

Java

Pier Donini (pier.donini@heig-vd.ch)

- Contributions
 - René Rentsch
 - Alexandre Duc
 - Bertil Chapuis
 - Marcel Graf
 - Grégoire Krahenbühl

Index

■	Introduction	3	■	Classes abstraites	119
■	Machine virtuelle	7	■	Interfaces	123
■	Types et références	13	■	Types énumérés	137
■	Classes	19	■	Systèmes de types	143
■	Tableaux	34	■	Redéfinition	156
■	Exceptions (aperçu)	40	■	Surcharge	159
■	Héritage	42	■	Égalité d'objets	162
■	Paquetages et visibilités	51	■	Records	175
■	Polymorphisme	62	■	Copie d'objets	179
■	Autoboxing/Unboxing	78	■	Classes internes	186
■	Collections	83	■	Événements	207
■	Généricité	96	■	Expressions lambda et streams	214
■	Mot clef <code>final</code>	104	■	Exceptions et erreurs	219
■	Mot clef <code>static</code>	107	■	Suppression des objets	229
■	Blocs et initialisations	114	■	Bonnes pratiques	236

Bibliographie

■ Oracle

- oracle.com/java
- Tutoriel : docs.oracle.com/javase/tutorial
- Java SE 21 documentation : docs.oracle.com/en/java/javase/21
- Bibliothèque des classes : docs.oracle.com/en/java/javase/21/docs/api

■ Livres Java

- Core Java Volume I – Fundamentals (13th edition), Cay S. Horstmann, Prentice Hall.
- Core Java Volume II – Advanced Features (13th edition), Cay S. Horstmann, Prentice Hall.
- Effective Java, (3rd edition), Joshua Bloch, Addison-Wesley Professional
- Java in a nutshell, (8th edition), David Flanagan, Ben Evans, O'Reilly.

Historique

- SUN Microsystems cherche un langage pour ses systèmes embarqués.
 - Nom préliminaire **Oak**, puis **Java**.
- Mêmes exigences qu'Internet :
 - fiabilité, sécurité ;
 - indépendance vis-à-vis du processeur ;
 - interactions avec l'extérieur ;
 - récupération d'erreurs.
- A dépassé le cadre spécifique d'Internet (applets) et est devenu un standard pour le développement d'applications professionnelles (entreprise, web, mobiles [Android]...).

Versions Java (2)

- Langage en constante évolution, versions LTS :
 - 5 ('04) extension majeure, intégration de concepts qui faisaient défaut (généricité, types énumérés, `foreach`...) ;
 - 8 ('14) extension majeure (méthodes par défaut, lambda expressions) ;
 - 11 ('18) extension mineure (inférence de type `var`, disparition des applets) ;
 - 17 ('21) extension mineure (blocs de texte, expressions `switch`, pattern matching `instanceof`, `record`, `sealed` classes) ;
 - 21 ('23) extension mineure (`record` patterns, pattern matching for `switch`, virtual threads, sequenced collections).
- Dans ce cours, **version 21**.

Java: aspects généraux

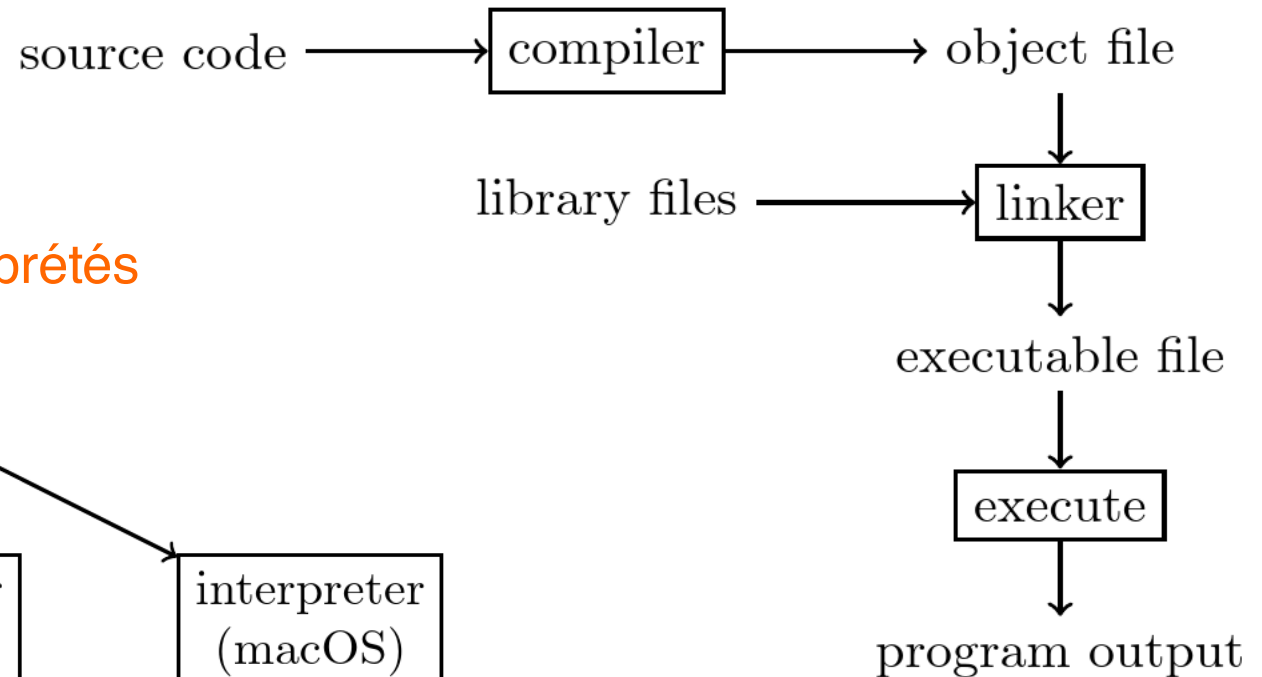
- **Multi plateforme** (génération de byte code), machine virtuelle.
- Syntaxe proche de C++ (un standard).
- Presque **tout objet** (sauf types primitifs : `int`, `double`, `boolean`...).
- **Robuste** (typage fort, pas de pointeurs, **ramasse-miettes**, encapsulation).
- Sûr (gestion des exceptions).
- Structure par paquetages (modules).
- **Bibliothèque** importante de classes (API).
- Gestion graphique intégrée et portable (swing, Java FX).
- Processus (threads).
- Dynamique (lier dynamiquement du code à l'exécution).
- Réparti : développement d'applications réparties, fournit des classes et paquetages de communication.

Langages compilés et interprétés

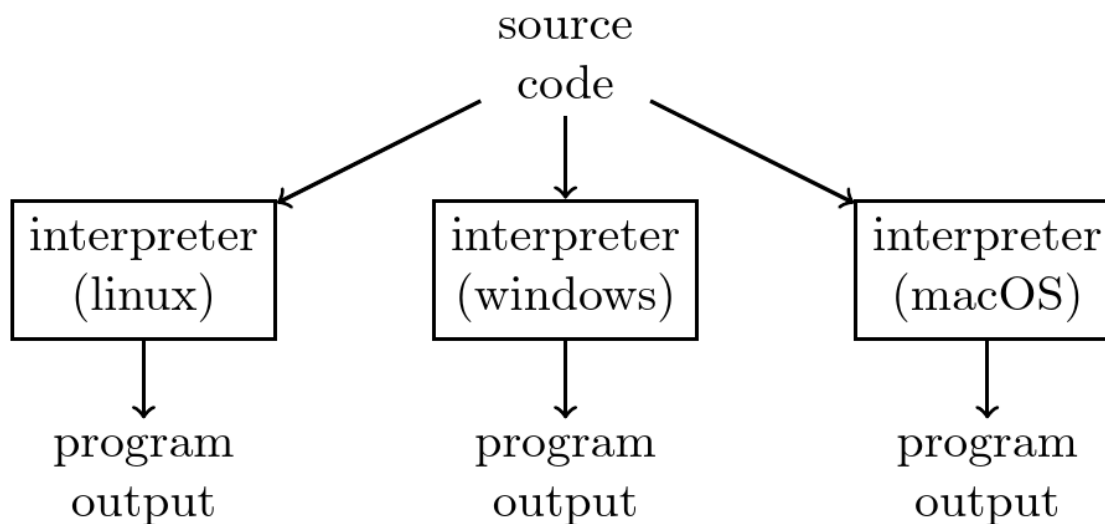
- La conception d'un langage de programmation implique des compromis.
- Les **langages compilés** sont traduits en code machine **avant** l'exécution par le compilateur.
 - Ils favorisent la performance au détriment des fonctionnalités et de la productivité (ex : C, C++).
- Les **langages interprétés** sont traduits dynamiquement **pendant** l'exécution par l'interpréteur.
 - Ils favorisent les fonctionnalités et la productivité au détriment de la performance (ex : JavaScript, Python).

Langages compilés et interprétés (2)

Langages compilés



Langages interprétés



Langages compilés et interprétés (3)

■ Langages compilés (+/-)

- (Souvent) plus rapide que les langages interprétés.
- Performances prédictibles.
- Contrôle fin du hardware.
- Mémoire difficile à gérer.
- Programmes difficiles à sécuriser (unsafe).
- Programmes difficiles à porter
- Etc.

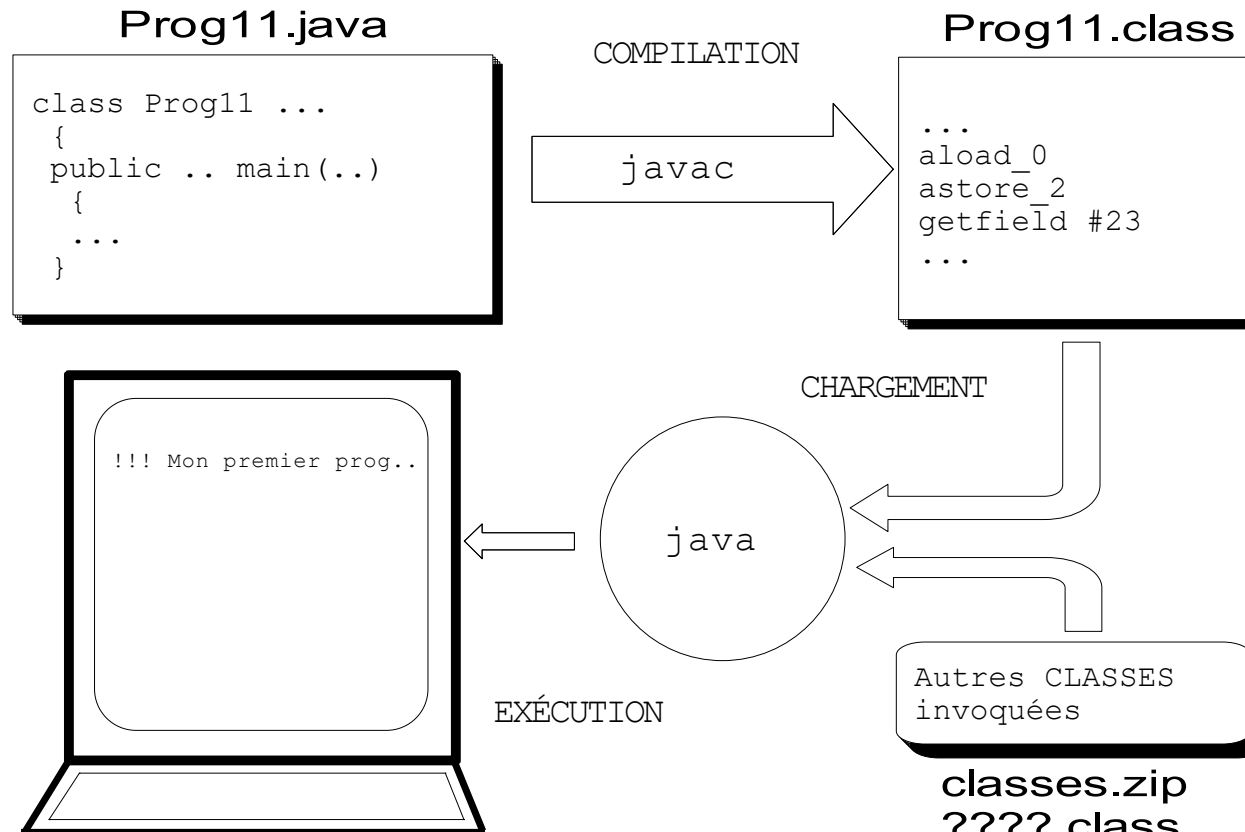
■ Langages interprétés (+/-)

- (Souvent) plus lents que langages compilés.
- Contrôle limité du hardware.
- Fonctionnalités avancées (eval).
- Programmes faciles à porter.
- Idéal pour le prototypage.
- Risque d'injection de code.
- Etc.

