



Introduction à Java

Aurince AKAKPO
Ronald HOUNKPONOU

Plan du cours (1/4)

1. Introduction générale et historique

- Le langage de programmation Java
- La plateforme Java

2. Première application en Java

3. Syntaxe et sémantique de Java

- Identificateurs
- Types primitifs et types de référence
- Tableaux et chaînes de caractères
- Arithmétique et opérateurs
- Instructions de contrôle

4. Premiers concepts de l'orienté objets

- Programmation procédurale v/s Programmation OO
- Concepts de l'OO
- Représentations UML

5. Programmation orientée objets en Java

- La création d'objets: Constructeurs et mot-clé « **new** »
- Les variables: Déclaration et portée
- Les méthodes: Déclaration, interface et surcharge
- L'encapsulation: « **public** », « **private** » et « **protected** »
- Les membres d'instance et de classe: « **static** »
- Utilisation de l'héritage: « **this** » et « **super** »
- Conversion de types
- Polymorphisme
- Classes abstraites
- Interfaces

Plan du cours (3/4)

- 6. Structure des API de Java
 -
- 7. La gestion des Exceptions
 - Principes et hiérarchie des classes d'exceptions
 - Interception par bloc try – catch – finally
 - Lancement par mots-clés throws et throw
- 8. Les collections
 - Interfaces
 - Implémentations □ « HashSet », « TreeSet », « ArrayList », « LinkedList »
 - Algorithmes
 -

Plan du cours (4/4)

9. Interfaces graphiques et Gestion des événements

- La philosophie Model-View-Control
- Interfaces graphiques: SWING
- Gestion d'événements

10. Multithreading

- Définition
- Raison d'être
- Création
- Gestion

11. Annexes

- Java et ses concurrents
- Les utilitaires de Java

Présentations

- Nom
- Activité / Centres d'intérêt
- Expérience de programmation
- Motivations pour suivre le cours / Intérêt pour Java
- Attentes par rapport au cours

Objectifs du cours (1/2)

- Décrire les éléments-clé de la plate-forme Java
- Compiler et exécuter une application Java
- Comprendre et utiliser la documentation en ligne de Java
- Décrire la syntaxe du langage
- Comprendre le paradigme OO et utiliser Java pour le mettre en œuvre
- Comprendre et utiliser les exceptions
- Développer une interface graphique
- Décrire les éléments de l'AWT

Objectifs du cours (2/2)

- Développer un programme qui utilise une interface graphique pour introduire les données à traiter
- Comprendre la gestion des événements
- Comprendre un diagramme de classes et de séquences (UML)
- Lire et écrire sur des fichiers
- Comprendre les bases du *multithreading*
- Développer des applications *multiprocess*

Quelques sujets non couverts

- Développement des applets
- Le package Swing
- Développement d'application clients/serveur
 - TCP/IP
 - UDP
- Enterprise Java Beans (EJB)
- Servlets et Java Server Pages (JSP)
- Connections à des bases de données (JDBC)

Références Web

- The Java Tutorial from Sun
<http://java.sun.com/docs/books/tutorial/>
- The Java Developer Connection
<http://developer.java.sun.com/developer/index.html>

Introduction à Java

I. Introduction et historique

Survol du chapitre

- Qu'est-ce que Java ?
- Java comme langage de programmation
- La plateforme Java
 - La Java Virtual Machine
 - Les interfaces de programmation d'application (API)
- Déploiement d'un programme
- Quelques notions historiques

Qu'est-ce que Java ?

- Java est un langage de programmation
 - Un programme Java est compilé et interprété
- Java est une plateforme
 - La plateforme Java, uniquement software, est exécutée sur la plateforme du système d'exploitation
 - La « Java Platform » est constituée de :
 - La « Java Virtual Machine » (JVM)
 - Des interfaces de programmation d'application (Java API)

Java comme langage de programmation

Java est un langage de programmation particulier qui possède des caractéristiques avantageuses:

- Simplicité et productivité:
 - Intégration complète de l'OO
 - Gestion mémoire (« Garbage collector »)
- Robustesse, fiabilité et sécurité
- Indépendance par rapport aux plateformes
- Ouverture:
 - Support intégré d'Internet
 - Connexion intégrée aux bases de données (JDBC)
 - Support des caractères internationaux
- Distribution et aspects dynamiques
- Performance

Java comme langage de programmation

Simple et orienté objet

- Java est un langage de programmation simple
 - Langage de programmation au même titre que C/C++/Perl/Smalltalk/Fortran mais plus simple
 - Les aspects fondamentaux du langage sont rapidement assimilés
- Java est orienté objet :
 - La technologie OO après un moment de gestation est maintenant complètement intégrée
 - En java, tout est un objet (à la différence du C++ par ex.)
- Simple aussi parce qu'il comporte un grand nombre d'objets prédéfinis pour l'utilisateur
- Java est familier pour les programmeurs C++

Java comme langage de programmation

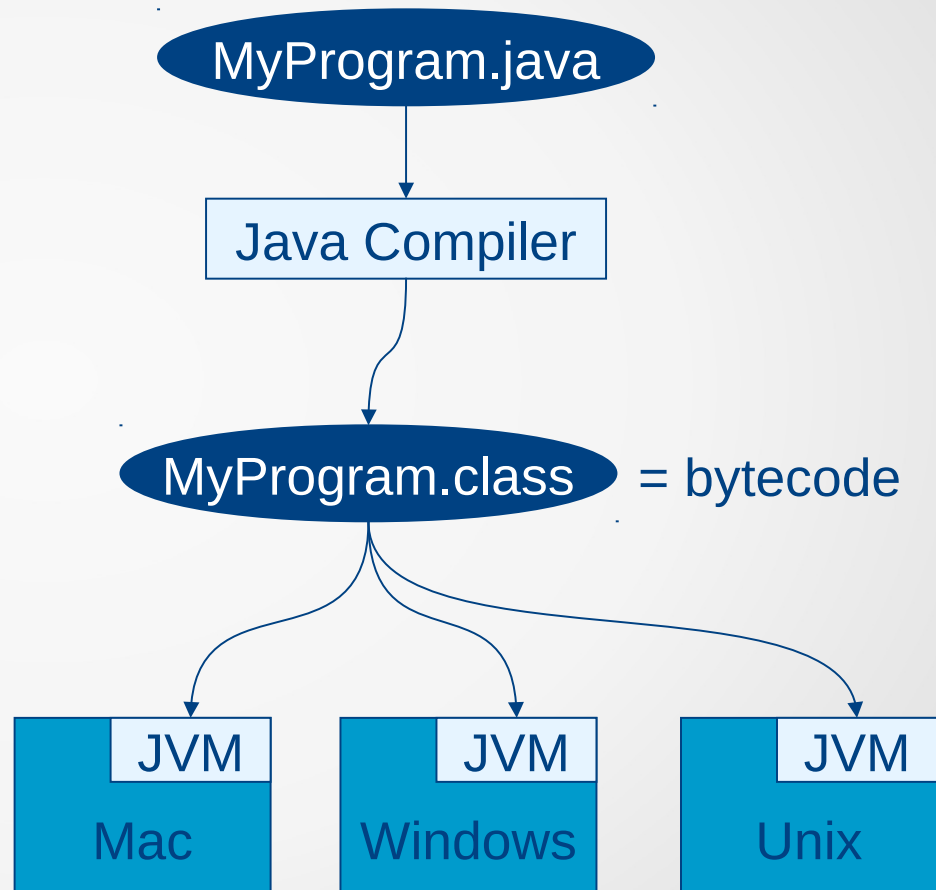
Robuste et sécurisé

- Conçu pour créer des logiciels hautement fiables
- Oblige le programmeur à garder à l'esprit les erreurs hardware et software
- Vérifications complètes à l'exécution et à la compilation
- Existence d'un « garbage collector » qui permet d'éviter les erreurs de gestion de la mémoire

Java comme langage de programmation

Neutre architecturalement

- Il existe une grande diversité de systèmes d'exploitation
- Le compilateur Java génère un bytecode, c'est à dire un format intermédiaire, neutre architecturalement, conçu pour faire transiter efficacement le code vers des hardware différents et/ou plateformes différentes
- Le bytecode ne peut-être interprété que par le processeur de la JVM
- On dit que Java est WORA



Java comme langage de programmation

Ouvert et distribué

- Support intégré d'Internet
 - La Class URL
 - Communication réseaux TCP et UDP
 - RMI, CORBA, Servlets
- Connectivité aux bases de données
 - JDBC: Java DataBase Connectivity
 - Offre des facilités de connexions à la plupart des BD du marché
 - Offre un pont vers ODBC
- Support des caractères internationaux
 - Java utilise le jeu de caractères UNICODE
 - JVM équipée de tables de conversion pour la plupart des caractères
 - JVM adapte automatiquement les paramètres régionaux en fonction de ceux de la machine sur laquelle elle tourne

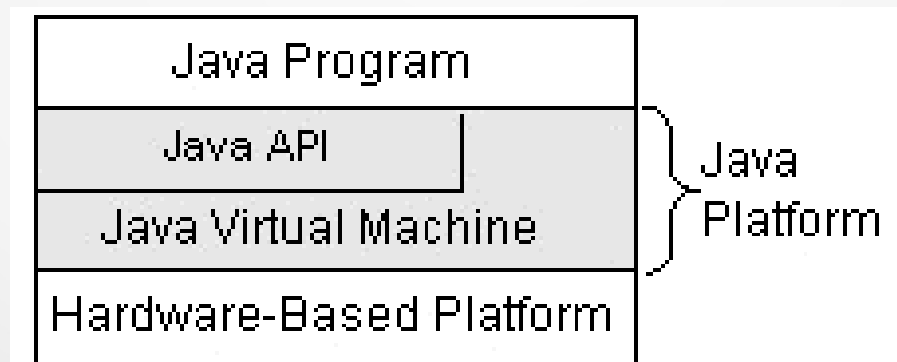
Java comme langage de programmation

Performant

- Considération basique
- Exécution ralentie à cause de l'interpréteur ?
- Le code natif généré par l'interpréteur est-il aussi rapide que celui réalisé par un compilateur classique (par ex C)?
- Plusieurs processus peuvent être exécutés en même temps
 - Comparable au multitâche d'un OS
 - Le temps du CPU est divisé (sliced)
 - Toutes les bibliothèques Java
- Edition de lien effectuée à l'exécution du programme
- Codes exécutables chargés depuis un serveur distant permet la mise à jour transparente des applications

Java comme Plateforme

- Plateforme = environnement hardware ou software sur lequel le programme est exécuté.
- La Java « Platform » se compose de:
 - la Java Virtual Machine (Java VM)
 - la Java Application Programming Interface (Java API)



Java comme Plateforme

Java Application Programming Interface (API)

- L'API Java est structuré en libraires (packages). Les packages comprennent des ensembles fonctionnels de composants (classes)..
- Le noyau (core) de l'API Java (incluse dans toute implémentation complète de la plateforme Java) comprend notamment :
 - Essentials (data types, objects, string, array, vector, I/O,date,...)
 - Applet
 - Abstract Windowing Toolkit (AWT)
 - Basic Networking (URL, Socket –TCP or UDP-,IP)
 - Evolved Networking (Remote Method Invocation)
 - Internationalization
 - Security
 -

Java Virtual Machine (1/2)

- « An imaginery machine that is implemented by emulating it in software on a real machine. Code for the JVM is stored in .class files, each of which contains code for at most one public class »
- Définit les spécifications hardware de la plateforme
- Lit le bytecode compilé (indépendant de la plateforme)
- Implémentée en software ou hardware
- Implémentée dans des environnements de développement ou dans les navigateurs Web

Java comme Plateforme

Java Virtual Machine (2/2)

La JVM définit :

- Les instructions de la CPU
- Les différents registres
- Le format des fichiers .class
- Le tas (« Heap ») des objets « garbage-collectés »
- L'espace mémoire

Java comme Plateforme

Java Runtime Environment

Trois tâches principales :

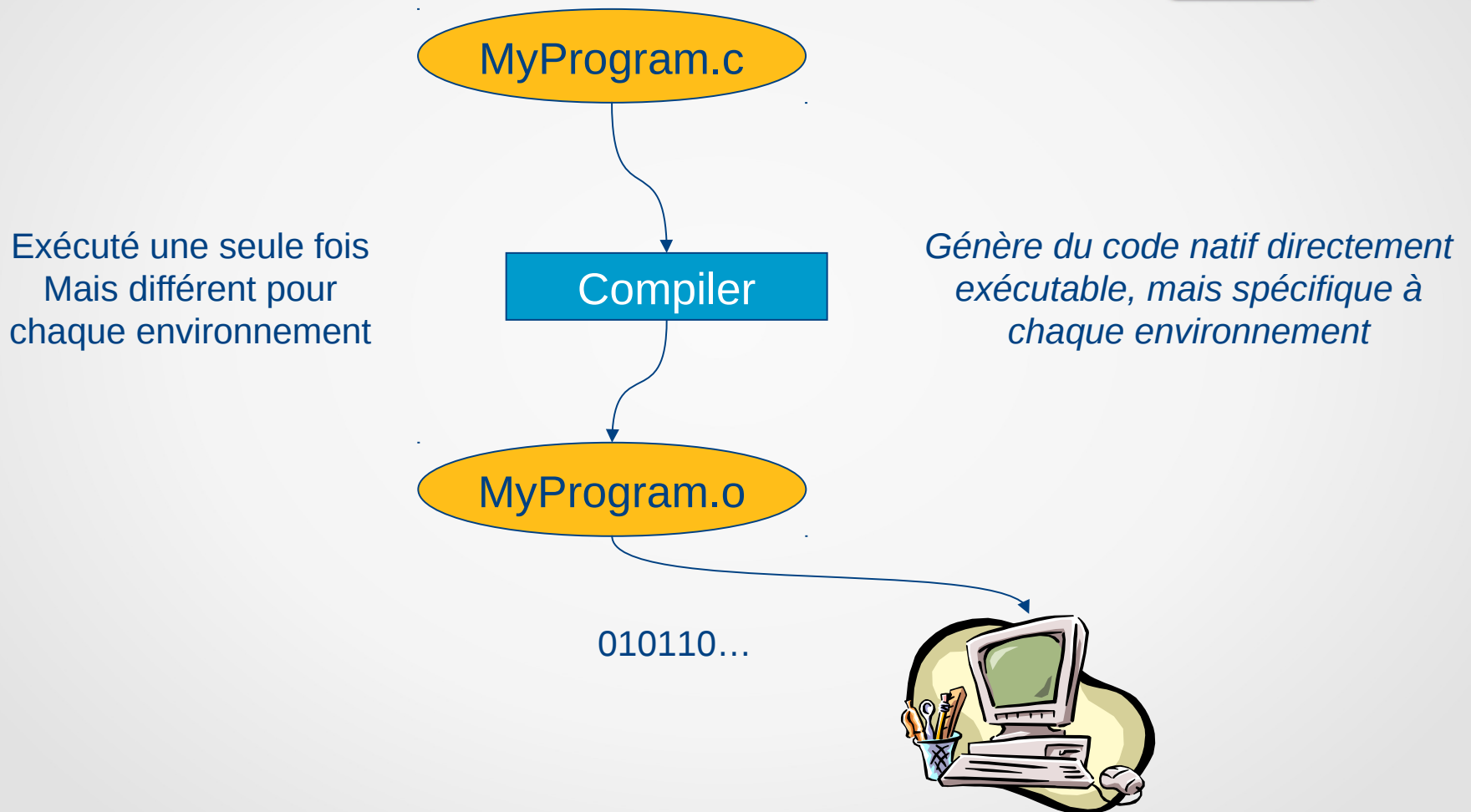
- Charger le code (class loader)
- Vérifier le code (bytecode verifier)
- Exécuter le code (runtime interpreter)

D'autres THREAD s'exécutent :

- Garbage collector

Déploiement d'un programme (1/2)

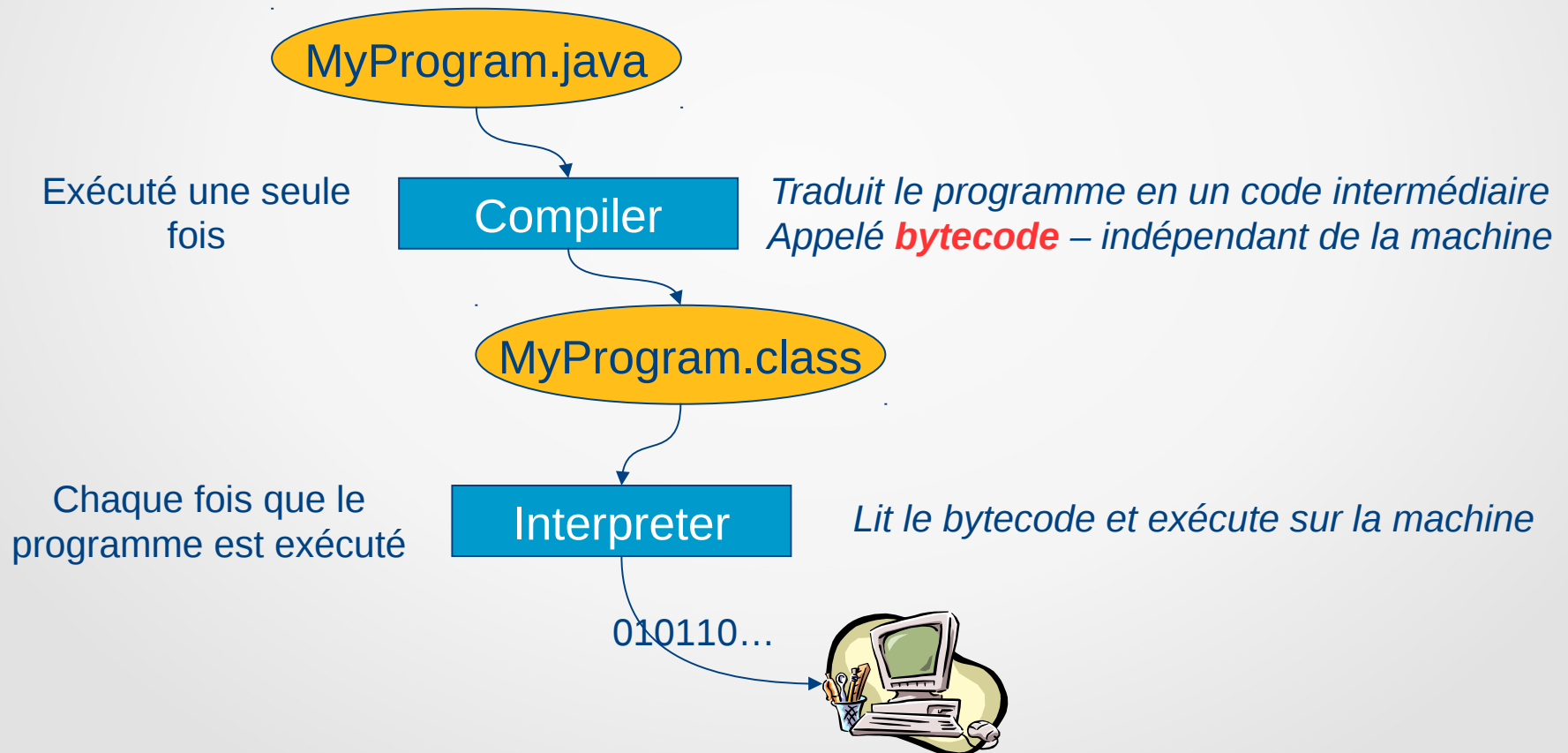
Paradigme classique de la compilation



Déploiement d'un programme (2/2)

Changement de la vision traditionnelle de la compilation

- Chaque programme est compilé et interprété
- « Write once run everywhere »



Bref Historique

- 1991: Développement de OAK
 - langage simple, portable et orienté objets
 - pour la programmation d'appareils électroniques ménagers
 - emprunte la portabilité du Pascal (VM) et la syntaxe de C++
- 1994: Abandon du projet OAK
 - Peu d'enthousiasme pour l'idée
- 1995 : Intégration de la JVM dans Netscape
 - Apparition des Applets
 - Explosion d'Internet → attrait grandissant pour Java
- 1999 : Apparition de JINI
 - Nouvelle technologie basée sur Java
 - Reprend l'ambition de départ d'un plug and play universel
 - Distribué sur tous les appareils munis d'un processeur

II. Première application en Java

Comment développer une application?

Deux façons d'écrire des programmes Java:

- En écrivant le code dans un simple éditeur de texte
 - Compilation et exécution du code en ligne de commande DOS
- En utilisant un environnement de développement (IDE)
 - Netbeans (<http://www.netbeans.com>)
 - Borland JBuilder (<http://www.borland.com/jbuilder>)
 - IBM WebSphere Studio (<http://www.ibm.com/software/awdtools>)
 - Sun ONE Studio (<http://wwws.sun.com/software/sundev>)
 - Microsoft .Net Studio (<http://msdn.microsoft.com/vstudio>)
 - ...

