Αντικειμενοστρεφής Προγραμματισμός

Παναγιώτης Αδαμίδης
adamidis@it.teithe.gr

"Υπέρβαση, Upcasting, final (χαρακτηριστικά, μέθοδοι, κλάσεις)"

Υπερφόρτωση - Υπέρβαση

- Η κλάση **Y** κληρονομεί την κλάση **X** (υπερκλάση). Ένα αντικείμενο **y** της κλάσης **Y** είναι «ουσιωδώς» και ένα αντικείμενο της κλάσης **X** με περισσότερα δεδομένα και λειτουργικότητα, αλλά όχι ακριβώς.
- Εάν και οι δύο κλάσεις δηλώσουν μία μέθοδο g() με την ίδια υπογραφή, η **y.g()** θα καλέσει την μέθοδο που δηλώθηκε στην κλάση **Y** και όχι αυτή που δηλώθηκε στην **X**. Σε αυτή την περίπτωση η μέθοδος **y.g()** υπερβαίνει (overrides) την **x.g()** .
- Η υπέρβαση(overriding) είναι όμοια με την υπερφόρτωση (overloading): διαφορετικές μέθοδοι έχουν το ίδιο όνομα.
- Γιατί ο διαφορετικός όρος; Υπάρχουν διαφορές;

Υπέρβαση - Παράδειγμα (1)

```
class Employee { // Represents one Employee
   protected String name;
   protected String address;
   protected String phoneNumber;
   protected String id;
   protected double payRate;
   public Employee (String empName, String empAddress,
            String empPhone,  String empId, double empRate) {
      name=empName;
      address=empAddress;
      phoneNumber=empPhone;
      id=empId;
      payRate=empRate;
   }
   public double pay() {  return payRate;  }
   public void print() {
       System.out.println (name+ "  "+id);
       System.out.println (address);
       System.out.println (phoneNumber);
   }
} // class Employee
```

Υπέρβαση - Παράδειγμα (2)

```
class Executive extends Employee { // Represents one
// executive as an employee that can earn a bonus
   private double bonus;
   public Executive (String execName, String execAddress,
          String execPhone, String execId, double execRate) {
      super(execName, execAddress, execPhone, execId, execRate);
      bonus=0;  // bonus yet to be awarded
   }
   public void awardBonus (double execBonus){
      bonus=execBonus;
   }
   public double pay() {
      double paycheck = super.pay() + bonus;
      bonus=0;
      return paycheck;
   }
} // class Executive
```

Υπέρβαση - Παράδειγμα (3)

```
class Firm {
   public static void main(String[] args) {
      Executive nikos = new Executive("Νίκος", "Μ. Αλεξάνδρου 12",
                 "999888", "M 111111", 1923.07);
      Employee maria = new Employee("Μαρία", "Αργοναυτών 35",
                 "123456",  "P 151515", 846.15);
      Employee john = new Employee("Γιάννης", "Ιπποκράτους 29",
                 "987654", "T 222222", 769.23);
      john.print();
      System.out.println ("Πληρωμή: " + john.pay());
      System.out.println ();
      maria.print();
      System.out.println ("Πληρωμή: " + maria.pay());
      System.out.println ();
      nikos.print();
      nikos.awardBonus (2000);
      System.out.println ("Πληρωμή: " + nikos.pay());
      System.out.println ();
   } //main
} // class Firm
```

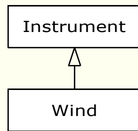
Upcasting

- Η πιο σημαντική προσφορά της κληρονομικότητας δεν είναι ότι παρέχει μεθόδους σε μια νέα κλάση, αλλά η σχέση της νέας με την βασική κλάση. Η νέα κλάση (υποκλάση) είναι ένας τύπος της υπάρχουσας κλάσης (υπερκλάση).
- Θεωρούμε την κλάση **Instrument** η οποία αναπαριστά μουσικά όργανα και την υποκλάση **Wind**. Η υποκλάση έχει όλες τις μεθόδους της υπερκλάσης. Έτσι μπορούμε να πούμε ότι ένα αντικείμενο **Wind**, είναι επίσης και τύπου **Instrument**.
- Μία αναφορά σε αντικείμενο κάποιας κλάσης μπορεί να χρησιμοποιηθεί για αναφορά σε αντικείμενο οιασδήποτε κλάσης η οποία την κληρονομεί.
- Στο παρακάτω παράδειγμα η μέθοδος **tune()** δέχεται αναφορά τύπου **Instrument**. Στην Wind.main() όμως καλείται με μία αναφορά τύπου **Wind**. Η μετατροπή της αναφοράς **Wind** σε **Instrument** ονομάζεται "upcasting".
- Η χρήση του "upcasting" είναι πάντα ασφαλής επειδή πηγαίνουμε σε ένα πιο γενικό τύπο.

