Programmation en Java

par Sylvain Ferey

pour le concours “JavaCup”

Tables des matières

Liste des programmes

Introduction

Ce cours de Java est destiné aux débutants, cependant il a été pensé comme un guide de référence qui servira également aux développeurs initiés à Java.

Si vous débutez en Java et ne disposez pas encore de compilateur, vous trouverez, sur le Forum des Programmeurs Francophones, la distribution du kit de développement Java de Sun pour Windows 95/NT ainsi que pour Unix/Linux.

Vous pouvez également obtenir gratuitement de nombreux outils directement sur le web, notamment sur les sites de :

- Sun (**http://java.sun.com/products/jdk/**)
- Microsoft (**http://www.microsoft.com/java/sdk/**)
- Apple.(**http://applejava.apple.com**)

Il existe également des produits commerciaux fournissant des compilateurs Java interfacés avec un environnement de développement incluant éditeur et débuggeur; parmi ces produits citons : **CodeWarrior** de MetroWerks et **Visual Café** de Symantec, tous deux sur Macintosh et Windows 95/NT; ou Microsoft Visual J++ Professional Edition pour Windows 95/NT.

Les codes exposés ici sont compatibles avec tout compilateur Java et toute machine virtuelle Java tant que nous utiliserons l'API 1.0.x; cette API est supportée par les navigateurs de Netscape version 2.2 et ultérieure et ceux de Microsoft version 3.0 et supérieure ainsi que par les "AppletViewer" des kits Sun.

L'utilisation de l'API 1.1.x que nous étudierons plus en détail requiert une version adaptée des navigateurs si vous souhaitez exécuter des applets écrites avec cette API.

Une recommandation évidente encouragerait l'usage des API 1.0 pour construire des applets et les API 1.1 pour construire des applications Java.

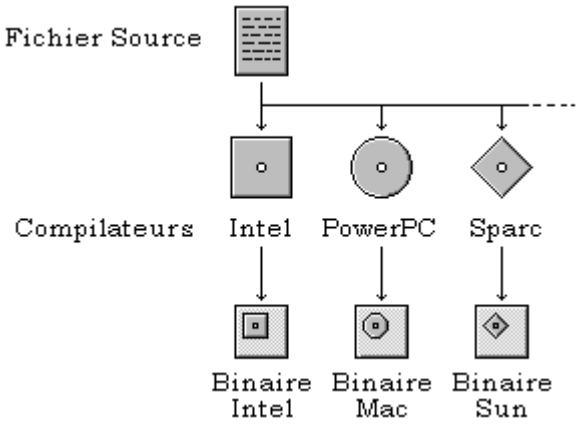
Les commandes de compilation, exposées ici, correspondront à l'utilisation du kit JDK de Sun; si vous utilisez un environnement de développement intégré, référez-vous à sa documentation pour apprendre à construire un projet et compiler un source Java.

Pourquoi utiliser Java ?

Le monde sans Java

Avec les langages évolués courants (C++, Pascal, etc.) nous avons pris l'habitude de coder sur une machine identique à celle qui exécutera nos applications; la raison est fort simple : à de rares exceptions près les compilateurs ne sont pas multi-plateformes et le code généré est spécifique à la machine qui doit l'accueillir.

Ainsi si nous souhaitons distribuer un produit sur plusieurs systèmes, nous devons le compiler pour chacun de ces systèmes :

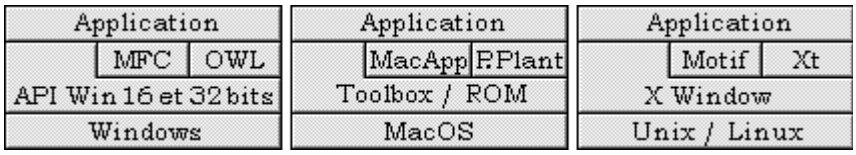


Nous devons alors utiliser soit *n* compilateurs différents sur *n* machines, soit un ou plusieurs compilateurs multi-plateformes. Outre l'investissement en matériels et logiciels, la connaissance de plusieurs systèmes et de leurs outils de développement (formation aux différents compilateurs et débuggeurs) représente un surcoût non négligeable.

Cette vision du développement logiciel s'accordait assez bien aux besoins tant que les applications produites disposaient d'une interface homme - machine (HIM) restreinte et que les sources étaient aisément portables d'un système à un autre. Les programmes fortran recouvrent bien cette définition.

Aujourd'hui, la généralisation des interfaces graphiques et l'usage de langage plus évolués compliquent encore davantage le problème. Ainsi pour développer une application destinée à plusieurs systèmes, il nous faut non plus seulement connaître plusieurs compilateurs, mais également plusieurs systèmes d'exploitation avec ses différentes couches de librairies et d'interfaces; les API de ces interfaces étant toutes différentes.

Le puzzle logiciel ressemble donc à :

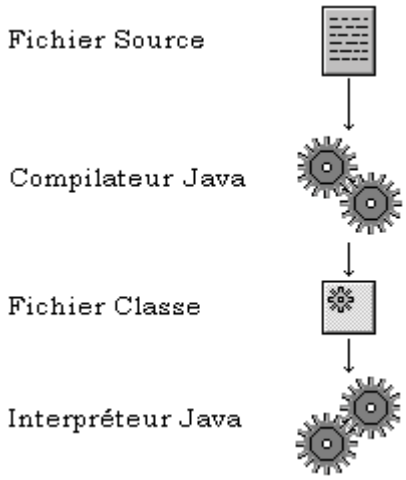


Ainsi nos applications sont fortement dépendantes des ressources (dont graphiques) du système hôte, dépendantes des API des interfaces utilisées, et le code produit ne peut s'exécuter que sur le système pour lequel il a été initialement produit.

Avec Java

Tout d'abord, Java simplifie le processus de développement; quelle que soit la machine sur laquelle vous réalisez le codage, le compilateur fournit le même code.

Ensuite, quel que soit le système utilisé par vos utilisateurs, cet unique code est directement opérationnel :



En effet, la compilation d'un source Java produit du pseudo-code Java qui sera exécuté par tout interpréteur Java sans aucune modification ou recompilation.

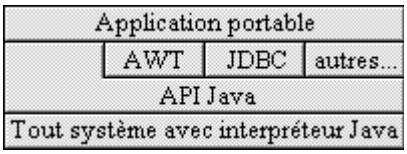
Cet "interpréteur" est couramment dénommé "machine virtuelle Java". De plus en plus, cette machine virtuelle utilise un compilateur JIT ("Just In Time") qui transforme, au moment de l'exécution, le pseudo-code en code natif afin d'obtenir la vitesse d'exécution maximale.

Le langage Java peut être utilisé pour créer des modules de code référencés au sein d'une page html et exécutés par un navigateur compatible Java (ou un simple interpréteur Java), on parle alors d'applets.

Ces modules de code ont rendu Java populaire car ils permettent à un créateur de site d'enrichir le contenu de son site de modules dynamiques et/ou interactifs qui tourneront à l'identique quelque soit la machine et le système utilisé par le visiteur de ce site.

Java permet également de créer des applications autonomes qui peuvent se substituer à des applications développés en langage compilé. Pour ces applications l'api Java apporte un ensemble très riche de classes répondant à de nombreux besoins et pouvant être étendue; cette unique api simplifie la création et le déploiement des applications, en effet cette application s'exécutera sur tout système en utilisant l'aspect visuel de ce système.

La construction d'une application Java répond au schéma :



Les applications ne sont plus dépendantes d'api propriétaires de leurs éditeurs (l'api Java est rendue publique par Sun); l'apprentissage d'un seul langage et d'une seule api vous ouvrent tous les marchés informatiques actuels et futurs (dont l'informatique domestique et embarquée).

Débuter en Java

Réalisation d'une Applet Java

Voyons tout de suite comment créer une applet, et, pour ne pas faillir à une vieille tradition, écrivons une applet qui dit simplement "Hello world!".

Voici le code de notre première applet :

Code 1a : Applet "Hello world"

```
1    /* Notre première applet */

2    import java.awt.Graphics;
3    import java.applet.Applet;

4    public class helloApplet extends Applet {
5       public void paint(Graphics graphics)
6       {
7           graphics.drawString("Hello world!", 10, 10);
8       }
9    }
```

Remarque : les mots en bleu sont des mots clés du langage Java, le violet sera pour les commentaires.

Les personnes familières du C ou C++ auront noté des similitudes avec ce langage, pour les autres n'ayez aucune crainte, le Java est peut être le langage idéal pour commencer avec un langage objet.

Détaillons ce source :

- la première ligne commençant par "/*" et finissant par "*/" est un commentaire, il est ignoré par le compilateur mais est toujours très utile pour le développeur.
- les seconde et troisième lignes utilisent le mot réservé "**import**", ce mot clé est utilisé pour préciser les bibliothèques de classes que nous allons utiliser. Ici nous importons la définition de la classe "Graphics" du paquetage "awt" (Abstract Window Toolkit) de la bibliothèque "java"; ainsi que la définition de la classe "Applet" du paquetage "applet" de la même bibliothèque "java".
- la quatrième ligne définit notre objet : "helloApplet". Cet objet est basé sur la classe "Applet", il en reprend les caractéristiques et les étends, pour signifier cela nous utilisons le mot réservé "**extends**".
- la cinquième ligne définit la fonction utilisée pour dessiner notre applet; la déclaration (ou prototype) de cette fonction est fixée par la classe "Applet", c'est une fonction publique (mot clé "**public**") afin que le navigateur puisse l'appeler; et elle ne retourne aucun résultat (mot clé "**void**" = rien); de plus cette fonction reçoit un unique paramètre que nous appelons ici arbitrairement "graphics" et qui est de type "Graphics" - notons au passage que le langage Java opère une distinction entre les caractères minuscules et majuscules.
- la sixième ligne ne comporte qu'une accolade ouvrante, c'est l'équivalent du mot clé "begin" du Pascal, elle sert à définir le début du corps de notre routine.
- la septième ligne fait un appel à la fonction "drawString" de la classe "Graphics", retenons pour l'instant que cela nous permet d'afficher un texte à une position désirée.
- la huitième ligne utilise une accolade fermante pour terminer la déclaration de notre routine, c'est l'équivalent du mot clé "end" du Pascal.

- la neuvième et dernière ligne utilise une autre accolade fermante pour terminer la déclaration de la classe “helloApplet” que nous avions commencé en ligne 4.

Remarque : le terme “classe” et d’autres principes de la programmation objet ont été utilisés ci-dessus sans beaucoup d’explications; ne vous inquiétez pas si tout ne vous parait pas très clair, vous allez rapidement comprendre tout cela avec la suite du cours et les suivants.

Pour exécuter notre première applet, nous devons maintenant compiler notre code source Java. Pour cela nous enregistrons le code ci-dessus (sans les numéros de lignes) dans un fichier nommé “helloApplet.java”. Notons que le nom du fichier doit correspondre au nom de la classe que ce fichier défini et que l'extension est toujours “java”; ici encore Java distingue les minuscules des majuscules, vous devez donc donner un nom de fichier avec la même syntaxe que celle utilisée pour nommer la classe. Nous utilisons maintenant le compilateur “javac” du JDK (toujours en respectant la syntaxe minuscules/majuscules) :

```
J:\>javac helloApplet.java
```

Remarque: les outils du JDK, pour Windows NT et 95, sont des applications 32 bits qui s'exécutent en mode console, vous devez donc taper les commandes dans une fenêtre de “commandes MS-DOS”. Sur Macintosh lancez l’outil “javac” est désignez le fichier à compiler grâce au dialogue d’ouverture de fichier.

Si votre source java ne contenait aucune erreur - vérifiez avec la version ci-dessus le cas échéant - le compilateur javac a créé un fichier de pseudo-code : “helloApplet.class”. Ce fichier ne peut pas être exécuté directement, comme le serait une application, il doit être chargé par un navigateur compatible Java ou un interpréteur Java. La contrepartie de ce fait est que ce code est portable sur toute machine et tout système où Java est disponible.

Utilisation de notre première applet

Pour visualiser notre applet, nous devons donc l'appeler depuis un navigateur (ou un interpréteur); pour cela nous créons un document html qui utilise cette applet, voici le source de ce document :

Code 1b : Référence de l'applet "Hello world" depuis le html

```
<html>
<body>
<applet code="helloApplet.class" width=100 height=50>
Ce message est affich&eacute si votre navigateur n'est pas compatible Java.
</applet>
</body>
</html>
```

Dans ce document html, nous utilisons la marque (ou balise) <applet> pour désigner l'applet à utiliser, le paramètre code permet de préciser le nom du fichier contenant la classe Java, les paramètres width et height précisent les dimensions de la zone réservée à notre applet. La marque <applet> est associée à la marque </applet>; tout ce qui est inséré entre ces deux marques sera affiché par un navigateur non compatible Java et ignoré si l'applet Java est correctement insérée.

Nous enregistrons ce document html sous le nom "helloApplet.html" - le choix du nom du fichier html est arbitraire, nous aurions pu ici choisir un nom différent de celui de notre applet.

Nous pouvons maintenant visualiser notre applet, ouvrez simplement le document html créé avec notre navigateur; alternativement vous pouvez utiliser l'interpréteur du JDK, soit:

```
J:\>appletviewer helloApplet.html
```

Dans les deux cas, "Hello world!" est affiché; nous verrons bientôt comment améliorer cet affichage.

Le minimum de théorie

La programmation orienté objet

Maintenant que nous avons réalisé et exécuté notre première applet, revenons sur les concepts de la programmation orienté objet (POO ou OOP en anglais).

Vous pourrez lire des dizaines de livres très différentes les uns des autres sur la POO; étant un sujet à la mode beaucoup se sont essayé à la présenter à leur façon; le propos suivant sera très illustré et sûrement peu académique, il est, en effet, inutile de maîtriser des termes techniques très compliqués pour percevoir les intérêts de la POO et l'utiliser, ici comme ailleurs l'école du "feeling" peut s'opposer à celle des règles.

Pour illustrer notre présentation, nous utiliserons un objet "voiture"; le terme "objet" est utilisé ici au sens commun du terme : une voiture est quelque chose qui existe, qui a certaines caractéristiques (forme, couleur, ...) et qui réalise certaines actions (roule, s'arrête, tombe en panne,...). Que l'on parle d'une voiture, d'un presse-purée ou d'un objet informatique créé grâce à un langage objet; pensez le terme "objet" avec exactement la même approche : il a des caractéristiques (on parlera de "variables", de "champs", de "propriétés" ou d' "attributs") et peut réaliser des actions (on parlera de "méthodes", de "comportements").

