

IN4R21 – Java

Java en bref et Programmation Objet

Damien MASSON
<http://esiee.fr/~massond/Teaching/IN4R21>

28 mars 2011

Références

- Les cours de Rémi Forax disponibles sur <http://forax.org>

2/108

Première partie

Java en Bref

La technologie Java

En quelques mots :

- Orienté Objet
- Simple, Robuste, Dynamique et Sécurisé
- Indépendant de la Plateforme (VM)
- Semi Compilé/Semi Interprété
- Bibliothèque Importante (JDK API)

4/108

La technologie Java en 2010

Trois environnements d'exécutions différents :

- Java ME (Micro Edition) pour PDA, téléphone
- Java SE (Standard Edition) pour ordinateurs personnels
- Java EE (Entreprise Edition) pour serveurs
 - Servlet/JSP/JSTL
 - Portlet/JSF
 - JTA/JTS, JDO/EJB
 - JavaMail, etc.
- API du JDK 1.6 (environ 7000 classes) :
 - java.lang, java.lang.reflect, java.lang.annotation
 - java.util, java.util.prefs, java.util.concurrent
 - java.awt, java.applet, javax.swing
 - java.io, java.nio, java.net
 - java.beans, java.sql, javax.sql
 - etc...

5/108

Java Standard Edition

- JDK 1.0 (1995)
- JDK 1.1 (1997)
- JDK 1.2 aka Java 2 (1999)
- JDK 1.3 (2001)
- JDK 1.4 (2002)
- JDK 1.5 aka Java 5 (2004)
- JDK 1.6 aka Java 6 (2006)

Compatibilité ascendante (un code écrit avec JDK 1.0 doit fonctionner sur JRE 1.6)

6/108

Java/Open Source

- Java est OpenSource depuis novembre 2006 (2008 complètement)
- Toutes les sources sont disponibles :
<http://www.openjdk.org>
- N'importe qui peut contribuer, trouver des bugs, etc.
- Installation pour linux (ubuntu, fedora) :
<http://www.openjdk.org/install/>

7/108

Qu'est-ce que Java ?

C'est trois choses différentes :

- un langage haut niveau proche du C
- un langage pseudo assembleur le bytecode
- un programme capable d'interpréter le bytecode, la machine virtuelle

Ces trois parties de Java correspondent à trois spécifications différentes.

8/108

Programmation en C

- l'humain écrit des fichiers .c
- la **compilation** de ces fichiers produit des fichiers .o
- l'**édition de liens** entre ces fichiers et des **librairies** produit un **exécutable**

Les fichiers .c :

- fichiers textes en ASCII
- contiennent le code C qui sera converti en code assembleur
- ces fichiers sont portables à condition de respecter **une** norme (ex. ISO C 90, ...)

9/108

Programmation en Java

- l'humain écrit des fichiers .java
- la **compilation** de ces fichiers produit des fichiers .class
- un programme, la **machine virtuelle** interprète ces fichiers

Les fichiers .java :

- fichiers textes en Unicode
- contiennent le code Java qui sera converti en code *byte-code*
- ces fichiers sont portables car respectent la spécification Java

Les fichiers .class :

- contiennent le code *byte-code*, pseudo assembleur défini par la spécification Java, qui sera interprété par la machine virtuelle
- ces fichiers sont portables car respectent la spécification Java

10/108

de C à Java et vice versa

Pareil.c

```
#include <stdio.h>
/* pas de classe */
int compare(double t[],
            double e){
    int i, c=-1;
    for(i=0; i<3; i++){
        if(t[i]< e || t[i]>e)
            c=0;
        else c=1;
    }
    return c;
} /* compare() */
int main(int argc, char*
        argv[]){
    double tab[]={1.1,2.2,3.3};
    int r=compare(tab,2.2);
    printf("r=%d\n",r);
    return 0;
} /* main() */
/* pas de classe */
```

Pareil.java

```
//import java.lang.System;
class Pareil{
    static int compare(double[] t,
                       double e){
        int i, c=-1;
        for(i=0; i<3; i++){
            if(t[i]< e || t[i]>e)
                c=0;
            else c=1;
        }
        return c;
    } /* compare() */
    public static void main(String
                               [] argv){
        double[] tab={1.1,2.2,3.3};
        int r=compare(tab,2.2);
        System.out.printf("r=%d\n",r);
        // la VM s'arrete
    } /* main() */
} /* Pareil */
```

Principales différences

- En C : tout est structure
- En Java : tout est Objet
- Un fichier = une Classe
- Une classe = description d'un Objet
- Méthodes de classe (static) = fonctions du C
- Méthodes d'objets = voir cours2 "programmation Objet"
- Gestion automatique de la mémoire

12/108

Une classe ?

- c'est un "template" d'objet, cela décrit un objet
- les champs et méthodes "static" appartiennent à la classe
- les autres appartiennent aux objets instanciés à partir de la classe
- on instancie des objets à partir d'une classe grâce à `new`
- accès aux champs et méthode avec un `.`
- `Truc.field` pour accéder à un champs static

13/108

src/Example.java

```
public class Example{
    public int normalField; // default value : 0
    public static int staticField; // default value : 0
    public void normalMethod(){
        System.out.println("nm:\t"+normalField);
    }
    public void staticMethod(){
        System.out.println("sm:\t"+staticField);
    }
    public static void main(String[] argv){
        // Static field/method can be accessed without any
        // instantiation:
        Example.staticField = 3;
        Example.staticMethod();// print sm: 3
        // But normal field/method require to instantiate objects:
        Example e1 = new Example();
        Example e2 = new Example();
        e1.normalField=1;
        e2.normalField=2;
        e1.normalMethod();// print nm: 1
        e2.normalMethod();// print nm: 2
    }
    // No explicit constructor, default constructor is generated
    // public Example(){super();}
}
```

14/108

La mémoire en bref

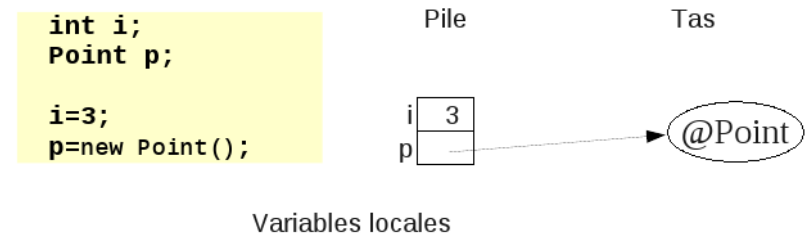
Différents types de mémoire d'un programme

- Texte (donnée read-only) : code du programme, constantes
- Globale (donnée r/w) : variables globales
- Pile : une par thread, exécute opération courante
- Tas : objets alloués (par malloc ou new)

15/108

La mémoire en bref

- En Java, les variables locales sont allouées sur la pile (ou dans les registres) et les objets dans le tas.
- Le développeur ne contrôle pas la zone d'allocation ni quand sera effectuée la désallocation.
- En C++, il est possible d'allouer des objets sur la pile, pas en Java.



