Programmation Java
Aspects impératifs …

Objets / valeurs

Types primitifs

Méthodes

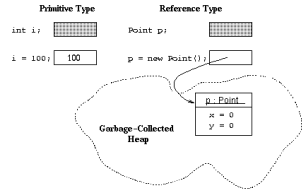
Instructions

Tableaux

Exceptions

Types Primitifs / Types d'Objets

- Les variables de types objets contiennent une référence (pointeur) vers la valeur de l'objet stockée dans le tas.
- Les variables de types primitifs contiennent directement la valeur



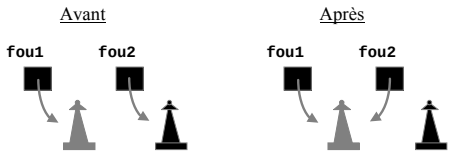
Pointeurs et Objets

- Plusieurs variables peuvent référencer le même objet (attention à l'affectation)
- Exemple:

```
Piece fou1 = new Piece();
Piece fou2 = new Piece();
```

affectation d'Objets

```
fou2 = fou1;
```



Affectation de valeurs de types Primitifs

```
num2 = num1;
```

Types non objet (primitifs ou simples)

Type	Taille/Format	Description
	(integers)	
byte	8-bit complément à 2	entier sur un octet
short	16-bit complément à 2	entier court
int	32-bit complément à 2	entier
long	64-bit complément à 2	entier long
	(nombres réel)	
float	32-bit IEEE 754	Simple-précision
double	64-bit IEEE 754	Double précision
	(autres types)	
char	16-bit Unicode	un simple caractère
boolean	true ou false	une valeur booléenne

Syntaxe et déclaration des variables

syntaxe: n rate x15 a_long_name time_is_$ helloWorld

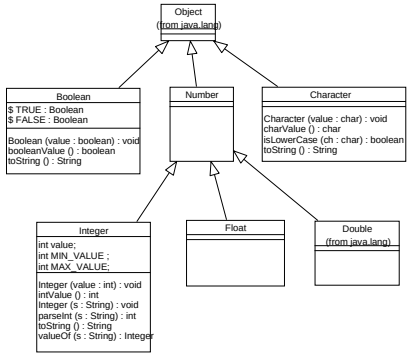
```
int n;
double x;
double rate = 0.07;
char space = ' ';

double x,y;
char   first = 'D',
       middle = 'J',
       last = 'E';
int i, j = 17;
```

Type ou Classe

- une variable à un type définit à la compilation
  int, int[], PrintStream, Integer ou String
- une valeur comme 3 est de type int
- un objet à une classe
  A toute classe correspond un objet (constant) de type Class
  - "Hello world" est de classe String.class
  - new int[3] est de classe int[].class

Par cohérence (?) ceci est généralisé aux objets primitifs
3 est dit de classe int.class

Objets et types de base: wrappers



Unicode

Les programmes Java sont écrits en Unicode (codage sur 16 bits)

Les 128 premiers codes correspondent aux caractères ASCII \u0041 unicode correspond à 0x41 ASCII, c'est à dire 'A'

En partant d'un flot ASCII on ajoute donc 0x00

Les identificateurs sont donc faits de caractères unicodes

Les caractères peuvent être notés 'a' ou '\u0061'

les caractères de fin de lignes doivent être notés \n \r (autres caractères \t \f,..)

Promotion dans les opér. binaires

Ce type de promotion s'applique aux opérandes de certaines opérations:

***,/,%,+,-,**

< <= > >=

== !=

& ^ | opérateurs entiers binaires

? : dans certains cas

si l'un des opérandes est de type double, l'autre est promu en double
sinon si l'un des opérandes est de type float, l'autre est promu en float
sinon si l'un des opérandes est de type long, l'autre est promu en long
sinon les deux sont promus en int

Promotion dans les opér. unaires

Cette promotion s'applique

dans l'évaluation des expressions de dimensionnement des tableaux
dans l'expression d'indexation d'un tableau
pour les opérateurs unaires + et -
dans le cas de l'opérateur unaire binaire ~
sur chaque opérande indépendamment dans les opérateurs >>,<<,>>>

La règle étant que:

si l'opérande est à la compilation de type byte, short ou char
il est promu en int
sinon le type n'est pas modifié

