

De l'esprit à la machine
L'approche Professo-Académique

Java - JavaEE

Introduction Java JEE

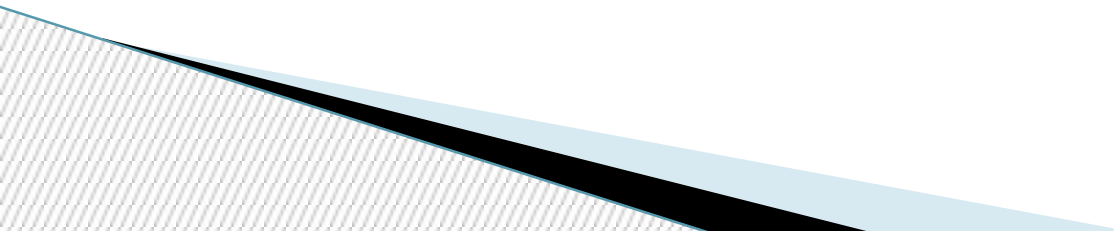
le marché actuel et les technologies demandées

Host : INPT

**Présenté par
Abdelahad SATOUR**



Plan

- ▶ Introduction à la machine Java
 - ▶ Java les concepts POO
 - ▶ Web le HTTP
 - ▶ Java EE
 - ▶ Serveur Web Serveur Applicatif
- 

1. Introduction

- ▶ Maple? Matlab? C?
- ▶ Compilateur? Interpréteur?
- ▶ Machine virtuelle? **Exemples?** JVM,ByteCode? Langage intermédiaire?

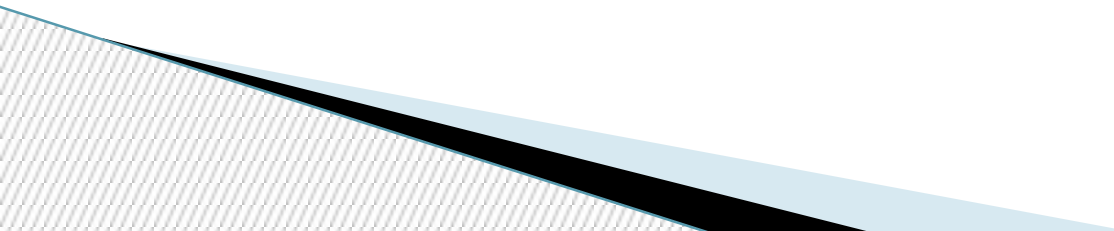
1. Introduction: **Compilateur**

- ▶ La première technique est dite **interprétation**, un programme (l'interpréteur) lit votre programme à vous et l'exécute. Cette technique tend à disparaître pour les langages d'usage général, mais elle survit bien dans des contextes plus spécialisés. Par exemple, la vaste majorité des imprimantes fabriquent les pages qu'elles impriment en interprétant un langage (le Postscript), les commandes que vous tapez en Unix sont interprétées (et exécutées) par le **shell** qui n'est rien d'autre qu'un interpréteur, on peut aussi citer **JavaScript**, interprété par les navigateurs (browsers) web, etc. L'autre technique consiste à compiler. En pratique, un compilateur lit votre programme et le transforme en un exécutable, c'est à dire quelque chose que la machine peut exécuter directement, disons une suite d'entiers que le processeur de votre machine comprend comme des instructions.

1. Introduction: Compilateur

- ▶ **L'interpréteur** doit lire les trois caractères, les transformer en un entier machine, puis ranger cet entier dans x. Le compilateur va, quant à lui, lire les caractères, les transformer en un entier machine, puis produire le code qui range l'entier dans la variable x. Dès lors, à l'exécution le programme **compilé** se contente de ranger l'entier machine dans x, soit grosso-modo une seule instruction machine, tandis que l'interpréteur doit exécuter des dizaines, voire des centaines d'instructions pour arriver au même résultat.

1. Introduction: Compilateur

- ▶ $N+1$ se compile vers N
 - ▶ Compilation: Règles algébriques etc
 - ▶ Cette idée de haut-niveau signifie que le langage source contient des constructions synthétiques, faciles à comprendre par un homme, tandis que le langage cible exprime des opérations élémentaires, faciles à réaliser par une machine.
- 

1. Introduction: Machine virtuelle

- ▶ D'une manière générale, une **Machine Virtuelle (VM)** est une implémentation logicielle d'une **architecture physique** ou **hypothétique**. Il existe deux types de machines virtuelles, **les VM système et les VM applicatives**. La JVM fait partie de la seconde catégorie. Elle est exécutée comme n'importe quelle application sur un système d'exploitation hôte et est associée à **un seul processus**. Son but est de fournir un **environnement** de programmation **indépendant** de la plate-forme qui fait **abstraction** du matériel sous-jacent et/ou du système d'exploitation, et permet à un programme de s'exécuter de la même manière sur n'importe quelle plate-forme.

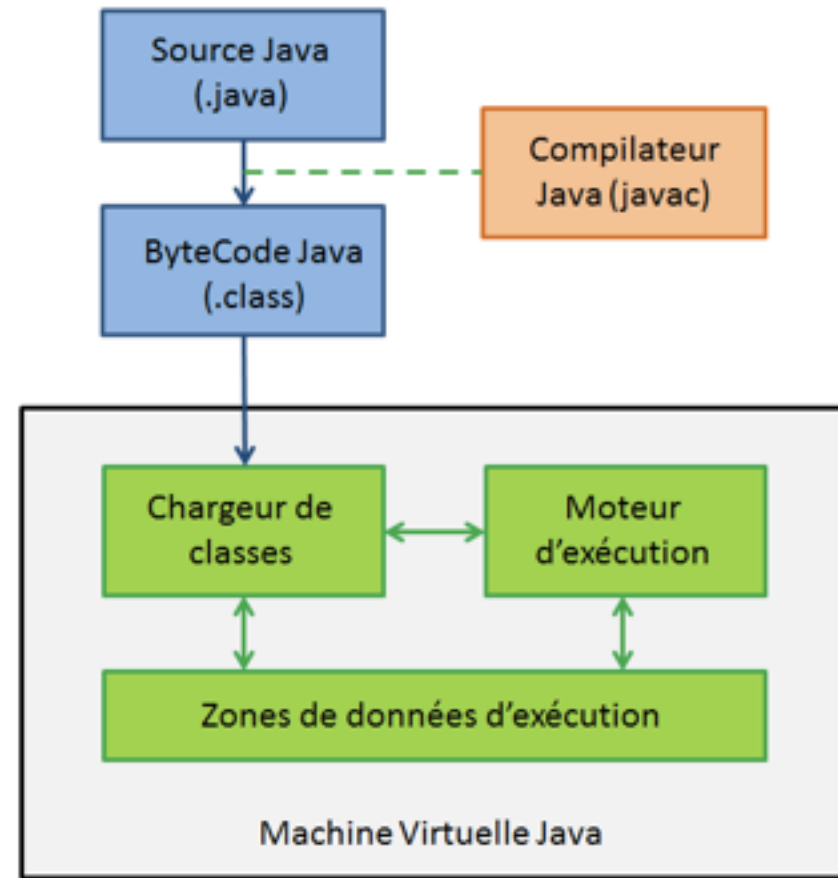
La JVM utilise un bytecode, un langage intermédiaire entre Java (le langage utilisateur) et le langage machine (Virtual machine en anglais).

