Java et principes SOLID

Qualité de code

IUT Clermont Auvergne

17 octobre 2023

Microcosme Java

Java en bref

- Langage créé au milieu des année 90
- Syntaxe proche du C++, simple mais un peu verbeuse
- Pas de pointeurs
- Multi-paradigmes (impératif, orienté objet, générique, quelques emprunts au fonctionnel)
- Portable car s'exécutant sur une machine virtuelle
- Langage à mémoire managée
- A évolué lentement au fil du temps

- Y a plus récent/sympa/concis/expressif : Kotlin de JetBrains
- Oui mais Java encore beauuuuucoup utilisé en entreprise

Les outils

- À l'IUT : OpenJDK version 17
- En ligne de commande :
  - `javac` : le compilateur Java
  - `java` : le lanceur de bytecode
  - `jdb` : un débogueur CLI
  - `javap` : désassembleur de bytecode
  - `javadoc` : générateur de documentation
- IDE pour ce module : IntelliJ IDEA (de JetBrains)
  - Pas juste un truc pour cacher les appels à javac/java derrière l'appui sur un bouton ▶
  - Utilisez le débogueur (points d'arrêt (jouez avec les conditions), watch sur les attributs, ...) !
  - Utilisez les outils de refactoring (refactor rename (Maj+F6), extraction de méthodes, ...) !

Le compilateur

```java
/**
 * A basic class to say hello.
 * @author IUT Clermont Auvergne
 */
public class HelloBUT {
    public static void main(String[] args) {
        System.out.println("Hello BUT 2 !");
    }
}
```

- Le compilateur Java : `javac`
  - Compile le code source Java (fichiers `.java`) en bytecode (fichiers `.class`)
    `javac HelloWorld.java`
  - Tout le code se trouve dans la classe
  - Une classe `public` par fichier (les 2 doivent avoir le m̂ nom)

Le lanceur

- Bytecode java (ouvert dans un éditeur de texte) :

```
A

java/lang/Object<init>()V


Hello BUT 2 !temoutLjava/io/PrintStream;

java/io/PrintStreamprintln(Ljava/lang/String;)
HelloBUTCodeLineNumberTablemain([Ljava/lang/String;)V
HelloBUT.java!* %
```

- Le lanceur : `java`
  - Lance une machine virtuelle Java (JVM)
  - Exécute le bytecode Java
    `java HelloWorld`

Le désassembleur (juste pour le fun)

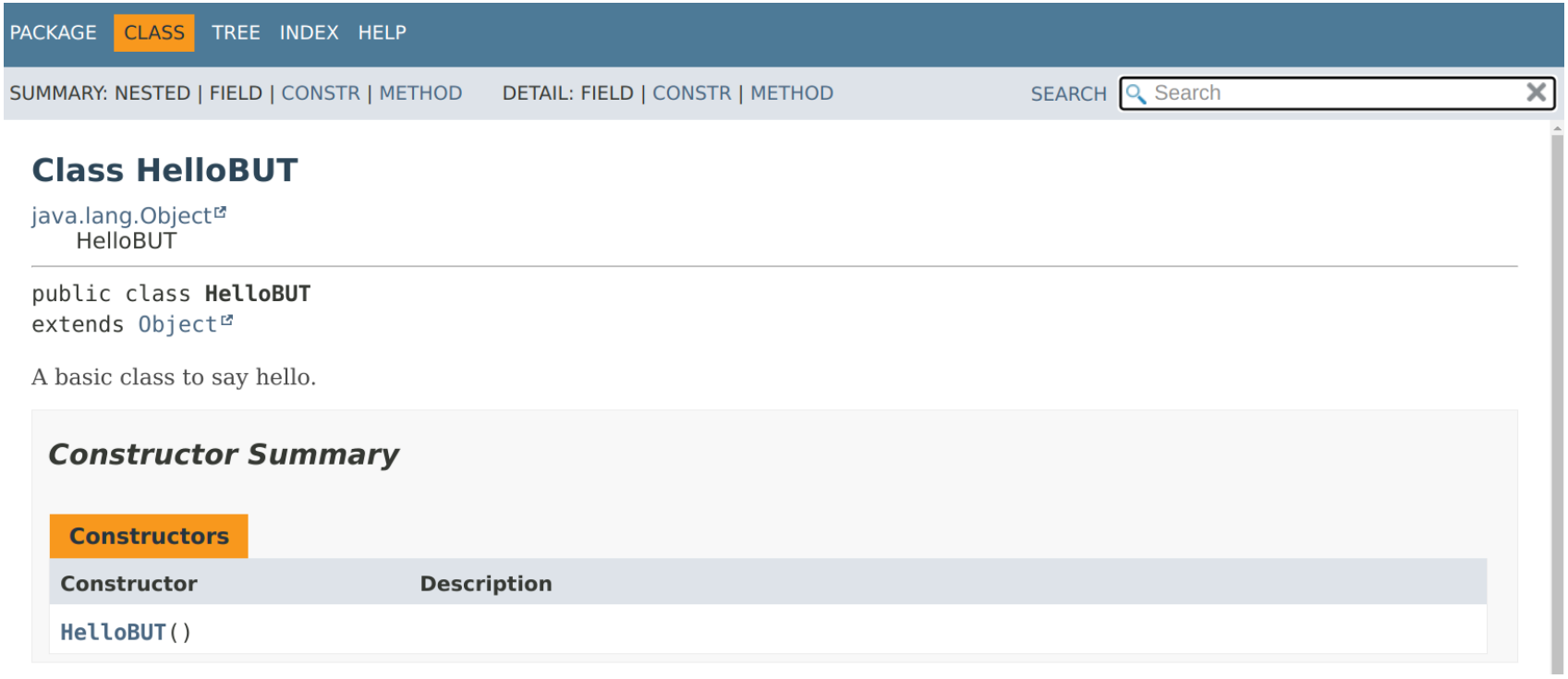
```
public class HelloBUT {
 public HelloBUT();
  Code:
   0: aload_0
   1: invokespecial #1   // Method java/lang/Object."<init>":()V
   4: return

 public static void main(java.lang.String[]);
  Code:
   0: getstatic     #7   // Field java/lang/System.out:Ljava/io/PrintStream;
   3: ldc           #13  // String Hello BUT 2 !
   5: invokevirtual #15  // Method java/io/PrintStream.println:(Ljava/lang/
                                                                String;)V
   8: return
}
```