Notre voiture, tel un programme informatique à réaliser, à beaucoup d'actions à réaliser; dans l'industrie on utilise des chaînes de montages où chaque poste est dédié à une tache; en POO, nous utiliserons une hiérarchie de d'objets où chacun d'eux réalisent une tache particulière.

Dans l'assemblage de notre objet voiture, nous ne savons, par exemple, pas comment utiliser l'électricité de la batterie pour faire démarrer le moteur; par contre nous savons qu'il existe un objet démarreur qui réalise cela très bien; nous réglerons donc le problème du démarrage en utilisant dans notre objet voiture un exemplaire d'un démarreur, le terme "instance" sera synonyme de "exemplaire". Avec le même raisonnement, nous munirons notre voiture d'un objet "moteur" et d'un objet "boite de vitesses"; votre voiture possède maintenant différents objets près à rendre des services, nous devons organiser leur utilisation; pour cela nous créons - avec une syntaxe inventée - notre objet :

```
Objet : Voiture                 //  définition de notre classe "voiture"
    Démarreur       leDémarreur //  on utilise la classe "Démarreur",
    Moteur          leMoteur    //  la classe "Moteur",
    BoiteDeVitesse  laBoite     //  et la classe "BoiteDeVitesse"
    Démarre()                   //  notre fonction pour démarrer
    Roule()                     //  notre fonction pour rouler
Fin Objet
```

Codons maintenant comment la "Voiture.Démarre( )" :

```
    leDémarreur.lance( leMoteur )
```

nous utilisons ici la fonctionnalité de l'objet "leDémarreur" de la classe "Démarreur", qui sait faire démarrer le moteur que nous fournissons en paramètre de la fonction "lance". Savoir comment le démarreur s'y prends exactement ne nous intéresse pas, et lors d'un codage réel nous n'aurez pas besoin de savoir obligatoirement comment un objet s'acquitte de sa tache, l'important est qu'il nous rende le service que l'on attends de lui.

Bon, le moteur tourne; il nous reste à faire en sorte que la "Voiture.Roule( )" :

```
pour leRapport allant de 1 à 4
    laBoite.PasseLaVitesse( leRapport )
    leMoteur.Accélère( )
fin de pour
```

Voila ! notre voiture roule.

Evidemment c'est une voiture sommaire, elle ne sait même pas tourner. Comment améliorer ses fonctionnalités ? Nous pourrions demander à qui nous a fournit cette voiture - à qui a codé les objets que nous allons utiliser - de prendre en compte certaines fonctions. Nous pouvons également les ajouter nous même grâce au principe d'héritage. Pour cela nous créons une nouvelle classe (ou famille d'objet) qui reprends les caractéristiques de "voiture" pour ajouter un volant et un moyen d'orienter le véhicule :

```
Objet : Véhicule hérite de Voiture // définition de notre classe "véhicule"
    Volant  leVolant           //  on utilise la classe "Volant"
    Tourne()                   //  notre fonction pour tourner
Fin Objet
```

nous disposons maintenant d'une nouvelle classe fournissant plus de services, nous pouvons également imaginer non pas d'ajouter des services mais d'améliorer les services existants, pour cela nous recouvrons (redéfinissons) des fonctions existantes :

```
Objet : Véhicule hérite de Voiture // définition de notre classe "véhicule"
    Démarre()                  //  nouvelle fonction pour démarrer
Fin Objet
```

nous pourrions alors réaliser quand le "Véhicule.Démarre( )" :

```
si leNiveauD_Essence > 0 alors
    Voiture.Démarre( )
sinon
    Affiche( "je ne peux pas démarrer..." )
fin si
```

dans cette méthode surchargée (redéfinie) nous testons le niveau d'essence; si celui-ci est supérieur à zéro, nous appelons la méthode "Démarrer" de la classe parent, sinon nous prévenons le conducteur que sa demande ne peut pas être satisfaite.

Voila les principes de la POO, il reste à apprendre la syntaxe propre à chaque langage mais d'ores et déjà nous savez penser objet.

La syntaxe de base

Revenons sur le source de notre première applet :

```
/* Notre première applet */

import java.awt.Graphics;
import java.applet.Applet;

public class helloApplet extends Applet {
    public void paint(Graphics graphics)
    {
        graphics.drawString("Hello world!", 10, 10);
    }
}
```

La clause “import”

Une des caractéristiques du langage Java est sa portabilité, une même applet (dont notre helloApplet.class) peut être exécuté sous Windows NT/95, MacOS, SunOS, Solaris, etc.

Du fait de cette portabilité, nous n’utilisons aucune librairie contenant les fonctions que nous rencontrons habituellement dans les runtime des autres langages; de même comme notre code ne dépend pas de services fournis par l’OS (Operating System = système d’exploitation) nous devons très souvent utiliser des services fournis par les bibliothèques de classes installées avec le navigateur compatible Java (par exemple Netscape Navigator) ou un runtime Java (par exemple MacOS Runtime for Java).

L’utilisation de ces services est dynamique; cela signifie qu’il n’existe aucune opération de résolutions des liens (link) avec Java et que les objets auxquels nous faisons référence seront chargés au moment de l’exécution de notre code.

Pour spécifier quelle classe doit être importée, nous utilisons la clause (le mot clé) “**import**”.

Cette clause peut être utilisé avec une désignation explicite d’une classe particulière, par exemple : dans le paquetage “java.awt” (abstract window toolkit) nous utilisons la classe “Graphics”; ou désigné un paquetage complet, par exemple “import java.awt.*” importe la description de toutes les classes définies dans le paquetage “java.awt”.

Cette clause “import” permet donc de recenser les différentes classes que nous utiliserons en nous permettant par la suite de les désigner simplement par leur nom; nous pouvons omettre cette clause, mais nous devrons alors qualifier le nom des classes que nous utilisons; le code suivant est fonctionnellement identique au premier :

```
public class helloApplet extends java.applet.Applet {
    public void paint(java.awt.Graphics graphics) {
        graphics.drawString("Hello world!", 10, 10);
    }
}
```

Remarque : par “librairie”, il faut comprendre “librairie de classes”; c’est généralement un fichier zip qui contient des fichiers *.class de pseudo-code et non une librairie binaire comme celles générés, par exemple, par un compilateur C.

La classe Applet

Un code en Java ne contient aucune définition de variable ou fonction globales; tous les types sont des classes et toutes les fonctions sont des méthodes d'objets. Ceci souffre en fait d'une exception, les types primitifs (types numériques et caractères) ne sont pas des classes; ils seront présentés plus loin.

Pour cela, il ne semble pas y avoir de point d'entré à notre code (routine "main") et nous ne déclarons même pas d'instance de la classe "helloApplet" (pas de variable de cette classe).

En fait l'allocation de l'instance de "helloApplet" est réalisé par le navigateur quand il rencontre la marque "`<applet code="helloApplet.class">`"; c'est pour cette raison que notre classe doit être déclaré "public", afin qu'un autre code (dont un navigateur) ait accès aux informations de cette classe et donc puisse l'allouer et la lancer.

Notre classe "helloApplet" descends (hérite) de la classe "Applet" (formellement: java.applet.Applet) et possède donc tout le mécanisme pour être initialisé par le navigateur (ou tout autre outil de visualisation d'applet, dont "appletviewer" du JDK), puis lancé par celui-ci. Ces opérations sont réalisées via les méthodes suivantes, définies par la classe "Applet" :

`public void init( );`	appelé au chargement de l'applet
`public void start( );`	appelé après l'initialisation ou après un interruption
`public void stop( );`	appelé lorsque l'applet n'est plus active (ex. on quitte la page qui la contient)
`public void destroy( );`	appelé quand on quitte le navigateur

le comportement de la classe Applet pour ces méthodes, nous convenait parfaitement, nous n'avons donc surchargé aucune de ces méthodes.

Par contre la méthode d'affichage "paint" est celle de "java.awt.Component" dont hérite indirectement la classe "Applet" (on a `Component->Container->Panel->Applet`); son comportement est de ne rien faire du tout; voila pourquoi nous avons choisi de redéfinir cette fonction afin d'afficher notre message.

Nous verrons plus loin comment et pourquoi surcharger également les autres méthodes de "Applet".

Types définis par Java

Java est un langage fondamentalement objet; l'absence de globales en est déjà un signe.

Le langage ne définit pas non plus de pointeurs bien que les objets soient dynamiques, c'est à dire alloués; de ce fait, une instance de classe est toujours une référence à un objet. Cependant cette référence est masquée par le compilateur Java, ceci implique que nous accédons toujours à un membre de l'objet (une de ses variables ou méthodes) en utilisant la syntaxe *nom de l'objet* "." (point) *nom de la variable ou fonction membre*; comme dans "graphics.drawString".

Ceci implique également que lorsque nous transmettons un objet à une méthode, c'est bien une référence à cet objet qui est transmis; si la routine appelée modifie l'objet, il le sera au retour de cette routine. Ainsi l'équivalent de la méthode "paint" serait :

en Pascal Object:
```
procedure paint(var g: Graphics);
```

en C++:
```
void paint(Graphics& graphics);
```

Hormis les classes, Java définit des types primitifs; ceux-ci sont, à l'opposé des classes, toujours transmis par valeur; cela signifie notamment que vous ne pouvez pas modifier la valeur d'un paramètre de type primitif dans une méthode et voir cette nouvelle valeur répercutée dans la routine appelante.

Types primitifs (numériques et caractère)

Type	Taille (bits)	Valeurs min. et max.	Désignation
boolean	1	true ou false	valeur logique
char	16	0 à 0xFFFF (65535)	caractère unicode (non signé)
byte	8	-128 à 127	entier signé sur 1 octet
short	16	-32 768 à 32 767	entier signé sur 2 octets
int	32	-2 147 483 648 à 2 147 483 647	entier signé sur 4 octets
long	64	-9 223 372 036 854 775 808 à 9 223 372 036 854 775 807	entier signé sur 8 octets
float	32	+/-1.40239846e-45 à +/-3.40282347e+38	réel 4 (norme IEEE 754)
double	64	+/- 4.94065645841246544e-324 à +/- 1.79769313486231570e+308	réel 8 (norme IEEE 754)

Remarque :

- la taille des différents types de données est indépendante de la machine hôte.
- il n'existe pas de modificateurs tel "unsigned", il est donc impossible de créer une variable contenant, par exemple, des valeurs comprises entre 0 et 255 en utilisant un "unsigned byte"; corollairement l'arithmétique est toujours signé.
- une chaîne de caractères correspond à la classe "String" et non à un tableau de "char"; par extension tout tableau (y compris de type primitif) est un objet.

Amélioration de notre applet

Assez de théorie et appliquons ce qui viens d'être exposé.

Positionnement de l'applet

Nous avons vu précédemment que la marque <applet> permettait de faire référence à une applet et de provoquer son affichage dans une page html; examinons la syntaxe complète de cette marque.

`<applet`	
`code = nom du fichier classe`	fichier (.class)contenant la définition de la classe
`[ codebase = répertoire ]`	localisation, relative ou absolue, du fichier ci-dessus
`width = largeur de l'applet`	largeur initiale en pixels réservé pour l'applet
`height = hauteur de l'applet`	hauteur initiale en pixels réservé pour l'applet
`[ align = top, middle ou bottom ]`	justification de l'applet dans la page
`[ hspace = marge horizontale ]`	marge, en pixel, insérée à gauche et à droite de l'applet
`[ vspace = marge verticale ]`	marge, en pixel, insérée au dessus et en dessous de l'applet
`[ name = nom de cette applet ]`	nom unique permettant un dialogue entre applets
`[ alt = texte ]`	texte affiché si le navigateur est compatible Java mais que l'exécution de code Java a été interdite

```
>
[ <param name = un nom  value = une valeur >]
[ text affiché par un navigateur non compatible avec Java ]
</applet>
```

Remarque : les paramètres entre crochets sont optionnels.

Le paramètre `codebase` vous permet d'organiser vos fichiers de classes en les séparant des documents html; par exemple si les applets sont dans un dossier "mesApplets" se trouvant au même niveau que le document html les utilisant, vous préciserez `<applet codebase="mesApplets" code="helloApplet.class">`.

Le paramètre `align` vous permet de contrôler l'alignement de l'applet; vous pouvez également utiliser `texttop`, `absmiddle`, `baseline` ou `absbottom`.

Amélioration de l'écriture

Pour afficher notre texte avec une autre police de caractères, nous devons utiliser une instance de la classe "Font"; cette classe est définie dans "java.awt", nous utiliserons donc une clause "import java.awt.Font".

De plus, nous souhaitons que notre instance de "Font" soit créé une fois pour toutes, pour cela nous surchargerons la méthode init( ) de la classe "Applet".

Voici notre applet modifiée :

Code 2 : Choix de la police d'écriture

```
import java.awt.Font;
import java.awt.Graphics;

public class helloApplet extends java.applet.Applet
{
    protected Font  font;

    public void init() {
        super.init();
        font = new Font("TimesRoman", Font.BOLD, 24);
    }

    public void paint(Graphics graphics) {
        graphics.setFont(font);
        graphics.drawString("Hello world!", 10, 25);
    }
}
```

Examinons le code :

- nous avons déclaré une variable "font" qui contiendra une instance de la classe "Font", cette variable est protégée ("protected") car nous ne voulons pas que son contenu puisse être altéré par un objet qui utiliserait une instance de cet objet.
- nous avons déclaré une fonction "init( )" qui surcharge "Applet.init( )". Ceci nous permet de créer notre objet de type "Font", pour cela nous utilisons le mot clé **new** suivi d'un constructeur de la classe "Font". Un constructeur est une méthode particulière d'un objet qui sert à l'initialiser et qui est toujours appelée en association avec le mot clé new (Note aux Cistes: on ne parle pas ici d' "opérateur new" car celui-ci ne peut pas être surchargé). Un objet peut définir plusieurs constructeurs ayant chacun un jeu de paramètres différents, le runtime choisira parmi ceux déclarés celui qui correspond à l'appel que nous utilisons. Nous appelons également la procédure d'initialisation du parent de notre classe (soit "Applet.init( )") grâce au mot clé "**super**" qui désigne toujours la classe parent. En fait "Applet.init( )" ne réalise aucune action, nous l'utilisons ici pour rappeler qu'il est toujours conseillé d'appeler la méthode de la classe parent de toute méthode surchargée; concernant cette méthode nous omettrons par la suite l'appel à la méthode de la classe Applet.
- dans la méthode "paint" nous utilisons notre fonte grâce à la méthode "setFont" de la classe "Graphics"

Compilons notre nouvelle applet :

```
J:\>javac helloApplet.java
```

et regardons le résultat :

```
J:\>appletviewer helloApplet.html
```

Si vous n'avez pas modifié le document “helloApplet.html”, vous risquez de ne pas voir l'intégralité du message; en effet la zone réservé à notre applet, dans notre tout premier exemple, est devenue trop petite; agrandissons largement les dimensions de cette zone pour ne plus être gêné dans cet exercice.

