة والذي تكون قيمته مساوية لقيمة المحدّد المعطـى. إذا لم تكن هذه القيمة موجودة في القائمة فلا يحصل شيء. التابع eliminate_fr( ) يحـــذف العنصر الموجود في رأس السلسلة. أما التابع first( ) فيعيد قيمة العنصر الأول في القائمـة دون تغييرها.

```
typedef  int Type;

enum bool {false = 0,true};

class Node {
  friend class List;
private :
  inline Node(Type, Node* n = 0);
  Type val;
  Node* next;
};

class List {
public :
  inline List();
  ~List();
  void insert(Type);
  void eliminate(Type);
  void eliminate_fr(void);
  void display();
  inline Type first() const;
  inline bool is_empty() const;
  bool exist(Type);
private :
  Node* l;
};

inline Node::Node(Type x ,Node* n) {
  val = x;
  next = n;
}

inline List::List() {
  l = 0;
};
```

```
inline bool List::is_empty() const {
  return l == 0 ? true : false;
}

inline Type List::first() const {
  return l->val;
}
```

list1.h

في هذا التحقيق، نستعمل مفهوماً جديداً في C++ وهو مفهوم الصديق friend، إذ يسمح بالتصريح عن صف أو تابع بأنه "صديق" لصف آخر. لنأخذ هنا الحالة التي نعرّف فيها صفاً صديقاً لصف آخر. إن الصف Node الذي صرّح عن كل أعضائه على أنها خاصة، يصرّح أن الصف List هو صف صديق له. يمكن أن يجري مثل هذا التصريح في أي قسم من الصف ويجب أن يبدأ بالكلمة المحجوزة friend. من وجهة نظر أسلوب البرمجة، يفضل كتابة جميع التصريحات عن الأصدقاء في بداية تعريف الصف، وقبل الفقرة العامة.

إن التصريح عن صف ثان كصديق لصف أول يسمح للصف الأول بالنفاذ إلى الأعضاء الخاصة في الصف الثاني كما لو كانت توابع أعضاء. تسمح هذه الآلية إذن بحل مسألتنا السابقة. يكفي هنا أن نعرف جميع الأعضاء الخاصة بحيث نمنع المستثمر النهائي من النفاذ إلى الصف Node، على حين يكون للصف الصديق كامل الحقوق للنفاذ بحرّية. ومع ذلك، يُفضّل أن يستعمل القارئ مفهوم "الصديق" بتقتير، إذ تودي إساءة استعماله إلى ثغرات في بناء البرنامج. في الواقع، يزيد تعريف الصفوف "الصديقة" من الترابط الموجود بين المجتزآت المختلفة، ويضر بالجودة العامة للبرنامج (صعوبة أكبر عند الصيانة...). ولن نستعمل مفهوم الصديق إلا في حالات مشابهة لحالتنا هذه، أو، كما سنرى لاحقاً، عند زيادة تحميل الموثرات.

في المثال السابق، جرى تعريف باني الصف Node سطرياً. هذا وإن تصريح الباني في القسم الخاص من الصف يمنع المستثمر النهائي من إنشاء أغراض من هذا النوع.

مثال:

```
# include " list1.h "
```

```
int  main ( )  {
        Node  n (5 ) ;    // error :    Node : : Node ( ) inaccessible!
}
```

```
#include <iostream.h>

#include "list1.h"


List::~List() {
  Node* tmp = l;
  while (tmp) {
    l = tmp->next;
    delete tmp;
    tmp = l;
  }
}


void List::insert(Type x) {
  Node* p = new Node(x,l);
  l = p;
} ;

bool List::exist(Type x) {
  if (l == 0) return false;
  Node* p = l;
  bool found = false;
  while ((p) && (!found))
    if (p->val == x)
      found = true;
    else
```

```
    p = p->next;
  return found;
}

void List::display() {
  cout << "( ";
  for (Node* p = l; p; p = p->next) cout << p->val << " ";
  cout << ")" << endl;
}

void List::eliminate_fr(void) {
  if (l == 0)
    return;
  else {
    Node* p = l;
    l = l->next;
    delete p;
  }
}

void List::eliminate(Type x) {
  if (l == 0) {
    cout << "empty List !" << endl;
    return;
  }
  // research of the element to eliminate
  Node* p = l;
  Node* pred = 0;
  bool found = false;
  while ((p) && (!found))
    if (p->val == x)
      found = true;
    else {
      pred = p;
      p = p->next;
    }
  if (!found) {
    cout << x << " does not exist in the list !" << endl;
    return;
  }
  else { // eliminate the found element, pointed by p
    if (pred) { // the element to eliminate has a predecessor
      (pred->next) = (p->next);
      delete p;
    }
    else {     // the element to eliminate is the first in the list
```

```
    l = (p->next);
    delete p;
   }
  }
}
```

نبين فيما يلي استخداماً بسيطاً للنمط المجرد List:

```
#include <iostream.h>

#include "list1.h"

void main()
{
  List l;
  Type i;

  cout << "Enter integers, 0 to finish : " << flush;
  cin >> i;
  while (i) {
  l.insert(i);

   cin >> i;

  }
  if (!(l.is_empty()))
    l.display();
  else
    cout << "empty list !" << endl;
  cout << "Research in the list, 0 to finish : " << flush;
  cin >> i;
  while (i) {
    if (l.exist(i))
          cout << i << " exist in the list" << endl;
    else
          cout << i << " does not exist in the list" << endl;
    cout << "Research in the list, 0 to finish : " << flush;
    cin >> i;
  }

  cout << "Elimination from the list, 0 to finish : " << flush;
  cin >> i;
  while (i) {
    l.eliminate(i);
    l.display();
    cout << "Elimination from the list, 0 to finish : " << flush;
    cin >> i;
```

```
  }
}
```

عند التنفيذ نحصل على:

```
Enter integers, 0 to finish : 1 23 4 5 6 1 3 0
( 3 1 6 5 4 23 1 )
Research in the list, 0 to finish : 1
1 exist in the list
Research in the list, 0 to finish : 4
4 exist in the list
Research in the list, 0 to finish : 0
Elimination from the list, 0 to finish : 4
( 3 1 6 5 23 1 )
Elimination from the list, 0 to finish : 1
( 3 6 5 23 1 )
Elimination from the list, 0 to finish : 23
( 3 6 5 1 )
Elimination from the list, 0 to finish : 0
```

يفيدنا هذا التنفيذ للنمط المجرد List عند بناء النمط المجرد Stack بناءً جديداً.

```
#include "list1.h"


class Stack {

public :

 inline Stack();          // constructor

 inline ~Stack();         // destructor

 inline bool is_empty() const;

 inline int top() const;
 inline void push(int);
 inline void pop();

private :
 List head;      // the stack is implemented as a list
};

inline Stack::Stack() {}

inline Stack::~Stack() {}
```

```
inline void Stack::push(int i) { head.insert(i); }

inline void Stack::pop() { head.eliminate_pr(); }

inline bool Stack::is_empty() const {
  return head.is_empty();
}

inline int Stack::top() const {
  return head.first();
}
```

stack4.h

يثير استعمال النمط List في الصف Stack عدة تساؤلات: ماهو الترتيب الذي تُستدعى فيه بواني وهوادم المعطيات الأعضاء؟

تُنَفَّذ بواني المعطيات الأعضاء وفق ترتيب تصريحها، قبل باني الصف الذي يتضمن تعريف المعطيات الأعضاء. أما هوادم المعطيات الأعضاء فتنفّذ بالترتيب المعاكس لترتيب البناء، بعد أن ينفذ هادم الصف الذي يصرح عن المعطيات الأعضاء. تطبق هذه القواعد عودياً إذا كانت المعطيات الأعضاء في نمط صف تتألف هي نفسها من معطيات أعضاء من نمط صف. نبين هذا الأمر فيما يلي:

```
#include <iostream.h>

[]
class X {
[]
public :
[]
  X()   {cout << "Constructor  X" << endl;};
  ~X()   {cout << "Destructor  X " <<endl;};

private :
  int m1;
};

class X {
public :
  X()   {cout << "Constructor  Y" << endl;};
  ~X()   {cout << "Destructor  Y " <<endl;};
```

```
private :
  X m2;
};

int main( ) {
Y vy;
}
```

عند التنفيذ يظهر على الشاشة مايلي:

```
Constructor X
Constructor Y
Destructor Y
Destructor X
```

وهكذا، في حالتنا يُستدعى بانى الصف List قبل بانى الصف Stack، علماً بأن هذا الأخير فارغ.

أما بقية تنفيذ النمط المجرّد Stack باستعمال النمط المجرد List فهي لا تحتوي على صعوبات خاصة.

3- 3- 5 الصفوف العمومية *generic classes*

أخيراً سنعرض هنا نسخة أخيرة من النمط Stack، وهي نسخة عمومية. يبيّن هذا المثال أنه من الممكن ضم مفهوم الصف إلى مفهوم القالب template، وهذا ما يسمح لنا بتحقيق العديد من الأنماط المجردة. في الواقع، إن الأشكال المختلفة لتحقيق النمط Stack التي عرضناها حتى الآن كانت حالات مبسّطة لأنها لم تكن تعمل إلا في حالة الأعداد الصحيحة.

يسمح لنا هذا المثال بإظهار إمكانات أخرى تقدّمها لغة C++. إن النحو المستعمل في مفهوم القالب مماثل للنحو الذي رأيناه في الفصل السابق عندما عرضنا التوابع العمومية.

يبدأ الصف العمومي بالكلمة المحجوزة "template" وتتبعها قائمة محدّدات القالب بين رمزي "<" و ">". تُفصل محدّدات القالب بفواصل. تفصل الفواصل بين المحدّدات. كل محدّد هو إما زوج من الشكل "*class* identifier" أو تصريح عادي عن موسط. إن اختيار معيّن هوية الصف أي (identifier) اختيار حر ويمثل النمط العمومي. عند تعريف الصف يمكن

بالطبع الرجوع إلى النمط العمومي من طريق معيّنات الهويّة الموجودة في ترويسة الصف.

عند تعريف أي تابع عضو خارج الصف، يجب أن يُسبق اسم التابع العضو بمرجع يدل على الصف العمومي الذي يصرّح عنه، إضافةً إلى الصفة *template*. يُكتب إذن الاسم الكامل للتابع العضو بضم اسم التابع العضو مع محدّداته الصورية إلى قائمة معيّنات هوية النمط الموضوعة ضمن الرمزين "<" و ">" في التابع. إن هذه المعيّنات المفصولة بالفواصل هي تلك التي استعملها المبرمج في الفقرة template *template* <list of arguments> التي تسبق اسم التابع العمومي، وهي ليست بالضرورة المعيّنات المستعملة عند تعريف الصف. يمكن أن تبدو هذه الرموز صعبة ومتعبة للوهلة الأولى، ولكنها في الحقيقة منطقية ومتجانسة.

سنستعمل في هذا التحقيق الجديد للنمط المجرد Stack مفهوم الجدول. يُظهر تحقيق المكدّس على شكل جدول بضعة مشاكل.

فمادام من الواجب أن يكون حجم الجدول معروفاً في لحظة الترجمة، يجدر إذن التعامل مع عنوان العنصر الأول من الجدول p، وحجز الذاكرة اللازمة لبقية عناصر الجدول ديناميكياً. ولما كان هذا الحجز يمكن أن يحصل بطريقة ديناميكية، فليس من الضروري أن يثبت المستخدم حداً أعلى في لحظة تحرير البرنامج.

لهذا السبب، سيزوّد النمط المجرد Stack ببانيين، وهما الباني الافتراضي وبانٍ آخر يمكّن المبرمج من تثبيت حجم المكدّس. كان من الممكن هنا أن نقتصر على بانٍ يعطي قيمة افتراضية للمجدّد، ولكن التحقيق المقترح يبين كيف يمكن زيادة تحميل الباني. قبل كتابة البواني، لنركّز على الجزء الخاص من الصف. يخفي هذا الجزء إمكانات لم نستثمرها بعد.

```
#include <iostream.h>
enum bool {false = 0, true};

template <class Type>
class Stack {

□
public :
□
  Stack();          // default constructor
```

```
 Stack(int);        // constructor with precise size
 ~Stack();          // destructor
 inline bool is_empty() const;
 inline Type top() const;
 void push(Type);
 inline void pop();

private :
 enum constants { EMPTY_STACK = -1, MAX_DEFAULT = 50 };
 Type* p;           // the stack is implemented as a table
 int pp;            // pointer on the top of the stack
 const int MAX_STACK; // maximal size of the stack
};

template <class Type1>
Stack<Type1>::Stack() : MAX_STACK(MAX_DEFAULT){
          p=new Type1[MAX_DEFAULT];
          pp=EMPTY_STACK;
}


template <class Type>
Stack<Type>::Stack(int max) : MAX_STACK(max) {
          p=new Type[max];
          pp=EMPTY_STACK;
}


template <class Type>
Stack<Type>::~Stack() {
 delete [] p;
```

```
□
}
□


□
template <class Type>
□
void Stack<Type>::push(Type t) {
□
 pp++;
□
 if (pp < MAX_STACK)
□
   p[pp] = t;
 else
   cerr << "full stack" << endl;  // simple treatment !
}

template <class Type>
inline void Stack<Type>::pop() { pp-- ;}

template <class Type>
inline bool Stack<Type>::is_empty() const {
 return (pp == EMPTY_STACK) ? true : false ;
}

template <class Type>
inline Type Stack<Type>::top() const {
 return p[pp];
}
```

stack 5.h

نلاحظ أولاً أن الصف Stack يحتوي تعريفاً محلياً لنمط معدود في الصف. يرى أعضاء الصف هذا التصريح مباشرةً. وليس للاسم المعطى له أي تأثير في فضاء الأسماء خارج الصف.

وبالطريقة نفسها، يسمح بالتصريح عن تعاريف typedef محلية بالنسبة إلى الصف. ويتميز الصف Stack بأنه يتضمن عنصر معطيات عضو ثابت. في الواقع، يكون الحجم الأعظمي للمكدّس ثابتاً فعلياً بعد بناء المكدّس. ولما كان الباني هو الذي يبني أية عيّنة (عنصر مخصّص)، وقع على عاتقه تهيئة الثابت.

يجب أن يهيأ عنصر معطيات عضو ثابت في غرض باستعمال بواني الصف. لذا، يُتبـــع توقيعُ الباني الرمزَ " : " يليه معيّن هوية الثابت وقيمته الموضوعة بين قوسين. في حالة احتـــواء الصف على عدة ثوابت، يتبع الرمزُ " : " قائمةَ الأزواج identifier (initial value) التي يُفصل بعضها عن بعضها بالفواصل.

ينصح بهذا الترميز عموماً في البواني لتهيئة جميع المعطيات الأعضاء في الغرض في الملـــف stack1.h من النسخة السابقة، كان من الممكن كتابة:

```
inline List :: List ( ) : l(0) { } ; // l is initiated with 0
```

وفي ملف الترويسة Stack5.h كان يفضل كتابة:

```
template <class Type1>
Stack<Type1>::Stack() :
        p(new Type1[MAX_DEFAULT]), pp(EMPTY_STACK),
MAX_STACK(MAX_DEFAULT) {
}

template <class Type>
Stack<Type>::Stack(int max) : p(new Type[max]), pp(EMPTY_STACK),
MAX_STACK(max) {
      }
```

يعود السبب في ذلك، إلى أن الترميز الذي يولّده المترجم، هو غالباً أكثر فاعلية، ولكـــن هذه الطريقة قد تكون، وللأسف، مصدراً لخطأ يصعب كشفه. مثال:

```
class Vector {
public:
   Vector (int=20);
private:
   double* v;
   int size;
};

Vector::Vector (int  t) : size(t), v(new double [size]) {
}
```

لا يعمل هذا الباني جيداً لأن المعطيات الأعضاء في الغرض تهيأ بترتيب التصريح عنـها، لا بالترتيب الذي كتبه المبرمج. وهكذا، ففي هذه الحالة تهيـــأ v بالقيمـــة الموجـــودة في size (الترتيب المعرّف في الصف) ولو أن المبرمج يحدد تهيئة size قبل تهيئة v (ترتيب محدّد سلفاً في

الباني). تعطي بعض المترجمات تحذيرات من هذا النوع من الأخطاء التي يصعب كشفها لاحقاً.

لنعد إلى مثال المكدّس. يبني الباني الافتراضي مكدّساً من MAX-DEFAULT عنصراً، في حين يبني الباني الآخر مكدّساً باستعمال القيمة المعطاة عن طريق المحدّد.

رأينا سابقاً أن المؤثر new يستعمل لحجز مصفوفة من العناصر من نمط محدد. لذا، يكفي أن تُتبع المؤثر new اسمَ النمط، يليه عدد العناصر المرغوب حجزها في المصفوفة، موضوعاً ضمن قوسين [ ]. إذا كان النمط صفاً، يُستدعى الباني الافتراضي لهذا الصف لكل عنصر من عناصر المصفوفة المحجوزة بهذه الطريقة. يعيد المؤثر عنوان العنصر الأول في المصفوفة.

أما هدم الشعاع المحجوز بهذه الطريقة، فيجري عن طريق استعمال شكل خاص من أشكال المؤثر delete، يجري فيه وضع القوسين [ ] بين المؤثر delete واسم المصفوفة، وذلك لإعلام المترجم بأنه يجب أن يهدم مصفوفة من العناصر، وأن يُطبَّق الهادم على جميع عناصر هذه المصفوفة.

يجري الحصول على مكدّسات خاصة بطريقة سهلة. فعند التصريح عن متحول من النمط Stack، يُتبع المعيّنُ Stack الأنماطَ المختارة موضوعة بين الرمزين "<" و ">".

```
#include <iostream.h>
□
#include <string.h>
□
#include "stack5.h"
□

□
int main1() {
  Stack<int> p1,p2;   // two stacks of integers, the default size of each is 50

  for(int i=0; i < 5; i++) p1.push(i);
  while (!p1.is_empty()) {
    cout << p1.top() << '\t';
    p2.push(p1.top());
    p1.pop();
  }
  cout << endl;
```

```
Stack<int> p3(6);   // declare a stack of 6 integers
while (!p2.is_empty()) {
  cout << p2.top() << '\t';
  p3.push(p2.top());
  p2.pop();
 }
 cout << endl;

/* inverse a word using a complicated way */
 cout << "Enter a word : " << flush;
 char word[50];
 cin >> word;
 Stack<char> p4(strlen(word));  // stack of char, with a variable size !
 char c;

 for (int i=0;c=word[i];i++) p4.push(c);
 cout << "Here is the inverted word : " << flush;
 while (!p4.is_empty()) {
   cout << p4.top();
   p4.pop();
  }
}
```

في هذا المثال، جرى تخصيص مكدّسين، هما مكدّسان للأعداد الصحيحة p1 ، p2 باستخدام الباني الافتراضي الذي يقوم بحجز خمسين عنصراً. أما المكدّس p3 فهو كذلك مكدّس أعداد صحيحة معرّف باستعمال الشكل الثاني من البواني ولا يحتوي سوى ستة عناصر على الأكثر. يقوم الجزء الأخير من المثال ببناء مكدّس من المحارف بحجم يعادل حجم الكلمة التي يراد تكديس محارفها. وفيما يلي تنفيذ البرنامج السابق.

```
4       3       2       1       0
0       1       2       3       4
Enter a word :    Hello
Here is the inverted word :  olleH
```

في هذه الفقرة، رأينا العديد من الإمكانات الجديدة التي تقدّمها C++، ولكن لم نصل بعد إلى نهاية استكشافاتنا لهذه اللغة. في الفقرة التالية سنرى كيف يمكن إجراء تحقيق لنمط مجرد بحيث يتكامل على أفضل وجهٍ ممكن مع لغة C++.

3- 3- 6 زيادة تحميل المؤثرات

بيّن لنا تحقيق النمط المجرد Stack بعداً جديداً في إمكانات لغة C++. في الحقيقة تتوفر في C++ إمكانات أكبر تسمح للمبرمج بزيادة تحميل بعض المؤثرات المعرّفة سابقاً. وفيما يلي قائمة المؤثرات التي يمكن زيادة تحميلها في C++.

+	-	*	/	%	^	&	|	~	!
,	=	<	>	<=	>=	++	--	<<	>>
==	!=	&&	||	+=	-=	/=	%=	^=	&=
|=	*=	<<=	>>=	[ ]	( )	->	->*	new	delete

new [ ] delete [ ]

يمكن زيادة تحميل المؤثرات "+" و"-" و "*" و "&" بشكلها الأحادي أو الثنائي. أما المؤثرات التي لا يمكن زيادة تحميلها فهي:

:: .* . ?: sizeof

ولئن كانت زيادة تحميل المؤثرات ممكنة، إنه لا يمكن تغيير قواعد الأولوية الخاصة بها. كذلك الأمر فيما يتعلق بالخاصة التجميعية للمؤثرات، وبعدد المعاملات الخاصة بها إذ لا يمكن تعديلها. ثُمّ إنه لا يمكن بناء مؤثرات جديدة.

سنستعمل في هذا الجزء مثالاً للنمط المجرد "سلسلة المحارف" أو ما يسمى String. هذا النمط المجرد مفيد جداً، إذ يعلم المبرمجون بلغة C أن التعامل مع سلاسل المحارف يكون غالباً مصدراً للعديد من الأخطاء.

إن مثال النمط المجرّد String مثال تقليدي، غير أنه يسمح بإظهار إمكانات اللغة ومزالقها إظهاراً مثالياً.

في جميع النسخ التي سنعرضها في هذا الفصل، سيجري تحقيق النمط String ببساطة على شكل مؤشر إلى char. وسنهتم في المقام الأول بالتوابع الأعضاء.

تتألف النسخة الأولى من الصف String من بانيين، وهادم ومن زيادة تحميل المؤثر الإسناد.

```
class String {
public :
  String(const char* = "");        // default constructor
  String(const String&);           // copy constructor
  ~String();                       // destructor
  String& operator=(const String&); // overloading of the assignment operator

private :
  char* ch;
};
```

String1.h

الباني الأول:

String (const char* = " ");

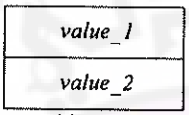
هو بانٍ افتراضي إذ يمكن استعماله دون تحديد المحدّدات. وهو يقبل محدّداً وحيداً محتملاً هو مؤشر إلى char. إذا لم يحدّد أي محدّد، يقوم الباني بتهيئته وإعطائه القيمة الافتراضية وهي سلسلة المحارف الفارغة " ". ثُمّ إن الباني يجب أن يهتم أيضاً بحجز الذاكرة اللازمة لاحتواء سلسلة المحارف المرّرة ضمن المحدد.

أما الباني الثاني، فهو بانٍ خاص يسمى باني النسخ (copy constructor). يستدعى النظام C++ باني النسخ عندما يكون عليه أن ينشئ غرضاً اعتباراً من غرض موجود من الصف نفسه. ويستعمل النظام هذا الباني عندما يُمرّر غرضاً من الصف بالقيمة كمحدّد لتابع، أو عندما يعيد التابع غرضاً من صف.

إن باني النسخ في صف ما X يجب أن يكون له توقيع من الشكل X(X&) أو X(const X&) وهو الأفضل.

إن لم يوصّف المبرمج باني النسخ أو نسي تعريفه (وقد رأينا في مثال المكدّس أنه لا يوجد بانٍ للنسخ)، فإن النظام يقوم بتوليد باني النسخ آلياً. يقوم باني النسخ الذي يولّده النظام

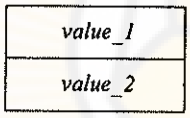
بنسخ كل عنصر معطيات عضو إلى عنصر معطيات عضو، وذلك عودياً عندما تكون المعطيات الأعضاء من نمط صف. تخطيطياً، يكون لدينا الوضع الحالي: ليكن لدينا الغرض object_X غرضاً من الصف X بعنصري معطيات أعضاء قيمتهما على الترتيب value_1 و value-2.



الشكل 3–5: object_X ذو عنصرين عضوين هو عيّنة من الصف X

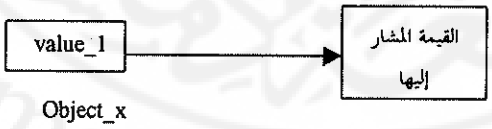
إذا مُرّر هذا الغرض كمحدّد لتابع يقبل غرضاً من النمط X كموسط ممرّر بالقيمة، وليكن التابع هو مثلاً: void f(X a) {.....}

عندئذ، في لحظة الاستدعاء f(object_X)، يجري بناء المحدّد a وتعطى معطياته الأعضاء قيماً ابتدائية آلياً، وهي نسخة من قيم المعطيات الأعضاء بالترتيب الموافق.



الشكل 3–6: محدّد التابع f هو نسخة مطابقة للغرض object_X

كان يمكننا الاكتفاء بهذا التصرف ولكن، ويالالأسف، للأمور خفايا أكثر مما يظهر علناً في الواقع، غالباً ما يحدث أن يكون عنصر المعطيات العضو هو مؤشر إلى قيمة.

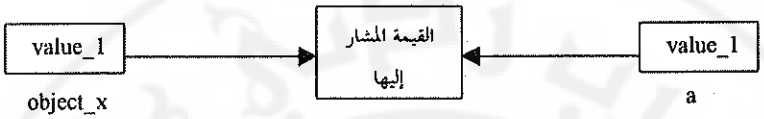


الشكل 3–7: الغرض object_X فيه عنصر معطيات عضو من نمط مؤشر

لنفترض الآن أن الصف X يقبل هادماً من النمط:

X :: ~X ( ) { delete data member; }

إن حصل كما في الحالة السابقة أن يستدعى تابع f( ) ذو محدّد object_X، وأن يطبـــق مبدأ النسخ السابق ذاته (عضو إلى عضو)، فعندها نجد أنفسنا في وضع خطير.



الشكل 3–8: الموسط الفعلي والموسط الصوري الموافق يشيران إلى نفس المعطيات

ذلك أن عنصري المعطيات الأعضاء في الغرضين يشيران إلى نفس القيمة.

في هذه الحالة، في نهاية تنفيذ التابع f( ) يُستدعى آلياً هادم الغرض a، وهذا مـــا يجعـــل الغرض object_X في البرنامج المستدعي يحتوي على مؤشر لا يدل على شيء محدد. فعنصـــر المعطيات العضو لديه يشير إلى كتلة من الذاكرة لم تعد ملكه! وينتج عن ذلك عموماً خطـــأ تنفيذي خطير.

من الواضح الآن إذن بأن التصرف الافتراضي للنظام لا يكفي في جميع الحالات. وهـــذا يفسّر ضرورة مفهوم باني النسخ.

في مثالنا الذي ننفذ فيه النمط المجرد String، يقوم باني النسخ بحجز مساحة في الذاكـــرة مخصّصة للنسخة التي ينيها، ثم يقوم بعمل نسخة من سلسلة محارف الغرض المحدِّد في المنطقة الخاصة به في الذاكرة.

```
#include <string.h>
□
#include "string1.h"
□

□
String::String(const char* arg) : ch(new char[strlen(arg)+1]) {
□
 strcpy(ch,arg);              // default constructor
□
}
□
```

```
String::String(const String& arg) : ch(new char[strlen(arg.ch)+1]) {
  strcpy(ch,arg.ch);              // copy constructor
}
String::~String() {               // destructor
  delete [] ch;
}
String& String::operator=(const String& arg) {
  if (this == &arg) return *this;   // case x = x
  delete [] ch;                  // liberate the old content
  ch = new char[strlen(arg.ch)+1];  // allocate new space
  strcpy(ch,arg.ch);
  return *this;
}
```

التابع العضو الأخير في الصف String الذي ستتفحصه في مثالنا هذا هو زيادة تحميل مؤثر الإسناد. تسمح زيادة تحميل مؤثر الإسناد بإسناد غرض من صف إلى غرض آخر هو عموماً من الصف نفسه.

وكما هو الحال في غياب بانيِ النسخ، إن لم يقدم المبرمج تعريفاً لمؤثر الإسناد الخاص بالصف، يقوم النظام بالتوليد الآلي لتابع إسناد، يعمل هذا التابع افتراضياً وفق مبدأ نسخ المعطيات الأعضاء في المعطيات الأعضاء، وذلك بطريقة عودية إذا كانت المعطيات الأعضاء هي نفسها من نمط صف.

لزيادة تحميل مؤثر من بين تلك المؤثرات الواردة في القائمة السابقة، يكفي تعريف تابع يكون اسمه مركباً من الكلمة المحجوزة operator متبوعاً بالمعامل نفسه. إن إعادة تعريف مؤثر لا تقبل قيماً افتراضية للمحدّدات.

يُبيّن توقيع التابع العضو operator =( ) أن هذا التابع لا يقبل سوى محدّد وحيد، وهو هنا مرجع ثابت لـ String.

هذا ما يتعلق بالجزء الأيمن من الإسناد. أما المعامل الأيسر من الإسناد، فهو الغرض الذي تطبق عليه العملية operator=( ). وهكذا فالإسناد

```
x = y
```

يترجم باستدعاء التابع

```
x.operator =(y)
```

عند إعادة تعريف مؤثر، يجب التنبّه دوماً إلى مراعاة فلسفة اللغة C++. فمعامل الإسناد في C و C++ معرّف بحيث يكون المعامل الموجود إلى اليسار قيمةً قابلة للتغيير. يفسّر هذا كـونَ نمطِ العودة لتابع زيادة تحميل التابع operator=( ) مرجعاً على String. بهذه الطريقة، يسـمح المبرمج لمستخدمي الصف أن يستعملوا مؤثر الإسناد مثلما يستعملون مؤثر الإسناد السـابق التعريف.

تكشف كتابة التابع operator=( ) عن بعض النقاط التي يجب التنبه إليها ومعاملتها بحذر. فما دامت قيمة المعامل في الطرف الأيسر قبل الإسناد ستضيع بعد الإسناد، يجب أن يحـرص المبرمج على هدم هذه القيمة فعلياً. إضافة إلى ذلك، يجب قبل البدء بالإسناد نفسه التحقـق من حالة خاصة يمكن أن تصيب مقتلاً إن وجدت. هذه الحالة هي التي يجب أن يتعامل فيـها التابع مع إسناد من الشكل x = x.

يجب اعتراض طريق مثل هذا الإسناد قبل كل شيء. ففي الواقع، مادامت المعالجة النظامية ستقوم لاحقاً بهدم قيمة المعامل الأيسر، فإن المعامل الأصلي سيضيع، وسيؤدي الإسـناد إلى خطأ قاتل.

بقي لنا أن نحل مشكلة أخيرة: ما هي القيمة المعادة لهذا التابع ؟ من الواضح أننا نعني قيمة المعامل الأيسر، أي الغرض الذي طُبق عليه التابع العضو والذي عُدّلت معطياته الأعضاء.

لحسن الحظ أن لغة C++ مجهّزة بما يسهّل هذه الحالة. هناك متحول معرّف ضمناً في جميع التوابع الأعضاء. يسمى هذا المتحول this، وهو مؤشر إلى الغرض الذي استدعي من أجلـه التابع العضو.

في حالتنا، ولأن القيمة الواجب إعادتها هي مرجع لـ String، أي هي المعامل الأيسـر، فإن القيمة المعادة ستكون الغرض المؤشر إليه بـ this وهو this*.

وفيما يلي مثال استعمال الصف String الذي قمنا بتعريفه:

```
#include <iostream.h>
□
#include "string1.h"
```

```
□
int main1 () {
□
  String s1("Nino"); // call default constructor
□
  String s2 = s1;    // call copy constructor
□
  s2 = s1;          // call the overloaded operator =
  return 0;
}
```

عند إنعام النظر في هذه الأسطر يدهش القارئ لدى رؤية السطر

```
String  s2 = s1;   // call of copy constructor
```

للوهلة الأولى، يخيّل إلينا أن النظام يستدعي مؤثر الإسناد بعد زيادة تحميله. غير أن ذلك غير صحيح! فإذا استعملنا رمز الإسناد عند تعريف المتحول، فإن هذا لا يعني إسناداً بل إعطاء قيم ابتدائية، وهو يعامل معاملةً مختلفةً تماماً من قبل المترجم. هناك طريقتان لإعطاء قيمة ابتدائية لمتحول:

```
String   st2 (st1) ;
String   st2 = st1 ;
```

وهما طريقتان متكافئتان إذا كان st1 من النمط String. يقوم المترجم عند أي منهما بإعطاء قيمة ابتدائية لـ st2 باستعمال باني النسخ مع المعامل st1. أما لو كنا نقوم بإعطاء قيمة ابتدائية لمتحول، وكانت هذه القيمة من نمط مختلف عن نمط المتحول المصرّح عنه، فتصبح هاتان الطريقتان في إعطاء القيمة الابتدائية مختلفتين:

```
String   st2 ( "test" ) ;
String   st2 = "test" ;
```

يعني السطر الأول أنه يجب إعطاء قيمة ابتدائية لـ st2 باستعمال بانٍ يقبل "test" محدّداً. أما السطر الثاني، فيعني أنه يجب بناء غرض مؤقت من النمط String باستعمال بانٍ يقبل "test" محدّد، ثم نَسْخُ هذا الغرض المؤقت في st2 باستعمال باني النسخ.

إن كل صف X يتضمن عنصر معطيات عضو من نمط مؤشر يجب أن يحتوي على الأقل على:

- بانٍ افتراضي من الشكل *X :: X() ;*

- باني نسخ من الشكل ; *X :: X(const X&)*
- زيادة تحميل لمؤثر الإسناد من الشكل: ; *X& X :: operator =(const X&)*
- هادم ; *~X ()*

إن هذا الشكل الأصغري لصف يسمى *الشكل القانوني للصف*، إذ إن صفاً من هذا الشكل يسهل فهمه لكونه مدعوماً من لغة C++ نفسها، ومن جهة أخرى، يوفر العدد الأدنى من القواعد التي تسمح للمترجم بتوليد الترميز الصحيح.

إذا لم نكن نرغب في تزويد صف ما بهذه التوابع الأعضاء، ونرغب في الحيلولة دون توليد النظام لهذه التوابع افتراضياً، يكفي أن نصرّح عنها في الجزء الخاص من الصف ونبقي محتواها فارغاً.

أصبح لدينا الآن شكلٌ أصغريٌ من الصف String صالحٌ للعمل، وسنعمل على توسيعه تدريجياً. وهكذا، ففي النسخة التالية، سنضيف زيادة تحميل المؤثر "[]".

```
class String {
public :

  String(const char* = "");        // default constructor

  String(const String&);           // copy constructor
  ~String();                       // destructor
  String& operator=(const String&); // assignment operator  overloading
  char& operator[](int);           // [ ] overloading
  int length(void) const;          // return the length
private :
  char* ch;
};
```

string2.h

عند التعامل مع سلسلة محارف، من المفيد أن نتمكن من النفاذ إلى المحارف المختلفة المشكّلة لهذه السلسلة. توجد طريقة ترميز بسيطة للتعبير عن ذلك، وهي إلحاق اسم السلسلة بموضع المحرف المراد النفاذ إليه محاطاً بقوسين. إن العدد الصحيح الدال على الموضع يجب أن يكون بين 0 وطول السلسلة.

ولأنه يمكن زيادة تحميل المؤثر "[ ]"، يمكننا إذن القيام بتحقيقه للصف String. ولكـــن السؤال الذي يطرح نفسه هنا هو: ما هو النمط الذي يعيده التابع ( ) operator[ ] ؟

نتوقع للوهلة الأولى أن يكون النمط هو char، لأن سلسلة المحارف تتكون فعليـــاً مـــن محارف. يصلح هذا التوقع ما دمنا لا نريد تغيير المحرف المعاد في التعليمة ذاتها كما في المثـال التالي:

ليكن s من نمط String عندئذ:

```
char   c = s[2] ;   // OK
s[2] = 's' ;    // ERROR
```

لكي يصبح بالإمكان استعمال تعليمات كهذه التعليمة الأخيرة في المثال، من الضـــروري أن يعيد التابع operator[ ] مرجعاً على محرف بحيث يمكن تغيير قيمة هذا المحرف.

إن تحقيق هذا التابع وكذلك التابع ( ) length الذي يعيد عدد المحارف المشكلة لسلســـلة المحارف، لا يطرح عوائق جديدة.

```
#include <string.h>
#include <assert.h>
#include "string2.h"

String::String(const char* arg) : ch(new char[strlen(arg)+1]) {
  strcpy(ch,arg);                  // default constructor
}
String::String(const String& arg) : ch(new char[strlen(arg.ch)+1]) {
  strcpy(ch,arg.ch);               // copy constructor
}
String::~String() {                // destructor
  delete [] ch;
}
String& String::operator=(const String& arg) {
  if (this == &arg) return *this;  // case x = x
  delete [] ch;                    // liberate the old content
  ch = new char[strlen(arg.ch)+1];  // allocate the new memory
  strcpy(ch,arg.ch);
  return *this;
}
```

```
char& String::operator[](int i) {
  assert(i >= 0 && i < strlen(ch));
  return ch[i];
}
int String:: length (void) const {
  return strlen(ch);
}
```

يتبين لنا أن معالجة الأخطاء في التابع ( ) operator[ ] هي معالجة مختصرة: يجري التحقق من كون الدليل يقع ضمن حدود السلسلة، وفي الحالة التي لا يكون فيها ذلك محققاً يتوقف تنفيذ البرنامج مع رسالة خطأ تتعلق بعملية التحقق.

إن التابع العضو ( ) length لا يقبل موسطات، وذلك لأنه تابع عضو يُطبّق حُكماً على غرض من هذا الصف.

يتضمن المثال التالي تطبيقاً لهذه النسخة الجديدة من الصف String. يبين المثال أنه من الممكن تعريف توابع غير أعضاء باستعمال محدّدات من نمط الصف String.

```
#include <iostream.h>
#include "string2.h"


void display(String s) {
  // displays s character by character
  for(int i=0; i <s.length(); i++) cout << s[i];
  cout << endl;
}

int main () {
  String s1("Nino"); // call default constructor
  String s2 = s1;   // call copy constructor
  s2 = s1;          // call the overloaded operator =

  s1 = "Rocco";     // "Rocco" is converted to String !
  cout << "Length of s1 : " << s1.length() << endl;
  display(s1);

  cout << "Length of s2 : " << s2.length() << endl;
  s2[0] = 'n';      // possible because [ ] returns a reference of char
  display(s2);
```

```
cout << s2[5];    // execution error (assert)

}
```

كما يوضّح المثال ميزة جديدة للبواني التي تقبل محدّداً واحداً فقط (وذلك إما لأن التابع في الأصل ليس له إلا محدّد وحيد، أو لأن جميع المحدّدات الأخرى لها قيم افتراضية). لمّا كان الصف يعرف بانياً ذا محدّد من النمط سلسلة محارف، فإن النظام قادر على إجراء تحويل للأنماط في حالة التعليمة "s1="ROCCO. وإذْ قد جرت زيادة تحميل المؤثر " = " في الصف String، إذْ أن المعامل الأيسر من نمط String، فإن هذه التعليمة تُفهم كالتالي: s1.operator=("ROCCO"). يتوقع التابع operator=( ) محدّداً من نمط String. ولمّا كان هنالك بانٍ يبني غرضاً من النمط String انطلاقاً من سلسلة من المحارف، فإنه ينفّذ هذا التحويل في النمط.

غير أنه يوجد حد لتحويل الأنماط، إذ لا يمكن القيام آلياً بأكثر من تحويل للنمط خلال عملية واحدة. في هذا المجال، يُقدّم المعيار ANSI/ISO إمكاناً جديداً من خلال الكلمة المحجوزة explicit. هذه الكلمة هي واصف لتابع (مثل inline) ولا يمكن استعمالها إلا مع بانٍ لصف.

إذا استعملناها، فإن النظام لا يقوم بتحويل آلي للنمط كما رأينا سابقاً. فالتحويل لا يجري (في حالة الباني المسبوق بـ explicit) إلا بناءً على طلب صريح من المبرمج الذي يقوم بذلك باستعمال الباني مع قسر النمط (type cast) وفيما يلي ناتج تنفيذ البرنامج السابق.

```
Length of s1  : 5
Rocco
Length of s2  : 4
Nino
Assertion failed: I>0  && I<strlen(ch), file STRING2.CPP, line 27
Abnormal program termination
```

يبين هذا البرنامج بوضوح أن تابع الإظهار display ( ) لا يتكامل البتة مع نمط البرمجة بلغة C++. هناك طريقة أفضل تعتمد على تكييف مؤثر الإدخال/الإخراج من القنوات وإليها عند تعريف المبرمج لصفوف جديدة.

```
#include <iostream.h>
□
```

```
class String {

  friend ostream& operator<<(ostream&, const String&);

  friend istream& operator>>(istream&, String&);

public :

  String(const char* = "");        // default constructor

  String(const String&);           // copy constructor
  ~String();                       // destructor
  String& operator=(const String&); // assignment overloading
  char& operator[](int);           // [] overloading
  int length(void) const;          // return the length
private :
  char* ch;
};
```

string3.h

إن الجديد في هذه النسخة هو التابعان الصديقان (friend) المصرّح عنهما في الصف. لقد قمنا في مثال سابق بتعريف الصف List والصف List_elt. في هذا الصف الأخير، كان الصف List مصرّحاً عنه كصف صديق، وهذا يعني أن التوابع الأعضاء في List يمكن أن تنفذ إلى الأعضاء الخاصة في List_elt.

يمكن استعمال مفهوم الصديق في مجال أضيق، وذلك بالسماح لبعض التوابع فقط بالنفاذ إلى الأعضاء الخاصة للصف. وهكذا، يكون التابع تابعاً صديقاً friend إذا كان تابعاً غير عضو في الصف ولكن له نفس حقوق النفاذ التي يملكها تابع عضو. ولأن التابع الصديق ليس عضواً في الصف، فلا يهم في أي جزء جرى التصريح عنه. نذكّر هنا بأنه يُفضّل كتابة جميع التصريحات friend في بداية الصف قبل الفقرة العامة وذلك لسهولة القراءة.

لنعد إلى مسألة زيادة تحميل معامل الإدخال. إذا أردنا زيادة تحميل مؤثر الإدخال "<<" فيجب تعريف تابع تكون بنيته كالتالي:

```
ostream&  operator<< (ostream& os, const class_type&) {
```

```
    os<< ...  // insert  data member of class_type
    return  os ;
}
```

لا يمكن أن يكون هذا التابع عضواً في صف، إذ إنه في تلك الحالة سيكون المعامل الأيسر لـ "<<" غرضاً من الصف. وهذا لا يتفق مع آلية عمل المؤثر "<<" كما رأيناها في الأنماط السابقة التعريف. ولأن التابع ليس تابعاً عضواً، فليس له حق النفاذ إلا إلى الجزء العام من class_type. سبق تعريف ostream في المكتبة iostream.h. ويمثل class_type صفاً يعرّفه المبرمج. إن هدف التابع operator<<( ) هو أن يقوم بإدخال المعطيات الأعضاء من نمط class_type في قناة الخرج القياسية، وذلك وفق التنسيق الذي يرغب فيه المبرمج، وبالترتيب الذي يريده. من الواضح، أنه لكي تنجح هذه العملية، يجب أن يكون مؤثر الإدخال معرّفاً للمعطيات الأعضاء في الصف، والتي نريد إدخالها في القناة. لنذكّر أن مؤثر الإدخال معروف سابقاً لجميع الأنماط السابقة التعريف في C++.

يجب ألّا ننسى هنا أن نعيد قناة الإدخال. إن إعادة مرجع على الغرض ostream نفسه الذي أدخلت فيه المعطيات الأعضاء التابعة للصف يسمح بتحقيق التسلسل في عمليات الإدخال.

مثال:

cout << i << j ; تُقيَّم من اليسار إلى اليمين. تعيد cout << i الغرض cout الذي سيكون عندها هو المعامل الأيسر لعملية الإدخال التالية cout << j .

```
#include <string.h>

#include <assert.h>

#include "string3.h"


static const int MAX_STRING = 200;


String::String(const char* arg) : ch(new char[strlen(arg)+1]) {
```

```
□
  strcpy(ch,arg);                  // default constructor
}
String::String(const String& arg) : ch(new char[strlen(arg.ch)+1]) {
  strcpy(ch,arg.ch);               // copy constructor
}
String::~String() {                // destructor
  delete [] ch;
}
String& String::operator=(const String& arg) {
  if (this == &arg) return *this;   // case x = x
  delete [] ch;                    // liberate the old content
  ch = new char[strlen(arg.ch)+1];  // allocate new memory
  strcpy(ch,arg.ch);
  return *this;
}
char& String::operator[](int i) {
  assert(i >= 0 && i < strlen(ch));
  return ch[i];
}
int String::length(void) const {
  return strlen(ch);
}
ostream& operator<<(ostream& os, const String& arg) {
  os << arg.ch;
  return os;
}

istream& operator>>(istream& is, String& arg) {
  char tmp[MAX_STRING];
  is.getline(tmp,MAX_STRING);
  delete [] arg.ch;                // destruct the old content
  arg.ch = new char[strlen(tmp)+1];
  strcpy(arg.ch,tmp);
  return is;
}
```

في مثالنا، لأن عنصر المعطيات العضو في الصف String خاص، فنحن مجبرون على التصريح عن زيادة تحميل مؤثر الإدخال كصديق للصف String.

يطبق المبدأ نفسه لزيادة تحميل مؤثر الاستخراج ">>"، وتعرف بنية التابع كالتالي:

```
istream&  operator >> ( istream& is , class_type& ) {
...
is >> ... . // extract  data
```

```
return is :
}
```

كما في تابع الإدخال، لا يمكن لهذا التابع أن يكون عضواً في الصف. سبق تعريف istream في المكتبة iostream.h. يمثل class_type غرضاً من صف يعرّفه المبرمج. إن هدف التابع ( ) <<operator هو استخراج المعطيات من قناة الدخل القياسية، وتحويل قيمها إلى المعطيات الأعضاء في الصف class_type، وذلك وفق التنسيق الذي يرغب فيه المبرمج، وبالترتيب الذي يريده. من الواضح أنه لكي تنجح هذه العملية، يجب أن يكون مؤثر الاستخراج معرّفاً للمعطيات الأعضاء في الصف، التي يجب استخراجها من القناة. نذكر هنا بأن مؤثر الاستخراج معرّف سابقاً لجميع الأنماط السابقة التعريف في C++.

في تحقيق الصف String، عرّفنا الثابت الذي يحدد الحجم الأعظمي لسلسلة محارف مستخرجة من ملف الدخل القياسي بطريقة ساكنة لحصر مجال رؤيتها في المجتزأ الذي يعرفها.

```
#include <iostream.h>
#include "string3.h"

int main () {
  String s1("Nino"); // call default constructor
  String s2 = s1;    // call copy constructor

  cout << "The length of s1 : " << s1 << " is " << s1.length() << endl;
  cout << "The length of s2 : " << s2 << " is " << s2.length() << endl;
  s1 = "Annekind";
  cout << "The length of s1 : " << s1 << " is " << s1.length() << endl;
  cout << "Enter the text" << endl;
  cin >> s2;
  cout << "The length of s2 : " << s2 << " is " << s2.length() << endl;
}
```

يبين البرنامج السابق الإمكانات الجديدة:

```
The length of s1 :  Nino is 4
The length of s2 :  Nino is 4
The length of s1 :  Annekind  is 8
Enter the text
blabla
The length of s2 :  blabla is 6
```

يبين هذا المثال أن تحقيقنا للنمط المجرد String يقترب أكثر فأكثر من الأنماط السابقة

التعريف في C++. تبقى مع ذلك بعض المؤثرات الهامة التي يجب زيادة تحميلها. فأحد أهـــم المؤثرات في التعامل مع السلاسل هو مؤثر الضم concatenation. والمؤثر الأكثر ملاءمة لترميز هذه العملية هو الرمز "+".

إن تحقيق هذه الزيادة في التحميل يكشف عن مشكلة. ففي الواقع، إذا قررنا جعل التابع operator+() تابعاً عضواً، فإن هذا يقتضي أن يكون الغرض الذي سيطبق عليه التابع هو من نمط String. أي يجب أن يكون المعامل إلى يسار مؤثر الضم من نمط String. *مبدئياً*، لاتوجد مشكلة سوى أن المعامل الأيمن ليس بالضرورة من نمط String.

مثال: ليكن s1 و s2 من نمط String، ومؤثر الضم منفذ كتابع عضو، عندها:

```
s1 + s2 ;       // OK !
s1 + "test" ;   // OK !
"test" + s1 ;   // ERROR !
```

لا تعتبر الحالة الثانية مشكلة بسبب أن الصف String يحتوي على بانٍ يقوم ببنـــاء String اعتباراً من سلسلة محارف عادية في C++.

أما الحالة الثالثة فهي تطرح مشكلة إذ يترجم تطبيق المؤثـر "+" بـــ "test".operator+(s1)، وهذا ليس له معنى لأن باني التحويل لا يمكن أن يُستدعى في مثل هذه الحال.

في النتيجة، إذا قمنا بتحقيق زيادة تحميل المؤثر "+" كتابع عضـــو في الصـــف String، فسيكون تصرف المؤثر غير متوقع للمستخدم النهائي. وهذا ما يجب تجنبه مهما كلّف الأمر. تُحلّ المسألة بجعل التابع operator+( ) تابعاً غير عضو في الصف. ومن ثَمّ فهو يقبل محدّديـن، وهما المعامل الأيسر والمعامل الأيمن. ولما كان هذا التابع، لأغراض الفاعلية، يجب أن ينفذ إلى الجزء الخاص في الصف String، فإنه يُصرّح عنه كتابع صديق للصف String.

```
#include <iostream.h>
□

□
class String {
□
  friend ostream& operator<<(ostream&, const String&);
□
```

```
 friend istream& operator>>(istream&, String&);
 friend String operator+(const String&, const String&);
public :
 String(const char* = "");        // default constructor
 String(const String&);           // copy constructor
 ~String();                       // destructor
 String& operator=(const String&); // assignment overloading
 char& operator[](int);           // [ ] overloading
 int length(void) const;          // return the length
 String& operator+=(const String&);
private :
 char* ch;
};
```

string4.h

والآن، يكفي أن يكون أحد معاملي المؤثر "+" من نمط String لكي يجري تحويل المعامل الآخر إلى النمط String باستدعاء الباني المناسب، إن وجد.

ليس في تحقيق تابع الضم أية مشكلة.

```
#include <string.h>

#include <assert.h>

#include "string4.h"


....cf. implementation of the class String (version 3)

String operator+(const String& g, const String& d) {

 String result;

 delete [] result.ch;

 result.ch = new char[strlen(g.ch)+strlen(d.ch)+1];

 strcpy(result.ch,g.ch);

 strcat(result.ch,d.ch);

 return result;
```

```
}
String& String::operator+=(const String& d) {
  char* result;
  result = new char[strlen(ch)+strlen(d.ch)+1];
  strcpy(result,ch);
  delete [] ch;
  strcat(result,d.ch);
  ch = result;
  return *this;
}
```

عندما نقرر زيادة تحميل مؤثر في C++ يجب ألّا ننسى أن اللغة تقدم مجموعة من المؤثرات التي تختصر الكتابة وتسمح عموماً بتوليد ترميز أكثر فاعلية.

وهكذا، إذا قمنا بزيادة تحميل المؤثر "+"، فمن المنطقي أن نقوم بزيادة تحميل المؤثر "+=". ولكن في هذه الحال، سيكون التابع operator+=() تابعاً عضواً لأن المعامل الأيسر سيكون دوماً غرضاً من الصف String. وكما في حالة زيادة تحميل معامل الإسناد، فإن هذا التابع يعيد مرجعاً إلى String.

نكمل النمط المجرد String بزيادة تحميل المؤثرات العلاقاتية وهي (< , <= , > , >=) ومؤثرات المساواة (==, !=). إضافةً إلى ذلك، يصبح النمط String مزوّداً بمؤثر يسمح له باستخراج سلسلة جزئية من المحارف من سلسلة محارف معطاة. سيكون المؤثر المراد زيادة تحميله لهذه الغاية هو مؤثر استدعاء تابع "()". ولتبيان أن التابع المؤثر يعامل كتابع عادي، سنقوم بزيادة تحميل المؤثر ">>" مرة أخرى في الصف String، ولكن كتابع عضو هذه المرة.

إن هذه الزيادة في تحميل المؤثر ">>" تحقق انزياحاً نحو اليسار للمحارف المكونة للـ String الذي يطبق عليه. يقبل المؤثر محدّداً صحيحاً يمثل عدد المواضع المراد إزاحتها. ولجعل استعمال النمط المجرد String أقرب للحدس، سنضيف إليه بانياً اعتباراً من عنصر معطيات من نمط char.

```
#include <iostream.h>
□

□
```

```
enum bool {false=0, true};
□

□
class String {
□
  friend ostream& operator<<(ostream&, const String&);
□
  friend istream& operator>>(istream&, String&);
  friend String operator+(const String&, const String&);
  friend bool operator>(const String&, const String&);
  friend bool operator>=(const String&, const String&);
  friend bool operator<(const String&, const String&);
  friend bool operator<=(const String&, const String&);
  friend bool operator==(const String&, const String&);
  friend bool operator!=(const String&, const String&);
public :
  String(const char* = "");       // default constructor
  String(const String&);          // copy constructor
  String(const char);             // constructor from a char
  ~String();                      // destructor
  String& operator=(const String&); // assignment overloading
  char& operator[](int);          // [] overloading
  int length(void) const;         // return the length
  String& operator+=(const String&);
  String operator()(int,int);     // extract a sub-chaine
  String operator<<(int);         // left shift
private :
  char* ch;
};
```

string5.h

إن المؤثرات العلاقاتية ومؤثرات المساواة يمكن أن تحقق باستعمال التابع ( ) strcmp المعرف في المكتبة String.h. من الواضح أنه يمكن تعريف هذه التوابع كتوابع سطرية، ولكن يجب عدم المبالغة في الاهتمام بمواضيع الاستمثال أثناء بناء النمط المجرد.

إن المشكلة الهامة في الاستمثال (مساحة الذاكرة/ زمن التنفيذ) هي أنها أصعب من أن تعالج أثناء مرحلة تطوير الصف. ويستحسن تأجيلها حتى الانتهاء من تنفيذ النمط المجرد، وإجراء العديد من الاختبارات التي تثبت موثوقية النمط المبني. عندها، وباستعمال أدوات مناسبة (مثل profiler)، يمكن لمطور الصف أن يهتم بموضوع مدى فاعلية التحقيق الذي أنجزه

مُزوّداً بكل إمكانات اللغة. تقدم C++، عدا إمكان كتابة ترميز مركّز، وسيلة لتعريف التوابع السطرية كما رأينا في الفصل السابق، وكذلك إمكان التصريح عن متحولات ومحـــدّدات register كما في C.

إن زيادة تحميل مؤثر استدعاء التابع "()" تمكّن المبرمج من تحديد نوع المحدّدات وعددها بنفسه. في حالتنا هذه، و لأن هذا المؤثر سيفيدنا في استخراج سلسلة محارف جزئيـــة مـــن السلسلة التي يطبق عليها المؤثر، فإن String operator ( ) (int , int) يقبل محدّدين من النمـــط الصحيح. يدل الأول على موضع المحرف الأول المراد استخراجه، ويدل الثاني علـــى عـــدد المحارف المراد استخراجها.

```
#include <string.h>

#include <assert.h>

#include "string5.h"


static const int MAX_STRING = 200;


String::String(const char* arg) : ch(new char[strlen(arg)+1]) {
  strcpy(ch,arg);                  // default constructor
}

String::String(const String& arg) : ch(new char[strlen(arg.ch)+1]) {
  strcpy(ch,arg.ch);               // copy constructor

}

String::String(const char c) : ch(new char[2]) {
  ch[0] = c;                       // constructor from a char
  ch[1] = 0;
}
```

```
String::~String() {                    // destructor
  delete [] ch;
}

String& String::operator=(const String& arg) {
  if (this == &arg) return *this;    // case x = x
  delete [] ch;                      // liberate the old content
  ch = new char[strlen(arg.ch)+1];   // allocate new memory space
  strcpy(ch,arg.ch);
  return *this;
}

char& String::operator[](int i) {
  assert(i >= 0 && i <= strlen(ch));
  return ch[i];
}

int String::length(void) const {
  return strlen(ch);
}

ostream& operator<<(ostream& os, const String& arg) {
  os << arg.ch;
  return os;
}

istream& operator>>(istream& is, String& arg) {
```

```
 char tmp[MAX_STRING];

 is >> tmp;

 // is.getline(tmp,MAX_STRING);
 delete [] arg.ch;              // destruct the old content
 arg.ch = new char[strlen(tmp)+1];
 strcpy(arg.ch,tmp);
 return is;
}

String operator+(const String& g, const String& d) {

 String result;

 delete [] result.ch;
 result.ch = new char[strlen(g.ch)+strlen(d.ch)+1];
 strcpy(result.ch,g.ch);
 strcat(result.ch,d.ch);
 return result;
}

String& String::operator+=(const String& d) {

 char* result;

 result = new char[strlen(ch)+strlen(d.ch)+1];
 strcpy(result,ch);
 delete [] ch;
 strcat(result,d.ch);
 ch = result;
 return *this;
}

String String::operator()(int start,int size) {
 String sub_chain;
 delete [] sub_chain.ch;
 int length = strlen(ch);
 assert(start >= 0 && start < length &&
                          size > 0 && start+size <= length);
 sub_chain.ch = new char[size+1];
 for(int i = 0; i < size; i++)
   sub_chain[i] = ch[start+i];
```

```
 sub_chain[size] = 0;
 return sub_chain;
}

String String::operator<<(int i) {
 int length = strlen(ch);
 assert(i > 0 && i <= length);
 char* new_chain = new char[length-i+1];
 strcpy(new_chain,ch+i);
 delete [] ch;
 ch = new_chain;
 return *this;
}

bool operator>(const String& s1, const String& s2) {
 return ((strcmp(s1.ch,s2.ch) > 0) ? true : false);
}

bool operator>=(const String& s1, const String& s2) {
 return ((strcmp(s1.ch,s2.ch) >= 0) ? true : false);
}

bool operator<(const String& s1, const String& s2) {
 return ((strcmp(s1.ch,s2.ch) < 0) ? true : false);
}

bool operator<=(const String& s1, const String& s2) {
 return ((strcmp(s1.ch,s2.ch) <= 0) ? true : false);
}

bool operator==(const String& s1, const String& s2) {
 return ((strcmp(s1.ch,s2.ch) == 0) ? true : false);
```

```
}

□
bool operator!=(const String& s1, const String& s2) {
 return ((strcmp(s1.ch,s2.ch) != 0) ? true : false);
}
```

لتحقيق الباني انطلاقاً من محرف، يكفي أن نحجز ذاكرة لمحرفين. الذاكرة الأولى للمحرف المحدَّد، والثانية للمحرف '\0' الذي يدل على نهاية سلسلة المحارف. ولما كان الصف يملك بانياً افتراضياً، فهو لا يسمح بإعطاء قيم افتراضية لمحدّد الباني، لأنه عندها سيحصل التباس ناتج عن كون الصف له بانيان افتراضيان، ولا يستطيع المترجم أن يحدد أيهما يستدعي (ولاحتى المبرمج)!

إن التابع العضو الذي يحقق استخراج سلسلة محارف جزئية، يبدأ بحجز مساحة في الذاكرة تكفي للسلسلة الجزئية بعد أن يتحقق من صلاحية المحدّدات. تُستخرج السلسلة الجزئية بعد ذلك وتُعاد. إن التابع يعيد String وليس String& ، لأن السلسلة الجزئية المعادة يجب ألّا تتعرض للتغيير.

يُنفّذ مؤثر الانزياح على وجهٍ مماثل، ولكن في هذه الحال، فإن الغرض الذي يطبق عليه التابع يُعدَّل. يعد التابع القيمة this* ، أي إن التابع operator>>(int) يعيد نسخة من الغرض الذي يعدّله. يدل هذا مرة أخرى على أهمية البواني والهوادم التي تُستدعى أكثر مما يتوقعه المبرمج.

في المثال التالي، تُستثمر الإمكانات المختلفة للصف String. تتكامل المؤثرات التي زيد تحميلها زيادة منسجمة مع أسلوب C++.

```
#include <iostream.h>
#include "string5.h"

int main () {
  String s1("Nino"); // call default constructor
  String s2 = s1;   // call copy constructor

  cout << "The length of s1 : " << s1 << " is " << s1.length() << endl;
  cout << "The length of s2 : " << s2 << " is " << s2.length() << endl;
  s1 = "Lisete";          // affectation
```

```
cout << "The length of s1 : " << s1 << " is " << s1.length() << endl;

cout << (s1+' '+s2) << endl; // concatenations with char

String s3 = s1+=" Nino";
cout << s1 << "\t" << s3 << endl;
cout << s1(2,5) << endl;     // extraction
cout << s1 << endl;
for (int i = s1.length(); i > 0; i--)
        cout << (s1 << 1) << endl; // use of shift operator

// sort the table T (composed of 5 Strings)

String t[5] = { {"ghi"},{"abcde"},{"stu"},{"abc"},{"ghi"} };

for (int i=0; i<5; i++) cout << t[i] << "\t";
cout << endl;
for (int i=0; i<5; i++) {
        int smallest = i;
        for(int j=i+1; j<5; j++)
                if (t[j] < t[smallest]) smallest = j;
        String tmp = t[i]; t[i] = t[smallest]; t[smallest] = tmp;
}
for (int i=0; i<5; i++) cout << t[i] << "\t";

return 0;
}
```

يُنفّذ الجزء الثاني من المثال فرز الشعاع t وهو شعاع عناصره من نمط String، وهذا مـــا يوضّح كيفية استعمال مؤثر المقارنة الذي جرت زيادة تحميله. يعطي تنفيذ البرنامج مايلي:

```
The length of s1 : Nino is 4
The length of s2 : Nino is 4
The length of s1 : Lisete is 6
Lisete Nino
Lisete Nino    Lisete Nino
sete
Lisete Nino
isete Nino
sete Nino
ete Nino
te Nino
e Nino
Nino
Nino
ino
```

```
no
o

ghi   abcde  stu   abc   ghi
abc
```

من خلال هذه الأمثلة المتعلقة بالصف String، قمنا بإظهار عدد من عمليات زيادة تحميل المؤثرات. سندرس في الفصل الأخير المخصص للبرمجة الغرضية التوجه مثالاً مثيراً للاهتمام حول زيادة تحميل المؤثر "<-". في المثال التالي، سندرس بالتفصيل إشكال زيادة تحميل المؤثراتnew . delete. لكن قبل الانتقال إلى هذه النقطة، لا بد من أن نضيف كلمة أخيرة حول زيادة تحميل المؤثرين"++" و "--".

يمتاز هذان المؤثران بخاصة أنهما يمكن أن يستعملا قبل المعامل (الشكل السابقي prefix) أو بعده (الشكل اللاحقي postfix). من المهم إذن أن نميز بين هاتين الحالتين عند زيادة التحميل.

ضمنياً، إذا قام المبرمج بزيادة تحميل أحد هذين المؤثرين بالطريقة التي اتبعناها حتى الآن، فإن هذا يعني أنه يقصد الشكل السابقي. على سبيل المثال، إذا كانX صفاً، عندها:

```
X &          X :: operator ++ ( ) ;
friend       X& operator ++ (X& ) ;
```

هما شكلان محتملان لزيادة تحميل المؤثر "++". الشكل الأول هو تابع عضو، والشكل الثاني هو تابع صديق. هذان الشكلان هما من النمط السابقي. لزيادة تحميل الشكل اللاحقي، يكفي أن نضيف محدّداً من النمط الصحيح إلى الشكلين السابقين.

```
X&     X :: operator ++ (int ) ;
friend  X& operator ++ ( X&, int ) ;
```

لا يفيد هذا المحدّد في شيء سوى مساعدة المترجم على التمييز بين الشكلين.

لنر الآن كيف يمكن زيادة تحميل المؤثرات new . new [ ] . delete . delete [ ]. إذا كان المبرمج يرغب في العمل مع كتل من الذاكرة محجوزة ديناميكاً، فإنه عادة يستعمل العمليات الأولية التي يقدمها نظام البرمجة C++. ويمكن أن يندهش القارئ من الإمكانات التي تقدمها C++ لزيادة تحميل هذه المؤثرات الشديدة الأهمية.

في الواقع، الميزة الأساسية في استعمال المؤثرات new , new [ ] , delete, delete [ ] السابقة التعريف تكمن في أنها تعمل على وجهٍ صحيح أي تقوم بما يطلب منها.

أما المساوئ الأساسية لهذه المؤثرات فهي أن المبرمج لا يعرف غالباً كيف تعمل. أضف إلى ذلك أنه غالباً ما يجري تحقيق مؤثرات إدارة الذاكرة الديناميكية هذه بأسلوب مختلف حسب بنية الحاسوب. يعني هذا وضوحاً أن التصرف العميق لهذه المؤثرات غير قابل للنقل من بنية إلى أخرى، وهذا ما قد يكون مصدراً لمشاكل يصعب السيطرة عليها عند نقل التطبيق من بيئة إلى بيئة أخرى (زمن الاستجابة...). هناك مشكلة أخرى تتعلق بكون هذه المؤثرات غير مخصّصة. لما كان المؤثر new يجب أن يعمل مهما كان نمط المعطيات، فإن تحقيقه يجب أن يكون عاماً ما أمكن. وهذا يعني أنه سيكون في بعض الحالات " غير فعّال " من وجهة نظر الأداء وفاعليته. هناك مشكلة أخرى قد تطرأ على بعض التطبيقات وهي الحصول على فُتات في الذاكرة يمكن أن يؤدي مع تراكمه إلى التوقف غير المتوقع للبرنامج مع أنه خالٍ تماماً من الأخطاء!

لتوضيح الطريقة التي سنعيد فيها تعريف هذه المؤثرات، سنعود إلى البرنامج الذي يدير قائمة من الأعداد الصحيحة. إن القائمة تتألف من عقد. وسنعرّف صفين: List , Node. يُبنى الصف Node بحيث لا يفيد إلا الصف List الذي جرى التصريح عنه كصديق في الصف Node:

```
#include <iostream.h>
□

□
typedef int Type;
□

□
const int MAX_NODES = 200;
□
const int END_LIST = -1;
□

□
```

```
//enum bool {false = 0,true};


class Node {
 friend class List;
 friend ostream& operator<<(ostream&,const List&);

private :
 static int first_eliminate;
 static int first_free;
 void* operator new(size_t);
 void operator delete(void*,size_t);

 Type val;
 Node* next;
};


class List {
 friend ostream& operator<<(ostream&,const List&);
public :
 inline List();
 ~List();
 void insert(Type);
 void eliminate(Type);
 void eliminate_fr(void);
 inline Type first() const;
 inline bool operator!() const;
 bool exist(Type);
private :
 Node* l;
};
```

```
inline List::List() : l(0) {;
};

inline bool List::operator!() const {
  return l == 0 ? false : true;
}

inline Type List::first() const {
  return l->val;
}
```

list2.h

توجد عدة تغييرات في الصف Node مقارنةً بالنسخة الأولى، فإضافةً إلى زيادة تحميل المؤثراتnew و delete، يُوجد عنصرا معطيات أعضاء ساكنة وهما: first_eliminate , *first*_free. كما نلاحظ اختفاء التابعين ( ) display , ( )is_empty والتعويض عنهما بزيادة تحميل لمؤثر الإدخال ومؤثر النفي على الترتيب. أخيراً، نلاحظ اختفاء باني الصف Node.

أما *المعطيات الأعضاء الساكنة* فلها تصرف مختلف تماماً عن المعطيات الأعضاء العادية. في الواقع، تكون المعطيات الأعضاء الساكنة في صف، مشتركة بين جميع الأغراض المأخوذة من هذا الصف. هي تتصرف إذن بطريقة ما كمتحولات شمولية global، و إن كانت هي متحولات محلية في الصف ذات ميزات منهجية محددة.