1. Introduction: Machine virtuelle

- ▶ Bien que dans **JVM**, le « **J** » signifie Java, il est théoriquement possible d'implémenter un compilateur compilant n'importe quel langage en **bytecode** pour la **JVM**. Mais en pratique certains langages sont très difficiles à implémenter de manière efficiente.

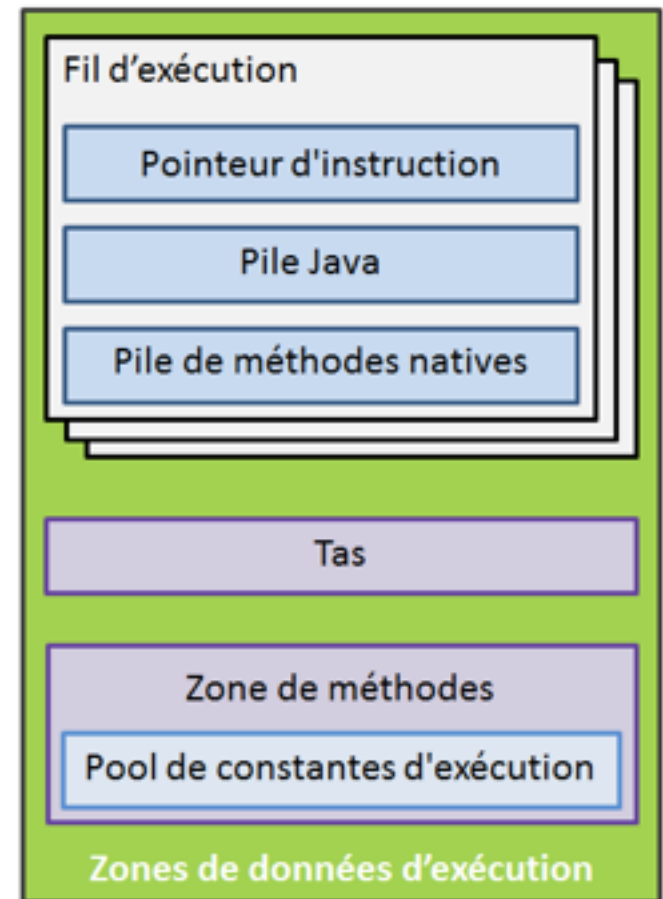
1. Introduction: Machine virtuelle

- ▶ Le JRE est composé de l'API Java et de la JVM. Le rôle de la JVM est de lire une application composée de fichiers .class sous différentes formes (zip, jar, war, un simple tableau d'octets, etc.) à l'aide d'un chargeur de classes et de l'exécuter, ainsi que l'API Java.

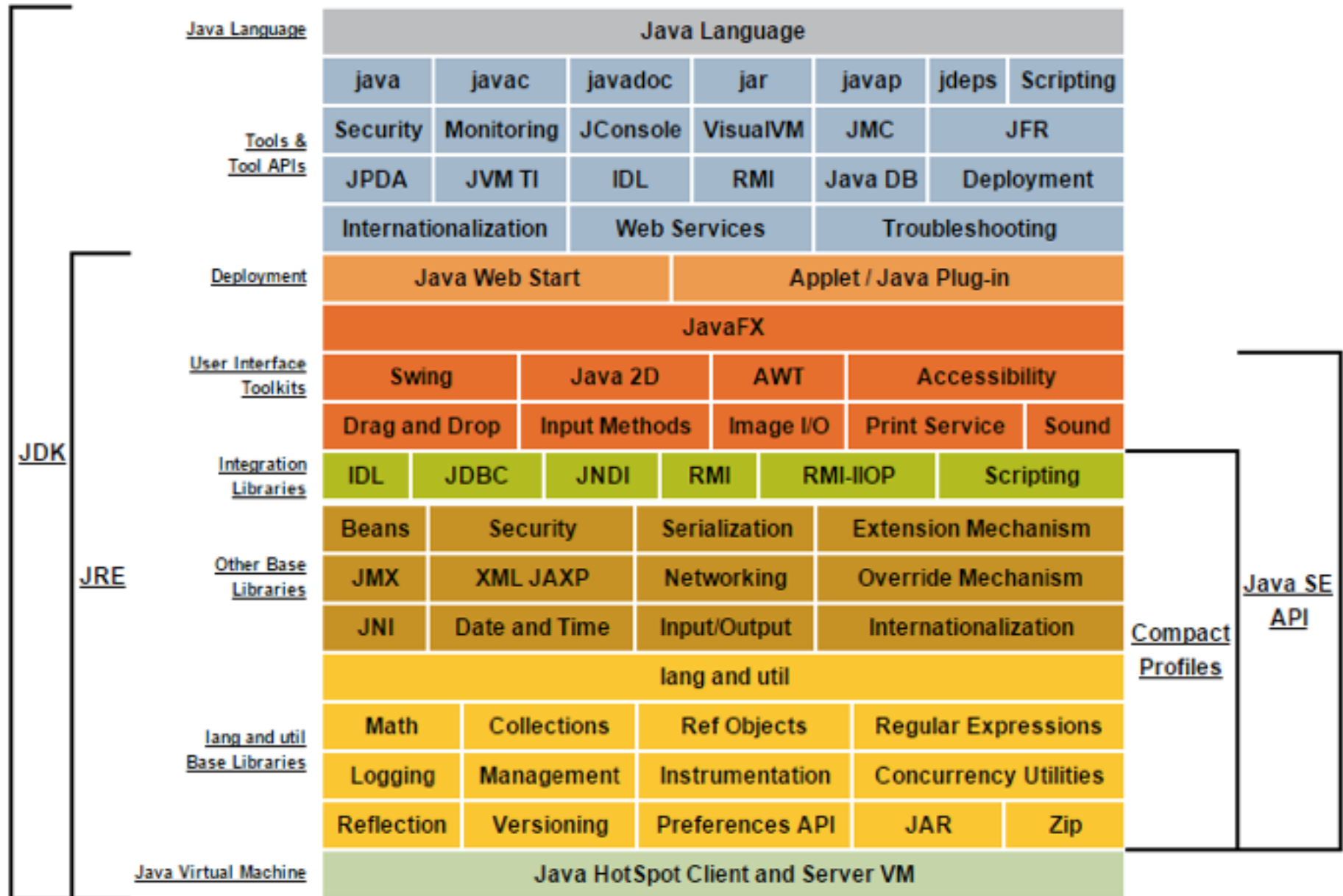


1. Introduction: Machine virtuelle

- ▶ Tas = Heap Memory
Heap(Class, Array)
Array, Class = structure



Description of Java Conceptual Diagram



1. Introduction: Machine virtuelle

```
.class org/isk/jvmhardcore/bytecode/parttwo/Adder
```

```
.method add(II)I
```

```
    iload_0
```

```
    iload_1
```

```
    iadd
```

```
    ireturn
```

```
.methodend
```

