

Java

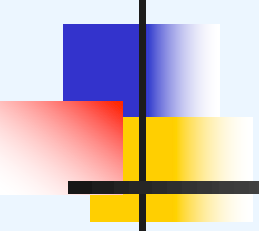


Bernard Archimède
Professeur de Universités
EC0908SI01



Plan

- ❑ Qu'est-ce que Java
- ❑ Historique
- ❑ Références
- ❑ Ses caractéristiques
- ❑ Les outils
- ❑ Les API (Application Programming Interfaces)



Qu'est-ce que Java ?

- ❑ Un langage de programmation orienté objets
- ❑ Une architecture de *Virtual Machine*
- ❑ Un ensemble d'API variées
- ❑ Un ensemble d'outils (le JDK)



Historique de Java (1)

- ❑ Développé à partir de décembre 1990 par une équipe de Sun Microsystems dirigée par James Gosling

- ❑ Au départ, il s'agissait de développer un langage de programmation pour permettre le dialogue entre de futurs ustensiles domestiques

- ❑ Les langages existants tels que C++ ne sont pas à la hauteur
 - ✧ Recompilation dès qu'une nouvelle puce arrive,
 - ✧ Complexité de programmation pour l'écriture de logiciels fiables...



Historique de Java (2)

- ❑ 1990 : Ecriture d'un nouveau langage plus adapté à la réalisation de logiciels embarqués, appelé OAK
 - ✧ Petit, fiable et indépendant de l'architecture
 - ✧ Destiné à la télévision interactive
 - ✧ Non rentable sous sa forme initiale
- ❑ 1993 : le WEB « décolle », Sun redirige ce langage vers Internet : les qualités de portabilité et de compacité du langage OAK en ont fait un candidat parfait à une utilisation sur le réseau. Cette réadaptation prit près de 2 ans.
- ❑ 1995 : Sun rebaptisa OAK en Java (*nom de la machine à café autour de laquelle se réunissait James Gosling et ses collaborateurs*)



Historique de Java (3)

- ❑ Langage indépendant de toute architecture
- ❑ Idéal pour programmer des applications utilisables dans des réseaux hétérogènes, notamment Internet.
- ❑ Enjeu stratégique pour Sun et l'équipe écrit un navigateur appelé HotJava capable d'exécuter des programmes Java.
1995
- ❑ Netscape 2.0 a été développée pour supporter Java, suivi de près par Microsoft (Internet Explorer 3) - **1995**
- ❑ L'intérêt pour la technologie Java s'est accru rapidement: IBM, Oracle et d'autres ont pris des licences Java.



Historique de Java (4)

- ❑ En mai 2007, Sun publie l'ensemble des outils Java dans un « package » OpenJDK sous licence libre.
- ❑ La société Oracle a acquis en 2009 l'entreprise Sun Microsystems. On peut désormais voir apparaître le logo Oracle dans les documentations de l'api Java.
- ❑ Août 2012: faille de sécurité importante dans Java 7



Les différentes versions de Java

- ✧ Java 1.0 en 1995
- ✧ Java 1.2 en 1999 (Java 2, version 1.2)
- ✧ Java 1.4 en 2002 (Java 2, version 1.4)
- ✧ Java 5 en 2004
- ✧ Java 6 en 2006
- ✧ Java 7 en 2011
- Évolution très rapide et succès du langage
- Une certaine maturité atteinte avec Java 2
- Mais des problèmes de compatibilité existaient
 - ✧ entre les versions 1.1 et 1.2/1.3/1.4
 - ✧ avec certains navigateurs



Références (1)

□ Bibliographie

- ✧ Au cœur de Java 2 : Volume I - Notions fondamentales.
C. Hortsman et G. Cornell. The Sun Microsystems Press.
Java Series. CampusPress.
- ✧ Au cœur de Java 2 : Volume II - Fonctions avancées.
C. Hortsman et G. Cornell. The Sun Microsystems Press.
Java Series. CampusPress.
- ✧ Passeport pour l'algorithmique objet. Jean-Pierre Fournier.
Thomsom Publishing International.



Références (2)

□ Webographie

✧ Pour récupérer le kit de développement de Sun

- <http://java.sun.com/> (racine du site)

✧ Outils de développement

- Eclipse : <http://www.eclipse.org>
- JBuilder 5 : <http://www.borland.fr/download/jb5pers/>

✧ Des exemples de programmes commentés

- <http://www.technobuff.com/javatips/> (en anglais)
- <http://developer.java.sun.com/developer/JDCTechTips/> (en anglais)



Les fausses rumeurs sur Java

- ❑ Java n'a rien de commun avec HTML
- ❑ Java n'est pas un langage de script (type Perl ou TCL)
- ❑ Java != JavaScript (c'est un langage généraliste, type C++)
- ❑ Java != C++ (c'est un langage purement objet, de plus haut niveau, plus proche de SmallTalk)
- ❑ Java ne sert pas seulement à faire des applets (Il est raisonnable d'envisager des développements importants : *cf.* JDK, HotJava, JigSaw, Castanet, Lotus Kona, Applix, ...)



Les points faibles de Java

- ❑ Java est jeune et encore en gestation.
- ❑ Les JVM sont encore lentes.
- ❑ Java est gourmand en mémoire
- ❑ Certaines constructions du langage sont décevantes.
- ❑ Certaines API sont décevantes.
- ❑ Java est, actuellement, un langage propriétaire (Sun).
- ❑ Les environnements de programmation commencent seulement à apparaître (debugger, profiler,...).

Les caractéristiques du langage Java

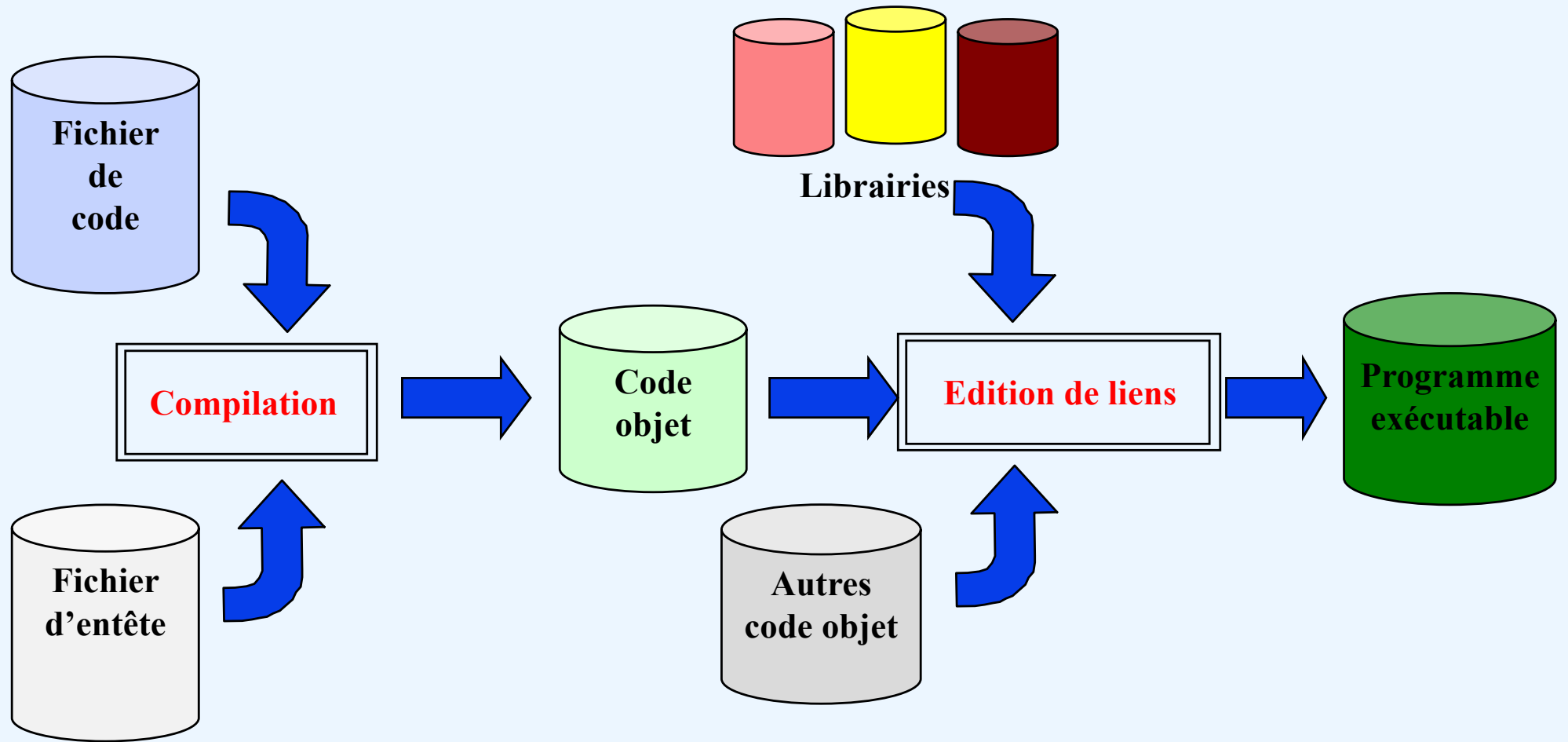
- ❑ Orienté objets
- ❑ Robuste
- ❑ Interprété
- ❑ Sécurisé
- ❑ Portable
- ❑ Multi-threads
- ❑ Simple
- ❑ Distribué



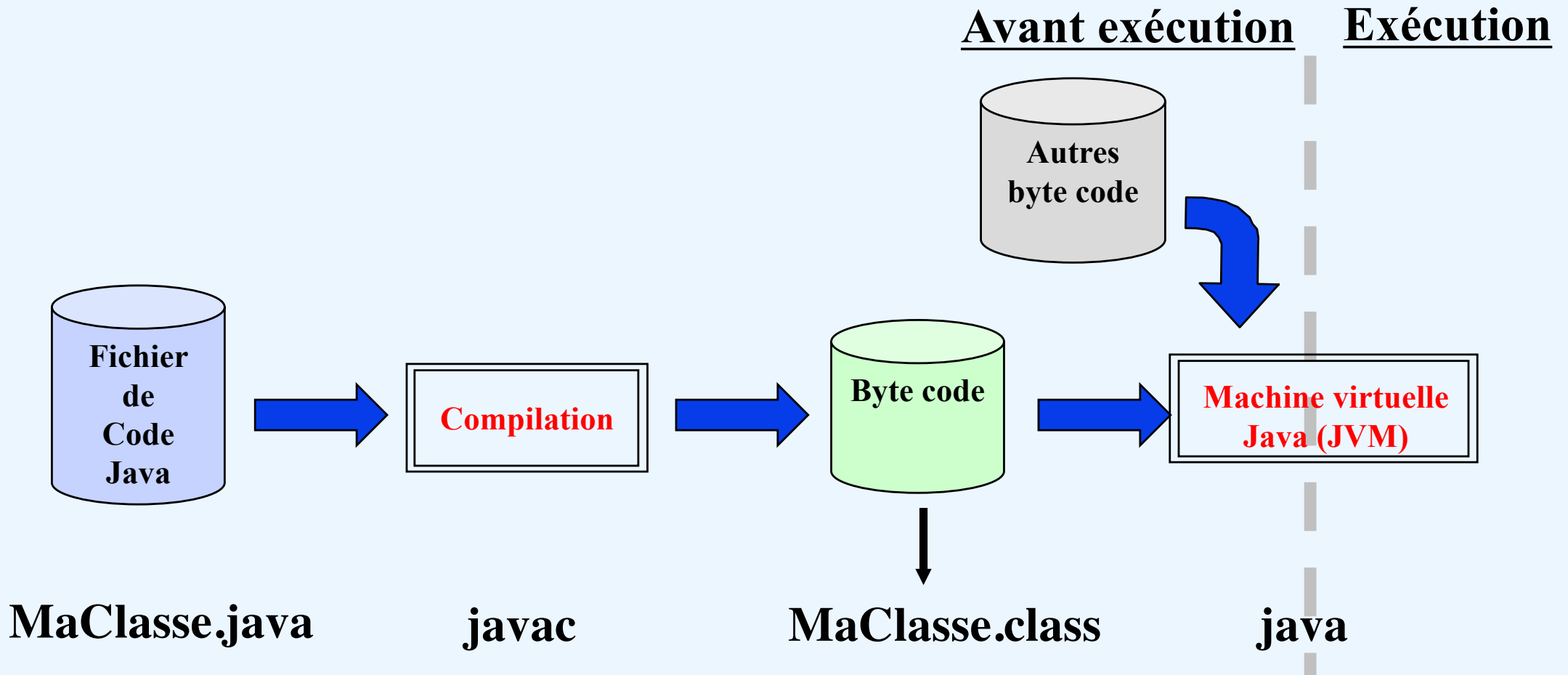
Java est un langage orienté objets

- ❑ Tout est classe (pas de fonctions) sauf les types primitifs (`int`, `float`, `double`, ...) et les tableaux
- ❑ Toutes les classes dérivent de `java.lang.Object`
- ❑ Héritage simple pour les classes
- ❑ Héritage multiple pour les interfaces
- ❑ Les objets se manipulent via des références
- ❑ Une API objet standard est fournie
- ❑ La syntaxe est proche de celle de C

