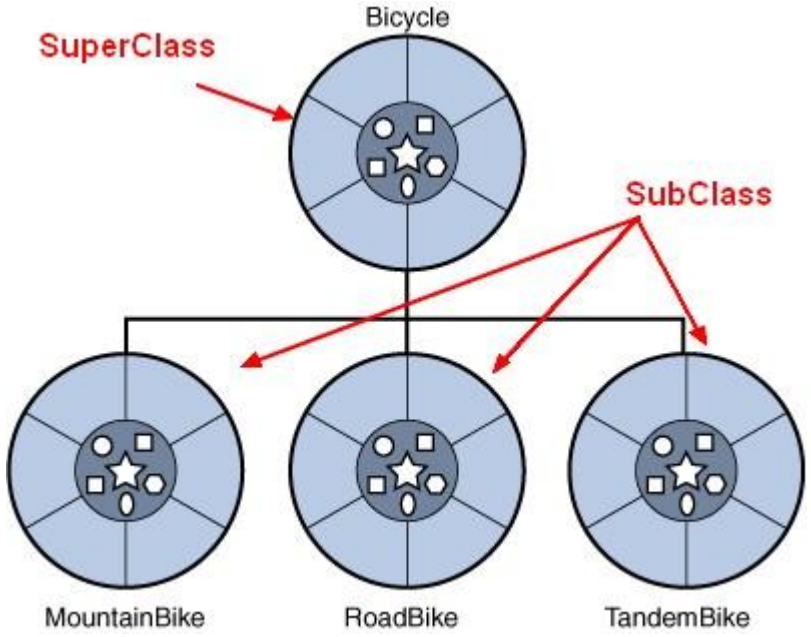
Κληρονομικότητα

Γιώργος Θάνος
gthanos@uth.gr
Γραφείο: Γ5/8, 3ος όροφος
Γκλαβάνη 37

Βασικό χαρακτηριστικό του αντικειμενοστραφούς προγραμματισμού είναι η δυνατότητα να παράγουμε νέες κλάσεις με βάση υφιστάμενες, εξειδικεύοντας και επεκτείνοντας τα χαρακτηριστικά τους.

Η διαδικασία επέκτασης των υφιστάμενων κλάσεων σε νέες, ειδικότερες κλάσεις ονομάζεται **κληρονομικότητα**.



Παράδειγμα Κληρονομικότητας

```
public class Bicycle {

private int cadence;
private int gear;
private int speed;

public Bicycle(int startCadence, int startSpeed,
int startGear) {
   gear = startGear;
   cadence = startCadence;
   speed = startSpeed;
}

public int getCadence() { return cadence;}
public int getGear() { return gear; }
public int getSpeed() { return speed; }
public void setCadence(int newVal) { cadence = newVal; }
public void setGear(int newVal) { gear = newVal; }
public void applyBrake(int dec) { speed -= dec; }
public void speedUp(int incr) { speed += incr; }

}
```

```
public class MountainBike extends Bicycle {

   private int seatHeight;

   public MountainBike(int startHeight,
.                      int startCadence,
.                      int startSpeed,
.                      int startGear) {
       super(startCadence, startSpeed, startGear);
       seatHeight = startHeight;
    }


  public void setSeatHeight(int newValue) {
     seatHeight = newValue;
  }

  public int getSeatHeight() {
      return seatHeight;
  }

}
```

Τι μπορούμε να κάνουμε σε μία υποκλάση...

Να χρησιμοποιήσουμε τα πεδία της γονικής κλάσης στα οποία έχουμε πρόσβαση (public, protected, package-private στο ίδιο package).

Να ορίσουμε νέα πεδία.

Να χρησιμοποιήσουμε τις μεθόδους της γονικής κλάσης στις οποίες έχουμε πρόσβαση (public, protected, package-private στο ίδιο package).

Μπορούμε να γράψουμε νέες στατικές ή μη στατικές μεθόδους για τη υποκλάση.

Μπορούμε να γράψουμε νέες μεθόδους που έχουν το ίδιο signature (ίδιο όνομα, ίδιο αριθμό και ίδιο τύπο ορισμάτων), ώστε να επαναορίσουμε (override) τις μεθόδους αυτές στην υποκλάση. Η πρακτική αυτή είναι συνήθης.

