CHAPITRE 3 : MANIPULATION DES CLASSES ET OBJETS

Dr. A. Nejeoui

POO JAVA: GI2

INTRODUCTION

objectifs :

- Création et manipulation des variables d'instance / variables de classe
- Déclaration et manipulation des méthodes d'instance / méthodes de classe
- Le mot clé this
- Passage d'arguments à une méthode
- La surcharge des méthodes
- Déclaration et manipulation des Constructeurs
- Création des objet / Opérateur new
- La redéfinition des méthodes
- Les tableaux
- Le transtypage

DÉFINITION D'UNE CLASSE

Une classe est définie par le mot clé **class** suivi du nom de la classe comme suit :

```
class UneClasse {
// le corps de la classe
}
```

Par défaut toute classe hérite de la classe Object. C'est la super-classe dans la hiérarchie des classes Java.

Si une classe hérite d'une super classe on utilise le mot clés **extends.**

```
class UneClasse extends UneSuperClasse {
// le corps de la classe
}
```

VARIABLES D'INSTANCE

Une variable est dite **variable d'instance** si elle est déclarée en dehors du corps d'une méthode et elle n'est pas modifiée par le mot clé static.

Par convention les variables membres (variables d'instance + variables de classe) sont déclarées juste après l'accolade ouvrante du corps de la classe.

Exemple :

```
class Voiture {
String couleur;
String marque;
int nbrPortes;
Moteur motor;
}
```

Mais c'est possible de déclarer les variables membres n'import où dans le corps de la classe en dehors du corps de toute méthode.

VARIABLES DE CLASSE

une variables de classe est une variable dont la valeur est partagée par tous les objets instances de cette classe par opposition à une variable d'instance qui peut changer de valeur d'un objet à l'autre.

On déclare un variable de classe par le mot clés **static**

Exemple :

```
class Voiture {
String couleur;
String marque;
int nbrPortes;
static int nbrRoues;
static int vitesseMax=300;
Moteur motor;
}
```

LES MÉTHODES D'INSTANCE/ CLASSE

Une méthode d'instance définit le comportement des objets de la classe (l'ensemble des tâche que ces objets peuvent accomplir).

En Java la définition d'une méthode comporte quatre parties :

1 Le nom de la méthode

2 La liste des paramètres

3 Le type d'objet ou type primitif retourné par la méthode

4 Le corps de la méthode

```
TypeRetourné nomMéthode (type1 arg1, type2 arg2, …, typeN argN)
{
// le corps de la méthode
    }
```

Rq: Pour simplifier nous avons ignoré deux parties optionnelles : les Modificateurs et le mot clés throws.

1 **Le nom de la méthode** : peut être tout identificateur valide.

2 **La liste des paramètres** : un ensemble de déclarations de variables séparées par des virgules délimité par deux parenthèses ().

LES MÉTHODES D'INSTANCE/ CLASSE

Les paramètres de la méthode sont traités dans le corps de la méthode comme des variables locales. Ils reçoivent leurs valeurs lors d'un appel à cette méthode.

3 Le type retourné peut être de type primitif, classe ou void si la méthode ne renvoie aucun valeur. Si le type retourné est de type tableau, les crochets doivent être placées soit après le type soit après la liste des paramètres.

Exemple :

```
String[ ] mots(String paragraphe) {
String[ ] parts = paragrapahe.split(“ ”);
return parts;
}
```

```
String mots(String paragraphe)[ ] {
String[ ] parts = paragrapahe.split(“ ”);
return parts;
}
```

Si le type retourné est différent de void la méthode doit obligatoirement retourner une valeur de type **compatible** avec le type retourné déclaré.

4 Le Corps de la méthode : Dans le corps de la méthode vous pouvez avoir des instructions, des expressions, des boucles et des appels à d'autres méthodes.

LE MOT CLÉ THIS

this représente l'objet courant, il est utilisé là où un objet instance de la classe est attendu.

Exemple :

```
classe Figure2D{
double largeur;
double longueur;
public void setLargeur(double largeur ){  this.largeur=largeur;  }
public double getLargeur() {
        return    this.largeur;
        }
public double getLongueur(){
        return   this.longueur; }
public double surface(){
return this.getLargeur()*this.getLongoeur();
        }
}
```

Les méthodes de classes déclarées avec le mot clé static ne peuvent pas utiliser **this**

PASSAGE D'ARGUMENTS À UNE MÉTHODE

```java
package ma.ac.ensa;
public class PassageArguments {
    public static int changer(int x) {
        x=x+5;
        return x;
    }
    public static void main(String[] args) {
        int a=0;
        changer(a);
        System.out.println(a);
    }
}
```

En java toutes les variables sont passées par valeur : La règle générale de Java est que les arguments sont passés par valeur. Cela signifie que l'appel de méthode se fait par copie des valeurs passées en argument et que chaque appel de méthode dispose de sa propre version des paramètres.