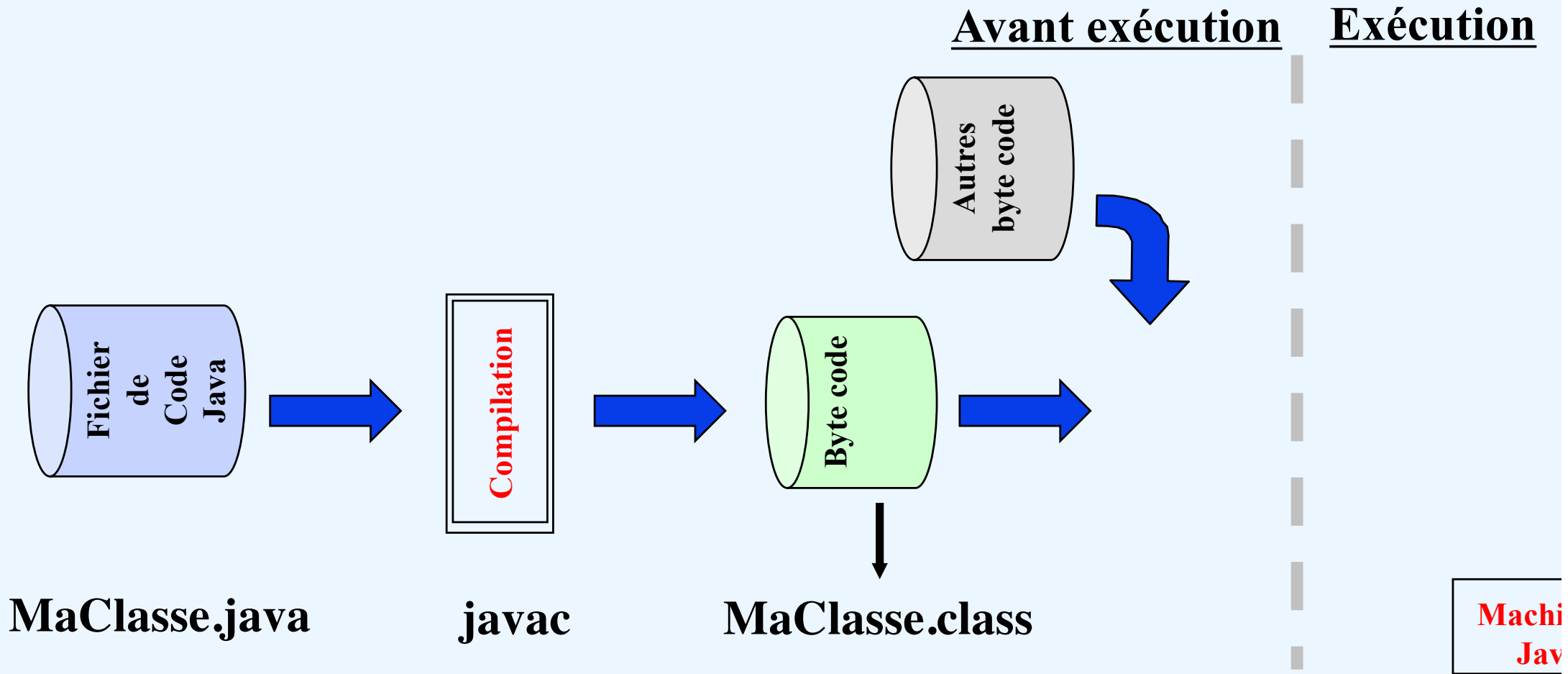
Langage compilé



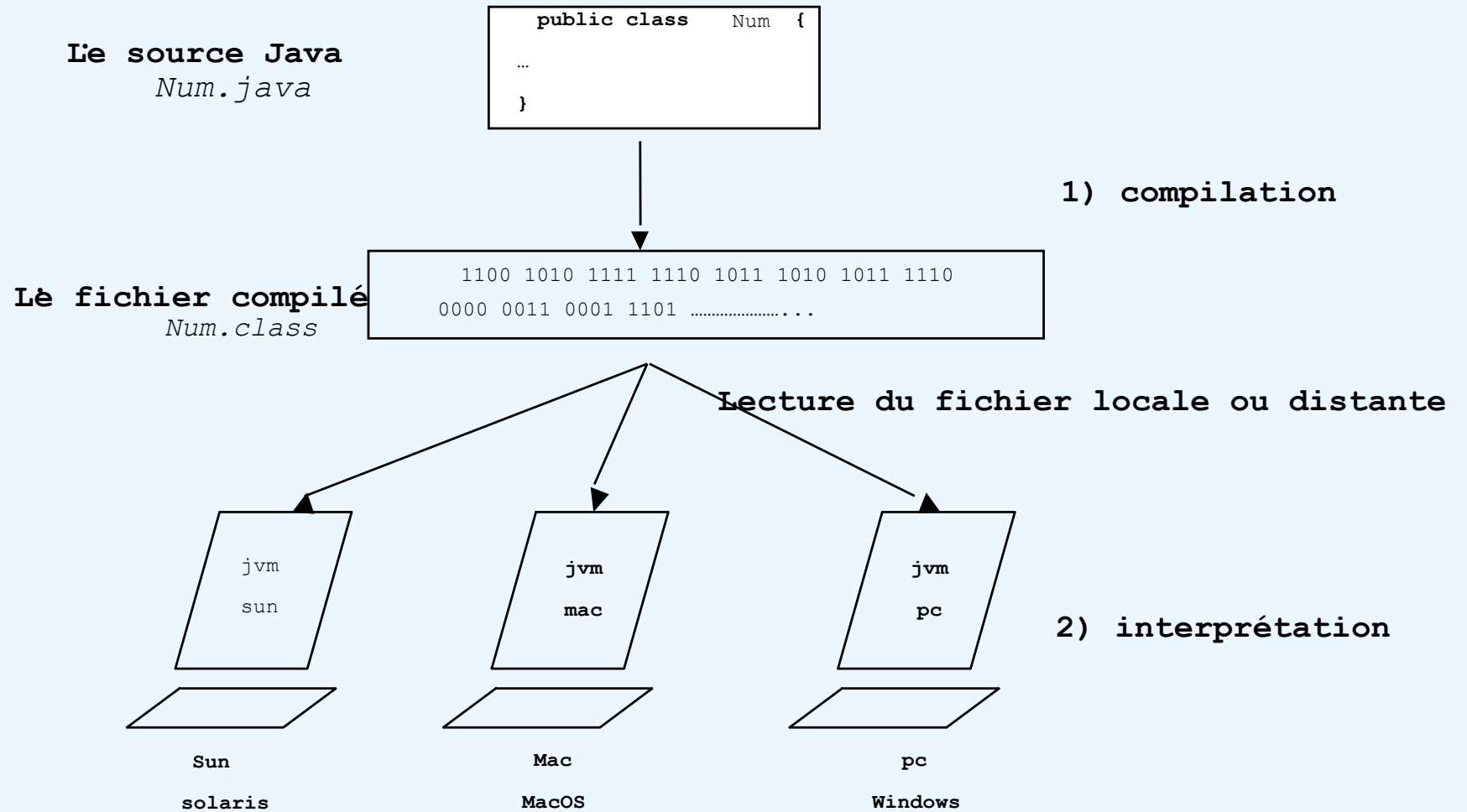
Langage interprété (Java)



Langage interprété (Java)



Java est portable





Java est portable

- ❑ Le compilateur Java génère du *byte code*.
- ❑ La *Java Virtual Machine* (JVM) est présente sur Unix, Win32, Mac, OS/2, Netscape, IE, ...
- ❑ Le langage a une sémantique très précise.
- ❑ La taille des types primitifs est indépendante de la plate-forme.
- ❑ Java supporte un code source écrit en Unicode.
- ❑ Java est accompagné d'une librairie standard.



Java est robuste

- ❑ A l'origine, c'est un langage pour les applications embarquées.
- ❑ Gestion de la mémoire par un *garbage collector*.
- ❑ Pas d'accès direct à la mémoire.
- ❑ Mécanisme d'exception.
- ❑ Accès à une référence *null* → exception.
- ❑ compilateur contraignant (erreur si exception non gérée, si utilisation d'une variable non affectée, ...).
- ❑ Tableaux = objets (taille connue, débordement → exception).
- ❑ Seules les conversions sûres sont automatiques.
- ❑ Contrôle des *cast* à l'exécution.

Java est robuste

❑ Deux types : primitif ou Object (et ses dérivés)

❑ primitif :

➤ int x = 1;

➤ int y = 2;

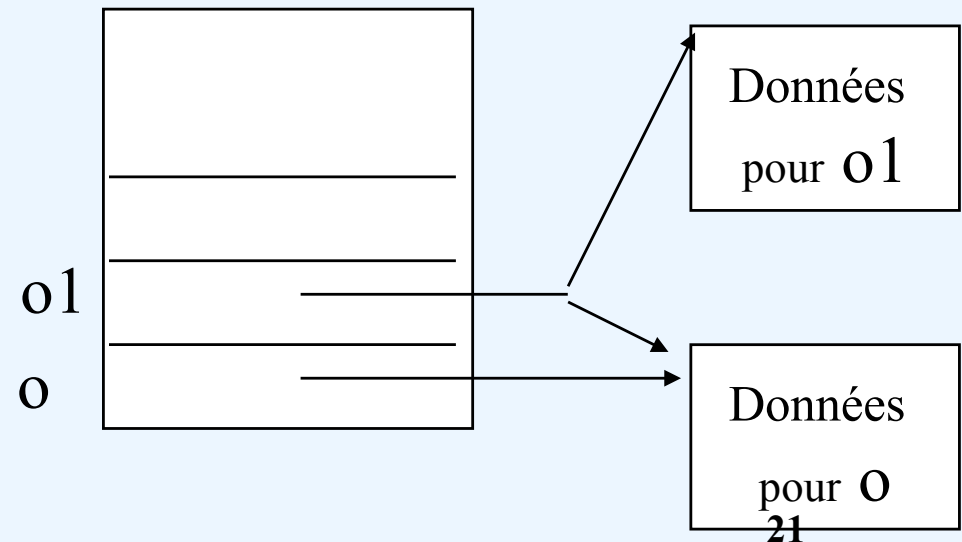
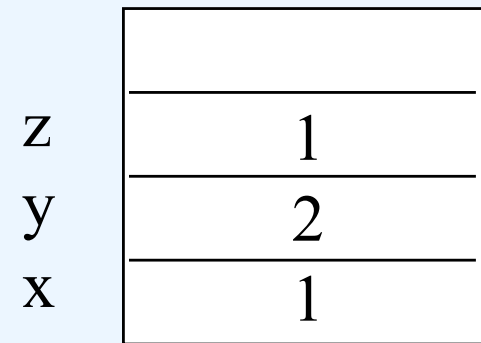
➤ int z = x;

❑ Object

➤ Object o = new Object();

➤ Object o1 = new Object();

➤ o1 = o;





Java est sécurisé

- ❑ Indispensable avec le code mobile.
- ❑ Pris en charge dans l'interpréteur.
- ❑ Trois couches de sécurité :
 - ✧ *Verifier* : vérifie le *byte code*.
 - ✧ *Class Loader* : responsable du chargement des classes.
 - ✧ *Security Manager* : accès aux ressources.
- ❑ Code certifié par une clé.



Java est multi-thread

- ❑ Intégrés au langage et aux API :
 - `synchronized`
 - *garbage collector* dans un thread de basse priorité
 - `java.lang.Thread`, `java.lang.Runnable`
- ❑ Accès concurrents à objet gérés par un *monitor*.
- ❑ Implémentation propre à chaque JVM.
- ❑ Difficultés pour la mise au point et le portage.



Java est distribué

- ❑ API réseau (java.net.Socket, java.net.URL, ...).
- ❑ Chargement / génération de code dynamique.
- ❑ Applet.
- ❑ Servlet.
- ❑ *Protocole / Content handler.*
- ❑ *Remote Method Invocation.*
- ❑ JavaIDL (CORBA).



Les performances

- ❑ Actuellement le *byte code* est interprété.
- ❑ Plusieurs types de génération de code machine :
 - Conversion statique en C (j2c, Tabo, ...)
 - Conversion statique en code natif.
 - Compilation en code machine à la volée (JIT).

Les différences avec C++

- ❑ Pas de structures ni d' unions
- ❑ Pas de types énumérés
- ❑ Pas de *typedef*
- ❑ Pas de préprocesseur
- ❑ Pas de variables ni de fonctions en dehors des classes
- ❑ Pas de fonctions à nombre variable d' arguments
- ❑ Pas d'héritage multiple de classes
- ❑ Pas de types paramétriques (*template*)
- ❑ Pas de surcharge d' opérateurs
- ❑ Pas de passage par copie pour les objets
- ❑ Pas de pointeurs, seulement des références



Les outils (1)

□ Environnements de développement :

- Sun JDK 1.1.x (compilateur, interpréteur, *appletviewer*,...)
- java-mode emacs
- IDE : Visual Age, Café, CodeWarrior, Java WorkShop, Jbuilder, Visual J++, ...

□ Browsers :

- Sun HotJava 1.1
- Netscape Navigator 4
- Internet Explorer 4



Les outils (2)

- ❑ JVM : Kaffe, Cacao, Harissa, ...
- ❑ Convertisseur : c2j, j2c, Tabo...
- ❑ Décompilateur/ 'obscurcisseur' : Mocha / Crema
- ❑ Générateur de parseurs : JavaCC, JavaCUP
- ❑ Profiler : Hyperprofiler, ProfileViewer



Les outils (3)

Java Development Kit

- ❑ **javac** : compilateur de sources java
- ❑ **java** : interpréteur de *byte code*
- ❑ **appletviewer** : interpréteur d'applet
- ❑ **javadoc** : générateur de documentation (HTML, MIF)
- ❑ **javah** : générateur de *header* pour l'appel de méthodes natives
- ❑ **javap** : désassembleur de *byte code*
- ❑ **jdb** : debugger
- ❑ **javakey** : générateur de clés pour la signature de code
- ❑ **rmic** : compilateur de *stubs* RMI
- ❑ **rmiregistry** : "*Object Request Broker*" RMI



Les *core* API 1.0.2 et 1.1

- ❑ **java.lang** : Types de bases, Threads, ClassLoader, Exception, Math, ...
- ❑ **java.util** : Hashtable, Vector, Stack, Date, ...
- ❑ **java.applet**
- ❑ **java.awt** : Interface graphique portable
- ❑ **java.io** : accès aux i/o par flux, wrapping, filtrage
- ❑ **java.net** : Socket (UDP, TCP, multicast), URL, ...



