ΕΛΛΗΝΙΚΗ ΔΗΜΟΚΡΑΤΙΑ

Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών

ΠΛΗΡΟΦΟΡΙΚΗ ΙΙ

Ενότητα 9: Κληρονομικότητα (Inheritance)

Μιχάλης Δρακόπουλος

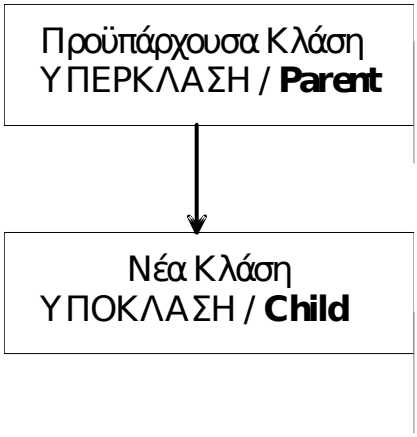
Σχολή Θετικών επιστημών

Τμήμα Μαθηματικών

ΠΛΗΡΟΦΟΡΙΚΗ ΙΙ (Java)
Ενότητα 9

Κληρονομικότητα (Inheritance)

Υπάρχουν κλάσεις που εμπεριέχουν μεταβλητές και μεθόδους που έχουν οριστεί σε προϋπάρχουσες κλάσεις:




```
public class Child extends Parent
{
...
}
```

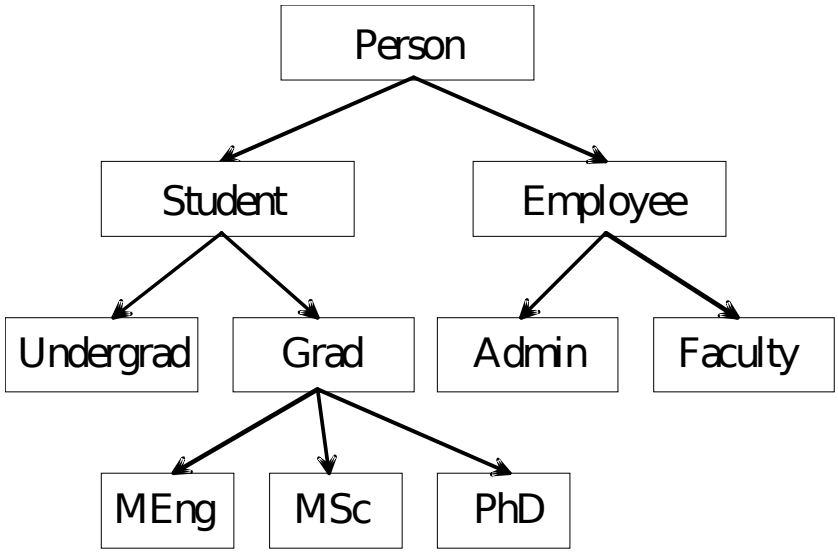
Η δήλωση λοιπόν της σχέσης κληρονομικότητας μεταξύ δύο κλάσεων είναι της μορφής:

```
public class <Υποκλάση> extends <Υπερκλάση>
```

Έτσι δημιουργείται μία ιεραρχία, όπου από τη γενικότερη κλάση μπορούμε να περάσουμε σε ειδικότερες, όπου κάθε μία έχει και τα χαρακτηριστικά της γενικότερης προγόνου της, αλλά και αυτά των ακόμα γενικότερων προγόνων της προγόνου της!

.

Π.χ., θέλουμε να φτιάξουμε ένα πρόγραμμα που να περιέχει στοιχεία για τα μέλη ενός πανεπιστημίου. Μια ομαδοποίηση των μελών μπορεί να είναι η εξής:



Στο παράδειγμα αυτό, θα είχαμε δηλώσεις κλάσεων της μορφής:

```
public class Person
{
  ...
}

public class Student extends Person
{
  ...
}

public class Undergrad extends Student
{
  ...
}

public class Grad extends Student
{
  ...
}
```

κτλ.