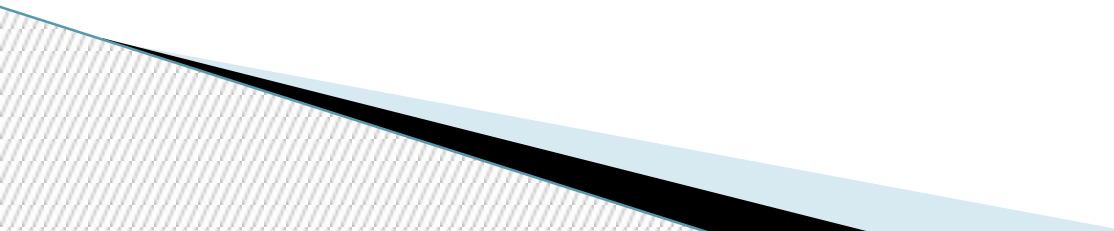
```
.classend
```



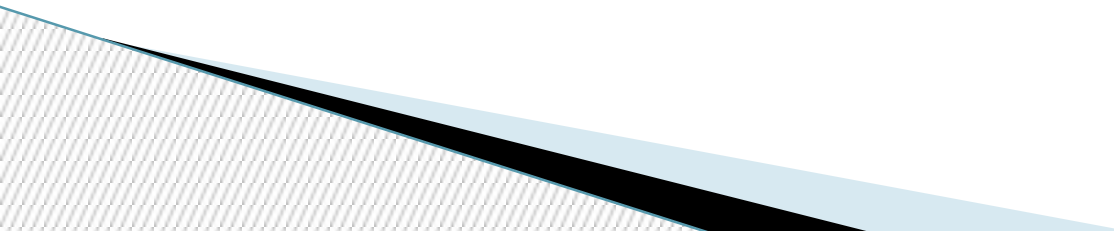
1. Introduction: Machine virtuelle

```
package org.isk.jvmhardcore.bytecode.parttwo;
```

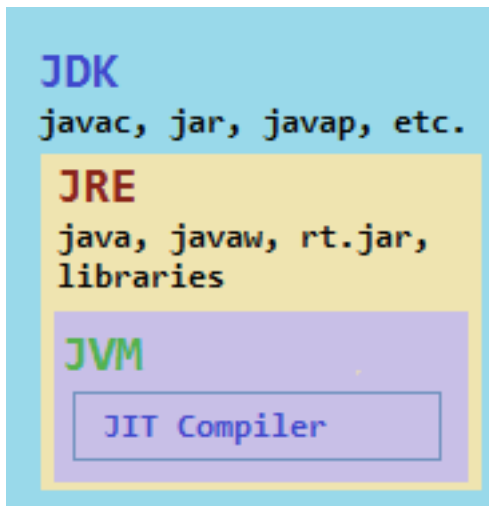
```
public class Adder {  
    public static int add (int i1, int i2) {  
        return i1 + i2;  
    }  
}
```



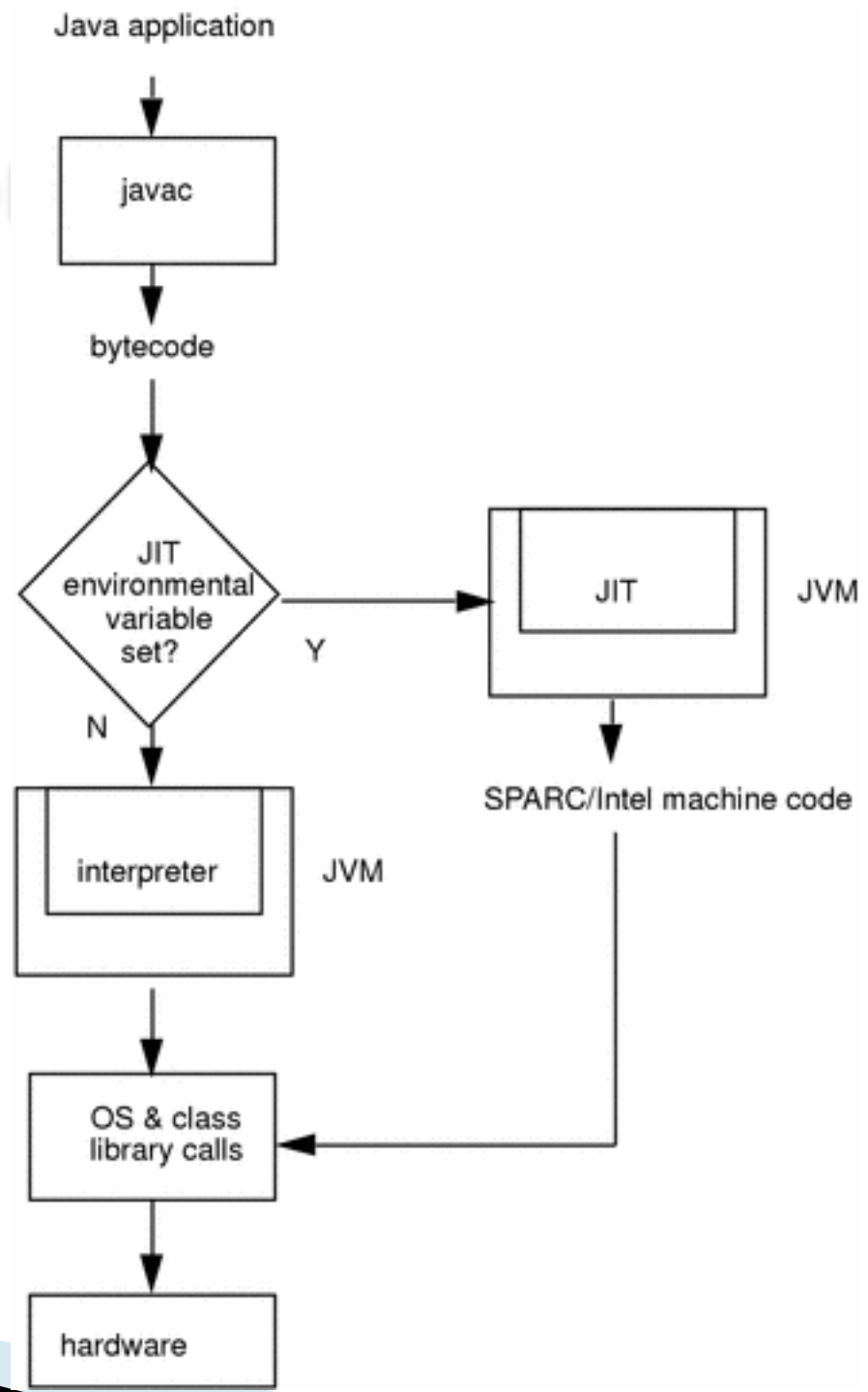
1. Introduction: Machine virtuelle

- ▶ JDK = JRE + Development/debugging tools
 - ▶ JRE = JVM + Java Packages Classes (like util, math, lang, awt, swing etc) + runtime libraries.
 - ▶ JVM = Class loader system + runtime data area + Execution Engine.
- 

1. Introduction



Compiler.disable();



1. Introduction: Machine virtuelle

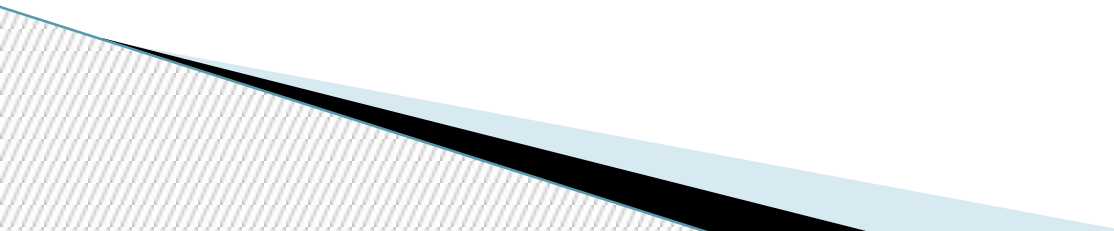
- ▶ -Djava.compiler=NONE

WHY?? disabling JIT compiler?