Les core API 1.1

- ❑ **java.lang.reflect** : introspection sur les classes et les objets
- ❑ **java.beans** : composants logiciels
- ❑ **java.sql (JDBC)** : accès homogène aux bases de données
- ❑ **java.security** : signature, cryptographie, authentification
- ❑ **java.serialisation** : sérialisation d'objets
- ❑ **java.rmi** : *Remote Method Invocation*
- ❑ **java.idl** : interopérabilité avec CORBA



Les autres API

- ❑ **Java Server** : jeeves / servlets
- ❑ **Java Commerce*** : JavaWallet
- ❑ **Java Management (JMAPI)** : gestion réseau
- ❑ **Java Média** : 2D*, 3D, Média Framework, Share, Animation*, Telephony



Structure du langage



Java : introduction

- Exemple de programme
- Types primitifs
- Variables de type primitif
- Types structurés
- Opérateurs
- Instructions

Exemple

```
public class Application
```

```
{
```

```
    public static void main( String args[])
```

```
    {
```

```
        int i = 5;
```

```
        i = i * 2;
```

```
        System.out.print(" i est égal à ");
```

```
        System.out.println( i);
```

```
    }
```

```
}
```

Nom de la classe -->
fichier Application.java

Point d'entrée unique
la procédure "main"
args : les paramètres de
la ligne de commande

Instructions

affichage

variable locale à "main"



Identificateurs (1)

- ❑ Les identificateurs commencent par une lettre, _ ou \$
- ❑ Conventions sur les identificateurs :
 - ✧ Si plusieurs mots sont accolés, mettre une majuscule à chacun des mots sauf le premier.
 - *uneVariableEntiere*
 - ✧ La première lettre est majuscule pour les classes et les interfaces
 - *MaClasse, UneJolieFenetre*
 - ✧ La première lettre est minuscule pour les méthodes, les attributs et les variables
 - *setLongueur, i, uneFenetre*
 - ✧ Les constantes sont entièrement en majuscules
 - *LONGUEUR_MAX*



Les mots réservés de Java

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



Types primitifs

- entier
 - signés seulement
 - **type** byte (**8 bits**), short (**16 bits**), int (**32 bits**), long (**64 bits**)
- flottant
 - standard IEEE
 - **type** float(**32 bits**), double (**64bits**)
- booléen
 - **type** boolean (**true,false**)
- caractère
 - unicode, <http://www.unicode.org>
 - **type** char (**16 bits**)



Variables de type primitif

- *Syntaxe :*
 - *type nom_de_la_variable;*
 - *type nom1,nom2,nom3;*
 - *type nom_de_la_variable = valeur;*
- *exemples :*
 - `int i;`
 - `int j = 0x55AA0000;`
 - `boolean succes = true;`
- Les variables peuvent être déclarées n'importe où dans un bloc.

Passage de paramètres des types primitifs

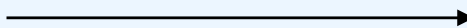
- ✧ l'adresse d'une variable ne peut être déterminée
- ✧ le passage de paramètre est par valeur uniquement

```
static int plusGrand(int x, int y, int z){...}
```

```
void MéthodeAppelante{int a=2, b=3, c=4; plusGrand(a,b,4);}
```

c	4
b	3
a	2

Appel de plusGrand(a,b,4);



z	4
y	3
x	2
c	4
b	3
a	2



Conversion de type

- Les affectations non implicites doivent être *castées* (sinon erreur à la compilation).
- Automatique
 - si la taille du type destinataire est supérieure
 - `byte a,b,c; int d = a+b/c;`
- Explicite

```
int i = 258;
long l = i;    // ok
byte b = i;    // error: Explicit cast needed to convert int to byte
byte b = 258; // error: Explicit cast needed to convert int to byte
byte b = (byte)i; // ok mais b = 2
```



Conversions de type

- *par défaut la constante numérique est de type int,*
- *suffixe L pour obtenir une constante de type long* 40L
- *par défaut la constante flottante est de type double,*
- *suffixe F pour obtenir une constante de type float* 40.0F

- **Implicite**

- *si la taille du type destinataire est supérieure*
 - byte ⇒ short,int,long,float,double
 - short ⇒ int, long, float, double
 - char ⇒ int, long, float, double
 - int ⇒ long, float, double
 - long ⇒ float, double
 - float ⇒ double



La classe Conversions : Conversions.java

```
public class Conversions
{
    public static void main( String args[])
    {
        byte b;short s;char c;int i;long l;float f;double d;
        b=(byte) 0; s = b; i = b; l = b; f = b; d = b;
        i = s; l = s; f = s; d = s;
        i = c; l = c; f = c; d = c;
        l = i; f = i; d = i;
        f = l; d = l;
        d = f;
    }
}
```



Type caractère

- Java utilise le codage Unicode
- représenté par 16 bits
- `\u0020` à `\u007E` code ASCII, Latin-1
 - `\u00AE` ©
 - `\u00BD` / la barre de fraction ...
- `\u0000` à `\u1FFF` zone alphabets
 -
 - `\u0370` à `\u03FF` alphabet grec
 -

<http://www.unicode.org>



Opérateurs

- Arithmétiques

`+, -, *, /, %, ++ +=, -=, /=, %=, --,`

Syntaxe C

- Binaires

`~, &, |, ^, &=, |=, ^=, >>, >>>, <<, >>=, >>>=, <<=`

`>>>` rotation à droite avec remplissage de zéros

`>>` rotation à droite avec conservation du signe

`<<` Bits à gauche perdus, zéros insérés à droite

- Relationnels

`==, !=, >, <, >=, <=`

Syntaxe C

- Booléens

`&, |, ^, &=, |=, ^=, ==, !=, !, ?:, ||, &&`

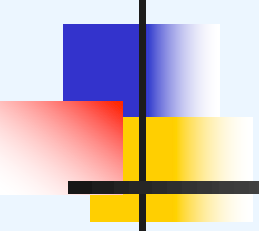


Opérateurs booléens et court-circuits, exemple

```
1 public class Div0
2 {
3     public static void main( String args[])
4     {
5         int den = 0, num = 1;
6         boolean b;
7         System.out.println("den == " + den);
8         b = (den != 0 && num / den > 10);
9         b = (den != 0 & num / den > 10);
10    }
11 }
```

*Exception in thread "main" java.lang.ArithmeticException :
by zero at Div0.main(Div0.java:9)*

Précédence des opérateurs



	()	[]	.		
+	++	--	~	!	
	*	/	%		
	+	-			
	>>	>>>	<<		
	>	>=	<	<=	
	==	!=			
	&				
	^				
	&&				
	?:				
-	=	<i>op=</i>			



Type structuré : tableau

- Déclarations de tableaux
 - `int[] mois; // mois est affecté à null`
 - ou `int[] mois = new int[12];`
 - ou `int[] mois={31,28,31,30,31,30,31,31,30,31,30,31};`
- Déclarations de tableaux à plusieurs dimensions
 - `double [][] m= new double [4][4];`



Type structuré : tableau

- Accès aux éléments
 - le premier élément est indexé en 0
 - vérification à l'exécution des bornes, levée d'exception

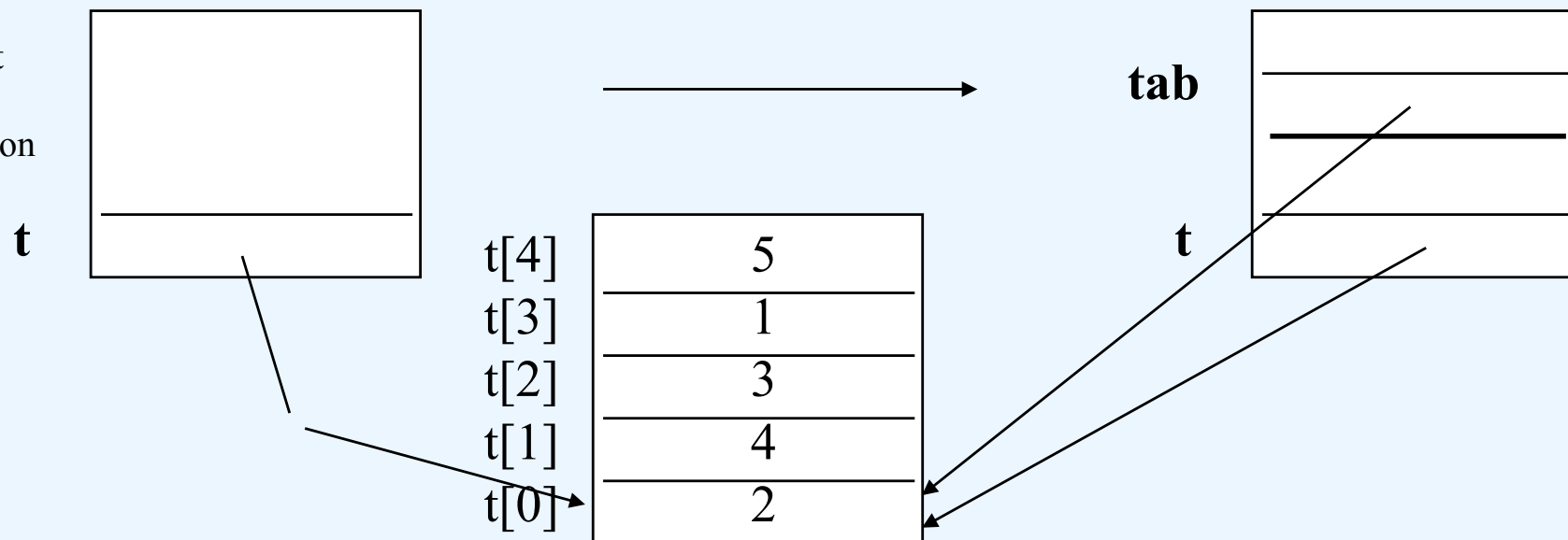
```
mois[0] = 31;  
int l = mois.length; // l = 12  
int e = mois[50]; // Lève une ArrayIndexOutOfBoundsException
```
- En paramètre
 - la variable de type tableau est une référence,
 - le passage par valeur de Java ne peut que transmettre la référence sur le tableau
- 2 syntaxes autorisées : `int mois[]` ou `int[] mois;`

Passage de paramètres par valeur uniquement

```
static void trier(int[] tab) {  
    tab[0] = 8; // --> t[0] == 8;
```

```
int[] t = {2, 4, 3, 1, 5};  
trier(t);
```

Un extrait
de la pile
d'exécution





Instructions de contrôle

- **branchement**
 - if else, break, switch, return // syntaxe C
- **Itération**
 - while, do-while, for, continue // syntaxe C
- **Exceptions**
 - try catch finally, throw



Instruction de branchement, *if else*

- *if (expression-booleenne) instructions1; [else instructions2]*

```
public class IfElse
```

```
{
```

```
public static void main( String[] args)
```

```
{
```

```
int mois = 4;
```

```
String saison;
```

```
if ( mois == 12 || mois == 1 || mois == 2)           {saison = "hiver"; }
```

```
else if ( mois == 3 || mois == 4 || mois == 5)      {saison = "printemps";} 
```

```
else if ( mois == 6 || mois == 7 || mois == 8)     {saison = "ete";   }
```

```
else if ( mois == 9 || mois == 10 || mois == 11)   {saison = "automne";} 
```

```
else { saison = "";} 
```

```
System.out.println("Avril est au " + saison + ".");
```

```
}
```

```
}
```



switch case

- *switch (expression) {*
- *case value1 :*
- *break;*
- *case value2 :*
-
- *case value3 :*
- *break;*
- *case valueN :*
- *break;*
- *default :*
- *}*



swieth case, exemple

```
public class SwitchSaison
{
    public static void main( String[] args)
    {
        int  mois = 4;   String saison;
        switch (mois)
        {
            case 12: case 1: case 2:  saison = "hiver";      break;
            case 3: case 4: case 5:   saison = "printemps";break;
            case 6: case 7: case 8:   saison = "ete";       break;
            case 9: case 10: case 11: saison = "automne";  break;
            default: saison = "";
        }
        System.out.println("Avril est au " + saison + ".");
    }
}
```



Itérations

- *while (expression) {*
- *instructions*
- *}*

- *for (initialisation; terminaison; itération) instructions;*
- *<===>*
- *initialisation;*
- *while (terminaison){*
- *instructions;*
- *itération;*
- *}*



Itération, *for(;;)*, exemple

```
public class Mois
{
    public static void main( String[] args)
    {
        String[] mois={"janvier","fevrier","mars","avril","mai","juin",
            "juillet","aout","septembre","octobre","novembre","decembre"};
        int[] jours={31,28,31,30,31,30,31,32,30,31,30,31};
        String printemps = "printemps";   String ete = "ete";
        String automne = "automne";       String hiver = "hiver";
        String[] saisons={ hiver,hiver,printemps,printemps,printemps,ete,ete,ete,
            automne,automne,automne,hiver};
        for(int m = 0; m < 12; m++)
        {
            System.out.println(mois[m] + " est au/en " +saisons[m] + " avec " + jours[m] + " jours.");
        }
    }
}
```




break

- Provoque la fin du bloc en cours

```
for ( ..... )  
  {  
    .....  
    if (...) break;  
    .....  
  }
```



continue

continue; *poursuite immédiate de l'itération*

```
for ( .....){  
.....  
if (...) continue;  
.....  
}
```

•Plus les blocs labelisés

```
UN: while(...) {  
DEUX: for(...) {  
TROIS: while(...) {  
    if (...) continue UN;    // Reprend sur la première boucle while  
    if (...) break DEUX;    // Quitte la boucle for  
    if (...) continue;      // Reprend sur la deuxième boucle while  
    .....  
}  
.....  
}  
.....  
}
```



Les unités de compilation

- Le code source d'une classe est appelé *unité de compilation*.
- Il est recommandé (mais pas imposé) de ne mettre qu'une classe par unité de compilation.
- L'unité de compilation (le fichier) doit avoir le même nom que la classe qu'elle contient.