Machine virtuelle Java

- La Machine Virtuelle Java (JVM) combine le meilleur des deux mondes.
- Le langage (Java, Scala, Kotlin, etc.) est compilé en une représentation intermédiaire : le **Bytecode**.
- Le Bytecode est **interprété** par la machine virtuelle Java.
- La machine virtuelle java identifie les portions *chaudes* du programme et compile **à la volée** le bytecode en code machine (*Just-in-time*) à l'exécution.
 - La machine virtuelle Java connaît les spécificités de l'environnement d'exécution.
 - Le profil d'exécution peut être utilisé pour mettre en œuvre différentes stratégies d'optimisation (*inlining*, etc.).
- Plusieurs JVMs : Oracle JDK (propriétaire), Open JDK (open source)...

Java : compilation et exécution



Machine Virtuelle Java (+/-)

- Portable (*Write once run anywhere*).
- Optimisé dynamiquement à l'exécution (*Just-In-Time compilation*).
- Performance difficilement prédictible.
- Temps de chauffe (*warmup, class loading*).
- Ramasse-miettes (*garbage collector*).
- Plus grande utilisation mémoire (*garbage collector*).
- Sécurité de la mémoire (*memory safety*).
- Etc.

Types primitifs

- **Entiers**, signés seulement :
`byte` (8 bits), `short` (16 bits), `int` (32 bits), `long` (64 bits).
- **Réels** : `float` (32 bits), `double` (64 bits).
- **Booléen** : `boolean` (`true`, `false`).
- **Caractère** : `char` (16 bits, Unicode).
- Attention : `String` (chaîne de caractères) est une **classe**, pas un type primitif.
- Type par défaut des littéraux :
 - un littéral entier est du type `int` (p. ex. `3`) ;
 - lors d'une affectation dans une variable de type `byte`, `short` ou `char` (et `long`), conversion implicite d'un littéral entier si sa valeur est compatible ;
 - un littéral flottant est du type `double` (p. ex. `3.14` ou `3.14E2`).

Variables de type primitif

■ Déclaration :

- `type nomVariable;`
- `type nom1, nom2, nom3;`
- `type nom = valeur; // affectation`

■ Une variable peut être déclarée n'importe où dans une méthode avant son utilisation (comme en C++ mais pas dans les vieilles versions de C [C99-]).

■ Exemples :

- `int i;`
- `int j = 1966, k;`
- `boolean test = true;`

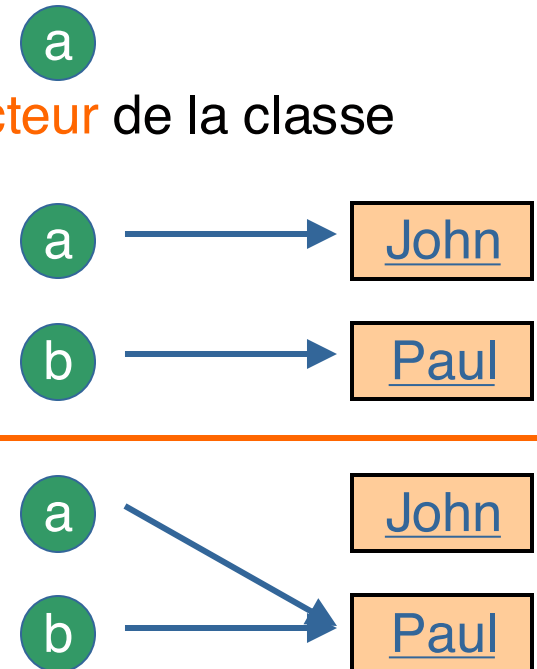
■ Pas de prise d'adresse de variable (opérateur `&` en C/C++).

Classes enveloppes (wrapper)

- A chaque type primitif correspond sa contrepartie en classe : `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Boolean` et `Character`.
 - Intérêt : utilisable partout ou peut l'être un objet + méthodes spécifiques. Ces types sont fréquemment rencontrés dans des tableaux et des collections d'objets.
 - Une fois l'objet créé, son état est **immutable** (non modifiable – pas de méthode `set`).
- Java permet la conversion implicite d'une variable de type primitif en sa contrepartie objet (*Autoboxing/Unboxing*).

Références

- En Java, tous les objets sont manipulés au moyen de références typées.
- Déclaration de variables référençant des objets : `NomClasse nomVariable;`
 - `Personne a;`
- Création d'une instance par l'invocation d'un **constructeur** de la classe (méthode permettant d'initialiser l'état d'un objet) :
 - `a = new Personne("John", ...);`
// appel d'un constructeur de la classe `Personne`
 - ou en une seule fois (déclaration + création),
`Personne b = new Personne("Paul", ...);`
- Assignment :
 - `a = b;`
 - `a` référence alors le **même objet** que `b`
(et la référence sur l'objet précédemment référencé par `a` est perdue).
- L'opérateur `==` compare l'**identité des objets référencés**, pas leurs états.



Chaînes de caractères

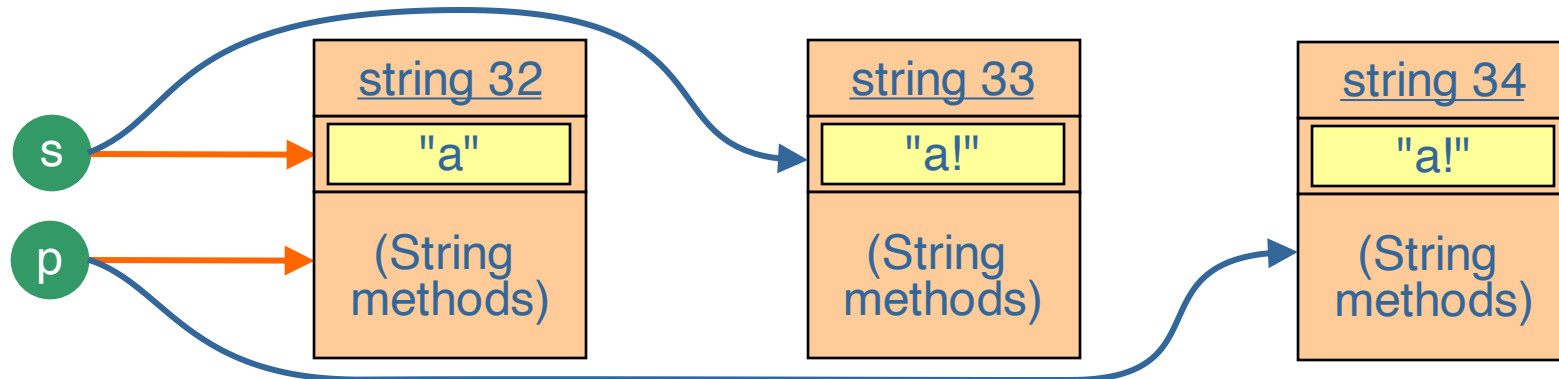
- La déclaration d'instances de la classe `String` peut être simplifiée :
 - au lieu de les construire au moyen de tableaux de caractères,

```
char[] data = {'D', 'e', ' ', 'b', 'e', 'l', 'l', 'e', 's'};  
String s = new String(data); // ne jamais appeler new String(),
```
 - simplement,

```
String s = "De belles"; // s = ""; pour un String vide.
```
- Surcharge de l'opérateur `+` entre la classe `String` et les types primitifs :
 - ```
System.out.println(s + " valeurs : " + 1 + ", " + true);
// affiche : De belles valeurs : 1, true.
```
  - Remarque : la surcharge utilisateur des opérateurs n'est pas possible.
- Les `Strings` sont vraiment des objets :
  - ```
String s = "foo";
```
 - ```
String t = s.substring(1); // contient "oo"
```
  - ```
String u = "Go! Now!".substring(1, 2); // contient "o!"
```

Chaînes de caractères (2)

- Les états des objets `String` sont **immutables** (utiliser la classe `StringBuilder`), les opérations usuelles créent de nouveaux objets `String`.
- Exemple :
 - `String s = "a", p = s; ⇒ p == s` est **true**.
 - `s += "!"; ⇒ s` référence un nouveau `String` ("a!"), `p` toujours l'ancien `String` ("a").
 - `p = "a!" ⇒ p` référence un nouveau `String` ("a!") et `p == s` est **false**.



- Remarque : le premier `String` ("a") n'est plus référencé et peut être supprimé par le ramasse-miettes.

Classes

- Une classe permet d'instancier des objets.
 - Toute classe hérite implicitement de la classe `Object` (racine).
 - **Convention** : un nom de classe commence par une majuscule.
 - Généralement une seule classe est déclarée par fichier.
 - Si la classe a une visibilité publique, le fichier doit porter le même nom que la classe + l'extension `.java`.
 - Il existera autant de fichiers `.class` produits par le compilateur (`javac`) qu'il existe de classes dans le fichier `.java`.
- Déclaration de classe (syntaxe la plus simple) :

```
class NomClasse
{
    // déclarations d'attributs et de méthodes
}
```

Attributs

- Une classe peut déclarer des **attributs** (ou champs ou variables membres) permettant de définir l'**état** de ses objets.
 - Ce sont des **variables de types primitifs** ou des **références sur des objets**.
 - Il est possible de leur modifier leur valeur par défaut.
 - Ils ont généralement une visibilité privée.
 - **Convention** : un nom d'attribut commence par une minuscule.

- Exemple :

```
public class Person
{
    // attributs public, à éviter !
    public String name;           // référence à null
    public int age = 18;          // au lieu de 0
    public Person father, mother; // références à null
}
```

- L'accès aux attributs de l'instance manipulée se fait par notation pointée :

```
Person aPerson = new Person();
aPerson.name = "John";
```

Méthodes

- Une **méthode** (ou fonction membre) est une fonction définie sur une classe qui s'applique sur les objets de cette classe.
 - Les paramètres et le type de résultat d'une méthode sont des types primitifs ou de références à des objets.
 - Comme en C/C++, une méthode ne retournant aucune valeur (procédure) doit déclarer un type de résultat **void**.
 - **Convention** : un nom de méthode commence par une minuscule.
- Déclaration analogue à celle des fonctions en C :

```
public class UneClasse
{
    TypeResultat nomMethode (Type1 arg1, ... TypeN argN) {
        // corps de la méthode
    }
}
```

Méthodes (2)

- Formellement, la **signature de type** d'une méthode est une expression de la forme :
 - $\text{méthode} : \text{Classe} \times \text{Type}_1 \times \dots \times \text{Type}_n \rightarrow \text{TypeR\u00e9sultat}$
 - Où *Classe* d\u00e9note la classe de l'instance manipul\u00e9e.
 - Invocation de m\u00e9thode
 - Syntaxe : `r\u00e9f\u00e9rence.m\u00e9thode(<param\u00e8tres effectifs de Typei>);`
 - La m\u00e9thode s'applique sur `r\u00e9f\u00e9rence`, l'instance de *Classe* manipul\u00e9e.
 - Le compilateur transforme cette syntaxe OO en un appel de fonction de la forme : `m\u00e9thode(r\u00e9f\u00e9rence, <param\u00e8tres>)`.
- ⇒ Toutes les propri\u00e9t\u00e9s de l'instance manipul\u00e9e (`r\u00e9f\u00e9rence`) sont implicitement disponibles dans le corps de la m\u00e9thode.
Cette instance peut \u00eatre explicitement d\u00e9not\u00e9e par le mot clef `this`.

Example

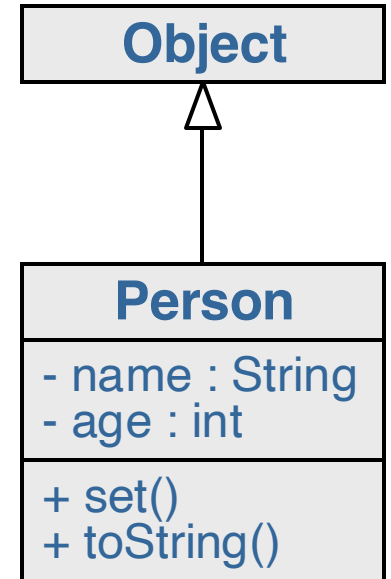
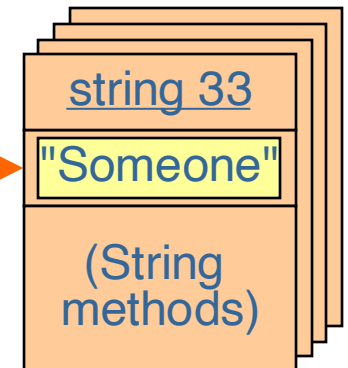
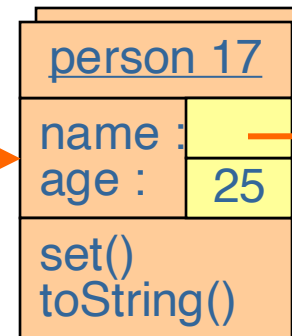
```
■ public class Person
{
    private String name;
    private int age;

    public void set(String aName, int anAge) {
        name = aName;
        age = anAge;
    }

    public String toString() {
        return name + ", " + age;
    }
}
```

```
Person p = new Person();
p.set("Someone", 25);
System.out.println(p.toString());
```

p



Constructeurs

- La méthode `set(...)` définie sur la classe `Person` permet d'initialiser les attributs d'une instance, mais rien ne garantit que l'utilisateur l'invoque.
- La notion de **constructeur** (+ encapsulation) permet d'automatiser cette initialisation et d'assurer qu'un objet sera construit dans un **état cohérent**.
 - Méthode possédant le **même nom que la classe**, **sans type de résultat**.
 - Il peut disposer d'un nombre variable de paramètres (ou aucun).
 - Toute classe définit **au moins un constructeur**.
 - Si aucun constructeur n'est spécifié pour une classe, le compilateur en définit implicitement un, **sans paramètres** le **constructeur par défaut** qui initialise les attributs à leur valeur par défaut :
 - nombres : `0` ;
 - Booléens : `false` ;
 - caractères : caractère de code nul ;
 - références (y compris celles de type `String`) : `null`.