Μπορούμε να γράψουμε νέες στατικές (static) μεθόδους που έχουν το ίδιο signature, ώστε να επαναορίσουμε (override) τις μεθόδους αυτές στην υποκλάση.

Μπορούμε να γράψουμε κατασκευαστές της υποκλάσης που χρησιμοποιούν κατασκευαστές της γονικής κλάσης.

Κριτήρια χρήσης της κληρονομικότητας

Ας επανέλθουμε στο παράδειγμα του ορθογωνίου παραλληλογράμμου το οποίο έχουμε συναντήσει αρκετές φορές στο παρελθόν και ας προσπαθήσουμε να δημιουργήσουμε την κλάση ενός κυβοειδούς με χρήση της υφιστάμενης κλάσης του ορθογωνίου παραλληλογράμμου **Rectangle**.

Θα πρέπει να αποφασίσουμε εάν θέλουμε να δημιουργήσουμε την κλάση του κυβοειδούς (**Cuboid**)

- χρησιμοποιώντας ως πεδίο της νέας κλάσης ένα αντικείμενο της κλάσης **Rectangle** ή
- επεκτείνοντας την κλάση **Rectangle**, ώστε να προκύψει το κυβοειδές.

Πεδίο της νέας κλάσης ένα αντικείμενο της κλάσης Rectangle

```
public class Cuboid {
  Rectangle rec;
  int length;
  public Cuboid(int l, int w, int h) {
    rec = new Rectangle(w,h);
    length = l;
  }
  public int getLength() { return length; }
  public void setLength(int l) { length = l; }
  public int volume() { return length *
rec.area(); }
}
```

Επέκταση της κλάσης Rectangle μέσω κληρονομικότητας

```
public class Cuboid extends Rectangle {
  int length;
  public Cuboid(int l, int w, int h) {
    super(w,h); length = l;
  }
  public int getLength() { return length; }
  public void setLength(int l) {length = l;}
  public int volume() { return length * area(); }
}
```

Κριτήρια χρήσης της κληρονομικότητας

Οι παραπάνω δύο κλάσεις μεταγλωττίζονται και παράγουν το ίδιο λειτουργικό αποτέλεσμα. Το ερώτημα είναι ποια από τις δύο προσεγγίσεις θα προτιμήσουμε.

Χρησιμοποιώντας την ιδιότητα της κληρονομικότητας, εάν ο νέος τύπος δεδομένων που προκύπτει αποτελεί εξειδίκευση ή επέκταση της γονικής κλάσης αλλά εξακολουθεί να είναι και τύπος δεδομένων της γονικής κλάσης τότε μπορούμε να χρησιμοποιήσουμε την κληρονομικότητα ως μέθοδο δημιουργίας της νέας κλάσης.

Εάν όμως ο νέος τύπος δεδομένων δεν εξακολουθεί να είναι τύπος δεδομένων και της γονικής του κλάσης τότε είναι προτιμότερο ο νέος τύπος να ενσωματώνει την παραπάνω κλάση ως πεδίο, αντί να την επεκτείνει μέσω της κληρονομικότητας.

Στο προηγούμενο παράδειγμα, εάν η κλάση **Cuboid** κληρονομεί την κλάση **Rectangle** θα πρέπει η κλάση **Cuboid** να είναι και τύπου **Rectangle**. Όμως το κυβοειδές ως τρισδιάστατο σχήμα δεν μπορεί να είναι παράλληλα και διδιάστατο. Η νέα κλάση δεν αποτελεί εξειδίκευση της προηγούμενης, αλλά αυτοτελή τύπο δεδομένων.

Η χρήση της κληρονομικότητας στο παραπάνω παράδειγμα εισάγει λογική αντίφαση.

