Les bases de la programmation orientée objet avec Java

Etienne Duris

UNIVERSITE PARIS-EST MARNE-LA-VALLEE

Bibliographie et sources

- Les cours de Rémi Forax
  http://igm.univ-mlv.fr/~forax/
- *Le cours de Marie-Pierre Béal*
  http://igm.univ-mlv.fr/~beal/
- *Java et Internet*
  G. Roussel, E. Duris, N. Bedon et R. Forax. Vuibert 2002.
- Documentations Java Oracle
  http://docs.oracle.com/javase/
- The Java Language Specification, Third Edition:
  http://java.sun.com/docs/books/jls/
- The Java Virtual Machine Specification, Second Ed:
  http://java.sun.docs/books/jvms

Il existe différents styles de programmation

- Style **applicatif**
  - Fondé sur l'évaluation d'expressions qui ne dépendent que de la valeur des arguments, et non de l'état de la mémoire
    - On parle aussi de programmation fonctionnelle
    - Proche des notations mathématiques, utilise beaucoup la récursivité
    - Accepte des arguments, produit un résultat (pas d'« effet de bord »)
    - Ex: Lisp, Caml, ML, Haskel
- Style **impératif**
  - Fondé sur l'exécution d'instructions qui modifient l'état de la mémoire
    - Utilise beaucoup les itérations et autres structures de contrôle
    - Les structures de données sont fondamentales
    - Ex: Fortran, C, Pascal

Le style objet

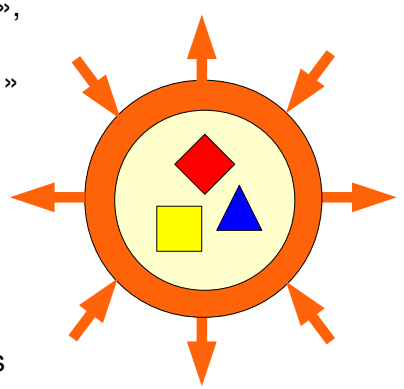
- C'est un style de programmation où l'on considère que des composants autonomes (les **objets**) disposent de ressources et de moyens d'interactions entre-eux.
- Ces objets représentent des données qui sont modélisées par des **classes** qui définissent des types
  - un peu comme `typedef struct` en C
- En plus de la manière dont sont structurés leurs objets, les classes définissent les actions qu'ils peuvent prendre en charge et la manière dont ces actions affectent leur état
  - ce sont des « messages » ou des « **méthodes** ».
- Java n'est pas le seul langage objet
  - Simula, Smalltalk, C++, OCaml...

Les avantages de la programmation objet

- Les caractéristiques de bases précédemment décrites peuvent être mises en oeuvre dans un style impératif, mais des fonctionnalités propres au style objet favorisent:
  - la **programmation modulaire**
  - l'**abstraction**
  - la **spécialisation**
- L'objectif est de produire du code
  - facile à développer, à maintenir, à faire évoluer,
  - réutilisable, tout ou en partie, sans avoir besoin de le dupliquer
  - générique, et dont les spécialisations sont transparentes

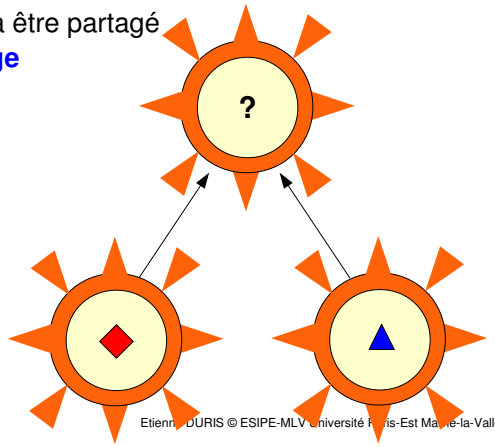
Programmation modulaire

- La conception par classes, représentant à la fois les données, les actions et les **responsabilités** des objets de cette classe, permet de bien **distinguer et séparer les concepts**
- Le fait de définir des « **interfaces** », au sens « moyens et modalités de communication avec l'extérieur » permet de **cacher** les détails d'**implémentation** et d'éviter les dépendances trop fortes
- Tout ça favorise la réutilisabilité et la **composition / délégation**: l'assemblage des composants en respectant leurs responsabilités



L'abstraction et la spécialisation

- L'**abstraction** demande à **séparer la définition** (d'un type, d'une classe, d'une méthode) **de l'implémentation**
  - Permet d'identifier un modèle commun à plusieurs composants
  - Ce modèle commun pourra être partagé via le mécanisme d'**héritage**
- La **spécialisation** traite des cas particuliers, mais elle doit autant que possible rester transparente:
  - C'est possible grâce à la dérivation



Le langage Java:

- est né en 1995 chez Sun Microsystems
  - Version actuelle Java 8, actuellement Oracle
- est **orienté objet**
- est **fortement typé**
  - Toute variable doit être déclarée avec un type
  - Le compilateur vérifie que les utilisations des variables sont compatibles avec leur type (notamment via un sous-typage correct)
  - Les types sont d'une part fournis par le langage, mais également par la définition des classes
- est **compilé**
  - En bytecode, i.e., code intermédiaire indépendant de la machine
- est **interprété**
  - Le bytecode est interpété par une machine virtuelle Java

Premier exemple

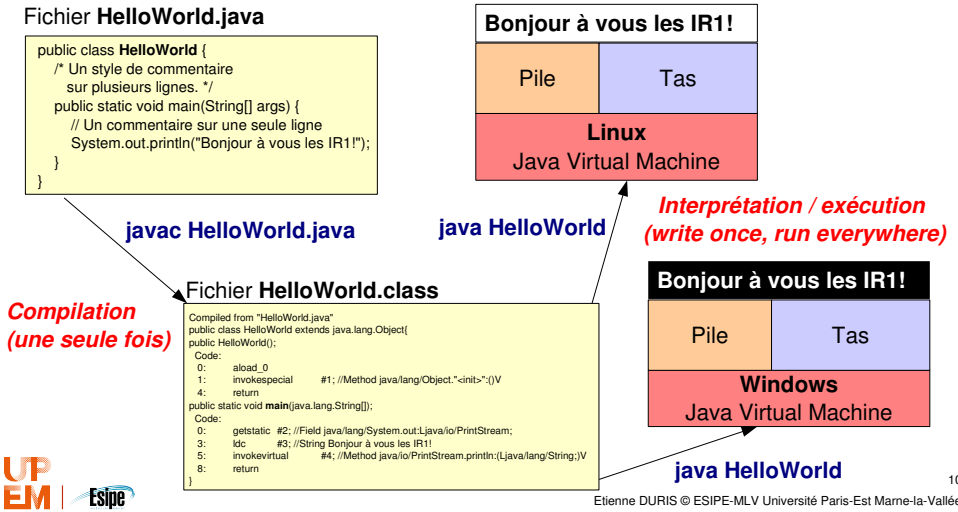
- Dans un fichier de nom `HelloWorld.java`
  - ***Règle: toute classe publique doit être dans un fichier qui a le même nom que la classe***
  - ***Règle: tout code doit être à l'intérieur d'une classe***

```
public class HelloWorld {
    /* Un style de commentaire
       sur plusieurs lignes. */
    public static void main(String[] args) {
      // Un commentaire sur une seule ligne
      System.out.println("Bonjour à vous les IR1!");
    }
}
```

  - Ça définit une classe, qui est une unité de compilation
  - Comme il y a une méthode `main`, cette classe est « *exécutable* »

Compilation, bytecode et JVM

- Compilation du langage source -> exécution du bytecode



Le bytecode

- Le ***langage source Java*** est défini par la JLS (*Java Language Specification*) éditée par Sun-Oracle
  - Dans sa syntaxe et sa sémantique
- Le code source d'une classe contenue dans un fichier est compilé avec la commande **javac**
  - Cela produit un code intermédiaire, appelé bytecode, qui est le « langage machine » de la machine virtuelle Java
- Le ***bytecode*** d'une classe est destiné à être ***chargé*** par une ***machine virtuelle*** qui doit l'exécuter avec la comande **java**
  - Soit par ***interprétation***, soit par ***compilation*** « juste à temps » (*just-in-time* ou JIT)
  - L'argument est le nom de la classe (sans l'extension .class)

La machine virtuelle (JVM)

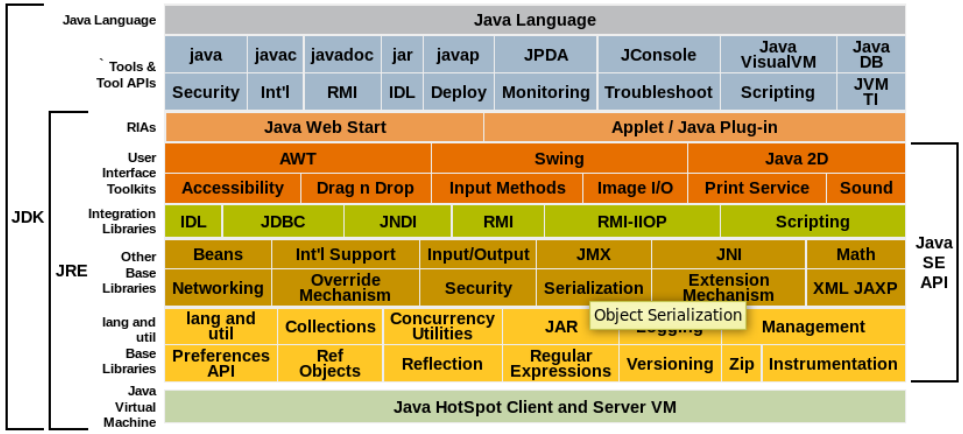
- Son rôle est d'**abstraire le comportement d'une machine**
  - Pour le rendre le + possible indépendant de la plateforme
  - Son comportement est défini par la JVM Spec édité par Sun-Oracle
- Une JVM est une implémentation de cette spec
  - Qui peut être adaptée à une plateforme d'accueil (Windows, Linux, Mac...)
  - Qui peut être développée par Sun (HotSpot: open source GPL depuis 2006) ou par d'autres: IBM, Jikes, etc.
- Une JVM **traduit** le **bytecode** dans le **langage machine** de la plateforme d'accueil

Java: un langage et une plateforme

- Dans la technologie Java, on a donc besoin
  - Du **langage de programmation** et du compilateur
    - Et plein de commandes bien utiles: jar, javap, javadoc, etc
  - De la **JVM et** des **APIs** (*Application Programming Interfaces*) regroupées dans une « plateforme »:
    - Java SE (Java Platform, Standard Edition): Java SE 6 pour applications classiques, desktop
    - Java EE (Java Platform, Enterprise Edition): Java EE 6 pour développer et déployer des applications serveur, Web services, etc.
    - Java ME (Java Platform, Micro Edition): J2ME pour les applications embarquées, PDA, téléphones, etc.
- Si on veut juste exécuter, il suffit du **JRE** (*Java Runtime Execution*) par opposition au **JDK** (*Java Developpement Kit*)

Java SE 7 Plateform at a Glance

(http://docs.oracle.com/javase/7/docs/)



Le langage Java

- Les variables, les opérateurs, les expressions, instructions, blocs, contrôle de flot sont très proches de ceux du C
  - Les exceptions sont une nouveauté
  - Les types primitifs ont une taille et une représentation normée
- S'y ajoutent des spécificités syntaxiques liées à la programmation objet, aux classes, à l'héritage...
- Un **style de nommage** (très fortement) conseillé
  - Style « chameau » (***CamelCase***) pour les indentificateurs
  - Première majuscule pour les classes (class HelloWorld)
  - Première minuscule pour les variables/champs et les fonctions/méthodes (radius, getRadius())
  - Tout en majuscule pour les constantes (MAX_SIZE)

Classes et objets

- Une classe Toto représente plusieurs choses:
  - Une **unité de compilation**
    - La compilation d'un programme qui contient une classe Toto produira un fichier Toto.class
  - La définition du **type** Toto
    - Il peut servir à déclarer des variables comme Toto t;
  - Un **moule pour la création d'objets** de type Toto
    - Cela nécessite en général la définition d'un ensemble de **champs** (***fields***) décrivant **l'état** d'un objet de ce type et d'un ensemble de ***méthodes*** définissant son **comportement** ou ses focntionnalités
  - Chaque objet de la classe Toto
    - Dispose de son **propre état** (la valeur de ses champs)
    - Répond au **même comportement** (via les méthodes de la classe)

Structure d'une classe

- Une classe est définie par son nom et son package d'appartenance (ex: java.lang.String)
  - En l'absence de directive, les classes sont dans un package dit « par défaut » (i.e., pas de packge).
- Une classe peut contenir trois sortes de **membres**
  - Des **champs** (fields) ou attributs
  - Des **méthodes** (methods) et **constructeurs**
  - Des ***classes internes***
- Les membres statiques (static) sont dits **membres de classe**
  - Ils sont définis sur la classe et non sur les objets
- Les **membres** non statiques (ou **d'instance**) ne peuvent exister sans un objet

Exemple

```
public class Pixel {
    public final static int ORIGIN = 0;
    private int x;
    private int y;
    public Pixel(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public void reset() {
        x = ORIGIN;
        y = ORIGIN;
    }
    public void printOnScreen() {
        System.out.println("("+x+","+y+")");
    }
    public static boolean same(Pixel p1, Pixel p2) {
        return (p1.x==p2.x) && (p1.y==p2.y);
    }
    public static void main(String[] args) {
        Pixel p0 = new Pixel(0,0);
        Pixel p1 = new Pixel(1,3);
        p1.printOnScreen();              // (1,3)
        System.out.println(same(p0,p1)); // false
        p1.reset();
        System.out.println(same(p0,p1)); // true
    }
}
```

Constante

Champs

Constructeur

Méthodes d'instances

Méthode de classe

Variables locales à la méthode main et objets de la classe Pixel

Les objets sont manipulés via des références (sortes d'adresses mémoire)