16/108

Organisation des fichiers

Organisation des fichiers :

- en général, on place les .java dans un répertoire src
- en général, on place les .class dans un répertoire bin

```
moi@mamachine:~/work/java/exemple$ ls
src/ bin/
moi@mamachine:~/work/java/exemple$ ls src/
src/Programme.java
moi@mamachine:~/work/java/exemple$ ls bin/
bin/Programme.class
```

17/108

Compilation

Les sources sont compilées avec le programme javac :

- -d destination répertoire des .class générés (le compilateur crée les sous répertoires)
- -classpath chemin1:chemin2 ou -cp ... indique au compilateur où chercher les fichiers .class et .jar dont dépendent les sources
- -sourcepath chemin:...: indiquent où chercher les .java

Exemple :

```
javac -d classes -cp ../other/classes:lib/truc.jar src/fr/
upei/esiee/projet/Toto.java
```

18/108

Variables d'environnement

Java et certains programmes annexes utilisent les variables d'environnement :

- **CLASSPATH** qui correspond aux répertoires des classes (valeur par défaut de -cp/-classpath si l'option n'est pas présente)
- **JAVA_HOME** qui correspond au répertoire racine du JDK

19/108

Versions de Java

Deux options pour indiquer la version des sources (.java) que l'on a en entrée :

```
-source [1.2, 1.3, 1.4, 1.5, 1.6]
```

et la version du bytecode (.class) que l'on veut en sortie :

```
-target [1.2, 1.3, 1.4, 1.5, 1.6]
```

- Cela permet de compiler pour un JRE moins récent.
- Toutes les combinaisons ne sont pas valides (Ex : -source 1.5 implique -target 1.5)

20/108

Exécution

- démarrer une machine virtuelle : commande java
- lui demander d'interpréter le bytecode correspondant à une méthode d'une classe : on passe en paramètre le nom de la classe (et **pas le nom du fichier .class**) dont on veut exécuter la méthode main()
- option -classpath idem que pour javac

```
java -classpath bin HelloWorld
```

21/108

Classe HelloWorld

Création du projet :

```
mkdir src bin
```

Création du fichier source HelloWorld.java dans le répertoire src :

```
vim src/HelloWorld.java
```

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

Compilation pour obtenir le fichier HelloWorld.class dans le répertoire bin :

```
javac -d bin src/HelloWorld.java
```

Interprétation du bytecode par la machine virtuelle :

```
java -cp bin HelloWorld
```

22/108

Autres compilateurs

Compilateurs Java → bytecode :

- javac (livré avec Java SDK)
- Jikes (écrit en C++, IBM)
- Eclipse (ex IBM)
- ...

Mais aussi compilateurs de Java vers code machine :

- GCJ (GCC + frontend/backend Java)
- Excelsior JET
- ...

23/108

Machines virtuelles

- Hotspot (SUN) pour Windows, Linux, Solaris et MacOS
- JVM (IBM)
- JRockit (BEA) (AOT :ahead of Time compiler)
- Kaffe, SableVM, Jikes RVM
- gj (avec GCJ)
- ...

24/108

Machines virtuelles et interprétation

- La machine virtuelle interprète le byte-code
- Tant qu'il y a des instructions
 - On lit une instruction de byte-code Ex : iadd
 - On effectue l'opération correspondante
- Problème : facteur 1 à 1000 par rapport à du code compilé en assembleur.

25/108

Le JIT (Just In Time Compiler)

Pour améliorer les performances, lors de l'exécution, on transforme le byte-code en code assembleur pour la machine courante

- Avantages :
 - Exécution, pas interprétation (très rapide)
 - On adapte l'assembleur au processeur courant (P4 si c'est un P4, Turion si c'est un Turion, etc.)
- Inconvénients :
 - La transformation prend du temps (allocation des registres, inlining, déroulement des boucles etc)
 - Impossible de produire l'assembleur de tout le programme sans exploser la mémoire

26/108

Un exemple

Mesure du temps passé dans une méthode qui calcule 20 fois le n^e nombre de la suite de Fibonacci (mal codé) :

```
public class JITExample {
    private static int fibo(int n) {
        if (n==0 || n==1)
            return 1;
        return fibo(n-1)+fibo(n-2);
    }
    private static long time(int n) {
        long time=System.nanoTime();
        for(int i=0; i<20; i++)
            fibo(n);
        long time2=System.nanoTime();
        return time2-time;
    }
    public static void main(String[] args) {
        for(int i=0; i<10; i++)
            System.out.println(time(10));
    }
}
```

27/108

Un exemple

Sur une machine monoprocesseur (listing issu du cours de R. Forax) :

```
C:\eclipse\workspace\java-avance\src>java JITExample
34082
32966
31568
31568
809041
61739
5867
5587
5587
5587
```

La compilation à la volée ralentit l'exécution, mais dès que le code est compilé, remplacement de l'interprétation et accélération.

28/108

Un exemple

Sur une machine avec 2 cœurs :

```
dm@nissin:~/sync/ens/PIM-INF1$ javac -d bin src/JITExample.java
dm@nissin:~/sync/ens/PIM-INF1$ java -client -cp bin JITExample
302063
280692
278108
238438
26819
26470
26261
25981
26330
26261
```

La compilation à la volée se fait sur le second cœur, pas de ralentissement pour les exécution. Dès que le code est compilé, remplacement de l'interprétation et accélération.