Le document “helloApplet.html” devient :

```
<html>
<head><title>hello Applet</title></head>
<body>

<applet code="helloApplet.class" width=250 height=100>
Ce message est affich&eacute; si votre navigateur n'est pas compatible Java.
</applet>

<hr>
<a href="helloApplet.java">Le source.</a>
</body>
</html>
```

Remarque : nous avons ajouté à notre document html une marque `<a href=...>` afin de pouvoir visualiser facilement le code source de notre applet si nous la lançons depuis un navigateur; voici ci-dessous le résultat obtenu avec “appletviewer” :

Cette fois, notre applet est entièrement visible; nous verrons dans un prochain cours comment régler ces problèmes de taille.

Passage de paramètres

Maintenant que nous savons fixer la police de caractères à utiliser, nous aimerions pouvoir la changer facilement, de plus après avoir “salué tout le monde” nous voudrions que cette applet personnalise un peu son message.

Pour réaliser cela, nous utilisons la marque `<param ...>` à l'intérieur de la marque `<applet...>`.

Soit le nouveau document “helloApplet.html” :

Code 3a : Passage de paramètres depuis un document html

```
<html>
<head><title>hello Applet</title></head>
<body>

<applet code="helloApplet.class" width=250 height=100>
Ce message est affich&eacute si votre navigateur n'est pas compatible Java.
<param name="taille" value=24>
<param name="prenom" value="Duke">
</applet>

<hr>
<a href="helloApplet.java">Le source.</a>
</body>
</html>
```

Note : “Duke”, c'est lui : 

Nous passons ici deux paramètres à notre applet; le premier correspondant à la propriété nommée “taille” est un nombre valant 24; la seconde propriété, identifiée par “prenom”, à pour valeur “Duke”. Voyons maintenant comment lire ces valeurs et les utiliser dans notre applet.

Modifications du code :

- nous déclarons une variable “fontSize” de type “int” pour mémoriser la taille choisie.
- une autre variable “name”, instance de la classe “String”, mémorisera le prénom à afficher.
- la méthode “init( )” doit lire les paramètres reçus; pour cela nous utilisons la méthode “getParameter( )” de la classe “Applet”. Notons que nous ne faisons pas précéder “getParameter( )” du nom d'une instance de classe; c'est inutile ici car la méthode appartient à l'objet qui l'appelle, c'est à dire notre classe “helloApplet”; formellement nous utilisons en fait le descripteur implicite “this” qui désigne l'objet lui-même.

Code source modifié :

Code 3b : Lecture de paramètres depuis une applet

```
import java.awt.Font;
import java.awt.Graphics;

public class helloApplet extends java.applet.Applet
{
    protected Font      font;
    protected int       fontSize;
    protected String    name;

    public void init() {
        String parameter = getParameter("taille");
        fontSize = Integer.parseInt(parameter);
        font = new Font("TimesRoman", Font.BOLD, fontSize);

        name = getParameter("prenom");
    }

    public void paint(Graphics graphics) {
        graphics.setFont(font);
        graphics.drawString("Bonjour " + name + " !", 10, 40);
    }
}
```

Notons que “getParameter( )” est utilisé en fournissant une chaîne de caractères (transformée en instance de la classe “String” par le compilateur) et nous retourne une référence à une instance de “String” qui contient la valeur du paramètre.

Quand nous lisons la taille des caractères, nous voulons obtenir une valeur entière et non une chaîne de caractères, pour l’obtenir nous utilisons la méthode “parseInt” de la classe “Integer” (rappelez-vous il n’existe pas de routines globales - dont de conversions - mais uniquement des classes nous proposant leurs services); cette méthode extrait de l’objet “String”, fournit en paramètre, une séquence représentant un nombre et nous retourne ce nombre.

Nous pouvons alors créer notre objet “Font” avec la taille voulue; notons au passage que le deuxième paramètre correspond au style des caractères et prends l’une des valeurs : BOLD, ITALIC ou PLAIN. Ces constantes sont définies par la classe “Font” elle-même, nous devons donc les préfixer par “Font.”.

Modifions le code comme présenté ci-dessus et lançons la compilation, puis l’exécution; nous obtenons :

Validité des paramètres

Tout semble fonctionner pour le mieux, mais supposez - et essayez pour voir le résultat - que l'on ait oublié de préciser les paramètres attendus dans notre document html ou que les valeurs transmises soient invalides, par exemple une chaîne de caractères a été transmise à la place de la valeur numérique décrivant la taille des caractères.

Dans de tels cas, notre applet risque de planter - si "parseInt" rencontre des caractères ne formant pas un nombre - et de ne rien afficher.

Pour gérer les erreurs pouvant survenir, Java propose un mécanisme de traitements des erreurs (un modèle semblable existe en C/C++).

Ce mécanisme repose sur les mots clé "**try**" et "**catch**"; littéralement vous pouvez comprendre ces mots clé comme "essaie donc de faire ce qui suit" et si quelque chose ne va pas "attrape les erreurs".

Le mot clé "catch" est suivi, entre parenthèses, du nom d'une classe correspondant à un type d'erreurs; ceci nous permettra plus tard d'adapter le traitement à la nature de l'erreur. Aujourd'hui nous utiliserons une instance de la classe "Throwable" qui recueille toutes les erreurs.

Modifications du code :

- tout d'abord nous testons l'existence du paramètre "taille" en le comparant à la valeur "**null**"; s'il existe nous réalisons la conversion au sein d'un bloc try/catch; si une erreur se produit, notre bloc de code désigné par catch est exécuté nous permettant de fixer "fontSize" à une valeur par défaut et de continuer l'exécution de notre applet.
- de la même façon, l'existence du paramètre "prenom" est testé et notre variable "name" est initialisée avec une valeur par défaut la cas échéant.

Code source modifié :

Code 4 : Lecture de paramètres avec contrôle de validité

```
import java.awt.Font;
import java.awt.Graphics;

public class helloApplet extends java.applet.Applet {
    protected Font      font;
    protected int       fontSize;
    protected String    name;

    public void init() {
        String parameter = getParameter("taille");
        if (parameter == null)
            fontSize = 24;
        else {
            try {
                fontSize = Integer.parseInt(parameter);
            }
            catch (Throwable t) {
                fontSize = 24;
            }
        }
        font = new Font("TimesRoman", Font.BOLD, fontSize);
        name = getParameter("prenom");
        if (name == null)
            name = "tout le monde";
    }

    public void paint(Graphics graphics) {
        graphics.setFont(font);
        graphics.drawString("Bonjour " + name + " !", 10, 40);
    }
}
```

Les méthodes de dessin

Dans les chapitres précédents, nous avons utilisé la méthode “drawString” de la classe Graphics (du paquetage java.awt.Graphics); voyons maintenant comment dessiner autre chose que du texte à savoir les formes géométriques de base.

Les coordonnées graphiques en Java

Le système de coordonnées de Java est semblable à celui d’autres environnement : le point (0, 0) est situé en haut à gauche et les coordonnées croissent vers la droite et vers le bas.

La plupart, sinon toutes, des routines graphiques reçoivent leurs paramètres sous formes de variables de type “int” (soit réel 32bits); ceci définit virtuellement un espace de tracé bien plus grand que tous les espaces associés aux périphériques d’affichage.

Les méthodes de Graphics

Parmi les méthodes définies par la classe Graphics certaines sont dites “abstract” cela signifie que la classe “Graphics” fixe la syntaxe de cette méthode mais qu’elle ne l’implémente pas (elle ne fournit pas le code réalisant le traitement); pour ce qui nous préoccupe aujourd’hui dites vous que vous pouvez utilisez sans crainte ces méthodes dans la méthode “paint” d’une classe dérivée de “Applet”.

Le choix de la couleur

Avant de tracé nous pouvons fixer la couleur à utiliser grâce à la méthode :

```
public abstract void setColor(Color c);
```

où “c” est une couleur constante choisie parmi : Color.white (blanc), Color.black (noir), Color.lightGray (gris clair), Color.gray (gris moyen), Color.darkGray (gris foncé), Color.red (rouge), Color.green (vert), Color.blue (bleu), Color.yellow (jaune), Color.magenta, Color.cyan, Color.pink (rose) ou Color.orange.

La couleur “c” peut également être une couleur définie grâce à ces trois composantes Rouge, Vert, Bleu exprimées par des nombres entiers compris entre 0 et 255 ou trois réels compris entre 0.0 et 1.0; nous utilisons l’un des deux constructeurs de la classe “Color” de la même façon, exemple :

```
// crée un objet “jaune” de type “Color”
Color jaune = new Color( 255, 255, 0 );

// crée un objet “rose” de type “Color”
Color rose  = new Color( 1.0, 0.7, 0.7 );

public void paint(Graphics g) {
    g.setColor(jaune);  // dessine maintenant en jaune
    .....
    g.setColor(rose);   // dessine maintenant en rose
    .....
}
```

Le tracé de lignes

```
public abstract void drawLine(int x1, int y1, int x2, int y2);
```

permet de tracer une ligne entre les points définis par les coordonnées (x1, y1) et (x2, y2), exemple :

```
public void paint(Graphics g) {
//  ligne entre les points (10,10) et (80,60)
    g.grawLine(10, 10, 80, 60);
}
```

le tracé de rectangles

les rectangles simples

```
public void drawRect(int x, int y, int largeur, int hauteur);
```

dessine un rectangle défini par son coin supérieur gauche, sa largeur et sa hauteur, exemple :

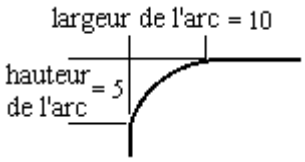
```
public void paint(Graphics g) {
//  rectangle de coords (10,10) et (90,70)
    g.drawRect(10, 10, 80, 60);
}
```

les rectangles à bord arrondi

```
public abstract void drawRoundRect(
     int x1, int y1,
     int largeur, int hauteur,
     int largeur_de_l_arc, int hauteur_de_l_arc);
```

dessine un rectangle avec des coins arrondis, ce rectangle s'inscrit dans le "rectangle vrai" défini comme précédemment par son coin supérieur gauche, sa largeur et sa hauteur; les coins sont formés par des arcs de cercle inscrits dans un rectangle de largeur et hauteur données; exemple :

```
public void paint(Graphics g) {
    g.drawRoundRect(10, 10, 80, 60, 10, 5);
}
```



les rectangles en relief

```
public void draw3DRect(
     int x, int y,
     int largeur, int hauteur,
     boolean en_avant);
```

dessine un rectangle avec un ombrage pour simuler un effet de 3D, ce rectangle est tracé à partir du point (x, y) avec la largeur et hauteur spécifiées; selon la valeur booléenne, le rectangle apparaît surélevé (true) ou enfoncé (false).

les rectangles simples remplis

```
public void fillRect(int x, int y, int largeur, int hauteur);
```

dessine un rectangle rempli avec la couleur courante, exemple : enregistrez le source ci-dessous dans le fichier “drawApplet.java” et compilez-le avec “javac” (“javac drawApplet.java”↵)

Code 5a : Tracé de carrés de couleur aléatoire

```
import java.awt.Color;
import java.awt.Graphics;

public class drawApplet extends java.applet.Applet
{
    public void paint(Graphics g) {
        for (int y = 20; y < (size().height - 20); y += 25)
        {
            for (int x = 20; x < (size().width - 20); x += 25)
            {
            //  génère une couleur aléatoire
            //  Math.random() renvoie un nombre compris entre 0.0 et 1.0
            //  sous forme de “double”, nous le convertissons en “float”
                g.setColor(new Color((float) Math.random(),
                    (float) Math.random(), (float) Math.random()));

            //  dessine un rectangle rempli avec cette couleur
            //  le rectangle est en fait un carré de 20 pixels de coté
                g.fillRect(x, y, 20, 20);

            //  utilise la couleur noire pour encadrer ce carré
                g.setColor(Color.black);
                g.drawRect(x-1, y-1, 21, 21);
            }
        }
    }
}
```

Remarque : “size( )” renvoie les dimensions de la zone réservée à l’applet sous forme d’une instance de la classe “Dimension” qui possède deux variables d’instances publiques “width” et “height”. Ainsi “size( ).width” est la largeur de la zone réservée pour l’applet (avec la marque “applet” ci-dessous, cette valeur est 340) et “size( ).height” est la hauteur de cette zone (pour la marque “applet” ci-dessous, cette valeur est 240).

Créons un fichier html faisant référence à cette Applet et enregistrons-le sous “drawApplet.html” :

Code 5b : Utilisation de l’applet de tracé de carrés

```
<html>
<head><title>draw Applet</title></head>
<body>

<applet code="drawApplet.class" width=340 height=240>
Ce message est affich&eacute si votre navigateur n'est pas compatible Java.
</applet>

<hr>
<a href="drawApplet.java">Le source.</a>
</body>
</html>
```

enfin, ouvrez ce fichier html avec un navigateur compatible Java ou utilisez l’ “AppletViewer” du JDK (“appletviewer drawApplet.html”↵)

les rectangles à bord arrondi remplis

```
public abstract void fillRoundRect(
     int x1, int y1,
     int largeur, int hauteur,
     int largeur_de_l_arc, int hauteur_de_l_arc);
```

dessine et remplis un rectangle avec des coins arrondis, la définition du rectangle et des coins arrondis est la même que pour “drawRoundRect”; exemple :

```
public void paint(Graphics g) {
    g.setColor(Color.blue)
    g.fillRoundRect(10, 10, 50, 50, 25, 25);    // dessine un rond bleu
}
```

les rectangles en relief remplis

```
public void fill3DRect(
     int x, int y,
     int largeur, int hauteur,
     boolean en_avant);
```

dessine et remplis un rectangle en relief; voir “draw3DRect” pour la définition des paramètres.

le tracé d’ellipse

les ellipses simples

```
public abstract void drawOval(int x, int y, int largeur, int hauteur);
```

dessine une ellipse inscrite dans le rectangle ayant pour coin supérieur gauche le point (x, y) et les largeur et hauteur précisées; si la largeur et la hauteur sont identiques on obtient un rond.

Contrairement à la largeur et hauteur des coins arrondis des rectangles vu précédemment, les tailles fixent ici la dimension totale de l’ellipse et non un quart d’ellipse comme ce fut le cas pour tracer un coin.