Και τα αντικείμενα της `Student` *και* τα αντικείμενα της `Employee` αντιπροσωπεύουν ανθρώπους, άρα έχουν κάποιες κοινές ιδιότητες. Επομένως, οι μέθοδοι που θα αρχικοποιούν, θα τροποποιούν ή θα κάνουν output π.χ. το όνομα κάποιου μέλους του πανεπιστημίου, είτε είναι φοιτητής (Student) είτε είναι εργαζόμενος (Employee), θα είναι ίδιες. Όλες αυτές τις μεθόδους τις βάζουμε σε μία υπερκλάση `Person`. Εκεί φυσικά βάζουμε και όλα τα κοινά στοιχεία (μεταβλητές) των δύο «κατηγοριών» (φοιτητών και εργαζόμενων). Με αυτή τη λογική δομούνται οι σχέσεις κληρονομικότητας των κλάσεων.

Μια απλή μορφή της κλάσης `Person` είναι η εξής:

```
public class Person
{
        // class variables
  // μεταβλητή για το όνομα του κάθε ατόμου:
  private String name;

        // constructors
  public Person()
  {
    name = "Δεν έχει ορισθεί ακόμα όνομα..";
  }

  public Person(String initName)
  {
    name = initName;
  }

  // modifier μέθοδος για αλλαγή του ονόματος
  public void setName(String newName)
  {
    name = newName;
  }
```

```
  // accessor μέθοδος
  public String accessName()
  {
    return name;
  }

  // μέθοδος για την εκτύπωση του αποτελέσματος
  public void writeOutput()
  {
    System.out.println("Name: " + name);
  }

}
```

Η κλάση `Student` κληρονομεί την κλάση `Person` (τις μεταβλητές της (το `name` δηλαδή) και τις μεθόδους της) και προσθέτει μια δικιά της μεταβλητή για τον αριθμό μητρώου του κάθε φοιτητή:

```
public class Student extends Person
{
  // μεταβλητή για τον αριθμό μητρώου του κάθε φοιτητή
  private int studentNumber;

  public Student()
  {
    super();        // -> κλήση του constructor της Person
    studentNumber = 0;
  }

  public Student(String initName, int initStudNo)
  {
    super(initName);  // -> κλήση του constructor της Person
    studentNumber = initStudNo;
  }

  public void reset(String newName, int newStudNo)
  {
    setName(newName);  // -> κλήση μεθόδου της Person!
    studentNumber = newStudNo;
  }

  public void setStudNumber(int newStudNo)
  {
    studentNumber = newStudNo;
  }

  public int accessStudentNumber()
  {
    return studentNumber;
  }

  // μέθοδος για την εκτύπωση του αποτελέσματος
  public void writeOutput()
  {
    System.out.println("Name: " + accessName());
    System.out.println("Student no.: " + studentNumber);
  }

}
```

Αν λοιπόν είχαμε τις δύο αυτές κλάσεις υποστήριξης, στην κλάση εφαρμογής θα μπορούσαμε να είχαμε τα εξής:

```
public class InheritanceExample
{
  public static void main (String [ ] args)
  {
    Student s = new Student();
    s.setName("Xxxxx Yyyyyyy"); // -> setName της Person
    s.setStudNumber(2010001); // -> setStudNumber της Student
    s.writeOutput();   // -> writeOutput της Student (Overriding)
  }

}
```

Αφού η `Student` κληρονομεί την `Person`, στην ουσία υπάρχουν δύο μέθοδοι `writeoutput()`. Αυτή που τελικά «τρέχει» είναι η `writeoutput()` της `Student` και όχι της `Person`. Αυτό λέγεται **overriding (υπερκάλυψη, ή παράκαμψη)**. Αν στην υποκλάση υπάρχει μέθοδος με ίδιο όνομα και ίδιο πλήθος και είδος παραμέτρων με κάποια μέθοδο της υπερκλάσης, τότε η μέθοδος της υποκλάσης κάνει override (υπερκαλύπτει ή παρακάμπτει) τη μέθοδο της υπερκλάσης.