29/108

Un exemple

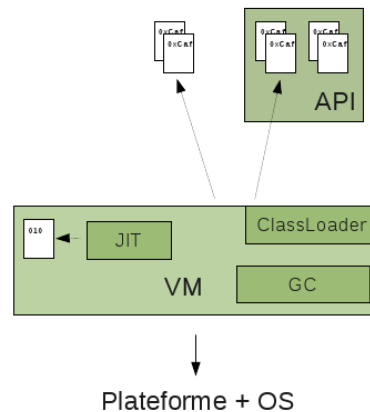
L'option `+PrintCompilation` permet d'afficher le code compilé :

```
dm@nissin:~/sync/ens/PIM-INF1$ java -XX:+PrintCompilation -client -cp bin JITExample
1      java.lang.String::hashCode (60 bytes)
2      java.lang.String::charAt (33 bytes)
3      sun.nio.cs.UTF_8$Encoder::encodeArrayLoop (490 bytes)
298711
280203
278946
278457
4      JITExample::fibonacci (25 bytes)
1162438
27518
27029
26539
27238
26959
```

30/108

Java

- Un chargeur de classe (classloader)
- Un JIT (transformation à la volée du byte-code)
- Le Garbage Collector (récupération des objets non utilisés)



31/108

Performances

- Même ordre de magnitude que le C ou le C++ (dépend de l'application)
- Théoriquement plus rapide car :
 - Compilation à l'exécution donc :
 - Optimisé en fonction de l'utilisation
 - Optimisé en fonction du processeur réel
 - Inline inter-bibliothèque
 - GC est plus efficace que malloc/free !!

http:
[//www.idiom.com/~zilla/Computer/javaCbenchmark.html](http://www.idiom.com/~zilla/Computer/javaCbenchmark.html)

32/108

Outils autour de Java

- jdb : débogueur
- jar : création d'archives éventuellement exécutables (java -jar)
- javap : décompilation de .class
- javadoc : génération de la doc automatique
- jstat, jconsole, jmap, jps, jinfo, jhat : monitoring de la machine virtuelle
- javah : génération header en C (interop avec C)
- keytool, policytool : gestion signature et sécurité
- rmiregistry, ordb, derby.jar (JavaDB) : Registry RMI, Orb CORBA, BD embarquée
- IDEs : blueJ (orienté apprentissage), Eclipse, netbeans

33/108

Plateforme Java - En résumé

- Code portable (VM)
- Syntaxe du C mais objet
- Accès à la mémoire fortement contrôlé
- (pas de pointeurs mais des références)
- Libération automatique de la mémoire (GC)
- Transformation en code machine à la volée

Et aussi :

- Exécution de morceaux de code en concurrence
- Introspection

34/108

Deuxième partie

Programmation Objet

But

Le but principal de la programmation objet est d'aider à la **conception** et la **maintenance** de logiciels. Elle apporte/facilite :

- la modularité
- l'extensibilité
- le polymorphisme
- la réutilisabilité

36/108

Un objet

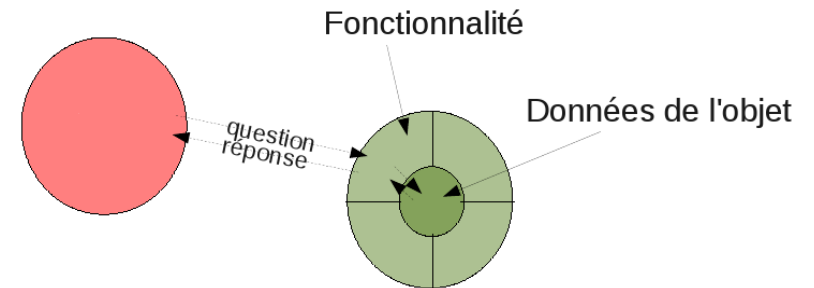
Un objet est la plus petite unité modulaire, extensible et réutilisable

- Il représente :
 - un objet physique (table, chaise, etc)
 - un concept (comparateur, factory, manager)
- Il possède :
 - des données propres qu'il ne montre à personne
 - des fonctionnalités accessibles à tous

37/108

Relations entre objets

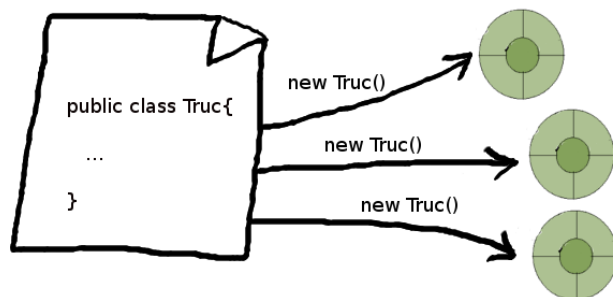
- Un objet évolue dans un monde au milieu d'autres objets
- Un objet protège son intégrité des autres objets, il n'expose que des fonctionnalités



38/108

Objets en Java

- ils sont décrits par des classes
- ils sont créés en instanciant une classe (new)



39/108

Membres d'un objet

- des champs qui définissent ses données
- des méthodes qui manipulent les données
- accès aux membres par "."

```
class Point {  
    void translate(int dx, int dy) {  
        x+=dx;  
        y+=dy;  
    }  
    int x;  
    int y;  
}
```

```
class OtherClass {  
    void otherMethod(Point p) {  
        System.out.println(p.x);  
        System.out.println(p.x);  
        p.translate(2,3);  
    }  
}
```

40/108

Les champs

- type primitif (int, short, long, char, byte, float et double)
- ou référence (pointeur sans arithmétique)

Références particulières :

- null : ne référence rien
- this : référence l'instance de la classe (l'objet) sur laquelle on invoque le code qui utilise this
- super : on verra quand on parlera d'héritage :)

41/108

Les champs

```
class Tutu {  
    void toto(){  
        Tata t1 = new Tata();  
        Tata t2 = new Tata();  
        t1.titi(t2);  
    }  
}
```

```
class Tata {  
    int titi(Tata t) {  
        if (t==null) return 1;  
        if (this==t) return 2;  
        return 3;  
    }  
}
```

Lors de l'appel `t1.titi(t2)`, `this` vaut `t1`, puisqu'on appelle la méthode `titi` sur `t1`, et `t` vaut `t2`, puisqu'on lui donne `t2` en paramètre.