Précédences

opérateurs postfixés	**[] . (paramètres) exp++ exp--**
opérateurs unaires	**++ expr --expr +expr -expr**
création et cast	**new (type)expr**
mult	***,/,%**
...	

exemple

(new t[10]).méthode(arg)

remarque

l'affectation est associative à droite ,

tous les autres opérateurs unaires sont associatifs à gauche

Conversions

implicites

une valeur numérique peut toujours etre affectée à une variable d'un type «plus large»

Un char peut être utilisé là où un int serait attendu

explicites

Une valeur flottante vers une variable entière (troncature)

voir la classe Math pour les arrondis

Un entier vers un type plus petit, on perd les bits de poids fort

byte b= (byte)(3*80); //la valeur est -16

(si on 'cast' un char vers un byte on perd les poids forts et il ne sont pas nécessairement nulls (caratère unicode non Ascii).

Cast d'Objets

```
class Etudiant { ... }
class Collegien extends Etudiant { ... }
class EleveIngenieur extends Etudiant { ... }
Etudiant etudiant1, etudiant2;
etudiant1 = new Collegien (); // ok
etudiant2 = new EleveIngenieur();      // ok
EleveIngenieur etudiant3;
etudiant3 = etudiant2; // erreur de compilation
etudiant3 =(EleveIngenieur) etudiant2;
            // cast explicite, ok
 etudiant3 = (EleveIngenieur) etudiant1;
      // compilation ok, run-time error
```

nombres flottants

de la forme $s.m.2^e$

avec s +1 ou -1

m un entier d'au plus 24 bits(float) ou 53 bits(double)

e un exposant entre -149 et +104 (f), -1075 et +970(d)

Zéro positif et zéro négatif (0.0==-0.0 égaux pour la comparaison)

MAX_VALUE et MIN_VALUE

NaN Not-a-Number

utilisé pour représenter le résultat de la division de zéro par zéro par exemple. Pas ordonné avec le reste.Les comparaisons avec NaN donnent tjs false.

cast et flottants

Les opérations flottantes ne produisent donc pas d'exception

la division par 0 par exemple donnera l'une des valeurs

POSITIVE_INFINITY NEGATIVE_INFINITY

correspondant aux infinis positifs et négatifs

Le cast de et vers d'autres types numériques est possible. Il ne fait pas un arrondi au plus proche (voir Math).

Si NaN il donne 0. Si infini il donne les valeurs min ou max du type cible.

Sinon il tronque vers le nombre inférieur le plus proche

Si le nombre ainsi obtenu est compatible avec un long par exemple, ce long est ensuite "casté" vers le type cible, idem s'il est compatible avec un int.

Exemple de Conversions

```
public class Conversions{
   public static void main(String[] args){
        double infNeg = Double.NEGATIVE_INFINITY;
        double infPos = Double.POSITIVE_INFINITY;

        System.out.println("float: "+(float)infNeg+" ... "+(float)infPos);
        System.out.println("long: "+(long)infNeg+" ... "+(long)infPos);
        System.out.println("int: "+(int)infNeg+" ... "+(int)infPos);
        System.out.println("short: "+(short)infNeg+" ... "+(short)infPos);
        System.out.println("char: "+Integer.toHexString((char)infNeg)+" ... "+
                                Integer.toHexString((char)infPos));
        System.out.println("byte: "+(byte)infNeg+" ... "+(byte)infPos);
   }
}
```

Résultat

float: -Infinity ... Infinity

long: -9223372036854775808 ... 9223372036854775807

int: -2147483648 ... 2147483647

short: 0 ... -1

char: 0 ... ffff

byte: 0 ... -1

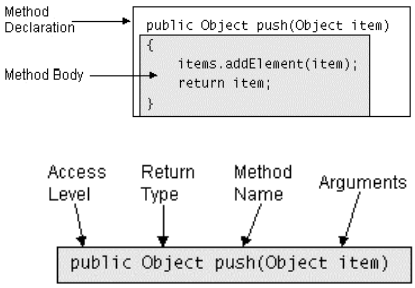
on voit ici que long, int et char donnent bien les valeurs les extrêmes correspondantes

