Διάλεξη 16-17: Πολυμορφισμός (Polymorphism)

Στην ενότητα αυτή θα μελετηθούν τα εξής επιμέρους θέματα:

- Υπερφόρτωση (Overloading), Μεθόδων (Method Overloading), Τελεστών (Operator Overloading (C++, C#))
- Υπερσκέλιση Μεθόδων (Method Overriding)
- Δυναμική Πρόσδεση (Late (Dynamic) Binding)
- Upcasting/Downcasting

Διδάσκων: Παναγιώτης Ανδρέου

Πολυμορφισμός (polymorphism)

- **Πολύ:** πολλές, πολλαπλές, …
- **Μορφή:** χαρακτήρας, εμφάνιση, απεικόνιση, …

Πολλαπλές μορφές

- Ο πολυμορφισμός είναι **μία από τις πιο βασικές έννοιες του αντικειμενοστρεφή προγραμματισμού.**
- Σχετίζεται με την **αποσύνδεση των μεθόδων από τους τύπους**

Είδη Πολυμορφισμού

- **Υπερφόρτωση (Overloading)**
  - Μεθόδων (Method Overloading)
  - Τελεστών (Operator Overloading (C++, C#))
- **Υπερσκέλιση Μεθόδων (Method Overriding)**
- **Δυναμική Πρόσδεση (Late (Dynamic) Binding)**
- **Upcasting/Downcasting**

Υπερφόρτωση Μεθόδων (Method Overloading)

- Παραδείγματα της πραγματικής ζωής:
  - **Παίζω** μπάλα
  - **Παίζω** το πιάνο
  - **Παίζω** ένα DVD
  - **Παίζω** με ένα παιχνίδι

  Η λέξη **“Παίζω”** χρησιμοποιείται με διαφορετική έννοια

- Παραδείγματα του προγραμματισμού:
  - **Constructor Overloading**
  - **Method Overloading**

Παράδειγμα Method Overloading

```
public static void method() { … }
public static void method(int x) { … }
public static void method(int x, String y) {
… }
```

Constructor Overloading

```
class Circle {
  double radius;
  Circle () { this (1.0); }
  Circle (double newRadius) {
     radius = newRadius; }
}
```

Υπερφόρτωση Τελεστών (Operator Overloading)

- Παραδείγματα της πραγματικής ζωής:
  - **Προσθέτω** δύο αριθμούς
  - **Προσθέτω** στη σακούλα μου ένα μήλο και ένα αχλάδι
  - **Προσθέτω** ζάχαρη στον καφέ μου
  - **Προσθέτω** ξύλα στο τζάκι

  Η έννοια της **"πρόσθεσης"** είναι αόριστη εκτός και αν είναι συσχετισμένη με κάποια συμφραζόμενα

- Προσοχή: Αν και οι τελεστές είναι συχνά συνδεδεμένοι με μαθηματικές πράξεις, δεν είναι τίποτα άλλο από καλέσματα μεθόδων

Παράδειγμα Operator Overloading (C#)

Υποθέστε κλάση MyObj με μεταβλητές int a, int b

```
public static MyObj operator + (MyObj x, MyObj y) {
      return new MyObj(x.a+y.a, x.b+y.b);
}
```

```
public static MyObj operator == (MyObj x, MyObj y) {
      if (x.a == y.a && x.b == y.b) return true;
      return false;
}
```

```
public static MyObj operator != (MyObj x, MyObj y) {
      if (x.a != y.a || x.b != y.b) return true;
      return false;
}
```

Υπερσκέλιση (Override) vs.Υπερφόρτωση (Overload)

- Η Υπερφόρτωση (overload) μπορεί να πραγματοποιηθεί:
  - είτε στην ίδια κλάση με ορισμό μεθόδων με το ίδιο όνομα
  - ή μέσω κληρονομικότητας με ορισμό μεθόδων με το ίδιο όνομα

- **Η Υπερσκέλιση (Overriding) μπορεί να συμβεί μόνο μέσω κληρονομικότητας**

- Οι ακόλουθες δηλώσεις **μεθόδων ΔΕΝ μπορούν υπερσκελιθούν:**
  - **private**
  - **static (εκτός και αν…)**
  - **final**