Une première application

Application versus Applet

- Une application Java
 - est composée d'une classe possédant une méthode main() :

```
public static void main (String[] args){  
    //code à exécuter pour initialiser l'application  
}
```

- L'environnement d'exécution dépend de l'OS de la machine
- Pas de restriction au niveau des API

Une première application

Application HelloWorld

- Créer un fichier texte : HelloWorld.java
- Règle de bonne pratique : 1 classe par fichier et 1 fichier par classe

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

La première ligne du programme doit être la déclaration de la classe

Tout programme doit contenir une méthode **main** qui porte la signature ci-contre

Écrire à l'écran "Hello the World"

Fermer les accolades

- Compiler le programme : `javac HelloWorld.java`
- Le compilateur génère le bytecode dans le fichier : `HelloWorld.class`
- Exécuter l'application : `java HelloWorld`
- « Hello the World » s'affiche à l'écran

III. Syntaxe du langage Java

Survol du chapitre

- Conventions d'écriture
- Commentaires dans le code source
- Identificateurs
- Mots-clé
- Types primitifs et types de références
- Les tableaux (« Array »)
- La classe String
- Arithmétique et opérateurs
- Instructions de contrôle
 - If, then, else
 - For
 - While
 - Do... While
 - Break et Continue
- Packages

Conventions d'écriture

Classes

class BankAccount

class RacingBike

Interfaces

interface Account

Méthodes

deposit()

getName()

Packages

package coursTechnofutur3.bank ;

Variables

int accountNumber

Constantes

MAXIMUM_SIZE

Commentaires dans le code source

Trois façons d'inclure des commentaires :

- Tout texte entre « // » et la fin de la ligne
`// Commentaires sur une seule ligne`
- Tout texte entre « /* » et « */ »
`/* Commentaires`
sur un nombre important voire très important
de lignes `*/`
- Les textes entre « /** » et « */ » sont utilisés pour créer des commentaires que l'exécutable JAVADOC pourra traiter afin de produire une documentation (cf. documentation de l'API Java)
`/** Commentaires destinés`
à la documentation `*/`

Identificateurs

- Un identificateur (*identifier*) permet de désigner une classe, une méthode, une variable ...
- On ne peut utiliser ce que l'on veut :
 - Interdiction d'utiliser les mots-clés
 - Commencent par :
 - Une lettre
 - Un « \$ »
 - Un « _ » (underscore)
 - Ne commencent pas par :
 - Un chiffre
 - Un signe de ponctuation autre que « \$ » ou « _ »

Mots-clé

abstract	double	int	strictfp **
boolean	else	interface	super
break	extends	long	switch
byte	final	native	synchronized
case	finally	new	this
catch	float	package	throw
char	for	private	throws
class	goto *	protected	transient
const *	if	public	try
continue	implements	return	void
default	import	short	volatile
do	instanceof	static	while

* Indique un mot clé qui est peu utilisé

** A partir de la plate-forme Java2

Types primitifs et types de référence

- Java est un langage fortement typé
- Le type de données précise
 - les valeurs que la variable peut contenir
 - les opérations que l'on peut réaliser dessus
- Deux types de données:
 - **Donnée primitive**: contient physiquement la valeur (caractère, nombre, booléen)
 - **Référence**: contient l'adresse mémoire où l'information relative à l'objet, l'interface, etc. est réellement stockée

Référence:

Adresse



Types primitifs et types de référence

Types de données primitifs (1/4)

Primitive Data Types

Integral

- byte 8 bits -128 to 127
- short 16 bits -32768 to 32767
- int 32 bits -2^{31} to $2^{31}-1$
- long 64 bits -2^{63} to $2^{63}-1$

Floating

- float 32 bits $1.4 \text{E}-45$ to $3.4 \text{E}38$
- double 64 bits $4.9 \text{E}-324$ to $1.8 \text{E}308$

Textual

- char 16 bits 0 to 65535

Logical

- one bit: true or false

Types primitifs et types de référence

Types de données primitifs (2/4)

- Explication:
 - byte : codé sur 8 bits $\rightarrow 2^8$ valeurs $\rightarrow (-2^7)$ to $(2^7-1) = -128$ à 127
 - int : codé sur 32 bits $\rightarrow 2^{32}$ valeurs $\rightarrow (-2^{31})$ to $(2^{31}-1)$
- Déclaration et initialisation :
 - int `int x=12;`
 - short `short x= 32; (short x=33000; // Hors limite)`
 - long `long x= 200L; // Nombre accolé à un L`
 - byte `byte x=012; // Nombre commençant avec un 0`
 - double `double x=23.2323;`
 - float `float x= 23.233F; // Nombre accolé à un F`
 - char `char c='a'; char c='\u0061'; char c=(char)97;`
 - boolean `boolean b=true;`

Types primitifs et types de référence

Types de données primitifs (3/4)

Déclaration, Initialisation et Assignment des types primitifs

<code>int t;</code>	Déclaration d'un entier t (t est l'identificateur)
<code>int u = 3;</code>	Déclaration et initialisation d'un entier
<code>t=7;</code>	Initialisation de t à la valeur 7
<code>u=t;</code>	Assignment (affectation) de la valeur de t à u
<code>m=9;</code>	Erreur déclaration préalable nécessaire
<code>char c;</code>	Déclaration
<code>c= 'a' ;</code>	Initialisation

Types primitifs et types de référence

Types de données primitifs (4/4)

Exemple:

```
int a = 5;
```

```
int b = 8;
```

```
a=b;
```



Déclaration et initialisation de 2 entiers: a et b

Affectation de la valeur de b à a

Désormais, il existe deux variables en mémoire qui ont la même valeur

Types primitifs et types de référence

Types de référence

- Tous les types hormis les types primitifs
- « Pointeur implicite » sur un objet

Référence: Adresse



Les tableaux ("Array") (1/3)

- Un tableau est utilisé pour stocker une collection de variables de même type
- On peut créer des tableaux de types primitifs ou de types de références
(cf. argument de la fonction main : `String[] args`)
- Un tableau doit être
 - Déclaré
 - Créé
 - Ses variables initialisées

```
int[] nombres;           // déclaration
nombres = new int[10];    // création
int[] nombres = new int[10]; // déclaration et création
nombres[0] = 28;
```

Les tableaux ("Array") (2/3)

- On peut construire des tableaux à plusieurs dimensions
- Des tableaux à plusieurs dimensions sont en fait des tableaux de tableaux

<pre>int[][] matrice = new int[3][];</pre>	« matrice » est une référence vers un tableau contenant lui-même 3 tableaux de taille non définie
<pre>matrice[0] = new int[4]; matrice[1] = new int[5]; matrice[2] = new int[3];</pre>	Le premier élément de la matrice est une référence vers un tableau de 4 entiers,...
<pre>matrice[0][0] = 25;</pre>	Le premier élément du premier tableau de la matrice est un entier de valeur 25

Exemple:

- Créer et initialiser une matrice contenant deux tableaux de 2 et 3 double respectivement
- Créer 5 variables de type «double»
- Affecter ces 5 variables aux 5 éléments de la matrice

Tableau (Résumé)

Array

Attention:

a.length

- Pour stocker une série de données de même nature
- Déclaration
`int [] nombre;` // une série de valeurs *int* dans le tableau nommé *nombre*
`String [][] etiquette;` // un tableau à deux dimensions de valeurs *String*
- Création
`nombre = new int[10];` // crée les cases `nombre[0]` à `nombre[9]`
`etiquette = new String[3][5];` // crée `etiquette[0][0]` à `etiquette[2][4]`
`int[] primes = {1, 2, 3, 5, 7, 7+4};` // déclare, crée de la bonne taille et initialise
- Utilisation
`nombre[0] = 4;`
`for (int i=1; i<nombre.length; i++) nombre[i]=nombre[i]+1;`
`etiquette[2][3] = "un texte";`
`String texte = etiquette[2][3];`

La classe String

- String n'est pas un type primitif, c'est une classe (un type de référence)
- Déclaration de deux String:
`String s1, s2;`
- Initialisation :
`s1 = "Hello";`
`s2 = "le monde";`
- Déclaration et initialisation :
`String s3 = "Hello";`
- Concaténation :
`String s4 = s1 + " " + s2;`

Arithmétique et opérateurs

Arithmétique élémentaire

- Quelle est la valeur de : $5+3*4+(12/4+1)$
- Règles de précédences sur les opérateurs:

Niveau	Symbole	Signification
1	()	Parenthèse
2	*	Produit
	/	Division
	%	Modulo
3	+	Addition ou concaténation
	-	Soustraction

Arithmétique et opérateurs

Arithmétique élémentaire

- Quelle est la valeur de : $5+3*4+(12/4+1)$
- Règles de précédences sur les opérateurs:

Niveau	Symbole	Signification
1	()	Parenthèse
2	*	Produit
	/	Division
	%	Modulo
3	+	Addition ou concaténation
	-	Soustraction

Arithmétique et opérateurs

Opérateurs de comparaison

- Pour comparer deux valeurs:

Opérateur	Exemple	Renvoie TRUE si
>	v1 > v2	v1 plus grand que v2
>=	v1 >= v2	Plus grand ou égal
<	v1 < v2	Plus petit que
<=	v1 <= v2	Plus petit ou égal à
==	v1 == v2	égal
!=	v1 != v2	différent

- Opérateurs logiques:

Opérateur	Usage	Renvoie TRUE si
&&	expr1 && expr2	expr1 et expr2 sont vraies
&	expr1 & expr2	Idem mais évalue toujours les 2 expressions
	expr1 expr2	Expr1 ou expr2, ou les deux sont vraies
	expr1 expr2	idem mais évalue toujours les 2 expressions
!	! expr1	expr1 est fausse
!=	expr1 != expr2	si expr1 est différent de expr2

Tests

- Pour les valeurs primitives (int, double, ...)
 - $x == y$: x et y ont la même valeur?
 - $x > y$, $x \geq y$, $x \neq y$, ...
 - Attention: ($== \neq =$)
- Pour les références à un objet
 - $x == y$: x et y pointent vers le même objet?
 - $x.compareTo(y)$: retourne -1, 0 ou 1 selon l'ordre entre le contenu des objets référés par x et y

Arithmétique et opérateurs

Opérateurs d'assignation (d'affectation)

- L'opérateur de base est '='
- Il existe des opérateurs d'assignation qui réalisent à la fois
 - une opération arithmétique, logique, ou bit à bit
 - et l'assignation proprement dite

Opérateur	Exemple	Équivalent à
+=	expr1 += expr2	expr1 = expr1 + expr2
-=	expr1 -= expr2	expr1 = expr1 – expr2
*=	expr1 *= expr2	expr1 = expr1 * expr2
/=	expr1 /= expr2	expr1 = expr1 / expr2
%=	expr1 %= expr2	expr1 = expr1 % expr2

Instructions et structures de contrôle

Déclarations, instructions, blocs

- Une instruction
 - Réalise un traitement particulier:
 - Renvoie éventuellement le résultat du calcul
 - Est comparable à une phrase du langage naturel
 - Constitue l'unité d'exécution
 - Est toujours suivie de « ; »
 - Instruction d'affectation (d'assignation), instruction de déclaration ...
- Un bloc
 - Est une suite d'instructions entre accolades « { » et « } »
 - Délimite la portée des variables qui y sont déclarées
- Une déclaration
 - Constitue la signature d'un élément (classe, variable ou méthode)
 - Annonce la définition de cet élément
 - Est (normalement) toujours suivie d'un bloc d'instructions

Instructions et structures de contrôle

Structures de contrôle

- Les structures de contrôles permettent d'arrêter l'exécution linéaire des instructions (de bas en haut et de gauche à droite)
- Elles permettent d'exécuter conditionnellement une instruction, ou de réaliser une boucle

Type d'instruction	Mots clés utilisés
Décision	if() else – switch() case
Boucle	for(; ;) – while () – do while()
Traitement d'exceptions	try catch finally – throw
Branchement	label : -- break – continue -- return

Instructions et structures de contrôle

Structures de contrôle

IF – THEN – ELSE

```
if (expression)
{
    //instructions
}
```

```
if (expression)
{
    //instructions
}
else
{
    //instructions dans les autres cas
}
```

Instructions et structures de contrôle

Structures de contrôle

SWITCH – CASE

```
switch (number)
{
    case 1 : instructions; break;
    case 2 : instructions; break;
    default : instructions;
}
```


Instructions et structures de contrôle

Structures de contrôle

FOR

```
for (initialisation; condition; mise à jour de valeurs){  
    // instructions  
}
```

- Initialisation: à exécuter lorsque le programme rentre pour la première fois dans la boucle
- Condition : à remplir pour recommencer le bloc d'instructions
- Mise à jour: instruction exécutée chaque fois que la boucle est terminée

Exemples:

```
for (int i=0 ; i<10 ; i++) {  
    System.out.println("The value of i is : " + i);  
}
```

Instructions et structures de contrôle

Structures de contrôle

WHILE – DO WHILE

```
while (test logique) {  
    //instructions  
}
```

Si le code de la boucle doit être exécuté la première fois de toute façon:

```
do {  
    // code to perform  
} while (logical_test)
```

Boucle infinie:

```
while (true) {  
    //instructions  
}
```

Instructions et structures de contrôle

Structures de contrôle

- BREAK / CONTINUE
- BREAK: achève immédiatement la boucle
- CONTINUE: ignore le reste des instructions et recommence au début de la boucle

```
for (int i=0; i<10 ;i++){  
    if (i==5) continue; // Si i=5, on recommence au début  
    if (i==7) break;     /* Si i=7, on sort de la boucle et  
    les instructions suivantes sont  
    exécutées */  
    System.out.println("The value of i is : " + i);  
}
```

Boucle Attention:
un ; après le for(), itère sur
la condition, et 'somme' ne sera
incrémentée qu'une seule fois

Attention:
'i' n'est déclarée ici qu'à
l'intérieur de la boucle for

Schéma d'exécution

- Pour traiter beaucoup de données en série

- Schémas

- Boucle *for*

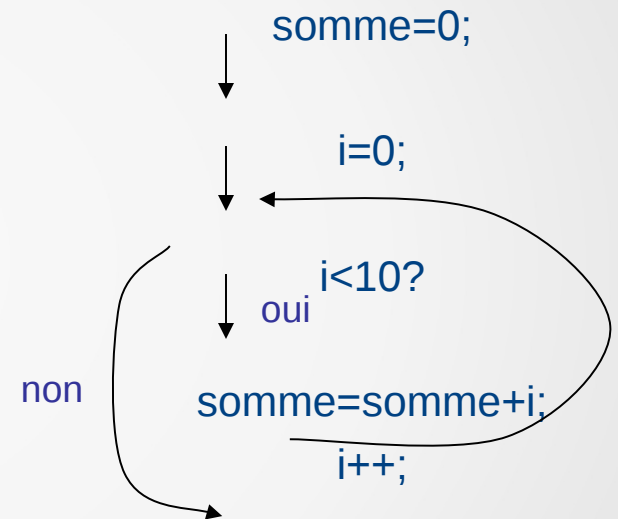
```
int somme = 0;  
for (int i = 0; i < 10; i++) somme = somme + i;
```

- Boucle *while*

```
int somme = 0;  
int i = 0;  
while (i < 10) {  
    somme = somme + i;  
    i++;  
}
```

- Que font ces deux boucles?

i:	0,	1,	2,	3,	4,	5,	6,	7,	8,	9,	10	
	↓	↓								↓		
somme:	0	→ 0,	→ 1,	3,	6,	10,	15,	21,	28,	36,	→ 45,	sortie



Boucle

- **do A while (*condition*)**
 - Faire A au moins une fois
 - Tester la condition pour savoir s'il faut refaire A

```
int somme = 0;  
int i = 15;  
while (i<10) { somme = somme + i;  
                i++;  
            }
```

somme = 0

```
int somme = 0;  
int i = 15;  
do { somme = somme + i;  
      i++;  
    }  
while (i<10)
```

somme = 15

Schéma d'exécution somme=0;

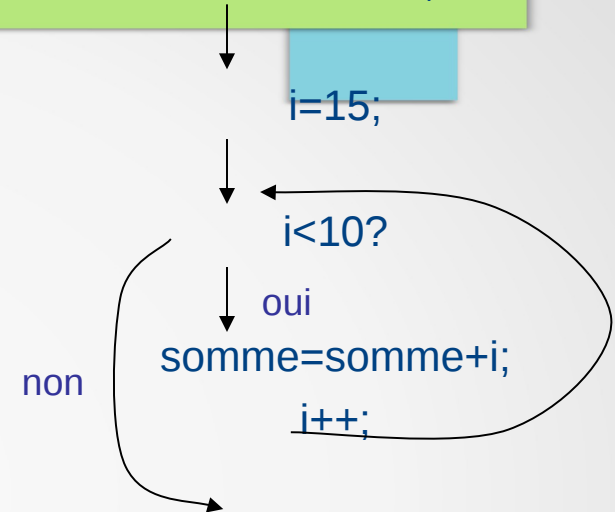
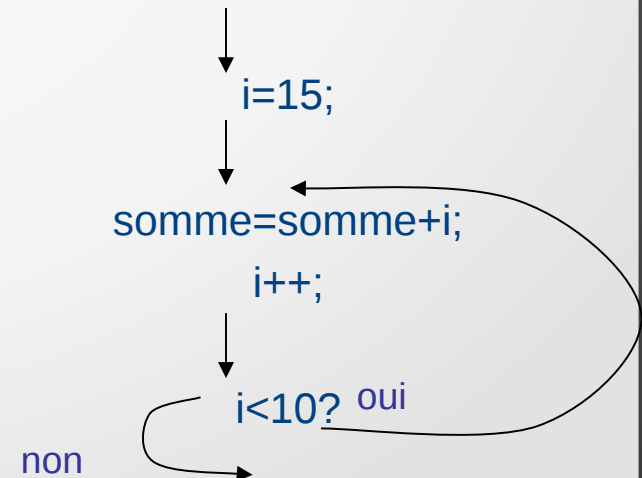


Schéma d'exécution somme=0;



Exemple

- Calcul des intérêts
- Étant donné le solde initial, le solde souhaité et le taux d'intérêt, combien d'années seront nécessaires pour atteindre le solde souhaité
 - au lieu d'utiliser une formule, on simule le calcul
- Algorithme (pseudocode):
 1. ans = 0;
 2. WHILE solde n'atteint pas le solde souhaité
 3. incrémenter ans
 4. ajouter l'intérêt au solde

Programme

```
public void nombreAnnees (double soldeInitial, double soldeCible,
    double taux ) {
    int ans = 0;
    while (soldeInitial < soldeCible) {
        ans++;
        double interet = soldeInitial * rate;
        soldeInitial += interet;
    }
    System.out.println(ans + " années seront requises");
}
```

Diagram illustrating the loop logic:

- From `ans++;` to `years = years + 1;`
- From `soldeInitial += interet;` to `balance = balance + interest;`

Appel de la méthode:

```
nombreAnnees(1000, 1500, 0.05);
```

Résultat:

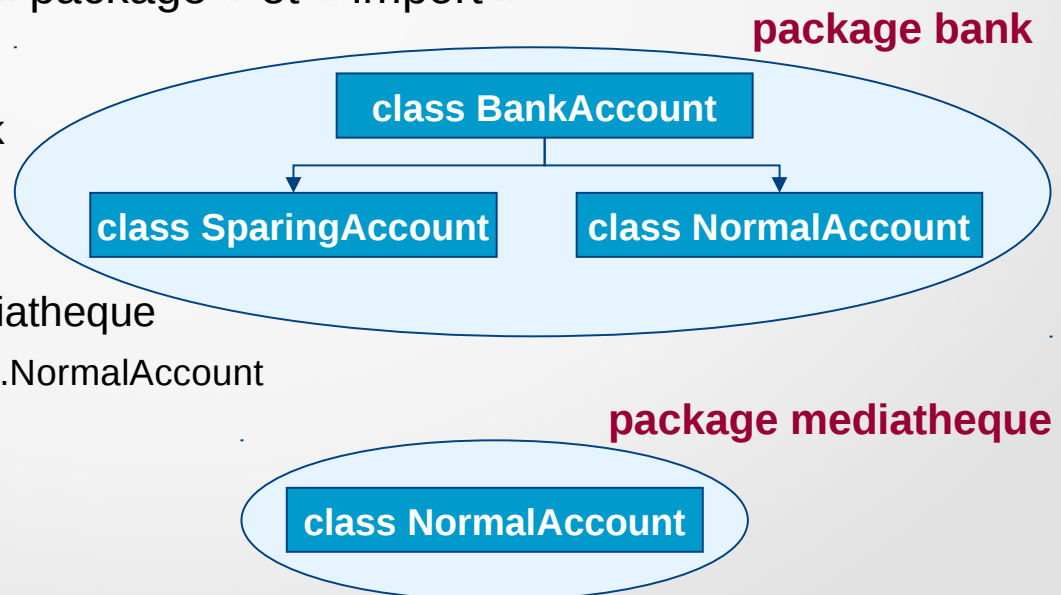
```
56 années seront requises
```

Les packages et les importations

- Les packages offrent une organisation structurée des classes
- La répartition des packages correspond à l'organisation physique
- Les packages conditionnent les importations de classes
- La variable CLASSPATH indique le chemin d'accès aux packages
- Les packages permettent la coexistence de classes de même nom
- Les mots-clé associés sont « package » et « import »

- Exemple:

- package technofutur3.bank
 - class NormalAccount
 - class SparingAccount
- package technofutur3.mediatheque
 - import technofutur3.bank.NormalAccount



Exercices

- Ding Ding Bottle

- Ecrire un programme qui compte de 1 à 100
- Affiche ces nombres à l'écran
- Remplace les multiples de 5 par « Bottle »
- Remplace les multiples de 7 par « Ding Ding »

EX 3.1

- Calcul de factorielle

- Ecrire un programme qui calcule la factorielle d'un nombre

EX 3.2

- Dessiner une Pyramide

*

**

- Décoder une chaîne de caractère

Compiler et exécuter les programmes créés

Factorielle

```
public class Factorielle
{
    public static double factorielle(int x) {
        if (x < 0) return 0.0;
        double fact = 1.0;
        while (x > 1) {
            fact = fact * x;
            x = x - 1;
        }
        return fact;
    }

    public static void main(String[] args) {
        int entree = Integer.parseInt(args[0]);
        double resultat = factorielle(entree);
        System.out.println(resultat);
    }
}
```

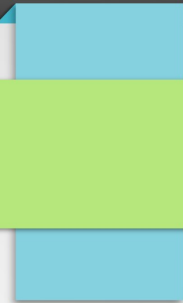
Si une méthode (ou un attribut, une variable) de la classe est utilisée par la méthode main (*static*), il faut qu'il soit aussi *static*.

Exercices

EX 3.3

- Etudiants
 - Vous recevez une à une les notes de vos N étudiants (/10). N est donné par l'utilisateur au début.
 - Notez pour chaque étudiant s'il réussit ou échoue et, à la fin, indiquez le nombre de ceux qui échouent, le nombre de ceux qui réussissent, la moyenne totale et les deux moyennes de ceux qui échouent et réussissent
- Exécuter les programmes et observer le résultat

IV. Premiers concepts orientés objets



Les concepts de l'OO

Orienté objet?

Un programme orienté objet est uniquement constitué de classes interagissant par envoi de messages

L'intégralité du code d'un programme orienté objet se trouve donc à l'intérieur de classes

Les concepts de l'OO Orienté objet?

Déclaration
de la classe

Variables d'instance
ou « champs »

Définition du
constructeur

Méthodes d'accès

Définition
des
méthodes

```
package bank;
import java.lang.*;
public class BankAccount {

    private String name ;
    private int solde ;
    private int interest ;
    public BankAccount(String n,int s,int i) {
        name=n ;
        solde=s;
        interest=i;
    }
    public String getName() { return name; }
    public void setName (String n) {name= n;}
    ...
    public void deposit (int amount) {
        solde += amount ;
    }
    public void withdrawal (int amount ){
        solde-=amount ;
    }
}
```

Class Body

Exercices

EX 4.1

- Les GSM
 - Créer une classe GSM
 - Quelles sont les variables d'états accessibles et non accessibles?
 - Quelles sont les fonctionnalités offertes à l'utilisateur?