- Le désassembleur : `javap`
  - Obtient des informations sur le bytecode Java
  - Affiche le bytecode sous forme plus lisible

    `javap -c HelloBUT.class`

Le générateur de documentation



- L'extracteur documentation : `javadoc`
  - Génère une documentation d'API au format HTML grâce aux commentaires de documentation
    `javadoc HelloWorld.java`
- Doc officielle de l'API Java sous ce même format : https://docs.oracle.com/en/java/javase/17/docs/api/index.html

Syntaxe de base du langage

Les variables

- Déclaration : `type nom;`
  - `int age;`
- Devra être initialisée avant utilisation
- Initialisation : `nom = valeur;`
  - `age = 42;`
- Définition : `int age = 42;`
- Convention de nommage camlCase
  - `float noteEnJava = 18.9f;`

Bonne pratique

Définissez vos variables quand vous en avez besoin.

- Si la valeur ne doit pas changer après affectation (immuable, `{frozen}` en UML)
  - `final int numeroAVie = 42;`

Les variables avec type inféré

- Depuis Java 10 inférence de type possible
  - À la définition avec `var`
  - Pour les variables locales uniquement
  - ```
    var num = 42L;   // num de type long
    ```
  - ```
    var list = new ArrayList<String>();
    ```
- À utiliser avec parcimonie
  - Peut rendre le code plus facile à lire
  - Peut rendre le code plus difficile à comprendre

Les types

1 Les types primitifs

- Types numériques signés
- Entiers : `byte` (8 bits), `short` (16 bits), `int` (32 bits), `long` (64 bits)
  - Littéraux de type `int` par défaut : `42`
  - Suffix `l` ou `L` pour type `long` : `1234567898765432 1L`
  - `_` pour lisibilité : `12_345_678_987_654_321L`
- Décimaux : `float` (16 bits), `double` (32 bits)
  - Littéraux de type `double` par défaut : `42.0`
  - Suffixe `f` ou `F` pour type `float` : `3.14f`, `2e-6f`
- Booléens : `boolean` (1 bit), littéraux `true` et `false`
- Caractères : `char` (16 bits, encodage UTF-16)
  - `'A'`, `'‰'` ou `'\u2031'`
- Manipulation par valeur

Les types

2 Les types objets

- Tout ce qui est défini avec le mot clé `class` (et `interface`)
- Convention de nommage PascalCase
  - `String`, `PrintStream`
- Manipulation par référence
- Instanciation avec `new`
  - `StringBuilder sb = new StringBuilder();`
- Les références peuvent avoir la valeur `null`
  - `sb = null;`
  - Donne la possibilité d'être garbage collecté
  - Appel de méthodes avec `.` : `sb.toString();`

Les types

3 Les tableaux

- Suite de taille fixe d'emplacements mémoires (même type donc)
- Déclaration : `int[] tab;`
  - Pas de mémoire allouée, devra être initialisé avant utilisation
- Initialisation : `tab = new int[3];`
  - Allocation de 3 entiers (initialisés à 0)
- Définition : `int tab = {3, 14, 42};`
  - Allocation de 3 entiers initialisés à 3, 14 et 42
- Accès/modification : opérateur `[]`
  - indexé à partir de 0
  - `int premier = tab[0]; tab[1] = 15;`
- Taille du tableau : `tab.length;`
- Manipulation par référence
- Utiliser classe `Arrays` pour opérations sur tableaux

Les chaînes de caractères

- Type `String`
- Définition : `String anonyme = "John Doe";`
- Ne pas faire `new String ("John Doe");`
- Chaînes de caractères immuables en Java
- `String nom = "Ano"; nom += "nyme";  // 3 alloc.`

Bonne pratique

Utilisez `StringBuilder` pour éviter les allocations temporaires.

