Plan du cours

- Chapitre 1 : un survol de Java.
- Chapitre 2 : compléments sur le langage.
  (classes abstraites, interfaces, String et StringBuffer, Threads, le package d'entrées/sorties java.io)
- Chapitres 3, 4, 5 et 6 : Java Swing.
  (composants, afficheurs, événements, dessins, images , dialogues et animation).

Chapitre 1: un survol de Java

- Un survol général
- Exceptions, paquetages
- Objets, Classes et Héritage

Généralités

"*Java est un langage simple, orienté objet, distribué, robuste, sûr, indépendant des architectures matérielles, portable, de haute performance, multithread et dynamique*"

[*définition Sun*]

Orienté objet :

- inspiré de C++ et Smalltalk
- Objet : unité ayant un ensemble de variables et de fonctions (ou "méthodes") qui lui sont propres.

Généralités

Orienté objet (suite) :

- les objets sont regroupés en classe, qui est l'unité de base en Java
- une classe organise un ensemble de variables et de fonctions qui seront disponibles dans les objets créés à partir de cette classe
- de nombreuses classes sont disponibles
- un objet créé à partir d'une classe est une "instance" de cette classe

Généralités

Distribué :

- les fonctions d'accès au réseau et les protocoles Internet courants sont intégrés

Robuste :

- typage des données très strict
- pas de pointeurs (un "garbage collector")

Sûr et portable :

- compilation en "byte code" pour une machine virtuelle. Ce code est vérifié (->"exception").

Généralités

Haute performance :

- très discutable car Java est un langage pseudo interprété.

Multi thread :

- on peut découper une application en unités d'exécution séparées et apparemment simultanées.

Dynamique :

- les classes peuvent être modifiées sans avoir à modifier les programmes qui les utilisent.

Un survol de Java

extrait en partie du livre de
Ken Arnold & James Gosling
"The Java programming Language",
Addison-Wesley, 1996

```
class HelloWorld {
        public static void main (String[ ] args ) {
                System.out.println ( "Hello, world" );
        }
)
```

```
class Fibonacci {
        /** Affiche la suite de Fibonacci pour les valeurs < 50 */

        public static void main (String[ ] args ) {
                int prec = 1;           // type int: entier signé sur 32 bits
                int suiv = 1;           // [ -2^32 ; 2^32 - 1]
                System.out.println (prec);
                while (suiv < 50) {
                        System.out.println (suiv);
                        suiv = prec + suiv ;          // nouveau suiv
                        prec = suiv - prec ;          // ancien suiv
                }
        }
}
```

Types primitifs

boolean	peut valoir **true** ou **false**
char	caractère 16 bits Unicode 1.1
byte	entier codé sur 8 bits (signé)
short	entier codé sur 16 bits (signé)
int	entier codé sur 32 bits (signé)
long	entier codé sur 64 bits (signé)
float	flottant codé sur 32 bits (IEEE 754-1985)
double	flottant codé sur 64 bits (IEEE 754-1985)

Commentaires

```
// ceci est un commentaire sur une seule ligne

/*  ceci est un commentaire sur plusieurs
  lignes, analogue aux commentaires de C
  ou C++. Il doit finir par  */

/** l'outil javadoc extrait les commentaires
commençant par /** pour générer une documentation
   du code en HTML  */
```

Constantes nommées

```
class TrucDeCercle {
   static final double  π = 3.1416; // π est un caractère unicode
}
class Couleur  {
   final static int TREFLES = 1;
   final static int CARREAUX = 2;
   final static int COEURS = 3;
   final static int PIQUES = 4;
}
```


Structures de contrôle

- Il existe en outre du **while**, un **do-while**, un **for**, un **if-else** et un **switch**.
- Pas de go-to, mais les instructions **break** et **continue** avec la possibilité d'utiliser un break étiqueté
- des instructions **try-catch** et une clause **finally** permettent de gérer le contrôle en cas d'erreurs exceptionnelles appelées "exceptions". (voir plus loin)

```
class FibonacciAmelioree {
        /** Affiche les premiers termes de la suite de Fibonacci
        en faisant suivre les termes pairs d'une étoile */
/* affichage          1: 1
                      2: 1
                      3: 2 *
                      4: 3                    etc.  */

        static final INDEX_MAX = 10;

        public static void main (String[ ] args ) {
                int prec = 1;
                int suiv = 1;
                String marque;

                System.out.println ("1: " + prec);
```

```
                for (int i = 2; i < INDEX_MAX; i++) {
                        if ( suiv%2 == 0 )
                                marque = " *";
                        else
                                marque = "" ;
                        System.out.println (i + ": " + suiv + marque );
                        suiv = prec + suiv ;          // nouveau suiv
                        prec = suiv - prec ;          // ancien suiv
                }
        } // fin de la méthode main
} // fin de la classe FibonacciAmelioree
```

Classes et objets

- Les objets ont tous un type: la classe de l'objet
- Chaque classe possède deux sortes de membres, des champs (ou encore variables, ou attributs) et des méthodes (i.e. des fonctions):
  - ✓ les champs sont des variables associées à une classe et à ses objets.
  - ✓ les méthodes contiennent le code exécutable, et la façon dont elles sont invoquées dirige l'exécution du programme.

```
class Point {
        public double x, y;
}
```

Objets

Conceptuellement, un objet est la modélisation complète d'une entité ou d'un concept du problème à traiter :

- les objets co-existent.
- ils communiquent par envoi de messages: un objet appelle une méthode d'un autre objet avec des paramètres.