قد يقرر المبرمج أن يجعلها متحولات عامة أو يبقي اسمها خاصاً في الصف، وفي جميع الحالات لا يؤدي إعطاء معيّن هوية عنصر المعطيات الساكنة نفسه لعنصر خارج الصف إلى حصول تضارب.

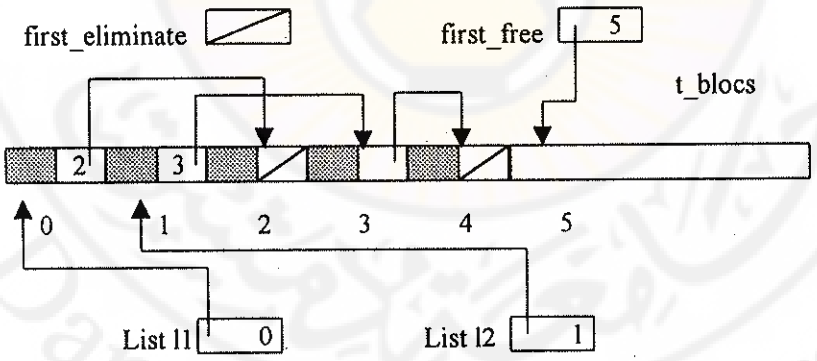
تقع هذه الإمكانات في إطار كبسلة (تغليف) متقدمة لمعطيات الصف الواحد، ويفضل أن تكون أكثر استعمالاً من المعطيات الشمولية التقليدية. وهنا نُطَمئن القارئ الذي تشغله المسائل المتعلقة بالأداء، أن حجم الذاكرة الذي يشغله غرض مشتق انطلاقاً من صف يحتوي على معطيات أعضاء ساكنة، لا يأخذ بعين الاعتبار سوى المعطيات الأعضاء غير الساكنة.

افتراضياً، تُعطى المعطيات الساكنة القيمة الابتدائية). يستطيع المبرمج أن يعطيها قيمة

ابتدائية، ولكن يجب أن يحدث هذا بمستوى شمولي خارج جميع التوابع، كما لـــو كـــانت متحولاً شمولياً عادياً. وأفضل ما يمكن عمله هو أن يجري ذلك في الملف الذي يعرّف التوابـع الأعضاء في الصف. بالطبع، يجب أن يصف المبرمج اسم عناصر المعطيات الساكنة بالاســم الكامل للصف. هذا وإنّ جَعْلَ المعطيات الساكنةِ خاصةً، لا يمنع من النفاذ إليها أثناء إعطـاء القيم الابتدائية. يمكن أن تُستعمل المعطيات الساكنة كقيم افتراضية في قائمة المحدّدات لأحـد توابع الصف.

نلاحظ أيضاً أنه من الممكن تعريف توابع أعضاء ساكنة. لا تستطيع مثل هذه التوابـــع النفاذ إلا إلى المعطيات الأعضاء الساكنة، وبالطبع يمكن أن تعدّل فيها. وحتى يكون بالإمكان استعمال هذه التوابع، يجب أن يصف المبرمج اسم التابع باسم الصف الذي يعرّف ضمنـــه التابع. عندها، نقول إن هذا التابع هو تابع صف، عكس تابع الغرض (التوابع الأعضاء).

في هذه النسخة الثانية من الصف List، نقترح أن ندير بأنفسنا عملية حجـــز الذواكـــر الخاصة بالعقد المكونة للقائمة. تحجز جميع العقد التابعة لمختلف القوائم في فضاء مشـترك في الذاكرة يسمىt_blocs، ويكون اسم الفضاء محلياً في الملف الذي ينفّذ التوابع المختلفة للنمـط List.



الشكل 3–9: تحجز عقد الصف List في القائمة t_blocs

يبين هذا الشكل القائمتين l1, l2. تتألف الأولى من عقدتين عنوانهمـــا 0 و2، وتتـــألف

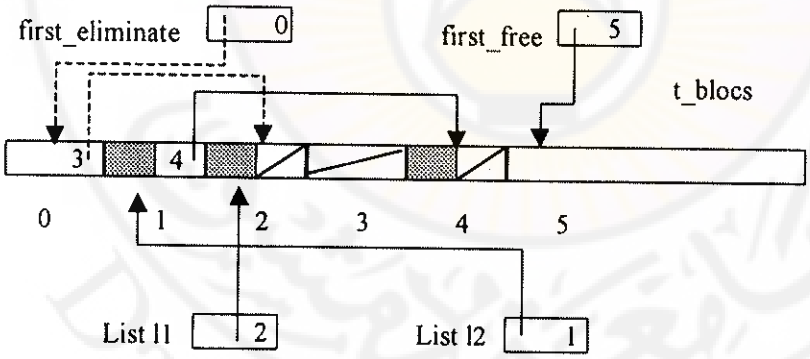
ألثانية من ثلاث عقد في المواضع 1 , 3 , 4.

مادام لم يحصل بعد أي حذف، فإن قيمة مؤشر first_eliminate هي END_LIST. ولأننا سنعمل على إدارة الذاكرة بطريقة داخلية في الصف، فسنعمد إلى استعادة الذاكرة المحرّرة بعد حذف كل عقدة، وذلك بربطها بقائمةِ العنصرُ الأول فيها معطى بـ first_eliminate. عند إجراء عمليات new لاحقة، يكون من الممكن إعادة استعمال العقد الموجودة في هذه القائمة، وذلك بأولوية تفوق أولوية استعمال العقد في القائمة المشار إليها بـ first_free.

نرى في الشكل 10 الوضع بعد أن جرت عمليات حذف متتالية للعقدة 3 ثم 0 .

وبسببٍ من أن الترتيب ليس مهمّاً، فإن قائمة العقد المحذوفة تعمل حسب المبدأ التالي: العقدة المحذوفة الأخيرة هي الأولى في القائمة. ولأن العقدة 0 هي الأخيرة المحذوفة، فستكون الأولى في القائمة المشار إليها بـ first_eliminate. تتألف القائمة l1 من عقدة في الموضع 2. وتتألف القائمة l2 من عقدتين في الموضعين 1 و 4.

يجب أن يكون واضحاً أنه في هذه التشكيلة، يمكن أن تكون كتلةٌ من فضاء الذاكرة t_blocs عقدةً، أو بكل بساطة مؤشراً إلى الكتلة التالية المحذوفة.



الشكل 3–10: تمثيل القائمتين في t_blocs بعدحذف العقد 3 و 0.

يفرض علينا هذا الوضع استعمال مفهوم التسجيلة المتغيرة الأشكال أو مايسمى union. ويجب أن نعلم أنه بعكس ما يحدث في الجداول التي لا تكون عناصرها من نمط union، فإن

البواني المتعلقة بالحقول المختلفة من نمط صف لا تُستدعى آلياً. والسبب بسيط، إذ يُمنَع تعريف union فيه حقل من نمط صف جرى تعريف باني وهادم فيه من قبل المبرمج، أو جرى القيام بزيادة تحميل مؤثر الإسناد فيه من قبل المستثمر. ولهذا السبب فإن صفنا Node لا يمتلك أيّاً من هذه التوابع الثلاثة، وهذا يفسر اختفاء الباني الخاص مقارنةً بالتنفيذ الأول للصف List.

عند تعريف مكان تخزين العقد t_blocs في البرنامج الآتي، اعتبرت الكتلة القابلة للحجز في الذاكرة للصف Node اجتماعاً union، يتألف إما من الحقل n من النمط Node أو من الحقل address الذي يؤشر إلى الكتلة المحذوفة التالية. لن تعطى هذه الكتلة في الذاكرة قيمة ابتدائية.

```
#include <iostream.h>


#include <assert.h>


#include "list2.h"
{}

{
static union Bloc {
  Node n;    // not initialised with a constructor
  int address;
} t_blocs[MAX_NODES];

int Node::first_eliminate = END_LIST;
int Node::first_free = 0;

void* Node::operator new(size_t) {
 if (first_eliminate == END_LIST) {
   assert(first_free < MAX_NODES);
   return &(t_blocs[first_free++]); }
 else {
   int tmp = first_eliminate;
   first_eliminate = t_blocs[first_eliminate].address;
   return &(t_blocs[tmp]);



}}
```

```
void Node::operator delete(void* p,size_t) {
  (static_cast<Bloc*>(p))->address = first_eliminate;
  first_eliminate = static_cast<unsigned int>(static_cast<Bloc*>(p) - t_blocs);
}

List::~List() {
  Node* tmp = l;
  while (tmp) {
    l = tmp->next;
    delete tmp;
    tmp = l;
  }
}

void List::insert(Type x) {
  Node* p = new Node;
  p->val = x;
  p->next = l;
  l = p;
}

bool List::exist(Type x) {
  if (l == 0) return false;
  Node* p = l;
  bool found = false;
  while ((p) && (!found))
    if (p->val == x)
      found = true;
    else
      p = p->next;
  return found;
}

ostream& operator<<(ostream& os,const List& l) {
  os << "( ";
  for (Node* p = l.l; p; p = p->next)
    os << (p->val) << " ";
  os << ")" << endl;
  return os;
}

void List::eliminate_fr(void) {
  if (l == 0)
    return;
  else {
    Node* p = l;
```

```
    l = l->next;
    delete p;
  }
}

void List::eliminate(Type x) {
 if (l == 0) {
  cout << "Empty list !" << endl;
  return;
 }

 // research of the element to eliminate
 Node* p = l;
 Node* pred = 0;
 bool found = false;
 while ((p) && (!found))
  if (p->val == x)
   found = true;
  else {
   pred = p;
   p = p->next;
  }
 if (!found) {
  cout <<  x << " does not exist in the list !" << endl;
  return;
 }
 else {
  if (pred) { // the element has a predecessor
   (pred->next) = (p->next);
   delete p;
  }
  else {     // the element is the first in the list
   l = (p->next);
   delete p;
  }
 }
}
```

في التنفيذ المقترح آنفاً، إذا ما استنفِدت الذاكرة المتاحة للأغراض Node، فإن التحقـــق الذي يقوم به التابع Node:: operator new ( ) سوف يؤدي إلى توقف التنفيذ. يمكن أن نغـير ذلك بحيث يعيد التابع قيمة تشير إلى هذا الوضع. وهكذا يمكن للتابع المستدعي أن يتلقـــى القيمة ويتصرف حسبها.

فيما يتعلق الآن بزيادة تحميل المؤثرات new , delete، نقدم بعض القواعد النحوية التي يجب معرفتها. إن التابع الذي يزيد تحميل new يجب أن يعيد مؤشراً إلى void، على حين يجب أن يُعرَّف delete كإجراء، أي يعيد ببساطة void.

يقبل المؤثر new محدّداً على الأقل من النمط size_t السابق التعريف قياسياً، والذي يعطيه النظام قيمة ابتدائية تعادل حجم الصف آلياً. يجب الإشارة إلى هذا المحدّد وإن لم يستعمل في حالتنا هذه. يعود السبب إلى كون جميع كتل الذاكرة المحجوزة بالمؤثر new الذي أعيد تعريفه، لها الحجم ذاته.

يقبل المؤثر delete محدّداً إجبارياً من نمط مؤشر إلى void، ومحدّداً آخر اختيارياً من نمط size_t.

ضمنياً، تعتبر جميع توابع زيادة تحميل المؤثرين delete و new توابع ساكنة، أي لا يرتبط بها متحول this، ومن ثَمّ لا تستطيع النفاذ إلى الأعضاء غير الساكنة في الصف.

إن هذا الأمر بديهي إذا فكرنا أن new سينفّذ قبل أن يوجد الغرض، و delete ستنفذ عندما لايعود لنا حاجة إليه.

يمكن زيادة تحميل المؤثر new عدة مرات في الصف نفسه، ولكن هذا الأمر ممنوع للمؤثر delete. يجب أن يكون المحدّد الأول في new من النمط size_t وأن يكون المحدّد الأول في delete من نمط *void. وفيما يلي التصريحات الاعتيادية:

```
void*   operator new (size_t) ;
void    operator delete (void *) ;
```

إذا رغبنا في النفاذ إلى المؤثرين new , delete الشموليين والسابقي التعريف، يجب عندها أن يُسبقا بمؤثر إبطال المدى " :: " . يمكن للمبرمج أيضاً زيادة تحميل المؤثرين new[ ] , delete[ ] لتعريف مصفوفات من الأغراض. وبذلك، تُستَعمَل تقنية إدارة الذاكرة الخاصة بالصف في حال الأغراض المستقلة، و في حال مصفوفة من هذه الأغراض.

في مثالنا لم نقم بزيادة تحميل new[ ] , delete[ ]، أي لا نستطيع حجز مصفوفة من العقد. لنعد إلى عمل تابعنا new.

في هذه الحالة الخاصة، إذا كانت قائمة العقد المحذوفة فارغة، فإن التابع Node:: operator new ( ) يعيد عنوان الكتلة الأولى الحرة في المصفوفة t_blocs، وإلا يَجْـــرِ تحديـــث المؤشـــر first_eliminate بإسناد قيمة جديدة له هي قيمة الحقل address المشار إليها بـــــ first_free. إن عنوان كتلة الذاكرة هو العنوان الذي يشير إليه أصلاً first_free.

يقوم التابع ( ) Node:: operator delete بوضع كتلة الذاكرة المشار إليها بالمحدّد في سلسـلة الكتل المحذوفة. ولما كان الإجراء يعلم أن المحدّد * void هو حقيقةً مؤشر إلى Blocs، فنحـــن نقسر المحدّد هنا على أن تجري معاملته كـــ * Blocs (عبر تحويل ساكن للأنماط)، بحيث يمكن النفاذ إلى الحقل address في union. إن الكتلة الأخيرة المحذوفة ستكون هي الأولى في سلسـلة المحذوفات. للحصول على دليلها، يكفي أن نقوم بعملية طرح للمؤشرات بين المحدّد والعنوان الأساسي لمصفوفة الكتل.

فيما يتعلق ببقية تحقيقات التوابع في الصف List، هنالك تغييرات طفيفة فقط. ولكن يجب أن يبقى في ذاكرتنا أنه عند كل تعليمة new Node يجري استدعاء المؤثر الذي جرت زيـــادة تحميله new لا المؤثر new:: . يُكرَّر الشيء نفسه في حالة delete.

لا يوجد أي تغيير في البرنامج الذي يستعمل النمط المجرد List.

```
#include <iostream.h>

#include "list2.h"

int main() {

        List l;

        Type i;


        cout << "Enter integers, ^Z to finish " << flush;

        while (cin >> i) {

          l.insert(i);
```

```
        }

        if (!l) cout << l;
        else cout << "Empty list !" << endl;

        /* modifications */
        char c;
        cin >> ws;
        cin.clear();
        do {
         cout << "<E>liminate, <A>dd, ^Z finish " << flush;
         cin >> c;
         switch (c) {
                case 'e' :
                case 'E' : {
        cout << "Enter integers " << flush; cin >> i;
        l.eliminate(i);
        cout << l;
        break;
                }
                case 'a' :
                case 'A' : {
        cout "Enter integers " << flush; cin >> i;
        l.insert(i);
        cout << l;
        break;
                }
                default : cout << l;
         }
        } while (!cin.eof());
        return 0;
}
```

في هذا المثال، يستطيع القارئ التحقق من عمل هذين المؤثرين اللذين تمت زيادة تحميلها عبر القيام بعمليات إضافة وحذف متتالية.

3– 4 تحويل الأنماط

بقي في تحقيق الأنماط المجرد جانب تناولناه جزئياً فقط، هو موضوع تحويل الأنماط. عند الحديث عن تحويل الأنماط، يجب أن نتناول الأمر من زاويتين:

- التحويل من نمط سابق التعريف، أو من نمط موجود نحو نمط الصف الذي نبنيه.

- التحويل من نمط الصف إلى نمط آخر موجود أو إلى نمط سابق التعريف.

فيما يتعلق بالشكل الأول من تحويل الأنماط، فقد سبق لنا أن تعرضنا له عند حديثنا عن البواني.

وهكذا، فالباني الذي يقبل محدّداً وحيداً، يوصّف تحويلاً لنمط المحدّد نحو نمط الصف. في الصف String، يعطي وجود الباني الذي يقبل محدّداً وحيداً من نمط char أو من نمط مؤشر إلى char إمكان تحويل غرض، من النمط char أو مؤشر إلى char ، إلى String. يُجرى هذا التحويل ضمنياً، إلا إذا قام المبرمج بإرفاق الكلمة المحجوزة explicit للباني. في هذه الحال، يظل التحويل ممكناً شريطة أن يبرمج صراحةً.

تقدم C++ إمكانات أكثر عمقاً في مجال تحويل الأنماط، وذلك بالسماح للمبرمج بتعريف توابع التحويل الخاصة به. وهكذا، يسمح له بتحقيق تابع يحوّل نمط صف إلى نمط آخر سابق التعريف، أو جرى تعريفه قبلاً.

إن تابع التحويل هو تابع عضو من الشكل operator type_identifier ( ) . إن مثل هذا التابع لا يقبل نمط عودة ولا محدّدات. لا يمكن إذن زيادة تحميل هذا التابع، وإن كان هذا لا يمنع أن يكون للصف نفسه عدة توابع تحويل.

يمكن لتابع التحويل أن يستعمل صراحة من قبل المستثمر من خلال القسر (cast) أو ضمناً، من قِبَل المترجم أثناء إجراء التحويلات الرياضية في التعابير. في الواقع، يستعمل المترجم تابع التحويل ضمنياً كلما وجد أمامه غرضاً من نمط صف ليس هو النمط المطلوب، ويوجد لهذا النمط تابع تحويل يمكّنه من تحويل نمط الصف إلى النمط المطلوب. تذهب C++ إلى ما هو أبعد من ذلك عند استعمال تابع تحويل، وذلك في الحالة التي يكون فيها النمط الناتج قابلاً للتحويل إلى النمط المرغوب فيه مروراً بتوابع تحويل الأنماط المقيّسة كما هي معرّفة في CANSI (int إلى double، int إلى char...). ولكن المترجم لا يقوم في جميع الحالات إلا بعملية تحويل واحدة فقط باستعمال تابع تحويل أو باني في حال قيمة معطاة.

ومع أن هذه الإمكانات تبدو مغرية، فإنه يجب توخي الحذر دوماً عند استعمالها خوفاً من

الآثار الجانبية الممكنة الحدوث والتي قد تبدو غريبة. لنعد إلى الصف String ولنضف إليه تابعاً "طبيعياً" للتحويل من النمط String إلى النمط char*.

```
#include <iostream.h>
□

□
enum bool {false=0, true};
□

□
class String {
□
  friend ostream& operator<<(ostream&, const String&);
  friend istream& operator>>(istream&, String&);
  friend String operator+(const String&, const String&);
  friend bool operator>(const String&, const String&);
  friend bool operator>=(const String&, const String&);
  friend bool operator<(const String&, const String&);
  friend bool operator<=(const String&, const String&);
  friend bool operator==(const String&, const String&);
  friend bool operator!=(const String&, const String&);

□
public :
□
  String(const char* = ""):        // default constructor
  String(const String&):           // copy constructor
  ~String():                    // destructor
  String& operator=(const String&); // affectation overloading
  char& operator[](int):           // [] overloading
  int length(void) const;        // return the length
  String& operator+=(const String&);
  String operator()(int,int);      // extract a sub-chain
  String operator<<(int);          // left shift
  operator const char*() const;    // type String --> char*

□
private :
□
  char* ch;
□
};
□
```

```
inline String::operator const char*() const {
  return ch;
}
```

String6.h

يهدف تابع التحويل إلى تحويل غرض من النمط String إلى char* مع الحفاظ على كـون الغرض لا يتغير بهذا التحويل. لهذا السبب قمنا بتعريف تابع التحويل كتابع ثابت أي تابع لا يغير من قيمة حقل المعطيات العضو char، ويعيد سلسلة ثابتة من المحارف، أي إنها لا يمكـن أن تعدّل لاحقاً. إن هذا الوضع يكون مرغوباً فيه إذا كنا لا نرغب، أثناء تغييرنا لسلسـلة المحارف التي نحصل عليها بهذا التحويل، في حصول تغيير غير مقصود على الغرض الذي كان يحتوي هذه السلسلة. بمعنى أدق، إذا كان تابع التحويل معرّفاً على النحو التالي:

```
operator  char * ( ) const ;  // type String - - > char *
```

يمكننا عندها أن نكتب :

```
String  s ("Toto" ) ;
char * p =s;   // conversion function implicitly used
 * P = ' M' ;
cout  << s << endl;  // displays Moto instead of Toto !!
```

هذا هو أحد الآثار السيئة. ولكن ما يزعج أكثر أثناء استعمال توابع التحويل، هو رسالة الخطأ التي يصدرها المترجم والتي تعبّر عن وجود التباس. فيما يلي مثال على برنامج يتسـبب في ظهور مثل هذه الرسالة.

```
#include <iostream.h>

#include <string.h>

#include "string6.h"


int main () {

  String s("toto");

```

```
 const char* p = s;
 cout << s << endl;  // the String
 cout << strcat("He is crazy ",p) << endl; // Ok !
 cout << ("He is crazy "+s);       // error : ambiguous
}
```

إن السطر الذي يتسبب بمشكلة في البرنامج هو ذلك الذي يستعمل مؤثر الضم. ينبّـــه المترجم إلى وجود خطأ سببه حصول التباس بين تابعين هما

String :: operator const char*( ) const و operator + (const String& ,const String&).

ما هو تفسير هذه الرسالة، في حين أنه يبدو أن تابع التحويل يعمل جيداً علـــى ضـــوء التعليمات الواردة قبل التعليمة التي تسبّبت بالخطأ ؟

تكمن المشكلة في كون المترجم في وضع يمكّنه من تطبيق كلا التابعين دون معرفة أيـــهما يجب أن يختار. في الموضع المشار إليه، يمكن تطبيق تابع الضم، أو إجراء تحويل نمط من String إلى *char. في الواقع، لن يتأخر تطبيق تابع التحويل في إعلان الخطأ، ولكن في اللحظة الـــتي يقوم فيها المترجم بهذا التحليل، لا يستطيع أن يحكم سلفاً على نتيجة عمله! هذا الوضع غـــير سليم ويجب تفاديه. وهذا يفسر السبب في أن المترجم ينبّه إلى خطأ في حالة مماثلة ولا يولّـــد ترميزاً للآلة.

في النتيجة، يجب أن يكون المبرمج يقظاً عندما يدرج توابع التحويل لأنها قد تـــؤدي إلى التباسات يصعب كشفها.

3– 5 مدى معيّنات الهوية

يقصد بمدى معيّن هوية ذلك الجزء من نص البرنامج الذي يحتفظ فيه المعيّن بالمعنى الـــذي أعطي له عند التصريح عنه. ويجب عدم الخلط بين مدى الغرض المشار إليه بالمعيّن ووجوده.

حتى الآن، تعاملنا مع هذه المسألة حدسياً، دون إعطاء القواعد الدقيقة التي يســـــتعملها مترجم C++ للتغلب على هذه الصعوبات. نقدم هنا شروحاً مبسّـــطة كافيـــة في معظـــم

الحالات. يميز C++ بين ثلاثة أنماط للمدى:

- مدى الملف
- المدى المحلي (الكتلة)
- مدى الصف

يتألف برنامج C++ من تابع واحد على الأقل. ومن غير المألوف أن يجمع البرنامج كلّه في ملف، لذا نجزئ البرنامج عموماً إلى عدة ملفات منفصلة. يضم كل ملف منفصل التوابع التي يوحد بينها ارتباط منطقي والتي تكوّن مجتزأً. لضمان الاتصال بين هذه الملفات المختلفة، تجمّع التصريحات ذات الاستعمال المشترك في ملفات ترويسة يجري احتواؤها في لحظة الترجمة. تترجم هذه الملفات التي تكوّن البرنامج منفصلةً، ثم تُربط، بعضها ببعض في مجتزأ تنفيذي بأداة تسمى محرر الارتباطات (linker).

إن *مدى الملف* هو ذلك الجزء من نص الملف الذي لا يكون محتوى في تعريف تابع أو صف. افتراضياً، تكون المعيّنات التي لها مدى الملف شمولية في البرنامج، بمعنى أنها تكون قابلة للنفاذ من قبل المجتزآت الأخرى في البرنامج نفسه. تستعمل هذه المجتزآت الكلمة المحجوزة extern قبل اسم المعيّن الذي جرى تعريفه في مجتزأ آخر، وذلك ليفهم المترجم أن هذه المجتزآت تشير إلى معيّن هوية شمولي مُعرَّف في ملف آخر. يحجز فضاء الذاكرة اللازم لمتحول له مدى الملف في لحظة الترجمة ويبقى محجوزاً حتى نهاية تنفيذ البرنامج.

يمكن أن يصرح المبرمج عن معيّن هوية له مدى الملف ويكون ساكناً static، وهذا يعني أنه يصبح غير قابل للنفاذ من قبل المجتزآت الأخرى في البرنامج. نذكّر بأن المتحولات الستاتيكية (الساكنة) يجري إعطاؤها القيمة الابتدائية0 من قبل المترجم.

أما *المدى المحلي* أو *مدى الكتلة*، فهو ذلك الجزء من نص البرنامج الذي يكون محدوداً بزوج الأقواس "{}" أي إنه تعليمة مركبة أو كتلة. وهكذا يعرَّف كل تابع كتلةً مختلفة واحدة على الأقل، وهي الكتلة التي تعرّف التابع. تكون جميع الكتل مختلفة، ويمكن أن تحتوي الكتلة كتلاً أخرى متداخلة. تشكّل محدّدات التابع جزءاً من الكتلة التي تعرّف التابع.

افتراضياً، يجري حجز الذاكرة لغرض أو لمتحول معرّف على مستوى الكتلة آلياً، أي تحجز مساحة الذاكرة اللازمة على مكدّس تنفيذ البرنامج في لحظة تنفيذ الكتلة. بعد انتهاء التنفيذ، تحرر المساحة المحجوزة. إن عمر غرض كهذا لا يتجاوز مدة حياة الكتلة التي تعرّفه لا تعطى المتحولات الآلية قيماً ابتدائية من قبل المترجم ومن ثمّ فليس لها قيم ابتدائية افتراضية.

يمكن التصريح عن متحول محلي على أنه static مما يجعل الذاكرة المحجوزة له ثابتة ومستمرة. وقد يكون هذا الأمر مفيداً عند كتابة توابع عودية. يتوقف مدى معين هوية على كامل جزء النص الخاص بكتلة تعرّف معين هوية محلياً مماثلاً. وهذا يعني أنه يسمح بالاستعمال المحلي للمعيّن نفسه المستعمل استعمالاً شمولياً أو في كتلة محيطة بهذه الكتلة. ومع أن المعيّن هو نفسه، فإنه يوجد كيانان مختلفان. في مثل هذه الحالة، نقول إن معيّن الهوية المحلي يخفي معين الهوية الشمولي. ولكن لمّا كان من الممكن أن تحوي الكتلة كتلاً أخرى، فإنه من الممكن إعادة تعريف معيّن الهوية المحلي في كتلة محتواة.

وهكذا، لكي يستطيع المترجم أن يعمل على وجهٍ صحيح، يجب أن يرتبط كل معيّن هوية بمداه. فعندما يجد المترجم معيّن هوية، يحاول أن يجد تعريفه في المدى الذي يوجد فيه. فإن لم يجده يتحقق من الكتلة الحاوية له مباشرةً وهكذا حتى يصل إلى المستوى الشمولي. وكما رأينا في الفصل2، يسمح باستعمال مؤثر إبطال المدى " :: " لتحديد البحث في المستوى الشمولي.

وأخيراً، فإن *مدى الصف* يخصّ معيّنات الهوية المعرّفة داخل الصف. إن معيّن هوية عضو في الصف هو محلي في هذا الصف ولا يمكن استعماله إلا في الحالات التالية: في تابع عضو لهذا الصف، أو بعد المؤثر " . " المطبق على غرض من الصف، أو بعد المؤثر " -> " المطبق على مؤشر إلى غرض من هذا الصف، أو بعد مؤثر إبطال المدى المطبق على اسم هذا الصف.

يجري داخل كل تابع عضو في صف، تحديد مدى معيّن هوية كما في الكتل، ما عدا أن الكتلة الحاوية للكتلة التي تعرف التابع العضو هي تعريف الصف.

إن مدى الكتلة الحاوية للصف شمولي في حال كون الصف معرّفاً على وجهٍ شمولي. أما إذا كان الصف داخل صف آخر، فإن المدى المحيط هو مدى هذا الصف الأخير. وأما إذا كان الصف محلياً في كتلة، فعندها يكون المدى المحيط هو مدى الكتلة التي تعرفه.

ومع أن عرض هذه القواعد قد يبدو جافاً، فإنه لا يحوي استثناءات شاذة. فالمبرمج الذي اعتاد التعاملَ مع المتحولات المحلية والشمولية في لغة C لن يجد صعوبة تذكر على هذا الصعيد في C++.

3- 6 الثوابت غير الثابتة

تحتضن لغة C++ منذ مدة إمكاناً جديداً، هو القدرة على تعريف معطيات أعضاء في صف على أنها متقلّبة mutable. مثل هذه المعطيات ليست static وليست const. هذه الكلمة الجديدة تعطي المبرمج وسيلة لتغيير قيمة هذه المعطيات الأعضاء حتى ولو كان الغرض الذي تعود إليه قد صرّح عنه أنه ثابت!

مثال:

```
class X {
public :
   X (int  k = 1) : i (k) { } ;
   int  f ( )  const ;
private :
   mutable  int  i ;
};
int  X :: f ( ) const  {
    if (I < 0) (const_cast < x * > (this)) -> i = 0 ;
     return   i ;
}
```

مع أن التابع العضو f ( ) معرّف كثابت، فإنه يمكن أن يغير العضو i المصرح عنه mutable. يحصل الأمر نفسه إذا قمنا لاحقاً بتخصيص مجموعة من الأغراض الثابتة اعتباراً من الصف X. كما في التابع الثابت، لا يسمح بإسناد قيمة مباشرة إلى عنصر معطيات عضو، بل يجب حذف الواصف const باستعمال تحويل صريح للنمط (const_cast). يمكن أن يفاجأ

القارئ بذلك لأنه لا يرى الواصف const في المحدّد this للمؤثر const_cast. في الواقع، في تابع عضو ثابت يكون this ضمنياً *مؤشراً ثابتاً* إلى الغرض الذي جرى استدعاء التابع العضو لأجله. في حالة تابع عضو غير ثابت، يكون this مؤشراً عادياً إلى الغرض الذي استدعي التابع العضو لأجله.

قد تبدو هذه القاعدة غريبة للوهلة الأولى، ولكنها تنتج بسبب ضعف تعريف مفهوم الثابت في النسخ السابقة من C++.

من الآن فصاعداً، يكون غرضٌ من صفٍ غيرَ متقلّب إذا كانت كل المعطيات الأعضاء في الصف معرّفة غير متقلبة، أي غير مزوّدة بالواصف mutable. وكل محاولة لتعديل غرض غير متقلب بعد إعطائه قيمة ابتدائية وقبل هدمه ستؤدي إلى آثار غير معرّفة (يمكن أن يشير إليها المترجم).

3– 7 فضاء الأسماء

برغم الإمكانات التي تقدمها لغة C++ والتي تسمح بإنشاء مكتبات من الصفوف القابلة لإعادة الاستعمال، كانت هذه اللغة تعاني إعاقةً عولجت منذ مدة ليست بالبعيدة. في الواقع، ليكون المبرمج أكثر إنتاجاً، فإن اللغة تحثه على استعمال صفوف موجودة في مكتبة المؤسسة، أو يمكن شراؤها من موزّع خارجي...). يمكن أن يطرح هذا الأمر مشكلة على مستوى تلويث فضاء الأسماء: فالاسم نفسه يمكن أن يكون مستعملاً لدى مصمم المكتبة A ولدى مصمم المكتبة B. في مثل هذه الحال، كان من غير الممكن للمبرمج سابقاً أن يستعمل هاتين المكتبتين A , B معاً في برنامجه. لمعالجة هذه المشكلة الهامة عملياً أدخل مصممو لغة C++ مفهوم فضاء الأسماء أو namespace. لنأخذ المثال التالي:

```
namespace A {     // Library of classes provided by the constructor A
        class X { ...} ;
        void   f ( ) {...} ;
}
namespace B {     // library of classes provided by the constructor B
        class X {...} ;
        void  f ( ) {...};
```

}

إن معيّني الهوية X . f مستعملان في المكتبتين المختلفتين. ولكن جرى تعريف هذين المعيّنين في فضائي أسماء مختلفين، وهما فضاء الأسماء A وفضاء الأسماء B.

في البرنامج الذي يستعمل المكتبتين A . B، يمكن للمبرمج أن يشير إلى أنه يعمل ضمنياً مع المكتبة B، وذلك باستعمال الكلمة المحجوزة using:

```
using   namespace   B ;
X   x ;              // creation of object of type   B : : X
f ( ) ;                    // call of function   B : :  f ( )
A:: f ( ) ;                // call of function   A ::  f ( )
A:: X  y;                  // creation of object of type  A : : X
```

كما يظهر في هذا المثال، يمكن للمبرمج في أية لحظة أن يحدد فضاء الأسماء الذي يختـــاره بالمؤثر " :: ".

الفصل 4 : البرمجة الغرضية التوجه في C++

سنعرض في هذا الفصل الجوانب الغرضية التوجه للغة C++. فقد استعرضنا في الفصلين السابقين ما أضافته C++ من تحسينات بسيطة، لكنها هامة بالنسبة إلى لغة C، وكيفية استخدامها لتحقيق أنماط مجردة abstract types بأسلوب "طبيعي" لإتمام وإغناء اللغة الأساسية. ولكن ما هي البرمجة الغرضية التوجه فعلاً؟

البرمجة الغرضية التوجه طريقةُ تنفيذٍ تُنَسَّقُ فيها البرامج بصورة مجموعاتٍ متعاونة من الأغراض التي يُمَثِّل كل منها مُستَنسَخاً (instance) من صف ما، وبحيث تنتظم جميع صفوفها (classes) في هرميةٍ مترابطة بعلاقات وراثية.

في الواقع لا يتفق جميع المختصين على هذا التعريف ويُحدِّدُ بعضهم سبع مراحل لكمال البرمجة الغرضية التوجه:

- يجب تقسيم النظم إلى مجتزآت *(modules)* وفقاً لبنى المعطيات (بنية اجتزائية غرضية التوجه)
- يجب تمثيل الأغراض على أنها تحقيق لأنماط معطيات مجردة (تجريد المعطيات *data abstraction*)
- على نظام التطوير "استعادة" الأغراض غير المستعملة دون تدخل صريح من المبرمج (إدارة آلية للذاكرة *garbage collection*).
- أي نمط غير بسيط هو مُجتزأ، وأيُّ مجتزأ ذي مستوى عالٍ هو نمط (صف).
- يمكن تعريف صفٍ ما بأنه توسيع أو تحديد لصف آخر (الوراثة *inheritance*).
- يجب أن يكون بإمكان الكيانات *entities* في البرامج الإشارة إلى أغراضٍ تنتمي إلى صفوف مختلفة، ويجب أن يكون بالإمكان تحقيق العمليات بطرق مختلفة ضمن صفوف

مختلفة (تعددية الأشكال *polymorphism* والربط الديناميكي *dynamic binding*).

- يجب أن يكون بالإمكان التصريح عن وراثة صف لأكثر من صف ولأكثر من مرة للصف ذاته (الوراثة المتعددة *multiple inheritance* والوراثة المتكررة *repeated inheritance*).

في الواقع، يتمم هذان التّعريفان كل منهما الآخر، إذ يوضح التعريفُ الثاني التعريفَ الأولَ إيضاحاً أكثر تقنية.

رأينا في الفصول السابقة كيفية الوصول إلى الخطوة الرابعة بإهمال الثالثة[3]، وسنفصل هنا الخطوات الثلاث الأخيرة.

4- 1 المفاهيم الأساسية

يجمع المختصون على أن المفاهيم الأساسية للبرمجة الغرضية التوجه هي:

- الغرض *object*
- الصف *class*
- الوراثة *inheritance*
- تعددية الأشكال *polymorphism*
- الربط الديناميكي *dynamic binding*

يعتبر مفهوم الوراثة، من بين المفاهيم السابقة، مفهوماً خاصاً بالنموذج الغرضي. لندرس

---

[3] إن سبب الانتقال من الخطوة الثانية إلى الرابعة هو أن نظام تطوير C++ لا يتضمن نظاماً آلياً لإدارة الذاكرة. وتلك كانت رغبة المبتكر الذي يعلل هذه الثغرة بأن: "مبرمجي ليسب Lisp يعرفون أن إدارة الذاكرة أمرٌ هام بحيث لا يمكن تركه لمستخدم اللغة، ويَعرِفُ مبرمجُ C (وبالتالي C++) أن إدارة الذاكرة أمرٌ هام بحيث لا يمكن تركه للنظام".

بالتفصيل كلاً من هذه المفاهيم قبل أن نرى كيفية تحقيقها في لغة C++.

4- 1- 1 الغرض *Object*

الغرض هو المفهوم الأساسي في البرمجة الغرضية التوجه. فالأغراض هي الكيانات الأساسية عند تنفيذ البرنامج. وكما رأينا في الفصل السابق، يحتل الغرض مكاناً في الذاكرة الرئيسية وتُحدِّدُ المعطياتُ المُضَمَّنَة فيه حالتَه. ولكن الغرض يتضمن إضافة إلى حالته جميع التوابع التي تحدد سلوكه، وهذا ما نسميه عادة *الكبسلة encapsulation*. تسمح الكبسلةُ بجمع المعطيات والتوابع التي تنمذجُ كياناً ما بعضها إلى بعض، وبذلك يمكن رؤيتها ككل. تُقلِّصُ الكبسلةُ إذن تشتت المعطيات والتوابع، وتمكِّنُ المبرمجَ من فهمِ المسألة فهماً أفضل، وهذا مما يؤدي إلى تقليص تعقيدها.

ومع أن جميع المعلومات تُخزَّن ضمن الغرض، فإن ذلك لا يعني أن بالإمكان الوصول إليها جميعها. إذ تقسم معلومات غرض ما إلى قسمين:

- قسم *عام* Public، مرئي ويمكن الوصول إليه من خارج الغرض.
- قسم *خاص* Private، غير مرئي ولا يمكن الوصول إليه من خارج الغرض. يمكن، بالطبع، الوصول إلى معلومات هذا القسم واستخدامها من قبل التوابع الأعضاء *(members)* في الغرض.

إن مبدأ *إخفاء المعلومات information hiding* مفيد جداً في تحقيق مستوى معيّنٍ من التجريد، وذلك بستر كيفية تحقيق الغرض عن المستخدم. يسمح إخفاء المعلومات أيضاً، فيما بعد، بإنشاء رمازٍ سهل التعديل والصيانة. ولما كان المستخدم النهائي للغرض لا يعلم كيفية تحقيق عملية ما، وإنما فقط ما تُحدِثُهُ من أثر في الغرض، فبالإمكان تعديل كيفية تحقيق العملية (عادة لجعلها أكثر فعالية) دون أي انعكاسات سلبية على مستخدمي الغرض.

لا يمكن إجراء اتصالات بغرض ما إلا عبر واجهته العامة، وذلك بإرسال *رسالة message* إلى الغرض (أي اسم عمليةٍ مع المُحَدِّدات arguments المناسبة). يُعَرَّف سلوكُ غرضٍ بمجموع

الرسائل التي يمكن للغرض أن يجيب عنها. يُنَفِّذُ الغرض، عند تلقيه الرسالة، ما يُعرَفُ *بالطريقة method*، أو ببساطة التابع الذي يحقق العملية المربوطة بالرسالة.

سنرى في سياق هذا الفصل أن مفهوم إرسالِ رسالةٍ وتنفيذِ طريقةٍ هو مفهومٌ أغنى من الطلب التقليدي للتابع.

4- 1- 2 الصف Class

يُعَرِّفُ الصفُ مجموعةً ممكنةً من الأغراض. يَنْتُجُ كلُ غرضٍ، عند تنفيذ البرنامج، من استنساخ instantiation صف موجود. أما الصفُ فهو تمثيلٌ ساكنٌ، جامدٌ (في البرنامج) لمجموعة ممكنة من الأغراض المُستَنسَخَة.

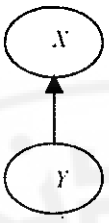
لتوضيح الفرق بين الغرض والصف، يُستَعمَلُ عادةً مجازُ المصنع (الصف) الذي ينتج منتجات (الأغراض). وبإمكان المبرمج تحديد الطريقة التي يجب اتباعها لإنتاج الغرض (البواني في C++).

لا نرى في البرنامج إلا الصفوف، في حين لا يوجد عند التنفيذ إلا الأغراض. يمكن مقارنة هذا السلوك بذلك الموافق للنمط، فتعريف صف يكافئ تعريفَ نمطٍ جديد.

4- 1- 3 الوراثة inheritance

الوراثة هي علاقةٌ بين صفوفٍ مختلفةٍ تَسمحُ بتعريفِ صفٍ جديدٍ اعتماداً على صفوفٍ موجودةٍ سابقاً. يُعتبر مفهوم الوراثة حقاً المفهومَ المبتكر والأقوى في البرمجة الغرضية التوجه. وهو الأداة التي تسمح فعلاً للمبرمج بكتابة رماز قابل للتوسيع ولإعادة الاستخدام (reuse).

عندما يرث الصف Y مباشرةً أو مُداورةً الصف X، نقول إن الصف Y هو صفٌ سليل *X descendant* أو مشتقٌ *derived* منه. نقول أيضاً إن X هو سلفٌ *ancestor* للصف Y. نمثل مفهومَ الوراثة بسهم يخرج من الوارث نحو السلف، مرمزين بذلك علاقة "يرث من".

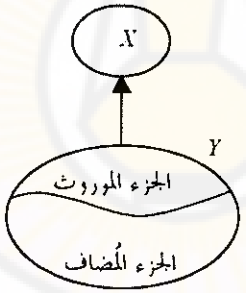


الشكل 4–1: Y يرث X، أو الصف Y سليل الصف X.

عند وجود علاقة الوراثة هذه بين Y و X فإن Y تتكون من جزأين:

- الجزء الموروث أو المُشتَق
- الجزء المُضَاف

فالجزء الموروث هو في الحقيقة تعريف X، في حين أن الجزء المُضاف هو مجموع الحقـــول والطرائق المعرَّفة خِصِّيصى للصف Y.



الشكل 4–2: يتكون الصف المشتق من الجزء الموروث من الصف السلف والجزء الخاص به.

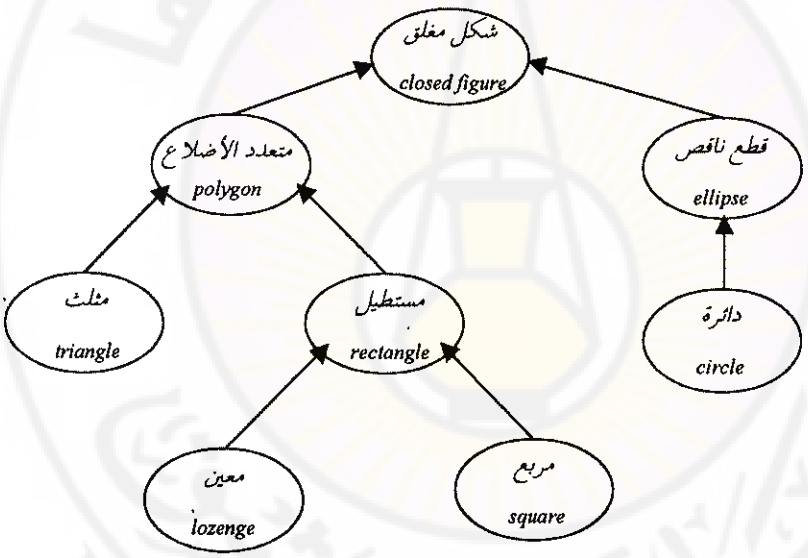
يمكن اعتبار علاقة الوراثة علاقةً من نمط "is" أي "هو" لأن الصف Y، المشتق من الصـف X، يرث كل شيء من X ومن ثَم فهو X. من جهة أخرى، Y أوسع من X. لأننا عندما ننشئ صفاً جديداً بالوراثة، نضيف إليه عادة حقولاً و/أو طرائق جديدة.

في الواقع، يمكن اعتبار الوراثة إجراء توسيعٍ وتخصيصٍ بآن واحد. كيف يمكن ذلـــك؟

يتوقف الأمر على كيفية اعتبار الصف. فإذا نظرنا إلى الصف على أنه نمط، يكون الإجـــراء تخصيصاً. لنوضح ذلك بمثال: عند إنشاء الصف *مستطيل Rectangle* وارثاً للصف *مضلـــع Polygon*، يظهر بوضوح أن الصف *Rectangle* يمثل شيئاً أكثر خصوصيـــة مـــن الصـــف *Polygon*.

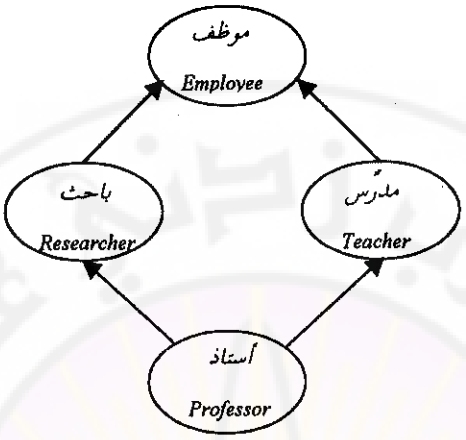
أما إذا نظرنا إلى الصف على أنه مُجتزأ، وكان *Y* وارثاً لـــ *X*، فإن المُجـــتزأ *Y* يتضمـــن طرائقَ وحقولَ *X* إضافةً إلى ملحقاته الخاصة. هنا يكون الإجراء توسيعاً.

فيما يلي مثالٌ تقليديٌ لمخطط وراثة في مجال الرسم البياني:



الشكل 4–3: مثال لهرمية الصفوف. "*شكل مغلق*" هو الصف الجذر لهذه الشجرة.

يمكن لصف واحد أيضاً أن يرث عدة صفوف (الوراثـــة المتعـــددة multiple inheritance). ومفهوم الوراثة المتعددة هام جداً لأنه يسمح طبيعياً بحل بعض المشاكل. تســـمح لغـــة C++ بالوراثة المتعددة، ولكن لم تُحَلَّ بَعد كافة المشاكل المتعلقة باستخدامه، وخاصـــة مشـــكلة *الوراثة المتكررة repeated inheritance*.



الشكل 4–4: مثال للوراثة المتكررة

تحدث الوراثة المتكررة، ضمن الوراثة المتعددة، عندما يكون صفٌ ما سلفاً لصف آخر عدة مرات. لنأخذ على سبيل المثال التوضيحي الصف *باحث Researcher* الذي يرث الصف *موظف Employee*، مثله مثل الصف *مدرس Teacher*. إذاً عرَّفنا الآن الصف *أستاذ Professor* الذي يرث في وقت واحد الوقت الصفين *Researcher* و*Teacher*، نصبح أمام حالة وراثة متكررة، لأن الصف *Professor* يرث مرتين الصف *Employee*.

المشكلة الآن أن نعرف: أيكون غرضٌ ما من الصف *Professor* مؤلفاً من غرض واحد أم من غرضين من الصف *Employee*؟ وكما سنرى لاحقاً، فإن C++ تعطي المبرمجَ إمكانَ اختيارِ التمثيل المرغوب.

إن الطريقة البدهية للإجابة عن التساؤل السابق هو القول بضرورة وجودِ غرضٍ وحيدٍ من الصف *Employee*، فيجب التشارك إذن في الورودين. المشكلة التي تُطرح هنا هي إمكان حدوث تضارب بين عمليات الصفين *Researcher* و*Teacher* التي تغير الحالة الداخلية للورود لمشترك لـ *Employee*.

4 – 1 – 4 تعددية الأشكال Polymorphism

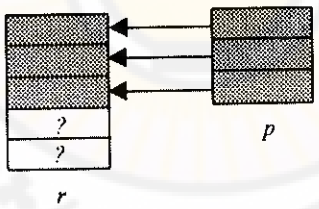
اشتُق المصطلح Polymorphism من اليونانية، ويعني القدرة على أخذ عدة أشكال. نقول عن كيان إنه متعدد الأشكال إذا كان بإمكانه، عند التنفيذ، الدلالة على مستنسخات من صفوف مختلفة.

تنتج تعددية الأشكال عن مفهوم الوراثة. والفكرة هي أن اشتقاق Y من X يعني أن Y "هو" X. وعلى هذا يمكن استخدام Y حيث يمكن استخدام X.

يترتب على ما سبق نتائج هامة، منها توسيع قاعدة توافق الأنماط: أي غرضٍ من الصف X هو متوافقٌ مع أي غرضٍ آخرَ من الصف X، ولكن لدينا أيضاً أن كل غرضٍ من الصف Y المشتق من الصف X، متوافقٌ مع جميع أغراضِ الصفِ X.

لنوضح ذلك بأخذ مؤثر الإسناد، فإذا كان p غرضاً من الصف *Polygon*، و r غرضاً من الصف *Rectangle* المشتق من الصف *Polygon*، فالإسناد التالي p=r صحيح.

يوجد حد لتعددية الأشكال في النظم الغرضية التوجه، وإلا فلن يبقى لمفهوم الغرض معنى كبير. افتراضياً، لا يُسمح بإسناد غرضٍ من الصف X إلى غرضٍ من الصف Y المشتق من X.

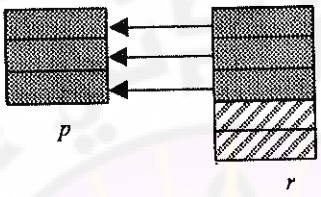


الشكل 4–5: بعد الإسناد r=p، تكون بعضُ مناطق الذاكرة في حالة عدم تعريف.

وبالعودة إلى المثال السابق يكون الإسناد بالاتجاه المعاكس r=p غير صحيح! ويمكن فهم ذلك عند النظر إلى حجم الذاكرة المحجوزة لكل من هذين الغرضين. فعادة يحتاج الغرض r المشتق من الغرض p إلى حجم ذاكرةٍ أكبر من الغرض السلف، لأن إنشاء صف بالوراثة هو توسيعٌ للصف الموجود. لذلك، فإن السماح بإجراء الإسناد r=p، يقتضي أن تكون بعض

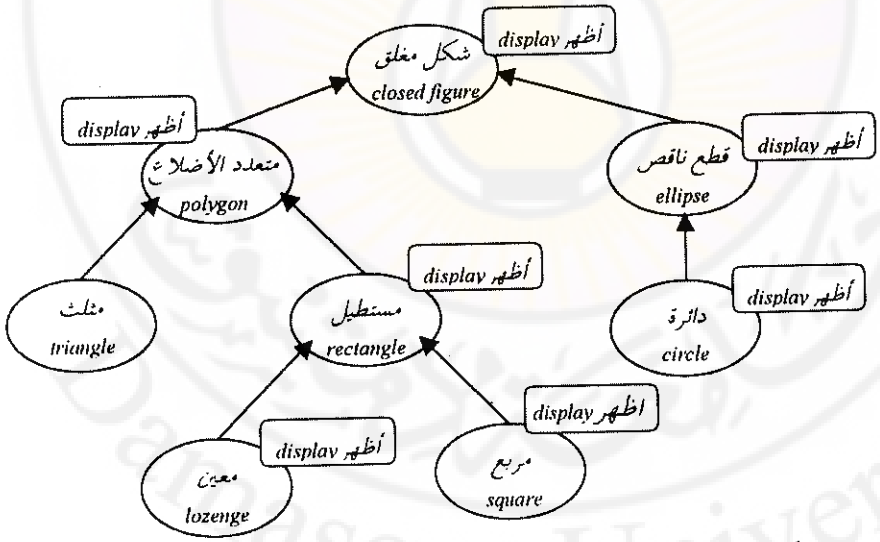
المعطيات والطرائق غيرَ معرفةٍ ضمن الغرض r بعد الإسناد! (الشكل 4–5).

لا يثير الإسنادُ بالاتجاه المعاكس أية مشكلة. فالشيء الوحيد الذي يحدث هـــو عدم نسخ المعلومات الإضافية التي يمتلكها الغرض المشتق، وهذا منطقياً صحيح !



الشكل 4–6: عند إجراء الإسناد p=r، لا تنسخُ مناطقُ الذاكرةِ الخاصةِ بـــ p, r.

ينتج شكل آخر من تعددية الأشكال عند إعادة تعريـــف overriding بعـــض الطرائق في صفٍ سليل تحقيقاً لمتطلباته. يمكن أن تحمل طريقة ما نفس الاسم، على طول شجرة الوراثة، ولكن تختلف نتائجها حسَب نمط الغرض المطبقة عليه.



الشكل 4–7: مثال لهرمية صفوف تمتلك جميع الصفوف فيها الطريقة display().

4- 1- 5 الربط الديناميكي Dynamic binding

يُقَارَنُ الربطُ الديناميكي بمفهوم الربط الساكن Static binding المستخدم عادة في أغلب لغات البرمجة التقليدية. نعني بالربطِ الآليةَ التي تنشئُ الرابطةَ بين رمازِ الطريقة (الإجرائية، التابع) واستدعائها.

يجب تناول آلية الربط الديناميكي مع تعددية الأشكال وآلية الوراثة، إذ يعتمد اختيار طريقةٍ لتطبيقها على غرضٍ ما على النمط الديناميكي لهذا الغرض.

لنأخذ الغرضين p وr من النمطين *Polygon* و*Rectangle* على التتالي، ولنطبق عليهما الطريقة *display()*:

```
p.display();
r.display();
```

لا توجد مشكلة في هذا التطبيق، إذ يجري، عند الترجمة، ربطُ الطريقةِ الصحيحةِ بكل من هذين الغرضين، اعتماداً على النمط الساكن لكل غرض.

إذا قمنا بالمقابل بإسناد الغرض r إلى p، ثم طبقنا الطريقة *display()* على الغرض p، فيقتضي مفهوم الربط الديناميكي عندها أن النمط الديناميكي لـ p هو الذي يفرض اختيار الطريقة التي يجب تطبيقها.

```
p. display();
r. display();
p=r ;
p. display();  //??
```

عند تطبيق الربط الديناميكي، تطبق على الغرض p الطريقة *display()* الخاصة بالصف *Rectangle*. ولا يستطيع المترجم طبعاً أن يعرف ذلك أثناء عملية الترجمة. فالربط الديناميكي آلية لا يمكن تَبَيُّنُ آثارها وتطبيقها إلا عند تنفيذ البرنامج.

ونذكر هنا أن الربط الوهمي virtual binding لا يعمل إلا إذا كان كلٌّ من p وr مؤشراً على الغرض الذي يمثله.

4- 2 مِيْزات البرمجة الغرضية التوجه فيما يتعلق بتصميم البرمجيات

تدعم البرمجة الغرضية التوجه (Object Oriented Programming) التصميمَ الجيد للبرمجيات. وذلك لأن المعايير المألوفة لجودة البرمجيات، وهي الاجتزاء modularity والكبسلة والتجريد وقابلية التوسيع، نتائج منطقية للبرمجة الغرضية التوجه.

ينتج اجتزاء البرمجية من الصفوف التي ينظر إليها على أنها مجتزآت. والكبسلة مدعومة طبيعياً لكون الصفوف تَتَضَمَّنُ المعطيات التي تعرف حالة الأغراض والطرائق التي تصف سلوكها. تسمح معظم لغات البرمجة الغرضية التوجه للمبرمج بتعريف الواجهة المرئية للمستخدم، وإخفاء التنفيذ وإبقائه بعيداً عن متناول المستخدم النهائي للصف (إخفاء المعلومات).

هناك طريقتان للتجريد: التجريد بالتوصيف abstraction by specification، والتجريد باستخدام المُوْسِطات abstraction by parameterization.

يفيد التجريد بالتوصيف في فصلِ توصيف الكيان عن تنفيذه. وتدعم أغلبُ نظمُ البرمجة هذا المفهوم. في الواقع، تؤدي الواجهة العامة public للصف دورَ التوصيف المجرَّد abstract specification لهذا الصف. وهذه الواجهة هي التي تحدد العمليات التي يستطيع المستخدم النهائي تطبيقها على الأغراض المستنسخة من صف ما. أما ما تبقى من تفاصيل التنفيذ فتحجب عن المستخدم.

يُفيد التجريدُ باستخدام الموسطات في فصل نمط المعطيات التي تعالجها بنية معطيات ما عن طريقة معالجتها. ويعني ذلك أن بالإمكان مثلاً توصيف الصف مكدَّس *Stack* بتجرد وباستقلال عن نمط الكيانات التي تعالجها الأغراض من الصف *Stack*.

ثُم إن قابلية التوسيع هي نتيجة طبيعية لآلية الوراثة، إذ يتقلص الجهد اللازم لإنشاء صفوف جديدة انطلاقاً من صفوف موجودة. فكلما ارتفعت هرمية الوراثة ازداد الحجم الذي يرثه الصف الجديد المُنشأ بالوراثة.

وإضافة إلى ذلك، فإن خاصة تعددية الأشكال المرتبطة بمفهوم الوراثة تجعل من غير الضروري، عند إضافة صف وارث إلى مجموعة من الصفوف المرتبطة سابقاً بالوراثة، تعديل الطرائق الموجودة عند الأسلاف للتمكن من تطبيقها على أغراض الصف المنشأ حديثاً.

4- 3 المفاهيم الغرضية التوجه في C++

سندرس في هذا الفصل كيفية تنفيذ المفاهيم الغرضية التوجه المعروضة آنفاً في لغة C++ . في هذه اللغة، لا يبرمج المستخدم "طبيعياً" بالنموذج الغرضي لأنها لغةٌ برمجة هجينة، أي لغة برمجة إجرائية تقليدية أضيفت إليها إمكانات التوجه الغرضي. تتنافس اللغات الهجينة مع نظم البرمجة الغرضية التوجه الصِّرْف، مثل سمول توك Smalltalk أو إيفل Eiffel.

يأتي نجاح لغات البرمجة الغرضية التوجه الهجينة من أن هذه اللغات مُطَمْئِنَةٌ أكثر، لأنها تندرج ضمن اللغات التقليدية، فإذا لم نستطع أو لم نرد تطبيق التقنيات الغرضية التوجه يمكننا دوماً استخدام اللغة بالطريقة التقليدية.

4- 3- 1 الوراثة البسيطة في C++

تعطي C++ المبرمج إمكان تعريف صفوفٍ جديدةٍ بالوراثة انطلاقاً من صفوف موجودة سابقاً.

4- 3- 1- 1 قابلية الوصول accessibility إلى أعضاء الصف

رأينا في الفصل 3 أن بالإمكان تحقيق إخفاء المعلومات بتعريف أعضاء عامة public وخاصة private ضمن الصف. لم يأخذ هذا التصنيف الثنائي لأعضاء الصف بعين الاعتبار الإمكانات الجديدة التي تضيفها الوراثة. في الواقع، يمكن تحديد قابلية الوصول إلى أعضاء الصف بثلاثة طرق:

- العضو خاصّ (private) بالصف، أي لا يمكن الوصول إليه من خارج الصف،

وتستطيع فقط التوابع الأعضاء و"الأصدقاء" *friends* الوصول إليه.

- العضو *عام* (public)، يمكن لأي كان الوصول إليه سواء كان عضواً أم لا؛
- العضو *محميّ* (protected)، هذا المفهوم جديدٌ، ويمكن تفسيره بأن من المفيد إعطاء إمكان الوصول إلى عضوٍ محميٍ للتوابع المنتمية إلى الصف وإلى الصفوف المشتقة، إضافةً إلى "أصدقاء" الصف. أما باقي التوابع فإمكان وصولها إلى العضو المحمي مستحيلة.

وهكذا يمكن للمبرمج، عند تعريف صفٍ في C++، تعريفُ عناصر خاصة *private* (وهي القيمة الافتراضية) ومحمية *protected* وعامة *public*.

يمكن أيضاً للمبرمج، عند تعريف صف جديد بالوراثة انطلاقاً من صفٍ موجود، التأثير في درجة الوصول إلى العناصر الموروثة.

في C++، عند تعريف صف باشتقاقه من صف آخر، يرث الصف الفرع جميع أعضاء الصف السلف سواء كان يمكن الوصول إليها أم لا (أعضاء الصف السلف إما public أو private أو protected). يمكن للمبرمج تعديل إمكان الوصول إلى الأعضاء الموروثة عند تعريف الصف الفرع.

للتعبير عن اشتقاقِ صفٍ من صفٍ آخر، يكفي أن يتبعَ اسمُ الصفِ المُشتقِ الرمزَ ":" يتبعه (اختيارياً) نمطُ الاشتقاق، يُذكر بعد ذلك اسمُ الصف السلف، ثم كالعادة تعريف الصف.

يسمحُ نمطُ الاشتقاق للمبرمج بتحديدِ قابلية الوصول إلى الأعضاء الموروثة. فيما يلي جدولٌ يلخصُ تأثيرَ نمط الاشتقاق في حالة الأعضاء في الصف المشتق تبعاً لحالتها في الصف السلف.

نمط الاشتقاق	حالة العضو في الصف السلف	حالة العنصر في الصف المشتق
private	private protected public	inaccessible private private
protected	private protected public	inaccessible protected protected
public	private protected public	inaccessible protected public

من الواضح أنه إذا جرى اشتقاق صف جديد بصورة public من صف سلف، تحتفظ العناصر الموروثة بحالتها التي كانت تملكها في الصف السلف. فلا يمكن أبداً الوصول إلى عضو اشتُقَ من عضو محمي private في الصف السلف. يجب الانتباه إلى أنه بالرغم من عدم إمكان الوصول إلى العضو المحمي في الصف السلف، فإن هذا العضو هو بالفعل جزء من الصف المشتق !

عند إنشاء صف جديد بالاشتقاق المحمي *protected* لصف سلف، يكون لجميع الأعضاء الموروثة الحالة "محمية" *protected* فيما عدا الأعضاء الخاصة *private* التي لا يمكن الوصول إليها بجميع الأحوال. هذا يعني أنه لا يمكن الوصول إلى الحقول الموروثة إلا من قبل التوابع الأعضاء في الصف المشتق، أو في "أصدقاء" الصف المشتق أو في الصفوف المشتقة من هذا الصف المشتق (في حال وجودها).

أما الاشتقاق الخاص *private* فهو أكثر تقييداً. في هذه الحالة، مهما كانت حالة العناصر الموروثة، فهي ستأخذ الحالة *private* في الصف المشتق، ومن ثَم لا يمكن الوصول إليها إلا من قبل أعضاء الصف المشتق أو "أصدقائه". هذا النمط من الاشتقاق يجعل الوصول المباشر إلى العناصر الموروثة مستحيلاً في الاشتقاقات اللاحقة. هذا الخيار هو الخيار الافتراضي المستخدم في نظام C++. ومن ثَم، عندما ينسى المبرمج وضعَ نمط الاشتقاق يطبَّقُ النمطُ *private*.

4-3-1-2 المعنى العملي للوراثة

عندما يرثُ صفٌ مشتقٌ صفاً سلفاً، يتألفُ الغرضُ من نمطِ الصف المشتق منَ المعطيات

التي تنتمي إلى الصف السلف (مهما كانت حالتها) إضافة إلى المعطيات المُعرَّفة في الصف المشتق.

يرثُ الصفُ المشتق جميعَ التوابع الأعضاء في الصف السلف باستثناء البواني والهوادم والتحميل الزائد overloading لمؤثر الإسناد.

لا يمكن أن يرثَ صفٌ مشتقٌ جزءا من أعضاء الصف السلف فقط. ولكن يمكن للمبرمج إعادة تعريف تابع موروث ومواءمتُه لحاجات الصف المشتق.

يُسمَحُ لتابعٍ عضوٍ في الصف المشتق باستدعاء تابعٍ عضوٍ في الصف السلف باستخدام مؤثر إبطال المدى scope resolution. لا تُورَثُ التوابعُ اللا أعضاء (الصديقة).

عند بناء غرضٍ مشتقٍ، يجري استدعاء باني الصفوف السلف ثم باني المعطيات الأعضاء، ثم باني الصف المشتق. أما ترتيب استدعاء الهوادم فهو عكسُ ترتيبِ استدعاء البواني (أولاً هادم الغرض، ثم هوادم المعطيات الأعضاء، وفي النهاية هوادم الصفوف السلف). إذا لم يُحدِّد الصفُ السلفُ بانياً افتراضياً فيجب على الصف المشتق أن يستدعي صراحةً بانياً من الصف السلف. لذلك فعند تعريف باني الصف المشتق، يجب أن يَتْبَعَ توقيعُ signature هذا الأخير الرمزَ ":" ثم اسم باني الصف السلف مع قائمة الموسطات.

فيما يلي مثالٌ على استخدام مفهوم الوراثة في C++. نُعرِّفُ صفين: الصف شخص *Person* والصف *موظف Employee* الذي سيكون صفاً مشتقاً بصورة عامة *public* من الصف الأول.

```
#include <iostream.h>
#include "string7.h"

class Person {
  friend ostream& operator<<(ostream&, const Person&);
public :
  Person(String="",unsigned short int=0);
  Person(const Person&);          // copy constructor
  String name() const;            // accessor
  unsigned short int age() const;
protected :
```

```
 String namep;
 unsigned short int agep;
};

class Employee : public Person {
 friend ostream& operator<<(ostream&, const Employee&);
public :
 Employee(String="",unsigned short int=0, unsigned int=0);
 Employee(const Employee&);          // copy constructor
  unsigned int salary() const;
private :
 unsigned int sal;
};
```

Pers 1. h

يُعرِّفُ الصفُ *Person* اثنتين من المعطيات المحمية وهما *namep* للاسم و*agep* للعمر، لـــذا يمكن الوصول إليهما مباشرة من جميع الصفوف المشتقة مـــن *Person*، ولكـــن لا يمكـــن للمستخدم النهائي لـــ *Person* الوصول إليهما مباشرة، بل يجب أن يمر عبر النوافذ accessors والمُعدِّلات modifiers التي يقدمها مُنشئُ الصف.

الصف *Person* ليس كاملاً، فهو لا يتضمن إلا بانياً افتراضياً وبـــانيَ نســـخ ونـــافذين accessors، أما الصف *Employee* فيعرِّفُ للعنصر المُعرَّف *sal* نافذاً إضافياً.

```
#include <iostream.h>
#include <iomanip.h>
#include "string7.h"
#include "pers1.h"

Person::Person(String n, unsigned short int a) : namep(n),agep(a) {
}
Person::Person(const Person& p) : namep(p.namep),agep(p.agep) {
}
String Person::name() const {
 return namep;
}
unsigned short int Person::age() const {
 return agep;
}
ostream& operator<<(ostream& os, const Person& p) {
 os.setf(ios::left);      // to write from the left
 os << "Name : " << setw(30) << p.namep;
 os << " Age : " << p.agep;
```

```
 return os;
}

/*-------------------------------------------------------------------*/

Employee::Employee(String n,unsigned short int a, unsigned int s)
                                        : Person(n,a),sal(s) {
}
Employee::Employee(const Employee& e) : Person(e.namep,e.agep),sal(e.sal) {
}
unsigned int Employee::salary() const {
 return sal;
}
ostream& operator<<(ostream& os, const Employee& e) {
 os << static_cast<Person>(e);     // os << (Person)e
 os << " Salary : " << e.sal;
 return os;
}
```

Pers1.cpp

إن تحقيقَ طرائقِ الصف *Person* عمليةٌ بسيطة، أما الصف *Employee*، الذي يرثُ الصفَ *Person*، فيجب الانتباه عند تحقيقه. إذ يبدأ باني غرضٍ من النمط *Employee* باستدعاء باني الصف *Person* بالمحددّات المناسبة. وذلك منطقي لأن *Employee* "هو" *Person*.