Lorsque la méthode

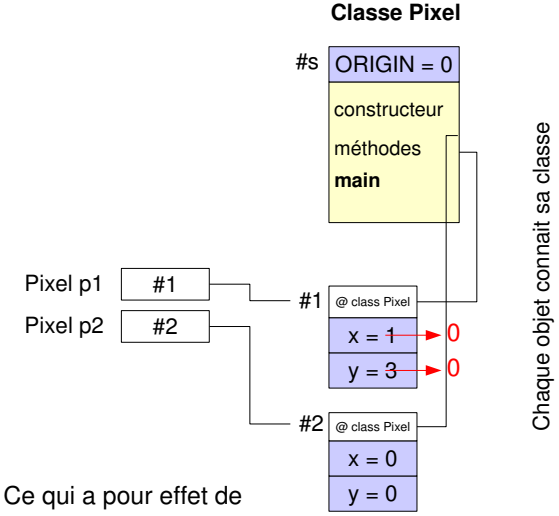
```
p1.reset();
```

est appelée, le code de la méthode défini dans la classe comme

```
public void reset() {
    x = ORIGIN;
    y = ORIGIN;
}
```

est exécuté sur la pile avec les références #1 à la place de p1 et #s à la place de ORIGIN

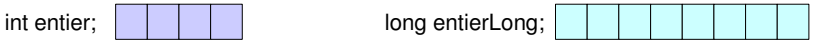
```
public void reset() {
    #1.x = #s;
    #1.y = #s;
}
```



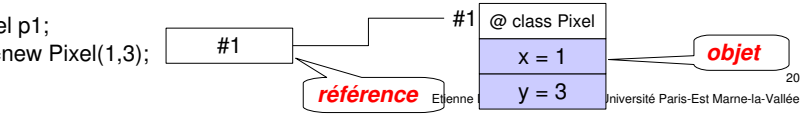
Ce qui a pour effet de mettre p1.x et p1.y à 0

La nature des variables en Java

- Les **variables locales** comme les **champs** des classes et des objets ne peuvent être que de deux natures
- De **type « primitif »**
  - Dans ce cas, la déclaration de la variable réserve la place mémoire pour stocker sa valeur (qui dépend de son type)



- De **type « objet », ou référence**
  - Dans ce cas, la déclaration de la variable ne fait que réserver la place d'une **référence** (une sorte de pointeur) qui permettra d'accéder à l'endroit en mémoire où est effectivement stocké l'objet en lui-même (vaut **null** si référence inconnue)



Les types primitifs

- **Types entiers signés** (représentation en complément à 2)
  - **byte** (octet) sur 8 bits: [-128 .. 127]
  - **short** sur 16 bits [-32768 .. 32767]
  - **int** sur 32 bits [-2147483648 .. 2147483647] (défaut pour entiers)
  - **long** sur 64 bits [-9223372036854775808 .. 9223372036854775807]
- **Type caractère non signé** (unités de code UTF-16)
  - **char** sur 16 bits ['\u0000' .. '\uffff']
- Types à **virgule flottante** (représentation IEEE 754)
  - **float** sur 32 bits
  - **double** sur 64 bits (défaut pour flottants)
- Type **booléen:** **boolean** (**true** ou **false**)

Promotion entière et flottants spéciaux

- Pour la plupart des opérations, les valeurs entières sont transformés en des « int » (promotion entière):
  - short s = 4;
    s = s+s; // Type mismatch: cannot convert int to short
    s = (short) (s+s); // cast nécessaire
- Les débordements ou cas d'erreurs sont prévus pour les flottants (Infinity et Not-a-Number):
  - double d = 1e308;
    System.out.println(d*10); // affiche: **Infinity**
  - d = 0.0/0.0;
    System.out.println(d); // affiche: **NaN**

Attention aux nombres à virgule flottante

- Ils ne sont que des **approximation** des valeurs
- Leur égalité au sens de l'opérateur **== n'a aucun sens**

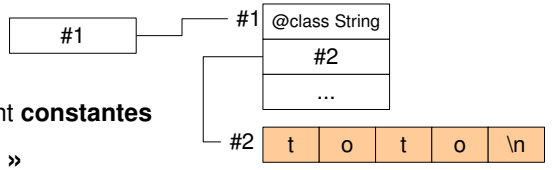
```
double d = 0.0; int nb = 0;
double expected = 1.0;
while (d != 1.0 /* && nb < 10 */) {  // boucle infinie!!!
    d += 0.1;  nb++;
}
System.out.println("nb: " + nb + " d: " + d);
// Si on décommente, affiche nb: 10 d: 0.9999999999999999
```

- Il faut tester leur « proximité » modulo un epsilon donné

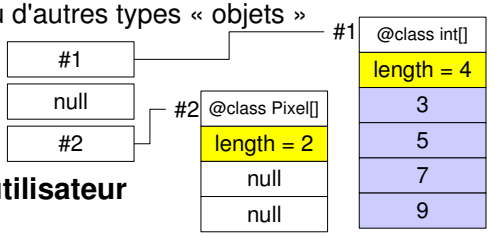
```
static final double EPSILON = 0.00001;
...
double d = 0.0;
double expected = 1.0;
while ( Math.abs(expected – d) > EPSILON )
    d += 0.1;
System.out.println(d);       // 0.9999999999999999
```


Tous les autres types sont « objets » et sont manipulés via des références

- soit des **types définis dans les APIs**
  - java.lang.Object, java.lang.String,java.util.Scanner, etc.
    - String chaine = "toto";
    - Différent du C!
    - En plus, les String sont **constantes**



- soit des **types « cachés »**
  - Tableau de types primitifs ou d'autres types « objets »
    - int[] tab = {3, 5, 7, 9};
    - String[] args;
    - Pixel[] array = new Pixel[2];



- soit des **types définis par l'utilisateur**
  - Pixel p = new Pixel(0,0);

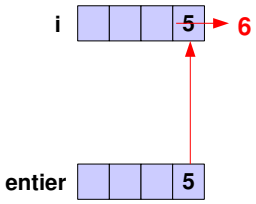
Types Java et passage de paramètre

- Lors des appels de méthode, les **arguments** sont toujours **passés par valeur**
- Dans le cas des **types primitifs**, c'est la **valeur** de l'argument qui est recopiée dans le paramètre de la méthode
  - **Les modifications sur le paramètre de la méthode sont sans effet sur l'argument**
- Dans le cas des **types « objet »**, c'est la valeur de la variable, (la **référence** à l'objet) qui est transmise à la méthode
  - Les **modifications effectuées en suivant cette référence** (e.g. modification d'un champ de l'objet) sont **répercutés dans la mémoire** et sont donc visibles sur l'argument
  - En revanche, **la modification de la référence elle-même est sans effet sur l'argument** (c'en est une copie)

Passage de paramètre: type primitif

- Dans le cas des **types primitifs**, c'est la **valeur** de l'argument qui est recopiée dans le paramètre de la méthode
  - **Les modifications sur le paramètre de la méthode sont sans effet sur l'argument**

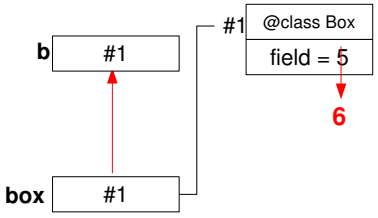
```
public static void m1(int i) {
    i++;
}

public static void main(String[] args) {
    int entier = 5;
    m1(entier);
    System.out.println(entier); // 5
}
```



Passage de paramètre: type référence

- Dans le cas des **types « objet »**, c'est la valeur de la variable, (la **référence** à l'objet) qui est transmise à la méthode
  - Les **modifications effectuées en suivant cette référence** (e.g. modification d'un champ de l'objet) sont **répercutés dans la mémoire** et sont donc visibles sur l'argument

```
public static void m2(Box b) {
    b.field++;
}

public static void main(String[] args) {
    Box box = new Box();
    box.field = 5;
    m2(box);
    System.out.println(box.field); // 6
}
```

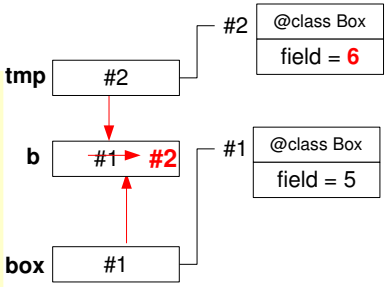



```
class Box {
    int field;
}
```


Passage de paramètre: type référence

- Dans le cas des **types « objet »**, c'est la valeur de la variable, (la **référence** à l'objet) qui est transmise à la méthode
  - En revanche, **la modification de la référence elle-même est sans effet sur l'argument** (c'en est une copie)

```
public static void m3(Box b) {
    Box tmp = new Box();
    tmp.field = b.field+1;
    b = tmp;
}
public static void main(String[] args) {
    Box box = new Box();
    box.field = 5;
    m3(box);
    System.out.println(box.field); // 5
}
```




```
class Box {
    int field;
}
```

Type référence et valeur null

- Lorsqu'on déclare une variable de type objet, seule la place de la référence est réservée sur la pile d'exécution (registre)
  - 4 octets, 32 bits, et ce quelque soit le type de l'objet référencé
  - Par défaut, cette référence vaut une valeur particulière, **null**.
  - Il est interdit de tenter d'y accéder, de la déréférencer
- Le compilateur vérifie ce qu'il peut

```
public static void main(String[] args) {
    Box b;
    System.out.println(b.field); // Variable b might not have been initialized
}
```

- On peut « forcer » pour que ça compile => lève une **exception**

```
public static void main(String[] args) {
    Box b = null;
    System.out.println(b.field); // lève NullPointerException
```


Allocation mémoire

- Pour qu'une variable objet prenne une autre valeur que null, il faut être capable d'y affecter une référence
  - Elle peut être produite (retournée) par l'opérateur d'allocation **new**
  - Cet opérateur à besoin de connaître la taille de l'objet qu'il doit réserver en mémoire
    - Le nom du type / de la classe suit immédiatement l'opérateur
      - `new Box();  // j'ai besoin de stocker 1 int (field)`
      - `new int[10]; // stocker 10 int + la taille du tableau`
    - La zone mémoire allouée doit être initialisée (affectation des valeurs)
      - `new Pixel(1,3); // utilise un « constructeur »`
      - Le terme de **constructeur** est mal choisi: **initialiseur** serait mieux
  - Ce que retourne l'opérateur new est **la référence** qui permet d'accéder à l'objet alloué en mémoire

Désallocation mémoire

- Elle n'est pas gérée par le programmeur, mais par un **GC (Garbage Collector)**
- Les objets qui ne sont plus référencés peuvent être « récupérés » par le GC, pour « recycler » l'espace mémoire qu'ils occupent
- Un même objet peut être référencé par plusieurs variables
- Il faut qu'aucune variable ne référence plus un objet pour qu'il soit réclamé
- Les variables cessent de référencer un objet
  - Quand on leur affecte une autre valeur, ou null
  - Quand on quitte le bloc où elles ont été définies: elles meurent, disparaissent... (sur la pile)

Références et Garbage Collector

- La quantitié de mémoire disponible dans le tas de la VM est fixé à l'avance (paramétrable):
  - `java -Xms<size> MyAppli`
- C'est le gestionnaire de la mémoire qui se charge de
  - L'allocation des nouveaux objets
    - Demandée par l'opérateur new
  - La récupération de la place occupée par les objets morts (devenus inaccessibles)
    - Lorsqu'il y a besoin de place supplémentaire ou quand il le décide
  - De la manière d'organiser les objets
    - Pour éviter une « fragmentation « de la mémoire, il « déplace » les objets en mémoire (zone des « vieux » des « récents », etc.)
  - Les références ne sont pas des adresses (indirection)

Regardons les classes de plus près

- « Avant » la classe, il peut y avoir 2 informations
  - Le paquetage d'appartenance de la classe:
    - `package fr.umlv.ir1.basics;`
    - Il permet de définir un « espace de nommage » qui donne un *nom complet* à la classe: `fr.umlv.ir1.basics.MyClass`
    - L'organisation des classes dans les répertoires du système de fichier doivent refléter l'organisation des paquetages: le répertoire `fr/umlv/ir1/basics` contient la classe `MyClass.java`
  - Des directives d'importation (là où aller chercher les classes dont on se sert dans le code); ce n'est pas une directive d'inclusion!
    - `import java.io.*;`
      `import java.util.Scanner;`
    - Les classes du paquetage `java.lang` sont visibles par défaut

Quelques règles d'hygiène

- (Très vite) indispensable: **regrouper les classes en paquetages**
- Obligatoire: avoir une hiérarchie de répertoire isomorphe à la hiérarchie des paquetages
- Séparer les **sources** des **classes**
-
```
[code$] ls
 classes  src
[code$] ls src/fr/umlv/ir1/basics/
 HelloWorld.java  Pixel.java
[code$] javac -d classes src/fr/umlv/ir1/basics/Pixel.java
[code$] ls classes/fr/umlv/ir1/basics/
 Pixel.class
[code$] java -cp classes/ fr.umlv.ir1.basics.Pixel
 (1,3) false true
[code$] javac -cp classes -d classes src/fr/umlv/ir1/basics/HelloWorld.java
[code$] java -cp classes/ fr.umlv.ir1.basics.HelloWorld
 Bonjour à vous les IR1!
 (1,3) false true
```

-d : répertoire de destination des classes

-cp : répertoire où trouver les classes compilées

Les deux si la compilation a besoin d'autres classes

La classe en elle-même: accessibilité

- `class MyClass`
  - La classe sera **accessible** depuis toutes les classes du même paquetage qu'elle (on parle quelque fois de ***visibilité de paquetage***)
- `public class MyClass`
  - La classe sera **accessible** de n'importe où (pourvu qu'on indique son nom de paquetage complet ou qu'on utilise la directive `import`)
- Cette notion d'accessibilité sert à définir des « composants » de plus grande granularité que les classes,
  - Permet de **limiter l'accès**
  - Peut éviter des **conflits de noms**

Les membres: champs et méthodes

- Dans une classe, il y a grossièrement
  - Une zone avec des déclarations de champs
    - Ils définissent ce qui sera stocké dans chaque objet de cette classe
  - Une zone avec des déclarations de méthodes
    - Elles définissent les actions/fonctionnalités/comportements acceptés par les objets de cette classe
    - Chaque méthode à deux parties:
      - Sa **signature** (type de retour, nom, types des paramètres
      - Son code, qui est constitué de **blocs** imbriqués
  - Il peut également y avoir aussi quelques blocs d'intitalisation, constructeurs ou autres classes « internes »...