- Depuis Java 15 : chaînes multi-lignes (*Text Blocks*)
  - ```
    String multiligne = """
        On conserve     le formatage
              sur plusieurs lignes (pas besoin de \\n)
     et l'indentation
    """
    ```
  - méthodes `formatted()`, `indent()`, …
    ```

Les types *wrapper*

- Chaque type primitif possède un type objet associé
- Même nom, commançant par une majuscule :
  `double -> Double`, `byte -> Byte`
- Exception pour `int -> Integer` et `char -> Character`
- Autoboxing : le compilo transforme pour vous
  - Boxing : `Integer i = 42;`
  - Unboxing : `int j = i;`
- Les valeurs de type wrappers sont immuables (alloc. si modif.)
- Possèdent des méthodes utilitaires sur le type en question
- Indispensable pour les collections (cf. plus loin)

Bonne pratique

Utilisez autant que possible les types primitifs.

Les instructions de contrôle

- `if`, `else`, `for`, `while`, `do`, `switch`, `break`, `continue`
- On ne revient pas dessus, tout comme en C

Bonne pratique

Utilisez toujours les accolades, même s'il n'y a qu'une instruction

```
if (answer == 42) {
    System.out.println("The only way");
} else {
    System.out.println("No no no no no");
    answer = 42;
}
```

Les fonctions

- Pas de fonctions (de premier ordre) en Java
- Toute code doit être dans une classe : il n'existe par conséquent que des méthodes
- La syntaxe de définition est la même qu'en C
- Convention de nommage camelCase

```
class Stuff {
  void nomDeLaFonction(int param1, String param2) {
    // code
  }
}
```

Paramètre varargs

- Possibilité de passer un nombre variable d'arguments à une méthode avec la syntaxe `Type...`
- Accessible sous forme de tableau dans le code

```
String funcVarargs(String... questions) {
    if (questions.length == 0) return null;
    String firstQuestion = questions[0];
}

// ailleurs dans le code
funcVarargs("Quelle est ?", "Avec qui ?", "Pourquoi ?"); // OK
funcVarargs("Quoi ?");      // Aussi OK
```

- Un seul paramètre peut être de type varagrs
- Il doit être en dernière position

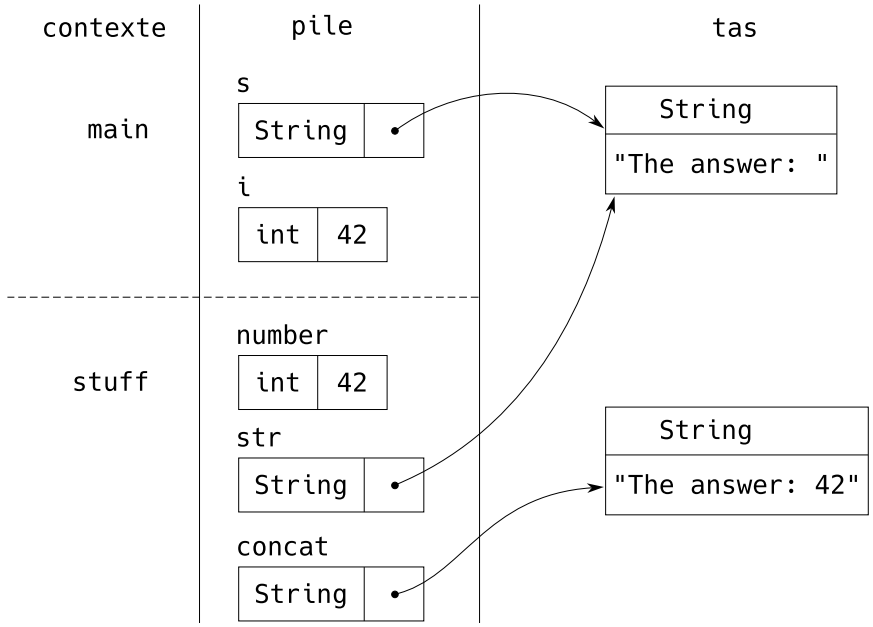
La mémoire

Gestion de la mémoire

- Toute allocation dynamique est faite sur le tas (*heap*)
- Les variables locales sont stockées sur la pile (*stack*)
  - Pour les types primitifs, la **valeur** est stockée
  - Pour les types objet, la **référence** est stockée
- Le passage de paramètres est fait par valeur en Java (on copie le contenu)
  - Type primitif : on copie la valeur (on duplique)
  - Type objet : on copie la référence (on partage donc)

```
void stuff(int number, String str) {
    String concat = str + number;
    // d'autres trucs ici
}