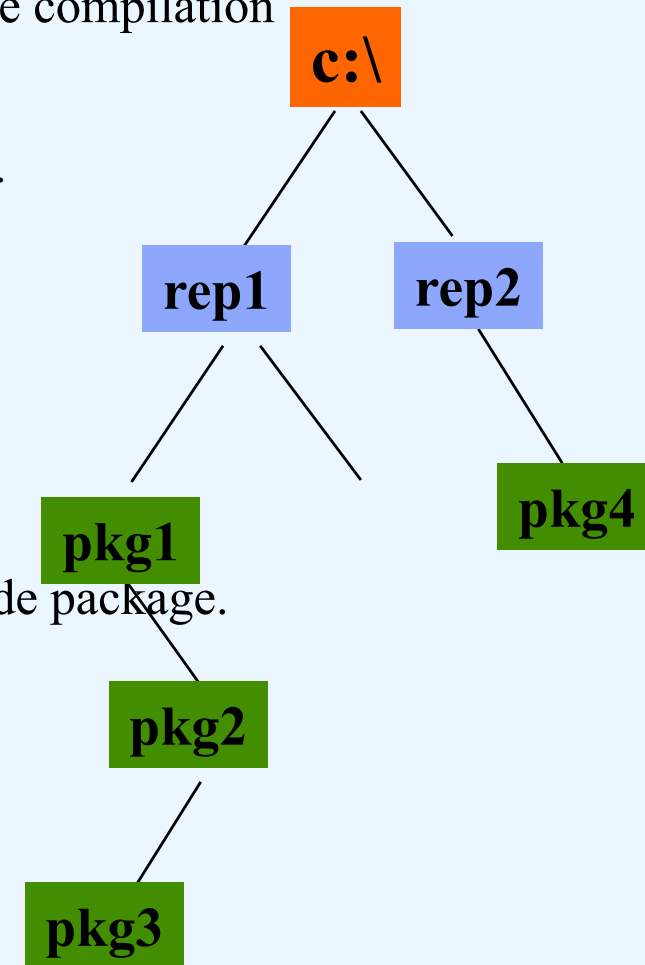


Les packages (1)

- **Fonction**
 - Unité logique par famille de classes
 - découpage hiérarchique des paquetages
 - Les API sont organisées en package (`java.lang`, `java.io`, ...)
 - Les packages permettent au compilateur et à la JVM de localiser les fichiers contenant les classes à charger.
- **But**
 - regrouper un ensemble de classes sous un même espace de 'nomage'.
 - restriction visibilité
- Une classe `Watch` appartenant au package `time.clock` doit se trouver dans le fichier `time/clock/Watch.class`

Les packages (2)

- Instructions
 - indique à quel package appartient la ou les classe(s) de l'unité de compilation (le fichier).
- Les noms des packages suivent le schéma : `name.subname ...`
 - `package pkg1[.pkg2[.pkg3];`
 - les noms sont en minuscules
 - `c'` est la première instruction du source java
- ils doivent être importés explicitement sauf `java.lang`.
 - L' instruction `import packageName`
 - permet d'utiliser des classes sans les préfixer par leur nom de package.
 - `import pkg1[.pkg2[.pkg3].(nomdeclasse|*);`
 - liés aux options de la commande de compilation
 - `dos> javac -classpath .;c:\rep1;c:\rep2`
- Les répertoires contenant les packages doivent être présents dans la variable d'environnement `CLASSPATH`





Les packages (3)

```
CLASSPATH = $JAVA_HOME/lib/classes.zip;$HOME/classes
```

```
~exemple/classes/graph/2D/Cercle.java
```

```
package graph.2D;  
public class Cercle()  
{ ... }
```

```
~exemple/classes/graph/3D/Sphere.java
```

```
package graph.3D;  
public class Sphere()  
{ ... }
```

```
~exemple/classes/paintShop/MainClass.java
```

```
package paintShop;  
  
import graph.2D.*;  
  
public class MainClass()  
{  
    public static void main(String[] args) {  
        graph.2D.Cercle c1 = new graph.2D.Cercle(50)  
        Cercle c2 = new Cercle(70);  
        graph.3D.Sphere s1 = new graph.3D.Sphere(100);  
        Sphere s2 = new Sphere(40); // error: class paintShop.Sphere not found  
    }  
}
```



Paquetages prédéfinis

- le paquetage `java.lang.*` est importé implicitement
 - ce sont les interfaces : *Cloneable*, *Comparable*, *Runnable*
 - et les classes : `Boolean`, `Byte`, `Character`, `Class`, `ClassLoader`, `Compiler`,
 - `Double`, `Float`, `InheritableThreadLocal`, `Long`, `Math`, `Number`,
 - `Object`, `Package`, `Process`, `Runtime`, `RuntimePermission`,
 - `SecurityManager`, `Short`, `StrictMath`, `String`, `StringBuffer`,
 - `System`, `Thread`, `ThreadGroup`, `ThreadLocal`,
 - `Throwable`, `Void`,
 - toutes les classes dérivées de `Exception`, `ArithmeticException`,....
 - et celles de `Error`, `AbstractMethodError`,....
- `java.awt.*` `java.io.*` `java.util.*`



Classes et objets

-



Plan

- Classe syntaxe
- Création d'instances
 - Ordre d'initialisation des champs
- Constructeur
- Surcharge
- Encapsulation
 - Règles de visibilité
 - Paquetage
- Classes imbriquées
- Interfaces
- Classes incomplètes/abstraites
- Classe et exception



Classe : Syntaxe

```
class NomDeClasse {  
    type variable1DInstance;  
    type variable2DInstance;  
    type variableNDInstance;  
  
    type nom1DeMethode ( listeDeParametres) {  
    }  
    type nom2DeMethode( listeDeParametres) {  
    }  
    type nomNDeMethode( listeDeParametres) {  
    }  
}
```



Classe : Syntaxe et visibilité

```
visibilité class NomDeClasse {  
    visibilité type variable1DInstance;  
    visibilité type variable2DInstance;  
    visibilité type variableNDInstance;  
  
    visibilité type nom1DeMethode ( listeDeParametres) {  
    }  
    visibilité type nom2DeMethode( listeDeParametres) {  
    }  
    visibilité type nomNDeMethode( listeDeParametres) {  
    }  
}
```

visibilité ::= public | private | protected | < vide >



Exemple: la classe des polygones réguliers

```
public class PolygoneRegulier{
    private int nombreDeCotes;
    private int longueurDuCote;

    void initialiser( int nCotes, int longueur){
        nombreDeCotes = nCotes;
        longueurDuCote = longueur;
    }
    int perimetre() { return nombreDeCotes * longueurDuCote; }
    int surface(){
        return (int) (1.0/4 * (nombreDeCotes * Math.pow(longueurDuCote,2.0) *
            cotg(Math.PI / nombreDeCotes)));
    }
    private static double cotg(double x){return Math.cos(x) / Math.sin(x); }
}
```



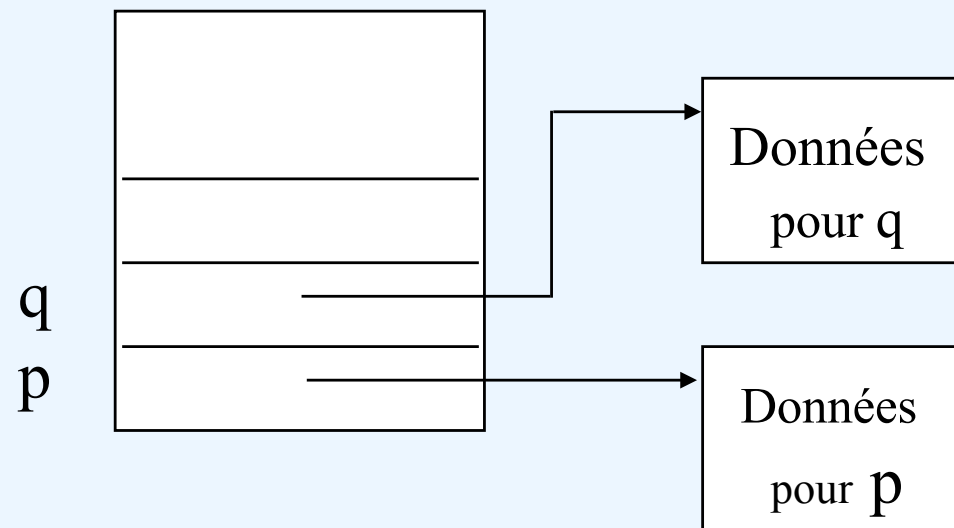
Création et accès aux instances

- Declaration d'instances et création
 - PolygoneRegulier p; *// ici p == null*
 - p = new PolygoneRegulier (); *// p est une référence sur un objet*
 - PolygoneRegulier q = new PolygoneRegulier (); *// en une ligne*
- La référence d'un objet est l' adresse
 - d'une structure de données contenant des informations sur la classe déclarée à laquelle se trouvent les champs d'instance et d'autres informations
 - (cette référence ne peut être manipulée)
- Opérateur "."
 - appels de méthodes (si accessibles)
 - accès aux champs (si accessibles)
- en passage de paramètre
 - par valeur, soit uniquement la référence de l'objet

Instances et mémoire

`p = new PolygoneRegulier ();` // *p est une référence sur un objet*

`PolygoneRegulier q = new PolygoneRegulier ();` // *en une ligne*



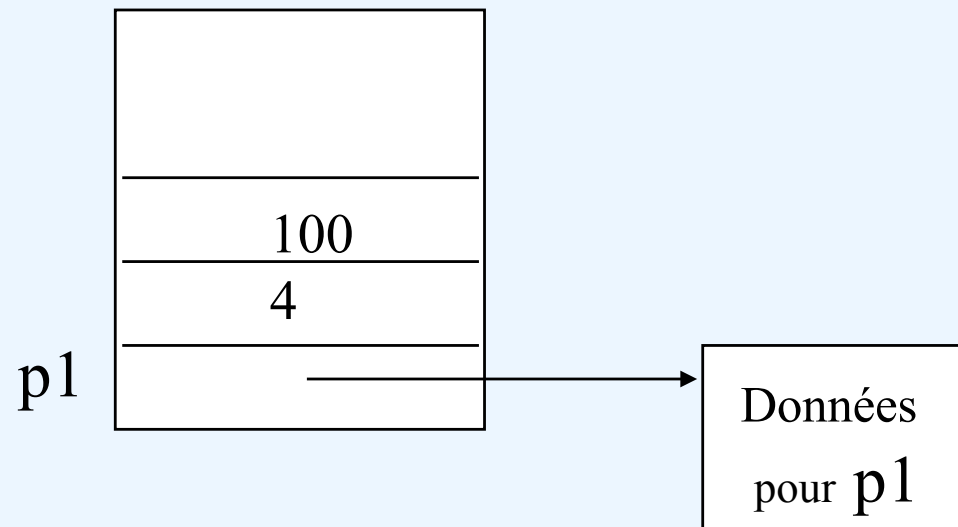


Utilisation de la classe

- **public class TestPolyReg{**
- **public static void main(String args[]){**
- **PolygoneRegulier p1 = new PolygoneRegulier();**
- **PolygoneRegulier p2 = new PolygoneRegulier();**
- **p1.initialiser(4,100);**
- **System.out.println(" surface de p1 = " + p1.surface());**
- **p2.initialiser(5,10);**
- **System.out.println(" perimetre de p2 = " + p2.perimetre());**
- **}**

Instances et appels de méthodes

p1.initialiser(4,100);



Pile d'exécution avant l'appel de la méthode initialiser



Initialisation des variables d'instance

- champs d'instance initialisés
 - à 0 si de type entier, 0.0 si flottant, 0 si char, false si booléen
 - à null si référence d'objet
- Variables locales aux méthodes
 - elles ne sont pas initialisées, erreur à la compilation si utilisation avant affectation



Constructeur

- **public class** PolygoneRegulier {
- **private int** nombreDeCotes;
- **private int** longueurDuCote;

- PolygoneRegulier (int nCotes, int longueur) { // void initialiser
- **nombreDeCotes = nCotes;**
- **longueurDuCote = longueur;**
- **}**
-
- le constructeur a le même nom que la classe
 - PolygoneRegulier P = new PolygoneRegulier(4, 100);
 - A la création d'instance(s) les 2 paramètres sont maintenant imposés : le seul constructeur présent impose son appel



Constructeur par défaut

- **public class PolygoneRegulier{**
- **private int nombreDeCotes;**
- **private int longueurDuCote;**

- **void initialiser(int nCotes, int longueur){...}**
- **int perimetre(){...}**
- **int surface(){....}**
- **}**

- **public static void main(String args[]){**
- **PolygoneRegulier p1 = new PolygoneRegulier();**
- **PolygoneRegulier p2 = new PolygoneRegulier();**

appel du constructeur
par défaut





Destructeurs et ramasse miettes

- pas de destructeurs (accessibles à l'utilisateur)
- La dé-allocation n'est pas à la charge du programmeur
- Le déclenchement de la récupération mémoire dépend de la stratégie du ramasse miettes