Techniquement, un objet est une entité informatique regroupant des données (les champs ou attributs), et les algorithmes permettant de manipuler ces données (les méthodes).

Exemple d'une interface graphique

Un algorithme capable de gérer tous les événements utilisateurs est inconcevable. L'idée de la programmation orientée objet est de modéliser chacun des éléments graphiques de l'interface par un objet en soi, capable de communiquer avec les autres par l'envoi de messages. L'utilisateur est également considéré comme un objet : il envoie des messages (clic souris, frappe clavier) qui déclenchent l'exécution des méthodes.

Chaque objet a un environnement composé des objets qu'il connaît. Un objet A connaît un objet B si l'un de ses champs est une référence sur B.

Principes de la POO

- L'algorithme est morcelé : chaque objet possède ses propres méthodes pour répondre aux messages, et chaque objet est responsable des interactions avec son environnement qui est constitué des objets qu'ils connaît, mémorisés dans ses champs (ou accessible par héritage)
- Les algorithmes de chaque objet sont plus faciles à concevoir et plus concis qu'un algorithme global qui tenterait de tout gérer

Principes de la POO

Principe de non-intrusion : on ne travaille pas sur un objet depuis l'extérieur de l'objet ; le bouton Clear ne va pas lui-même nettoyer les champs de saisie d'une fenêtre.

- Intérêt : si on rajoute un champ de saisie, on ne modifiera pas le code du bouton Clear.

Principe de délégation : quand un travail doit être fait, on demande aux autres objets de le faire.

- intérêt : on encapsule et on localise le code.

Création d'objets

- Les objets sont créés au moyen du mot clé **new**.
- L'allocation est effectuée par Java à l'intérieur d'une zône spécifique: le tas. Il y a un ramasse miettes qui gère automatiquement les références aux objets dans le code.
- Pour **initialiser** les champs des objets créés, on peut définir des méthodes de classes particulières : les constructeurs.
- Toutes les classes étendent implicitement la classe générique `Object` et les objets héritent de ses méthodes.

Création d'objets

```
Point  coinGauche = new Point ();
Point  coinDroit = new Point ();

coinGauche.x = 0.0 ;
coinGauche.y = 0.0;

coinDroit.x = 1280.0 ;
coinDroit.y = 1024.0 ;
```

Chaque objet possède sa propre copie de x et y

Champs statiques ou de classe

```
class Point {
   public static Point origine = new Point ();
   public double x, y;
   …
}
```

Du fait de la déclaration static, il n'existera qu'une copie de la donnée appelée Point.origine et faisant référence à un objet (0,0).

On parle alors de variables de classe, car leurs valeurs sont partagées par toutes les instances.

Méthodes et paramètres

```
class Point {
        public double x, y;
        public void clear ( ) {
                x = 0 ;
                y = 0 ;
        }
        public double distance (Point autre) {
                double xdiff, ydiff;
                xdiff = x - autre.x;
                ydiff = y - autre.y;
                return  Math.sqrt(xdiff * xdiff + ydiff * ydiff);
        }
}
```

Surcharge de méthodes

- ➢ Une méthode peut être définie plusieurs fois à condition de se différencier par le nombre et/ou le type de ses paramètres.
- Le compilateur décidera de la bonne méthode à utiliser en fonction du type des arguments d'appel.
- ☹ Java ne supporte pas la surcharge des opérateurs prédéfinis (contrairement à C++).

```
class BarreDeProgression {
        float pourcent ;
        …
        void setPourcent (float valeur ) {   pourcent = valeur;       }
        void setPourcent (int nb, int total ) {
                pourcent = total/nb;
        }
}
```


Méthodes statiques ou de classe

- Des méthodes déclarées **static** peuvent également être associées aux classes.
- Dans ce cas, elles ne pourront accéder qu'aux membres statiques de la classe et définiront donc des méthodes portant sur les variables de classe.
- Elles sont utilisées par la classe Math qui comporte des méthodes statiques implémentant des fonctions mathématiques. Cette classe ne fait que regrouper un ensemble de fonctions indépendantes.

Champs statiques ou de classe

Résumé: Les attributs et les méthodes déclarées `static` ne sont rien d'autre que des variables et des fonctions globales appartenant à la classe.

- un attribut static est une variable de classe ;
- une méthode static est une fonction ordinaire.

> Les attributs static peuvent permettre de faire communiquer entre eux les objets d'une même classe.

> De manière générale, ces éléments permettent de mettre en place des schémas de programmation impérative.