PASSAGE D'ARGUMENTS À UNE MÉTHODE

```java
package ma.ac.ensa;
public class PassageArguments {
    public int a;
    public String str;
    public static int changer(int x) {
        x=x+5;
        return x;
    }
    public static void changer(PassageArguments x) {
        x.a+=5;
    }
    public static void main(String[] args) {
        PassageArguments pa=new PassageArguments();
        changer(pa.a);
        System.out.println(pa.a);
        changer(pa);
        System.out.println(pa.a);
    }
}
```

Lorsqu'un objet est passé en paramètre, ce n'est pas l'objet lui même qui est passé mais une référence sur l'objet. La référence est bien transmise par valeur et ne peut pas être modifiée mais l'objet peut être modifié via l'appel d'une méthode.

Pour transmettre des arguments par référence à une méthode, il faut les encapsuler dans un objet qui prévoit les méthodes nécessaires pour les mises à jour.

MÉTHODE DE CLASSE

Une méthode de classe définit un comportement de la classe entière et non d'un objet spécifique. Pour définir une méthode de classe on utilise le mot clé static. Une méthode de classe n'a pas besoins d'un objet pour s'exécuter, on peut y accéder directement par le nom de la classe exemples :

Character.isUpperCase('d');

int n=PassageArguments.changer(102);

On utilise une méthode static lorsqu'on veut définir un comportement de tous les objets instance de la classe, lorsqu'il s'agit d'un comportement d'un objet spécifique on utilise une méthode d'instance.

SURCHARGE DE MÉTHODES

```java
package ma.ac.ensa;
public class Surcharge {
    public int a;
    public String str;
    public  int changer(int x) {
        x=x+5;
        return x;
    }
    public  double changer(double x) {
        x=x+5;
        return x;
    }
    public  void changer(Surcharge x) {
        x.a+=5;
    }
}

```

Surcharger une méthode consiste à déclarer une deuxième méthode qui porte le même nom que la première avec une liste d'arguments différente.

Avantages :

- Éliminer la recherche des noms de plusieurs méthodes qui font la même chose.
- Permettre de définir des méthode qui agissent différemment selon les arguments passés.

LES CONSTRUCTEURS

Un constructeur d'une classe est une méthode utilisée par la MVJ lors de la création des objets instances de cette classe. Par contre aux autres méthodes les constructeurs ne peuvent être appelés directement dans une application. Ils sont appelés automatiquement par la MVJ quand un objet est crée par l'opérateur new.

Java effectue trois tâche lorsque l'opérateur new est utilisé pour créer un objet instance d'une classe :

1- Allouer la mémoire

2- Initialiser les variables d'instance de cet objet, soit par des valeur initiales soit par les valeurs par défaut (0 pour les nombres, false pour boolean, null pour les objets et '\0' pour char).

3- Exécuter les bloques d'initialisation.

4- Appeler l'un des constructeurs de la classe

LES CONSTRUCTEURS

```java
package ma.ac.ensa;
public class Constructeurs {
    public int a;
    public String str;
    public   Constructeurs() {
        super();
    }
    public   Constructeurs(int a) {
        super();
        this.a=a;
    }
     public   Constructeurs(int a,String str) {
        this(a);
        this.str=str;
    }

}
```

- Un constructeur diffère d'une méthode d'instance en trois points :

1- Il porte toujours le nom de la classe

2- Il n'a pas un type de retour

3- Il ne renvoi aucun valeur (pas de mot clé return dans le corps du constructeur).

- La premiere instruction d'un constructeur doit être l'appel d'un constructeur de la même classe ou de sa super classe. **Ce qui assure que les variables membres des super classes sont initialisées en premier.**
- **super**(arg1,arg2,…,argN) : appel d'un constructeur de la super classe.
- **this**(arg1,arg2,…,argN) : appel d'un constructeur de la même classe.