Constructeurs (2)

- L'invocation d'un constructeur s'effectue à la création de l'objet au moyen de l'opérateur `new` : `new NomClasse(<paramètres>)`.
- Définition d'un constructeur :

```
public class Person
{
    // ...

    public Person(String aName, int anAge) {
        name = aName;
        age = anAge;
    }
}

Person p = new Person("John", 22);
```

- Dès qu'un constructeur est explicitement défini celui par défaut est perdu.
 - Permet de garantir qu'un objet sera créé seulement par un constructeur utilisateur (se devant d'initialiser correctement les attributs).

Constructeurs (3)

- Il peut exister plusieurs constructeurs par classe :
 - différentes manières d'instancier un objet ;
 - mais ils ne peuvent pas avoir les mêmes paramètres (surcharge).
- Un constructeur peut invoquer un autre constructeur de la même classe par l'instruction `this(<paramètres>) ;`.
 - Attention : utilisable que comme **première instruction** du constructeur.

```
public class Person
{
    // ...

    public Person(String aName) {
        this(aName, 0);
    }
}
```

- Pas de destructeurs : les objets qui ne sont plus référencés sont éliminés automatiquement (ramasse-miettes).

Mode de transmission des paramètres

- Il existe deux modes de transmission des paramètres :
 - par valeur ;
 - par référence (ou par adresse).
- Par **valeur** : la méthode reçoit une **copie** de la valeur du paramètre effectif et travaille sur cette copie sans incidence sur l'original.
- Par **référence** : la méthode reçoit une référence sur le paramètre effectif avec lequel elle travaille directement. Elle peut donc modifier la valeur du paramètre effectif.
- Certains langages (p. ex. C++) autorisent les deux modes de transmission.

Mode de transmission des paramètres (2)

- Java emploie exclusivement le mode de transmission par **valeur**.
- Cependant les objets ne sont manipulés en Java qu'au moyen de références.
 - A l'appel d'une méthode possédant un paramètre formel de type référence sur un objet, c'est une **copie** de cette référence que reçoit la méthode, non une copie de l'objet.
 - La méthode peut donc modifier l'objet référencé.
- En définitive, tout se passe comme si le mode de transmission était :
 - par valeur pour les types primitifs (et les objets **immutables** : **String**, types enrobeurs, etc.) ;
 - par référence pour les objets mutables.

Exemples

```
■ public class Z
{
    public void swap(int a, int b) {
        int tmp = a;
        a = b;
        b = tmp;
    }
}

int x = 1, y = 2;
Z z = new Z();

z.swap(x, y);           // x = 1, y = 2, inchangés.
```

Examples (2)

```
■ public class Int
{
    private int value;

    public Int(int i) {
        value = i;
    }

    public int value() {
        return value;
    }

    public void swap(Int other) {
        int tmp = value;
        value = other.value;
        other.value = tmp;
    }
}

Int x = new Int(1), y = new Int(2);
x.swap(y); // x.value = 2, y.value = 1, modifiés.
```

« this »

- Dans le corps d'une méthode, le mot clef `this` dénote l'instance manipulée.
- Utile afin de :
 - passer l'instance en paramètre à une autre méthode ;
 - accéder aux propriétés de l'instance qui pourraient être masquées ;
 - rendre l'instance comme résultat de la méthode (p. ex. pour pouvoir chaîner des appels).
- Exemple :

```
class Person
{
    // ...

    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

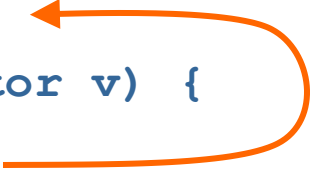


`age` est le paramètre
dans ce contexte

« this » (2)

- Exemple, définition d'une méthode effectuant un += entre vecteurs :

```
class Vector
{
    private int x, y;
    Vector addTo(Vector v) {
        this.x += v.x;
        this.y += v.y;
        return this;
    }
}
```



Rem. : il est possible d'accéder aux attributs privés de **v** (même classe) car l'unité d'encapsulation est la classe, non l'objet.

Q : comment définir une méthode + ?

```
Vector v1 = new Vector(1, 0), v2 = new Vector(42, 42),
        v3 = new Vector(-1, -1);
v1.addTo(v2).addTo(v3); // Résultat : v1 : (42, 41)
```

- Ne pas confondre **this.attribut** ou **this.methode(...)** avec l'invocation, depuis un constructeur d'un autre constructeur par **this(...)**.
De même, **super.attribut** et **super.methode(...)** accèdent aux propriétés de la super-classe, tandis que **super(...)** invoque l'un de ses constructeurs.

Programme

- Il doit exister une méthode publique `main()` définie dans une classe d'un fichier pour pouvoir exécuter le programme.
- Syntaxe :
 - ```
public static void main(String[] args) {
 // instructions
}
```
  - où `args` est un tableau de `Strings` contenant les éventuels arguments de la ligne de commande.
- Compilation et exécution d'un programme (en ligne de commande) :
  - compilation `javac <NomClasse>.java ;`
  - exécution `java <NomClasse> [arg1, arg2, ... argn] ;`
  - où `<NomClasse>` est une classe définissant une méthode `main()`.

# Tableaux

---

- Un **tableau** est un **objet** :
  - dérivé de la classe de base **Object** ;
  - sa taille est fournie par l'attribut **length** ;
  - ne contient que des éléments du même type ;
  - accès à un élément grâce à un **[indice]** (de 0 à **length - 1**).
- Tableaux d'**objets** ou de valeurs de **types primitifs**.
- Utilisation
  - Déclaration : **Type[] nomReference; OU Type nomReference[];**
  - Instanciation : **nomReference = new Type[taille];**
  - Affectation : **nomReference[indice] = valeur;**

# Tableaux (2)

---

- L'objet (resp. valeur) d'indice `i` dans le tableau référencé par `ref`, est dénoté par `ref[i]` et peut être utilisé comme une référence (resp. variable) normale (p. ex. `ref[i].methode()`).
- Exemples :
  - ```
int[] array;  
array = new int[10];  
array[4] = 2000;  
array[3] = array[4] - 2;
```
 - ```
Object[] objectArray = new Object[30];
objectArray[0] = "toto"; // un String, donc Object
objectArray[1] = new Person(...);
System.out.println(objectArray.length); // affiche 30
```
  - ```
String[][] matrix = new String[5][10];  
matrix[2][3] = "Néo";
```

Tableaux (3)

- Déclaration générale d'un tableau
 - Déclarer une référence à l'objet tableau (`Object[] array`).
 - Créer le tableau (`array = new Object[10]`), groupe de réfs/vars.
 - La création d'un tableau d'objets ne crée pas ses éléments (après `Person[] p = new Person[10]`, `p[0]` vaut `null`).
 - Pour les types primitifs, le tableau est initialisé aux valeurs par défaut (après `int[] i = new int[10]`, `i[0]` vaut `0`).
 - Créer les objets ou initialiser les valeurs qu'il contient.
- Déclaration simplifiée d'un tableau, si les éléments sont connus
 - `double[] tableau = { 1.2, 12.33, 33 };`
 - `String[] rgb = { "rouge", "vert", "bleu" };`
 - `int[][] matrice = { {2, 3}, {3, 4}, {5, 6} };`
 - `Integer[] tabInteger = { new Integer(1), new Integer(12) };`

↑
préférer la méthode de classe `Integer.valueOf(1)` ;

Tableaux (4)

- Possibilité de définir un tableau comme paramètre d'une méthode « à la volée ».

- Exemple :

- ```
public class Notes
{
 // ...

 public void definirNotes(float[] notes) {
 // ...
 }
}
```

```
Notes n = new Notes();
```

- L'appel suivant est **invalide** : `n.definirNotes({ 4, 2.5f, 3 });`  
La notation simplifiée n'est autorisée uniquement que lors de la déclaration d'une référence, sinon ambiguïté pour le compilateur.
- Il faut donc invoquer la méthode en définissant le tableau « à la volée » :  
`n.definirNotes(new float[] { 4, 2.5f, 3 });`

# Arguments de main()

---

- La méthode `main`, point d'entrée d'une application, reçoit en paramètre un tableau de `String` contenant les arguments de la ligne de commande.
- Exemple :

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

> java Test un deux trois

0: un

1: deux

2: trois

# Ellipse (...)

---

- Définition d'une méthode acceptant un nombre de paramètres variables.
  - Syntaxe : `TypeRetour methode( [ paramètres ] Type ... args)`.
  - D'autres paramètres peuvent être définis, mais la déclaration d'un nombre de paramètres variables doit être la dernière de la liste.
  - Les paramètres effectifs sont récupérés dans un tableau de `Type` (le type précédant l'ellipse `...` définit le type des arguments, contrairement au C).
  - Remarque : la méthode `main` peut aussi s'écrire,  
`public static void main(String ... args)` (ou `String[] args`)

- Exemple :

```
public class A {
 public void m(String ... args) {
 for (int i = 0; i < args.length; i++)
 System.out.println(i + ": " + args[i]);
 }
}
```

```
new A().m("foo", "bar"); // objet anonyme sur lequel est invoqué m()
```

# Exceptions (aperçu)

---

- Une exception est un **objet**, décrivant un problème qui ne devrait pas survenir, instancié dans une sous-classe de la classe **Exception**.
  - La plus fréquente est la **RuntimeException** (et ses sous-classes).
- Génération d'une exception
  - Automatiquement par le système (p. ex. accès illégal à une propriété d'une référence à **null**, ouverture d'un fichier inexistant...).
  - Manuellement par l'utilisateur en utilisant l'instruction **throw** :

```
if (o == null)
 throw new NullPointerException("Object is null!");
```
- Traitement d'une exception
  - Si l'instruction pouvant lever une exception est incluse dans un bloc **try { ... }**, alors un bloc **catch (TypeDeLException e) { ... }** peut récupérer l'exception et rétablir un contexte d'exécution cohérent,
  - Sinon, l'exécution du programme est abandonnée.



# Exemple

```
■ public class Division
{
 public static void main(String ... args) {
 if (args.length != 2)
 throw new RuntimeException("java Division <n1> <n2>");
 String result = args[0] + " / " + args[1] + " = ";
 try {
 int a = Integer.parseInt(args[0]);
 int b = Integer.parseInt(args[1]);
 result += "" + a / b;
 }
 catch (RuntimeException e) {
 result += "<undefined>";
 }
 System.out.println(result);
 }
}
```

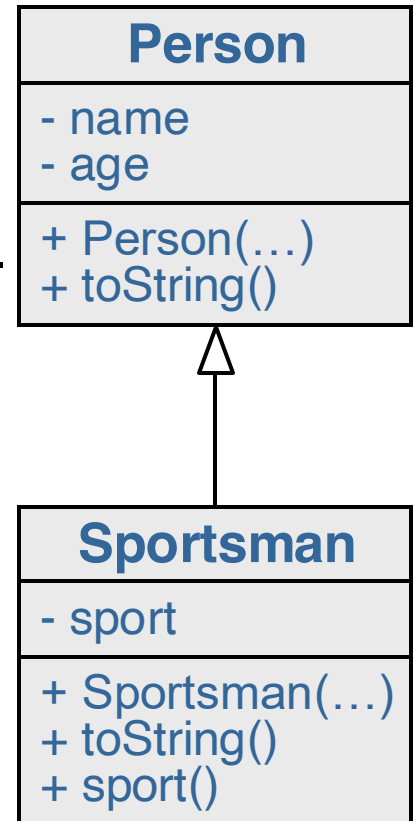
exception si n'est pas un nombre

exception si division par 0

# Héritage

- Une **sous-classe** (ou classe dérivée) permet de définir une **sous-population** des objets de la super-classe.
  - Elle est spécifiée par le mot clef **extends**.
  - Pas d'héritage multiple en Java : une seule classe parent.
  - Elle hérite de **toutes** les propriétés définies dans la classe parent et peut en **définir** ou **redéfinir** d'autres.
  - Elle hérite indirectement de la super-classe **Object**.
- Exemple :