La référence `this`

```
class Point {
        public double x, y;

        public void clear ( ) {
                this.x = 0 ;
                this.y = 0 ;
        }
        …
        public void placer (double x, double y) {
                this.x = x ;  // le x à droite désigne le paramètre
                this.y = y ;
        }
}
```

Tableaux

- Un tableau est une collection de variables de même type, accessibles par un indice.

```
class Jeu {
        static final int TAILLE_JEU = 52;

        Carte [ ]  paquet = new Carte [TAILLE_JEU];

        public void print () {
        for (int i = 0; i < paquet.length; i++ )
                        System.out.println (paquet[i]);
        }
}
```


Tableaux

- On peut déclarer les tableaux avec les deux syntaxes :
  - type[ ] nom;
  - type nom [ ];
- Leur taille est invariable (fixée à la création) mais est allouée dynamiquement par l'opérateur **new.**
- Il n'y a donc pas de tableau statique.
- La taille est consultable par length. Si on sort des bornes 0 et length-1: IndexOutOfBoundsException

```
int table [ ] = { 1, 2, 3, 4, 5 };
int  [ ] [ ] matrice = new int [10][20];
System.out.println (table.length);
System.out.println (matrice.length); // 1ère dimension
System.out.println (matrice[0].length); // 2ème dimension
```

Les objets `String`

- La classe `String` fournit des méthodes de manipulation de chaînes de caractères.
- La longueur des objets String n'a pas à être déterminée à la création, mais on n'accède à leur valeur qu'en lecture. (Si on change la valeur d'une variable de type `String`, on change en réalité sa référence).

```
String monNom = "Recanati";
String specialite = "Informatique";

monNom = monNom + " Catherine";
monNom += " " + "(" + "specialite " + specialite + ")";
System.out.println("Nom = " + monNom);
```


La méthode `main`

On invoque toujours la commande **java** avec le nom d'une classe qui dirige l'application. Le système localise cette classe et exécute alors la méthode `main` de cette classe avec en argument unique, les arguments d'appel du programme. Cette méthode est nécessairement statique :

```
public static void main (String [ ] args)  {
    // arg!!! ce ne sont pas les mêmes args qu'en C !
}
```

Exemple

```
class Echo  {
        public static void main (String [ ] args)  {
                for (int i = 0; i < args.length;  i++)
                        System.out.print (args[i]  +  " ");
                System.out.println ();
        }
}
```

Sous Unix (ou sous DOS), on aurait par exemple l'exécution

```
% java Echo ici et maintenant
ici et maintenant
```

Ainsi, si on crée une classe principale intitulée `MyAppli`, on devra mettre son code dans un fichier intitulé MyAppli.java et ce code devra contenir une procédure `main`.
On compile le code avec

```
% javac MyAppli.java
```

Ce qui génère un fichier MyAppli.class
On lance ensuite l'exécution du programme avec

```
% java MyAppli
```

Qui exécute le code de la procédure `main`.

> Généralement, il faut instancier dans le `main` la classe MyAppli pour accéder à ses fonctions.

Exemple

```
import java.io.*;

class MyAppli {

        public void myProc (Object o) {
        … code de la procedure qui imprime par exemple
        le nom de la classe de l'objet o …

        }

        public static void main (String[ ] args) {

                MyAppli  prog = new MyAppli();

                prog.myProc (System.out);

        }
}
```

Héritage

L'héritage de classe a deux types de fonctions:

• une fonction de modélisation.

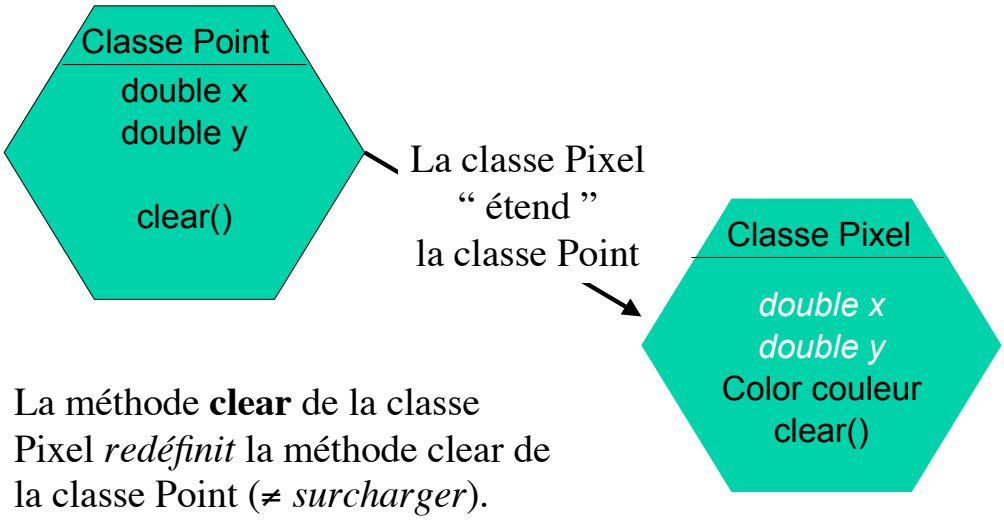
Etant donnée une classe d'objets, on peut la **partitionner** en sous-classes. Ex : la classe ObjetGraphique peut être partitionnée en Cercle, Rectangle, etc. On peut aussi la **raffiner** en créant une sous-classe. Ex : la classe Etudiant peut être spécialisée en EtudiantMaster.