// quelque part dans le thread prinicipal
String s = "The answer: ";
int i = 42;
stuff(i, s);
```



L'orienté objet

Type utilisateur

- Squelette d'une classe concrète

```
public class MonObjet {
  private Type attribut;                          // attributs
  ...
  public MonObjet() {...}
  public MonObjet(...) {...}                      // constructeurs
  public MonObjet(MonObjet o) {...}
  ...
  public Type getAttribut() {...}                 // accesseurs
  public void setAttribut(Type attr) {...}
  ...
  public Type interfaceMethod(...) {...}          // API publique
  ...
  private Type maMethod(...) {...}                // bidouilles internes
  ...
  public String toString() {...}                  // méthodes utilitaires
  public boolean equals(Object obj) {...}
}
```

Les accesseurs

- Pas de notion de *propriété* en Java
- On adopte des conventions de nommage pour certaines méthodes
- Pour un attribut `type xxx;` :
  - getteur : `type getXxx() { return xxx; }`

    (ou si type booléen `boolean isXxx() { return xxx; }`)
  - un setteur :
    `void setXxx(type newXxx) { xxx = newXxx; }`

Attention

Si `type` est un type objet non immuable, le getteur expose l'attribut (qui est une référence).

Bonne pratique

Ne pas les ajouter machinalement, pensez à l'encapsulation.

Les visibilités

- Visiblilités des constituants (attributs, méthodes, classes internes, ...) d'une classe :

1. `private` : visible uniquement depuis la classe en question
2. celle par défaut, si on ne met rien (*package-private*) : visible dans la classe et toute classe du même package
3. `protected` : visible dans la classe, celles du même package et dans ses classes dérivées
4. `public` : visible partout

Bonne pratique (*encapsulation*)

N'exposez que ce qui est nécessaire. Réduisez au plus vos visibilités.

La notion de package

- Équivalent du `namespace` en C++
- Permet d'organiser un ensemble de classes proches
- Appui l'encapsulation (visibilité *package-private*)
- Permet d'éviter les collisions de noms
- Déclarer un package : `package fr.uca.iut;`
  - Doit être la première ligne du fichier
  - Convention de nommage : tout en minuscule
- Doit être reflété dans la hiérarchie du code source
- Fichiers `.java` des classes de ce package dans dossier `$SRC/fr/uca/iut/` où `$SRC` est la racine des sources de votre projet

```java
package fr.uca.iut;

public class Rectangle { … }
```

La notion de package

- Utilisation (pénible !) :

```
fr.uca.iut.Rectangle r = new fr.uca.iut.Rectangle(…);
```

- Pour éviter d'avoir à taper le nom complet :

```
import fr.uca.iut.Rectangle;
// ...
Rectangle r = new Rectangle(…);
```

- Permet aussi d'éviter les collisions de noms si 2 classes avec même nom

Bonne pratique

N'importez que les dépendances nécessaires à votre code. Évitez les `import fr.uca.iut.*` qui importent tout.

Les constructeurs

- Même nom que la classe, pas de type de retour

```
public class Point {
  private double x;
  private double y;
  …
  public Point(double x, double y) {
    this.x = x;
    this.y = y;
  }
}
```

- On peut appeler un Ctor depuis un autre grâce à `this`
- Doit être la première instruction

```
public Point(Point p) {
  this(p.x, p.y);
}
```

Les constructeurs

- On ne peut pas préciser de valeur par défaut comme en C++
- On utilise une surcharge et un appel à un autre Ctor

```
public Point() {
  this(0,0);
}
```

Bonne pratique

Utilisez le chaînage de constructeurs pour éviter la duplication de code.

- Les attributs sont par défaut initialisés à `0`, `'\u0000'`, `false`, `null` suivant leur type

Destructeur

- Il n'y a pas de destructeur en Java
- La JVM est doté d'un ramasse-miettes (*garbage collector*)
- Quand aucune référence ne pointe plus sur un objet celui-ci devient candidat à la libération de mémoire

Attributs et méthodes de classe

- Déclarés avec le mot-clé `static`
- Pas besoin d'instance de classe pour les utiliser

```java
public class Point {
  public static final double ORIGINE_AXE = 0.0;
  private double x;
  private double y;
  …
  public static Point fromPolar()(double rho, double theta) {
    return new Point(rho * Math.cos(theta),
                     rho * Math.sin(theta));
  }
}
// Dans le code
Point p = Point.fromPolar(2.0, Math.PI / 2.0);
```

- Les constantes (de classe) : `static` + `final`
  - Type primitif ou `String`
  - Convention de nommage UPPER_SNAKE_CASE

Classes de données

- Depuis Java 16, moins de boilerplate code pour les classes de données immuables grâce aux *records*