Αν η μέθοδος που παρακάμπτεται (της υπερκλάσης) επιστρέφει (`return`) κάποια μεταβλητή, η νέα μέθοδος που την κάνει override (στην υποκλάση) πρέπει να επιστρέφει μεταβλητή του ίδιου τύπου.

→	**Overriding (υπερκάλυψη)**	/	**Overloading (υπερφόρτωση)**
	- Ίδιο όνομα μεθόδων		- Ίδιο όνομα μεθόδων
	- και ίδιο πλήθος παραμέτρων		- και διαφορετικό πλήθος παραμ.
	- και ίδιο είδος παραμέτρων		- ή διαφορετικό είδος παραμ.
	- και ίδιο είδος επιστροφής (return)		- ανεξαρτήτως είδους επιστροφής

Overriding: μεταξύ μεθόδων διαφορετικών κλάσεων που συνδέονται με κληρονομικότητα

Overloading: μεταξύ μεθόδων της ίδιας κλάσης ή μεθόδων κλάσεων που συνδέονται με κληρονομικότητα

Η λέξη `final` δηλώνει μέθοδο που δεν μπορεί να γίνει overridden (να παρακαμφθεί).

π.χ.
```
public final void specialMethod()
{
  ...
}
```

Η λέξη `super`:

i) `super.writeOutput(); //` → κλήση μεθόδου που έχει γίνει overridden (δηλαδή, καλεί τη μέθοδο της υπερκλάσης)

ii) Κλήση του κατασκευαστή της υπερκλάσης: `super();`

ΠΡΟΣΟΧΗ! Η λέξη super όταν χρησιμοποιείται για την κλήση ενός κατασκευαστή της υπερκλάσης μέσα σε κάποιον κατασκευαστή της υποκλάσης, πρέπει να είναι η πρώτη εντολή αυτού του κατασκευαστή. Δεν μπορεί να χρησιμοποιηθεί αργότερα μέσα στον κατασκευαστή.

Στην πραγματικότητα, αν δεν υπάρχει κλήση του κατασκευαστή της υπερκλάσης (κάποια μορφή της `super()` δηλαδή) μέσα στον κατασκευαστή της υποκλάσης, τότε η Java καλεί *αυτόματα* τον default constructor της υπερκλάσης.

Άρα, στον κατασκευαστή της `Student`:

```
public Student()
{
  super();
  studentNumber = 0;
}
```

θα ήταν το ίδιο αν γράφαμε:

```
public Student()
{
  studentNumber = 0;
}
```

Συμβουλή: Καλύτερα να γράφουμε την κλήση της `super()` κι ας μην είναι απαραίτητο, για να είναι πιο σαφής ο κώδικάς μας.

Η λέξη `this` αναφέρεται στον κατασκευαστή της υποκλάσης.

π.χ., ένας ακόμη constructor της `Student` θα μπορούσε να είναι ο εξής:

```
public Student(String initName)
{
  this(initName, 0); → κλήση του constructor: Student(String, int)
}
```

Σημείωση: Πρέπει και η κλήση της `this` να είναι η *πρώτη* εντολή του κατασκευαστή. Προφανώς, όταν σε έναν κατασκευαστή της υποκλάσης υπάρχει κλήση άλλου κατασκευαστή της μέσω της `this`, τότε δεν μπορεί να υπάρχει ταυτόχρονα και η εντολή `super`, αφού ο κατασκευαστής της υπερκλάσης θα κληθεί τελικά μέσω του κατασκευαστή στον οποίο αναφέρεται η `this`.

Από αυτά που είπαμε μέχρι τώρα για τη `super`, προκύπτει ότι θα μπορούσαμε να γράψουμε πιο σωστά τη μέθοδο `writeOutput()` της `Student`, χωρίς επανάληψη κώδικα:

Η αρχική μέθοδος:

```
public void writeOutput()
{
  System.out.println("Name: " + accessName());
  System.out.println(Student no.: " + studentNumber);
}
```

μπορεί να γίνει:

```
public void writeOutput()
{
  super.writeOutput();  // -> κλήση της writeOutput της Person
  System.out.println(Student no.: ” + studentNumber);
}
```

Η ορατότητα `protected`:

Μέχρι τώρα είχαμε δει τις εξής δηλώσεις ορατότητας μεταβλητών ή μεθόδων:

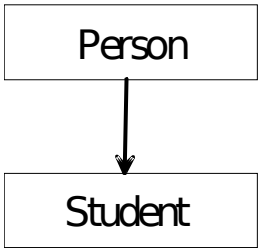
`public`, `private` και χωρίς δήλωση.