• une fonction d'architecture logicielle.

Les sous-classes d'une classe partage les méthodes et les attributs de la classe mère.

(Cette fonction de **partage de code** est en fait plus importante dans les langages à héritage multiple : Eiffel et C++).

Héritage



La méthode **clear** de la classe Pixel *redéfinit* la méthode clear de la classe Point (≠ *surcharger*).

Héritage

```
class Pixel extends Point {
    Color   couleur;

    public void clear () {
        super.clear();          // méthode clear de Point
        couleur = null;
    }
};
```

On peut ainsi soit *étendre* le comportement, soit le *restreindre* (en limitant par exemple les valeurs).
[Rem: ambiguïté fondamentale]

Héritage

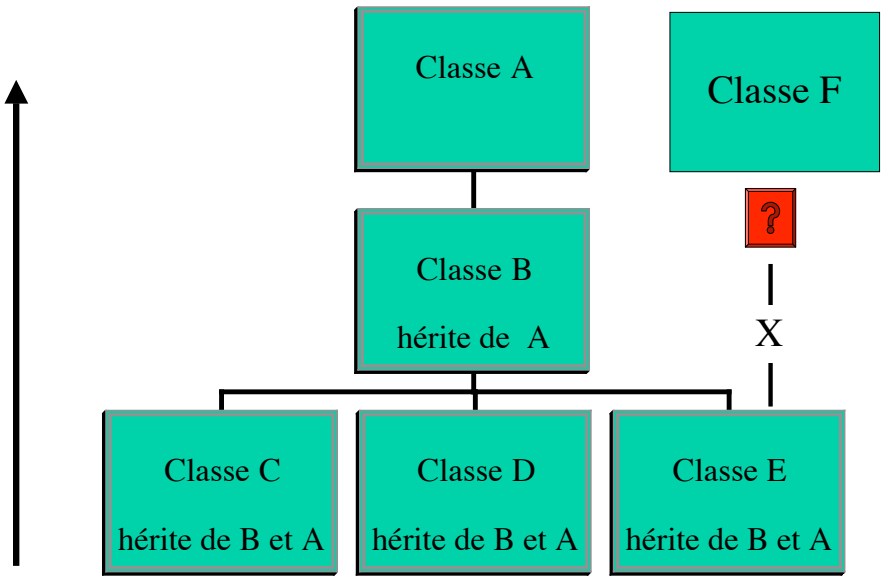
L'héritage de classe peut donc avoir différentes significations de modélisation :

- . . . est une sorte de . . .
- . . . est une extension de . . .
- . . . est une spécialisation de . . .

Sans oublier la simple fonction de « factorisation » i.e. de « partage de code ».

Attention aussi à ne pas confondre héritage et composition. (ex: Point et Segment).

Héritage simple



```
public class Animal {
   public void dormir () {System.out.print(" dormir-animal.");}
   public void jouer () {System.out.print(" jouer-animal.");}
   public void procreer () {System.out.println(" procreer-animal.");}
}
public class Mammifere extends Animal {
   public void procreer ()  {System.out.println(" procreer-mamif.");}
}
public class Chat extends mammifere {
   public void jouer () {System.out.println(" jouer-chat!");}
   public void miauler () {System.out.println(" miauler-chat!");}
   public static void main (String [ ] args) {
         Chat minet = new Chat ();
         minet.dormir ();  minet.procreer (); minet.jouer ()
   }
}
>   dormir-animal. procreer-manif. jouer-chat!
```

Les interfaces

Les interfaces permettent de déclarer la signature de méthodes, sans en fournir l'implémentation. Une classe qui implémente une interface s'engage (**par contrat**) à donner une implémentation effective de ces méthodes.

```
interface Recherche {
    /** retourne la valeur asociée au nom, ou null si une
    telle valeur n'existe pas */
    Object find (String name);
}
```

Les interfaces

Les interfaces permettent de décrire ce que des objets peuvent faire. Une interface est en général juste composée de signatures de méthodes.

Une signature de méthode est constituée :
- du nom de la méthode ;
- des noms et des types des paramètres de la méthode ;
- du type du résultat renvoyé par la méthode.

Une classe d'objet implémente une ou plusieurs interfaces. Cela signifie qu'elle donne une implémentation de toutes les méthodes décrites dans les interfaces qu'elle implémente.


```
class RechercheSimple implements Recherche {
        private String [ ]  noms;
        private Object [ ] valeurs;

        public Object find(String nom) {
                for (int i = 0; i < noms.length; i++) {
                        if (noms[i].equals(nom) )
                                return valeurs[i];
                }
                return null;   // pas trouvé
        }
}
```

Les interfaces

☺ Une interface peut aussi contenir des variables constantes (nommées). Elle joue ainsi un rôle analogue à celui des fichiers .h (inclus par #include) en C, qui pouvaient être partagés par plusieurs fichiers et définissaient des constantes.

☺ Une classe peut implémenter autant d'interfaces qu'elle le souhaite. Les interfaces compensent donc un peu l'absence d'héritage multiple en Java.

