PROGRAMMATION PAR OBJETS

Java

Environnement et constructions spécifiques

Initialisation d'objets : Constructeurs

- il faut créer les instances selon le modèle de la classe pour concrétiser les entités permettant la résolution du problème

  utilisation de constructeurs

- Chaque appel à un constructeur crée un nouvel objet (instance) qui obéit au modèle défini par la classe

Constructeurs

- Premier traitement exécuté par l'instance
- Permet d'automatiser le mécanisme d'initialisation d'un objet :
  - Initialisation des variables d'instance
  - N'importe quelles actions utiles au bon fonctionnement de l'objet

Construction en Java :

new + nom de la classe (+ param)

- Exemple :

```
new Livre()
new Livre("JRR Tolkien", "Le Seigneur des Anneaux", 1954)
```


Constructeurs

- `<Classe>()` est appelé « constructeur par défaut »
- Initialisation des variables d'instance :
  - aux valeurs déclarées, si elles existent
  - par défaut sinon (et en standard) :
    - valeur d'init pour les types primitifs (cf. tableau des types)
    - `null` pour les types d'objets
- Il est possible de le redéfinir, de le surcharger en le paramétrant :

```
new <Classe>([<parametres>])
```

- Exemple :

```
class Livre{
 Livre(String aut, String tit) {
      auteur = aut;
      titre = tit;
 }
}
```

Quelques règles concernant les constructeurs

- Par essence un constructeur ne retourne aucune valeur.
  - Dans son en-tête, aucun type ne doit figurer devant son nom.
  - Même la présence de `void` est une erreur.
- Une classe peut ne disposer d'aucun constructeur
  - L'instanciation se fait comme s'il existait un constructeur par défaut
    - Sans argument
    - Ne faisant rien
  - Dès qu'une classe possède au moins un constructeur, le constructeur par défaut ne peut plus être appelé
    - `Livre()` ne peut plus être appelé
- Un constructeur peut appeler un autre constructeur de la même classe.
  - Utilisation de this() en première instruction

Construction d'objets composites

- Construction d'objets par composition d'autres objets

```
class Rectangle {
 Point origin, corner;
}
```

- Initialisation par défaut : null, d'où :

```
class Rectangle {
 Rectangle(Point p1, Point p2) {
      origin=p1;
      corner=p2;
 }
 Rectangle(double x1, double y1, double x2, double
 y2) {
      origin = new Point(x1,y1);
      corner = new Point(x2,y2);
 }
}
```


Destructeur ?

- But du destructeur : libérer l'espace mémoire occupé par l'objet
- En Java : pas de destructeur (contrairement à C++) :
  - automatique par garbage-collector (objets non-référencés).
  - Il existe cependant un protocole de "finalisation" utilisable dans des cas particuliers (libération de ressources systèmes...) : méthode `finalize()`.

Déclaration

- Il est possible de nommer un objet créé pour pouvoir y faire référence par la suite.
  - on précise le type (classe) de la référence (donc de l'objet référencé)
  - on nomme la référence
  - on affecte une valeur (existante ou résultante d'une construction) = l'objet
- Exemple :

```
Auteur unAuteur = new Auteur();
Livre unLivre = new Livre("JRR Tolkien", "Le
Seigneur des Anneaux", 1954);
```

- Remarques :
  - conventions d'écriture (majuscules/minuscules)
  - instructions se terminent par un ";"

Référence

Identificateur d'objet = référence
Référence = un pointeur vers l'identité de l'objet

```
String chaine; // "chaine" référence "null"
chaine = new String("Le Seigneur des Anneaux");
Livre id1Livre = new Livre();
Livre id2Livre = id1Livre;
```

Important :
La référence permet d'accéder à l'objet mais n'est pas l'objet lui-même.
Une variable référence d'objet contient l'information sur comment trouver l'objet.