42/108

1 Objet = 1 Responsabilité

- Si l'on veut réutiliser des objets il ne faut pas qu'il apporte trop de fonctionnalités
- 2 objets = 2 responsabilités, etc
- Utiliser la délégation entre objet (le fait qu'un objet en connaisse un autre) pour séparer les responsabilités

Exemple

- On veut une application qui gère les places de parking et calcule à tout instant combien le parking va rapporter si toutes les voitures sortent maintenant
- Le prix à payer dépend uniquement du type de voiture

43/108

Exemple

4 responsabilités :

- stocker les voitures
- représenter la voiture
- représenter le modèle de la voiture
- trouver le prix en fonction du modèle

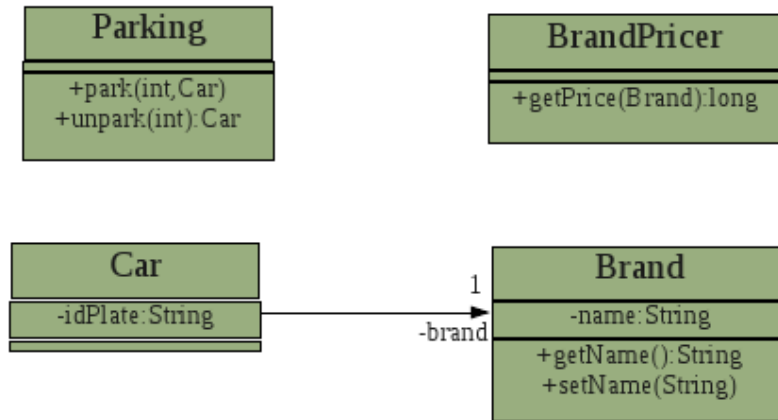
4 objets :

- Parking
- Car
- Brand
- BrandPricer

44/108

Exemple

Diagramme de classe correspondant :



Chaque objet Car possède une référence sur un objet Brand.

45/108

Encapsulation

- L'encapsulation est un principe qui garantit qu'aucun champ d'un objet ne pourra être modifié pour corrompre l'état d'un objet sans une vérification préalable
- L'état ne pourra être modifié (s'il est modifiable) que par des méthodes vérifiant les données rentrées

```
class Point {
    void init(double x, double y)
    {
        rho=Math.hypot(x,y);
        theta=Math.atan2(x,y);
    }
    double rho;
    double theta;
}
```

```
class OtherClass {
    void otherMethod(Point p) {
        p.init(2.0,3.0);
        System.out.println(
            p.theta);
        p.rho=-1; // aie aie !!
    }
}
```

- L'encapsulation n'est pas garantie par cette classe Point (on ne garantit pas $\rho \geq 0$)

46/108

Visibilité des membres

- public : tout le monde accède au membre
- $\emptyset$: les classes du même paquetage accèdent au membre
- protected : les sous-classes accèdent au membre
- private : personne n'accède au membre

Règles de bonne conduite :

- tous les champs sont **privés** : on ne modifie les données de l'objet que par des appels de méthodes, qui peuvent garantir l'encapsulation
- seules les méthodes qui le nécessitent (qui définissent une fonctionnalité externe de l'objet) sont publiques (sinon privées ou protégées)

47/108

Un meilleur Point

```
public class Point {
    public void init(double x, double y){
        rho=Math.hypot(x,y);
        theta=Math.atan2(x,y);
    }
    private double rho;
    private double theta;
}
```

```
public class OtherClass {
    public void otherMethod(Point p) {
        p.init(2.0,3.0);
        System.out.println(
            p.theta); // ne compile pas
        p.rho=-1; // ne compile pas
    }
}
```

Mais si on veut accéder/modifier les coordonnées du point ?

48/108

Accesseurs/Modificateurs

- accéder à une valeur **peut** être une **fonctionnalité** de l'objet
- changer une valeur **peut** être une **fonctionnalité** de l'objet

```
public class Point {
    public void init(double x,
                    double y){
        rho=Math.hypot(x,y);
        theta=Math.atan2(x,y);
    }
    public double getTheta(){
        return theta;
    }
    public void setRho(double d){
        if(d>=0)rho=d; //OK
    }
    private double rho;
    private double theta;
}
```

```
public class OtherClass {
    public void otherMethod(
        Point p) {
        p.init(2.0,3.0);
        System.out.println(
            p.getTheta());
        p.setRho(-1); //ne fait
                       rien
    }
}
```

Si le champ est une référence vers un objet mutable,
l'encapsulation n'est plus garantit. Voir *copie défensive*.

49/108

Vers un bon Point

- et si on veut garantir $\rho > 0$, comment empêcher d'appeler l'accesseur avant init ?

```
public class OtherClass {
    public void otherMethod(Point p) {
        System.out.println(p.theta); // 0,0
        p.init(2.0,3.0);
    }
}
```

- tout objet doit être créer en appelant un **constructeur**

50/108

Un bon Point

```
public class Point {
    public Point(double x,
                double y){
        rho=Math.hypot(x,y);
        theta=Math.atan2(x,y);
    }
    public double getTheta(){
        return theta;
    }
    public void setRho(double d){
        if(d>=0)rho=d; //OK
    }
    private double rho;
    private double theta;
}
```

```
public class OtherClass {
    public void otherMethod(){
        Point p=new Point(2.0,3.0);
        System.out.println(
            p.getTheta());
    }
}
```

51/108

Un bon Point

```
public class Point {
    public Point(double x,
                double y){
        rho=Math.hypot(x,y);
        theta=Math.atan2(x,y);
    }
    public double getTheta(){
        return theta;
    }
    public void setRho(double d){
        if(d>=0)rho=d; //OK
    }
    private double rho;
    private double theta;
}
```

```
public class OtherClass {
    public void otherMethod(){
        Point p=new Point(2.0,3.0);
        System.out.println(
            p.getTheta());
    }
}
```

```
Point p2 = new Point();
//cannot find symbol constructor Point()
```