Les exceptions

• Beaucoup de méthodes sont susceptibles de "lever"(ou "lancer" = *to throw*) une exception, qui sera détectée juste avant qu'elle ne se produise.

• Une exception peut être gérée par le programmeur (on dit qu'elle est "capturée") en utilisant la séquence `try-catch-finally`. C'est la manière de programmer le traitement des erreurs.

•Si aucune méthode ne "capture" (= *to catch*) l'exception, le code sera interrompu par un signal d'erreur.

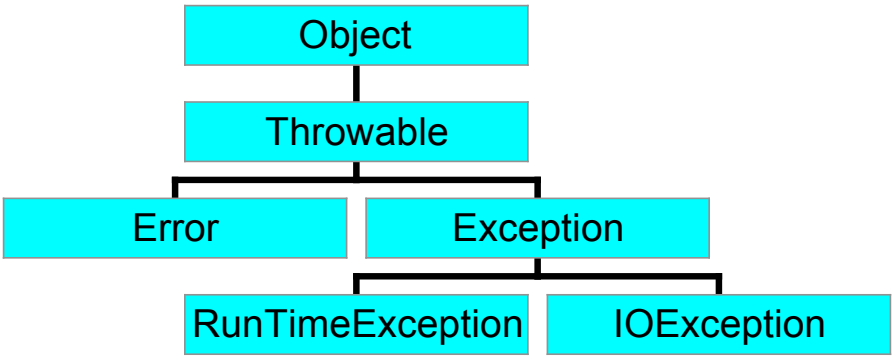
Les exceptions

Structure d'un bloc try-catch-finally :

```
try  <instruction>;    // ou blocd'instructions: { }
catch (<type e1> e) <instruction>; // ou bloc ibid { }
[ catch (<type e2> e) <instruction>; ]
[ catch (<type e3> e) <instruction>; ]
…
[ finally <intruction>;  ]
```

Les exceptions

Une exception est un objet du type Throwable



Les exceptions

•Si une méthode peut lancer une exception vérifiée, directement par `throw`, ou *indirectement* en invoquant une méthode qui lance elle-même une exception, elle doit le déclarer explicitement (avec `throws`.)

•Limitation de cette règle : les exceptions de type `Error` ou `RuntimeException` n'ont pas besoin d'être déclarées.

```
        class MoyenneIllegaleException extends Exception {
        }

        class MonUtilitaire {
        public double moyenne(double [ ] vals, int i, int j)
                        throws MoyenneIllegaleException {
              try {
                    return (vals[i] + vals [j] ) / 2;
              } catch (IndexOutOfBoundsException e) {
                    throw new MoyenneIllegaleException();
              }
         }
}
```

Les exceptions

Rem: on doit gérer les erreurs par des exceptions mais les exceptions peuvent aussi parfois servir à programmer :

```
public class DataTree
{
     public boolean search(Data d) {
          try {
               auxSearch(d) ;
               return false ;
          } catch (Exception e)
               { return true ; }
     } ;
```

Les exceptions

```
    private void auxSearch(Data d)
                throw Exception {
          if (isLeaf()) {
                if (d.equal(leafValue())) {
                      throw new Exception() ;
                } ;
          } else {
                auxSearch(leftSubTree()) ;
                auxSearch(rightSubTree()) ;
          } ;
    }
}
```


Les paquetages

Lorsque l'on attaque un problème, on commence par identifier les catégories d'objets présentes dans le problème. Chacune de ces catégories doit donner lieu à un *package*.

Un *package* est un ensemble de classes d'objets regroupées dans un répertoire spécifique. Les *packages* sont placés dans une arborescence correspondant à une hiérarchie de répertoires.

Les paquetages

• En Java, la notion de *package* qui regroupe un ensemble de classes et d'interfaces permet d'éviter des confusions de noms (comme `get`, `set`, `put`, `clear`, etc.).

• Les *packages* sont nommés et peuvent être importés. Ils peuvent ainsi être imbriqués.

• Les noms des *packages* sont également hierarchisés, les composants étant séparés par des points.

Les paquetages

☹ Pb insoluble des noms : deux projets distincts pourraient cependant avoir un même nom de paquetage.

➢Pour éviter ce type d'erreur, la convention standard est d'utiliser le nom de domaine Internet inversé comme préfixe au nom du paquetage: ainsi la société Acme, dont le domaine est acme.com, nommera ses paquetages `com.acme.nom`.