Référence

- Déclaration : `String chaine`
  - ↳ réservation d'un nom pour potentiellement un futur objet `chaine`
- Création : `new String("Le Seigneur des Anneaux")`
  - ↳ création de l'objet grâce à un constructeur
  - Le système (la JVM) possède un moyen de repérer l'objet.
- Liaison :

```
String chaine = new String("Le Seigneur des
Anneaux");
```

moyen de retrouver que l'objet est stocké dans `chaine`

Identificateur

- on fait référence à un objet en utilisant son identificateur
- tout identificateur doit être initialisé avant d'être utilisé
  - ↳ il faut lier l'identificateur à une référence.
- un identificateur non initialisé à la valeur null
- un identificateur correspond à un seul objet
- deux identificateurs peuvent faire référence au même objet
- Exemple :

```
Livre leLivre = new Livre("Tolkien", "Le Seigneur des
Anneaux", 1954);
leLivre.affiche();
Livre unLivre;
unLivre = new Livre("Frank Herbert","Dune",1965);
leLivre.getAuteur();
unLivre.getAuteur();
new Livre(...).getAuteur();
```


La notation "."

- si l'on possède une référence sur un objet, on peut envoyer un message à cet objet

notation "." (->)

`objet.message`

- `objet.attribut` envoi du message "accès à l'attribut `attribut`" à `objet`
- `objet.methode([params*])` envoi à `objet` le message "exécute la méthode `methode` avec les paramètres `params`"
  - ↳ le traitement décrit dans le corps de la méthode est exécuté.

La notation "."

- `Livre unLivre = new Livre(...)`
- `new TV().on()`
- `unLivre.auteur = "Tolkien"`
- `unLivre.imprime()`
- les “cascades” sont possibles :

`unLivre.auteur`.nom
un objet `Auteur`
`unLivre.auteur`.fixeDeces()

Attention à ne pas abuser des cascades, mais à déléguer au maximum (loi de Demeter)

- `Livre unLivre;`
- `unLivre.auteur` ➡ erreur : référence non initialisée

Envoi de message

- La dynamique d’un programme (le “traitement”) se fait par une succession de constructions d’objets et d’envois de messages à ces objets.
- Un envoi de message se fait toujours sur un objet (ou exceptionnellement une classe).
  - Jamais dans le vide
- Toujours se poser les questions :
  - Quel est l’objet à qui ce message est envoyé ?
  - Ai-je le droit de lui envoyer ce message ?
    - ↳ sa définition (classe) accepte t-elle ce message ?

Cas de l’autoréférence

- Dans le traitement de l’une de ses méthodes un objet peut avoir à s’envoyer un message (pour accéder à un de ses attributs ou invoquer une des ses méthodes)
  - utilisation de l’auto-référence, en Java : this
- this ne peut être utilisé que dans une méthode
  - (n’a aucun sens ailleurs)
- exemple : on se place dans le corps d’une méthode de la classe Livre
  - ↳ lors du traitement, l’objet invoquant est une instance de Livre

`this.imprime()` signifie “envoyer à `this` (= moi-même) le message `imprime()`”

- Si pas d’ambigüité, `this` peut être omis :

`imprime()` ≡ `this.imprime()`
`auteur` ≡ `this.auteur`

Egalité entre objets

- `String ch1 = new String("Le Seigneur des Anneaux");`
- `String ch2 = new String("Le Seigneur des Anneaux");`
- 2 références différentes sur 2 objets différents
- `ch1 == ch2` ➡ `false`
- pour comparer les états des instances on utilise la méthode equals
- `ch1.equals(ch2)` ➡ `true`
- La méthode equals, doit être définie et adaptée pour chaque classe.
  - Par défaut, elle se comporte comme == !

- `String ch3 = ch1;`
  - ↳ range dans `ch3` l’information stockée dans `ch1`
    - `ch1 == ch3` ➡ `true`
    - `ch1.equals(ch3)` ➡ `true`

Classes vs types primitifs

- 2 catégories de variables :
  - de type d'objets (classes et interfaces) : contiennent des références
  - de type primitif : contiennent des valeurs

- Déclaration des variables primitives :
  - Pas de `new` (pas d'objet)

```
int i;
boolean fini = true;
```

- Egalité de variables primitives
- Type primitif : variable contient valeur donc

```
int i = 5;
int j = 5;
i == j ➡ true
```

- == ne regarde que le contenu des variables

Types primitifs

- C (C++) + `boolean` et `byte`
- de taille constante quelque-soit la machine
- gérés par valeur, **ce ne sont pas des objets**, mais «enrobables» par les Wrapper classes
- les `boolean` ne sont pas des entiers
- les règles de compatibilité de l'affectation se fondent sur une hiérarchie de types.
- les `char` sont codés sur deux octets Unicode compatible ASCII. les caractères spéciaux sont notés (comme en C)
  `\n \t \b \r \f \\ \' \“`