```
public class Sportsman extends Person
{
 private String sport;
 public String sport() {
 return sport;
 }
 // autres méthodes
}
```



# Héritage et constructeurs

---

- Les constructeurs d'une sous-classe sont responsables de l'initialisation de tous les attributs de la classe, y compris ceux hérités.
  - L'implémentation du constructeur par défaut comporte l'instruction `super()`, invoquant le constructeur sans paramètre de la classe parent (par défaut, celui de la classe `Object`, qui ne fait rien).
  - Un constructeur utilisateur peut invoquer explicitement un autre constructeur de la classe parent :
    - `super();` // non indispensable, rajouté par le compilateur
    - `super(<paramètres>);`
    - Attention : utilisable que comme **première instruction** du constructeur.
- ⇒ Intérêt de `super(...)` pour invoquer un constructeur de la classe parent :
- permet de factoriser les initialisations (comme `this(...)`) ;
  - permet d'initialiser des attributs privés définis dans la super-classe.

# Héritage et constructeurs : exemple

---

```
■ public class Sportsman extends Person
{
 private String sport;

 public String sport() {
 return sport;
 }

 public Sportsman(String aName, int anAge, String aSport)
 {
 super(aName, anAge);
 sport = aSport;
 }
}

Sportsman s = new Sportsman("John", 22, "tennis");
```

# Héritage et constructeurs (2)

---

## ■ Fonctionnement d'un constructeur

- Soit invocation explicite d'un autre constructeur (première instruction) par `super(...)` ou `this(...)`,  
soit invocation implicite du constructeur sans paramètres du parent
    - Attention : ce dernier doit exister.  
Soit le constructeur par défaut soit, si un constructeur existe, un constructeur sans paramètres explicitement défini.
- ⇒ Construction d'un l'objet d'abord dans la (les) super-classe(s).
- Initialisation explicite (`int i = 2`) ou par défaut des valeurs des attributs locaux (`int x`).
  - Les autres instructions du constructeur sont **ensuite** exécutées.

# Héritage et constructeurs : exemple

```
■ class A
{
 A(){ System.out.print("A() "); }

 A(int x) {
 System.out.print("A(int)"); }
}

class B extends A
{
 B() { System.out.println(" B()"); }

 B(int x) {
 System.out.println(" B(int)"); }

 B(String x) {
 super(0);
 System.out.println(" B(String)"); }
}
```

## ■ Code

```
B b;
b = new B();
b = new B(0);
b = new B("");
```

## ■ Résultat

```
A() B()
A() B(int)
A(int) B(String)
```

super() implicite  
(A() doit exister)



# Héritage et constructeurs : exemple (2)

```
■ class A
{
 A() { System.out.println("A()"); }
 A(double x) { this(); System.out.println("A(double)"); }
}

class B extends A
{
 A a = new A();
 B(int i) { super(i); System.out.println("B(int)"); }
}
```

■ Résultat de `B b = new B(3);` // Rem.: `B b = new B()` impossible!

```
A()
A(double)
A()
B(int)
```

# Héritage et redéfinition

---

- Il est possible de **redéfinir** une méthode héritée dans une sous-classe.
  - Simplement en définissant à nouveau la méthode.
  - Exactement le même **nom**, le même **nombre**, le même **ordre** et les mêmes **types** de **paramètres** (mais pas forcément les mêmes noms de variables) et (en première approche) le même **type de résultat** ;  
⇒ **même signature de type de la méthode** (au nom de la classe près).
  - Mécanisme de **liaison dynamique** entre la méthode originale et la méthode redéfinie.
  - Sinon c'est une **surcharge** (même nom), pas une redéfinition.  
⇒ Pas de mécanisme de liaison dynamique possible.
- Dans le corps de la méthode redéfinie, il est possible (lire, « **il faut souvent** ») d'invoquer la méthode originale par **super.methode(<paramètres>)**.
  - Permet de spécialiser un comportement (p. ex. pour une méthode **draw**).



# Tour de magie : toString()

---

- Toutes les classes possèdent une méthode `public String toString()` héritée de la super-classe `Object`.
- Cette méthode est **automatiquement** invoquée dès qu'une instance doit être représentée sous la forme d'une chaîne de caractères.
  - `Person p = new Person(...);`  
`System.out.println(p);`
  - `println` prend en paramètre un `Object`  $\Rightarrow$  `p.toString()` est invoquée.
- La méthode `toString` définie dans la classe `Object` n'est pas très parlante.
  - Elle rend `NomClasse@hashCodeInstance` (parfois basé sur son adresse).
  - Il est souvent utile de la redéfinir dans les sous-classes pour :
    - afficher des informations plus pertinentes (valeur des attributs) ;
    - bénéficier du mécanisme de liaison dynamique (p. ex. pour afficher une collection d'`Object` pouvant contenir des instances de `Person`).

# Example

---

```
■ public class Sportsman extends Person
{
 private String sport;

 public Sportsman(String aName, int anAge, String aSport) {
 super(aName, anAge);
 sport = aSport;
 }

 public String sport() {
 return sport;
 }

 public String toString() {
 return super.toString() + ", " + sport;
 }
}

Sportsman s = new Sportsman("John", 22, "Tennis");
Person p = s;
System.out.println(p);
```

redéfinition

// Affectation polymorphique  
// Affiche : John, 22, Tennis

# Paquetages

---

- Un paquetage (package) est une bibliothèque de classes.
- Déclaration d'un paquetage
  - Déclarer en début de fichier : `package nomPackage;`  
⇒ Toutes les classes définies dans un fichier appartiennent au même paquetage.
  - Si aucun paquetage n'est déclaré dans un fichier, les classes correspondantes appartiennent au **paquetage par défaut** (i.e., il contient toutes les classes hors paquetage accessibles).
- Les fichiers compilés (**.class**) d'un paquetage doivent être placés dans un répertoire de même nom.
- Les paquetages peuvent être organisés de manière hiérarchique :
  - `package nom1.nom2...nomN;`  
⇒ hiérarchie de répertoires pour les fichiers .class: `nom1/nom2/.../nomN`.

# Utilisation des paquets

---

- Par défaut, le compilateur recherche la définition des classes utilisées dans le paquetage courant (ou par défaut). Pour les autres, il est nécessaire de préciser dans quel paquetage elles se trouvent.

- Utilisation d'une classe, notation pointée : `nomPaquetage.NomClasse`

```
people.Person p = new people.Person("John");
```

- Importation d'une seule classe : `import nomPaquetage.NomClasse;`. `NomClasse` est ensuite utilisable directement.

```
import people.Person;
```

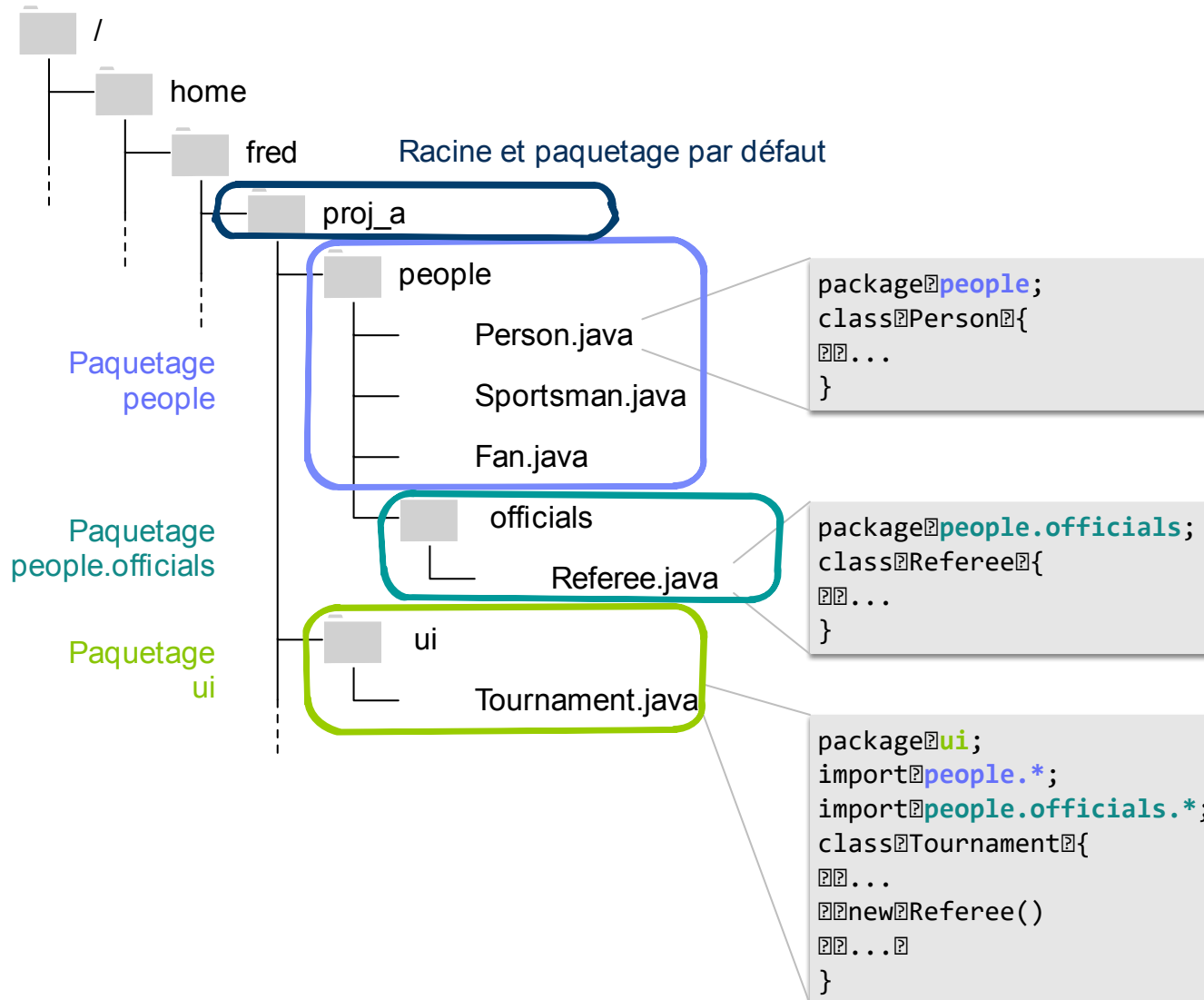
```
// ...
```

```
Person p = new Person("John");
```

- Importation de toutes les classes d'un paquetage : `import nomPaquetage.*;`  
Attention : n'importe pas les sous-paquetages d'un paquetage.

```
import people.*;
```

# Utilisation des paquetsages (2)



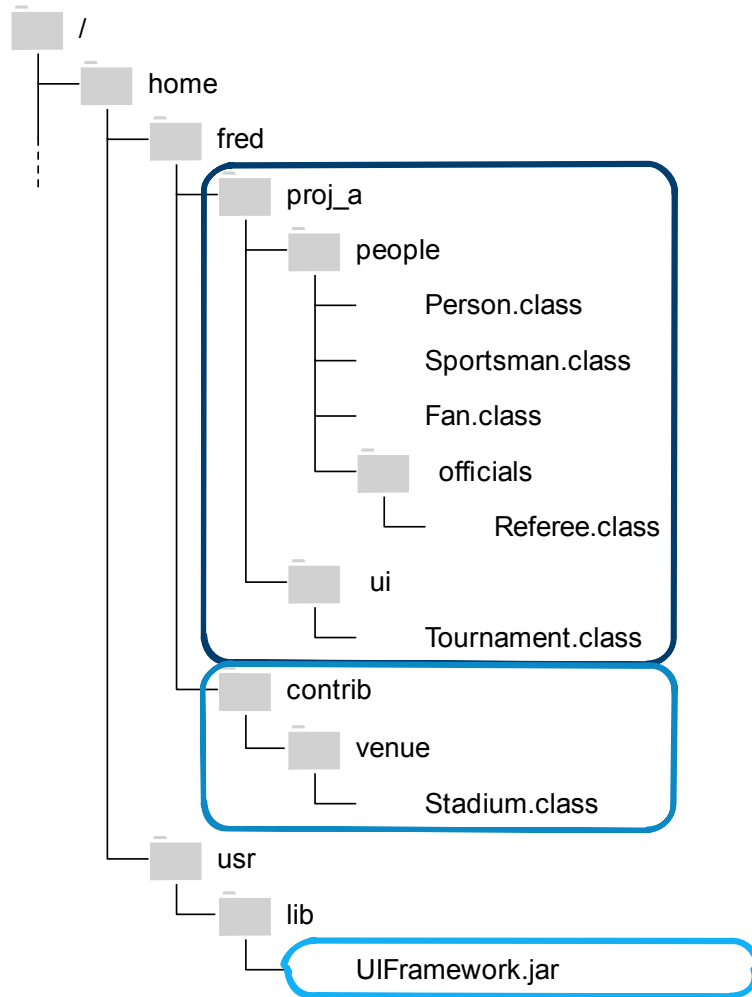
# Accès aux paquetages

---

## ■ CLASSPATH

- Variable d'environnement permettant de localiser les paquetages à la compilation et à l'exécution.
  - Si non initialisée (par défaut), contient le répertoire courant.
- Le répertoire racine d'un paquetage utilisateur doit être accessible depuis les répertoires définis dans le CLASSPATH.
- Par défaut, si un fichier utilise un paquetage **p** et est compilé ou exécuté depuis un répertoire **r**, **p** doit être un sous-répertoire de **r** (i.e., .../r/p).
  - Pour compiler (depuis **r**) une classe **c** du paquetage **p**: **javac p/C.java**.
  - Pour exécuter (depuis **r**) la méthode **main** de la classe **c**: **java p.C**.
- Il est toujours possible d'accéder aux classes prédéfinies Java (**java.\***, **javax.\*...**).
- Les classes prédéfinies de **java.lang.\*** sont implicitement importées.

# Accès aux paquetages (2)



Vue logique des paquetages et classes  
après application du CLASSPATH :