➢Ainsi, un nom complet sera non ambigu

Les paquetages

```
/* class Date1 {
        public static void main (String[ ] args) {
                java.util.Date  now = new java.util.Date ();
                System.out.println(now);
        }
} */
import java.util.Date
class Date2 {
        public static void main (String[ ] args) {
                Date  now = new Date ();
                System.out.println(now);
        }
}
```


Les paquetages

- java.util fournit des classes « utiles », par exemple: `BitSet`, `Vector`, `Stack` et `Date`.
- Les packages de base sont :

java.lang	principales classes de java
java.io	Entrées/Sorties vers des périphériques
java.util	Utilitaires (tables de hachage, etc.)
java.net	Support réseau (sockets, URL, etc.)
java.applet	Classes de base pour les applets
java.awt	Interface graphique de base

Les paquetages

```
import java.lang.*; // inutile, car ce package est implicite
import java.awt.*;  // pour l'exemple, mais non utilisé ici
-----
package MonPaquetage; // le package est nommé en tête
                      // du fichier source
public class MaClasse {
        public void test () {
                System.out.printl ("test !");
        }
}
```


Les paquetages

- une instruction package toto; doit figurer en tête du fichier source
- les packages peuvent être imbriqués: dans ce cas, le fichier n'a pas d'autre contenu que la déclaration d'autres packages comme import toto.*;
- de nombreux enironnements de développement reflètent les noms de paquetages dans le système de fichier, souvent en demandant que tout le code d'un unique packetage soit dans le même répertoire, et que le nom du répertoire reflète le nom du packetage. Renseignez-vous.

Les paquetages

- Les classes java sont importées par le compilateur au moment de la compilation et par la machine virtuelle au moment de l'exécution
- Les techniques de chargement des classes varient avec les machines
- la variable d'environnement CLASSPATH référence dans la plupart des cas les répertoires qui hébergent des packages susceptibles d'être importés. On y trouvera généralement, le package par défaut, identifié par le . du répertoire par défaut, les packages standards (/usr/java/..), et les packages personnels qu'il vous faudra rajouter

Objets, Classes et Héritage

Les constructeurs

- Un objet créé reçoit un état initial. Le code effectuant l'initialisation peut avoir besoin de données initiales, ou peut vouloir effectuer d'autres opérations. Les constructeurs remplissent ce rôle.
- Un constructeur porte le même nom que sa classe et peut prendre des paramètres.
- On peut avoir plusieurs constructeurs, distingués par le type de leurs paramètres.


```
class CorpsCeleste {
        public long numId;
        public String nom = "<sans nom>";
        public CorpsCeleste tourne = null;
        private static long idSuiv = 0;

        CorpsCeleste () {
                numId = idSuiv++;
        }
}
//  … ailleurs
CorpsCeleste soleil = new CorpsCeleste ();
soleil.nom = "Soleil";
CorpsCeleste terre = new CorpsCeleste ();
terre.nom = "Terre";
terre.tourne = soleil;
```

```
CorpsCeleste (String name, CorpsCeleste centre) {
        this () ;       // invoque le précédent constructeur
        nom = name;
        tourne = centre;
}
…
CorpsCeleste soleil = new CorpsCeleste ("Soleil", null);
CorpsCeleste terre = new CorpsCeleste ("Terre", soleil);
```

Remarques:

1. On pourrait aussi avoir un constructeur à un argument pour les corps célestes qui ne sont pas en orbite autour d'un autre.
2. Définir un constructeur non public permet de restreindre son utilisation.

Environnement d'un objet

L'environnement d'un objet est constitué des objets qu'il connaît (=la plupart du temps ses attributs). Cet environnement doit être minimal en fonction du principe de délégation.

Ex: le bouton Clear n'a pas à connaître le contenu de la fenêtre qu'il efface.

> Par défaut, les attributs d'un objet doivent donc être protégés (par le mot clé `private`) pour empêcher qu'un objet extérieur ne les utilise.

– Rem : l'environnement d'un objet doit être établi à la création de l'objet. Cela peut poser de petits problèmes lorsque deux objets doivent mutuellement se connaître.

Contrôle d'accès et héritage

- 4 niveaux d'accès aux membres d'une classe:

`public`	accessible partout où la classe est accessible, et hérité par les sous-classes.
`private`	accessible seulement depuis de la classe.
`protected`	accessible depuis la classe et ses sous-classes, et également depuis le package.
(défaut) paquetage	accessibles seulement depuis le **même** paquetage, et hérités uniquement par les sous-classes y figurant.

Les méthodes

- Les méthodes sont invoquées sur des classes ou sur des objets via des références et l'opérateur noté "."
  - `référence.méthode(paramètres)`
- La plupart du temps, les champs des objets sont privés (`private`) et on utilise des méthodes publiques (`public`) pour en contrôler l'accès.
- Remarque: Java supporte les méthodes à nombre d'arguments variable

Les méthodes

- Tous les attributs d'un objet doivent être protégés avec le mot-clé private.
- Si nécessaire, une méthode doit permettre d'accéder à la valeur de l'attribut :

```
private int valeur ;
public int getValeur() { return valeur ; }
```

- Si des objets extérieurs doivent pouvoir modifier l'attribut, ne leur permettre de le faire qu'à travers une méthode:

```
public void setValeur(int _valeur) { valeur = _valeur ; }
```


Intérêt des attributs privés

- Faciliter les évolutions du logiciel.
  - Un attribut peut avoir à être remplacé par une fonction.
- Maintenir la cohérence des objets.
  - Dans une classe Rectangle, le fait de modifier l'attribut `largeur` entraîne une modification de l'attribut `surface`.
- Maintenir la cohérence de l'application.
  - Dans un éditeur graphique avec une classe `Rectangle`, le fait de modifier l'attribut `largeur` entraîne le réaffichage de l'objet.