short et byte résulte du fait que le int tonqué donne 0x00 .. 0xff

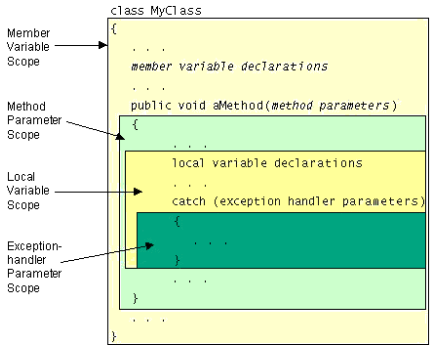
Math

java.lang.Math
double E ; **double PI ;**
double round(double a); **double ceil(double a);** **double floor(double a);** **double rint(double a);** **int abs (int a);** **..** **double random()** **double min (double a ; double b);** **float max (float a ; float b);** **double sin (double a);** **double pow (double a ; double b)** **double sqrt (double a) ;...**

Déclaration des méthodes java



Portée des déclarations de variables



Passage par valeur

```
. . .
int r = -1, g = -1, b = -1;
pen.getRGBColor(r, g, b);
System.out.println("red = " + r + ", green = " + g + ", blue = " + b);. . .

//Avec
class Pen {
    int redValue, greenValue, blueValue;
    void getRGBColor(int red, int green, int blue) {
        // red, green, and blue sont créés
        // et leur valeur est –1
        . . .
        red = redValue;     //ceci ne modifie pas la variable r appelante
    }
}
```

Passage par valeur des références

```
RGBColor penColor = new RGBColor();
pen.getRGBColor(penColor);
System.out.println("red = " + penColor.red + ", green = " + penColor.green + ", blue = " +
penColor.blue);
. . .
//avec
class RGBColor {   public int red, green, blue; }
//et
class Pen {
    int redValue, greenValue, blueValue;
    void getRGBColor(RGBColor aColor) {
        aColor.red = redValue;          //ceci modifie la variable d'instance de penColor
        aColor.green = greenValue;
        aColor.blue = blueValue;
    }
}
```

Surcharge des méthodes

```
class Point { int x, y; }
class ColoredPoint extends Point { int color; }
class Test {
    static void test(ColoredPoint p, Point q) {
        System.out.println("(ColoredPoint, Point)");
    }
    static void test(Point p, ColoredPoint q) {
        System.out.println("(Point, ColoredPoint)");
    }
    public static void main(String[] args) {
        ColoredPoint cp = new ColoredPoint();
        test(cp, cp); // compile-time error
    } // on ne sait pas quelle méthode choisir !
}
```

Méthode la plus spécifique

Pas de problème si une troisième méthode existait telle que

```
static void test(ColoredPoint p, ColoredPoint q) {
    System.out.println("(ColoredPoint, ColoredPoint)");
}
```

Type de retour et sélection

```
class Point { int x, y; }
class ColoredPoint extends Point { int color; }

class Test {
    static int test(ColoredPoint p) {return p.color; }
    static String test(Point p) { return "Point"; }
    public static void main(String[] args) {
        ColoredPoint cp = new ColoredPoint();
        String s = test(cp); // compile-time error
    } // le type de retour n'intervient pas dans la sélection de méthode !
} // la première méthode est sélectionnée mais le type int est incompatible !
```

Spécialisation de méthodes

```
class Point {
    final int EDGE = 20;
    int x, y;
    void move(int dx, int dy) {
        x += dx; y += dy;
        if (Math.abs(x) >= EDGE || Math.abs(y) >= EDGE) clear();
    }
    void clear() {
        System.out.println("\tPoint clear"); x = 0; y = 0; }
}
class ColoredPoint extends Point { int color;
    void clear() {
        System.out.println("\tColoredPoint clear"); super.clear(); color = 0; }
}
```

Exemple

```
class Test {
    public static void main(String[] args) {
        Point p = new Point();
        System.out.println("p.move(20,20):"); p.move(20, 20);
        ColoredPoint cp = new ColoredPoint();
        System.out.println("cp.move(20,20):"); cp.move(20, 20);
        p = new ColoredPoint();
        System.out.println("p.move(20,20), p colored:"); p.move(20, 20); }
}
```

which is:

```
p.move(20,20): Point clear
cp.move(20,20): ColoredPoint clear Point clear
p.move(20,20), p colored: ColoredPoint clear Point clear
```