Les concepts de l'OO

Héritage

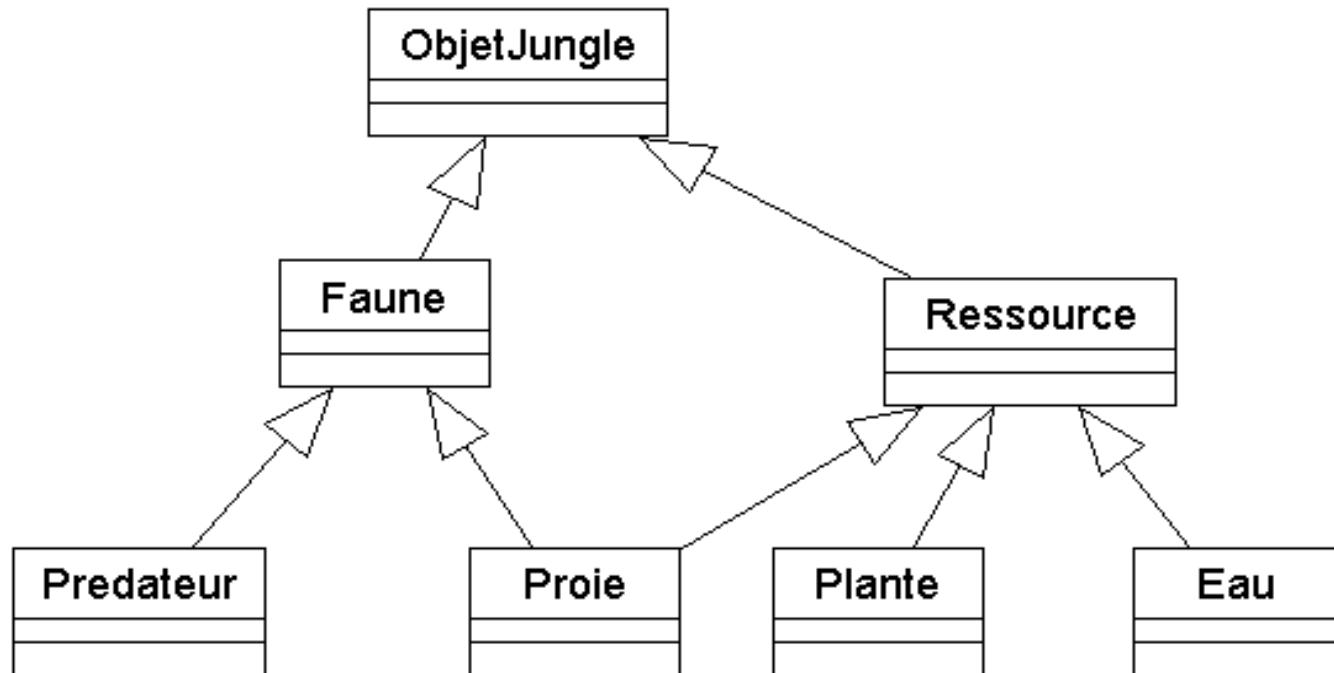
- En quoi consiste l'héritage?
 - Supposons qu'il existe déjà une classe qui définit un certain nombre de messages et qu'on ait besoin d'une classe identique mais pourvue de quelques messages supplémentaires
- ➔ Comment éviter de réécrire la classe de départ?
- ➔ Regrouper les classes en super-classes en factorisant et spécialisant
- ➔ La sous-classe hérite des attributs et méthodes et peut en rajouter de nouveaux
- ➔ Quid du multi-héritage?



Les concepts de l'OO

Héritage

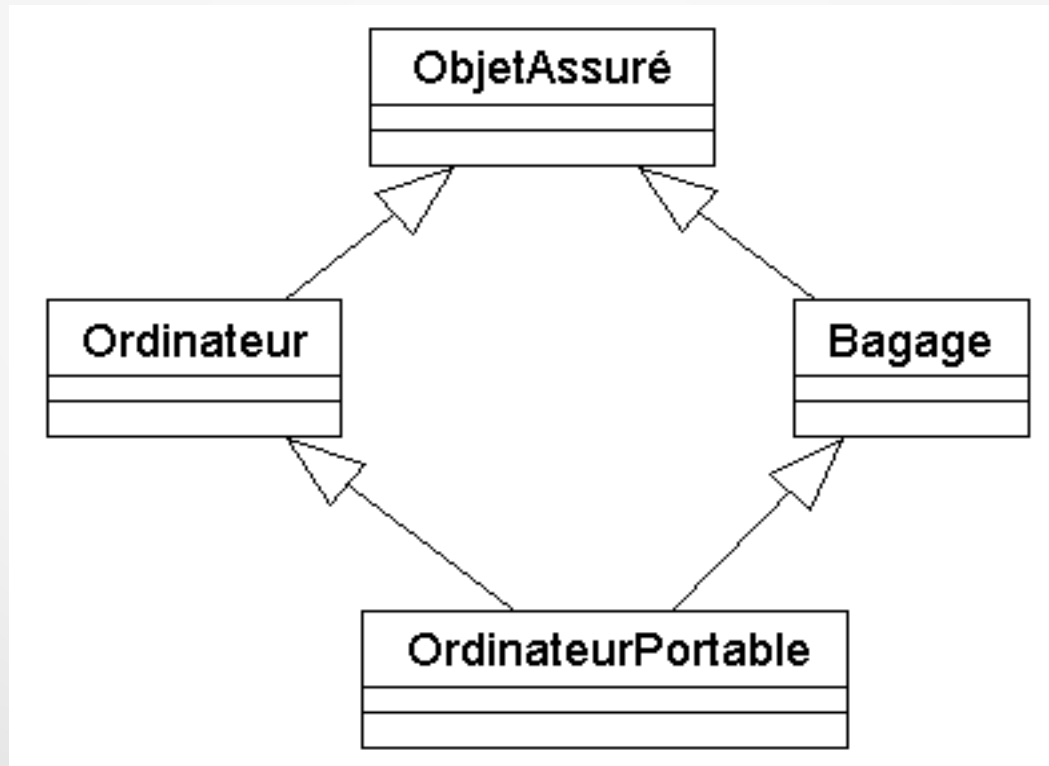
- Ex: Dans l'écosystème
 - La classe Faune regroupe les animaux
 - La classe Ressource regroupe l'eau et la plante



Les concepts de l'OO

Héritage

- Quid du multi-héritage?
 - Possible ou non d'un langage à un autre



Les concepts de l'OO

Héritage

- Qu'est-ce que le polymorphisme?
 - Concept basé sur la notion de redéfinition de méthodes
 - Permet à une tierce classe de traiter un ensemble de classes sans connaître leur nature ultime
 - Permet de factoriser des dénominations d'activité mais pas les activités elles-mêmes
 - Consiste à permettre à une classe de s'adresser à une autre en sollicitant un service générique qui s'appliquera différemment au niveau de chaque sous-classe du destinataire du message
 - En d'autres termes, permet de changer le comportement d'une classe de base sans la modifier ➔ Deux objets peuvent réagir différemment au même appel de méthode
 - Uniquement possible entre classes reliées par un lien d'héritage
- Exemple dans l'écosystème?
 - Demander à tous les animaux de se déplacer (selon leurs propres règles) en leur adressant un message en tant qu'objets de type "Faune"

Exercices

EX 4.2

- Les êtres vivants
 - Subdiviser les être vivants en une hiérarchie de classes
 - A chaque niveau dans la hiérarchie, indiquer:
 - Quelles sont les variables d'états accessibles et non accessibles aux autres ?
 - Quelles sont les comportements exploitables par les autres ?
 - Quelles sont les comportements internes, cachés aux autres ?
 - Quelle hiérarchie de classes pourrait-on proposer ?
 - Réaliser un diagramme de classes UML du résultat

Les concepts de l'OO

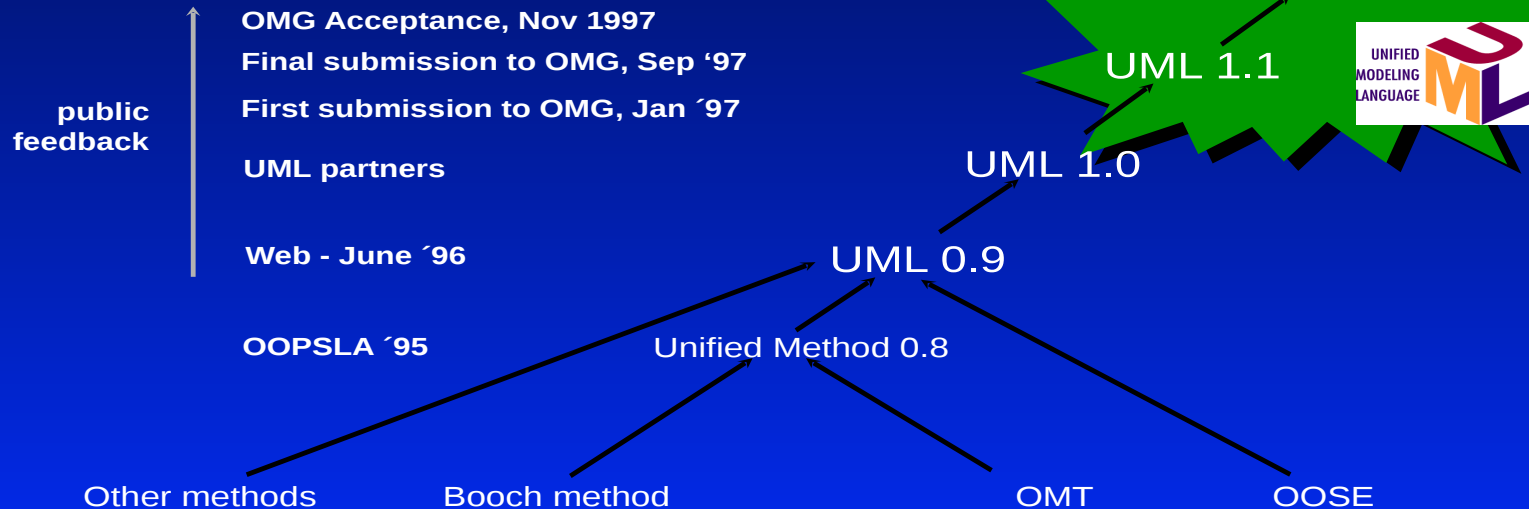
La modélisation devient la référence

- Pourquoi la modélisation?
 - La conception OO est entièrement basée sur une modélisation des objets intervenant dans le problème
 - Avant de programmer quoi que ce soit, il faut donc modéliser les classes et leurs relations au minimum
- Comment?
 - Sur base d'UML (Unified Modeling Language)
 - ➔ Notation standardisée pour tout le développement OO de la conception au déploiement
 - ➔ Définition de 9 diagrammes
 - 1. Identifier les classes
 - ➔ Attributs, comportements, polymorphisme
 - 2. Déterminer les relations entre les classes
 - ➔ Associations / Dépendance / Héritage
 - 3. Construire les modèles

Les concepts de l'OO

La modélisation devient la référence

Creating the UML



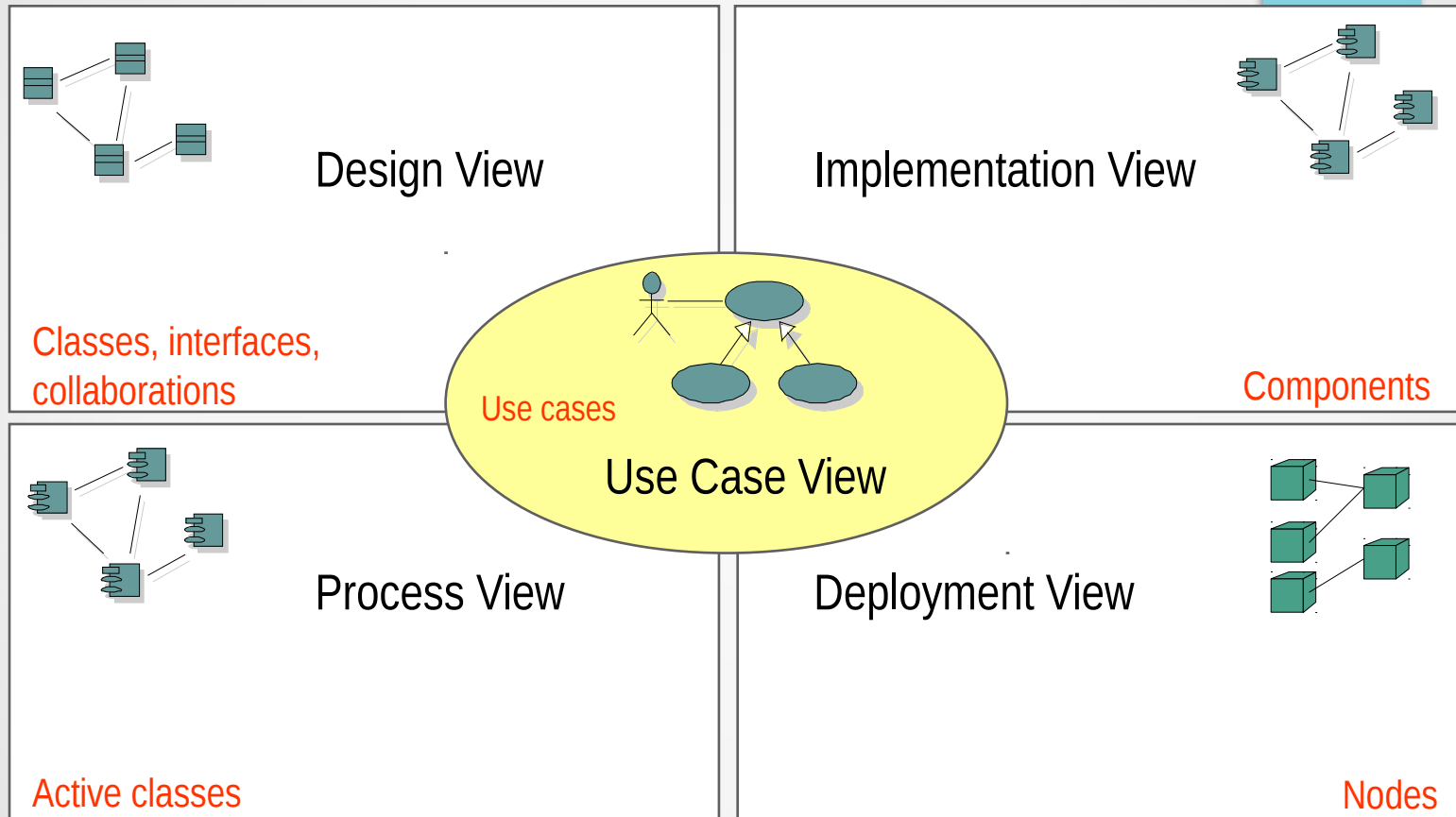
Les concepts de l'OO

La modélisation devient la référence

- Qu'est-ce qu'UML?
 - UML est un langage objet graphique - un formalisme orienté objet, basé sur des diagrammes (9)
 - UML permet de s'abstraire du code par une représentation des interactions statiques et du déroulement dynamique de l'application.
 - UML prend en compte, le cahier de charge, l'architecture statique, la dynamique et les aspects implémentation
 - Facilite l'interaction, les gros projets
 - Générateur de squelette de code
 - UML est un langage PAS une méthodologie, aucune démarche n'est proposée juste une notation

Les concepts de l'OO

La modélisation devient la référence



Les concepts de l'OO

La modélisation devient la référence

- Diagrammes UML

- Les diagrammes des cas d'utilisation: les fonctions du système, du point de vue de l'utilisateur ou d'un système extérieur - l'usage que l'on en fait
- Les diagrammes de classes: une description statique des relations entre les classes
- Les diagrammes d'objet: une description statique des objets et de leurs relations. Une version « instanciée » du précédent
- Les diagrammes de séquence: un déroulement temporel des objets et de leurs interactions
- Les diagrammes de collaboration: les objets et leurs interactions en termes d'envois de message + prise en compte de la séquentialité

Les concepts de l'OO

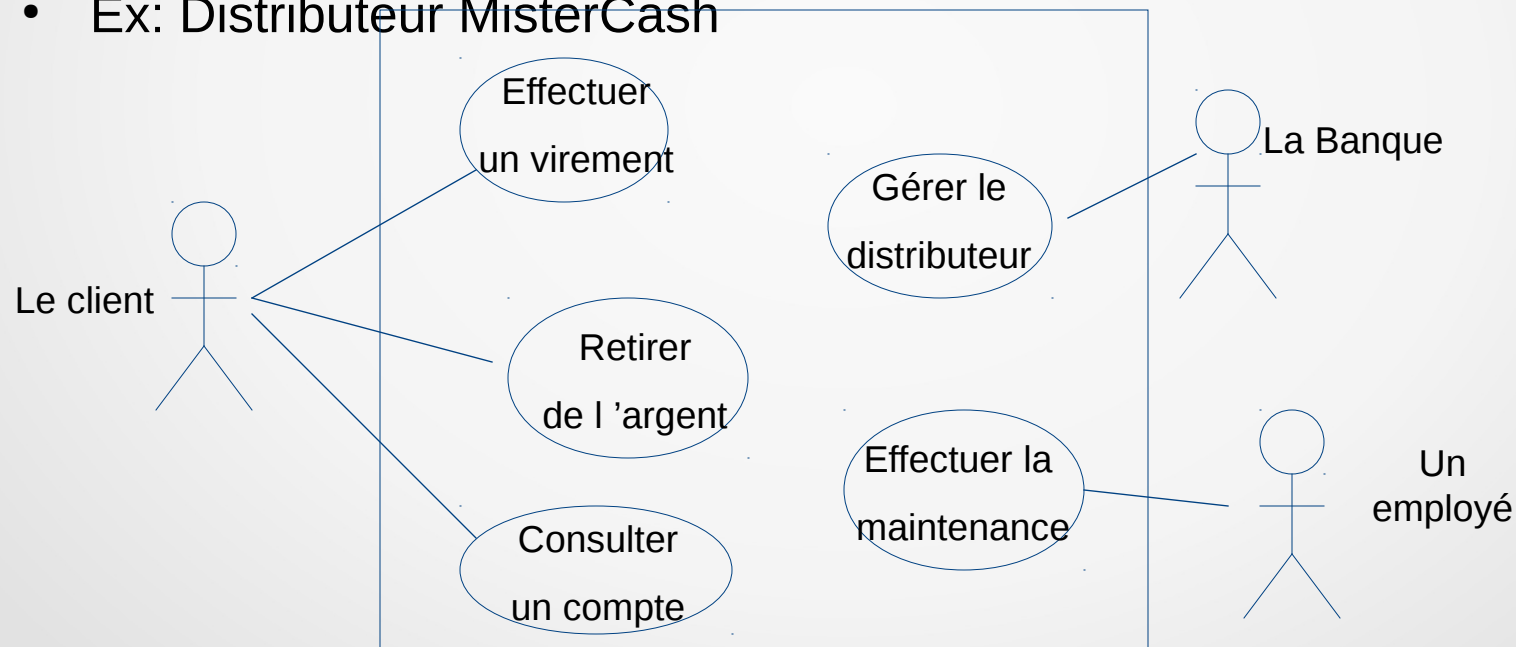
La modélisation devient la référence

- Diagrammes UML
 - Les diagrammes d'états-transitions: scrute les cycles de vie d'une classe d'objet, la succession d'états et les transitions
 - Les diagrammes d'activité: le comportement des différentes opérations en termes d'actions
 - Les diagrammes de composants: représente les composants physiques d'une application
 - Les diagrammes de déploiements: le déploiement des composants sur les dispositifs et les supports matériels

Les concepts de l'OO

La modélisation devient la référence

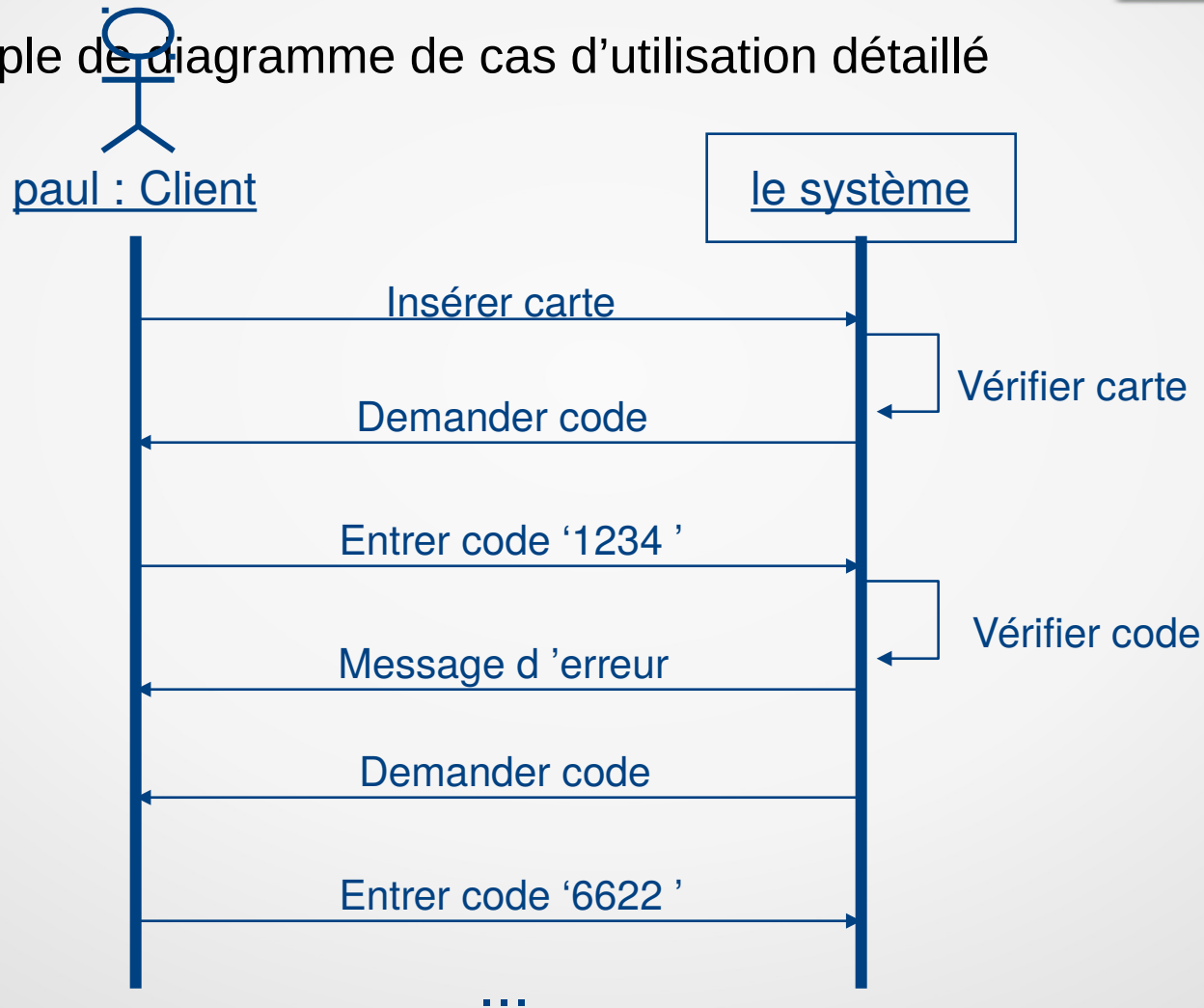
- Diagramme de cas d'utilisation
 - Cela répond aux spécifications du système:
 - Ses fonctionnalités, son utilisation, les attentes de l'utilisateur
- Ex: Distributeur MisterCash