Les blocs et les variables locales

- Une variable locale est visible dans le bloc dans lequel elle est déclarée
- Les paramètres des méthodes sont considérés comme des variables locales
- 2 variables de même nom doivent être dans des blocs disjoints

```
public class LocalVariable {
   private static double sum(double[] values) {
      double sum=0.0;
      for(double v:values) {
         sum+=v;
      }        // v n'est plus accessible
      return sum;
   }       // values et sum pas accessible
   public static void main(String[] args) {
      sum(new double[]{2,3,4,5});
   }
}
```


Variable locale constante

- Le mot-clé **`final`** signifie en Java « **affectation unique** »
- Il s'applique sur la variable qui est déclarée
- Un objet ou un tableau « final » peut avoir ses champs ou ses éléments modifiés

```
public class FinalExample {
  public static void main(String[] args) {
    final int val;
    if (args.length==2)
      val=3;
    else
      val=4;
    val = 0; // error: variable val might already have been assigned

    final String[] tab = args;
    for(int i=0;i<args.length;i++)
      tab[i]="toto"; // ok, car cela ne change pas la référence
  }
}
```


Les champs, par rapport aux variables?

- Leur existence et leur durée de vie sont **associées aux objets** (ou au pire à la classe elle-même)
  - Tandis que les variables locales sont associées à une exécution de la méthode qui les utilise... sur la pile!
- Les champs possèdent une **valeur par défaut** qui leur est affectée lors de la création d'un objet
  - **`0`** pour les types numériques primitifs
  - **`false`** pour les booléens
  - **`null`** pour les types référence
  - Tandis que les variables locales doivent nécessairement être initialisées avant d'être utilisées

Accessibilité des membres

- Tous les membres ont une **accessibilité** qui est spécifiée à la déclaration par un « modificateur »
- De manière **complémentaire à celui de la classe**, il permet de déterminer qui, parmi ceux qui ont accès à la classe A, ont accès à ce membre
  - **`private`** : accessible uniquement depuis l'intérieur de la classe A
  - ***`Par défaut`*** (pas de modificateur) : accessible depuis toutes les classes qui sont dans le même paquetage que A
  - **`protected`** : accessible depuis toutes les classes qui sont dans le même paquetage que A, et également depuis celles qui ne sont pas dans le même paquetage mais qui héritent de la classe A
  - **`public`** : accessible depuis n'importe où

Accès aux membres: champs et méthodes

- Avec le point « . » sur une référence à un objet:
  - `p.x, p0.sameAs(p1);`
- Le compilateur regarde le type déclaré de la variable (ici `p` ou `p0`) et vérifie que le membre (ici `x` ou `sameAs`) existe.
- Le compilateur vérifie également les droits d'accessibilité
- Un champ et une méthode peuvent avoir le même nom
- `this` représente l'instance courante

```
public class Pixel {
  private int x;
  private int y;
  public Pixel(int x, int y) {
    this.x = x;
    this.y = y;
  }
  public boolean sameAs(Pixel p) {
    return (this.x==p.x) && (this.y==p.y);
  }
  public static void main(String[] args) {
    Pixel p0 = new Pixel(0,0);
    Pixel p1 = new Pixel(1,3);
    boolean b = p0.sameAs(p1); // false
  }
}

class OtherClass {
 public static void main(String[] args) {
    Pixel p0 = new Pixel(0,0);
    p.sameAs(p);  // true
    p0.x = 1; // error: x has private
  }           // access in Pixel
}
```


Création d'instances

- Une instance, ou un objet, d'une classe est créée en 3 temps

  `Pixel p1 = new Pixel(1,3);`

  - `new` est l'opérateur d'instanciation: comme il est suivi du nom de la classe, il sait quelle classe doit être instanciée
    - Il initialise chaque champ à sa valeur par défaut
    - Il peut exécuter un bloc d'initialisation éventuel
    - Il retournera la référence de l'objet ainsi créé
  - `Pixel` est le nom de la classe
  - `(1,3)` permet de trouver une fonction d'initialisation particulière dans la classe, qu'on appelle un constructeur

Constructeur

- Un constructeur est une méthode particulière, qui sert à initialiser un objet une fois que la mémoire est réservée par new
  - Permet de garantir des invariants sur un objet sont conservés, par exemple pour initialiser un objet avec des valeurs particulières
    - Par exemple, on veut qu'un Pixel soit en (1,1)
  - Le faire avec une méthode « normale » ne garantirait pas qu'on accède pas à l'état de l'objet avant son initialisation
  - Le constructeur a le même nom que la classe et pas de type de retour
- **En l'absence de constructeur explicitement défini, le compilateur ajoute un constructeur public sans paramètre**

```
class Box {
  private int field;
  public static void main(String[] a){
    Box b = new Box();
  }
}
```


Appel à un autre constructeur

- Plusieurs constructeurs peuvent cohabiter dans la même classe
  - Ils ont typiquement des rôles différents et offrent des « services » complémentaires à l'utilisateur, par exemple:
    - `new Pixel(1,3)` crée un pixel avec les coordonnées explicites
    - `new Pixel()` crée un pixel avec les coordonnées à l'origine
    - `new Pixel(1)` crée un pixel sur la diagonale (x == y), etc.
- Quand c'est possible, il est préférable qu'il y en ait un « **le plus général** » et que **les autres y fassent appel**
  - Plutôt que de dupliquer le code dans plusieurs constrcuteurs
  - L'appel à un autre constructeur de la même classe se fait par `this(...)`

```
public class Pixel {
  private int x;
  private int y;
  public Pixel(int x, int y) {
    this.x = x;
    this.y = y;
  }
  public Pixel() {
    this(0,0);
  }
  public Pixel(int v) {
    this(v, v);
  }
}
```

Champ constant

- Comme pour les variables locales, il est possible de déclarer un champ avec le modificateur **final**.
  - Cela signifie qu'il doit avoir une **affectation unique**
  - Le compilateur vérifie que il a **bien été initialisé**, et ce **quelque soit le constructeur** mais également qu'il n'a été **affecté qu'une seule fois**

```
public class Pixel {
  private final int x;
  private int y;
  public Pixel(int x, int y) {
    this.x = x;
    this.y = y;
  }
  public Pixel() {
    // error:  final field x may not
    // have been initialized
  }
  public Pixel(int v) {
    this(v, v);
  }
  public static void main(String[] a){
    Pixel p = new Pixel(1);
    p.x = 0; // error: final field x
    // cannot been assigned
  }
}
```


Surcharge

- Les méthodes comme les constructeurs peuvent être **surchargées**: (***overloaded***)
  - Leur nom est le même
  - Le nombre ou le type de leurs paramètres varie
- Objectif: fournir plusieurs définitions pour la même méthode
- Le compilateur doit pouvoir trouver celui qui convient le mieux au « site d'appel », c'est à dire au nombre et au type des arguments
- Si aucune méthode ne correspond exactement, le compilateur peut prendre une méthode approchée en fonction du typage

Le typage et la surcharge

- Les types déclarés des variables pevent quelque fois être ordonnés par une relation de sous-typage
  - Il y a du **sous-typage** sur les types références
    - Lié aux classes, aux interfaces, aux classes abstraites, à l'héritage...
  - Sur les types primitifs, on parle de **conversion** implicite
    - `byte < short < int < long < float < double`
      `char < int`
- En règle générale, il est préférable d'utiliser la surcharge entre deux méthodes quand elles ont la même sémantique
  - `float sqrt(float value) {...}  // et`
    `double sqrt(double value) {...} // OK`
  - `void remove(Object value) {...} // et`
    `void remove(int index) {...}  // KO (liste)`

Ambiguité de surcharge

- Le compilateur peut ne pas savoir laquelle choisir

```
public class Overloading {
  public static void m(int i, long l) {
    System.out.println("m(int, long)");
  }
  public static void m(long l, int i) {
    System.out.println("m(long, int)");
  }
  public static void main(String[] args) {
    int i = 1;
    long l = 1L;
    m(i,l); // m(int, long)
    m(l,i); // m(long, int)
    m(i,i); // reference to m is ambiguous,
    // both method m(int,long) and method m(long,int)  match
  }
}
```

Les membres statiques

- Déclarés avec le mot-clé **`static`**, les membres statiques sont liés à la classe et non à une instance particulière (un objet)
  - **Champs**: sa valeur n'est pas propre à un objet mais à la classe (le champ lui-même, et bien sûr sa valeur, est la même pour tous les objets de la classe)
    - On y accède à partir du nom de la classe ou de la référence à n'importe quel objet de la classe
  - **Méthodes**: son code ne dépend pas d'un objet de la classe et peut être exécuté même si aucun objet existe (e.g. main)
  - Classes internes
- Tout code utilisé dans les membres statiques ne peut pas faire référence à l'instance courante (**`this`**)

Le mystère de println()...

```
public static void main(String[] args) {
   System.out.println("Hi les geeks!");// Hi les geeks!
                      void java.io.PrintStream.println(String x)
   System.out.println('c');            // c
                      void java.io.PrintStream.println(char x)
   System.out.println(2.5f);           // 2.5
                      void java.io.PrintStream.println(float x)
   Object o = new Object();
   System.out.println(o);              // java.lang.Object@7a9664a1
                      void java.io.PrintStream.println(Object x)
   int[] tab = new int[5];
   System.out.println(tab);            // [I@27a8c4e7
                      void java.io.PrintStream.println(Object x)
   Pixel p1 = new Pixel(1,3);
   System.out.println(p1);             // Pixel@20cf2c80
}                     void java.io.PrintStream.println(Object x)
```


Comment ça marche?

- Les méthodes println() sont **surchargées** pour être adaptées:
  - À chaque valeur de type primitif (boolean char double float int long)
  - Aux chaînes de caractères (String et char[])
  - Aux objets (Object)
- Par défaut, toute classe **`A`** « hérite » de **`Object`**
  - Cela induit une relation de **sous-typage** : **`A < Object`**
  - **`A`** définit un « sous-type » de **`Object`**
  - On peut donc appliquer sur **`A`** ce qu'on sait faire sur **`Object`**
- Quand on appelle println() sur un Pixel, ça appelle la méthode sur Object
  - Affichage **Pixel@20cf2c80** cohérent avec **java.lang.Object@7a9664a1**

Ce que fait la méthode println(Object x)

- Si on suit les commentaires de la javadoc (ou le code)
  - PrintStream.println(Object x) appelle String.valueOf(x)
  - String.valueOf(Object x) retourne
    - "null" si jamais l'objet x vaut null
    - x.toString() si x ne vaut pas null
  - Dans la classe Object, toString() affiche une chaîne construite par getClass().getName() + '@' + Integer.toHexString(hashCode())
  - Pour notre cas, si `Pixel p = new Pixel(1,3);`
    - Comme p est un sous-type de Object, l'appel à println(p) exécute la méthode println(Object x) sur l'objet p, ce qui appelle String.valueOf(p) qui lui-même retourne p.toString()
    - Comme il n'y a pas de méthode toString() explicitement définie dans Pixel, c'est la méthode toString() de Object « héritée » dans Pixel qui est appelée. CQFD.

Là où c'est fort...

- Quand une méthode m() est appelée sur une variable v qui contient une référence
  - Le *compilateur* vérifie qu'il y a bien une méthode m() définie pour le **type déclaré** de la variable v (« ***au pire, je sais faire*** »)
  - La *JVM* (à l'exécution) recherche une définition pour cette méthode m() qui soit **la plus précise possible**,i.e., la plus proche du type « réel » de la référence contenue dans v
  - Par exemple, si Object v = new Pixel(1,3);
    - C'est possible car Pixel est un sous-type de Object
    - v est déclarée de type Object (***info compile time***), mais contient en réalité une référence à un objet de type Pixel (***info run time***)
  - Si m() est la méthode toString(), le compilateur dit « banco » pour Object.toString() et la JVM recherche Pixel.toString()
    - Si Pixel.toString() n'existe pas, celle de Object qui est « héritée »

Et voilà le travail...

- Il « suffit » alors de donner **sa propre définition** de toString() dans la classe Pixel
  - On dit qu'on « **redéfinit** » (***Override***) la méthode toString()

```
public class Pixel {
  // ...
  @Override
  public String toString() {
    return "("+x+","+y+")";
  }
  public static void main(String[] args) {
    Object o = new Pixel(1,3);
    System.out.println(o);              // (1,3)

    // et a fortiori
    Pixel p = new Pixel(5,7);
    System.out.println(p);              // (5,7)
  }
}
```

L'annotation `@Override` demande au compilateur de vérifier qu'on est bien en train de redéfinir une méthode qui sinon serait héritée

`void java.io.PrintStream.println(Object x)`

`void java.io.PrintStream.println(Object x)`

Le sous-typage

- L'idée (de la redéfinition de toString()) est que:
  - Le comportement dépend de l'objet réellement contenu dans la variable
  - L'affichage d'un objet est différent de l'affichage d'un Pixel
  - Mais les deux peuvent s'afficher...
  - Ils disposent tous les deux de la « méthode » toString()
- Plus généralement, on voudrait avoir des **types**
  - Sur lesquels un **ensemble de méthodes est disponible**
  - Mais dont la **définition exacte dépend du sous-type**
  - La méthode finalement exécutée sera la plus précise possible
- Exemple: toute figure a une surface, mais la surface d'un carré ne se calcule pas comme la surface d'un cercle...

Comment définir des sous-types

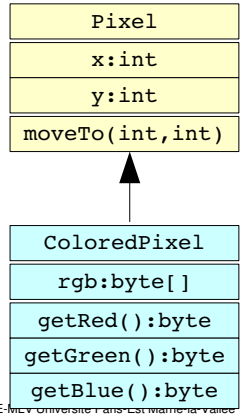
- On a vu les relations de conversions qui sont autorisées entre **types primitifs**
  - `byte < short < int < long < float < double`
    `char < int`
- On a vu que toute classe A hérite de la classe Object, et définit ainsi un type A qui est **sous-type** du type Object
- L'**héritage** définit des sous-types:
  - Soit explicitement: `class Student extends Person { ... }`
  - Soit implicitement `Pixel` ou `int[]` héritent de `Object`
- L'**implémentation d'interface** définit des sous-types
  - Une interface **déclare les méthodes applicables** par les objets des classes qui l'implémentent
  - Une **classe** implémente l'interface en **définissant ses méthodes**: `class Carre implements Mesurable { ... }`

L'héritage

- Consiste à définir une classe, dite classe dérivée ou classe fille, à partir d'une autre classe, dite classe de base ou classe mère, en récupérant automatiquement dans la classe dérivée tous les membres de la classe de base, et en lui en ajoutant éventuellement de nouveaux membres

```
public class Pixel {
    private int x;
    private int y;
    public void moveTo(int newX, int newY) {
        this.x = newX;
        this.y = newY;
    }
}
```

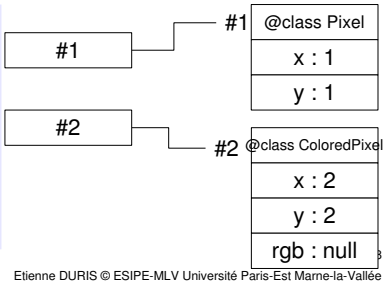
```
public class ColoredPixel extends Pixel {
    private byte[] rgb;
    public byte getRed() { return rgb[0]; }
    public byte getGreen() { return rgb[1]; }
    public byte getBlue() { return rgb[2]; }
}
```



Que sont les objets de la classe dérivée?

