Java
Eléments de programmation

1. Structure du langage

Structure du langage

Les types primitifs

- boolean(true/false), byte (1 octet), char (2 octets), short (2 octets), int (4 octets), long (8 octets), float (4 octets), double (8 octets).
- Les variables peuvent être déclarées n'importe où dans un bloc.
- Les affectations non implicites doivent être *castées* (sinon erreur à la compilation).

```
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
```

Structure du langage

Les structures de contrôle et expressions

- Essentiellement les mêmes qu'en C
  - `if`, `switch`, `for`, `while`, do `while`
  - `++`, `+=`, `&&`, `&`, `<<`, `?:`
- 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
                continue;             // Reprend sur la deuxieme boucle while
        }
    }
}
```

Structure du langage

Les tableaux

- Déclaration

```
int[] array_of_int; // équivalent à : int array_of_int[];
Color rgb_cube[][][];
```

- Création et initialisation

```
array_of_int = new int[42];
rgb_cube = new Color[256][256][256];
int[] primes = {1, 2, 3, 5, 7, 7+4};
array_of_int[0] = 3
```

- Utilisation

```
int l = array_of_int.length; // l = 42
int e = array_of_int[50];    // Lève une ArrayIndexOutOfBoundsException
```

Structure du langage

Les exceptions (1)

- Elles permettent de séparer un bloc d'instructions de la gestion des erreurs pouvant survenir dans ce bloc.

```
try {
   // Code pouvant lever des IOException ou des SecurityException
}
catch (IOException e) {
   // Gestion des IOException et des sous-classes de IOException
}
catch (Exception e){
   // Gestion de toutes les autres exceptions
}
```

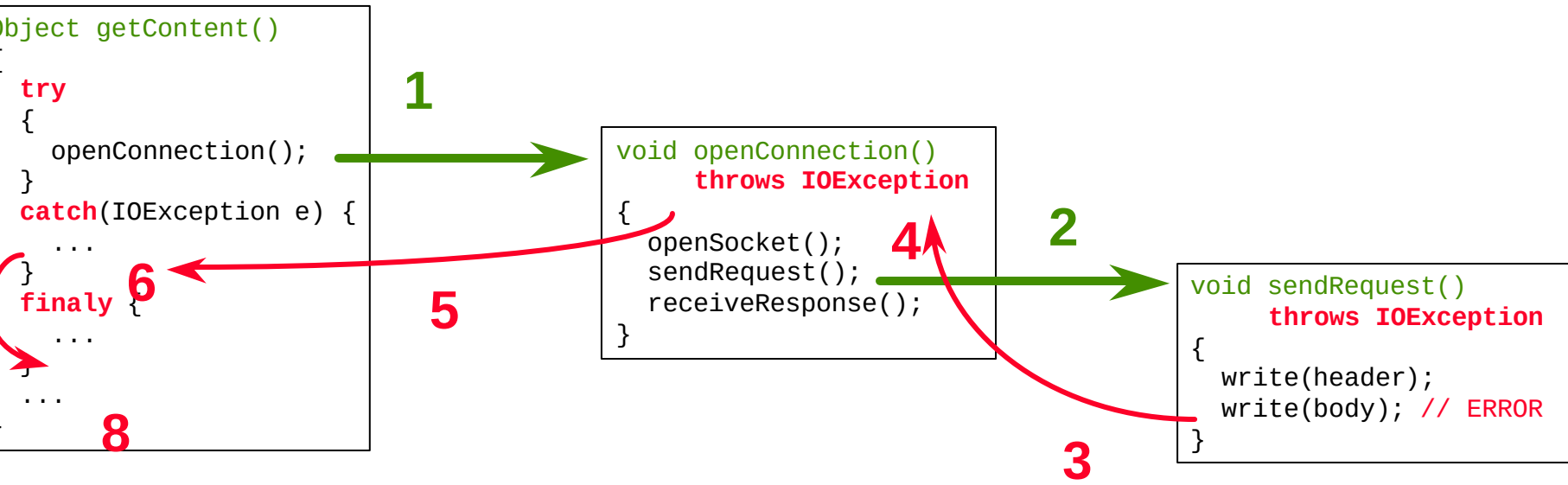
Structure du langage

Les exceptions (2)

- Ce sont des instances de classes dérivant de `java.lang.Exception`
- 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`

Structure du langage

Les exceptions (3)



Structure du langage

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.

Structure du langage

Les packages (1)

- Un package regroupe un ensemble de classes sous un même espace de 'nomage'.
- Les noms des packages suivent le schéma : `name.subname ...`
- Une classe `Watch` appartenant au package `time.clock` doit se trouver dans le fichier `time/clock/Watch.class`
- Les packages permettent au compilateur et à la JVM de localiser les fichier contenant les classes à charger.
- L'instruction `package` indique à quel package appartient la ou les classe(s) de l'unité de compilation (le fichier).

Structure du langage

Les packages (2)

- Les répertoires contenant les packages doivent être présents dans la variable d'environnement `CLASSPATH`
- En dehors du package, les noms des classes sont : `packageName.className`
- L'instruction `import` ***`packageName`*** permet d'utiliser des classes sans les préfixer par leur nom de package.
- Les API sont organisées en package (`java.lang`, `java.io`, ...)

Structure du langage

Les packages (3)

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

~dedieu/classes/graph/2D/Circle.java

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

~dedieu/classes/graph/3D/Sphere.java

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

~dedieu/classes/paintShop/MainClass.java