Exemple :

```
public void paint(Graphics g) {
    g.drawOval(10, 10,  50, 50);    // dessine un rond de rayon 25 pixels
    g.drawOval(10, 80, 100, 50);    // dessine une ellipse de grand axe 100
                                    // pixels et de petit axe 50 pixels
}
```

les ellipses remplies

```
public abstract void fillOval(int x, int y, int largeur, int hauteur);
```

dessine une ellipse inscrite dans le rectangle ayant pour coin supérieur gauche le point (x, y) et les largeur et hauteur précisées; si la largeur et la hauteur sont identiques on obtient un rond.

Contrairement à la largeur et hauteur des coins arrondis des rectangles vu précédemment, les tailles fixent ici la dimension totale de l’ellipse et non un quart d’ellipse comme ce fut le cas pour tracer un coin.

Exemple : enregistrez le source ci-dessous dans le fichier “drawApplet.java” et compilez-le avec “javac” (“javac drawApplet.java”↵):

Code 6 : Tracé d'ellipses de couleur aléatoire

```
import java.awt.Color;
import java.awt.Graphics;

public class drawApplet extends java.applet.Applet
{
    public void paint(Graphics g) {
        for (int i = 0; i < 100; i++){
        //  génère une couleur aléatoire
            g.setColor(new Color((float) Math.random(),
                (float) Math.random(), (float) Math.random()));

        //  dessine une ellipse remplie avec cette couleur
        //  Math.floor retourne sous forme de "double" la valeur
        //  arrondie du réel passé en paramètre
            g.fillOval(
                (int) Math.floor(Math.random() * size().width),
                (int) Math.floor(Math.random() * size().height),
                (int) Math.floor(Math.random() * 50),
                (int) Math.floor(Math.random() * 50));
        }
    }
}
```

Créez un fichier html faisant référence à cette Applet (son contenu est identique à celui présenté plus haut), puis ouvrez ce fichier html avec notre navigateur (compatible Java) ou avec "AppletViewer".

le tracé d'arc de cercle

les arcs de cercles simples

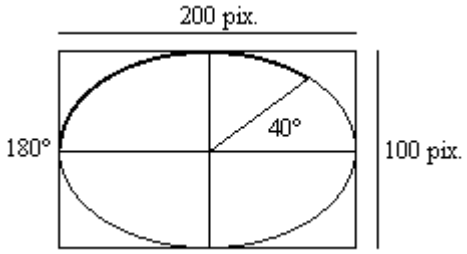
```
public void drawArc(
     int x, int y,
     int largeur, int hauteur,
     int angle_début, int angle_fin);
```

dessine un arc d'ellipse représentant une partie de l'ellipse qui s'inscrirait dans le rectangle défini par son coin supérieur gauche et ayant les largeur et hauteur fixées. Les angles de début et de fin sont données en degré et sont comptés sur le cercle trigonométrique avec 0° à 3 heures d'un cadran horaire, 90° à 12h., 180° à 9h. et 270° à 6h.

Exemple :

```
public void paint(Graphics g) {
    g.drawArc(10, 10, 200, 100, 40, 180);
}
```

dessine l'arc d'ellipse suivant (partie en gras) :



les arcs de cercles remplis

```
public abstract void fillArc(
     int x, int y,
     int largeur, int hauteur,
     int angle_début, int angle_fin);
```

dessine et remplie un arc d'ellipse défini comme ci-dessus.

Exercices :

1. modifiez le code de l'exemple n°2 pour tracer des arcs d'ellipse dont l'origine et la fin du secteur angulaire seront déterminés aléatoirement.
2. simplifiez le code obtenu pour tracer des arcs de cercle.
3. modifiez la classe "drawApplet" pour éliminer le code redondant (code invariant exécuté plusieurs fois)

le tracé de polygone

les polygones simples

```
public abstract void drawPolygon(int[] x, int[] y, int nombre_points);
public void drawPolygon(Polygon p);
```

la première implémentation attends deux tableaux de coordonnées (le premier pour les abscisses des points formant le polygone, le second pour leur ordonnées) et le nombre de points dans ce tableau, exemple :

```
public void paint(Graphics g) {
    int X[] = { 10, 90, 50 };
    int Y[] = { 10, 10, 90 };
    int taille = X.length;          // ceci récupère la taille du tableau X
    g.drawPolygon(X, Y, taille);    // devinez quelle forme est dessinée
}
```

la seconde implémentation utilise une instance de la classe "Polygon", cette classe permet d'ajouter des points aux polygones grâce à la méthode "addPoint".

Exemple : enregistrez le source ci-dessous dans le fichier "drawApplet.java" et compilez-le avec "javac" ("javac drawApplet.java"☞):

Code 7 : Définition et tracé de polygones

```
import java.awt.*;

public class drawApplet extends java.applet.Applet {
    protected Polygon   forme;
    protected Dimension surface;

    public void init() {
        surface = size();
        int X[] = { 10, 90, 50 };
        int Y[] = { 10, 10, 90 };
        forme = new Polygon(X, Y, X.length);
    }

    public void paint(Graphics g) {
    //  on ajoute un point à chaque fois que l'applet est réaffichée
        forme.addPoint(
            (int) Math.floor(Math.random() * surface.width),
            (int) Math.floor(Math.random() * surface.height));
        g.drawPolygon(forme);
    }
}
```

Créez un fichier html faisant référence à cette Applet (son contenu est encore identique à celui présenté plus haut), puis ouvrez ce fichier html avec notre navigateur (compatible Java) ou avec “AppletViewer”.

les polygones remplis

```
public abstract void fillPolygon(int[] x, int[] y, int nombre_points);
public void fillPolygon(Polygon p);
```

ces méthodes dessinent et remplissent un polygone défini comme ci-dessus.

Un complément de théorie

Dans les chapitres précédents, nous avons souvent défini une méthode “paint” de la classe “Applet”; une telle définition polymorphe nous a permis de faire écrire ou dessiner quelque chose à notre applet; examinons cette classe “Applet” (ou “java.applet.Applet”) en détail.

Le polymorphisme

Mais tout d’abord qu’est-ce qu’une définition polymorphe ?

Le dictionnaire nous dirait “qui peut se présenter sous différentes formes”, en programmation orientée objet - qui repose fondamentalement sur cette notion - nous dirions plutôt “qui peut être présent dans différents objets”.

Pour illustrer cela reprenons notre voiture-objet, nous l’avions muni d’une instance (“laBoite”) de la classe fictive “BoiteDeVitesse”, et pour faire rouler la voiture nous utilisions la méthode “Roule( )” que nous écrivons (grâce à nos nouvelles connaissances de Java) :

```
public void Roule() {
    for (int leRapport = 1; leRapport <= 4; leRapport++){
        laBoite.PasseLaVitesse( leRapport );
        leMoteur.Accélère( );
    }
}
```

Donc notre classe “BoiteDeVitesse( )” définit une méthode nommée “PasseLaVitesse” qui reçoit comme unique paramètre une variable de type “int”. En supposant que cette classe était dans un paquetage nommé “mécanique.auto”, nous avions quelque chose comme :

```
public class BoiteDeVitesse
{
    public void PasseLaVitesse( int unRapport ){
    /* traitement adéquat */
        ......
    }
/* d'autres méthodes */
        ......
}
```

Mais lorsque nous avons écrit ce code nous ne pensions pas à une boite de vitesse particulière, devons-nous réécrire le code si la boite à 4 rapports est remplacée par une boite 5 vitesses ou, pire, par une boite automatique ?

Grâce aux objets, non !

En fait il suffit pour une nouvelle classe de redéfinir une méthode existante avec la même syntaxe pour que ce nouveau code se substitue à l’ancien à chaque fois que cette méthode est appelée pour une instance de cette nouvelle classe; exemple :

```
public class BoiteAutomatique extends mécanique.auto.BoiteDeVitesse
{
    public void PasseLaVitesse( int unRapport ){
        /* t'inquiètes pas, je gère ... */
    }
}
```

ainsi selon la nature réelle de l’instance “laBoite”, c’est cette méthode ou la première qui sera appelée; il est des cas où nous voudrions également profiter de ce que savait faire la classe parent et juste apporter une fonctionnalité en plus, nous réalisons cela sans difficulté en appelant la méthode de la classe parent; comme cette méthode porte le même nom (forcément) nous sommes obligé de le faire précéder de “super” qui désigne toujours cette classe parent, nous coderions par exemple :

```
public class Boite_5_vitesses extends mécanique.auto.BoiteDeVitesse
{
```

```
        public void PasseLaVitesse( int unRapport ){
            if (unRapport == 5) {
            /* code adéquat pour accrocher le petit pignon */
                .....
            }
            else {
            //  la méthode de la classe BoiteDeVitesse
            //  est responsable des autres vitesses
                super.PasseLaVitesse( unRapport );
            }
        }
    }
```

Les méthodes de “Applet”

La classe “Applet” définit les méthodes suivantes :

```
public void init();

public void start();

public void stop();

public void destroy();
```

ces méthodes ont été introduites avec les chapitres précédents; nous avons déjà recouvert “init( )” pour initialiser des variables d’instances de nos classes.

D’autres méthodes hérités de “Component” (“Applet” hérite de “Panel”, “Panel” de “Container” et “Container” de “Component”) vont nous intéresser aujourd’hui; ces méthodes permettent de gérer les événements, il s’agit de :

```
public boolean handleEvent        (Event evt);

public boolean mouseDown          (Event evt, int x, int y);
public boolean mouseDrag          (Event evt, int x, int y);
public boolean mouseEnter         (Event evt, int x, int y);
public boolean mouseExit          (Event evt, int x, int y);
public boolean mouseMove          (Event evt, int x, int y);
public boolean mouseUp            (Event evt, int x, int y);

public boolean keyDown            (Event evt, int key);
public boolean keyUp              (Event evt, int key);
```

Toutes ces méthodes reçoivent une instance de la classe “Event” nous détaillerons cette classe au paragraphe suivant; la méthode “handleEvent” est le cœur du gestionnaire événements; c’est cette routine qui identifie la nature de l’événement à traiter et qui appelle la méthode concernée parmi les suivantes de la liste ci-dessus.

Les méthodes “mouseXxxx” traitent des événements liés à la souris, elles reçoivent en paramètre supplémentaire les coordonnées du pointeur au moment où l’événement a été généré.

La méthode “mouseDown” est appelé lors qu’un bouton de la souris est appuyé, “mouseUp” est appelé lorsque ce bouton est relâché; dès que le pointeur est déplacé d’au moins un pixel horizontalement ou verticalement, “mouseMove” est appelé si aucun bouton n’est enfoncé sinon “mouseDrag” est appelé; enfin quand la souris entre dans la zone réservée à notre applet “mouseEnter” est appelé, de même pour “mouseExit” dès que la souris sort de cette zone.

les méthodes “keyDown” et “keyUp” sont appelées lorsqu’une touche est pressée puis relâchée, le paramètre supplémentaire contient des informations sur la touche impliquée.

Pour traiter un événement particulier, nous pouvons implémenter un de ces gestionnaires pour réaliser la tâche souhaitée; si nous recouvrons le gestionnaire “handleEvent” nous

devrons impérativement appeler la méthode polymorphe de la classe parent afin que les événements non traités par notre méthode le soient par la classe parent; remarquez à ce propos que tous ces méthodes retournent une valeur logique “true” ou “false” sera permet au gestionnaire de savoir si l'événement à été correctement traité - auquel cas il n'y a plus qu'à attendre l'événement suivant - où si au contraire cet événement est toujours à la recherche de quelqu'un qui le prendra en compte - auquel cas le gestionnaire appellera le gestionnaire d'événements des autres éléments d'interface présents susceptibles de traiter de tels événements.

La classe “Event”

Cette classe est définie dans le package “java.awt.Event”, nous devons donc importer ce package pour l'utiliser.

La classe “Event” définit de nombreuses constantes qui permettent d'identifier la nature des événements, nous en détaillerons certaines ci-dessous; elle définit également les variables d'instances suivantes :

`public Object arg;`	argument arbitraire (varie selon l'événement)
`public Object target;`	la cible de cet événement
`public Event evt;`	l'événement suivant dans la liste des messages
`public long when;`	le moment où s'est produit l'événement
`public int id;`	le type d'événement (cf ci-dessous)

la variable d'instance “id” est fixée avec une des constantes de la classe “Event”; ces constantes sont définies comme “public final static int” : “public” pour les rendre publiques, ceci nous permet de les utiliser dans les méthodes de la classe “Event” ainsi que depuis une quelconque autre méthode (sous réserve d'importer le paquetage “java.awt.Event”); “final” signifie que cette variable ne pourra pas être modifiée, ce qui en fait une variable de classes; “static” enfonce le clou en nous rappelant que, en effet, il s'agit d'une variable de classe; enfin ces variables sont de type “int”.

Selon la valeur de “id” les variables d'instances suivantes sont définies :

- si “id” est (la liste n'est pas exhaustive puisque le langage évolue) :

  MOUSE_DOWN, MOUSE_UP,
  MOUSE_MOVE, MOUSE_DRAG,
  MOUSE_ENTER, MOUSE_EXIT

alors

`public int x;`	contient la position horizontale de la souris
`public int y;`	contient la position verticale de la souris

pour la valeur MOUSE_DOWN, une variable supplémentaire est renseigné, elle est à zéro sinon,

`public int clickCount;`	est à 1 pour un simple clic, 2 pour un double clic, etc...

- si “id” est :

  KEY_PRESS, KEY_RELEASE,
  KEY_ACTION, KEY_ACTION_RELEASE

alors

`public int key;`	contient le code de la touche impliquée
`public int modifiers;`	renseigne sur l'état des touches de modification (Control, Alt, ...)

Variable de classe et variable d'instance

Ces termes ont été utilisés plusieurs fois, quelle est la différence exacte entre les deux ?

Une classe donnée a souvent besoin de constantes, il est peu intéressant que chaque nouvel objet de cette classe contiennent une copie de la variable définissant la valeur de cette constante, puisque la valeur est, précisément, constante; pour remédier à cela nous utilisons le mot clé "static" qui rend la variable commune à tous les objets de cette classe; cela signifie que si un objet modifie cette variable (les variables dites "static" ne sont pas obligatoirement constantes), elle sera modifié pour tous les objets de cette classe.

Concrètement : notre voiture a besoin de quatre sièges (c'est une quatre places); donc nous fournirons quatre variables de la classe "siège" (ou une classe dérivée - qui étends "siège" - selon les options désirées par le client : cuir, appui-tête, etc..); ces quatre variables-sièges sont des variables d'instance car elles appartiennent en propre à cette voiture (toutes les voitures du parking ne partagent pas les quatre même sièges !).