```
people.Person
people.Sportsman
people.Fan
people.officials.Referee
ui.Tournament
venue.Stadium
uiframework.widgets.Window
uiframework.widgets.Button
...
```

```
$ export CLASSPATH=/home/fred/proj_a:/home/fred/contrib:/usr/lib/UIFramework.jar
```

# Visibilité

---

- La déclaration d'une classe ou d'une propriété peut être préfixée par un **modificateur de visibilité**.
- Permet de réaliser le concept d'encapsulation.
- **Visibilité des propriétés** (attributs et méthodes)
  - **private** : visible seulement dans la classe.
  - aucun : visibilité « package » (paquetage).
  - **protected** : visibilité « package » et pour les sous-classes (même si dans un paquetage différent).
  - **public** : visible partout.
  - La plupart du temps, préférer une visibilité **private** pour les attributs (encapsulation).



# Visibilité (2)

---

- Il est possible de déclarer un constructeur `private`, mais cela est pertinent :
  - si un autre constructeur public l'invoque (par `this(...)`) ;
  - ou s'il faut **interdire la création d'instances** de cette classe (p. ex. classes ne déclarant que des propriétés statiques, comme `java.lang.Math`), le constructeur par défaut étant `public`.
- **Visibilité des classes**
  - aucun : visibilité « package ». Avantage, classe pas exposée.
  - `public` : visible partout, déclarée dans un fichier de même nom.
  - private/protected : possible seulement pour les classes internes.
    - `private` : visible pour la classe et la classe englobante.
    - `protected` : visibilité « package » et pour les sous-classes.

# Exemple

- `package test;` fichierClazz.java

```
public class Clazz
{
 private int privateInt;
 protected int protectedInt;
 int defaultInt;
}

class Visibility
{
 void test() {
 Clazz c = new Clazz();

 //c.privateInt = 1;
 c.defaultInt = 1;
 c.protectedInt = 1;
 }
}
```

- `import test.*;`

```
class SubClazz extends Clazz
{
 void test() {
 //privateInt = 1;
 //defaultInt = 1;
 protectedInt = 1;
 }
}
```

illégal, hors package

légal, hors package mais sous-classe

illégal, privé

légal, même package

# Visibilité et héritage

- Dans une sous-classe il est possible d'augmenter la visibilité d'une méthode redéfinie (mais pas de la restreindre, cela violerait les spécifications de l'interface héritée) .

- `private`
- aucun
- `protected`
- `public`



- Exemple :

```
class Invisible
{
 protected void m() { ... }
}

class Visible extends Invisible
{
 public void m() { ... }
}
```

mais pas `private` : d'après l'interface d'`Invisible`, une méthode `m()` doit être accessible dans `Visible` pour le packaging.

# Exercice : Star Wars

---

## ■ Types

- Humanoïde
  - Ont un nom
  - Affichage : Humanoid <nom>
- Jedi et Sith
  - Sont des humanoïdes
  - Ont une force
  - Affichage : type <nom>, force <force>, lightsaber <couleur>, master <maître> (p.ex. *Jedi Luke, force: 23000, lightsaber: blue, master: Yoda*)
- Jedi
  - Ont un maître Jedi et un sabre laser bleu, vert, jaune, violet, orange ou blanc
- Sith
  - Ont un maître Sith et un sabre rouge
  - Peuvent corrompre un Jedi en Sith (p.ex. *Darth Sidious corrupts Anakin into Darth Vader*)

# Exercice : StarWars (2)

---

## ■ Main

- Créer un humanoïde Paul
- Créer les Jedi :
  - Yoda (19000, green)
  - Obi-Wan (17000, blue, master Yoda)
  - Anakin (27000, green, master Obi-Wan)
  - Luke (23000, green, master Obi-Wan)
- Créer le Sith (Darth Sidious, 20000)
- Faire corrompre Anakin par Darth Sidious en Darth Vader
- Mettre Paul, Yoda, Dath Sidious, Obi-Wan, Darth Vader et Luke dans un tableau
- Afficher les éléments du tableau

# Exercice : StarWars (2)

---

## ■ Main

- Créer un humanoïde Paul
- Créer les Jedi :
  - Yoda (19000, green)
  - Obi-Wan (17000, blue, master Yoda)
  - Anakin (27000, green, master Obi-Wan)
  - Luke (23000, green, master Obi-Wan)
- Créer le Sith (Darth Sidious, 20000)
- Faire corrompre Anakin par Darth Sidious en Darth Vader
- Mettre Paul, Yoda, Dath Sidious, Obi-Wan, Darth Vader et Luke dans un tableau
- Afficher les éléments du tableau
- Comment afficher tous les éléments du tableau qui possèdent un sabre vert en n'utilisant que les concepts vus jusqu'ici ?

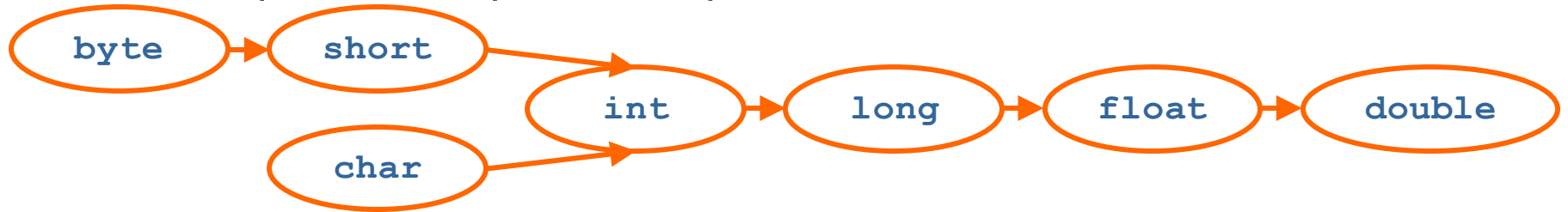
# Polymorphisme

---

- Plusieurs polymorphismes ([Strachey 67] [Cardelli 85] [Meyer 88] [Booch 91]...).
  - Universel
    - Paramétrique opération applicable sur plusieurs types (généricité).
    - **Inclusion** opération applicable sur des types liés par une relation de sous-typage (rem: héritage  $\Rightarrow$  sous-typage).
  - Ad hoc
    - Surcharge même identificateur pour des opérations différentes.
    - Conversion conversion automatique de type.
- Principe de **substitution** en POO (polymorphisme d'inclusion) :
  - si  $C'$  est une sous-classe de  $C$ , une instance de  $C'$  peut être utilisée là où une instance de  $C$  est attendue (affectations, passages de paramètres) ;  
`Person p = new Sportsman(...); // affectation polymorphique`
  - fortement lié au mécanisme de **liaison dynamique**.  
`System.out.println(p); // invoque Sportsman::toString`

# Polymorphisme (2)

- Un entier peut toujours être utilisé là où un réel est attendu, mais non nécessairement l'inverse.
- Pour les types primitifs, les conversions élargissantes suivantes sont effectuées implicitement par le compilateur :



- Les autres conversions restrictives de types primitifs (sauf pour des valeurs connues à la compilation qui sont dans le domaine du type) doivent être effectuées explicitement.

```
int i = 1; double d = i; // conversion élargissante
i = (int) (d * 2.3); // transtypage nécessaire, i == 2
```



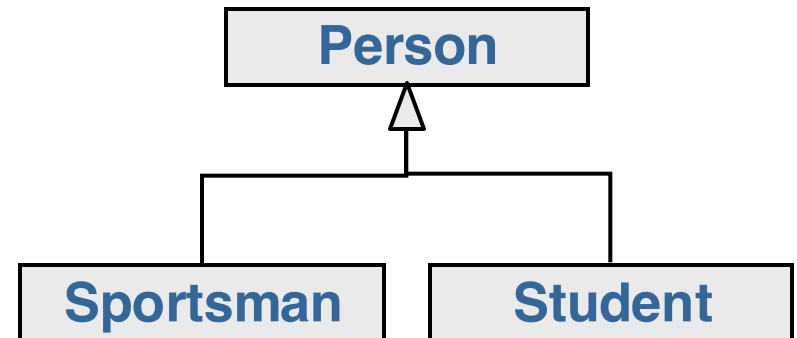
# Polymorphisme (3)

- Autant un objet d'une sous-classe peut être utilisé comme un objet d'une super-classe (un **Student** est une **Person**), autant l'inverse n'est pas vrai.
- Exemple :

```
class Test
{
 void person(Person arg) { ... };
 void student(Student arg) { ... };
}

// ...
Test t = new Test();
Student s = new Student();
t.person(s);

Person p = s; // polymorphisme
// t.student(p);
t.student(s);
```

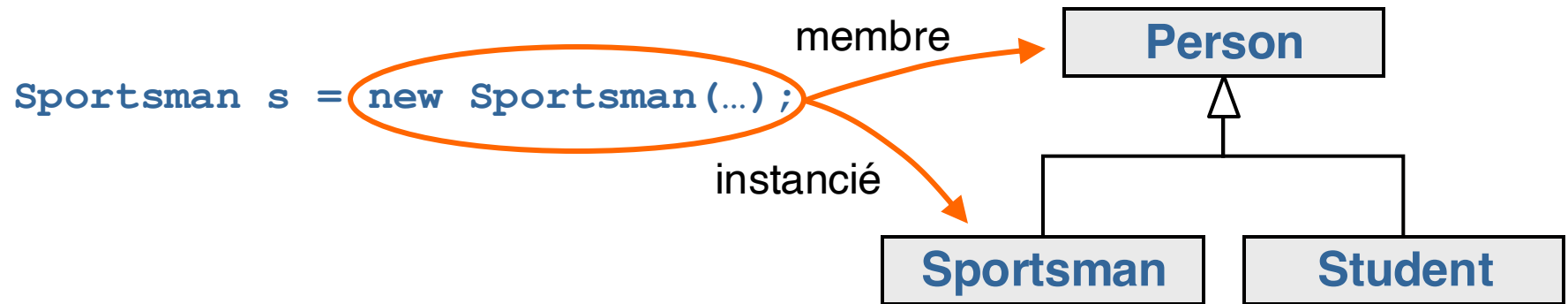


**s** référence un **Student** qui est également une **Person**

illégal : **p** référence bien un objet **Student**, mais vu depuis le contexte **Person**

# Polymorphisme (4)

- Une **instance** d'une classe *C* est **membre** de toutes les super-classes de *C*.



- **Transtypage** (*cast*)
  - Permet de forcer le type d'une référence, d'une variable ou d'un littéral.
  - Extrêmement **dangereux** si on ne sait pas ce que l'on fait.
  - Pour les références, il *faut* que l'**objet** référencé soit effectivement **membre** de la nouvelle classe, sinon une exception `ClassCastException` est levée.
  - Syntaxe : `(Type) variable`.

# Polymorphisme (5) : instanceof

## ■ Exemple :

```
Person p = new Sportsman(...);
// p.sport();
System.out.println(p);
((Sportsman) p).sport();
Student s = (Student) p;
```

Illégal car `sport()` n'existe pas dans les méthodes de `Person`

liaison dynamique sur `toString()`

légal car `p` référence bien un `Sportsman`

syntactiquement correct mais génère une erreur à l'exécution !

- Pour éviter ces erreurs de transtypage à l'exécution, l'opérateur `instanceof` permet de déterminer si un objet est **membre** d'une classe.
  - Syntaxe : `expression instanceof Classe → boolean`
  - `instanceof` rend `true` pour les classes dont l'objet rendu par l'expression `expression` est membre.

# Polymorphisme (6) : instanceof

## ■ Exemple :

```
Person p = new Sportsman(...);

if (p instanceof Sportsman) {
 Sportsman s = (Sportsman) p;
 s.sport();
 boolean b = p instanceof Person; // Toujours true
}
```

- Ne pas abuser de l'opérateur `instanceof` (dénote un code pas très OO...).
- Le *pattern matching* (filtrage par motif) pour l'opérateur `instanceof` permet le transtypage *à la volée* dans une nouvelle référence et d'éviter des erreurs (p.ex. mauvais transtypage en `(Student) p`) :
  - Syntaxe : `expression instanceof Classe [identifiant] → boolean`
  - ```
if (p instanceof Sportsman s)  
    s.sport();
```

Polymorphisme (7) : instanceof

- Dans l'expression `exp instanceof C [id]` :
 - `exp` est une expression rendant une référence à un objet ;
 - `c` peut être un nom de classe, d'interface ou un tableau.
- Si `c` est une classe, détermine si `exp` référence un objet de la classe `c` ou d'une de ses sous-classes.
- Si `c` est une interface, détermine si `exp` référence un objet dont la classe implémente l'interface `c`.
- Si `c` dénote un tableau, `T[]`, détermine si `exp` référence un tableau d'éléments de type `T` (où `T` peut être un type primitif, une classe ou une interface).
- Remarque :
 - A la compilation, le type dénoté par `c` doit être compatible avec `exp`.
P. ex. si le type de la référence rendue par `exp` est une classe `R` et `c` dénote une classe, `c` doit être une sur-classe ou une sous-classe de `R`.

Polymorphisme : exemple

```
■ class Color {}  
  class Red extends Color {}  
  class Blue extends Color {}  
  // ...  
  