Παράδειγμα Overloading vs. Overriding

```
public class Test {
  public static void main(String[] args) {
    A a = new A();
    a.p(10);
    a.p(10.0);
  }
}

class B {
  public void p(double i) {
    System.out.println(i * 2);
  }
}

class A extends B {
  // This method overrides the method in B
  public void p(double i) {
    System.out.println(i);
  }
}
```

```
public class Test {
  public static void main(String[] args) {
    A a = new A();
    a.p(10);
    a.p(10.0);
  }
}

class B {
  public void p(double i) {
    System.out.println(i * 2);
  }
}

class A extends B {
  // This method overloads the method in B
  public void p(int i) {
    System.out.println(i);
  }
}
```

- Όταν γίνεται χρήση override πως μπορούμε να έχουμε πρόσβαση στην μέθοδο της υπερκλάσης;;;
- Μόνο μέσω του super (κάλεσμα μέσα από άλλη μέθοδο);

Overriding: Αποδυνάμωση/Ενδυνάμωση

- Μία υποκλάση δεν μπορεί να αποδυναμώσει την πρόσβαση σε μία μέθοδο που γίνεται override
- Μπορεί όμως να την ενδυναμώσει

- Παράδειγμα:

```
class parent {
  //package-level
  void m(){ }
}
```

```
class child extends parent {
  void m(){ }              //OK
  protected void m(){ } //OK
  public void m(){ }     //OK
  private void m(){ }    //NOT OK
}
```

Overriding static, final methods

- Μία υποκλάση **δεν μπορεί να κάνει override** μία **static method εκτός και αν είναι static**. Γιατί;;;;;;;;;;;;;;;

```
class parent {
  //package-level
  static void m(){ }
}
class child extends parent {
  static void m(){ }  //OK
  void m(){ }         //NOT OK
}
```

Η πρόσδεση στατικών μεθόδων και μεταβλητών γίνεται πριν την έναρξη του προγράμματος
➔early (static) binding

- **Μία υποκλάση δεν μπορεί να κάνει override μία final method.**

Πρόσδεση (Binding)

- **Πρόσδεση (Binding) συμβαίνει όταν:**
  - Συνδέεται μία μεταβλητή με μία τιμή
  - Συνδέεται το κάλεσμα μίας μεθόδου με υλοποίησή της
- Δύο είδη Πρόσδεσης
  - **Early (static) Binding:** συμβαίνει πριν να τρέξει το πρόγραμμα από τον μεταγλωττιστή ή linker. Η compilers της C υποστηρίζουν μόνο αυτό.
  - **Late (dynamic) Binding:** συμβαίνει όταν η πρόσδεση γίνεται κατά τη διάρκεια εκτέλεσης του προγράμματος

Late (Dynamic) Binding

- Ο μεταγλωττιστής δεν γνωρίζει εκ' των προτέρων ποιος τύπος θα χρησιμοποιηθεί
- Για να πραγματοποιηθεί το late binding πρέπει να υπάρχει κάποιος μηχανισμός που καθορίζει τον τύπου του αντικειμένου και την μέθοδο που πρέπει να καλεστεί την συγκεκριμένη στιγμή εκτέλεσης

- **Η JAVA πραγματοποιεί όλα τα method bindings με late binding**

- **Εξαίρεση:** αν οι μεθόδοι είναι δηλωμένες σαν static ή final
- **Προσοχή:** οι μεθόδοι που είναι δηλωμένες σαν private είναι εμμέσως final

Πολυμορφισμός και Late (Dynamic) Binding

- Παραδείγματα της πραγματικής ζωής:
  - Διάβασε **το**
  - Εξέτασε **το**
  - Τύπωσε **το**
  - Αντίγραψέ **το**

  Η έννοια του αντικείμενου **“το”** είναι αόριστη εκτός και αν είναι συσχετισμένη με κάποια συμφραζόμενα

- Ο πολυμορφισμός δεν έχει σχέση τόσο με τα χαρακτηριστικά (μεταβλητές) της κλάσης όσο με τη συμπεριφορά (μεθόδους) της