- Tout objet d'une classe dérivée est considéré comme étant avant tout un objet de la classe de base
  - Un pixel coloré est avant tout un pixel
- Tout objet d'une classe dérivée « cumule » les champs dérivés dans la classe de base avec ceux définis dans sa propre classe
  - Il y a un int x et un int y dans un objet de la classe ColoredPixel

```
public static void main(String[] args) {
    Pixel p = new Pixel();
    p.moveTo(1, 1);

    ColoredPixel cp = new ColoredPixel();
    cp.moveTo(2, 2);

    cp.getRed();  // NullPointerException
}
```



Tous les champs sont hérités

- Ils peuvent être manipulés si leur accessibilité le permet
  - Si x n'est pas private dans Pixel, on peut dire this.x dans ColoredPixel
- Ils peuvent être **masqués** par la définition de champs qui ont le même nom dans la classe dérivée
  - Si String x est déclaré dans ColoredPixel, c'est celui qui sera considéré dans cette classe quand on parle de this.x
  - Il est possible de manipuler celui qui est masqué (s'il est accessible) par la notation **super.x**

```
public class Pixel {
    int x;
    private int y;
    // ...
}
```

```
public class ColoredPixel extends Pixel {
    private byte[] rgb;
    String x;
    void test() {
        System.out.println(this.x);   // null
        System.out.println(super.x);  // 0
    }
}
```


Résolution du champ à accéder

- La **détermination du champ** qui doit être accédé s'appelle « **la résolution** »
  - il s'agit de savoir où on ira chercher la valeur à l'exécution
- La résolution des champs est effectuée par le compilateur, en **fonction du type déclaré de la variable** qui contient la référence

```
public static void main(String[] args) {
    ColoredPixel cp = new ColoredPixel();
    // le type déclaré de cp est ColoredPixel

    System.out.println(cp.x);   // null

    Pixel p = cp;
    // le type déclaré de p est Pixel, même si la référence
    // contenue dans p est celle d'un ColoredPixel

    System.out.println(p.x);    // 0
}
```

Le masquage des champs

- Si c'est possible de créer dans une classe héritée un champ qui a le même nom qu'un champs d'une superclasse
  - C'est rarement très utile, souvent une mauvaise idée, source d'erreur
- **super**, c'est **this** vu avec le **type de la super-classe**
- **super.super.x** n'existe pas...
- Pas plus que ref.super ni ref.super.x...
- En revanche, le **transtypage** (***cast***) permet d'accéder en changeant le type déclaré de la référence ref

```
class A {
  int x = 1;
}

class B extends A {
  String x = "zz";
}

class C extends B {
  boolean x = true;
  public static void main(String[] args) {
    C c = new C();
    System.out.println(c.x);       // true
    System.out.println(((B)c).x); // zz
    System.out.println(((A)c).x); // 1
  }
}
```


Transtypage, type déclaré et type « réel »

- Le transtypage de référence est le fait de considérer explicitement (forcer) une référence comme étant d'un type donné (qui n'est pas nécessairement le type de l'objet accessible via cette référence)
- La machine virtuelle vérifiera, à l'exécution, que le type en question est bien compatible et que voir cette référence comme étant de ce type là est possible; dans le cas contraire, l'exécution provoque une **ClassCastException**

```
class A { }
class B extends A { }
class C extends B { }
```

```
B b = new B();
A a = b;
// B b2 = a;  // incompatible types
B b2 = (B) a; // OK
C c = (C) a;  // ClassCastException
```

```
Object o;
if(Math.random() > 0.5)
  o = "toto";
else
  o = new Object();
String s = (String) o;
// Compile toujours mais a une
// chance sur deux de lever une
// ClassCastException...
```

L'opérateur instanceof

- Il est possible d'assurer un transtypage sans exception en utilisant l'opérateur **`x instanceof T`**
  - **`x`** doit être une (variable contenant une) référence ou **`null`**
  - **`T`** doit rerpésenter un type
  - Le résultat vaut true si x n'est pas null et s'il peut être affecté dans T sans ClassCastException; sinon c'est false.

```
class A { }
class B extends A { }
class C extends B { }
```

```
A ab = null;
System.out.println(ab instanceof A);  // false
ab = new B();
System.out.println(ab instanceof A);  // true
System.out.println(ab instanceof B);  // true
System.out.println(ab instanceof C);  // false
```

```
Object o; String s;
if(Math.random()>0.5)
  o = "toto";
else
  o = new Object();
if (o instanceof String)
  s = (String) o;  // OK...
```


Constructeurs et héritage

- La construction (initialisation) de toute instance d'une classe débute par la construction (initialisation) d'une instance d'Object
  - En pratique, tout constructeur débute par un appel au constructeur de sa super-classe: **`super()`**
  - Doit obligatoirement être la première instruction du constructeur
  - Le constructeur implicite (ajouté par le compilateur) fait appel au constructeur sans argument de la super-classe

```
public class Pixel {
    private int x;
    private int y;
    public Pixel(int x, int y) {
        this.x = newX;
        this.y = newY;
    }
    // ...
}
```

```
public class ColoredPixel extends Pixel {
  private byte[] rgb;
  public ColoredPixel() {
    // super(); // Constructeur Pixel() is undefined
    super(0,0); // OK; notez que x et y sont private!
    rgb = new byte[3];
  }
}
```

Constructeurs et initialisations

- Il faut voir le constructeur d'une classe comme une étape dans l'initialisation des objets de cette classe:
  - Commence par l'initialisation des champs de l'objet « en tant qu'instance de la super-classe »: c'est l'objectif de l'appel à super(...)
  - Ensuite il se charge d'initialiser les objets qui lui sont propres (en tant qu'instance de la classe dérivée)
  - L'appel à super ne peut pas utiliser des champs dont l'existence ou la valeur dépendrait de l'instance de la classe dérivée...

```
public class ColoredPixel extends Pixel {
  private int v = 0;
  private static int s = 0;
  public ColoredPixel() {
    // super(v,v);
    // error: cannot reference v before supertype constructor has been called
    super(s,s); // OK
  }
}
```

Redéfinition de méthode

- Fournir une nouvelle définition pour la même méthode:
  - Même nom, mêmes arguments, code différent
  - L'annotation **`@Override`** demande au compilateur de vérifier

```
public class ColoredPixel extends Pixel {
  private byte[] rgb;
  // ...
  @Override
  public String toString() {
    return super.toString()+"["+rgb[0]+":"+rgb[1]+":"+rgb[2]+"]";
  }
  public static void main(String[] args) {
    ColoredPixel cp = new ColoredPixel(2,2);
    System.out.println(cp);  // (2,2)[0:0:0]
    Pixel p = new Pixel(5,5);
    System.out.println(p);   // (5,5)
    Object o =  new ColoredPixel(2,2);
    System.out.println(o);   // (2,2)[0:0:0]
  }
}
```


L'héritage des méthodes

- En plus des champs, en tant que « membres », la **classe dérivée hérite des méthodes** de la classe de base
- Seuls **les constructeurs ne sont pas hérités**
  - Ils restent propres à leur classe
- Attention: le **code** (sémantique) d'une méthode de la super classe **peut ne plus être correct** dans la classe dérivée
  - Pour l'exemple de Pixel et ColoredPixel:
    - moveTo() héritée dans ColoredPixel a une sémantique correcte
    - Mais que dire de toString() qui donne une représentation textuelle?
    - Que dire de sameAs() qui compare un pixel à un autre pixel?
  - Dans certains cas, il faut **donner une nouvelle définition** de la même méthode à utiliser pour la classe dérivée

Redéfinition (méthodes) *versus* masquage (champs)

- Les **champs** définis dans les classes dérivées sont tous présents dans l'objet instance de la classe dérivée
  - Même s'ils ont même nom et même type
  - On peut accéder à celui immédiatement supérieur par super.x
  - La **résolution dépend du type déclaré** du paramètre
  - Ça permet d'accéder à chacun d'entre eux par transtypage
- Pour la **méthode**, une seule est conservée
  - On peut accéder à celle immédiatement supérieure par super.m()
  - La **résolution** est faite en deux temps
    - **Compile-time**: on vérifie que c'est possible sur le type déclaré
    - **Runtime**: on cherche la plus précise étant donnée le type « réel »
  - Les autres ne sont plus accessibles

Redéfinition versus surcharge

- Si la signature de la méthode qu'on définit dans la classe dérivée n'est pas le même que celle de la classe de base, il s'agit de **surcharge**:
  - Les deux méthodes cohabitent dans la classe dérivée

```
class A {
  void m1() { ... }
  void m2() { ... }
  Pixel m3() { ... }
  void m4(Pixel p) { ... }
}
class B extends A {
  @Override void m1() { ... }              // redefinition
            void m2(int a) { ... }         // surcharge
  @Override ColoredPixel m3() { ... }      // redefinition
  @Override void m4(Pixel p) { ... }       // redefinition
            void m4(ColoredPixel p) { ... } // surcharge
}
```


Les principes de la redéfinition

- Quand on redéfinit une méthode m() dans B alors qu'elle était définie dans A, où B est un sous-type de A
  - L'objectif est de lui donner une définition plus précise *(mieux adaptée à B qu'à A)*, de sorte qu'elle soit appelée à run-time, y compris si à compile-time le compilateur n'avait vu que celle qui est définie dans A
- Le compilateur est sensé éviter les mauvaises surprises (découvrir un problème à run-time): c'est ce qui gouverne les règles
  - Une méthode d'instance ne peut pas redéfinir une méthode static
  - L'accessibilité de la méthode redéfinie ne peut pas être plus restrictive
  - Le type de retour ne peut pas être d'un super-type (références)
  - Les exceptions propagées ne peuvent être que d'un sous-type

Le cas de sameAs...

- On a redéfinit la méthode toString()
  - Donner une représentation textuelle plus précise dans ColoredPixel : c'est facile (même signature)
- On voudrait aussi redéfinir sameAs()
  - Actuellement, la méthode héritée est fausse!
  - Redéfinition: pas si simple avec la signature de **sameAs()**

```
// public boolean sameAs(ColoredPixel p) { // Surcharge et pas redéfinition

@Override
public boolean sameAs(Pixel p) {
  // return (x==p.x) && (y==p.y) & p.rgb[0]=rgb[0] ...
  // Aïe! x and y: private access in Pixel
  // return super.sameAs(p) && p.rgb[0]=rgb[0] ...
  // Aïe! rgb is not a field
  // return super.sameAs(p) && ((ColoredPixel)p).rgb[0]==rgb[0] ...
  // Compile, mais si p n'en est pas un ColoredPixel... Aïe!
  // Il faut utiliser instanceof...
}
```


La méthode equals()

- De la même manière qu'il existe une méthode toString() dans la classe Object, que toute sous-classe peut redéfinir
- Il existe dans Object une méthode equals(Object obj) dont le « contrat » est clairement établi par la documentation
  - Par défaut, elle teste l'**égalité primitive des références** (même objet)
  - C'est celle-là qu'il faut redéfinir!

```
public class Pixel {
  private int x, y;
  // ...
  @Override
  public boolean equals(Object obj) {
    if(!(obj instanceof Pixel))
      return false;
    Pixel p = (Pixel) obj;
    return (x==p.x) && (y==p.y);
  }
}
```

```
public class ColoredPixel extends Pixel {
  private byte[] rgb;
  @Override
  public boolean equals(Object obj) {
    if(!(obj instanceof ColoredPixel))
      return false;
    ColoredPixel cp = (ColoredPixel) obj;
    return super.equals(obj) &&
         rgb[0]==cp.rgb[0] &&
         rgb[1]==cp.rgb[1] &&
         rgb[2]==cp.rgb[2];
  }
}
```

Le contrat de la méthode equals()

- Définit une relation d'équivalence sur les références non-nulles
  - **Reflexive**
    - Pour toute référence x non nulle, x.equals(x) vaut true
  - **Symétrique**
    - Pour toutes références x et y non nulles, x.equals(y) ssi y.equals(x)
  - **Transitive**
    - Pour toutes références x, yet z non nulles, si x.equals(y) et y.equals(z) alors x.equals(z)
  - **Cohérente**
    - Tant qu'on ne modifie pas les valeurs utilisées pour tester l'égalité, la valeur de x.equals(y) retourne toujours la même valeur
  - Pour toute référence x non nulle, x.equals(null) vaut false
- Des objets égaux au sens de equals doivent avoir le même **hashcode**
  - Redéfinition de equals() implique en général redéfinition de hashCode()

Bof bof... dans notre cas de ColoredPixel, c'est limite...

La symétrie peut se discuter...

- Demandez à un Pixel en (2,2) s'il est égal à un ColoredPixel en (2,2), il dira que oui!
  - Il teste uniquement les coordonnées...
- Demandez à un **ColoredPixel magenta** en (2,2) s'il est égal à un Pixel en (2,2), il dira que non!
  - Il est sensé tester la couleur que le Pixel n'a même pas...
- On peut trouver que ce code est acceptable... il faut juste être conscient de la nuance...

```
public class ColoredPixel extends Pixel {
  // ...
  public static void main(String[] args) {
    Object o1 = new Pixel(2,2);
    Object o2 = new ColoredPixel(2,2);
    System.out.println(o1.equals(o2)); // true
    System.out.println(o2.equals(o1)); // false
  }
}
```


Pour être plus strict...