1. Introduction: `Javac`

- ▶ `javap java.lang.Object`
- ▶ `javac Simple.java`
- ▶ `javac -d $(pwd) Simple.java`
- ▶ `javap Simple`
- ▶ `javap -c Simple`
- ▶ `Java class` or `java class arg1 arg2`
- ▶ `jconsole pid`
- ▶ `jar cvf Simple.jar *.class`
- ▶ `jar cvfm Simple.jar META-INF/MANIFEST.MF *.class`
- ▶ `java -jar 'Simple.jar'`

Plan

- ▶ Introduction à la machine Java
 - ▶ **Java les concepts POO**
 - ▶ Web le HTTP
 - ▶ Java EE
 - ▶ Serveur Web Serveur Applicatif
- 

Java et POO

- ▶ Introduction
- ▶ Package
- ▶ Class encapsulation
- ▶ Class héritage
- ▶ Class polymorphisme
- ▶ Class static, final, abstract
- ▶ Les interfaces
- ▶ Collections

Java et POO

Historique: langages de programmations.

Première génération : langage machine

Deuxième génération 1955: langage assembleur

Troisième génération 1960 : procédural & orientés objet

Java et POO

Historique : Programmation orienté objet

Fondateurs : les Norvégiens Ole-Johan Dahl et Kristen Nygaard

Premier langage : Simula (1967)

Concept : une représentation abstraite, sous forme d'objets

Java et POO

Historique : Java

Fondateur : James Gosling

Premier lancement: 1990

Sun Microsystems:

Créé en 1982;

Une compagnie vendant des serveurs, des composants informatiques, des logiciels et des services informatiques;

Créateur du système d'exploitation SOLARIS, utilisé sur le matériel de Sun;

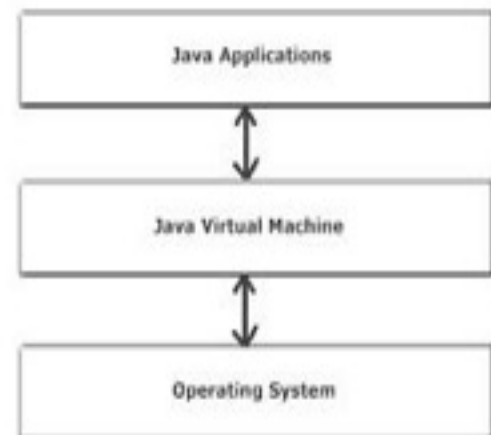
Créateur du langage de programmation Java

Oracle Corporation a acquis Sun Microsystems en 2010, afin de construire des systèmes intégrés et des solutions optimisées conçues pour atteindre des niveaux de performance très élevés dans l'industrie.

Java et POO

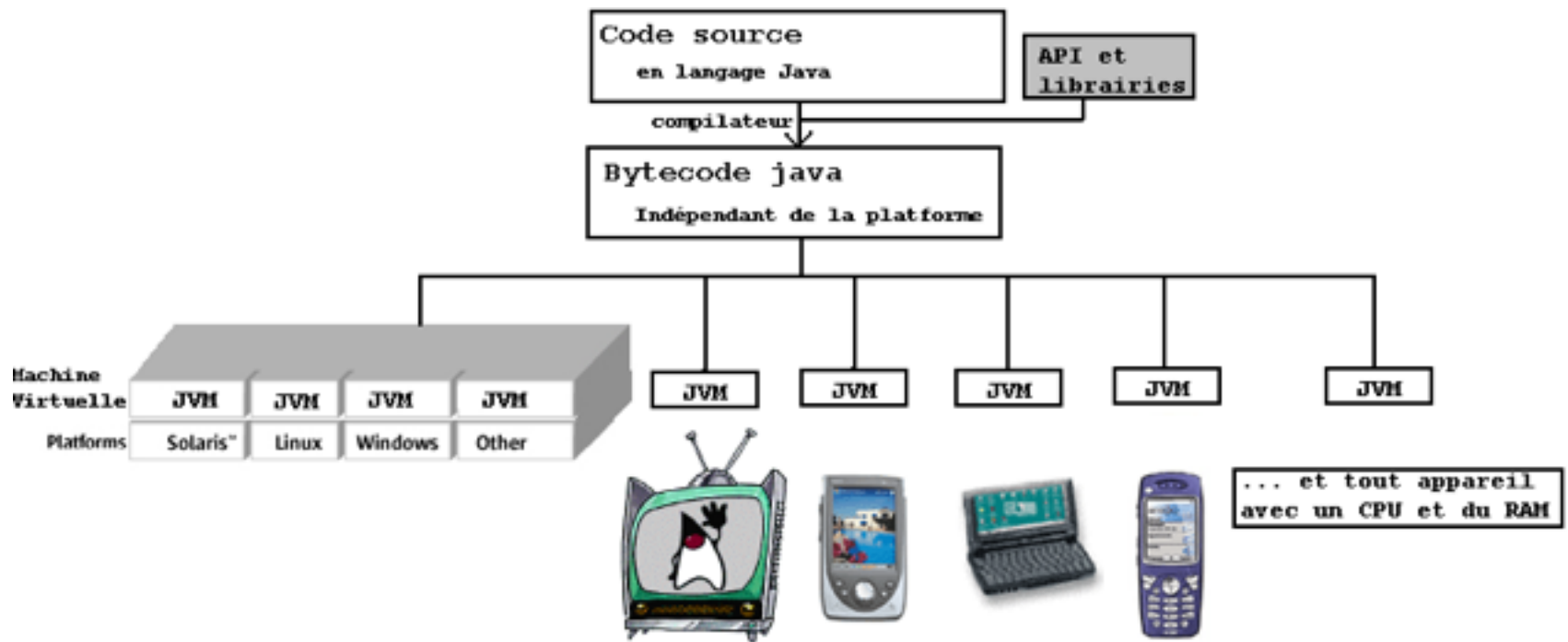
Java Virtuelle machine (JVM)

- Environnement d'exécution pour les applications Java
- Assure l'indépendance du matériel et du système d'exploitation lors de l'exécution des applications Java