LES BLOQUES D'INITIALISATION

```
 4 import java.util.Random;
 5 import org.junit.Test;
 6 public class BloquesInitialisation {
 7     public byte[] a;
 8     public static final byte[] PRIME;
 9     {  a=new byte[32];
10         new Random().nextBytes(a);
11     }
12     static {  PRIME=new byte[32];
13     BigInteger bigPrime=
14         new BigInteger("78F33FFF66FFFFFFFFFABCFFF8889C", 16);
15     int size=bigPrime.toByteArray().length;
16         System.arraycopy(bigPrime.toByteArray(),0,PRIME,32-size,size);
17     }
18     @Test
19     public  void testInitialization() {
20         assertEquals(-100, PRIME[PRIME.length-1]);
21     }
22 }
```

REDÉFINITION D'UNE MÉTHODE

```
1  package ma.ac.ensa;
2  public class SuperClasse {
3      public byte[] a;
4      public Object m() {
5          return a;
6      }
7      @Override
8      public String toString() {
9          return ""+a.length;
10     }   }
11 class SouClasse extends SuperClasse{
12     String str;
13     @Override
14     public  byte[]  m() {
15         return a;   }
16     @Override
17     public String toString() {
18         return super.toString()+str;
19     }   }
```

La redéfinition d'une méthode consiste à re-déclarer cette méthode dans la sous classe avec le même nom et la même liste d'arguments **(même signature)**. et **un type de retour compatible** avec le type de retour de la super classe.

pour différencier les méthodes déclarées dans la super classe de celles déclarées dans la classe courante on utilise les mot clés **super** et **this** respectivement.

CRÉATION DES OBJET(NEW)

L'instantiation des objets nécessite l'appel de l'opérateur new suivi par un constructeur de la classe à instancier. Exemple :

Date date =**new** Date (1234567890L);

La gestion de mémoire en java se est automatique, Elle est assurée par la ramasse-miette Java (JGC). Lorsque on crée un nouvel objet par l'opérateur **new** , java alloue automatiquement la taille exacte de la mémoire nécessaire pour stocker cet objet.

Quand l'objet n'est plus référencé dans l'application, java le détruit automatiquement et libère l'espace mémoire occupé.

NB : les performances du JGC différent d'une JVM a une autres.

Pour des JGC optimisés voir Zing (azul.com).

LES TABLEAUX

Dans certains cas on a besoins de gérer plusieurs variables de même type, par exemple 100 variables de type entier, au lieu de déclarer 100 différentes variables avec 100 identificateurs différents, il convient d'utiliser un tableau d'entiers de taille 100.

On peut déclarer un tableau qui peut stocker des valeurs primitives, des références à des objets d'une classe, des références à des interfaces ou bien des références à des tableaux.

Une fois on a déclaré un tableau d'un type donné on ne peut y stocker que des valeurs de types compatibles au type déclaré.

Ex : byte[] a=**new** byte**[32]**;

ce tableau peut stocker des valeurs de type byte.

Object[] b=**new** Object**[64]**;

ce tableau peut stocker des références à des objets de type Object ou de type **compatible**.

LES TABLEAUX

Pour créer un tableau en java on suit les étapes suivantes :
1- déclarer une variable qui va stocker une référence au tableau
2- réserver l'espace mémoire pour stocker les éléments du tableau
3- stocker les éléments du tableau

Déclaration d'une variable qui va stocker une référence au tableau

Int [] n;
Point[] pt;

C'est le même principe utilisé pour déclarer une variable régulière (seule valeur), les deux crochets sont ajoutés pour différencier les tableaux des autres variables.

On peut aussi mettre les deux crochets après le nom de la variable comme suit

Int n[];
Point pt[];

Réservation de l'espace mémoire

Car les tableau en java sont aussi des objets, on utilise l'opérateur new pour créer un tableau.

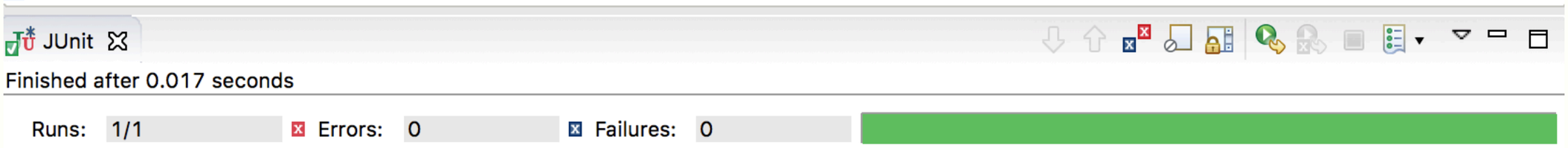
Point[] pt=new Point[10];