```java
public record Personne (String nom, int age) {}
```

- Génère automatiquement (impliciement)
  - le Ctor associé aux paramètres
  - les getters associés aux paramètres
  - le `equals()` et le `hashCode()` (cf. collections dans la suite)
  - le `toString()` (ex. `Personne[nom=John Doe, age=42]`)
- Pratique pour stocker des données (POJO)
- Tout est immuable
- Pas d'héritage possible

Héritage

- L'héritage s'exprime en Java grâce au mot-clé `extends`

```
class Derivee extends Base {
  …
}
```

- Tous les constituants de `Base` déclarés `public` ou `protected` sont accessibles dans `Derivee`
- L'héritage multiple n'est pas possible
- Toute classe dérive ultimement de la classe `Object` (même si non précisé)
- Une classe estampillée `final` ne peut pas être dérivée

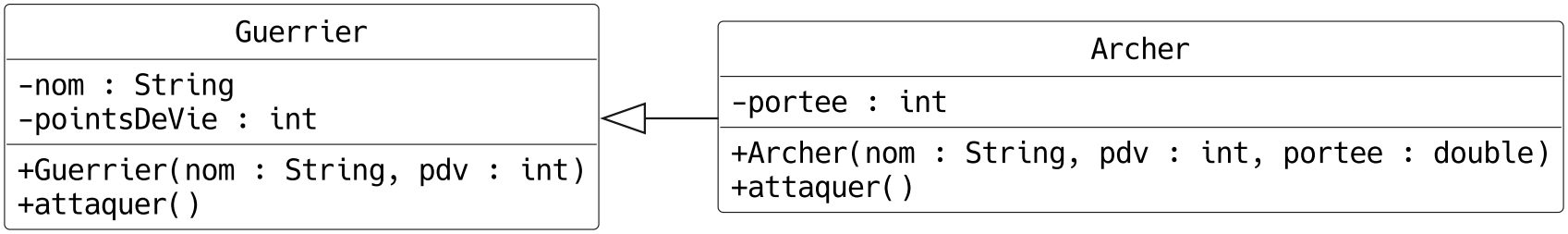
La classe `Object`

Principales méthodes de la classe `Object` :

- `public final Class<?> getClass()`
  renvoie l'instance de la méta-classe représentant l'objet implicite (utile pour l'introspection)
- `public boolean equals(Object obj)`
  teste l'égalité (en terme de contenu) de l'objet implicite avec l'objet passé en paramètre (`this == obj` par défaut)
- `public int hashCode()`
  renvoie une valeur de hachage relative à l'objet (utile pour les conteneurs basés sur des tables de hachage)
- `public String toString()`
  renvoie une chaîne de caractères représentant l'objet (utile pour le débogage) (nom_classe + @ + hashcode_hexa par défaut)

Constructeur de classe dérivée

- Les constructeurs ne sont pas hérités
- Appel au constructeur de la classe de base : mot-clé `super`




```
public class Archer extends Guerrier {
  private int porteeAttaque ;
  public Archer (String nom, int pdv, int portee) {
    super (nom, pdv);
    porteeAttaque = portee ;
  }
}
```

- `super(…)` : doit être la première instruction du constructeur

Masquage

- Même nom d'attribut dans `Base` et `Derivee` : la nouvelle définition *masque* la définition de base

```
public class Base {
  protected int x = 42 ;
}
public class Derivee extends Base {
  protected double x = 3.14 ;
}
public class EncoreDerivee extends Derivee {
  protected char x = 'A' ;
}
```

Depuis la classe `EncoreDerivee` :

- `x` ou `this.x` vaut `'A'`
- `super.x` ou `((Derivee) this).x` vaut `3.14`
- ~~`super.super.x`~~ `((Base) this).x` vaut `42`

Polymorphisme

- Instance dérivée manipulable avec référence type de base
  `Guerrier robin = new Archer("Robin", 10, 100);`
- Par défaut les méthodes sont virtuelles en Java
- Le *polymorphisme* s'applique naturellement sans rien faire
- `robin.attaquer();` appelle la méthode de la classe `Archer`

Redéfinition

- Méthode de `Base` doit être visible dans `Derivee`
- Méthode de même signature dans `Derivee` que dans `Base`

```java
public class Base {
  public void faireUnTruc(int i) {}
}
public class Derivee1 extends Base {
  public void faireUnTruc(int i) {}
}
public class Derivee2 extends Base {
  public void faireUnTruc(float i) {}
}
```

```java
Base bd1 = new Derivee1();
Base bd2 = new Derivee2();

// appel faireUnTruc de Derivee1
bd1.faireUnTruc(42);

// appel faireUnTruc de Base
bd2.faireUnTruc(42);
```

Bonne pratique

Estampillez dans `Derivee` la méthode à redéfinir grâce à l'annotation `@Override` (permet au compilateur de vérifier que c'est bien une redéfinition).

Redéfinition

- Les méthodes de classe ne peuvent pas être redéfinies
- Les méthodes déclarées `final` ne peuvent pas être redéfinies
- Utiliser le mot-clé `super` pour appeler la méthode de la classe de base