  Color c = new Blue();  
  Red   r = new Red();  
  
  System.out.println("Object c: " + (c instanceof Object)); true  
  System.out.println("Color  c: " + (c instanceof Color));  true  
  System.out.println("Blue   c: " + (c instanceof Blue));   true  
  System.out.println("Red    c: " + (c instanceof Red));     false  
  System.out.println("Color  r: " + (r instanceof Color));   true  
  // System.out.println("Blue  r: " + (r instanceof Blue));   erreur  
  System.out.println("Red    r: " + (r instanceof Red));     true
```

Polymorphisme : exemple (2)

■ `Integer i[] = new Integer[5];`

`System.out.println("I[] i: " + (i instanceof Integer[]));` `true`

`System.out.println("O[] i: " + (i instanceof Object[]));` `true`

`System.out.println("O i: " + (i instanceof Object));` `true`

`// System.out.println("S[] i: " + (i instanceof String[]));` `erreur`

`Object o = i;`

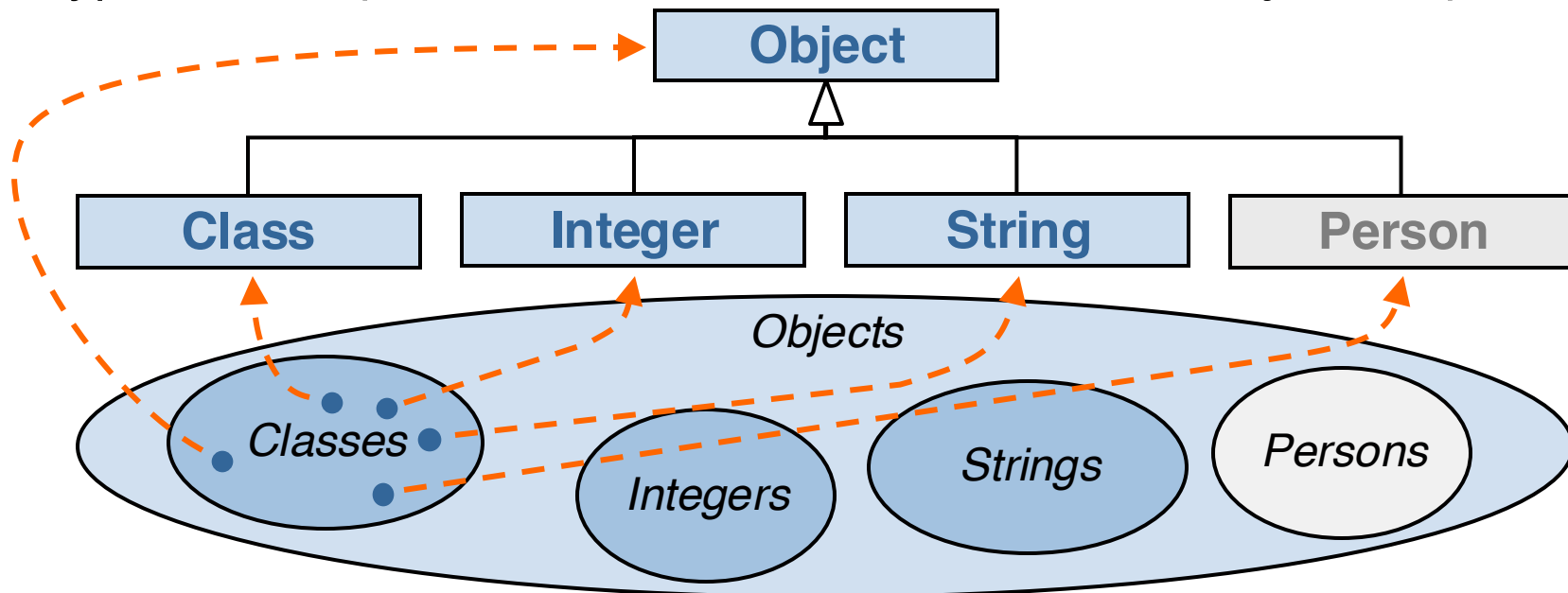
`System.out.println("S[] o: " + (o instanceof String[]));` `false`

`System.out.println("I[] o: " + (o instanceof Integer[]));` `true`

`System.out.println("O[] o: " + (o instanceof Object[]));` `true`

Polymorphisme : Class (8)

- La classe prédéfinie `Class` représente tous les types de l'application.
 - A chaque classe/interface/type primitif correspond un **unique** objet `Class`.
 - Un attribut `class` est associé à chaque nom de type et référence l'objet `Class` correspondant. Syntaxe: `UnType.class`
 - La méthode `getClass()` définie sur la classe `Object` rend un objet de type `Class` représentant la classe où est instancié l'objet manipulé.



Polymorphisme : Class (9)

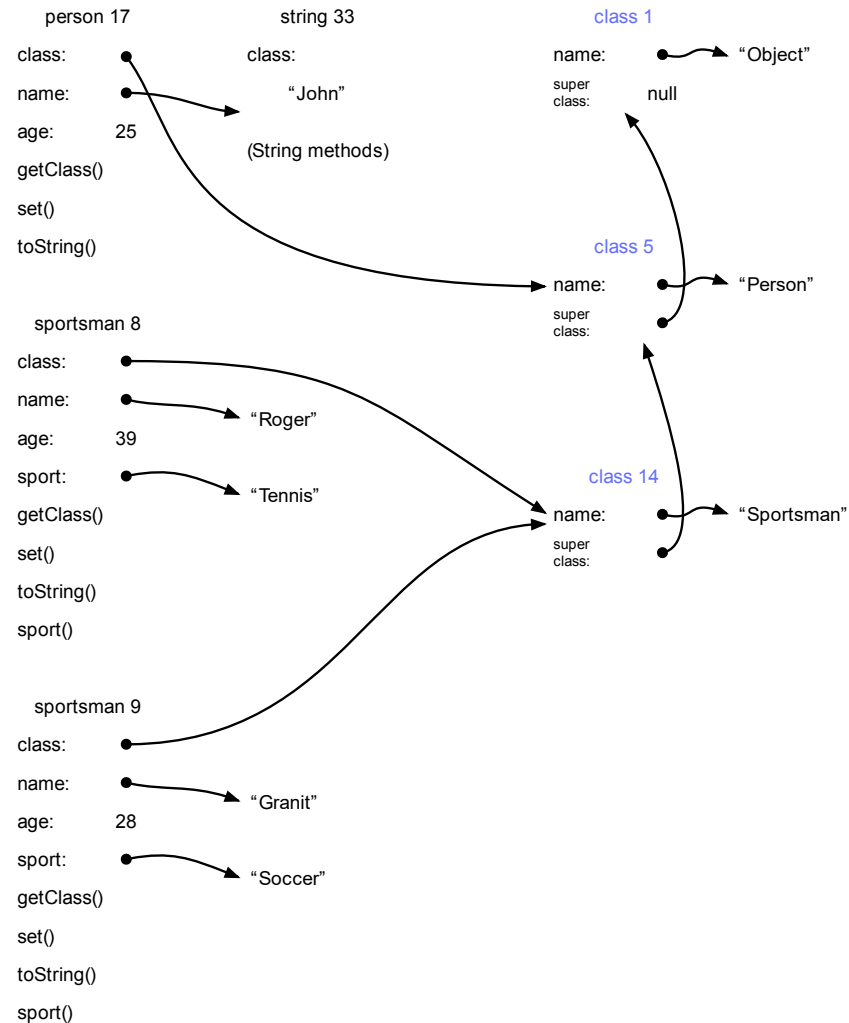
- Pour tester si un objet est **instancié** dans une classe donnée (et non pas seulement **membre**) il est nécessaire d'utiliser la classe **Class**.

- Exemple :

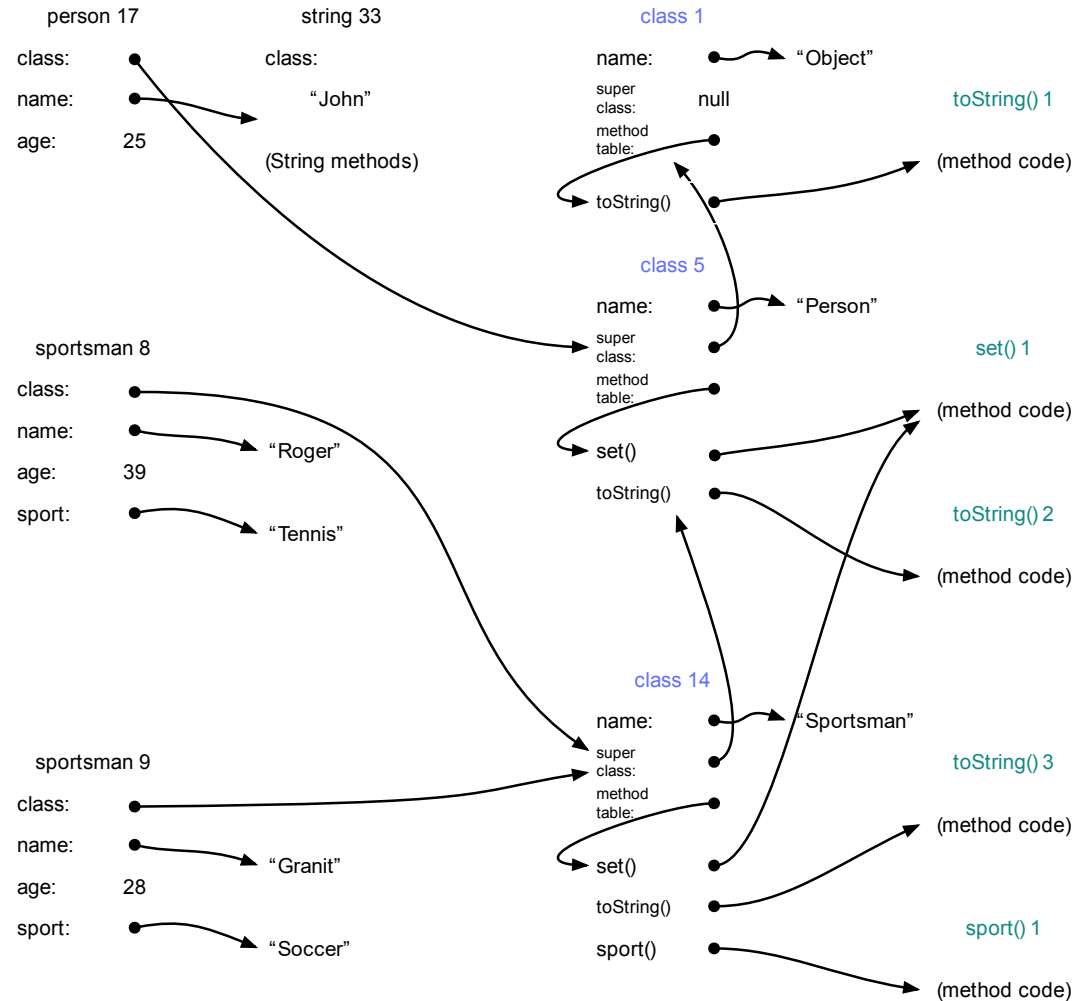
```
Person p = new Sportsman(...);  
  
if (p.getClass() == Sportsman.class) {  
    Sportsman s = (Sportsman) p;  
    s.sport();  
    boolean b = p.getClass() == Person.class; // false...  
}
```

- Par ailleurs, la classe **Class** définit des méthodes de **réflexion**, comme :
 - **String getName()** et **String getSimpleName()** rendant respectivement le nom complet et court de la classe (p. ex. **java.lang.String** / **String**),
 - **Class getSuperClass()**, rendant l'éventuelle superclasse de la classe,
 - **Object newInstance()**, permettant de créer un objet de la classe, etc.

Polymorphisme : exemple



Polymorphisme : exemple (2)



Polymorphisme : instanceof (10)

- La méthode `boolean instanceof(Object o)` définie sur la classe `Class` permet d'invoquer dynamiquement l'opérateur `instanceof`.

- Exemple :