51/108

Constructeur

- Un constructeur est une méthode particulière
- Pour garantir que les invariants d'un objet sont conservés, il faut pouvoir initialiser un objet avec des valeurs particulières
- Le constructeur va être appelé par `new` une fois la mémoire réservée pour initialiser l'objet.
- Un constructeur possède le même nom que la classe et pas de type de retour
- Si aucun constructeur n'est défini, le **compilateur** rajoute un constructeur public sans paramètre
- L'appel à un autre constructeur peut se faire avec la notation `this(...)` (doit être la première instruction)

52/108

Point final

```
public class Point {  
    public Point(double x, double y){  
        init(x,y);  
    }  
    public Point(){  
        this(0.0,0.0);  
    }  
    public double getTheta(){  
        return theta;  
    }  
    public double getRho(){  
        return rho;  
    }  
    public void init(double x, double y){  
        rho = Math.hypot(x,y);  
        theta = Math.atan2(x,y);  
    }  
    private double rho;  
    private double theta;  
}
```

53/108

final ?

- Il est possible de déclarer un champs constant (final)
- Le compilateur vérifie que la valeur est assignée une seule fois et par chaque constructeur

```
public class Point {  
    public Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
    public Point() {  
    } //variable x might not have been initialized  
    private final int x;  
    private final int y;  
}
```

54/108

Types de méthodes

- Constructeurs : Initialisent l'objet
- Accesseurs :
 - Getters (TypeX getX()) → Exportent les données (souvent partiellement)
 - Setters (void setX(TypeX value)) → Importent les données (en les vérifiant)
- Méthodes métier (*business methods*) : Effectuent des calculs en fonction des données

Les accesseurs permettent d'assurer l'encapsulation, mais aussi de changer l'implantation de l'objet sans changer le code d'appel (maintenabilité).

55/108

Types et Java

- Philosophiquement, Java sépare les types primitifs des types Objets.
 - Les types primitifs : ex. boolean, int, double, ...
 - Les types objet : ex. String, int[], Point, ...
- Les types primitifs sont manipulés par leur valeur
- Les types objets par leur référence

56/108

Surcharge de méthode

- La surcharge (*overloading*) d'une méthode consiste à fournir plusieurs définitions pour une méthode (ou un constructeur)
- Même nom, même type de retour et types des paramètres différents
- Le compilateur cherche la meilleure méthode en fonction du type des arguments

```
public class PrintStream{
    public void println(String text){
        ...
    }
    public void println(double value){
        ...
    }
}
```

```
public static void main(
    String [] args){
    PrintStream out=
        System.out;
    out.println("toto");
    out.println(3.0);
}
```

57/108

Surcharge de méthode

- La surcharge (*overloading*) d'une méthode consiste à fournir plusieurs définitions pour une méthode (ou un constructeur)
- Même nom, même type de retour et types des paramètres différents
- Le compilateur cherche la meilleure méthode en fonction du type des arguments

```
public class PrintStream{
    public void println(String text){
        ...
    }
    public void println(double value){
        ...
    }
}
```

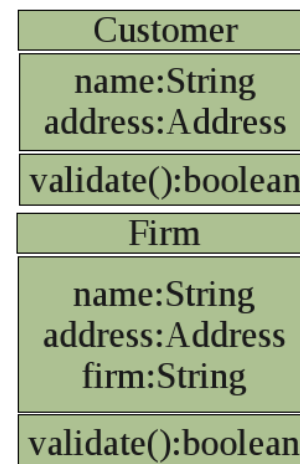
```
public static void main(
    String [] args){
    PrintStream out=
        System.out;
    out.println("toto");
    out.println(3.0);
    out.println(3); //ok
    out.println(out); //erreur
}
```

- Si aucune méthode ne possède les types exactes, le compilateur prend une méthode approchée en fonction du type des paramètres

57/108

Héritage

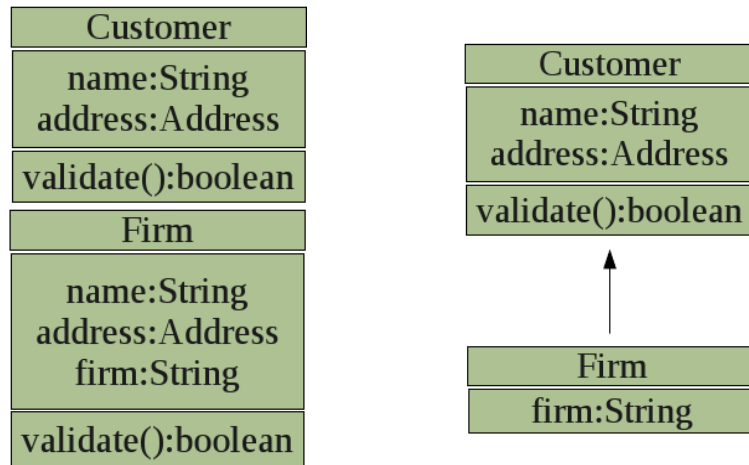
Exemple



58/108

Héritage

Exemple



58/108

Héritage

formellement

L'héritage, c'est 3 choses :

- Récupération des membres (héritage structurelle)
- Possibilité de redéfinir les méthodes
- Sous-typage

59/108

Héritage

en Java

En Java, avec le mot clé `extends` :

```
public class Firm extends Customer {
    final String firm;
}
```

```
public class Customer {
    public Customer(String name, Address address) {
        this.name=name;
        this.address=address;
    }
    public boolean validate() {
        return name!=null && address.validate();
    }
    final String name;
    final Address address;
}
```

60/108

Héritage

La classe `Object`

En Java, toutes classes héritent de `java.lang.Object`, directement ou indirectement.