```
public class Button {
  …
  void paint (Graphics g) {
    g.drawRect(0, 0, this.width, this.height);
  }
}
public class ButtonIcon extends Button {
  …
  @Override
  void paint (Graphics g) {
    super.paint (g); // dessin du contour du bouton
    g.drawImage (2, 2, this.icon);
  }
}
```

Méthodes et classes abstraites

- Mot-clé `abstract`
- En plus pour les méthodes : pas de corps
- Une classe avec une méthode abstraite doit être absraite
- Une classe abstraite ne peut pas être explicitement instanciée avec `new`

```
public abstract class Forme {      // Classe abstraite
  private double x;
  private double y;
  …
  public void deplacer(double dx , double dy) {
    x += dx;
    y += dy;
  }
  public abstract perimetre();     // Méthode abstraite
}
```

Interfaces

- Pas d'héritage multiple en Java (évite pb héritage en diamant)
- On passe par la notion d'interface : entité purement abstraite
- Pas d'attributs d'instance (que des constantes)
- Méthodes sans corps (implicitement `abstract` et `public`)
- Mot-clé `interface`

```java
interface Attaquant {
  void attaquer (Unite ennemi);
}
```

- Les classes peuvent réaliser une ou plusieurs interfaces : mot-clé `implements`

```java
public class Guerrier implements Attaquant {
  @Override
  void attaquer (Guerrier ennemi) { /* code */ }
}
```

Interfaces

- Une interface peut étendre une autre interface : mot-clé `extends`
- Depuis Java 8
  - Méthodes de classe autorisées (ne participent pas à l'API de la classe qui l'implémente)
  - Méthodes avec implémentation par défaut (mot-clé `default`) autorisées (pratique pour factoriser un code abstrait)

```
interface Assembleur {
  default Voiture assembler() {
    ajouterCarrosserie();
    ajouterRoues();
  }
  void ajouterCarrosserie();
  void ajouterRoues();
  void ajouterOptions();
}
```

Transtypage descendant

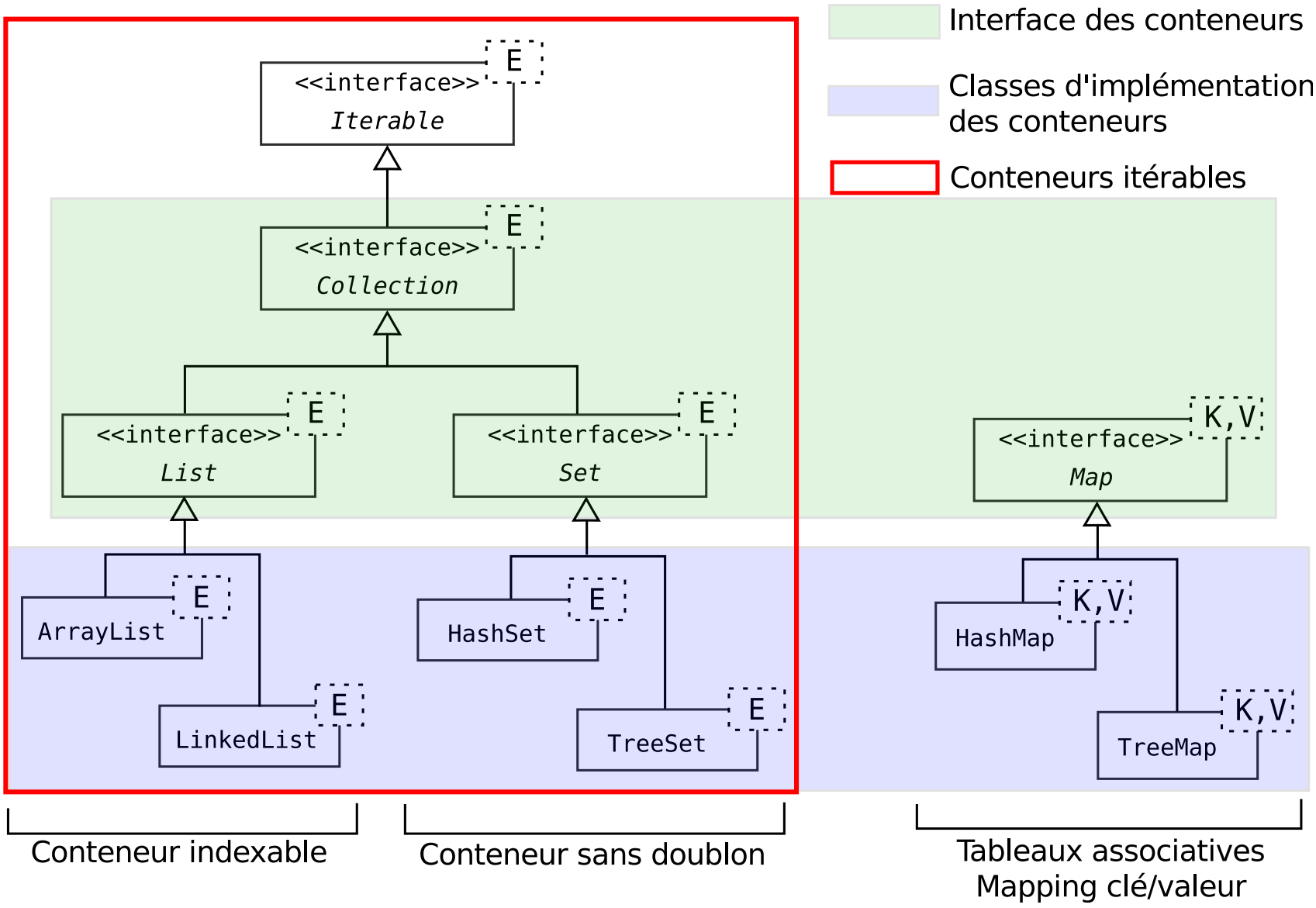
- Instance dérivée manipulable avec référence type de base `Guerrier robin = new Archer("Robin", 10, 100);`
- On peut vouloir utiliser une spécificité de `Archer` qui n'est pas dans `Guerrier` et on n'a que `robin` sous la main
- On *downcast* : `Archer vraiRobin = (Archer) robin;`
- Opération non sûre si en réalité `robin` pas de type `Archer` (`ClassCastException`)
- On peut vérifier dynamiquement le type : `intanceof`