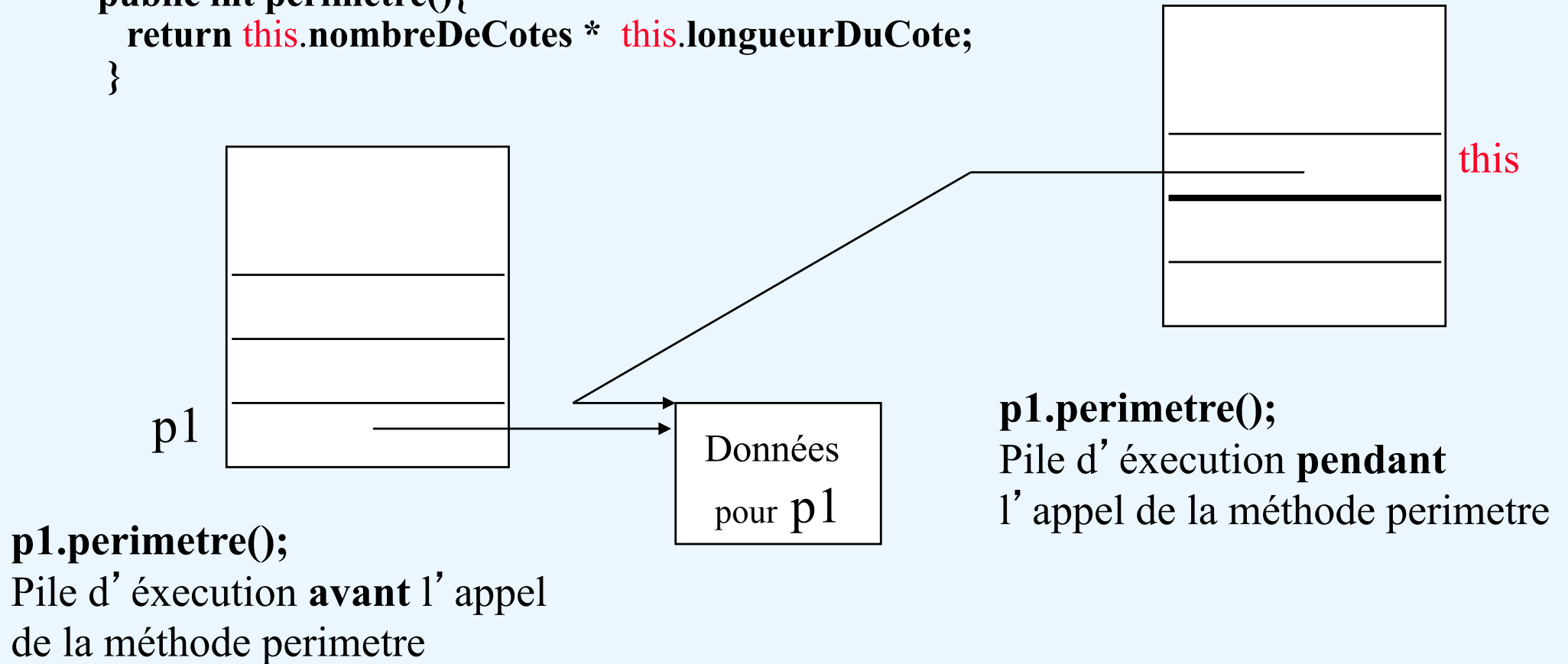


La référence this

```
public class PolygoneRegulier{  
    private int nombreDeCotes;  
    private int longueurDuCote;  
  
    public PolygoneRegulier ( int nombreDeCotes, int longueur){  
        this.nombreDeCotes = nombreDeCotes;  
        this.longueurDuCote = longueur;  
    }  
  
    public int perimetre(){  
        return this.nombreDeCotes * this.longueurDuCote;  
    }  
}
```

this et l'appel d'une méthode

```
public int perimetre(){  
    return this.nombreDeCotes * this.longueurDuCote;  
}
```





Surcharge, polymorphisme ad'hoc

- Polymorphisme ad'hoc
 - Surcharge(overloading),
 - plusieurs implémentations d'une méthode en fonction des types de paramètres souhaités, le choix de la méthode est résolu statiquement dès la compilation
 - Opérateur polymorphe : $3 + 2$; $3.0 + 2.0$, "bon" + "jour"
 - ```
public class java.io.PrintWriter {
```
  - ....
  - ```
public void print(boolean b){...};
```
 - ```
public void print(char c){...};
```
  - ```
public void print(int i){...};
```
 - ```
public void print(long l){...};
```
  - ....
  - ```
}
```



Surcharge, présence de plusieurs constructeurs

- **public class PolygoneRegulier{**
- **private int nombreDeCotes;**
- **private int longueurDuCote;**

- **PolygoneRegulier (int nCotes, int longueur) {**
- **nombreDeCotes = nCotes;**
- **longueurDuCote = longueur;**
- **}**

- **PolygoneRegulier () { nombreDeCotes = 0; longueurDuCote = 0; }**
- **.....**

- **public static void main(String args[]){**
- **PolygoneRegulier p1 = new PolygoneRegulier(4,100);**
- **PolygoneRegulier p2 = new PolygoneRegulier();**



Surcharge (2)

- **public class PolygoneRegulier{**
- **private int nombreDeCotes;**
- **private double longueurDuCote;**

- PolygoneRegulier (int nCotes, int longueur) { **nombreDeCotes = nCotes;**
- **longueurDuCote = longueur;**
- **}**

- PolygoneRegulier (int nCotes, double longueur) { **nombreDeCotes = nCotes;**
- **longueurDuCote = longueur;**
- **}**

- PolygoneRegulier () {... }

- **public static void main(String args[]){**
- **PolygoneRegulier p1 = new PolygoneRegulier(4,100);**
- **PolygoneRegulier p2 = new PolygoneRegulier(5,101.6);**



Un autre usage de this

- `public class PolygoneRegulier{`
- `private int nombreDeCotes;`
- `private int longueurDuCote;`

- `PolygoneRegulier (int nCotes, int longueur) {`
- `nombreDeCotes = nCotes;`
- `longueurDuCote = longueur;`
- `}`

- `PolygoneRegulier () {`
- `this(1, 1);` *// appel du constructeur d'arité 2*
- `}`

- `}`

Affectation

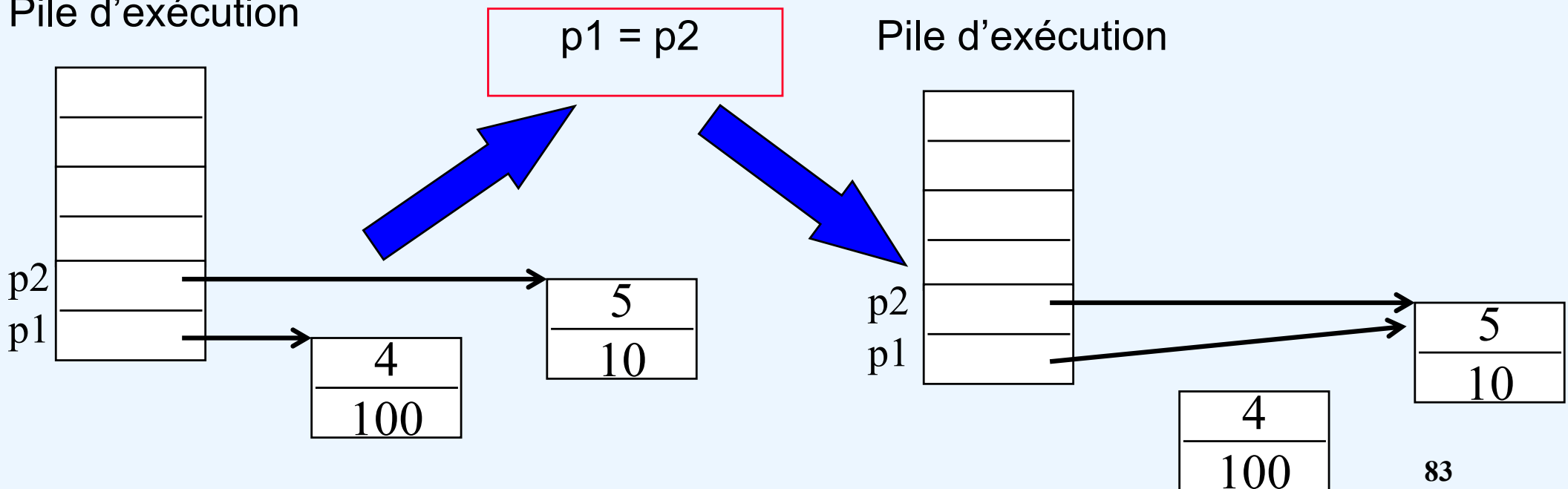
✧ Affectation de références =

✧ `public static void main(String args[]){`

✧ `PolygoneRegulier p1 = new PolygoneRegulier(4,100);`

✧ `PolygoneRegulier p2 = new PolygoneRegulier(5,10);`

Pile d'exécution





Affectation : attention

- Affectation entre deux variables de types primitifs :
 - Affectation des valeurs : `int i = 3; int j = 5; i=j;`
- Affectation entre deux instances de classes :
 - Synonymie : `Polygone p1, p2; ... p1 = p2;`
 - `p1` et `p2` sont deux noms pour le même objet



Objet et Partage

- Un Objet est *mutable* si son état peut changer. Par exemple les tableaux sont mutable.
- Un Objet est *non mutable* si son état ne change jamais. Par exemple, strings sont non mutable.
- Un Objet est *shared* par deux variables s' il peut être accédé par les deux variables.
- Si un objet mutable est shared par deux variables, les modifications effectuées par l' intermédiaire d' une variable sont visible quand l' objet est utilisé depuis l' autre variables



Variables & méthodes static

- Les variables `static` sont communes à toutes les instances de la classe.
- Il n'est pas nécessaire d'instancier une classe pour accéder à un de ses membres statiques.
- Les méthodes statiques ne peuvent pas accéder à `this`.



Variables & méthodes static

```
public class Cercle {
    public static int count = 0;
    public static final double PI = 3.14;
    public double x, y, r;
    public Cercle (double r) {this.r = r; count++;}
    public Cercle bigger(Cercle c)
        {if (c.r > r) return c; else return this;}

    public static Cercle bigger(Cercle c1, Cercle c2)
        {if (c1.r > c2.r) return c1; else return c2;}
}

Cercle c1 = new Cercle (10); Cercle c2 = new Cercle (20);

n = Cercle.count;

Cercle c3 = c1.bigger(c2);

Cercle c4 = Cercle.bigger(c1, c2);
```

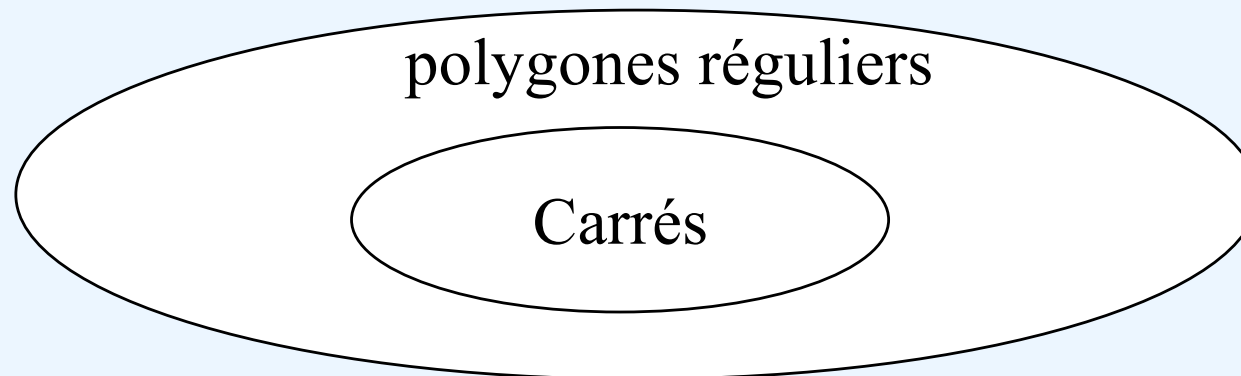


Sommaire

Héritage

Héritage et classification

- définir une nouvelle classe en ajoutant de nouvelles fonctionnalités à une classe existante
 - ajout de nouvelles fonctions
 - ajout de nouvelles données
 - redéfinition de certaines propriétés héritées (masquage)
- Une approche de la classification en langage naturel
- *Les carrés* sont *des polygones réguliers*





Polymorphisme : définitions

- Polymorphisme ad'hoc
 - **Surcharge(overloading),**
 - **plusieurs implémentations d'une méthode en fonction des types de paramètres souhaités, le choix de la méthode est résolu statiquement dès la compilation**
- Polymorphisme d'inclusion (**overriding**),
 - **est fondé sur la relation d'ordre partiel entre les types, relation induite par l'héritage. si le type B est inférieur selon cette relation au type A alors on peut passer un objet de type B à une méthode qui attend un paramètre de type A, le choix de la méthode est résolu dynamiquement en fonction du type de l'objet receveur**
- Polymorphisme paramétrique
 - **ou généricité,**
 - **consiste à définir un modèle de procédure, ensuite incarné ou instancié avec différents types, ce choix est résolu statiquement**



Classe : Syntaxe

```
class NomDeClasse extends NomDeLaSuperClasse {  
    type variableDeClasse1;  
    type variableDeClasse2;  
    type variableDeClasseN;  
  
    type nomDeMethodeDeClasse1( listeDeParametres) {  
  
    }  
    type nomDeMethodeDeClasse2( listeDeParametres) {  
  
    }  
    type nomDeMethodeDeClasseN( listeDeParametres) {  
  
    }  
}
```



Héritage (1)

- Une classe ne peut hériter (`extends`) que d'une seule classe.
- Les classes dérivent, par défaut, de `java.lang.Object`.
- Une référence d'une classe `C` peut contenir des instances de `C` ou des classes dérivées de `C`.
- L'opérateur `instanceOf` permet de déterminer la classe d'une instance.
- Les classes `final` ne peuvent pas être redéfinies dans les sous-classes.