Υπάρχει και η δήλωση ορατότητας **`protected`** η οποία κάνει τις μεταβλητές ή μεθόδους ορατές μόνο από κλάσεις που σχετίζονται με κληρονομικότητα (και από κλάσεις του ίδιου πακέτου, όπως θα δούμε). Δηλαδή μια μεταβλητή ή μέθοδος που δηλώνεται σαν `protected`, λειτουργεί σαν `public` για τις κλάσεις απογόνους της κλάσης ορισμού της και σαν `private` για όλες τις υπόλοιπες κλάσεις του προγράμματος.

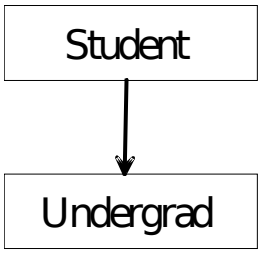
Συνοπτικά:

Ορατότητα:	`public`	`protected`	χωρίς δήλωση ορατότητας	`private`
Μεταξύ μεθόδων της ίδιας κλάσης	ναι	ναι	ναι	ναι
Μεταξύ κλάσεων του ίδιου πακέτου	ναι	**ναι**	ναι	όχι
Μεταξύ κλάσεων εκτός πακέτου	ναι	όχι	όχι	όχι
Από υποκλάσεις του ίδιου πακέτου	ναι	**ναι**	ναι	όχι
Από υποκλάσεις εκτός πακέτου	ναι	**ναι**	όχι	όχι

Όπως είχαμε τη σχέση κληρονομικότητας:



έτσι μπορούμε να έχουμε και τη σχέση:



Στη δεύτερη αυτή σχέση κληρονομικότητας, η κλάση `Student` γίνεται πλέον <u>υπερκλάση</u> και το ρόλο της υποκλάσης παίζει η κλάση `Undergrad`. Έτσι δημιουργείται μια αλυσίδα κληρονομικότητας που αποτελείται από απλά ζευγάρια κληρονομικότητας, η οποία οδηγεί στην *κληρονομικότητα πολλών επιπέδων*. Η αλυσίδα αυτή ξεκινάει από το «γενικό» και προχωράει προς το «ειδικό». Η «εξειδίκευση» όμως αυτή κάνει τις κλάσεις «ευρύτερες», αφού κληρονομούν τα στοιχεία των γενικότερων κλάσεων και μπορούν να τα χρησιμοποιούν σαν δικά τους.

Ο παρακάτω πίνακας δείχνει την αύξηση του εύρους των διαθέσιμων στοιχείων των κλάσεων, καθώς αυτές εξειδικεύονται και συγχρόνως κληρονομούν στοιχεία των προγόνων τους, για το παράδειγμά μας:

Οι κλάσεις →	`Person`	`Student`	`Undergrad`
χρησιμοποιούν μετ/τές & μεθόδ. της κλάσης:			
`Person`	ναι	ναι	**ναι**
`Student`	όχι	ναι	ναι
`Undergrad`	όχι	όχι	ναι

Προφανώς κάθε κλάση χρησιμοποιεί μεταβλητές και μεθόδους του εαυτού της, αλλά βλέπουμε ότι η `Person` χρησιμοποιεί απλά τα δικά της στοιχεία, ενώ η `Student` χρησιμοποιεί τα δικά της και της `Person`, ενώ η `Undergrad` χρησιμοποιεί και τα δικά της και της `Student`, *αλλά και της `Person`*. Όλα αυτά βέβαια αφορούν μεταβλητές και μεθόδους ορατότητας `public` ή `protected`.