ويقوم التابع الصديق بزيادة تحميل overloading مؤثر الإدراج insertion، ليكون بالإمكان كتابة الموظف بكامل معطياته. نقوم هنا بإعادة استخدام الرماز المكتوب من أجل الصف *Employee* بإجبار المترجم على رؤية الغرض كـ *Person* ثم باستدعاء تابع الإدراج المكتوب من أجل *Person*.

فيما يلي برنامجٌ يستخدمُ الصفين المعرفين سابقاً:

```
#include <iostream.h>
#include "pers1.h"

int main() {
 Person p1("Paul",40), p2("Pierre",30);
 Employee e1("Claudine",36), e2("Anne",34,30000);
 cout << p1 << endl << p2 << endl;
 cout << e1 << endl << e2 << endl;
 Person p(p1);
 Employee e(e2);
```

```
 cout << p << endl;
 cout << e << endl;
 p = e2;
// e = p1; // illicit !
 cout << p << endl;
 return 0;
}
```

Tpers1.cpp

وهذا هو الخرج الناتج عن تنفيذه:

```
Name : Paul              Age : 40
Name : Pierre            Age : 30
Name : Claudine          Age : 36 Salary : 0
Name : Anne              Age : 34 Salary : 30000
Name : Paul              Age : 40
Name : Anne              Age : 34 Salary : 30000
Name : Anne              Age : 34
```

تحقق C++ الأغراضَ من النمط صف كتسجيلات records تقليدية. تبدو هذه الطريقة بديهية ومنطقية لأول وهلة، ولكنها تعقد تحقيق البرمجة الغرضية التوجه (تعددية الأشكال والربط الديناميكي). أما في نظم البرمجة الغرضية التوجه الصِّرْف مثل إيفل Eiffel أو اللغات الهجينة مثل Object Pascal أو المِقيَس الجديد لـ Cobol فيعمل المبرمج، دون علمه بالضرورة، باستخدام مراجعَ references على الأغراض فقط.

يمكن للمبرمج، مع ذلك، أن يتعامل في C++ صراحةً مع مؤشرات على الأغراض، كما يبينه التحقيق الجديد للمثال السابق:

```
#include <iostream.h>
#include "string7.h"

class Person {
  friend ostream& operator<<(ostream&, const Person&);
public :
  Person(String="",unsigned short int=0);
  Person(const Person&);          // copy constructor
  ~Person();                      // destructor
  String name() const;            // accessor
  unsigned short int age() const;
protected :
  String namep;
  unsigned short int agep;
```

```
};
class Employee : public Person {
  friend ostream& operator<<(ostream&, const Employee&);
public :
  Employee(String="",unsigned short int=0, unsigned int=0);
  Employee(const Employee&);          // copy constructor
  ~Employee();                        // destructor
  unsigned int salary() const;
private :
  unsigned int sal;
};
```

pers2.h

يبقى تعريف الصفوف وكيفية تحقيقها دون تعديل.

```
#include <iostream.h>
#include <iomanip.h>
#include "pers2.h"
#include "string7.h"

Person::Person(String n, unsigned short int a) : namep(n),agep(a) {
}
Person::Person(const Person& p) : namep(p.namep),agep(p.agep) {
}
Person::~Person() {
}
String Person::name() const {
  return namep;
}
unsigned short int Person::age() const {
  return agep;
}
ostream& operator<<(ostream& os, const Person& p) {
  os.setf(ios::left);
  os << "Name : " << setw(30) << p.namep;
  os << " Age : " << p.agep;
  return os;
}
Employee::Employee(String n,unsigned short int a, unsigned int s)
                                          : Person(n,a),sal(s) {
}
Employee::Employee(const Employee& e) : Person(e.namep,e.agep),sal(e.sal) {
}
Employee::~Employee() {
}
unsigned int Employee::salary() const {
```

```
  return sal;
}
ostream& operator<<(ostream& os, const Employee& e) {
  os << static_cast<Person>(e);     // os << (Person)e;
  os << " Salary : " << e.sal;
  return os;
}
```

pers2.cpp

يكون التغيير فقط على مستوى الاستخدام. وتعتبر استعادة الذاكرة المحجوزة ديناميكياً إحدى المشاكل الهامة.

إن البرنامجَ الذي لا يقوم باستعادة الذاكرة التي يحجزها ديناميكياً برنامجٌ غيرُ صحيح، لأنه سيصل إلى لحظةٍ (عادة في مرحلةِ استخدامٍ متقدمةٍ للبرنامج) تَنْفَدُ فيها الذاكرةُ المخصصة للبرنامج، وهذا مما يؤدي عادة إلى جمود البرنامج. لذلك، يجب عدم إهمال هذا الجانب من صحة البرامج.

```
#include <iostream.h>
#include "pers2.h"

int main() {
  Person* p1 = new Person("Paul",40);
  Person* p2 = new Person("Pierre",30);
  Employee* e1 = new Employee("Claudine",36);
  Employee* e2 = new Employee("Anne",34,30000);
  cout << *p1 << endl << *p2 << endl;
  cout << *e1 << endl << *e2 << endl;
  Person* p = new Person(*p1);
  Employee* e = new Employee(*e2);
  cout << *p << endl;
  cout << *e << endl;
  delete p;
  p = e2;
// e = p1;  // illicit !
  cout << *p << endl;

  delete p1;
  delete p2;
  delete e1;
  delete e2;
// delete p;
  delete e;
```

```
 return 0;
}
```

Tpers2.cpp

يجب أن يحرص المبرمج على إعادة جميع كتل الذاكرة التي حجزها ديناميكياً (بمؤثر *new*)، بواسطة المؤثر *delete*. يجب أن نلاحظ أيضاً أنه:

- لا يجب ألا تَستخدِمَ المؤثر *delete* قائمةً من المتحولات المفصولة بفاصلة (مع أن ذلك صحيح نَحْوياً)، لأن الفاصلة هي مؤثر! فكتابة:

```
delete a,b;
```

لا يعطي عادة النتيجة المرجوة، بل يجب كتابة:

```
delete a;
delete b ;
```

- لا يكفي تحريرُ الذاكرة المحجوزة والمُؤشَّرِ عليها مِن قِبَلِ المتحولات من النمط مؤشر، بل يجب ألا يغيب عن بال المبرمج العمليات التي أُجريَت على هذه المتحولات. ففـــــي مثالنا مثلاً، يؤشر *e2* و*p* على كتلة الذاكرة نفسها، فيجب ألا نحاول، بعد إعادة الذاكرة التي يؤشر عليها *e2*، إعادة العملية على *p*.

يوضح المثال التالي إعادة استخدام الرماز وإمكانات توسيع الرماز الموجود بفضل آليـــــة الوراثة.

لنفترض أن لدينا الصف *Rat* الذي يحقق نمط الأعداد العادية بواجهته العامة.

```
#include <iostream.h>

class Rat {
  friend ostream& operator<<(ostream&, const Rat&);
  friend Rat operator+(const Rat&, const Rat&);
  friend Rat operator-(const Rat&, const Rat&);
public :
  Rat(int = 0, int = 1);
  Rat(const Rat&);
  int numerator() const;
  int denominator() const;
protected :
```

```
 int num, denom;
};
```

rat.h

في هذا المثال المُبسَّط، لدى مستخدمِ الصفِ المؤثراتُ الحسابية "+" و"-"، وتـابعُ إدراجِ عددٍ عاديٍ ضمن دفق الخرج، وإجراءاتٌ للنفاذِ إلى البسطِ والمقام، إضافةً إلى بانٍ افـتراضي وبانِ نسخ.

```
#include <iostream.h>
#include <stdlib.h>
#include "rat.h"

Rat::Rat(int n, int d) : num(n), denom(d) {
 if (denom < 0) {
   denom = abs(denom);
   num *= -1;
 }
}
Rat::Rat(const Rat& r) : num(r.num), denom(r.denom) {
}
int Rat::numerator() const {
 return num;
}
int Rat::denominator() const {
 return denom;
}
ostream& operator<<(ostream& os, const Rat& r) {
 if (r.num < 0) os << '(';
 os << r.num << '/' << r.denom;
 if (r.num < 0) os << ')';
 return os;
}
Rat operator+(const Rat& r1, const Rat& r2) {
 Rat sum(r1.num * r2.denom + r1.denom * r2.num,r1.denom * r2.denom);
 return sum;
}
Rat operator-(const Rat& r1, const Rat& r2) {
 Rat sum(r1.num * r2.denom - r1.denom * r2.num,r1.denom * r2.denom);
 return sum;
}
```

rat.cpp

يقوم البانِي الافتراضي بإنشاء عددٍ عادي و"ضبط" إشارته بحيث تكون إشارته في البسـط

إذا كان سالباً. تجري عمليات جمع وطرح الأعداد العادية وفق الصيغتين التاليتين:

$$\frac{a}{b}+\frac{c}{d}=\frac{ad+bc}{bd},\frac{a}{b}-\frac{c}{d}=\frac{ad-bc}{bd}$$

يمكن استخدام الصف *Rat* مباشرة كما في هذا المثال:

```
#include <iostream.h>
#include "rat.h"

int main() {
  Rat a,b(-1,-3),c(1,-3);
  cout << a << '\t' << b << '\t' << c << endl;
  cout << (a+b) << '\t' << (a+c) << '\t' << (b-c) << endl;
  cout << (2+c) << endl;
  return 0;
}
```

Trat1.cpp

يعطي تنفيذُ البرنامج ما يلي:

```
0/1     1/3     (-1/3)
1/3     (-1/3)  6/9
5/3
```

يمكن تعديل هذه النتائج وتبسيطها بتعريف صفٍ جديدٍ *My_Rat* مشتقٍ من *Rat*، تكـــونُ فيه الأعداد العادية مبسطة. يجدر الإشارة إلى أن هذه العملية ممكنةٌ بمعرفة الواجهة العامـــــة والمحمية فقط للصف السلف *Rat*.

```
#include <iostream.h>
#include <math.h>
#include <stdlib.h>
#include "rat.h"

class My_Rat : public Rat {
public :
  My_Rat(int = 0, int = 1);   // default constructor
  My_Rat(Rat);                // conversion constructor
  My_Rat(const My_Rat&);      // copy constructor
private :
  int gcd(int,int);
};

My_Rat::My_Rat(int n, int d) : Rat(n,d) {
```

```
 int tmp = gcd(n,d);
 num /= tmp;
 denom /= tmp;
}
My_Rat::My_Rat(const My_Rat& mr) : Rat(mr.num, mr.denom) {
 int tmp = gcd(num,denom);
 num /= tmp;
 denom /= tmp;
}
My_Rat::My_Rat(Rat r) : Rat(r) {
 int tmp = gcd(num,denom);
 num /= tmp;
 denom /= tmp;
}
int My_Rat::gcd(int a,int b) {
 a = abs(a); b = abs(b);
 while (a!=b)
   (a > b) ? (a -= b) : (b -= a);
 return a;
}

int main() {
 Rat a(-1,-3),b(1,-3);
 cout << "a=" << a << '\t' << "b=" << b << endl;
 cout << a << '-' << b << '=' << (a-b) << endl;
 My_Rat x(-3,-9),y(1,-3);
 cout << "x=" << x << '\t' << "y=" << y << endl;
 cout << x << '-' << y << '=' << (x-y) << endl;
 My_Rat z = x-y;
 cout << x << '-' << y << '=' << z << endl;
}
```

Trat2.cpp

يتضمن الصف *My_Rat* إضافة إلى باني النسخ والباني الافتراضي بانياً للتحويل. يسمح باني التحويل بإنشاء غرضٍ من النمط *My_Rat* انطلاقاً من غرضٍ من النمط *Rat*. يُعرِّف هذا الصف تابعاً خاصاً private وهو gcd() (Greatest Common Divider) لحساب القاسم المشترك الأعظم لبسط ومقام العدد العادي بغية إجراء التبسيط. ومع أن الصف *My_Rat* لا يرثُ التابعَ الصديقَ الذي يعيدُ تعريفَ مؤثر الإدراج بالنسبة إلى غرض من النمط *Rat* فإن بالإمكان استخدامه لأنه يمكن استخدام الغرض المشتق حيثما يمكن استخدام الغرض السلف. يعطي تنفيذ البرنامج السابق ما يلي:

```
a=1/3              b=(-1/3)
1/3-(-1/3)=6/9
x=1/3              y=(-1/3)
1/3-(-1/3)=6/9
1/3-(-1/3)=2/3
```

4- 3- 2 تعددية الأشكال والربط الديناميكي

تكمن في مفهوم الوراثة، إضافةً إلى إمكان ربط الطريقة التي يجب استدعاؤها عند التنفيذ، قوة البرمجة الغرضية التوجه. لتفعيل *الربط المتأخر late binding* أو *الربط الديناميكي dynamic binding* لتابع ما بدل الربط الساكن المستخدم عادة، يجب على المبرمج وضع الكلمة المحجوزة *virtual* (وهمي) عند التصريح عن التابع العضو.

ولأن باني الغرض يقوم فعلياً بوضع الآلية التي تسمح بالقيام بالربط الديناميكي، فلا يمكن أن يُعرَّف الباني *virtual*. بالمقابل يمكن أن تعرف الهوادم virtual، ومن الأفضل تعريفها كذلك في معظم الحالات.

```
#include <iomanip.h>
#include "string7.h"

class Person {
public :
  Person(String="",unsigned short int=0);
  Person(const Person&);          // copy constructor
  virtual ~Person();              // destructor
  virtual String name() const;    // accessor
  virtual unsigned short int age() const;
  virtual void display();
protected :
  String namep;
  unsigned short int agep;
};
class Employee : public Person {
public :
  Employee(String="",unsigned short int=0, unsigned int=0);
  Employee(const Employee&);          // copy constructor
  virtual ~Employee();                // destructor
  virtual unsigned int salary() const;
  virtual void display();
private :
  unsigned int sal;
```

```
};
```

pers3.h

ليس من الضروري التصريح بأنّ التابعَ المعرَّف *virtual* في الصف السلف هـــو *virtual* في الصف السليل. مع هذا سنقوم بذلك دوماً لسهولة الفهم ووضوح الرماز.

لكي تعمل آلية الربط الديناميكي لتابع ما، يجب أن يكون معرفاً *virtual* على مدى هرمية الوراثة، وأن يكون له التوقيع ونمط الخرج ذاته. جرى في الحقيقة تخفيف القيد علـــى نمـــط الخرج وأصبحت القاعدة: إذا كان التابع معرفاً *virtual* في صف سلف وكان يعيد مَرجعـــاً reference أو مؤشراً pointer على هذا الصف السلف، يمكن زيادة تحميل هذا التابع الوهمـــي في الصف السليل بحيث يكون نمط خرجه مَرجعاً أو مؤشراً على هذا الصف المشتق.

يعمل الربط الديناميكي في هذه الحالة برغم أن أنماط الخرج مختلفة. لا يُطبَّـــقُ الربـــطُ الديناميكي إذا لم يتحقق أحد الشروط السابقة.

```
#include <iostream.h>
#include <iomanip.h>
#include "pers3.h"
#include "string7.h"

Person::Person(String n, unsigned short int a) : namep(n),agep(a) {
}
Person::Person(const Person& p) : namep(p.namep),agep(p.agep) {
}
Person::~Person() {
}
String Person::name() const {
  return namep;
}
unsigned short int Person::age() const {
  return agep;
}
void Person::display() {
  cout.setf(ios::left);
  cout << "Name : " << setw(30) << namep;
  cout << " Age : " << agep;
}
Employee::Employee(String n,unsigned short int a, unsigned int s)
                                        : Person(n,a),sal(s) {
}
```

```
Employee::Employee(const Employee& e) : Person(e.namep,e.agep),sal(e.sal) {
}
Employee::~Employee() {
}
unsigned int Employee::salary() const {
  return sal;
}
void Employee::display() {
  Person::display();
  cout << " Salary : " << sal;
}
```

pers3.cpp

ليس لتعريف التابع *virtual* تأثيرٌ في تحقيقه. ولكن لتطبيق الربط الديناميكي عند التنفيـذ، يجب تطبيق التابع على عنوان الغرض، لا على الغرض مباشرة.

```
#include <iostream.h>
#include "pers3.h"

void display(Person& p) {
  p.display(); cout << endl;
}

int main() {
  Person* p1 = new Person("Paul",40);
  Person* p2 = new Person("Pierre",30);
  Employee* e1 = new Employee("Claudine",36);
  Employee* e2 = new Employee("Anne",34,30000);
  p1->display(); cout << endl;
  p2->display(); cout << endl;
  e1->display(); cout << endl;
  e2->display(); cout << endl;
  Person* p = new Person(*p1);
  Employee* e = new Employee(*e2);
  p->display(); cout << endl;
  e->display(); cout << endl;
  delete p;
  p = e2;
// e = p1;                  // illicit
  p->display(); cout << endl;

  delete p1;               // not delete p1,p2,e1,e2,e !!!
  delete p2;
  delete e1;
  delete e2;
```

```
// delete p;
  delete e;

  Person rp("Lisete",35);
  Employee re("Nino",36,30000);
  rp.display(); cout << endl;
  re.display(); cout << endl;
  display(rp);
  display(re);
  rp = re;
  rp.display(); cout << endl;
  (&rp)->display(); cout << endl;
  display(rp);
  display(re);
  return 0;
}
```

Tpers3.cpp

فيما يلي تنفيذ البرنامج السابق:

```
Name : Paul          Age : 40
Name : Pierre        Age : 30
Name : Claudine      Age : 36 Salary : 0
Name : Anne          Age : 34 Salary : 30000
Name : Paul          Age : 40
Name : Anne          Age : 34 Salary : 30000
Name : Anne          Age : 34 Salary : 30000
Name : Lisete        Age : 35
Name : Nino          Age : 36 Salary : 30000
Name : Lisete        Age : 35
Name : Nino          Age : 36 Salary : 30000
Name : Nino          Age : 36
Name : Nino          Age : 36
Name : Nino          Age : 36
Name : Nino          Age : 36 Salary : 30000
```

بإجراء المقارنة بين النسختين السابقتين نرى أن السطر الأخير قد تغير بسبب تطبيق الربط الديناميكي. فقد أصبحت *p* من النمط *Employee* لأنه جرى إسناد e2 إلى p. في القسم الأخير من البرنامج جرى العمل مباشرة على الغرضين *rp* و *re*. تربط الإجرائية *displaylg*، التي محددها الوحيد مرجعٌ على *Person* ديناميكياً التابع *display()*، باعتبارها تُنَفِّذُ على مَرجِع الغرض المُحَدَّد.

بالمقابل يبقى *rp*، بعد إجراء الإسناد rp=re، غرضاً من *Person* كما شرحنا آنفاً في المقطع 4-1-4.

4- 4 تنفيذُ صندوقِ أدواتٍ غرضي التوجه

يسمح التطبيق المنهجي للمفاهيم الغرضية التوجه ببناء مكتبةٍ من بنى المعطيات التي لهـــا مزايا هامة من أهمها تباين بنى المعطيات هذه. سنقوم فيما يلي بتحقيق الأنماط المجردة التاليـة: *اللائحة List والمِكدَس Stack والشجرة الثنائية Binary Tree*.

4- 4- 1 الصفوف المجردة *abstract classes*

الفكرة الجوهرية في تحقيق مكتبةٍ كهذه هي اشتقاق جميع الصفوف الفعلية من الصـــف السلف نفسه للاستفادة بأكبر قدر من تعدديةِ الأشكال والربطِ الوهمي. يُعرِّف هذا الصـف السلف العمومي (المولِّد) هيكلاً مشتركاً لجميع الصفوف السليلة. فيما يلي تعريف الصـــف السلف المشترك "عمومي" *Generic* الذي سنستخدمه في هذا الفصل.

```
#ifndef _GENERIC_
#define _GENERIC_

#include <iostream.h>
#include "boolean.h"

class Generic {              // abstract base classe
  friend ostream& operator<<(ostream&, const Generic*);
public :
  virtual void display(ostream&) const=0;
  virtual Generic* clone()=0;
  virtual ~Generic();        // virtual destructor
  virtual bool operator==(const Generic*)=0;
  virtual bool operator>(const Generic*)=0;
  virtual bool operator<(const Generic*)=0;
};
#endif
```

generic.h

لا يُعرِّف هذا الصفُ معطياتِ أعضاء ولا بواني، وإنما فقط توابعَ وهميةً إضافة إلى تـــابعِ

صديقٍ يزيد تحميل مؤثر "الإدراج ضمن دفق". يسمى مثل هذا الصف *بالصف المجرد abstract class* لأنه يُعرِّف *تابعاً وهمياً صرفاً pure virtual function* واحداً على الأقل.

نقول عن تابع وهمي عضوٍ إنه صِرْف إذا عرفنا واجهته فقط دون تعريف جسمه. كما يسمى أحياناً تابعاً متأخراً، ويسمى الصف الذي يتضمن مثل هذا التابع الصفَ المتأخر.

ويعني ذلك أن تعريفَ جسم التابع الوهمي الصرف مؤجلٌ إلى وقتٍ لاحق، وذلك ضمن الصفِ المشتقِ من الصف المجرد المتأخر.

للتابع الوهمي الصرف، في C++ ، تركيبٌ نحويٌ خاص، حيث يجب أن تليَ العبارةُ "=0" توقيعَ التابع. وهذه حالة التوابع *display()* و *clone()* و *operator==()* و *operator>()* و *operator<()* في الصف *Generic*.

نقول عن صفٍ إنه مجردٌ إذا حوى تابعاً وهمياً صرفاً واحداً على الأقل. بالنتيجة لا يُسمح بإنشاء أغراض من هذا النمط من الصفوف. لا حاجة إذن إلى تزويد هذه الصفوف ببوانٍ.

يخدم الصف المجرد كصفٍ سلفٍ عند إنشاء صفوف جديدة. ويجب على الصف الذي هو سليل صفٍ مجرد تقديمُ تعريفٍ لكل من التوابع الوهمية الصرفة، وإلا اعتُبِر هو أيضاً صفاً مجرداً ولا يستخدم في إنشاء الأغراض.

```
#include <iostream.h>
#include "generic.h"
#include "boolean.h"

Generic::~Generic() { }

ostream& operator<<(ostream& os, const Generic* g) {
  g->display(os);
  return os;
}
```

generic.cpp

يَستخدِمُ التابعُ غير العضو ومن ثَم غير الوهمي *operator<<(ostream&, const Generic*)* الربطَ الديناميكي، وذلك بتزويد جميع الصفوف غير المتأخرة وسليلة *Generic* بتابعٍ للإدراج ضمن دفق الخرج باستخدام التابع الوهمي *display()*.

تلك هي إحدى مواطن قوة البرمجة الغرضية التوجه. فبالإمكان كتابةُ رمازٍ عمومي يمكن إعادة استخدامه دون أن يعلم مصممه سلفاً أنماط الأغراض التي ستستخدمه. ولا ضرورة لإجراء تمييز للحالات فالربط الديناميكي هو الذي يتكفل بالأمر.

تفيد التوابع الوهمية *operator==()* و *operator>()* و *operator<()* في تحقيق المؤثرات العلاقاتية. أما التابع الوهمي *clone()* فيعيد نسخة عميقة من الغرض الذي يستلم الرسالة.

4- 4- 2 اللوائح الخطية المتباينة *heterogeneous linear list*

رأينا في الفصل الثالث وسائل لغة C++ لإنشاء تحقيقٍ عمومي للأنماط المجردة. يسمح مفهوم *القالب template* بتحقيق بنى معطيات عمومية ولكن متجانسة، أي إن لجميع العناصر التي تعالجها البنيةُ النمطَ ذاته.

نقترح فيما يلي تحقيقاً غرضيَ التوجه للنمط المجرد List (تحقيقٌ متعدد الأشكال ومتباين).

```
#ifndef _LIST_
#define _LIST_

#include <iostream.h>
#include "generic.h"
#include "boolean.h"

class List : public Generic {
  friend ostream& operator<<(ostream&, const List&);
public :
  List();                              // default constructor
  List(const List&);                   // copy constructor de copie
  virtual ~List();                     // destructor
  List& operator=(const List&);        // affectation operator
  bool empty() const;                  //  true if empty
  void insert_first(Generic*);         // insert at the  head of the list
  void insert_last(Generic*);          // insert at the  queu of the list
  void eliminate_first();              // eliminate the head of the list
  Generic* head() const;               // return the head
  virtual void display(ostream&) const;
  virtual List* clone();
private : class Node;                  // forward declaration
protected :
  Node* first                          // point to the first node
```

```
  Node* last;                    // point to the last node
private :
  class Node : public Generic {
  public :
          Node(Generic* g=0,Node* n=0);
          Node(const Node&);
          virtual ~Node();
          virtual void display(ostream&) const;
          virtual Node* clone();
          virtual bool operator==(const Generic*);
          virtual bool operator>(const Generic*);
          virtual bool operator<(const Generic*);
  public :
          Generic* value;
          Node* next;
  };
private :
  void destruct_list(void);
  void copy_list(const List&);
  virtual bool operator==(const Generic*);
  virtual bool operator>(const Generic*);
  virtual bool operator<(const Generic*);
};
#endif
```

liste.h

إن هذا التحقيق للنمط المجرد List (لائحة ذات ترابط بسيط) ليس كاملاً، ولكنه كـــاف لتوضيح الأفكار العامة المعروضة. يتضمن هذا التحقيـــق بعـــض أدوات المعالجـــة وهـــي: *insert_first()* الذي يُدرِجُ مُحَدِّدَه في بداية اللائحة، و *insert_last()* الذي يضيف مُحَدِّدَه في نهاية اللائحة، و *delete_first()* الذي يحذف العنصر الموجود في بداية اللائحة، و *head()* الذي يعيد العنصر الموجود في بداية اللائحة، و *empty()* الذي يبيِّن اللائحة أفارغة هي أم لا؟

ولما كانت اللائحة مؤلفةً من عقد، نُعرِّف الصف *عقدة Node* داخل الصف List، وبحيث يكون خاصاً private. لاحظ أن هذين الصفين يزيدان تحميل التابع الوهمي *clone()* بتغيير نمط خرجه. الميِّزة الأساسية هي انتفاء حاجتنا إلى القيام بتحويل للأنماط لاحقاً.

يُشتَق الصفان *Node* و *List* بصورة عامة *public* من الصف الأساسي المجـــرد *Generic*. ومن ثَم، إذا أردنا استنساخ أغراضٍ انطلاقاً من هذين الصفين، فيجب حتماً تعريف التوابـــع

المتأخرة. هدف التابع *display()* هو إظهار محتوى الغرض المتلقي للرسالة بالصورة المناسبة، أما التابع *clone()* فيعيد عنوانَ نسخةٍ من الغرض الذي طُبِّقَ عليه. ليس من المفيد تحقيق الموثرات العلاقاتية الثلاث في هذه الحالة.

```
#include <iostream.h>
#include <assert.h>
#include "list.h"
#include "boolean.h"

List::Node::Node(Generic* g,Node* n) : value(g),next(n) {
}
List::Node::Node(const Node& n) : value((n.value)->clone()),next(n.next) {
}
List::Node::~Node() {
  delete value;
}
void List::Node::display(ostream& os) const {
  os << value;
}
List::Node* List::Node::clone() {
  return new Node(*this);
}
bool List::Node::operator==(const Generic*) {
  return false;
}
bool List::Node::operator>(const Generic*) {
  return false;
}
bool List::Node::operator<(const Generic*) {
  return false;
}

List::List() : first(0), last(0) {
}
void List::copy_list(const List& l) {
 if (l.first!=0) {
          first = new Node(*l,first);
          Node* last_copy=first;
          for(Node* p = (*first).next; p; p=p->next) {
                  (*last_copy).next = new Node(*p);
                  last_copy = (*last_copy).next;
          }
          last = last_copy;
  } else first = last = 0;
```

```
}
List::List(const List& l) {
  this->copy_list(l);
}
void List::destruct_list(void) {
  Node* tmp = first;
  while (tmp) {
            first = tmp->next;
            delete tmp;
            tmp = first;
  }
}
List::~List() {
            destruct_list();
}
List& List::operator=(const List& l) {
            if (this!=&l) {
                        this->destruct_list();
                        this->copy_list(l);
            }
            return *this;
}
bool List::empty() const {
  return (0 == first) ? true : false;
}
void List::insert_first(Generic* g) {
  first = new Node(g, first);
  if (0 == last) last = first;
}
void List::insert_last(Generic* g) {
  Node* n = new Node(g);
  if (last != 0) {
    last->next = n;
    last = last->next;
  } else last = n;
  if (first == 0) first = last;
}
void List::eliminate_first() {
  assert (first != 0);          // must be done by the one who call
  if (first != last) {
    Node* tmp = first;
    first = first->next;
    delete tmp;
  }
  else {                        // elimination of the last node
    delete first;
```

```
    first=last=0;
  }
}
Generic* List::head() const {
  return first->value;
}
void List::display(ostream& os) const {
  os << '(';
  for(Node* p = first; p; p=p->next) os << p->value << ' ';
  os << ')';
}
List* List::clone() {
  return new List(*this);
}
bool List::operator==(const Generic*) {
  return false;
}
bool List::operator>(const Generic*) {
  return false;
}
bool List::operator<(const Generic*) {
  return false;
}
```

liste.cpp

يُنشئُ باني النسخ في الصف *Node* عقدةً جديدةً بنسخ القيمة الأصلية. يجب ألا يَنسَخ باني النسخ قيم المؤشرات فقط، لأن الأصل والنسخة سيشيران عندها إلى البنية نفسها، ويمكن تصور المشاكل التي تنجم عن الاستدعاء الضمني لبواني النسخ وهوادم الأغراض.

يُنشئُ باني النسخ نسخةً من النمط *List* بحيث تكون مواقع الذاكرة للائحتين منفصلتين تماماً. يعتمد التابع *clone()* على باني النسخ هذا. تُعيدُ التوابعُ التي تُعرِّفُ المؤثرات العلاقاتية القيمةَ *false* مبدئياً.

يمكن انطلاقاً من هذا التحقيق إنشاء لائحةٍ متباينةٍ (أي لائحة أنماطُ عناصرها مختلفة). لِنَرَ كيفية تنفيذ لائحةٍ عناصرها إما أعداد طبيعية أو سلاسل محرفية أو لوائح.

القيد الوحيد الواجب احترامه هو ضرورة اشتقاق جميع العناصر التي تندرج في هذه اللائحة من الصف الأساسي المجرد *Generic*. فيما يلي تعريف للصف *Integer*:

```
#ifndef _INTEGER_
```

```
#define _INTEGER_

#include <iostream.h>
#include "generic.h"
#include "boolean.h"

class Integer : public Generic {
  friend istream& operator>>(istream&, Integer&);
  friend ostream& operator<<(ostream&, const Integer*);
public :
  Integer(int=0);                    // default constructor
  Integer(const Integer&);           // copy constructor
  virtual ~Integer();                // destructor

  virtual void display(ostream&) const;
  virtual Integer* clone();
  virtual bool operator==(const Generic*);
  virtual bool operator>(const Generic*);
  virtual bool operator<(const Generic*);
private:
  int i;
};
#endif
```

Integer.h

نلاحظ هنا أن خرج التابع الوهمي *clone()* هو من النمط *Integer** وليس *Generic** كما هو في الصف *Generic*.

```
#include <iostream.h>
#include <typeinfo.h>
#include <string.h>
#include "Integer.h"
#include "boolean.h"

istream& operator>>(istream& is, Integer& e) {
  return is >> e.i;
}
Integer::Integer(int j) : i(j) {
}
Integer::Integer(const Integer& e) : i(e.i) {
}
Integer::~Integer() {
}
void Integer::display(ostream& os) const {
  os << i;
```

```
}

Integer* Integer::clone() {
  return new Integer(*this);
}
bool Integer::operator==(const Generic* e) {
  if (strcmp(typeid(*e).name(),"Integer") != 0) {
            cerr << "Error : wrong type" << endl;
            return false;
  }
  // return ((*(Integer*)e).i == i) ? true : false;
  return ((*dynamic_cast<Integer*>(const_cast<Generic*>(e))).i == i) ? true : false;
}
bool Integer::operator>(const Generic* e) {
  if (strcmp(typeid(*e).name(),"Integer") != 0) {
            cerr << "Error : wrong type" << endl;
            return false;
  }
  return (i > (*dynamic_cast<Integer*>(const_cast<Generic*>(e))).i) ? true : false;
}
bool Integer::operator<(const Generic* e) {
  if (strcmp(typeid(*e).name(),"Integer") != 0) {
            cerr << "Error : wrong type" << endl;
            return false;
  }
  return (i < (*dynamic_cast<Integer*>(const_cast<Generic*>(e))).i) ? true : false;
}
```

Integer.cpp

نفترض عند كتابة توابع وهمية تُحقّقُ المؤثرات العلاقاتية أنه إذا جرى استدعاء أحد هـذه التوابع يكون الغرض الذي استلم الرسالة من النمط *Integer* وكذلك المعامل الأيمن أيضـاً. يسمح مِقيَس لغة C++ بالحصول على معلومات عن نمط المعطيات المعالجة، عنـد التنفيـذ، وذلك باستخدام التعبير *typeid*.

يعيدُ التعبيرُ *typeid (expression)* مرجعاً على غرض ثابت من النمـط *(const* type_info *type_info&)* تتضمنُ معلومات تُمثّل نَمطَ المعامل. الصف *type_info* معرفٌ سلفاً ضمـن أي نظام C++. يمكن مقارنة غرضين مستنسخين من هذا الصف ("==" و "!="). يُعرَّف هـذا الصف أيضاً الطريقةَ *name()* التي تعيد قيمةً من النمط *const char** يُعرِّفُها مُحقق المـترجم وتمثلُ اسم الصف.

لذلك فإذا لم يكن المعامل ضمن الصف *Integer* من النمط *Integer* تُعـــرَض، عندئـــذ، رسالة خطأ وتعيد التوابع القيمة *false*. سنرى لاحقاً كيفية تحقيق آلية أكـــثر رُقيـــاً وهـــي *الاستثناءات exceptions*.

بعد أن تأكد لنا أن المعامل الأيمن للمؤثر العلاقاتي هو من النمط *Integer*، نصل إلى القيمة المتضمنة في هذا الغرض. هنا تظهر مشكلة، وهي أن للتوابع الوهمية المعنيّة مُحدِّداً من النمـط *const Generic** ونحن نعلم أنه *Integer*. لم يبق إذن إلا إجراء عملية قسرٍ casting للنمـط. في البداية يحذف المُوصِّف *const* باستخدام المؤثر *const_cast*، ثم يجري تحويل الـــــ *Generic* الناتج إلى *Integer* باستخدام مؤثر *dynamic_cast*.

يؤدي تنفيذ التعبير *dynamic_cast <T>(e)* إلى تحويل التعبير *e* إلى النمـــط *T*، حيـــث *T* بالضرورة إما مؤشر أو مَرجِع على صف موجود أو *void**. في الحالة الأخيرة، يجب أن يكون e مؤشراً أيضاً. يُستخدَمُ هذا المؤثر لتحويل مؤشرٍ أو مَرجِع على صفٍ إلى مؤشرٍ أو مَرجِـــع على صفٍ آخر ضمن هرمية الوراثة ذاتها. إذا استحال إجراءُ التحويل يعيد المؤثرُ القيمـــةَ 0 ويُرسَلُ الاستثناء *bad_cast*. لا يمكنُ إجراء التحويل من صفٍ سلف إلى صـــفٍ مشـــتق أو بالعكس إلا إذا كان الصفُ السلف صفاً متعدد الأشكال، أي صفاً يُصرِّح عن تابع وهمـــي، أو وريثاً لتابع وهمي.

يمكن، بعد إجراء التحويل، الوصول إلى عنصر المعطيات العضو في الغـــرض ومقارنتـــه بمقابله في الغرض المتلقي للرسالة. يمكن تعريف صف *سلسلة محرفية* بسهولة.

```
#ifndef _CHAIN_
#define _CHAIN_

#include <iostream.h>
#include "string7.h"
#include "generic.h"

class Chain : public Generic {
  friend istream& operator>>(istream&, Chain&);
public :
  Chain();                  // default constructor
  Chain(const Chain&);      // copy constructor
```

```
  Chain(const String&);     // constructor based on String
  Chain(double);            // constructor based on double
  virtual ~Chain();         // destructor
  bool numeric(double&);  // if the object is a number, it returns the value in the
argument
  String chain();
  virtual void display(ostream&) const;
  virtual Chain* clone();
  virtual bool operator==(const Generic*);
  virtual bool operator>(const Generic*);
  virtual bool operator<(const Generic*);
private:
  String i;
};
#endif
```

chain.h

يستند الصف *Chain* إلى الصف *String* المُعرَّف في الفصل السابق. لا يمكن استعمال الصف *String* كما هو لأنه ليس مشتقاً من الصف *Generic*. الصف *Chain* صفٌ أغنى من الصف *Integer*.

يعيد التابع *chain()* غرضاً من النمط *String* يمثل قيمة السلسلة المحرفية. إذا كانت السلسلة المحرفية مؤلفة من محارف تشكل *double*، يعيدُ التابع *numeric(double&)* القيمةَ *true*، ويتضمن المحدِّد المُمرَّر بالمرجع (العنوان) هذا العددَ بصيغةِ *double*. في الحالة المعاكسة يعيدُ التابعُ القيمةَ *false* ويكون المُحدِّد غير مُعرَّفٍ عند العودة.

الصف *chain* مزوّد أيضاً ببانٍ يسمح بإنشاء سلسلة محرفية انطلاقاً من *double*.

```
#include <iostream.h>
#include <typeinfo.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "string7.h"
#include "chain.h"
#include "boolean.h"

istream& operator>>(istream& is, Chain& e) {
  return is >> e.i;
}
Chain::Chain() : i(String()) {
```

```
}
Chain::Chain(const Chain& e) : i(e.i) {
}
Chain::Chain(const String& s) : i(s) {
}
Chain::Chain(double d) {
 char tmp[20];
 sprintf(tmp, "%g",d);
 i = String(tmp);
}
Chain::~Chain() {
}
bool Chain::numeric(double& d) {
 const char* tmp = i.vers_char();
 char* ptrfin;

 d=strtod(tmp,&ptrfin);
 return (*ptrfin) ? false : true;
}
String Chain::chain() {
 return i;
}
void Chain::display(ostream& os) const {
 os << i;
}

Chain* Chain::clone() {
 return new Chain(*this);
}
bool Chain::operator==(const Generic* e) {
 if (strcmp(typeid(*e).name(),"Chain") != 0) {
        cerr << "Error : wrong type" << endl;
        return false;
 }
 return ((*dynamic_cast<Chain*>(const_cast<Generic*>(e))).i == i) ? true : false;
// return (*(Chain*)e).i == i ? true : false;
}
bool Chain::operator>(const Generic* e) {
 if (strcmp(typeid(*e).name(),"Chain") != 0) {
        cerr << "Error : wrong type" << endl;
        return false;
 }
 return ((*dynamic_cast<Chain*>(const_cast<Generic*>(e))).i > i) ? true : false;
}
bool Chain::operator<(const Generic* e) {
 if (strcmp(typeid(*e).name(),"Chain") != 0) {
```

```
        cerr << "Error : wrong type" << endl;
        return false;
 }
 return ((*dynamic_cast<Chain*>(const_cast<Generic*>(e))).i < i) ? true : false;
}
```

chain.cpp

بعد الانتهاء من تعريف هذين الصفين، يصبح من السهل تحقيق لائحة متباينة.

```
#include <iostream.h>
#include "boolean.h"
#include "generic.h"
#include "Integer.h"
#include "chain.h"
#include "list.h"

int level = 1;        // indicates the imbrication level

bool get_value(Generic*& pg) {
 char i;
 cout << level << ":: <I>nteger, <C>hain, <L>ist ? " << flush;
 cin >> i;
 switch (i) {
  case 'i' :
  case 'I' : {
                Integer e;
                cout << "Integer ? " << flush;
                cin >> e;
                pg = new Integer(e);
                return true;
  }
  case 'c':
  case 'C' : {
                Chain c;
                cout << "Chain ? " << flush;
                cin >> ws >> c;
                pg = new Chain(c);
                return true;
  }
  case 'l' :
  case 'L' : {
                List l;
                Generic* element;
                level++;
                while (get_value(element)) l.insert_first(element);
                level--;
```

```
                pg = new List(l);
                return true;
    }
        default : return false;
  }
}

int main() {

  List l,l1;
  Generic* pg;
  while (get_value(pg)) l.insert_last(pg);
  cout << &l << endl;
  l1 = l;
  cout << &l1 << endl << &l << endl;
  while(!l.empty()) {
    l.eliminate_first();
        cout << &l << endl;
  }
  cout << &l1 << endl;
  return 0;
}
```

يبين هذا المثالُ طريقةً لإنشاء لائحـــة أنمـــاط عناصرهـــا مختلفـــة. يحـــاول التـــابع *get_value(Generic*&)* إنشاءَ غرضٍ مشتقٍ من الصف *Generic*. يُطلَبُ من المســـتخدم في هذه الحالة إدخال القيمِ الملائمة. يعيدُ هذا التابع القيمة *true* إذا نجحَ في إنشاءِ هذا الغـــرض ويضعُه في نهاية اللائحة. لإنشاء غرض من هذا النمط يَطلبُ التابعُ من المستخدمِ إدخالَ عددٍ صحيح أو سلسلة محرفية أو لائحة. بذلك يمكن أن يكون عنصر اللائحة المبني بهذه الطريقـــة في حد ذاته لائحة أخرى وهكذا. يبين المثال أيضاً أن زيادة تحميل مؤثر الإسناد تُنشِئُ لائحة منسوخة من اللائحة الأصلية. يجري بعد ذلك إظهار اللائحة المنشأة بعد أن حُذف منـــها العنصر الأول. بهدف التوضيح يَسبقُ إظهارَ اللائحة عددٌ يدلُ المســـتخدمَ علـــى مســـتوى التداخل imbrication الذي توجد فيه. فيما يلي مثال على كيفية الاستخدام:

```
1:: <I>nteger, <C>hain, <L>ist ? l
Integer ? 2
1:: <I>nteger, <C>hain, <L>ist ? C
Chain ? Nino
1:: <I>nteger, <C>hain, <L>ist ? L
2:: <I>nteger, <C>hain, <L>ist ? I
```

```
Integer ? 34
2:: <I>nteger, <C>hain, <L>ist ? X
1:: <I>nteger, <C>hain, <L>ist ? C
Chain ? Lisete
1:: <I>nteger, <C>hain, <L>ist ? X
(2 Nino (34 ) Lisete )
(2 Nino (34 ) Lisete )
(2 Nino (34 ) Lisete )
(Nino (34 ) Lisete )
((34 ) Lisete )
(Lisete )
()
(2 Nino (34 ) Lisete )
```

يبين هذا المثال أن اللائحتين l و ll تعملان على كتلتين منفصلتين من الذاكرة. لنؤكد هنا أهميةَ كون الهوادم وهمية. في الواقع، عند هدم لائحة من قبل الهادم تُفحَصُ جميعُ العقد على التتالي وتُهدَمُ باستدعاء المؤثر *delete*. فإذا لم تكن هذه العقد مرتبطة ديناميكياً وجب على المترجم الاعتماد على المعلومات التي بحوزته عند الترجمة: تتألفُ اللائحةُ من عقدٍ تشير عناصرها إلى أغراض من النمط *Generic*. أي إذا كان العنصر ضمن اللائحة من النمط *List* فمن الواضح أن المترجمَ سيَستدعي فقط هادم *Generic* بدلاً من استدعاء هادمِ الصف *List*! توجد إذن طريقتان في C++ لإنشاء بنى عمومية (مولدة generic): مفهوم القالب *template* وآلية الوراثة. يسمح مفهوم القالب بإنشاء أنماط جديدة مجردة، عمومية ومتجانسة. بالمقابل، تسمح التقنية المعتمدة على مفهوم الوراثة، كما جرى توضيحه آنفاً من أجل النمط *List*، بتحقيق أنماط مجردة متباينة. يمكن أن تتضمن *List* إذن جميع الأغراض الممكنة شرطَ أن تكون سليلة الصف *Generic*! هذه إحدى نقاط قوة البرمجة الغرضية التوجه.

يمكننا محاولة دمج الطريقتين للحصول على فعالية أكبر. ولكن يجب الحذر لأن المشاكل التي يمكن أن تظهر تكون في بعض الأحيان صعبة الاكتشاف. لتوضيح ذلك نأخذ مثالَ النمط المجرد مِكدَس *Stack* المحقق باستخدام مفهوم القالب. ليس هناك ما يمنعنا لاحقاً من استنساخ مِكدَسٍ لعناصر من النمط *Generic*.

```
#ifndef _STACK_
#define _STACK_
```

```
#include <iostream.h>
#include <assert.h>
#include <mem.h>
#include "boolean.h"

template <class Type>
class Stack {
  friend ostream& operator<<(ostream&, const Stack<Type>&);
public :
  Stack();                          // default constructor
  Stack(int);                       // constructor with a given size
  ~Stack();                         // destructor
  Stack(const Stack&);              // copy constructor
  Stack& operator=(const Stack&);
  inline bool is_empty() const;
  inline Type top() const;
  void push(Type);
  void pop();
private :
  enum constants { STACK_EMPTY = -1, MAX_DEFAULT = 50 };
  const int MAX_STACK;              //  maximal size of the stack
  Type* p;                          // the stack implemented as a table
  int pp;                           // pointer on the top of the stack
};

template <class Type>
Stack<Type>::Stack() : MAX_STACK(MAX_DEFAULT),p(new
Type[MAX_DEFAULT]),pp(STACK_EMPTY) {
  assert(p!=0);
  memset(p,'\x00',MAX_STACK*sizeof(Type));
}

template <class Type>
Stack<Type>::Stack(int max) : MAX_STACK(max),p(new Type[max]),
pp(STACK_EMPTY) {
  assert(p!=0);
  memset(p,'\x00',MAX_STACK*sizeof(Type));
}

template <class Type>
Stack<Type>::~Stack() {
          for(int i=0;i <= pp; i++) delete p[i];
          delete [] p;
//     delete [] p;   doesn't work, because there is no destructor for Generic*!!
}
```

```
template <class Type>
Stack<Type>::Stack(const Stack<Type>& stack)
                              : MAX_STACK(stack.MAX_STACK),p(new Type
[stack.MAX_STACK]),pp(stack.p) {
  assert(p!=0);
  memset(p,'\x00',MAX_STACK*sizeof(Type));
  memcpy(p,stack.p,sizeof(Type)*(pp+1));
}

template <class Type>
Stack<Type>& Stack<Type>::operator=(const Stack<Type>& stack) {
  if (this != &stack) {
    for(int i=0;i <= pp; i++) delete p[i];
          delete [] p;
    assert(MAX_STACK >= stack.MAX_STACK);
    p = new Type [stack.MAX_STACK];
    assert(p!=0);
    pp = stack.pp;
    memcpy(p,stack.p,sizeof(Type)*(pp+1));
  }
  return *this;
}

template <class Type>
void Stack<Type>::push(Type t) {
  pp++;
  if (pp < MAX_STACK)
    p[pp] = t;
  else
    cerr << "Stack pleine" << endl;

}

template <class Type>
void Stack<Type>::pop() {
  delete p[pp];
  pp-- ;
}

template <class Type>
inline bool Stack<Type>::is_empty() const {
  return (pp == STACK_EMPTY) ? true : false;
}

template <class Type>
inline Type Stack<Type>::top() const {
```

```
 return p[pp];
}

template <class Type>
ostream& operator<<(ostream& os, const Stack<Type>& stck) {
 Stck<Type> tmp(stck);
 while (!tmp.is_empty()) {
        os << tmp.top() << '\t';
        tmp.pop();
 }
 os << endl;
 return os;
}
#endif
```

Stack.h

رأينا مثالاً مماثلاً في الفصل الثالث، ولكن يجب علينا هنا تغيير الهادم لأننـــا نرغـــب في استنساخ مِكدَس من المؤشرات على *Generic* الذي لم يُعرَّف له أي هادم! يجب علينـــا إذن مسح جدول المؤشرات واستدعاء الهادم صراحة. نلاحظ هنا أيضاً ضرورة تعريف الهـــوادم كتوابع وهمية للتوثق من استدعاء الهادم الصحيح في جميع الحالات.

تتضمن هذه النسخة، مقارنة بالنسخة الموجودة في الفصل الثالث، باني نسخ وتحميـــلاً زائداً لمؤثر الإسناد إضافيين. تُعرَّف هذه التوابع كما لو كانت تعمل على الأغراض نفســها، لا على عناوينها. ينتج من ذلك أن للأصل وللنسخة كتلة الذاكرة نفسها وما يولده ذلك من مشاكل وأخطاء.

يجب إذن على المبرمج، عند الاستنساخ، تعديل هذا السلوك الافتراضي بزيـــادة تحميـــل هذين التابعين للنمط **Generic*. أي نُنشئُ "نسخة" من المِكدَس الأصلي لتفادي أن يعمـــل المِكدَس الأصلي والنسخة على المعطيات نفسها عند إجراء النسخ. وهكذا، إذا هُدِم لاحقـــاً المِكدَس الأصلي أو النسخة لا يؤدي ذلك إلى ضياع الآخر.

```
#include <iostream.h>
#include <new.h>
#include "Stack.h"
#include "boolean.h"
#include "Generic.h"
#include "Integer.h"
```

```
template<>
Stack<Generic*>::Stack(const Stack<Generic*>& stack)
                                        : MAX_STACK(stack.MAX_STACK),
p(new Generic*[stack.MAX_STACK]), pp(stack.pp) {
  assert(p!=0);
  for(int i=0; i<=pp; i++) p[i] = ((stack.p[i])->clone());
}

template<>
Stack<Generic*>& Stack<Generic*>::operator=(const Stack<Generic*>& stack) {
  if (this != &stack) {
     for(int i=0;i <= pp; i++) delete p[i];
     delete [] p;
     assert(MAX_STACK >= stack.MAX_STACK);
           p = new Generic*[stack.MAX_STACK];
     assert(p!=0);
     pp = stack.pp;
     for(int i=0; i<=pp; i++) p[i] = (stack.p[i])->clone();
  }
  return *this;
}

template ostream& operator<<(ostream& os, const Stack<Generic*>& stack);

int main() {
  set_new_handler(0);
  Stack<Generic*> p,p1;
  int e;
  while (cin >> e) {                    // create a stack of integers
    Integer* pe = new Integer(e);
    p.push(pe);
  }
  cout << p;
  p1 = p;                               // copy a stack into another
  cout << p1 << endl << p << endl;      // display the two stacks
}
```

Stack.cpp

يبين هذا المثال وجوب الحذر عند دمج نموذجين، وأن من الأفضل غالباً فصلهما.

4-4-3 تركيبٌ أم وراثة

يجب أحياناً على المبرمج، عند تحقيق نمطٍ مجرد، الاختيار بين الوراثة والتركيب. فإذا كان

يَعتَبِرُ أن صفهُ الجديد "هو" صف موجود، عندها يستخدم مفهوم الوراثة. أما إذا كان يعتبر أن صفه الجديد يتألف من غرضٍ من صفٍ موجود، عندها يستخدمُ التركيبَ بإنشاءِ عنصر معطيات عضو من نمط الصف الموجود.

لنأخذ على سبيل المثال النمطَ المجرد مِكدَس *Stack* يمكن رؤية المِكدَس على أنه لائحة *List* يمكن تخصيصها، ويمكن من ثَم تعريفه بالوراثة انطلاقاً من الصف *List*.

```
#ifndef _STACKG_
#define _STACKG_

#include <iostream.h>
#include "generic.h"
#include "list.h"
#include "boolean.h"

class Stack : public List {
public :
  Stack();                              // default constructor
  Stack(const Stack&);                  // copy constructor
  virtual ~Stack();                     // destructor
  Stack& operator=(const Stack&);
  bool is_empty() const;
  Generic* top() const;
  void push(Generic*);
  void pop();
};
#endif
```

Stackg.h

لم يجر إغناء النمط الجديد Stackنسبة إلى النمط List. يمكن تحقيق التوابع بسهولة:

```
#include <iostream.h>
#include "boolean.h"
#include "Stackg.h"
#include "List.h"

Stack::Stack() {}

Stack::~Stack() {}

Stack& Stack::operator=(const Stack& p) {
  List::operator=(p);
  return *this;
```

```
}

void Stack::push(Generic* t) {
  insert_first(t);
}

void Stack::pop() {
  eliminate_first();
}

bool Stack::is_empty() const {
  return List::empty();
}

Generic* Stack::top() const {
  return head();
}
```

Stackg.cpp

لا تُضيفُ الوراثة الكثير في هذه الحالة. بل لأن *Stack* يرث جميع طرائق الصف *List*، يمكن للمبرمج أن يكتب:

```
Stack p;
p.insert_first (new Integer(7)) ;
```

وهذا خطأٌ منطقي كبير في استخدام النمط المجرد *Stack*.

يمكن أيضاً النظر إلى المِكدَس على أنه مكوّن من لائحة. عندما يكون بالإمكان الاختيار بين العلاقتين التاليتين "هو" و "يتألف من" فمن الأفضل استخدام العلاقة الأخيرة دون استخدام مفهوم الوراثة. يجب استخدام مفهوم الوراثة فقط في حالة كون العلاقة من النمط "هو". نحصل عندها على:

```
#include "generic.h"
#include "list.h"
#include "boolean.h"

class Stack : public Generic {
public :
  Stack();                          // default constructor
  Stack(const Stack&);              // copy constructor
  virtual ~Stack();                 // destructor
  Stack& operator=(const Stack&);
  bool is_empty() const;
```

```
  Generic* top() const;
  void push(Generic*);
  void pop();
  virtual void display(ostream&) const;
  virtual Generic* clone();
private :
  List l;
private :
  virtual bool operator==(const Generic*);
  virtual bool operator>(const Generic*);
  virtual bool operator<(const Generic*);
};
#endif
```

Stackl.h

تحقيق التوابع السابقة عملية بسيطة:

```
#include <iostream.h>
#include "boolean.h"
#include "stackl.h"

Stack::Stack() {
}

Stack::~Stack() {
}

Stack::Stack(const Stack& p) : l(p.l) {
}

Stack& Stack::operator=(const Stack& p) {
  l=p.l;
  return *this;
}

void Stack::push(Generic* t) {
  l.insert_first(t);
}

void Stack::pop() {
  l.eliminate_first();
}

bool Stack::is_empty() const {
  return l.empty();
}
```

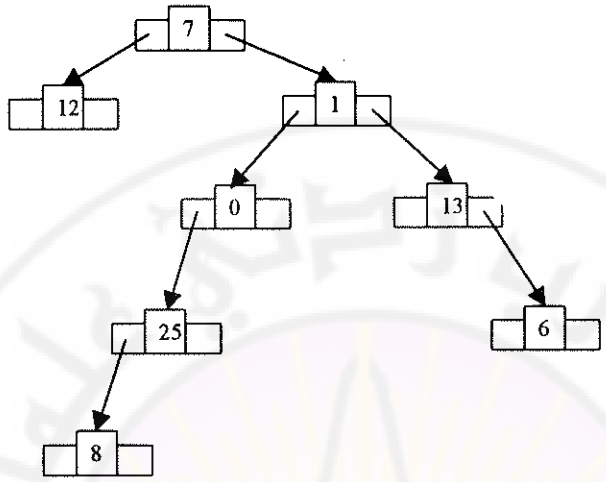
```
Generic* Stack::top() const {
  return l.head();
}
void Stack::display(ostream& os) const {
  os << &l;
}
Generic* Stack::clone() {
  return new Stack(*this);
}
bool Stack::operator==(const Generic* p) {
  return false;
}
bool Stack::operator>(const Generic* p) {
  return false;
}
bool Stack::operator<(const Generic* p) {
  return false;
}
```

Stackl.cpp

4- 4- 4 الشجرة الثنائية المتباينة

نقوم فيما يلي بتحقيق النمط المجرد *للشجرة الثنائية Binary Tree* بُغية توضيح مزايا البرمجة الغرضية التوجه على أمثلة لبنى غير خطية، حيث يكون لإعادة استخدام الرماز قيمةٌ كبـــيرةٌ لأن خوارزميات المعالجة، عادةً، أكثر تعقيداً.

الشجرةُ الثنائيةُ هي إما شجرةٌ خالية أو عقدةٌ (*stl, n, str*) حيث تمثل *n* القيمةَ، أمـــا *stl* و *str* فهما شجرتان ثنائيتان فرعيتان الأولى يُسرى والثنائية يُمنى. فالشجرة الثنائية إذن بنيـــةٌ مؤلفةٌ من عقد، تتألفُ كلُ عقدةٍ من قيمةٍ ومن سليلين على الأكثر. تسمى العقدة التي ليس لها أي سليل *ورقة leaf*. تتألف الشجرة في المثال التالي من ثلاث أوراقٍ 12 و 8 و 6. تسمى العقدة التي قيمتها 7 *جذرَ root* الشجرة.



الشكل 4–7: مثال لشجرة ثنائية عمقها 5 ومؤلفة من ثمانِ عقد، منها ثلاث أوراق.

هذا التعريف ليس كافياً لإنشاء الغرض المدروس، فمن الضروري تعريفُ مجموعة من التوابع لمعالجة هذا الغرض. في البداية، يجب تعريف التوابع الأولية الأساسية التالية:

- بانٍ يأخذ كمحددات stl ,str وقيمة العقدة؛
- النوافذ left() و right() و root() التي تعيد على التتالي stl و str وجذرَ الشجرةِ الثنائيةِ المتلقيةِ للرسالة؛
- يدل التابع المنطقي is_empty() على كون الشجرة الثنائية خالية أو لا؛
- nods_nb() يعيد عدد العقد المكونة للشجرة؛
- height() يدل على ارتفاع الشجرة، ويُعرَّف ارتفاع الشجرة بأنه عدد العقد الموجودة على مسارٍ يبدأ من الجذر وينتهي بأبعد ورقة عنه. ارتفاع الشجرة في الشكل السابق يساوي 5؛
- المُعدِّل affect_root() يسمحُ بتغييرِ جذرِ الشجرة؛
- close() و display() لأن الشجرة الثنائية مشتقةٌ من الصف Generic.

لنأخذ مسألةَ فحصِ عُقدِ الشجرة. كيف يمكنُ إجراء معالجةٍ منهجيةٍ لعقدِ الشجرة الثنائية (المرور بكل عقدة من عقد الشجرة وفق ترتيبٍ معينٍ بغية إجراء معالجةٍ معينة عليها)، فيما يلي خوارزمية عودية للحل:

```
scan(a) = if not (is_empty(a)) then
                scan(left(a)),
                treat(root(a)),
                scan(right(a))
          fin_if
```

إذا كانت المعالجة المطلوبة هي إظهار قيمة العقدة، نحصل في المثال السابق على: 12، 7، 8، 25، 0، 1، 13، 6.

نسمي هذا الشكل من العبور بالعبور وفق الترتيب البيني *inorder* حيث تجري المعالجة بين استدعاءين عوديين. إذا جرت المعالجة قبل عبور *stl* و *str* نتحدث عن عبور وفق الترتيب السابقي *preorder*. نحصل إذن في حالة المثال السابق نفسه على: 7، 12، 1، 0، 25، 8، 13، 6. أما إذا جرت المعالجة بعد عبور *stl* و *str* فنقول إن العبور وفق الترتيب اللاحقي *postorder*، ونحصل بذلك على 12، 8، 25، 0، 6، 13، 1، 7.

يقودنا هذا إلى تعريف ثلاثة توابع أولية إضافية وهي *postorder(), inorder(), preorder()*.

نفترض في المثال التالي أن المعالجة في حالة التابعين *postorder(), preorder()* هي إظهار قيمة العقد فقط، في حين أن مُحدِّد التابع *inorder()* هو تابع المعالجة الذي يجب أن يعطيه المبرمج.

```
#ifndef _ARBREBIN_
#define _ARBREBIN_

#include <iostream.h>
#include "generic.h"
#include "boolean.h"

typedef void (*P_FUNCTION)(Generic*);

class BinTree : public Generic {
```

```
public :
  class Node;                                  // forward declaration
  BinTree();                                   // default constructor
  BinTree(const BinTree&);                     // copy constructor
  BinTree(BinTree*, Generic*, BinTree*);       // compose
  virtual ~BinTree();                          // destructor
  BinTree& operator=(const BinTree&);          // affectation overloading
  void treat_inorder(P_FUNCTION) const;        // inorder scanning with treatment
  void preorder(ostream&) const;               // preorder scanning
  void postorder(ostream&) const;              // postorder scanning
  virtual void display(ostream&) const;
  virtual BinTree* clone();
  void affect_root(Node*);          // root modifier
  bool is_empty() const;            // empty tree?
  Node* root() const;               // root accessor
  BinTree* left() const;            // return a copy of stl
  BinTree* right() const;           // return a copy of str
  int nb_nodes() const;             // number of nodes
  int height() const;               // height og the tree
protected :
  Node* rt;
public :
  class Node : public Generic {
    friend void destruct(BinTree::Node*);
  public :
            Node(Generic* g=0);
            Node(const BinTree::Node&);
            BinTree::Node& operator=(const BinTree::Node&);
            virtual ~Node();
            BinTree::Node* left();
            BinTree::Node* right();
            Generic* value();
            void affect_left(BinTree::Node*);
            void affect_right(BinTree::Node*);
            void affect_value(Generic*);
            virtual void display(ostream&) const;
            virtual Node* clone();
            virtual bool operator==(const Generic*);
            virtual bool operator>(const Generic*);
            virtual bool operator<(const Generic*);
  public :
            BinTree::Node* l;
            BinTree::Node* r;
            Generic* val;
  };
private ;
```

```
 virtual bool operator==(const Generic*);
 virtual bool operator>(const Generic*);
 virtual bool operator<(const Generic*);
};
#endif
```

BinTree.h

يُعرِّف الصف المسمى شجرة ثنائية *BinTree* صفاً مساعداً وهو صف العقدة *Node*. ولما كان تحقيق عدة توابع أولية لـ *Node* يتطلّب النفاذ إلى المعطيات المتضمنة، فهي بالضرورة عامة. يتطلّبُ الصف *Node* أيضاً بوانيَ وهادماً وهمياً ونوافذ ومُعدِّلات، إضافة إلى زيادة تحميل التوابع الموروثة من الصف السلف *Generic*.

```
#include <iostream.h>
#include "BinTree.h"
#include "Generic.h"
#include "boolean.h"

BinTree::Node::Node(Generic* v) : l(0), r(0), val(v) {
}
BinTree::Node::Node(const BinTree::Node& n) : l(0), r(0),val(n.val->clone()) {
// no need to copy the pointers
}
BinTree::Node::~Node() {
 delete val;
}
BinTree::Node& BinTree::Node::operator=(const BinTree::Node& n) {
 if (this != &n) {
   delete val;          // destruct left operand
   l = r = 0;           // no need to copy the pointers
   val = n.val->clone();
 }
 return *this;
}
BinTree::Node* BinTree::Node::left() {
 return l;
}
BinTree::Node* BinTree::Node::right() {
 return r;
}
Generic* BinTree::Node::value() {
 return val;
}
void BinTree::Node::affect_left(BinTree::Node* n) {
```

```
 l = n;
}
void BinTree::Node::affect_right(BinTree::Node* n) {
 r = n;
}
void BinTree::Node::affect_value(Generic* v) {
 val = v;
}
BinTree::Node* BinTree::Node::clone() {
 return new BinTree::Node(*this);
}
void BinTree::Node::display(ostream& os) const {
 val->display(os);
}
bool BinTree::Node::operator==(const Generic* l) {
 return val->operator==(l);
}
bool BinTree::Node::operator>(const Generic* l) {
 return val->operator>(l);
}
bool BinTree::Node::operator<(const Generic* l) {
 return val->operator<(l);
}

BinTree::BinTree() : rt(0) {
}
static void copy_tree(BinTree::Node* k, BinTree::Node*& n) {
 if (k!=0) {
        n = (BinTree::Node*)k->clone();
        copy_tree(k->l,n->l);
        copy_tree(k->r,n->r);
 }
 else n = 0;
}
BinTree::BinTree(const BinTree& a) {
 BinTree::Node* copy;
 copy_tree(a.rt,copy);
 rt = copy;
}
BinTree::BinTree(BinTree* stl, Generic* v, BinTree* str) {
 BinTree::Node* tmp = new BinTree::Node(v);
 tmp->l = stl->rt;
 tmp->r = str->rt;
 rt = tmp;
 stl->rt = 0; delete stl;
 str->rt = 0; delete str;
```

```
}
void destruct(BinTree::Node* n) {
 if (n) {
  destruct(n->l);
  destruct(n->r);
  delete n;
 }
}
BinTree::~BinTree() {
 destruct(rt);
}
BinTree& BinTree::operator=(const BinTree& a) {
 if (this != &a) {
  destruct(rt);
        copy_tree(a.rt,rt);
 }
 return *this;
}
static void scan_inorder(P_FUNCTION pf, BinTree::Node* p) {
 if (p) {
  scan_inorder(pf,p->left());
  pf(p->value());
  scan_inorder(pf,p->right());
 }
}
void BinTree::treat_inorder(P_FUNCTION pf) const {
 if (rt) scan_inorder(pf,rt);
}
static void scan_preorder(ostream& os, BinTree::Node* p) {
 if (p) {
  os << (p->value()) << ' ';
  scan_preorder(os,p->left());
  scan_preorder(os,p->right());
 }
}
void BinTree::preorder(ostream& os) const {
 if (rt) {
  scan_preorder(os,rt);
  os << endl;
 }
 else os << "Empty Tree" << endl;
}
static void scan_postorder(ostream& os, BinTree::Node* p) {
 if (p) {
  scan_postorder(os,p->left());
  scan_postorder(os,p->right());
```

```
    os << (p->value()) << ' ';
  }
}
void BinTree::postorder(ostream& os) const {
  if (rt) {
    scan_postorder(os,rt);
    os << endl;
  }
  else os << "Empty Tree" << endl;
}
static void preorder_output(ostream& os, BinTree::Node* n) {
  if (n) {
          os << n->value() << ' ';
          preorder_output(os,n->left());
          preorder_output(os,n->right());
  }
  else os << "nil ";
}
void BinTree::display(ostream& os) const {
  preorder_output(os,root());
}
void BinTree::affect_root(BinTree::Node* l) {
  rt = l;
}
BinTree* BinTree::clone() {
  return new BinTree(*this);
}
bool BinTree::is_empty() const {
  return (rt == 0) ? true : false;
}
BinTree::Node* BinTree::root() const{
  return rt;
}
BinTree* BinTree::left() const {
  BinTree* tmp = new BinTree;
  copy_tree(rt->l,tmp->rt);
  return tmp;
}
BinTree* BinTree::right() const {
  BinTree* tmp = new BinTree;
  copy_tree(rt->r,tmp->rt);
  return tmp;
}
static int count_nodes(BinTree::Node* n) {
  if (n==0) return 0;
  else return 1+count_nodes(n->l)+count_nodes(n->r);
```

```
}
int BinTree::nb_nodes() const {
  return count_nodes(rt);
}
static int max(int a, int b) {
  return (a > b) ? a : b;
}
static int mesure_height(BinTree::Node* n) {
  if (n==0) return 0;
  else return 1+max(mesure_height(n->l),mesure_height(n->r));
}
int BinTree::height() const {
  return mesure_height(rt);
}
bool BinTree::operator==(const Generic* l) {
  return false;
}
bool BinTree::operator>(const Generic* l) {
  return false;
}
bool BinTree::operator<(const Generic* l) {
  return false;
}
```

BinTree.cpp

ولما كنّا نقوم بمعالجة بنى ديناميكية، فمن الضروري وجود التابع *Clone()* الذي يسمح بإجراء عمليات نسخ "عميقة". مهما كان تعقيد البنية النهائية للشجرة، يستطيع هذا التابع المربوط ديناميكياً، إذا عُرِّفَ تعريفاً صحيحاً من أجل جميع المشاركين، بناء هذه النسخة العميقة.