Héritage (2)

Les polygones réguliers sont des "objets java"

- **class PolygoneRegulier extends java.lang.Object{ ...}**

Les carrés sont des polygones réguliers

- **class Carre extends PolygoneRegulier{ ...}**

Les carrés en couleur sont des carrés

- **class CarreEnCouleur extends Carre {.....}**



La classe Object, racine de toute classe Java

- **public class java.lang.Object {**
- **public boolean equals(Object obj) {...}**
- **public final native Class getClass();**
- **public native int hashCode() {...}**
-
- **public String toString() {...}**
- ...
- **protected native Object clone() ...{...}**
- **protected void finalize() ...{...}**
- ...
- **}**



La classe PolygoneRegulier re-visité

```
public class PolygoneRegulier extends Object{  
    private int nombreDeCotes;  
    private int longueurDuCote;  
  
    PolygoneRegulier ( int nCotes, int longueur) {...}  
  
    public boolean equals( Object obj ) {  
        if( !( obj instanceof PolygoneRegulier ) ) return false;  
        PolygoneRegulier poly = ( PolygoneRegulier ) obj;  
        return poly.nombreDeCotes == nombreDeCotes &&  
            poly.longueurDuCote == longueurDuCote; }  
  
    public String toString() {  
        return "<" + nombreDeCotes + "," + longueurDuCote + ">";  
    }  
  
    .....  
}
```



Exemple: utilisation

```
public class TestPolyReg{  
  
    public static void main( String [ ] args ) {  
        PolygoneRegulier p1 = new PolygoneRegulier ( 4, 100);  
        PolygoneRegulier p2 = new PolygoneRegulier ( 5, 100);  
        PolygoneRegulier p3 = new PolygoneRegulier ( 4, 100);  
  
        System.out.println( "poly p1: " + p1.toString() );  
        System.out.println( "poly p2: " + p2 ); // appel implicite de toString()  
        System.out.println( "poly p3: " + p3 );  
        if ( p1.equals( p2))      System.out.println( "p1 == p2");  
        else    System.out.println( "p1 != p2");  
  
        System.out.println( "p1 == p3 ? : " + p1.equals( p3 ) );  
    }  
}
```



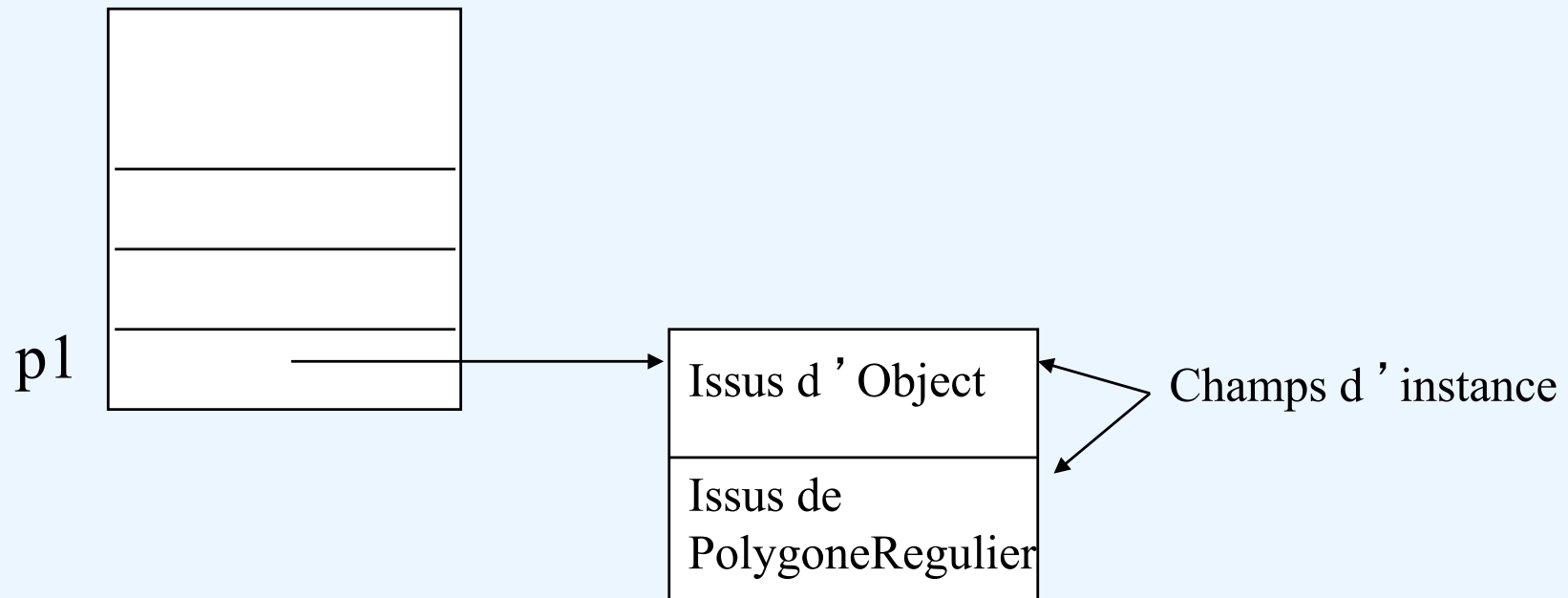

Héritage (3)

- Les instances des classes dérivées effectuent :
 - Le cumul des données d'instance,
 - Le "cumul" du comportement,
- **le comportement des instances issu de la classe dérivée dépend de :**
 - La surcharge des méthodes (la signature est différente) et du masquage des méthodes (la signature est identique)

Champs d'instance

Class PolygoneRegulier extends Object{....}

```
p1 = new polygoneRegulier(4,100);
```





Exemple la classe Carre

```
class Carre extends PolygoneRegulier
{
    // pas de champ d'instance supplémentaire
    Carre( int longueur)
    {
        nombreDeCotes = 4;
        longueurDuCote = longueur;
    }
    // masquage de la méthode PolygoneRegulier.surface()
    int surface() {return longueurDuCote* longueurDuCote;}
    String toString() {return "<4,"+ longueurDuCote +">";}
}
```

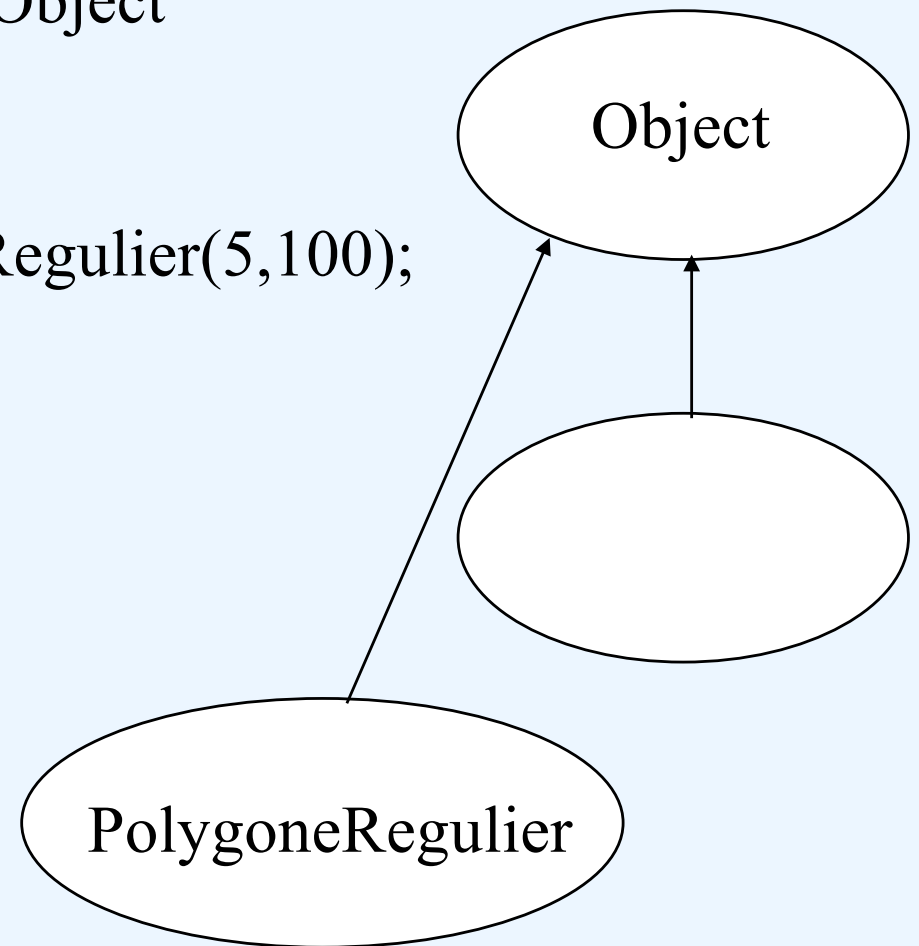


Création d'instances et affectation

- Création d'instances et création
 - Carre c1 = new Carre(100);
 - Carre c2 = new Carre(10);
 - PolygoneRegulier p1 = new PolygoneRegulier(4,100);
- Affectation
 - c1 = c2; *// synonymie, c1 est un autre nom pour c2*
- Affectation polymorphe
 - p1 = c1;
- Affectation et changement de classe
 - c1 = (Carre) p1; *// Hum, Hum ...*

Affectation polymorphe

- Toute instance de classe Java est un "Object"
- `Object obj = new Object();`
- `PolygoneRegulier p = new PolygoneRegulier(5,100);`
- `System.out.print(p.toString());`
- `obj = p;`
- `System.out.print(obj.toString());`





Liaison dynamique

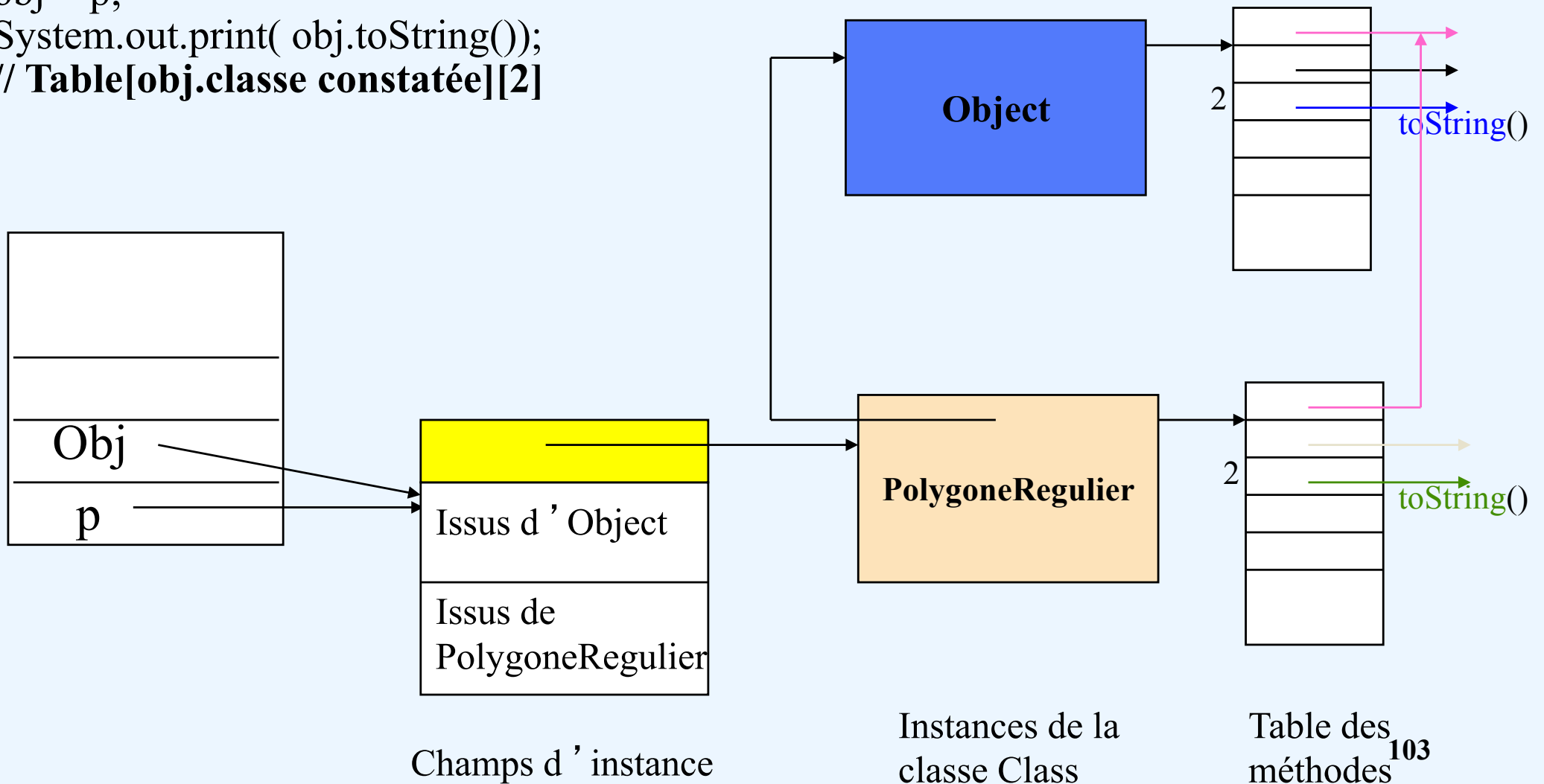
- Sélection de la méthode en fonction de l'objet receveur
- type déclaré / type constaté à l'exécution