- Il faut considérer que deux objets qui ne sont pas de la même classe ne peuvent pas être égaux
  - instanceof ne suffit plus
  - Il faut connaître la classe « exacte » de l'objet (à runtime)
    - Méthode Class getClass de la classe Object

Dans Pixel :

```
@Override
public boolean equals(Object obj) {
  if(obj.getClass() != Pixel.class)
    return false;
  Pixel p = (Pixel) obj;
  return (x==p.x) && (y==p.y);
}
```

```
@Override
public boolean equals(Object obj) {
  if(obj.getClass() != ColoredPixel.class)
    return false;
  ColoredPixel cp = (ColoredPixel) obj;
  return super.equals(obj) &&
    Arrays.equals(this.rgb, cp.rgb);
}
```

```
public static void main(String[] args) {
  Object o1 = new Pixel(2,2);
  Object o2 = new ColoredPixel(2,2);
  System.out.println(o1.equals(o2)); // false
  System.out.println(o2.equals(o1)); // false
}
```

Dans ColoredPixel :

La méthode hashCode()

- Cette méthode est utlisée lorsqu'on stocke des objets dans une table de hachage (exemple java.util.HashMap)
- Elle établit également un « **contrat** » (de pair avec equals())
  - **`public int hashCode()`**
  - Étant donnée une exécution de la JVM, différents appels à la méthode hashCode() doivent retourner la même valeur tant qu'on ne modifie pas les valeurs utilisées pour tester l'égalité (equals())
  - Si deux objets sont égaux au sens de equals(), la méthode hashCode() appelée sur les deux doit produire la même valeur
  - Deux objets distincts au sens de equals() peuvent avoir des hashCode() identiques (c'est une « collision »), mais fournir des hashCode() distincts pour des objets distincts au sens de equals() améliore la performance des tables de hachage.

Utilisation de hashCode() et equals()

- Les ensembles, les tables de hachage, etc.
- Si equals est redéfinie, mais pas hashCode, voilà ce qui arrive

```
import java.util.HashSet;

public class Pixel {
  // ...
  public static void main(String[] args) {
    Pixel zero = new Pixel(0,0);
    Pixel def = new Pixel();

    HashSet set = new HashSet();
    set.add(zero);

    System.out.println(set.contains(def)); // false

    System.out.println(zero.hashCode());   // 1522065175
    System.out.println(def.hashCode());    //  524193161

    System.out.println(zero.equals(def));  // true
  }
}
```

Incohérence entre equals() et hashCode()

Exemple de hashCode() pour nos pixels

```
public class Pixel {
  // ...
  @Override
  public boolean equals(Object obj) {
    if(!(obj instanceof Pixel))
      return false;
    Pixel p = (Pixel) obj;
    return (x==p.x) && (y==p.y);
  }
  @Override
  public int hashCode() {
    return Integer.rotateLeft(x,16) ^ y;
  }
```

```
public static void main(String[] a){
  Pixel zero = new Pixel(0,0);
  Pixel def = new Pixel();
  HashSet set = new HashSet();
  set.add(zero);
  set.contains(def); // true
  zero.hashCode();   // 0
  def.hashCode();    // 0
  zero.equals(def);  // true
}
```

```
public class ColoredPixel extends Pixel {
  private byte[] rgb;
  // ...
  @Override
  public int hashCode() {
    // return super.hashCode() ^ Integer.rotateLeft(rgb[0],16)
    //        ^ Integer.rotateLeft(rgb[1],8) ^ rgb[0];
    return super.hashCode() ^ Arrays.hashCode(rgb);
  }
}
```


Les classes et méthodes « final »

- Le mot-clé **final** existe pour les méthodes:
  - Il signifie que la méthode ne pourra pas être redéfinie dans une sous-classe
  - Peut être utile pour garantir qu'aucune autre définition ne pourra être donnée pour cette méthode (sécurité)
- Le mot-clé **final** existe pour les classes:
  - Il devient alors impossible d'hériter de cette classe
  - Les méthodes se comportent comme si elles étaient final

Les interfaces

- Une **classe** définit:
  - Un type
  - Une structure de données pour les instances (les champs)
  - Des méthodes avec leur code (leur définition)
- Une **interface** définit:
  - Un type
  - Des méthodes sans leur code (méthodes abstraites)
- Une interface **ne peut pas être instanciée**
- Elle est destinée à être « **implémentée** » par des classes
  - À qui elle donnera son type
  - Qui fourniront des définitions pour les méthodes déclarées (code)

Intérêt des interfaces

- Donner un type commun à des classes différentes pour en faire un même usage
- Exemple:
  - Manipuler des tableaux de « trucs » qui ont chacun une surface
  - Faire la somme des surfaces des trucs qui sont dans le tableau

```
public interface Surfaceable {
  public double surface();
}
```

```
public class AlgoOnTrucs {
  public static double totalSurface(Surfaceable[] array) {
    double total = 0.0;
    for(Surfaceable truc : array)
      total += truc.surface();
    return total;
  }
}
```


Utilisation d'interface

- Les 2 principaux avantages:
  - L'algo de la méthode totalSurface(Surfaceable[] array) fonctionne indépendamment de la classe réelle des objets qui sont stockés dans array: c'est le **sous-typage**
  - La méthode surface() effectivement appelée sur les objets contenus dans le tableau sera la plus précise possible, en fonction du type réel de cet objet: c'est le **polymorphisme**

```
public class AlgoOnTrucs {

  public static double totalSurface(Surfaceable[] array) { ... }

  public static void main(String[] args) {
    Rectangle rectangle = new Rectangle(2,5);
    Square square = new Square(10);
    Circle circle = new Circle(1);
    Surfaceable[] t = {rectangle, square, circle};
    System.out.println(totalSurface(t));  // 113.1415926535898
  }
}
```

Implémentation d'interface

```
public class Square implements Surfaceable {
  private final double side;
  public Square(double side) {
    this.side = side;
  }
  @Override
  public double surface() {
    return side * side;
  }
}
```

```
public class Rectangle implements Surfaceable {
  private final double height;
  private final double width;
  public Rectangle(double height, double width) {
    this.height = height;
    this.width = width;
  }
  @Override
  public double surface() {
    return height * width;
  }
}
```

```
public class Circle implements Surfaceable {
  private final double radius;
  public Circle(double radius) {
    this.radius = radius;
  }
  @Override
  public double surface() {
    return Math.PI * radius * radius;
  }
}
```


Les membres des interfaces

- Contiennent des déclarations de méthode publiques
  - Toutes les méthodes sont **abstract public**
    - même si non spécifié

```
public interface Surfaceable {
  double surface(); // equivaut à
  public abstract double surface();
}
```

- Peuvent définir des champs publiques constants
  - Tous les champs sont **public final static**
    - Le compilateur ajoute les mot-clés

```
public interface I {
  int field = 10; // equivaut à
  public final static int field = 10;
}
```

- Il n'est **pas possible d'instancier** une interface
  - On ne peut que déclarer des variables avec leur type
  - Ces variables pourront reçevoir des références à des objets qui sont des instances d'une classe qui implémente l'interface

Implémentation d'interface et sous-typage

- Une classe peut **implémenter** une interface
  - Mot clé **`implements`**

```
public class Rectangle implements Surfaceable {
  ...
}
```

  - La classe Rectangle définit un **sous-type** de Surfaceable

```
Surfaceable s = null;

s = new Rectangle(2,5);
```

- Une interface ne peut pas implémenter une autre interface
  - On ne saurait pas comment implémenter les méthodes

Sous-typage entre interfaces

- Une interface peut **hériter** d'**une ou plusieurs** autres interfaces
  - Mot clé **`extends`**

```
public interface Paintable extends Surfaceable {
  double paint(byte[] color, int layers);
}
```

  - Le type Paintable est un **sous-type** de Surfaceable

```
Surfaceable[] array = new Surfaceable[3]; // On peut créer des tableaux!
Paintable p = null;
array[0] = p; // OK: Paintable < Surfaceable
p = array[1]; // Cannot convert from Surfaceable to Paintable
```

  - Séparer les super-types avec des virgules

```
public interface SurfaceableAndMoveable
  extends Surfaceable, Moveable { ... }
```

  - Le type SurfaceableAndMoveable définit un **sous-type** des deux types **Surfaceable** et de **Moveable** (**sous-typage multiple**)
    - SurfaceableAndMoveable < Surfaceable et SurfaceableAndMoveable < Moveable
    - Ces deux dernières n'ont aucune relation entre-elles

Héritage de classe et implémentation d'interfaces: sous-typage multiple

- Une classe peut **hériter** d'une classe et **implémenter** plusieurs interfaces

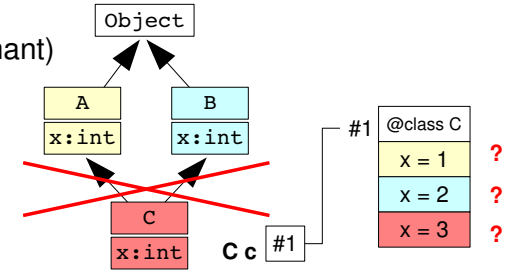
```
public class SolidCircle extends Circle implements Paintable, Moveable {
  private final Point center;
  public SolidCircle(Point center, double radius) {
    super(radius);
    this.center = center;
  }
  @Override  // Pour pouvoir implémenter Paintable
  public double paint(byte[] color, int layers) {
    // doThePaintingJob(color,layers);
    return layers * surface(); // SolidCircle < Circle < Surfaceable
  }
  @Override  // Pour pouvoir implémenter Moveable
  public void moveTo(int x, int y) {
    center.moveTo(x,y);
  }
  public static void main(String[] args) {
    SolidCircle sc = new SolidCircle(new Point(0,0), 3);
    Circle c = sc;      double d = c.surface();      // SolidCircle < Circle
    Paintable p = sc;   p.paint(new byte[]{0,0,0},2);// SolidCircle < Paintable
    Moveable m = sc;    m.moveTo(1, 1);              // SolidCircle < Moveable
  }
}
```

Vérifications du compilateur

- **Toutes les méthodes déclarées** (abstract) dans l'ensemble des interfaces dont on revendique l'implémentation **doivent être implantées**
  - Définies avec leur code
- Le modificateur de visibilité **ne peut pas être autre chose que `public`**
  - Même si on a mis la visibilité par défaut dans l'interface, le compilateur y ajoute **`public abstract`**
- Que se passe t il si plusieurs méthodes de même nom et même signature de différentes interfaces doivent être implémentées dans la même classe?

Pourquoi pas hériter de plusieurs classes

- A cause des **champs** (diamant)
  - Qui pourraient provenir de deux classes dont on hérite et qui devraient « **coexister** » dans un même objet!



- L'héritage multiple d'interfaces ne pose pas de problèmes
  - Au même titre que l'implémentation multiple d'interfaces
  - On « récupére » par héritage uniquement
    - des **déclarations** de méthodes (qui ne possèdent pas de définition)
    - des **constantes** dont la portée est limitée à l'interface dans laquelle elles sont définies (espace de nom)

Design: interface ou héritage

- On **hérite d'une classe**
  - pour créer un nouveau type qui est « **une sorte particulière** » de classe de base
- On définit une interface et on l'implémente
  - Pour une fonctionnalité transverse
    - Comparable, Closeable, *Mesurable, Déplacable...*
  - Pour regrouper un ensemble de fonctionalités qui pourront être implémentées par des instances qui en implantent déjà d'autres (ou qui héritent d'une autre classe)
    - `class RandomAccessFile extends Object implements DataOutput, DataInput, Closeable {...}`

Les classes abstraites

- Dans une **interface**, **tout doit être abstrait** (méthodes)
  - Les méthodes doivent toutes être abstraites
  - Elles ne peuvent être que publiques
- Dans une **classe, tout doit être concret**
  - Les méthodes doivent être définies (leur code)
  - Elles peuvent avoir des modificateurs de visibilités différents
- Une **classe abstraite** permet de créer une type à mi-chemin
  - Il s'agit d'une classe qui peut avoir des **méthodes abstraites**
  - Elle est considérée comme partiellement implantée, donc **non instanciable**
  - Elle peut éventuellement n'avoir aucune méthode abstraite
  - Le mot-clé abstract fait qu'elle n'est pas instanciable

Si mon algo manipule un type...

- ... sur lequel je sais faire des choses, mais pas tout!

```
public class AlgoOnTrucs {
  public static double totalSurface(Surfaceable[] array) {
    double total = 0.0;
    for(Surfaceable truc : array)
      total += truc.surface();
    return total;
  }
  public static Surfaceable theBigger(Surfaceable[] array) {
    Surfaceable bigger = array[0];
    for(int i = 1; i<array.length; i++) {
      if (array[i].biggerThan(bigger))
        bigger = array[i];
    }
    return bigger;
  }
  public static void main(String[] args) {
    Rectangle rectangle = new Rectangle(2,5);
    Square square = new Square(10);
    Circle circle = new Circle(1);
    Surfaceable[] t = {rectangle, square, circle};
    System.out.println(totalSurface(t));  // 113.1415926535898
    System.out.println(theBigger(t)); // Square@a6eb38a
  }
}
```

Le code de surface() ne peut être défini que dans la classe « concrète »

Mais je peux écrire un code pour biggerThan() qui ne dépend pas de cette classe concrète

Exemple de classe abstraite

- L'implémentation de la méthode biggerThan() marchera dès qu'on l'appelera sur un objet (instance) d'une classe concrète, puisque celle-ci aura donné l'implémentation de la méthode surface().

```
public abstract class Surfaceable {
  public abstract double surface();
  public boolean biggerThan(Surfaceable bigger) {
    return surface() > bigger.surface();
  }
}
```

```
public class Rectangle extends Surfaceable {
  ...
}
```

```
public class Circle extends Surfaceable {
  ...
}
```

```
public class Square extends Surfaceable {
  private final double side;
  public Square(double side) {
    this.side = side;
  }
  @Override
  public double surface() {
    return side * side;
  }
}
```


Les Exceptions

- Mécanisme qui permet de reporter des erreurs vers les méthodes appelantes
  - « à travers » la pile d'appel des méthodes
  - avec la possibilité d'intercepter/traiter ou de propager
- Problème en C :
  - prévoir une plage de valeurs dans la valeur de retour pour signaler les erreurs.
  - Propager les erreurs “manuellement”
- En Java comme en C++, le mécanisme de remonté d’erreur est gérée par le langage.