Ces sièges ont eux-mêmes différentes variables, parmi elles se trouve la couleur; notre client s'attend à trouver quatre sièges de couleur identique dans sa voiture, il est donc inutile que chaque siège possède sa propre variable d'instance "couleur"; au contraire nous définirons la variable "couleur" de la classe "siège" comme "static" ainsi il n'existera dans notre code qu'une seule telle variable et si de plus elle est déclarée "public", le concessionnaire n'aura qu'à fixer cette variable avant de livrer la voiture pour satisfaire à tous les goûts.

Lorsque nous utilisons des valeurs constantes, il apparaît encore plus évident de déclarer ces valeurs comme "static"; imaginons par exemple un code faisant des bulles; chaque bulle est caractérisé par une variable d'instance "rayon" - car, en effet, chaque bulle peut avoir un rayon différent - et sait calculer son volume grâce à une méthode "double Volume( )"; pour calculer ce volume nous avons besoin de la valeur du nombre pi; en supposant que pi ne soit pas défini dans un paquetage mathématique, nous serions obligé de définir une variable pour cette valeur (d'accord on pourrait aussi utiliser la valeur numérique...), il parait alors naturel de rendre cette variable statique afin que toutes les bulles ne créent pas cette variable dont la redondance serait inutile.

Cet exemple de bulles pourrait être codé (enregistrez ce source dans un fichier “bulle.java” et compilez-le avec “javac”) :

Code 8a : Définition de constantes

```
public class bulle {
    static double QuatreTiersDePi = 4.0 / 3.0 * Math.PI;
    public double rayon;

    public bulle( double unRayon ) {
        rayon = unRayon;
    }

    public double Volume(){
        return QuatreTiersDePi * rayon * rayon * rayon;
    }
}
```

On utilisera alors cette classe, par exemple, comme suit : (enregistrez ce second source dans un fichier “FaisDesBulles.java”, compilez-le avec “javac” et lancez cette *application* par “java FaisDesBulles”; les plus impatients d’entre vous noterons au passage le prototype de la fonction principale d’une application Java).

Code 8b : Faisons des bulles

```
class FaisDesBulles {
    public static void main(String args[]){
        for (short i = 0; i < 10; i++){
            bulle sphere = new bulle( Math.random() );
            System.out.println(
                "rayon : " + sphere.rayon +
                "  volume : " + sphere.Volume());
        }
    }
}
```

La gestion des événements (modèle API 1.0)

Nous allons, pour illustrer cette gestion, mettre en place une applet de dessin étape par étape.

Répondre au clic de la souris

Commençons par créer une méthode qui dessine un point pour chaque appui sur un bouton de la souris.

Enregistrez le source ci-dessous dans le fichier “eventApplet.java” et compilez-le (vous pouvez obtenir un avertissement - warning - dû à l’utilisation d’une API dépréciée (désapprouvée), ne tenons pas compte de ces messages pour l’instant).

Code 9 : Réponse à l’appui d’un bouton de la souris

```
import java.awt.Color;
import java.awt.Event;
import java.awt.Graphics;

public class eventApplet extends java.applet.Applet
{
    public void init() {
        setBackground(Color.white); // fond en blanc
        setForeground(Color.blue);  // tracé en bleu
    }

    public boolean mouseDown(Event evt, int x, int y){
        Graphics g = getGraphics(); // récupère le contexte graphique
        g.fillOval(x, y, 5, 5);
        return true;
    }
}
```

Créez un fichier html faisant référence à cette Applet et enregistrez-le sous “eventApplet.html” :

```
<html>
<head><title>event Applet</title></head>
<body>

<applet code="eventApplet.class" width=250 height=250>
Ce message est affich&eacute; si votre navigateur n'est pas compatible Java.
</applet>

<hr>
<a href="eventApplet.java">Le source.</a>
</body>
</html>
```

enfin, ouvrez ce fichier html avec un navigateur compatible Java ou avec “AppletViewer”.

A chaque clic (quand le bouton est enfoncé) un petit rond bleu est dessiné; agrandissez alors l’applet (en redimensionnant la page qui le contient le cas échéant), l’applet est effacée ! en effet nous n’avons pas défini de méthode “paint” donc la méthode de la classe parent (classe ancêtre en fait, car cette méthode est définie en bas de la hiérarchie de classe) réinitialise la surface réservée à l’applet; un programme de dessin devra donc mémoriser les dessins à réaliser (au lieu de les dessiner directement) puis utiliser une méthode polymorphe “paint” pour redessiner ce qui aura été sauvegardé afin de reconstruire l’intégralité du dessin; pour l’heure contentons-nous de ce comportement pour mieux examiner la gestion des événements.

Répondre au relâchement de la souris

Ajoutons une méthode qui dessine un point d'une autre couleur quand l'utilisateur relâche le bouton de la souris; ici nous pouvons toujours utiliser "setBackground(*instance de Color*)" afin de définir la couleur de fond de l'applet, par contre "setForeground(*instance de Color*)" qui fixe la couleur de tracé ne peut pas être utilisée car nous utiliserons deux couleurs différentes, nous fixerons donc la couleur active avant chaque tracé; nous pourrions mémoriser la couleur actuellement sélectionnée grâce à une variable logique afin de changer cette couleur uniquement lorsque cela est nécessaire, cependant nous n'écrivons pas pour l'instant un code où le temps d'exécution est un critère important (au contraire puisque nous *attendons* que l'utilisateur clique...).

Code 10 : Réponse au relâchement d'un bouton de la souris

```
import java.awt.Color;
import java.awt.Event;
import java.awt.Graphics;

public class eventApplet extends java.applet.Applet
{
    public void init() {
        setBackground(Color.white);
    }

    public boolean mouseDown(Event evt, int x, int y){
        Graphics g = getGraphics();
        g.setColor(Color.blue);
        g.fillOval(x, y, 5, 5);
        return true;
    }

    public boolean mouseUp(Event evt, int x, int y){
        Graphics g = getGraphics();
        g.setColor(Color.red);
        g.fillOval(x, y, 5, 5);
        return true;
    }
}
```

Implémenter son propre gestionnaire

Pour obtenir plus de souplesse, et minimiser le code redondant, nous allons redéfinir le gestionnaire “handleEvent”, nous en profitons pour matérialiser le déplacement du pointeur lorsque le bouton est enfoncé (événement de type Event.MOUSE_MOVE) :

Code 11 : Gestionnaire d’événement souris

```
import java.awt.Color;
import java.awt.Event;
import java.awt.Graphics;

public class eventApplet extends java.applet.Applet
{
    public void init() {
        setBackground(Color.white);
    }

    public boolean handleEvent(Event evt){
        Graphics g = getGraphics();

        switch (evt.id){           // selon le type d'événement...
        case Event.MOUSE_DOWN:
            g.setColor(Color.blue);
            g.fillOval(evt.x, evt.y, 5, 5); // un rond bleu
            return true;

        case Event.MOUSE_UP:
            g.setColor(Color.red);
            g.fillOval(evt.x, evt.y, 5, 5); // un rond rouge
            return true;

        case Event.MOUSE_DRAG:
        //  matérialise le déplacement de la souris
            g.setColor(Color.green);
            g.fillOval(evt.x, evt.y, 2, 2); // un point vert
            return true;

        default:
        //  ne pas oubliez l'appel à la classe parent pour
        //  traiter les événements non pris en compte ici
            return super.handleEvent(evt);
        }
    }
}
```

Le tracé en mode Xor

Nous venons de voir comment gérer les événements souris, nous allons modifier notre applet afin qu'elle trace des lignes et non plus des points quelque peu désordonnés.

Le but recherché est le suivant : lors de l'appui d'un bouton de la souris, nous stockons la position du curseur (Java génère l'événement "Event.MOUSE_DOWN" quel que soit le bouton pressé; pour l'instant nous nous contenterons de ce modèle de souris à un seul bouton), tant que le bouton est pressé, nous traçons une ligne entre le point d'ancrage initial et la position courante, cela suppose que nous effacions l'ancienne ligne et redessinions la nouvelle à chaque changement de position de la souris; enfin quand l'utilisateur relâche le bouton, nous traçons la ligne définitive.

Pour réaliser cette opération nous allons utiliser la fonction "setXORMode" de la classe "Graphics" qui active le mode de tracé en mode Xor; dans ce mode un premier tracé dans une couleur donnée dessine le trait et un deuxième, avec la même couleur, l'efface en réalisant l'opération binaire Xor.

Avec cette fonction nous pourrons donc effacer une ligne, simplement en la redessinant à la même position dans la même couleur; ensuite la fonction "setPaintMode", nous réactiverons le mode de tracé normal et afficherons la position courante ou définitive.

Code 12 : Une applet de dessin

```
import java.awt.*;
import java.applet.*;

public class Dessin extends Applet {

    private Point   ancre;              // position initiale
    private Point   ruban;              // position courante
    private boolean bouge;              // la souris a bougée ?

    public void init() {
        setBackground(Color.white);     // arrière-plan en blanc
        setForeground(Color.white);     // premier-plan en blanc

        ancre = new Point(-1000, -1000);
        ruban = new Point(-1000, -1000);
        bouge = false;
    }

    public boolean handleEvent(Event event){
        Graphics g = getGraphics();     // récupère le contexte graphique

        switch (event.id) {
            case Event.MOUSE_DOWN:
                ancre.x = event.x;      // sauvegarde la position
                ancre.y = event.y;      // de début de tracé
                bouge = false;
                return true;

            case Event.MOUSE_UP:
                DrawXorLine(g, event);  // trace la ligne finale
                return true;

            case Event.MOUSE_DRAG:
                DrawXorLine(g, event);  // mise à jour du tracé
                ruban.x = event.x;      // sauvegarde les nouvelles
                ruban.y = event.y;      // coordonnées de l'extrémité
                bouge = true;
                return true;

            default:
                return super.handleEvent(event);
        }
    }

    protected void DrawXorLine(Graphics g, Event position)
```

```
        {
        //  Efface l'ancienne ligne
            if (bouge){
                g.setXORMode(Color.blue);
                g.drawLine(ancre.x, ancre.y, ruban.x, ruban.y);
            }
        //  Trace la nouvelle ligne
            g.setColor(Color.blue);
            g.setPaintMode();
            g.drawLine(ancre.x, ancre.y, position.x, position.y);
        }
    }
```

Ce code crée une applet nommée “Dessin”; nous l’enregistrons donc dans un fichier nommé “Dessin.java”, nous le compilons avec “Javac” (javac Dessin.java); nous créons un fichier html qui utilise cette applet, ce fichier “Dessin.html” contient :

```
<html>
<head><title>Applet de dessin</title></head>
<body>

<applet code="Dessin.class" width=250 height=250
    alt="Votre navigateur est compatible Java mais
        l'interpr&eacute;teur Java semble d&eacute;sacvit&eacute;.">
Ce message est affich&eacute si votre navigateur n'est pas compatible Java.
</applet>

<hr>
<a href="Dessin.java">Le source.</a>
</body>
</html>
```

Enfin, nous pouvons lancer l’applet grâce à “AppletViewer” (appletviewer Dessin.html).

Cliquez dans la fenêtre de l’applet et faîtes danser la souris; que ce passe-t-il ?

➡ La ligne en cours de positionnement est correctement tracée mais elle efface les lignes précédentes; parfois, quand on sort du cadre (bouton pressé) et que l’on revient dans le cadre (bouton toujours pressé) la ligne est scindée en deux avec un nouveau point à la limite du cadre; enfin si nous redimensionnons le cadre, tout est effacé... Heureusement ce n’était qu’un début, nous allons voir comment corriger ces points un à un; mais commençons tout d’abord par transformer cette applet en application.

Applet et Application de dessin

Une Application Java

Nous avons déjà rencontrer des applications Java, elle se caractérise par la présence d'une fonction "main"; il suffirait donc d'en ajouter une ... Mais que doit-elle faire ?

Simplement la même chose que ce qui est réalisé par le navigateur, c'est à dire instancier notre applet et l'insérer dans un cadre; ce cadre ne doit pas être confondu avec un cadre ("frame") du navigateur; en Java ces cadres correspondent à la classe "Frame" du paquetage "java.awt".

Cette classe "Frame" (java.awt.Frame) définit la fenêtre principale de l'application, elle peut avoir un titre fixé par la méthode "setTitle(String)", une barre de menus fixée par la méthode "setMenuBar(MenuBar)", la forme du curseur peut être définie grâce à "setCursor(int typeDeCurseur)" afin cette fenêtre est peut être redimensionnée si "setResizable(boolean)" a été appelé avec la valeur "true" - ou pas appelé du tout car c'est le comportement par défaut.

Une application Java est donc composée d'un cadre (une instance de la classe "Frame" ou d'une classe dérivée) à l'intérieur duquel nous insérons d'autres éléments visuels généralement des descendants de la classe "Panel" (java.awt.Panel) qui définit un conteneur générique.

Examinons le code qui définit une application Java

Code 13 : Une application Java

```
import java.awt.*;

public class Dessin extends Frame
{
//  point d'entrée pour l'application Dessin
    public static void main(String args[]) {
    //  création d'un conteneur principal avec titre
        Frame cadre = new Frame("Dessin");

    //  insertion d'un panneau de dessin au centre du cadre
        cadre.add("Center", new Surface_de_dessin());

    //  redimensionnement du cadre et affichage
        cadre.pack();               // prépare l'affichage
        cadre.resize(250, 250);     // redimensionne
        cadre.show();               // affiche la fenêtre
    }
}

class Surface_de_dessin extends Panel
{
    private Point   ancre;          // position initiale
    private Point   ruban;          // position courante
    private boolean bouge;          // la souris a bougée ?

    public Surface_de_dessin() {    // constructeur de la classe
        setBackground(Color.white);
        setForeground(Color.white);

        ancre = new Point(-1000, -1000);
        ruban = new Point(-1000, -1000);
        bouge = false;
    }

    public boolean handleEvent(Event event){
        codage identique à celui de l'applet du "code 12"
    }
```

```
    protected void DrawXorLine(Graphics g, Event position){
        codage identique à celui de l'applet du "code 12"
    }
}
```

Ce code est compilé avec Javac mais est maintenant exécuté avec **Java** (Java Dessin).

```
J:\>javac Dessin.java
```

puis :

```
J:\>java Dessin
```

Remarque : notez que le compilateur a créé autant de fichiers classes que nous avons défini de classes dans notre source, soit : "Dessin.class" et "Surface_de_dessin.class".

Qu'observons-nous ?

➡ Evidemment les bugs "cosmétiques" ne se sont pas guéris tout seul (patience), mais de plus on ne peut pas quitter cette application : un clic sur la case de fermeture est sans effet ! Seul remède tapez "Control + C" dans la fenêtre de commandes Dos à partir de laquelle vous avez lancé Java.