// classe déclarée

- PolygoneRegulier p1 = new PolygoneRegulier(5,100);
- Carre c1 = new Carre(100);
int s = p1.surface(); // la méthode surface() de PolygoneRegulier
- p1 = c1; // affectation polymorphe
- s = p1.surface(); // la méthode surface() de Carre est sélectionnée
- la recherche de la méthode s'effectue uniquement dans l'ensemble des méthodes masquées associé à la classe dérivée

Sélection de la bonne méthode(1)

```
obj = p;  
System.out.print( obj.toString());  
// Table[obj.classe constatée][2]
```





Sélection de la bonne méthode(2)

- Chaque instance référencée possède un lien sur sa classe
 - `instance.getClass()`
- Chaque descripteur de classe possède un lien sur la table des méthodes héritées comme masquées
- Le compilateur attribue un numéro à chaque méthode rencontrée
- Une méthode conserve le même numéro tout au long de la hiérarchie de classes
 - `obj.p()` est transcrit par l'appel de la 2^{ème} méthode de la table des méthodes associée à `obj`.
 - Quel que soit le type à l'exécution de `obj` nous appellerons toujours la 2^{ème} méthode de la table
- Comme contrainte importante
 - La signature doit être strictement identique entre les classes d'un même graphe
 - `boolean equals(Object)` doit se trouver dans la classe `PolygoneRegulier` !!
 - (alors que `boolean equals(PolygoneRegulier p)` était plus naturel)



Exercice

- **class A{**
- **void m(A a){ System.out.println(" m de A"); }**
- **void n(A a){System.out.println(" n de A"); }**
- **}**

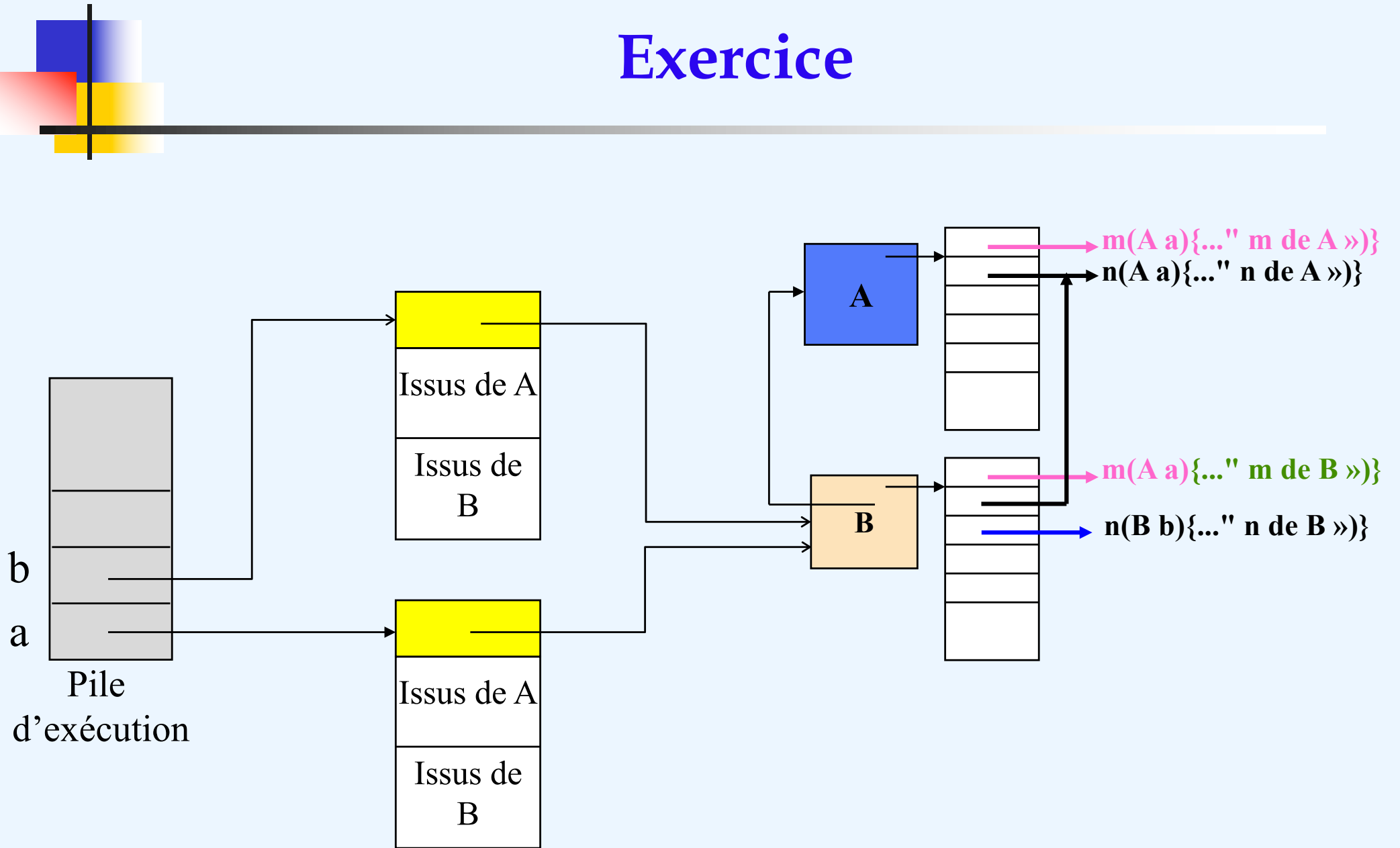
public class B extends A{

- **public static void main(String args[]){**
- **A a = new B();**
- **B b = new B();**
- **a.m(b);**
- **a.n(b);**
- **}**
- **void m(A a){ System.out.println(" m de B"); }**
- **void n(B b){ System.out.println(" n de B"); }**
- **}**

Quelle est la trace d'exécution ?

• m de B
• n de A

Exercice





Le masquage des variables

- Une classe peut définir des variables portant le même nom que celles de ses classes ancêtres.
- Une classe peut accéder aux attributs redéfinis de sa classe mère en utilisant `super` ou par *cast*.
- Une classe peut accéder aux méthodes redéfinies de sa classe mère en utilisant `super`.



super

- Appel d'une méthode de la super classe
- Appel du constructeur de la super classe

```
class Carre extends PolygoneRegulier
{
    Carre( int longueur) {super(4,longueur);}
    int surface() {return super.surface();}
    .....
}
```



super

```
class A {int x; void m() {...}}
class B extends A{int x;void m() {...}}
class C extends B {
    int x, a;
    void m() {...}
    void test() {
        a = super.x;
        a = super.super.x;
        a = ((B)this).x;
        a = ((A)this).x;
        super.m();
        super.super.m();
        ((B)this).m();
    }
}
```

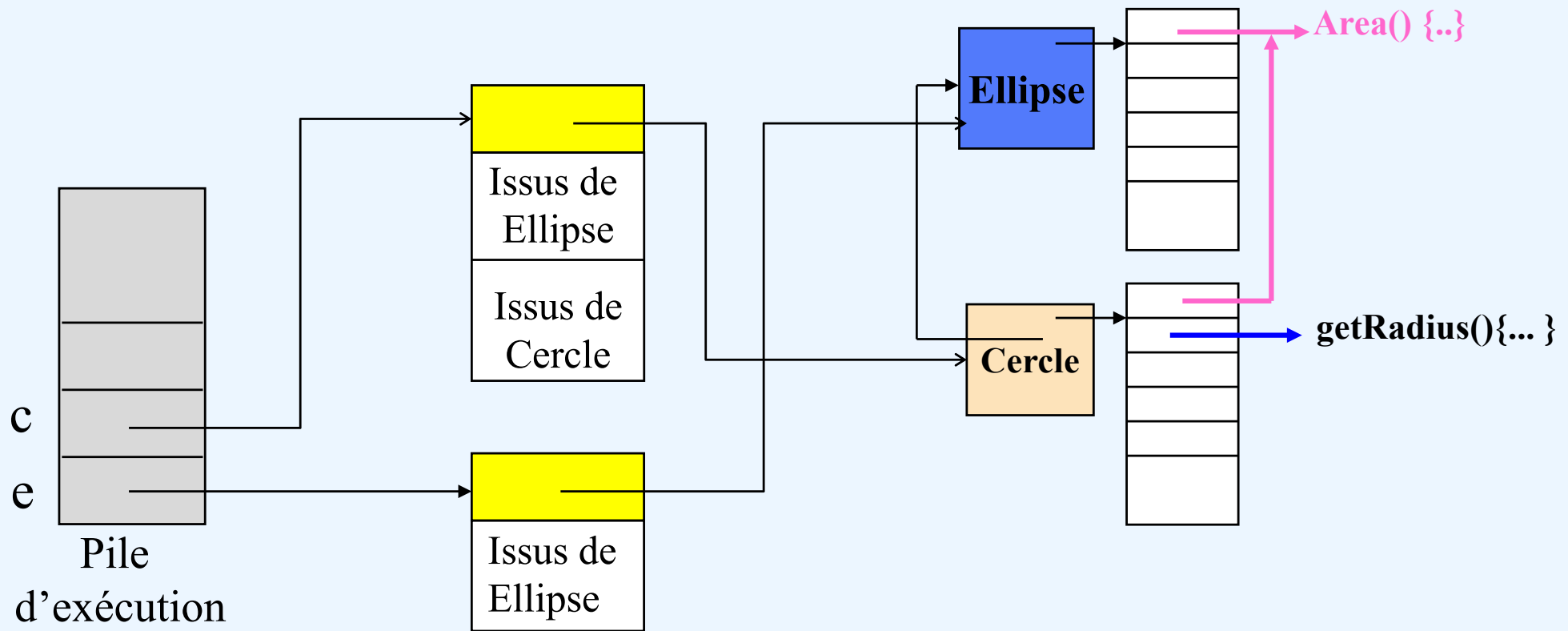
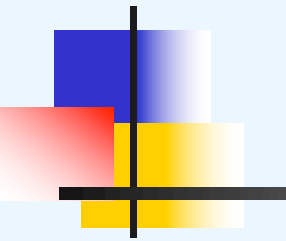


instanceOf (1)

```
public class Ellipse
{
    public double r1, r2;
    public Ellipse(double r1, double r2)
        {this.r1 = r1; this.r2 = r2;}
    public double area{...}
}

final class Cercle extends Ellipse
{
    public Cercle(double r) {super(r, r);}
    public double getRadius() {return r1;}
}
```

Exercice





instanceOf (2)

```
Ellipse e = new Ellipse(2.0, 4.0);
Cercle c = new Cercle(2.0);
System.out.println("Aire de e:" + e.area() + ", Aire de
    c:" + c.area());
System.out.println((e instanceof Cercle)); // false
System.out.println((e instanceof Ellipse)); // true
System.out.println((c instanceof Cercle)); // true
System.out.println((c instanceof Ellipse));
    // true car dérive de Ellipse
e = c;
System.out.println((e instanceof Cercle)); // true
System.out.println((e instanceof Ellipse)); // true
int r = e.getRadius();
    // Error: method getRadius not found in class Ellipse.
c = e; // Error: Incompatible type for =. Explicit cast needed.
```




Encapsulation

- Une seule classe
 - contrat avec le client
 - interface publique, en Java: outil javadoc
 - implémentation privée
 - Classes imbriquées
 - ==> Règles de visibilité
- Plusieurs classes
 - Paquetage : le répertoire courant est le paquetage par défaut
 - Paquetages utilisateurs
 - Paquetages prédéfinis
 - liés à la variable d'environnement CLASSPATH



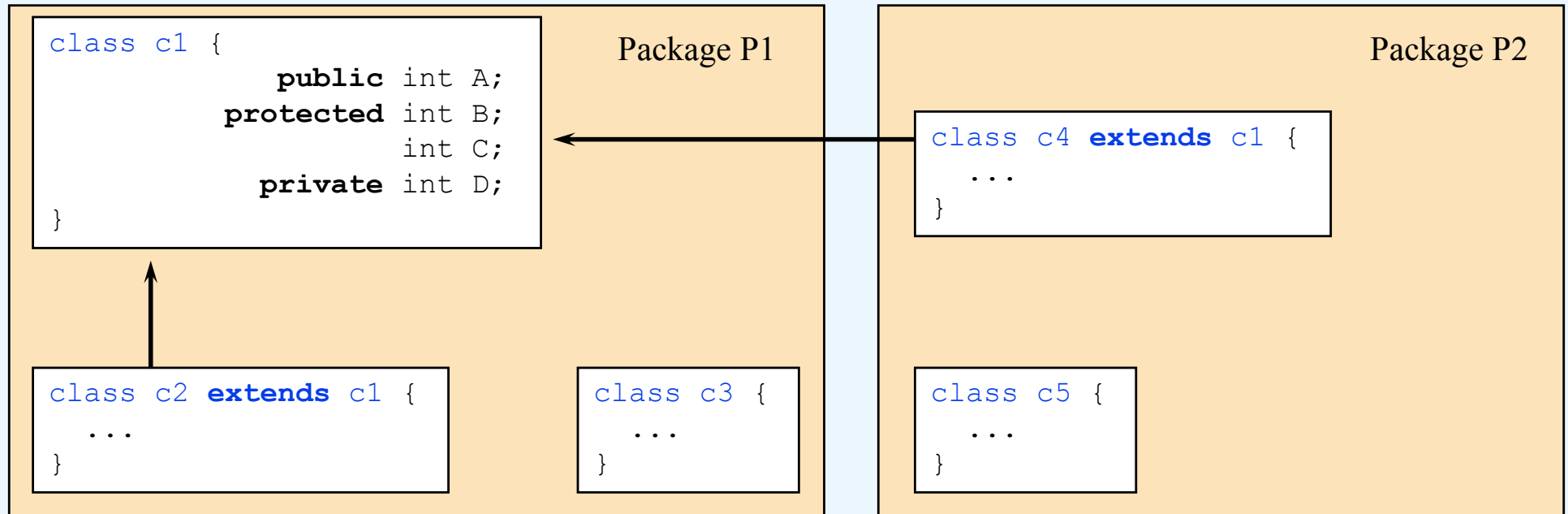
Règles de visibilité: public, privée et protégée