Παράδειγμα: Late (Dynamic) Binding

```
public class PolymorphismDemo {
  public static void m (Object x) {
    System.out.println(x.toString());}
  public static void main(String[] args) {
    m( new GraduateStudent() );
    m( new Student() );
    m( new Person() );
    m( new Object() );
  }
}
class GraduateStudent extends Student { }
class Student extends Person {
  public String toString() { return "Student"; }
}
class Person extends Object {
  public String toString() {return "Person"; }
}
```

Η μέθοδος m() παίρνει σαν παράμετρο Object

➔Μπορεί να καλεστεί με οποιοδήποτε αντικείμενο αφού όλα τα αντικείμενα κληρωνόμουν από το Object

Παράδειγμα: Late (Dynamic) Binding

```
public class PolymorphismDemo {
  public static void m (Object x)   {
    System.out.println(x.toString());}
  public static void main(String[] args) {
    m( new GraduateStudent() );
    m( new Student() );
    m( new Person() );
    m( new Object() );
  }
}
class GraduateStudent extends Student { }
class Student extends Person {
  public String toString() { return "Student"; }
}
class Person extends Object {
  public String toString() {return "Person"; }
}
```

Η μέθοδος m() μπορεί να καλεστεί με οποιοδήποτε αντικείμενο κληρονομεί από το αντικείμενο που δέχεται σαν παράμετρος

➔ **πολυμορφισμός**

Παράδειγμα: Late (Dynamic) Binding

```
public class PolymorphismDemo {
  public static void m (Object x)   {
    System.out.println(x.toString());}
  public static void main(String[] args) {
    m( new GraduateStudent() );
    m( new Student() );
    m( new Person() );
    m( new Object() );
  }
}
class GraduateStudent extends Student { }
class Student extends Person {
  public String toString() { return "Student"; }
}
class Person extends Object {
  public String toString() {return "Person"; }
}
```

Όταν εκτελείτε η μέθοδος m() το αντικείμενο μπορεί να είναι του τύπου:

- **GraduateStudent ή**
- **Student ή**
- **Person ή**
- **Object**

Η κάθε κλάση έχει δική της υλοποίηση για τη μέθοδο toString()

Ποια θα καλεστεί; Αποφασίζεται από την JVM run-time

Late (Dynamic) Binding

- Έστω ότι ένα αντικείμενο ο είναι ένα στιγμιότυπο της κλάσης C_1, C_2,… C_n.

- Αν το αντικείμενο ο καλέσει την μέθοδο p() το JVM ψάχνει την υλοποίηση της μεθόδου p() στις κλάσεις C_1, C_2,… C_n με την σειρά μέχρι να τη βρει.

- Η πρώτη εκτέλεση που θα βρεθεί, θα εκτελεστεί

Ταίριασμα(Match) vs. Πρόσδεση(Binding) Μεθόδων

- Το ταίριασμα μία υπογραφής μεθόδου γίνεται σε επίπεδο μεταγλώττισης σύμφωνα με το όνομα της μεθόδου, τη λίστα των παραμέτρων και το είδος των παραμέτρων

- Η μέθοδος μπορεί να υλοποιείται από πολλές υποκλάσεις

- Η JVM προσδέτει δυναμικά την υλοποίηση της μεθόδου κατά τη διάρκεια εκτέλεσης του προγράμματος

Μετατροπές αντικειμένων (Casting Objects)

- Το casting μπορεί να χρησιμοποιηθεί για να μετατραπεί ένα αντικείμενο από ένα τύπο σε κάποιο άλλο της ιεραρχία της κληρονομικότητας