لا تُعقّدُ توابعُ المقارنةِ على مستوى العقدة *Node* عملها، ذلك أنها توكل مهمة المقارنة إلى عنصر المعطيات *val* الذي ينجز ذلك بالاستعانة بالربط الديناميكي.

يقوم باني النسخ *BinTree* فقط بابتداء الإجراء العودي الذي يقع في مستوى المعطيات ذات النمط *BinTree::Node*. من الضروري إذن إضافة تابع عودي مساعد *copy_tree()* يقوم فعلاً بهذا الإجراء بمسح الشجرة الأصلية وفق الترتيب البيني، منشئاً بالتدريج النسخةَ ضمن المُحَدِّد المَمَرّ بالعنوان.

يأخذ الباني الثاني ثلاثة محدِّدات، وينشئُ عقدةً جديدةً حقلاها *l* و *r* هما جذرا *stl* و *str*

(محدِّدا البانى). تُصبح هذه العقدةُ جذرَ الشجرة الناتجة. يجب الانتباه إلى هدم *str* و *stl* (وهما من النمط *BinTree*) دون هدم عقدتيهما لأنهما يكوِّنان جزءاً من الشجرة الجديدة الناتجة!

يحتاج هادم الصف *BinTree* إلى تابع مساعدٍ أيضاً يسمى *destruct()* يقوم بالمهمة بعبور البنية وفق الترتيب اللاحقي وباستدعاء الهادم في آخر عبور له في كل عقدة.

يَدمجُ التحميلُ الزائدُ لمؤثر الإسناد في الصف *BinTree* هذين التابعين المساعدين، وذلك بهدم البنية الشجرية لعقد المعامل الأيسر، ثم بإجراء نسخ للمعامل الأيمن.

تحتاج جميع إجرائيات عبور الشجرة *postorder(), inorder(), preorder()* إلى تابع مساعدٍ لأن العمل يجري على مستوى العقد. وكما ذكرنا آنفاً، تتميز الإجرائية *inorder()* عن الإجرائيتين الباقيتين بأنها تستدعي التابع الموجود ضمن مُحدِّداتها بتمرير قيمة العقدة مُوْسِطاً له، بدلاً من إدراج قيمة العقدة في دفق الخرج.

يُعيد كلا التابعين *right()* و *left()* على التتالي نسخةً من الشجرة الفرعية اليسرى واليمنى للشجرة المتلقية للرسالة.

فيما يلي تطبيقٌ على الصف *BinTree*. لنوضح أنه لا يوجد حتى الآن أي ترتيب لتوزع القيم في الشجرة.

```
#include <iostream.h>
#include "BinTree.h"
#include "Integer.h"

static void preorder_input(istream& is,BinTree::Node*& n) {
  int value;
  BinTree::Node* tmp;
  is >> value;
  if (value != -1) {
    n = new BinTree::Node(new Integer(value));
    preorder_input(is,tmp);
    n->affect_left(tmp);
    preorder_input(is,tmp);
    n->affect_right(tmp);
  } else n = 0;
}
```

```
istream& operator >>(istream& is,BinTree& b) {
  BinTree::Node* tmp;
  cout << "Enter the integers of the tree, -1 for NIL : " << flush;
  preorder_input(is,tmp);
  b.affect_root(tmp);
  return is;
}
void treat_integer(Generic* p) {
  cout << p << ' ';
}
int main() {
  BinTree b,b2;
  Integer c;
  cin >> b;
  b.treat_inorder(treat_integer);          // display the tree using inorder
  cout << endl;
  b.preorder(cout);                        // display the tree using preorder
  b.postorder(cout);                       // display the tree using postorder
  cout << &b << endl;
  cout << "Height : " << b.height() << endl;
  cout << "Number of nodes : " << b.nb_nodes() << endl;
  BinTree b1(b);
  b1.treat_inorder(treat_integer);         // display the tree using inorder
  cout << endl;
  cout << &b1 << endl;                     // display the tree using preorder
  cout << "Height : " << b1.height() << endl;
  cout << "Number of Nodes : " << b1.nb_nodes() << endl;
  b2 = b1;
  cout << &b2 << endl;
  cout << &b1 << endl;                     // display the tree using preorder

  BinTree* stl = b1.right();
  stl->treat_inorder(treat_integer);       // display the tree using inorder
  cout << endl;
  cout << stl << endl;                     // display the tree using preorder
  cout << "Height : " << stl->height() << endl;
  cout << "Number of Nodes : " << stl->nb_nodes() << endl;
  delete stl;
  return 0;
}
```

Tbin1.cpp

من الضروري أن نذكر هنا أن هذه الشجرة ذات طبيعة متباينة heterogeneous شـرط أن تكون القيم الموضوعة في الشجرة مشتقةً من الصف *Generic*.

بالرغم من تَضمُّنِ الصف *BinTree* لبوانٍ، فلا شيء يمنعنا من كتابة إجرائيـــة لإنشـــاء شجرة. تَسمحُ زيادةُ تحميل مؤثر الاستخراج، المقترحة في المثال السابق، للمستخدم بإنشـــاء شجرةٍ بإدخال أعداد صحيحة، بحيث يكون ترتيب الأعداد هو نفسه ترتيب عبور الشـــجرة المنشأة وفق الترتيب السابقي. يجب ألا ننسى طبعاً إعطاء القيم *nil*. نحصل بتنفيذ البرنـــامج السابق على الشجرة الثنائية في الشكل السابق على:

```
Enter the integers of the tree, -1 for NIL : 7 12 -1 -1 1 0 25 8 -1 -1 -1 -1 13 -1 6 -1 -1
12 7 8 25 0 1 13 6
7 12 1 0 25 8 13 6
12 8 25 0 6 13 1 7
7 12 nil nil 1 0 25 8 nil nil nil nil 13 nil 6 nil nil
Height : 5
Number of nodes : 8
12 7 8 25 0 1 13 6
7 12 nil nil 1 0 25 8 nil nil nil nil 13 nil 6 nil nil
Height : 5
Number of Nodes : 8
7 12 nil nil 1 0 25 8 nil nil nil nil 13 nil 6 nil nil
7 12 nil nil 1 0 25 8 nil nil nil nil 13 nil 6 nil nil
8 25 0 1 13 6
1 0 25 8 nil nil nil nil 13 nil 6 nil nil
Height : 4
Number of Nodes : 6
```

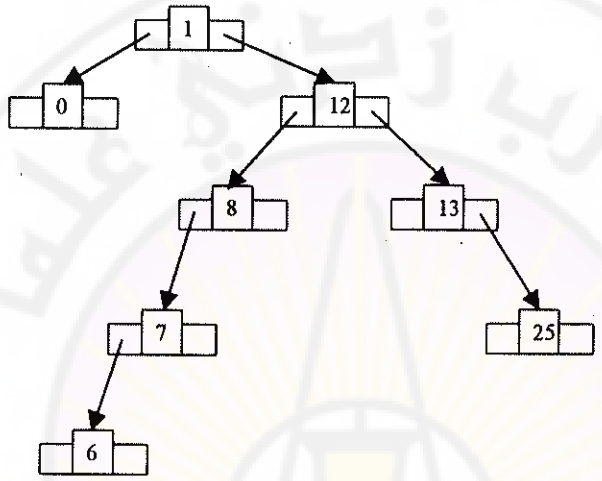
من السهل جداً تعريفُ صف شجرة *البحث Research Tree* بالوراثة. وشجرة البحـــث الثنائية هي شجرة ثنائية تحقق أية عقدة فيها *n* ما يلي:

- قيمة جميع عقد الشجرة الفرعية اليسرى (بالنسبة إلىn) أصغر من قيمة n؛
- قيمة جميع عقد الشجرة الفرعية اليمنى (بالنسبة إلى n) أكبر أو تساوي قيمة n؛
- كل شجرة فرعية هي أيضاً شجرة بحثٍ ثنائية.

بالطبع، يجب أن يُعرِّف المستخدم العلاقات *أصغر وأكبر وتساوي*.

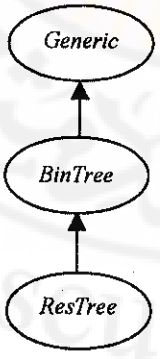
إن مفهوم شجرة البحث الثنائية هام جداً لأنه يسمح بالحصول آلياً، بفضل عبور الشجرة وفق الترتيب البيني، على تتالي قيم الشجرة وفق الترتيب التصاعدي بالنسبة إلى علاقة المقارنة المستخدمة.

مثال: لنبن شجرة بحث ثنائية مزودة بعلاقة الترتيب العادية على الأعداد الصحيحة. نقوم لذلك بإضافة الأعداد التالية بالترتيب 1، 0، 12، 13، 8، 7، 25، 6. نحصل على شجرة البحث الثنائية التالية:



الشكل 4–8: شجرة البحث الثنائية بإدخال الأعداد 1، 0، 12، 13، 8، 7، 25،6 بالترتيب.

لعبور هذه الشجرة وفق الترتيب البيني، يجب العبور بالتتالي على العقد 0، 1، 6، 7، 8، 12، 13، 25 وهذا يوافق مجموعة العقد مرتبة بالترتيب المتزايد. إن شجرة البحث الثنائية هي شجرة ثنائية خاصة. تسمح البرمجة الموجهة بالغرض بالاستفادة من هذه الملاحظة.



يكفي لذلك تعريف الصف *ResTree* بحيث يكون سليلاً للصف *BinTree*، ثم إغناؤه بالتابع الأولي *insert_val()* الذي يضيف غرضاً مُستنسخاً من النمط *Generic* إلى الشجرة مع مراعاة شروط تعريف شجرة البحث الثنائية. كذلك نضيف التابع المنطقي *find()* الذي يسمح بالتحقق من وجود قيمة ما ضمن شجرة البحث الثنائية، والطريقة *eliminate_val()* التي تهدم القيمة المحددة ضمن الشجرة في حال وجودها مع مراعاة كون الشجرة الجديدة شجرة بحث ثنائية.

```
#ifndef _RESTREE_
#define _RESTREE_

#include "BinTree.h"
#include "generic.h"
#include "boolean.h"

class ResTree : public BinTree {
public :
  bool find(Generic*);
  void insert_value(Generic*);
  void eliminate_value(Generic*);
};
#endif
```

ResTree.h

تستند هذه التوابع الثلاثة إلى التعريف الصحيح للتابعين الوهميين *operator<()* و *operator>()*. إذ يجب على مستخدم الغرض *ResTree* التحقق من أن هاتين الطريقتين معرّفتان تعريفاً صحيحاً بالنسبة إلى نمط المعطيات التي يضعها في الشجرة.

قبل إدراج قيمةٍ ما، يجري البحثُ عودياً عن المكان المناسب لإدراجها. بعد العثور على الورقة المطلوبة تُضاف القيمة، التي تصبح ورقة جديدة ضمن الشجرة. تجري إضافة هذه الورقة باستخدام آلية تمرير المحددات بالعنوان للمؤشر إلى الشجرة الفرعية التي تتضمن القيمة الجديدة.

يَستخدمُ تابعُ البحثِ عن قيمةٍ ضمنَ الشجرةِ مبدأ خوارزميةِ الإدراج نفسها، غير أنه يكتفي بإعادة جواب منطقي.

```
#include "ResTree.h"
#include "generic.h"
#include "boolean.h"

static bool search_node(BinTree::Node* n, Generic* value) {
 if (n==0)
          return false;
 else if ( *(n->val) > value)
   return search_node(n->l,value);
 else if ( *(n->val) < value)
   return search_node(n->r,value);
 else return true;
}
bool ResTree::find(Generic* value) {
 return search_node(rt, value);
}
static void insert_node(BinTree::Node*& n, Generic* value) {
 if (n==0)                              // we are on the level of a leaf, insert the node
   n = new BinTree::Node(value);
 else
   if (*(n->val) > value)
     insert_node(n->l,value);
   else
         insert_node(n->r,value);
}
void ResTree::insert_value(Generic* value) {
 insert_node(rt,value);
}
static BinTree::Node* succes(BinTree::Node* x, BinTree::Node*& prec) {
 prec = x;
 x = x->r;
 while (x->l) {
  prec = x;
  x = x->l;
 }
 return x;
}
static void eliminate_node(BinTree::Node*& n, Generic* value) {
 BinTree::Node* tmp;
 if (n==0) return;                      // node not found
 else
   if (*(n->val) > value)
         eliminate_node(n->l,value);
   else if (*(n->val) < value)
         eliminate_node(n->r,value);
   else                                 // node found, n point to it
```

```
        if (n->l == 0) {                    // leaf or empty left branch
          tmp = n;
          n = n->r;
          delete tmp;
        }
        else
        if (n->r == 0) {                    // empty right branch
          tmp = n;
          n = n->l;
          delete tmp;
        }
        else {                              // internal of the tree
          BinTree::Node* prec;
          BinTree::Node* tmp1 = succes(n,prec);
          if (prec != n)
            prec->l = tmp1->r;              // the succ. may have a str
          else
            n->r = tmp1->r;
          tmp1->l = n->l;
          tmp1->r = n->r;
          tmp = n;
          n = tmp1;
          delete tmp;
        }
}
void ResTree::eliminate_value(Generic* value) {
  eliminate_node(rt,value);
}
```

ResTree.cpp

يجب أن تقومَ خوارزميةُ حذفِ عقدةٍ من الشجرة بالتحقق من أن الشجرةَ الناتجةَ تبقى شجرةَ بحثٍ ثنائية. لذلك، بعد تحديدِ موقعِ القيمة المراد هدمها، يجري التمييز بين الحالات التالية:

- إذا كان stl للقيمة المراد هدمها خاليا، يكفي رفع الشجرة الفرعية اليمنى؛
- وإلا، إذا كان str للقيمة المراد هدمها خالياً، يكفي رفع الشجرة الفرعية اليسرى؛
- وإلا، فهي عقدةٌ داخليةٌ في الشجرة ولها stl و str . عندها، ولأن القيمة تفصلُ مجموعتين من القيم، يكفي تبديلُ القيمةِ المراد حذفها بالقيمةِ التي هي أكبر مباشرة (أو أصغر) والموجودة ضمن الشجرة، ثم حذف هذه القيمة من مكان وجودها.

المفيد في هذه الطريقة أن القيمة التي هي أكبر مباشرة (أو أصغر) هي دوماً العقدة الموجودة في أقصى اليسار ضمن الشجرة الفرعية اليمنى (أقصى اليمين ضمن الشجرة الفرعية اليسرى) للعقدة المراد حذفها. ولما كانت هذه العقدة هي "الصغرى" ("الكبرى") ضمن هذه الشجرة الفرعية، فمن خواصها أن ليس لها شجرة فرعية يسرى (يمنى). وتُحذف من ثَم ببساطة لكونها تدخل ضمن الحالة الأولى من الحالات المذكورة آنفاً.

نذكر أن حذفَ ورقة مُتَضمَّنٌ في الحالة الأولى، لأنه عند رفع الشجرة الفرعية اليمنى (اليسرى) لورقة نجدها خالية، ومن ثَم تُحذف الورقةُ من الشجرة.

فيما يلي برنامج يستخدم الصف *ResTree* ويُوضِّح قوة الوراثة. تُورَّثُ جميعُ طرائق الصف *BinTree* ويمكن للصف الجديد استخدامها دون مشاكل.

```
#include <iostream.h>
#include "ResTree.h"
#include "Integer.h"

void treat_integer(Generic* p) {
 cout << p << ' ';
}

int main() {
 // creation of a reaserch binary tree
 ResTree ar;
 Integer e;
 cout << "Enter the integers of the Tree, ^Z = stop " << flush;
 while (cin >> e) {
        Integer* pg = new Integer(e);
  if (!ar.find(pg))                    // each integer is unique
    ar.insert_value(pg);
  else
    delete pg;
 }
ar.treat_inorder(treat_integer);        // display the Tree using inorder
cout << endl;
cout << "Height: " << ar.height() << endl;
cout << "Nomber of nodes : " << ar.nb_nodes() << endl,


cin.clear();
cout << "Enter the integers to eliminate, ^Z = stop " << flush;
```

```
while (cin >> e) {
        Integer* pg = new Integer(e);
  if (ar.find(pg)) ar.eliminate_value(pg);
  delete pg;
        ar.treat_inorder(treat_integer);   // display the tree using inorder
  cout << endl;
 }
}
```

Tbin2.cpp

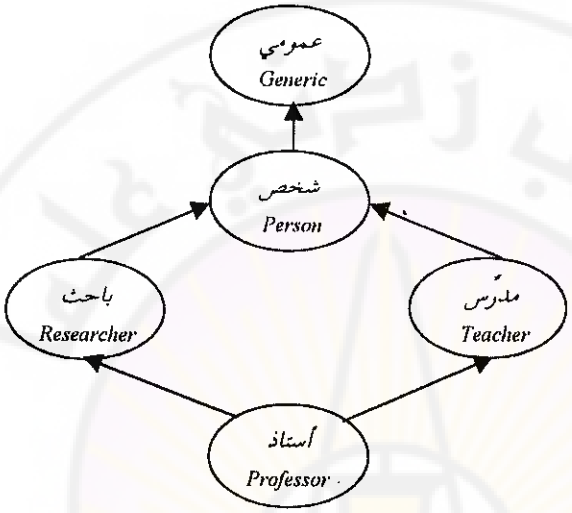
لتوضيح عمل هذا البرنامج، ندخل القيم التالية بالترتيب المبين للحصول على شجرة البحث الموجودة في الشكل السابق.

```
Enter the integers of the Tree, ^Z = stop 1 0 12 13 8 7 25 6
^Z
Height: 5
Number of nodes : 8
Enter the integers to eliminate, ^Z = stop 1
0 6 7 8 12 13 25
12
0 6 7 8 13 25
6
0 7 8 13 25
8
0 7 13 25
^Z
```

4– 5 الوراثة المتعددة multiple inheritance

إضافة إلى الوراثة البسيطة، تسمح C++ بالوراثة المتعددة، أي إنشاء صف مشتق مباشرة من عدة صفوف سالفة. من وجهة نظر نَحْوية (تركيبية) يكفي ذكر قائمةٍ بأسماء الصفوف السالفة مفصولة بفواصل ",". عند حدوث تضاربٍ في أسماء التوابع في الصف المشتق، يجب على المبرمج تفصيل الاسم باستخدام مؤثر إبطال المدى scope resolution. يتمثل الحل الآخر لهذه المسألة في زيادة تحميل التابع المدروس ضمن الصف المشتق.

تطرح الوراثة المتعددة مسألة معالجة الوراثة المتكررة المذكورة في بداية هذا الفصل. نبين هذه المشكلة بالاستعانة بمثال:



الشكل 4–9: مثال على الوراثة المتكررة للصفوف

يظهر المخطط السابق هرمية وراثة المثال الذي سنفصّله. نقترح هنا تحقيق الصف *Professor* انطلاقاً من الصفين *Researcher* و *Teacher*. لدينا هنا إذن وراثةٌ متعددةٌ. للزيادة في تعقيد المسألة يُشتق الصفان *Researcher* و *Teacher* من الصف *Person*.

```
#ifndef _PERSON_
#define _PERSON_

#include <iostream.h>
#include "boolean.h"
#include "generic.h"
#include "string7.h"

class Person : public Generic {
public :
  Person(String="");
  Person(const Person&);
  virtual ~Person();
  virtual void display(ostream&) const;
```

```
  virtual Person* clone();
  String name() const;
protected :
  String namep;
  virtual bool operator==(const Generic* g){ return false; };
  virtual bool operator>(const Generic* g) { return false; };
  virtual bool operator<(const Generic* g) { return false; };
};
#endif
```

Pers.h

لا يسبب اشتقاق *Person* من *Generic* أية مشكلة. يتألف *Person* من عنصر معطيـــات عضو وحيد: namep.

```
#include <iostream.h>
#include "pers.h"

Person::Person(String n) : namep(n) {
}

Person::Person(const Person& p) : namep(p.namep) {
}

Person::~Person() {
}

void Person::display(ostream& os) const {
  os << namep << endl;
}

Person* Person::clone() {
  return new Person(*this);
}

String Person::name() const {
  return namep;
}
```

pers.cpp

فيما يلي تعريف الصفين *Researcher* و *Teacher* :

```
#ifndef _RESEARCH1_
#define _RESEARCH1_

#include <iostream.h>
#include "string7.h"
```

```
#include "pers.h"

class Researcher : public Person {
public :
  Researcher(String n="",String tr="");
  Researcher(const Researcher&);
  virtual ~Researcher();
  virtual void display(ostream&) const;
  virtual Researcher* clone();
  String domain() const;
protected :
  String domain_research;
};
#endif
```

research1.h

الباحث "هو" شخص له مجال بحث معين.

```
#include <iostream.h>
#include "research1.h"

Researcher::Researcher(String n,String tr) : Person(n),domain_research(tr) {
}

Researcher::Researcher(const Researcher& cc)
                    : Person(cc.namep),domain_research(cc.domain_research) {
}

Researcher::~Researcher() {
}

void Researcher::display(ostream& os) const {
  Person::display(os);
  os << "research domain : " << domain() << endl;
}

Researcher* Researcher::clone() {
  return new Researcher(*this);
}

String Researcher::domain() const {
  return domain_research;
}
```

research1.cpp

نحصل بنفس الطريقة من أجل الصف *Teacher* على:

```
#ifndef _TEACH_
#define _TEACH_

#include <iostream.h>
#include "string7.h"
#include "pers.h"

class Teacher : public Person {
public :
  Teacher(String n="",String tc="");
  Teacher(const Teacher&);
  virtual ~Teacher();
  virtual void display(ostream&) const;
  virtual Teacher* clone();
  String course() const;
protected :
  String course_title;
};
#endif
```

teach1.h

نفترض لتبسيط المثال، أن المدرّس يُدرّس مادةً واحدةً اسمها محـــددٌ بعنصـــر المعطيـــات *course_title*. جرى تحقيق ذلك كما يلي:

```
#include <iostream.h>
#include "teach1.h"

Teacher::Teacher(String n,String tc)
        : Person(n),course_title(tc) {
}

Teacher::Teacher(const Teacher& cc)
                    : Person(cc.namep),course_title(cc.course_title) {
}

Teacher::~Teacher() {
}

void Teacher::display(ostream& os) const {
  Person::display(os);
  os << "Teached course : " << course() << endl;
}

Teacher* Teacher::clone() {
  return new Teacher(*this);
```

```
}

String Teacher::course() const {
  return course_title;
}
```

teach1.cpp

تستدعي البواني باني الصف السلف صراحة، كما تستدعي الطريقة الوهمية *display()* صراحةً التابع *display()* من الصف السلف *Person*.

لنعد الآن إلى الصف *Professor*:

```
#ifndef _PROF_
#define _PROF_

#include <iostream.h>
#include "string7.h"
#include "teach1.h"
#include "research1.h"

class Professor : public Teacher, public Researcher {
public :
  Professor(String n="",String tc="",String tr="",String tch="");
  Professor(const Professor&);
  virtual ~Professor();
  virtual void display(ostream&) const;
  virtual Professor* clone();
  String chair() const;
protected :
  String chair_title;
};
#endif
```

prof1.h

في هذا التعريف يملك الأستاذ لقباً (كرسياً) *chair*. يعني اشتقاق *Professor* من *Teacher* و *Researcher* أنه يدرّس مادةً وأن له مجال بحث معين.

عند تحقيق هذا الصف يستدعي الباني الافتراضي وباني النسخ البانيين الموافقين في الصفين السالفين. ليس لترتيب الاستدعاء أهمية.

```
#include <iostream.h>
#include "prof1.h"
```

```
Professor::Professor(String n,String tc,String tr,String tch)
                        : Teacher(n,tc),Researcher(n,tr),chair_title(tch) {
}

Professor::Professor(const Professor& p)
                        : Teacher(p.Teacher::namep,p.course_title),
                                Researcher(p.Teacher::namep,p.domain_research),
                                chair_title(p.chair_title) {
}

Professor::~Professor() {
}

void Professor::display(ostream& os) const {
  os << "Professor : " << Teacher::name() << endl;
//  os << "Professor : " << name() << endl;
  os << "Chair of : " << chair() << endl;
  os << "Research domain : " << domain() << endl;
  os << "Teached course : " << course() << endl;
}

Professor* Professor::clone() {
  return new Professor(*this);
}

String Professor::chair() const {
  return chair_title;
}
```

prof1.cpp

يسبب التابع *display()* مشكلةً في استخدام النافذ *name()* في الصف *Person*. فعند محاولة برمجة السطر الذي يُظهر اسم الأستاذ:

```
os  << " Professor : " << name() << endl;
```

يرد المترجم برسالة خطأ من النمط "Ambiguous Number". بتحديد التابع العضو *name()* بإضافة اسم الصف *Teacher* أو *Researcher* تختفي هذه الرسالة! لماذا تظهر رسالة الخطأ هذه؟ تحدث مفاجأة مماثلة في البرنامج الذي يستخدم هذه الصفوف المختلفة:

```
#include <iostream.h>
#include "pers.h"
#include "teach1.h"
#include "research1.h"
#include "prof1.h"
```

```
int main() {
 Person* p = new Person("Nino");
 p->display(cout);
 cout << endl;
 Teacher* cc = new Teacher("Pierre", "Unix");
 cc->display(cout);
 cout << endl;
 Researcher* ch = new Researcher("Raymond","Cognitive Sciences");
 ch->display(cout);
 cout << endl;
 Professor* pr = new Professor("Jean-Claude","UML",
                  "Information Systems","Analyses and Conception");
 pr->display(cout);
 cout << endl;
 // Generic* pg = pr->clone();                    // Error : create a copy of pr
 Generic* pg = dynamic_cast<Teacher*>(pr->clone());
 pg->display(cout);
 delete p;
 delete cc;
 delete ch;
 delete pr;
 delete pg;
}
```

توجد هنا مشكلة في إنشاء نسخة من المتحول pr، يكمن حلها في إجراء تحويل صريح للنمط إلى *Teacher* أو *Researcher*.

يعود السبب إلى الطريقة التي اشتققنا بها على التتالي الصفين *Researcher* و *Teacher*. حيث يرث *Professor* مرتين من *Person*، ومن ثَم فإن الصف *Professor* يتضمن فعلياً عنصري معطيات *namep*. فللبنية الداخلية لغرض من النمط *Professor*، الشكل التالي:

namep
Research domain
namep
course title
chair title

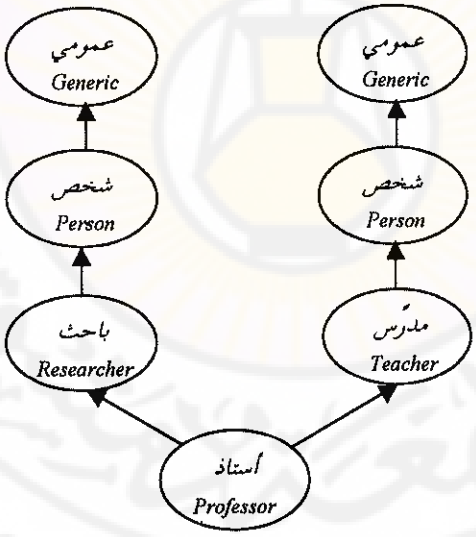
الشكل 4–10: البنية الداخلية لغرض من النمط *Professor*

نلاحظ أن عنصر المعطيات *namep* يظهر مرتين! كذلك الأمر في حالة التوابع الأعضاء الأخرى.

يولّدُ نظامُ C++ في حالة الوراثةِ المتكررةِ، ضمنَ الصفِ المشتقِ، عدداً من تكررات الصفوف المشتركة مساوياً لعدد الصفوف السلف المشتقة من الصف المتكرر.

ففي المثال السابق، لا يكون لدينا مخطط الوراثة التي نتصوره، بل مخطط الوراثة المبين في الشكل 4-11.

هذا يفسر كون النافذ *name()* مبهماً وذلك لوجود اثنين منه. على المبرمج إذن أن يُحدِّدَ الاسم *name* المقصود فعلياً: أهو اسم الباحث أم اسم المدرّس؟ ينطبق الأمر نفسه على التسليع *clone()*. لا يستطيع المبرمج قسر النمط مباشرة على *Generic* لأن ذلك يعتمد على فرع الوراثة الذي سيجري اختياره.



الشكل 4-11: هرمية الصفوف المنشأة في المثال السابق.

لمنع تطبيق هذا السلوك الافتراضي، غير الملائم في أغلب الحالات، يجب على المبرمج

التدخل في كيفية الوراثة عند تعريف الصفوف. ويمكنه ذلك باستخدام الكلمة المحجوزة virtual عند الاشتقاق.

عند اشتقاق صفٍ انطلاقاً من صفوف لها أسلافٌ مشتركةٌ وهميةٌ *virtual* ، لن يكون هناك إلا مستنسخ وحيد للسلف المشترك في الصف المشتق حديثاً. ومن ثَم سيكون من غير الضروري في هذه الحالة أن نحدد أعضاء الصف المشترك. حيث يأخذ الصف السلف الوهمي (هنا *Person*) قيماً بدائية دائماً من الصف الأعمق في شجرة الاشتقاق (*Professor*). إذا لم يستدع الأخير صراحة باني الصف السلف المشترك (*Person*) فيُستدعَى الباني الافتراضي لهذا الصف. في مثالنا، لن يأخذ الصف الوهمي السلف المشترك (*Person*) أية قيمة عن طريق باني الصفوف المشتقة مباشرةً (*Teacher* و *Research*). وهكذا إذا عَرَّف المبرمج صفاً في نهاية هرمية صفوف تتضمن صفوفاً مشتقة بصورة وهمية، فستبنى هذه الصفوف دائماً قبل الصفوف السالفة غير الوهمية، وذلك من اليسار إلى اليمين وفق ترتيب ظهورها في التصريح عن الصف المشتق. وإذا لم تُحدَّد في تعريف باني الصف المشتق قيمةٌ بدائية للصف الوهمي السلف، فيجب أن يكون للأخير بانٍ افتراضي أو ألا يكون له بانٍ أبداً. يسمحُ هذا للنظام بتوليد البناء الصحيح للصف الوهمي السلف آلياً.

```
#include <iostream.h>
#include "Research.h"

Researcher::Researcher(String n,String tr) : Person(n),domain_research(tr) {
}

Researcher::Researcher(const Researcher& cc)
                        : Person(cc.namep),domain_research(cc.domain_research) {
}

Researcher::~Researcher() {
}

void Researcher::display(ostream& os) const {
  Person::display(os);
  os << "Reseach domain : " << domain() << endl;
}
```

```
Researcher* Researcher::clone() {
  return new Researcher(*this);
}

String Researcher::domain() const {
  return domain_research;
}
```

research.h

لا يهم الترتيب في كتابة *virtual public* أو *public virtual*. ولأن الصف *Teacher* مُشتقٌ أيضاً بصورة وهمية من الصف *Person*، يمكننا كتابة:

```
#include <iostream.h>
#include "Teach.h"

Teacher::Teacher(String n,String tc)
                                        : Person(n),course_title(tc) {
}

Teacher::Teacher(const Teacher& cc)
                                : Person(cc.namep),course_title(cc.course_title) {
}

Teacher::~Teacher() {
}

void Teacher::display(ostream& os) const {
  Person::display(os);
  os << "Teached course : " << course() << endl;
}

Teacher* Teacher::clone() {
  return new Teacher(*this);
}

String Teacher::course() const {
  return course_title;
}
```

teach.h

ليس هناك تعديل آخر للنسخة السابقة، فتبقى تعاريف التوابع الأعضاء في الصفوف *Person* و *Teacher* و *Course* دون تعديل. أما الصف *Professor* فتجري عليه بعض التعديلات.

```
#ifndef _PROF_
#define _PROF_

#include <iostream.h>
#include "string7.h"
#include "teach.h"
#include "research.h"

class Professor : public Teacher, public Researcher {
public :
  Professor(String n="",String tc="",String tr="",String tch="");
  Professor(const Professor&);
  virtual ~Professor();
  virtual void display(ostream&) const;
  virtual Professor* clone();
  String chair() const;
protected :
  String chair_title;
};
#endif
```

prof.h

ما يزال الصف *Professor* مشتقاً بصورة عامة من *Teacher* و *Researcher*. ولكن لما كان لا يوجد إلا مستنسخ وحيد من الصف المشترك السلف *Person*، بُنيت الأغراض من النمط *Prefessor* بطريقة مختلفة. وكما رأينا آنفاً، تقعُ على الصف *Professor*، باعتباره الصف الأكثر اشتقاقاً، مسؤولية وضع قيم بدائية لجزء الغرض *Person* الموروث بصورة وهمية. يتحقق ذلك باستدعاء *Person(n)* ضمن تعليمات وضع قيم بدائية (استبداء) للباني.

```
#include <iostream.h>
#include "prof.h"

Professor::Professor(String n,String tc,String tr,String tch) :
                Person(n),                      // must figure
                Teacher(n,tc),
                Researcher(n,tr),
                chair_title(tch) {
}

Professor::Professor(const Professor& p) :
                Person(p.namep),                // must figure
                Teacher(p.namep,p.course_title),
                Researcher(p.namep,p.domain_research),
                chair_title(p.chair_title) {
```

```
}

Professor::~Professor() {
}

void Professor::display(ostream& os) const {
  os << "Professor : " << name() << endl;
  os << "Chair of : " << chair() << endl;
  os << "Research domain : " << domain() << endl;
  os << "Teached course : " << course() << endl;
}

Professor* Professor::clone() {
  return new Professor(*this);
}

String Professor::chair() const {
  return chair_title;
}
```

prof.cpp

ولأن الصف الأكثر اشتقاقاً من الصف السلف المشترك هو الذي يضع قيماً بدائية لـــه، يستدعي باني الصف *Professor* باني الصف *Person*. وإذا لم يفعل ذلك، فسيُستدعى البـاني الافتراضي للصف *Person*! يبين التابع العضو *display()* أن التحديد الصريح لمعيّـــن هويـــة identifier السلف المشترك وقسر النمط casting ليسا ضروريين لأنه لا يوجد ســـوى ورود وحيد للسلف المشترك. يتصرف الصف كما في المخطط الأول لهرمية الوراثة. نحصل عنـــد التنفيذ في الحالتين على:

```
Nino

Pierre
Teached course : Unix

Raymond
Reseach domain : Cognitive Sciences

Professor : Jean-Claude
Chair of : Analyses and Conception
Research domain : Information Systems
Teached course : UML

Professor : Jean-Claude
```

Chair of : Analyses and Conception
Research domain : Information Systems
Teached course : UML

إن استخدام الوراثة المتعددة ليس سهلاً في C++ ويتطلّب من المبرمج التروّي قبل البدء بها. على المبرمج عند استخدامه للوراثة المتعددة، اشتقاق الصفوف بصورة *virtual* من الصفوف السالفة، وذلك لأن التسيير الافتراضي لـ C++ في حالة الوراثة المتكررة غير بدهي. وفي جميع الأحوال، الوراثة البسيطة قادرة، في أغلب الأحيان، على حل كثير من المشاكل بأناقة.

4– 6 إدارة الاستثناءات Exception management

تعتبر كتابة برامج متينة، قادرة على الاستجابة لجميع الأوضاع المحتملة، من أكبر الصعوبات في مهنة المبرمج. يسمح مفهوم الاستثناءات في C++ بالقيام بإدارة جيدة للأخطاء التي تحدث ضمن تطبيق ما. حتى الآن، لم يكن لدى المطوّر إلا التقنيات التقليدية المستنبطة من لغات البرمجة التقليدية. نذكر من هذه التقنيات:

- التوابع التي تعيد حالة خطأ، إما ضمن قيمة التابع، أو عن طريق محدداتها؛
- استخدام متحولات عامة للخطأ؛
- استخدام توابع أوّلية مثل setjmp و longjmp من لغة C؛
- التوكيدات assertions.

لكل من هذه التقنيات مساوئها، وخاصة في سياق لغة C++. فلا يمكن تطبيق التقنية الأولى في حالة البواني الافتراضية والهوادم والتوابع المؤثراتية operator functions. للتقنية الثانية مساوئها المتعلقة باستخدام المتحولات العامة. من جهة أخرى يجب عدم استخدام التوابع الأولية *setjmp* و *longjmp* في C++ لأنها لا تتعامل مع مسألة هدم الأغراض عند الخروج من كتلة ما. في النهاية، تعتبر التوكيدات وسيلةً لضبط البرامج أكثر منها أداةً لإجراء معالجة الأخطاء.

إن الفكرة الأساسية لإدارة الأخطاء في C++ هو أنه عندما يواجه تابعٌ ما مشكلةً لا يعلم كيف يحلها، يقوم بإرسال (throw) استثناء exception آملاً أن يتمكن أحد التوابع في شجرة الاستدعاء من التقاطه (catch). لنَرَ على مثال بسيط لصف الأعداد العادية *Rat* (المقطع 1.3.4) كيفية تنفيذ معالجة الأخطاء.

عند تنفيذ الصف Rat سابقاً، لم نأخذ بعين الاعتبار الحالة التي ينشئُ المبرمج فيها عدداً عادياً مقامه صفر. سيكون ذلك متوقعاً في هذه النسخة، وعند حدوثه نقوم بإرسال استثناء. الاستثناء هو غرض من نمط معين (أيةُ بنيةٍ يمكنها نقل المعلومات). سننشئ في مثالنا الأول صفاً خاصاً بالأخطاء نسميه *NullDenominator*.

```
#include <iostream.h>

class NullDenominator {}; // exception class

class Rat {
  friend ostream& operator<<(ostream&, const Rat&);
  friend Rat operator+(const Rat&, const Rat&);
  friend Rat operator-(const Rat&, const Rat&);
public :
  Rat(int = 0, int = 1);
  Rat(const Rat&);
  int numerator() const;
  int denominator() const;
protected :
  int num, denom;
};
```

rat1.h

يَفحصُ البانِي البسطَ الذي يُمرَّر له كمحدِّد. فإذا كان صفراً يرسلُ استثناءً من نمط *NullDenominator*، وينقل بذلك معلومة هامة لمن يرغب في التقاطها.

```
#include <iostream.h>
#include <stdlib.h>
#include "rat1.h"

Rat::Rat(int n, int d) : num(n), denom(d) {
  if (denom == 0) throw NullDenominator();
  else
  if (denom < 0) {
```

```
    denom = abs(denom);
    num *= -1;
  }
}
Rat::Rat(const Rat& r) : num(r.num), denom(r.denom) {
}
int Rat::numerator()   const { return num; }
int Rat::denominator() const { return denom; }

ostream& operator<<(ostream& os, const Rat& r) {
  if (r.num < 0) os << '(';
  os << r.num << '/' << r.denom;
  if (r.num < 0) os << ')';
  return os;
}
Rat operator+(const Rat& r1, const Rat& r2) {
  Rat sum(r1.num * r2.denom + r1.denom * r2.num,r1.denom * r2.denom);
  return sum;
}
Rat operator-(const Rat& r1, const Rat& r2) {
  Rat sum(r1.num * r2.denom - r1.denom * r2.num,r1.denom * r2.denom);
  return sum;
}
```

rat1.cpp

لنرَ الآن كيفية التقاط ومعالجة الاستثناءات. تقضي اللغة أن بإمكان المبرمج أن يُعـــرِّفَ مناطقَ تُلتقط فيها الاستثناءات. تبدأ هذه المناطق بالكلمة المحجوزة *try* وتسمح بتعريف كتلةٍ من التعليمات. يجب أن يتبعَ هذه الكتلةَ مباشرةً برنـــامجُ إدارة الاســـتثناءات exceptions manager، الذي يجب أن يبدأ بالكلمة المحجوزة *catch* يتبعها بين قوسين نمـــط الاســـتثناء. يُعرِّف برنامجُ الإدارةِ هذا كتلةً من التعليمات التي تُنفَّذُ إذا كان غرضُ الاستثناءِ الذي تلقّـاه من النمطِ المحددِ بترويسته. يمكن أن يتبعَ برنامجَ الإدارةِ هذا مباشرةً عددٌ غير محددٍ من برامج الإدارة (catch).

```
#include <iostream.h>
#include "rat1.h"

int main() {
  try {
        Rat a,b(-1,-3);
        cout << a << '\t' << b << '\t' << endl;
        Rat c(5,0);
```

```
        cout << (a+b) << endl;          // we never reach here !
 }
 catch (NullDenominator) {
        cerr << "Intercepted Exception: Denominator 0" << endl;
 }
 return 0;
}
```

Trat1.cpp

يودي إرسالُ استثناءٍ إلى تحويل التحكم، بصورة مطلقة، إلى أول برنامج إدارة استثناءات مُحيط. لذلك ففي مثالنا لا يُنفَّذ الإظهارُ الثاني ضمن كتلة try أبداً. يُظهر هذا البرنامج عند التنفيذ:

```
0/1     1/3
Intercepted Exception: Denominator 0
```

بعدَ أن ينهي برنامجُ إدارة الاستثناءات تنفيذَ كتلةِ التعليمات الموافقة، يستمرُ التنفيذُ ابتداءً بأول تعليمة موجودة بعد آخر برنامج إدارة ضمن الكتلة try المعنية.

قد يتساءل القارئُ ما الذي يحدث إذا أُرسِل استثناءٌ خارجَ كتلةِ try، كما في البرنامج التالي؟

```
#include <iostream.h>
#include "rat1.h"

int main() {
 Rat a,b(-1,-3);
 cout << a << '\t' << b << '\t' << endl;
 Rat c(5,0);
 cout << (a+b) << endl;                    // we never reach here !
 return 0;
}
```

السلوك الافتراضي بسيطٌ وفج: إذا أُرسلَ استثناءٌ و لم يُلتقط، يتوقفُ تنفيذُ البرنامج. في هذه الحالة يُستدعى التابع *terminate()* الذي يَستدعي بدوره التابعَ *abort()*، وهي أكثر الطرق جذرية لإنهاءِ تنفيذِ برنامج. يمكن للمبرمج أن يُعدِّل هذا الوضع بإضافةِ نسختِه الخاصة من التابع *terminate()* بمساعدة التابع *set_terminate()*.

```
#include <iostream.h>
#include <stdlib.h>
#include <except.h>          // set_terminate()
```

```
#include "rat1.h"

void my_function_exception_not_treated() {
  cout << "An exception has been started" << endl;
  cout << "It has not been intercepted, so the porogram will stop";
  exit(EXIT_FAILURE);
}

int main() {
  set_terminate(my_function_exception_not_treated);
  Rat a,b(-1,-3);
  cout << a << '\t' << b << '\t' << endl;
  Rat c(5,0);
  cout << (a+b) << endl;              // we never reach here !
}
```

يجب ألا يكون للتابع الذي يُمرَرُ عنوانه إلى *set_terminate* محـــددات وأن يعيـــد *void*، لنعدِّل مرَّة أخرى الصف *Rat* بإضافةِ موثر القسمة. يجب علينا هذه المرة الانتباه لالتقاط أيـــة قسمة على صفر.

```
#include <iostream.h>

class NullDenominator {};  // exception class
class DivisionByZero {};

class Rat {
  friend ostream& operator<<(ostream&, const Rat&);
  friend Rat operator+(const Rat&, const Rat&);
  friend Rat operator-(const Rat&, const Rat&);
  friend Rat operator/(const Rat&, const Rat&);
public :
  Rat(int = 0, int = 1);
  Rat(const Rat&);
  int numerator() const;
  int denominator() const;
protected :
  int num, denom;
};
```

rat2.h

نُعرِّف صفاً آخر للأخطاء وهو *DivisionByZero*. تجدر الملاحظة أنه ليس من الضـــروري أن يكون الاستثناءِ من النمط "صف"، بل يمكن أن يكون من أي نمط آخر سواء كان معرَّفاً سابقاً أم لا (enum, struct, ...).

```
#include <iostream.h>
#include <stdlib.h>
#include "rat2.h"

Rat::Rat(int n, int d) : num(n), denom(d) {
  if (denom == 0) throw NullDenominator();
  else
  if (denom < 0) {
     denom = abs(denom);
     num *= -1;
  }
}
Rat::Rat(const Rat& r) : num(r.num), denom(r.denom) {
}
int Rat::numerator()   const { return num; }
int Rat::denominator() const { return denom; }

ostream& operator<<(ostream& os, const Rat& r) {
  if (r.num < 0) os << '(';
  os << r.num << '/' << r.denom;
  if (r.num < 0) os << ')';
  return os;
}
Rat operator+(const Rat& r1, const Rat& r2) {
  return Rat(r1.num * r2.denom + r1.denom * r2.num,r1.denom * r2.denom);
}
Rat operator-(const Rat& r1, const Rat& r2) {
  return Rat(r1.num * r2.denom - r1.denom * r2.num,r1.denom * r2.denom);
}
Rat operator/(const Rat& r1, const Rat& r2) {
  if (r2.num == 0) throw DivisionByZero();
  return Rat(r1.num * r2.denom,r1.denom * r2.num);
}
```

rat2.cpp

فيما يلي برنامجٌ يستخدمُ هذه النسخة الجديدة من الصف *Rat* وفيه تعليمتان يمكنهما إرسال استثناء.

```
#include <iostream.h>
#include <stdlib.h>
#include "Rat2.h"

int main() {
  try {
        Rat a,b(1,3);
```

```
        cout << a << '\t' << b << '\t' << endl;
        Rat d = b-b;
        cout << (a/b) << d << (b/d) << endl;   // Exception : DivisionByZero
        Rat c(5,0);                            // Exception : NullDenominator
        cout << (a+b) << endl;                 // we never reach this point !
}
catch (NullDenominator) {
        cerr << "Intercepted exception : Denominator 0" << endl;
}
catch (DivisionByZero) {
        cerr << "Intercepted exception : Division by 0" << endl;
}
catch (...) {
        cerr << "Unexpected Exception has been intercepted" << endl;
        exit(EXIT_FAILURE);
}
cout << "We continue here, except in the case of an unexpected exception";
return 0;
}
```

إن برنامجَ إدارةِ الاستثناءات الافتراضي، والذي يُكتَبُ توصيفُ نمطه على الشكل "..." خاصٌ من حيث أنه يقبل جميع الاستثناءات، مهما كان نمطها. ولما كانت برامج الإدارة تُفحصُ على التتالي، فمن الضروري أن يكون هذا البرنامجُ آخر القائمة، فهو بصورة ما بديلٌ أنيقٌ للتابع *terminate()*. يُظهِر هذا البرنامجُ أيضاً أن التنفيذَ يستمرُ فعلاً من أولِ تعليمةٍ تلي آخرَ برنامجِ إدارةٍ ضمن الكتلة *try*. فيما يلي خرج تنفيذ البرنامج:

```
0/1     1/3
Intercepted exception : Division by 0
We continue here, except in the case of an unexpected exception
```

لكنَّ نظامَ إدارة الاستثناءات، كما هو منفَّذٌ فعلاً في لغة C++، أكثرُ تعقيداً مما تظهره هذه الأمثلة. حيث تُهدم، عند إرسال استثناءٍ، جميعُ الأغراض المعرَّفة محلياً في الكتلة *try* باستدعاء هوادمها الموافقة. إذا بدأ استثناءٌ أثناءَ بناءِ غرضٍ (أي ضمن بانيه)، تُستدعى آلياً هوادم الأغراض الفرعية، المنفّذة كلُّها فقط، والتي يمكن أن تُكوِّنَ لاحقاً الغرضَ الذي في طور البناء. يَضمَنُ ذلك هدمَ جميع المتحولات (المحلية) عند حدوث مشكلة (مثلاً: الهادم الذي يغلق ملفاً متعلقاً بالغرض، الخ.).

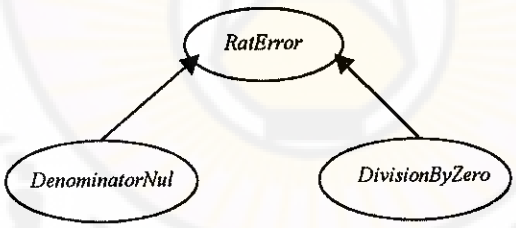
صفوفُ الاستثناءات هي صفوف C++ عاديةٌ جداً. من الممكن إذن تعريفُ هرميةٍ لصفوف الاستثناءات. ولكن يجب الانتباه هنا إلى توصيف النمط في برامج إدارة الاستثناءات. حيث يقبلُ برنامجُ إدارةِ الاستثناءاتِ من النمط *T* أو *const T* أو *T&* أو *const T&* استثناءً من النمط *E* إذا كان:

- T و E من نفس النمط
- T صف سلف عام لـ E
- T مؤشر و E مؤشر يمكن تحويله إلى T بإجراء تحويل مِقيَس للمؤشرات (لا تحويل أنماطٍ خاصٍ بصفٍ باستخدامِ بانٍ أو تابعِ تحويلٍ عضو).

عند إرسال استثناءٍ يُستدعى باني النسخ لإنشاءِ غرضٍ بالقيمةِ التي جرى إرسالها.

أيضاً، برنامج إدارة الاستثناءات ليس مجبراً على معالجةِ استثناء معالجة كاملة، بل يمكنه إجراء جزءٍ من العمل وتمريرُ الغرض (الاستثناء) إلى برنامجِ إدارة أعم منه (محيطي).

نعرّف في النسخة التالية من الصف *Rat* هرمية صفوف للاستثناءات.



الشكل 4–12: هرمية صفوف الأخطاء التي استُخدِمَت في التحقيق المُبسَّط للأعداد العادية

تتضمن أغراض هذه الصفوف عنصر معطيات يمثل العدد العادي الذي حرض الاستثناء. وتُزوَّد هذه الصفوف أيضاً ببواني نسخٍ لأنها مستدعاة آلياً من قِبَل النظام عند إرسال استثناء.

```
#include <iostream.h>

class Rat {
  friend ostream& operator<<(ostream&, const Rat&);
  friend Rat operator+(const Rat&, const Rat&);
  friend Rat operator-(const Rat&, const Rat&);
```

```
  friend Rat operator/(const Rat&, const Rat&);
public :
  Rat(int = 0, int = 1);
  Rat(const Rat&);
  ~Rat();
  int numerator() const;
  int denominator() const;
protected :
  int num, denom;
};

class ErrorRat {                                    // exception classes
public :
  Rat r;
  ErrorRat(Rat i) : r(i) {}
};
class NullDenominator : public ErrorRat {
public :
  NullDenominator(Rat i) : ErrorRat(i) {}           // constructor
  NullDenominator(const NullDenominator& d) : ErrorRat(d) {}   // copy constructor
};
class DivisionByZero : public ErrorRat {
public :
  DivisionByZero(Rat i) : ErrorRat(i) {}            // constructor
  DivisionByZero(const DivisionByZero& d) : ErrorRat(d) {}      // copy constructor
};
```

Rat3.h

يمكن تحقيق التوابع الأعضاء المختلفة ببساطة.

```
#include <iostream.h>
#include <stdlib.h>
#include "rat3.h"

Rat::Rat(int n, int d) : num(n), denom(d) {
//  cout << "Default " << num << "/" << denom << " is constructed" << endl;
  if (denom == 0) throw NullDenominator(*this);
  else
  if (denom < 0) {
          denom = abs(denom);
          num *= -1;
  }
}
Rat::Rat(const Rat& r) : num(r.num), denom(r.denom) {
//  cout << "Copie " << num << "/" << denom << " is constructed" << endl;
}
```

```
Rat::~Rat() {
// cout << num << "/" << denom << " is destructed" << endl;
}
int Rat::numerator()   const { return num; }
int Rat::denominator() const { return denom; }

ostream& operator<<(ostream& os, const Rat& r) {
 if (r.num < 0) os << '(';
 os << r.num << '/' << r.denom;
 if (r.num < 0) os << ')';
 return os;
}
Rat operator+(const Rat& r1, const Rat& r2) {
 return Rat(r1.num * r2.denom + r1.denom * r2.num,r1.denom * r2.denom);
}
Rat operator-(const Rat& r1, const Rat& r2) {
 return Rat(r1.num * r2.denom - r1.denom * r2.num,r1.denom * r2.denom);
}
Rat operator/(const Rat& r1, const Rat& r2) {
 if (r2.num == 0) throw DivisionByZero(r2);
 return Rat(r1.num * r2.denom,r1.denom * r2.num);
}
```

Rat3.cpp

في هذا التحقيق الجديد، يُنقَلُ العددُ العادي المسببُ للاستثناء إلى غرضِ الاستثناء، ليتمكن برنامجُ إدارةِ الأخطاء من معالجته بطريقته الخاصة (بإظهاره مثلاً).

```
#include <iostream.h>
#include <stdlib.h>
#include "rat3.h"
void f(const Rat&) throw(DivisionByZero);
void f(const Rat& r) throw(DivisionByZero) {
 try {
 Rat r1(5,7);                    // local objects in f()
 Rat r2 = r-r;                   // equal 0/9
 cout << (r1/r2) << endl;        // provoke exception
 }
 catch(DivisionByZero) {
 cout << "Division by 0, we do nothing ..." << endl;
   throw;
 }
}
int main() {
 try {
 Rat a(1,3);
```

```
    f(a);
cout << a << endl;                  // we never reach this point !
Rat c(5,0);                         // Exception : NullDenominator
 }
 catch (NullDenominator d) {                // copy constructor
cerr << "Intercepted exception : Denominator 0 in " << d.r << endl;
 }
 catch (DivisionByZero d) {
cerr << "Intercepted exception : Division by 0 provoked by " << d.r << endl;
 }
 catch (ErrorRat) {
cerr << "Exception provoked by a Rat " << endl;
 }
 catch (...) {
cerr << "Unexpected exception has been intercepted" << endl;
exit(EXIT_FAILURE);
 }
 cout << "We continue here, except in the case of an unexpected exception";
 return 0;
}
```

يوضح هذا المثال تداخل الكتلِ *try* عند التنفيذ. يُستدعى التابع f() ضمن الكتلة *try* في التابع *main()*. لكن لهذا التابع كتلة *try* الخاصة به والتي أُرسل فيها استثناءٌ فعلاً مِن جرّاء محاولة القسمة على صفر. يُؤدي هذا الإرسال إلى إنشاءِ غرضٍ من نمط *DivisionByZero* قيمته البدائية العدد "0/9".

يُلتقط هذا الاستثناء من قبل برنامج إدارة الاستثناءات الخاص بالتابع *f()*. يقوم هذا البرنامج بتعرُّف هذا الغرض فقط ويُمرِّره إلى برنامج الإدارة على المستوى الأعلى، وذلك بإرسال الاستثناء الذي تلقاه (تعليمة *throw*). تُهدَم، أثناء ذلك، الأغراض المحلية(*r1* و *r2*) في *f()*. إن الكتلة *try* التي تغلّف مباشرة الكتلة التي أرسل منها الاستثناء هي تلك الخاصة بالتابع *main()*. تُرتَّب برامج الإدارة ضمن هذه الكتلة ترتيباً منطقياً، ابتداءً من صفوف الخطأ المشتقة إلى صفوف الخطأ السلف، يتبعها برنامج إدارة الاستثناءات الافتراضي. نحصل عند التنفيذ على:

```
Division by 0, we do nothing...
Intercepted exception : Division by 0 provoked by 0/9
We continue here, except in the case of an unexpected exception
```

يتوفر إمكان آخر في C++ فيما يتعلق بالاستثناءات، ويتمثل في توصيف الاستثناءات التي يمكن لتابع أن يرسلها، إذ يمكن للمبرمج، عند التصريح عن التابع، بيان الاستثناءات التي يمكن أن يرسلها التابع أثناء تنفيذه. يكفي لذلك أن تُتبِعَ لائحة المحددات الصورية بتوصيف للاستثناءات مُمثل بالكلمة المحجوزة *throw* يتبعها لائحةُ أنماط بين قوسين. يمكن أن تكون هذه اللائحة خالية، وهذا يعني أن التابع لا يستطيع إرسال أي استثناء. إذا لم يكن هذا التوصيف موجوداً يمكن للتابع إرسال استثناءات من أي نمط. يمكن للتابع الذي يُسمح له بإرسال استثناءٍ من النمط X إرسال أي استثناءٍ من نمطٍ مشتقٍ من النمط X. يجب الانتباه هنا إلى أن هذا التوصيف لا يشكّل جزءاً من توقيع التابع. إذا حدث وأرسلَ تابعٌ استثناء آخر مختلفاً عن الاستثناءات التي صرَّح عنها، عندئذ يُستدعى التابع المعرّف مقدَّماً *unexpected ()*. يَستدعي هذا التابعُ التابعَ *terminate()*. يمكن للمبرمج تعديل هذا السلوك الافتراضي بتحديد تابعٍ خاصٍ به لاستدعائه باستخدام التابع الأولي set_unexpected (). يجب أن يكون محدِّدُ هذا التابع عنوانَ إحرائيةٍ (تابعٌ يعيد *void*) ليس لها محددات. وفيما يلي مثالٌ لتوصيف استثناء:

```
void f (const Rat&) throw (DivisionByZero) ;
```

عند تعريف تابع يجب استخدام نفس توصيف الاستثناءات الموجود في التصريح!

يولّد نظام C++ في بعض الحالات استثناءات. فعند عدم قدرة المؤثر *dynamic_cast* على إجراء التحويل تُرسِلُ استثناءً من النمط *bad_cast*.

يُرسِل كلا المؤثرين *new* و *new[]* عند فشلهما استثناءً من نمط *bad_alloc*. بأخذ ما سبق بعين الاعتبار يكون توقيعهما كالتالي:

```
void* operator new(size_t vol) throw (bad_alloc);
void* operator new[] (size_t vol) throw (bad_alloc);
```

4– 7 مداومة الأغراض (Object Persistence) في C++

نعني بالمداومة إمكان إطالة عمر أغراضِ برنامجٍ ما إلى ما بعد حياة البرنامج الذي يُنشئُها، وذلك بتخزين الأغراض على القرص الصلب مثلاً. يسمح ذلك للمبرمج لاحقاً بإعادة قراءة

هذه الأغراض من الذاكرة الرئيسية ومعالجتها، أو بإجراء اتصال بين برنامجين يعالجان الصفوف نفسها.

لا تقدّم C++ إلا الإمكان المقدم في الفصل الثاني، أي إلحاق ملفٍ بقناة دخلٍ أو خرجٍ. لذلك فإن جعل الأغراض مداومة يتطلّب جهداً من المبرمج.

4– 8 المكتبة المِقيَس للقوالب Standard Template Library

تضم هذه المكتبة مجموعة كاملة تقريباً وعمومية من الصفوف المستخدمة بتكرار. أُخِذَ جزء منها من مكتبةٍ موجودة سلفاً ومطورة في مخابر البحث في Hewlett-Packard من قبل Alexander Stepanov و Meng Lee[4].

تعتمد هذه المكتبة STL، كما يوحي اسمها، كلياً على مفهوم القالب. وتتألف من خمسة مكونات أساسية:

- *الحاويات* containers: وهي بنى (أغراض) قادرة على احتواء أغراض أخرى؛
- *المُكَرِّرات* iterators: التي تقدم وسيلةً تسمح لخوارزمية بعبور حاوية ما؛
- *الخوارزميات* algorithms: التي تُعَرِّف إجرائية للحساب؛
- *"الأغراض التوابع"* function objects: التي تُكبسِلُ تابعاً ما ضمن غرض، بحيث يمكن لمكونات أخرى استخدامه؛
- *الموائمات* adapters: وتسمح بموائمة المكونات بإعطائها واجهة أخرى.

اعتنى مطورو هذه المكتبة بتنفيذها بصيغة فعالة جداً، بحيث يمكن تطبيق خوارزميات

---

[4] يمكن الحصول على معلومات حديثة حول لغة C++ (توفرها على المنصات المختلفة ، المنتجين ، دروس، تطور المِقيَس ...) على العناوين التاليين :

http://lomond.maths.warwick.ac.uk/C++

http://info.desy.de/user/projects/C++.html

المكتبة أيضاً على بنى معرفة سلفاً في ++C.

لن نعرض هنا جميع التوابع الأولية (يستوجب ذلك كتاباً كاملاً)، ولكن بعض الأمثلة فقط بحيث يُقَدِّرُ القارئ إمكانات STL:

تتضمن المكتبة صنفين من الحاويات:

- الحاويات المتتالية التي تُخزِّن الأغراض التي تتضمنها ضمن ترتيب تسلسلي؛
- الحاويات التجميعية التي تُمكِّنُ من العثور بسرعة على غرضٍ مُضَمَّنٍ فيها أو أكثر، اعتماداً على مفاتيح للنفاذ.

من ضمن الحاويات المتتالية يسمح *vector<T>* و *deque<T>* بالنفاذ، كما في حالة الجداول، إلى متتاليةٍ متغيرة الطول من الأغراض التي لها نفس النمط. وإضافة إلى ذلك هنالك النمط *liste<T>* الذي يسمح بإدارة متتاليةٍ بزمن ثابت. افتراضياً، يجب استخدام النمط *vector*، لكن إذا كانت المتتالية المستخدمة تحتاج إلى الكثير من الإدراج/الحذف في وسط المتتالية، فمن الأفضل استخدام النمط *liste*. أما إذا كانت هذه العمليات تجري على رأس أو نهاية المتتالية فيجب استخدام النمط *deque*.

هناك أربع حاويات تجميعية: *multimap, map, multiset , set*. المفاتيح ضمن *set* و *map* وحيدة بعكس *multimap*. لا يمكن لـ *set* أن تتضمن سوى المفاتيح، في حين تسمح *map* بقرن معلومة من النمط *T* بكلٍ من المفاتيح.

هناك أنماطٌ مختلفة من المكررات: فبعضها يسمح بالتقدم فقط، وبعضها بالتراجع فقط، وبعضها بالتوضع مباشرة على غرض ضمن الحاوية.... يدل المؤثر "++"، حسب نمط المكرر، على التقدم إلى الغرض التالي و "--" على الرجوع إلى الغرض السابق. ويستخدم المؤثران "=+" و "=-" لإجراء الانتقالات الطويلة، كما يستخدمان لإضافةٍ (أو طرح) عددٍ إلى (من) مكرِّر. يمكن مقارنة المكررات، كما يمكن الوصول إلى قيمتها باستخدام "*".

تُصنَّفُ الخوارزمياتُ أيضاً ضمن أصناف مختلفة. نذكر منها العمليات التي لا تعدل أغراض المتتالية (*for_each, find, count, ...*) والعمليات التي تعدِّل أغراض المتتالية (*replace,*

(*remove, reverse,...*) وعمليات الفرز (*binary_search , partial_sort, stable_sort , sort,...*)، الخ.

لتوضيح ما سبق، سنعطي مثالاً على استخدام نمط *vector*.

```
#include <vector.h>
#include <algorithm.h>
#include <functional.h>
#include <iostream.h>
#define _BOOLEAN_     // type bool defined in <vector.h>
#include "string7.h"

int display_element(const vector<String>::value_type& v) {
        cout << v << '\t';
}

void display(const vector<String>::iterator& deb, const vector<String>::iterator& fin) {
  for_each(deb, fin, ptr_fun(display_element)); cout << endl;
}

int main() {
  String input;
  vector<String> v(6, "toto");          // the vector is initialised then displayed
  display(v.begin(),v.end());
  vector<String>::iterator i;
  for(i = v.begin(); cin >> input; ++i)
          (*i) = input;   .
  vector<String>::iterator last = i;
  display(v.begin(),last);

  sort(v.begin(),last);                  // sort (quick sort) then displayed
  display(v.begin(),last);

  i = v.begin();                         // insert an element in the first position
  cout << (*i) << endl;
  i = v.insert(i,"blabla");              // the dimension of the vector is now 11
  display(v.begin(),v.end());

  reverse(v.begin(),v.end());           // return the vector
  display(v.begin(),v.end());
}
```

لما كان النمط *bool* معرَّفاً في STL، فإننا نحذف تعريفنا لهذا النمط. يجري في هذا المثال إنشاء شعاع مؤلف من ستة عناصر، القيمة البدائية لجميع عناصره "toto". يجري إظهار

الشعاع، ابتداءً من العنصر الأول حتى الأخير. تعيد الطريقتان *begin()* و *end()* متحولاً من النمط مكرِّر *iterator* على العنصرين الأول والأخير من الغرض. وهذا ما يظهر في توقيع المحددات الصورية للإجرائية *display()*. يظهر في هذه الإجرائية تابع أولي آخر *for_each()* يسمح بعبور متتالية، حيث يُطبِّقُ هذا التابعُ الغرضَ التابعَ المعطى بمحدده الثالث، على جميع الأغراض التي يمكن أن يصل إليها اعتباراً من القيمة البدائية للمكرر حتى قيمته النهائية.

يُبنى الغرضُ التابعُ بتطبيقِ الموائم *ptrfun()*. النمط *value_type* هو *typedef* مُعرَّف سلفاً ويعيد نمط عنصر الجدول (هنا String). تُبيّن حلقة *for* التالية كيفية تعريف مكرِّر مرتبطٍ بالشعاع وكيفية عبوره بوضع السلاسل المحرفية المقروءة على ملف الدخل المِقيَس (استخدام المؤثرين "++" و "*"). يجري بعد ذلك إنشاء المكرر *last* بهدف تخزين موقع آخرِ غرضٍ فعالٍ في الشعاع. ثم يجري إظهار الجزء الفعال من الشعاع ويفرز (يرتب). افتراضياً، تستخدم خوارزمية الفرز sort() العملية operator<() لإجراء المقارنات. يسمح التابع الأولي *insert()* بإدراج عنصرٍ ضمن الجدول، وتحديث أبعاد هذا الجدول. تسمح الخوارزمية *reverse()* المعرَّفة سلفاً بقلب شعاع. فيما يلي خرج تنفيذ البرنامج السابق:

```
toto  toto  toto  toto  toto  toto
nino
like
lisete
^Z
nino  like  lisete
like  lisete nino
like
blabla like  lisete nino  toto  toto  toto
toto  toto  toto  nino  lisete like  blabla
```

يقوم المثال التالي بإظهار جميع الكلمات التي يمكن تشكيلها بتغيير ترتيب محارف السلسلة المحرفية المقروءة من ملف الدخل المِقيَس. يجري إنشاء شعاعٍ مؤلف من عشرة محارف char، القيمة البدائية لجميع عناصره '\0'. بعد إظهار السلسلة المحرفية يُدخِلُ المستخدمُ السلسلةَ التي يريدها محرفاً محرفاً. تُحذف من سلسلة محارف الدخل جميع محارف '\0'، ويشير المكرر *last* إلى المحرف الأخير من الشعاع المتبقي. يُفرَزُ بالنهاية هذا الشعاع وفق الترتيب التصاعدي،

وتُولّدُ وتُظهرُ جميع التباديل الممكنة لعناصر الشعاع المحصورة بين الموقعين i و j. تستخدم لذلك الخوارزمية المُعرَّفة سلفاً *next_permutation*.

```
#include <vector.h>
#include <iostream.h>
#include <algorithm.h>
#include <functional.h>

int display_element(const vector<char>::value_type& v) {
cout << v;
}

void display(const vector<char>::iterator& deb, const vector<char>::iterator& fin) {
  for_each(deb, fin, ptr_fun(display_element)); cout << '\t';
}

int main () {
  vector<char> v(10,'\0');
  display(v.begin(),v.end()); cout << endl;
  char ch;
  vector<char>::iterator i;
  for(i = v.begin(); cin >> ch; ++i) (*i)=ch;
  display(v.begin(),v.end()); cout << endl;

  vector<char>::iterator last = remove(v.begin(),v.end(),'\0');
  display(v.begin(),last); cout << endl;
  sort(v.begin(),last);
  display(v.begin(),last); cout << endl;

  i = v.begin();
  vector<char>::iterator j = last;
  while(next_permutation(i,j)) display(v.begin(),last);
}
```

نحصل مثلاً عند تنفيذ البرنامج على:

```
lina^z
lina
ailn
ainl   alin   alni   anil   anli   ialn   ianl   ilan   ilna   inal
inla   lain   lani   lian   lina   lnai   lnia   nail   nali   nial
nila   nlai   nlia
```

كمثال أخير، سنعيد التمرين المدروس سابقاً في الفصلين 2 و3: الأداة التي تحصي جميع الكلمات التي تُدخلها وتحسب عدد مرات ظهور كل كلمة ضمن النص.