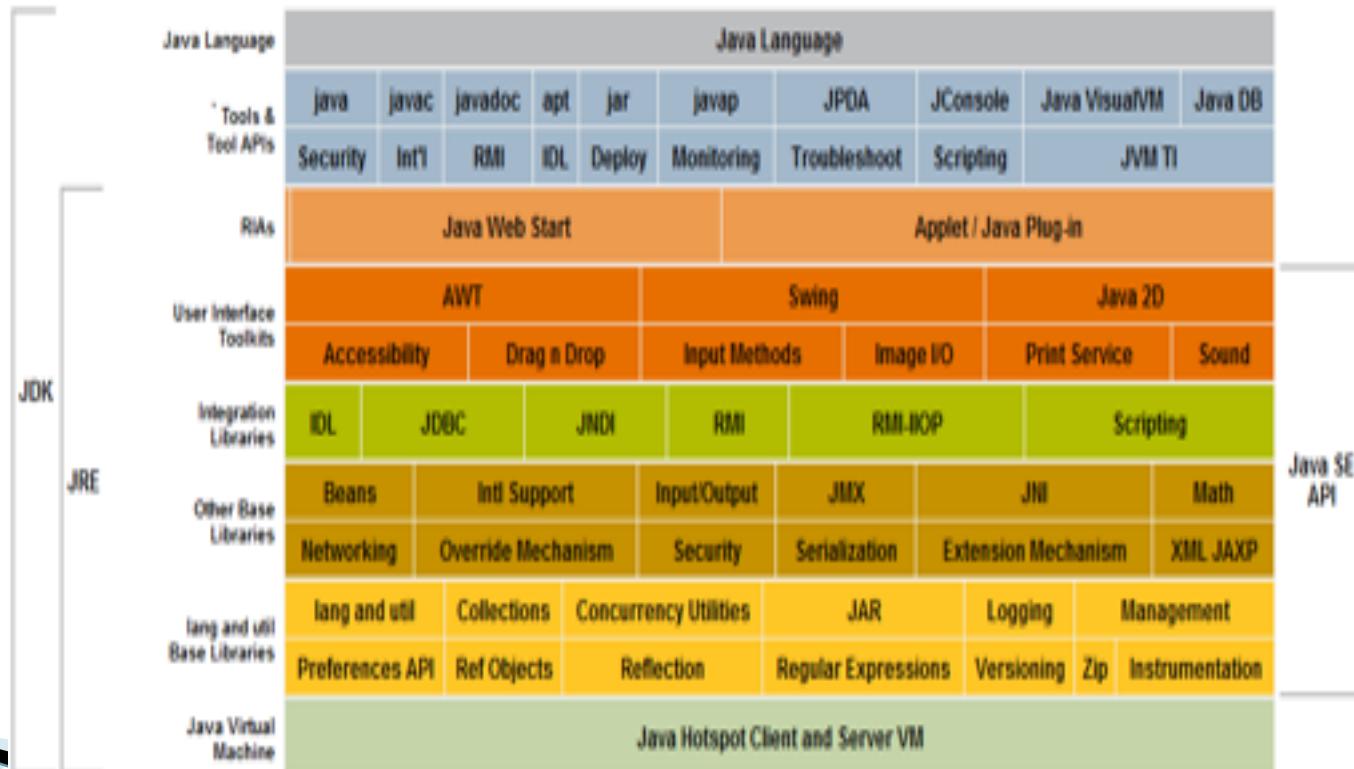
Java et POO

Java Virtuelle machine (JVM)



Java et POO

JDK



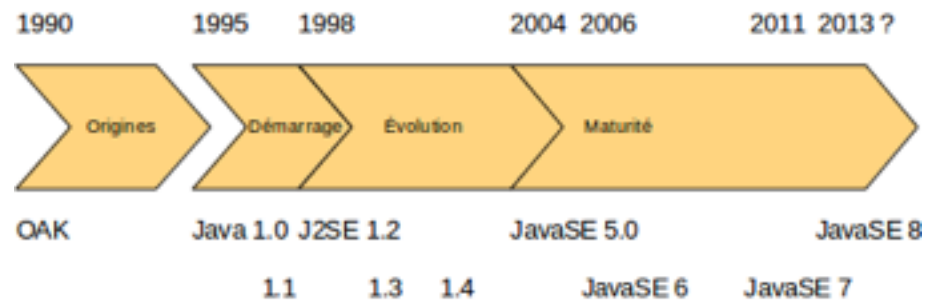
Java et POO

Gouvernance de la plateforme Java

- 97 % des bureaux d'entreprise exécutent Java
- 9 millions de développeurs Java dans le monde
- 3 milliards de téléphones mobiles exécutent Java
- 5 milliards de cartes Java utilisées
- 125 millions de périphériques TV exécutent Java
- Les 5 fabricants d'équipement d'origine principaux fournissent Java ME

Java et P00

Evolution de la plateforme Java



Java et POO

Java par rapport aux autres technologies

- Simple
- Orienté objet
- Distribuée
- Multi-thread
- Robuste
- Sécurisée
- Architecture neutre

Java et POO

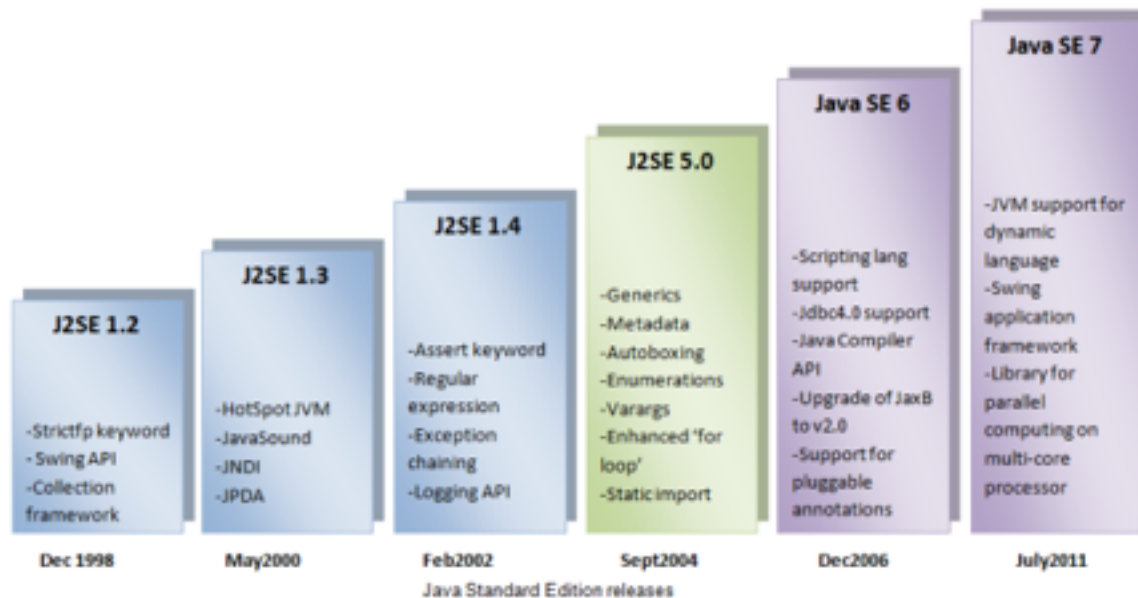
Quelles sont les caractéristiques des éditions de La plateforme Java ?

- Java Standard Edition
- Java Enterprise Edition
- Java Micro Edition

Java : une plateforme très riche qui s'avère difficile.