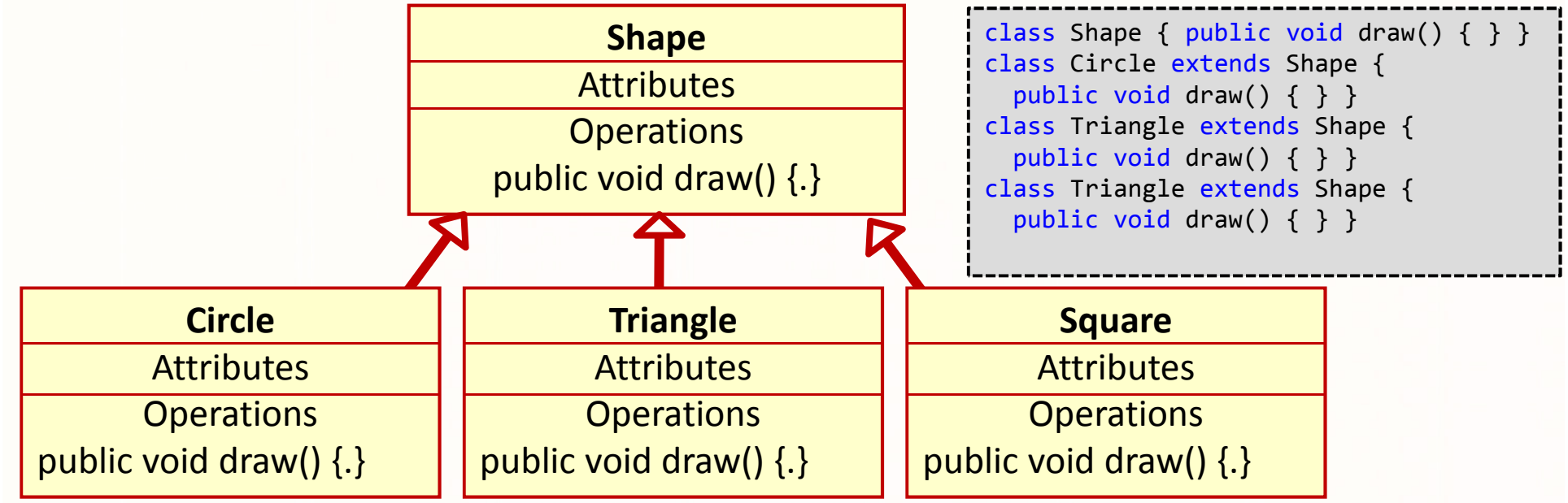
- Παράδειγμα: `m( new Student() );`
  μετατρέπει το Student σε Object

- Το πιο πάνω είναι ισοδύναμο με

```
Object o = new Student();
m( o );
```

- Έχουμε μετατρέψει ένα αντικείμενο μίας υποκλάσης σε ένα αντικείμενο μίας υπερκλάσης ➔ upcasting

Χωρίς Πολυμορφισμό




```
class Shape { public void draw() { } }
class Circle extends Shape {
  public void draw() { } }
class Triangle extends Shape {
  public void draw() { } }
class Triangle extends Shape {
  public void draw() { } }
```

```
public class GUI {
  public static void
    erase( Circle c ) { }

  public static void
    erase( Triangle t ) { }

  public static void
    erase( Square s ) { }
```

Υλοποιήσαμε 3 διαφορετικές overloaded μεθόδους

```
  …
```

```
…
public static void main(String args[]) {
    Circle   c = new Circle();
    Triangle t = new Triangle();
    Square   s = new Square();

    erase( c );
    erase( t );
    erase( s );
}
```

Μειονεκτήματα απουσίας του Upcasting

- Χρειάζεται να γράψουμε 3 ειδικές μεθόδους erase
  ➔ περισσότερος κώδικας.
- Αν θελήσουμε να προσθέσουμε καινούριες μεθόδους (σαν την erase) ή καινούριες κλάσεις (τύπου Shape)
  ➔ περισσότερη προσπάθεια
- Μπορεί να μην υλοποιήσουμε μεθόδους για όλα τα είδη αντικειμένων
  ➔ γίνεται πιο εύκολα λογικό λάθος

- Το upcasting (πολυμορφισμός) μας επιτρέπουν να ορίσουμε μια μέθοδο, στην κλάση-βάσης, και αυτή τη μέθοδο να την χρησιμοποιήσουμε και σε αντικείμενα κλάσεων-κληρονόμων.