Exemple d'exception

Un exemple simple

```
public class ExceptionExample {
  public static char charAt(char[] array,int index) {
    return array[index];
  }

  public static void main(String[] args) {
    char[] array=args[0].toCharArray();
    charAt(array,0);
  }
}
```

Lors de l’exécution :

```
C:\eclipse\workspace\java-avancé>java ExceptionExample
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0
        at ExceptionExample.main(ExceptionExample.java:18)
```


Exemple d'exception (suite)

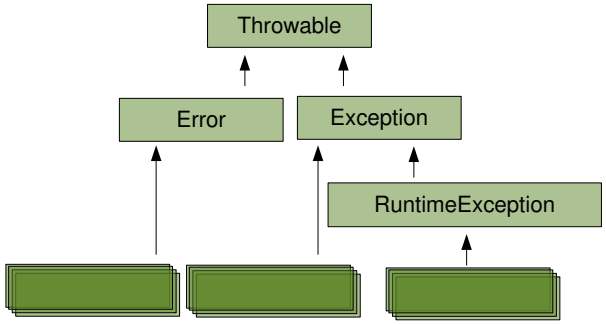
En reprenant le même exemple :

```
public class ExceptionExample {
  public static char charAt(char[] array,int index) {
    if (index<0 || index>=array.length)
      throw new IllegalArgumentException("bad index "+index);
    return array[index];
  }

  public static void main(String[] args) {
    char[] array=args[0].toCharArray();
    charAt(array,0);
    charAt(array,1000);
  }
}
```

```
C:\eclipse\workspace\java-avancé>java ExceptionExample toto
Exception in thread "main" java.lang.IllegalArgumentException: bad index
1000
        at ExceptionExample.charAt(ExceptionExample.java:13)
        at ExceptionExample.main(ExceptionExample.java:20)
```

Types d'exceptions

Il existe en Java tout une hiérarchie de « types » d'exceptions



Arbre de sous-typage des exceptions

Types d'exceptions (2)

Throwable est la classe « mère » de toutes les exceptions

Les **Error** correspondent à des exceptions qu'il est rare d'attraper.

Les **RuntimeException** que l'on peut rattraper mais que l'on n'est pas obligé.

Les **Exception** que l'on est obligé d'attraper (**try**/**catch**) ou de dire que la méthode appelante devra s'en occuper (**throws**).

Exceptions levées par la VM

Les exceptions levées par la VM correspondent :

Erreur de compilation ou de lancement

NoClassDefFoundError, ClassFormatError

problème d'entrée/sortie :

IOException, AWTException

problème de ressource :

OutOfMemoryError, StackOverflowError

des erreurs de programmation (runtime)

NullPointerException, ArrayIndexOutOfBoundsException, ArithmethicException

Attraper une exception

try/catch permet d'attraper les exceptions

```
public class CatchExceptionExample {
  public static void main(String[] args) {
    int value;
    try {
      value=Integer.parseInt(args[0]);
    } catch(NumberFormatException e) {
      value=0;
    }
    System.out.println("value "+value);
  }
}
```

parseInt

```
public static int parseInt(String s)
                    throws NumberFormatException
```

Parses the string argument as a signed decimal integer. The characters in the string must all be decimal digits, except that the first character may be an ASCII minus sign `'-'` (`'\u002D'`) to indicate a negative value. The resulting integer value is returned, exactly as if the argument and the radix 10 were given as arguments to the `parseInt(java.lang.String, int)` method.

Parameters:
`s` - a `String` containing the `int` representation to be parsed
Returns:
the integer value represented by the argument in decimal.
Throws:
`NumberFormatException` - if the string does not contain a parsable integer.

Attraper une exception

try/catch définit obligatoirement un bloc.

```
public class CatchExceptionExample {
  public static void main(String[] args) {
    int value;
    try {
      value=Integer.parseInt(args[0]);
    } catch(ArrayIndexOutOfBoundsException e) {
      System.err.println("no argument");
      e.printStackTrace();
      return;
    } catch(NumberFormatException e) {
      value=0;
    }
    System.out.println("value "+value);
  }
}
```

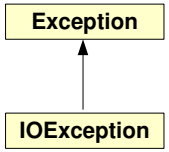
Sinon cela ne compile pas car value n'est pas intitialisée

Attraper une exception

Attention, les blocs **catch** sont testés dans l'ordre d'écriture !

Un catch inatteignable est une erreur

```
public class CatchExceptionExample {
  public static void main(String[] args) {
    int value;
    try {
      value=...
    } catch(Exception e) {
      value=1;
    } catch(IOException e) { // jamais appelé
      value=0;
    }
    System.out.println("value "+value);
  }
}
```



Ne pas attraper tout ce qui bouge

Comment passer des heures à débugger;-)

```
public static void aRandomThing(String[] args) {
  return Integer.parseInt(args[-1]);
}
public static void main(String[] args) {
  ...
  try {
    aRandomThing(args);
  } catch(Throwable t) {
    // surtout ne rien faire sinon c'est pas drôle
  }
  ...
}
```

Éviter les catch(Throwable) ou catch(Exception) !!

La directive throws

Indique qu'une exception peut-être levée dans le code mais que celui-ci ne la gère pas (pas de try/catch).

```
public static void f(String author) throws OhNoException {
  if ("dan brown".equals(author))
    throw new OhNoException("oh no");
}
public static void main(String[] args) {
  try {
    f(args[0]);
  } catch(OhNoException e) {
    tryToRecover();
  }
```

throws n'est obligatoire que pour les Exception

pas les erreurs (Error)

ni pour ou les runtimes (RuntimeException)

Alors throws ou catch

- Si on appelle une méthode qui lève une exception non runtime
  - **`catch`** si l'on peut reprendre sur l'erreur et faire quelque chose de cohérent (appliquer une action corrective)
  - Sinon **`throws`** pour propager l'exception vers celui qui a appelé la méthode qui fera ce qu'il doit faire

Un exemple

La même exception peut être capturée ou propagée selon le contexte

```
public static Config initConfig(String userName) throws IOException {
    Config conf = null;
    FileInputStream fis = null;
    try {
        fis = new FileInputStream(userName+"_config");
        conf = new Config(userName);
    } catch (FileNotFoundException e) {
        // no config file for this user...
        e.printStackTrace(System.err);
        fis = new FileInputStream("default_config");
        conf = new Config(userName);
    }
    readConfFromInputStream(conf,fis);
    fis.close();
    return conf;
}
```


Le bloc finally

Sert à exécuter un code quoi qu'il arrive
(fermer un fichier, une connexion, libérer une ressources)

```
public class FinallyExceptionExample {
  public static void main(String[] args) {
    ReentrantLock lock = new ReentrantLock();
    lock.lock();
    try {
      doSomething();
    } finally {
      lock.unlock()
    }
  }
}
```

Le **catch** n'est pas obligatoire.

Exception et StackTrace

Lors de la création d'une exception, la VM calcule le *StackTrace*

Le *StackTrace* correspond aux fonctions empilées (dans la pile) lors de la création

Le calcul du *StackTrace* est quelque chose de coûteux en performance.

Le chaînage des exceptions

- Il est possible d'encapsuler une exception dans une autre
  - **new Exception("msg", throwable)**
  - **new Exception("msg").initCause(throwable)**
- Permet de lever une exception suffisamment précise tout en respectant une signature fixée

Le chaînage des exceptions (2)

Comme close ne peut pas renvoyer d'**Exception**, on encapsule celle-ci

```
interface Closable {
  public void close() throws IOException;
}
public class DB {
  public void flush() throws OhNoException {
    throw new OhNoException("argh !");
  }
}
public class ExceptionChain implements Closable {
  private final DB db=new DB();
  public void close() throws IOException {
    try {
      db.flush();
    } catch(OhNoException e) {
      throw (IOException)new IOException().initCause(e);
      // ou à partir de la 1.6
      throw new IOException(e);
    } } }
```


Le chaînage des exceptions (3)

Exécution de l'exemple précédent :

```
Exception in thread "main" java.io.IOException
    at ExceptionChain.close(ExceptionChain.java:27)
    at ExceptionChain.main(ExceptionChain.java:32)
Caused by: fr.umlv.OhNoException: argh !
    at DB.fush(ExceptionChain.java:20)
    at ExceptionChain.close(ExceptionChain.java:25)
    ... 1 more
```

Désencapsulation :

```
public static void caller() throws OhNoException {
  ExceptionChain chain=new ExceptionChain();
  try {
    chain.close();
  } catch(IOException e) {
    Thowable t=e.getCause();
    if (t instanceof OhNoException)
      throw (OhNoException)t;
    ...
```


Le mot-clé assert

Le mot-clé **assert** permet de s'assurer que la valeur d'une expression est vraie

Deux syntaxes :

assert *test*; `assert i==j;`

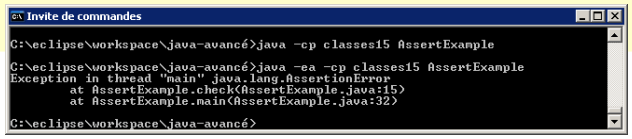
assert *test* : *mesg*; `assert i==j:"i not equals to j";`

Par défaut, les **assert** ne sont pas exécutés, il faut lancer **java -ea** (enable assert)

assert et AssertionError

Si le test booléen du **assert** est faux, la VM lève une exception **AssertionError**

```
public class AssertExample  {
  private static void check(List list)  {
    assert list.isEmpty() || list.indexOf(list.get(0))!=-1;
  }
  public static void main(String[] args) {
    List list=new BadListImpl();
    list.add(3);
    check(list);
    ...
  }
}
```



Exceptions et programmeur

- Le programmeur va utiliser des exceptions pour assurer :
  - Que son code est bien utilisé (pré-condition)
  - Que l'état de l'objet est bon (pré-condition)
  - Que le code fait ce qu'il doit faire (post-condition/invariant)
- Il va de plus gérer toutes les exceptions qui ne sont pas runtime.

Exception et prog. par contrat

- Habituellement, les :
  - Pré-conditions sont utilisées pour :
    - vérifier les paramètres
      NullPointerException et IllegalArgumentException
    - vérifier l'état de l'objet
      IllegalStateException
  - Post-conditions sont utilisées pour :
    - vérifier que les opérations ont bien été effectués
      assert, AssertionError
  - Invariants sont utilisées pour :
    - Vérifier que les invariants de l'algorithme sont préservés. assert, AssertionError

Exemple

```
public class Stack {
  public Stack(int capacity) {
    array=new int[capacity];
  }
  public void push(int value) {
    if (top>=array.length)
      throw new IllegalStateException("stack is full");
    array[top++]=value;
    assert array[top-1]==value;
    assert top>=0 && top<=array.length;
  }
  public int pop() {
    if (top<=0)
      throw new IllegalStateException("stack is empty");
    int value=array[--top];
    assert top>=0 && top<=array.length;
    return value;
  }
  private int top;
  private final int[] array;
}
```

Pré-condition

Post-condition

Invariant

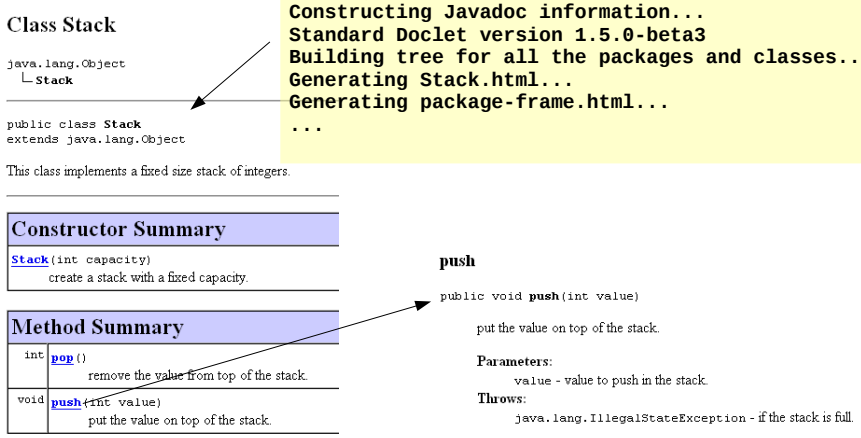
Exemple avec commentaires

Le code **doit** être commenté

```
public class Stack {
  /** This class implements a fixed size
      stack of integers.
   * @author remi
   */
  public class Stack {
    /** put the value on top of the stack.
     * @param value value to push in the stack.
     * @throws IllegalStateException if the stack
                  is full.
     */
    public void push(int value) {
      ...
    }
    /** remove the value from top of the stack.
     * @return the value on top of the stack.
     * @throws IllegalStateException if the stack
                  is empty.
     */
    public int pop() {
```


Utilisation de Javadoc

```
C:\java-avancé>javadoc src\Stack.java
Loading source file src\Stack.java...
Constructing Javadoc information...
Standard Doclet version 1.5.0-beta3
Building tree for all the packages and classes...
Generating Stack.html...
Generating package-frame.html...
...
```



Programmation par exception

Déclencher une exception pour l'attraper juste après est très rarement performant (la création du stacktrace coûte cher)

```
public class CatchExceptionExample {
  public static int sum1(int[] array) { // 5,4 ns
    int sum=0;
    for(int v:array)
      sum+=v;
    return sum;
  }
  public static int sum2(int[] array) { // 7,2 ns
    int sum=0;
    try {
      for(int i=0;;)
        sum+=array[i++];
    } catch(ArrayIndexOutOfBoundsException e) {
      return sum;
    }
  }
}
```


StrackTrace et optimisation

Pour des questions de performance, il est possible de pré-créer une exception

```
public class StackTraceExample {
  public static void f() {
    throw new RuntimeException();
  }
  public static void g() {
    if (exception==null)
      exception=new RuntimeException();
    throw exception;
  }
  private static RuntimeException exception;
  public static void main(String[] args) {
    f(); // 100 000 appels, moyenne()=3104 ns
    g(); // 100 000 appels, moyenne()=168 ns
  }
}
```

Appel de méthode

L'algorithme d'appel de méthode s'effectue en deux temps

1) A la compilation, on cherche la méthode la mieux adaptée

   On recherche les méthodes applicables
   (celles que l'on peut appeler)

   Parmi les méthodes applicables, on recherche
   s'il existe une méthode plus spécifique
   (dont les paramètres seraient sous-types des
   paramètres des autres méthodes)

2) À l'exécution, on recherche l'implémentation de cette méthode qui soit la plus précise étant donné le type réel de l'objet receveur de l'appel

Méthodes applicables

Ordre dans la recherche des méthodes applicables :

Recherche des méthodes à nombre fixe d'argument en fonction du sous-typage & convertions primitifs

Recherche des méthodes à nombre fixe d'argument en permettant l'auto-[un]boxing

Recherche des méthodes en prenant en compte les varargs

Dès qu'une des recherches trouve une ou plusieurs méthodes la recherche s'arrête

Exemple de méthodes applicables

Le compilateur cherche les méthodes applicables

```
public class Example {
  public void add(Object value) {
  }
  public void add(CharSequence value) {
  }
}
```

```
public static void main(String[] args) {
  Example example=new Example();
  for(String arg:args)
    example.add(arg);
}
```