هذه المرة سنستخدم بنية معطيات تجميعية وهي الـ *map* بدلاً من الشعاع. مفتاح الجدول من النمط *String* وهو غير مكرر. المعلومة المربوطة من النمط *int*، وتابع المقارنة المستخدم هو التابع الافتراضي *less (operator<())* من النمط *String*. تُقرأ كل كلمةٍ من ملفِ الدخلِ المِقيَس وتُدرَج ضمن الجدول، أي الـ *map*. بنية المعطيات التي تستخدمها STL في إدراج وإعادة عنصر من وإلى الـ *map* هي ثنائيةٌ، لعنصرها الأول نمط المفتاح نفسه والنافذُ *first*، ولعنصرها الثاني نمط المعلومة المربوطة والنافذ *second*.

تُنشئ التعليمة map<String,int,less<String> >:: value_type (word, 0) دخلاً كهذا، حيث تُعطي العدد المربوط القيمة البدائية 0. يُدرج هذا الدخل في جدولنا، ولأن ورود المفتاح وحيد فلا تدرج الكلمة إذا كانت موجودة سلفاً ضمن الجدول. أما إذا لم تكن موجودة فتدرج وتصبح قيمة العداد الموافق 0. في الحالتين، تُرفَعُ قيمة العداد الموافق للكلمة. تُخزَّن المفاتيح في الجداول وفق الترتيب المتزايد.

```
#include <map.h>
#include <algorithm.h>
#include <functional.h>
#include <iostream.h>
#define _BOOLEAN_     // type bool defined in <vector.h>
#include "string7.h"

int display_input(const map<String,int,less<String> >::value_type& v) {
        cout << v.first << '\t' << v.second << endl;
  return 0;
}

void display(const map<String,int,less<String> >::iterator& deb, const
map<String,int,less<String> >::iterator& fin) {
  for_each(deb, fin, ptr_fun(display_input)); cout << endl;
}

int main() {
  map<String,int,less<String> > table;
  String wrd;
  while(cin >> wrd) {
          table.insert(map<String,int,less<String> >::value_type(wrd,0));
    ++table[wrd];
  };
```

```
  display(table.begin(),table.end());
}
```

يمكن للقارئ أن يلاحظ أن المكتبة STL التي ليست سوى جزءٍ من مكتبة C++ المِقيَـــس، قويةٌ جداً وسهلةُ الاستخدام. ثم إن الفوائد التي تتحقق باستخدامها عديدة:

- ليس على المبرمج أن يعيد اختراع الدولاب في كل تطبيق.
- مكتبة *STL* قوية وفعالة جداً، وذلك أمر يصعب تنفيذه دون خبرة كبيرة.
- يصبح عمل المبرمج في مستوى أعلى، وهذا ما يجنبه إدارة تفاصيل التنفيذ.
- يكون التطبيق المنفذ محمولاً بين منصات عديدة تدعم الـــ *C++* المِقيَس.

فهرس أبجدي

220 ..... iterators

L

152 ..... late binding

M

64 ..... Modificators
195 ,133 ..... multiple inheritance
125 ..... mutable

N

126 ..... namespace

O

130 ..... Object
219 ..... Object Persistence
90 ..... operator

P

135 ..... Polymorphism
139 ..... private
58 ..... procedural abstraction
140 ..... protected
140 ..... public

R

133 ..... repeated inheritance

S

142 ..... scope resolution
220 ..... Standard Template Library
137 ..... Static binding

T

158 ,80 ..... template
209 ..... throw
210try

A

156 ..... abstract classes
58 ..... abstract data types
138 ..... abstraction by parameterization
138 ..... abstraction by specification
64 ..... Accessors
220 ..... adapters
220 ..... algorithms
131 ..... ancestor
208 ..... assertions

C

209 ..... catch
63 ..... class
131 ..... Class
64 ..... Constructors
220 ..... containers

D

69 ..... default constructor
131 ..... descendant
64 ..... Destructor
137 ..... Dynamic binding

E

130 ..... encapsulation
208 ..... Exception management

F

74 ..... friend
220 ..... function objects

G

79 ..... generic classes

I

130 ..... information hiding
131 ..... inheritance
131 ..... instantiation

U

using 126

V

virtual 152
virtual binding 137

أ

أنماط معطيات مجردة 58

إ

إبطال المدى 142
إخفاء المعلومات 130
إدارة الاستثناءات 208

ا

استنساخ 131
الأغراض التوابع 220
الباني الافتراضي 69
البواني 64
التابع الوهمي الصرف 157
التجريد الإجرائي 58
التجريد بالتوصيف 138
التوكيدات 208
الحاويات 220
الحاويات التجميعية 221
الحاويات المتتالية 221
الخوارزميات 220
الربط الديناميكي 137
الربط الساكن 137
الربط المتأخر 152
الربط الوهمي 137

الصديق 74
الصف 63, 131
الصفوف العمومية 79
الصفوف المجردة 156
الغرض 130
القالب 80, 158
الكبسلة 130
المعدّلات 64
المكتبة المعيارية للقوالب 220
المُكَرِّرات 220
الموائمات 220
النوافذ 64
الهادم 64
الوراثة 131
الوراثة البسيطة 139
الوراثة المتعددة 133, 195
الوراثة المتكررة 133

ت

تحويل الأنماط 119
تعددية الأشكال 135, 152

خ

خاصّ 139

ز

زيادة تحميل المؤثرات 85

س

سلفٌ 131
سليل 131

ع

عام 140

ف

فضاء الأسماء 126

م

عمي 140

مداومة الأغراض 219

و

والتحريد باستخدام المُوسِطات 138

Damascus University

الجزء الثاني

Java

وقل رب زدني علما
جامعة دمشق
Damascus University

الفهرس

الفصل 1 : مقدمة 241

الفصل 2 : لمحة عامة 243

2-1 المترجم (COMPILER) والمفسر (INTERPRETER) 243
2-2 تطبيق جافا 244
2-3 برمجيات جافا 246
2-4 الحزم (PACKAGES) 248
2-5 تعليمة IMPORT 249
2-6 مُعيّنات الهُويّة (IDENTIFIERS) و الكلمات المحجوزة (RESERVED WORDS) 249

الفصل 3 : عناصر البرنامج 251

3-1 الأنماط البسيطة 251
3-2 الأنماط المركبة 253
3-3 التعابير (EXPRESSIONS) والعمليات (OPERATORS) 254
3-4 التعليمات STATEMENTS 255
3-4-1 التعليمة الفارغة (Empty Statement) 256
3-4-2 تعليمة الكتلة (Block Statement) 256
3-4-3 استدعاء الطرائق (Method Invocation) 256
3-4-4 تعليمات الحجز (Allocation Statements) 256
3-4-5 تعليمات الإسناد (Assignment Statements) 257
3-4-6 تعليمة if 257
3-4-7 لصاقات التعليمات (Statement Labels) 257
3-4-8 تعليمة الابتدال (switch Statement) 257
3-4-9 تعليمة break 258
3-4-10 تعليمة for 259
3-4-11 تعليمة while 259
3-4-12 تعليمة do 259
3-4-13 تعليمة continue 259
3-4-14 The synchronized Statement 259

3-4-15 تعليمة try 260
3-4-16 تعليمة return 260
3-4-17 تعليمة throw 260

الفصل 4 : الصفوف والأغراض 261

4-1 الصف 261
4-2 الأغراض 264
4-3 الوراثة 265
4-4 الربط الديناميكي وتعددية الأشكال 268
4-5 المعدّلات MODIFIERS 268
4-5-1 معدلات قابلية الرؤية 269
4-5-2 معدلات أخرى 269

الفصل 5 : الدخل والخرج 271

5-1 القنوات STREAMS 271
5-2 صوانات (BUFFERS) الدخل/الخرج 272
5-3 الدخل الرقمي 273
5-4 المزيد عن صفوف الدخل والخرج 273

الفصل 6 : أغراض لتنظيم المعطيات 276

6-1 المصفوفات (ARRAY) 276
6-2 VECTOR 278
6-3 الصف STRING 279
6-4 الصف HASHTABLE 281

الفصل 7 : بُرَيْمجات جافا (APPLET) 284

الفصل 8 : البيانيات 288

8-1 الصف GRAPHICS 288
8-2 نظام الإحداثيات البيان 289
8-3 الألوان 290
8-3-1 الألوان المعرفة سلفاً 291

8-3-2 تعريف الألوان 292
8-3-3 النمط XOR 293
8-4 أشكال الرسم 294
8-4-1 القطوع (ovals) 295
8-4-2 المستطيلات 298
8-4-3 الأقواس Arcs 301
8-4-4 المضلعات 303
8-4-5 متعدد المستقيمات 305
8-4-6 البنوط Fonts 306
8-4-7 مثال: الكرة الوثّابة Bouncing Ball 308

الفصل 9 : واجهات الاستخدام البيانية GUI 312

9-1 عناصر GUI 312
9-2 البرمجة المقودة بالأحداث 314
9-2-1 واجهات الأحداث 318
9-3 المكونات والحاويات 323
9-3-1 الحاويات Containers 323
9-3-2 هيكلية المكونات 325
9-4 مكونات GUI 327
9-4-1 اللصاقات Label 328
9-4-2 حقول النصوص TextField و مساحات النصوص TextArea 328
9-4-3 القوائم: lists 331
9-4-4 الأزرار Buttons 332
9-4-5 Scrollbars 334
9-5 LAYOUT MANAGER (متولي الترتيب) 337
9-5-1 Flow Layout (الترتيب التدفقي) 337
9-5-2 Grid Layout (الترتيب الشبكي) 339
9-5-3 Border Layout (الترتيب الحافّ) 341
9-5-4 Card Layout (الترتيب البطاقي) 342
9-5-5 الترتيب الشبكي الحقائبي Grid Bag Layout 344
9-6 مثال: مقتطفات QUOTES 349

الفصل 10 : التحكم المتقدم بسير البرنامج 356

10-1 معالجة الاستثناءات (EXCEPTIONS) 356
10-2 تعدد المسالك (MULTI-THREADING) 359
10-2-1 إنشاء المسالك 359
10-2-2 مشكلة مع المسالك 361
10-2-3 المزامنة (Synchronization) 362
10-2-4 التحكم في المسالك 364

الفصل 11 : خاتمة 370

Damascus University

الفصل 1 : مقدمة

جافـــا هي لغة برمجة غرضية التوجه مشتقة من لغة C++ ومبسطة لها. طَوَّرَت جافا أساساً شركة *Sun Microsystems* ضمن إطار مشروع تطوير برمجية خاصة بالتجهيزات الإلكترونية المنزلـــية مـــثل التلفاز والفيديو وسخانة الخبز الخ. ولكن تطور الويب جعل *Sun* توجِّه هذا المشروع توجيهاً مختلفاً قليلاً لينفرج سنة 1991 عن لغة برمجة سهلة الاستعمال، ونقالة على مختلف منصات العمل، وأمثلية لتوزيع البرامج التنفيذية على الويب.

تسمح جافا، إضافةً إلى إنشاء تطبيقات مستقلة، بإنشاء بُرَيمجات جافا (*java applet*[1]) التي تمـــتلك نفـــس الوظائف الأساسية للتطبيقات، ولكن سبب وجودها هو قدرتها على العمل ضـــمن متصـــفح ويب متوافق مع جافا. تشبه بريمجات جافا، عندما نراها ضمن متصفح الويب، الصور ولكنها تطبيقات ديناميكية وتفاعلية. يمكن لهذه البريمجات أن تسمح بإنشاء إحياءات أو أشكال أو مساحات تستجيب لأعمال المستخدم، أو ألعاب أو تأثيرات تفاعلية أخـــرى على صفحة ويب واحدة تتضمن نصوصاً وصوراً. يُنشئ مُبَرمِج جافا بريمجَه مرةً واحدةً ويأخذ المتصفح، المتوافق مع جافا، على عاتقه تفسير هذا البريمج وتشغيله على منصة العمل التي يعمل عليها المتصفح. كان *HotJava* (متصفح *Sun*) أول متصفح ويب قادر على تشغيل بريمجات جافا، ثم تبِعَهُ *Netscape* و *Internet Explorer*.

منذ عام 1995 طرحت *Sun* ثلاثة إصدارات هامة من *Java*:

- خـــريف 1995: طرحت *Java 1.0* وهو إصدار مصمم للاستعمال على الويب، وذو كمون كبير ليكون مفيداً في مجالات أخرى.
- ربـــيع 1997: طرحت *Java 1.1* وهو تحديث تضمن عدداً كبيراً من التحسينات في

[1] كلمة *applet* هي تصغير *application* أي برنامج تطبيقي صغير.

طريقة إنشاء واجهات الاستخدام ومعالجتها.

- صـــيف/خـــريف 1998: طرحت *Java 1.2* (عُرف بعد ذلك بـــ *Java 2*) وهو آخر الإصدارات وأكبر بثلاث مرات من *Java 1.0*، يتضمن تحسينات ترفع جافا إلى مستوى لغات البرمجة المعممة (*generalized*).

احتوت جافا 2 على جافا 1.1 وأضافت إليها حزماً جديدة مثل *Swing*. سنتحدث في هذا الكـــتاب بالمقام الأول عن جافا *1.1* لأنها نواة جافا 2 وهي قاعدة كافية لاستيعاب المفاهيم الأساسية لجافا وأسلوب عملها.

الفصل 2 : لمحة عامة

يُعتبر أسلوب التعلم بالأمثلة من أكثر أساليب التعليم فعاليةً، حيث يُعطى مثالٌ على كلّ سمة من سمات لغة البرمجة. إلا أن كتابة مثال ما يتطلب أكثر من تعليمة، ويتطلب تشغيل هذا المثال معرفةَ بيئة العمل وأسلوب عمل المترجم أو المفسر. لذلك يبدأ تعليم لغات البرمجة من مثال بسيط، وهو البرنامج الشهير بـ *Hello World* الذي نشرحه ونفصله ونترجمه دون الدخول في تفاصيل التعليمات المستخدمة أو في التركيب النحوي للُّغة، ثم نعدّله لاحقاً لاختبار السمات الأولية للغة البرمجة، إلى أن نمتلك من الأدوات ما يسمح بكتابة برامج متطورة بلغة البرمجة المعنية.

قبل أن نتمكن من ترجمة برنامج ما، لا بد لنا من امتلاك مترجم. يزودنا *JDK* (*Java Developpement Kit*) بمترجم ومفسر، كما يزودنا بـ *JDK Applet viewer* لتشغيل برمجات جافا التي يمكن تنفيذها ايضاً باستعمال أي متصفح ويب متوافق مع جافا مثل *HotJava* و *Netscape* و *Internet Explorer*.

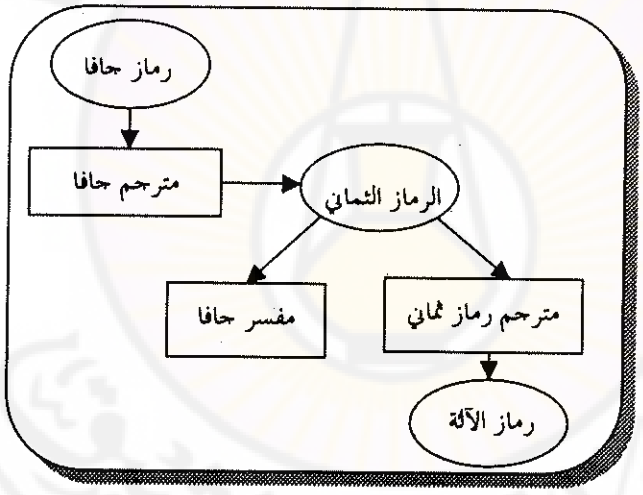
2-1 المترجم (compiler) والمفسر (interpreter)

بعد كتابة برنامجٍ ما وتخزينه على القرص الصلب، لا بد من ترجمته إلى لغة الآلة قبل أن نصبح قادرين على تنفيذه. تقوم معظم المترجمات بتحويل الرماز المصدر (*source code*) إلى لغة الآلة (*machine language*) ويصبح عندها بالإمكان تنفيذ هذا البرنامج في أي وقت نشاء.

يشبه المفسرُ المترجمَ من حيث الوظيفة، لكنه يدمج عمليتي الترجمة والتنفيذ، حيث يترجم التعليمات الواحدة تلو الأخرى وينفذها قبل الانتقال إلى التعليمة التالية وهكذا. بالطبع، فإن ذلك يبطئ من سرعة التنفيذ، لأن عملية الترجمة تتكرر كلما نُفِّذَت تعليمة ما، ولو وجدت

هذه التعليمة ضمن حلقة. يتميز المفسر بأنه يجعل بالإمكان تنفيذ البرنامج على أية آلة تمتلك المفسر المناسب.

تتضمن عملية ترجمة وتنفيذ برامج جافا استعمالاً لمترجم ومفسر معاً، فتُتَرجم برامج جافا إلى لغة خاصة تشبه لغات الآلة تُسمى الرماز الثماني (*bytecode*) وتختلف عن لغات الآلة النظامية بعـــدم وجود معالج يُنَفذ تعليمات هذه اللغة. يقوم مفسر جافا بقراءة الرماز الثماني وتنفيذه على آلة معينة. ويمكن استعمال مترجم آخر (غير المترجم الذي ترجم برنامج جافا إلى الرماز الثماني) لترجمة الرماز الثماني إلى لغة الآلة المعنية. يوضح الشكل 2-1 مراحل تنفيذ برنامج جافا.



الشكل 2-1: مراحل تنفيذ برنامج جافا

2-2 تطبيق جافا

نســـتطيع أن نـــبدأ بتحقيق *Hello World* وهو تطبيق جافا يعمل مستقلاً عن أي متصفح، ويعلمـــنا بعـــض مفاهـــيم جافـــا وكيف نعرف صفوف جافا، وكيف نستعمل الصفوف

والأغراض المتاحة سلفاً. لنتبع الخطوات التالية:

- ننشئ الملف المصدري التالي ونسميه *HelloWorldApp.java*

```
/*
 * The HelloWorldApp class implements an application that
 * simply displays "Hello World!" to the standard output.
 */
class HelloWorldApp {
   public static void main(String[] args) {
      System.out.println("Hello World!"); //Display the string.
   }
}
```

- نترجم الملف المصدر باستعمال *javac* (وهو جزء من *JDK*) على سبيل المثال. إذا نجحت عملية الترجمة فسوف يتولد ملف يسمى *HelloWorldApp.class* أما إذ فشلت فيجـــب التوثق من اسم البرنامج والرماز. تتطلب تطبيقات جافا أن يكون اسم الملف مطابقاً لاسم الصف الذي يحتوي الطريقة *main*.

- نشغل البرنامج باستعمال مفسر جافا، فنرى كلمة *"Hello world"* قد ارتسمت على الشاشة.

يبدأ هذا البرنامج بتعليق عام وذلك باستعمال علامات التعليق /* */ المشابهة لما نعرفه في لغة *C++* ونشاهد أيضاً في هذا البرنامج نوعاً آخر من التعليقات وهو المُعرَّف بالعلامة // المأخوذة أيضـــاً عن لغة *C++*. إلا أن جافا تمتلك نوعاً ثالثاً من التعليقات يستعمل التركيب /** **/ وهي التعليقات التي يستعملها *javadoc* وهو برنامج متضمن في *JDK* يولد آلياً، عند تنفيذه على ملف أو حزمة جافا، دليلَ استثمارٍ للصفوف المعنية، وذلك اعتماداً على التعليقات ذات التركيـــب /** **/ التي يعطيها المبرمج (راجع دليل استثمار المترجم الذي تستعمله لمزيد من التفاصيل).

يتضمن هذا البرنامج صفاً واحداً يسمى *HelloWorldApp* يحتوي على طريقة واحدة تسمى *main*. هذه الطريقة معرفة لتكون *public* و*static*. نلاحظ أن التعريف *public* يكوِّن جزءاً من ترويسة الطريقة *main* ولا يتعداها إلى الطرائق الأخرى، على النقيض من لغة *C++* حيث

يؤثـــر المعدِّل *public* في جميع الطرائق التي تليه ما لم يحدد غير ذلك. أما فيما يتعلق بالمعدِّل *static* فيُستَعمل بنفس أسلوب *C++*.

لا يوجد في لغة جافا إجرائيات مستقلة كما في لغة *C++* وإنما طرائق متضمنة في الصفوف. وتكون نقطة البدء للبرنامج هي الطريقة *main* الموجودة ضمن الصف الذي يحمل اسم الملف الذي يحتويه. لهذه الطريقة ترويسة ثابتة وهي *public static void main(String[] args)*.

تـــأخذ الطـــريقة *println*، الخاصة بالغرض *io* المُعَرَّف ضمن الصف *system*، سلسلة محرفية وتقوم بطباعتها على الخرج المِقيَسيّ. سنتحدث عن الدخل والخرج بتفصيل أكبر في الفصل الخامس.

3-2 بُرَيمجات جافا

لكتابة بريمج جافا نتبع الخطوات التالية:

- ننشئ الملف المصدر وليكن *HelloWorld.java*:

```
import java.applet.Applet;
import java.awt.Graphics;

public class HelloWorld extends Applet {
   public void paint(Graphics g) {
      g.drawString("Hello world!", 50, 25);
   }
}
```

- نترجمه باستعمال مترجم جافا فنحصل على *HelloWorld.class*
- ننشـــئ ملف *hello.html* لاحتواء البريمج. يجب أن يحتوي هذا الملف على الرماز التالي:

```
<HTML>
<HEAD>
<TITLE> A Simple Program </TITLE>
</HEAD>
<BODY>

Here is the output of my program:
```

```
<APPLET CODE="HelloWorld.class" WIDTH=150 HEIGHT=25>
</APPLET>
</BODY>
</HTML>
```

- حمّـــل الملـــف *Hello.html* في تطبيق قادر على تنفيذ برمجات جافا مثل متصفح *Netscape* أو *Internet Explorer* أو بـــرنامج *Applet viewer* المـــرفق بـــ *JDK*. بعد تحميل الملف سوف تشاهد شيئاً يشبه الصورة التالية:

Here is the output of my program: Hello world!

تقـــوم التعليمـــتان الأولى والثانية من البرمج باستيراد الصف *Applet* الموجود ضمن الحزمة *java.applet* وباســـتيراد الصف *Graphics* الموجود ضمن الحزمة *java.awt*. في الواقع تجمع جافـــا الصفوف المتعلق بعضها ببعض ضمن حزم وتجمع الحزم المتعلق بعضها ببعض ضمن مجلدات كما هو حال المجلد *java* الذي يحتوي على العديد من الحزم المعرفة للغة جافا (انظر المقطع *packages*).

يـــلي تعلـــيمات استيراد الصفوف المساعدة تعريفاً للصف *HelloWorld* المشتق من الصف *Applet*. تدل الكلمة المفتاحية *public* على أنه من الممكن استعمال الصف *HelloWorld* من قـــبل أي صـــف آخـــر. لو لم يكن هذا الصف معرفاً *public* لبقي استعماله مقصوراً على الصفوف التي تنتمي إلى نفس الحزمة (انظر 4.5 لمزيد من المعلومات)

يلي استيراد الصفوف المطلوبة التعليمات التالية:

```
public class HelloWorld extends Applet {
   public void paint(Graphics g) {
      g.drawString("Hello world!", 50, 25);
   }
}
```

تُعرف هذه السطور برمجاً جديداً يسمى *HellolWorld* وذلك باشتقاقه من الصف المقيسي *Applet* المعرَّف في الحزمة *java.applet*. الطريقة *paint* معرفة أساساً في الصف الأب *Applet* وتُطلـــب آلـــياً عندما يحتاج المتصفح إلى إعادة عرض البرمج، ويمرَّر الغرض البياني *g* الممثِل لسطح البرمج، لتمكين المبرمج من رسم ما يشاء على سطح البرمج. نستعمل في هذا المثال

الطـــريقة *drawString* مـــن الصـــف *Graphics* لكتابة نص ضمن البرنمج، في الموقع المحدد بالإحداثيات المُمَرَّرة لهذه الطريقة. (لمزيد من المعلومات عن الصف *Graphics*.انظر 8).

4-2 الحزم (Packages)

تُرَتَّب برامج جافا ضمن حزم (*packages*) تحتوي على الرماز المصدر لصفوف وواجهات (*interface*) جافا (انظر 4.3). تمتلك الصفوف التي تنتمي إلى نفس الحزمة حقوقاً مختلفة عن تلك التي تنتمي إلى الحزم الأخرى (انظر 4.5). تحدد الحزم باستعمال تعليمة *package* التي يجب أن تكون أول تعليمة في البرنامج المصدر ولها الشكل التالي:

package packageName;

حيث يعرف *packageName* اسم الحزمة التي تنتمي إليها جميع الصفوف المعرفة ضمن الملف الحالي. يُستعمل مسار الحزمة واسم الصف لتحديد الصف المطلوب مثل *java.applet.Applet*. يمكن لصف واحد فقط أو واجهة (*interface*) واحدة فقط أن تُعرف عمومية *public* ضمن ملـــف رماز مصدر (وحدة ترجمة). يمكن تعريف صفين وواجهة ضمن ملف وحيد ولكن يمكـــن لأحدهـــا فقط أن يعرف *public*. يجب أن يكون اسم الملف مطابقاً لاسم الصف أو الواجهة العمومية متبوعاً باللاحقة *java*.

تحتوي جافا قياسياً على العديد من الحزم وهي :

الحزمة	وصفها
Language	تحتوي على النواة الرئيسة للغة جافا
Utilities	تحتوي بعض بنى المعطيات المفيدة
I/O	تعرف عدة أنواع من الدخل/خرج
Networking	دعم لـــ *TCP/IP* وبرمجة المقابس
AWT[2]	تصميم واجهات الاستخدام ومعالجة الأحداث

[2] *Abstract Windowing Toolkit*

Text	دعم للتدويل
Security	دعم للتشفير الأمني
RMI	دعم للبرمجة الموزعة
Reflection	تستعمل للحصول على معلومات أثناء التنفيذ
SQL	تستعمل لاستجواب قواعد المعطيات

تحتوي الحزمة *java.lang* على الصفوف الأساسية المعرفة للغة جافا (انظر الملحق أ) وهي:

Boolean	*Byte*	*Character*
Class	*ClassLoader*	*Compiler*
Double	*Float*	*Integer*
Long	*Math*	*Number*
Object	*Process*	*Runtime*
SecurityManager	*Short*	*String*
StringBuffer	*System*	*Thread*
ThreadGroup	*Throwable*	*Void*

5-2 تعليمة import

تستعمل تعليمة *import* باستيراد الصفوف والواجهات المعرفة ضمن حزمٍ أخرى، بغرض الإشارة إليها ضمن صفوف الحزمة الحالية. ولهذه التعليمة ثلاثة أشكال:

```
import packageName.className;
import packageName.interfaceName;
import packageName.*;
```

تسمح الصيغتان الأولى والثانية بالإشارة إلى الصف أو الواجهة المذكورة في التعليمة ضمن البرنامج دون تحديد اسم الحزمة التي تحويهما بينما تسمح الصيغة الثالثة بالإشارة إلى جميع الصفوف والواجهات المحتواة ضمن الحزمة المعنية دون تحديد اسم الحزمة. يمكن الإشارة إلى صفٍ ما دون استيراد حزمته وذلك بإعطاء اسمه الكامل إي اسم الصف مسبوقاً باسم الحزمة وهذا الحل هو الحل الوحيد لتتميز بين صفين لهما نفس الاسم وينتميان إلى حزمتين مختلفتين.

6-2 مُعيِّنات الهُويَّة (Identifiers) و الكلمات المحجوزة (Reserved Words)

تُستعمل معينات الهوية لإعطاء أسماء لعناصر برامج جافا، وتبدأ هذه الأسماء بحرف أبجدي أو

بالشَّرْطة التحتية (_) أو بعلامة الدولار ($) ويليها محارف من نفس النوع إضافةً إلى الأرقام. وبالطبع تميز جافا بين الحروف الصغيرة والكبيرة. نتجنب عادة استعمال إشارة $ لأنها تستعمل في أسماء معينات الهوية المُوَلّدة آلياً من قبل المترجم.

الكلمات التالية محجوزة من قبل جافا ولا يمكن استعمالها كمعينات هوية:

abstract	*boolean*	*break*	*byte*
case	*catch*	*char*	*class*
*const**	*continue*	*default*	*do*
double	*else*	*extends*	*final*
finally	*float*	*for*	*goto**
if	*implements*	*import*	*instanceof*
int	*interface*	*long*	*native*
new	*package*	*private*	*protected*
public	*return*	*short*	*static*
super	*switch*	*synchronized*	*this*
throw	*throws*	*transient*	*try*
void	*volatile*	*while*	

* هذه الكلمات محجوزة ولكنها غير مستعملة في جافا.

الفصل 3 : عناصر البرنامج

اســـتُلهِمَت جافـــا من لغة C++ وتشابه تراكيبُها النحوية، التراكيب النحوية للغة C++، وسنســـتفيد مـــن ذلك في هذا الفصل لتقديم لغة جافا بأقصر طريق ممكن. وكمعظم لغات البرمجة يتألف البرنامج من أنماط بسيطة ومن أنماط مركبة وتعابير وتعليمات. وسنكتفي هنا بالإشـــارة إلى الفـــروق بين جافا وC++ وسنفرد مقاطع خاصة لاحقاً لتفصيل الحديث عن الأغراض والصفوف والمسالك *(threads)* ومعالجة الاستثناءات، لأن الاختلاف مع لغة C++ عديدة وتتطلب الوقوف عندها.

3-1 الأنماط البسيطة

تُعـــرِّف جافا ثمانية أنماط بسيطة. المتحولات المعرفة كنمط بسيط ليست بأغراض، وإنما هي مســـاحات في الذاكرة لحفظ القيمة الموافقة للنمط البسيط فقط. هذه الأنماط البسيطة هي *byte, short, int, long, float, double, char, and boolean.*

وصف	الحجم	النمط
عدد صحيح حجمه بايت واحد	*8-bit*	*byte*
عدد صحيح قصير	*16-bit*	*short*
عدد صحيح	*32-bit*	*int*
عدد صحيح طويل	*64-bit*	*long*
عدد عائم الفاصلة بسيط الدقة	*32-bit*	*float*
عدد عائم الفاصلة مضاعف الدقة	*64-bit*	*double*
حرف واحد	*16-bit*	*char*
قيمة منطقية (صح أو خطأ)	*true or false*	*boolean*

تكتب القيم التصريحية لهذه المتحولات باستعمال أعداد صحيحة عشرية أو ست عشرية أو ثمانية. تُسبَق الأعداد الست عشرية بـ *0x* أو *0X* وتستعمل الأحرف من *a* (أو *A*) إلى *f* (أو *F*) لتمثيل الأرقام من 10 إلى 15. تُسبق الأعداد الثمانية بالرقم 0. كما تُلحق قيم الأرقام العشرية الطويلة بالحرف *l* أو *L*. تمتلك القيم من نمط *float* اللاحقة *f* أو *F* كما تمتلك القيم من نمط *double* اللاحقة *d* أو *D* وهي اللاحقة الاصطلاحية في حال غياب اللاحقة.

تُمثل المحارف باستعمال 16 بت (بمجموعة المحارف *Unicode*) وهذا مايمكننا من تمثيل جميع محـــارف اللغـــات السائدة بدلاً من مجموعة المحارف *ASCII* المستعملة في لغة *C++* والتي لا تســـمح بتمثـــيل أكثر من 256 محرفاً مختلفاً. تُمثل القيم المحرفية باستعمال علامة الاقتباس البسيطة ' مثل *'a'* ويستثنى من ذلك ثلاثة محارف لأن لها معنىً خاصاً وهي (') و (") و (\) وتستعمل \ رمزَ هروبٍ لتمثيل قيم بعض المحارف الخاصة والممثلة بالجدول التالي:

Code	Character
\b	*Backspace*
\t	*Tab*
\n	*Linefeed*
\f	*Form feed*
\r	*Carriage return*
\"	*Double quote*
\'	*Single quote*
\\	*Backslash*

يُمثل النمط *boolean* قيماً منطقية تأخذ القيم *true* أو *false*. ولا يكافئ النمط *boolean* في جافـــا أي قيمة صحيحة كما هو الحال في لغة *C++* وإنما هو نمط مستقل في ذاته ولا تعامل التعابير الصحيحة كتعابير منطقية.

لكل نمط أولي في جافا صفٌ يُغلفه (*wrapper class*) ويمتلك هذا الصف نفس نمط المعطيات، لكـــنه يُمـــثل بصـــف يمتلك العديد من الطرائق (*methods*) المفيدة للتعامل مع هذا النمط، وخاصـــة طرائق التحويل بين الأنماط. على سبيل المثال يحتوي الصف *Integer* الذي يغلف

النمط البسيط *int*، على طريقة *parseInt* لتحويل سلسلة من محرفية تحتوي على عدد صحيح إلى القيمة الصحيحة الموافقة.

تمتلك جافا الصفوف المغلفة التالية: *Boolean*، *Byte*، *Character*، *Double*، *Float*، *Integer*، *Long*، *Short*، *Void* وتوجد جميعاً في الحزمة *java.lang* من *API* جافا.

تحتوي جافا أيضاً على نمط سلاسل محرفية، ولكن السلاسل هذه ليست بأنماط بسيطة وإنما أغراض. انظر 6.3.

جافا لغة شديدة التنميط ولا يمكن إسناد قيمة من نمط ما إلى متحول من نمط مختلف.

2-3 الأنماط المركبة

كجميع لغات البرمجة تزودنا جافا بتراكيب نحوية تسمح ببناء أنماط أكثر تطوراً انطلاقاً من الأنماط البسيطة. لا تمتلك جافا التركيب النحوي *enum* لكنها تقبل الثوابت ذات الأسماء (انظر *final*) ولا تقبل كذلك التراكيب النحوية *struct* و *union* ذلك لأنها صممت لتكون غرضية التوجه كلياً وما وجود هذه التراكيب في لغة *C++* إلا نتيجة الإرث الذي خلفتها لها لغة *C* والذي يهدف إلى بقاء لغة *C++* متوافقة مع لغة *C*. نُذكر أنه من الممكن الاستغناء عن التركيب *struct* باستعمال التركيب *class* كما يمكن الاستغناء عن التركيب *union* باستعمال تقنيات الوراثة وتعددية الأشكال.

في الواقع تمتلك جافا تركيباً نحوياً وحيداً لبناء الأنماط المركبة وهو الصف (*class*) الذي يفيد، كما في حال لغة *C++*، في تعريف الصفوف التي سنتطرق لها لاحقاً في هذا الكتاب. كما تحتوي في صلبها (*java.lang*) على صفين لتنظيم المعطيات هما *Array* و *String*. وتتضمن أيضاً غرضين لتنظيم معطيات موجودين ضمن الحزمة *java.util* وهما *vector* و *hashTable* (انظر الفصل السادس).

3-3 التعابير (expressions) والعمليات (operators)

تتضمن معظم لغات البرمجة تعابيرَ حسابية وهي مزيج من العمليات (*operators*) والمعاملات (*operands*) التي تنفذ عمليةً ما. وليست القيمة المحسوبة نتيجةً لتقويم هذه التعابير رقمية دائماً، إلا أنها غالباً كذلك. تعرف جافا عمليات حسابية وعمليات مقارنة، و عمليات منطقية وعمليات معالجة البتات، وعمليات تحويل بين الأنماط وعمليات على الصفوف وعمليات انتقاء وعمليات إسناد. يبين الجدول التالي هذه العمليات (يجدر بنا الإشارة هنا إلى أن جافا لا تتقبل التحميل الزائد للعمليات كما هو حال *C++*):

نوع العملية	العملية	وصف	مثال
حسابية	+	جمع	*a + b*
	-	طرح	*a - b*
	*	ضرب	*a * b*
	/	قسمة	*a / b*
	%	باقي القسمة الصحيحة	*a % b*
مقارنة	>	أكبر تماماً من	*a > b*
	<	أصغر تماماً من	*a < b*
	>=	أكبر من أو يساوي	*a >= b*
	<=	أصغر من أو يساوي	*a <= b*
	!=	لا يساوي	*a != b*
	==	يساوي	*a == b*
منطقية	!	عكس	*!a*
	&&	و	*a && b*
	||	أو	*a || b*
معالجة بتية	~	متممة	*~a*
	&	و	*a & b*
	|	أو	*a | b*

	^	أو مقصورة	a ^ b
	<<	إزاحة يسارية	a << b
	>>	إزاحة يمينية	a >> B
	>>>	إزاحة يمينية مع تعبئة صفرية	a >>> b
إسناد	=	إسناد	a = b
	++	زيادة مع إسناد	a++
	--	إنقاص مع اسناد	a--
	+=	جمع وإسناد	a += b
	-=	طرح وإسناد	a -= b
	*=	ضرب وإسناد	a *= b
	/=	تقسيم وإسناد	a /= b
	%=	حساب باقي القسمة الصحيحة وإسناد	a %= b
	|=	أو وإسناد	a |= b
	&=	و وإسناد	a &= b
	^=	أو المقصورة وإسناد	a ^= b
	<<=	إزاحة يسارية مع إسناد	a <<= b
	>>=	إزاحة يمينية مع إسناد	a >>= b
	>>>=	إزاحة يسارية وتعبئة صفرية وإسناد	a >>>= b
تحويل بين الأنماط	(type)	تحويل إلى نمط جديد	(char) b
نسخة	instanceof	نسخة من الصف ؟	a instanceof b
حجز	new	إنشاء نسخة جديدة من الصف	new A()
التقاء	? :	الاختيار if then	a ? b : c

4-3 التعليمات Statements

تمتلك جافا مجموعة من التعليمات مشابهةً جداً للغة C++ وسندرج هذه التعليمات فيما يلي مع ذكر الاختلاف فقط مع C++ إن وجد، وإلا سنكتفي بذكر نَحْوِ التعليمة فقط.

3-4-1 التعليمة الفارغة (Empty Statement)

وتتمثل بعلامة ; كما في لغة C++.

3-4-2 تعليمة الكتلة (Block Statement)

تتكون هذه التعليمة من سلسلة من التعليمات والتصريحات التي تعامل معاملة التعليمة الوحيدة، وتجمع ضمن القوسين } و { كما في لغة C++.

3-4-3 استدعاء الطرائق (Method Invocation)

لا يوجد في لغة جافا أية توابع أو إجرائيات مستقلة وإنما طرائق معرفة ضمن الصفوف. وتستدعى هذه الطرائق بأحد الأسلوبين التاليين:

```
objectName.methodName(argumentList);
className.methodName(argumentList); // for static method
```

يوجد هنا اختلاف مع لغة C++ إذ تستدعى الطرائق الساكنة باستعمال :: بعد اسم الصف، على حين تستعمل جافا النقطة (.).

3-4-4 تعليمات الحجز (Allocation Statements)

عندما نحجز غرضاً ما في الذاكرة فإننا نُسنده عادةً إلى متحولٍ ما، وإن لم يكن هذا قيداً من قيود اللغة. لتعليمة الحجز الشكل العام التالي:

```
new constructor(argumentList);
```

لتعليمة الحجز هذه نفس معنى الحجز في لغة C++ مع اختلافين:

- تعيد تعليمة *new* في لغة C++ مؤشراً إلى الغرض المحجوز، في حين تعيد في جافا مرجعاً (*reference*) للغرض الجديد.
- لا يوجد في لغة جافا تعليمة *delete* وإنما يُدَمَّر الغرض آلياً عندما لا يبقى هنالك أي مرجع يشير إليه. تسمى الإجرائية التي تقوم بهذا العمل بجامع القمامة (*Garbage*

Collector) وتُـنَفذ في مسـلكٍ (*thread*) مختلفٍ عن مسلك البرنامج الرئيسي (انظر الفصل العاشر).

3-4-5 تعليمات الإسناد (Assignment Statements)

تسند تعليمةُ الإسناد قيمةً أو غرضاً إلى متحولٍ ولها الشكل العام التالي:

```
variableName = expression;
```

حيـث يعيد التعبير *expression* قيمة لها نمط متوافق مع نمط المتحول. كما يوجد أسوةً بلغة *C++* تعليمات إسناد أخرى (انظر فصل التعابير والعمليات).

3-4-6 تعليمة if

تستعمل للاختيار بين مسارات بديلة للتنفيذ ولها أحد الشكلين التاليين

```
if ( booleanExpression ) statement
if ( booleanExpression ) statement1 else statement2
```

يـستطابق تعريف تعليمة *if* في لغة حافا مع تعريفها في لغة *C++* إلا أنها في حافا تقتضي أن يكون الشرط تعبيراً منطقياً ولا تُقبل الأرقام الصحيحة كقيم منطقية كما هي حال *C++*.

3-4-7 لصاقات التعليمات (Statement Labels)

يمكن ربط لصاقة أو رمز بتعليمة ما كما يلي:

```
label: statement
```

ويمكن لهذه اللصاقة أن تكون اسماً أو عدداً صحيحاً.

3-4-8 تعليمة الابتدال (switch Statement)

تشبه تعليمة *switch* في حافا نظيرتها في *C++* وتكتب كما يلي:

```
switch (expression){
  case value1: codeBlock1; break;
```

```
  case value2: codeBlock2; break;
  .
  .
  case valuen: codeBlockn; break;
  default : codeBlock;
}
```

حيث يعيد التعبير *expression* قيمة من نمط *int* أو *short* أو *byte* أو *char* كما في المثال التالي:

```
switch (c){
 case '1': case '3': case '5': case '7': case '9':
  System.out.println ("c is an odd number");
  break;
 case '0': case '2': case '4': case '6': case '8':
  System.out.println ("c is an even number");
  break;
 case ' ':
  System.ou..println ("c is a space");
  break;
 default :
  System.out.println ("c is not a number or a space");
  }
```

9-4-3 تعليمة break

تنقل تعليمة *break* التحكم إلى تعليمة موسومة (*labeled statement*) أو تُخرِج التحكم خارج تعليمة الكتلة وتأخذ الشكل التالي:

```
break;
break label;
```

حيث يُستعمَل الشكل الأول لنقل التحكم إلى أول تعليمة بعد الكتلة الحالية ويُستعمَل الشكل الثاني لنقل التحكم إلى التعليمة ذات اللصاقة *label*.

3-4-10 تعليمة for

مماثلة لتعليمة *for* في لغة *C++* ولها الشكل العام التالي:

```
for (initializationStatement ; booleanExpression ; incrementStatement)
  iteratedStatement
```

3-4-11 تعليمة while

تستعمل لتنفيذ تعليمة معينة ما تحقّق شرط ما ولها الشكل العام التالي:

```
while (booleanExpression) iteratedStatement
```

3-4-12 تعليمة do

لها نفس دلالة تعليمة *do* في لغة *C++*، ولها الشكل العام التالي:

```
do iteratedStatement while (booleanExpression);
```

3-4-13 تعليمة continue

هذه التعليمة خاصة بحافا، وتُستعمل لمتابعة تنفيذ حلقة (*do* أو *for* أو *while*) بدون إكمال تنفيذ التعليمات (*iteratedStatement*) المكررة ضمن الحلقة. يمكن لهذه التعليمة أن تأخذ لصاقة كمعامل وتُكتب عندها على الشكل:

```
continue label;
```

في هذه الحالة يُتابع التنفيذ انطلاقاً من اللصاقة *"label"*.

3-4-14 The synchronized Statement

تُستعمل لتنفيذ تعليمة بعد إقفال غرض معين (انظر الفصل العاشر) ولها الشكل العام التالي:

```
synchronized ( expression ) statement
```

يعيد التعبير *expression* الغرض الذي نود قفله.

15-4-3 تعليمة try

تُنفذ هذه التعليمة كتلة من التعليمات وترصد مجموعة من الاستثناءات (انظر الفصل العاشر) الـــتي قد تنجم عن تنفيذ هذه الكتلة. عند حدوث استثناءٍ ما، يتوقف تنفيذ كتلة التعليمات وينتقل التحكم إلى الترميز المناسب، إن وجد، لمعالجة الاستثناء. تتميز تعليمة *try* في لغة جافا عـــن نظيرتهـــا في لغة C++ بأنها قد تمتلك عبارة *finally* التي تحدد مجموعة من التعليمات الواجب تنفيذها حتماً سواءً حدث استثناءٌ أم لا. لهذه التعليمة الشكل العام التالي:

try block catchClauses finallyClause

يجب أن يكون هنالك على الأقل عبارة *catch* أو *finally*.

لعبارة *catch* الشكل العام التالي:

catch (exceptionDeclaration) block

ولعبارة *finally* الشكل:

finally block

16-4-3 تعليمة return

تُستعمل لإعادة غرض أو قيمة نتيجةً لتنفيذ تابع ما، وتشبه في الوظيفة تعليمة *return* الخاصة بلغة C++.

17-4-3 تعليمة throw

تُســـتَعمل لإرســـال استثناء ونقل التحكم إلى معالج الاستثناءات الموافق (انظر فصل معالجة الاستثناءات).

الفصل 4 : الصفوف والأغراض

كمـــا ذكـــرنا سابقاً، اشتقت لغة جافا من لغة C++ وهي تشبهها كثيراً من حيث السلوك وأسلوب تعريف الصفوف واشتقاقها. لن نتناول هنا أية مفاهيم جديدة لأنه سبق شرحها في الجزء الأول من هذا الكتاب، ولكننا سنعرض بالدرجة الأولى الفروق النحوية والدلالية (إن وُجدت) بين صفوف جافا وصفوف C++.

نذكر أن عناصر البرمجة الغرضية التوجه هي:

1- الصف *(Class)* الذي يعرف مجموعة كامنة من الأغراض (معمل).

2- الغرض *(Object)* وهو الكيان الرئيسي أثناء تنفيذ البرنامج (مُنتج).

3- الوراثة *(Inheritance)*: علاقة بين الصفوف تسمح بتعريف صف اعتماداً على آخر.

4- الـــربط الديناميكي *(Dynamic Binding)* : الربط هو العلاقة بين التعريف والطلب ويكون ديناميكياً عندما تجري هذه العملية أثناء التنفيذ لا أثناء الترجمة.

5- تعددية الأشكال *(Polymorphism)*: هو القدرة على أخذ أشكال مختلفة.

1-4 الصف

تُعـــرَّف صفوف جافا كما في C++ باستعمال الكلمة المحجوزة *class* وتختلف عنها بالنقاط التالية:

- يقســـم تعـــريف الصـــف في C++ إلى جزأين: التعريف الموجود ضمن الملف *h*. والتحقـــيق ضمن الملف *cpp*. ويمكن أحياناً تحقيق الطرائق ضمن تعريف الصف نفسه ولكن هذا يعني للغة C++ أن الطريقة المعنية هي طريقة من نمط *inline*. لا تمتلك جافا طرائق *inline* ولا تسمح بتحقيق الإجرائيات خارج إطار تعريف الصف.

- لا تسـمح C++ بإعطاء قيم بدائية لأعضاء الصف (*members*) أثناء التعريف، ويجـب أن يجـري ذلك ضمن البواني (*constructors*) فيما يتعلق بالأعضاء العادية، وضمن الملف وخارج أي تعريف لأية إجرائية، فيما يتعلق بالأعضاء الساكنة. في حين تسمح جافا بإعطاء قيماً بدائية للأعضاء الساكنة والعادية على حدٍ سواء كما في المثال التالي:

```
// C++ version
class Point {
  private:
   int x,y;
 public:
   Point() {x=y=0;}
   ...
   static Point center;
}
// within a .cpp file only
Point Point::center(0,0);
```

```
// Java version
class Point {
  private int x=0,y=0;
  public Point() {};
  ...
static Point center=new Point();
}
```

- تدخـل المعدلات *public* و *private* و*protected* في بداية تعريف كل متحول أو كل طريقة على حدة ولا تشمل الطرائق اللاحقة لها.

```
class Point {
    private int x,y;
   public Point(int x,int y) {...};
   public getX() {...};
  public getY() {...};
public static Point center=new Point(0,0);
...
}
```

- تدخل المعدلات أيضاً على تعريف الصف نفسه، فيمكن أن يكون الصف *public* أي يُـرى من قبل جميع الحزم، أو أن لا يكون له معدِّل فلا يُرى عندها إلا من قبل أغراض الحزمة التي ينتمي إليها. وكما أكدنا سابقاً لا يمكن لملف أن يحتوي أكثر من صف *public* واحد وهو الصف الذي يمتلك نفس اسم الملف الذي يحويه.

- تمـيز لغة C++ بين أربعة أنواع من البواني وهي *copy constructor* و *default constructor* و *casting* و *normal constructor* في حـين لا تمـتلك جافـا *copy*

constructor لأن أغراضها مراجع، وتُنسخ العناصر صراحةً باستعمال الطريقة *clone* المعرفة ضمن الصف *Object* والتي يجب إعادة تعريفها في الصفوف المشتقة إذا اقتضت الحاجة. ولا تسمح جافا أيضاً بتعريف البواني الخاصة بالتحويل بين الأنماط *(casting)* ويجب طلب التحويل صراحةً باستعمال التعليمة المناسبة، إلا في حال التحويل بين الصفوف المشتقة وآبائها. باختصار تُمَيز جافا بين الباني الافتراضي (*default constructor*) والباني العادي (*normal constructor*) فقط.

- لمّا كانت أغراض جافا تُحجز ديناميكياً ولا يُحَرَّرُها المستثمر وإنما جامع قمامة *Garbage Collector* يعمل في مسلك آخر، فإن صفوف جافا لا تمتلك هوادم (*destrcutors*) وإنما تمتلك طريقة خاصة تُسمى *finalize* يطلبها جامع القمامة آلياً عند تحرير الذاكرة الخاصة بالغرض. هذه الطريقة معرفة ضمن الصف *Object* ويجب إعادة تعريفها في الصفوف المشتقة إذا احتجنا إليها. لا يحرر جامع القمامة الذاكرة فوراً عندما تصبح غير نافعة، وإنما يمتلك ستراتيجية خاصة به للقيام بذلك لاحقاً. يمكن أن نستعجل جامع القمامة للقيام بعمله بطلب الطريقة *System.gc()* ولكن ذلك لا يعني أن جامع القمامة سيقوم بالعمل فوراً. وجامع القمامة هذا هو أحد الأسباب الرئيسية التي تجعل جافا غير صالحة للتطبيقات التي تتطلب حسابات بزمن حقيقي (*real time*).

- تمتلك *C++* عبارة خاصة لتهيئة أعضاء الصف، إذ تُعطى قيمها البدائية أو تُطلب بوانيها قبل الدخول في متن الباني وهذا غير متوفر في جافا ولا حاجة إليه لأن أغراض جافا مراجع تمتلك دوماً قيماً بدائية افتراضية هي الأصفار. ويمكن إعطاء القيم البدائية أثناء التعريف إذا احتجنا إلى ذلك كما شاهدنا سابقاً. في الواقع احتاجت *C++* إلى هذا التركيب النحوي الخاص لنتمكن من بناء الأغراض باستعمال الباني المناسب. نفترض في المثال التالي أن الصف *Point* لا يمتلك بانياً افتراضياً:

// This is the C++ version *class Segment {* *private:* *// p1 and p2 are values that needs to be*	*//This is the Java version* *class Segment {* *// p1 and p2 are not values they are* *// references that have already a null*

```
// initialized by a constructor before
// calling the constructor of the class
// Segment.
 Point p1, p2;
public:
// This is how C++ can explicitly call the
appropriate
// constructors for p1 and p2
 Segment (x1, y1, x2, y2): p1 (x1, y1), p2
(x2, y2)
   {...}
}
```

```
// value.
private point p1,p2;
public Segment(x1,y1,x2,y2)
   {
    p1=new Point(x1,y1);
    p2=new Point(x2,y2);
    ...
   }
}
```

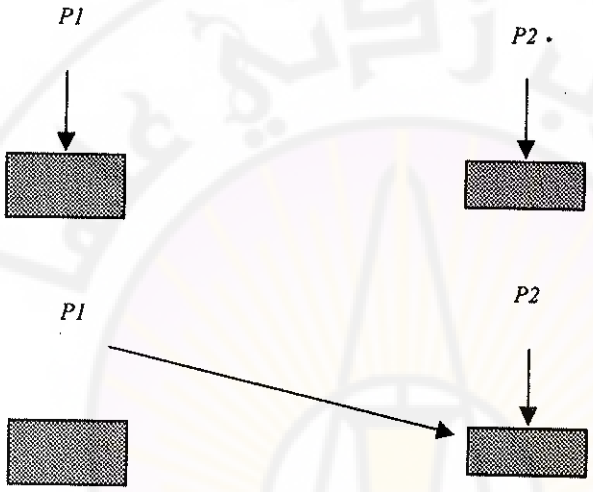
- يُشار إلى الأعضاء والطرائق الساكنة في C++ باستعمال العملية :: بعد اسم الصف على حين تستعمل حافا العملية ".".
- لا تمتلك حافا الكلمة المحجوزة *friend* بسبب وجود فكرة الحزم، إذ يمكن أن ترى الصفوف التي تنتمي إلى نفس الحزمة على أنها صفوف صديقة، فيمكن للصفوف التي تنتمي إلى نفس الحزمة أن ترى ما لا يمكن لبقية الصفوف أن تراه. كما يمكن في حافا أن يكون مجال تعريف بعض الصفوف مقصوراً على ملف واحد (انظر 4.5).
- لا تمتلك حافا الكلمة المحجوزة *namespace* بسبب وجود فكرة الحزم، إذ يؤدي إضافة اسم الحزمة قبل اسم الصف، إلى التمييز بين صفين لهما نفس الاسم وينتميان إلى حزمتين مختلفتين.

2-4 الأغراض

في حين أن أغراض لغة C++ هي قيم *(Values)* فإن أغراض حافا هي مراجع *(References)* إلى القيم الفعلية الممثلة لها. المرجع هو مؤشر ضمني إلى قيمة ما. ولما كانت الأغراض مراجع والمراجع هي مؤشرات ضمنية، شغلت أغراض حافا نفس الحجم في الذاكرة ولا يؤدي إسناد غرض من صف ما إلى غرض من صف آخر (الصف الأب) إلى أي ضياع في المعلومات كما في لغة C++. وقد شاهدنا أن تعددية الأشكال *(polymorphism)* في لغة C++ لا تعمل على

وجـــه صحيح ما لم نستعمل مؤشرات إلى الأغراض بدلاً من الأغراض نفسها، وهو شرط أساسي في اللغات الموجهة بالغرض لم تحترمه C++ للمحافظة على التوافق مع لغة C.

يؤدي إسناد غرض ما إلى غرض آخر في جافا إلى تغيير العنوان المرتبط بالغرض، ولا يؤدي



الشكل 4-1: أغراض جافا مراجع وليست بقيم.

إلى نســـخ الغرض (نستعمل لنسخ الأغراض الطريقة *clone* المعرفة في الصف *Object*، انظر 4.3)، كما هو موضح في الشكل 4-1.

ثم أن عملـــية المقارنة == بين أغراض جافا لا تعني تساوي القيم، وإنما تساوي العناوين في الذاكرة. ولذلك يجب استعمال أو إعادة تعريف الطريقة *equals* من الصف *Object* لمقارنة الأغراض بعضها ببعض (انظر 4-3).

4-3 الوراثة

هي آلية تعريف نمط جديد اعتباراً من نمط سابق:

- يمكـــن للصفوف في لغة C++ أن تكون أصلية غير مشتقة من صفوف أخرى، في حين تُشتق جميع صفوف جافا من الصف *Object* الذي يحتوي عدة طرائق يجب إعادة تعريفها في الصفوف المشتقة إذا احتجنا إليها. نذكر من هذه الطرائق: الطريقة *clone* التي تفيد في نسخ الأغراض. والطريقة *toString* التي تحول الأغراض إلى سلاسل محرفية تعبر عنها، وتُستعمل هذه الطريقة من قبل الطريقة *println* لطباعة الأغراض. والطريقة *finalize* التي يستدعيها جامع القمامة عند تحريره للذاكرة. والطريقة *getClass* التي تعيد غرضاً من نمط *Class* يتضمن معلومات عن الغرض، والطريقة *equals* التي تقارن قيمتي غرضين.

- يختلف التركيب النحوي للاشتقاق في جافا بعض الشيء عنه في لغة C++. ففي حـــين أن الاشتقاق في لغة C++ يمكن أن يكون *public* أو *protected* أو *private* فإن الاشـــتقاق في جافا لا نمط له ويُعرف باستعمال الكلمة المحجوزة *extends*. فلاشتقاق الصف *Point3d* من الصف *Point* نستعمل التعريف التالي:

```
class Point3d extends Point {
...
}
```

- تُستعمل جافا الكلمة المحجوزة *super* لطلب بانٍ معين من بواني الصف الأب، وإن لم تفعل فإن الباني الافتراضي للصف الأب سيطلب آلياً.

```
class Point3d extends Point {
  private int z;
  Point3d(int x, int y, int z) {
    // calling the constructor Point(int,int) of the super class Point.
    super(x,y);
    this.z=z;
  }
...
}
```

- كما هو مبين في المثال السابق، المتحول *this* المعبر عن الغرض الطالب للطريقة في جافا هو غرض (مرجع) وليس مؤشراً كما في حالة C++ ولذلك استعملنا معه العملية '.' ولم نستعمل <-.

- تستعمل الكلمة المحجوزة *super* أيضاً في طلب طرائق وأعضاء الصف الأب في حال وجود التباس في الأسماء كما يلي:

```
class Point3d extends Point {
...
   bool equals(PoInt3d p) {
     return super.equals(p) && z==p.z;
   }
...
}
```

- تمتلك جافا إمكان إنشاء صفوف مجردة باستعمال الكلمة المحجوزة *abstract* التي يمكن أن تدخل على الصف أو على الطريقة المجردة كما يلي:

```
abstract class C {
   abstract method(...) {
   };
}
```

- لا يوجد وراثة متعددة في جافا توخياً للبساطة، وتحل المشكلة باستعمال الواجهات *interfaces*:

تشبه الواجهات الصفوف المجردة لكنها تختلف عنها بالنقاط التالية:

- لا يمكن للواجهة أن ترث إلا واجهة ولا يمكنها أن ترث صفاً أو أن تحقق واجهة أخرى.
- لا يمكن أن يكون لأي طريقة من طرق الواجهة متن (*body*). بمعنى آخر جميع طرائقها مجردة.
- جميع طرائق الواجهة هي *public* و *abstract* وإن لم يذكر ذلك.
- جميع متحولات الواجهة معرفة *public static final* (أي ثوابت) وإن لم يذكر ذلك.

فيما يلي الشكل العام لتعريف الواجهات:

```
public interface interface_name extends other_interface{
  public abstract type method(..);
```

```
  public static final int x;
}
```

يمكن لصف ما أن يحقق ما يشاء من الواجهات ولكنه يرث صفاً واحداً فقط، وذلك باستعمال التركيب التالي:

```
class class_name extends mother_class
                implement interface_1, ..., interface_n {

}
```

عندما يصرح صفٌ ما بأنه يحقق واجهةً ما، فإن عليه أن يعرف جميع الطرائق المعرفة ضمن هذه الواجهة ما لم يكن الصف نفسه معرفاً كصف مجرد.

مثال: يمكن أن تُعرَّف النقطة الثنائية البعد كواجهة تمتلك الطرائق اللازمة لقراءة وتعديل الإحداثيات كما يلي:

```
interface Point2D {
    int getX();
    int getY();
    void setX(int x);
    void setY(int y);
}
```

4-4 الربط الديناميكي وتعددية الأشكال

من أهم الاختلافات بين *Java* و*C++* غياب الربط الساكن في جافا. في حين تستعمل *C++* الكلمة المحجوزة *virtual* للتمييز بين الربط الساكن والديناميكي فإن جافا لا تمتلك إلا ربطاً ديناميكياً. ولما كانت أغراض جافا هي مراجع (مؤشرات ضمنية) فلا حاجة إلى استعمال مؤشرات كي تعمل تعددية الأشكال على وجه سليم (*polymorphism*) كما في لغة *C++*. وعلى أي حال، لا يوجد مؤشرات في لغة جافا.

4-5 المعدِّلات Modifiers

تقسم معدلات جافا إلى فئتين: معدلات قابلية الرؤية *Visibility Modifiers* والمعدلات الأخرى.

4-5-1 معدلات قابلية الرؤية

يوضح الجدول التالي أثر معدلات قابلية الرؤية عند استعمالها مع الصفوف والواجهات، وعند استعمالها مع الطرائق والأغراض. نلاحظ أنه ليس من الضروري أن ينتمي الصف والصف المشتق منه إلى نفس الحزمة

المعدل	الصفوف والواجهات	الطرائق والأعضاء
default (no modifier)	مرئية ضمن حزمتها فقط	ترثها جميع الصفوف التحتية والموجودة ضمن نفس حزمة الصف الذي يحويها، ويمكن الوصول إليها من أي صف ينتمي إلى نفس حزمة الصف الذي يحويها.
public	مرئية في أي مكان	ترثها جميع الصفوف ويمكن الوصول إليها من أي مكان.
protected	لا معنى له	ترثها جميع الصفوف التحتية للصف الذي تنتمي إليه، ولا يمكن الوصول إليها من الصفوف الأخرى.
private	لا معنى له	لا تورث ولا يمكن الوصول إليها من قبل صفوف أخرى.

4-5-2 معدلات أخرى

يبين الجدول التالي أثر المعدلات الأخرى في مختلف عناصر اللغة.

المعدل	الصف	الواجهة	الطريقة	المتحول
abstract	يمكن للصف أن يحتوي على طرائق مجردة، ولذا ي لا يمكن استنسائه.	جميع الواجهات مجردة ولذا فهذا المعدل اختياري	لا تمتلك الطريقة متناً ويجب تحقيق متن لها عندما تُوَرث	لا يُستعمل
final	لا يمكن اشتقاق صفوف أخرى من هذا الصف	لا يُستعمل	لا يمكن إعادة تعريف الطريقة[3]	المتحول ثابت القيمة.
native	لا يُستعمل	لا يُستعمل	لا تحتاج الطريقة إلى متن لأنها محققة بلغة برمجة أخرى	لا يُستعمل
static	لا يُستعمل	لا يُستعمل	هذه الطريقة متعلقة	تعرف متحول صف فلا

[3] في هذه الحالة يقوم مترجم جافا بعملية استمثال ويعتمد الربط الساكن بدلاً عن الربط الديناميكي.

			بالصف لا بالغرض فلا نحتاج إلى غرض لاستدعائها وهي ضمنياً نهائية (*final*).	حاجة لغرض للدلالة على هذا المتحول.
synchro-nized	لا يُستعمل	لا يُستعمل	لا يمكن تنفيذها من قبل مسلكين في آن واحد.	لا يُستعمل
transient	لا يُستعمل	لا يُستعمل	لا يُستعمل	لا يؤثر هذا المتحول على حالة الصف ولا داعي لحفظه عند حفظ حالة الغلرض.
volatile	لا يُستعمل	لا يُستعمل	لا يُستعمل	يمكن أن تتغير قيمة المتحول بطريقة غير متزامنة. يجب على المترجم ألا يقوم بأية عمليات استمثال عليه.

الفصل 5 : الدخل والخرج

لكي تكون البرامج مفيدة، عليها أن تقبل دخلاً وأن تعطي خرجاً ما. إلا أن مصدر الدخل ووجهة الخرج يمكن أن تتغير. غالباً ما تقرأ البرامج وتكتب ضمن ملفات موجودة على القرص الصلب، ويأتي الدخل أحياناً من المستثمر عن طريق لوحة المفاتيح أو جهاز دخل آخر، ويرسل الخرج إلى الشاشة. أما في حالة البرامج التي تتخاطب عبر الشبكة، فإن مصدر الدخل ووجهة الخرج يمكن أن تكون اتصالاً شبكياً ما.

5-1 القنوات Streams

تتعامل جافا مع جميع أنواع الدخل والخرج بنفس الأسلوب. نسمي قناة أي مصدر دخل أو أي وجهة خرج. تتحقق جميع عمليات الدخل/الخرج باستعمال مكتبات من الصفوف معرفة ضمن بيئة جافا، ولا تحتوي جافا ذاتها أي أدوات دخل/خرج. تُعرَّف معظم عمليات الدخل/الخرج ضمن الحزمة *java.io* من الواجهة البرمجية لتطبيقات جافا (*Java API*) وقد سبق أن رأينا العمليتين *println* و *print*. يوجد ثلاث قنوات قياسية معرفة سلفاً:

- *System.in* لقراءة الدخل
- *System.out* للكتابة على الخرج
- *System.err* لكتابة رسائل الخطأ

ما لم يذكر غير ذلك تقرأ القناة *System.in* من لوحة المفاتيح وتُرسل القناتان *System.out* و *System.err* خرجهما إلى الشاشة. يوجد أنواع أخرى من القنوات التي يمكن ضبطها للقراءة والكتابة سنتطرق إليها في أماكن أخرى. تمتلك جافا عدة أنماط من القنوات التي يمكن استعمالها، لكننا سنركز على القنوات المقيسية.

ترسل الطريقتان *print* و*println* معلومات للخرج القيسي لأنهما جزء من *System.out*.

لنتفحص البرنامج التالي الذي يقرأ سلسلة محرفية ويعيد طباعتها:

```
import java.io.*;

// Reads a string from the user and prints it.
class Echo {
  public static void main(String[] args) throws IOException {
     BufferedReader stdin = new BufferedReader
          (new InputStreamReader(System.in));
     String message;

     System.out.println ("Enter a line of text");
     message = stdin.readLine();
     System.out.println ("You entered: \""+ message +"\"");
  } // method main
} // class Echo
```

فيما يلي خرج ممكن للدخل *This is my test*

```
Enter a line of text:
This is my test
You entered: " this is my text "
```

استعملنا الصف *BufferedReader* لأنه يستطيع قراءة سلسلة محرفية وهو معرف ضمن الحزمة *java.io*. أعدنا تعريف الدخل المقسيس باستعمال التعليمة:

```
BufferedReader stdin = new BufferedReader
        (new InputStreamReader(System.in));
```

ترسل الطريقة *readLine* استثناءً إذا حدث خطأ ما أثناء القراءة. يتفحص المترجم بعض الاستثناءات ويقتضي إما معالجتها أو إرسالها إلى الإجرائية الطالبة، وهذا ما اخترناه إذ تُرسل الطريقة *main* الاستثناء المحتمل *IOException* في حال حدوثه إلى المستوى الأعلى (انظر الفصل العاشر).

5-2 صوانات (Buffers) الدخل/الخرج

عند إدخال معطيات إلى برنامج ما باستعمال لوحة المفاتيح، نستطيع استعمال مفتاح العودة *Backspace* لحذف حرف أو أكثر في حالة الخطأ. ولا يُعالج السطر من قبل البرنامج حتى نضغط على مفتاح *Return*. توضع خروف الإدخال هذه في صوان *buffer* مؤقت حتى

الضغط على *Return*.

كذلـــك الأمـــر فيما يتعلق بالخرج، إذ يوضع في صوان خرج، وتؤدي بعض التعليمات إلى إظهـــاره فوراً وهو ما يسمى بـــ *flushing* وهو ما تقوم به عملية *println* آلياً ولا تقوم به عملية *print* ويمكن القيام بذلك صراحةً باستعمال الطريقة *System.out.flush()*.

5-3 الدخل الرقمي

الحل الوحيد لقراءة معلومات رقمية هو قراءتها كسلسلة محرفية ثم تحويل محتوى السلسلة إلى رقم باستعمال الطريقة *parseInt* المعرفة ضمن الصف المغلف *Integer*:

```
my_int = Interger.parseInt(my_string);
```

يمكنـــنا بطريقة مماثلة قراءة سلاسل محرفية وتحويلها إلى قيم من أنماط أخرى مثل *double* أو *float* باستعمال الصفوف المناسبة.