De plus, chargé dans un navigateur (via le document "Dessin.html") ce cadre n'affiche rien du tout, une erreur est même reportée; commençons par corriger cela.

Une Applet-Application

Lorsque nous chargeons le fichier "Dessin.class", le navigateur s'attends à rencontrer une applet et à appeler les routines fournies par cette classe ("init( )", "start( )", ...); la classe "Frame" n'implémente pas ces méthodes et n'est pas un descendant de "Applet" aucune chance pour que notre source ne permettent donc de créer un composant utilisable à la fois en tant qu'applet et en tant qu'application ... sauf si ce cadre est créé par une applet !

Continuons donc à dériver nos classes : la classe "Dessin" devient une sous classe de "Applet" (afin d'être applet compatible) et notre classe "Frame" doit définir un gestionnaire d'événements (afin de pouvoir quitter l'application); ces modifications nous conduisent à :

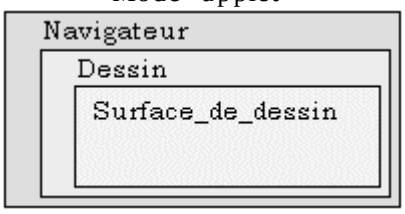
Code 14 : Une applet - application

```
import java.applet.*;
import java.awt.*;

public class Dessin extends Applet {

//  point d'entrée pour l'application 'Dessin'
//  réalise ce que ferait un navigateur

    public static void main(String args[]) {

    //  création du cadre principal
        Frame cadre = new Cadre_de_dessin("Dessin");
    //  définition de la stratégie d'insertion
        cadre.setLayout(new BorderLayout());

    //  création et démarrage d'une applet
        Dessin applet = new Dessin();
        applet.init();
        applet.start();

    //  insertion de cette applet dans le cadre principal
        cadre.add("Center", applet);

    //  redimensionnement du cadre et affichage
        cadre.pack();
        cadre.resize(250, 250);
        cadre.show();
    }
```

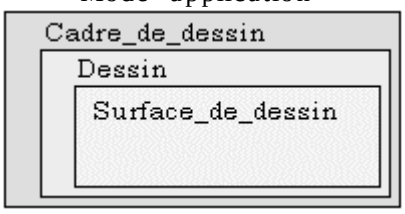
```
//  initialisation de l'applet 'Dessin'
//  appelé directement par le navigateur (le cas échéant)

    public void init() {
    //  création d'un positionneur
        setLayout(new BorderLayout());
    //  ajout d'un panel de dessin
        add("Center", new Surface_de_dessin());
    }
}

class Cadre_de_dessin extends Frame {
//  Un constructeur avec un titre
    public Cadre_de_dessin(String title){
        super(title);
    }

//  Gestionnaire d'événements au niveau du cadre
    public boolean handleEvent(Event event) {
        switch (event.id) {
        //  la fermeture de ce cadre quitte l'application
            case Event.WINDOW_DESTROY:
                hide();             // cache le cadre
                dispose();          // et le détruit
                System.exit(0);     // quitte 'Java'
                return true;
            default:
                return super.handleEvent(event);
        }
    }
}

class Surface_de_dessin extends Panel {
    codage identique à celui du "code 13"
}
```

Cette nouvelle classe peut être lancée depuis un navigateur, grâce à un fichier html contenant une référence identique à celle de "Dessin.html"; et peut également être lancée comme une application indépendante. L'organisation des éléments est alors, selon le mode d'utilisation (applet ou application) du fichier class :

Mode "applet"



Mode "application"



De meilleurs événements

Nous pouvons observer que les ruptures brutales de lignes n'apparaissent pas dans la version applet de notre classe mais persistent dans la version application; en fait ceci est dû à la gestion des événements dans notre classe "Cadre_de_dessin" (sous classe de "Frame); ainsi (par exemple) lors du premier clic - à l'intérieur du cadre, la classe "Surface_de_dessin" reçoit un événement MOUSE_DOWN; lorsque nous quittons cette surface et que la souris survole l'objet "Cadre_de_dessin" cet événement est à nouveau envoyé à notre Panel qui donc fixe une nouvelle valeur aux champs de "ancre".

Pour cerner l'origine de tels problèmes vous pouvez utiliser la console standard pour afficher des messages permettant de mieux comprendre les événements distribués; ainsi pour chaque étiquette ("case") de la méthode "handleEvent" de la classe "Surface_de_dessin", l'utilisation de "System.out.println(xxx)" permet de voir quels événements sont reçus et à quel moment. Les messages générés sont affichés dans la fenêtre de commandes Dos lorsque vous exécutez l'application et dans la "console Java" lorsque que l'applet tourne dans un navigateur.

Le problème cerné, nous pouvons contourner ce problème en surveillant les entrées et sorties de la souris de la zone de "Surface_de_dessin" (événements MOUSE_ENTER et MOUSE_EXIT) et ainsi savoir si les événements d'appui et de relâchement du bouton de la souris sont pertinents; voici un moyen de faire cela (en gras les modifications faites) :

Code 15 : Gestion d'événements souris supplémentaires

```
import java.applet.*;
import java.awt.*;

public class Dessin extends Applet {
    codage identique à celui du "code 14"
}

class Cadre_de_dessin extends Frame {
    codage identique à celui du "code 14"
}

class Surface_de_dessin extends Panel
{
    .....
//  nous ajoutons les déclarations de variables privées suivantes :
    private boolean dans_le_cadre;      //  où sommes-nous ?
    private boolean trace_en_cours; //  traçons-nous ?

    public Surface_de_dessin() {
        .....
    //  nous ajoutons les initialisation suivantes :
        dans_le_cadre = false;
        trace_en_cours= false;
    }

    public boolean handleEvent(Event event){
        Graphics g = getGraphics();

        switch (event.id) {
            case Event.MOUSE_DOWN:
                if (dans_le_cadre){
                    trace_en_cours = true;
                    ancre.x = event.x;
                    ancre.y = event.y;
                    bouge = false;
                }
                return true;
```

```
                case Event.MOUSE_UP:
                    if (trace_en_cours)
                        DrawXorLine(g, event, dans_le_cadre);
                    trace_en_cours = false;
                    return true;

                case Event.MOUSE_ENTER:
                    dans_le_cadre = true;
                    return true;

                case Event.MOUSE_EXIT:
                    dans_le_cadre = false;
                    return true;

                case Event.MOUSE_DRAG:
                    DrawXorLine(g, event, true);
                    ruban.x = event.x;
                    ruban.y = event.y;
                    bouge = true;
                    return true;

                default:
                    return super.handleEvent(event);
            }
        }

        protected void DrawXorLine(Graphics g, Event position, boolean retrace){
            if (bouge){
                g.setXORMode(Color.blue);
                g.drawLine(ancre.x, ancre.y, ruban.x, ruban.y);
            }
            if (retrace){
                g.setColor(Color.blue);
                g.setPaintMode();
                g.drawLine(ancre.x, ancre.y, position.x, position.y);
            }
        }
    }
```

Bien, voila notre application de dessin qui sait tracer des lignes ... tout en effaçant les précédentes et perdant tout au moindre rafraîchissement (“update”).

Quels objets stocker ?

Pour se souvenir des lignes tracées, nous devons le sauvegarder (évident, non ?); pour cela nous allons doter notre classe “Surface_de_dessin” d’un tableau d’objets qui permettent de récupérer les coordonnées des lignes tracées. Mais souhaitons-nous vraiment ne tracer que des lignes, il serait plus judicieux d’avoir une classe générique qui précise les services attendus, créer un tableau d’éléments de cette classe abstraite (abstraite car ce n’est jamais directement cette classe que nous utiliserons) et ensuite remplir ce tableau avec des instances de classes dérivées de celle-ci.

Concrètement, si nous imaginons une classe “Forme” qui contiendrait deux variables d’instances (propres à chaque objet créé, c’est à dire chaque instance de cette classe) de type “Point” ainsi qu’une méthode de tracé, par exemple “Trace” à qui nous fournirions une instance de “Graphics” (afin que l’objet sache où se dessiner), nous pourrions alors aisément définir de nouvelles classes qui définirait chacune leur façon de se dessiner; une première définition de cela pourrait être :

```
abstract class Forme {
    protected Point hautGauche;
    protected Point basDroite;

//  constructeur vide
    public Forme(){
        hautGauche  = new Point(0, 0);
        basDroite   = new Point(0, 0);
    }
//  constructeur avec deux points
    public Forme(Point upLeft, Point downRight){
        hautGauche  = new Point(upLeft);
        basDroite   = new Point(downRight);
    }
//  méthode abstraite de tracé
    public abstract void Trace(Graphics g);
}
```

Nous pourrions alors dériver cette classe abstraite pour définir des classes utilisables :

```
class monRectangle extends Forme {
    public void Trace(Graphics g){  // implémentation de la méthode de tracé
        g.drawRect(hautGauche.x, hautGauche.y,
            basDroite.x - hautGauche.x, basDroite.y - hautGauche.y);
    }
}

class maLigne extends Forme {
    public void Trace(Graphics g){  // implémentation de la méthode de tracé
        g.drawLine(hautGauche.x, hautGauche.y, basDroite.x, basDroite.y);
    }
}
```

Remarques :

1. la classe “Forme” doit être déclarée abstraite (“abstract”) car elle contient une méthode abstraite; cette méthode abstraite (la routine “Trace”) utilise également le mot clé “abstract” et ne fournit aucune implémentation pour cette fonction, par contre elle fixe la prototype de la fonction. Une telle classe ne peut pas être créée, par exemple “Forme forme = new Forme( );” provoquera une erreur de compilation puisque cette classe est inutilisable sans que la méthode “Trace” ne soit définie.

2. nous fournissons ici deux constructeurs différents, notez qu’il est possible d’en créer autant que l’on désire; nous aurions ainsi pût définir un troisième constructeur invoqué avec quatre entiers.

3. les classes “monRectangle” et “maLigne” étendent “Forme” en définissant la méthode “Trace”; ces nouvelles classes sont tout à fait instanciables et utilisables puisqu’elles ont défini toutes les (l’unique) méthodes abstraites de “Forme” (elles ont bouché tous les trous dans la définition de “Forme”).

Ayant défini ce modèle, nous pourrions doter notre classe “Surface_de_dessin” d’un tableau de “Forme” dans lequel nous rangerions, non pas des vrais objets “Forme”, mais des références à des instances des classes dérivées de “Forme”, dont “monRectangle” et “maLigne”.

Mais sommes-nous bien sur que ce modèle est ce que nous désirions; pour définir une ligne nous n’avons rencontré aucun problème et pour d’autres formes nous n’en rencontrons sûrement pas, mais nous avons par contre de grande chance de “réinventer la roue” ! En effet, la classe “Rectangle”, par exemple, existe dans le paquetage “java.awt”; elle dispose de constructeurs propre à satisfaire nos besoins ainsi que de nombreuses méthodes qui pourront s’avérer utiles quant nous aurons à tracer des rectangles.

Nous pourrions, puisque les méthodes de java.awt.Rectangle et de Forme nous intéressent, vouloir créer un objet qui étends à la fois “Rectangle” et “Forme”, soit quelque chose comme :

```
class monRectangle extends Rectangle, Frame { ...
```

ce procédé est connu en C++ comme l’héritage multiple; il est interdit en Java ... tant mieux, il y a beaucoup mieux !

En fait ce dont nous souhaitons disposer est un **comportement** : étant donné des objets (déjà définis ou non) nous voulons les doter d’une capacité à se dessiner; pour réaliser de telles taches Java fournit les interfaces; une interface ressemble à une définition de classes mais ne fait que fixer des prototypes pour les fonctions qui seront définies - c’est précisément ce que nous avions commencé à faire avec notre méthode abstraite.

N’importe quel objet peut ensuite implémenter une ou plusieurs interfaces en utilisant le mot clé “implement”; voyons comment définir nos classes avec cette stratégie :

```
public interface Dessinable {
//  définition de l'interface de tracé
    public abstract void Trace(Graphics g);
}
```

Nous utilisons alors cette interface dans de nouvelles classes qui “implémentent” ces services :

```
class Carre extends Rectangle implements Dessinable {
//  utilise les champs et les méthodes de la classe Rectangle
//  et implémente les méthodes de l’interface Dessinable
    public void Trace(Graphics g){
        g.drawRect(x, y, width, height);
    }
}

class Ligne extends Carre {
//  implémentation de la méthode de tracé
    public void Trace(Graphics g){
        g.drawLine(x, y, x + width, y + height);
    }
}
```

Avec cette approche nous récupérons dans “Carre” tout le savoir-faire de la classe “Rectangle” (dont ses champs “x”, “y”, “width” et “height”); de même, “Ligne” qui hérite de “Carre” aura les mêmes “pouvoirs”.

Nous allons utiliser ce principe pour mémoriser les lignes tracées, nous reviendrons ensuite sur le concept d’interface pour mieux le définir.