- ```
class Person { /* ... */ }
class Student extends Person { /* ... */ }
class Sportsman extends Person { /* ... */ }

Class[] types = { Student.class, Sportsman.class,
 Person.class, Object.class };

Person p = new Student(...);

for (int i = 0; i < types.length; ++i)
 if (types[i].instanceof(p))
 System.out.println("p : " + types[i].getSimpleName());
```

- Résultat

```
p : Student
p : Person
p : Object
```

importance de l'ordre : classes plus spécifiques en premier pour avoir le vrai type (en ajoutant un `break` dans le `for`)

# Exercice

---

- Ecrire une méthode `static printInstancesCount(Object[] array)` affichant tous les types des objets du tableau `array` ainsi que leur nombre d'instances.
- Indications :
  - utiliser uniquement des tableaux (pas de listes) ;
  - définir une classe permettant de stocker les informations nécessaires.
- Exemple de résultat pour un tableau de `Person` :

```
Person Yoko, 22 years
Student Paul, 20 years, number: 234, orientation: IL
Sportsman Ringo, 21 years, sport: Tennis
Student George, 20 years, number: 456, orientation: SI
Student John, 23 years, number: 123, orientation: IL

Person: 1
Student: 3
Sportsman: 1
```

# Pattern matching pour switch

21

- Des *patterns* peuvent apparaître dans les *labels* des expressions `switch`.
  - Si le *label* est une classe/interface/tableau, l'opérateur `instanceof` est appliqué (attention à l'ordre des labels dans les hiérarchies).
- Exemple :

```
class Cat { String meow() { return "meeeeow"; } }
class Human { int age() { /* ... */ } }

Object o = /* ... */

String sound =
 switch (o) {
 case null -> {
 System.out.println("No animal...");
 yield "";
 }
 case Cat c -> c.meow();
 case Human h when h.age() > 2 -> "Hello!";
 default -> "Argle-bargle";
 };
```

# Autoboxing/Unboxing

---

## ■ Autoboxing

- Conversion automatique d'une valeur de type primitif en une instance de la classe wrapper (enrobeur) correspondante.
- `int`  $\Rightarrow$  `Integer`  
`Integer obj = 1; //` au lieu de `Integer obj = Integer.valueOf(1);`
- Optimisation: pour certaines valeurs d'un type primitif (en tout cas de -128 à 127 pour les `int`), les objets enrobeurs correspondants ne sont créés qu'une seule fois (leur valeur n'étant pas modifiable).

## ■ Unboxing

- Conversion automatique d'une instance d'une classe représentant un type primitif en la valeur correspondante.
- `Integer`  $\Rightarrow$  `int`  
`int i = obj; //` au lieu de `int i = obj.intValue();`

# Exemples

## ■ Affichage des classes des objets enrobeurs :

- `Object array[] = { 1, 2.4, 4.0 / 2 }; // Autoboxing`  
`for (int i = 0; i < array.length; ++i)`  
 `System.out.println(array[i].getClass()); // toString()`

- Résultat

```
class java.lang.Integer
class java.lang.Double
class java.lang.Double
```

## ■ Comparaison d'identités d'objets :

```
Integer integer = 1; // Autoboxing
int value = integer; // Unboxing
Object object = value; // Autoboxing
```

```
System.out.println(integer == 1); // true
System.out.println(integer == object); // true
System.out.println(integer == new Integer(1)); // false
System.out.println(integer == Integer.valueOf(1)); // true
```

mais false si 1000



# Autoboxing/Unboxing (2)

---

- Simplification d'écriture, mais **une valeur de type primitif n'est pas un objet !**
- Attention aux performances ; l'autoboxing n'est pas *gratuit*.  
⇒ préférer l'utilisations de types primitifs lorsque cela est possible.
- Exemple :

```
private static long sum() {
 Long sum = 0L;
 for (long i = 0; i <= Integer.MAX_VALUE; ++i)
 sum += i; // unboxing, somme, creation d'un nouveau Long
 return sum;
}
```

code environ 5 fois plus lent que la version avec un `long` au lieu du `Long`.

# Sortie standard

- La classe `System` définit les attributs de classe `out` et `err`, de type `PrintStream`, représentant les flux de sortie et d'erreur standards.
- La classe `PrintStream` offre les méthodes d'affichage :
  - Surcharges de `print()` et `println()` acceptant en paramètre des valeurs de type primitif, des instances de `String` et des objets.
  - Une méthode `printf(String format, Object ... args)`, largement inspirée de celle de C acceptant un nombre variable d'arguments.
  - Si un `String` est requis et qu'un objet est fourni la conversion s'effectue automatiquement par l'invocation de la méthode `toString()` sur l'objet.
- Exemple, affichage d'un tableau d'objets `Person` :

```
Person tableau[] = /* ... */;
```

```
for (int i = 0; i < tableau.length; ++i)
 System.out.printf("%2d : %s\n", i, tableau[i]);
```

appel de `Person::toString`

autoboxing de `i` en objet `Integer`

# Entrée standard

---

- La classe `System` déclare également l'attribut de classe `in` (de type `InputStream`) représentant le flux d'entrée standard.
- La classe `InputStream` est peu utilisable en tant que telle (lecture d'octets). La classe `Scanner` est un analyseur textuel simple permettant d'extraire des valeurs de type primitif et des instances de `String` d'un flux donné (`InputStream`, fichier ou chaîne de caractères).
- Exemples :
  - ```
Scanner sin = new Scanner(System.in);
```
 - ```
System.out.print("Name: ");
String nom = sin.next();
System.out.print("Address: ");
String adresse = sin.nextLine();
```
  - ```
int sum = 0;  
while (sin.hasNextInt())  
    sum += sin.nextInt();  
System.out.printf("Sum: %d\n", sum);
```

Collections

- Tableaux
 - Avantages : accès rapide, fonctionne avec les types primitifs.
 - Inconvénient : taille fixe.
- Java fournit un ensemble de classes permettant de gérer les collections de taille variable (listes, ensembles, dictionnaires...) dans le module `java.util`.
- Ce module fournit également des méthodes statiques définies sur la classe `Arrays` pour, entre autres, trier et comparer les tableaux (`sort()`, `equals()`)
- Exemple :

```
int[] array = { 1, 22, 8, 33, -1, 12 };
Arrays.sort(array); // appel de méthode statique
for (int i = 0; i < array.length; ++i)
    System.out.println(array[i]);
```

Encapsulation et tableaux/collections

- Les tableaux (et les collections) sont des objets.
- Si une méthode rend une référence sur un tableau (collection) privé elle compromet l'encapsulation de ses éléments.
- Exemple :

```
public class Array
{
    private int[] array = { 1, 2, 3 };
    public int[] m() { return array; }
}
```

```
Array a = new Array();
a.m()[2] = 66;
```



modification !

Solutions

- Rendre une copie du tableau (collection) ou prévoir des méthodes d'accès à ses éléments. P.ex. :

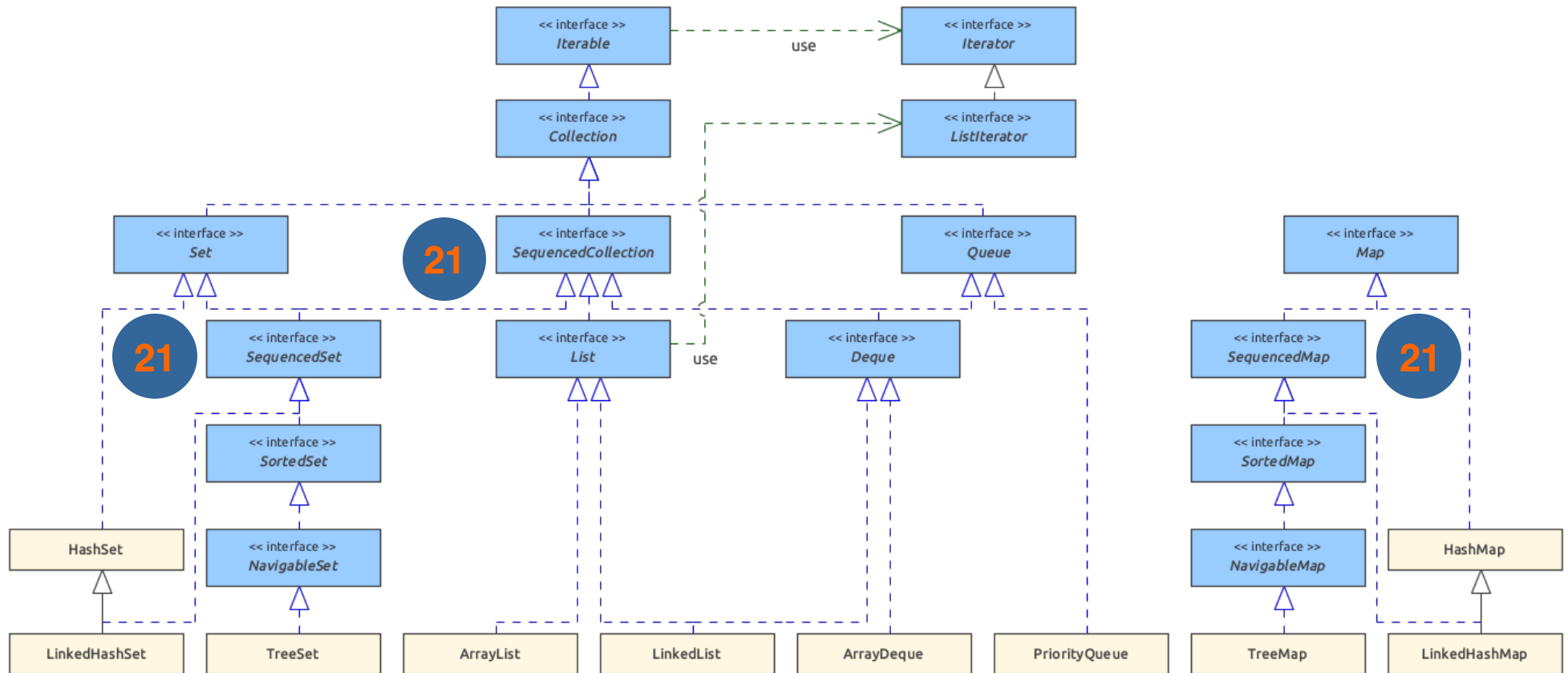
```
public class Array
{
    private int[] array = { 1, 2, 3 };

    int getElement(int i) {
        return array[i];
    }
}

Array a = new Array();
System.out.println(a.getElement(2));
```

- Utiliser des collections non modifiables pouvant être rendues sans risques (voir la méthode `Collections.unmodifiableCollection`).

Collections (2)



Collections (3)

- L'interface `Collection` qu'implémentent les classes listes, ensembles, etc., de `java.util` spécifie (entre autres) les méthodes :
 - `boolean add(Object o)` insérer un élément ;
 - `boolean remove(Object o)` supprimer un élément ;
 - `int size()` connaître la taille de la collection ;
 - `boolean contains(Object o)` savoir si un objet existe ;
 - `Iterator iterator()` obtenir un itérateur sur la collection.
- Les **collections** stockent des références sur des **objets**, non des valeurs de types primitifs.
- Il est nécessaire de convertir une valeur d'un type primitif en objet au moyen de la classe correspondante, avant de pouvoir l'insérer dans une collection.
 - Explicitement : `collection.add(Integer.valueOf(2)) ;`
 - Implicitement par *autoboxing* : `collection.add(2) ;`

Itérateurs

- Un itérateur permet de parcourir une collection donnée.
 - A sa création, se réfère au début de la collection (avant le 1er élément).
- L'interface `Iterator` déclare les méthodes suivantes :
 - `boolean hasNext()` rend `true` si l'itérateur n'est pas en fin de collection ;
 - `Object next()` rend l'élément suivant de l'itérateur ;
 - `void remove()` supprime l'élément courant.
- Utilisation, soit une collection `c` :