```
package paintShop;

import graph.2D.*;

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

2. Classes et objets

Classes et objets

Exemple de programme

```
class Circle(){
    public double x, y;        // Coordonnée du centre
    private double r;          // rayon du cercle

    public Circle(double r) {
        this.r = r;
    }
    public double area() {
        return 3.14159 * r * r;
    }
}

public class MonPremierProgramme() {
    public static void main(String[] args) {
        Circle c;           // c est une référence sur un objet Circle, pas un objet
        c = new Circle(5.0); // c référence maintenant un objet alloué en mémoire
        c.x = c.y = 10;
        System.out.println("Aire de c :" + c.area());
    }
}
```

Classes et objets

Création d'un objet (1)

- Pour manipuler un objet, on déclare une référence sur la classe de cet objet :
  `Circle c;`
- Pour créer un objet, on instancie une classe en appliquant l'opérateur `new` sur un de ses constructeurs. Une nouvelle instance de cette classe est alors allouée en mémoire :
  `c = new Circle(5);`
- Toute classe possède un constructeur par défaut, implicite. Il peut être redéfini. Une classe peut avoir plusieurs constructeurs qui diffèrent par le nombre et la nature de leurs paramètres.

Classes et objets

Création d'un objet (2)

```
class Circle() {
   double x, y, r;

   public Circle(double x, double y, double r) {
      this.x = x; this.y = y; this.r = r;
   }
   public Circle(Circle c) {
      x = c.x; y=c.y; r=c.r;
   }
   public Circle() {
      this(0.0, 0.0, 0.0);
   }
}
```

Destruction d'un objet (1)

- La destruction des objets est prise en charge par le *garbage collector* (GC).
- Le GC détruit les objets pour lesquels il n'existe plus de référence.
- Les destructions sont asynchrones (le GC est géré dans une *thread* de basse priorité).
- Aucune garantie n'est apportée quant à la destruction d'un objet.
- Si l'objet possède la méthode `finalize`, celle-ci est appelée lorsque l'objet est détruit.

Classes et objets

Destruction d'un objet (2)

```
public class Circle {
    ...
    void finalize() { System.out.println("Je suis garbage collecte"); }
}
...
    Circle c1;
    if (condition) {
        Circle c2 = new Circle(); // c2 référence une nouvelle instance
        c1 = c2;
    }
    // La référence c2 n'est plus valide mais il reste une référence,c1,
   // sur l'instance

    c1=null; // L'instance ne possède plus de référence. Elle n'est plus
             // accessible. A tout moment le gc peut detruire l'objet.
```

Classes et objets

La structure des classes (1)

- Une classe est un agrégat d'attributs et de méthodes : les *membres*
- Les membres sont accessibles via une instance de la classe ou via la classe (pour les membres statiques)

```
c.r = 3;            // On accède à l'attribut 'r' de l'instance 'c'
a = c.area();       // On invoque la méthode 'area' de l'instance 'c'
pi = Math.PI;       // On accède à l'attribut statique 'PI' de la classe 'Math'
b = Math.sqrt(2.0);// On invoque la méthode statique 'sqrt' de la classe 'Math'
```

- L'accessibilité des membres est pondérée par des critères de visibilité (public, private, ...)
- Les méthodes sont définies directement au sein de la classe.