Recherche de méthode

```
class T1 { String s() { return "1"; } }

class T2 extends T1 { String s() { return "2"; } }

class T3 extends T2 {
    String s() { return "3"; }
    void test() { System.out.println("s()=\t\t"+ s() );
                  System.out.println("super.s()=\t"+ super.s() );
                  System.out.print("((T2)this).s()=\t");
                  System.out.println( ((T2)this).s() );
                  System.out.print("((T1)this).s()=\t");
                  System.out.println( ((T1)this).s() ); }
}
```

Résultats

```
class Test {
    public static void main(String[] args) {
        T3 t3 = new T3();
        t3.test();
    }
}
```

which produces the output:

s()= 3

super.s()= 2

((T2)this).s()= 3

((T1)this).s()= 3

masquage des identificateurs

Une méthode d'instance ne peut masquer une méthode static

Au contraire une variable 'static' peut masquer une variable d'instance et inversement

l'accès à la méthode masquée peut se faire par super .

Le cast n'a pas d'effet contrairement au cas des attributs

((SuperClasse)this).m() # ((SuperClasse)this).a

super.m() ~ super.a

On ne peut pas utiliser super plusieurs fois.

une déclaration de variable locale masque non seulement une autre déclaration de variable mais aussi de type (**mieux vaut respecter les conventions**).

les noms de package ne masquent jamais les autres déclarations.

Une classe peut hériter des attributs de même nom de classes et interfaces différentes, l'ambigüité doit seulement être levée à l'usage.

On peut hériter plusieurs fois des mêmes déclarations.

Initialisation des variables

Les variables de classe et d'instance sont initialisées par défaut à 0, 0.0,null,\u0000, ou false

Attention:

Les variables locales aux méthodes,à l'instruction for ou aux blocs ne le sont pas.

la portée d'une déclaration locale à un bloc est limitée au bloc.

les variables locales n'existent que le temps de vie de leur contexte

les déclarations locales masquent les déclarations de même nom

Instructions

bloc

```
{  // This block exchanges the values of x and y
        int temp = x;          // declare temp and store x in it
        x = y;                 // copy value of y into x
        y = temp;              // copy value of temp into y
}
```

if then else

attention aux ambiguités de lecture

```
if(c1) if(c2) i1;
else i2;
```

A qui appartient le else ?

Java comme C ou C++ attribue le else au if le plus imbriqué auquel il peut légitimement appartenir.

mieux vaut donc écrire

```
if(c1){
    if(c2) i1;
    else i2;
}
```

Switch

```
int i;
....
switch(i){    //i est un chiffre décimal
    case 2:
    case 4:
    case 6:
    case 8: System.out.println(" i est pair "); break;
    default:System.out.println(" i est impair ");
}
```

remarque: Il est recommandé de mettre un break même dans le dernier case car une erreur surviendrait aisément en cas d'ajout d'un cas supplémentaire.

while

```
while (boolean-expression)
        statement

do
        statement
while (boolean-expression);
```

for

En java l'opérateur ',' de C est limité au for

```
for (int i= 0, j=0 ; j<10 ; i++, j++){
        ...
}
```

En 1.5

```
int[] tab = new int[10];
for (int val: tab){
        ...val..
}
```

Etiquettes

syntaxe

Identifier : *Statement*

Elles sont utilisées par les instructions break et continue.

Les étiquettes ne masquent aucun autre nom,ils peuvent donc être identiques à un nom de variable de type,de package ou de méthode. (pas d'ambiguïté)

Deux étiquettes identiques ne peuvent pas être imbriquées.

break

sans label

permet de sortir 'normalement' de la structure switch, while, do ou for (et seulement de celles-là sinon erreur de compilation) immédiatement englobante

avec label

sort vers la structure englobante (il en faut une mais celle ci n'est pas nécessairement un switch, while, do ou for) qui possède cette étiquette. (attention de toutes façon si le break est dans des try on fera avant de sortir les éventuels finally).

continue

seulement dans une boucle (while, do ou for)

sans label elle provoque le démarrage de l'itération suivante de la boucle immédiatement englobante.

avec label (il doit être devant un boucle englobante) c'est alors celle-ci qui passe à l'itération suivante.

Ici encore d'éventuels blocs finally seraient exécutés avant le passage à la boucle suivante.