Java Standard Edition

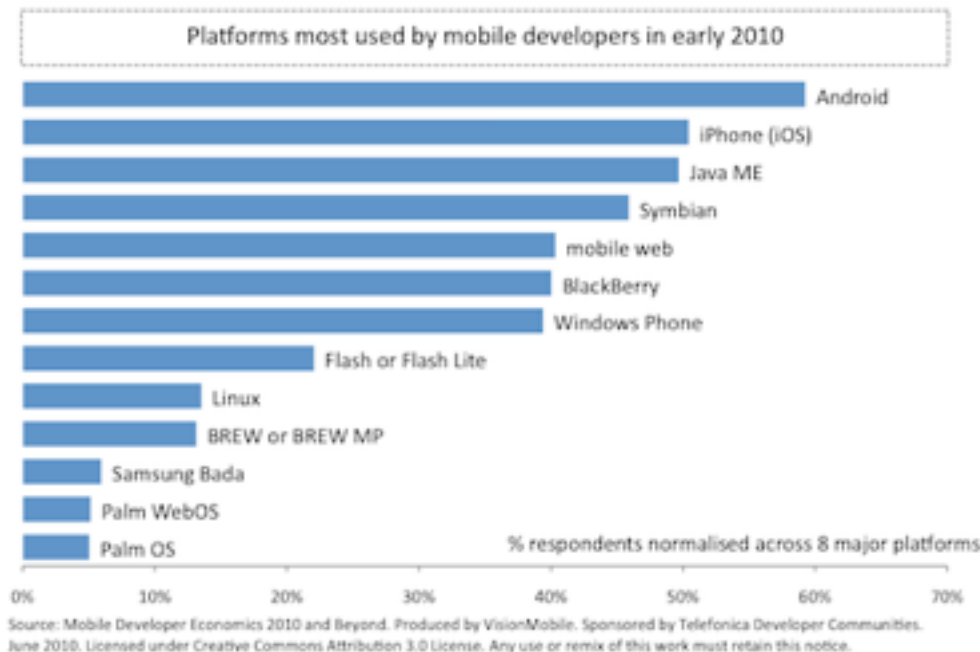
Java SE : est la plate-forme de base de Java, elle est utilisée dans le développement des applications et des applets Java. La bibliothèque standard inclue les packages tels que : `java.io`, `java.rmi` et `java.swing`



Java : une plateforme très riche qui s'avère difficile.

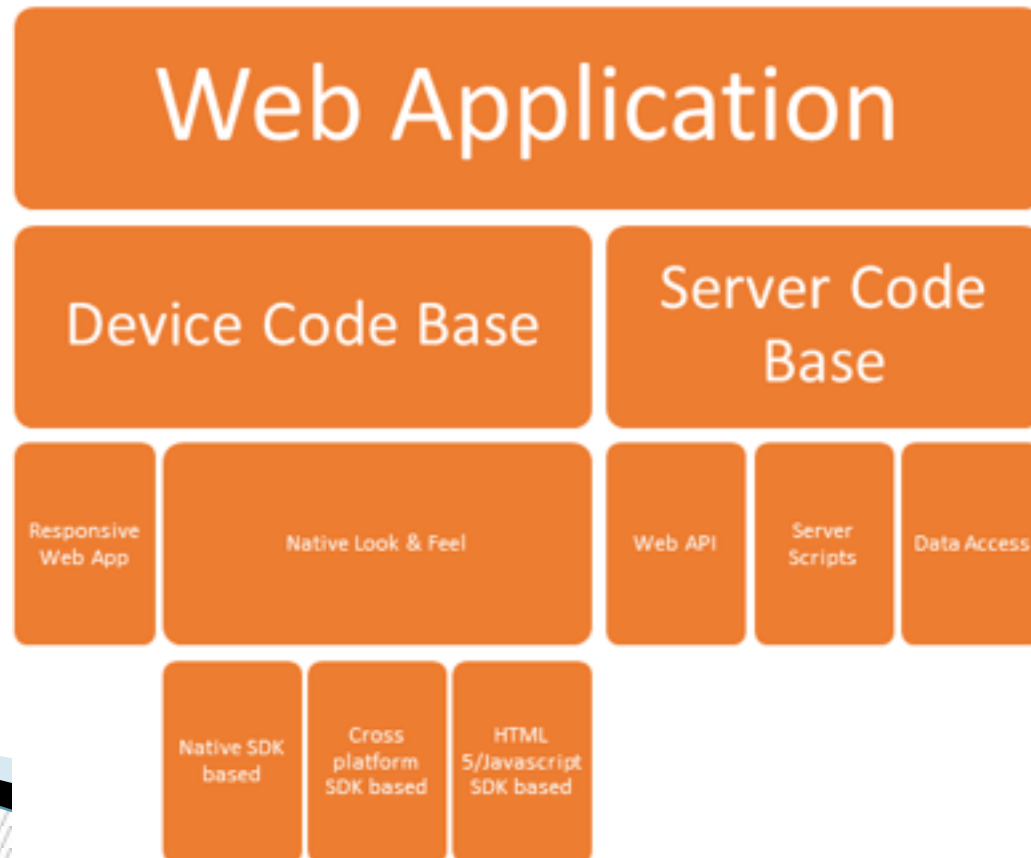
Java Micro Edition

Java ME : est un environnement d'exécution Java optimisé destiné à un usage dans les produits électroniques tels que les téléphones cellulaires



Java et POO: Java EE

Enjeux du développement d'application web



Java et POO

- ▶ Introduction
- ▶ **IDE**
- ▶ Visibilité
- ▶ Encapsulation
- ▶ Héritage
- ▶ Surcharge
- ▶ Polymorphisme
- ▶ Abstraction
- ▶ Class static, final, abstract
- ▶ Les interfaces
- ▶ Collections

2. Java & POO IDE

- ▶ Which eclipse?
- ▶ <https://eclipse.org/downloads/compare.php>

2. Java & POO IDE

RELEASES

Oxygen Packages
Neon Packages
Mars Packages
Luna Packages
Kepler Packages
Juno Packages
Indigo Packages
Helios Packages
Galileo Packages
Ganymede Packages
All Releases



Eclipse IDE for Java EE Developers

Package Description

Tools for Java developers creating Java EE and Web applications, including a Java IDE, tools for Java EE, JPA, JSF, Mylyn, EGit and others.

This package includes:

- Data Tools Platform
 - Eclipse Git Team Provider
 - Eclipse Java Development Tools
 - Eclipse Java EE Developer Tools
 - JavaScript Development Tools
 - Maven Integration for Eclipse
 - Mylyn Task List
 - Eclipse Plug-in Development Environment
 - Remote System Explorer
 - Code Recommenders Tools for Java Developers
 - Eclipse XML Editors and Tools
- Detailed features list

Download Links

Windows 32-bit
Windows 64-bit
Mac OS X (Cocoa) 64-bit
Linux 32-bit
Linux 64-bit

Downloaded 484,713 Times

► Checksums...

Bugzilla

► Open Bugs: 60

► Resolved Bugs: 143

File a Bug on this Package

2. Java & POO IDE

Releases

Since 2006, the Eclipse Foundation has coordinated an annual [Simultaneous Release](#). Each release includes the Eclipse Platform as well as a number of other Eclipse projects. Until the Galileo release, releases were named after the moons of the solar system.

So far, each Simultaneous Release has occurred at the end of June.

Release	Main Release	Platform version	Projects
Oxygen	June 2017		
Neon	22 June 2016	4.6	
Mars	24 June 2015	4.5	Mars Projects
Luna	25 June 2014	4.4	Luna Projects
Kepler	26 June 2013	4.3	Kepler Projects
Juno	27 June 2012	4.2	Juno Projects
Indigo	22 June 2011	3.7	Indigo projects
Helios	23 June 2010	3.6	Helios projects
Galileo	24 June 2009	3.5	Galileo projects
Ganymede	25 June 2008	3.4	Ganymede projects
Europa	29 June 2007	3.3	Europa projects
Callisto	30 June 2006	3.2	Callisto projects
Eclipse 3.1	28 June 2005	3.1	
Eclipse 3.0	28 June 2004	3.0	

Java et POO

- ▶ Introduction
- ▶ IDE
- ▶ **Visibilité**(+Class=Type, vs Value= Obj)
- ▶ Encapsulation
- ▶ Héritage
- ▶ Surcharge
- ▶ Polymorphisme
- ▶ Abstraction
- ▶ Class static, final, abstract
- ▶ Les interfaces
- ▶ Collections