Types primitifs

Type	***Valeurs***	***Init***	***Taille***	***Wrapper class***
boolean	true false	false	1 bit	Boolean
char	Unicode	'\u0000'	16 bits	Character
byte	entier signé	0	8 bits	Byte
short	entier signé	0	16 bits	Short
int	entier signé	0	32 bits	Integer
long	entier signé	0L|0l	64 bits	Long
float	IEEE 754 (0.5E-3)	0.0F|0.0f	32 bits	Float
double	IEEE 754	0.0D|0.0d	64 bits	Double

Wrapper Classes

- « Pont » entre valeurs primitives et objets
- Permet de considérer une valeur de type primitif comme un objet quand cela est requis (cf. collections d'objets)
- Dispose d'un constructeur recevant un argument d'un type primitif

```
Integer nObject = new Integer(2); // wrapping
```

- Dispose d'une méthode xxxValue (xxx représentant le nom du type primitif) qui permet de retrouver la valeur dans le type primitif

```
int n = nObject.intValue();// unwrapping
```

Wrapper Classes

- Offre des utilitaires (`static`) comme le parsing `String ->` valeur inverse de `String.valueOf(…)` (cf. entrée standard avant 5.0, paramètres du `main`, saisies de champs texte dans les interfaces)

```
public class Plus {
    public static void main(String[] args) {
        double x = Double.parseDouble(args[0]);
        double y = Double.parseDouble(args[1]);
        System.out.printf("x+y=%.2f\n", x+y);
    }
}
```


Tableaux

- Les tableaux s'apparentent à des objets :
  - créés dynamiquement (avec leur length) par instanciation : `new <type des elements>[<length>]`
  - libérés automatiquement (gc=garbage collector)
  - manipulés par référence : variables tableaux et passage en paramètre
  - compatibles avec le type Object dont les méthodes sont applicables.
- Mais beaucoup de syntaxe et de manipulation spécifique: création avec initialisation par : {} , accès par : [], …
- Type des éléments : types primitifs (homogènes) ou classes (tableaux polymorphes).
- Tableaux multidimensionnels = vrais tableaux de tableaux

Tableaux

- Exemples

On peut utiliser indifféremment `int[] t1` ou `int t1[]`

```
int[] t1= new int[10];

// declaration avec initialisation:
int[] t2= {1,2,3,4,5};

// affectation de variables tableaux
t1=t2; // t1 et t2 sont des variables

int[][] matrice = new int[50][100];

//int[][] matrice = new int[][100]; impossible!
```


Tableaux

```
public class test {
  static void uns(int tab[]) { // passage en parametre
    for (int i=0;i<tab.length;i++) tab[i]=1;
  }
  public static void main(String args[]) {
    int tabtab[][]=new int[3][]; //tableau de 3 tableaux d'int
    tabtab[0]=new int[10]; // de tailles quelconques...
    tabtab[1]=new int[20];
    tabtab[2]=new int[30];
    uns(tabtab[0]); uns(tabtab[1]); uns(tabtab[2]);
    for (int i=0;i<tabtab.length;i++) {
     for (int j=0;j<tabtab[i].length;j++)
        System.out.print(tabtab[i][j]);
     System.out.print("\n");
    }
  }
}

//for en 5.0
  for (int[] ligne : tabtab) { // ligne : variable tableau
   for (int x : ligne)
      System.out.print(x);
```

Chaînes de caractères

- **Ce sont des objets à part entière**
  - instances de la classe `String`
  - mais admettent une forme littérale :

```
String s = "deux\nlignes";
```

- **Deux classes principales**
  - **String =** objets chaînes de taille constante
  - **StringBuffer** = objets chaînes de taille variable

Chaînes de caractères

- **String** : quelques opérations
  - `opérateur + (String)`
  - `les méthodes valueOf(...)`
  - `int length()`
  - `int compareTo(String) (équivalent de strcmp)`
  - `boolean equals(Object)`
  - `char charAt(int) throws StringIndexOutOfBoundsException`
  - `String substring(int,int) throws StringIndexOutOfBoundsException`
- **StringBuffer : chaine modifiables, en contenu et en taille** :
  - `StringBuffer append(String)`
  - `StringBuffer insert(int,String) throws StringIndexOutOfBoundsException`
  - `void setCharAt(int, char) throws StringIndexOutOfBoundsException`