5-4 المزيد عن صفوف الدخل والخرج

تمـــتلك حافـــا العديد من صفوف الدخل/الخرج (انظر الشكل 5-1) منها ما يُشتق من *InputStream/OutputStream* وهي أجهزة الدخل/خرج التسلسلية الثنائية (أو البيتية) ومنه ما يشتق من *Reader/Writer* وهي صفوف الدخل/ خرج التسلسلية المحرفية (*Unicode*) ومنها ما يُشتق من *Object* وهي صفوف الدخل/خرج غير التسلسلية مثل *RandomAccessFile*.

يشـــتق مـــن القنوات نوعين من الصفوف يفيد أحدها في ربط القناة مع جهاز دخل/خرج فـــيزيائي ويضـــاف اســـم هـــذا الجهاز على اسم القناة مثل ByteArrayOutputStream أو *StringBufferInputStream* أو *FileInputStream/FileOutputStream*، ويفيد النوع الثاني في إغـــناء القـــنوات بعمليات إضافية وهي الصفوف المشتقة من الصف */FilterInputStream* FilterOutputStream ونذكر منها *DataInputStream/DataOutputStream* التي تمتلك طرائق إضـــافية للقراءة والكتابة وكذلك *BufferedInputStream* التي تمتلك صوان لتخزين الدخل

والسماح بالعودة خلفاً.

```
java.lang.Object
  |
  + --Reader
  + -- Writer
  +--java.io.InputStream
       |
       + ByteArrayInputStream
       + FileInputStream
       + FilterInputStream
       + ObjectInputStream
       + PipedInputStream
       + SequenceInputStream
       + StringBufferInputStream
  +-- File
  +-- FileDescriptor
  +-- ObjectStreamClass
  +-- OutputStream
    |
    + ByteArrayOutputStream
    + FileOutputStream
    + FilterOutputStream
    + ObjectOutputStream
    + PipedOutputStream
  +-- StreamTokenizer
  +-- RandomAccessFile
```

الشكل 5-1: هيكلة صفوف الدخل/الخرج في جافا.

تبنى القنوات حول العديد من أجهزة الدخل والخرج مثل الملفات (الصف *File*) والسلاسل المحرفية (الصف *String*) والمقابس (الصف *Socket*) والمصفوفات (الصف *Array*). نلاحظ أن الصف *File* يحتوي على معلومات حول عن الملف ولا يمثل الملف نفسه. كما نلاحظ أن الصف *RandomAccessFile* ليس مشتقاً من القنوات ولا تبنى القنوات حول هذا النوع من الملفات العشوائية الوصول، لأن القنوات هي أدوات دخل/خرج تسلسلية. يوضح المثال التالي كيفية بناء القنوات على ملفات:

```
import java.io.*;

class FileStreamsTest {
  public static void main(String[] args) {
```

```
    try {
      File inputFile = new File("farrago.txt");
      File outputFile = new File("outagain.txt");

      FileInputStream fis = new FileInputStream(inputFile);
      FileOutputStream fos = new FileOutputStream(outputFile);

      DataInputStream dfis=new DataInputStream(fis);

      int i=dfis.readInt();

      int c;

      while ((c = fis.read()) != -1) {
          fos.write(c);
        }

        fis.close();
        fos.close();
      } catch (FileNotFoundException e) {
        System.err.println ("FileStreamsTest: " + e);
      } catch (IOException e) {
        System.err.println ("FileStreamsTest: " + e);
      }
  }
}
```

Damascus University

الفصل 6 : أغراض لتنظيم المعطيات

نتـــناول في هـــذا الفصل الأغراض الخاصة بتنظيم المعطيات. تمتلك جافا مصفوفات *Array* كغيرهـــا مـــن اللغـــات المتقدمة، ولكنها تمتلك أيضاً أغراضاً أخرى لتنظيم المعطيات مثل السلاســـل المحرفية *String* وتحتوي الحزمة *Java.util* على أغراض تنظيم أخرى متطورة مثل *Vector* و *Hashtable*.

1-6 المصفوفات (Array)

تشبه مصفوفات جافا مصفوفات *C++* من حيث كونها تبدأ العد من الصفر، إلا أنها في لغة جافا أغراض وليست مؤشرات. لإنشاء مصفوفة في جافا يجب تعريف مرجع (*Reference*) لهـــذه المصـــفوفة أولاً ثم استعمال العملية *new* لإنشاء نسخة من المصفوفة، والتعليمة التالية تنشئ مصفوفةً من أحد عشر عدداً صحيحاً مفهرسة من 0 إلى 10.

```
int[] height= new int[11];
```

الأقـــواس المستطيلة *[]* في جافا هي عمليات كما في لغة *C++* ولهذه العملية الأفضلية العليا في جافا وتسمى عملية الفهرسة. تقوم عملية الفهرسة بفحص آلي لحدود المصفوفة، وتتطلب أن ينـــتمي الدلـــيل إل مجـــال المصفوفة، أي أكبر أو يساوي الصفر وأصغر تماماً من طول المصـــفوفة الأعظمـــي (في الحالة السابقة ينتمي الدليل إلى المجال 0..10) فإذا تبين أن قيمة الفهرس خاطئة، يرسل استثناءً يسمى *ArrayIndexoutOfBoundsException*.

وتتميز جداول جافا باحتوائها على طول الجدول في ثابت خاص (لا يمكن تعديله بعد إنشاء المصـــفوفة) ضـــمن الغـــرض "مصـــفوفة" يسمى *length*. في مثالنا السابق يساوي الثابت *height.length* القيمة 11.

يمكن تعريف الجدول باستعمال تركيبين نحويين متكافئين هما:

```
type_name[] variable_name;
```

و

type_name variable_name[];

مثال:

int[] x;

و

int x[];

التركيـــب الأول متوافق أكثر مع بقية التراكيب النحوية للتعاريف، فعلى سبيل المثال يمكن تعريف ثلاثة متحولات صحيحة باستعمال العبارة:

int i1, i2, i3;

وكذلك ثلاثة متحولات جدولية باستعمال العبارة:

int[] a1, a2, a3;

أما العبارة

int a1[], a2, a3[];

فإنها تعرف *a1* و *a2* كجداول و*a2* كعدد صحيح.

يمكـــن تعريف الجداول واستنساخها دون استعمال العملية *new* وذلك بإعطائها قيماً بدائية كما في *C++*. تعطى القيم البدائية هذه عند تعريف الجدول وتوضع بين قوسين *{}*. مثال

```
int[] scores={87, 98, 69, 54, 76, 87, 99};
char[] letter_grades={'A', 'B', 'C', 'D', 'F'};
String[] names={"Jane", "Megan", "Eric"};
```

عـــند تعريف جدول من الأغراض فإن هذه الأغراض لا تعطى قيماً بدائية، فهي ليست إلا مراجع تشير إلى قيم، ولذا فلا بد من بناء كلٍ من هذه الأغراض على حدة.

مثال:

```
Member[] member_list;
member_list= new Member[4];
member_list[0]= new Member();
```

تقنياً لا تحتوي جافا على جداول متعددة الأبعاد، بيد أنه يمكن الحصول عليها بتعريف جدول كل عنصر من عناصره هو بدوره جدول آخر. في الواقع، التركيب النحوي لتعريف الجداول هو:

type_name[] variable_name;

فإذا كان *type* من نمط جدول أي من الشكل *type[]* لحصلنا على جدول ذي بعدين (ليس مربعاً بالضرورة) كما يلي:

type_name[][] variable_name;

ويمكن إعطاء هذا الجدول الجديد قيماً بدائية كما يلي:

int[][] x= {{1, 2, 3}, {3,4}, {5}};

ويكـــون المتحول *length* هو طول الجدول الأساسي، وهو في مثالنا السابق *x* وطوله 3. في حين يكون *x[0].length* مساوياً لـــ 3 و*x[1].length* مساوياً لـــ 2 و*x[2].length* مساوياً لـــ 1.

2-6 Vector

وهو جزء من الحزمة *java.util* ويقدم خدمه مشابهة للمصفوفات من حيث إنه يسمح بتخزين مجموعـــة عناصر، والوصول إليها انطلاقاً من فهرس. في حين يبقى بعد المصفوفة ثابتاً طوال فترة حياتها، فإن حجم الشعاع يتغير وفقاً للحاجة، ويمكن إضافة عناصر معطيات إلى الشعاع باستدعاء طريقة بسيطة.

صُمم الصف *Vector* ليحتوي على عناصر من نمط *Object* وبذلك يمكنه احتواء أي غرض كـــان، لأن جميع الصفوف مشتقة من الصف *Object*، ويجب وضع الأنماط البسيطة ضمن صفوف مُغَلَّفَة لنتمكن من إضافتها إلى *Vector*.

يحتوي الشعاع على الطرائق التالية لمعاملة مجموعة العناصر:

addElement(Object element)
removeElement(Object element)
contains(Object element)
elementAt(int index)
firstElement()
lastElement(
size()

تضـــيف الطـــريقة *addElement* عنصـــراً إلى نهايـــة الشـــعاع في حـــين تحذف الطريقة *removeElement* الغرض المرغوب من الشعاع. تعيد الطريقة *contains* القيمة *true* إذا وُجد

العنصر المعطى ضمن الشعاع وتعيد الطريقة *elementAt* الغرض الموجود في الموقع *index* من الشـــعاع. كمـــا تعيد الطريقتان *firstElement* و*lastElement* العنصرين الأول والأخير على التوالي، وتعطي الطريقة *size()* عدد العناصر الفعلي في الشعاع.

يوضح المثال التالي استعمالاً بسيطاً للصف *Vector* فيضع مجموعة من الأسماء ضمن شعاع، ثم يطبعها على الشاشة ثم يحذف أحدها ويستبدل به آخر ويعيد الطباعة.

```
import java.util.Vector;

// Uses a vector object to store a list of strings.
class Beatels {

   public static void main (String[] args)  {
        Vector band= new Vector();
        band.addElement("Paul");
       band.addElement("Pete");
      band.addElement("John");
      band.addElement("George");
      System.out.println (band);

      band.removeElement("Pete");
      band.addElement("Ringo");
      System.out.println (band);
   } // method main
} // Beatels
```

ويكون خرج البرنامج كالتالي:

```
[Paul, Pete, Johne, George]
[Paul, Johne, George, Ringo]
```

يجب استعمال الصف *Vector* بحذر، فالعمليات على الشعاع ليست آنية. عند إنشاء الشعاع يكون له سعة بدائية عظمى يودي تجاوزها (بإضافة عناصر إلى الشعاع) إلى توسيع الشعاع وهـــي عملـــية تستغرق بعض الوقت. ثم إن إدخال عنصر في مكان محدد يودي إلى إزاحة العناصر التالية له بمقدار خانة إلى الأمام.

3-6 الصف String

ليست السلاسل المحرفية بأنماط بسيطة في جافا، وإنما تُمثل بصف يسمى *String*. نمط *String*

حقـــيقي يمثل السلاسل المحرفية كأغراض لا كمصفوفات خطية كما هو حال C++ التي لا تمتلك النمط *string* في صلب اللغة وإنما في مكتبة *STL* الملحقة. ولما كانت السلاسل المحرفية هامـــة جداً وكثيرة الاستعمال جعلت لها جافا تركيباً نحوياً إضافياً لتعريفها وإعطائها قيمةً بدائية وهو:

```
String name= "Ammar";
```

ويمتلك الصف *String* بانياً خاصاً يسمح ببناء السلسلة انطلاقاً من سلسلة أخرى، فالعبارة

```
String name2= new String(name);
```

تنشئ سلسلة أخرى تمتلك نفس القيمة *"Ammar"*. ويمكن إضافة السلاسل المحرفية بعضها إلى بعض باستعمال عملية الجمع +. السلاسل المحرفية غير قابلة للتبدل فلا يمكن زيادة طولها أو إنقاصـــه أو تعديل أي حرف ضمنها. ولذلك تزودنا جافا بصف آخر قابل للتبدل يسمى *StringBuffer*.

فيما يلي بعض الطرائق المفيدة في الصف *String*:

تعيد الحرف الموجود في الموقع المحدد	*charAt(int index)*
تعيد موقع الحرف المعطى	*indexOf(char ch)*
تعيد موقعَ آخرِ ورود للحرف المعطى	*lastIndexOf(char ch)*
محققة إذا انتهت السلسلة بالسلسلة المعطاة	*endsWith(String suffix)*
محققة إذا ابتدأت السلسلة بالسلسلة المعطاة	*startsWith(String prefix)*
محققة إذا احتوت السلسلة والوسيط نفس الأحرف	*equals(Object obj)*
محققة إذا تساوت السلسلتان بصرف النظر عن طبيعة الحروف	*equalsIgnoreCase(String str)*
تقارن السلسلتين	*compareTo(String str)*
تعيد سلسلة مساوية للأصلية لكن بحروف صغيرة	*toLowerCase()*
تعيد سلسلة مساوية للأصلية لكن بحروف كبيرة	*toUpperCase()*
تعيد طول السلسلة	*length()*

يشبه الصف *StringBuffer* الصف *String* ولكنه قابل للتعديل، ويتضمن عدة طرائق منها:

تضيف الحرف المعطى في نهاية السلسلة	*append(char c)*
تضيف الحرف المعطى في المكان المعطى من السلسلة	*insert(int index, char c)*
تعيد الحرف الموجود في الموقع المعطى	*charAt(int index)*
تبدل بالحرف الموجود في الموقع المعطى الحرف الجديد المعطى	*setCharAt(int index, char c)*
تعدل الطول الأعظمي للسلسلة	*setLength(int new_length)*
تعيد مقلوب السلسلة	*reverse()*
تعيد طول السلسلة	*length()*

أمـــا الصف *Stringtokenizer* فيستعمل لتحليل مكونات سلسلة ما إلى مفردات (*Tokens*) اعـــتماداً على حروف خاصة تسمى الفواصل، مثل الفراغ ونهاية السطر، ويمكن تحديد هذه الفواصل من قبل المبرمج. كثيراً ما يستعمل هذا الصف لتحليل دخل نصي لبرنامج ما.

4-6 الصف Hashtable

يعـــرف الصـــف *Hashtable* ضمن الحزمة *java.util* ويسمح بإنشاء علاقة بين مجموعة من المفاتيح والقيم المرتبطة بها، ويتميز بسرعة الوصول إلى هذه القيم انطلاقاً من المفاتيح، وهي عـــادة بـــزمن ثابت أو شبه ثابت (لوغاريتمي مثلاً). يمكن أن يكون المفتاح أي غرض غير معدوم، ويمكن أن تكون القيمة أي غرض كان.

كي نتمكن من إنشاء الأغراض والبحث عنها ضمن *Hashtable* يجب على هذه الأغراض أن تحقق الطريقة *hashCode* والطريقة *equals* المعرفتين ضمن الصف *Object*.

تمتلك كل نسخة من *Hashtable* وسيطين يؤثران في فعاليته وهما سعته ومعامل تحميله. ينتمي معـــامل التحمـــيل إلى المجال 0..1 وعندما يتجاوز عدد مداخل *Hashtable* حداء معامل التحميل في السعة الحالية، تُزاد السعة باستعمال الطريقة *rehash*. كلما ازداد معامل التحميل كان استعمال الذاكرة أمثلياً أكثر على حساب سرعة البحث في *Hashtable*.

من المستحسن إنشاء *Hashtable* بسعة كافية من البداية فهذا أفضل من توسيعه آلياً.

ينشئ المثال التالي *Hashtable* من الأعداد، ويستعمل أسماءها كمفاتيح:

```
Hashtable numbers = new Hashtable();
numbers.put("one", new Integer(1));
numbers.put("two", new Integer(2));
numbers.put("three", new Integer(3));
```

للعثور على رقم ما نستعمل الرماز التالي:

```
Integer n = (Integer)numbers.get("two");
if (n != null) {
   System.out.println ("two = " + n);
}
```

يمتلك الصف *Hashtable* ثلاثة بوانٍ الأول لا يأخذ أي وسطاء ويأخذ الثاني وسيطاً يمثل سعة الـــ *Hashtable* ويـــأخذ الثالث وسيطين السعة ومعامل التحميل. كما يمتلك العديد من الطرائق المفيدة نذكر منها:

clear()	تمسح جميع المداخل
clone()	تعيد نسخة منه
contains(Object)	تتحقق من أن القيمة المعطاة موجودة ضمن الجدول
containsKey(Object)	تتحقق من وجود المفتاح المعطى ضمن الجدول
elements()	تعيد غرض تعداد[4] خاص بقيم الجدول
get(Object)	تعيد القيمة المرتبطة بالمفتاح المعطى
isEmpty()	تفحص خلو الجدول من العناصر
keys()	تعيد غرض تعداد خاص بمفاتيح الجدول
put(Object, Object)	تربط المفتاح المعطى بالغرض المعطى
rehash()	توسع سعة الجدول
remove(Object)	تزيل المفتاح والقيمة الموافقة من الجدول

[4] يسمى غرض التعداد في جافا بـــ *enumeration*.

تعيد عدد المفاتيح في الجدول	*size()*

وقل رب زدني علما
جامعة دمشق
Damascus University

الفصل 7 : بُرَيْمجات جافا (Applet)

تُحَقِّــق بريمجات جافا بإنشاء صف مشتق من الصف *Applet* الذي يظهر لنا الشكل التالي شجرته الوراثية:

```
java.lang.Object
    |
    +----java.awt.Component
          |
          +----java.awt.Container
                |
                +----java.awt.Panel
                      |
                      +----java.applet.Applet
```

سنستعرض البريمجات من خلال بريمج بسيط، يظهر سلسلة محرفية تصف الأحداث الأساسية في حياة البريمج، مثل أول زيارة للبريمج من قبل مستخدم ما.

```
import java.applet.Applet;
import java.awt.Graphics;

public class Simple extends Applet {

    StringBuffer buffer;

    public void init() {
      buffer = new StringBuffer();
      addItem("initializing... ");
    }

    public void start() {
      addItem("starting... ");
    }

    public void stop() {
      addItem("stopping... ");
    }

    public void destroy() {
      addItem("preparing for unloading...");
```

```
    }

    void addItem(String newWord) {
        System.out.println (newWord);
        buffer.append(newWord);
        repaint();
    }

    public void paint(Graphics g) {
        //Draw a Rectangle around the applet's display area.
        g.drawRect(0, 0, size().width - 1, size().height - 1);

        //Draw the current string inside the rectangle.
        g.drawString(buffer.toString(), 5, 15);
    }
}
```

نستطيع تَعَرُّف دورة حياة البرنمج من خلال المثال *Simple* الذي يمتلك الشكل العام التالي:

```
public class Simple extends Applet {
        ...
        public void init() { . . . }
        public void start() { . . . }
        public void stop() { . . . }
        public void destroy() { . . . }
        ...
    }
```

يعيد البرنمج *Simple* تعريف أربع طرائق معرفة ضمن الصف *Applet* وهذا ما يسمح له بالاستجابة للأحداث الرئيسية:

- *init()*: تقوم هذه الطريقة بتهيئة البرنمج كلما حُمّل البرنمج أو أعيد تحميله.
- *start()*: تُبدئ عمل البرنمج كلما حُمّل أو كلما أعاد المستخدم زيارة الصفحة التي تحتوي على البرنمج.
- *stop()*: توقف عمل البرنمج عندما يغادر المستخدم الصفحة أو يغلق المتصفح.
- *destroy()*: يقوم بتنظيف نهائي تحضيراً لإنهاء التحميل وذلك عند إغلاق المتصفح عادةً.

ليس من الضروري أن يعيد كل برنمج تعريف هذه الطرائق جميعها، فكما رأينا لا يعرف

البرنمج *"Hello World"* أي منها.

تُستدعى الطريقة *init()* مرةً واحدةً لتهيأة البرنمج، وهي تحتوي عادةً على الرماز الذي نضعه ضمن البناني (*constructor*). لا تمتلك البرنمجات عادةً بواني لأنه لا يمكن ضمان امتلاك البرنمج لبيئة كاملة قبل استدعاء الطريقة *init()*.

تحتاج البرنمجات التي تقوم بعملٍ ما بعد التهيئة (باستثناء الأعمال التي تجيب على أعمال المستخدم) إلى تعريف الطريقة *start()*. تنجز الطريقة *start()* عمل البرنمج أو تستهل مسلكاً أو أكثر لانجاز العمل. سنتحدث عن المسالك لاحقاً في الفصل العاشر ونتناول معالجة الأحداث *events* في الفصل التاسع.

يجب على معظم البرنمجات التي تعرف الطريقة *start()* أن تعرف الطريقة *stop()* أيضاً والتي تعلق تنفيذ البرنمج بحيث لا تستهلك موارد النظام، في الوقت الذي لا يعرض فيه المستخدم صفحة الويب الموافقة.

لا تحتاج معظم البرنمجات إلى تعريف الطريقة *destroy()* لأن الطريقة *stop()* تُستدعى قبلها وتقوم باللازم. إلا أن هذه الطريقة متاحة للبرنمجات التي تحتاج إلى تحرير موارد إضافية.

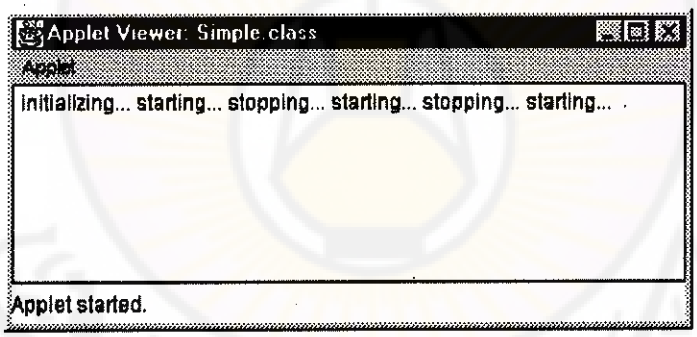
يعرف البرنمج *Simple* شاشته الخاصة وذلك بإعادة تعريف الطريقة *paint()* وهي إحدى طريقتي الرسم التي يمكن للبرنمج إعادة تعريفهما وهما *paint()* و*update()* والموروثتين أصلاً من الصف *Component* ضمن الحزمة *AWT(Abstract Windowing Toolkit)* الذي سنتحدث عنه بالتفصيل لاحقاً.

يرث الصف *Applet* أيضاً مجموعة من الطرائق من الصف *Component* تفيد في معالجة بعض الأحداث مثل *action()* و *mouseDown*، كما يرث طريقة لالتقاط جميع الأحداث وهي *handleEvent()*. للاستجابة لحدث ما يجب على البرنمج إما أن يعيد تعريف الطريقة الخاصة بمعالجته أو أن يعيد تعريف الطريقة *handleEvent()*. على سبيل المثال، إضافة الرماز التالي إلى البرنمج *Simple*، تجعله يستجيب لضغطات الفأرة.

```
import java.awt.Event;
...
   public boolean mouseDown(Event event, int x, int y) {
    addItem("click!... ");
   return true;
  }
```

يرث الصف *Applet* الصف *Container* أي حاوية، وهذا يعني أنه معد لاحتواء *Component* أي أغـــراض واجهـــات اســـتخدام مثل *Button* أو *labels* أو *pop-up list* أو *scrollbars*. وكجمـــيع الحاويات، تستعمل البرمجات متوليات تراتيب (*layout manager*) للتحكم في مواقع المكونات (*Componenent*). (انظر الفصل التاسع).

بعد الانتهاء من البرمج يجب إضافته إلى صفحة ويب لتنفيذه كما هو مبين في الفصل الأول ويوضح الشكل التالي خرج البرمج *Simple*.



الفصل 8 : البيانيات

تؤدي البيانيات دوراً هاماً في النظم الحاسوبية الحديثة، فالصورة تساوي الألوف من الكلمات في قوة تعبيرها.

سنـــبق أن استعملنا، في الفصول السابقة، بعض الطرائق البيانية لرسم مستقيم أو كتابة نص بسيط، إلا ان إمكانات جافا البيانية تتجاوز ذلك بكثير، إذ تزودنا الحزمة *java.awt*[5] بالعديد من من الصفوف البيانية والطرائق.

ســـنتحدث في هذا الفصل عن الألوان ورسم الأشكال البيانية، وسنتحدث في الفصل التالي عن إنشاء واجهات الاستخدام البيانية.

8-1 الصف Graphics

الصـــف *Graphics* هو حجر الأساس لجميع الرسوم وواجهات الاستخدام *(GUI)* في جافا، وهـــو مُعـــرَّف في الحـــزمة *java.awt* من *API* جافا كما هو حال جميع الصفوف المرتبطة بالبيانيات. يرمز *awt* إلى *abstract windowing toolkit*.

يتضمن هذا الصف مجموعة من الطرائق لرسم المستقيمات والأشكال الهندسية المغلقة، ويُمثل كل صف بياني مساحة رسم خاصة. فعلى سبيل المثال يمتلك برنامج جافا غرضاً بيانياً مرتبطاً به ويتأثر باستدعاء أي طريقة من طرائق هذا الغرض. ويعرف هذا الغرض سياقاً بيانياً يسمح بالتعامل مع الفعاليات البيانية على السطح (*surface*) الموافق.

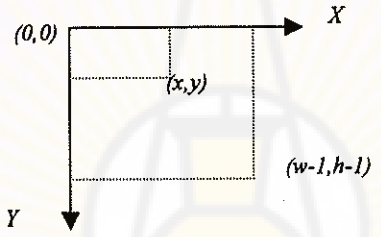
لا يمكـــن للمستثمر إنشاء نسخة من الصف *Graphics* لأن هذا الصف وحيد من أجل كل

[5] نجـــد في إصدار جافا 2، مكتبة بيانية أخرى تسمى *Swing* ذات إمكانات أفضل ولكنها تولد رمازاً أكبر حجماً يحتاج شحنه على الويب زمناً أطول.

حاســـوب، فهو متعلق بالآلة الفيزيائية. يمكن الحصول على الأغراض البيانية المرتبطة بسطح مـــا، إما باستعمال طرائق مثل *getGraphics* (خاصة بالسطح) أو بقبول الغرض *Graphics* كمعامل لإجرائية تُستدعى آلياً مثل *paint*.

8-2 نظام الإحداثيات البياني

ترتبط الرسوم ضمن غرض بياني بجملة إحداثيات حيث تعبر كل نقطة *(x,y)* في هذه الجملة عن بيكسل (*picture element*) وتعد الزاوية العليا اليسارية مبدأً للإحداثيات، ولها الإحداثي *(0,0)* فإذا كانت أبعاد السطح البياني هي *w* عرضاً و*h* ارتفاعاً، تكون إحداثيات أبعد نقطة فيه عن المبدأ هي *(w-1,h-1)*.



الشكل 8-1: جملة الإحداثيات البيانية لجافا

يوضـــح الـــبرنامج التالي استعمال الصف *Graphics* ويطلب الطريقة *setSize* لتحديد أبعاد البريمج بالبكسل.

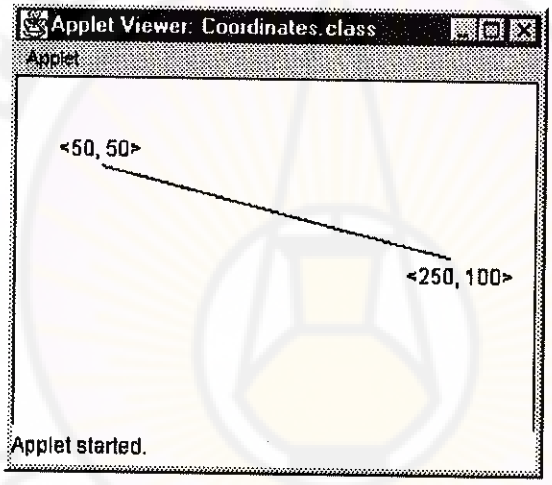
```
import java.applet.Applet;
import java.awt.*;

//show two points on the coordinate system.
public class Coordinates extends Applet {

   public void paint (Graphics page)  {
      setSize (300, 200);
      page.drawRect (0, 0, 299, 299);
      page.drawLine (50, 50, 250, 100);
     page.drawString ("<50, 50>", 25, 45);
    page.drawString ("<250, 100>", 225, 115);
```

```
    } // method paint
} // class coordinates
```

ترسم الطريقة *drawRect* مستطيلاً حول الحافات الخارجية لمساحة الرسم على البرنمج، وتأخذ أربعة وسطاء. يمثل الوسيطان الأول والثاني إحداثيات الزاوية العلوية اليسرى للمستطيل، ويمثل الوسيطان الثالث والرابع إحداثيات الزاوية السفلية اليمنى. سبق أن رأينا كيفية عمل الطريقة *drawString* التي تكتب النص المعطى لها في الإحداثيات المعطاة ويظهر الشكل 8-2 خرج البرنامج.



الشكل 8-2: البرنمج *Coordinates*.

8-3 الألوان

تستعمل الألوان لتحسين مظهر البيانيات على الشاشة. يستعمل مبرمج جافا الصف *Color* المعرف ضمن الحزمة *java.awt* لتعريف الألوان ومعالجتها. يُعرف كل غرض من هذا الصف لوناً وحيداً. ويحتوي الصف على نسخ ساكنة من نفسه، لتزويدنا بعدد من الألوان الأساسية المعرفة سلفاً. كما يحتوي هذا الصف على طرائق لتعريف ومعالجة العديد من الألوان الأخرى.

يمكـــن توصـــيف اللون على أنه مزيج من ثلاثة ألوان أولية هي الأحمر والأخضر والأزرق. تُعـــرف الألوان في جافا (كما هو حال العديد من اللغات الأخرى) بثلاثة أرقام نشير إليها عادةً بالقيمة *RGB* التي ترمز إلى *Red* (أحمر) و*Green* (أخضر) و*Blue* (أزرق). وتقع كل قيمة من هذه القيم بين 0 و 255 وتعبر عن مدى مشاركة اللون الموافق لهذه القيمة في هذا اللون، وتحدد هذه القيم الثلاث اللون النهائي. على سبيل المثال إذا إعطينا قيمة كبيرة للأحمر والأخضر، والقليل من الأزرق، حصلنا على الأصفر.

يُربط عادة بكل مساحة رسم لونٌ أمامي (*foreground color*) وهو اللون المستعمل في رسم الخطوط والأشكال والنصوص. يمكن تغيير هذا اللون باستعمال الطريقة *setColor* والحصول عليه باستعمال الطريقة *getColor*.

8-3-1 الألوان المعرفة سلفاً

يحتوي الصف *Color* على بعض الأغراض الساكنة (*static*) والنهائية (*final*) من نوع *Color* ولأنها *static* يمكن الإشارة إليها باستعمال اسم الصف، ولأنها نهائية (*final*) فلا يمكن إعادة تعريفها وهذه الألوان هي الموضحة في الشكل 8-3.

color	Object	RGB Value
Black	Color.black	0,0,0
Blue	Color.blue	0,0,255
Cyan	Color.cyan	0,255,255
Gray	Color.gray	128,128,128
Dark gray	Color.darkGray	64,64,64
Light gray	Color.lightGray	192,192,192
Green	Color.green	0,255,0
Magenta	Color.magenta	255,0,255
Orange	Color.orange	255,200,0
Pink	Color.pink	255,175,175
Red	Color.red	255,0,0
White	Color.white	255,255,255
yellow	Color.yallow	255,255,0

الشكل 8-3: الألوان المعرفة سلفاً في الصف *Color*.

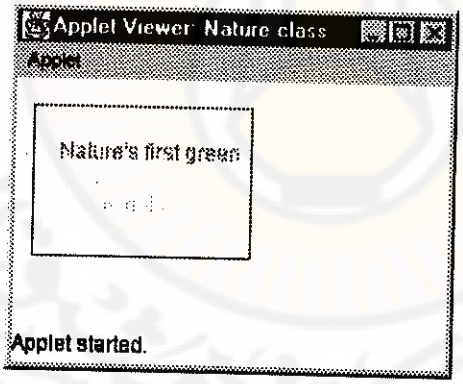
يظهـــر المـــثال التالي المسمى *Nature* استعمال الصف *Color* والطرائق *setColor* لتغيير لون الرســـم (*foreground*) و الطريقة *setBackground* لتغيير لون الخلفية ويظهر الشكل 8-4 خرج هذا البرنامج.

```
import java.applet.Applet;
import java.awt.*;

// Writes two strings in different colors.
public class Nature extends Applet {

    public void paint (Graphics page) {
      setBackground (Color.white) ;
      page.setColor (Color.red) ;
      page.drawRect (10,15,125,85) ;
      page.setColor (Color.green);
      page.drawString ("Nature's first green",25,45) ;
      page.setColor (Color.yellow) ;
      page.drawString ("is gold.",50,75) ;
    } // method paint

} // class Nature
```



الشكل 8-4: البرنامج *Nature*.

8-3-2 تعريف الألوان

يقدم لنا الصف *Color* عدة بوانٍ لتعريف ألوان جديدة وهي:

```
Color(int red, int green, int blue)
Color(float red, float green, float blue)
```

Color(int rgb)

يأخذ الباني الأول ثلاثة أعداد صحيحة تنتمي إلى المجال *[0..255]* تمثل القيمة *RGB*. مثال:

Color brown=new Color(107, 69, 38)

في حين يتوقع الباني الثاني هذه القيم في المجال *[0..1]*، أما الباني الثالث فيأخذ عدداً صحيحاً واحـــداً ويفســـره بأن البتات من 0 إلى 7 تمثل قيمة الأزرق والبيتات من 8 إلى 15 تمثل الأخضـــر والبتات من 16 إلى 23 تمثل الأحمر. تسمح بذلك جافا بتمثيل 256^3 لون أي أكـــثر من 16 مليون لون، ولكن التمييز بين لونين متجاورين من هذه الألوان صعب على العين المجردة.

يمتلك هذا الصف أيضاً طرائق لتحليل اللون من حديد وهي *getRed* و*getGreen* و*getBlue*. تعـــيد الطـــريقة *brighter* لوناً أكثر سطوعاً بقليل من اللون الأصلي في حين تعيد الطريقة *darker* لوناً معتماً أكثر بقليل من اللون الأصلي. وتفيد هاتان الطريقتان في تعريف تدرجات لون ما.

Color brighter_brown= brown.brighter();

3-3-8 النمط XOR

عـــند رســـم شكلين مصمتين متراكبين فإن الشكل الأخير سوف يغطي جزءاً من الشكل الأول. وعندما نرسم باللون نفسه، يندمج الشكلان ولا يمكننا التمييز بين حافاتهما. يمكن أن يجـــري الرســـم بالنمط الطبيعي أوالنمط *XOR* . يسمح النمط *XOR* ببقاء حافات الشكلين متميزة بعضها عن بعض، ويمكن تفعيله باستعمال الطريقة *setXORMode* المعرفة ضمن الصف *Graphics* وهي طريقة تقبل وسيطاً من نمط *Color* يحدد لون النمط *XOR*. عند رسم بيكسل بالنمط الطبيعي فإنه يأخذ لون الرسم الأمامي *foreground* أما عند استعمال النمط *XOR* فإن لونه يُحَدَّد بدلالة اللون الحالي للبيكسل ولون النمط *XOR* وبذلك يبقى لون أي شكل مغايراً للون الذي رُسم عليه.

آثار النمط *XOR* عكوسة، فلو رُسم بيكسل ما مرتين في النمط *XOR* لعاد لون البيكسل إلى

لونه الأصلي ولذلك فإن إعادة رسم شكل ما بالنمط *XOR* تمحي الشكل الأول.

يوضح المثال التالي استعمال النمط *XOR* في الرسم والمحو.

```
import java.awt.*;
import java.applet.*;

// Demonstrates the use of XOR Mode
public class XOR_Demo extends Applet {

    public void paint (Graphics page) {

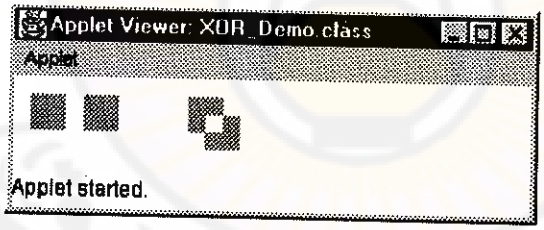
      page.setXORMode (Color.gray) ;

      page.fillRect (10,10,20,20);
      page.fillRect (40,10,20,20);

      page.fillRect (100,10,20,20);
      page.fillRect (110,20,20,20);
      page.fillRect (140,10,20,20);
      page.fillRect (140,10,20,20);

    } // method paint

} // class XOR_Demo
```



الشكل 8-5: البرنامج *XOR_Demo*

8-4 أشكال الرسم

تسمح جافا بواسطة الصف *Graphics* رسم المستقيمات والقطوع و المستطيلات والأقواس والمضـــلعات المغلقة والمفتوحة. تُعامل الدائرة كحالة خاصة من القطوع، كما يعامل المربع كحالة خاصة من المستطيل. تتضمن المضلعات العديد من الأنواع الخاصة من الأشكال من

مثلـــثات ومسدســـات، لأن المضلعات يمكن أن تمتلك أي عدد من الأضلاع وبأي طول. المضـــلعات المفـــتوحة التي تُسمى أيضاً بمتعددات المستقيمات، هي سلسلة من المستقيمات المتصل بعضها ببعض.

يسمح الصف *Graphics* أيضاً بتحديد خواص الشكل: فارغ أو مصمت. فإذا كان فارغاً فلـــن يُـــرى منه إلا حافاته ويكون شفافاً تُرى الأشكال الواقعة خلفه. أما إذا كان مصمتاً فيمتلئ ما بين أضلاعه ويخفي ما خلفه. يُستعمل لون الواجهة الأمامية (*foreground color*) لرسم حافات الشكل وداخله إذا كان مصمتاً.

تُرسم المستقيمات باستعمال الطريقة *drawLine* (التي سبق أن رأيناها). لا يمكن للمستقيمات أن تكون مصمتة، ولا يمكن تحديد عرضها الذي يساوي دوماً بيكسل واحد. يمكن محاكاة مســـتقيم ثخين برسم مستقيمين أو أكثر جنباً إلى جنب أو برسم مستطيل مصمت إذا كان المســـتقيم شاقولياً أو أفقياً. يُفَصِّل المقطع التالي طرائق رسم القطوع والمستطيلات والأقواس والمضلعات ومتعددات المستقيمات.

1-4-8 القطوع (*ovals*)

يمكـــن تعريف القطع بالمستطيل المغلف له. الوسطاء المعطاة لطرائق رسم القطوع في الصف *Graphics* هـــي الـــزاوية العليا اليسرى للمستطيل المغلف وكذلك عرض وطول المستطيل بالبيكسل.

يحتوي الصف *Graphics* على طريقتين لرسم القطوع:

- drawOval(int x,int y, int width, int height)
- fillOval(int x,int y, int width, int height)

وكما ذكرنا فـــ *x* و*y* هما إحداثيا الزاوية العليا اليسرى للمستطيل المغلف و*width* و*height* هما عرض وطول المستطيل المغلف بالبيكسل، وأضلاع هذا المستطيل موازية دوماً للمحورين *x,y* فلا يمكن رسم قطوع مائلة. تستعمل الطريقة *fillOval* لرسم قطع مصمت. يختبر البرنامج التالي الطريقتين *drawOval* و*fillOval* ويظهر الشكل 8-6 خرج هذا البرنامج:

يختـــبر البرنامج التالي *drawRoundRect* و *fillRoundRect* كما يظهر الشكل 8-9 خرج هذا البرنامج.

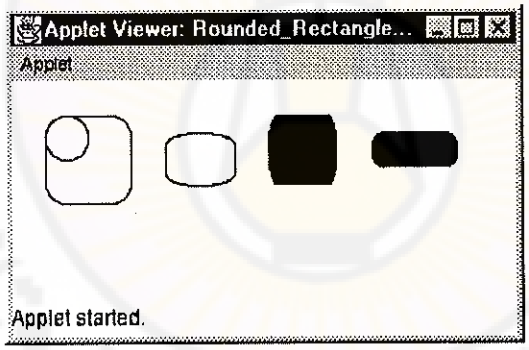
```
import java.applet.Applet;
import java.awt.*;

// Draws various types of rounded rectangles.
public class Rounded_Rectangles extends Applet {

   public void paint (Graphics page) {
      page.drawRoundRect (20, 20, 50, 50, 25, 25);
      page.drawOval (20, 20, 25, 25); // oval defines the rounded corner
      page.drawRoundRect (90, 30, 40, 30, 40, 15);

      page.fillRoundRect (150, 20, 40, 40, 10, 30);
      page.fillRoundRect (210, 30, 50, 20, 15, 15);
   } // method paint

} // class Rounded_Rectangles
```



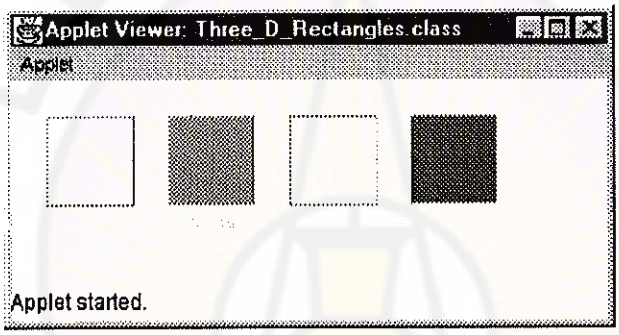
الشكل 8-9: البرنامج *Rounded_Rectangles*.

ترسم الطريقتان *draw3DRect* و*fill3DRect* مستطيلاً يتمتع بتأثير بسيط ثلاثي الأبعاد. يتحقق هذا التأثير بإضافة ظلال بسيطة على ضلعين متجاورين. يحدد الوسيط الأخير حالة المستطيل مـــن حيث ظهوره مرتفعاً (قيمة الوسيط هي *true*) أو منخفضاً (قيمة الوسيط هي *false*). يختبر البرنامج التالي هاتين الطريقتين ويُظهر الشكل 8-10 خرج البرنامج:

```
import java.applet.Applet;
import java.awt.*;
```

```
// Draw various 3D rectangles.
public class Three_D_Rectangles extends Applet {

    public void paint (Graphics page) {
        setBackground (Color.white);
        setForeground (Color.gray);
        page.draw3DRect (20, 20, 50, 50, true);
        page.fill3DRect (90, 20, 50, 50, true);
        page.draw3DRect (160, 20, 50, 50, false);
        page.fill3DRect (230, 20, 50, 50, false);
      } // method paint
} // class Three_D_Rectangles
```



الشكل 8-10: البرنامج *Three_D_Rectangles*.

3-4-8 الأقواس *Arcs*

القوس هو جزءٌ من قطع. يوفر الصف *Graphics* الطريقتين التاليتين لرسم الأقواس:

```
drawArc( int x, int y, int width, int height, int start_angle, int arc_angle)
fillArc( int x, int y, int width, int height, int start_angle, int arc_angle)
```

يطمس القوس المصمت المساحة المحصورة بين القوس نفسه، وبين المستقيمين المرسومين من طـــرفي القوس إلى مركز القطع المعرَّف للقوس. تمتلك كلتا الطريقتين الوسطاء نفسها فيعبر الوسيطان الأول والثاني عن الزاوية العلوية اليسرى للقطع، والثالث والرابع عن عرض القطع وارتفاعه. يعبر الوسيط الخامس عن زاوية البداية للقطع، في حين يعبر الوسيط السادس عن زاوية القوس (زاوية النهاية= زاوية البداية + زاوية القوس). تقاس الزوايا بالدرجة، ويكون

المستقيم الأفقي المار بمركز القطع هو مبدأ الإحداثيات أي الزاوية 0.

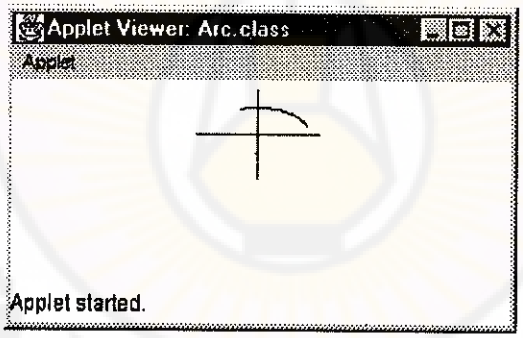
القوس المعرف باستدعاء *drawArc* في الشكل 7-13 هو الجزء من القطع الممتد من الزاوية 20 حتى الزاوية 110 ويغطي امتداد القوس 90 درجة. يرسم البرنامج التالي هذا القوس ويظهر الشكل *8-11* خرج البرنامج.

```
import java.applet.Applet;
import java.awt.*;

// Draw an arc.
public class Arc extends Applet {
    public void paint (Graphics page) {

        page.drawLine (140, 5, 140, 55);
        page.drawLine (105, 30, 175, 30);
       page.drawArc (110, 15, 60, 30, 20, 90);
  } // method paint

} // class Arc
```



الشكل 8-11: البرنامج *Arc*.

تقاس الزوايا الموجبة بعكس عقارب الساعة وتساوي الزاوية 45 درجة الزاوية -315 ويكون بذلك لكل قوس أربعة تمثيلات ممكنة.

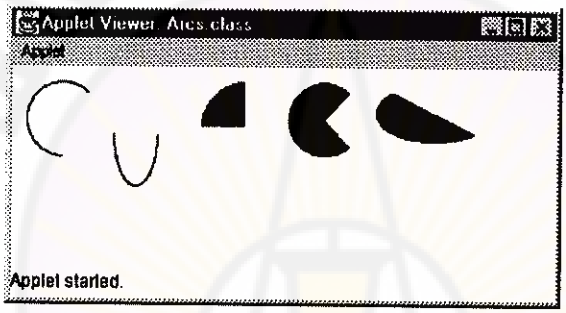
يختبر البرنامج التالي الطريقتين *drawArc* و *fillArc* باستعمال زوايا بدء موجبة وسالبة، وكذلك زوايا أقواس موجبة وسالبة. ويظهر الشكل 8-12 خرج هذا البرنامج.

```
import java.applet.Applet;
import java.awt.*;
```

```
// draw various arcs.
public class Arcs extends Applet

   public void paint (Graphics page)
      page.drawArc (10, 10, 50, 50, 45, 225);
      page.drawArc (70, 10, 30, 70, -180, 180);
      page.fillArc (130, 10, 60, 60, -180, -90);
     page.fillArc (190, 10, 50, 50, 45, 270);
     page.fillArc (250, 10, 80, 40, -225, 180);
   } // method paint

} // class Arcs
```



الشكل 8-12: البرنامج *Arcs*.

8-4-4 المضلعات

يُعرَّف المضلع في حافا بسلسلة من النقاط <x,y> تدل على نهايات القطع المستقيمة المشكلة لـــه. تحتوي الحزمة *java.awt* صفاً يسمى *Polygon* يُستعمل لتوصيف المضلع المراد رسمه، في حين أن طريقة الرسم ذاتها معرفة في الصف *Graphics*. طرائق رسم المضلع هي:

```
drawPolygon( int[] xpoints, int[] ypoints, int numpoints)
drawPolygon( Polygon poly)
fillPolygon( int[] xpoints, int[] ypoints, int numpoints)
fillPolygon( Polygon poly)
```

يعـــبر الوســـيط *xpoints* عـــن الإحداثيات *x* لرؤوس المضلع، ويعبر الوسيط *ypoints* عن الإحداثيات *y* لها. تأخذ النسختان المُحَمَّلتان (*overloaded*) من *drawPolygon* و *fillPolygon* وسيطاً واحداً هو مضلع يحتوي سلفاً على هذه النقاط، حيث تستعمل الطريقة *addPoint* من

الصف *Polygon* لإضافة النقاط.

المضلع دائماً مغلق، فإذا لم تكن إحداثيات آخر نقطة مساوية لإحداثيات أول نقطة، فإنهما سيوصلان آلياً عند رسم الشكل.

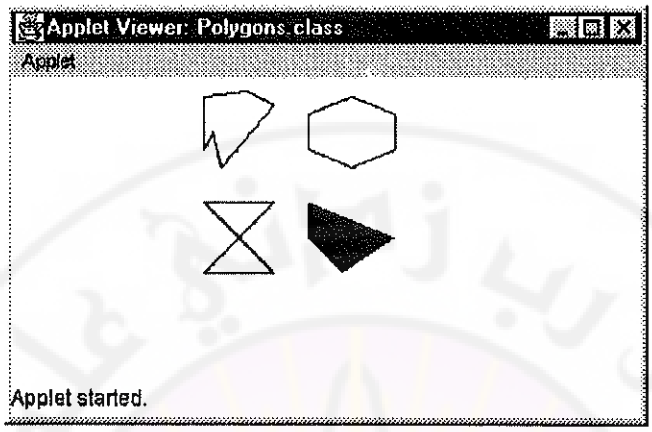
يختبر البرنامج التالي الطريقتين *fillPolygon* و *drawPolygon*، ويظهر الشكل 8-13 خرج هذا البرنامج.

```
import java.applet.Applet;
import java.awt.*;

// Draws various polygons.
public class Polygons extends Applet {
   private int[] xset1 = {110, 110, 115, 120, 150, 135};
   private int[] yset1 = {10, 40, 30, 50, 15, 7};
   private int[] xset2 = {195, 170, 170, 195, 220, 220};
  private int[] yset2 = {10, 20, 40, 50, 40, 20};
 private int[] xset3 = {110, 150, 110, 150, 110};
 private int[] yset3 = {70, 70, 110, 110, 70};
 private Polygon poly = new Polygon();

 public void paint (Graphics page) {
    page.drawPolygon (xset1, yset1, 6);
    page.drawPolygon (xset2, yset2, 6);
    page.drawPolygon (xset3, yset3, 5);
    poly.addPoint (170, 70);
    poly.addPoint (170, 90);
    poly.addPoint (190, 110);
    poly.addPoint (220, 90);
    page.fillPolygon (poly); // closed automatically
 } // method paint

} // class Polygons
```



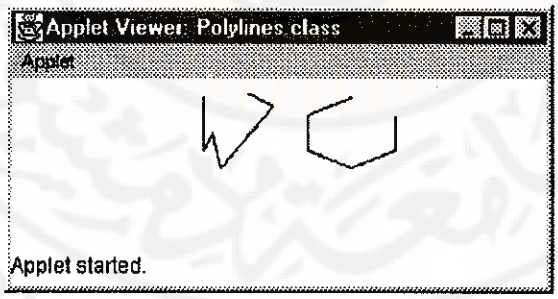
الشكل 8-13: البرنمج *Polygons*.

8-4-5 متعدد المستقيمات

تشـــبه مـــتعددات المستقيمات المضلعات لكنها ليست بالضرورة أشكالاً مغلقة. وتُستعمل الطريقة التالية لرسمها:

drawPolyline(int[] xpoints, int[] ypoints,int numpoints)

يمثل الوسيط الأول الإحداثيات x لرؤوس متعدد المستقيمات، ويمثل الوسيط الثاني الإحداثيات y لها، في حين يمثل الوسيط الثالث عدد هذه الرؤوس.



الشكل 8-14: البرنمج *Polylines*.

يختـــبر البرنامج التالي عدة استدعاءات للطريقة *drawPolyline* ويُظهر الشكل 8-14 خرج

هذا البرنامج:

```
import java.applet.Applet;
import java.awt.*;

// Draws various polylines.
public class Polylines extends Applet {
   private int[] xset1 = {110, 110, 115, 120, 150, 135};
   private int[] yset1 = {10, 40, 30, 50, 15, 7};
   private int[] xset2 = {195, 170, 170, 195, 220, 220};
  private int[] yset2 = {10, 20, 40, 50, 40, 20};

 public void paint (Graphics page) {
    page.drawPolyline (xset1, yset1, 6);
    page.drawPolyline (xset2, yset2, 6);
} // method paint

} // class Polylines
```

لاحـــظ أن الأشكال الناتجة هي نفسها المرسومة في الشكل 8-13 ولكنها أغلقت آلياً في الشكل 8-13 لأنها تمثل مضلعاً.

6-4-8 البنوط Fonts

يقـــدم لنا الصف *Font* عدة طرائق لتحديد البنط المستعمل في برنامج ما. يحدد البنط شكل الحروف أثناء رسمها أو كتابتها. تمتلك نظم التشغيل المختلفة بنوطاً مختلفة، ولذلك، من الهام معرفة أنواع البنوط المتوفرة في نظام التشغيل الذي نستعمله.

يســـتعمل البرنامج التالي الطريقة الساكنة *getFontList* المعرفة في الصف *Toolkit* والتي تعيد مصفوفة خطية من السلاسل المحرفية *Strings* تحتوي على البنوط المتوفرة:

```
import java.awt.*;

// Lists the available fonts.
public class Font_Lister {

   public static void main (String[] args) {
      String[] list;
      list = Toolkit.getDefaultToolkit().getFontList();
```

```
    for (int item = 0; item < list.length; item++)
      System.out.println (list[item]);
    } // method main

} // clas Font_Lister
```

يمكن أن يكون خرج البرنامج شيئاً من القبيل التالي:

```
Dialog
SansSerif
Serif
Monospaced
Helvetica
TimesRoman
Courier
DialogInput
ZapfDingbats
```

يقدم لنا الصف *Font* بانياً يسمح للمبرمج بإنشاء بنط معتمداً على أحد بنوط النظام، ولكن بنمط وقياس خاصين. هذا الباني هو

```
Font( String name, int style, int size)
```

يجب أن يعبر الوسيط *name* عن اسم صحيح لأحد بنوط النظام ويعبر *Style* عن النمط المرغوب لهذا البنط (عادي، غامق، مائل) ويمكن استعمال أحد الثوابت الرقمية *PLAIN, BOLD, ITALIC* المعرفة في الصف *Font*. يمكن مراكبة هذه الثوابت باستعمال عملية الجمع (+). يؤدي، على سبيل المثال، تمرير *Font.ITALIC+Font.BOLD* إلى إنشاء بنط مائل وغامق في آن واحد. يدل الوسيط *size* على حجم كل نقطة من البنط، وبالطبع كلما ازداد هذا الوسيط كان البنط أكبر.

فيما يلي مثال ينشئ عدة بنوط، ويحدد البنط الفعال باستعمال الطريقة *setFont*. تستعمل الطريقة *drawString* البنط الفعال دائماً لترسم السلسلة المطلوبة ويوضح الشكل 7-19 خرج هذا البرنامج

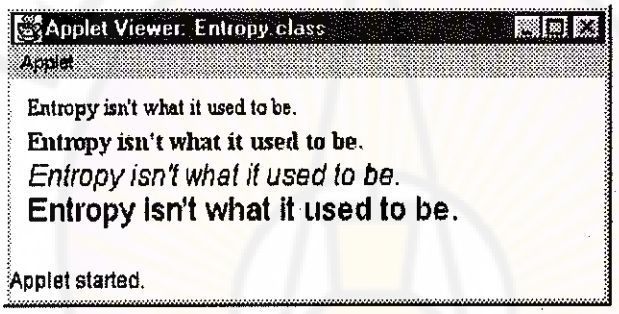
```
import java.applet.Applet;
import java.awt.*;

// Shows different fonts with different styles and sizes.
public class Entropy extends Applet {

    private String quote = "Entropy isn't what it used to be.";
```

```
    public void paint (Graphics page) {
        page.setFont (new Font ("TimesRoman", Font.PLAIN, 12));
        page.drawString (quote, 10, 20);
        page.setFont (new Font ("TimesRoman", Font.BOLD, 14));
        page.drawString (quote, 10, 40);
        page.setFont (new Font ("Helvetica", Font.ITALIC, 16));
        page.drawString (quote, 10, 60);
        page.setFont (new Font ("Helvetica ", Font.PLAIN, 18));
        page.drawString (quote, 10, 80);
    } // method paint

} // class Entropy
```



الشكل 7-19: البرنامج *Entropy*.

7-4-8 مثال: الكرة الوثّابة *Bouncing Ball*

لنتناول مثالاً يستعمل طرائق بيانية لتحقيق إحياء بسيط. يحقق البرنامج التالي صفاً يمثل كرة مربوطة بحبل مطاطي. ويحاكي البرنامج ما يحدث عندما نشد الكرة ثم نتركها.

تُمثل الكرة بدائرة ويمثل الحبل بمستقيم. يُوَلَّد تأثير الوثب بمحو الكرة والحبل المرسومين سابقاً باستعمال تقنية النمط *XOR* وإعادة رسم الكرة مع حبل أقصر أو أطول. تكرار هذا العمل يـــؤدي إلى الحصول على تأثير الوثب. لاحظ أننا نستعمل عدة ثوابت لإبطاء تنفيذ البرنامج وهذا ضروري للحصول على تأثير طبيعي.

```
import java.applet.Applet;
import java.awt.*;
```

```
// Simulates a ball on a rubber band.
public class Bouncing_Ball extends Applet {

   private final int PAUSE = 100000;
   private final int SIZE = 300;

  private Ball ball = new Ball(150, 10, 250, 200);
  private Graphics page;

  public void init() {
    setVisible(true);
    setSize(SIZE, SIZE);
    page = getGraphics();
   page.setXORMode (getBackground () ) ;
  } // method init

    public void start () {
        for (int pause = 1; pause< PAUSE; pause++) ;
        while (ball.moving () )
              ball.bounce (page) ;
     } // method start

} // class Bouncing_Ball

// Draws the ball and simulates its movement.
class Ball {

    private final int MOVE =2;
    private final float DISTANCE = 0.97f ;
    private final int size = 20;
    private final int PAUSE = 1000000 ;

    private int x;
    private int start_y;
    private int end_y;
    private int length;

    private boolean moving_up = true;

    public Ball (int new_x, int new_start_y,
          int new_end_y, int new_length) {
       x = new_x ;
       start_y = new_start_y ;
       end_y = new_end_y;
       length = new_length ;
```

```
    } // constuctor Ball

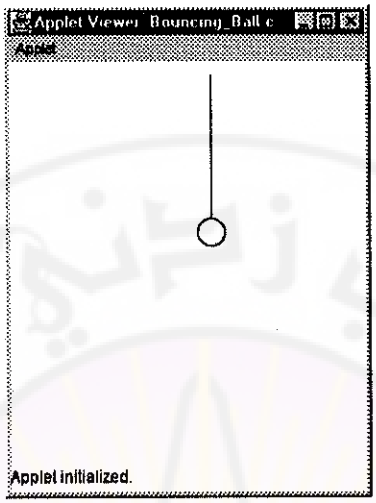
    private void move () {
        if (moving_up) {
          end_y = end_y -MOVE ;
        } else {
          end_y = end_y +MOVE;
        }
    } // method move

    private void draw_ball (Graphics page ) {
        page.drawOval (x-(size/2), end_y, size, size);
        page.drawLine (x, start_y, x, end_y) ;
    } // draw_ball

    public boolean moving () {
       return length != 0 ;
    } // method moving

    public void bounce (Graphics page) {
      for (int count = 1; count < length; count += MOVE) {
          draw_ball (page) ;
          for (int pause = 1; pause < PAUSE/length; pause++ ) ;
          draw_ball (page) ;
          move () ;
        }
        moving_up = !moving_up;
        length = (int) (DISTANCE * length) ;
    } // method bounce

} // class Ball
```



الشكل 8-15: لقطة للبرنمج *Bouncing_Ball*.

Damascus University

الفصل 9 : واجهات الاستخدام البيانية GUI

تتضـــمن معظم البرمجيات الحديثة واجهة استخدام بيانية *GUI* (*Graphical User Interface*) توفـــر التفاعل بين المستخدم والبرنامج. يشرح هذا الفصل تقنيات إنشاء واجهات استخدام بيانية باستعمال الواجهة البرمجية لتطبيقات جافا *API* (*Application Programming Interface*). كما يهتم هذا الفصل بوصف البرمجة المقُوْدة بالأحداث *event-driven programming* وهي فكرة هامة جداً لتحقيق الواجهات البيانية. نفحص في هذا الفصل العديد من مكونات *GUI* ومتوليات الترتيب (*Layout manager*) التي تحكم وضعية مكونات *GUI* على الشاشة.

9-1 عناصر GUI

تتكون *GUI* من عدة أجزاء. تزودنا جافا بعدة صفوف مشتقة من الصف *Component* لبناء *GUIs* ونشـــير عـــادة إلى هرمـــية هذا الصف بمكونات *GUI*. يصف الشكل 9-1 بعض مكونات *GUI* الشائعة.

المكونة	وصف
Label	تُظهر سطراً نصياً
TextField	تُظهر سطراً نصياً قابلاً للتعديل.
TextArea	تُظهر عدة أسطر نصية قابلة للتعديل.
List	تُظهر عدة عناصر يمكن انتقاؤها.
PushButton	زر يُستعمل للقيام بعملٍ ما عند الضغط عليه.
Choicebuttons	زر يُظهر قائمة من الخيارات عند الضغط عليه، ويظهر الخيار المنتقى إلى جانب الزر.

CheckBoxButtons	زر يأخذ حالتين مُحَدَّد/غير مُحَدَّد.
RadioButtons	مجموعة أزرار تأخذ إحدى حالتين. لا يمكن انتقاء اثنتين منها في آن واحد.
Scrollbar	شريط إزاحة يستعمل لإظهار وضعية ما أو قيمة نسبية.
Scroll panel	منطقة تمتلك شريط إزاحة.
PullDownMenu	مجموعة خيارات يمكن اختيار أحدها وتوجد في أعلى نافذة.
PopUpMenu	مجموعة خيارات يمكن اختيار أحدها وتوجد في مكان يحدده المستخدم.
Canvas	مساحة عامة تُستعمل لإنشاء مكونات *GUI* خاصة.

الشكل 9-1: بعض مكونات *GUI*

يمكن ضم المكونات معاً باستعمال مكونات *GUI* خاصة تعرف بالحاويات *Containers*. أما ترتيب المكونات ضمن الحاويات فهو مهمة متولي الترتيب (*Layout Manager*) الذي يختار مكـــان كل مكونة اعتماداً على مجموعة من القيود. ننشئ هيكلية كاملة بمداخلة المكونات والحاويـــات وباختيار متولي الترتيب المناسب، وتستعمل هذه الهيكلية في التمثيل المرئي لـــ *GUI* البرنامج.

تســـمح واجهـــة الاستخدام البيانية للمستثمر بتنفيذ عدة أعمال ممكنة. يمكن للمستثمر أن يضـــغط عـــلى زر، أو يُدخل حرفاً ضمن حقل نصي، أو يحرك شريط ازاحة (*Scrollbar*) اعتماداً على مجموعة من الخيارات المتاحة من قبل البرنامج. ويجب على البرنامج أن يتحسس هذه الأعمال وأن يستجيب بطريقة مناسبة.

تولـــد مكونـــات *GUI* أحداثـــاً تشير إلى أن عملاً ما قد حدث. تُفيد تقنية البرمجة المقودة بالأحداث في بناء برمجيات تعمل بالاستجابة للأحداث فور حدوثها. يوصف الشكل 9-2 بعـــض الأحداث الشائعة المستعملة في جافا. يجب على البرنامج التقاط الأحداث ومعالجتها بطريقة مناسبة.

الحدث	وصف
Adjustment	يُولّد نتيجة حركة شريط إزاحة ويشير إلى حدوث تعديل في قيمة ما.
Component	يُولّد نتيجة تغيير حجم المكونة أو تحريكها.
Focus	يولد نتيجة لتغيير في التركيز كأن تفقد مكونة ما التركيز أو تخسره.
Item	يولد نتيجة اختيار عنصر من قائمة أو مجموعة، ويشير إلى أن حالة العنصر قد تغيرت كأن يُنتقى *checkbox*.
Key	يولد نتيجةً لعمل لوحة المفاتيح كأن يُضغط مفتاح ما أو يترك.
Mouse	يولد من قبل مفاتيح الفأرة وحركتها.
Text	يولد من قبل غرض نصي ويدل على أن النص الممثل بالغرض قد تغير.
Window	يولد نتيجة للتأثير على نافذة من إغلاق وتصغير.

الشكل 9-2: أحداث *GUI*

9-2 البرمجة المقودة بالأحداث

لنتناول مثالاً على واجهة بيانية بسيطة. يحتوي البرمج التالي غرضين *TextField* و *Label*.

عـــندما تُدخل الأحرف ضمن حقل النص و تُلحق بالمحرف *newline* فإن محتوى حقل النص يُنسخ ضمن *Label*. و يُظهر الشكل 9-3 هذا البرمج أثناء العمل.

يحقق الرماز التالي البرمج *Mimic*. حيث تهيئ الطريقة *init* واجهة الاستخدام البيانية للبرمج. تـــنفذ الطريقة *update_ label* الهدف الأساسي للبرمج و هو تحديث الغرض *Label* اعتماداً عـــلى النص الذي أدخله المستخدم ضمن حقل النص. لاحظ أنه لا يوجد استدعاء صريح للطريقة *update_Label* ضمن البرمج، و إنما تُستدعى هذه الطريقة عند ورود حدث ما.

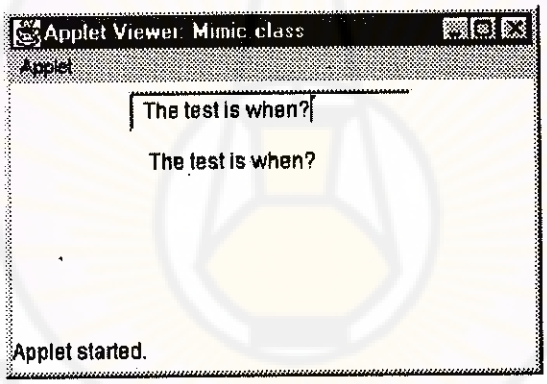
في حالتنا هذه يُولّد الحدث *ActionEvent* من قبل الحقل النصي عندما يُدخِل المستخدم محرف السطر الجديد *newline* .

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

// Demonstrates a simple component and an action event
public class Mimic extends Applet {

    Mimic_GUI gui = new Mimic_GUI (this) ;
    public void init () {
      gui.init () ;
    } // method init

    public void update_label () {
      System.out.println ("Action") ;
      gui.update_label (gui.get_quote()) ;
    } // method set_uppercase
} // class Mimic
```



الشكل 9-3 خرج البرنامج *Mimic*

يستعمل البرنامج صفين إضافيين لإكمال الهدف و هما *Mimic_GUI* و *Mimic_Action_Listener*. يعرف الصف *Mimic_GUI* أجزاءً عديدة من واجهة الاستخدام البيانية ويرتبها بطريقة مناسبة.

يتكون البرنامج من مكونتين *GUI* وهما *Text Field* و *Label* . يُربط كلٌ منهما إلى البرنامج باستعمال الطريقة *add* للبرنامج.

يربط الصف *Mimic_ GUI* الغرض *listener* من الصف *Mimic_ Action_ Listener* إلى الغرض

quote من الصف *TextField*. ويجري هذا الربط باستعمال الطريقة *addActionListener* من الصف *TextField*. عندما يكتشف الغرض *quote* حدوث *ActionEvent* (مثل إدخال سطر جديد في حقل النص) فإنه يستدعي جميع المتنصتات (*listener*) المسجلة باستعمال *addActionListener* وفي حالتنا هذه لم يُسجل إلا متنصت وحيد.