```
Iterator i = c.iterator();
while (i.hasNext()) {
    Object o = i.next();
    // ...
}
```
- Ne pas modifier une collection pendant qu'elle est itérée, sauf par les méthodes fournies par l'itérateur (p. ex. `remove()`)

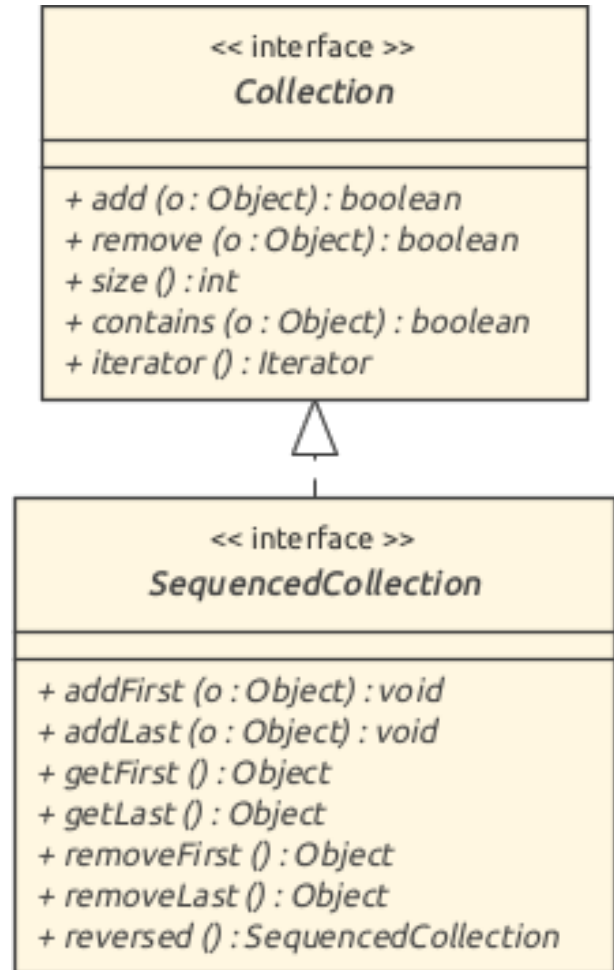
Types de collections

- Deux classes implémentent le concept de liste.
- `ArrayList`
 - Tableaux extensibles.
 - Peu performants en insertion et en suppression.
 - Accès à un élément de manière indexée.
- `LinkedList`
 - Listes doublement chaînées.
 - Accès séquentiel optimal (accès indexé relativement lent).
 - Rem : la méthode `listIterator()` rend un itérateur de type `ListIterator` aux fonctionnalités plus complètes qu'`Iterator`.
- Autres collections
 - Ensembles (listes sans doubles) : implémentations de l'interface `Set`.
 - Dictionnaires (paires clés-valeurs) : implémentations de l'interface `Map`.

Types de collections (2)

21

- Certaines collections ne préservent pas l'ordre des éléments (p. ex. `HashSet` et `HashMap`), d'autres le font (p. ex. `LinkedList`).
- L'interface `SequencedCollection` (et pour les dictionnaires, `SequencedMap`) déclare les méthodes nécessaires aux collections préservant l'ordre :
 - `void add[First|Last](Object o)`
insérer un élément au début ou à la fin ;
 - `Object get[First|Last]()`
obtenir le premier ou le dernier élément ;
 - `Object remove[First|Last]()`
supprimer le premier ou le dernier élément ;
 - `SequencedCollection reversed()`
obtenir une collection dont l'ordre des éléments est inversé.



Conteneurs C++ vs Collections Java

- Les langages C++ et Java offrent des structures de données dans leur bibliothèque standard.
 - C++ : *containers library*.
 - Java : *collections framework*.
- La plupart des structures de données sont les mêmes, mais nommées différemment (attention aux “faux amis”).

Exemples :

- tableau redimensionnable en C++, `vector` ; en Java, `ArrayList` ;
- liste doublement chaînée en C++ : `list` ; en Java, `LinkedList`.

Différences de conception C++ vs Java

- Conteneurs d'objets ou valeurs vs collections de références :
 - les conteneurs C++ peuvent contenir des objets directement ou des valeurs de types primitifs (ou des pointeurs sur des objets/valeurs) ;
 - les collections Java contiennent uniquement des références sur les objets.
- Templates de conteneurs vs interfaces de collections :
 - le C++ utilise des templates pour écrire du code générique qui s'applique à n'importe quel type de conteneur ;
 - les collections Java par contre définissent des interfaces communes à plusieurs collections (p.ex. `List`).

Conteneurs C++ vs Collections Java (3)

C++ containers library

Sequence containers

Sequence containers implement data structures which can be accessed sequentially.

<code>array</code> (C++11)	static contiguous array (class template)
<code>vector</code>	dynamic contiguous array (class template)
<code>deque</code>	double-ended queue (class template)
<code>forward_list</code> (C++11)	singly-linked list (class template)
<code>list</code>	doubly-linked list (class template)

Associative containers

Associative containers implement sorted data structures that can be quickly searched ($O(\log n)$ complexity).

<code>set</code>	collection of unique keys, sorted by keys (class template)
<code>map</code>	collection of key-value pairs, sorted by keys, keys are unique (class template)
<code>multiset</code>	collection of keys, sorted by keys (class template)
<code>multimap</code>	collection of key-value pairs, sorted by keys (class template)

Unordered associative containers (since C++11)

Unordered associative containers implement unsorted (hashed) data structures that can be quickly searched ($O(1)$ average, $O(n)$ worst-case complexity).

<code>unordered_set</code> (C++11)	collection of unique keys, hashed by keys (class template)
<code>unordered_map</code> (C++11)	collection of key-value pairs, hashed by keys, keys are unique (class template)
<code>unordered_multiset</code> (C++11)	collection of keys, hashed by keys (class template)
<code>unordered_multimap</code> (C++11)	collection of key-value pairs, hashed by keys (class template)

Java collection framework

Collection Interfaces

The *collection interfaces* are divided into two groups. The most basic interface, `java.util.Collection`, has the following descendants:

- `java.util.Set`
- `java.util.SortedSet`
- `java.util.NavigableSet`
- `java.util.List`
- `java.util.Queue`
- `java.util.concurrent.BlockingQueue`
- `java.util.concurrent.TransferQueue`
- `java.util.Deque`
- `java.util.concurrent.BlockingDeque`

The other collection interfaces are based on `java.util.Map` and are not true collections. However, these interfaces contain *collection-view* operations, which enable them to be manipulated as collections. Map has the following offspring:

- `java.util.SortedMap`
- `java.util.NavigableMap`
- `java.util.concurrent.ConcurrentMap`
- `java.util.concurrent.ConcurrentNavigableMap`

Collection Implementations

Classes that implement the collection interfaces typically have names in the form of `<Implementation-style><Interface>`. The general purpose implementations are summarized in the following table:

General purpose implementations					
Interface	Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table + Linked List
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Queue, Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

Exemple (non générique)

```
■ LinkedList list = new LinkedList();
list.add(new Person("John"));
list.add(new Sportsman("Paul", "tennis"));
list.add("foo");
list.add(2); // Autoboxing

Iterator i = list.iterator();
while (i.hasNext()) {
    Object o = i.next();
    System.out.printf("[%s] %s\n", o.getClass(), o);
    // liaison dynamique sur toString()
}
```

■ Résultat

```
[class Person           ] Nom: John
[class Sportsman        ] Nom: Paul, Sport: tennis
[class java.lang.String ] foo
[class java.lang.Integer] 2
```

Itérations simplifiées

- Simplification du parcours des éléments d'une collection ou d'un tableau.
 - Syntaxe : `for (type element : expression) bloc`
 - Effet : pour chaque `element` de type `type` de la collection ou tableau `expression`, exécuter le bloc `bloc`.
 - Elimine l'utilisation d'itérateurs sur la collection ou d'index sur le tableau.
- Exemple, calcul de la somme des éléments d'un tableau :

```
int[] array = { 1, 2, 3, 4 }; int sum = 0;
for (int i : array)
    sum += i;
```
- Restrictions :
 - Les itérateurs (resp. index) sont toujours nécessaires pour pouvoir modifier la collection (resp. le tableau) pendant le parcours.
 - Pour pouvoir bénéficier de cette boucle `for` sur des collections définies par l'utilisateur il est nécessaire d'implémenter l'interface `Iterable` (qui requiert une méthode `public Iterator iterator()`).

Désavantage des collections non génériques

- Perte de l'information sur la classe de base des objets référencés : tous sont vus comme instances d'`Object`.
 - Transtypage des objets référencés pour pouvoir invoquer leurs méthodes.
 - **Danger** : le programmeur doit être sûr du type des objets manipulés.
 - Permet des collections hétérogènes (stockant des `String`, `Person`...).
⇒ Désambiguïsation du type des éléments par l'opérateur `instanceof`.
- Soit une collection `c` censée stocker des références sur des objets de la classe `Person` où est définie une méthode `print()` :

```
Iterator i = c.iterator();
while (i.hasNext()) {
    Object o = i.next(); // Impossible d'invoquer print() directement
    ((Person) o).print(); // Transtypage, mais si o n'est pas une
                          // Person une exception sera levée...
}
```
- Ne **jamais** utiliser de collection non générique.

Collections génériques

- La généricité permet de définir des collections (et ses itérateurs) d'un type donné (et ses sous-types) et plus seulement de type `Object`.
 - Evite l'opération de transtypage pour les collections homogènes.
 - Interdit d'insérer d'autres objets de types différents que celui désiré et ainsi évite les erreurs de transtypage (`BadCastException`).
 - ```
LinkedList<String> list = new LinkedList<String>();
list.add("One");
list.add("Two"); // list.add(new Person(...)); interdit
```
  - ```
String s = list.get(0); // et plus s = (String) list.get(0);
```
- Itérateurs :
 - ```
for (String s : list) System.out.println(s);
```
  - ```
Iterator<String> it = list.iterator();  
while (it.hasNext()) { String s = it.next(); /* ... */ }
```
- Lors de l'initialisation, il n'est pas nécessaire de respécifier le type :
 - ```
LinkedList<String> list = new LinkedList<>();
```

# Collections génériques (2)

---

- Une collection sans type explicite (p. ex. `LinkedList`) est appelée une collection de type *raw* (brut).
  - Une collection sans type explicite peut toujours être utilisée et équivaut à une collection d'`Object` (p. ex. `LinkedList<Object>`).
  - Les collections de type *raw* doivent être *évitées* à tout prix.
- Si une collection d'`Object` est souhaitée, les bonnes pratiques veulent de l'expliciter (p. ex. `LinkedList<Object>`) ; ce qui permet d'éviter de (mauvaises) surprises lors de l'exécution.
- Les collections génériques sont utilisables comme paramètres ou type de retour :
  - `public void setNames(List<String> names) { /* ... */ }`
  - `public List<String> getNames() { /* ... */ }`

# Généricité

---

- Définition d'un type (interface ou classe) générique :

```
interface List<E>
 extends Collection<E>, Iterable<E>, SequencedCollection<E>{
 void add(E x);
 E get(int index);
 Iterator<E> iterator();
 /* ... */
}
```

- Un type générique est compilé une seule fois (en un `.class`) comme pour une classe ou interface (contrairement aux templates C++).
- Le typage est vérifié à la compilation, mais est ensuite effacé (*type erasure*) et remplacé par le type `Object` ; ce qui permet d'avoir un seul fichier `.class`.
- Un type générique peut être lié à une hiérarchie : `<T extends Type>`.
  - `class PersonList<T extends Person> { ... }`  
// paramétrable avec `Person` et ses sous-classes

⇒ Permet d'utiliser le super-type (ici `Person`) et ses méthodes spécifiques.

# Généricité (2)

---

- Un type générique peut être paramétré par un ou plusieurs types :

```
class Pair<T, U>
{
 private T first;
 private U second;

 public Pair(T first, U second) {
 this.first = first; this.second = second;
 }

 public T first() { return first; }
 public U second() { return second; }

 public String toString() {
 return "(" + first + ", " + second + ")"; // toString()
 }
}

Pair<String, Integer> p1 = new Pair<>("two", 2); // autoboxing
System.out.println(p1.second() / 2); // unboxing
```

# Méthodes génériques

---

- Comme les types, les méthodes peuvent être paramétrées par un ou plusieurs types, entre `< >`.
- Le(s) type(s) ne sont pas transmis à la méthode, ils sont automatiquement déduits lors de l'invocation par le type des paramètres.
- Par exemple, dans la classe `Pair<T, U>`:

```
public <V> Pair<T, V> merge(Pair<U, V> other) {
 if (second.equals(other.first))
 return new Pair<>(first, other.second);
 else
 return new Pair<>(null, null);
}

Pair<Integer, String> pair1 = new Pair<>(10, "Key");
Pair<String, Double> pair2 = new Pair<>("Key", 3.14);

Pair<Integer, Double> result = pair1.merge(pair2);
System.out.println(result); // (10, 3.14)
```

# Exercice : SimpleList

---

- Implémentation d'une classe `SimpleList` définissant une liste générique, simplement chaînée d'objets de type `T` (paramètre) offrant les méthodes :
  - `int size()`
  - `void insert(T o)`
  - `void append(T o)`
  - `void remove(T o)`
  - `T get(int index)`
  - `String toString()`
  - `Examinator<T> examiner()`
- `Examinator` est une classe itérateur offrant les méthodes :
  - `boolean hasNext()`
  - `T next()`
  - `void remove()`
- Ne pas laisser transparaître les détails d'implémentation (encapsulation) : définir un paquetage et des attributs privés.

# Exercice : SimpleList (2)

---

- Liste : référence sur le premier élément.
- Chaque élément contient une donnée et une référence sur l'élément suivant.

```
class Element<T>
{
 T data;
 Element<T> next;
}

public class SimpleList<T>
{
 private Element<T> head;
}

public class Examiner<T>
{
 private Element<T> current;
}
```

