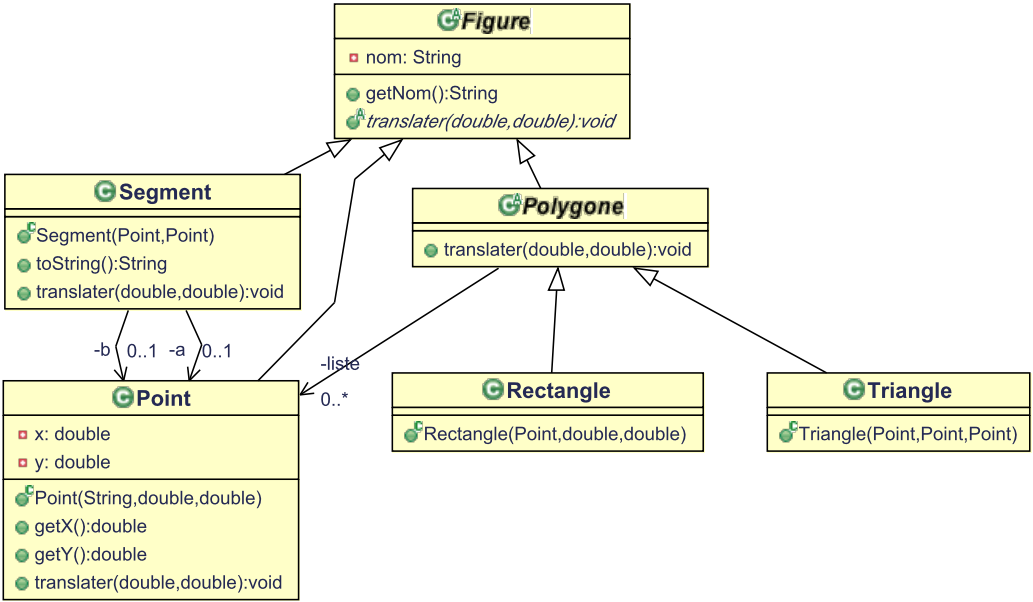
LU2IN002
CONVERSION DES TYPES DE VARIABLE (CAST...) TESTS DYNAMIQUES (INSTANCEOF)

Vincent Guigue & Christophe Marsala

SCIENCES SORBONNE UNIVERSITÉ
LIP6
MLIA Machine Learning & Deep Learning for Information Access

Exemple du logiciel de dessin



Cas amusant:

le type des instances est parfois (souvent) inconnu

Exemple:

```
1 Figure f;
2 if(Math.random()>0.5)
3   f = new Point(2,3);
4 else
5   f = new Segment(new Point(1,2), new Point(5,3));
6
7 // pour le compilateur f est de type Figure
8 // pour la JVM, quel est le type de l'instance f ???
```

Récupérer le type d'une instance dynamiquement

- Les limites du polymorphisme sont imposées par le compilateur qui vérifie (statiquement) le type des variables

1) Contournement = connaitre le type des instances en cours de programme (dynamiquement)
2) Revenir au type de l'instance pour accéder aux méthodes spécifiques = faire un cast sur la variable

Exemple:

```
1 Figure f;
2 if(Math.random()>0.5)
3   f = new Point(2,3);
4 else
5   f = new Segment(new Point(1,2), new Point(5,3));
6
7 // pour le compilateur f est de type Figure
8 // pour la JVM, quel est le type de l'instance f ???
```

variable **instanceof** *NomClasse*

⇒ retourne un `boolean`

```
1  Figure f;
2  if(Math.random()>0.5)
3    f = new Point(2,3);
4  else
5    f = new Segment(new Point(1,2), new Point(5,3));
6
7  if(f instanceof Point)
8     System.out.println("C'est un Point");
9  else
10    System.out.println("C'est un Segment");
```

- Syntaxe particulière
- Souvent, il existe d'autre moyen de faire...
  Globalement, sauf exception:

`instanceof` = mauvaise programmation

Procédure qui spécialise ses traitements par type de figure

```
1  public static void afficheType(Figure f) {
2    if(f instanceof Point)
3        System.out.println("C'est un Point");
4    else if(f instanceof Segment)
5        System.out.println("C'est un Point");
6    else if(f instanceof Rectangle)
7        System.out.println("C'est un Rectangle");
8      // etc ... un cas par figure !
```

⇒ Très mauvais... Mais pourquoi ?