Les concepts de l'OO

La modélisation devient la référence

- Exemple de diagramme de cas d'utilisation détaillé



Les concepts de l'OO

La modélisation devient la référence

- Diagramme de classes
 - Le but du diagramme de classes est de représenter les classes au sein d'un modèle
 - Dans une application OO, les classes possèdent:
 - Des attributs (variables membres)
 - Des méthodes (fonctions membres)
 - Des relations avec d'autres classes
 - C'est tout cela que le diagramme de classes représente
 - L'encapsulation est représentée par:
 - (private), + (public), # (protected)
 - Les attributs s'écrivent:
+/-/# nomVariable : Type
 - Les méthodes s'écrivent:
+/-/# nomMethode(Type des arguments) : Type du retour ou « void »

Nom Classe
Attributs
Méthodes()

Les concepts de l'OO

La modélisation devient la référence

- Les relations d'héritage sont représentées par:

—▶
– A B signifie que la classe A hérite de la classe B

- L'agrégation faible est représentée par:

◊—
– A B signifie que la classe A possède
un ou plusieurs attributs B

- L'agrégation forte (ou composition) est représentée par:

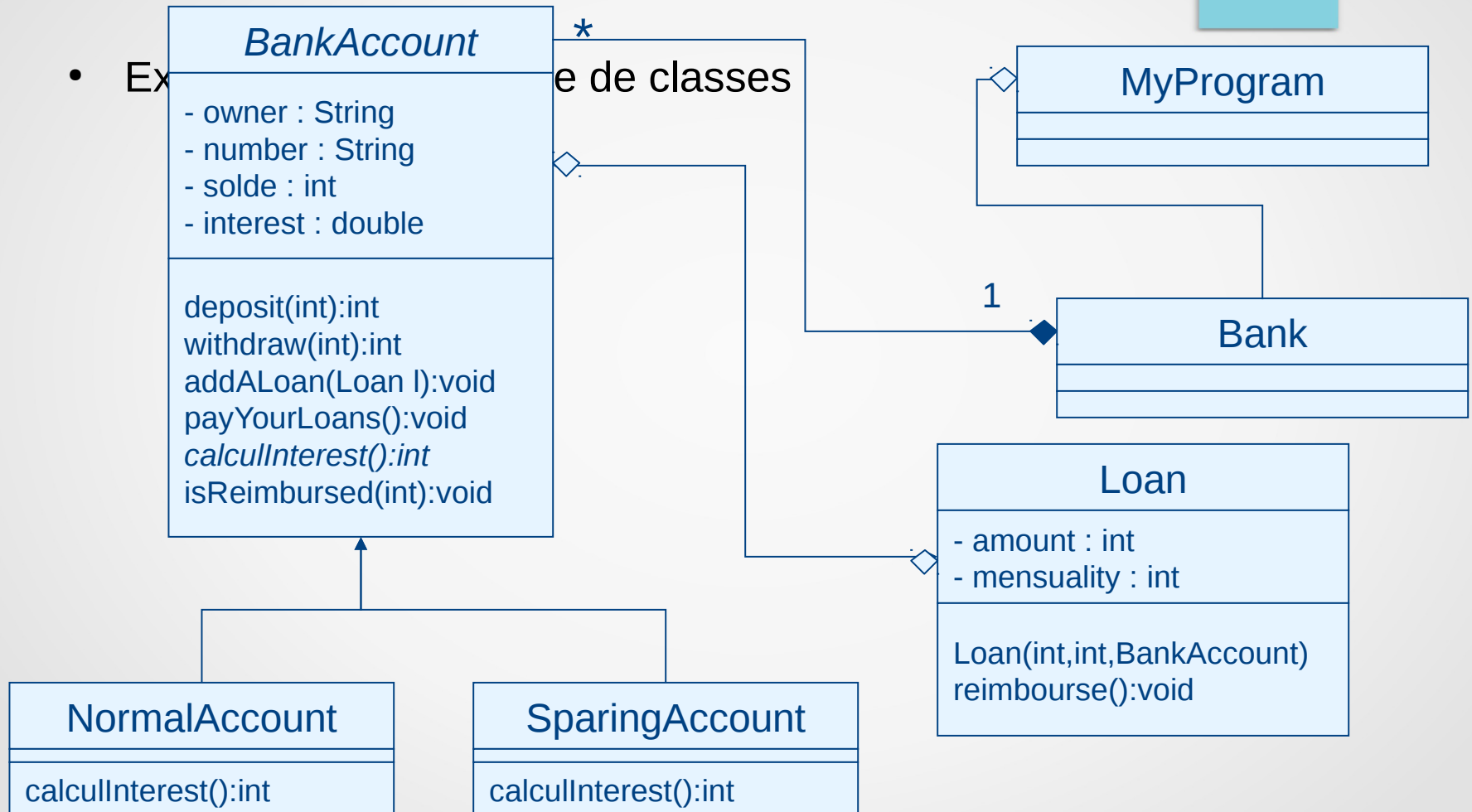
◆—
– A B signifie que les objets de la classe B ne peuvent exister
qu'au sein d'objets de type A

Les concepts de l'OO

La modélisation devient la référence

• Ex

e de classes

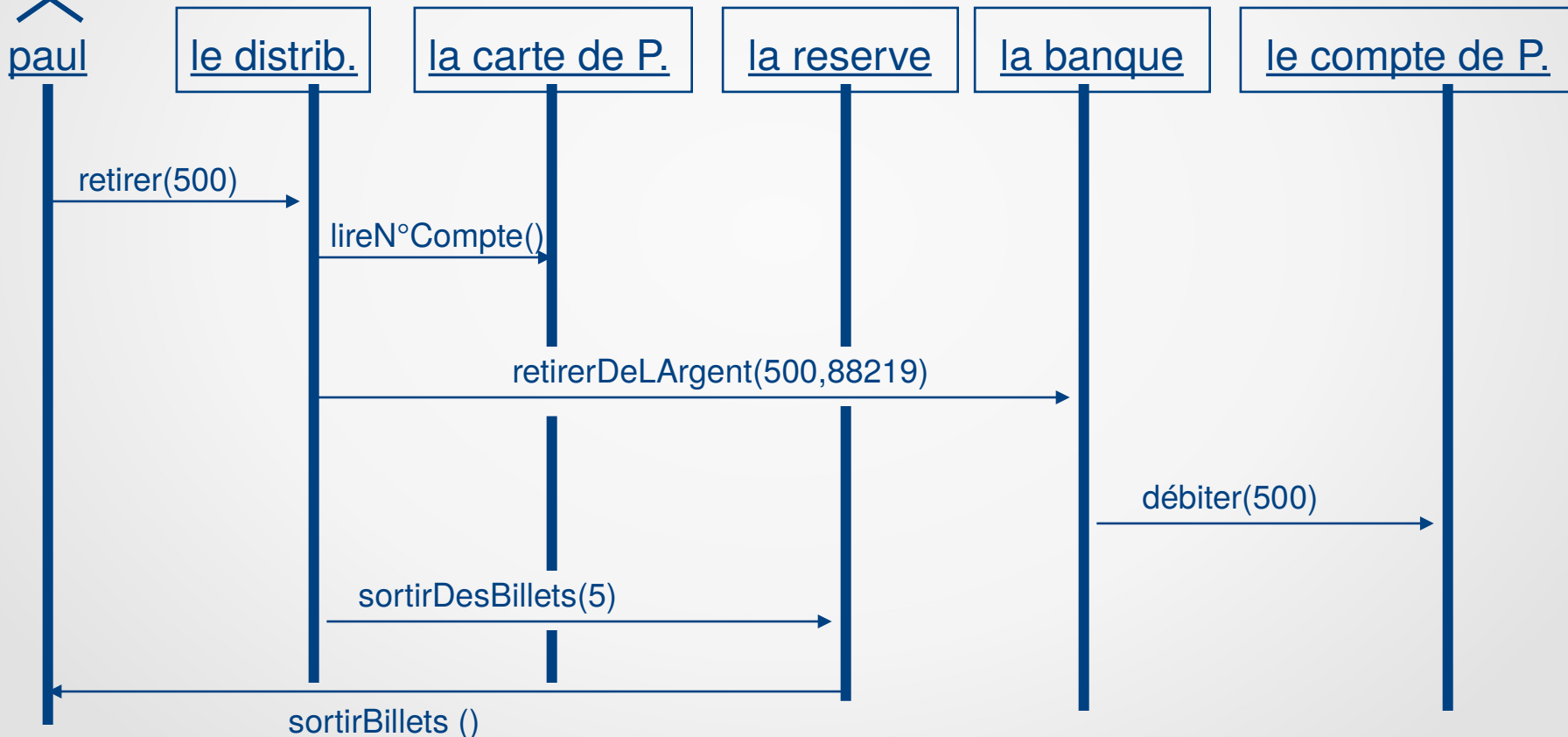


Les concepts de l'OO

La modélisation devient la référence



Exemple de diagramme de séquence



Les concepts de l'OO

Les avantages de l'OO

- Les programmes sont plus stables, plus robustes et plus faciles à maintenir car le couplage est faible entre les classes («encapsulation»)
- elle facilite grandement le ré-emploi des programmes: par petite adaptation, par agrégation ou par héritage
- émergence des «design patterns»
- il est plus facile de travailler de manière itérée et évolutive car les programmes sont facilement extensibles. On peut donc graduellement réduire le risque plutôt que de laisser la seule évaluation pour la fin.
- l'OO permet de faire une bonne analyse du problème suffisamment détachée de l'étape d'écriture du code - on peut travailler de manière très abstraite → UML
- l'OO colle beaucoup mieux à notre façon de percevoir et de découper le monde

Les concepts de l'OO

Les avantages de l'OO

- Tous ces avantages de l'OO se font de plus en plus évidents avec le grossissement des projets informatiques et la multiplication des acteurs. Des aspects tels l'encapsulation, l'héritage ou le polymorphisme prennent vraiment tout leur sens. On appréhende mieux les bénéfices de langage plus stable, plus facilement extensible et plus facilement ré-employable
- JAVA est un langage strictement OO qui a été propulsé sur la scène par sa complémentarité avec Internet mais cette même complémentarité (avec ce réseau complètement ouvert) rend encore plus précieux les aspects de stabilité et de sécurité

Les concepts de l'OO

Langages et plateformes

- Quels sont les principaux langages orienté objet aujourd'hui?
 - C++
 - Hybride, permet la coexistence d'OO et procédural
 - Puissant et plus complexe
 - Pas de « ramasse-miettes », multihéritage, etc.
 - Java
 - Très épuré et strictement OO
 - Neutre architecturalement (Multi-plateformes)
 - Ramasse-miettes, pas de multihéritage, nombreuses librairies disponibles
 - C#
 - Très proche de Java
 - Conçu par Microsoft

Les concepts de l'OO

En résumé

- Tout est un objet
- L'exécution d'un programme est réalisée par échanges de messages entre objets
- Un message est une demande d'action, caractérisée par les paramètres nécessaires à la réalisation de cette action
- Tout objet est une instance de classe, qui est le « moule » générique des objets de ce type
- Les classes définissent les comportements possibles de leurs objets
- Les classes sont organisées en une structure arborescente à racine unique : la hiérarchie d'héritage
- Tout le code des programmes se trouve entièrement et exclusivement dans le corps des classes
- A l'exception toutefois de deux instructions:
 - package → définit l'ensemble auquel la classe appartient
 - import → permet l'utilisation de classes extérieures au package
- UML permet la représentation graphique des applications

V. Programmation orientée objets en Java

Survol du chapitre

- La création d'objets: Constructeurs et mot-clé « new »
- Les variables: Déclaration et portée
- Les méthodes: Déclaration, interface et surcharge
- L'encapsulation: « public », « private » et « protected »
- Les membres d'instance et de classe: « static »
- Utilisation de l'héritage: « this » et « super »
- Conversion de types
- Polymorphisme
- Classes abstraites
- Interfaces

La création d'objets (1/2)

Le constructeur

- A le même nom que la classe
- Quand un objet est créé, on invoque tout d'abord le constructeur
- Un constructeur utilise comme arguments les arguments de la méthode interne
- On peut surcharger les constructeurs, i.e. en créant plusieurs constructeurs
- Il existe toujours un constructeur. S'il n'en existe pas, le compilateur crée un constructeur par défaut, sans arguments
- Signature d'un constructeur:
 - Modificateur d'accès (en général public)
 - Pas de type de retour
 - Le même nom que la classe
 - Les arguments sont utilisés pour initialiser les attributs

```
public BankAccount(String n,int s,int i)
{
    name=n ;
    solde=s;
    interest=i;
}

public BankAccount(String n)
{
    name=n ;
    solde=0;
    interest=10;
}
```

La création d'objets (2/2)

L'appel au constructeur

- Se fait pour initialiser un objet
 - ➔ Provoque la création réelle de l'objet en mémoire
 - ➔ Par l'initialisation de ses variables internes propres
- Se fait par l'emploi du mot clé « new »

```
BankAccount ba1, ba2;
```

```
ba1 = new BankAccount("Bersini", 10.000, 7);
```

```
ba2 = new BankAccount("Bersini");
```


Les variables

Déclaration des variables membres

(1/2)

- Une variable est un endroit de la mémoire à laquelle on a donné un nom de sorte que l'on puisse y faire facilement référence dans le programme
- Une variable a une valeur, correspondant à un certain type
- La valeur d'une variable peut changer au cours de l'exécution du programme
- Une variable Java est conçue pour un type particulier de donnée

Les variables

Déclaration des variables membres

(2/2)

- Rappel: toute variable doit être déclarée et initialisée
- Les variables membres sont des variables déclarées à l'intérieur du corps de la classe mais à l'extérieur d'une méthode particulière, elles sont donc accessibles depuis n'importe où dans la classe.
- La signature de la variable :
 - Les modificateurs d'accès: indiquent le niveau d'accessibilité de la variable
 - [static]: permet la déclaration d'une variable de classe
 - [final]: empêche la modification de la variable
 - [transient]: on ne tient pas compte de la variable en sérialisant l'objet
 - [volatile]: pour le multithreading
 - Le type de la variable (ex: int, String, double, RacingBike,...)
 - Le nom de la variable (identificateur)

optionnel

Les variables

Portée d'une variable et des attributs

- Portée = Section du programme dans laquelle une variable existe
- La variable ne peut donc pas être utilisée en dehors de cette section
- La portée est définie par les accolades qui l'entourent directement
- Exemple:

```
if(solde < 0){  
    String avertissement = "Attention, solde négatif !"  
}  
else{  
    String avertissement = "Tutti va bene !"  
}  
System.out.println(avertissement);
```

```
// Une erreur apparaîtra dès la compilation, car la variable  
// « avertissement » n'existe pas en dehors du bloc IF
```

- Avantages
 - Rend les programmes plus faciles à corriger
 - Limite le risque d'erreurs liées au réemploi d'un nom pour différentes variables

Exercices

EX 5.1

- Analyser la classe Point
 - Que se passe-t-il à l'exécution du programme?
 - Exécuter le programme pour vérifier

EX 5.2

- Analyser la classe Scope
 - Que se passe-t-il à l'exécution du programme?
 - Exécuter le programme pour vérifier
 - Même question si la méthode job() est redéfinie comme suit:

```
public void job() {  
    i=5;  
    for(int i=0;i<10;i++) {  
        System.out.println(++this.i);  
    }  
}
```

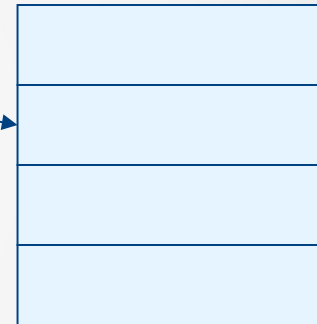
Les variables

Types de référence (1/3)

- Tous les types hormis les types primitifs
- « Pointeur implicite » sur un objet

Référence: Adresse

```
class Point{  
    int x=7, y=10;  
    Point(int x,int y){this.x=x;this.y=y;}  
    void move(int dx,int dy){x+=dx;y+=dy;}  
}
```



Les variables

Types de référence (2/3)

Déclaration et création d'objets

- Déclaration : `Point p;`
- Création : `p = new Point(2,3);`

1. Recherche une place

p

????

p

????

x	0
y	0

2. Assignment d'une valeur

p

????

x	7
y	10

3. Exécution du constructeur

p

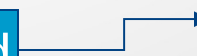
????

x	2
y	3

4. Création du pointeur

p

ox0123abcd



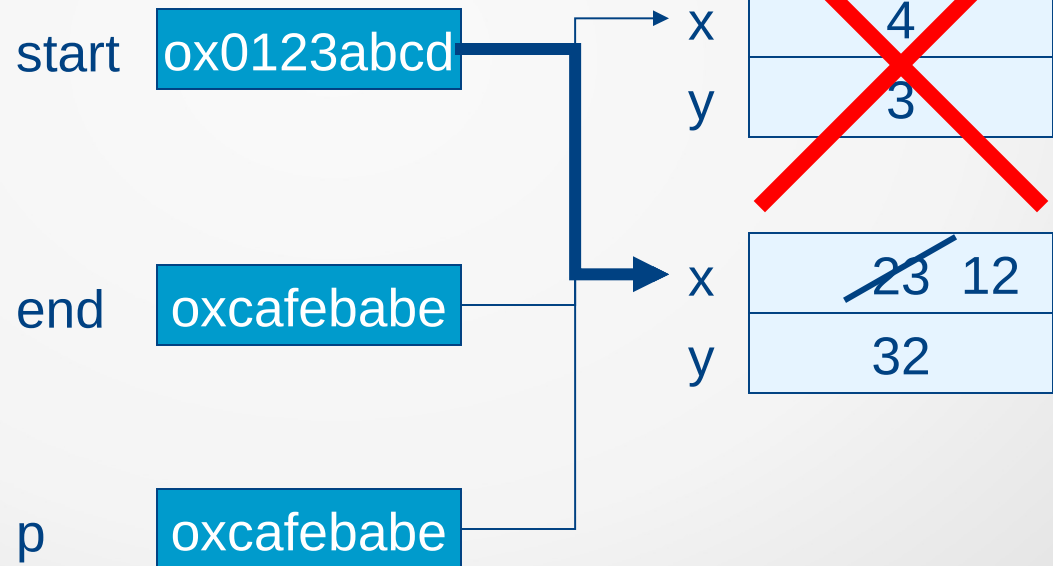
x	2
y	3

Les variables

Types de référence (3/3)

Assignation d'un type de référence

- `Point start=new Point(4,3);`
- `Point end=new Point(23,32);`
- `Point p=end;`
- `p.x=12;`
- `start=p;`



Exercice

- Classe MyPoint
 - Lire le code et prévoir ce qui apparaîtra à l'écran à l'exécution
 - Observer en particulier les mécanismes de création et d'assignation de variables de références dans la classe

Les méthodes (1/4)

Déclaration d'une méthode

```
public void deposit (int amount) {  
    solde+=amount ;  
}
```

Sa déclaration

Son corps

- Une méthode est composée de:

- Signature d'une méthode:

Signature

- Modificateurs d'accès : public, protected, private, aucun
- [modificateurs optionnels] : static, native, synchronized, final, abstract
- Type de retour : type de la valeur retournée
- Nom de la méthode (identificateur)
- Listes de paramètres entre parenthèses (peut être vide mais les parenthèses sont indispensables)
- [exception] (throws Exception)

- Au minimum:

- La méthode possède un identificateur et un type de retour
- Si la méthode ne renvoie rien → le type de retour est **void**

- Les paramètres d'une méthode fournissent une information depuis l'extérieur du "scope" de la méthode (idem que pour le constructeur)

Les méthodes (2/4)

Passage d'arguments

- Les arguments d'une méthode peuvent être de deux types
 - Variable de type primitif
 - Objet
- Lorsque l'argument est une variable de type primitif, c'est la valeur de la variable qui est passée en paramètre
- Lorsque l'argument est un objet, il y a, théoriquement, deux éléments qui pourraient être passés en paramètre:
 - La référence vers l'objet
 - L'objet lui-même
- A la différence de C++, Java considère toujours que c'est la valeur de la référence et non la valeur de l'objet qui est passée en argument

Les méthodes (3/4)

L'interface d'une méthode

- L'interface d'une méthode, c'est sa signature
- Cette signature, qui définit l'interface de la méthode, correspond en fait au message échangé quand la méthode est appelée
- Le message se limite de fait uniquement à la signature de la méthode
 - Type de retour
 - Nom
 - Arguments
- L'expéditeur du message n'a donc jamais besoin de connaître l'implémentation ou corps de la méthode
- On a donc:
 - Déclaration = Signature = Message de la méthode
 - Bloc d'instruction = Corps = Implémentation de la méthode

Les méthodes (4/4)

La surcharge de méthodes

- La surcharge est un mécanisme qui consiste à dupliquer une méthode en modifiant les arguments de sa signature

- Exemple: `int solde ;`

```
public void deposit(int amount){  
    solde+=amount;  
}
```

```
public void deposit(double amount) {  
    solde +=(int) amount;  
}
```

L'encapsulation (1/2)

Raisons d'être

Les modificateurs d'accès qui caractérisent l'encapsulation sont justifiés par différents éléments:

- Préservation de la sécurité des données
 - Les données privées sont simplement inaccessibles de l'extérieur
 - Elles ne peuvent donc être lues ou modifiées que par les méthodes d'accès rendues publiques
- Préservation de l'intégrité des données
 - La modification directe de la valeur d'une variable privée étant impossible, seule la modification à travers des méthodes spécifiquement conçues est possible, ce qui permet de mettre en place des mécanismes de vérification et de validation des valeurs de la variable
- Cohérence des systèmes développés en équipes
 - Les développeurs de classes extérieures ne font appel qu'aux méthodes et, pour ce faire, n'ont besoin que de connaître la signature. Leur code est donc indépendant de l'implémentation des méthodes

L'encapsulation (2/2)

Accès aux membres d'une classe

- En java, les modificateurs d'accès sont utilisés pour protéger l'accessibilité des variables et des méthodes.
- Les accès sont contrôlés en respectant le tableau suivant:

Mot-clé	classe	package	sous classe	world
private	Y			
protected	Y	Y	Y	
public	Y	Y	Y	Y
[aucun]	Y	Y		

Seul les membres publics sont visibles depuis le monde extérieur.