Stockage des éléments du tableau

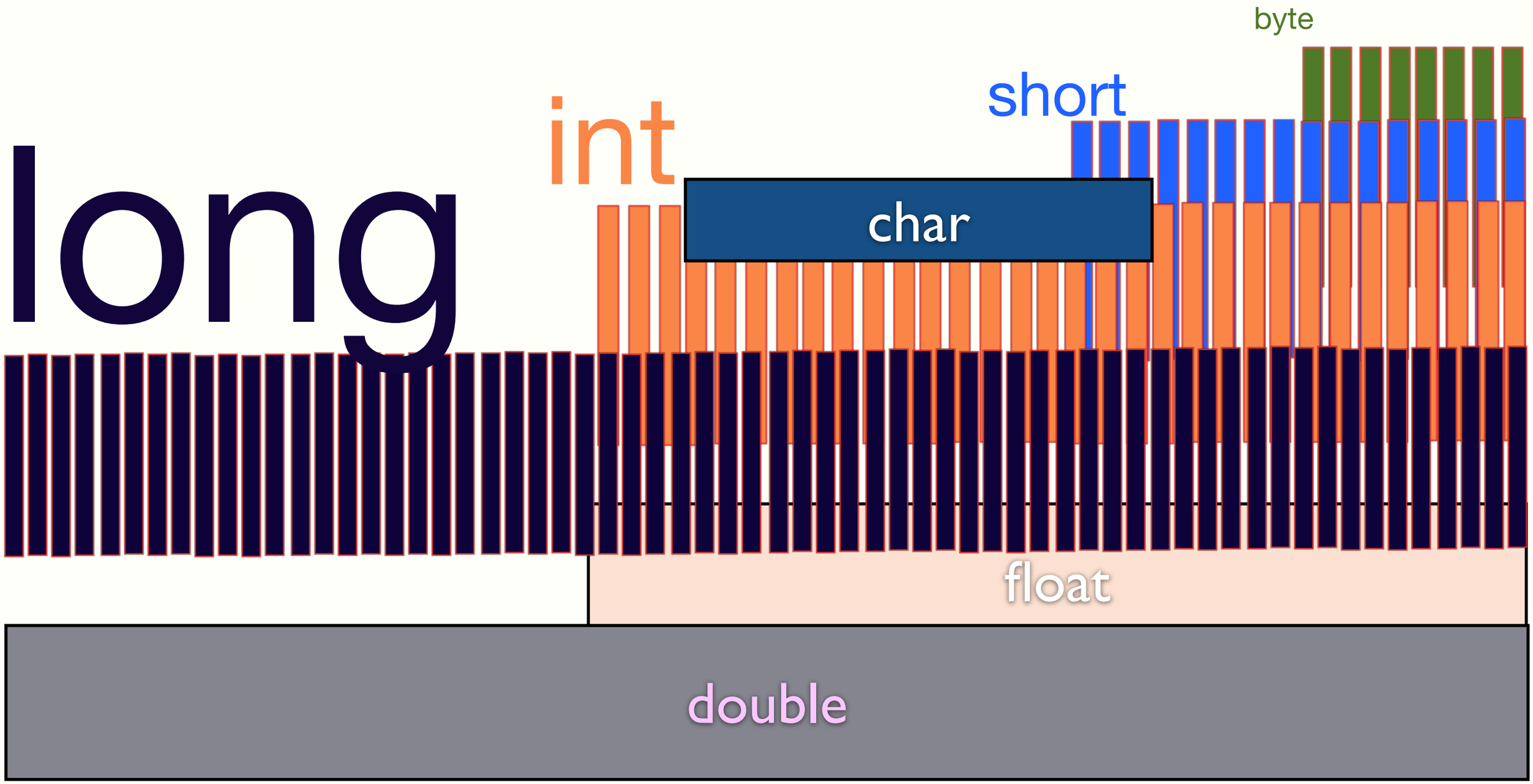
for (int i=0;i<10;i++) pt[i]=new Point();