Tableaux

- Un tableau est une liste ordonnée de valeurs de même type. Ce type peut être un type primitif ou une classe (ou interface)

•Les tableaux sont des objets. (types références)

•Les tableaux sont de taille fixe et leur limites sont controlées.

•La longueur d'un tableau ta est: **`ta.length`**

•Un tableau de taille N est indicé de 0 à N-1

Exemples

```
int[] ia = new int[3];
ia[0] = 1; ia[1] = 2; ia[2] = 3;


int[] ia = { 1, 2, 3};


float[][] mat = new float[4][4];
for (int y = 0; y < mat.length; y++) {
  for (int x = 0; x < mat[y].length; x++)
    mat[y][x] = 0.0;
}
```

Déclaration et initialisation des tableaux

int[] t = new int[3];

int[] t={ 0,5,2};

int[] t = new int[]{0,5,2};

String et Array

Les tableaux doivent être considérés comme des objets (un peu spéciaux puisqu'il n'y a pas de classe explicite).

la classe de t dans

int [] t = new int[3];

est int[].class

Attention String.class et char[].class sont différents.

attention : pour les longueurs dans un cas on a un attribut dans l'autre une méthode

"abc".length() t.length

On peut passer de l'un à l'autre par de méthodes de String

char[] toCharArray()

String valueOf(char[] data)

Arrays et Paramètres de méthodes

- Un tableau peut être passé en paramètre à une méthode
- Comme pour tout autre objet c'est la référence qui est passée.
- Paramètres formel et effectif désignent donc le même tableau
- Changer un élément du tableau dans la méthode change donc l'élément original.

Arrays et Objets

- Les éléments d'un tableau peuvent être des objets (références)

```
String[] words = new String[25];
```

- Cette déclaration ne crée **pas** les objets String
- Chaque objet rangé dans un tableau doit être créé et instancié séparement
  - Words[0]= new String("abcd");

Arguments de la ligne de Commande

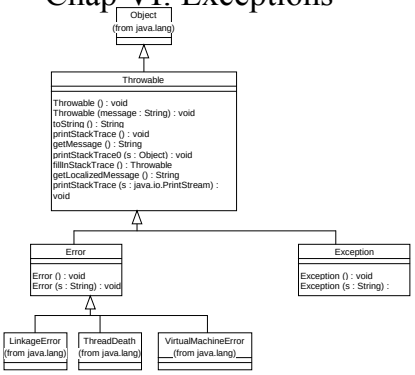
```
public class ArgumentsLigneCommande {

  public static void main(String[] args) {
    int length = args.length;
    System.out.println("args.length=" + length);
    for (int i = 0; i < length; i++) {
      System.out.println("args[" + i + "]=" + args[i]);
    }
  }

}
```

Sortie

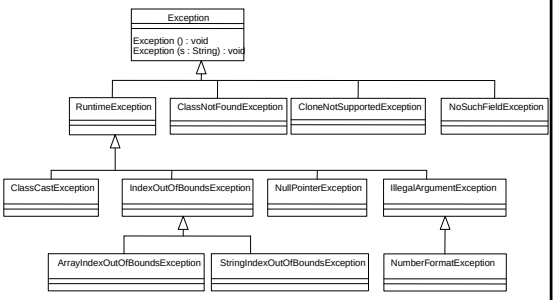
```
C:\Examples>java CommandLineArguments
args.length=0

C:\Examples>java CommandLineArguments Hello World!
args.length=2
args[0]=Hello
args[1]=World!
```

Chap VI: Exceptions

java.lang



Sous classes d' Exception



printStackTrace

originating the exception in f()
Inside g(), e.printStackTrace()
java.lang.Exception: thrown from f()
    at Rethrowing.f(Rethrowing.java:8)
    at Rethrowing.g(Rethrowing.java:12)
    at Rethrowing.main(Rethrowing.java:24)
Caught in main, e.printStackTrace()
java.lang.Exception: thrown from f()
    at Rethrowing.g(Rethrowing.java:18)
    at Rethrowing.main(Rethrowing.java:24)

Exception Handling

- bloc Try-catch et bloc finally

```
try {
  // code may cause/throw exceptions
} catch (Exception1 e1) {
  // handle Exception1
} catch (Exception2 e2) {
  // handle Exception2
} finally {
  // optional
  // code always executed whether an exception
  //   is thrown or not
}
```