Une classe a toujours accès à ses membres.

Les classes d'un même package protègent uniquement leurs membres privés (à l'intérieur du package)

Une classe fille (ou dérivée) n'a accès qu'aux membres publics et protected de la classe mère.

Exercice

VilleCapitale (1^{ère} partie)

1. Créer une classe Ville correspondant au schéma UML indiqué. Ecrivez la méthode decrisToi() qui affiche à l'écran le nom de la ville et le nombre d'habitants
2. Créez une classe Monde qui utilise la classe Ville. La fonction main dans Monde comprendra les lignes suivantes:

```
Ville[] mesVilles=new Ville[5];  
mesVilles[0]= new Ville(« Gosselies »);  
mesVilles[1]= new Ville(« Namur »);  
...
```

3. Compilez et exécutez en ligne de commande
4. Surchargez le constructeur de la classe Ville. Définissez un constructeur, à deux arguments (un String et un entier), utilisant celui défini en (1)
5. Peut-on modifier la variable d'instance nomVille depuis l'extérieur de la classe ? et depuis l'intérieur ?

Ville

- nbHabitants : int
- nomVille : String

+ Ville(String)

+ decrisToi(): void
+ getNomVille(): String
+ getNbHabitants(): int
+ setNbHabitants(int): void

Membres d'instance et membres de classe (1/2)

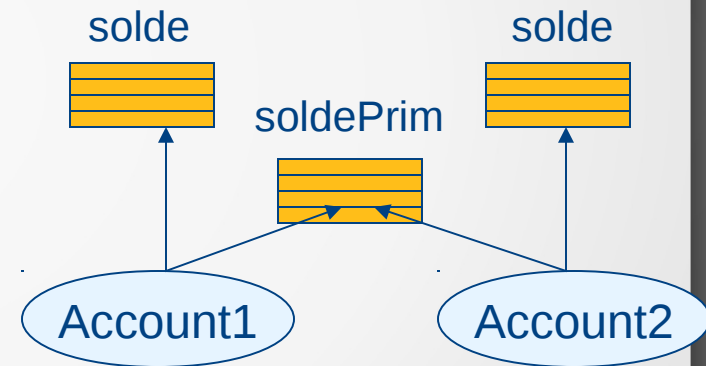
Le mot-clé « static »

- Chaque objet a sa propre “mémoire” de ses variables d'instance
- Le système alloue de la mémoire aux variables de classe dès qu'il rencontre la classe. Chaque instance possède la même valeur d'une variable de classe.

```
class BankAccount {  
    int solde;  
    static int soldePrim;  
    void deposit(int amount){  
        solde+=amount;  
        soldePrim+=amount;  
    }  
}
```

variable d'instance → (pointing to `int solde;`)

variable de classe → (pointing to `static int soldePrim;`)

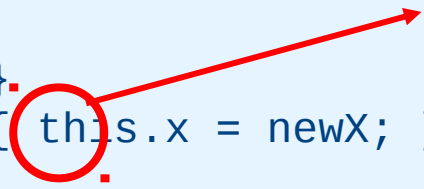


Membres d'instance et membres de classe (2/2)

Le mot-clé « static »

- Variables et méthodes statiques
 - Initialisées dès que la classe est chargée en mémoire
 - Pas besoin de créer un objet (instance de classe)
- Méthodes statiques
 - Fournissent une fonctionnalité à une classe entière
 - Cas des méthodes non destinées à accomplir une action sur un objet individuel de la classe
 - Exemples: `Math.random()`, `Integer.parseInt(String s)`, `main(String[] args)`

```
class AnIntegerNamedX {  
    int x;  
    static public int x() { return x; }.  
    static public void setX(int newX) { this.x = newX; }  
}
```



x est une variable d'instance, donc inaccessible pour la méthode static setX

Exercice

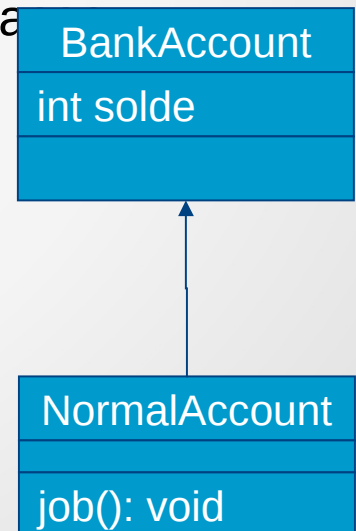
Cercles et leurs surfaces

1. Créer une classe « Cercle » modélisant des ronds destinés à s'afficher à l'écran
 1. Quels attributs devraient être d'instance ou de classe?
 2. Quelles pourraient être ses méthodes d'instance?
 3. Créer les méthode de calcul du périmètre et de la surface
 - Ne peut-on pas en faire des méthodes de classe?
 - Quel avantage y aurait-il à faire cela?
2. Créer une classe « Dessin » qui contient la méthode main
 1. La méthode maininstanciera un Cercle, en affichera les coordonnées et en calculera le périmètre et l'aire

Utilisation de l'héritage (1/5)

- Java n'offre pas la possibilité d'héritage multiple
- La « super super » classe, est la classe *Object* (parente de toute classe)
- Une sous-classe hérite des variables et des méthodes de ses classes parentes
- La clause *extends* apparaît dans la déclaration de la classe

```
class BankAccount {  
    protected int solde;  
    ...  
}  
class NormalAccount extends BankAccount {  
    public void job(){solde+=1000;}  
}
```



Utilisation de l'héritage (2/5)

Cacher des données membres

- La variable aNumber du compte normal cache la variable aNumber de la classe générale compte en banque. Mais on peut accéder à la variable aNumber d'un compte en banque à partir d'un compte normal en utilisant le mot-clé super :

super.aNumber

```
class BankAccount{  
    Number aNumber;  
}  
class NormalAccount extends BankAccount{  
    Float aNumber;  
}
```

Utilisation de l'héritage (3/5)

Les mots-clé « this » et « super »

- Dans une méthode
 - « this » est une référence sur l'objet en cours lui-même
 - « super » permet d'accéder aux membres de la superclasse (peut être nécessaire en cas de redéfinition, par ex.)
- Dans le constructeur
 - Il existe toujours un constructeur. S'il n'est pas explicitement défini, il sera un constructeur par défaut, sans arguments
 - « this » est toujours une référence sur l'objet en cours (de

```
class MyClass{
    int x;
    MyClass(int x){
        this.x=x;// constructeur parent
    }
}
```

```
class Child extends MyClass {
    Child(){
        super(6); // appel du constructeur parent
    }
}
```

Utilisation de l'héritage (4/5)

Les mots-clé « this » et « super »

- En cas de surcharge du constructeur:

```
class Employee {  
    String name,firstname;  
    Address a;  
    int age;  
    Employee(String name,String firstname,Address a,int age){  
        super();  
        this.firstname= firstname;  
        this.name=name;  
        this.a=a;  
        this.age=age;  
    }  
    Employee(String name,String firstname){  
        this(name,firstname,null,-1);  
    }  
}
```

Utilisation de l'héritage (5/5)

Redéfinition de méthodes

- La redéfinition n'est pas obligatoire !! Mais elle permet d'adapter un comportement et de le spécifier pour la sous-classe.

```
class BankAccount {  
    public void computeInterest(){  
        solde+=300;           //annual gift  
    }  
}  
  
class NormalAccount extends BankAccount {  
    public void computeInterest(){  
        super.computeInterest();//call the overridden method  
        solde*=1.07;          //annual increase  
    }  
}
```

- Obligation de redéfinir les méthodes déclarées comme abstraites (abstract)
- Interdiction de redéfinir les méthode déclarées comme finales (final)

Exercice

VilleCapitale (2^{ème} partie)

1. Créez une classe Capitale qui hérite de la classe Ville. Celle-ci comprendra une variable d'instance supplémentaire : *nomPays*. Expliquez le mécanisme de construction d'une classe dérivée. Testez différents cas : appel explicite ou non au constructeur de la classe mère ; existence ou non d'un constructeur sans arguments.
2. Redéfinissez la méthode *decrisToi()*, en faisant appel à la méthode de la classe mère. (*decrisToi()* affichera à l'écran Capitale de *nomPays*: *nomVille* ; *nbHabitants*). Testez.
3. Changez les modificateurs d'accès des données membres de la classe mère, en remplaçant *private* par *protected*. Peut-on accéder à ces variables depuis l'extérieur de la classe Ville ? Quel danger cela présente-t-il (en termes d'encapsulation)?

Conversion de types (1/2)

Définition

- Java, langage fortement typé, impose le respect du type d'un objet
- Toutefois, il est possible de convertir le type d'un objet vers un type compatible
 - Un type A est compatible avec un type B si une valeur du type A peut être assignée à une variable du type B
 - Ex: Un entier et un double
- La conversion de type peut se produire
 - Implicitement (conversion automatique)
 - Explicitement (conversion forcée)
- La conversion explicite s'obtient en faisant précéder la variable du type vers lequel elle doit être convertie entre parenthèses (casting)

```
double d = 3.1416;  
int i = (int) d;
```

Conversion de types (2/2)

Application

- Appliquer un opérateur de « cast » au nom d'une variable
 - Ne modifie pas la valeur de la variable
 - Provoque le traitement du contenu de la variable en tant que variable du type indiqué, et seulement dans l'expression où l'opérateur de cast se trouve
- S'applique aux variables de types primitifs et aux variables de types de références
- Types primitifs:
 - Seulement vers un type plus large (ou risque de perte de données)
 - Interdit pour le type *boolean*
 - Ex: Short → Integer → Long
- Types de références:
 - Vers une classe parent ou une interface implémentée (ou risque d'erreur)
 - Dans ce cas, l'opérateur de cast n'est pas nécessaire (en fonction du contexte)
 - Peuvent toujours être castés vers OBJECT
 - Ex: Voiture → VéhiculesMotorisés → Véhicules → Object

Exercice

Conversion de types

- Analyser le code de l'application « Circulation »
- Observer les mécanismes de conversion de types
- Déterminer les conversions valides et celles qui provoqueront une erreur à la compilation
- Corriger la classe Circulation pour que l'application puisse être compilée et exécutée correctement
- Analyser ensuite la même application avec cette fois la classe « Circulation2 » pour classe principale (sans erreurs) et en observer les mécanismes de conversion de types
- Quels risques pourraient se présenter en utilisant ces techniques sur des collections hétérogènes?

Polymorphisme (1/2)

Définition

- Concept basé sur la notion de redéfinition de méthodes
- Consiste à permettre à une classe de s'adresser à une autre en sollicitant un service générique qui s'appliquera différemment au niveau de chaque sous-classe du destinataire du message
- En d'autres termes, permet de changer le comportement d'une classe de base sans la modifier → Deux objets peuvent réagir différemment au même appel de méthode
- Uniquement possible entre classes reliées par un lien d'héritage et

```
class Bank{  
    BankAccount[] theAccounts = new BankAccount[10];  
    public static void main(String[] args){  
        theAccounts[0] = new NormalAccount("Joe",10000);  
        theAccounts[0].computeInterest();  
    }  
}
```

Polymorphisme (2/2)

Utilisation du polymorphisme sur des collections hétérogènes

```
BankAccount[] ba=new BankAccount[5];
```

```
ba[0] = new NormalAccount("Joe",10000);  
ba[1] = new NormalAccount("John",11000);  
ba[2] = new SpecialAccount("Jef",12000);  
ba[3] = new SpecialAccount("Jack",13000);  
ba[4] = new SpecialAccount("Jim",14000);
```

```
for(int i=0;i<ba.length();i++)  
{  
    ba[i].computeInterest();  
}
```

Exercice

VilleCapitale (3^{ème} partie)

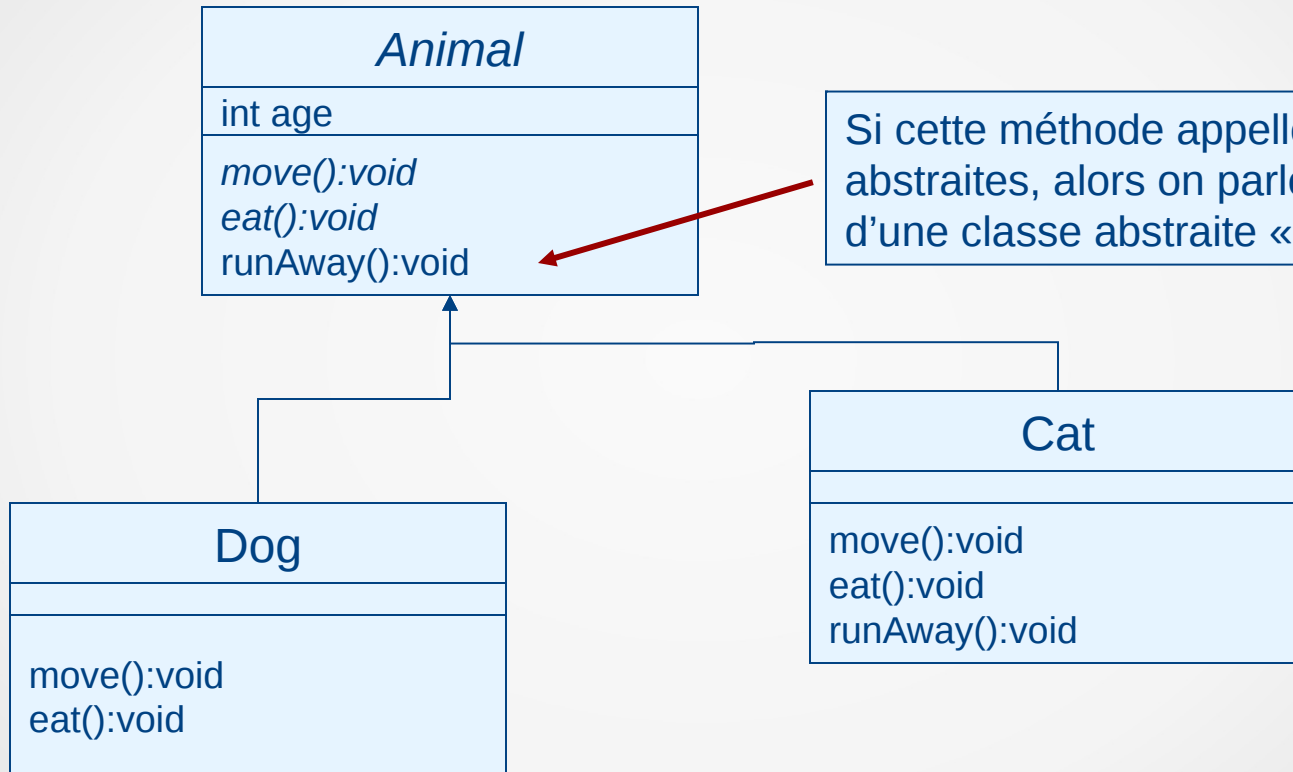
1. Montrez que l'héritage permet le polymorphisme. Pour ce faire créez un tableau de Ville comprenant des villes et des capitales.
2. Soit une Capitale *londres*. Que se passe-t-il à l'exécution de la commande `System.out.println(londres)` ? Changez le nom des methodes `decrisToi()` en le remplaçant par `toString()`. Que se passe-t-il a l'exécution de `System.out.println(londres)` ? Expliquez.
3. Peut-on rendre `decrisToi()` polymorphique dès la conception de la classe mère. Utilisez les méthodes `getClass()` et `getName()`

Les classes abstraites (1/2)

- Une classe abstraite
 - Peut contenir ou hériter de méthodes abstraites (des méthodes sans corps)
 - Peut contenir des constantes globales
 - Peut avoir des méthodes normales, avec corps
- Une classe abstraite ne peut être instanciée
 - On peut seulement instancier une sous-classe concrète
 - La sous-classe concrète doit donner un corps à toute méthode abstraite
- La déclaration d'une classe abstraite contenant une méthode abstraite ressemble à ceci:

```
abstract class Animal {  
    abstract void move();  
}
```

Les classes abstraites (2/2)



Si cette méthode appelle des méthodes abstraites, alors on parle d'une classe abstraite « template »

Les interfaces (1/3)

Définition

- L'interface d'une classe = la liste des messages disponibles
= signature des méthodes de la classe
- Certaines classes sont conçues pour ne contenir précisément que la signature de leurs méthodes, sans corps. Ces classes ne contiennent donc que leur interface, c'est pourquoi on les appelle elles-mêmes *interface*
- Ne contient que la déclaration de méthodes, sans définition (corps)
- Permet des constantes globales
- Une classe peut implémenter une interface, ou bien des interfaces multiples

```
public interface Runnable {  
    public void run();  
}
```

```
public interface GraphicalObject {  
    public void draw(Graphics g);  
}
```

Les interfaces (2/3)

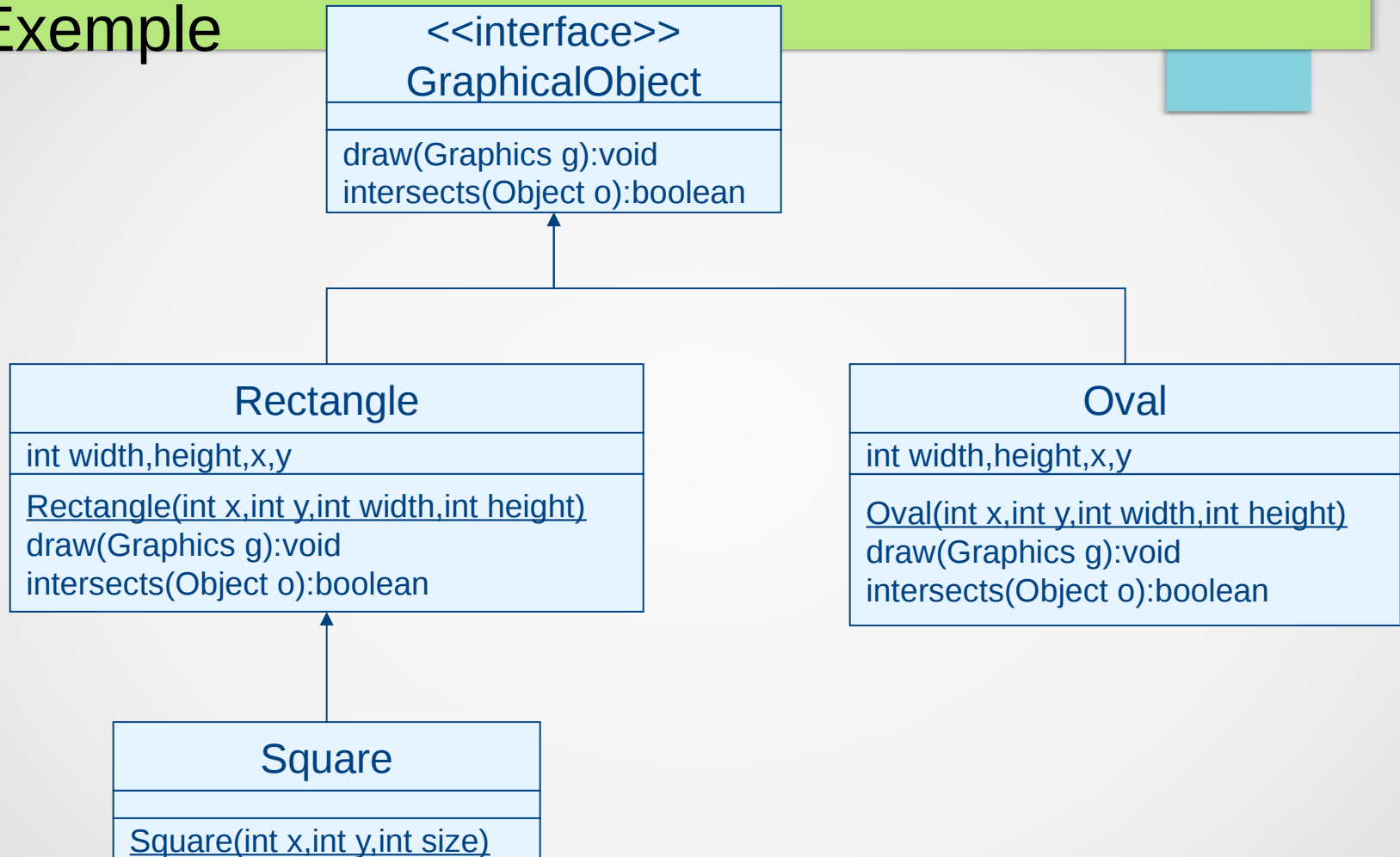
Raisons d'être

- Forcer la redéfinition / l'implémentation de ses méthodes
- Permettre une certaine forme de multi-héritage
- Faciliter et stabiliser la décomposition de l'application logicielle
- D'une classe qui dérive d'une interface, on dit qu'elle implémente cette interface
- Le mot clé associé est donc logiquement: `implements`
- Exemple:

```
public class monApplet extends Applet implements Runnable,
    KeyListener
```

Les interfaces (3/3)