Ici, **add(Object)** et **add(CharSequence)** sont applicables

Méthode la plus spécifique

Recherche parmi les méthodes applicables, la méthode la plus spécifique

```
public class Example {
  public void add(Object value) {
  }
  public void add(CharSequence value) {
  }
}
```

```
public static void main(String[] args) {
  Example example=new Example();
  for(String arg:args)
    example.add(arg); // appel add(CharSequence)
}
```

La méthode la plus spécifique est la méthode dont tous les paramètres sont sous-type des paramètres des autres méthodes

Méthode la plus spécifique (2)

Si aucune méthode n'est plus spécifique que les autres, il y a alors ambiguité

```
public class Example {
  public void add(Object v1, String v2) {
  }
  public void add(String v1, Object v2) {
  }
}
```

```
  public static void main(String[] args) {
    Example example=new Example();
    for(String arg:args)
      example.add(arg,arg);
    // reference to add is ambiguous, both method
    // add(Object,String) and method add(String,Object) match
}
```

Dans l'exemple, les deux méthodes **add()** sont applicables, aucune n'est plus précise

Surcharge et auto-[un]boxing

Le boxing/unboxing n'est pas prioritaire par rapport à la valeur actuelle

```
public static void main(String[] args) {
  List list=...
  int value=3;
  Integer box=value;

  list.remove(value); // appel remove(int)
  list.remove(box);   // appel remove(Object)
}
```

```
public class List {
  public void remove(Object value) {
  }
  public void remove(int index) {
  }
}
```


Surcharge et Varargs

Une méthode à nombre variable d'arguments n'est pas prioritaire par rapport à une méthode à nombre fixe d'arguments

```
public class VarargsOverloading {
  private static int min(int... array) {
  }

  private static int min(double value) {
  }

  public static void main(String[] args) {
    min(2); // appel min(double)
  }
}
```


Surcharge et Varargs (2)

Surcharge entre deux varargs :

```
public class VarargsOverloading {
  private static void add(Object... array) { }
  private static void add(String ... array) { }

  public static void main(String[] args) {
    add(args[0],args[1]); // appel add(String...)
  }
}
```

Choix entre deux varargs ayant même wrapper :

```
public class VarargsOverloading {
  private static int min(int... array) { }
  private static int min(Integer... array) {
}
  public static void main(String[] args) {
    min(2);  // reference to min is ambiguous, both method
             // min(int...) and method min(Integer...) match
  }
}
```

Late-binding

A l'exécution, on recherche **l'implémentation la plus précise possible de la méthode identifiée à la compilation**

```
public class A {
  public static void main(String[] args) {
    B b = new B();
    C c = new D();
    c.m(b);   // Affiche C.m(A)
  }
}
class B extends A { }
```

```
class C {
  public void m(A a) {
    System.out.println("C.m(A)");
  }
}
class D extends C {
  public void m(B b) {
    System.out.println("D.m(B)");
  }
}
```


Quelques mots sur java.util

- Contient de nombreuses classes « utilitaires »
  - Autour des tableaux, des collections...
- Contient également la plupart des classes, classes abstraites et interfaces sur les structures de données permettant de stocker des éléments
  - Listes
  - Ensembles
  - Tables d'associations
  - Files d'attente...

Opération sur les tableaux

- La classe **java.util.Arrays** définit des méthodes statiques de manipulation des tableaux :
  - equals(), hashCode(), toString()...
  - binarySearch(), sort(), fill()
- Pour la copie de tableau, on utilise :
  - Object.clone(),System.arraycopy(), Arrays.copyOf()

equals(), hashCode(), toString()

- Les méthodes equals(), hashCode(), toString() ne sont pas redéfinies sur les tableaux
- Arrays.equals(), Arrays.hashCode(),Arrays.toString() marchent pour Object et tous les types primitifs

```
int[] array=new int[]{2,3,5,6};
System.out.println(array);                          // [I@10b62c9
System.out.println(Arrays.toString(array));         // [2, 3, 5, 6]
System.out.println(array.hashCode());               // 17523401
System.out.println(Arrays.hashCode(array));         // 986147
int[] array2=new int[]{2,3,5,6};
System.out.println(array2.hashCode());              // 8567361
System.out.println(Arrays.hashCode(array2));        // 986147
System.out.println(array.equals(array2));           // false
System.out.println(Arrays.equals(array,array2));    // true
```

binarySearch, sort, fill

- Dichotomie (le tableau doit être trié)

```
binarySearch(byte[] a, byte key)
...
binarySearch(Object[] a, Object key)
<T> binarySearch(T[] a, T key, Comparator<? super T> c)
```

- Tri

```
sort(byte[] a)
sort(byte[] a, int fromIndex, int toIndex)
...
<T> sort(T[] a, Comparator<? super T> c)
<T> sort(T[] a, int fromIndex, int toIndex, Comparator<? super T> c)
```

- Remplissage

```
fill(boolean[] a, boolean val)
fill(boolean[] a, int fromIndex, int toIndex, boolean val)
...
fill(Object[] a, Object val)
fill(Object[] a, int fromIndex, int toIndex, Object val)
```


Ordre naturel

- Une classe peut spécifier un **ordre naturel** en implantant l'interface Comparable<T>

```
public interface Comparable<T> {
   int compareTo(T t);
}
```

- T doit être la classe spécifiant l'ordre
- Valeur de retour de **compareTo(T t)** :
  - <0 si this est **inférieur** à t
  - ==0 si this est **égal** à t
  - >0 si this est **supérieur** à t
- Exemple avec une classe **`Person`**


```
public class Person implements Comparable<Person> {
  private final String name, firstName;
  private final int year;
  public Person(String name, String firstName, int year) {
    this.name = name;
    this.firstName = firstName;
    this.year = year;
  }
  @Override
  public int compareTo(Person p) {
    int dName = name.compareTo(p.name);
    if(dName != 0)
      return dName ;
    int dFirstName = firstName.compareTo(p.firstName);
    if(dFirstName != 0)
      return dFirstName;
    return year - p.year;
  }
  @Override public boolean equals(Object o) {
    if (!(o instanceof Person))
      return false ;
    Person p=(Person)o ;
    return name.equals(p.name)
      && firstName.equals(p.firstName)
      && year == p.year;
  }
  @Override
  public String toString() {
    return "(" + name + " " + firstName + " " + year + ")";
  }
```

compareTo() et equals() doivent être « compatibles »

Cet ordre « naturel » est intrinsèque

- Il est défini « dans » la classe

```
public class Person implements Comparable<Person> {
  // ...
  @Override public boolean equals(Object o) { //...
  @Override public String toString() { //...
  @Override public int compareTo(Person p) { //...
  public static void main(String[] args) {
    Person[] array = {
      new Person("Toto","Titi",1990),
      new Person("Toto","Titi",1900),
      new Person("Toto","Abdel",1990),
      new Person("Toto","Zora",1990),
      new Person("Dudu","Titi",1950),
      new Person("Paupau","Seb",2010)};
    System.out.println(Arrays.toString(array));
    Arrays.sort(array);
    System.out.println(Arrays.toString(array));
  }
}
```

Affiche:
[(Toto Titi 1990), (Toto Titi 1900), (Toto Abdel 1990), (Toto Zora 1990), (Dudu Titi 1950), (Paupau Seb 2010)]
[(Dudu Titi 1950), (Paupau Seb 2010), (Toto Abdel 1990), (Toto Titi 1900), (Toto Titi 1990), (Toto Zora 1990)]

Comparaison externe

- L'interface java.util.Comparator permet de spécifier un ordre externe

```
public interface Comparator<T> {
   int compare(T o1, T o2);
}
```

- Un ordre externe est un ordre valable juste à un moment donné (rien de naturel et d'évident)
- La valeur de retour de **compare** suit les mêmes règles que **compareTo**
- Par exemple, pour notre classe Person,
  - L'ordre de la date de naissance
  - L'ordre des prénoms, ou tout autre ordre « moins naturel »...

Peut se décrire par une classe anonyme

- Si on a besoin de cette classe qu'à un seul endroit...

```
// ...
public static void main(String[] args) {
  Person[] array = { … };
  Arrays.sort(array);
  System.out.println(Arrays.toString(array));
  Arrays.sort(array, new Comparator<Person>() {
    @Override
    public int compare(Person o1, Person o2) {
      return o1.year - o2.year;
    }
  }) ;
  System.out.println(Arrays.toString(array));
}
}
```

Ordre naturel

Ordre de l'age de naissance

Affiche :

[(Dudu Titi 1950), (Paupau Seb 2010), (Toto Abdel 1990), (Toto Titi 1900), (Toto Titi 1990), (Toto Zora 1990)]

[(Toto Titi 1900), (Dudu Titi 1950), (Toto Abdel 1990), (Toto Titi 1990), (Toto Zora 1990), (Paupau Seb 2010)]

Comparator inverse

- Il existe deux méthodes **static** dans la classe java.util.Collections qui renvoie un comparator correspondant à :
  - L'inverse de l'ordre naturel

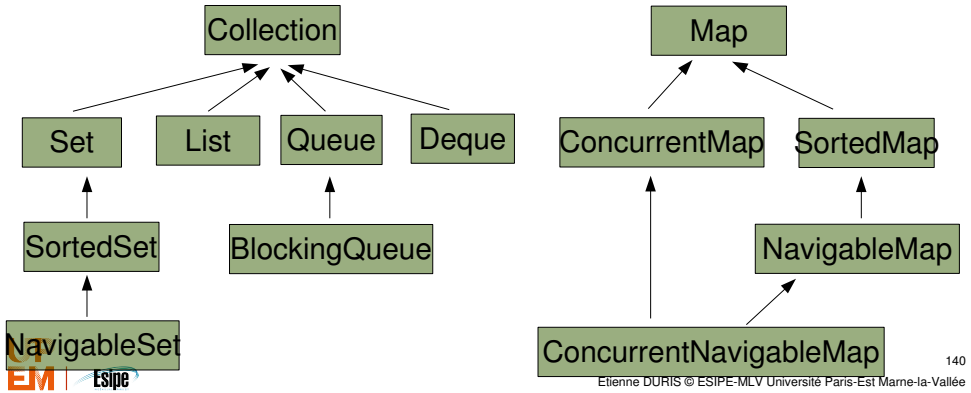
```
<T> Comparator<T> Collections.reverseOrder();
```

  - L'inverse d'un ordre externe sur **T**

```
<T> Comparator<T> Collections.reverseOrder(Comparator<T> c);
```


L'API des collections

- 2 paquetages : java.util, java.util.concurrent
- 2 hiérarchies d'interfaces : Collection, Map



Design

- Séparation Interface/implantation
- Plusieurs implantations pour une interface
  permet d'obtenir en fonction de l'algorithme que l'on veut écrire la meilleure complexité
- Deux classes contenant des algorithmes communs (méthodes statiques dans java.util.Arrays et java.util.Collections)

Interfaces des collections

- Définition abstraite des collections :
  - **Collection** ensemble de données
  - **Set** ensemble de données sans doublon
  - **SortedSet** ensemble ordonné et sans doublon
  - **NavigableSet** ensemble ordonné, sans doublon avec précédent suivant
  - **List** liste indexée **ou** séquentielle
  - **Queue** file (FIFO) de données
  - **BlockingQueue** file bloquante de données
  - **Deque** double-ended queue

Interfaces des maps

- Permet d'associer un objet (la clé) à un autre :
  - **Map** association sans relation d'ordre
  - **SortedMap** association avec clés triées
  - **NavigableMap** association avec clés triées avec suivant/précédent
  - **ConcurrentMap** association à accès concurrent

Iterator

- Pour parcourir une collection, on utilise un objet permettant de passer en revue les différents éléments de la collection
- java.util.Iterator<E> définie :
  - **boolean hasNext()** qui renvoie vrai s'il y a un suivant
  - **E next()** qui renvoie l'élément courant et décale sur l'élément suivant
  - **void remove()** qui retire un élément précédemment envoyé par next() – opération « optionnelle »

next() et NoSuchElementException

- L'opération **next()** est sécurisée et lève une exception runtime NoSuchElementException dans le cas où on dépasse la fin de la collection
  (c-a-d si **hasNext()** renvoie false)

```
public static void main(String[] args) {
  Collection<Integer> c=new ArrayList<Integer>();
  c.add(3);
  Iterator<Integer> it=c.iterator();
  it.next();
  it.next(); // NoSuchElementException
}
```


Exemple d'iterateur

- Conceptuellement un iterateur s'utilise comme un pointeur que l'on décale sur la collection

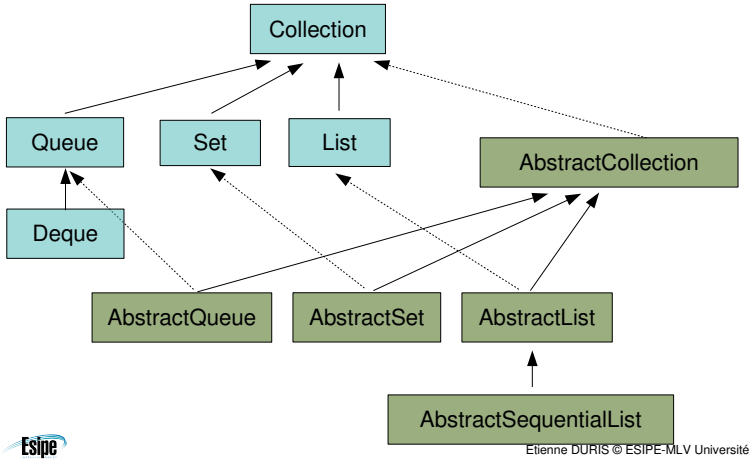
```
public static void main(String[] args) {
  Collection<Integer> c=new ArrayList<Integer>();
  c.add(3);
  c.add(2);
  c.add(4);
  Iterator<Integer> it=c.iterator();
  for(;it.hasNext();) {
    System.out.println(it.next()*2);
  } // affiche 6, 4, 8
}
```


Intérêt des itérateurs

- Pour le parcours d'une collection:
  - Pas toujours possible d'effectuer un parcours avec un index (Set, Queue, Deque)
  - Problème de complexité (List séquential)
- Les iterateurs offrent un parcours garanti en O(n)