Mémoriser le tracé

Pour enregistrer les formes dessinées (pour l'instant continuons avec des lignes uniquement) nous devons donc déclarer puis utiliser un tableau de "Dessinable" qui pourra contenir tout objet implémentant cette interface; sur l'événement MOUSE_UP nous ne dessinons plus directement mais insérons une nouvelle forme dans ce tableau et provoquons le réaffichage de toutes les formes grâce à un "update" qui provoquera l'appel de la procédure "paint" où nous parcourons le tableau; ceci conduit à :

Code 16 : Mémorisation des lignes tracées

```
import java.applet.*;
import java.awt.*;

public class Dessin extends Applet {
    codage identique à celui du "Code 15"
}

class Cadre_de_dessin extends Frame {
    codage identique à celui du "Code 15"
}

// définition de l'interface de tracé
interface Dessinable {
    public void Trace(Graphics g);
}

class Carre extends Rectangle implements Dessinable {
    public Carre(int x1, int y1, int x2, int y2){
        super(x1, y1, (x2 - x1), (y2 - y1));
    }

//  implémentation de la méthode de tracé
    public void Trace(Graphics g){
        g.drawRect(x, y, width, height);
    }
}

class Ligne extends Carre {
    public Ligne(int x1, int y1, int x2, int y2){
        super(x1, y1, x2, y2);
    }

//  implémentation de la méthode de tracé
    public void Trace(Graphics g){
        g.drawLine(x, y, x + width, y + height);
    }
}

class Surface_de_dessin extends Panel
{
    nous ajoutons les déclarations de variables privées suivantes :
    private Dessinable figures[];      // tableau de formes
    private int compteur;              // nombre de formes stockées

    public Surface_de_dessin() {
        nous ajoutons les initialisation suivantes :
        figures = new Dessinable[100];
        compteur = 0;
        .....
    }
```

```
    public boolean handleEvent(Event event){
        Graphics g = getGraphics();

        switch (event.id) {
            case Event.MOUSE_DOWN:
            case Event.MOUSE_ENTER:
            case Event.MOUSE_EXIT:
            case Event.MOUSE_DRAG:
                codage identique à celui du “Code 15”

            case Event.MOUSE_UP:
                if (trace_en_cours == false)
                    return true;
                DrawXorLine(g, event, false);
                trace_en_cours = false;

                if (dans_le_cadre == false || compteur == 100)
                    return true;

            //  créé une Ligne
                Dessinable ligne = new Ligne(
                    ancre.x, ancre.y, event.x, event.y);
            //  insère cette ligne dans le tableau et incrémente le compteur
                figures[ compteur++ ] = ligne;
            //  redessine toutes les formes
                repaint();
                return true;

            default:
                return super.handleEvent(event);
        }
    }

    public void paint(Graphics g){
        g.setPaintMode();
        g.setColor(Color.blue);
        for (int i = 0; i < compteur; i++)
            figures[i].Trace(g);
    }

    protected void DrawXorLine(Graphics g, Event position, boolean retrace){
        codage identique à celui du “Code 15”
    }
}
```

A ce stade notre application commence remplir ces objectifs, il nous reste encore à implémenter d'autres formes et à fournir un moyen de sélectionner la forme à dessiner et sa couleur; de même l'affichage doit être amélioré, en effet lorsque toutes les lignes sont retracées, nous obtenons un effet de clignotement; ce phénomène est dû à l'implémentation de la méthode “update” (appelé via “repaint”) qui efface toute la surface avant d'appeler “paint” or ici nous savons bien que rien n'a besoin d'être effacé puisque justement nous cherchons à redessiner ce qui l'a été; pour améliorer notre application nous pouvons alors facilement redéfinir “update” afin qu'elle se contente d'appeler “paint” (à placer dans la définition de la classe “Surface_de_dessin”) :

```
public void update(Graphics g){
    paint(g);
}
```

Ceci corrige le clignotement, cependant l'effacement partiel des lignes déjà tracées durant le positionnement d'une nouvelle ligne reste désagréable à l'œil; nous pouvons pour corriger cela modifier la méthode “DrawXorLine(...)” afin que celle-ci redessine les lignes précédentes après avoir effacé la ligne en cours d'édition, soit :

```
protected void DrawXorLine(Graphics g, Event position, boolean retrace){
    if (bouge){
        g.setXORMode(Color.blue);
        g.drawLine(ancre.x, ancre.y, ruban.x, ruban.y);
    }
    if (retrace){
        repaint();
        g.setColor(Color.blue);
        g.setPaintMode();
        g.drawLine(ancre.x, ancre.y, position.x, position.y);
    }
}
```

Ces modifications suppriment les “bugs cosmétiques” et préservent la fluidité de fonctionnement; avant de poursuivre les améliorations revenons sur la définition du langage et des interfaces.

Héritage et Interface

L'héritage et la redéfinition de méthodes héritées

Pour hériter d'une classe "parent", une classe "fille" doit être déclarée :

```
[public] class fille extends parent {
    ...
}
```

la classe "fille" peut alors redéfinir (recouvrir) toutes ou aucune (et tous les intermédiaires) des méthodes de la classe "parent" (qui peuvent l'être, ceci exclus les "static"), c'est à dire fournir des méthodes ayant le même nom, même jeu de paramètres et même valeur de retour que des méthodes de la classe "parent" quand on utilisera une de ces méthodes, c'est la nouvelle méthode qui sera appelée.

Insistons sur ce dernier point : il est possible dans une méthode d'un objet d'appeler la méthode redéfinie de la classe parent mais il est impossible pour l'utilisateur de cette classe d'accéder à l'ancienne définition.

Exemple :

```
class A {
    public void foo(){
        System.out.println("Bonjour d'A");
    }
}

class B extends A {
    public void foo(){
        System.out.println("Bonjour de B");
    }
}
```

avec ces définitions, le code :

```
A a = new A();
a.foo();
```

affiche "Bonjour d'A"; tandis que :

```
B b = new B();
b.foo();
```

affiche "Bonjour de B"; ceci est sans surprise.

Modifions la définitino de "B" :

```
class B extends A {
    public void foo(){
        super.foo();
        System.out.println("et de B");
    }
}
```

Un appel à "foo( )" de la classe "B" afficherait maintenant "Bonjour d'A [retour ligne] et de B", donc une classe peut accéder aux méthodes qu'elle recouvre, ceci n'est pas nouveau puisque nous l'avions déjà vu précédemment; par contre utilisant toujours une instance de "B" il est impossible d'appeler la méthode "A.Foo( )".

Ainsi même le code :

```
B b = new B();      // instanciation de B
A a = (A) b;        // tentative de retypage
a.foo();
```

où nous retypons “b” en instance de “A” que nous stockons dans “a”, affichera le message de “B.foo( )”.

Nous le voyons, le principe d’héritage permet de spécialiser des classes d’objets en adaptant certaines de leurs méthodes (par la redéfinition) à une tache plus précise.

La Surcharge de méthodes

Une classe quelconque peut également surcharger ses méthodes ou des méthodes héritées, pour cela elle fournit des méthodes de nom équivalent mais utilisant un jeu de paramètres différents; la différenciation ne peut pas se faire uniquement sur le type retourné, en effet les conversions implicites d’un de ces types en un autre, rendrait équivalente les définitions et le compilateur ne saurait pas laquelle utiliser.

Exemple de surcharge :

```
class A {
    public void foo(){
        System.out.println("Bonjour tout le monde");
    }
    public void foo(String s){
        System.out.println("Bonjour " + s);
    }
}
```

nous pouvons alors coder :

```
A a = new A();
a.foo();                        //  affiche "Bonjour tout le monde"
a.foo("Duke");              // affiche "Bonjour Duke"
```

Les plus attentifs se disent déjà “ça y est, il nous refait le coup du ‘Hello world’ ”, rassurez-vous je referme ce rappel.

Notez néanmoins que le constructeur d’une classe accepte également la surcharge, et il est donc possible de fournir plusieurs constructeurs chacun avec un jeu de paramètres différents.

La surcharge est souvent utilisée pour permettre la réalisation d’une tache unique à partir de différentes situations. Supposons, par exemple, une classe où une méthode attends comme paramètre quatre entiers définissant les coordonnées de deux points extrêmes :

```
public void maFonction(int x1, int y1, int x2, int y2){
   ...
}
```

Il se trouvera, immanquablement, des situations où, lorsque nous voudrons appeler cette méthode, nous disposerons de ces coordonnées sous forme de deux instances de la classe Point (java.awt.Point) ou encore une instance de la classe Rectangle (java.awt.Rectangle); nous serons alors obligé, à chaque appel à “maFonction” d’extraire de ces instances les informations utiles.

Pour faciliter l'écriture du code et lever cette contrainte, nous pouvons également surcharger la définition de "maFonction" pour fournir d'autres interfaces à cette méthode :

```
public void maFonction(Point a, Point b){
   maFonction(a.x, a.y, b.x, b.y);
}

public void maFonction(Rectangle r){
   maFonction(r.x, r.y, r.x + r.width, r.y + r.height);
}
```

Grâce à cette surcharge, l'extraction des données de ces différents types d'instances est réalisée par notre classe et évite donc au code client ces opérations

Remarque : ici les différentes versions de la méthode ne font que "traduire" les paramètres reçus pour finalement réaliser le traitement via la première définition de la méthode.

La surcharge peut également être utilisée pour adapter le traitement à effectuer selon la nature des paramètres à traiter; imaginons, par exemple, une classe gérant les nombres complexes :

Code 17.1 : Surcharge du Constructeur

```
public class Complex {
    protected double re;
    protected double im;

    public Complex(){
        this.re = 0.0;
        this.im = 0.0;
    }

    public Complex(double re){
        this.re = re;
        this.im = 0.0;
    }

    public Complex(double re, double im){
        this.re = re;
        this.im = im;
    }

    public Complex(Complex z){
        this.re = z.re;
        this.im = z.im;
    }
}
```

Notez au passage que cette classe utilise la surcharge du constructeur pour offrir quatre façons de créer une instance de Complex.

Pour pouvoir additionner un nombre ou un complexe à une instance de Complex, nous fournirons les méthodes "add" surchargées suivantes :

Code 17.2 : Surcharge de méthodes

```
public class Complex {
    cf ci-dessus pour les champs et la définition des constructeurs

    public Complex add(double x){
        return new Complex(re + x, im);
    }

    public Complex add(Complex z){
        return new Complex(re + z.re, im + z.im);
    }
}
```

De même pour supporter la multiplication, nous définirons :

```
public class Complex {
    cf ci-dessus pour la définition de la classe

    public Complex mul(double x){
        return new Complex(x * re, x * im);
    }

    public Complex mul(Complex z){
        return new Complex(re * z.re - im * z.im, re * z.im + im * z.re);
    }
}
```

Ainsi, la surcharge permet d'offrir un service générique - car toujours identifié par une méthode nommée "add" ou "mul" - qui réalise différentes opérations selon la nature des données à traiter.

L'implémentation d'interface

Définition d'une interface

Pour définir une interface, on utilise la syntaxe :

```
[public] interface service [extends[1] interface1, interface2, ...] {
    modifieur[4] static final[2] type constante;
    modifieur[4] [abstract[3]] methode( parametres );
}
```

Explications :

1. Une interface peut étendre une ou plusieurs interface, ainsi si nous avons défini :

```
public interface serviceDeBase { ... }
public interface serviceEtendu { ... }
```

nous pouvons définir :

```
public interface tousServices extends serviceDeBase, serviceEtendu { ... }
```

qui regroupe alors toutes les méthodes des deux interfaces; cette nouvelle interface peut bien sur également ajouter de nouvelles déclarations de constantes et / ou méthodes supplémentaires.

2. Une interface peut définir des variables, ce sont obligatoirement des constantes donc "static" et "final" doivent être employés.

3. Les méthodes d'une interface sont toujours abstraites - puisque le code n'est jamais fourni - le mot clé "abstract" est donc optionnel, précisez-le pour améliorer la lisibilité du code si cela vous parait utile.

4. Les méthodes et les constantes, aussi bien déclarées dans une interface que dans une classe, définissent leur visibilité grâce à un modifieur parmi :

public	la variable / la méthode est utilisable par tout le monde
protected	la variable / la méthode n'est accessible que dans la classe et les classes dérivées
private	la variable / la méthode n'est accessible que dans la classe qui la définie
aucun	la variable / la méthode est accessible dans la classe et les classes dérivées, ainsi que dans les classes du même paquetage (sorte de "friend" du C++)

Utilisation d'une interface

Pour implémenter une interface "service", la classe "fille" doit être déclarée :

```
[public] class fille implements service {
    ...
}
```

La classe “fille” doit alors définir **toutes** les méthodes définies dans l’interface “service”.

Si, de plus, “service” est une interface étendant d’autres interfaces (comme “tousServices” ci-dessus), la classe “fille” doit implémenter toutes les méthodes de cette interface et des interfaces héritées.

Il est possible, pour une classe, d’implémenter plusieurs interfaces; ainsi si l’interface “tousServices” n’ajoutait avec définitions de méthodes, les deux définitions suivantes sont équivalentes :

```
[public] class fille implements tousServices {
    ...
}
```

et

```
[public] class fille implements serviceDeBase, serviceEtendu {
    ...
}
```

Voila qui clarifie les définitions; continuons à rendre les choses plus explicites en réorganisant notre programme de dessin.

Organisation du projet

Les paquetages

Un des reproches évident que nous pourrions faire au programme de dessin, tel que nous l'avons laissé, est son coté brouillon voire désordre avec la multitude de classes définies dans le même fichier source et la profusion de fichiers classe générés à la compilation.

La façon dont nous avons codé jusqu'alors est convenable pour la mise au point mais dès que nous désirons figer les classes ou interfaces définies, nous devons recourir aux paquetages.

Un paquetage permet d'organiser les classes en leur associant une sorte de nom de famille plus riche que leur simple nom; ainsi le nom d'une classe dans un paquetage est construit avec les éléments suivants :

nom du fournisseur **.** catégorie **.** nom de la classe

le paquetage "java.*" est l'exception qui ne comporte pas de nom de fournisseur puisque les classes qu'il contient sont génériques et présentes dans tout ensemble de Java.

Illustrons ceci, notre classe "Ligne" appartenait jusqu'alors au "paquetage par défaut" - puisque nous n'avons pas précisé de nom de paquetage - nous voudrions maintenant qu'elle appartienne au paquetage "frprog.dessin", nous réaliserons cela en utilisant simplement le mot clé "package" qui doit figurer obligatoirement en première ligne (hormis les commentaires) de notre ficher source, soit :

```
package frprog.dessin;
```

grâce à cette information notre classe devient plus sure car si nous utilisons simultanément une autre classe "Ligne", notre classe sera explicitement désignée par "frprog.dessin.Ligne" (de la même façon que "Applet" est entièrement qualifié par "java.applet.Applet") et il n'y a aucun risque qu'elle soit confondue avec une autre définition portant le même nom de classe mais appartenant à un autre paquetage.

Cette désignation de paquetage nous oblige à réorganiser nos sources, comme c'était un peu l'objectif, ce n'est pas gênant; le nom du fichier classe généré doit, en effet, correspondre à l'organisation des fichiers sur le disque.

Ainsi notre classe "frprog.dessin.Ligne" sera enregistrée dans un fichier placé sous :

[*classes*] \ frprog \ dessin \ Ligne.class

où *classes* désigne un répertoire listé dans la variable d'environnement "CLASSPATH"; **nous choisissons le dossier ".\classes" comme dossier *classes***; ce choix est arbitraire, par la suite nous nous référerons à cette organisation de fichiers; vous pouvez utiliser une autre localisation si vous le souhaitez.

Grâce à cette convention, l'organisation d'un paquetage correspond exactement à celle des fichiers classes; ceci aide à organiser des projets volumineux, même si le procédé force un peu la main.

Appliquons ces principes, nous allons créer les fichiers sous ".\classes\frprog\dessin\" et nous ajoutons le chemin ".\classes" à notre variable "CLASSPATH", nous définissons donc (dans notre fichier "autoexec.bat") :

```
set classpath=.\;.\classes\;c:\jdk\lib\classes.zip
```

où, le cas échéant, vous remplacerez ".\classes" par le chemin choisi et "c:\jdk" pour le chemin d'accès au kit Java de Sun; si vous utilisez un autre compilateur adaptez la modification en conséquence; de même si vous n'utilisez pas le système Dos/Windows,

reportez vous à la documentation du Kit de développement Java de votre plate forme pour définir la variable “CLASSPATH”.