Η κλάση `Undergrad` θα μπορούσε να περιέχει τα ακόλουθα:

```
public class Undergrad extends Student
{
         // class variables
  // μεταβλητή για το έτος φοίτησης του κάθε φοιτητή
  private int year;

  // constructors
```

```
  public Undergrad()
  {
    super();  // καλεί τον default constructor της Student
    year = 1;
  }

  public Undergrad(String initName, int initStudNo, int initYear)
  {
    super(initName, initStudNo);  // → constructor της Student
    year = initYear;
  }

  // methods

  // modifier method
  public void reset(String newName, int newStudNo, int newYear)
  {
    reset(newName, newStudNo); //→ κλήση της overloaded reset της Student
    year = newYear;
  }

  // modifier method
  public void setYear(int newYear)
  {
    year = newYear;
  }

  // accessor method
  public int accessYear()
  {
    return year;
  }

  // μέθοδος για εκτύπωση του αποτελέσματος
  public void writeOutput()
  {
    super.writeOutput(); // → Η writeOutput της Student - Overriding
    System.out.println("Student year: " + year);
  }
}
```

Στην κλάση αυτή παρουσιάζονται τόσο το φαινόμενο του overriding όσο και αυτό του overloading. Overriding μπορεί να γίνει και μεταξύ κλάσεων όχι "άμεσης" κληρονομικότητας αλλά "έμμεσης", δηλαδή μεταξύ της `Undergrad` και της `Person` στο παράδειγμά μας (οι κλάσεις αυτές έχουν *έμμεση σχέση κληρονομικότητας* γιατί η `Undergrad` κληρονομεί τη `Student` και η `Person` κληρονομείται από τη `Student`). Π.χ., θα μπορούσε στην `Undergrad` να υπάρχει και η ακόλουθη μέθοδος:

```
public void setName(String newName)   // μέθοδος της Undergrad
{
   System.out.println("Αλλαγή ονόματος προπτυχιακού φοιτητή");
   super.setName(newName);    // -> setName της Person!
}
```

Η μέθοδος αυτή της `Undergrad` *υπερκαλύπτει τη μέθοδο* `setName` ***της Person*** (παρόλο που η `Person` σε σχέση με την `Undergrad` είναι δύο "γενιές" πίσω). Προσοχή χρειάζεται στο ότι και εδώ η κλήση υπερκαλυμμένης μεθόδου γίνεται με τη χρήση της `super`, παρόλο που η μέθοδος αυτή δεν βρίσκεται στην υπερκλάση (`Student`) αλλά έχει κληρονομηθεί εκεί από μια δική της

υπερκλάση (`Person`). Ουσιαστικά δηλαδή, η `super` αναφέρεται και σε μεθόδους *που έχει κληρονομήσει* η υπερκλάση.

(Φυσικά η ύπαρξη της συγκεκριμένης αυτής μεθόδου `setName` στην κλάση `Undergrad` δεν έχει νόημα, αφού στην ουσία δεν κάνει κάτι διαφορετικό από τη μέθοδο `setName` της `Person`, η οποία μπορεί φυσικά να χρησιμοποιηθεί με οποιοδήποτε αντικείμενο της `Undergrad`, αφού κληρονομείται από αυτή (και είναι `public`)).

Προχωρημένα θέματα κληρονομικότητας:

Όπως προαναφέρθηκε, η κληρονομικότητα *ορίζεται* μεταξύ δύο και μόνο κλάσεων,

από υπερκλάση → σε υποκλάση
ή αλλιώς, από parent → σε child

και ο ορισμός γίνεται στην υποκλάση (child) ως εξής:

```
public class Child extends Parent
```

αλλά ουσιαστικά *ισχύει σε πολλαπλά επίπεδα*, όταν μια κλάση που κληρονομεί κάποια άλλη κλάση, κληρονομείται με τη σειρά της από κάποια τρίτη κλάση. Δηλαδή, στη Java δεν υπάρχει *πολλαπλή κληρονομικότητα* όπως υπάρχει σε άλλες γλώσσες προγραμματισμού (π.χ. C++), αλλά υπάρχει *κληρονομικότητα πολλών επιπέδων*. Από τα όσα έχουν αναφερθεί μέχρι τώρα, γίνεται σαφές ότι ο βασικός σκοπός της κληρονομικότητας είναι η *επαναχρησιμοποίηση κώδικα*.