Upcasting - Παράδειγμα

```
class Instrument {
    public void play() {}
    static void tune(Instrument i) {
        // ...
        i.play();
    }
}

class Wind extends Instrument {
    public static void main(String[] args) {
        Wind flute = new Wind();
        Instrument.tune(flute); // Upcasting
    }
}
```



Σύνθεση - Κληρονομικότητα

- Στον αντικειμενοστραφή προγραμματισμό το πιθανό είναι να χρησιμοποιούμε αντικείμενα κλάσεων οι οποίες απλά ενσωματώνουν στον κώδικά τους δεδομένα και μεθόδους. Επίσης θα χρησιμοποιήσουμε προϋπάρχουσες κλάσεις για να δημιουργήσουμε νέες με σύνθεση. Η κληρονομικότητα θα χρησιμοποιείτε πολύ λιγότερο, αντίθετα με την βαρύτητα που της δίνεται κατά την εκμάθηση.
- Ένας άλλος απλός τρόπος (εκτός του είναι-ένα ή έχει-ένα) επιλογής ανάμεσα σε σύνθεση και κληρονομικότητα είναι να αναρωτηθούμε αν θα χρειαστεί να χρησιμοποιήσουμε "upcasting" από την νέα κλάση στην υπερκλάση. Εάν χρειάζεται "upcasting" τότε η κληρονομικότητα είναι απαραίτητη, αλλιώς θα πρέπει να εξετάσουμε πιο προσεκτικά κατά πόσο χρειαζόμαστε κληρονομικότητα.

Πεδία "final"

```
class Value { int i = 1;}
public class FinalData {  // Can be compile-time constants
  final int i1 = 9;
  static final int VAL_TWO = 99;
  // Typical public constant:
  public static final int VAL_THREE = 39;
  // Cannot be compile-time constants:
  final int i4 = (int)(Math.random()*20);
  static final int i5 = (int)(Math.random()*20);
  Value v1 = new Value();
  final Value v2 = new Value();
  static final Value v3 = new Value();
  final int[] a = { 1, 2, 3, 4, 5, 6 };
  public void print(String id) {
     System.out.println( id + ": " + "i4 = " + i4 + ", i5 = " + i5);
  }
```

συνεχίζεται

Πεδία "final"

```
  public static void main(String[] args) {
     FinalData fd1 = new FinalData();
     //! fd1.i1++;              // Error: can't change value
     fd1.v2.i++;               // Object isn't constant!
     fd1.v1 = new Value();     // OK -- not final
     for(int i = 0; i < fd1.a.length; i++)
     fd1.a[i]++;               // Object isn't constant!
     //! fd1.v2 = new Value();  // Error: Can't
     //! fd1.v3 = new Value();  // change reference
     //! fd1.a = new int[3];

     fd1.print("fd1");
     System.out.println("Creating new FinalData");
     FinalData fd2 = new FinalData();
     fd1.print("fd1");
     fd2.print("fd2");
  }
} // class FinalData
```

Ορίσματα "final"

```
class Gizmo {  public void spin() {} }
public class FinalArguments {
   void with(final Gizmo g) {
      //! g = new Gizmo();          // Illegal -- g is final
   }
   void without(Gizmo g) {
      g = new Gizmo();             // OK -- g is not final
      g.spin();
   }
   // void f(final int i) { i++; }     // Can't change
   // You can only read from a final primitive:
   int g(final int i) { return i + 1; }
   public static void main(String[] args) {
      FinalArguments bf = new FinalArguments();
      bf.without(null);
      bf.with(null);
   }
}
```

Μέθοδοι "final"

- Λόγοι:
  - Παρεμπόδιση υπέρβασης, δηλ. αλλαγής λειτουργίας της μεθόδου σε υποκλάσεις οι οποίες κληρονομούν την κλάση που περιέχει την "final" μέθοδο.
  - Inline calls
- Μέθοδοι οι οποίες είναι private είναι εμμέσως και final αφού δεν υπάρχει πρόσβαση σε αυτές και δεν μπορούμε να τις υπερβούμε.
- Εάν προσπαθήσουμε να υπερβούμε μία μέθοδο "private" φαίνεται να λειτουργεί, αλλά στην πραγματικότητα δημιουργείται μία νέα μέθοδος
- ΔΕΝ επιτρέπεται η δήλωση μεθόδου ως final και abstract. *(Προσοχή: Λάθος βιβλίου)*

Μέθοδοι "final" - Παράδειγμα

```
class WithFinals {
    // Identical to "private" alone:
   private final void f() { System.out.println("WithFinals.f()"); }
   // Also automatically "final":
   private void g() {  System.out.println("WithFinals.g()"); }
}

class OverridingPrivate extends WithFinals {
   private final void f() {  System.out.println("OverridingPrivate.f()"); }
   private void g() { System.out.println("OverridingPrivate.g()"); }
}

class OverridingPrivate2 extends OverridingPrivate {
   public final void f() { System.out.println("OverridingPrivate2.f()"); }
   public void g() {  System.out.println("OverridingPrivate2.g()"); }
}
```

Μέθοδοι "final" - Παράδειγμα

```
public class FinalOverridingIllusion {
    public static void main(String[] args) {
        OverridingPrivate2 op2 = new OverridingPrivate2();
        op2.f();
        op2.g();
        // You can upcast:
        OverridingPrivate op = op2;
        // But you can't call the methods:
        //! op.f();
        //! op.g();
        // Same here:
        WithFinals wf = op2;
        //! wf.f();
        //! wf.g();
    }
}
```

Final Classes

- **ΔΕΝ κληρονομούνται**

```
class SmallBrain {}
final class Dinosaur {
    int i = 7;    int j = 1;
    SmallBrain x = new SmallBrain();
    void f() {}
}
//! class Further extends Dinosaur {}
// Compile error: Cannot extend final class 'Dinosaur'
public class Jurassic {
    public static void main(String[] args) {
        Dinosaur n = new Dinosaur();
        n.f();
        n.i = 40;
        n.j++;
    }
}
```