Syntaxe et éléments de base

- Commentaires

```
/* ceci est
un commentaire sur plusieurs lignes*/
// ceci est un commentaire ligne
```

- Identificateurs
  - variables, classes, méthodes, packages
  - syntaxe

```
identificateur = initiale suivant*
initiale = "a"|...|"z"|"A"..."Z"|"$"|"_"
suivant = initiale |"0"..."9"|unicode > 00C0
```

  - pas de limitation de longueur, tout caractère significatif (minuscules et majuscules)
  - conventions :
    - ne pas utiliser $ et _ (librairies C)
    - ceciEstUnIdentificateur
    - NomDeClasse
    - CONSTANTE

Mots réservés

```
abstract boolean break byte byvalue case cast
catch char class const continue default do
double else extends false final finally float
for future generic goto if implements import
inner instanceof int interface long native new
null operator outer package private protected
public return short static super switch
synchronized this thread throw throws
transient true try void volatile while
```

Expressions et structures de contrôle

- Pour l'essentiel, très semblables à C (C++).
- L'appel de fonction est remplacé par l'envoi de message:
  - c'est une instruction si la méthode est de type void
  - une expression sinon.
- **Opérateurs**
  - en moins: *, &, ->, sizeof (inutiles)
  - en plus: instanceof et + (concaténation de chaînes)
  - les opérateurs logiques procèdent sur le type boolean
  - mêmes règles de priorité et d'associativité
- **Structures de contrôle**

  `if/else while, do/while, switch, for, break`
  - les prédicats sont de type `boolean`
  - `for (int i=0;i<n;i++) // indice local a la boucle`
  - depuis Java 5.0 le `for` permet d'itérer sur toute séquence de valeurs « itérable », en particulier tableaux et collections

Applets

Applet : Programme Java, non autonome, destiné à être invoqué dans des documents HTML :
- sous un navigateur intégrant un interprète Java (JVM)
- ou un visualisateur d'applets (outil appletviewer du JDK)

```
//fichier Salut.java a compiler:
//javac Salut.java => Salut.class
import java.applet.*;
import java.awt.*;
public class Salut extends Applet {
  public void paint(Graphics g) {
        g.drawString("Salut!",20,20);
  }
}
```

.java

```
<HTML>
<!-- fichier salut.html sur la meme machine -->
<APPLET CODE="Salut.class" WIDTH=200 HEIGHT=50> </APPLET>
</HTML>
```

.html

Applications autonomes

- Une classe, dite principale et `public`, introduit une méthode "`main`" particulière qui détermine une application exécutable par la commande `java`

```
// fichier HelloWorld.java
public class HelloWorld {
  public static void main(String[] args) {
        System.out.println("Hello World!");
  }
}
```

.java

```
bash>javac HelloWorld.java
bash>java HelloWorld
```

`static` signifie que la méthode n'est associée à aucun objet particulier de la classe, mais à la classe elle même.

Protocole du main

- Un seul paramètre : tableau d'objets `String`
  - n'incluant pas le nom du programme,
  - sa taille (équivalent de `argc`) peut être obtenue comme pour tout tableau par son champ `length`

```
public class echo {
 public static void main(String[] argv) {
 for (int i=0; i<argv.length; i++)
  System.out.print(argv[i]+" ");
 System.out.print("\n");
 }
}
```

.java

```
bash>java echo bonjour le monde
bash>bonjour le monde
```

Applications autonomes

Exemple depuis java 5.0

```
public class echo {
 public static void main(String[] argv) {
  //sequence de valeurs : « for each »
  for (String chaine : argv)
  // sortie formattee a la C :
   System.out.printf("%s ", chaine);
 System.out.print("\n");
 }
}
```

La variable `chaine` prendra successivement les différentes valeurs du tableau `argv`.
Attention, structure adaptée pour des consultations de valeurs et non des modifications

Quelques règles

- Fichiers source
  - Un fichier peut contenir plusieurs classes,
    - `javac` générera autant de `.class`
  - Mais un fichier ne peut contenir qu'une classe `public` et doit porter son nom
  - **Règle : un fichier par classe (compilable séparément).**
  - Il faut une méthode main par application (sauf applet).
  - Les noms de classes commencent par une majuscule. Les noms de méthodes et d'attribut par des minuscules.