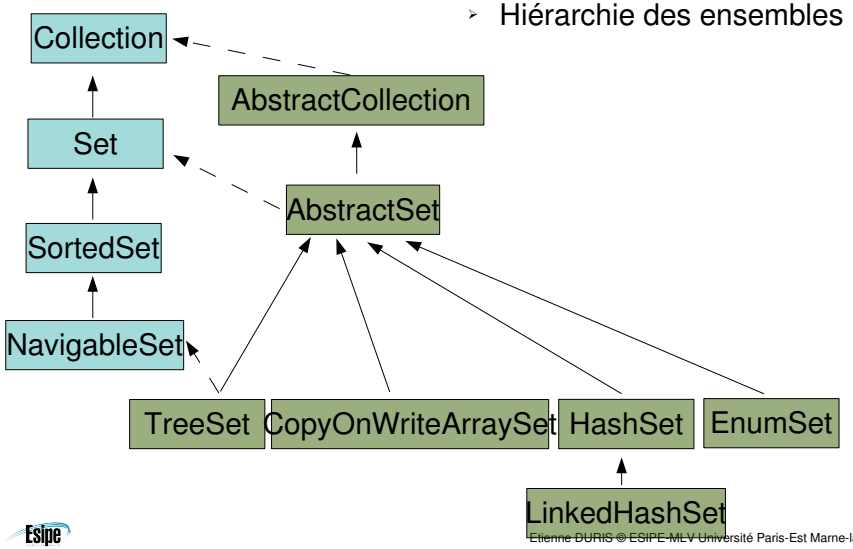
Implantation des collections

- Chaque collection possède une classe abstraite

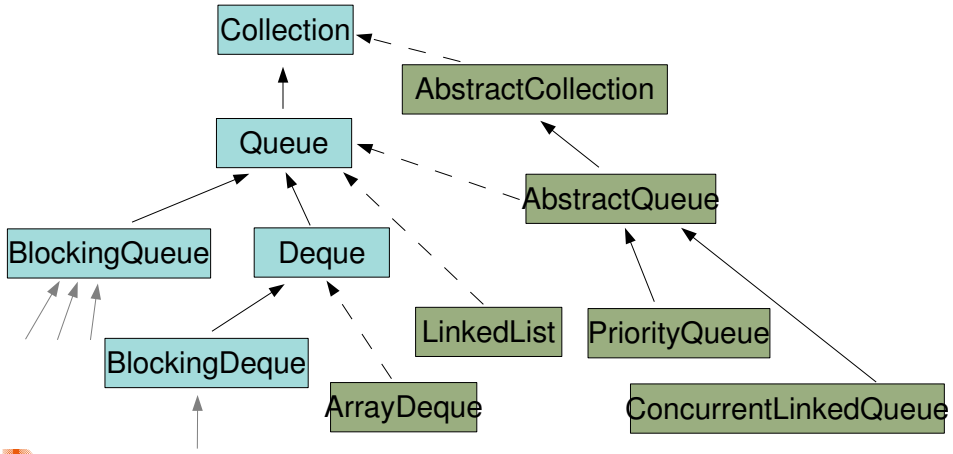


Set

- Hiérarchie des ensembles

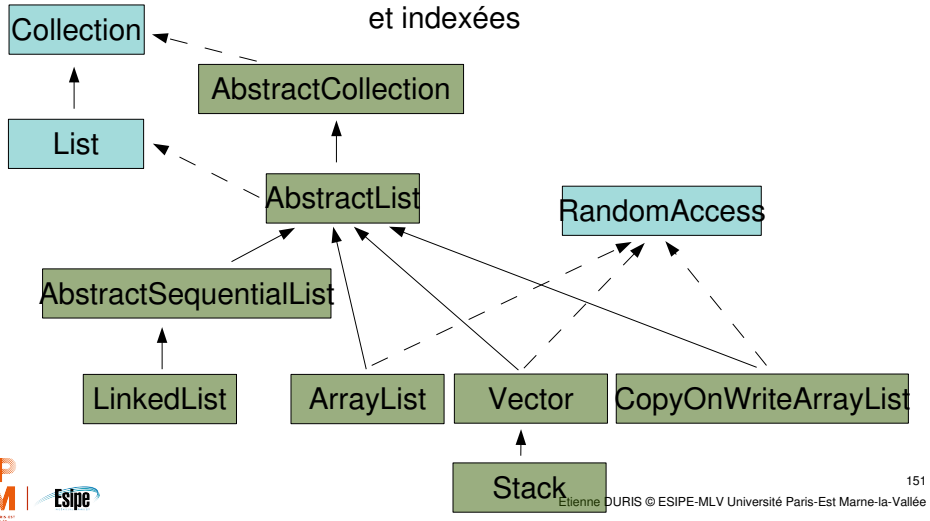


Queue & Deque



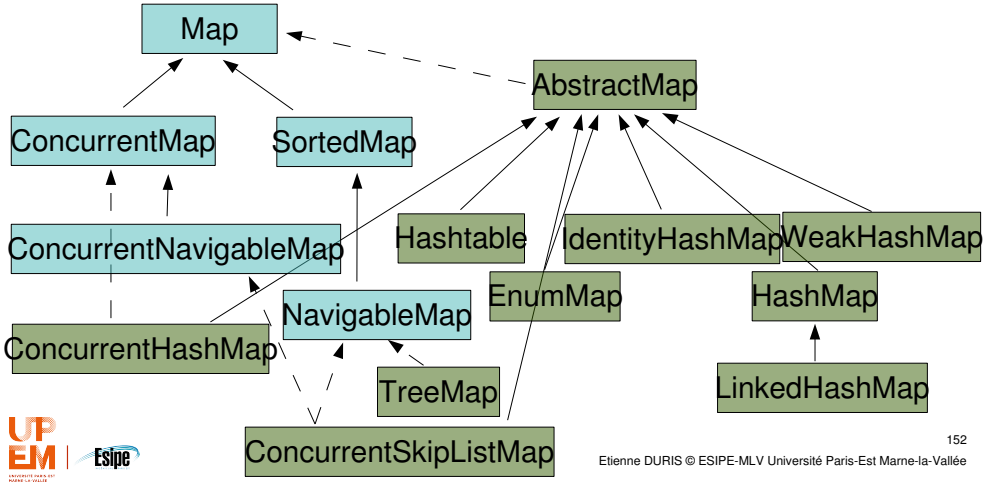
List

- Gére les listes séquentielles et indexées



Map

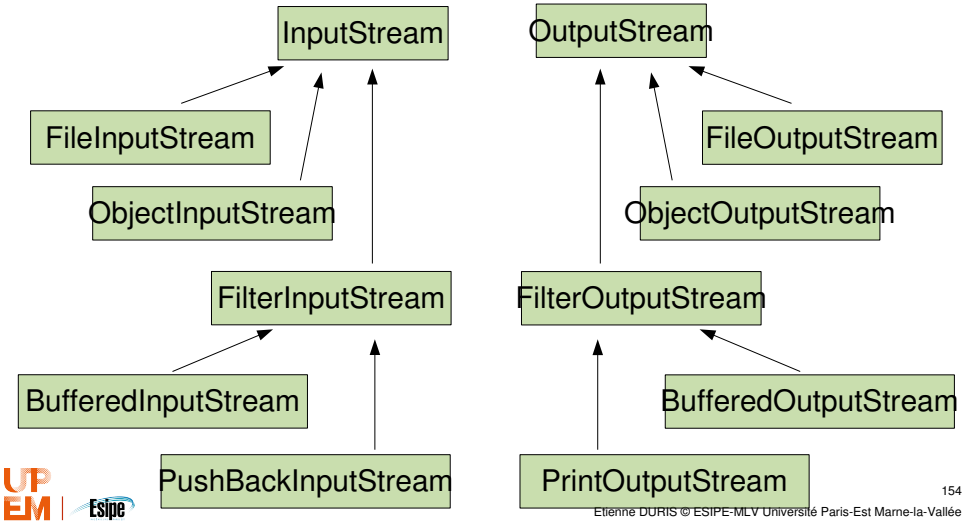
- Association entre une clé et une valeur



Quelques mots sur java.io

- **2 versions des entrées/sorties**
  - **InputStream**/**OutputStream** manipulent des octets (byte en Java) donc dépendant de la plateforme
  - **Reader**/**Writer** manipulent des caractères (char en Java) indépendant de la plateforme mais qui nécessitent d'indiquer un codage des caractères
  - On utilise l'une ou l'autre des versions en fonction de ce que l'on veut faire
    - Lire un fichier de propriété (**Reader**)
    - Ecrire un fichier binaire (**OutputStream**)

Entrées/sorties par byte



InputStream

- Flux de **byte** en entrée
  - Lit un **byte** et renvoie ce byte ou -1 si c'est la fin du flux
    - abstract int **read**()
  - Lit un tableau de byte (plus efficace)
    - int **read**(byte[] b)
    - int **read**(byte[] b, int off, int len)
  - Saute un nombre de bytes
    - long **skip**(long n)
  - Ferme le flux
    - void **close**()

InputStream et appel bloquant

- Les méthodes **read**() sur un flux sont **bloquantes** s'il n'y a pas au moins un byte à lire
- Il existe une méthode **available**() dans InputStream qui est sensée renvoyer le nombre de byte lisible sans que la lecture sur le flux soit bloquée
  mais mauvais le support au niveau des OS
  (à ne pas utiliser)

InputStream et read d'un buffer

- Attention, la lecture est une demande pour remplir le buffer, le système essaye de remplir le buffer au maximum mais peut ne pas le remplir complètement
  - Lecture dans un tableau de bytes
    - int **read**(byte[] b)
      renvoie le nombre de bytes lus
    - int **read**(byte[] b, int off, int len)
      renvoie le nombre de bytes lus

InputStream et efficacité

- Contrairement au C (stdio) par défaut en Java, les entrées sorties ne sont pas bufferisés
- Risque de gros problème de performance si on lit les données octets par octest
- Solution :
  - Lire utilisant un buffer
  - Uitliser un BufferedInputStream qui utilise un buffer intermédiaire

InputStream et IOException

- Toutes les méthodes de l'input stream peuvent lever une **IOException** pour indiquer que
  - Il y a eu une erreur d'entrée/sortie
  - Que le stream est fermé après un appel à **close**()
  - Que le thread courant a été interrompu en envoyant une **InterruptedIOException**
    (cf cours sur la concurrence)
- Il faut penser un faire un **close**() sur le stream dans ce cas

OutputStream

- Flux de byte en sortie : **OutputStream**
  - Ecrit un byte, en fait un int pour qu'il marche avec le read
    - abstract void write(int b)
  - Ecrit un tableau de byte (plus efficace)
    - void **write**(byte[] b)
    - void **write**(byte[] b, int off, int len)
  - Demande d'écrire ce qu'il y a dans le buffer
    - void **flush**()
  - Ferme le flux
    - void **close**()

Copie de flux

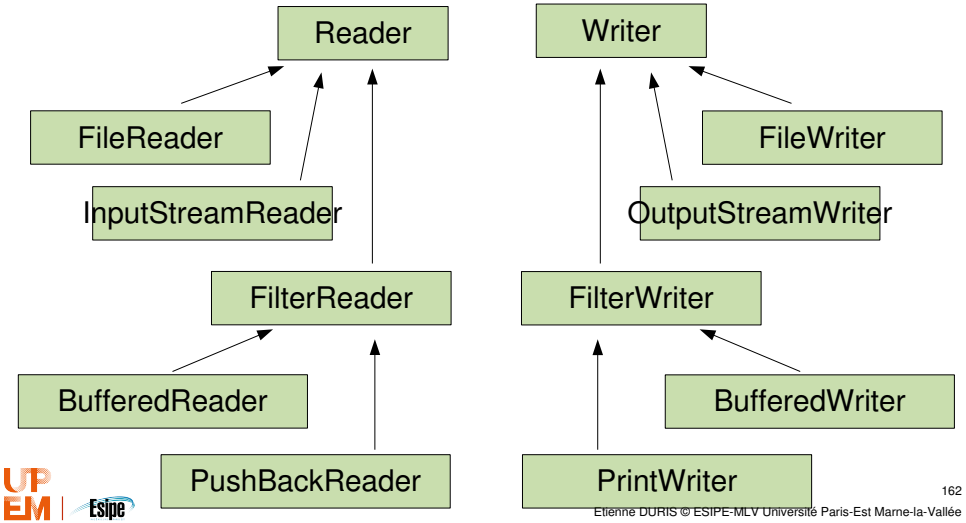
- Byte par byte (mal)

```
public static void copy(Inputstream in,OutputStream out)
  throws IOException {

  int b;
  while((b=in.read())!=-1)
    out.write(b);
}
```

- Par buffer de bytes

```
public static void copy(Inputstream in,OutputStream out)
  throws IOException {

  byte[] buffer=new byte[8192];
  int size;
  while((size=in.read(buffer))!=-1)
    out.write(buffer,0,size);
}
```


Entrées/sorties par char



Reader

- Flux de **char** en entrée (méthode bloquante)
  - Lit un char et renvoie celui-ci ou -1 si c'est la fin du flux
    - abstract int **read**()
  - Lit un tableau de char (plus efficace)
    - int **read**(char[] b)
    - int **read**(char[] b, int off, int len)
  - Saute un nombre de caractères
    - long **skip**(long n)
  - Ferme le flux
    - void **close**()

Writer

- Flux de caractère en sortie
  - Ecrit un caractère, un int pour qu'il marche avec le read
    - abstract void write(int c)
  - Ecrit un tableau de caractère (plus efficace)
    - void **write**(char[] b)
    - void **write**(char[] b, int off, int len)
  - Demande d'écrire ce qu'il y a dans le buffer
    - void **flush**()
  - Ferme le flux
    - void **close**()

Writer et chaine de caractères

- Un Writer possède des méthodes spéciales pour l'écriture de chaîne de caractères
  - Ecrire une String,
    - void **write**(String s)
    - Void **write**(String str, int off, int len)
- Un writer est un Appendable donc
  - Ecrire un CharSequence
    - Writer **append**(CharSequence csq)
    - Writer **append**(CharSequence csq, int start, int end)

Copie de flux

- Caractère par caractère (mal)

```
public static void copy(Reader in,Writer out)
  throws IOException {

  int c;
  while((c=in.read())!=-1)
    out.write(c);
}
```

- Par buffer de caractères

```
public static void copy(Reader in,Writer out)
  throws IOException {

  char[] buffer=new char[8192];
  int size;
  while((size=in.read(buffer))!=-1)
    out.write(buffer,0,size);
}
```