Les fichiers sources seront, pour leur part, enregistrer sous une arborescence identique à la racine près; celle-ci peut être identique, et les fichiers sources (.java) seront alors au même niveau que les fichiers compilés (.class) associés; nous pouvons également choisir de stocker les sources du projets à part, nous utiliserons alors, pour le fichier définissant la classe “Ligne”, l’arborescence :

[*source*] \ frprog \ dessin \ Ligne.java

où *source* est arbitrairement choisi; pratiquement ce sera le répertoire courant, c’est à dire le répertoire parent de notre répertoire “classes”

Nous créons maintenant le fichier “Dessinable.java” (qui reprend la définition de notre interface).

Contenu de “[*source*]\frprog\dessin\Dessinable.java” :

Code 18.1 : L’interface “Dessinable”

```
package frprog.dessin;

import java.awt.Graphics ;
import java.awt.Rectangle ;

public interface Dessinable
{
    public abstract boolean traverse(Rectangle r);
    public abstract boolean intersecte(Rectangle r);
    public abstract void trace(Graphics g);
    public abstract void remplis(Graphics g);
}
```

nous profitons de cette réécriture pour définir d’autres méthodes dans notre interface :

- traverse détermine si le tracé de l’objet traverse un rectangle
- intersecte détermine si le remplissage de l’objet traverse un rectangle
- trace réalise le tracé du contour de l’objet
- remplis réalise le remplissage de la surface de l’objet

L’analyse de la classe “Ligne” (frprog.dessin.Ligne) montre que celle-ci devra disposer de variables d’instances pour stocker ses points extrêmes ainsi qu’une couleur de tracé; nous codons donc :

Contenu de “[*source*]\frprog\dessin\Ligne.java” :

Code 18.2 : La classe “Ligne”

```
package frprog.dessin;

import java.awt.*;

public class Ligne implements Dessinable {

    protected Point a;
    protected Point b;
    protected Color color;

    public Ligne(int x1, int y1, int x2, int y2, Color color){
        cf source joint pour le codage exact
    }

    public Ligne(Point a, Point b, Color color){
        cf source joint pour le codage exact
    }
```

```
public boolean traverse(Rectangle r){
    cf source joint pour le codage exact
}

public boolean intersecte(Rectangle r){
    return traverse(r);
}

public void trace(Graphics g){
    g.setColor(color);
    g.drawLine(a.x, a.y, b.x, b.y);
}

public void remplis(Graphics g){
    trace(g);
}
```

Remarques :

1. les champs (a, b, color) sont “protected” donc visibles uniquement par cette classe et les sous classes de “Ligne”.
2. nous fournissons deux constructeurs différents (Cf. “La Surcharge de méthodes”).
3. les méthodes “intersecte” et “remplis” sont peu pertinentes pour une ligne, nous devons néanmoins implémenter ces méthodes (Cf. “Utilisation d’une interface”)

Lors de la compilation des fichiers, nous devons préciser le dossier abritant l’arborescence des classes, c’est à dire notre dossier *classes*; pour cela nous utilisons l’option “-d *nom de dossier*” du compilateur Javac; de plus, le nom du fichier source doit comporter le chemin d’accès dans notre arborescence basée sur *source*, cela conduit à :

[*source*]\>javac -d [*classes*] frprog\dessin\Dessinable.java
et
[*source*]\>javac -d [*classes*] frprog\dessin\Ligne.java

ces dernières commandes supposent que [*classes*] figure dans la liste des répertoires de la variable CLASSPATH, si ce n’est pas le cas, vous etes obligé d’utiliser la commande :

```
[source]\>javac -classpath [classes];%CLASSPATH% -d [classes] source java
```

Dans les deux cas, la compilation créé les fichiers classes :

[*classes*]\frprog\dessin\Dessinable.class
et
[*classes*]\frprog\dessin\Ligne.class

Documentation des classes

Autant votre premier souci, si vous êtes encore un peu novice en Java, va être de comprendre l’intérêt et le rôle de chacune des fonctions, autant cette étape vite passée, il vous paraîtra indispensable d’avoir sous la main (pas sur une fiche déjà perdue trois fois) une documentation des classes que vous aurez définies afin de les utiliser sans hésitation; continuons donc notre grand nettoyage en faisant cette documentation.

Cette opération va être réalisée grâce à l’outil “JavaDoc” fournit avec le kit Sun; ce processeur récupère les commentaires que nous insérons dans notre code pour créer automatiquement une aide au format html.

Ceci est très commode puisque nous avons tous l’habitude de commenter nos sources; le travail est donc quasiment réalisé.

Pour réaliser les pages d’aide, JavaDoc recueille un certain type de commentaires, pas tous; il traite en effet les blocs commençant par le symbole “/**” et ignore ceux qui commencent par “/*”. De plus certains mots clé permettent de définir certaines rubriques - comme la

définition des paramètres reçus - et puisque la documentation générée est au format html, nous pouvons y insérer nos propres marques en cas de besoin.

Voici par exemple le fichier “Dessinable.java” avec ses commentaires de documentation; les autres classes sont également documentées dans les sources fournis; n’hésitez pas à consulter la documentation fournie avec le kit Sun pour plus d’informations.

Contenu de “[*source*]\frprog\dessin\Dessinable.java” :

Code 18.3 : L’interface “Dessinable” documentée

```
//  cette classe est stockée dans le paquetage suivant
package frprog.dessin;

//  importation de définition de classes
import java.awt.Graphics ;
import java.awt.Rectangle ;

/**
 *  Une interface fournissant des informations g&eacute;om&eacute;triques
 *  et des m&eacute;thodes d'affichage.
 *  @author     Sylvain Ferey
 *  @version    1.0, 10/11/97
 *  @since      frprog1.0
 */

public interface Dessinable {
    /**
    * Evalue l'intersection entre le trac&eacute; de l'objet et un rectangle.
    * L'objet d&eacute;termine si son contour traverse le rectangle
    *   reçu en param&egrave;tre.
    * @param    r   un rectangle
    * @return   une valeur logique indiquant si le contour traverse
    *           le rectangle (<font color='#0000FF'>true</font>)
    *           ou non (<font color='#0000FF'>false</font>)
    * @since    frprog1.0
    */
    public abstract boolean traverse(Rectangle r);

    /**
    * Evalue l'intersection entre la surface de l'objet et un rectangle.
    * L'objet calcule le plus petit rectangle le contenant
    *   et d&eacute;termine son intersection avec le rectangle
    *   reçu en param&egrave;tre.
    * @param    r   un rectangle
    * @return   une valeur logique indiquant si l'intersection
    *           est non nulle (<font color='#0000FF'>true</font>)
    *           ou nulle (<font color='#0000FF'>false</font>)
    * @since    frprog1.0
    */
    public abstract boolean intersecte(Rectangle r);

    /**
    * R&eacute;alise le trac&eacute; de l'objet.
    * Seul le contour est dessin&eacute;.
    * L'objet est responsable de la s&eacute;lection
    *   de la couleur &agrave; utiliser.
    * @param    g   une instance de Graphics
    * @since    frprog1.0
    */
    public abstract void trace(Graphics g);

    /**
    * R&eacute;alise le remplissage de l'objet.
    * L'objet est responsable de la s&eacute;lection
    *   de la couleur &agrave; utiliser.
    * @param    g   une instance de Graphics
    * @since    frprog1.0
    */
    public abstract void remplis(Graphics g);
}
```

Pour générer la documentation correspondant à cette classe, il suffit de lancer “javadoc Dessinable.java”; pour documenter toutes les classes nous utiliserons “javadoc *.class”; notez que l’option “-d *dossier*” permet d’indiquer l’endroit où les fichiers doivent être créés.

Pour créer la documentation de nos classes disposées selon l’arborescence présentée, nous utiliserons :

```
JavaDoc -d doc frprog\dessin\*.*
```

ceci suppose que le répertoire “doc” existe au niveau du répertoire courant.

Les fichiers d’archives

Maintenant que tous nos fichiers sont bien rangés et chaque classe documentée, il n’en reste pas moins que le nombre de fichier classe est toujours important et que mettre une telle applet sur le web avec tous ces fichiers ne va pas être élégant, de plus le transfert, réalisé un à un, de chacun de ces fichiers va être long; les archives nous viennent à la rescousse.

Un fichier archive ressemble beaucoup à une archive zip; les classes insérés dans une archive conserve leur désignation du chemin relatif et donc leur appartenance à un paquetage.

Ces archives de classes sont générées grâce à l’outil “jar” du kit Sun; la syntaxe est :

```
jar {ctx} [vfmM0] archive.jar dossier ou fichiers à inclure
```

vous devez utiliser une des options “c”, “t” ou “x” pour **c**réer, lis**t**er ou e**x**traire les classes de l’archive nommée; “v” active le mode ba**v**ard (“**v**erbose”); “f” spécifie que l’on fournit le nom de l’archive (“**f**ile”); le paramètre “0” (zéro) désactive la compression, les fichiers sont alors simplement ajoutés.

N’hésitez pas à consulter la documentation fournie avec le kit Sun pour plus d’informations.

Pour créer une librairie de nos classes disposées selon l’arborescence présentée, nous utiliserons :

```
[classes]\jar cf0 frprog.jar frprog
```

L’archive créée peut alors être utilisée à la place des fichiers classes de notre arborescence de classes, cependant pour indiquer à “java” qu’il trouvera nos fichiers dans cette archive, nous devons ajouter à la variable “CLASSPATH” le nom du fichier archive; ceci permet donc les deux stratégies suivantes :

1. vous ne créez pas d’archive ou vous préférez que vos applications et applets utilisent les fichiers classes, la racine de l’arborescence des fichiers classes sera alors précisée dans “CLASSPATH”.
2. Vous créez et utilisez une archive, il suffit alors d’ajouter le nom de cette archive (avec le chemin complet) dans “CLASSPATH”.

Remarque : pour utiliser la version applet de notre programme de dessin, il faut préciser dans la marque “<applet ...>” une référence à notre librairie, cela donne :

```
<applet
    code="Dessin.class"
    archive="[classes]/frprog.zip"
    width=250 height=250>
    Ce message est affich&eacute si votre navigateur
    n'est pas compatible Java.
</applet>
```

Dans les deux cas, l’organisation (l’arborescence) de vos fichiers sources et classes sera identique (afin qu’ils puissent être compilés), mais l’option archive vous permet, quand la mise au point est terminée, de produire un jeu minimum de fichiers pour distribuer le programme et surtout pour installer l’applet sur le web, en effet un seul fichier archive sera

alors téléchargé - ce qui est beaucoup plus rapide - et il n'est pas besoin de recréer l'arborescence sur la machine hôte.

Construction du paquetage frprog

Résumons, nous avons isolé chaque classe ou interface dans un fichier propre qu'il va falloir compiler avec "javac"; la documentation sera produite grâce à "javadoc" en traitant tous les fichiers sources enfin une archive sera créée par "jar" avec tous les fichiers classes (elle peut également contenir les sources).

Les étapes sont beaucoup plus nombreuses que la simple compilation d'un unique fichier comme nous le faisions précédemment; essayons d'automatiser ce processus.

Si vous avez l'expérience des outils en ligne de commandes, vous aurez certainement pensé à créer un "makefile", c'est à dire un fichier qui décrit quand les éléments doivent être reconstruits et comment.

Vous trouverez avec les sources fournis, dans le répertoire "Code.20", un tel fichier "makefile" prévu pour l'outil "nmake.exe" de Microsoft, un fichier "make.bat" invoque cet outil.

Pour ceux qui ne disposeraient pas de cet outil, nous allons créer un outil équivalent, notre besoin est simple : nous voudrions savoir si un fichier source est plus vieux qu'un fichier de classe afin de déterminer s'il doit être recompilé, par analogie déterminer si l'archive est reconstruite selon les fichiers classes et la documentation selon l'archive.

Un outil minimum est fourni (source et exécutable) dans les exemples, il est écrit en C et compilé avec Microsoft Visual C++ 5.0 afin de supporter les noms longs; ne le compilez pas avec le gnu gcc celui-ci ne guère pas ces noms longs.

Muni de cet outil et de deux fichiers batch, nous pouvons tout reconstruire (et que ce qui doit l'être) simplement en lançant le fichier "doIt.bat"; ces fichiers batch ne sont pas commentés ici car leur contenu est assez trivial.

Voici l'arborescence finale et les fichiers en présence :

```
[source]\
    doIt.bat            -- script pour tout reconstruire sans "nmake.exe"
    make.bat            -- script pour lancer "nmake.exe"
    makefile            -- fichier de dépendance pour "nmake.exe"

[source]\frprog\dessin\
    Dessinable.java     -- le source de l'interface "Dessinable"
    Ligne.java          -- le source de la classe "Ligne"

[source]\[classes]\
    frprog.jar          -- l'archive générée

[source]\[classes]\frprog\dessin\
    Dessinable.class    -- l'interface "Dessinable"
    Ligne.class         -- la classe "Ligne"

[source]\doc\
    les fichiers générés par "javadoc" (dont tree.html)

[source]\doc\images\
    des ressources graphiques dupliquées depuis le kit Sun

[source]\tools\
    UpToDate.exe        -- l'outil de vérification de dépendances
    UpToDate.c          -- le source de cet outil
    compile.bat         -- script batch pour compiler un source Java
```

Révisions du source "Dessin.java"

Notre librairie graphique (le paquetage "frprog.dessin") est complétée en créant un fichier source pour chacune des classes; ces classes étant toutes insérées dans le paquetage

“frprog.dessin”, il est inutile d’importer les classes de ce package, en effet, toute classe d’une paquetage “connaît” toutes les classes de ce paquetage.

Voir le dossier “Code.18” pour l’ensemble de ces sources”.

Pour exécuter l’application, vous pouvez cliquer le fichier “doIt.bat” qui recompilera si besoin les fichiers source et lancera java en fixant la liste des répertoires et fichiers archives à utiliser.

Si vous disposez de l’outil “nmake.exe” vous pouvez également utilisez le fichier “make.bat” pour obtenir le même résultat.

Conclusion

Ce cours a présenté les principes de la construction d’applets ou d’applications Java orientées dessin; à ce stade vous maîtrisez tous les éléments pour dynamiser vos sites web grâce aux applets.

La richesse de Java s’étends bien au delà de ce nous avons pu aborder ici; aussi nous vous encourageons à explorer ce fascinant langage qui sait répondre à tous vos besoins.