Μετατροπή τύπου αντικειμένων

Στη Java τα αντικείμενα της κλάσης `Parent` είναι μόνο τύπου `Parent`, ενώ τα αντικείμενα της κλάσης `Child` είναι τύπου `Child` *και* τύπου `Parent`. Στην ουσία, εάν μία κλάση έχει πολλούς προγόνους (έμμεσα, μέσω της κλάσης που κληρονομεί η υπερκλάση της, κ.ο.κ.), τα αντικείμενά της είναι τύπου της ίδιας της κλάσης, τύπου της υπερκλάσης της, αλλά και τύπου κάθε μιας από τις κλάσεις που είναι πρόγονοί της. Άρα, στο προηγούμενο παράδειγμα, τα:

```
Person sp1 = new Student();
```

και

```
Student us1 = new Undergrad();
```

είναι **σωστά** γιατί ένα αντικείμενο `sp1` της `Student` μπορεί να είναι και τύπου `Person`, όπως και ένα αντικείμενο `us1` της `Undergrad` μπορεί να είναι και τύπου `Student`, ενώ το ίδιο ισχύει και για το:

```
Person up1 = new Undergrad();
```

αφού τα αντικείμενα της `Undergrad` μπορούν να είναι *και* τύπου `Student` αλλά *και* τύπου `Person` (αφού και οι δύο αυτές κλάσεις είναι πρόγονοί της).

Αντιθέτως, το:

```
Student ps1 = new Person();  // ΛΑΘΟΣ!
```

είναι **λάθος**, αφού τα αντικείμενα μιας κλάσης `Parent` (εδώ το `ps1` της `Person`) είναι μόνο τύπου της κλάσης αυτής και όχι τύπου των απογόνων της.

Τι νόημα όμως έχει μια δημιουργία αντικειμένου τύπου `Parent` με τον κατασκευαστή μιας `Child`, όπως π.χ. η εντολή:

```
Person sp1 = new Student();
```

Μια πιο σωστή διατύπωση της έκφρασης *"δημιουργία αντικειμένου τύπου `Person` με τον κατασκευαστή της `Student`"* θα ήταν η *"δημιουργία αντικειμένου της `Student` που δηλώνεται σαν τύπου `Person`"*. Στην ουσία, αυτό που γίνεται είναι μια μετατροπή (*type casting*) ενός αντικειμένου της `Student` σε τύπο `Person`. Το αντικείμενο αυτό (`sp1`) είναι πρακτικά κάτι ενδιάμεσο από αντικείμενο της `Person` και της `Student`. Μπορεί να χρησιμοποιηθεί σαν όρισμα εισόδου σε μεθόδους που δέχονται αντικείμενα τύπου `Person` αλλά όχι σε μεθόδους που δέχονται αντικείμενα τύπου `Student`. Αντιθέτως, ένα κανονικό αντικείμενο της `Student` θα μπορούσε να χρησιμοποιηθεί και στις δύο περιπτώσεις. Μια άλλη διαφορά θα γίνει σαφής στην αμέσως επόμενη ενότητα.

Το συγκεκριμένο casting ονομάζεται *upcasting*, επειδή μία κλάση τύπου `Child` "αναβαθμίζεται" σε τύπου `Parent`. Το αντίθετο (*downcasting*) είναι δυνατό με την εντολή:

```
Student s1 = (Student) sp1;
```

Έτσι μετατρέπεται το αντικείμενο `sp1` σε ένα κανονικό αντικείμενο της `Student (s1)`.

Παρατήρηση: Εάν η μετατροπή εφαρμοζόταν σε ένα κανονικό αντικείμενο της `Person` (π.χ. στο `Person p = new Person()`), δηλαδή:

```
Student s2 = (Student) p;
```

τότε η εντολή θα γινόταν compile αλλά θα προέκυπτε *σφάλμα εκτέλεσης* (run-time error). Δηλαδή, downcasting γίνεται μόνο σε αντικείμενα που έχουν υποστεί upcasting (ή έχουν δημιουργηθεί με αυτόν τον τρόπο, όπως τα `sp1`, `up1` κλπ παραπάνω).