```
// Constructs the graphical user interface
class Mimic_GUI {

    private Label label = new Label("No news is good news.") ;
    private TextField quote = new TextField(20) ;

    private Mimic applet;
    private Mimic_Action_Listener listener ;

    public Mimic_GUI (Mimic mimic_applet) {
      applet = mimic_applet ;
      listener = new Mimic_Action_Listener (applet);
    } // constructor Mimic_GUI

    public void init() {
      applet.add (quote) ;
      applet.add (label) ;

      applet.resize (250,100);

      //Add all associated listeners
      quote.addActionListener (listener) ;
    } // method init

    public void update_label (String message) {
      label.setText (message) ;
    } // method update_label

    public String get_quote() {
      return quote.getText() ;
    } // method get_quote
} // class Mimic_GUI
```

يجب على صفوف التنصت أن توفر طرائق معينة. تُجبَر الصفوف على توفير هذه الطرائق عن طريق طلب تحقيق بعض الواجهات (*interface*) الخاصة، وفي هذه الحالة يحقق الصف *Mimic_Action_Listener* الواجهة *ActionListener*.

تطلب هذه الواجهة من المتنصت أن يوفر الطريقة *actionPerformed*. عندما يتحسس غرض ما، مرتبط بمتنصت ما، حدثاً فإنه يستدعي آلياً الطريقة *actionPerformed* المرتبطة بالمتنصت.

يقوم المتنصت *Mimic* باستدعاء الطريقة *update_label* للبرمج .

```
// Handles the action events for Mimic
class Mimic_Action_Listener implements ActionListener{

    private Mimic applet ;

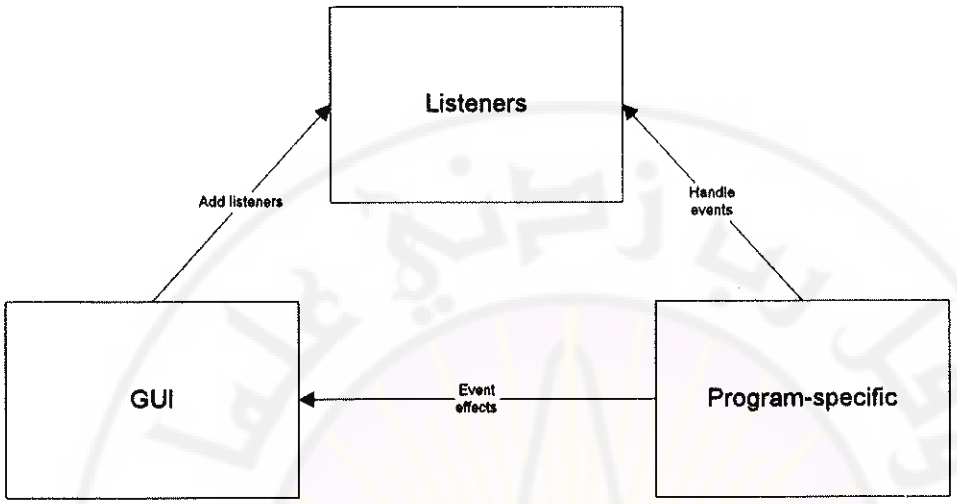
    public Mimic_Action_Listener (Mimic listening_applet) {
      applet = listening_applet ;
    } // constructor Mimic_Action_Listener

    public void actionPerformed (ActionEvent event) {
      applet.update_label();
    } // method actionPerFormed

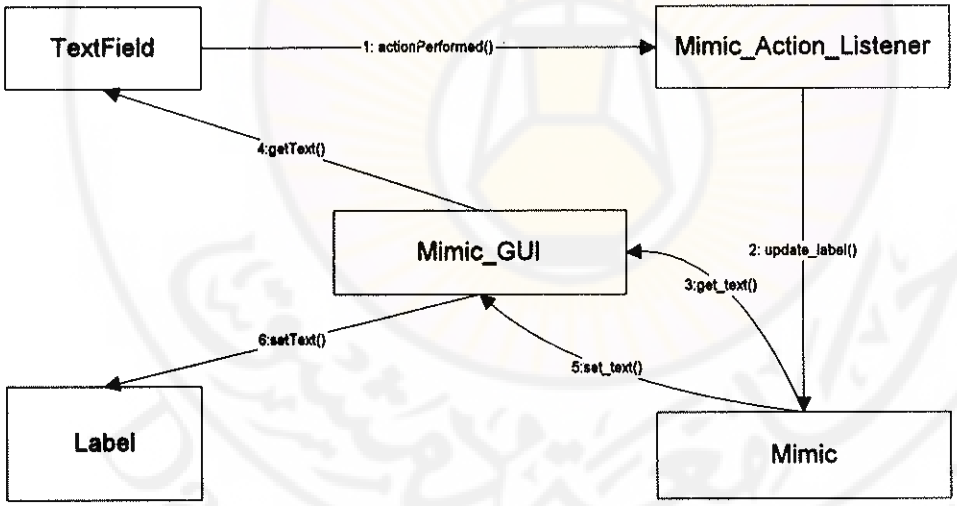
} // class Mimic_Action_Listener
```

لنميِّز أجـــزاء المـــثال *Mimic* التي يمكن استعمالها في برامج *GUI* أخرى. يقوم النموذج الأساسي لبرامج *GUI* بتقسيم البرنامج إلى ثلاثة أجزاء مختلفة .

- مكونات *GUI* المسؤولة عن بناء واجهة المستخدم وربط الأحداث إلى المتنصتات.
- الجزء المتنصت الذي يحقق الواجهات (*interface*) المناسبة لنمط الحدث المقصود.
- الرماز الذي يحقق أعمال البرنامج المرتبطة بالحدث.



الشكل 9-4: نموذج البرنامج *GUI*



الشكل 9-5: سيناريو *Mimic*

9-2-1 واجهات الأحداث

لكـــل نمـــط مـــن الأحداث واجهة تنصت خاصة به. يظهر الشكل 9-6 جدولاً كاملاً

بـــالأحداث وواجهات التنصت الخاصة بها، والطرائق المرتبطة بها والصفوف التي تتقبل هذه الأحداث.

لننظر إلى برنمج يسمح للمستثمر بمعرفة كيفية توالد الأحداث انطلاقاً من أعمال المستخدم. يستعمل هذا البرنمج متنصتاً يحقق العديد من واجهات التنصت. يُعلمنا البرنامج بسلسلة من أحداث الفأرة ولوحة المفاتيح، وذلك بطباعة رسائل تصف كل حدث عند حدوثه، وتظهر أيضاً عدد الأحداث وتظهر المعلومات كلما تغيرت . كما هو مبين في الشكل 9-7.

يســـتعمل الـــبرنامج صفين، الأول هو الصف *Events* المشتق من *Applet* ويتضمن رماز البرنامج ورماز واجهة الاستخدام البيانية. يُنشئ هذا الصف مكونات *GUI* ويرتبها ويربطها بالمتنصتات المناسبة. يسمى الصف الثاني *Universal_ Listener* وهو صف وحيد يحقق العديد من واجهات التنصت المختلفة. تسجل المعلومات الناتجة عن الأحداث في سجل (*log*) يُحقّق باستعمال *TextArea*.

صف الحدث	واجهة التنصت	طرائق الربط	منشأ من قبل صف
ActionEvent	*interface ActionListener* *actionPerformed(ActionEvent)*	*addActionListener* *removeActionListener*	*Button List* *TextField MenuItem*
WindowEvent	*interface WindowListener* *windowClosing(WindowEvent)* *windowOpened(WindowEvent)* *windowIconified(WindowEvent)* *windowClosedWindowEvent)* *windowActivated(WindowEvent)* *windowDeactivated(WindowEvent)*	*addWindowListener*	*Window*
Component-Event	*interface ComponentListener* *ComponentMoved(ComponentEvent)* *ComponentHidden(ComponentEvent)* *ComponentResized(ComponentEvent)* *ComponentShown(ComponentEvent)*	*addComponentListener* *removeComponentListener*	*Component*
Adjustment-Event	*interface AdjustmentListener* *adjustmentValueChanged(AdjustedEvent)*	*addAdjustmentListener* *removeAdjustmentListener*	*Scrollbar*
ItemEvent	*interface ItemListener* *itemStateChanged(ItemEvent)*	*addItemListener* *removeItemListener*	*Checkbox* *Checkbox-MenuItem* *Choise* *List*
MouseEvent	*interface MouseListener* *mouseDragged(MouseEvent)* *mouseMoved(MouseEvent)* *interfaceMouse((MouseEvent)* *mousePressed(MouseEvent)* *mouseReleased(MouseEvent)*	*addMouseListener* *removeMouseListener* *addMouseMotionListener* *removeMouseMotionListener*	*Component*

		mouseEntered(MouseEvent) mouseExited(MouseEvent) mouseClicked(MouseEvent)	
Component	addKeyListener removeKeyListener	interface KeyListener keyPressed(KeyEvent) keyReleased(KeyEvent) keyTyped(KeyEvent)	KeyEvent
Component	addFocusListener removeFocusListener	interface FocusListener focusGained(FocusEvent) focusLost(FocusEvent)	FocusEvent
TextComponent	addTextListener removeTextListener	interface TextListener textValueChanged(TextEvent)	TextEvent

الشكل 9-6: واجهات التنصت والصفوف المرتبطة بها.

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

//Interactively shows when and how events are generated
public class Events extends Applet {

    private TextArea log = new TextArea (10,65) ;
    private int count = 0;
    private Label count_label = new Label (" ", Label.CENTER) ;

    private Universal_Listener listener = new
      Universal_Listener (log, count_label) ;
    public void init () {
       add ("North", new Label("Event Logging", Label.CENTER) ) ;
       add ("Center", log) ;
       add ("South", count_label) ;

       addComponentListener(listener) ;
       log.addFocusListener(listener) ;
       log.addKeyListener(listener) ;
       log.addMouseListener(listener) ;
      setSize (500,300) ;
    } // method init
} // class Events

class Universal_Listener implements ComponentListener,
      MouseMotionListener, MouseListener, KeyListener,
      FocusListener {

    private TextArea log ;
    private Label count_label;
    private String count_text = "Number of Events:" ;
```

```
private int count=0;

public Universal_Listener ( TextArea log, Label count_label ) {
   this.log =log;
   this.count_label = count_label ;
}// constructor Universal_Listener
private void log_event (AWTEvent event) {
  count++ ;
  count_label.setText (count_text + count) ;
  log.append(event.toString()+ "\n");
} // method log_event

public void componentMoved (ComponentEvent event) {
  log_event(event) ;
} // method componentMoved
public void componentHidden (ComponentEvent event) {
  log_event(event) ;
} // method componentHidden
public void componentResized (ComponentEvent event) {
  log_event(event) ;
} // method componentResized
public void componentShown (ComponentEvent event) {
  log_event(event) ;
} // method componentShown
public void mouseDragged (MouseEvent event) {
  log_event (event) ;
} // method mouseDragged
public void mouseMoved (MouseEvent event) {
  log_event (event) ;
} // method mouseMoved
public void mousePressed (MouseEvent event) {
  log_event(event) ;
} // method mousePressed
public void mouseReleased (MouseEvent event) {
  log_event(event) ;
} // method mouseReleased
public void mouseEntered (MouseEvent event) {
  log_event(event) ;
} // method mouseEntered

public void mouseExited (MouseEvent event) {
  log_event(event) ;
} // method mouseExited

public void mouseClicked (MouseEvent event) {
  log_event(event) ;
```

```
    } // method mouseClicked

    public void keyPressed (KeyEvent event) {
      log_event(event) ;
    } // method keyPressed

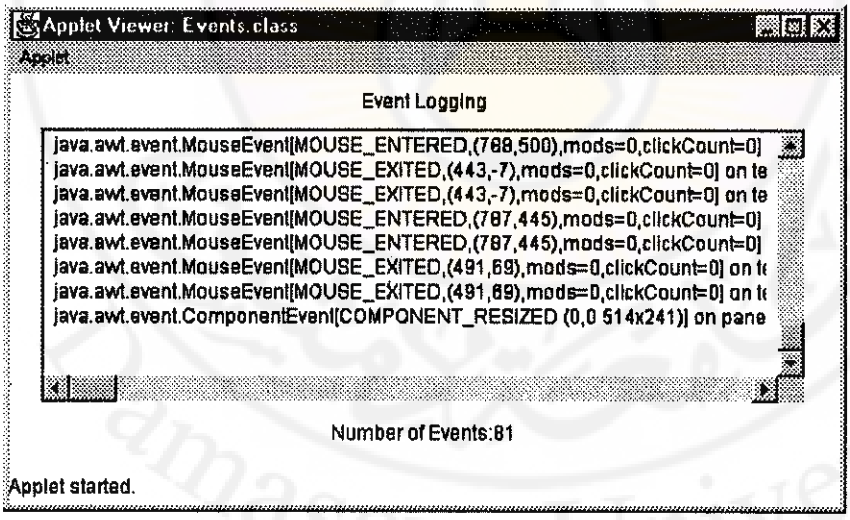
    public void keyReleased (KeyEvent event) {
      log_event(event) ;
    } // method keyReleased

    public void keyTyped (KeyEvent event) {
      log_event (event) ;
    } // method keyTyped

    public void focusGained (FocusEvent event) {
      log_event(event) ;
    } // method focusGained

    public void focusLost (FocusEvent event) {
      log_event(event) ;
    } // method focusLost

} // class Universal_Listener
```



الشكل 9-7: لقطة للبرنامج *Events*.

9-3 المكونات والحاويات

كمـــا رأيـــنا في البريمج *Mimic* تتألف واجهات الاستخدام من مكونات تسمح للمستثمر بالـــتفاعل مـــع الـــبرنامج بطريقة خاصة. على سبيل المثال يتضمن البريمج *Mimic* مكونين *TextField* و *Label*.

يعرَّف كلٌ من هذه المكونات بصف أو مجموعة من الصفوف ضمن الحزمة *java.awt*. تُشتق معظـــم هذه الصفوف من الصف *Component* الذي يزودنا بعدة طرائق مشتركة بين جميع المكونات. يوفر الصف *Component* أدوات رسم ومعالجة أحداث ومعالجة صور وتحكم في الألوان والبنوط.

نـــناقش لاحقـــاً في هذا الفصل الطرائق الخاصة المعرّفة في الصف *Component* ويمكن رؤية وصف كامل لمحتويات الصف في الملحق أ.

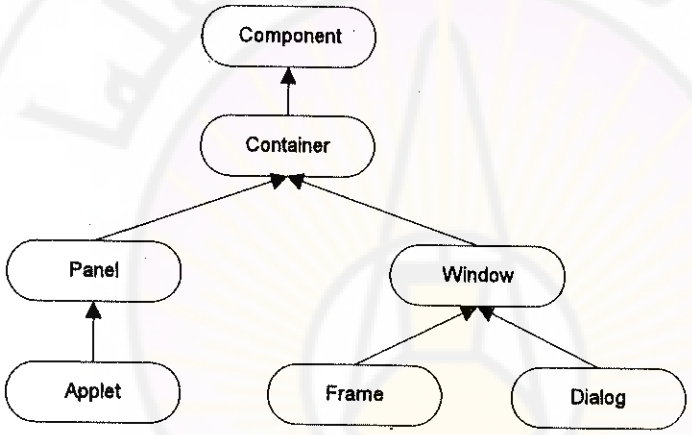
قـــبل أن نبدأ باستعراض مكونات *GUI* خاصة بالتفصيل، علينا أولاً مناقشة كيفية استعمال هذه المكونات وتنظيمها وتقديمها للمستخدم. علينا أيضاً أن نتفحص كيف يحدد البرنامج أن المستثمر قد تسبب بحدث ما مثل الضغط على زر فأرة.

9-3-1 الحاويات *Containers*

هنالك مجموعة خاصة من المكونات تسمى بالحاويات، تفيد في جمع عدة مكونات معاً ضمن وحـــدة قابلة للمعالجة كمكونة وحيدة. وتفيد الحاويات أيضاً في إظهار هذه المكونات على الشاشة. وما البريمج إلا حاوية. وسنستعمل البريمجات كأمثلة على الحاويات في هذا الفصل، وسنضيف إليها مكونات *GUI* أخرى مثل الأزرار *Buttons* بغية إظهارها.

توجد حاويات أخرى ضمن الواجهة البرمجية لتطبيقات جافا (*Java API*) وهي جميعاً مشتقة مـــن الصـــف *Container* المُشتق بدوره من الصف *Component* ، ولكل من هذه الحاويات اســـتعمالها الخـــاص. يظهر الشكل 9-8 هيكلية هذه الحاويات ويعطي الشكل 9-9 شرحاً بسيطاً لهذه الصفوف.

تُقسم الحاويات إلى جزأين يتضمن الأول الحاويات التي يجب ربطها بحاويات أخرى مثل *Applet* و *Panel* والحاويات التي يمكن أن تتحرك مستقلةً مثل *Window* ومشتقاتها. يمكن إظهار *Panel* أو *Applet* حصراً على سطح بياني آخر. يُربط *Applet* إلى متصفح ويب أو مُظهِر برمجات، ويجب إضافة *Panel* إلى حاوية أخرى كي نستطيع إظهاره. تُنشأ *Frame* أو *Dialog* أو *Window* كسطوح مستقلة ومتحركة، يمكن للمستخدم التقاط حافاتها بالفأرة وتحريكها إلى أماكن أخرى على الشاشة.



الشكل 9-8: هيكلية صفوف الحاويات.

الحاوية	وصف
Container	الصف الأب لجميع الصفوف التي يمكنها احتواء مكونات، ولا يمكن استنساخها.
Panel	تستعمل لجمع عدة مكونات، ويجب أن تنتمي بدورها إلى حاوية أكبر منها.
Applet	هي لوحة *Panel* يمكن إظهارها ضمن متصفح ويب.
Window	حاوية يمكن تغيير موضعها من قبل المستثمر، وليس لها حافات ولا شريط عنوان.
Frame	نافذة تمتلك حافات وشريط عنوان، ويمكن أن يربط بها شريط قوائم، ويمكن

	للمستخدم تغيير أبعادها وتصغيرها.
Dialog	نافذة تُنشأ للتعامل مع مواقف خاصة، ويمكنها تجميد جميع منافذ دخل البرنامج إلى حين انتهاء مهمتها.

الشكل 9-9: صفوف الحاويات في جافا

تُضـــاف المكونـــات إلى الحاويات باستعمال الطريقة *add* من الصف *Container*. لا يمكن لمكونـــة ما أن تكون جزءاً من حاويتين في آن واحد . فإذا أضفنا مكونة ما إلى حاوية فإنها ستُحذف آلياً من أية حاوية أخرى.

9-3-2 هيكلية المكونات

لمـــا كانت الحاويات مكونات، أمكن إذن إضافتها إلى مكونات أخرى. نتحكم في مظهر الواجهـــة البيانـــية جزئياً بطريقة وضع المكونات والحاويات معاً. هذه العلاقة بين المكونات والحاويات تكون ما نسميه بهيكلية المكونات.

لنأخذ مثالاً آخر يَشتق صفاً يسمى *Rings* من الصف *Canvas* ويَستعمل الطريقة *paint* من *Rings* لرسم شكلٍ بيانيٍ بسيط. يظهر الشكل 9-10 لقطة لهذا البرنمج.

أنشـــئ ثلاثة أغراض من هذا الصف أضف اثنين منهما إلى *Panel* يسمى *group* وأضف الغرض الثالث من *Rings* و *Panel* الذي يحوي الغرضين الآخرين إلى هذا البرنمج.

```
import java.applet.Applet;
import java.awt.*;

   public class Rings_Display extends Applet {

    private Panel group = new Panel () ;
    private Rings rings1 = new Rings () ;
    private Rings rings2 = new Rings () ;
    private Rings rings3 = new Rings () ;

    public void init () {
      add (rings1) ;
     group.add (rings2) ;
     group.add (rings3) ;
```

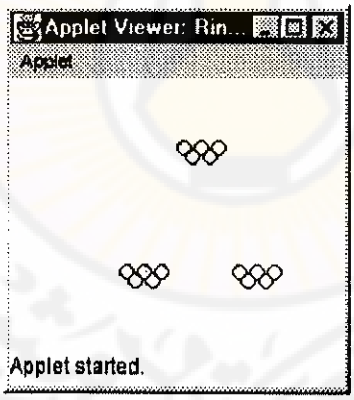
```
        add (group) ;
        resize (200,150) ;
      } // method init
} //  class Rings_Display

class Rings extends Canvas {
    private Dimension size = new Dimension (60,60) ;

    public void paint (Graphics page) {
        page.drawOval (30,30,8,8) ;
        page.drawOval (40,30,8,8) ;
        page.drawOval (50,30,8,8) ;
        page.drawOval (35, 35,8,8) ;
        page.drawOval (45,35,8,8) ;
      } // method paint

    public Dimension getMinimumSize () {
        return size ;
      } // method getMinimumSize

    public Dimension getPreferredSize () {
         return getMinimumSize () ;
      } // method getPreferredsize
} // class Rings
```

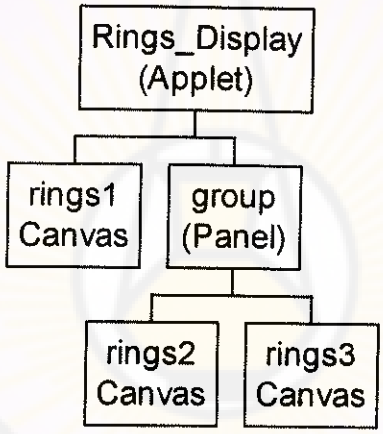


الشكل 9-10 البرنمج *Rings_Display*

لما كان لـــ *Canvas* أبعاد معدومة ما لم يُعرف غير ذلك، فإنها لن تكون مرئية ما لم تُعرف الطريقـــتان *minimumSize* و *preferredSize*. يُظهر الشكل 9-10 خرج هذا البرنامج.

الشكلان البيانيان الثاني والثالث مجموعان في *Panel* وحيد، وإن لم يكن ذلك مرئياً، ولكن عند تعديل حجم النافذة وإعادة توضع الأشكال فإن الأشكال على الـ *Panel* ستتحرك معاً كوحدة. يمكن أن توصف هيكلية المكونات للبرمج *Rings_Display* بالمخطط الظاهر في الشكل 9-11. لهيكلية مكونات برنامج ما، أهمية كبيرة عند معالجة أحداث المستخدم مثل الضغط على زر. سنناقش هذه المسألة لاحقاً في هذا الفصل.

ثم أن لطريقة ترتيب المكونات ضمن الحاوية أهميةً كبيرة أيضاً. يُحدد طريقة اختيار هذا الترتيب متولي الترتيب (*layout manager*). تستعمل متوليات الترتيب استراتيجيات مختلفة لتقرير كيفية عرض المكونات ضمن الحاويات.



الشكل 9-11: هرمية مكونات البرمج *Rings_Display*.

4-9 مكونات GUI

تُعرف مكونات *GUI* ضمن *API* جافا، وهي مجموعة آليات تستخدم للتفاعل مع المستخدم. تشتق هذه المكونات جميعاً من الصف *Component* وتستعمل للرسم ومعالجة الأحداث والتعامل مع الصور والتحكم في البنوط والألوان . تعرف المكونات كصفوف في الحزمة *java.awt*.

9-4-1 اللصاقات Label

عـــند إضـــافة لصاقة إلى حاوية فإن موقعها ضمن الحاوية يحدد بالترتيب الخاص بالحاوية، ويُخصص له مساحة *area* معينة يتوضع النص ضمنها إما إلى اليمين أو اليسار أو الوسط، ويعرف التوضع باستعمال أحد ثلاثة ثوابت صحيحة وهي *CENTER* أو *RIGHT* أو *LEFT* وهيَ القيمة الافتراضية.

لهذا الصف ثلاثة بوان:

```
Label()
Label ( String label) ,
Label (String label, int alignet)
```

لا يـــأخذ الـــباني الأول أي وسطاء، في حين يأخذ الثاني وسيطاً واحداً وهو النص البدئي للصـــاقة، ويأخذ الثالث إضافةً إلى النص البدئي نوعَ المحاذاة. وفيما يلي بعض الطرائق هذا الصف:

```
void setText (String label)
void setAlignn.ent (int Alignment)
void getText()
int getAlignment()
```

كمـــا تبين أسماء هذه الطرائق، فهي تستعمل لتغير نص اللصاقة *setText* وتغيير نمط المحاذاة *setAlignment* والحصول على نص اللصاقة *getText* والحصول على نوع المحاذاة *getAlignment* . البرنمج *Mimic* هو مثال على استعمال *Label*.

9-4-2 حقول النصوص TextField و مساحات النصوص TextArea

حقـــل النص هو مكونة *GUI* تحتوي على سطر نصي واحد، يستطيع المستخدم تعديله إذا حـــدد المـــبرمج ذلك. لما كان من الممكن تعديل قيمة هذا الحقل، كان بالإمكان استعماله للحصول على دخل من المستخدم.

مساحة النص هي مكونة مشابهة لحقل النص، يمكنها الامتداد على أكثر من سطر، ويُربط بها شـــريطا إزاحة (*Scrollbars*) أفقي وعمودي يسمحان برؤية كامل النص ولو كانت النافذة

أصغر من حجم النص الفعلي.

يعـــرَّف حقل النص بالصف *TextField* كما تُعرف مساحة النص بالصف *TextArea* وكلا الصفين مشتق من الصف *TextComponent* الذي يحتوي على طرائق عامة يستعملها الصفان معاً. تنتمي الصفوف الثلاثة إلى الحزمة *java.awt*. لا يمتلك الصف *TextComponent* أي بانٍ عمومي، لذا لا يمكن خلق أي نسخة منه.

يمتلك الصف *TextField* أربعة بوان:

```
TextField ()
TextField (int columns)
TextField (string text)
TextField (string text, int columns)
```

لا يحدد البانيان الأول ولا الثاني النص البدئي لحقل النص، ويقبل الثاني والرابع عدداً صحيحاً يحدد عرض حقل النص بالحروف (أو الأعمدة).

تشـــبه بـــواني مساحة النص بواني حقل النص، إلا أن حجم مساحة النص يعطى بالأسطر والأعمدة معاً:

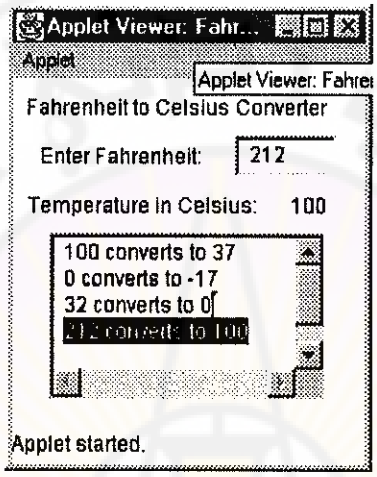
```
TextArea ()
TextArea (int columns)
TextArea (string text)
TextArea (string text,int rows, int columns)
```

الطـــرائق التالية معرَّفة في الصف *TextComponent* وتستعمل في حقول النصوص ومساحات النصوص في آن واحد:

```
void setText (string text)
string getText ()
string getSelectedText ()
void setEditable (boolean b)
```

تحدد الطريقة *setText* صراحة النص الواجب إظهاره، وتعيد الطريقة *getText* النص الحالي. تعـــيد الطريقة *getSelectedText* سلسلة محرفية موافقة للنص المحدد (المختار) وتقبل الطريقة *setEditable* قيمة منطقية تحدد إمكان تعديل النص أو لا. يَستعمل البرنمج التالي حقل نص للحصـــول على دخل المستخدم الموافق لدرجة حرارة فهرنهاتية. عندما يضغط المستثمر على

مفـــتاح *Enter* تُحول الحرارة إلى قيمة مئوية تُكتب ضمن لصاقة وضمن مساحة نصية تفيد كـــتقرير عن كل التحويلات السابقة. يستطيع المستخدم الاستمرار في التحويل حتى توقُّف البرنمج. يظهر الشكل 9-12 لقطة من هذا البرنمج.



الشكل 9-12: البرنمج *Fahrenheit*.

يحقـــق البرنامج التالي البرنمج *Fahrenheit* ويستعمل *ActionEvent* مرتبطاً بالمكونة *TextField* ليحدد متى يستجيب:

```
import java.awt.event.*;

// converts temperatures from Fahrenheit to Celsius.
public class Fahrenheit extends Applet {

    private Label title = new Label ("Fahrenheit to Celsius Converter");
    private Label question = new Label ("Enter Fahrenheit:");
    private Label answer = new Label ("Temperature in Celsius:");

    private TextField fahrenheit = new TextField (5) ;
    private Label celsius  = new Label ("N/A") ;
    private TextArea log = new TextArea (5,20) ;

    private Fahrenheit_Listener listener= new Fahrenheit_Listener (this) ;

    public void init () {
```

```
    fahrenheit.addActionListener (listener) ;
    log.setEditable (false) ;
    add (title) ;
    add (question) ;
    add (fahrenheit) ;
    add (answer) ;
    add (celsius) ;
    add (log) ;
    resize (200,200);
  } // method init

  public void convert () {
    int fahrenheit_temperature, celsius_temperature ;
    String text = fahrenheit.getText () ;

    fahrenheit_temperature = Integer.parseInt (text) ;
    celsius_temperature = (fahrenheit_temperature-32) *5/9 ;

    celsius.setText (Integer.toString (celsius_temperature)) ;
    log.append (text+ " converts to " +
                celsius.getText () + "\n") ;
    } // method convert
  } // class fahrenheit

  // The listner class for the textfield
  class Fahrenheit_Listener implements ActionListener {
    private Fahrenheit applet;

    public Fahrenheit_Listener(Fahrenheit listening_applet) {
      applet = listening_applet;
    } // constructor Fahrenheit_Listener

    public void actionPerformed(ActionEvent action) {
      applet.convert();
    } // method actionPerformed
  } // class Fahrenheit_Listener
```

3-4-9 القوائم: lists

تُستعمل المكونة *GUI* المعرفة بالصف *List* لإظهار قائمة من سلاسل محرفية قابلة للانتقاء (*selectable string*)، يُحدد قياس القائمة بعدد الأسطر الظاهرة ضمنها. ويمكنها أن تحتوي أي عدد من العناصر. وعند الحاجة يظهر شريط إزاحة (*scrollbar*) إلى يمينها ليساعدنا على

الوصول إلى العناصر الخفية.

يمكـــن اختـــيار عنصر من القائمة بضغطة واحدة من الفأرة، كما يمكن أن يَسمح المبرمج باختيار أكثر من عنصر. تفيد طرائق الصف *List* بإيجاد النصوص المختارة بغية معالجتها. كما يُحَدَّد كل عنصر من القائمة برقم يدل على موقعه. يقع العنصر الأول من القائمة في الموقع *0* . للصف *List* بانيان هما:

List ()
List (int rows, boolean multipleSelections)

ينشئ الباني الأول قائمة بدون حجم مرئي، ولا يسمح باختيار عدة عناصر من هذه القائمة في آنٍ واحـــد. ينشـــئ الباني الثاني قائمة عدد أسطرها هو *rows* و تقبل الاختيار المتعدد إذا كانت قيمة *multipleSelections* مساوية *true*.

من الطرائق الأساسية في هذا الصف ما يلي:

- *void addItem (String item)*
- *void addItem (Strig item, int index)*
- *boolean isMultipleMode ()*
- *void setMultioleMode (boolean b)*
- *init countItems ()*
- *void clear ()*
- *void delItem (int index)*
- *String getitem (int index)*
- *String getSelectedItem ()*
- *String -[] getSelectedItems ()*

تضـــيف الطريقة المحملة (*addItem (over loaded*) العنصر إما في نهاية القائمة أو في موضع معين فيها. و تعيد الطريقة *countItems* عدد العناصر في القائمة وتُستعمل الطريقة *delItem* لحـــذف العنصـــر الموجود في موضع معين. وأخيراً تعيد الطريقة *getSelectedItem* السلسلة المحرفية المختارة حالياً، وإذا كان نمط الاختيار المتعدد مُفعَّلاً تعيد الطريقة *getSelectedItemes* جدول *array* بالسلاسل المختارة.

4-4-9 الأزرار *Buttons*

تقدم لنا الحزمة *java.aunt* أربعة أنواع مختلفة من الأزرار. كل منها يخدم في حالات مختلفة

كما هو موضح بالشكل 9-13.

يعرف زر الضغط بالصف *Button* ويمكن إنشاؤه مع أو بدون لصاقة باستعمال البواني التالية

Button ()
Button (String label)

يعـــرف زر الاختـــيار بالصـــف *Choice* ويمتلك بانياً وحيداً لا يقبل أي وسطاء. تضاف الخيارات إلى قائمة الخيارات ويُستعلم عنها باستعمال الطرائق التالية:

نمط الزر	وصف
Push button	زر وحيد مجهز للقيام بعمل ما عند الضغط عليه
Choice button	زر وحيد يظهر قائمة خيارات عند الضغط عليه ويظهر الخيار الحالي إلى جانب الزر
Checkbox button	زر يأخذ قيمتين محدد/ غير محدد. تستعمل بمجموعة من هذه الأزرار لتعريف حالة ما
Radio buttons	مجموعة أزرار تأخذ إحدى حالتين. لا يمكن انتقاء اثنين منها في آن واحد

الشكل 9-13 : أنواع الأزرار المعرفة في الحزمة *java.awt*.

- *void addItem (String item)*
- *int getItemCount ()*
- *String getItem (int item)*
- *int getSelectedItem ()*
- *String getSelectedItem ()*
- *void select (int index)*
- *void select (String string)*

يمكن تمييز العناصر في قائمة الخيارات إما باسمها أو بموقعها ضمن القائمة. يمكن اختيار عنصر ما باستعمال إحدى الطرائق المحملة *select* ونحصل عليه باستعمال الطريقة *getselectedItem*. تسمح الطريقة *addItem* بإضافة عنصر في الموقع المختار حالياً.

تعرف أزرار التحديد (*chekbox*) وأزرار الراديو (*radio*) باستعمال الصف *Checkbox*. ليس هـــنالك حاجة لتجميع أزرار التحديد ضمن كتلة واحدة، لأن الاختيار المتعدد مسموح به، في حـــين لا يمكن اختيار أكثر من زر راديو في آنٍ واحد، ولذلك تجمع هذه الأزرار ضمن مجموعة تُعرف بالصف *CheckboxGroup*. للصف *Chekbox* ثلاثة بوانٍ:

- *Checkbox ()*
- *Checkbox (String label)*
- *Checkbox (String label, CheckboxGroup group, Bollean state)*

يمكن إنشاء *Checkbox* مع سلسلة محرفية تشرح الخيار المقصود ومجموعة *CheckboxGroup* التي ينتمي إليها، وقيمة منطقية تدل على حالته البدائية. يبين المقطع 10-6 مثالاً لاستعمال مختلف أنواع الأزرار.

5-4-9 *Scrollbars*

كما رأينا سابقاً تمتلك المساحة النصية (*TextArea*) شريط انزياح (*Scrollbar*) مرتبط بأسفلها، وآخر إلى يمينها يفيدان في رؤية أجزاء النص التي لا ترى من خلال النافذة الحالية، وذلك بسحبها إلى داخل النافذة. كذلك هي حال القوائم، إذ يُضاف إليها آلياً شريط إزاحة في حال تجاوز حجم القائمة حجم المساحة المرئية منها.

إلا أن شريط الإزاحة نفسه هو مكونة *GUI* مستقلة بحد ذاتها، يمكن استعمالها مستقلة لضبط قيمة ما ضمن مجال محدد. ويفيد شريط الإزاحة أيضاً في تحديد الموقع النسبي للأغراض على الشاشة. يُعرف شريط الإزاحة باستعمال الصف *Scrollbar* الذي يملك ثلاثة بوان:

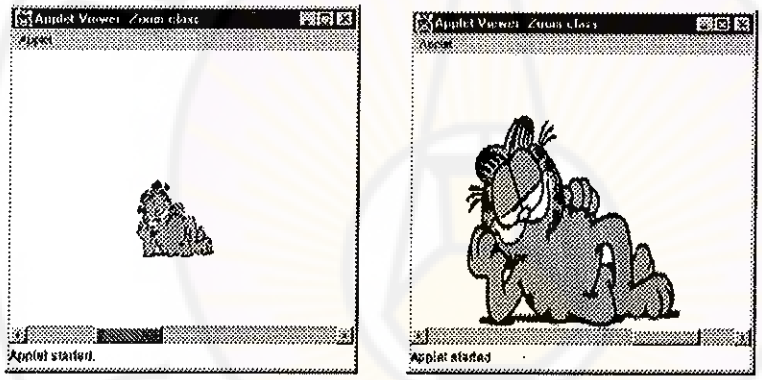
- *Scrollbar ()*
- *Scrollbar (int orientation)*
- *Scrollbar (int orientation, int value, int visible minimum, int maximum)*

يفيد الباني الأخير في تحديد اتجاه شريط الإزاحة وذلك بإعطاء الوسيط *orientation* إحدى القيمتين الثابتتين *HORIZONTAL* أو *VERTICAL* وفي تحديد القيمة البدائية لشريط الإزاحة والحجم المرئي له وقيمتيه الصغرى والكبرى. في غياب وسيط الاتجاه *orientation* يأخذ القيمة *HORIZONTAL* أما المعاملات *value* و *visible* و *minimum* و *maximum* فإنها تأخذ القيمة 0 في حال استعمال الباني الأول أو الثاني. يمكن الضغط على شريط الإزاحة في أماكن مختلفة. وتختلف المسافة التي يتحركها شريط الإزاحة وفقاً للمكان الذي نضغط عليه. فالضغط على الأسهم في طرفي شريط الإزاحة يؤدي إلى تحرك واحدي. في حين أن الضغط على المنطقة الواقعة بين المستطيل المتحرك والأسهم تدل على تحرك بمقدار كتلة (*block*). يمكننا تحديد قيم التحرك الواحدي والكتلي باستعمال الطرائق التالية:

- *void setUnitIncrement (int increment)*
- *void setBlockIncrement (int increment)*

- *void setValue (int value)*
- *void setValues (int value, int visible, int minimum)*
- *int getUnitIncrement ()*
- *int getBlockIncrement ()*
- *int getValue ()*
- *int getMaximum ()*
- *int getMinimum ()*
- *int getOrientation ()*
- *int getVisibleAmount ()*

تحتوي القائمة السابقة على طرائق أخرى تفيد في الحصول على الحالة الحالية لبعض خواص شـــريط الإزاحة. يَستعمل البرنمج التالي شريط إزاحة لتكبير صورة وتصغيرها. تبدأ الصورة كمستطيل صغير فوق شريط إزاحة أفقي، وعند تحريك شريط الإزاحة يكبر حجم الصورة. يُظهر الشكل 9-14 لقطة لهذا البرنمج.



الشكل 9-14: لقطة للبرنمج

يستعمل البرنامج الواجهة *AdjustmentListener* لإنشاء صف يستجيب لتغييرات موقع شريط الإزاحة، فيضبط القياس الأعظمي لشريط الإزاحة مع أكبر حجم ممكن للصورة ويربط أبعاد الصورة بموقع شريط الإزاحة الحالي.

عـــند التقاط الحدث *AdjustmentEvent* تُستدعى الطريقة *adjustValueChanged* من الصف *Zoom_Adjust_listner* وتطلـــب هذه الطريقة بدورها الطريقة *Zoom_image* للبرنمج *Zoom* لضبط أبعاد الصورة وإعادة رسم البرنمج.

import java.applet.;*

```
import java.awt.*;
import java.awt.event.*;

// zooms in and out on an Image controlled by a scrollbar
public class Zoom extends Applet {

    private final int SIZE = 300 ;

    private Scrollbar bar =
       new Scrollbar (Scrollbar.HORIZONTAL, 10,64,10,SIZE) ;
    private Zoom_Listener listener = new Zoom_Listener (this) ;

    private int current_size = 10 ;
    private Image picture ;

    public void init ()  {
      setLayout (new BorderLayout() ) ;
      setBackground (Color.white) ;
      bar.addAdjustmentListener (listener) ;
      picture = getImage (getDocumentBase(), "Garfield.gif") ;
      add (bar, "South") ;
      setSize (SIZE,SIZE) ;
      setVisible (true) ;
    } // method init

    public void zoom_image (int size) {
      current_size = size ;
      repaint () ;
    } // method zoom_image

    public void paint (Graphics page) {
      page.drawImage (picture, SIZE/2-current_size/2,
            SIZE/2- current_size/2, current_size, current_size, this) ;
     } // mathod paint

} // class Zoom

// The AdjustmentEvent Listener for zoom
class Zoom_Listener implements AdjustmentListener {
    private Zoom applet ;

   public Zoom_Listener (Zoom zoom_applet) {
      applet = zoom_applet ;
   } // constructor zoom_Listener

   public void adjustmentValueChanged (AdjustmentEvent event) {
```

```
        applet.zoom_image (event.getValue() ) ;
    } // method adjustmentValueChanged

} // class zoom_Listener
```

5-9 Layout Manager (متولي الترتيب)

تستعمل كل حاوية متولّي ترتيب (*Layout Manager*) لترتيب مكوناتها. هنالك عدة متوليات ترتيب معرفة في الحزمة *Java.awt*. نشرح في هذا المقطع كيف تؤثر كل من متوليات الترتيب هذه في أسلوب ترتيب المكونات. يصف الشكل 9-15 كلاً من متوليات الترتيب هذه ويُعرف كل منها بصف خاص ضمن الحزمة *Java.awt*.

متولي الترتيب	وصفه
Flow Layout	يرتب المكونات من اليمين إلى اليسار وينتقل إلى سطر جديد عند الحاجة وهو الترتيب الافتراضي للوحات والبريمجات.
Border Layout	يعرف خمسة مواقع يمكن إضافة المكونات إليها وهي الشرق والغرب والشمال والجنوب والوسط. وهو الترتيب الافتراضي للنوافذ *Window* ومشتقاتها.
Card Layout	يظهر مكونة واحدة في آن واحد تشغل كامل مساحة الحاوية.
Grid Layout	ترتب المكونات في أسطر وأعمدة وتجعلها جميعاً متساوية الأبعاد.
Grid Bag Layout	ترتب المكونات ضمن عدد محدد من الأسطر والأعمدة وتسمح لمكونة ما أن تشغل أكثر من سطر أو عمود.

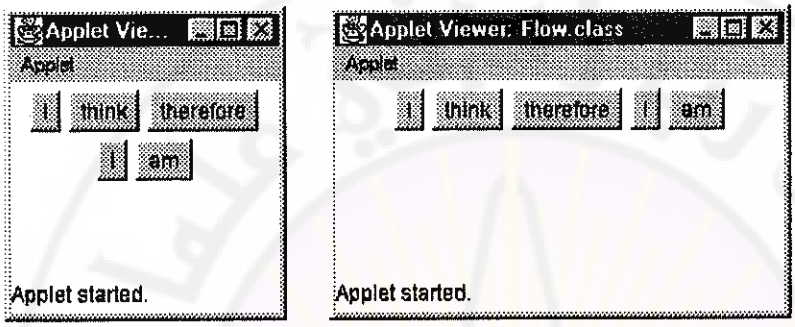
الشكل 9-15: متوليات الترتيب.

يمكن ربط متولي الترتيب إلى حاوية باستعمال الطريقة *SetLayout* المعرفة ضمن الصف *Container* وتعيد الطريقة *getLayout* متولي الترتيب الذي تستعمله الحاوية.

1-5-9 *Flow Layout* (الترتيب التدفقي)

تستعمل جميع البريمجات واللوحات *(Pannels)* متولي الترتيب التدفقي *Flowlayout* ما لم يصرح عن متولي ترتيب آخر. عند إضافة مكونات الحاوية محكومةً بمتولي الترتيب التدفقي،

فإنهـــا تُرتب على سطر من اليسار إلى اليمين؛ فإن لم يبق مكان لإضافة مكونة جديدة يبدأ عـــندها ســـطر جديـــد. تُرتب المكونات على السطر الواحد بحيث تتحاذى منتصفاتها ولو امتلكت أبعاداً مختلفة. يظهر الشكل 16-9 لقطتين لبرنمج يستعمل الترتيب التدفقي.



الشكل 9-16: البرنمج *Flow*.

يظهـــر الـــبرنمج خمسة أزرار مرتبة وفقاً لترتيب طلب الطريقة *add* لكل منها. تظهر السمة المميزة للترتيب التدفقي تغيير أبعاد البرنمج: ففي الحالة الأولى ترتب المكونات على سطرين، على حين يعاد ترتيبها في الحالة الثانية بعد تكبير عرض البرنمج ليتسع لها جميعاً سطر واحد. يحدد قياس كل زر بحيث يتسع لإظهار لصاقته، وفيما يلي تحقيق لهذا البرنمج:

```
import java.applet.Applet;
import java.awt.*;

// Show how the flow layout manager works
public class Flow extends Applet {

    private Button button1 = new Button ("I") ;
    private Button button2 = new Button ("think") ;
    private Button button3 = new Button ("therefore") ;
    private Button button4 = new Button ("I") ;
    private Button button5 = new Button ("am") ;

    public void init () {
      setLayout (new FlowLayout () ) ;
      add (button1) ;
      add (button2) ;
      add (button3) ;
      add (button4) ;
```

```
        add (button5) ;
        setVisible (true) ;
    } // method init
} // class Flow
```

مـــا لم يحـــدد غير ذلك، تكون الأسطر المختلفة في الترتيب التدفقي متمركزة في منتصف البريمج، ويمكن تحديد غير ذلك باستعمال الثوابت *LEFT, CENTER, RIGHT*. يسمح متولي الترتيـــب الـــتدفقي للمستثمر بتحديد الفراغات الأفقية بين المكونات على السطر الواحد، والفـــراغات العموديـــة بين الأسطر. تقاس هذه الفراغات بالبكسل. يعرف متولي الترتيب التدفقي بالصف *FlowLayout* الذي يمتلك البواني التالية:

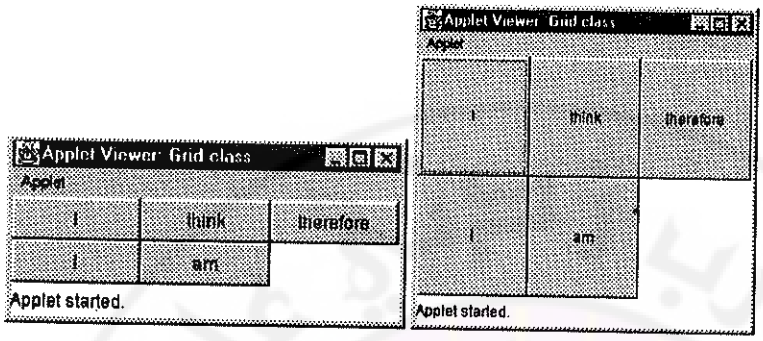
```
FlowLayout()
FlowLayout(int align)
FlowLayout(int align, int hgap, int vgap);
```

يـــأخذ الباني الأخير ثلاثة وسطاء هي نوع المحاذاة والفراغ الأفقي والفراغ العمودي. إذا لم تعطَ قيمة للمحاذاة فهي *CENTER*، وإن لم تعطَ قيم للفراغات فهي اصطلاحاً 5 بكسل.

2-5-9 *Grid Layout* (الترتيب الشبكي)

يضع متولي الترتيب الشبكي المكونات ضمن شبكة ذات عدد معين من الأسطر والأعمدة. تحـــتل كل مكونة خلية واحدة من هذه الشبكة المرتبة من اليسار إلى اليمين ومن الأعلى إلى الأسفل، وتمتلك جميع الخلايا الحجم نفسه.

يظهـــر الشكل 9-17 لقطتين لبريمج يستعمل الترتيب الشبكي. يستعمل هذا المثال الخاص شبكة 2*3 معبأة بخمسة أزرار. عند تغيير أبعاد البريمج يزداد حجم جميع الأزرار بما يتوافق مع الأبعاد الجديدة.



الشكل 9-17: البرنامج *Grid*.

```
import java.applet.Applet ;
import java.awt.*;
// show how the grid layout manger works
public class Grid extends Applet {

    private Button button1 = new Button ("I") ;
    private Button button2 = new Button ("think") ;
    private Button button3 = new Button ("therefore") ;
    private Button button4 = new Button ("I") ;
    private Button button5 = new Button ("am") ;

    public void init () {
       setLayout (new GridLayout (2,3) ) ;
       add (button1) ;
       add (button2) ;
       add (button3) ;
       add (button4) ;
       add (button5) ;
       setVisible (true) ;
    } // method init
} // class Grid
```

يمتلك الصف *GridLayout* البواني التالية:

- *GridLayout (int rows, int columns)*
- *GridLayout (int rows, int columns, int hgap, int vgap)*

يحـــدد كلا البانيين عدد الأسطر والأعمدة في الشبكة، ويحدد الباني الثاني الفراغات الأفقية والعمودية. إذا أعطي عدد أسطر معدوم يكون عددها هو عدد المكونات مقسوماً على عدد الأعمدة، والعكس صحيح ولا يمكن إعطاء الأسطر والأعمدة صفراً في آن واحد.

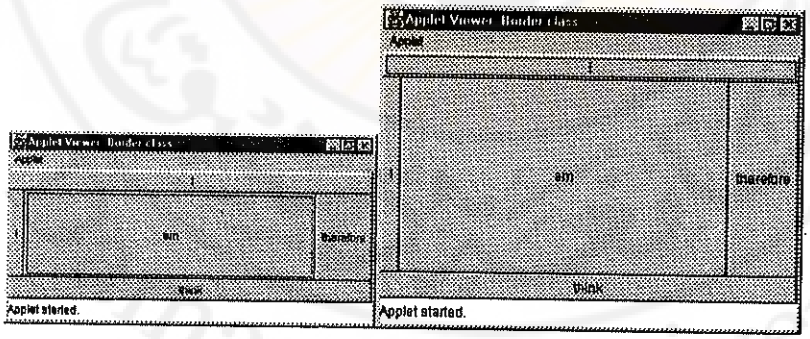
3-5-9 Border Layout (الترتيب الحافّي)

يعـــرف الترتيب الحافي خمسة مواقع يمكن أن توضع فيها المكونات وهي: الشمال *(North)* والجـــنوب *(South)* والشرق *(East)* والغرب *(West)* والوسط *(Center)*. تتوضع هذه المواقع بعضـــها بالنسبة إلى بعض كما هو موضح بالشكل 9-18 ويمكنها أن تأخذ أبعاداً مختلفة لكنها تشغل كامل سطح الحاوية.

عـــند إضافة مكونة إلى حاوية تستعمل الترتيب الحافي، يجب تحديد موقع هذه المكونة. يجب وضـــع مكونة وحيدة في كل موقع. فإذا أردنا وضع أكثر من مكونة في موقع ما وجب أن نجمعها أولاً ضمن لوحة *(Pannel)* ثم إضافة اللوحة في الموقع المطلوب. يمكن رؤية مثال على هذا النوع من التجميع في البرنامج المشروح في المقطع 6-10.

تتعلق أبعاد كل موقع بأبعاد المكونة المضافة إليه. تمتلك المواقع *North* و *South* نفس عرض الحاويـــة ولهمـــا نفس الارتفاع المفضل للمكونة. تمتلك المواقع *East* و *West* نفس العرض المفضل للمكونة ويحدد ارتفاعها بالمسافة المتبقية بين المواقع *North* و *South*. على حين يحتل الموقع *Center* ما تبقى من سطح الحاوية.

يظهر الشكل 9-18 لقطتين لبرنمج يستعمل الترتيب الحافي بأبعاد مختلفة ويحقق الرماز التالي هذا البرنمج.



الشكل 9-18: الترتيب الحافي

```
import java.applet.Applet ;
```

```
import java.awt.*;
// show how the border layout manger works
public class Border extends Applet {

    private Button button1 = new Button ("I") ;
    private Button button2 = new Button ("think") ;
    private Button button3 = new Button ("therefore") ;
    private Button button4 = new Button ("I") ;
    private Button button5 = new Button ("am") ;

    public void init () {
       setLayout (new BorderLayout () ) ;
       add (button1, "North") ;
       add (button2, "South") ;
       add (button3, "East") ;
       add (button4, "West") ;
       add (button5, "Center") ;
       setVisible (true) ;
    } // method init
} // class Border
```

لاحظ أنه لا يتحدد موقع المكونات بترتيب إضافتها بالطريق *add* وإنما بالوسيط الثاني الذي يأخذ القيم *"North"* و *"South"* و *"East"* و *"West"* و *"Center"* يمتلك الصف *BorderLayout* البانيين التاليين:

```
BorderLayout()
BorderLayout(int hgap, int vgap)
```

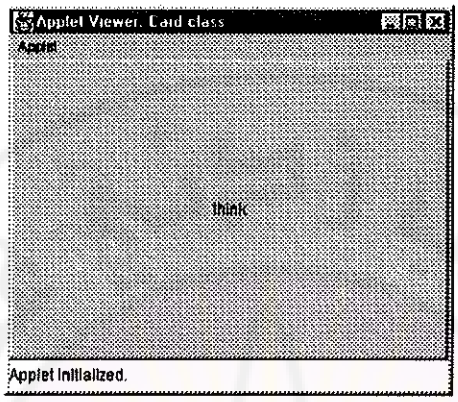
يحدد الوسيطان *hgap* و *vgap* الفراغات الأفقية والعمودية بين المواقع، وفي حال غياب هذين الوسيطين تكون هذه الفراغات معدومة.

4-5-9 Card Layout (الترتيب البطاقي)

مع أنه من الممكن إضافة أي عدد من المكونات إلى حاوية تستعمل الترتيب البطاقي، فإنه لا يمكن إظهار إلا مكونة واحدة فقط.

ترتب المكونات حسب ترتيب إضافتها إلى الحاوية، وتعدل أبعاد المكونة المرئية في الوقت الحالي لتحتل كامل المساحة المرئية من الحاوية. يظهر الشكل 9-19 برنامجاً يستعمل الترتيب البطاقي، وتضاف خمسة أزرار إلى هذا البرنامج، إلا أن الزر المرئي هو الزر الأخير

فقط.



الشكل 9-19: الترتيب البطاقي

يحقق البرنامج التالي هذا البرمج:

```
import java.applet.Applet ;
import java.awt.*;

// show how the card layout manger works
public class Card extends Applet {

    private Button button1 = new Button ("I") ;
    private Button button2 = new Button ("think") ;
    private Button button3 = new Button ("therefore") ;
    private Button button4 = new Button ("I") ;
    private Button button5 = new Button ("am") ;
    private CardLayout layout = new CardLayout () ;

    public void init () {
       setLayout (layout ) ;
       add (button1, "one") ;
       add (button2, "Two") ;
       add (button3, "Three") ;
       add (button4, "four") ;
       add (button5, "five") ;
       setVisible (true) ;
    } // method init

    public void start () {
       for (int cards = 1; cards < 50; cards++) {
```

```
            layout.next (this) ;
            for (int pause = 1; pause <400000; pause++);
        }
    } // method start
} // class Card
```

يملك الصف *CardLayout* البانيين التاليين:

- *CardLayout ()*
- *CardLayout (int hgap, int vgap)*

يعطي الباني الثاني المسافات الأفقية و الشاقولية المراد تركها بين المكونة وحافات الحاوية، وفي حال غياب هذين الوسيطين فإنهما يأخذان القيمة الافتراضية صفر.

تُعرف الطرائق التالية ضمن الصف *CardLayout*:

- *void first (Container cont)*
- *void last (Container cont)*
- *void next (Container cont)*
- *void pervious (Container cont)*
- *void show (Container cont, String name)*

تحـــدد هـــذه الطرائق المكونة الواجب ظهورها ضمن الحاوية المحددة بالوسيط *Cont*. تظهر الطريقـــتان *first* و *last* المكونـــة الأولى أو الأخيرة (على التوالي) ضمن قائمة المكونات، وتظهر الطريقة *next* المكونة التالية للمكونة الظاهرة حالياً، في حين تظهر الطريقة *previous* المكونة السابقة لها.

تُســـتعمل الطـــريقة *show* لتحديد المكونة المراد إظهارها صراحة، ولا يمكن استعمال هذه الطريقة إلا بعد استعمال الطريقة *addLayoutComponent* لربط سلسلة محرفية بمكونة ما.

9-5-5 الترتيب الشبكي الحقائبي *Grid Bag Layout*

كما هو حال الترتيب الشبكي، صمم الترتيب الشبكي الحقائبي على شكل شبكة ثنائية البعد من أعمدة وأسطر، ولكن لا تمتلك الخلايا في هذه الشبكة نفس الأبعاد. ويمكن لمكونة ما أن تمتد على أكثر من خلية.

يعرف متولي الترتيب الشبكي الحقائبي بالصف *GridBagLayout* الذي يمتلك بانياً وحيداً لا

يقبل أي وسيط. مبدئياً لا تحتوي الشبكة الحقائبية على أي خلية، ولكن الخلايا تُنشأ آلياً عند إضافة المكونات. وليس هناك ضرورة لإشغال جميع الخلايا.

يـــربط بكل مكونة في ترتيب شبكي حقائبي سلسلة من القيود، كما هو معرف في الصف المسمى *GridBagConstraints*. تحدد هذه القيود أبعادَ وموقع المكونة ضمن الشبكة الحقائبية. يظهر الشكل 9-20 الخيارات الممكنة لتقييد مكونة.

يمكن إنشاء تنوع كبير من التراتيب باستعمال الخيارات الموضحة في الشكل 9-20. يُظهر الشكل 9-21 لقطتين لبرنمج يستعمل الترتيب الشبكي الحقائبي. يوضح هذا البرنمج كيفية تعـــريف أسطر للحقيبة الشبكية، وكيف تتغير هذه الأسطر بتأثير تغيُّر أبعاد الحاوية. لا يغير حجم الزر "*I*" الأول والزر "*am*" في حين يتوسع الزر"*I*" الثاني في الاتجاهين ويتوسع الزران "*think*" و "*therefore*" في اتجاهات مختلفة . يحقق البرنامج التالي هذا البرنمج

```
import java.applet.Applet ;
import java.awt.*;

// show how the grid bag layout manger works
public class Grid_Bag extends Applet {

   private Button button1 = new Button ("I") ;
   private Button button2 = new Button ("think") ;
   private Button button3 = new Button ("therefore") ;
   private Button button4 = new Button ("I") ;
   private Button button5 = new Button ("am") ;

   private GridBagLayout gridbag = new GridBagLayout () ;

   private void first_row (Button button) {
     GridBagConstraints constraints = new GridBagConstraints () ;

      constraints.gridwidth = GridBagConstraints.REMAINDER;
      constraints.anchor    = GridBagConstraints.WEST;

      gridbag.setConstraints (button, constraints) ;
      add (button) ;
   } // method first_row

   private void second_row (Button button1, Button button2) {
      GridBagConstraints constraints = new GridBagConstraints () ;
```

```
    constraints.gridwidth = 8;
    constraints.fill      = GridBagConstraints.HORIZONTAL;
    constraints.weightx   = 1.0;

    gridbag.setConstraints (button1, constraints) ;
    add (button1);
    constraints = new GridBagConstraints () ;

    constraints.gridwidth = GridBagConstraints.REMAINDER;
    constraints.fill      = GridBagConstraints.VERTICAL;
    constraints.weighty   = 1.0;

    gridbag.setConstraints (button2, constraints) ;
    add (button2) ;
  } // method second_row

  private void third_row (Button button) {
  GridBagConstraints constraints = new GridBagConstraints () ;

  constraints.gridwidth = GridBagConstraints.REMAINDER;
  constraints.fill      = GridBagConstraints.BOTH;
  constraints.weighty   = 1.0;
  gridbag.setConstraints (button, constraints);
  add (button);
  } // method third_row

  private void fourth_row (Button button) {
    GridBagConstraints constraints = new GridBagConstraints () ;

    constraints.gridwidth = GridBagConstraints.REMAINDER;

    gridbag.setConstraints (button, constraints) ;
    add (button) ;
   // method fourth_row

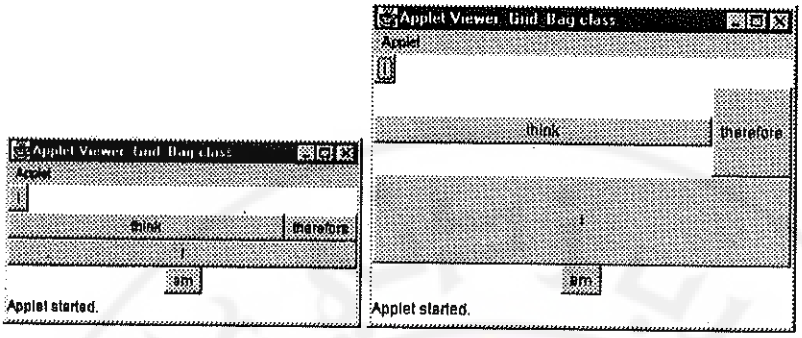
  public void init () {
    setLayout (gridbag) ;

    first_row (button1) ;
    second_row (button2, button3) ;
    third_row (button4) ;
    fourth_row (button5) ;

    setVisible (true) ;
  } // method init
} // class Grid_Bag
```

القيمة الافتراضية	القيم الممكن اسنادها	وصف	حقل القيد
CENTER	*CENTER* *EAST* *NORTH* *NORTHEAST* *NORTHWEST* *SOUTH* *SOUTHEAST* *SOUTHWEST* *WEST*	يحدد هذا الحقل موقع المكونة عندما تكون أصغر من موقعها ضمن الشبكة.	*anchor*
NONE	*BOTH* *HORIZONTAL* *NONE* *VERTICAL*	اتجاه توسع المكونة عندما يتوفر المكان لذلك.	*fill*
RELATIVE	*RELATIVE* *x interger value of a grid location*	يمثل الإحداثي الأول موقع المكونة ضمن إحداثيات الشبكة.	*gridx*
RELATIVE	*RELATIVE* *x interger value of a grid location*	يمثل الإحداثي الثاني موقع المكونة ضمن إحداثيات الشبكة.	*gridy*
1	*RELATIVE* *REMAINDER* *Size integer value in cells*	عرض المكونة معطى بالخلايا.	*gridwidth*
1	*RELATIVE* *REMAINDER* *Size integer value in cells*	ارتفاع المكونة معطى بالخلايا.	*gridheight*
0	*Size integer value of pixels for padding*		*ipadx*
0	*Size integer value of pixels for padding*		*ipady*
0, 0, 0, 0	*insets class representing appropriate internal padding*		*insets*
0	*Integer value to be used to create a growth ratio with other components in the grid*		*weightx*
0	*Integer value to be used to create a growth ratio with other components in a grid.*		*weighty*

الشكل 9-20: حقول الصف *GridBagConstraints*.



الشكل 9-21: الترتيب *GridBag*

يبين البرنمج كيفية بناء الأسطر. تعرف الطريقة *first_row* قيداً على الزر "*I*". يأخذ المتحول *gridwidth* القيمة *REMAINDER* وهذا ما يدل على أن هذا الزر هو آخر مكونة على هذا السطر. يأخذ متحول القيد *anchor* القيمة *WEST* وهذا ما يدل على أن المكونة ستتوضع إلى أقصى اليسار ما أمكن. و لما كانت القيمة الافتراضية للمتحول *fill* هي *NONE* فإن أبعاد هذا الزر لن تتغير بتغيُّر أبعاد البرنمج. تسمح الطريقة *second_row* للزر "*think*" بالتمدد أفقياً و ذلك باعطاء المتحول *fill* القيمة *HORIZENTAL* والمتحول *weightx* القيمة 1. وقد ضُبطت أبعاد الزر "*think*" لتكون أكبر بثماني مرات من أبعاد الزر في السطر الأول وذلك بإعطاء *gridwidth* القيمة 8. سُمح للزر "*therefere*" بالتمدد شاقولياً وذلك بإعطاء المتحول *fill* القيمة *VERTICAL* والمتحول *weighty* القيمة 1. وهذا الزر هو آخر مكونة على السطر الثاني لأن *gridwidth* قد أعطي القيمة *REMINDER*. سمح للزر "*I*" على السطر الثالث بالتمدد أفقياً وعمودياً بإعطاء المتحول *fill* القيمة *BOTH*، يعطي السطر الأخير القيمة *REMINDER* للمتحول *width* وبذلك فهو لايحتوي إلا على مكونة وحيدة. يبقى هذا الزر في المنتصف لأن *CENTER* هي القيمة الافتراضية للمتحول *anchor* ويحافظ على أبعاده لأن القيمة الافتراضية للمتحول *fill* هي *NONE*.

6-9 مثال: مقتطفات Quotes

يوضـــح البرنمج التالي استعمال الأنواع المختلفة للأزرار. يوضح الشكل 9-22 خرج هذا البرنمج. تكتب المقتطفات باستعمال لصاقة. تفيد الأزرار المختلفة في تغيير العبارة نفسها أو تغيير طريقة عرضها. يحدد صنف العبارة (فكاهي، فلسفي، تجاري) بسلسلة من أزرار الراديو في الوسط. يتحكم زرا دفع بمقاس الأحرف: صغيرة أم كبيرة. ويسمح زر اختيار موجود في الزاوية اليسارية السفلى باختيار بنط كتابة العبارة. كذلك يحدد زرا تحديد النمط المستعمل في كتابة العبارة. يستعمل البرنمج الترتيب الشبكي والتدفقي.