Ρητές και άρρητες μετατροπές τύπων

Άρρητη μετατροπή τύπου

Ας υποθέσουμε ότι έχουμε τη σχέση κληρονομικότητας

Bicycle → MountainBike

```
MountainBike myBike = new MountainBike();
```

Επειδή ο τύπος **MountainBike** κληρονομεί από την μεταβλητή **Bicycle** η μεταβλητή myBike είναι και τύπου Bicycle. Επομένως, θα μπορούσαμε να γράψουμε

```
Bicycle myBicycle = myBike;
```

ή

```
Bicycle yourBicycle = new MountainBike();
```

Ο compiler αντιλαμβάνεται την αλήθεια του παραπάνω ισχυρισμού.

Ρητή μετατροπή τύπου

Ας δοκιμάσουμε το ανάποδο

```
Bycycle myBicycle = new MountainBike();
MountainBike myBike = myBicycle;
```

Σε αυτή την περίπτωση ο *compiler* διαμαρτύρεται, διότι η μεταβλητή myBicycle είναι τύπου Bicycle και δεν είναι απαραίτητο ότι είναι και τύπου MountainBike.

Για να ξεπεράσουμε το παραπάνω μήνυμα λάθους θα πρέπει να βεβαιώσουμε τον *compiler* ότι το παρακάτω ισχύει:

```
Bycycle myBicycle = new MountainBike();

MountainBike myBike = (MountainBike) myBicycle;
```

Εάν η σχέση δεν ισχύει κατά την εκτέλεση του προγράμματος θα παραχθεί μία εξαίρεση (exception).

Ρητή μετατροπή τύπου

Ένας πιο ασφαλής τρόπος για να επαναλάβουμε τον παραπάνω κώδικα, χωρίς να παραχθεί εξαίρεση είναι ο εξής:

```
Bycycle myBicycle = new Bicycle();
MountainBike myBike;
if (myBicycle instanceof MountainBike) {
    myBike = (MountainBike)myBicycle;
}
```

Final Κλάσεις και Μέθοδοι

Μπορείτε να δηλώσετε μία ή περισσότερες μεθόδους μία κλάσης ως **final**. Δηλώνοντας μία μέθοδο ως **final** η μέθοδος αυτή δεν μπορεί να επαναοριστεί σε μία υποκλάση. Ο βασικός λόγος για να δηλώσετε μία μέθοδο ως **final** είναι αν η υλοποίηση της μεθόδου δεν πρέπει να αλλάξει.

```
class ChessAlgorithm {
    enum ChessPlayer { WHITE, BLACK }
    ...
    final ChessPlayer getFirstPlayer() {
        return ChessPlayer.WHITE;
    }
}
```

Μέθοδοι που καλούνται από τους κατασκευαστές της κλάσης ή για την αρχικοποίηση των μελών της κλάσης εκτός κατασκευαστών θα πρέπει να ορίζονται ως **final**.

Μπορείτε να προσδιορίσετε μία κλάση ως **final**, όταν θέλετε να δηλώσετε ότι η συγκεκριμένη κλάση δεν πρέπει να έχει υποκλάσεις. Ένα παράδειγμα τέτοια κλάσης είναι η κλάση **String** της standard βιβλιοθήκης

Κληρονομικότητα πολλαπλών γονικών κλάσεων?

```
public class Animal {
public String name;

public void setName(String
 name) {
   this.name = name;
   println("Animal's name”);
}
}
```

```
public class Mammal {
public String name;

public void setName(String name) {
   this.name = name;
   println("Mammal's name");
}
}
```

```
public class Dog
     extends Animal,
              Mammal {

public static void main(...) {
   Dog dog = new Dog();
   dog.setName(“Oscar”);
}
}
```

Ο τελεστής super

Πρόσβαση στους κατασκευαστές της γονικής κλάσης μέσω του τελεστή super


```
public class Bicycle {
   public int cadence;
   public int gear;
   public int speed;

   public Bicycle(int startCadence,int startSpeed, int
 startGear) {
      gear = startGear; cadence = startCadence;
      speed = startSpeed;
   }

   public void setCadence(int newValue) { cadence = newValue; }
   public void setGear(int newValue) {gear = newValue; }
   public void applyBrake(int decrement) {speed -= decrement; }
   public void speedUp(int increment) {speed += increment; }
 }
```

```
public class MountainBike extends Bicycle {

   public int seatHeight;

   public MountainBike(int startHeight,

        int startCadence, int startSpeed, int startGear) {

      super(startCadence, startSpeed, startGear);

      seatHeight = startHeight;

   }


   public void setHeight(int newValue) {

      seatHeight = newValue;

   }

}
```