- Directement : si l'on déclare une classe sans héritage, le compilateur rajoute `extends Object`
- Indirectement : si l'on hérite d'une classe celle-ci hérite de `Object` (directement ou indirectement)

61/108

Héritage

Le sous-typage

C'est le fait de pouvoir considérer une référence à une sous-classe comme étant une référence à une super-classe :

```
public static main(String[] args) {
    Firm firm=new Firm();
    System.out.println(firm);
    Customer customer=firm; // sous-typage
    System.out.println(customer);
}
```

- Si Firm hérite de Customer alors Firm est un sous-type de Customer
- Partout où on attend un objet de type Customer, il est possible de donner un objet de type Firm

62/108

Héritage

Membres

Une classe hérite de tous les membres de la super-classe :

```
public class Firm extends Customer {
    final String firm;
    public static main(String[] args) {
        Firm firm=new Firm();
        System.out.println(firm.firm+" "+
            firm.name+" "+firm.address);
    }
}
```

63/108

Héritage

Champs

Il est possible de nommer un champ de façon identique dans une classe et sa sous-classe. Les deux champs cohabitent :

```
public class Firm extends Customer {
    public int f() {
        return this.name;
    }
    final String firm;
    final int name;
}
```

```
public class Customer {
    public String g() {
        return this.name;
    }
    final String name;
    final Address address;
}
```

Le champ name de Firm masque le champ name de Customer.

64/108

Héritage

Champs

```
public class Firm extends Customer {
    public String f() {
        return super.name;
    }
    final String firm;
    final int name;
}
```

```
public class Customer {
    public String g() {
        return this.name;
    }
    final String name;
    final Address address;
}
```

Le mot clé super permet d'accéder au champ de la super classe

65/108

Héritage

Champs

```
public static main(String[] args) {
    Firm firm=new Firm();
    System.out.printf("%d",firm.name);
    Customer customer=firm; // sous-typage
    System.out.printf("%s",customer.name);
}
```

On accède au champs en fonction du type de la référence

66/108

Héritage

Constructeur

- La première instruction d'un constructeur est un appel au constructeur de la superclasse.
- Si elle n'est pas écrite, le constructeur implicite de la super classe est appelé
- idem pour le constructeur implicite de la sous classe :

```
public class Firm extends Customer {
    final String firm;
    public Firm() { // implicite
        super(); // bang
    }
} // cannot find symbol constructor Customer()
```

- Si il n'y en a pas il faut appeler explicitement un autre constructeur.

67/108

Héritage

Constructeur

- super(...) doit être la première instruction du constructeur
- Si l'instruction super(...) n'existe pas, le compilateur ajoute un appel au constructeur par défaut de la classe mère (super())
- La grosse différence entre un constructeur et une méthode, c'est que l'on n'hérite pas des constructeurs

68/108

Héritage

Constructeur

Le constructeur de la classe héritée a pour mission de :

- demander l'initialisation de la classe mère au travers de son constructeur
- d'initialiser ses propres champs

```
public class Firm extends Customer {
    public Firm(String firm, String name, Address address) {
        super(name,address); // appel a Customer(String,Address)
        this.firm=firm;
    }
    private final String firm;
}
public class Customer {
    public Customer(String name, Address address) {
        this.name=name;
        this.address=address;
    }
    private final String name; // notez les private !
    private final Address address;
}
```

69/108

Héritage

Redéfinition de méthode

```
public class Firm extends Customer {
    public Firm(String firm, String name, Address address) {
        super(name, address);
        this.firm=firm;
    }
    private final String firm;
    public static main(String[] args) {
        Address address=...
        Firm firm=new Firm("renault","Dubois",address);
        System.out.println(firm.validate()); // true
        Firm firm=new Firm(null,"Dubois",address);
        System.out.println(firm.validate()); // true
    }
}
```

validate() ne verifie pas le champs firm !

70/108

Héritage

Redéfinition de méthode

Il est possible de redéfinir le code de validate() dans Firm :

```
public class Firm extends Customer {
    public Firm(String firm, String name, Address address) {
        super(name, address);
        this.firm=firm;
    }
    public boolean validate() {
        return firm!=null && name!=null && address.validate();
    } // marche pas
    private final String firm;
}
```

71/108

Héritage

Redéfinition : @Override

L'annotation @Override indique au compilateur de générer une erreur si une méthode ne redéfinit pas une autre :

```
public class Firm extends Customer {
    public Firm(String firm, String name, Address address) {
        super(name, address);
        this.firm=firm;
    }
    public @Override boolean validate() {
        ...
    }
    private final String firm;
}
```

72/108

Héritage

Redéfinition ou surcharge ?

- La surcharge correspond à avoir des méthodes de même nom mais de profils différents dans une même classe
- La redéfinition correspond à avoir deux méthodes de même nom et de même profils dans deux classes dont l'une hérite de l'autre

```
public class A {
    public void m(int a) {...} // surcharge
    public void m(double a) {...} // surcharge
}
public class B extends A {
    public void m(long a) {...} // surcharge
    public void m(double a) {...} // redefinition
}
```

73/108

Héritage

Méthodes et sous-typage

On accède aux méthodes en fonction du type réel de la référence :

```
public static main(String[] args) {
    Firm firm=new Firm(...);
    System.out.println(firm.validate()); // Firm::validate
    ()
    Customer customer=firm; // sous-typage
    System.out.println(customer.validate()); // Firm::validate
    ()
}
```

Comme il y a redéfinition, la méthode validate() de Customer n'est pas accessible !

74/108

Héritage

Méthodes et sous-typage

La notation **super.** permet d'avoir accès aux membres non static de la superclasse :

```
public class Firm extends Customer {
    public Firm(String firm, String name, Address address) {
        super(name, address);
        this.firm=firm;
    }
    public boolean validate() {
        return firm!=null && super.validate();
    }
    private final String firm;
}
```

super.super.validate() ne marche pas.

75/108

Héritage

Méthodes et final

Si une méthode est déclarée final, celle-ci ne peut être redéfinie

- Important en terme de sécurité
- Peu voir pas d'impact en terme de performance

```
public class PasswordValidator {
    public final boolean checkPassword(char[] password) {
        ...
    }
}
public class YesValidator extends PasswordValidator {
    @Override
    public boolean checkPassword(char[] password) { //illegal
        return true;
    }
}
```

76/108

Héritage

Classes et final

- Si une classe est déclarée final, il est impossible de créer des sous-classes
- Mêmes raisons que pour les méthodes

```
public final class PasswordValidator {
    public boolean checkPassword(char[] password) {
        ...
    }
}
public class YesValidator extends PasswordValidator{//
    illegal
    ...
}
```

77/108

Héritage

Héritage multiple ?