Java et POO

Modificateur	Rôle
public	Une variable, méthode ou classe déclarée public est visible par tous les autres objets. Depuis la version 1.0, une seule classe public est permise par fichier et son nom doit correspondre à celui du fichier. Dans la philosophie orientée objet aucune donnée d'une classe ne devrait être déclarée publique : il est préférable d'écrire des méthodes pour la consulter et la modifier
par défaut : package friendly	Il n'existe pas de mot clé pour définir ce niveau, qui est le niveau par défaut lorsqu'aucun modificateur n'est précisé. Cette déclaration permet à une entité (classe, méthode ou variable) d'être visible par toutes les classes se trouvant dans le même package.
protected	Si une méthode ou une variable est déclarée protected, seules les méthodes présentes dans le même package que cette classe ou ses sous-classes pourront y accéder. On ne peut pas qualifier une classe avec protected.
private	C'est le niveau de protection le plus fort. Les composants ne sont visibles qu'à l'intérieur de la classe : ils ne peuvent être modifiés que par des méthodes définies dans la classe et prévues à cet effet. Les méthodes déclarées private ne peuvent pas être en même temps déclarées abstract car elles ne peuvent pas être redéfinies dans les classes filles.

Java et POO

- ▶ Introduction
- ▶ IDE
- ▶ Visibilité
- ▶ **Encapsulation**
- ▶ Héritage
- ▶ Surcharge
- ▶ Polymorphisme
- ▶ Abstraction
- ▶ Class static, final, abstract
- ▶ Les interfaces
- ▶ Collections

Java et POO encapsulation

- ▶ **L'encapsulation** permet de **sécuriser** l'accès aux données d'une classe. Ainsi, les données déclarées private à l'intérieur d'une classe ne peuvent être accédées et modifiées que par des méthodes définies dans la même classe. Si une autre classe veut accéder aux données de la classe, l'opération n'est possible que par l'intermédiaire d'une méthode de la classe prévue à cet effet. Ces appels de méthodes sont appelés « échanges de messages ».
- ▶ Un **accesseur** est une méthode publique qui donne l'accès à une variable d'instance privée. Pour une variable d'instance, il peut ne pas y avoir d'accesseur, un seul accesseur en lecture ou un accesseur en lecture et un autre en écriture. Par convention, les accesseurs en lecture commencent par get et les accesseurs en écriture commencent par set.

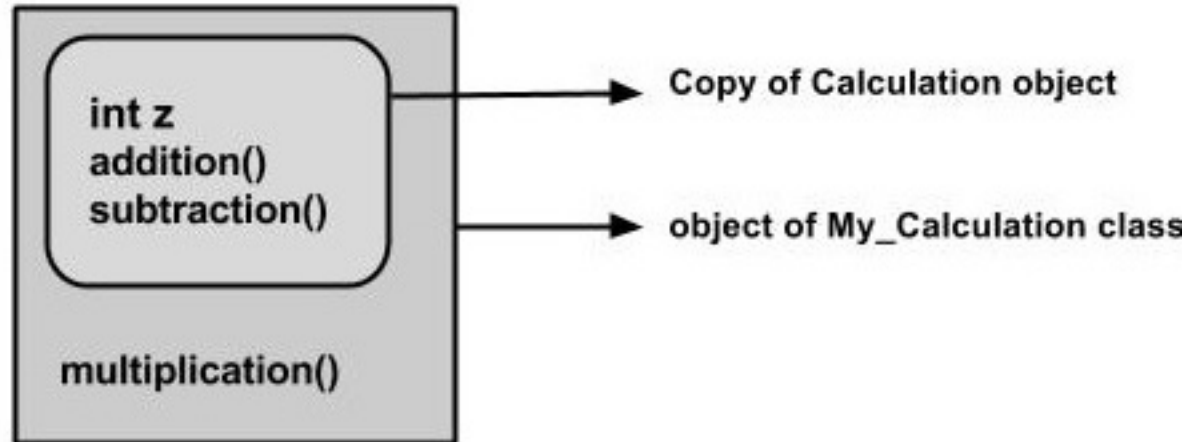
Java et POO encapsulation

- ▶ Cas réel d'échec?!
- ▶ Patterns That Provide Forms of Encapsulation
 - ▶ Facade
 - ▶ Strategy
- ▶ Patterns That "Violate" Encapsulation
 - ▶ Observer
 - ▶ Iterator
 - ▶ Memento

Java et POO

- ▶ Introduction
- ▶ IDE
- ▶ Visibilité
- ▶ Encapsulation
- ▶ **Héritage**
- ▶ Surcharge
- ▶ Polymorphisme
- ▶ Abstraction
- ▶ Class static, final, abstract
- ▶ Les interfaces
- ▶ Collections

Java et POO Héritage



Java et POO Héritage

- ▶ Exemple de surcharge
- ▶ Super a = new Parent()

Java et POO

- ▶ Introduction
- ▶ IDE
- ▶ Visibilité
- ▶ Encapsulation
- ▶ Héritage
- ▶ **Surcharge**
- ▶ Polymorphisme
- ▶ Abstraction
- ▶ Class static, final, abstract
- ▶ Les interfaces
- ▶ Collections

Java et POO Surcharge

```
public String ouSuisJe() {  
    return "Je suis dans A !" ;  
}  
  
public int ouSuisJe(int a) {  
    return 100;  
}  
  
public String ouSuisJe(String a) {  
    return "Je suis dans A !" ;  
}
```

Java et POO Redéfinition

```
public void demanderFormation(String dateDebut, String dateFin) {  
    System.out.println("Monsieur " + getPrenom() + " " + getNom()  
        + " pose une formation du " + dateDebut + " au " + dateFin);  
    System.out.println("Etant manager, il valide ses propres formations");  
}
```

```
public void demanderFormation(String dateDebut, String dateFin) {  
    super.demanderFormation(dateDebut, dateFin);  
    System.out.println("Etant manager, il valide ses propres formations");  
}
```

Java et POO

- ▶ Introduction
- ▶ IDE
- ▶ Visibilité
- ▶ Encapsulation
- ▶ Héritage
- ▶ Surcharge
- ▶ **Polymorphisme**
- ▶ Abstraction
- ▶ Class static, final, abstract
- ▶ Les interfaces
- ▶ Collections

Java et POO Polymorphisme

Overloading et overriding

```
public class A { // déclarée dans le fichier A.java
    public void ouSuisJe() {
        System.out.println("Je suis dans A !") ;
    }
}

public class B extends A { // déclarée dans le fichier B.java
    public void ouSuisJe() {
        System.out.println("Je suis dans B...") ;
    }
}

public class Main { // déclarée dans le fichier Main.java

    public static void main(String... args) {
        A a = new A() ; // mêmes déclarations que dans l'exemple précédent
        B b = new B() ;
        A ba = b ;

        System.out.println("a.ouSuisJe ? " + a.ouSuisJe()) ;
        System.out.println("b.ouSuisJe ? " + b.ouSuisJe()) ;
        System.out.println("ba.ouSuisJe ? " + ba.ouSuisJe()) ;
    }
}
```

Java et POO

- ▶ Introduction
- ▶ IDE
- ▶ Visibilité
- ▶ Encapsulation
- ▶ Héritage
- ▶ Surcharge
- ▶ **Polymorphisme**
- ▶ Abstraction
- ▶ Class static, final, abstract
- ▶ Les interfaces
- ▶ Collections