Σειρά αρχικοποίησης μεταξύ της γονικής και και της απογόνου κλάσης

Στις περιπτώσεις που μία κλάση είναι απόγονος άλλης κλάσης η διαδικασία αρχικοποίησης έχει ως εξής. Αρχικά αρχικοποιείται το τμήμα της κλάσης που αφορά την γονική κλάση καλώντας των κατασκευαστή αυτής μέσω του τελεστή **super** και στη συνέχεια αρχικοποιείται η απόγονος κλάση.

Εάν δεν υπάρχει κλήση του κατασκευαστή της γονικής κλάσης μέσω του τελεστή **super** στον κατασκευαστή της απογόνου κλάσης, τότε ο ***compiler*** αναζητά εάν υπάρχει ο default κατασκευαστής για την γονική κλάση (δηλ. κατασκευαστής χωρίς ορίσματα) και καλεί αυτόματα αυτόν. Εάν δεν υπάρχει ούτε default κατασκευαστής εμφανίζει μήνυμα λάθους.

Δείτε το συγκεκριμένο παράδειγμα κώδικα που συμπυκνώνει τη διαδικασία αρχικοποίησης. Ακολουθήστε τα εξής βήματα:

1. Προσπαθήστε να μεταγλωττίσετε ως έχει και δείτε το μήνυμα λάθους του μεταγλωττιστή.
2. Αφαιρέστε από τα σχόλια την γραμμή 11 και μεταγλωττίστε.
3. Προσθέστε σε σχόλια την γραμμή 11 και αφαιρέστε τα σχόλια από την γραμμή 15. Μεταγλωττίστε ξανά.

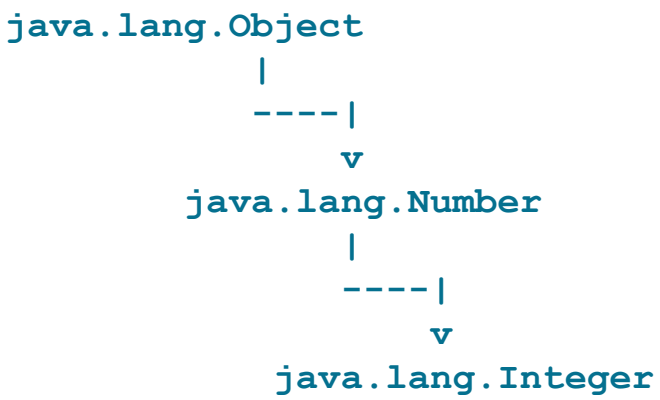
Πρόσβαση σε πεδία και μεθόδους της γονικής κλάσης μέσω του τελεστή super


```
public class MountainBike extends Bicycle {

  public int seatHeight;

  public MountainBike(int startHeight, int
 startCadence,
            int startSpeed, int startGear) {
    //super(startCadence, startSpeed, startGear);
    super.cadence = startCadence;
    super.speed = startSpeed;
    super.gear = startGear;
    seatHeight = startHeight;
  }
  public void setHeight(int newValue) {
    seatHeight = newValue;
  }
}
```

```
class Mammal {
  public String toString() { return "I am a mammal.";}
}
class Cat extends Mammal {
  public String toString() {
    return super.toString() + " In particular, I am a
 cat.";
  }
}
public class KittenBirth {
  public static void main(String []args) {
    Cat kitten = new Cat(); System.out.println(kitten);
  }
}
```

Η κλάση Object

Όλες οι κλάσεις στην Java είναι απόγονοι της κλάσης **Object**. Για παράδειγμα, η κλάση **Integer** κληρονομεί την κλάση **Number** που κληρονομεί την κλάση **Object**.



Όλες οι κλάσεις κληρονομούν συγκεκριμένες μεθόδους που ορίζονται στην κλάση **Object**. Τις μεθόδους αυτές μπορείτε αν θέλετε να τις επαναορίσετε στις κλάσεις που δημιουργείτε ή να τις αφήσετε ως έχουν.