```java
if (robin instanceof Archer) {
  Archer rob = (Archer) robin;
  rob.utiliserArc();
}
```

```java
// Ou bien, depuis Java 17
if (robin instanceof Archer rob) {
  rob.utiliserArc();
}
```

Les collections

Framework Collection



Généricité

- Le framework Collection repose fortement sur la généricité
- Un peu différent des templates en C++
- Utilisé à compilation pour vérifications de types
- À l'exécution tout devient `Object` (*type erasure*)
- Les types génériques en Java ne peuvent donc pas être remplacés par des types primitifs
- `ArrayList<int>` interdit, `ArrayList<Integer>` OK (d'où les types wrapper)

Invitation à l'exploration

Pas le temps dans ce petit cours, mais créer ses propres méthodes ou classes génériques peut quelquefois éviter l'écriture de code très semblable. N'hésitez pas à aller regarder tout ça par vous même et poser vos questions.

Utiliser un type objet dans une collection

- Une classe doit respecter certains protocoles pour être utilisable correctement dans des collections

1. Protocole d'égalité (pour pouvoir utiliser `contains()` ou `remove()` par exemple)

- Redéfinir la méthode `boolean equals(Object o)`
- Faire en sorte de comparer les contenus
- `equals()` doit également vérifier (relation d'équivalence) :
  - `o1.equals(o1) == true`
  - `o1.equals(o2) == o2.equals(o1)`
  - `o1.equals(o2)` et `o2.equals(o3)` $\Rightarrow$ `o1.equals(o3)`

Protocole d'égalité : exemple

```java
public class Personne {
  private int age;
  private String nom;
  …
  public boolean equals(Object o) {
    if(o == null) return false;
    if(this == o) return true;
    if (getClass() != o.getClass()) return false;
    Personnage other = (Personnage) o;
    return nom.equals(other.nom) && age == other.age;
  }
}
```

Utiliser un type objet dans une collection

2. Protocole de hashage (nécessaire pour collections à base de tables de hashage, par exemple `HashSet` ou `HashMap`)

- Redéfinir la méthode `int hashCode()`
- Transforme l'instance en un nombre
- Doit être cohérent avec `equals()` (mêmes attributs)

  `o1.equals(o2)` $\Rightarrow$ `o1.hashCode() == o2.hashCode()`

```java
public class Personne {
  private int age;
  private String nom;
  …
  public int hashCode(){
    int result = age;
    if(nom != null) result = 31 * result + nom.hashCode();
    return result;
  }
}
```

Utiliser un type objet dans une collection

3. Protocole de comparaison (nécessaires aux collections triées, par exemple `TreeSet` ou `TreeMap`)

- Pour une classe `XXX` implémenter l'interface `Comparable<XXX>`
- Redéfinir la méthode `int compareTo(XXX x)` qui retourne un nombre :
  - négatif si `this` est inférieur à `x`
  - nul si `this` et `x` sont égaux
  - positif si `this` est supérieur à `x`
- Ça définit une relation d'ordre canonique sur le type `XXX`
- Attention à rendre `compareTo()` et `equals()` cohérentes entre-elles

Protocole de comparaison : exemple

```
public class Personne implements Comparable<Personne> {
  private int age;
  private String nom;
  …
  public int compareTo(Personnage pers) {
     int compNom = nom.compareTo(pers.nom);
     if (compNom != 0) return compNom;
     return age - pers.age;
  }
}
```

Parcours d'une collection

- Pour les collections indexables (`List`)