Utiliser les méthodes pour protéger les accès :

```
class CorpsCeleste {
        private long numId;  // numId sera ainsi protégé
        public String nom = "<sans nom>";
        public CorpsCeleste tourne = null;
        private static long idSuiv = 0;

        CorpsCeleste () {
                numId = idSuiv++;
        }
        public long getNumId()  {
                return numId;
        }
        //  … mais il n'y aura pas de setNumId
}
```

Une autre méthode de la classe `CorpsCeleste` :

```
public String toString() {
    String desc = numId + " (" + nom + ")";

   if   (tourne != null)
            desc += " tourne autour de " + tourne.toString();
   return desc;
}
```

Si un objet dispose d'une méthode `toString`, elle sera utilisée pour convertir cet objet en chaîne si nécessaire, ainsi :

```
System.out.println ("Corps céleste " + soleil);
System.out.println ("Corps céleste " + terre);
```

donneront les affichages suivants :

Corps céleste 0 (Soleil)
Corps céleste 1 (Terre) tourne autour de 0 (Soleil)

Attention: Les paramètres sont passés par valeur

```
class PassByValue {
    public static void main (String [ ] args)  {
            double one = 1.0;
            System.out.println("avant: one = " + one);
            divise(one);
            System.out.println("après: one = " + one);
    }
    public static void divise(double arg)  {
            arg  /=  2.0;          // divise arg par 2
            System.out.println("divise: arg = " + arg);
    }
}
```

produit l'affichage

```
avant: one = 1
divise: arg = 0.5
après: one = 1
```

Mais si le paramètre est une référence d'objet :

```
class PassByRef {
    public static void main (String [ ] args)  {
        CorpsCeleste sirius = new CorpsCeleste("Sirius", null);
        System.out.println("avant: " + sirius);
        nomCommun(sirius);
        System.out.println("après:  " + sirius);
    }
    public static void nomCommun(CorpsCeleste corpsRef)  {
        corpsRef.nom ="Etoile du berger";
        corpsRef = null;
    }
}
```

produira l'affichage

```
avant: 0 (Sirius)
après: 0 (Etoile du berger)
```

Le mot-clef `this`

On a vu que `this` permettait d'invoquer en première instruction d'un constructeur, un autre constructeur

On peut aussi l'utiliser pour passer la référence de l'objet courant (sauf dans les méthodes statiques). Par exemple, pour ajouter l'objet à une liste d'objets :

```
Service.add(this)
```

A l'intérieur d'une méthode, il est implicite devant toute référence à un membre

L'affectation sur `str` dans le constructeur `Nom` :

```
class Nom {
        public String str;

        Nom () {
                str = "<sans nom>";
        }
}
```

est équivalente à

```
this.str = "<sans nom>";
```

Par convention, on n'utilise `this` que lorsque cela est nécessaire, c'est-à-dire lorsque le nom du champ de l'objet est masqué par celui d'une variable.

La méthode `finalize`

- On crée les objets avec `new`, et le "delete" correspondant est la méthode `finalize`.
- Un *garbage collector* récupère l'espace occupé par les objets java qui ne sont plus référencés, mais la méthode `finalize` sera exécutée avant que l'espace ne soit réclamé, et lorsque la machine virtuelle achève son exécution. Elle permet donc de gérer les ressources non java.

La méthode `finalize`

Elle a le prototype suivant:

```
protected void finalize () throws Throwable  {
    super.finalize ();
    //  … et on libère ici les ressources externes
}
```

Mais la présence du garbage collector rend la méthode `finalize` inutile dans la plupart des cas.

```
public class ProcessFile {
        private Stream file;
        public ProcessFile (String path) {
                file = new Stream(path);
        }
                // …
        public void close () {
                if (file != null) {
                        file.close();
                        file = null;
                }
        }
        protected void finalize () throws Throwable {
                super.finalize();
                close();
        }
}
```

Le mot-clef `super`

- Le mot-clef `super` peut être utilisé dans toutes les méthodes non statiques. Il désigne une référence à l'objet courant, en supposant que celui-ci est une instance, non pas de sa classe, mais de sa superclasse.

➢ On l'utilise ainsi pour accéder à des méthodes redéfinies.

➢ Il permet aussi de définir les constructeurs des sous-classes.