Classes et objets

La structure des classes (2)

- Le mode de passage des paramètres dans les méthodes dépend de la nature des paramètres :
  - par *référence* pour les objets
  - par *copie* pour les types primitifs

```
public class C {
   void methode1(int i, StringBuffer s) { i++; s.append("d");}

   void methode2() {
       int i = 0;
       StringBuffer s = new StringBuffer("abc");
       methode1(i, s);
       System.out.println(i=" + i + ", s=" + s); // i=0, s=abcd
   }
}
```

Classes et objets

La structure des classes (3)

- 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`.

Classes et objets

La structure des classes (4)

```
public class Circle {
   public static int count = 0;
   public static final double PI = 3.14; // final pour éviter Circle.PI = 4;
   public double x, y, r;

   public Circle(double r) { this.r = r; count++;)

   public Circle bigger(Circle c) {
       if (c.r > r) return c; else return this;
   }

   public static Circle bigger(Circle c1, Circle c2) {
       if (c1.r > c2.r) return c1; else return c2;
   }
}
Circle c1 = new Circle(10); Circle c2 = new Circle(20);
n = Circle.count;                    // n = 2;
Circle c3 = c1.bigger(c2);           // c3 = c2;
Circle c4 = Circle.bigger(c1, c2);   // c4 = c2
```

Classes et objets

L'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.

Classes et objets

L'héritage (2)

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

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

Ellipse e = new Ellipse(2.0, 4.0);
Circle c = new Circle(2.0);
System.out.println("Aire de e:" + e.area() + ", Aire de c:" + c.area());
System.out.println((e instanceOf Circle));  // false
System.out.println((e instanceOf Ellipse)); // true
System.out.println((c instanceOf Circle));  // true
System.out.println((c instanceOf Ellipse)); // true (car Circle dérive de Ellispe)
e = c;
System.out.println((e instanceOf Circle));  // 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.
```

Le masquage des variables (1)

- 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`.

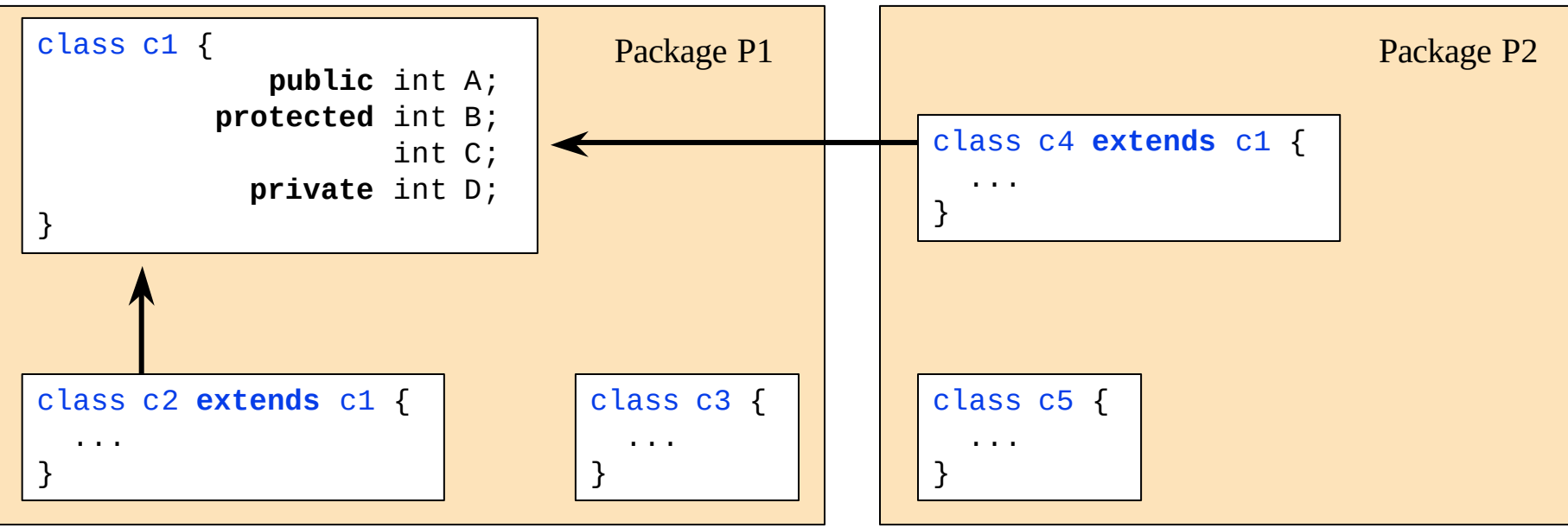
Classes et objets

Le masquage des variables (2)