- Problèmes de l'héritage multiple :
 - Si on hérite de deux méthodes ayant même signature dans deux super classes, quelle code choisir ?
 - Performance en cas de sous-typage
- Solution :
 - Il n'y a pas d'héritage multiple en Java
 - Java définit des interfaces et permet à une classe d'implanter plusieurs interfaces

78/108

Interfaces

Définition

- Une interface définit un type sans code
- On utilise le mot-clé **interface**

```
public interface List {  
    public Object get(int index);  
    public void set(int index, Object element);  
}
```

- Une interface déclare des méthodes sans indiquer le code (implantation) de celles-ci
- On dit alors que les méthodes sont **abstraites**

79/108

Interfaces

Instantiation

Il n'est pas possible d'instantier une interface car celle-ci ne définit pas le code de ses méthodes

```
public interface List {  
    public Object get(int index);  
    public void set(int index, Object element);  
}  
  
public class Main {  
    public static void main(String[] args) {  
        List list=  
            new List(); //illegal  
    }  
}
```

80/108

Interfaces

Implantation

- Implanter une interface consiste à déclarer un classe qui fournira le code pour **l'ensemble** des méthodes abstraites

```
public interface List {  
    public Object get(int index);  
    public void set(int index, Object element);  
}  
  
public class FixedSizeList implements List {  
    public FixedSizeList(int capacity) {  
        array=new Object[capacity];  
    }  
    public Object get(int index) {  
        return array[index];  
    }  
    public void set(int index, Object element) {  
        array[index]=element;  
    }  
    private final Object[] array;  
}
```

81/108

Interfaces

Implantation

- Implanter une interface consiste à déclarer une classe qui fournira le code pour **l'ensemble** des méthodes abstraites
- Le compilateur vérifie que toutes les méthodes de l'interface sont implantées par la classe
- Le fait qu'une classe implante une interface implique que la classe est un sous-type de l'interface

82/108

Interfaces

Héritage

- Une interface peut hériter d'une ou plusieurs interfaces
- Les méthodes de cette interface correspondent à l'union des méthodes des interfaces héritées

```
public interface Channel {  
    void close();  
}  
public interface ReadChannel extends Channel {  
    int read(byte[] buffer);  
}  
public interface WriteChannel extends Channel {  
    int write(byte[] buffer);  
}  
public interface RWChannel extends ReadChannel, WriteChannel  
{  
}
```

83/108

Interfaces

Méthodes et champs

- Les méthodes déclarées dans une interface sont obligatoirement :
 - abstraite (abstract)
 - publique (public)
 - donc : pas de méthode statique dans une interface !
- Les champs déclarés dans une interface sont obligatoirement :
 - constant (final)
 - public (public)
 - statique (static)
- Impossible d'implanter une méthode d'une interface avec une visibilité autre que public

84/108

Interfaces

Méthodes et champs

```
public class FixedSizeList implements List {  
    public FixedSizeList(int capacity) {  
        array=new Object[capacity];  
    }  
    // get(int) in FixedSizeList cannot implement  
    // get(int) in List; attempting to assign  
    // weaker access privileges; was public  
    Object get(int index) {  
        return array[index];  
    }  
    ...  
    private final Object[] array;  
}
```

85/108

Classe Abstraite

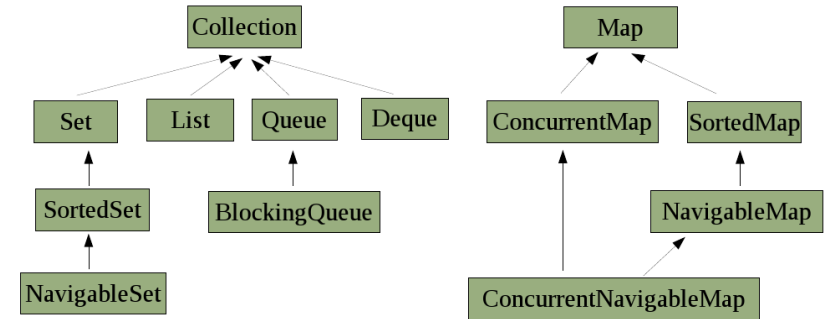
Il est possible de définir en Java des **classes** ayant des méthodes abstraites

```
public interface List {  
    public boolean isEmpty();  
    public int size();  
}  
  
public abstract class AbstractList implements List {  
    public boolean isEmpty() {  
        return size() == 0;  
    }  
}
```

- Une classe abstraite est une classe partiellement implémentée donc non instanciable
- Une classe abstraite peut s'intercaler dans l'arbre d'héritage entre l'interface et les classes concrètes
- Permet de partager du code commun à des sous-classes
- Le code commun peut supposer la présence de certaines méthodes (donc abstraites)

86/108

Collections



87/108

Collection

Définition abstraite des collections :

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

88/108

Map

Permet d'associer un objet (la clé) à un autre (la valeur) :

- **Map** : association sans relation d'ordre
- **SortedMap** : association avec clés triées
- **NavigableMap** : association avec clés triées avec suivant/précédent
- **ConcurrentMap** : association à accès concurrent

89/108

Collection<E> et types paramétrés

- Toutes les collections sont des types paramétrés par le type des éléments (E) qu'elles contiennent
- Les collections sont des conteneurs homogènes
- Si l'on veut stocker des éléments de types différents, on utilise le super-type commun, voir Object

90/108

Collections

Propriétés

Sauf exceptions :

- Toutes les collections acceptent null comme un élément valide
- Toutes les collections testent si un objet existe ou non par rapport à la méthode equals() de l'objet

91/108

Collections

Interface Collection

Interface de base des ensembles, listes et files.