Exemple

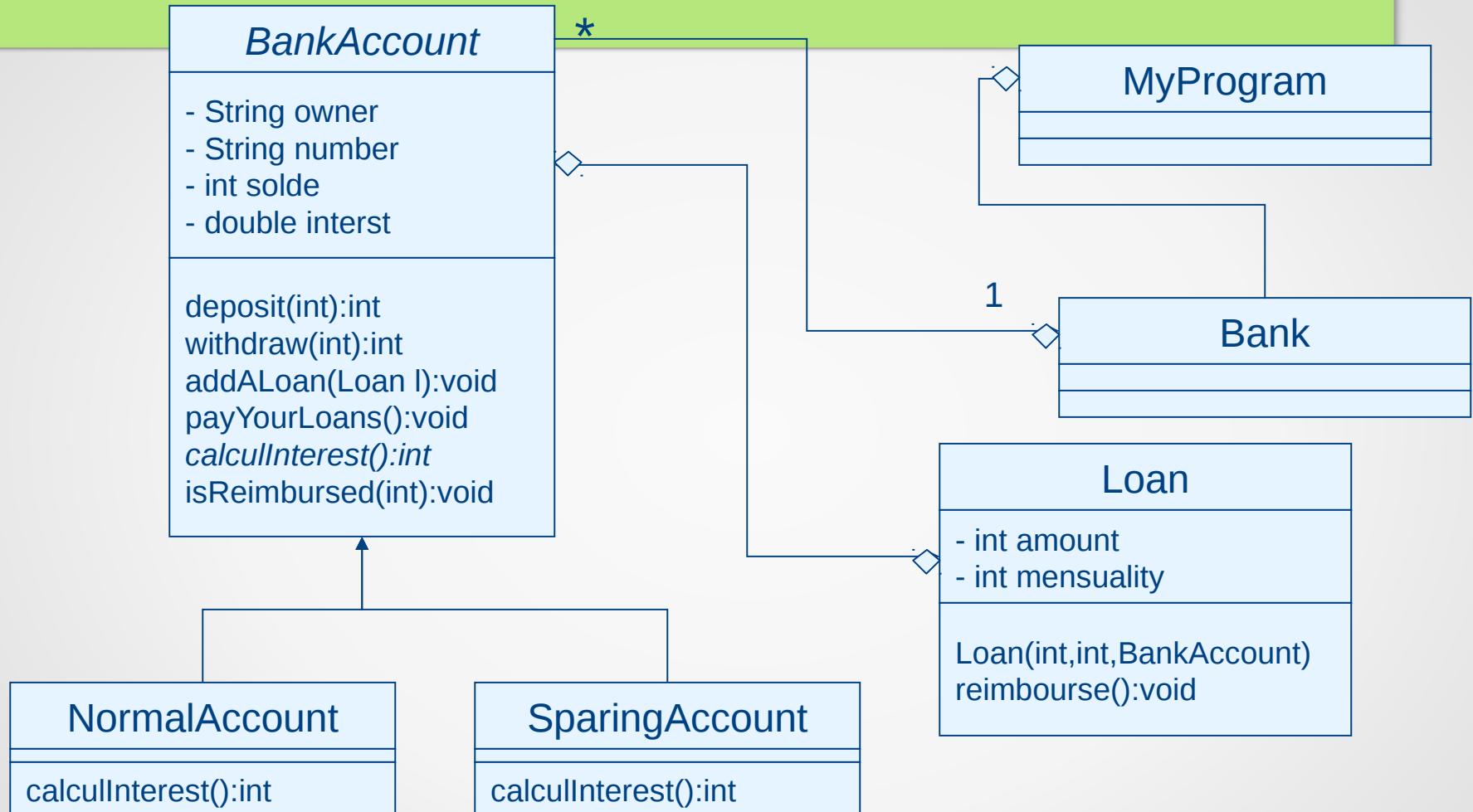


Exercice

Gestion de comptes en banque

- Une banque gère un ensemble de comptes de différents types : comptes courant, comptes d'épargne ... Les comportements et les caractéristiques liés à un compte sont à un niveau abstrait fort similaires : chacun possède un propriétaire, un solde, un numéro, un taux d'intérêt ; on peut y déposer de l'argent, retirer de l'argent (s'il en reste suffisamment)
- La banque propose également un système d'emprunts. Chaque emprunt est caractérisé par le montant total à rembourser, la valeur des mensualités et le compte duquel les mensualités doivent être débitées
- Le but de l'exercice consiste à créer une banque et à tenir à jour l'état des comptes en fonction des retraits, des intérêts et des remboursements d'emprunts sur chaque compte

Exercise



Exercice

- Envoi de messages
 - Ecrire un programme le plus simple possible pour envoyer un message d'un objet A à un objet B, en plaçant néanmoins le main dans une classe spécialement dédiée à cet effet

VI. Structure des API Java

Survol du chapitre

- Introduction
 - Organisation générale des API
- Packages
 - JAVA
 - JAVAX
 - ORG

Introduction

Organisation générale des API Java

- Les différentes API Java sont regroupées en packages
- Ces packages sont eux-mêmes rassemblés dans trois grands groupes, JAVA, JAVAX et ORG
 - JAVA
 - Contient tous les API d'origine de Java 1.0 dont la plupart sont toujours utilisés
 - JAVAX
 - Contient des API réécrits depuis Java 2 et qui sont destinés à les remplacer
 - ORG
 - API provenant de spécifications définies par des organismes internationaux
- La documentation complète des API est toujours disponible sur le site Web de Java
 - <http://java.sun.com/>

Packages JAVA

Package	Description
java.applet	Classes nécessaires à la création d'applets
java.awt	Abstract Windowing Toolkit → Interfaces graphiques, événements...
java.beans	Pour le développement de composants JavaBeans
java.io	Pour la gestion des IO systèmes (système de fichiers, etc.)
java.lang	Classes fondamentales du langage (toujours importées par défaut)
java.math	Pour les traitements arithmétiques demandant une grande précision
java.net	Pour les connexions et la gestion réseau
java.nio	Définit des tampons
java.rmi	Toutes les classes liées au package RMI (Remote Method Invokation)
java.security	Classes et interfaces du framework de sécurité Java
java.sql	Pour l'accès et la gestion des bases de données → JDBC
java.text	Pour la manipulation de texte, dates, nombres et messages
java.util	Collections, modèle événementiel, dates/heures, internationalisation

Packages JAVAX

Package	Description
<code>javax.accessibility</code>	Définit un contrat entre l'U.I. et une technologie d'assistance
<code>javax.crypto</code>	Pour les opérations liées à la cryptographie
<code>javax.imageio</code>	Pour la gestion des IO liées aux images
<code>javax.naming</code>	Pour la gestion de la Java Naming and Directory Interface (JNDI)
<code>javax.net</code>	Pour les connexions et la gestion réseau
<code>javax.print</code>	Pour les services liés à l'impression
<code>javax.rmi</code>	Toutes les classes liées au package RMI (Remote Method Invokation)
<code>javax.security</code>	Classes et interfaces du framework de sécurité Java
<code>javax.sound</code>	Pour le développement d'application gérant le son (Midi / Sampled)
<code>javax.sql</code>	Pour l'accès et la gestion des bases de données → JDBC
<code>javax.swing</code>	Interfaces graphiques « légères », identiques sur toutes plateformes
<code>javax.transaction</code>	Exceptions liées à la gestion des transactions

Packages ORG

Package	Description
org.ietf	Framework pour le développement d'applications avec services de sécurité provenant de mécanismes comme le Kerberos
org.omg	Contient tous les packages liés aux spécifications de l'Object Management Group tels que CORBA, IDL et IOP
org.w3c	Contient un parseur DOM pour XML
org.xml	Contient des parseurs SAX pour XML

VII. Interfaces graphiques et Gestion d'événements

Survol du chapitre

- Introduction
 - Le modèle « Model-View-Control » (MVC)
- Les interfaces graphiques (GUI): Le package AWT
 - AWT v/s SWING
 - La structure de l'AWT
 - Les « Components »
 - Les « Containers »
 - Les « LayoutManagers »
- Gestion d'événements
 - Mécanismes et structure
 - Mise en œuvre
- Graphisme 2D en Java
 - La classe « Rectangle »

Introduction

Le modèle MVC

- Fondement: Séparer
 - Les responsabilités relatives à la saisie des événements
 - Celles relatives à l'exécution des commandes en réponse aux événements
- Séparer au mieux
 - La gestion de l'affichage
 - Le contrôle du composant
 - Les informations intégrées dans le composant
- Avantages:
 - Un même événement peut être envoyé à plusieurs objets écouteurs
 - Utile si un événement est potentiellement intéressant pour plusieurs écouteurs
 - Facilite la réutilisation des composants
 - Permet le développement de l'application et de l'interface séparément
 - Permet d'hériter de super-classes différentes suivant les fonctionnalités
 - Règle essentielle en OO: modulariser au plus ce qui est modularisable

Les interfaces graphiques

AWT v/s SWING

- Les composants des interfaces graphiques sont fournis en Java par deux packages particuliers (et concurrents):
- Java 1: AWT (Abstract Window Toolkit)
 - Composants graphiques « lourds »
 - Chaque composant est relié à son équivalent dans l'OS par un « peer »
 - Look & Feel dépendant de l'OS
- Java 2: SWING
 - Nouveau package
 - Composants graphiques « légers », en pur Java
 - Tous les composants sont détachés de l'OS
 - Look & Feel indépendant de l'OS

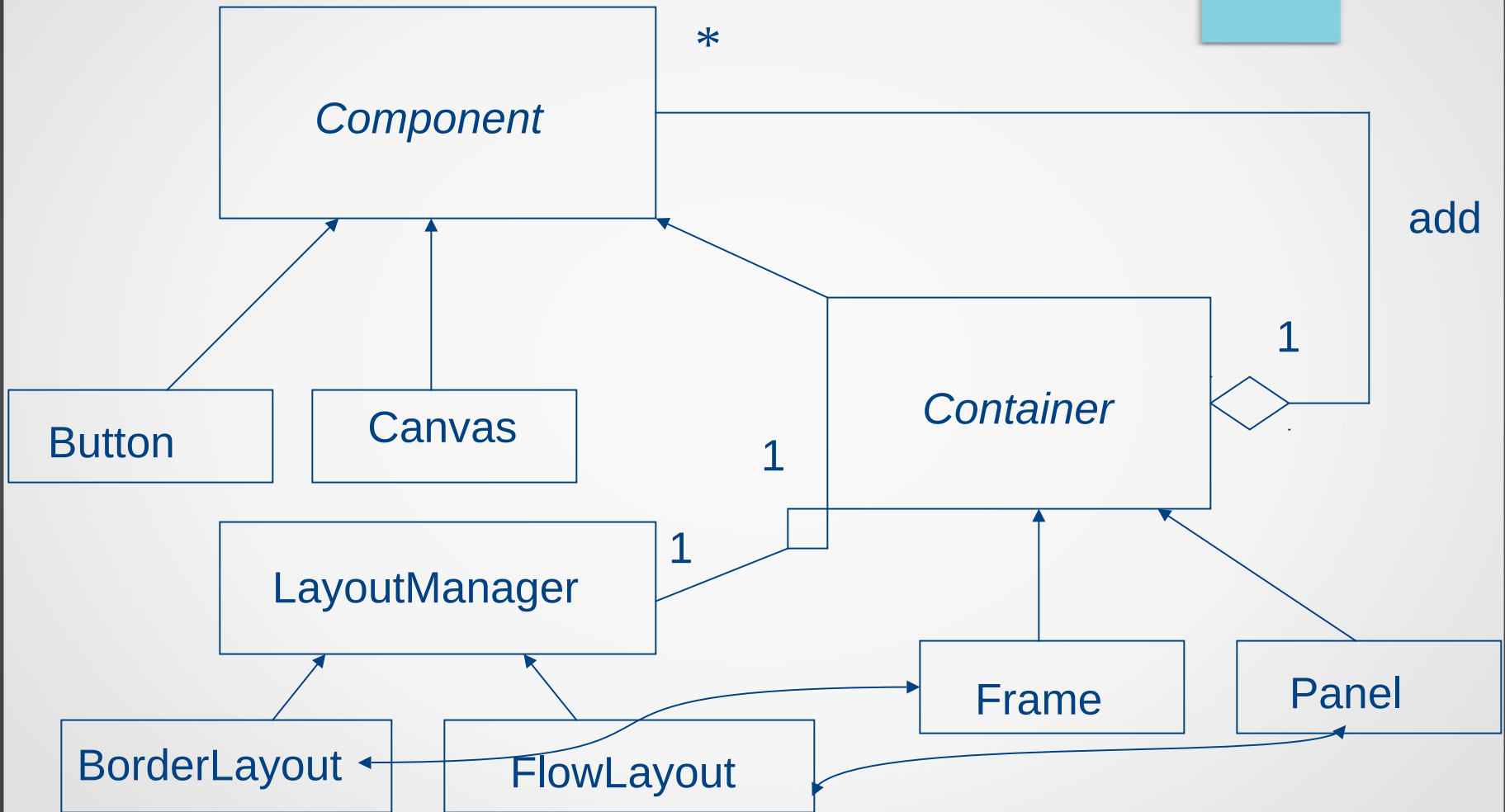
Les interfaces graphiques

Structure de l'AWT (1/2)

- L'AWT offre trois types d'éléments graphiques
 - Les « Containers » (contenants)
 - Les « Components » (composants ou contenus)
 - Les « LayoutManagers » (disposition des objets d'un contenant)
- Les « Containers »
 - Sont destinés à accueillir des composants
 - Gèrent l'affichage des composants
 - Ex: Frame, Panel, Window
- Les « Components »
 - Constituent différents éléments de l'affichage (boutons, barres de menus, etc.)
 - Ex: Button, Canvas, Label, Scrollbar, Checkbox
- Les « LayoutManagers »
 - Gèrent la disposition des composants au sein d'un conteneur

Les interfaces graphiques

Structure de l'AWT (2/2)



Les interfaces graphiques

Les « Components »

- Héritage de méthodes:
 - `paint(Graphics g)` : Affiche le composant
 - `repaint()` : Rafraîchit l'affichage du composant (rappelle la méthode `paint`)
 - `getGraphics()` : Crée un contexte graphique pour le composant
 - Etc. voir documentation Java en ligne
- Composants de formulaires (exemples)
 - `Button` (bouton)
 - `CheckBox` (case à cocher)
 - `Label` (case de texte)
- Composants de fenêtre (exemples)
 - `Menu` (Menu d'une barre de menus)
 - `MenuItem` (Elément d'un menu)

Les interfaces graphiques

Les « Containers »

- Héritage de méthodes:
 - `add(Component c)` : Intègre le composant spécifié à la fin du container
 - `setLayout(LayoutManager l)` : Configure le `LayoutManager` du container
 - Etc. voir documentation Java en ligne
- La classe « `Frame` »
 - Composant du plus haut niveau
 - La fenêtre d'une application est une instance de cette classe
 - Le `Frame` contient les différents composants graphiques de l'application
 - Ne peut être intégré dans un autre conteneur
- Les classes « `Panel` », « `Window` », « `ScrollPane` », etc.
 - Conteneurs essentiels
 - Peuvent être intégrés au sein d'un `Frame`

Les « LayoutManagers » (1/5)

- Rôle
 - Gérer la disposition des composants au sein d'un conteneur
- Types principaux:
 - BorderLayout: divise le conteneur en 5 zones
 - FlowLayout: rajoute les composants au fur et à mesure
 - GridLayout: applique une grille au conteneur pour aligner les composants
 - CardLayout: pour un conteneur qui contient plusieurs cartes
 - GridBagLayout: grille de cellules élémentaires

Les interfaces graphiques

Les « LayoutManagers » (2/5)

BorderLayout

- Principe
 - Positionne les composants suivants les points cardinaux
 - Layout par défaut des « Frame »
- Répartition
 - La largeur l'emporte pour le Nord et le Sud
 - La hauteur l'emporte pour l'Est et l'Ouest
 - Le centre occupe tout ce qui reste
- Utilisation
 - `add(unBouton, BorderLayout.NORTH)`
 - `new BorderLayout(int, int)`
 - Intervalles horizontal et vertical entre les éléments

NORTH		
WEST	CENTER	EAST
SOUTH		

Les interfaces graphiques

Les « LayoutManagers » (3/5)

FlowLayout

- Principe
 - Rajoute les composants au fur et à mesure
 - La taille des composants l'emporte
- Utilisation
 - new FlowLayout (int alignment)
 - alignment: LEFT, CENTER, RIGHT
 - new FlowLayout (int alignment, int hintervalle,int vintervalle)
 - alignment: LEFT, CENTER, RIGHT
 - hintervalle: L'intervalle horizontal entre les composants
 - vintervalle: L'intervalle vertical entre les composants

Les interfaces graphiques

Les « LayoutManagers » (4/5)

GridLayout

- Principe
 - Définit une grille a 2 dimensions
- Utilisation
 - `new GridLayout (int, int)`
 - Définit le nombre de lignes et de colonnes
 - `new GridLayout (int, int, int, int)`
 - Définit le nombre de lignes et de colonnes
 - Définit l'alignement horizontal et vertical

CardLayout

- Principe
 - Pour un container qui contient plusieurs cartes
 - Seule une carte est visible à chaque fois
 - L'utilisateur peut passer d'une carte à l'autre

Les interfaces graphiques

Les « LayoutManagers » (5/5)

GridBagLayout

- Principe
 - Le plus compliqué et le plus flexible
 - Une grille de cellules élémentaires
 - Les composants graphiques peuvent s'étaler indifféremment sur ces cellules
 - Pour positionner un composant, il faut utiliser un objet GridBagConstraints
 - On peut spécifier le comportement des composants quand on étire la fenêtre
- Utilisation
 - `new GridBagLayout()`
 - `new GridBagConstraints();`

Exercice

Une première application graphique (1/2)

- Créer une classe « Fenetre » héritant de *Frame*
- Construire la fenêtre d'application en ajoutant:
 - Au Nord: un *Label*
 - Au Centre: un *Panel*
 - ➔ Contenant lui-même trois *Button*
 - ➔ A placer dans un vecteur
 - Au Sud: un *Button*
- Créer une classe principale contenant uniquement le main:

```
public static void main(String args[]) {  
    Fenetre f = new Fenetre();  
}
```

Gestion d'événements

Mécanismes et structure (1/4)

- Une source d'événements
 - Génère des objets événements
 - Les fait écouter par un ensemble d'écouteurs d'événements
 - En général: un composant ou conteneur graphique
- Les objets événements
 - xxxEvent
 - Contiennent la description et les caractéristiques d'un événement
- Les objets écouteurs
 - xxxListener ou xxxAdapter
 - Concrétisent des méthodes définies dans les Interfaces
 - Indiquent leur réaction en réponse aux événements
 - Sont des interfaces implémentables dans les classes
 - Peuvent être implémentés par les sources d'événements elles-mêmes (Une source d'événements peut « s'écouter » elle-même)

Gestion d'événements

Mécanismes et structure (2/4)

1. Un événement se produit
2. La source d'événement dans laquelle il se produit génère un objet de type événement
3. La source transmet l'événement à son (ses) écouteur(s)
4. L'écouteur appelle la méthode correspondant au type d'événement et lui passe en argument l'objet événement
5. La méthode en question spécifie les traitements à réaliser lorsqu'un événement du type correspondant se produit
6. Dans ses traitements, la méthodes peut examiner les caractéristiques de l'événement (position du curseur de la souris, code de la touche pressée au clavier...) et adapter son comportement en fonction de ces caractéristiques

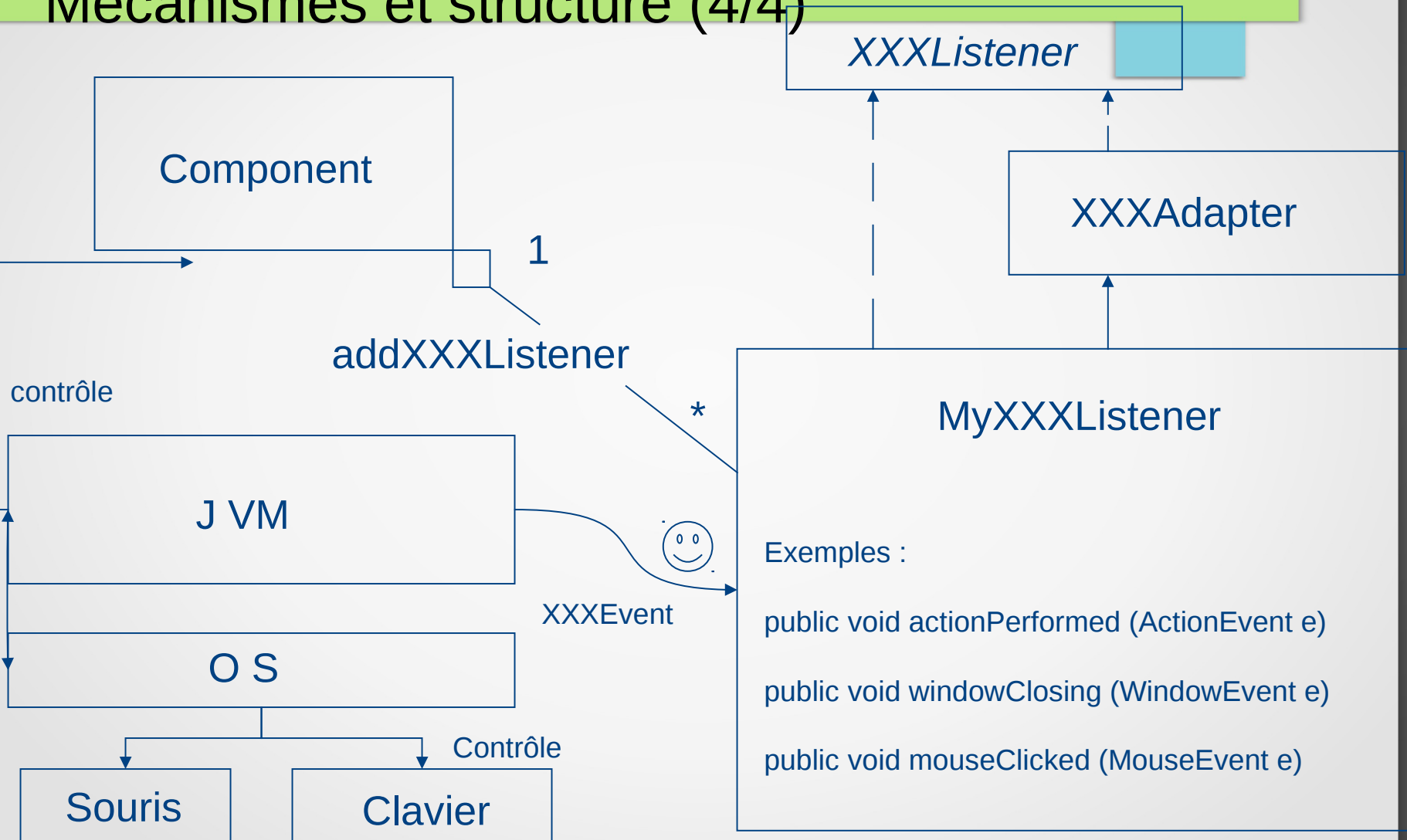
Gestion d'événements

Mécanismes et structure (3/4)