```
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 reçoit la valeur de la variable x de la classe B
        a = super.super.x; // Syntax error
        a = ((B)this).x;    // a reçoit la valeur de la variable x de la classe B
        a = ((A)this).x;    // a reçoit la valeur de la variable x de la classe A
        super.m();          // Appel à la méthode m de la classe B
        super.super.m();    // Syntax error
        ((B)this).m();      // Appel à la méthode m de la classe C (et non B)
    }
}
```

Classes et objets

L'encapsulation des membres



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

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.

Classes et objets

Les classes abstraites (2)

```
class abstract Shape {
    public abstract double perimeter();
}

class Circle extends Shape {
    ...
    public double perimeter() { return 2 * Math.PI * r ; }
}

class Rectangle extends Shape {
    ...
    public double perimeter() { return 2 * (height + width); }
}


...
Shape[] shapes = {new Circle(2), new Rectangle(2,3), new Circle(5)};
double sum_of_perimeters = 0;
for(int i=0; i<shapes.length; i++)
    sum_of_perimeters = shapes[i].perimeter();
```

Classes et objets

Les interfaces (1)

- Une interface correspond à une classe où toutes les méthodes sont abstraites.
- Une classe peut implémenter (`implements`) une ou plusieurs interfaces tout en héritant (`extends`) d'une classe.
- Une interface peut hériter (`extends`) de plusieurs interfaces.

Classes et objets

Les interfaces (2)

```
abstract class Shape { public abstract double perimeter(); }

interface Drawable { public void draw(); }

class Circle extends Shape implements Drawable, Serializable {
    public double perimeter() { return 2 * Math.PI * r ; }
    public void draw() {...}
}

class Rectangle extends Shape implements Drawable, Serializable {
    ...
    public double perimeter() { return 2 * (height + width); }
    public void draw() {...}
}

...
Drawable[] drawables = {new Circle(2), new Rectangle(2,3), new Circle(5)};
for(int i=0; i<drawables.length; i++)
    drawables[i].draw();
```

Classes et objets

Les *inner classes* (1)

- Introduites avec java 1.1
- Elles permettent de
  - Déclarer une classe dans une bloc (*inner class*)
  - Instancier une classes anonymes (*anonymous class*)
- Elles affinent la localisation des classes
- Elles simplifient le développement
- Elles sont une caractéristique du compilateur et non de la JVM
- **Attention : elles peuvent réduire la lisibilité des sources.**

Classes et objets

Les *inner classes* (2)

- Exemple :

```
public class FixedStack {
    Object array[];
    int top = 0;

    public void push(Object item) { ... }
    public Object pop() { ...}
    public isEmpty() { ...}
    public java.util.Enumeration element() { return new Enumerator(); }

    class Enumerator implements java.util.Enumeration {
        int count = top;
        public boolean hasMoreElements() { return count > 0; }
        public Object nextElement() {
                if (count == 0) throw NoSuchElementExceptio("FixedStack");
                return array[--count];
        }
    }
}
```

Classes et objets

Les *inner classes* (3)

- Exemple de classe locale :

```
...

Enumeration myEnumeration(final Object array[])
{
    class E implements java.util.Enumeration
    {
        int count = top;
        public boolean hasMoreElements() { return count > 0; }
        public Object nextElement() {
                if (count == 0) throw NoSuchElementExceptio("FixedStack");
                return array[--count];
        }
    }

    return new E();
}
```

Classes et objets

Les *inner classes* (4)

- Exemple de classe anonyme :

```
...

Enumeration myEnumeration(final Object array[]) {

    return new java.util.Enumeration() {
        int count = 0;
        public boolean hasMoreElements() { return count < array.length; }
        public Object nextElement() {
                return array[count++];
        }
    }
}
```