Οι κυριότερες μέθοδοι της κλάσης Object είναι οι εξής:

1. **public boolean equals(Object obj):** Η μέθοδος ελέγχει αν δύο αντικείμενα είναι ίδια.
2. **public int hashCode():** Επιστρέφει έναν αριθμό (hash) για το συγκεκριμένο αντικείμενο.
3. **public String toString():** Επιστρέφει μία αναπαράσταση σε μορφή String για το αντικείμενο.

Στατικός και Δυναμικός Πολυμορφισμός

Στατικός Πολυμορφισμός

Κατά την εισαγωγή στις μεθόδους της κλάσης είδαμε την δυνατότητα ορισμού σε μία κλάση μεθόδων με το ίδιο όνομα αλλά με διαφορετικό αριθμό ή/και τύπο παραμέτρων.

Η παραπάνω δυνατότητα αναφέρεται συχνά και ως **στατικός πολυμορφισμός (static polymorphism)**.

Στο διπλανό παράδειγμα, ποια από τις 3 μεθόδους sum θα κληθεί κάθε φορά αποφασίζεται κατά την μεταγλώττιση του προγράμματος, με βάση τις παραμέτρους με τις οποίες την καλούμε την κάθε μέθοδο.

```
class Calculation {
  void sum(int a,int b){ System.out.println(a+b); }
  void sum(double a, double b) { System.out.println(a+b); }
  void sum(int a,int b,int c){ System.out.println(a+b+c); }

  public static void main(String args[]) {
    Calculation obj=new Calculation();
    obj.sum(10,10,10);
    obj.sum(20,20);
    obj.sum(12.2, 11.3);
  }
}
```

Δυναμικός Πολυμορφισμός

```
class Animal {
   public void signature(){
      System.out.println("I am an
 animal.");
   }
}
class Mammal extends Animal {
   public void signature() {
      System.out.println("I am a mammal!");
   }
}
class Dog extends Mammal {
   public void signature() {
      System.out.println("I am a dog!");
   }
}
```

```
public class TestDog {

   public static void main(String args[]) {
      Animal a = new Animal(); // Animal reference and object
      Animal b = new Mammal();  // Animal reference but Mammal object
      Animal c = new Dog();     // Animal reference but Dog object

      a.signature(); //output: I am an animal.
      b.signature(); //output: I am a mammal!
      c.signature(); //output: I am a dog!
   }
}
```

Δυναμικός Πολυμορφισμός

Παρατηρούμε ότι η κλήση της μεθόδου signature από τρεις μεταβλητές τύπου Animal όπου:

- η πρώτη (a) δείχνει σε αντικείμενο της γονικής κλάσης (Animal),
- η δεύτερη (b) σε αντικείμενο της υποκλάσης (Mammal) και
- η τρίτη (b) σε αντικείμενο της υποκλάσης (Dog) έχει ως αποτέλεσμα την κλήση διαφορετικών μεθόδων.
- Η απόφαση για το ποια μέθοδος θα κληθεί σε κάθε μία από τις τρεις περιπτώσεις **δεν μπορεί να αποφασιστεί κατά την μεταγλώττιση του προγράμματος**. Το JVM καλείται να αποφασίσει, με βάση τον τύπο του αντικειμένου στο οποίο δείχνει κάθε μεταβλητή κατά την εκτέλεση.

Δυναμικός Πολυμορφισμός

Στο προηγούμενο παράδειγμα, το JVM γνωρίζει το είδος του αντικειμένου στο οποίο δείχνουν οι μεταβλητές **a**, **b** και **c** ανεξάρτητα εάν η κάθε μία reference μεταβλητή είναι του τύπου της γονικής κλάσης ή της υποκλάσης.

Η παραπάνω διάκριση είναι γνωστή ως δυναμικός πολυμορφισμός (***dynamic polymorphism***), διότι το ποια μέθοδος θα κληθεί τελικά αποφασίζεται κατά την εκτέλεση και όχι κατά την μεταγλώττιση.