Java et POO

- ▶ Introduction
- ▶ IDE
- ▶ Visibilité
- ▶ Encapsulation
- ▶ Héritage
- ▶ Surcharge
- ▶ Polymorphisme
- ▶ **Abstraction**
- ▶ Class static, final, abstract
- ▶ Les interfaces
- ▶ Collections

Java et POO

- ▶ Introduction
- ▶ IDE
- ▶ Visibilité
- ▶ Encapsulation
- ▶ Héritage
- ▶ Surcharge
- ▶ Polymorphisme
- ▶ Abstraction
- ▶ **Class static, final, abstract**
- ▶ Les interfaces
- ▶ Collections

Java et POO Final

```
public final class FinalClass { // déclarée dans le fichier A.java, ne peut être étendue
    public void ouSuisJe() {
        System.out.println("Je suis dans A !") ;
    }
}

class Z { // déclarée dans B.java
    public final void ouSuisJe() { // ne peut être surchargée, bien que
        // B puisse être étendue
        System.out.println("J'y suis j'y reste !") ;
    }
}
```

Java et POO static

```
public final class FinalClass { // déclarée dans le fichier A.java, ne peut être étendue
    public void ouSuisJe() {
        System.out.println("Je suis dans A !") ;
    }
}

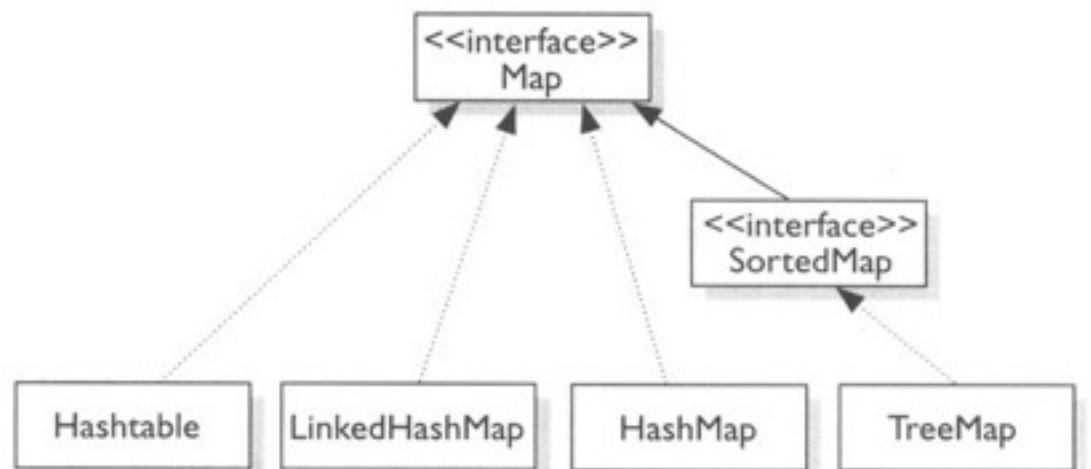
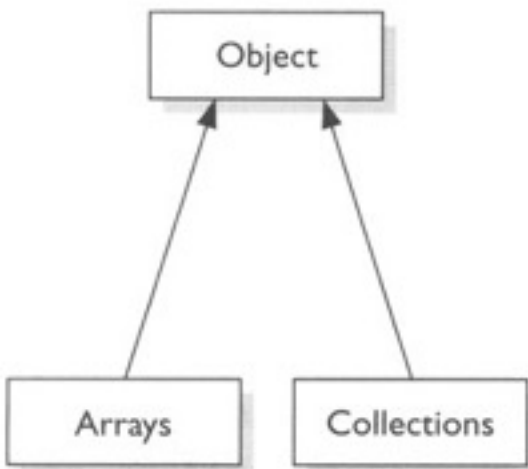
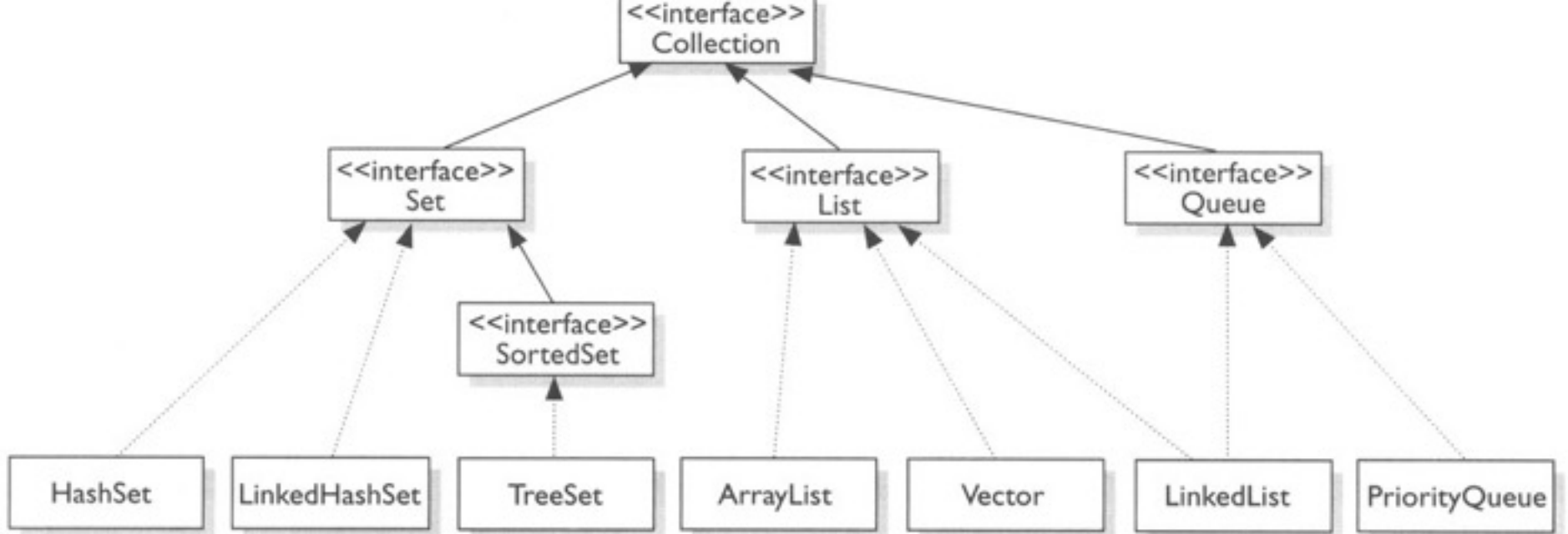
class Z { // déclarée dans B.java
    public final void ouSuisJe() { // ne peut être surchargée, bien que
        // B puisse être étendue
        System.out.println("J'y suis j'y reste !") ;
    }
}
```

Java et POO

- ▶ Introduction
- ▶ IDE
- ▶ Visibilité
- ▶ Encapsulation
- ▶ Héritage
- ▶ Surcharge
- ▶ Polymorphisme
- ▶ Abstraction
- ▶ Class static, final, abstract
- ▶ **Les interfaces**
- ▶ Collections

Java et POO

- ▶ Introduction
- ▶ IDE
- ▶ Visibilité
- ▶ Encapsulation
- ▶ Héritage
- ▶ Surcharge
- ▶ Polymorphisme
- ▶ Abstraction
- ▶ Class static, final, abstract
- ▶ Les interfaces
- ▶ **Collections**



implements

extends

ではまた。