Exemple: une classe de paires nom-valeur

```
class Attribut {
        private String name;
        private Object value;

        public Attribut(String nom) {
                name = nom;
        }
        public Attribut(String nom, Object valeur) {
                name = nom;
                value = valeur;
        }
        public String getName() {
                return name;
        }
```

```
        public Object getValue() {
                return value;
        }
        public Object setValue(Object newVal) {
                Object oldVal = value;
                value = newVal;
                return oldVal;
        }
}
class AttributCouleur extends Attribut  {
        private ScreenColor  color;  // sa valeur (String) décodée
        public AttributCouleur (String name, Object value) {
                super (name, value);
                decodeColor();    // traduit value (String) dans color
        }
```

```
        public AttributCouleur (String name) {
                this (name, "transparent");        // valeur par défaut
        }
        public AttributCouleur (String name, ScreenColor val){
                super (name, val.toString() );
                color = val;
        }
        /** Retourne l'objet ScreenColor */
        public ScreenColor getColor ( ) {
                return color;
        }
        public Object setValue (Object newVal) {
        // exécuter d'abord la méthode setValue de la superclasse
                Object retVal = super.setValue(newVal);
                decodeColor();
                return retVal;
        }
```

```
    /** Affecte la couleur directement avec un ScreenColor */
    public ScreenColor setValue(ScreenColor newVal) {
    // exécuter d'abord la méthode setValue de la superclasse
        super.setValue(newVal.toString() );
        color = newVal;
        return newVal;
    }
    /** Traduit la valeur (String) en un ScreenColor */
    protected void decodeColor () {
        if (getValue() == null)
                color = null;
        else
                color = new ScreenColor(getValue());
    }
}
```

IMPORTANT

- On utilise généralement un appel au constructeur de la super-classe pour définir le constructeur d'une sous-classe.
- De même, on appelle souvent les méthodes de la super-classe pour les redéfinir ou les surcharger dans une sous-classe.
- On appelle toujours la procédure `finalize` de la super-classe dans la procédure `finalize` d'une sous-classe.

➢ Ce mécanisme d'extension est une technique de base de la programmation orientée objet.

➢ On en verra d'autres exemples avec les packages graphiques.

Méthodes et classes finales

- Une méthode déclarée `final` ne peut être redéfinie par aucune sous-classe. Il s'agit donc de la *version finale* de cette méthode.
- Une classe finale ne peut pas être étendue. (Toutes ses méthodes sont donc implicitement finales).
- Les méthodes finales sont plus efficaces, car l'invocation au moment de l'exécution peut être optimisée.

La classe `Object`

public boolean equals (Object obj)

teste l'égalité du receveur et de obj

public int hashCode ()

retourne le code de hachage utilisé par les Hashtable

protected Object clone () throws CloneNotSupportedException

retourne un duplicata de l'objet qui l'invoque

public final Class getClass()

retourne un Object de type Class

protected void finalize () throws Throwable

Comparaison d'objets

On ne compare pas les objets en comparant leurs variables d'instance

```
rect1 = new Rectangle (10,20);
rect2 = new Rectangle (10,20);
rect3 = new Rectangle (30,60);
```

rect1 == rect2 → false

Mais on peut utiliser une méthode `equals`, ici contenue dans la classe Rectangle

rect1.equals(rect2) → true

rect1.equals(rect3) → false

Comparaison d'objets

- Deux objets distincts ne sont pas égaux par défaut au sens de `equals`, et ils ont des codes de hachage différents.
- Il faut redéfinir `hashCode` et `equals` si on veut une notion d'égalité différente.
- Si une classe propose une notion d'égalité dans laquelle deux objets distincts peuvent être égaux, ces deux objets devraient retourner le même code de hachage. `String` par exemple retourne true si deux chaînes ont le même contenu, et elle redéfinit `hashCode` (car le mécanisme de hachage utilise `equals`).

Duplication d'objets

On peut faire une copie de surface

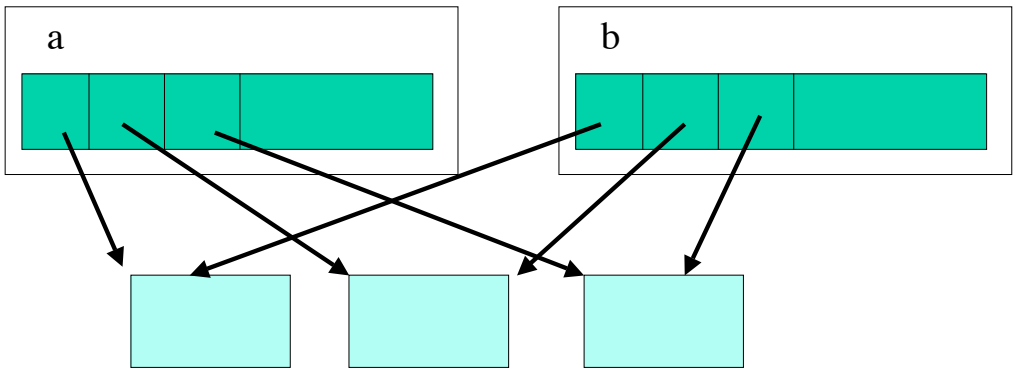
```
class ListeRectangle {
        Rectangle rectangles [ ] ;
        int nbRect;
        …
}

ListeRectangle  a, b;

//  copie de surface
        for (int i = 0; i < nbRect ; i++)
                a.rectangles [i] = b.rectangles [i] ;
```

Duplication d'objets

- Copie de surface



Duplication d'objets

… ou une copie profonde, avec `clone ()`

```
class ListeRectangle {
        Rectangle rectangles [ ] ;
        int nbRect;
        …
}

ListeRectangle  a, b;

//  copie profonde
        for (int i = 0; i < nbRect ; i++)
                a.rectangles [i] = b.rectangles [i].clone ( ) ;
```

Duplication d'objets

Copie profonde