LES TABLEAUX

```
 4 import org.junit.Test;
 5 public class Tableaux {
 6     @Test
 7     public  void test() {
 8         int[] tab= {1,2,3};
 9         int[] matrice[]= {{1},{2},{3}};
10         int[][] matrice2= {{1},{2},{3}};
11         int matrice3[][]= {{1},{2},{3}};
12     //  tab= {1,2,3};//Erreur de copilation
13         tab=new int[] {1,2,3};
14         Arrays.sort(tab);
15         int pos=Arrays.binarySearch(tab, 2);
16         System.arraycopy(new int[] {1, 2, 3},0, new int[3],0, 3);
17         assertEquals(pos, 1);
18     }
19 }
```



TRANSTYPAGE IMPLICITE : ELARGISSEMENT
TYPES PRIMITIFS



TRANSTYPAGE IMPLICITE : ELARGISSEMENT TYPES PRIMITIFS

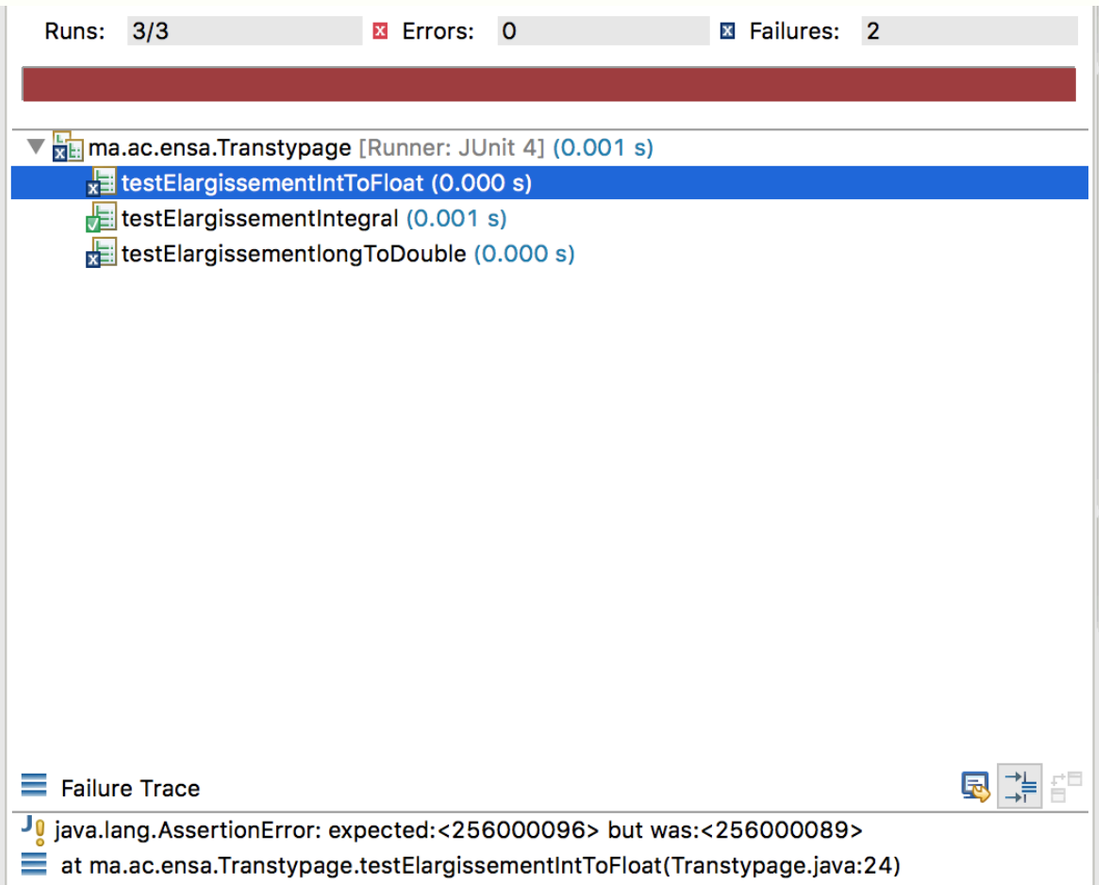
- double > float **>** long > int > short > byte
- int > char
- boolean incompatible avec les autres types
- char et short sont incompatible
- char et byte sont incompatibles