Throw Statement et Throws Clause

```
public class MyClass {

  public void aMethod(int i, int j) throws MyException {
    // ...

    throw new MyException(reason);
    // ...
  }
}
```

Appel de méthode et Exception

- Si on appelle une méthode qui déclare lever une exception, on doit
  - Soit encadrer cet appel d'un bloc try/catch
  - Soit déclarer dans l'entête de la méthode englobante que celle-ci lève cette même exception non traitée localement

Créer ses propres Exceptions

```
package com.me.mypackage;

public class ResourceLoadException extends Exception {

        public ResourceLoadException(String message) {
          super(message);
        }

}

...
public class ResourceLoader {
  public getResource(String name) throws ResourceLoadException {
    ...
}
```

Exceptions: Quelles infos ?

Une exception comporte trois type d'info

Le type de l'exception grâce auquel on peut choisir de n'en récupérer que certaines et de laisser passer les autres

La trace de la pile des appels de procédure qui est source de l'exception grâce à laquelle on peut en connaître l'origine

Le message renvoyé par l'exception qui doit être destiné à l'utilisateur

Trace de la pile des appels

```
try {
        int a[] = new int[2];
        a[4];
} catch (ArrayIndexOutOfBoundsException e) {
        System.out.println(     "exception: " +
                                e.getMessage());
                                e.printStackTrace();
}
```

```
Exception in thread "main" java.lang.NumberFormatException: For input string:
"51,319.3506"
    at java.lang.NumberFormatException.forInputString(NumberFormatException.java:48)
    at java.lang.FloatingDecimal.readJavaFormatString(FloatingDecimal.java:1213)
    at java.lang.Double.parseDouble(Double.java:202)
    at Cal_to_MJD.Cal_MJD(Cal_to_MJD.java:43)
    at Cal_to_MJD.main(Cal_to_MJD.java:54)
```

Héritage et exceptions

Essayer de regrouper les classes d'exceptions pertinentes en une hiérarchie de classes

les procédures déclarent renvoyer le parent tandis que l'objet effectivement levé est une instance d'une sous-classe et l'appelant peut décider de traiter ou non indépendamment

Exemple:

public final char **readChar**() throws IOException

Returns: the next two bytes of this input stream as a Unicode character.

Throws:

EOFException - if this input stream reaches the end before reading two bytes.

IOException - if an I/O error occurs.

Exceptions : Conception

-Une procédure ne doit pas lever plus de trois exceptions

-Les types levés doivent être conçus du point de vue de l'appelant

remonter une exception de bas niveau (implémentation) fait perdre le niveau d'abstraction des couches (abstraction)

```
public class ResourceLoader {

  public getResource(String name) throws ResourceLoadException {

        try {
            // try to load the resource from the database
          ...
        }
        catch (SQLException e) {
            throw new ResourceLoadException(e.toString());
        }

}
```

Attention

Renvoyer une exception nouvelle c'est perdre la trace de la pile initiale

```
public class ResourceLoader {

public loadResource(String resourceName) throws ResourceLoadException {
 Resource r;
      try {
            r = loadResourceFromDB(resourceName);
      } catch (SQLException e) {
            throw new ResourceLoadException("SQL Exception loading resource "
            + resourceName: " + e.toString());
      }
}
}
```

Chaîner les exceptions

```
public class ResourceLoader {

public loadResource(String resourceName) throws ResourceLoadException {
 Resource r;
      try {
            r = loadResourceFromDB(resourceName);
      } catch (SQLException e) {
            throw new ResourceLoadException("Unable to load resource "
                                                + resourceName, e); );
      }
}
}
```

Etendre ChainedException

```
public class ResourceLoadException extends ChainedException {
      public ResourceLoadException(String message) {
            super(message);
      }
      public ResourceLoadException(String message, Throwable cause) {
            super(message, cause);
      }
}
```

Chaînage des exceptions

```
public class ChainedException extends Exception {
      private Throwable cause = null;
      public ChainedException() { super(); }
      public ChainedException(String message) { super(message); }
      public ChainedException(String message, Throwable cause) {
            super(message); this.cause = cause;
      }
      public Throwable getCause() { return cause; }
      public void printStackTrace() {
            super.printStackTrace();
            if (cause != null) { ps.println("Caused
            by:");cause.printStackTrace(); }
      }
      }
```