Με Πολυμορφισμό (Upcasting)

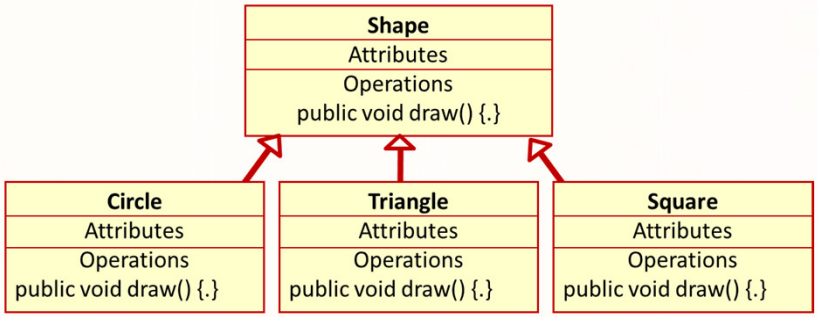
```
public class GUI {


  public static void
    erase( Shape s ) { }


  public static void
    main(String args[]) {

    Circle   c = new Circle();
    Triangle t = new Triangle();
    Square   s = new Square();

    erase( c );
    erase( t );
    erase( s );
}
```

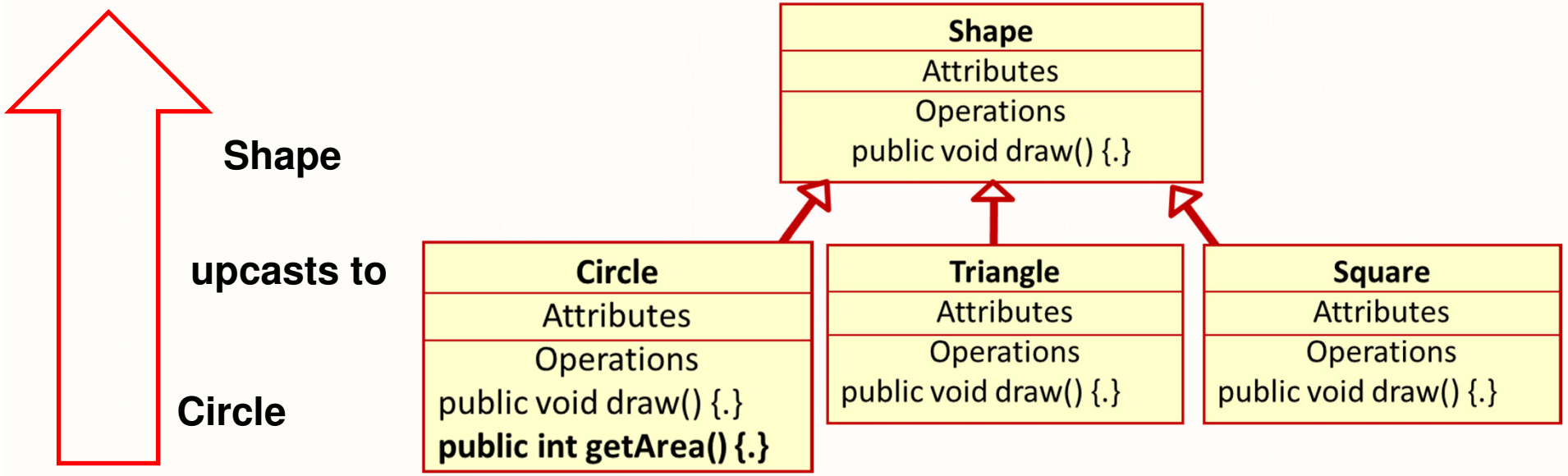


Υλοποιήσαμε 1 μέθοδο μόνο για όλους τις κληρονόμους της κλάσης Shape

Το JVM δεν γνωρίζει κατά το compile time τι χειριστήριο θα περάσει στην μέθοδο erase Καθορίζεται στην φάση της εκτέλεσης (late binding)

Upcasting

- `Shape s = new Circle ();`



- Μέσα στο heap δημιουργείται ένα καινούριο αντικείμενο Circle (= new Circle).
- **Υπακούει όμως την διαπροσωπεία της κλάσης Shape**
- Αυτό «περιέχει» ένα αντικείμενο τύπου Shape
- `s.draw()`: late binding ➔πολυμορφική κλήση στην Circle.draw()
- s.getArea() ➔ **compile error (δεν υπακούει την διαπροσωπεία Shape)**