Procédure qui spécialise ses traitements par type de figure

```
1  public static void afficheType(Figure f) {
2    if(f instanceof Point)
3        System.out.println("C'est un Point");
4    else if(f instanceof Segment)
5        System.out.println("C'est un Point");
6    else if(f instanceof Rectangle)
7        System.out.println("C'est un Rectangle");
8      // etc ... un cas par figure !
```

⇒ Très mauvais... Mais pourquoi ?

Que se passe-t-il si on ajoute un nouveau type de Figure ?

Il faut toucher au code utilisateur !!!

⇒ l'outil existe, mais souvent, il faut mieux **ne pas l'utiliser**!

Code spécifique dans les classes:

- dans Figure :

```
1 public String getTypeFigure() {};
```

- dans Point :

```
1 public String getTypeFigure() { return "Point"; }
```

- dans Rectangle :

```
1 public String getTypeFigure() { return "Rectangle"; }
```

- Code générique (éventuellement en dehors des classes):

```
1   public static void afficheType(Figure f) {
2     System.out.println("C'est␣un␣"+f.getTypeFigure());
3   }
```

instanceof: attention à la hierarchie

```java
public static void main(String[] args) {
    Point p1 = new Point("toto", 0, 2);
    Point p2 = new Point( "toto 2", 3, 2);
    Figure f = new Segment(p1, p2);

    if(f instanceof Point)
        System.out.println("f est un Point");
    if(f instanceof Segment)
        System.out.println("f est un Segment");
    if(f instanceof Figure)
        System.out.println("f est une Figure");
}
```

Le programme suivant retourne:

```
f est un Segment
f est une Figure
```

Le résultat est logique: il faut comprendre `instanceof` comme «EST UN?»

public Class getClass()

- Méthode de `Object`
- S'utilise sur une instance (syntaxe différente de `instanceof`)
- Retourne la classe (complète) de l'instance
  - Unique pour une classe donnée
  - Accès à différents descripteurs de la classe

```
1 Figure f = new Segment(p1, p2);
2 System.out.println("f est de type : "+f.getClass());
3 // retour :
4 // f est de type : class cours2.Segment
```

Usage classique pour comparer le type de deux instances:

```
1 // soit deux Objets obj1 et obj2
2 if (obj1.getClass() != obj2.getClass())
3   ...
```

Cast = 2 modes de fonctionnement

- Conversion sur les types basiques: le codage des données change. Souvent implicite dans votre codage...

```
1 double d = 1.4;
2 int i = (int) d;  // i=1
```

- Conversion dans les hiérarchies de classes:

la variable est modifiée, l'instance est inchangée

```
1 Figure f = new Segment(p1, p2);
2 Segment s = (Segment) f;
3 // Pour verifier que vous avez compris:
4 // donner un diagramme memoire
```

- Utile pour accéder aux méthodes spécifiques d'une instance
- Dangereux: aucun contrôle du compilateur...

Un système peu sécurisé à la compilation:

```
1 Point p1 = new Point("toto", 0, 2);
2 Point p2 = new Point( "toto␣2", 3, 2);
3 Figure f = new Segment(p1, p2);
4
5 Figure f2 = (Figure) p1;
```

Un système peu sécurisé à la compilation:

```
1 Point p1 = new Point("toto", 0, 2);
2 Point p2 = new Point( "toto␣2", 3, 2);
3 Figure f = new Segment(p1, p2);
4
5 Figure f2 = (Figure) p1; // OK mais inutile
6 Figure f3 = p1; // OK Subsomption classique
7 Segment s = (Segment) f;
```

Un système peu sécurisé à la compilation:

```
1 Point p1 = new Point("toto", 0, 2);
2 Point p2 = new Point( "toto␣2", 3, 2);
3 Figure f = new Segment(p1, p2);
4
5 Figure f2 = (Figure) p1; // OK mais inutile
6 Figure f3 = p1; // OK Subsomption classique
7 Segment s = (Segment) f; // compilation OK
8 Point p3 = (Point) f;
```

Un système peu sécurisé à la compilation:

```
1 Point p1 = new Point("toto", 0, 2);
2 Point p2 = new Point( "toto␣2", 3, 2);
3 Figure f = new Segment(p1, p2);
4
5 Figure f2 = (Figure) p1; // OK mais inutile
6 Figure f3 = p1; // OK Subsomption classique
7 Segment s = (Segment) f; // compilation OK
8 Point p3 = (Point) f; // compilation OK (!)
```