- Evénements
 - ActionEvent, AdjustmentEvent, ComponentEvent, ContainerEvent, FocusEvent, ItemEvent, KeyEvent, MouseEvent, TextEvent, WindowEvent
- Les Interfaces Ecouteurs
 - ActionListener, AdjustmentListener, ComponentListener, ContainerListener, FocusListener, ItemListener, KeyListener, MouseListener, MouseMotionListener, WindowListener
- Les Adapteurs correspondants
 - ActionAdapter, WindowAdapter, KeyAdapter, MouseAdapter, etc.
- Les Sources d'événements
 - Button, List, MenuItem, TextField, ScrollBar, CheckBox, Component, Container, Window

Gestion d'événements

Mécanismes et structure (4/4)



Gestion d'événements

Mise en œuvre

- Par implémentation de l'interface
 - Usage
 - ➔ `public class MaClasse implements MouseListener`
 - Avantages et inconvénients
 - 😊 Meilleur sur le plan orienté objet
 - 😊 La classe peut hériter d'une autre classe
 - 😊 Consistance
- Par héritage de l'adaptateur
 - Usage
 - A chaque interface correspond un adaptateur qui l'implémente lui-même
 - ➔ `public class MaClasse extends MouseAdapter`
 - Avantages et inconvénients
 - 😊 Code simple (l'adaptateur redéfinit déjà les méthodes, etc.)
 - 😞 La classe ne peut plus hériter d'une autre classe

Gestion d'événements

Mise en œuvre – Clics de souris

```
class MonFrame extends Frame implements MouseListener
{
    public MonFrame()
    {
        addMouseListener(this);
    }
    public void mousePressed(MouseEvent e) {}
    public void mouseClicked(MouseEvent e)
    {
        if(e.getX()>50 && e.getY()<100){...}
    }
    public void mouseReleased(MouseEvent e) {}
    public void mouseEntered(MouseEvent e) {}
    public void mouseExited(MouseEvent e) {}
}
```


Gestion d'événements

Mise en œuvre – Déplacements de souris

```
class MonFrame extends Frame implements MouseMotionListener
{
    public MonFrame()
    {
        addMouseMotionListener(this);
    }
    public void mouseDragged(MouseEvent e) {}
    public void mouseMoved(MouseEvent e)
    {
        if(e.getX()>50 && e.getY()<100){...}
    }
}
```

Gestion d'événements

Mise en œuvre – Clavier

```
class MonFrame extends Frame implements KeyListener
{
    public MonFrame()
    {
        addKeyListener(this);
    }
    public void keyPressed(KeyEvent e) {}
    public void keyTyped(KeyEvent e)
    {
        if(e.getKeyCode()==KeyEvent.VK_Q){System.exit(0)}
    }
    public void keyReleased(KeyEvent e) {}
}
```

Gestion d'événements

Mise en œuvre – Événements de fenêtre

```
class MonFrame extends Frame implements WindowListener
{
    public MonFrame()
    {
        addWindowListener(this);
    }

    public void windowClosed(WindowEvent e) {}
    public void windowIconified(WindowEvent e) {}
    public void windowOpened(WindowEvent e) {}
    public void windowClosing(WindowEvent e) {System.exit(0);}
    public void windowDeiconified(WindowEvent e) {}
    public void windowActivated(WindowEvent e) {}
    public void windowDeactivated(WindowEvent e) {}
}
```

Gestion d'événements

Mise en œuvre – Actions

```
public class MonPanel extends Panel implements ActionListener
{ // MonPanel devient un écouteur d'événements
    public void actionPerformed(Action evt)
    { // Ce qui est fait en réponse à l'événement...
        // je peux extraire de l'information sur l'événement
        // transmis par evt
        // je suis obligé de re-définir cette méthode abstraite
        // dans ActionListener
    }
}
```

```
MonPanel panel = new MonPanel();
JButton button = new JButton("OK");// Crée un bouton qui est source
// d'évènements de type ActionEvent
button.addActionListener(panel) // Associer un écouteur
// d'événements panel au bouton
```

Exercice

Une première application graphique (2/2)

- Implémenter l'interface « WindowListener » dans la classe Fenetre
- Redéfinir les méthodes de l'interface et en particulier *windowClosing(WindowEvent e)* pour qu'elle provoque la fermeture de la fenêtre et la sortie de l'application lorsque l'utilisateur clique sur la croix de fermeture

Exercice

- Calculatrice
 - Créer une classe « *Calculette* » héritant de *Frame* et y ajouter:
 - Un *Label* pour l'écran d'affichage
 - Un *Panel* pour le clavier
 - Ajouter un *GridLayout* au *Panel*, le divisant en 4 x 4 zones
 - Un tableau de 16 *Button* qui s'afficheront dans le *Panel* en fonction du *GridLayout*
 - Instancier les boutons du tableau dans une boucle FOR
 - Créer une classe « *GestionnaireDeFenetre* » héritant de *WindowAdapter*
 - Redéfinir sa méthode *windowClosing(WindowEvent e)* pour qu'une boîte de dialogue s'ouvre pour demander la confirmation de la fermeture
 - Créer une classe « *GestionnaireDeBoutons* » implémentant l'interface *ActionListener*
 - Redéfinir sa méthode *actionPerformed(ActionEvent e)* pour qu'elle identifie la touche pressée et déclenche l'action correspondante
 - Créer une classe « *Arithmetique* » effectuant les calculs et renvoyant le résultat qui sera affiché dans le *Label* de la fenêtre principale
 - Créer enfin une classe « *StartCalculatrice* » qui contiendra uniquement le *main*

Graphisme 2D

Fonctionnement

- Tout ce qui est nécessaire aux graphismes en 2D est fourni par la classe Graphics
- Chaque composant est toujours doté d'un objet de ce type
 - Graphics g
- On peut donc invoquer, sur cet objet graphique « g », toutes les méthodes de graphisme:
 - g.drawLine(int, int, int, int)
 - g.drawRect(int, int, int, int)
 - g.fillOval(int, int, int, int)
 - g.setColor(Color)
 - Etc.

Graphisme 2D

Classe `java.awt.Rectangle`

- Classe fournie dans le package `java.awt`
- Idéale pour servir de squelette à la plupart des objets graphiques
 - ➔ On peut inscrire toutes les formes dans un rectangle
- Attributs publics:
 - `int x,y, width, height`
- Méthodes publiques
 - `contains(int x, int y): boolean`
 - Renvoie *true* si les coordonnées (x,y) sont contenues dans le rectangle
 - `contains(Rectangle r): boolean`
 - Renvoie *true* si le rectangle **r** est contenu dans le rectangle en cours
 - `intersects(Rectangle r): boolean`
 - Renvoie *true* si le rectangle **r** touche le rectangle en cours
 - ...

Exercice

Paint (1/4)

- L'objectif du projet est la réalisation d'un éditeur graphique permettant de dessiner des segments de droites colorés (une version BETA de Paint 1.0)
- La réalisation de l'éditeur graphique commencera par la construction d'une fenêtre (Frame), support de l'application, que nous personnalisons grâce à divers éléments : boutons (Button), listes déroulantes (Choice), zone de dessin (Canvas), etc.
- Ensuite nous gérerons les événements générés par l'utilisateur (clic de souris, ...) pour donner vie à cette application
- Enfin en utilisant le contexte graphique fourni par la JVM, nous aborderons les mécanismes de construction graphique en 2D

VIII. Les collections

Survol du chapitre

- Introduction
 - Qu'est-ce qu'une Collection?
 - Le Java Collections Framework
- Interfaces
 - Collections ➔ « Set » et « List »
 - Maps ➔ « Map » et « SortedMap »
 - Itérateurs ➔ « Iterator »
 - Compérateurs ➔ « Comparable » et « Comparator »
- Implémentations
 - HashSet et TreeSet
 - ArrayList et LinkedList
- Algorithmes
 - Tri
 - Autres: Recherche, Mélange, etc.

Introduction

Qu'est-ce qu'un collection?

- Collection
 - Un objet utilisé afin de représenter un groupe d'objets
- Utilisé pour:
 - Stocker, retrouver et manipuler des données
 - Transmettre des données d'une méthode à une autre
- Représente des unités de données formant un groupe logique ou naturel, par exemple:
 - Une main de poker (collection de cartes)
 - Un dossier d'emails (collection de messages)
 - Un répertoire téléphonique (collection de correspondances NOM – N°)

Introduction

Java Collections Framework

- Définit toute la structure des collections en Java
- Constitue une architecture unifiée pour
 - la représentation des collections
 - la manipulation des collections
- Contient des:
 - Interfaces
 - Types de données abstraits représentant des collections
 - Permettent de manipuler les collections indépendamment de leur représentation
 - Organisées en une hiérarchie unique
 - Implémentations
 - Implémentations concrètes des interfaces ➔ Structures de données réutilisables
 - Fournies pour accélérer le développement
 - Algorithmes
 - Méthodes standards fournissant des opérations comme le tri, la recherche, etc.

Interfaces Structure

Il existe 4 groupes d'interfaces liées aux collections

- Collection
 - Racine de la structure des collections
 - Représente un groupe d'objets (dits « éléments »)
 - Peut autoriser ou non les duplicats
 - Peut contenir un ordre intrinsèque ou pas
- Map
 - Un objet qui établit des correspondances entre des clés et des valeurs
 - Ne peut en aucun cas contenir des duplicats
 - ➔ Chaque clé ne peut correspondre qu'à une valeur au plus
- Interfaces de comparaison
 - Permettent le tri des objets du type implémentant l'interface
 - Deux versions: « Comparator » et « Comparable »
- Iterator
 - Permet de gérer les éléments d'une collection

Interfaces

Collection (1/2)

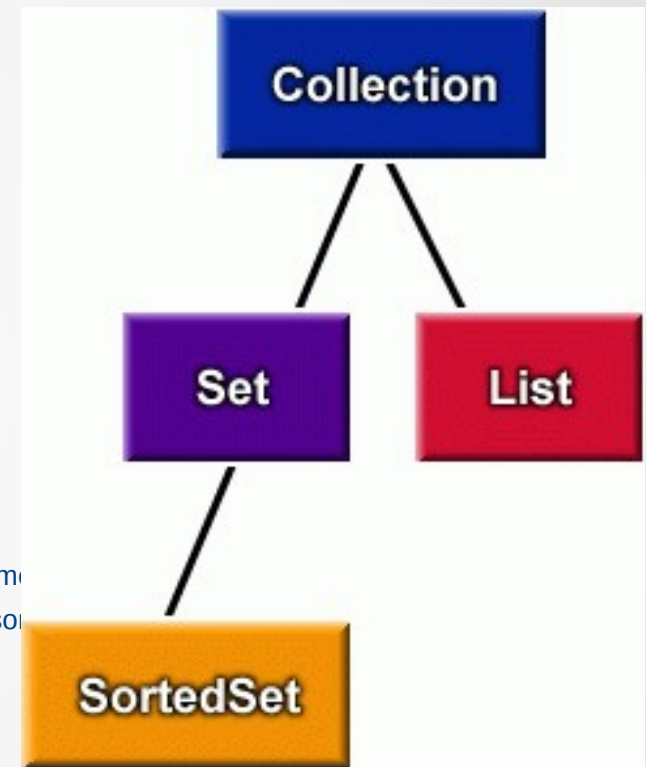
- La racine de la structure des collections
 - Sans ordre spécifique
 - Duplicats permis
- Définit les comportements standards des collections
 - Vérification du nombre d'éléments
 - ➔ *size()*, *isEmpty()*
 - Test d'appartenance d'un objet à la collection
 - ➔ *contains(Object)*
 - Ajout et suppression d'éléments
 - ➔ *add(Object)*, *remove(Object)*
 - Fournir un itérateur pour la collection
 - ➔ *iterator()*
 - Bulk Operations
 - ➔ *addAll(Collections)*, *clear()*, *containsAll(Collections)*

Interfaces

Collection (2/2)

Trois variantes principales:

- **Set**
 - Duplicats interdits
 - Sans ordre spécifique
- **List**
 - Duplicats permis
 - Contient un ordre spécifique intrinsèque
 - Parfois appelé « Séquences »
 - Permet 2 méthodes d'accès particulières:
 - Positional Access: manipulation basée sur l'index numérique des éléments
 - Search: recherche d'un objet en particulier dans la liste et renvoi de son index
- **SortedSet**
 - Duplicats interdits
 - Contient un ordre spécifique intrinsèque



Interfaces

Map

- Map = Objet qui contient des correspondances Clé – Valeur
- Ne peut contenir de doublons (clés sont primaires)
- Fournit des opérations de base standards:
 - `put(key,value)`
 - `get(key)`
 - `remove(key)`
 - `containsKey(key)`
 - `containsValue(value)`
 - `size()`
 - `isEmpty()`
- Peut générer une collection qui représente les couples Clé – Valeur
- Il existe aussi
 - des « Map » permettant de faire correspondre plusieurs valeurs à chaque clé
 - des « SortedMap » fournissant des « Map » naturellement triés

Interfaces

Iterator

- Permet de gérer les éléments d'une collection
- Toute collection peut fournir son « Iterator »
- Permet également de supprimer des éléments d'une collection au cours d'une phase d'itération sur la collection
- Contient 3 méthodes essentielles:
 - `hasNext()`: *boolean* → Indique s'il reste des éléments après l'élément en cours
 - `next()`: *Object* → Fournit l'élément suivant de la collection
 - `Remove()`: *void* → Supprime l'élément en cours

- Exemple:

```
static void filter(Collection c) {  
    for (Iterator i = c.iterator(); i.hasNext(); )  
        if (! cond(i.next()))  
            i.remove();  
}
```

Interfaces

Comparaison (1/3)

Deux interfaces

- Comparable
 - Fournit une méthode de comparaison au sein d'une classe
 - Impose de redéfinir une seule méthode `public int compareTo(Object o)` qui renvoie un entier:
 - 1 si l'objet courant > l'objet « o » fourni dans la méthode
 - 0 si l'objet courant = l'objet « o » fourni dans la méthode
 - 1 si l'objet courant < l'objet « o » fourni dans la méthode
- Comparator
 - Permet de définir une classe servant de comparateur à une autre
 - Impose de redéfinir une seule méthode `public int compare(Object o1, Object o2)` qui renvoie un entier:
 - 1 si `o1 > o2`
 - 0 si `o1 = o2`
 - 1 si `o1 < o2`

Interfaces

Comparaison (2/3)

Exemple d'implémentation de *Comparable*

```
import java.util.*;
public class Name implements Comparable {
    private String  firstName, lastName;

    public Name(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public int compareTo(Object o) {
        Name n = (Name) o;
        int lastCmp = lastName.compareTo(n.lastName);
        if(lastCmp!=0)
            lastCmp = firstName.compareTo(n.firstName);
        return lastCmp;
    }
}
```

Interfaces


Comparaison (3/3)

- Les types primitifs contiennent toujours un ordre naturel

Class	Natural Ordering
Byte	signed numerical
Character	unsigned numerical
Long	signed numerical
Integer	signed numerical
Short	signed numerical
Double	signed numerical
Float	signed numerical
BigInteger	signed numerical
BigDecimal	signed numerical
File	system-dependent lexicographic on pathname.
String	lexicographic
Date	chronological

Implémentations Structure

- Les implémentations sont les types d'objets réels utilisés pour stocker des collections
- Toutes les classes fournies implémentent une ou plusieurs des interfaces de base des collections

		Implementations			
		Hash Table	Resizable Array	Balanced Tree	Linked List
Interfaces	Set	HashSet		TreeSet	
	List		ArrayList		LinkedList
	Map	HashMap		TreeMap	

Implémentations

Sets (1/3)

Deux principales implémentations de l'interface « Set »

- HashSet (Set)
 - Plus rapide
 - N'offre aucune garantie en termes de d'ordre
- TreeSet (SortedSet)
 - Contient une structure permettant d'ordonner les éléments
 - Nettement moins rapide
 - A n'utiliser que si la collection doit être triée ou doit pouvoir être parcourue dans un certain ordre

Implémentations

Sets (2/3)

```
public class FindDups {  
    public static void main(String args[]){  
        Set s = new HashSet();  
        for (int i=0; i<args.length; i++)  
            if (!s.add(args[i]))  
                System.out.println("Duplicate  
detected: "+args[i]);  
        System.out.print(s.size()+" distinct words : "+s);  
    }  
}
```

Produira le résultat suivant:

```
C:> java FindDups i came i saw i left  
Duplicate detected: i  
Duplicate detected: i  
4 distinct words detected: [came, left, saw, i]
```


Implémentations

Sets (3/3)

```
public class FindDups {  
    public static void main(String args[]){  
        Set s = new TreeSet();  
        for (int i=0; i<args.length; i++)  
            if (!s.add(args[i]))  
                System.out.println("Duplicate  
detected: "+args[i]);  
        System.out.print(s.size()+" distinct words : "+s);  
    }  
}
```

Produira le résultat suivant:

```
C:> java FindDups i came i saw i left  
Duplicate detected: i  
Duplicate detected: i  
4 distinct words detected: [came, i, left, saw]
```

Implémentations

Lists (1/2)

Deux principales implémentations de l'interface « List »

- ArrayList (et Vector)
 - La plus couramment utilisée
 - Offre un accès positionnel à vitesse constante → Particulièrement rapide
- LinkedList
 - A utiliser pour
 - Ajouter régulièrement des éléments au début de la liste
 - Supprimer des éléments au milieu de la liste en cours d'itération
 - Mais plus lent en termes d'accès positionnel

Implémentations

Lists (2/2)

Opérations spécifiques aux listes:

- Obtenir la position d'un objet
 - *indexOf(Object o): int*
 - *lastIndexOf(Object o): int*
- Récupérer l'objet en position i
 - *get(int i)* ➔ Renvoie un objet de type « Object » ➔ Devra être converti (cast)
- Placer un objet à une certaine position
 - *set(int i, Object o)*
- Supprimer l'objet en position i
 - *remove(int i)*
 - *removeRange(int fromIndex, int toIndex)*

Algorithmes

Tri

On peut trier un tableau/collection au moyen de méthodes simples:

- Trier un tableau → méthode « sort » de la classe « Arrays »
 - `Arrays.sort(<type>[])`
- Trier une collection → méthodes « sort » de la classe « Collections »
 - Ne fonctionne qu'avec les collections dérivant de « List »
 - Si les éléments de la collection sont comparables (implémentent l'interface « Comparable »)
 - `Collections.sort(List)`
 - Si les éléments de la collection ne sont pas comparables, il faut alors indiquer quelle classe servira de comparateur
 - `Collections.sort(List, Comparator)`

Algorithmes

Autres

- D'autres opérations sont fournies par le Java Collections Framework:
 - Recherche binaire
 - *Collections.binarySearch(liste, clé)*
 - Mélange
 - *Collections.shuffle(liste)*
 - Inversion de l'ordre
 - *Collections.reverse(liste)*
 - Réinitialisation des éléments (remplace tous les éléments par l'objet spécifié)
 - *Collections.fill(liste, objetParDefaut)*
 - Copie des éléments d'une liste dans une autre
 - *Collections.copy(listeSource, listeDestination)*
 - Recherche d'extrema
 - Sur base de la position des éléments *min(liste)* et *max(liste)*
 - Sur base d'un comparateur *min(liste, comparateur)* et *max(liste, comparateur)*

Exercice

Paint (2/4)

- Définissez les classes Droites et Dessin permettant d'assurer la persistance de votre œuvre malgré les redimensionnements de la fenêtre, réductions, changements de focus... Utilisez pour ce faire la classe `java.util.Vector`.
- Redéfinissez la méthode `paint(Graphics g)` héritée de la classe abstraite `Component`.
- Adaptez aussi les méthodes liées aux boutons « Défaire » et « Annuler » afin qu'ils suppriment respectivement la dernière droite tracée et toutes les droites du dessin.
- Implémentez l'interface `Comparable` au sein de la classe « Dessin » afin de permettre le tri des droites sur base de leur longueur et redéfinissez à cette fin la méthode `compareTo(Object o)`
- Ajoutez un bouton permettant de trier les droites du dessin dans le vecteur et vérifiez que la commande « Défaire » supprime désormais les droites par ordre de longueur et non plus en fonction de l'ordre dans lequel elles ont été tracées.