On peut Convertir implicitement un type plus petit en un type plus large

NB : Voir Section 5.1.2 de la Specification du language Java

TRANSTYPAGE IMPLICITE : ELARGISSEMENT INT OU LONG VERS FLOAT

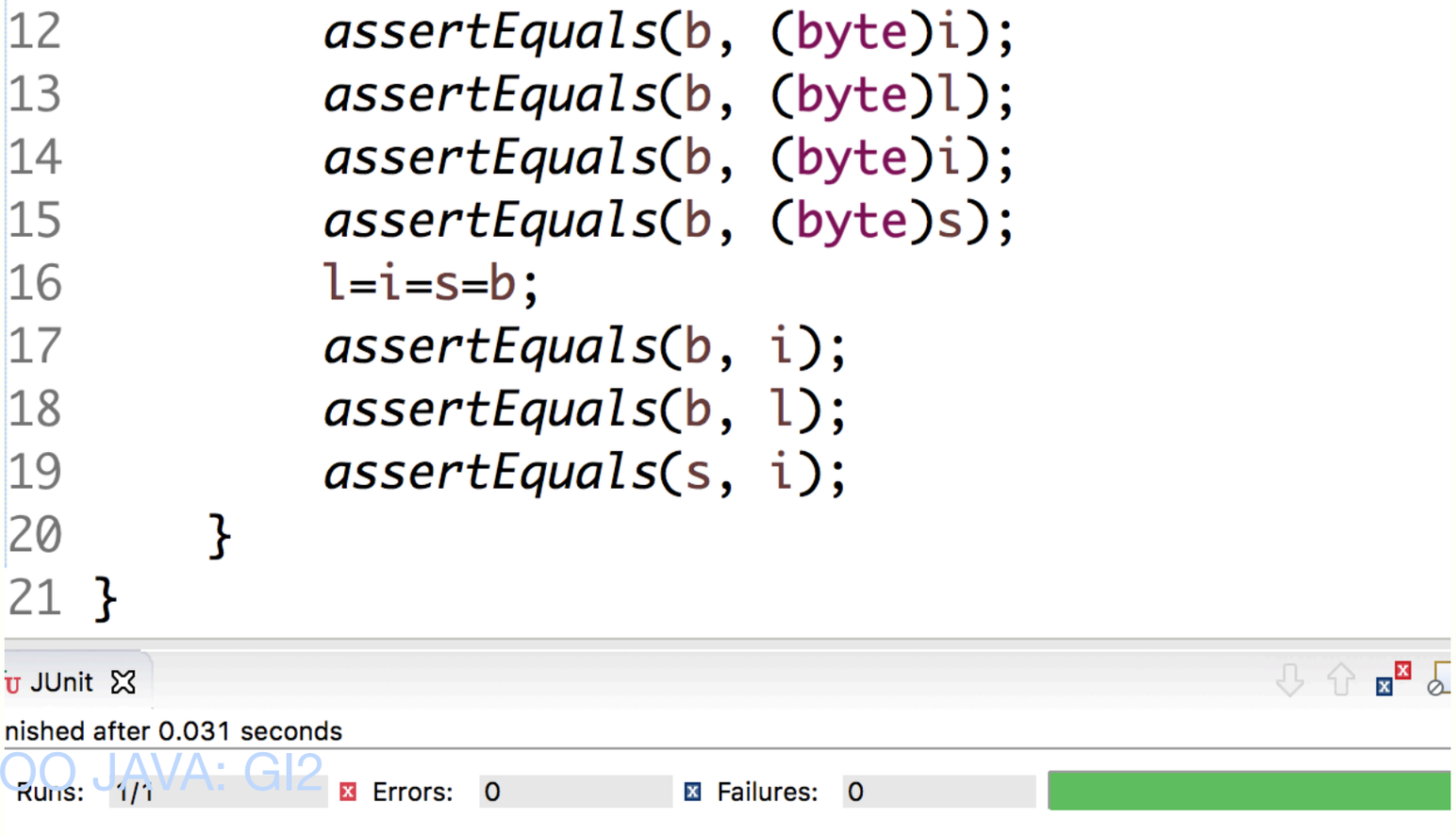
```
5  public class Transtypage {
6      byte b=89;
7      short s=25689;
8      int i=256000089;
9      long l=2560000000000000089l;
10     float f=256000089;
11     double d=2560000000000000089l;
12     @Test
13     public  void testElargissementIntegral() {
14         assertEquals(b, (byte)i);
15         assertEquals(b, (byte)l);
16         assertEquals(b, (byte)i);
17         assertEquals(b, (byte)s);
18         l=i=s=b;
19         assertEquals(b, i);
20         assertEquals(b, l);
21         assertEquals(s, i); }
22     @Test
23     public  void testElargissementIntToFloat() {
24         assertEquals((int)(f=i), i);  }
25     @Test
26     public  void testElargissementlongToDouble() {
27         assertEquals((long)(d=l), l); }
28 }
```