Κλήση στατικών μεθόδων από την γονική κλάση και την υποκλάση


```
public class Animal {
  public static void testClassMethod() {
    System.out.println("The static method in
Animal");
  }
  public void testInstanceMethod() {
    System.out.println("The instance method in
Animal");
  }
}
```

```
public class Cat extends Animal {
  public static void testClassMethod() {
    System.out.println("The static method in Cat");
  }
  public void testInstanceMethod() {
    System.out.println("The instance method in Cat");
  }
}
```

Κλήση στατικών μεθόδων από την γονική κλάση και την υποκλάση


```
public class TestCat {
  public static void main(String[] args) {
    Cat myCat = new Cat();
    Animal myAnimalCat = myCat;
    Animal myAnimal = new Animal();
    Animal.testClassMethod();
    Cat.testClassMethod();

    myAnimalCat.testClassMethod();
    myAnimal.testClassMethod();
    myCat.testClassMethod();

    myAnimalCat.testInstanceMethod();
    myAnimal.testInstanceMethod();
    myCat.testInstanceMethod();
  }
}
```

```
The static method in Animal
The static method in Cat

The static method in Animal
The static method in Animal
The static method in Cat

The instance method in Cat
The instance method in Animal
The instance method in Cat
```

Κλήση στατικών μεθόδων από την γονική κλάση και την υποκλάση

Παρατηρούμε ότι ο τύπος των δεδομένων από τον οποίο καλούμε μία στατική μέθοδο είναι καθοριστικός για το ποια μέθοδος θα κληθεί τελικά.

Για παράδειγμα, η μεταβλητή **`myAnimal`** ορίζεται ως τύπου **`Animal`** αλλά δείχνει σε ένα αντικείμενο τύπου **`Cat`**.

Η στατική μέθοδος που τελικά καλείται είναι εκείνη που ορίζεται από τον τύπο της reference μεταβλητής (**`Animal`** ή **`Cat`**) και όχι από το ίδιο το αντικείμενο στο οποίο δείχνει αυτή, όπως ίσως θα περίμενε ίσως κανείς να ισχύει με βάση τον δυναμικό πολυμορφισμό που είδαμε προηγούμενα.

Abstract Κλάσεις

Abstract Κλάσεις

Η Java επιτρέπει ορισμό κλάσεων οι οποίες είναι πιο γενικές και δεν μπορούν να υλοποιήσουν απευθείας αντικείμενα.

Αν και οι συγκεκριμένες κλάσεις δεν μπορούν να παράγουν αντικείμενα μπορούν να έχουν υποκλάσεις οι οποίες παράγουν αντικείμενα.

Οι κλάσεις αυτές έχουν τον προσδιοριστή **`abstract`** μπροστά από το όνομα της κλάσης.

Abstract Κλάσεις

Μία abstract κλάση μπορεί να έχει "κανονικές" μεθόδους ή **abstract** μεθόδους ή συνδυασμό των παραπάνω.

Με τον όρο **abstract method** εννοούμε μία μέθοδο της οποίας ορίζεται μόνο το *prototype* αλλά δεν ορίζεται η υλοποίηση. Η υλοποίηση της θα οριστεί σε κάποια υποκλάση της συγκεκριμένης **abstract** κλάσης.

Επίσης, από μία **abstract** κλάση μπορεί να οριστεί μία υποκλάση που και αυτή να είναι **abstract**.

Abstract Κλάσεις - Παράδειγμα

```
public abstract class Employee {
   private String name;
   private String address;
   private int id;
   public Employee(String name, String address, int id) {
      System.out.println("Constructing an Employee");
      this.name = name; this.address = address; this.id = id;
   }
   public abstract int computeSalary();
   public String toString() { return name + " " + address + " " + id; }
   public String getName() { return name; }
   public String getAddress() { return address; }
   public void setAddress(String newAddress) { address = newAddress; }
   public int getNumber(){ return id; }
}
```

```
public class Secretary extends Employee {
   int salary;
   public Secretary(String name, String address,
                    int id, int yearly_salary) {
      super(name, address, id);
      salary = yearly_salary;
   }
   public int computeSalary() {return salary/12; }
   public String toString() {
      return super.toString() + " " + salary;
   }
}
```