```
List<Personne> liste = Arrays.asList(
  new Personne("Pierre"),
  new Personne("Paul"),
  new Personne("Jacques")
);
```

1. Utilisation d'un `for` classique

```
for (int i = 0; i < liste.size(); i++) {
   System.out.println(liste.get(i).getNom());
}
```

- Pas très passe-partout

Parcours d'une collection

2. Pour les collections itérables... utiliser un itérateur
   - Type `Iterator<T>`
   - Méthode `Iterator<T> iterator()` sur la collection pour obtenir le point de départ
   - Méthode `boolean hasNext()` sur l'itérateur pour savoir si on peut continuer
   - Méthode `T next()` pour obtenir l'élément courant et avancer

```
Iterator<Personne> it = liste.iterator();
while(it.hasNext()) {
    System.out.println(it.next().getNom());
}
```

   - Suppression possible du dernier élément retourné avec `remove()`

Parcours d'une collection

- Syntaxe un peu pénible avec les itérateurs

3. Quand on veut parcourir toute la collection : for de type *for each*

```
for(Personne pers : liste) {
    System.out.println(pers.getNom());
}
```

4. Grâce à la méthode `forEach()` de l'interface `Iterable<T>`

```
liste.forEach(p -> System.out.println(p.getNom()));
```

- Dans ces 2 cas on perd la possibilité de modifier la collection pendant le parcours

Transformations de collection

- Depuis Java 8, *fluent API* pour traiter les collections
- Repose sur la notion de `Stream` (séquentiel et parallèle)
- Application paresseuse de fonctions (*lazy computation*)

```
List<String> persos = Arrays.asList("Mario", "Luigi", "Peach",
                                    "Toad", "Yoshi");
int somme = persos.stream().filter(m -> m.contains('i'))
                           .mapToInt(String::length)
                           .sum();              // somme = 15
```

- Repose fortement sur l'utilisation de la généricité, des interfaces fonctionnelles et des lambdas

```
Stream<T> filter(Predicate<? super T> predicate)
```

Les lambdas

- Objectif : passer des « fonctions » en paramètre pour appliquer des actions

1. Sans lambdas : objets anonymes

- On crée une instance locale d'une classe anonyme qui implémente l'interface désirée

```
Predicate<String> contientI = new Predicate<>() {
  @Override
  public boolean test(String s) {
    return s.contains("i");
  }
};
persos.stream().filter(contientI).forEach(System.out::println);
```

- Pénible !

Les lambdas

2. Avec lambas : sucre syntaxique qui fait tout ça pour vous

```
persos.stream().filter(m -> m.contains("i"))
                .forEach(System.out::println);
```

- Repose sur les interfaces fonctionnelles (*SAM*)
- Interfaces qui ne possèdent qu'une seule méthode abstraite
- Annotées avec `@FunctionalInterface`
- Permet aussi d'utiliser à la place des références de méthodes (opérateur `::`)... il faut évidemment que les prototypes correspondent

Entrées/sorties

- Sortie standard/erreur : `System.out` et `System.err` (type `PrintStream`)
- Écriture en utilisant les méthodes de la classe `PrintStream`
- Entrée standard : `System.in` (type `InputStream`)
- Lecture en utilisant la classe utilitaire `Scanner`
- `Scanner` fonctionne un peu comme un itérateur

```
Scanner clavier = new Scanner(System.in);
List<Integer> entiers = new ArrayList<>();
List<String> mots = new ArrayList<>();
while (clavier.hasNext()) {
  if (clavier.hasNextInt()) {
    entiers.add(clavier.nextInt());
  } else {
    mots.add(clavier.next());
  }
}
```

Lecture/écriture dans un fichier

- Nombreuses classes disponibles suivant façon de lire
  - Texte ou binaire
  - Octet par octet ou données typées
  - Bufferisé ou pas
- Ressource ⇒ peut générer des erreurs ⇒ gestion d'exceptions

```java
// Utilisation d'un try-with-resource (depuis Java 7)
try (BufferedReader in = new BufferedReader(
                            new FileReader("donnees.txt"))) {
    String s;
    StringBuilder sb = new StringBuilder();
    while ((s = in.readLine()) != null) {
        sb.append(s);
        sb.append("\n");
    }
    System.out.println(sb);
} catch (IOException e) {
    // Gestion erreur
}
```

Sérialisation

- Possibilité de sauvegarder des instances d'objets en laissant Java gérer le processus pour vous
- Les classes doivent implémenter l'interface `Serializable` et leurs attributs doivent être sérialisables
- La lecture/écriture se fait grâce à `ObjectInputStream` / `ObjectOutputStream`

```java
public record Personne (String nom, int age) implements Serializable {}

// dans le code
Personne john = new Personne("John Doe", 42);
try (ObjectOutputStream out = new ObjectOutputStream(
                              new FileOutputStream("data.ser"))) {
    out.writeObject(john);
} catch (IOException e) {
    // Gestion erreur
}
```