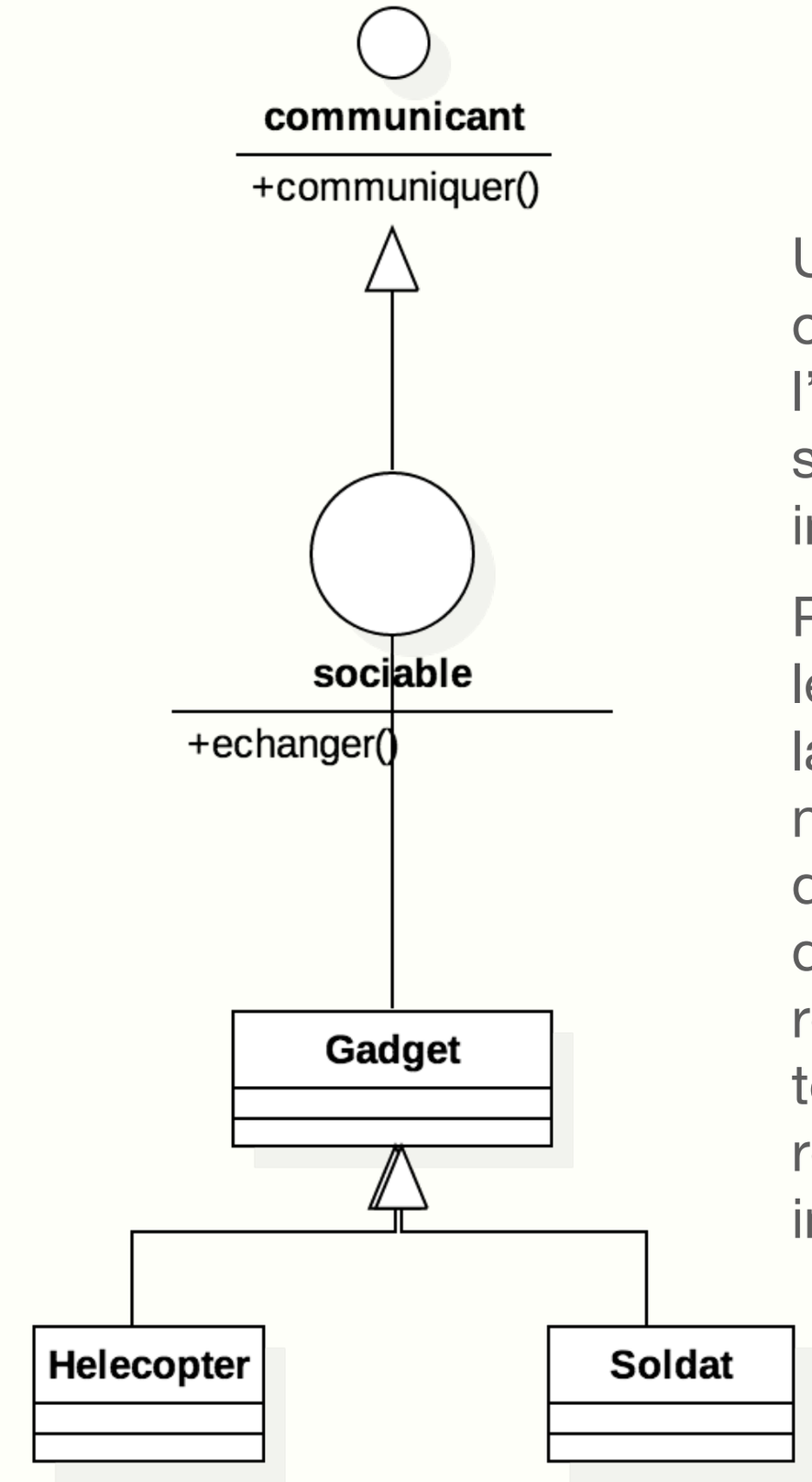
NB : Voir Section 5.1.2 de la Specification du language Java