IX. Gestion des exceptions

Survol du chapitre

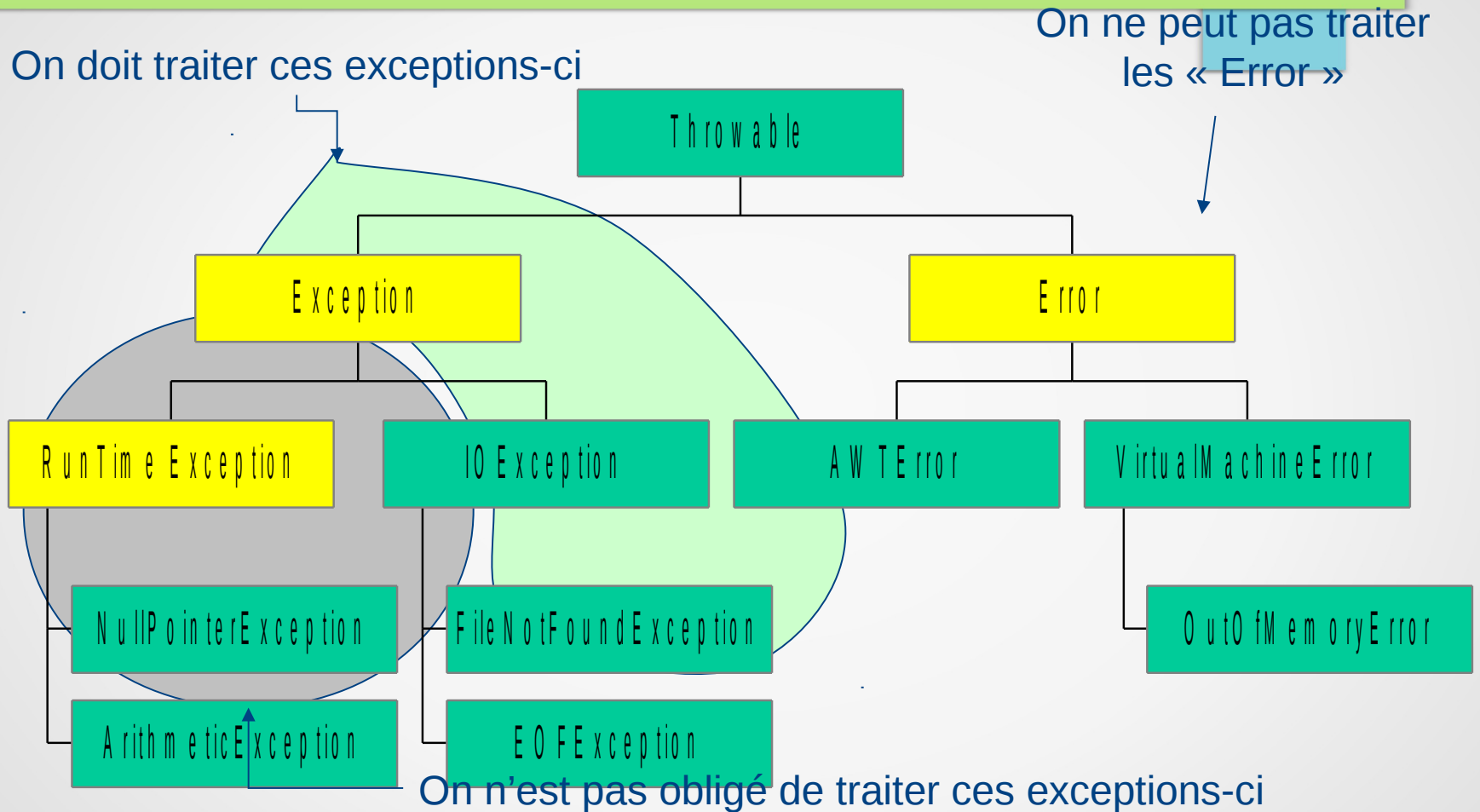
- Introduction
- Hiérarchie des exceptions
- Traitement des exceptions
 - Interception d'exceptions: bloc *try* – *catch* – *finally*
 - Lancement (génération) par une méthode: *throws* et *throw*

Introduction

La gestion des exceptions en Java

- S'approche du C++
- Des erreurs surviennent dans tout programme
- Distinction entre Exception et Error: deux classes apparentées
- La classe Exception traite les erreurs prévisibles qui apparaissent dans l'exécution d'un programme:
 - Panne du réseau
 - Fichier inexistant
 - Problème propre à la logique « business »
- La classe Error traite les conditions sérieuses que le programmeur n'est pas censé traiter

Hiérarchie des exceptions



Traitement des exceptions

Principes

- Le traitement des exceptions contient deux aspects:
 - L'interception des exceptions
 - Utilisation du bloc *try – catch – finally* pour récupérer les exceptions
 - Et réaliser les actions nécessaires
 - Le lancement (la génération) d'exceptions
 - Automatiquement par l'environnement run-time ou la machine virtuelle pour certaines exceptions prédéfinies par Java
 - Explicitement par le développeur dans une méthode avec « throws » et « throw » (en tout cas pour les exceptions créées par le développeur)

Traitement des exceptions

Interception par bloc *try – catch – finally*

(1/2)

```
try
{
    // quelques actions potentiellement risquées
}
catch(SomeException se)
{
    // que faire si une exception de ce type survient
}
catch(Exception e)
{
    // que faire si une exception d'un autre type survient
}
finally
{
    // toujours faire ceci, quelle que soit l'exception
}
```

Traitement des exceptions

Interception par bloc *try – catch – finally*

(2/2)

```
public ExampleException() {  
    for(int i=0;i<3;i++) {  
        test[i]=Math.log(i);  
    }  
    try {  
        for(int i=0;i<4;i++) {  
            System.out.println("log("+i+") = "+test[i]);  
        }  
    } catch(ArrayIndexOutOfBoundsException ae) {  
        System.out.println(« Arrivé à la fin du tableau »);  
    }  
    System.out.println(« Continuer le constructeur »);  
}
```

Traitement des exceptions

Lancement avec les mots-clés *throws* et *throw* (1/4)

- Si une exception peut survenir, mais que la méthode n'est pas censée la traiter elle-même, il faut en « lancer » une instance
- Il faut préciser que la méthode peut lancer ces exceptions

```
public void writeList() {  
    PrintWriter out = new PrintWriter(new FileWriter("OutFile.txt"));  
    for (int i = 0; i < vector.size(); i++)  
        out.println("Valeur = " + vector.elementAt(i));  
}
```

- Peut lancer une `IOException` → doit être attrapée
- Peut lancer une `ArrayIndexOutOfBoundsException`

```
public void writeList() throws IOException,  
    ArrayIndexOutOfBoundsException {
```

Traitement des exceptions

Lancement avec les mots-clés *throws* et *throw* (2/4)

- Une méthode avec une propriété *throws* peut lancer des exceptions avec *throw*
- Toute exception ou erreur lancée pendant l'exécution provient d'un *throw*
- Fonctionne avec les exceptions qui héritent de *Throwable* (la classe de base)

```
class MyException extends Exception {  
    MyException(String msg){  
        System.out.println("MyException lancee, msg =" + msg);  
    }  
}  
  
void someMethod(boolean flag)throws MyException {  
    if(!flag) throw new MyException(«someMethod»);  
    ...  
}
```

Traitement des exceptions

Lancement avec les mots-clés *throws* et *throw* (3/4)

- Une fois l'exception lancée avec *throw*, il faut soit l'attraper (avec *catch*), soit la relancer. Si vous la relancez, votre méthode doit la déclarer.

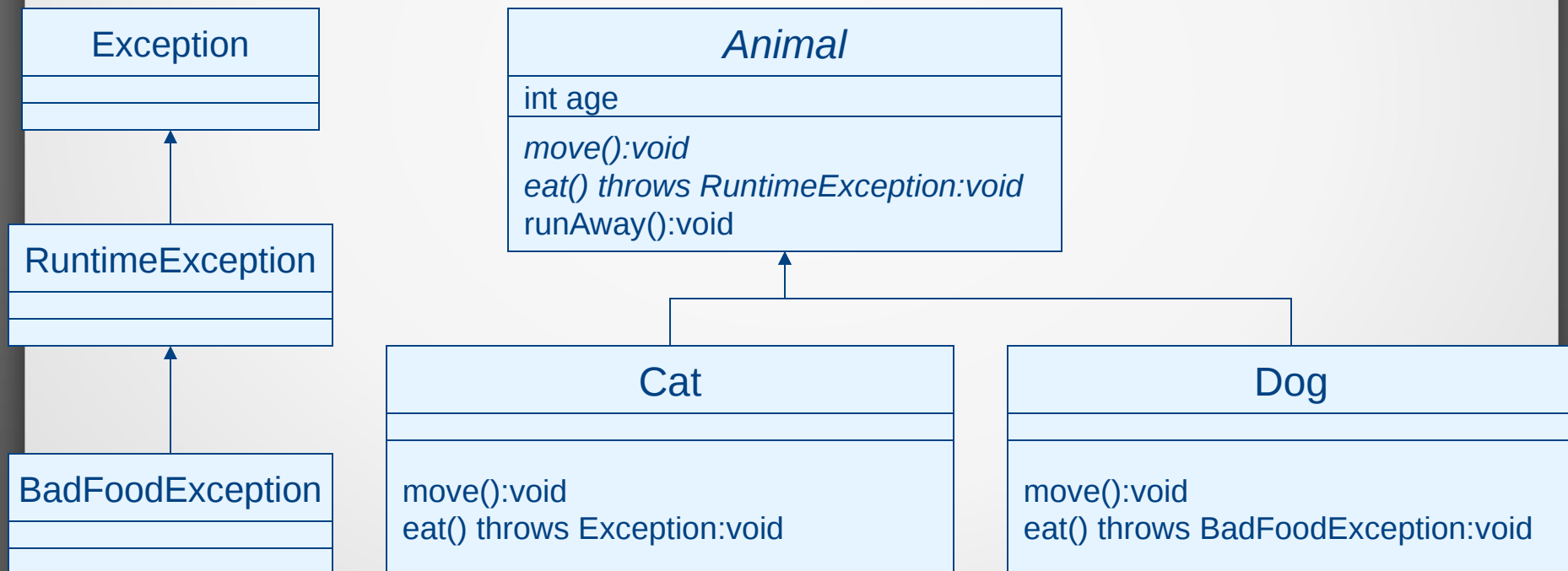
```
public void connectMe(String serverName) throws ServerTimedOutException {
    boolean success;
    int portToConnect = 80;
    success = open(serverName, portToConnect);
    if (!success) {
        throw new ServerTimedOutException("Connection impossible", 80);
    }
}
```

```
public void connectMe(String serverName) {
    boolean success;
    int portToConnect = 80;
    success = open(serverName, portToConnect);
    if (!success) {
        try{ throw new ServerTimedOutException("Connection impossible ", 80);
        } catch(ServerTimedOutException stoe){ }
    }
}
```


Traitement des exceptions

Lancement avec les mots-clés *throws* et *throw* (4/4)

- Si la méthode à redéfinir lance une exception
 - ➔ On ne peut pas lancer une « autre » exception
 - ➔ Mais on peut utiliser l'héritage



Exercice

Paint (3/4)

- Prévoir l'interception d'exceptions de type `ArrayIndexOutOfBoundsException` dans les méthodes de la classe `Dessin` et provoquer délibérément une sortie des limites du vecteur pour vérifier son bon fonctionnement
- Créer une nouvelle exception « `DroiteTropLongueException` », qui devra être lancée chaque fois que l'utilisateur tente de tracer une droite dont la longueur dépasse une certaine limite
- Lancer cette exception dans la méthode *`termineDroite(int, int)`* de la classe *`ZoneGraphique`*
- Intercepter cette nouvelle exception dans la méthode *`mouseReleased(MouseEvent)`* de la classe *`GestionSouris`* et faire en sorte qu'elle provoque l'affichage d'un message d'erreur dans la barre d'état

X. Multithreading

Survol du chapitre

- Introduction
 - Définition
 - Raison d'être
- Création de Thread
 - Par implémentation
 - Par héritage
- Gestion de Thread
 - Méthodes de gestion
 - Diagrammes d'état

Introduction

Qu'est-ce qu'un Thread?

- Un ordinateur qui exécute un programme :
 - Possède un CPU
 - Stocke le programme à exécuter
 - Possède une mémoire manipulée par le programme➔ Multitasking géré par l'OS
- Un thread (« file ») a ces mêmes capacités
 - A accès au CPU
 - Gère un processus
 - A accès à la mémoire, qu'il partage avec d'autres files➔ Multithreading géré par la JVM

Introduction

Pourquoi le multithreading?

- Un programme moderne est composé de
 - Une interface graphique
 - Quelques composantes pouvant agir de manière autonome
- Sans multithreading
 - Les composantes ne pourraient agir que lorsque l'interface est suspendue
- Plus généralement, le multithreading
 - Permet de réaliser plusieurs processus indépendants en parallèle
 - Permet de gérer les files d'attente
 - Permet au besoin de synchroniser les différents processus entre eux

Création de Thread

Mise en œuvre (1/3)


- Par implémentation de l'interface
 - Usage
 - ➔ `public void MaClasse implements Runnable`
 - Avantages et inconvénients
 - 😊 Meilleur sur le plan orienté objet
 - 😊 La classe peut hériter d'une autre classe
 - 😊 Consistance
- Par héritage de la classe Thread elle-même
 - Usage
 - ➔ `public void MaClasse extends Thread`
 - Avantages et inconvénients
 - 😊 Code simple (l'objet est un Thread lui-même)
 - 😞 La classe ne peut plus hériter d'une autre classe

Création de Thread

Mise en œuvre (2/3)

```
public class MaFile implements Runnable {
    public void run(){
        byte[] buffer=new byte[512];
        int i=0;
        while(true){
            if(i++%10==0)System.out.println(""+i+" est divisible par 10");
            if (i>101) break;
        }
    }
}
```

```
public class LanceFile {
    public static void main(String[]arg){
        Thread t=new Thread(new MaFile());
        t.start();
    }
}
```

 Le constructeur de la classe Thread attend un objet Runnable en argument


Création de Thread

Mise en œuvre (3/3)

```
public class MyThread extends Thread {
    public void run(){
        byte[] buffer=new byte[512];
        int i=0;
        while(true){
            if(i++%10==0) System.out.println(""+i+" est divisible par 10");
            if(i>101) break;
        }
    }
}

public class LaunchThread{
    public static void main(String[]arg){
        MyThread t=new MyThread();
        t.start();
    }
}
```

Grâce à l'héritage, un objet de type MyThread est lui-même Runnable, on peut donc appeler un constructeur sans argument



Gestion des Thread

Méthodes de gestion (1/4)

- `t.start()`
 - Appeler cette méthode place le thread dans l'état "runnable"
→ Eligible par le CPU
- `t.yield()` throws `InterruptedException`
 - La VM arrête la file active et la place dans un ensemble de files activables. (runnable state)
 - La VM prend une file activable et la place dans l'état actif (running state)
- `t.sleep(int millis)` throws `InterruptedException`
 - La VM bloque la file pour un temps spécifié (état « d'attente »)
- `t.join()` throws `InterruptedException`
 - Met la file en attente jusqu'au moment où la file t est terminée (a fini sa méthode `run()`). Le thread appelant redevient alors activable.

Gestion des Thread

Méthodes de gestion (2/4)

- `Thread.yield()` throws `InterruptedException`
 - La VM arrête la file active et la place dans un ensemble de files activables. (runnable state)
 - La VM prend une file activable et la place dans l'état actif (running state)
- `Thread.sleep(int millis)` throws `InterruptedException`
 - La VM bloque la file pour un temps spécifié (état « d'attente »)
- `t.join()` throws `InterruptedException`
 - Met la file en attente jusqu'au moment où la file t est terminé (a fini sa méthode `run()`). Le thread appelant redevient alors activable.

Gestion des Thread

Méthodes de gestion (3/4)

```
public class ExJoin {  
    public static void main(String[]arg){  
        Thread t=new Thread(new FileSecondaire());  
        t.start();  
        for(int i=0;i<20;i++){  
            System.out.println("File principale en cours d'exécution "+i);  
            try{  
                Thread.sleep(10);  
            } catch(InterruptedException ie){}  
        }  
        try {  
            t.join();  
        } catch (InterruptedException ie) {}  
        System.out.println("t termine son exécution, fin programme");  
    }  
}
```

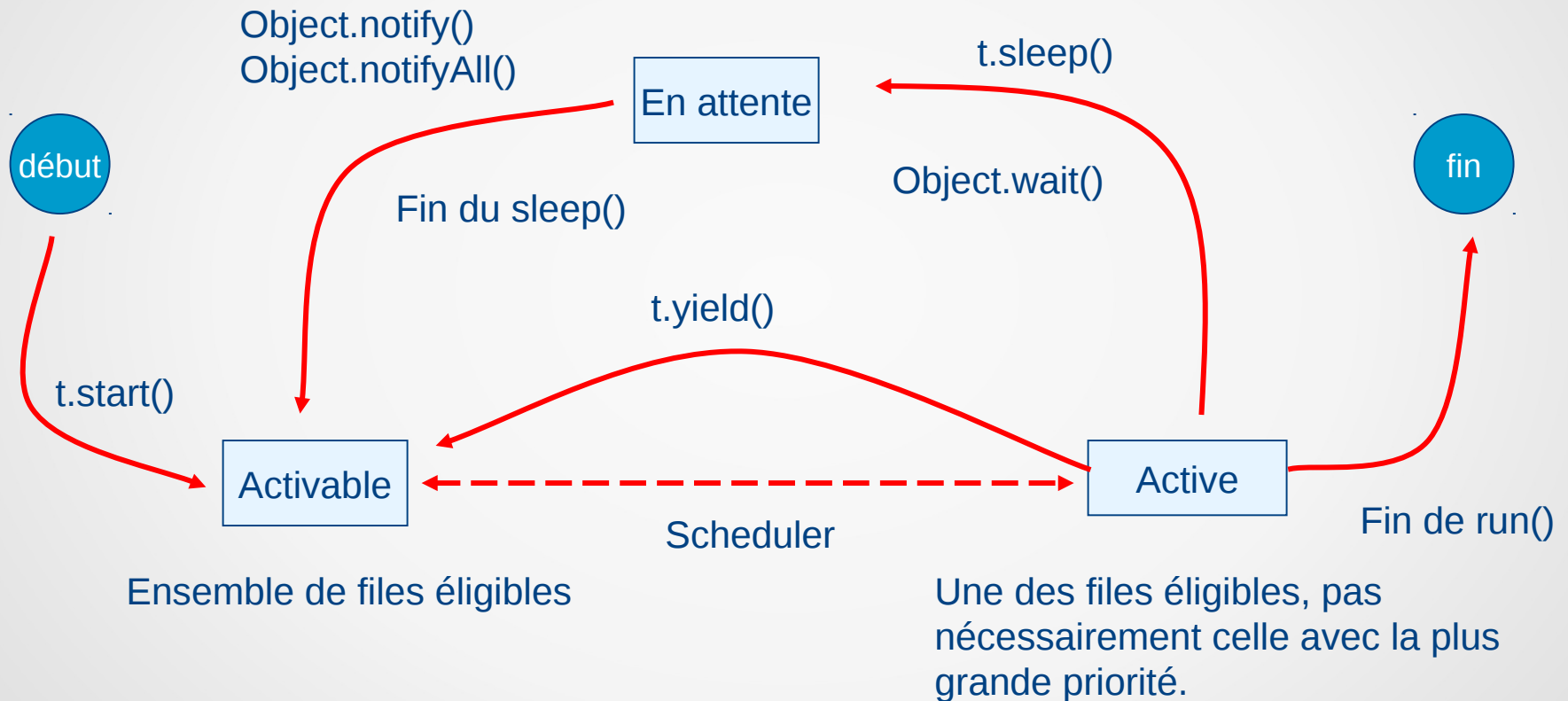
Gestion des Thread

Méthodes de gestion (4/4)

```
class FileSecondaire implements Runnable{
    public void run(){
        for(int i=0;i<40;i++){
            System.out.println("File secondaire en execution "+i);
            try{
                Thread.sleep(10);
            } catch (InterruptedException ie) {}
        }
        System.out.println("Fin de file secondaire");
    }
}
```

Gestion des Thread

Diagrammes d'état



Exercice

Paint (4/4)

- Créer une classe « Balle » héritant de « Thread »
- Définir ses variables et ses méthodes d'affichage *paint(Graphics g)* et *deplaceToi()*
- Redéfinir la méthode *run()* contenant une boucle infinie dans laquelle le déplacement de la balle est provoqué
- Instancier deux balles dans la zone graphique et lancer leurs *Thread* associés
- Implémenter l'interface « Runnable » dans la classe *ZoneGraphique* afin que l'écran soit automatiquement rafraîchi à intervalles réguliers.
- Redéfinir la méthode *run()* et lancer le *Thread* dans le constructeur de la classe

XI. Annexes

Survol du chapitre

- Java et ses concurrents
 - Java v/s CGI
 - Java v/s Microsoft .Net
 - Java v/s Perl
- Les utilitaires de Java
 - Javac
 - Java
 - Javadoc
 - Jar

Java et ses concurrents

Java v/s CGI

- CGI est une interface, non un langage
 - ➔ Convention pour invoquer des exécutables à partir d'un serveur Web
 - En CGI: `program?Hello%21+Here%20I%20come%21`
 - En Java: `args[0] = "Hello!", args[1] = "Here I come!«`
- Un programme CGI peut être implémenté dans différents langages
 - C
 - Perl
 - C++
 - Java
 - VB
 - Etc.

Java et ses concurrents

Java v/s Microsoft .Net

- Fonctionnalité « équivalente »
 - Pages Web dynamiques: JSP ⇔ ASP
 - Objets distribués: DCOM ⇔ RMI, EJB
 - Accès DB: ODBC ⇔ JDBC
- Java
 - Code Java ⇒ Bytecode ⇒ Multiples OS
- .NET
 - Tout Code ⇒ MS IL ⇒ Windows

Java et ses concurrents

Java v/s Perl

- Aspects communs:
 - Langages à usage général
 - Interprétés
 - Portables
 - Grandes bibliothèques à disposition
- Différences
 - Perl: pour administrateurs systèmes, faiblement typé, intégration facile avec le shell
 - Java: pour développeurs d'applications, fortement typé, objets distribués avec les EJB
 - Modèle Client/Serveur:
 - Java peut être utilisé sur un client en tant qu'applet et sur un serveur
 - Perl peut être utilisé sur un serveur, mais pas en tant qu'applet

Les utilitaires de Java

- javac
 - Compilateur, traduit fichier source .java en fichier bytecode .class
- java
 - Interpréteur java, lance des programmes
- javadoc
 - Générateur de documentation d'API
- jar
 - Utilitaire d'archivage et de compression
- javah
 - Générateur de fichiers C/C++ « .h »

Les utilitaires de Java

Javac et Java

- Javac
 - Compile un fichier source .java ou un package entier
 - Exemples:
 - `javac MyBankAccount.java`
compile le fichier mentionné, qui doit se trouver dans le package par défaut
 - `javac be\newco*.java -d c:\classes`
compile tout le package `be.newco` et génère du code compilé dans `c:\classes`, qui doit exister
- Java
 - Lance un programme principal
 - Exemples:
 - `java bankStream.MyProgram`
Lance le programme spécifié par la méthode `public static void main(String[] args)` dans la classe `MyProgram` qui se trouve dans le package `bankStream`.
 - Possibilité de spécifier un `classpath`: un chemin de recherche où `java` est censé de trouver ses classes

Les utilitaires de Java

Javadoc – Générateur de documents

- Nécessité d'une documentation suffisante
 - pour aider les membres de l'équipe
 - pour s'aider soi-même
 - javadoc: une partie de votre documentation
- Intégrer code et documentation
 - résoudre problème de maintenance de la documentation
 - informations dans le code lui-même
- Lancer Javadoc
 - Se mettre dans le répertoire parent de vos packages
 - Pour créer la javadoc, taper `javadoc -d c:\mydir\html demo`
 - Conditions
 - javadoc dans le PATH
 - répertoire destination (-d) doit exister
 - « demo » est le nom d'un package
 - Commentaires délimités par `/**` et `*/`

Les utilitaires de Java

Jar – Utilitaire d'archivage

- Permet de grouper et compresser des fichiers utilisés par un programme Java
- Syntaxe d'utilisation similaire à tar
 - `jar cf myjarfile.jar *.class`
archivage de tout fichier .class, trouvé dans le répertoire courant et tout sous-répertoire, dans le fichier myjarfile.jar
 - `jar xf myjarfile.jar`
Extrait tout fichier contenu dans myjarfile.jar vers une structure de répertoires
- l'interpréteur java reconnaît les fichiers .jar et peut les traiter comme un répertoire.
 - `java -cp myarchive.jar be.newco.MyMain`
Lance le `main()` contenu dans `be.newco.MyMain`, tout en ajoutant les fichiers de `myarchive.jar` dans le `classpath`