Upcasting με Abstract Classes/Interfaces

- Μία αφαιρετική κλάση μπορεί να χρησιμοποιηθεί σαν τύπος, π.χ., GeometricObject array = new GeometricObject[10]; (μέσω από πολυμορφισμό)
- Δείκτες σε αντικείμενα μπορούν να έχουν τύπο διαπροσωπείας (μέσω από πολυμορφισμό)
- Για παράδειγμα:
  `IAnimals a = new Lion();`
  - Το `IAnimals` είναι διαπροσωπεία
  - Το Lion είναι κλάση
- Τι έχουμε πετύχει με την πιο πάνω δήλωση;
- **Μόνο μεθόδοι που ανήκουν στην διαπροσωπεία `IAnimals` μπορούν να κληθούν!**

Down-Casting vs. Up-Casting

- Το πιο κάτω δημιουργεί σφάλμα μεταγλώττισης
  `Student s = new Object(); //downcasting`
- Αυτό `(Object o = new Student();)` όμως ΟΧΙ.
- Ο λόγος είναι ότι
  - Ένα αντικείμενο τύπου Student περιέχει/είναι ένα αντικείμενο τύπου Object
  - Το αντίστροφο όμως δεν ισχύει,
    π.χ., `Object o = new Apple(); //OK`
    `Student s = o;` σημαίνει ότι ο s είναι μήλο!
- Πρέπει να χρησιμοποιηθεί άμεσο/explicit casting για να επιτραπεί από τον μεταγλωττιστή
  π.χ., `Student s = (Student) new Object();`

Πολυμορφικές κλήσεις μέσω Κατασκευαστών

- Τι θα τυπωθεί;

```
abstract class Shape{
  abstract void draw();
  Shape(){
    System.out.println("Shape: before draw()");
    draw();
    System.out.println("Shape: after draw()");
  }
}
class Circle extends Shape{
  Circle(){
    System.out.println("Circle constructor");
  }
  void draw() { System.out.println("Circle: draw()"); }
}
public class PolymorphismConstructor {
  public static void main(String[] args) {
    Circle c = new Circle();
    Shape s = new Circle();
  }
}
```

Κλήση μία late binded abstract method μέσα στον κατασκευαστή της υπερκλάσης, χρησιμοποιεί την overrided method στην υποκλάση

Ο τελεστής <instanceof>

- Στην java υπάρχει ο τελεστής <instanceof> ο οποίος μας επιτρέπει να ελέγξουμε τον τύπο ενός αντικειμένου

- Επιστρέφει true ή false

- Σύνταξη: <object> instanceof <class>

- Παράδειγμα:

```
if( c instanceof child)
    System.out.println("child");
```

Παράδειγμα <instanceof>

- Τι θα τυπώσει ο ακόλουθος κώδικας;

```
class parent { }

public class child extends parent {

  public static void main(String[] args) {

    parent p = new child();

    if( p instanceof parent)
        System.out.println("parent");

    if( p instanceof child)
        System.out.println("child");
  }
}
```

Πολυμορφισμός και Covariant return types

- Τι θα τυπωθεί;

```
class ParentA {
  public String toString() {
    return "ParentA";
  }
}
class ChildA extends ParentA {
  public String toString() {
    return "ChildA";
  }
}
class ParentB {
  ParentA process() {
    return new ParentA();
  }
}
class ChildB extends ParentB {
  ChildA process() {
    return new ChildA();
  }
}
```

```
public class CovariantReturn {
  public static void main(String[] args) {
    ParentB pb = new ParentB();
    ParentA pa = pb.process();
    System.out.println(pa);
  }
}
```

`pb = new ChildB();` → **upcasting**

`pa = pb.process();` → **process() is overridden in ChildB, pa = new ChildA()**

`System.out.println(pa);` → **toString() is overridden in ParentA by ChildA**

Πολυμορφισμός και Σύνθεση

- Τι θα τυπωθεί;

```
class Actor {
  public void act() { }
}
class HappyActor extends Actor {
  public void act() {
   Sout.println("HappyActor");
  }
}
class SadActor extends Actor {
  public void act() {
    Sout.println("SadActor");
  }
}
class Stage {
  private Actor actor = new HappyActor();
  public void change() {
    actor = new SadActor();
  }
  public void performPlay() {
    actor.act();
  }
}
```

Composition

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

    Stage stage=new Stage();

    stage.performPlay();

    stage.change();

    stage.performPlay();

  }

}
```