TRANSTYPAGE EXPLICITE : RÉTRÉCISSEMENT TYPES PRIMITIFS

```
 1 package ma.ac.ensa;
 2 import static org.junit.Assert.assertEquals;
 5 public class Transtypage {
 6     @Test
 7     public  void test() {
 8         byte b=89;
 9         short s=25689;
10         int i=256000089;
11         long l=2560000000000000891;
12         assertEquals(b, (byte)i);
13         assertEquals(b, (byte)l);
14         assertEquals(b, (byte)i);
15         assertEquals(b, (byte)s);
16         l=i=s=b;
17         assertEquals(b, i);
18         assertEquals(b, l);
19         assertEquals(s, i);
20     }
21 }
```



TRANSTYPAGE IMPLICITE : ELARGISSEMENT TYPES RÉFÉRENCES

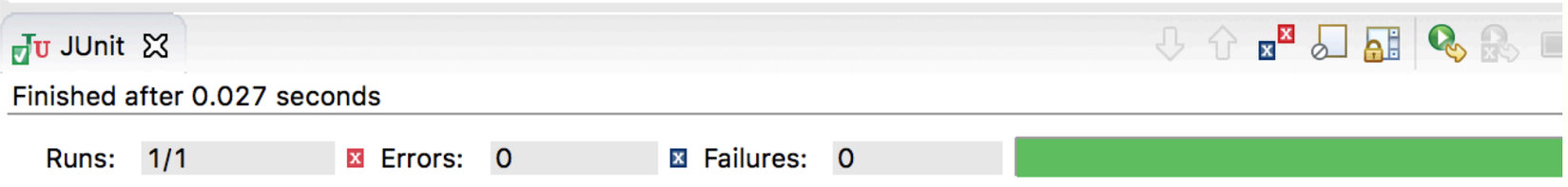


Une référence à un objet peut référencer l'un des trois types suivants: classe, interface ou tableau.

Pour les type référence, le type devient plus large lorsque vous naviguez vers le haut dans la hiérarchie d'héritage. C'est une règle générique pour tout les trois type de références.(classe, interface, tableau)

TRANSTYPAGE IMPLICITE : ELARGISSEMENT
TYPES RÉFÉRENCES

```
 6 public class TestTranstypage{
 7     @Test
 8     public void typeClasse() {
 9         SuperClasse SuperC=new SouClasse();
10         assertEquals(SuperC.fct(), "SousClasse");
11     }   }
12  class SuperClasse{
13 public   byte n=10;
14 public String fct() {return "SuperClasse";}
15 }
16
17  class SouClasse extends SuperClasse {
18 public   double d=3.8;
19 public String fct() {return "SousClasse";}
20 }
```



SuperClasse su=new **SouClasse()**;

Verification Statique :

Le compilateur vérifie que les méthodes et attributs invoqués par la variable **su** sont déclarés dans le type statique.

Par exemple l'instruction : su.d=34; génère une **erreur de compilation**

Liaison Dynamique :

Les méthodes invoquées sur une variable sont liées au type dynamique.

TRANSTYPAGE EXPLICITE : RÉTRÉCISSEMENT TYPES RÉFÉRENCES

```
3 import static org.junit.Assert.assertEquals;
6 public class TestTranstypageExplicit{
7     @Test
8     public void typeClasse() {
9         I1 i;
10    if(Math.random()>0.5)    i=new SouClasse();
11    else i=new SuperClasse();
12    SouClasse c=(SouClasse)i;
13    assertEquals(c.d, 38l);
14    }   }
15
16 class SuperClasse implements I1{
17 public  byte n=10;
18 public String fct() {return "SuperClasse";}
19 }
20
21  class SouClasse extends SuperClasse {
22 public  long d=38l;
23 public String fct() {return "SousClasse";}
24 }
25
26 interface I1{
27     String fct();
28 }
```

Verification statique :

Lors du transtypage explicite entre deux types references A et B : le compilateur vérifie que :

1- A et B sont liées par une relation d'heritage.

2- A est une interface et B une classe non finale.

3-Le transtypage entre deux tableaux de type primitifs différents est interdit.

4-Le transtypage entre deux tableaux de type référence A[] et B[] est possible si A et B acceptent le transtypage explicite.

Verification dynamique :

B b;

A a=(A)b;

le type dynamique de B doit être un sous type de A.

CONCLUSION

Dans ce chapitre on a détaillé les points suivants :

- Déclaration des classes
- différence entre variables d'instance et variables de classe
- Déclaration des méthodes
- Déclaration des constructeurs
- Instanciation des objets (Opérateur new)
- Surcharge de méthodes et constructeurs
- Redéfinition des méthodes
- Transtypage implicite et explicite entre types primitifs et références