هـــذا البرنامج هو مثال موسع لنموذج البرامج المعتمدة على *GUI*. يقسم البرنامج إلى أربعة صـــفوف. الصـــف الأول هو البرنمج الذي يحدد الأوامر المتاحة في هذا البرنامج والطرائق المرتبطة بكل أمر. *Bold* هو أمر بسيط يؤدي إلى كتابة العبارة بالنمط الغامق. الصف الثاني هو *Quote_GUI* الذي ينشئ أغراضاً للمكونات *GUI* ويبني هرمية المكونات. يحقق الصفان الثالـــث والـــرابع عـــدة أنواع من متنصتات الأحداث التي يستعملها الصف *Quote_GUI*. تستعمل صفوف التنصت متحولاً خاصاً لتخزين الأمر الواجب تنفيذه. يختلف هذا المثال عن ســـابقه بأن الصف *Quote_GUI* ينشئ غرضاً منفصلاً لكل نوع من المتنصات ويربط كل غرض بمكونة *GUI* خاصة.

```
import java.applet.Applet ;
import java.awt.*;
import java.awt.event.*;

// Demonstrates several types of events that interact to
// change and highlight several quotes.
public class Quotes extends Applet {

   String[] text = {"Take my wife, please.",
                    "We move our legs ourselves.",
                    "Measure twic, cut once."} ;
   Label quote = new Label (text[0], Label.CENTER) ;

   final public static int UPPERCASE  = 1;
   final public static int LOWERCASE = 2;
   final public static int FONT       = 3;
```

```
final public static int COMEDY       = 4;
final public static int PHILOSOPHY = 5;
final public static int CARPENTRY= 6;
final public static int BOLD             = 7;
final public static int ITALIC       = 8;

Quote_GUI gui = new Quote_GUI (this) ;

 public void init () {
   quote.setFont (new Font ("Times", Font.PLAIN, 12)) ;
   gui.init (quote) ;
} // method init


public void  set_uppercase () {
     quote.setText (quote.getText().toUpperCase () ) ;
} // method set_uppercase

public void set_lowercase () {
  quote.setText (quote.getText ().toLowerCase () );
} // method set_lowercase

public void set_comedy ()  {
  quote.setText ( text[0] );
} // method set_comedy

public void set_philosophy ()  {
  quote.setText ( text[1]);
} // method set_philosophy

public void set_carpentry ()  {
  quote.setText ( text[2]);
} // method set_carpentry

public void set_font (String font )  {
  quote.setFont ( new Font (font, quote.getFont().getStyle(), 12));
} // method set_font

public void set_italic (boolean on )  {
   Font font = quote.getFont() ;
   int style   = font.getStyle () ;

 // Turn italic on or off if not already set.
 if (on) {
    if (! font.isItalic() )
       style += font.ITALIC;
```

```
    } else {
      if ( font.isItalic () )
          style -= Font.ITALIC;
    }
    quote.setFont (new Font (font.getName(), style, 12)) ;
    } // method set_italics

    public void set_bold (boolean on) {
      Font font = quote.getFont() ;
      int style  = font.getStyle() ;

    // Turn bold on or off if not already set.
    if (on) {
       if(! font.isBold () )
           style += font.BOLD;
    } else {
       if (font.isBold ())
           style -= Font.BOLD;
     }
     quote.setFont (new Font (font.getName(), style, 12)) ;
    } // method set_bold
} // class Quotes

//constructs the graphical user interface and associates went listeners
class  Quote_GUI {
    private Quotes applet;

    private  Panel letters = new Panel () ;
    private Panel domain = new Panel () ;
    private Panel style = new Panel () ;

    private Button uppercase = new Button ("Uppercase" ) ;
    private Button lowercase = new Button ("Lowercase" ) ;
    private Choice font_choice = new Choice () ;

    private CheckboxGroup topic = new CheckboxGroup () ;
    private Checkbox comedy =
        new Checkbox ("Comedy", topic, true ) ;
    private Checkbox philosophy =
        new Checkbox ("Philosophy", topic, false ) ;
    private Checkbox carpentry =
        new Checkbox ("Carpentry", topic, false ) ;

    private Checkbox bold = new Checkbox ("Bold" ) ;
     private Checkbox italic = new Checkbox ("Italic") ;
```

```
private Quote_Action_Listener uppercase_listener;
private Quote_Action_Listener lowercase_listener;
private Quote_Item_Listener bold_listener;
private Quote_Item_Listener italic_listener;
private Quote_Item_Listener font_listener;
private Quote_Item_Listener comedy_listener;
private Quote_Item_Listener philosophy_listener;
private Quote_Item_Listener carpentry_listener;

public Quote_GUI (Quotes quote_applet) {
  applet = quote_applet;
  uppercase_listener =
     new Quote_Action_Listener (applet, Quotes.UPPERCASE) ;
  lowercase_listener =
      new Quote_Action_Listener(applet,Quotes.LOWERCASE) ;
  bold_listener =
      new Quote_Item_Listener (applet, Quotes.BOLD) ;
  italic_listener =
     new Quote_Item_Listener (applet, Quotes.ITALIC) ;
  font_listener =
     new Quote_Item_Listener (applet, Quotes.FONT) ;
  comedy_listener =
     new Quote_Item_Listener (applet, Quotes.COMEDY) ;
  philosophy_listener =
     new Quote_Item_Listener (applet, Quotes.PHILOSOPHY) ;
  carpentry_listener =
     new Quote_Item_Listener (applet, Quotes.CARPENTRY) ;
  } // constructor Quote_GUI

  public void init (Label quote) {

    // setup the component hierarchy
    font_choice.addItem ("Courier") ;
    font_choice.addItem ("Times") ;
    font_choice.addItem ("Helvetica") ;

    letters.setLayout (new GridLayout (4,1,3,3) );
    letters.add (uppercase) ;
    letters.add (lowercase) ;
    letters.add (font_choice) ;

    domain.setLayout (new GridLayout (4,1)) ;
    domain.add (comedy) ;
    domain.add (philosophy) ;
    domain.add (carpentry) ;
```

```
        style.setLayout (new GridLayout (4,1) ) ;
        style.add (bold) ;
        style.add (italic) ;

        applet.setLayout (new BorderLayout (10,2) ) ;
        applet.add ("North", quote) ;
        applet.add ("West", letters) ;
        applet.add ("Center", domain) ;
        applet.add ("East", style) ;

        applet.resize (260,150) ;

        // add all assoiciated listeners
        uppercase.addActionListener (uppercase_listener) ;
        lowercase.addActionListener(lowercase_listener) ;
        font_choice.addItemListener(font_listener) ;
        comedy.addItemListener(comedy_listener) ;
        philosophy.addItemListener(philosophy_listener) ;
        carpentry.addItemListener (carpentry_listener) ;
        bold.addItemListener (bold_listener) ;
        italic.addItemListener (italic_listener) ;
    } // method init
} // class Quote_GUI

// Handles the item events for the domain of the quote,
// style, and font.
class Quote_Item_Listener implements ItemListener {

    private Quotes applet ;
    private int command ;

    public Quote_Item_Listener (Quotes listening_applet,
                                int listening_command) {
      applet = listening_applet ;
      command = listening_command ;
    } // constructor Quote_Item_Listener

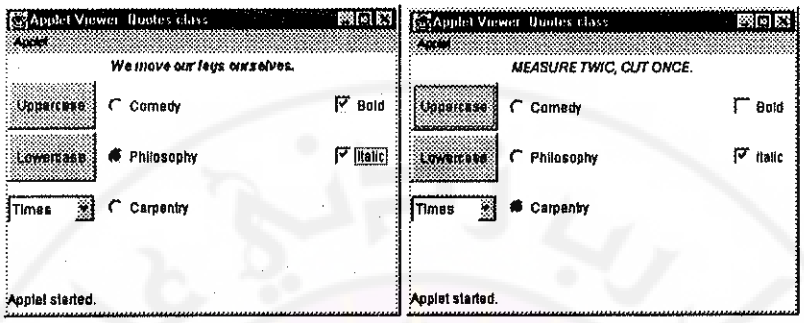
    public void itemStateChanged (ItemEvent event) {
      switch (command) {
          case Quotes.FONT:
              applet.set_font ( ( String) (event.getItem() ) ) ;
               break ;
          case Quotes.COMEDY:
              applet.set_comedy () ;
               break ;
          case Quotes.PHILOSOPHY:
```

```
            applet.set_philosophy () ;
             break ;
         case Quotes.CARPENTRY:
            applet.set_carpentry () ;
             break ;
         case Quotes.BOLD:
            applet.set_bold
               (event.getStateChange () == ItemEvent.SELECTED) ;
             break ;
         case Quotes.ITALIC:
            applet.set_italic
              (event.getStateChange () == ItemEvent.SELECTED) ;
             break ;
         default:
           System.out.println ("Illegal Item" +
               event.toString () ) ;
         }
    } // method itemStateChanged
} // class Quote_Item_Listener

// Handles the action events for the Uppercase and Lowercase
// buttons
class Quote_Action_Listener implements ActionListener {

    private Quotes applet ;
    private int command ;
    public Quote_Action_Listener (Quotes listening_applet,
                                  int listening_command) {
        applet = listening_applet ;
        command = listening_command ;
    } // constructor Quote_Action_Listener

    public void actionPerformed (ActionEvent event) {
        switch (command) {
          case Quotes.UPPERCASE:
            applet.set_uppercase () ;
            break ;
          case Quotes.LOWERCASE:
            applet.set_ lowercase () ;
            break ;
          default:
            System.out.println ("Illegal Action" +
               event.toString () ) ;
          }
    } // method actionPerFormed
} // class Quote_Action_Listener
```



الشكل 9-22: البرنامج *quote*

الفصل 10 : التحكم المتقدم بسير البرنامج

تقـــدم جافـــا تقنيتين للتحكم في تدفق تنفيذ برنامج ما، هما الاستثناءات والمسالك. تسمح الاســـتثناءات بـــتعريف تدفق تنفيذي لمعالجة المواقف غير الطبيعية أو الخاطئة، وتسمح لنا المسالك بتعريف عدة تدفقات تنفيذ تعمل في آن واحد.

10-1 معالجة الاستثناءات (Exceptions)

ينتج عن المشاكل التي تظهر في برنامج جافا أخطاءً واستثناءات. الاستثناء هو غرضٌ يُعرِّف حالة غير طبيعية. يُرسِل البرنامج أثناء التنفيذ استثناءً يمكن التقاطه ومعالجته.

تشبه الأخطاء الاستثناءات ولكنها لا تُلتقط ولا يمكن الاستمرار في العمل بعد حدوثها.

تمتلك جافا مجموعة من الأخطاء والاستثناءات المُعَرّفة سلفاً، والتي يمكن أن تحدث أثناء تنفيذ بـــرنامج مـــا. يُعرِّف المفهوم العام للاستثناءات والأخطاء في الصفوف *Exception* و*Error* المشـــتقة من الصف *Throwable*. ويُشتق من هذين الصفين استثناءات وأخطاء خاصة (انظر المـــلحق أ). وكما رأينا في حديثنا عن *C++* فإن الاستثناءات هامة لأنها تقسم البرنامج إلى تدفقين: طبيعي واستثنائي.

هنالك ثلاث طرائق للتعامل مع الاستثناءات في جافا:

- عدم معالجتها نهائياً
- معالجتها في أماكن حدوثها
- معالجتها في مكان آخر

في حال عدم معالجة الاستثناء فإن البرنامج يتوقف توقفاً طبيعياً، ويعطي رسالة تشرح نوع الاستثناء. يرسل البرنامج في المثال التالي الاستثناء *ArithmeticException* الذي يعني أن عملية

حسابية غير طبيعية قد حدثت مثل التقسيم على صفر في حالتنا هذه:

```
// Produce an exception
public class Zero {

    public static void main (String[] args) {
        int numerator = 10;
        int denomerator = 0;
        System.out.println (numerator / denomerator);
    } // method main

} // class Zero
```

عند تنفيذ البرنامج السابق، يتوقف عند القسمة على صفر ويعطي الرسالة:

```
java.lang.ArithmeticException: / by zero
  at Zero.main(Zero.java:7)
```

يحدد السطر الأول نوع الاستثناء وهو هنا *ArithmeticException*، ويحدد أيضاً سبب حدوث الاستثناء وهو القسمة على صفر المعبر عنها بالرسالة *./ by zero*. يعطي السطر الثاني حالة المكدس أثناء الاستدعاء الذي يحدد مكان حدوث استثناء، بإعطاء اسم الطريقة والملف ورقم السطر. نستطيع أيضاً الحصول على هذه المعلومات باستعمال الطريقتين *getMessage()* و *printStackTrace()* المعرفتين في الصف *Throwable*. تطبع الطريقة *printStackTarce()*، مكدس الاستدعاء على القناة *System.err*.

تُعالج الاستثناءات كما في حال *C++* باستعمال الكتلة *try ... catch* التي لها الشكل العام التالي:

```
try {
      statement-list-1
    }
catch (exception-class1 variable-1)
    {
     statement-list-2
    }
.
.
.
catch (exception-class-n variable-n)
    {
     statement-list-n
    }
```

يوجد في جافا استثناءات مراقَبة *checked* وأخرى غير مراقبة *unchecked*. يجب التصريح عن الاستثناءات المراقبة وإن لم تُلتقط وتعالج. إذا كان من المحتمل أن ترسل طريقة ما استثناءً فإن عليها أن تلتقطه وتعالجه، أو أن تصرح عنه بحيث يُرسل آلياً إلى المستوى الأعلى، وذلك بإضافة عبارة *throws* إلى ترويستها: على سبيل المثال ترسل الطريقة *System.in.read()* الاستثناء *IOException* فإذا لم نلتقطه ونعالجه فعلينا أن نضيف إلى الطريقة *main* عبارة *throws IOException* كما في المثال التالي:

```
class ExceptionTest {
   public static void main(String[] args) throws IOException
       {
        System.out.println (System.in.read());
       }
}
```

تُستعمل عبارة *throw* لإرسال الاستثناءات. وفي حين يمكن للغة *C++* أن ترسل أي غرض من أي صف كان، فإن جافا لا ترسل إلا الأغراض التي تنتمي إلى صفوف مشتقة من الصف *Throwable* مباشرةً أو بطريقة غير مباشرة كما يلي:

```
class ThrowableClass extends throwable
{
....
}
```

اختلاف آخر بين استثناءات جافا واستثناءات *C++* هو عبارة *finally*. الشكل العام لعبارة *try* التي تحوي عبارة *finally* هو:

```
try {
     statement-list-1
   }
catch (exception-class1 variable-1)
   {
    statement-list-2
   }


catch (exception-class-n variable-n)
   {
    statement-list-n
```

```
}
finally
  {
   statement-list-n+1
  }
```

تُـــنفذ التعلـــيمات الموجودة في عبارة *finally* سواءً حدث استثناءً أم لا. فإذا استمر تنفيذ البرنامج تنفيذاً طبيعياً أو عولج الاستثناء ضمن عبارة *catch* ما انتقل التنفيذ بعدها إلى عبارة *finally* ثم إلى العـــبارات الأخـــرى في الطريقة المعنية. فإن حدث استثناء غير معالج ضمن الطريقة الحالية فإن تعليمات *finally* ستنفذ قبل انتقال التحكم إلى الطريقة الطالبة.

2-10 تعدد المسالك (multi-threading)

تنفذ تعليمات البرنامج عادةً بالتسلسل الواحدة تلو الأخرى. المسلك هو آلية تسمح بتنفيذ أكثر من مهمة في آنٍ واحد ضمن البرنامج الواحد. يمكن أن تنفّذ المهمتان في آن واحد (على التوازي) في حال وجود أكثر من وحدة معالجة مركزية، كما يمكن أن تتنافسا على وحدة المعالجة المركزية الوحيدة المتوفرة على الحاسوب. تسمى البرامج التي تمتلك أكثر من مسلك، مـــتعددةَ المسالك *multithreaded*. ومع أن البرامج المتعددة المسالك أكثر تعقيداً من غيرها، فإنها فعالة جداً وتسمح للبرامج بمحاكاة فعاليات حقيقية لأشخاص أو لأشياء تعمل في إطار واحد في نفس الوقت. تتميز جافا عن *C++* بامتلاكها إمكان إنشاء المسالك في صلبها، ولا تحتاج إلى مكتبات خاصة تختلف باختلاف نظام التشغيل.

1-2-10 إنشاء المسالك

يمكـــن إنشاء المسالك في جافا بإحدى طريقتين. تقوم الأولى على اشتقاق صف جديد من الصف *Thread* وإعادة تعريف الطريقة *run* التي يجب أن تحتوي على العمليات المراد تنفيذها ضمن المسلك. تقوم الطريقة الثانية على إنشاء صف يحقق الواجهة *Runnable* (التي تقتضي تعريف طريقة تسمى *run*) وتمرير غرض من هذا الصف إلى باني غرض من الصف *Thread* مُنشـــأ حديـــثاً. تسمح التقنية الثانية باشتقاق الصف، المطلوب تنفيذ محتواه ضمن مسلك

مستقل، من صف آخر مختلف عن الصف *Thread*. التركيبان التاليان متكافئان ويؤديان نفس الغرض:

// First solution *class MyThread extends Thread {* *public void run() { ..code...}* *}* *MyThread myThread;*	*// Second solution* *class MyThread implements Runnable {* *public void run { ...code... }* *}* *Thread myThread(new MyThread());*

باستعمال الأسلوب الأول ينشئ المثال التالي ثلاثة مسالك تقوم بطباعة سلسلة محرفية 100 مرة.

```
// Creates three threads that run concurently.
public class Simultaneous {

    public static void main (String[] args) {
        Soda one = new Soda ("Coke");
        Soda two = new Soda ("Pepsi");
        Soda three = new Soda ("Diet Coke");

        one.start();
        two.start();
        three.start();
    } // method main.

} // class Simultaneous

// A thread that prints a message 100 times.
class Soda extends Thread {

    private String name;

    Soda (String str) {
        name = str;
    } // constructor Soda

    public void run() {
        for (int count = 0; cout < 100; count++)
            System.out.println (name);
    } // method run

} // class Soda
```

كل مسلك من المسالك الثلاثة السابقة هو نسخة من الصف *Soda* المشتق من *Thread*. يبدأ المسلك في العمل عند استدعاء الطريقة *start* التي تسبب استدعاء الطريقة *run* ضمن مسلك جديد. تعمل المسالك الثلاثة معاً ويطبع كلٌ منها عبارته الخاصة، ويعطي الشكل التالي خرجاً محتملاً:

```
Coke
Coke
Pepsi
Diet Coke
Pepsi
Pepsi
...
```

بالطبع، هذا ليس كل شيء عن المسالك، فهي تتسبب في تعقيد البرنامج، وهذا ما سنناقشه في تتمة هذا الفصل.

2-2 مشكلة مع المسالك

من الهام للعديد من التطبيقات امتلاك إمكان تقسيم البرامج إلى عدة أجزاء تعمل معاً، إلا أن ذلك يعقّد تحقيق هذه البرامج. في المثال السابق كان هنالك عدة مسالك لكل منها معطياته الخاصـــة، ولكن عندما تتشارك المسالك في معطياتٍ ما، يجب الانتباه إلى أن هذه المعطيات تُعَامل بوجهٍ صحيح. لنتناول المثال التالي:

يمتلك زوجٌ وزوجُه حساباً مشتركاً في البنك، ويمكنهما استعمال بطاقة ائتمان لسحب المال مـــن آلـــة *ATM* (*Automatic Teller Machine*). لنفترض أن كليهما سحبا المال من آلتين مختلفتين في نفس الوقت، وأنه يهتم مسلك من برنامج التخديم بكل آلة ويتشارك المسلكان في نفس المعطيات. إذا لم يكن البرنامج مكتوباً بعناية فقد يحدث الموقف التالي.

- يطلـــب الزوج مبلغاً من المال وليكن 100 دولار وتطلب الزوجة مبلغاً من المال وليكن 100 دولار أيضاً.
- تتحقق الآلة الأولى من توفر المبلغ وليكن هنالك 500 دولار وتتحقق الآلة الثانية

من توفر المبلغ وتجد أيضاً أن هنالك 500 دولار.

- تعطـــي الآلـــة الأولى المبلغ إلى الزوج وتنقصه من الرصيد الأصلي ليصبح الرصيد 500-100 أي 400 دولار وتضـــبط رقم الحساب ليصبح 400 دولار بدلاً من 500. وتعطي الآلة الثانية 100 دولار إلى الزوجة وتنقصه من المبلغ الأصلي ليصبح 400 دولار وتضبط رقم الحساب ليصبح 400 دولار بدلاً من 500.

نلاحظ في هذا المثال أن المبلغ المسحوب هو 200 دولار، ومع ذلك فالمبلغ المتبقي هو 400 ولـــيس 300 دولار. ومنه فقد يتسبب تعدد المسالك بوجود ذاكرة مشتركة بسلوك غير سليم للبرامج يجب الانتباه له، وإيجاد الحلول المناسبة كما يبين ذلك الجزء التالي من الفصل.

3-2-10 المزامنة (Synchronization)

يمكـــن حل مشكلة تشارك المعطيات إذا استطعنا أن نضمن عدم استعمال هذه المعطيات من قـــبل أكثر من مسلك في آنٍ واحد. تسمح لنا جافا بمزامنة تنفيذ طريقة ما. فإذا ما أضيف المعدِّل *synchronized* إلى تعريف طريقة ما، فلن يمكن تنفيذها من قبل أكثر من مسلك في آن واحد. ولا يمكننا تحديد المسلك الذي سيستعمل الطريقة، وإنما نضمن فقط أن مسلكاً ما لن يستعمل الطريقة ما لم ينتهي منها المسلك الذي يستعملها في الوقت الحالي إن وُجد.

يحـــاكي البرنامج التالي مسألة الحساب البنكي الذي ناقشناه في المقطع الأخير. إذا استعملنا المعدِّل *synchronized* مع الطريقة *withdrawal* فإن العمليات ستكون آمنة.

```
// Manages savings accounts using ATMs.
public class ATM_Accounts {

    public static void main (String[] args) {
        Savings_Account savings = new Savings_Account (4321, 531);

        ATM west_branch = new ATM (savings);
        ATM east_branch = new ATM (savings);

        west_branch.start();
        east_branch.start();
    } // method main
```

```
} // class ATM_Accounts

// Represents a single savings account with the ability
// to make withdrawals.
class Savings_Account {
    protected int account;
    protected int balance;

    public Savings_Account (int account_num, int initial) {
        account = account_num;
        balance = initial;
    } // constructor Savings_Account

    public synchronized boolean withdrawal (int amount) {
        boolean result = false;
        System.out.println ("Withdrawal from account "+ account);
        System.out.println ("Amount: "+ amount);

        if (amount <= balance) {
           balance -= amount;
           System.out.println ("New balance: "+ balance);
           result = true;
        } else
            System.out.println ("insufficient funds.");

        System.out.println ();

        return result;
     } // method withdrawal

} // class Savings_Account

// A thread that makes a withdrawal.
class ATM extends Thread {

    Savings_Account account;

    ATM (Savings_Account savings) {
        account = savings;
    } // constructor ATM

    public void run() {
        account.withdrawal (300);
    } // method run
```

```
} //class ATM
```

فيما يلي خرجٌ محتمل لبرنامجنا هذا:

```
Withdrawal from account 4321
Amount: 300
New balance: 231
Withdrawal from account 4321
Amount: 300
insufficient funds.
```

لا نعـــلم أي مسلك سيبدأ العمل، ولكن المعدِّل *synchronized* يضمن عدم تداخل الرسائل لأنه لا يمكن تنفيذ الإجرائية *withdrawal* إلا من قبل مسلك واحد.

عندما يبدأ مسلك ما بتنفيذ الطريقة فإن الغرض المعني يصبح مقفولاً *locked* ولن يستطيع أي مســـلك آخـــر تنفيذ أي طريقة متزامنة على ذلك الغرض، وهذا يسمح بكتابة عدة طرائق تتعامل مع معطيات مشتركة.

10-2-4 التحكم في المسالك

يوجد عدة طرائق معرفة ضمن الصف *Thread* تتحكم في عمل المسالك نذكر منها:

- *stop()*: تنهي عمل المسلك نهائياً.
- *suspend()*: توقف عمل المسلك مؤقتاً.
- *resume()*: تـــتابع عمـــل مسلك، كان متوقفاً عن العمل، انطلاقاً من النقطة التي توقف عندها.
- *sleep(long millisecond)* توقف عمل مسلك مدةً معينة.

لنَعُد إلى مثال الكرة الوثابة المذكور في الفصل الثامن، ولنجعل منه برنامجاً ذا مسالك ولنضع الإحياء في مسلك خاص يمكننا التحكم فيه. كنا، على سبيل المثال، نحقق التوقف باستعمال حلقة *for* بدون متن، وكان علينا التحكم في عدد التكرارات ليبدو الإحياء سلساً ومع ذلك، ســـيتعلق تنفيذ البرنامج بسرعة المعالج في تنفيذ حلقة *for* هذه. نستطيع الآن الاستعاضة عن

حلقة *for* بالطريقة *sleep* الخاصة بالمسلك، وهذا ما يؤدي إلى توقفه مدةً زمنية ثابتة لا تتعلق بسرعة المعالج.

نستطيع أيضاً إضافة إطار ذي زرّين *suspend* و*resume* يستدعيان الطريقة الموافقة لإيقاف البرنامج عن العمل، وجعله يستمر ثانية من النقطة التي توقف عندها. وفيما يلي البرنامج المعدل:

```
import java.awt.event.*;
import java.applet.Applet;
import java.awt.*;

// Simulates a ball on a rubber band.
public class Bouncing_Ball2 extends Applet {

   private final int SIZE = 300;

  private Ball ball = new Ball(150, 10, 250, 200);
  private Graphics page;

  private Control_Panel controls;

  public void init() {
    setVisible(true);
    setSize(SIZE, SIZE);
    page = getGraphics();
   page.setXORMode (getBackground () ) ;
  } // method init

    public void start () {
        controls = new Control_Panel (Thread.currentThread());
        controls.start();
        ball.pause();
       while (ball.moving () )
            ball.bounce (page) ;
    } // method start

} // class Bouncing_Ball2

// Various buttons that suspend and resume execution of the
// applet thread.
class Control_Panel extends Thread {

     private Button suspend = new Button ("suspend");
```

```
    private Button resume = new Button ("resume");
    private Frame frame = new Frame ("Bouncing Ball Control Panel");

    private Thread applet_thread;

    Control_Panel (Thread applet_thread) {
        this.applet_thread=applet_thread;
    } // constructor Control_Panel

    public void run() {
        Resume_Action resume_action =
                new Resume_Action (applet_thread);
        Suspend_Action suspend_action =
                new Suspend_Action (applet_thread);
        suspend.addActionListener (suspend_action);
        resume.addActionListener (resume_action);

        frame.setLayout (new FlowLayout());
        frame.add (suspend);
        frame.add (resume);
        frame.pack();
        frame.setLocation (250, 200);
        frame.setVisible (true);
    } // method run
} // class Control_Panel

// Action for the suspend button
class Suspend_Action implements ActionListener {

    Thread applet_thread;
    Suspend_Action (Thread applet_thread) {
        this.applet_thread=applet_thread;
    } // constructor Suspend_Action

    public void actionPerformed(ActionEvent action) {
        applet_thread.suspend();
    } //method actionPerformed
} // class Suspend_Action

// Action for the resume button
class Resume_Action implements ActionListener {

    Thread applet_thread;
    Resume_Action (Thread applet_thread) {
        this.applet_thread=applet_thread;
    } // constructor Resume_Action
```

```
    public void actionPerformed(ActionEvent action) {
        applet_thread.resume();
    } //method actionPerformed
} // class Resume_Action

// Draws the ball and simulates its movement.
class Ball {

    private final int MOVE =2;
    private final float DISTANCE = 0.97f ;
    private final int SIZE = 20;
    private final int PAUSE = 5;

    private int x;
    private int start_y;
    private int end_y;
    private int length;

    private boolean moving_up = true;

    public Ball (int new_x, int new_start_y,
            int new_end_y, int new_length) {
        x = new_x ;
        start_y = new_start_y ;
        end_y = new_end_y;
        length = new_length ;
    } // constuctor Ball

    public void pause() {
        try {
            Thread.currentThread().sleep (PAUSE);
        } catch (InterruptedException exception) {
            System.out.println ("have an axception");
            // ignore the exception and continue
        }
    } // method pause

    private void move () {
        if (moving_up) {
            end_y = end_y -MOVE ;
        } else {
            end_y = end_y +MOVE;
        }
    } // method move
```

```
    private void draw_ball (Graphics page ) {
        page.drawOval (x-( SIZE /2), end_y, SIZE, SIZE);
        page.drawLine (x, start_y, x, end_y) ;
    } // draw_ball

    public boolean moving () {
        return length != 0 ;
    } // method moving

    public void bounce (Graphics page) {
        for (int count = 1; count < length; count += MOVE) {
            draw_ball (page) ;
            pause();
            draw_ball (page) ;
            move () ;
        }
        moving_up = !moving_up;
        length = (int) (DISTANCE * length) ;
    } // method bounce

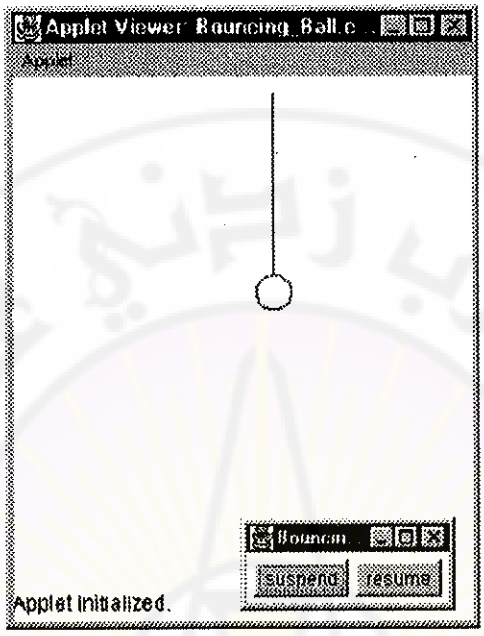
} // class Ball
```

يبين الشكل 10-1 لقطة من هذا البرنامج.

يمكـــن تنفـــيذ تعلـــيمات متزامـــنة دون استعمال طرائق متزامنة وذلك باستعمال المعدل *synchronized* ضمن التركيب النحوي التالي:

```
synchronized(expression) statement;
```

حيث يعيد التعبير *expresssion* غرضاً يقفل أثناء تنفيذ التعليمة *statement*.



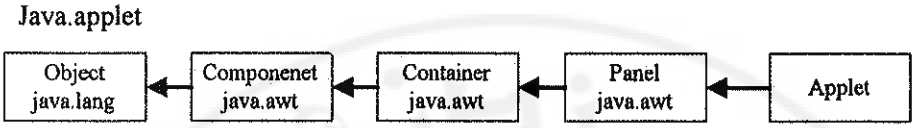
الشكل 10-1: البرنامج *Bouncing_Ball2*.

الفصل 11 : خاتمة

تناولنا في هذا الجزء من الكتاب مفاهيم وأسس لغة جافا من خلال الإصدار جافا 1.1 الأكثر شيوعاً على الوب، والمتوافق مع المتصفحات المتوفرة. هذا الإصدار من جافا هو نواة جافا التي ارتكز عليها الإصدار الأخير 1.2 والمعروف بجافا 2. أضاف الإصدار 2 إلى الإصدار السابق حزماً جديدة مبنية على الحزم القديمة لكنه بالطبع لم يغير شيئاً في لغة جافا وإنما أغناها بأدوات جديدة مثل حزمة *Swing* و إغناء الحزمة *net* بـ *Servlet*.

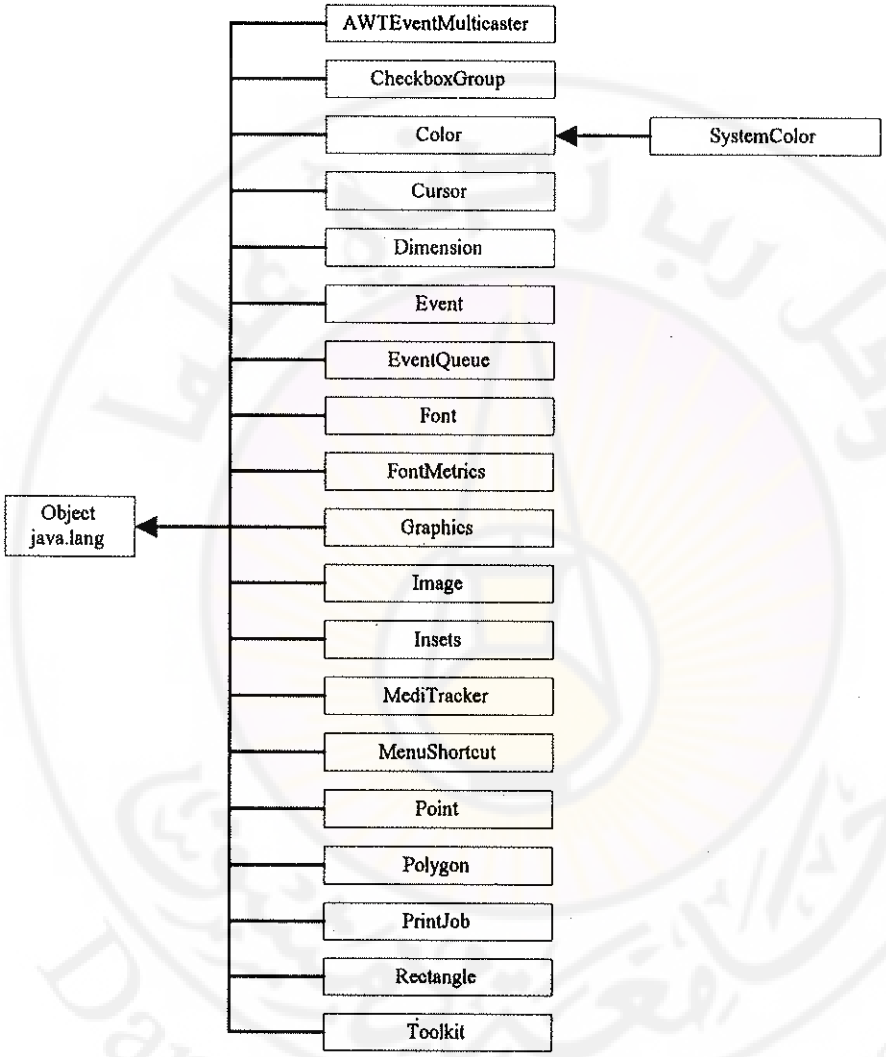
كما رأينا ليست جافا لغة برمجة فحسب، ولكنها آلة افتراضية يجب أن تؤدي جميع الوظائف التي تؤديها الآلات الفيزيائية ولهذا فدراسة جافا مع الحزم الملحقة بها تكافئ دراسة جميع المكتبات المتوفرة على نظام تشغيل ما، والتي يمكن أن تستعملها لغات البرمجة المختلفة. من هنا تظهر صعوبة تناول جميع هذه الحزم في كتاب تعليمي، وخاصة لأن بعضها مرتبط بمفاهيم قد لا يمتلكها القارئ بعد. ركز هذا الكتاب على المفاهيم الأساسية للغة جافا وعلى الحزم الخاصة بالبيانيات وبناء واجهات الاستخدام، وكذلك على مفاهيم معالجة الاستثناءات وتعدد المسالك.

الملحق أ: هيكلية صفوف جافا



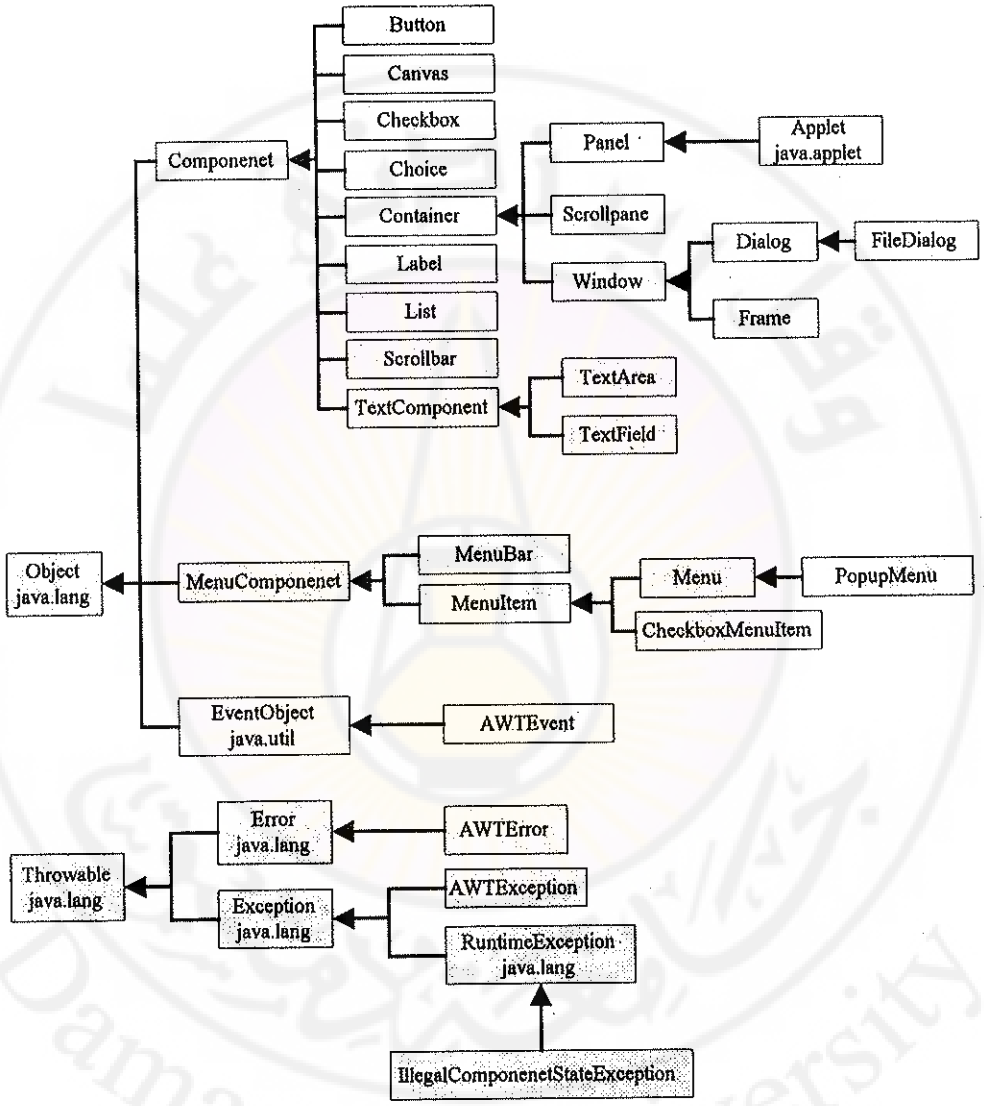
Damascus University

Java.awt (part 1 of 3)

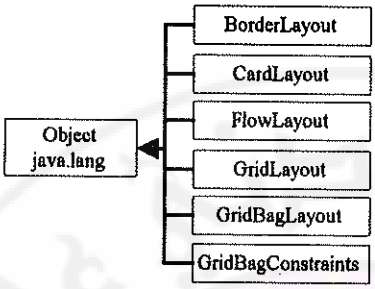


Damascus University

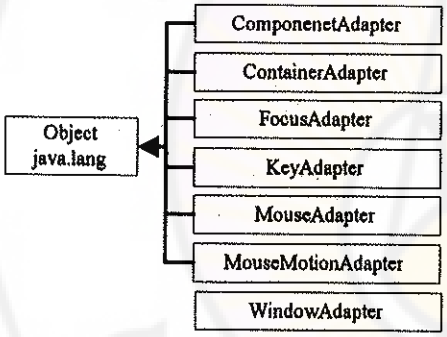
Java.awt (part 2 of 3)

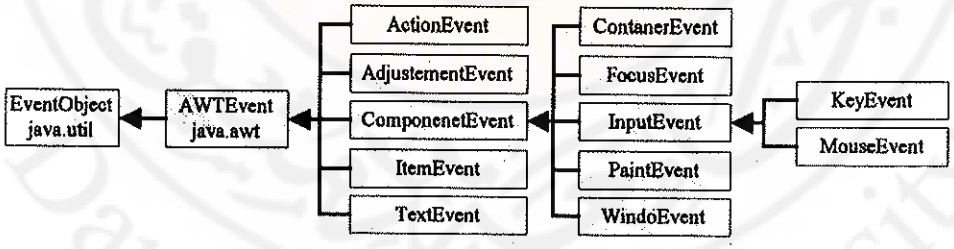


Java.awt (part 3 of 3)

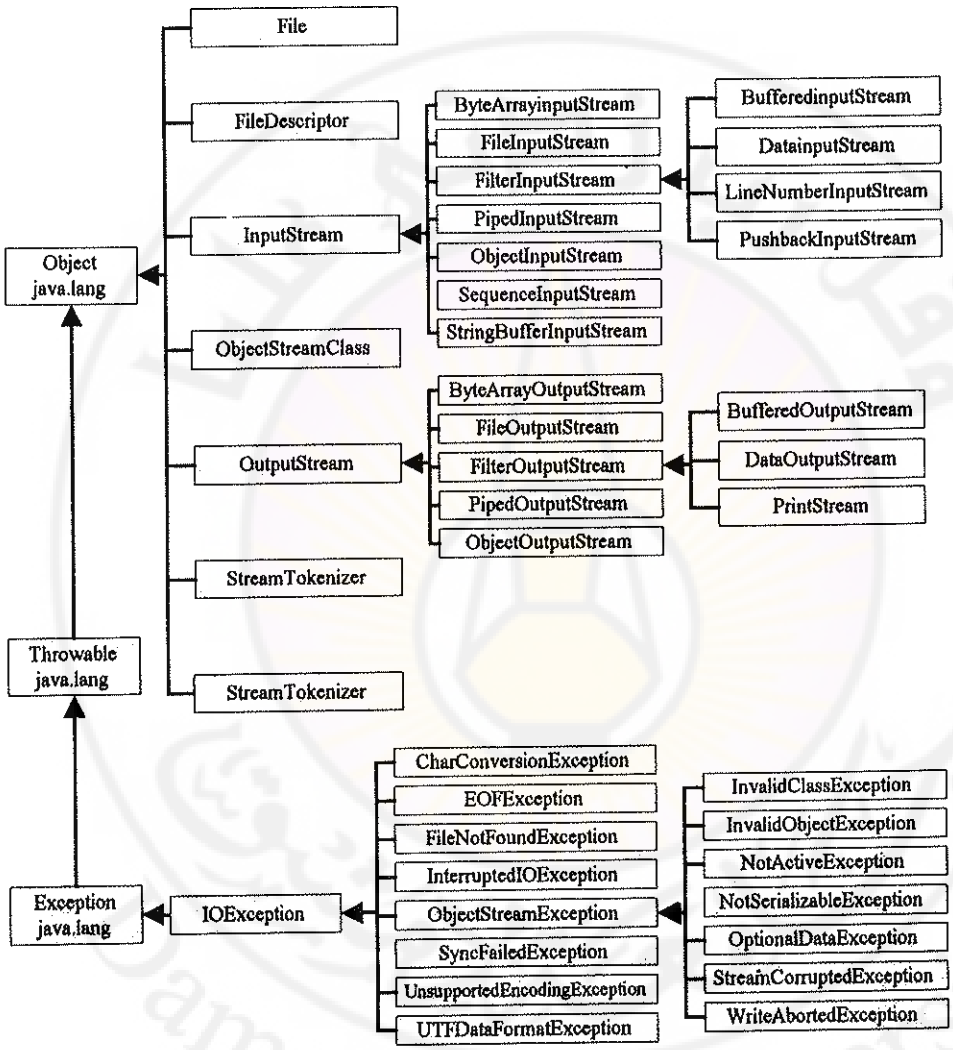


Java.awt.event



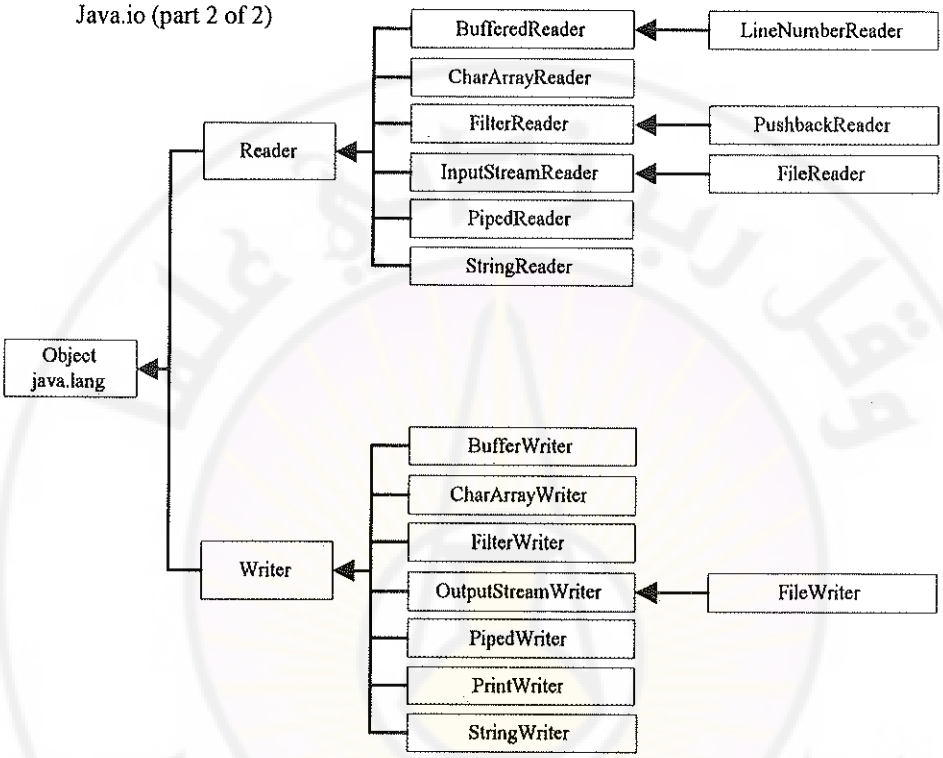


Java.io (part 1 of 2)

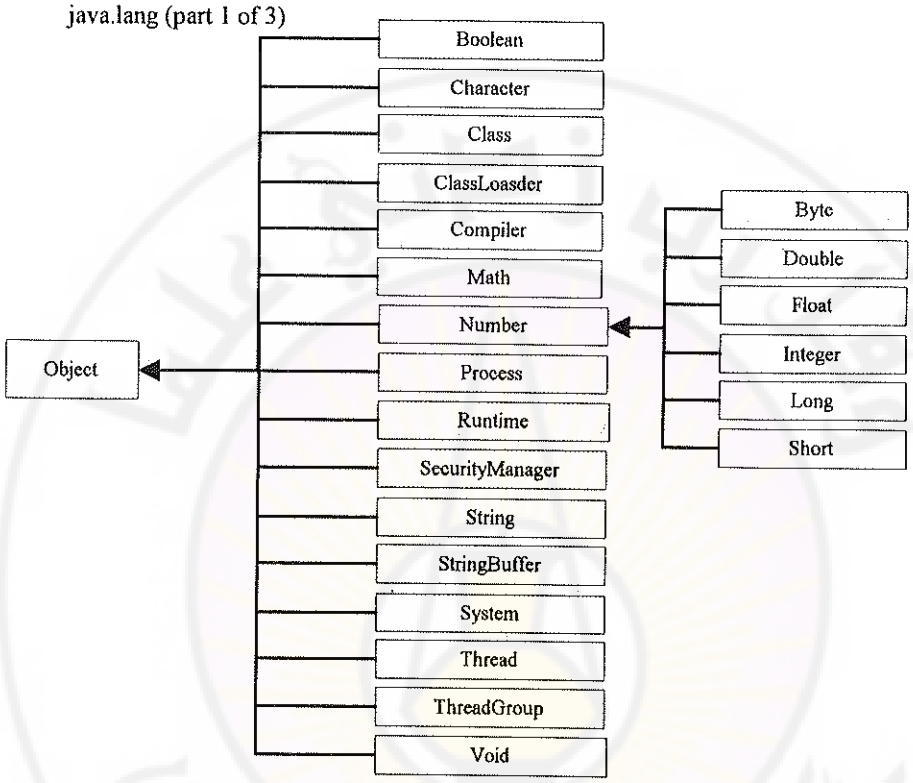


Damascus University

Java.io (part 2 of 2)

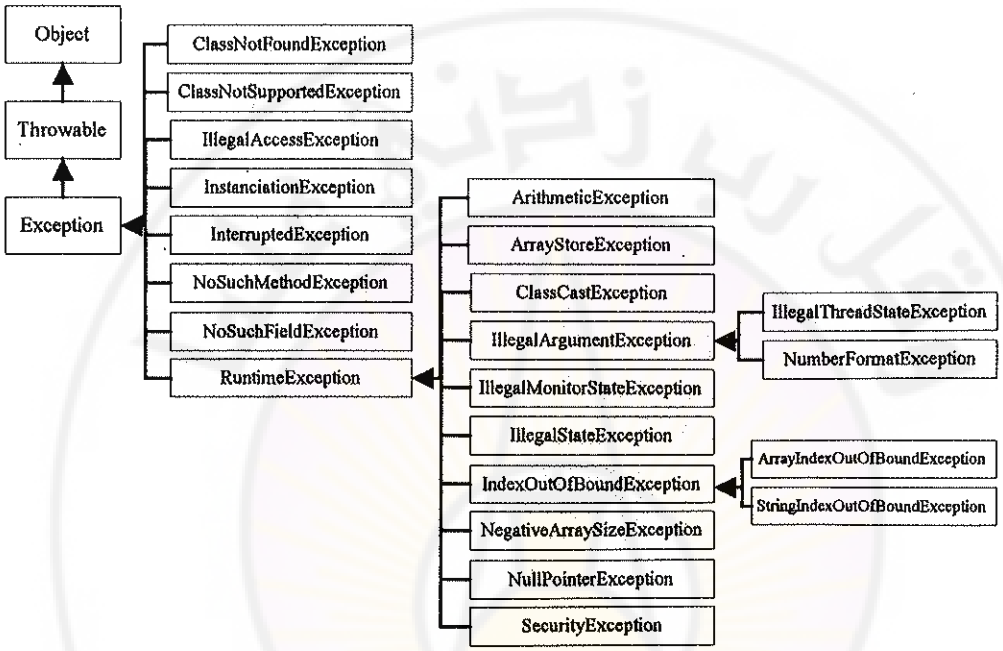


Damascus University

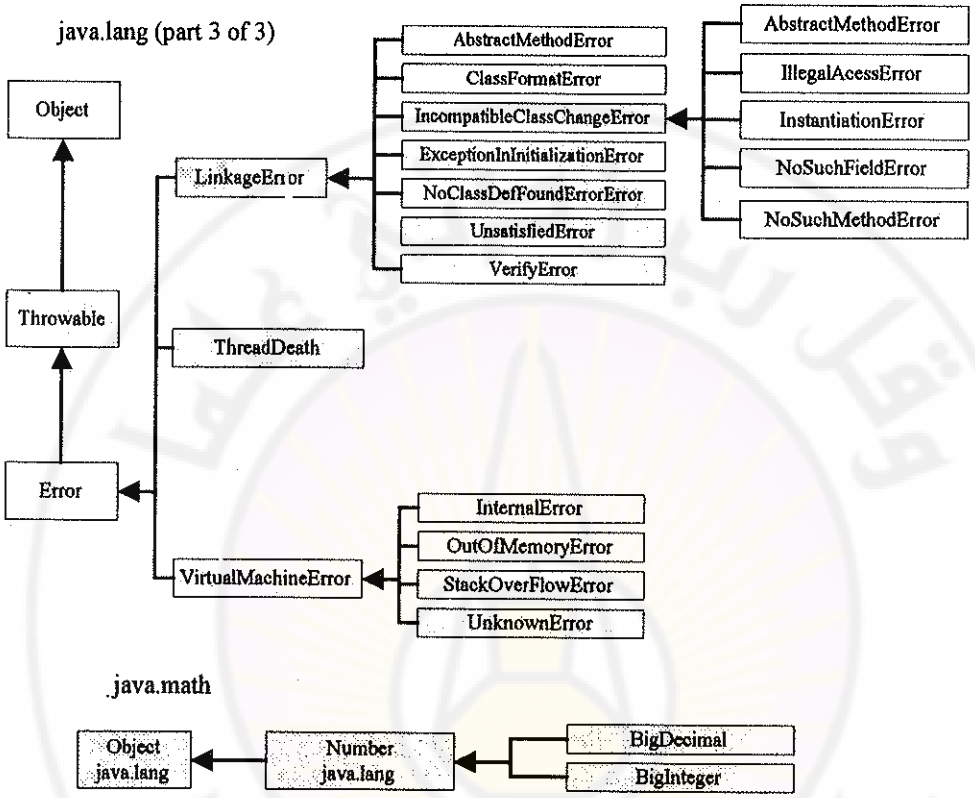


Damascus University

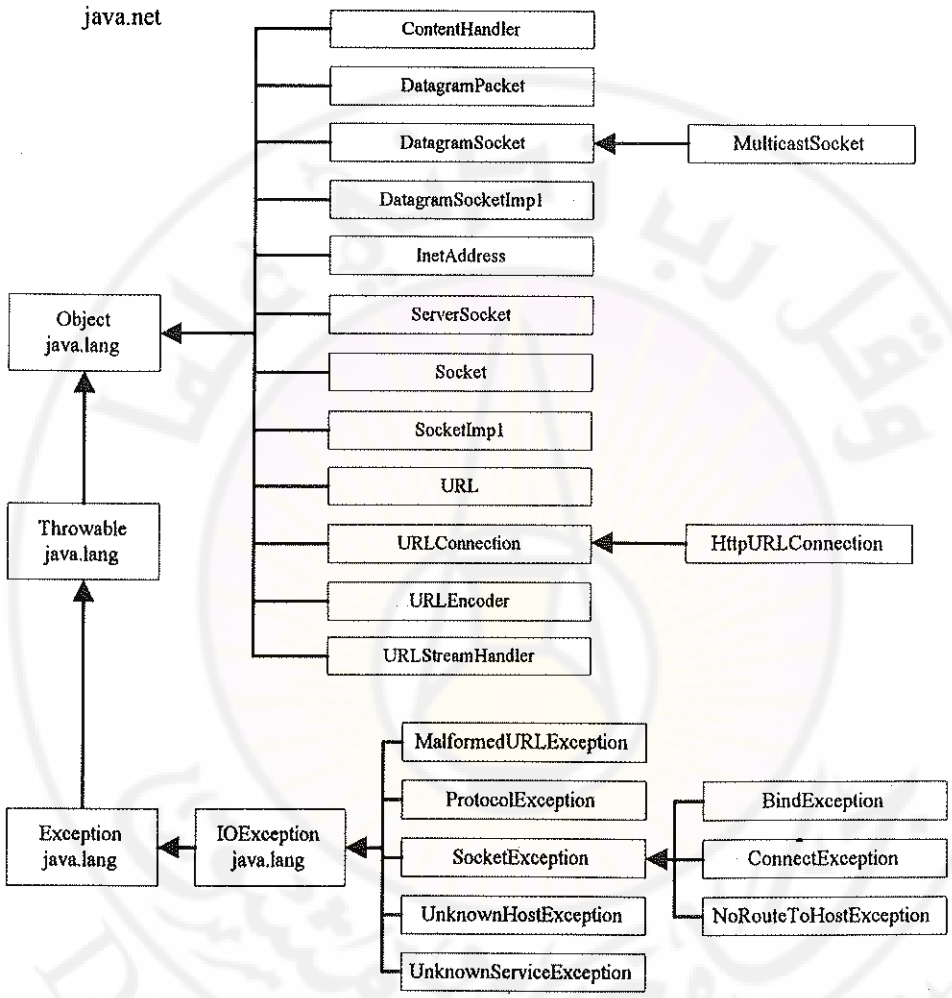
java.lang (part 2 of 3)



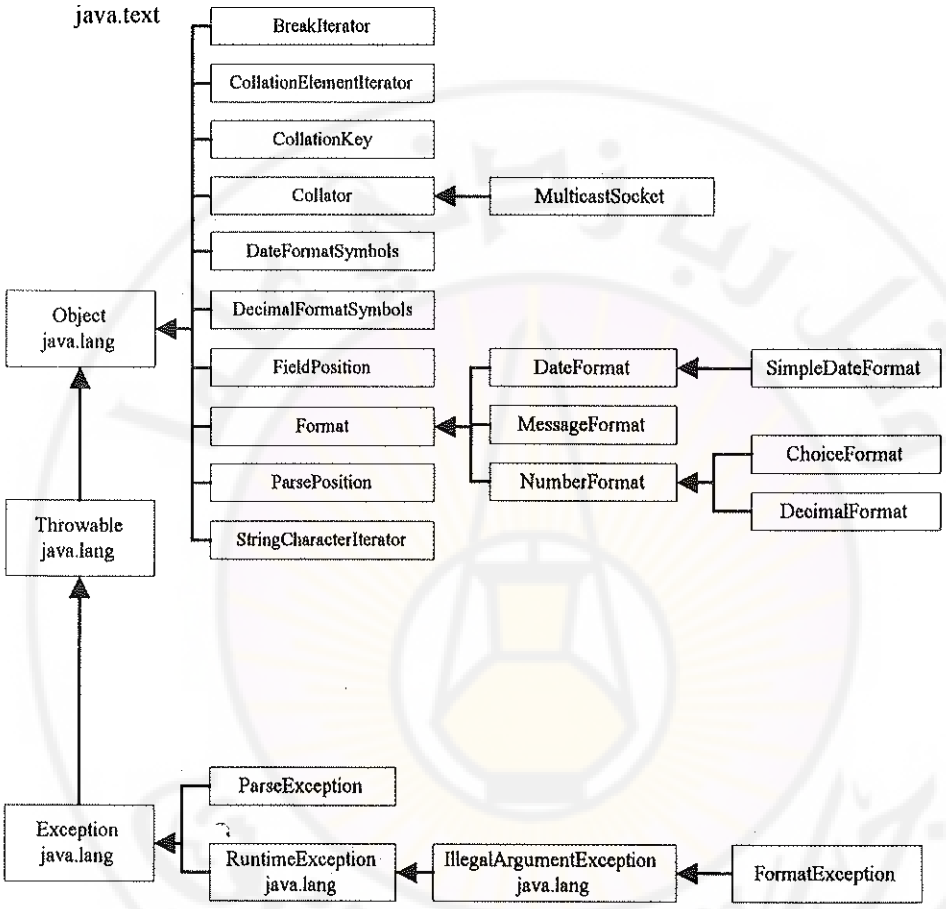
Damascus University

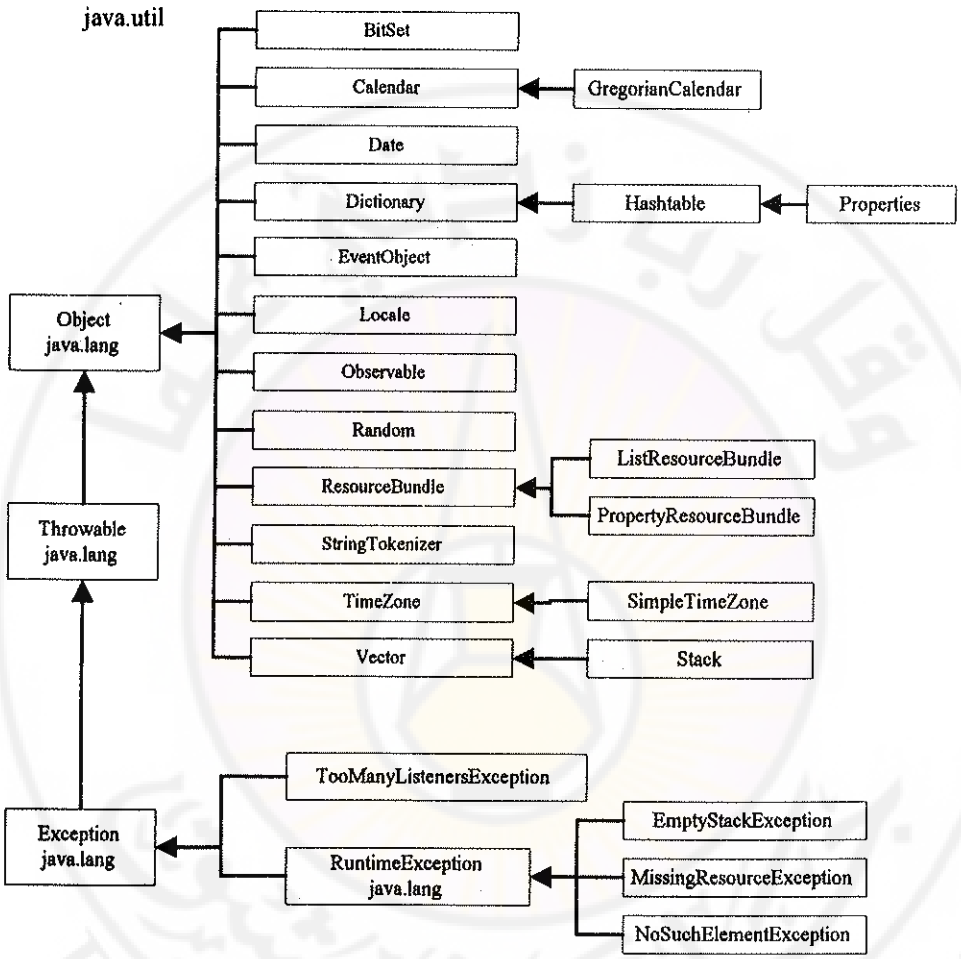


Damascus University



Damascus University


java.text
BreakIterator
CollationElementIterator
CollationKey
Collator
MulticastSocket
DateFormatSymbols
Object
java.lang
DecimalFormatSymbols
FieldPosition
DateFormat
SimpleDateFormat
Format
MessageFormat
ChoiceFormat
ParsePosition
NumberFormat
DecimalFormat
Throwable
java.lang
StringCharacterIterator
ParseException
Exception
java.lang
RuntimeException
java.lang
IllegalArgumentException
java.lang
FormatException



Damascus University

المراجع

- *JAVA Software Solutions. John LEWIS and William LOFTUS. ADDISON WISSLEY 1998.*
- *JAVA2. CAPUSPRESS 1999. FRANCE.*

وقل رب زدني علما
جامعة دمشق
Damascus University

فهرس أبجدي

263 :

264 ::

277 ,276 ..... Array

252 ..... boolean

252 ..... char

261 ..... classes

245 ..... comment

262 ..... constructors

268 ..... dynamic binding

356 ..... Exceptions

358 ..... finally

257 ..... if statement

266 ..... inheritance

262 ..... initial values

246 ..... methods

245 ..... Modifiers

256 ..... new

264 ..... Objects

254 ..... operator overloading

268 ..... polymorphism

256 ..... static method and members

280 ..... string

266 ..... this

359 ..... threads

358 ..... throw

260 ..... try statement

253 ..... union and struct

325 ..... Canvas

342 ..... Card Layout

344 ..... CardLayout

342 ..... Center

348 ,339 ,328 ..... CENTER

251 ..... char

281 ,280 ..... charAt

333 ..... Checkbox

334 ..... CheckboxGroup

333 ..... Choice

266 ..... Class

282 ..... clear

299 ..... clearRect

A

286 ..... action

316 ..... ActionListener

317 ..... actionPerformed

342 ,338 ,325 ..... add

316 ..... addActionListener

278 ..... addElement

333 ,332 ..... addItem

344 ..... addLayoutComponent

303 ..... addPoint

335 ..... AdjustmentListner

335 ..... adjustValueChanged

296 ..... Animation

281 ..... append

324 ,284 ,247 ..... Applet

301 ..... Arcs

356 ..... ArithmeticException

276 ..... Array

B

268 ..... binding

307 ..... BOLD

251 ..... boolean

342 ..... BorderLayout

348 ..... BOTH

258 ..... break

293 ..... brighter

272 ..... buffer

333 ..... Button

251 ..... byte

244 ..... bytecode

C

241 ..... C++

C++ and Java

312 .....event-driven programming
254 ..... expressions

F

302 ..... fillArc
303 ..... fillPolygon
299 ,298 ..... fillRect
358 ..... finally
344 ..... first
279 ..... firstElement
251 ..... float
337 .....Flowlayout
339 .....FlowLayout
306 ..... Font
259 ..... for
324 .....Frame

G

263 ,257 ..... Garbage Collector
282 ..... get
328 .....getAlignment
335 ..... getBlockIncrement
266 .....getClass
291 ..... getColor
306 .....getFontList
289 .....getGraphics
337 ..... getLayout
335 ..... getMaximum
357 ..... getMessage
335 ..... getMinimum
335 .....getOrientation
293 .....getRed
333 .....getselectedItem
332 ..... getSelectedItem
332 ..... getSelectedItemes
329 ..... getSelectedText
328 .....getText
335 ..... getUnitIncrement
335 ..... getValue
335 ..... getVisibleAmount
312 .....Graphical User Interface
282 ..... clone
290 ..... Color
291 .....foreground
280 ..... compareTo
243 .....compiler
323 ,312 .....Component
323 ,287 ..... Container
313 .....Containers
282 ,278 ..... contains
282 .....containsKey
259 ..... continue
332 .....countItems
365 ..... currentThread

D

293 ..... darker
332 ..... delItem
263 .....destrcutors
285 .....destroy
324 ..... Dialog
259 do
251 ..... double
300 .....draw3DRect
302 .....drawArc
289 .....drawLine
295 ..... drawOval
303 .....drawPolygon
305 .....drawPolyline
299 ,298 .....drawRect
299 .....drawRoundRect
307 ,248 .....drawString

E

342 .....East
279 .....elementAt
282 .....elements
280 .....endsWith
280 ..... equals
280 ..... equalsIgnoreCase
252 ..... esacape characters

245 ,243 ..... JDK

K

282 ..... keys

L

328 ..... Label
280 ..... lastIndexOf
337 ..... layout manage
327 ..... manager layout
312 ..... Layout manager
339 ,328 ..... LEFT
281 ,280 ..... length
331 ..... List
251 ..... long

M

245 ..... main
326 ..... minimumSize
268 ..... modifiers
286 ..... mouseDown
359 ..... multithreaded

N

247 ,243 ,241 ..... Netscape
344 ..... next
342 ..... North

P

248 ..... packages
286 ..... paint
324 ..... Panel
273 ..... parseInt
307 ..... PLAIN
303 ..... Polygon
268 ..... polymorphism
288 ,247 ..... Graphics
345 ..... GridBagConstraints
344 ..... GridBagLayout
340 ..... GridLayout
312 ..... GUI

H

286 ..... handleEvent
281 ..... hashCode
281 ,276 ..... Hashtable
244 ,243 ..... Hello World
334 ..... HORIZONTAL
241 ..... HotJava
246 ..... HTML

I

249 ..... Identifier
257 if
249 ..... import
280 ..... indexOf
285 ..... init
261 ..... inline
281 ..... insert
251 int
273 ..... Integer
248 ..... interface
267 ..... interfaces
247 ,243 ,241 ..... Internet Explorer
243 ..... Interpreter
358 ..... IOException
282 ..... isEmpty
307 ..... ITALIC

J

284 ..... java.applet.Applet
288 ..... java.awt
249 ..... java.lang
281 ..... java.util
245 ..... javac

251 ..... short
344 ..... show
283 ,279 ..... size
364 ..... sleep
342 ..... South
361 ,285 ..... start
280 ..... startsWith
255 ..... statements
280 ..... STL
285 ..... stop
271 ..... stream
279 ..... String
280 ..... StringBuffer
281 ..... Stringtokenizer
364 ..... suspend
242 ..... Swing
257 ..... switch
362 ,259 ..... synchronized

T

329 ..... TextArea
329 ..... TextComponent
329 ..... TextField
359 ..... Thread
260 ..... throw
357 ..... Throwable
358 ..... throws
280 ..... toLowerCase
306 ..... Toolkit
280 ..... toUpperCase
260try

U

286 ..... update

V

278 ,276 ..... Vector
334 ..... VERTICAL
326 ..... preferredSize
344 ..... previous
363 ,273 ,246 ,245 ..... println
357 ..... printStackTrace
282 ..... put

R

263 ..... real time
298 ..... Rectangle
282 ,281 ..... rehash
348 ..... REMAINDER
282 ..... remove
278 ..... removeElement
250 ..... reserved words
364 ..... resume
260 ..... return
281 ..... reverse
291 ..... RGB
339 ,328 ..... RIGHT
361 ,359 ..... run
359 ..... Runnable

S

334 ..... Scrollbar
333 ..... select
328 ..... setAlignment
292 ..... setBackground
334 ..... setBlockIncrement
281 ..... setCharAt
291 ..... setColor
329 ..... setEditable
307 ..... setFont
337 ..... SetLayout
281 ..... setLength
289 ..... setSize
329 ,328 ..... setText
334 ..... setUnitIncrement
335 ..... setValue
335 ..... setValues
293 ..... setXORMode

W

West 342
while 259
Window 324
wrapper class 252

X

XOR 293, 308

أ

أقواس 294, 301
ألوان 290
أنماط بسيطة 251
أنماط مركبة 253

ا

البرمجة المقودة بالأحداث 312, 313
الترتيب البطاقي 342
الترتيب الخاني 341
الترتيب الشبكي الحقائي 344
الحاويات 313
المكونات 313

ب

بان افتراضي 263
برمجيج 241, 246, 247, 284
بزمن حقيقي 263
بنط 306

ت

تحور 268
تعابير 254
تعدد المسالك 359
تعليمات 255
تعليمة الإسناد 257
تعليمة الابتدال 257

ث

ثماني 252

ج

جافا 2 242
جافا و C++ 251
جامع القمامة 256
جملة الإحداثيات البيانية 289

ح

حاويات 323
حزم 248
حقل النص 328

ر

ربط 268
رماز ثماني 244
رمز هروب 252

ز

زر 333

س

ست عشري 252

ش

شريط الإزاحة 334

ص

صفوف مغلفة 253
صوان 272

ق

قائمة 331
قطوع 294, 295
قناة 271

ك

كلمات محجوزة 250

ل

لصاقات التعليمات 257
لصاقة 328

م

مترجم 243
متعددات المستقيمات 305
متولي الترتيب 327
متولي الترتيب التدفقي 337
متولي الترتيب الشبكي 339
متولي ترتيب 337
متوليات الترتيب 312
مثلثات 295
مساحة النص 328
مستطيلات 294, 298
مستقيمات 294
مسدسات 295
مصفوفات 276
مضلعات 294, 303
معدلات 268
معدلات قابلية الرؤية 269
معينات الهوية 249
مفسر 243

ه

هوادم 263

و

وfill3DRect 300
وfillOval 295
وfillRoundRect 299
وgetBlue 293
وgetGreen 293
وlast 344
وlastElement 279
واجهات 248, 267
واجهة استخدام بيانية 312


وقل رب زدني علما
جامعة دمشق
Damascus University