- Tout est visible au sein de la même classe
- Visibilité entre classes
 - sous-classes du même paquetage
 - classes indépendantes du même paquetage
 - sous-classes dans différents paquetages
 - classes indépendantes de paquetages différents
- modificateurs d'accès
 - private
 - par défaut, sans modificateur
 - protected
 - public

Règle d'accès

	private	défaut	protected	public
même classe	oui	oui	oui	oui
même paquetage et sous-classe	non	oui	oui	oui
même paquetage et classe indépendante	non	oui	oui	oui
paquetages différents et sous-classe	non	non	oui	oui
paquetages différents et classe indépendante	non	non	non	oui

Règle d'accès



	A	B	C	D
Accessible par c2	0	0	0	-
Accessible par c3	0	0	0	-
Accessible par c4	0	0	-	-
Accessible par c5	0	-	-	-



Exercice : une explication

- mécanisme de liaison dynamique en Java :
 - La liaison dynamique effectue la sélection d'une méthode en fonction du type constaté de l'objet receveur, la méthode doit appartenir à l'ensemble des méthodes masquées,
 - (la méthode est masquée dans l'une des sous-classes, si elle a exactement la même signature)
 - Sur l'exercice, nous avons uniquement dans la classe B la méthode `m(A a)` masquée
 - en conséquence :
 - `A a = new B();` *// a est de type déclaré A, mais constaté B*
 - `a.m` --> sélection de `((B)a).m(...)` car `m` est bien masquée
 - `a.n` --> sélection de `((A)a.n(...)` car `n` n'est pas masquée dans B
 - Choix d'implémentation : vitesse d'exécution / sémantique ...



Les classes abstraites (1)

- Une classe abstraite est une classe ayant au moins une méthode abstraite.
- Une méthode abstraite ne possède pas de définition.
- Une classe abstraite ne peut pas être instanciée (`new`).
- Une classe dérivée d'une classe abstraite ne redéfinissant pas toutes les méthodes abstraites est elle-même abstraite.



Syntaxe

```
abstract class A
{
    abstract void p();
    int i;                // éventuellement données d'instance
    static int j;        // ou variables de classe
    void q()
    {
        // implémentation de q
    }
}
```



Classe incomplète: java.lang.Number

```
abstract class Number .... {  
    public abstract double doubleValue();  
    public abstract float floatValue();  
    public abstract int intValue();  
    public abstract long longValue();  
}
```

Dérivée par BigDecimal, BigInteger, Byte, Double, Float, Integer, Long, Short



Les classes abstraites

```
class abstract Forme {public abstract double perimetre() ;}
```

```
class Cercle extends Forme {
```

```
...
```

```
public double perimetre() { return 2 * Math.PI * r ; }  
}
```

```
class Rectangle extends Forme{
```

```
...
```

```
public double perimetre() { return 2 * (longueur + largeur); }  
}
```

```
...
```

```
Forme[] formes = {new Cercle(2), new Rectangle(2,3),new Cercle(5)} ;  
double somme = 0 ;  
for(int i=0; i< formes.length; i++) somme += formes[i]. perimetre() ;
```



Les exceptions (1)

- Elles permettent de:
 - prendre en compte les conditions anormales d'exécution d'un programme
 - séparer un bloc d'instructions de la gestion des erreurs pouvant survenir dans ce bloc.
- Ce sont des instances de classes dérivant de `java.lang.Exception`



Les exceptions (2)

```
try
    {
        instructions;
        instructions;
    }
catch( ExceptionType1 e)
    {
        traitement de ce cas anormal de type ExceptionType1;
    }
catch( ExceptionType2 e)
    {
        traitement de ce cas anormal de type ExceptionType2;
        throw e; //l'exception est propagée ( vers le bloc try/catch) englobant
    }
finally                               ⇐ Optionnel
    {
        traitement de fin de bloc try ;
    }
```



Les exceptions (3)

- La levée d'une exception provoque une remontée dans l'appel des méthodes jusqu'à ce qu'un bloc `catch` acceptant cette exception soit trouvé. Si aucun bloc `catch` n'est trouvé, l'exception est capturée par l'interpréteur et le programme s'arrête.
- L'appel à une méthode pouvant lever une exception doit :
 - soit être contenu dans un bloc `try/catch`
 - soit être situé dans une méthode propageant (`throws`) cette classe d'exception
- Un bloc (optionnel) `finally` peut-être posé à la suite des `catch`. Son contenu est exécuté après un `catch` ou après un `break`, un `continue` ou un `return` dans le bloc `try`



Les exceptions sont des classes

```
java.lang.Object
```

```
|
```

```
+--java.lang.Throwable
```

```
|
```

```
+--java.lang.Exception
```

```
|
```

```
+--java.lang.RuntimeException
```

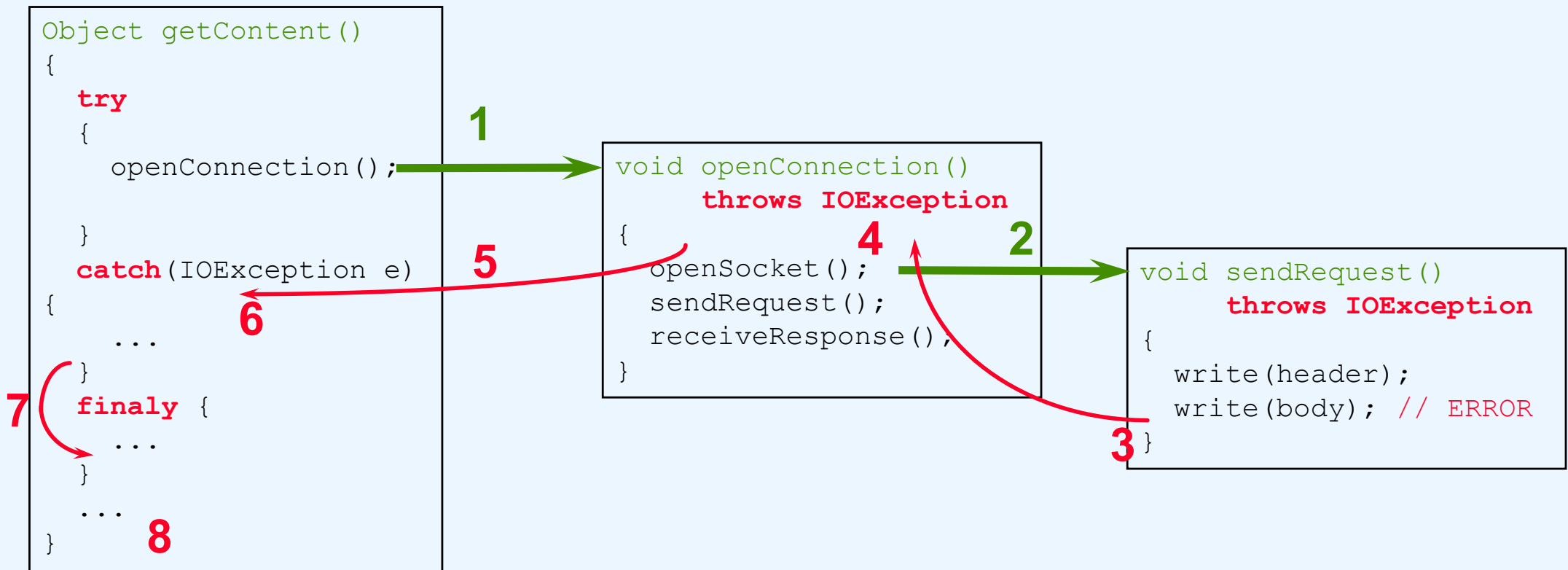
```
|
```

```
+--java.lang.IndexOutOfBoundsException
```

```
+--java.lang.RuntimeException
```

```
ArithmeticException, ArrayStoreException, CannotRedoException,  
CannotUndoException, ClassCastException, CMMException,  
ConcurrentModificationException, EmptyStackException,  
IllegalArgumentException, IllegalMonitorStateException,  
IllegalPathStateException, IllegalStateException, etc.
```

Les exceptions (5)





Les exceptions : exemple

Soit l' instruction suivante : `m = Integer.parseInt(args[0]);`

peut provoquer au moins deux erreurs :

1) `args[0]` n' existe pas

⇒ `arrayIndexOutOfBoundsException`

2) le format de '`args[0]`' n' est pas celui d' un nombre

⇒ `NumberFormatException`



Les exceptions : exemple

```
public class MoisException
{
    public static void main( String[] args)
    {
        String[] mois={"janvier","fevrier","mars","avril","mai","juin", "juillet", "aout",
            "septembre","octobre","novembre","decembre"};

        int[] jours = {31,28,31,30,31,30,31,32,30,31,30,31};

        String printemps = "printemps";    String ete    = "ete";
        String automne   = "automne";      String hiver = "hiver";
        String[] saisons = {hiver,hiver,printemps,printemps,printemps,ete,ete,ete,
            automne,automne,automne,hiver};

        int m;
        .....
```




Les exceptions : exemple

```
try
{
    m = Integer.parseInt(args[0]) -1;
    System.out.println(mois[m] + " est au/en " +saisons[m] + " avec " + jours[m] + " j.");
}
catch(ArrayIndexOutOfBoundsException e)
{
    System.out.println("usage : DOS>java MoisException unEntier[1..12]");
}
catch(NumberFormatException e)
{
    System.out.println("exception " + e + " levee");
}
finally
{
    System.out.println("fin du bloc try ");
}
}
```



Exemple 2

```
public class Div0
{
    public static void main( String args[])
    {
        int den = 0, num = 1;
        boolean b;
        System.out.println("den == " + den);
        b = (den != 0 && num / den > 10);
        b = (den != 0 & num / den > 10);
    }
}
```

Il existe un bloc try/catch prédéfini interne à la machine virtuelle

try

{

Div0.main({""});

}

catch (RuntimeException e) {System.err.println(e); }



NullPointerException

```
static void trier(int[] tab)
{
    // levée d'une exception instance de la classe NullPointerException,
}
```

```
int[] t; // t == null;
trier(t); // filtrée par la machine
```

ou bien

```
int[] t;
try
    {
        trier(t);
    }
catch(NullPointerException e) {}
```



Les interfaces

- Protocole qu'une classe doit respecter
- Caractéristiques
 - Identiques aux classes, mais pas d'implémentation des méthodes (toutes les méthodes sont abstraites)
 - Toutes les variables d'instances doivent être "final", (constantes)
- Classes et interfaces
 - Une classe peut implémenter (`implements`) une ou plusieurs interfaces tout en héritant (`extends`) d'une classe
 - Les méthodes implémentées des interfaces doivent être publiques
 - Une interface peut hériter (`extends`) de plusieurs interfaces.
- la clause *interface*
 - *visibilité interface NomDeLInterface{*
 - *}*
- la clause *implements*
 - *class NomDeLaClasse implements nomDeLInterface, Interface1{*



Exemple prédéfini : Interface Enumeration

- **public interface java.util.Enumeration{**
 - **public boolean** hasMoreElements();
 - **public Object** nextElement();
- **}**
- **Toute classe implémentant cette interface doit générer une suite d'éléments, les appels successifs de la méthode *nextElement* retourne un à un les éléments de la suite**
- **exemple une instance v la classe java.util.Vector**
 - for(Enumeration e = v.elements(); e.hasMoreElements();) {
 - System.out.println(e.nextElement());}
 - avec
 - **public class java.util.Vector{**
 -
 - **public final Enumeration** elements(){ }



Interface et référence

- une "variable Objet" peut utiliser une interface comme type
- une instance de classe implémentant cette interface peut lui être affectée

```
interface I {void p();}
```

```
class A implements I { public void p() { ...} }
```

```
class B implements I { public void p() { ...} }
```

```
class Exemple {
```

```
    I i;
```

```
    i = new A(); i.p(); // i référence le contenu d ' une instance de type A
```

```
    i = new B(); i.p(); // i référence le contenu d ' une instance de type B
```

```
}
```



L'interface implémentée

```
interface ObjetGraphique
{
    public void dessiner();
}
```

```
class PolygoneRegulier implements ObjetGraphique
{
    .....
    public void dessiner()
    {
        .....
    }
}
```

.....



Interface et « spécification »

```
interface PileSpec(  
    public void empiler(Object o);  
    public Object depiler();  
    public boolean EstVide ();  
    public boolean estPleine();  
}  
  
public class PileImpl implements PileSpec {  
    public void empiler(Object o){  
        .....    }  
    public Object depiler(){  
        .....    }  
    .....}
```




Les interfaces

```
abstract class Forme
```

```
{  
    public abstract double perimetre();  
}
```

```
interface Drawable
```

```
{  
    public void dessiner();  
}
```

```
class Cercle extends Forme implements Drawable, Serializable
```

```
{  
    public double perimetre() { return 2 * Math.PI * r ; }  
    public void dessiner() {...}  
}
```



Les interfaces

```
class Rectangle extends Forme implements Drawable, Serializable
{
    ...
    public double perimetre() { return 2 * (height + width); }
    public void dessiner() {...}
}
```

```
...
Drawable[] dd = {new Cercle(2), new Rectangle(2, 3), new Cercle(5)};

for(int i=0; i<dd.length; i++) dd[i].dessiner();
```