Entrées/sorties standards

- Les e/s (fichiers, "standards") sont définies par une hiérarchie de Streams (flux) dans le package java.io
- Les classes de la bibliothèque d'e/s Java sont divisées par entrée et sortie.

```
InputStream //lecture d'octets dont System.in
OutputStream
  FilterOutputStream
     PrintStream //sortie standard System.out
```

- Les flux chargés des e/s «standards » sont fournis dans 3 « variables statiques » de la classe System:

```
public class System {...
// variables de classe
 public static PrintStream err;
 public static InputStream in;
 public static PrintStream out;
...}
```


Sortie standard

- **System.out.print(..) et println(...)**
  - Ces méthodes sont *surchargées* pour chaque type de base (char, int, double, boolean...)
  - pour un objet, invoque sa méthode `toString()` qui doit fournir une `String` de représentation textuelle de l'objet
  - `toString()` est fournie par défaut dans `Object`. Il suffit de la redéfinir.
  - Exemple de la méthode `toString()`

```
class And {...
 public String toString() {
  return (
       "e1= "
     + String.valueOf(e1)
     + " e2= "
     + e2 // transformation automatique
     +  " s= "
     + s;
     );
 } ...}
```

Java 5.0 : sortie standard formatée

- **System.out.printf(String format, Object… args)**
  - printf « à la C »
  - format : %d, %f, %s, ...
  - Remarque :
    dans le cas d'objet utiliser %s => appel automatique à `toString()`
  - Exemple :

```
And a1 = new And(), a2 = new And();
// utilisation de a1 et a2
// affichage :
 System.out.printf("%d ands a1: %s a2: %s\n", 2, a1, a2);
```


Entrée standard (depuis Java 5.0)

- **System.in**
  Flux de bytes à "scanner" en l'enrobant ("wrapper") dans la classe `Scanner`

```
public class java.util.Scanner {
 public String next()
 public int nextInt()
 public double nextDouble()
 public String nextLine()
...}
```

- Exemple :

```
import java.util.*;
Scanner in = new Scanner(System.in);
System.out.printf("entrer 1 int, 1 double, une chaine, et le
  reste : \n");
int i = in.nextInt();
double d = in.nextDouble();
String s = in.next();
String reste = in.nextLine();
System.out.printf("i=%d\n d=%f\n s=%s\n reste=%s
  \n",i,d,s,reste);
```


Variables de classe : `static`

- Déclarées au niveau de la classe par le « modifier » `static`
- Permet de définir une ressource
  - attachée à la classe
  - en exemplaire unique
  - commune à toutes ses instances (accessibles directement)
  - et même "globale" puisque la classe l'est!
    Accessible en désignant la classe, comme c'est le cas de :
    `System.out, System.in, ...`

```
public class StaticExample {
   private static int compteur;
   public static double pi = 3.14159;
}
```


`final` : création de constantes

- La déclaration `final` rend la variable non modifiable et permet donc de déclarer des constantes

```
public class Circle {
  static final double PI = 3.14159265;
 // variables d'instance
  double rayon;

 // methodes d'instance
  double circonference() {
        return 2*PI*rayon; // ou 2*Circle.PI*rayon
        // final => constante => calculee statiquement
  }
}
```

- convention de nommage : les identifiants des constantes sont en majuscules et usage "_").

`Boolean.TRUE, Double.MAX_VALUE`

- NB : on peut utiliser final sans `static` et réciproquement

Méthodes de classe

```
public class System
  public static void exit(int status)
  public static Properties getProperties()

public class Math {
  public static double min(double a, double b)
  public static double sin(double a)
```

- Invocation : pas besoin d'instance !

```
Math.min(3, 0.7)
```

- NB : pas d'instance donc `this` n'a aucun sens dans le corps d'une méthode statique

Méthodes de classe

l'usage de `static` doit être limité et justifié

- a priori quasiment **jamais**
  - pas "objet", mais pratique...
  - plutôt, réservé pour les méthodes "utilitaires"
  - Intérêt : éviter la création d'objet "jetable".

    cf. dans `java.lang.Math`, `java.net.InetAddress.getLocalHost()`, ...

- cas particulier, la méthode `main`