- Opérations sur l'ensemble de données :
 - isEmpty/size/clear*
 - add*/remove*/contains
 - Itérateur : Iterator (hasNext,next,remove*)
- Opération Groupées (bulk) :
 - addAll*/removeAll*/retainAll*/containsAll
 - Création d'un tableau : toArray

opération* : opération optionnelle qui peut donc lever
UnsupportedOperationException

92/108

Collection

Les méthodes de Object

- Toutes les collections sauf exception testent si un objet existe en utilisant la méthode equals() de l'objet

```
public class MyPoint {  
    public MyPoint(int x, int y) {  
        this.x=x;  
        this.y=y;  
    }  
    // ne definit pas equals()  
    private final int x,y;  
    public static void main(String[] args) {  
        Collection<MyPoint> c=new ArrayList<MyPoint>();  
        c.add(new MyPoint(1,2)); // true  
        c.contains(new MyPoint(1,2)); // false  
    }  
}
```

- equals() dans Object teste l'égalité des références
- il faut donc redéfinir equals() si l'objet peut être utilisé dans une collection (donc tout le temps)

93/108

Collection

equals()

```
public class MyPoint {
    public MyPoint(int x, int y) {
        this.x=x;
        this.y=y;

        public boolean equals(MyPoint p){
            return p.x==this.x && p.y==this.y;
        }

        private final int x,y;
        public static void main(String[] args) {
            Collection<MyPoint> c=new ArrayList<MyPoint>();
            c.add(new MyPoint(1,2)); // true
            c.contains(new MyPoint(1,2)); // false
        }
    }
```

94/108

Collection

equals()

```
public class MyPoint {
    public MyPoint(int x, int y) {
        this.x=x;
        this.y=y;

        @Override
        public boolean equals(MyPoint p){
            return p.x==this.x && p.y==this.y;
        } // NE COMPILE PAS !

        private final int x,y;
        public static void main(String[] args) {
            Collection<MyPoint> c=new ArrayList<MyPoint>();
            c.add(new MyPoint(1,2));
            c.contains(new MyPoint(1,2));
        }
    }
```

95/108

Collection

equals()

```
public class MyPoint {
    public MyPoint(int x, int y) {
        this.x=x;
        this.y=y;

        @Override
        public boolean equals(Object o){
            if(o!=null && o instanceof MyPoint){
                MyPoint p = (MyPoint)o;
                return p.x==this.x && p.y==this.y;
            }
            return false;
        }

        private final int x,y;
        public static void main(String[] args) {
            Collection<MyPoint> c=new ArrayList<MyPoint>();
            c.add(new MyPoint(1,2)); // true
            c.contains(new MyPoint(1,2)); // true
        }
    }
```

96/108

Collection

Iterator

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

97/108

List

ArrayList

- Les éléments sont stockés dans un tableau redimensionnable
- Le facteur de redimensionnement est 1,5
- Pour éviter les allocations inutiles, il est intéressant de passer une taille par défaut (new ArrayList(1000))

98/108

List

LinkedList

- Liste toujours doublement chaînée
- Maintient une référence sur le début et la fin de la liste
- La taille est conservée pour éviter le parcours

99/108

Les exceptions

- Mécanisme qui permet de reporter des erreurs vers les méthodes appelantes.
- Problème en C :
 - prévoir une plage de valeurs de retour pour signaler les erreurs.
 - Propager les erreurs "manuellement"
- En Java comme en C++, le mécanisme de remonté d'erreur est gérée par le langage.

100/108

Les exceptions

Un exemple simple

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

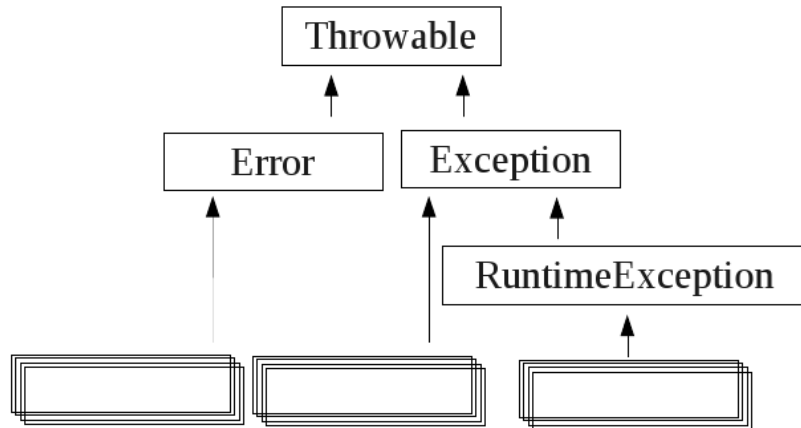
Lors de l'exécution :

```
>$java ExceptionExample  
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0  
    at ExceptionExample.main(ExceptionExample.java:18)
```

101/108

Les exceptions

Il existe 3 types d'exceptions organisés comme ceci :



102/108

Les exceptions

- Les Error correspondent à des exceptions qu'il est rare d'attraper
- Les RuntimeException que l'on peut rattraper mais on n'est pas obligé
- Les Exception :
 - soit on les attrape (**try/catch**)
 - soit on doit spécifier que la méthode appelante devra s'en occuper (**throws**)

103/108

Les exceptions

Les exceptions levées par la VM correspondent :

- erreur de compilation ou de lancement :
NoClassDefFoundError, ClassFormatError
- problème d'entrée/sortie : IOException, AWTException
- problème de ressource : OutOfMemoryError, StackOverflowError
- des erreurs de programmation (runtime) :
NullPointerException, ArrayIndexOutOfBoundsException, ArithmeticException

104/108

Les exceptions

try catch pour attraper une exception

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

105/108

Les exceptions

La directive throws

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

```
public static void f(String author) throws OhNoException {  
    if ("dan_brown".equals(author))  
        throw new OhNoException("oh_no");  
}  
public static void main(String[] args) {  
    try {  
        f(args[0]);  
    } catch (OhNoException e) {  
        tryToRecover();  
    }  
}
```

throws n'est nécessaire que pour les Exception (pas es erreurs ni les runtimes)

106/108

Les exceptions

Throws ou try/catch ?

Si l'on appelle une méthode qui lève une exception non runtime :

- try/catch si l'on peut reprendre sur l'erreur et faire quelque chose de cohérent
- Sinon throws propage l'exception vers celui qui a appelé la méthode, c'est lui qui fera ce qui doit être fait

107/108

Pour aller plus loin

- Les cours de Rémi Forax disponibles sur <http://forax.org> (encore une fois...)
- Des exercices sur <http://igm.univ-mlv.fr/ens>

108/108