Exécution:

Crash du programme avec le message suivant

```
1 Exception in thread "main" java.lang.ClassCastException:
2 cours2.Segment cannot be cast to cours2.Point
3         at cours2.Test.main(Test.java:26)
```

Cast: avec plusieurs niveaux de hiérarchie

```
1  // subsomption
2  Figure f = new Segment(p1, p2);
3  Figure f2 = new Carre(p1, cote);
4  Figure f3 = new Triangle(p1, p2, p3);;
5  Polygone p = new Triangle(p4, p5, p6);
6
7  // cast OK
8  Triangle t = (Triangle) p; // OK (comme precedemment)
9
10 Polygone p2 = (Polygone) f2; // OK compil + exec:
11                             // un Carre EST UN Polygone
12
13 Polygone p3 = (Triangle) f3; // OK:
14                     // f3 est un Triangle => conversion OK (JVM)
15                     // p3 peut referencer un Triangle OK (compil.)
16
```

Cast: avec plusieurs niveaux de hiérarchie

```
1  // subsomption
2  Figure f = new Segment(p1, p2);
3  Figure f2 = new Carre(p1, cote);
4  Figure f3 = new Triangle(p1, p2, p3);;
5  Polygone p = new Triangle(p4, p5, p6);
6
7  // cast OK
8  Triangle t = (Triangle) p; // OK (comme precedemment)
9
10 Polygone p2 = (Polygone) f2; // OK compil + exec:
11                             // un Carre EST UN Polygone
12
13 Polygone p3 = (Triangle) f3; // OK:
14                     // f3 est un Triangle => conversion OK (JVM)
15                     // p3 peut referencer un Triangle OK (compil.)
16
17 // cast KO
18 Carre c = (Carre) f;    // KO JVM
19 Cercle c = (Cercle) p; // KO Compil: opération impossible
```

Idée

Vérifier le type de l'instance avant la conversion

```
1        Figure f = new Segment(p1, p2);
2        Segment s;
3        if(f instanceof Segment)
4            s = (Segment) f;
```

$\Rightarrow$ vous utiliserez systématiquement cette sécurisation

Idée

Vérifier le type de l'instance avant la conversion

```
1        Figure f = new Segment(p1, p2);
2        Segment s;
3        if(f instanceof Segment)
4            s = (Segment) f;
```

$\Rightarrow$ vous utiliserez systématiquement cette sécurisation

Que penser de:

```
1        // Figure -> Polygone -> Carre
2        Figure f = new Carre(cote);
3        Polygone p;
4        if(f instanceof Polygone)
5            p = (Polygone) f;
```

- Est ce que ça marche (compil+exec)?
- Est ce que ça peut être utile?

fonction equals

Il y a toujours un cast (sécurisé) dans equals pour pouvoir accéder aux attributs à comparer

Exemple sur la classe Point:

```
1   public boolean equals(Object obj) { // V1
2       if (this == obj)
3           return true;
4       if (obj == null)
5           return false;
6       if (getClass() != obj.getClass())
7           return false;
8       Point other = (Point) obj;
9       if (x != other.x)
10          return false;
11      if (y != other.y)
12          return false;
13      return true;
14  }
```

Imaginons la redéfinition suivante pour `equals`:

```
1    public boolean equals(Object obj) {// V2
2        if (this == obj)
3            return true;
4        if (obj == null)
5            return false;
6        // if (getClass() != obj.getClass()) en V1
7        if (!(obj instanceof Point))
8            return false;
9        Point other = (Point) obj;
10       if (x != other.x)
11           return false;
12       if (y != other.y)
13           return false;
14       return true;
15   }
```

Quel est le défaut de l'implémentation v2?

Prise en défaut:

```
1 Point p = new Point(1,2);
2 PointTrafique p2 = new PointTrafique("toto",1,2);
3 if(p.equals(p2))
4             System.out.println("ils␣sont␣egaux␣!!!␣");
5 if(p2.equals(p))
6             System.out.println("et␣ici␣???");
```

Avec :

```
1 public class PointTrafique extends Point{
2     private String qqch;
3     public PointTrafique(String nom, double x, double y) {
4         super( x, y);
5         qqch = nom; // un attribut en plus
6     }
7 }
```

V1: pas d'égalité

V2: égalité détectée... Est ce légitime?
Pb: quid de la symétrie?