Παράδειγμα:

Έστω ότι υπάρχει στην κλάση `Student` η ακόλουθη μέθοδος `equals`:

```
public boolean equals(Student otherStudent)
{
  return (this.studentNumber == otherStudent.studentNumber);
}
```

ενώ στην κλάση `Undergrad` υπάρχει η ακόλουθη έκδοσή της που την κάνει overload (υπερφόρτωση), αφού έχει διαφορετικού τύπου παράμετρο εισόδου:

```
public boolean equals(Undergrad otherUndergrad)
{
  return (super.equals(otherUndegrad)
         && (this.year == otherUndergrad.year) );
}
```

Παρατηρούμε ότι η `equals` της `Undergrad` καλεί την `equals` της `Student` χρησιμοποιώντας τη λέξη `super`, παρόλο που δεν πρόκειται για μια μέθοδο που έχει υπερκαλυφθεί (παρακαμφθεί – override) αλλά για μια μέθοδο που έχει υπερφορτωθεί (overload)! Η λέξη `super` είναι απαραίτητη διότι ο σκοπός είναι να κληθεί η `equals` της `Student` με ένα αντικείμενο `Undergrad` ώστε να ελεγχθεί η ταύτιση των αριθμών μητρώου. Αυτό μπορεί να γίνει (παρόλο που η `equals` της `Student` δέχεται αντικείμενο τύπου `Student`) γιατί ένα αντικείμενο τύπου `Undergrad` είναι *και* τύπου `Student`. Μια κλήση όμως της `equals` με το `otherUndergrad` χωρίς τη `super` θα έδινε προτεραιότητα στην `equals` της `Undergrad` που δέχεται αντικείμενο τύπου `Undergrad` (δηλαδή η μέθοδος θα καλούσε τον εαυτό της..). Στην ουσία ένα αντικείμενο τύπου `Undergrad` είναι πρώτα τύπου `Undergrad` και μετά τύπου `Student`.

Αλλαγή ορατότητας υπερκαλυμμένης (overriden) μεθόδου

Μια μέθοδος που παρακάμπτεται (υπερκαλύπτεται) σε κάποια υποκλάση, μπορεί να αλλάξει ορατότητα, δηλαδή μπορεί ο προγραμματιστής να αλλάξει τα δικαιώματα προσπέλασης μιας overriden μεθόδου. Η αλλαγή μπορεί να γίνει μόνο προς την κατεύθυνση της διεύρυνσης της ορατότητας της μεθόδου, π.χ. από `private` σε `public` ή από `protected` σε `public` και ποτέ προς την αντίθετη κατεύθυνση. Άρα, μια μέθοδος της υπερκλάσης:

```
private void doSomething()
```

θα μπορούσε να παρακαμφθεί στην υποκλάση από την ακόλουθη μέθοδο:

```
public void doSomething()
```

Παρατήρηση: Στο παράδειγμα δημιουργίας ενός αντικειμένου της `Student` τύπου `Person` που αναφέρθηκε παραπάνω (`Person sp1 = new Student();`), μια κλήση της μεθόδου `writeOutput` με αυτό το αντικείμενο (π.χ. από την κλάση εφαρμογής), θα αναφερόταν όπως προαναφέρθηκε στη `writeOutput` της `Student`, αφού αυτή της `Person` έχει παρακαμφθεί (όπως θα γινόταν και με ένα κανονικό αντικείμενο της `Student` δηλαδή). Όμως, εάν η `writeOutput` της `Person` είχε ορισθεί ως `private` (ενώ αυτή της `Student` κανονικά ως `public`), τότε δεν θα ήταν δυνατή κλήση της `writeOutput` της `Student` με αυτό το αντικείμενο, παρόλο που η `writeOutput` της `Student` η οποία καλείται, είναι `public`. *Άρα, στην ουσία η δυνατότητα προσπέλασης των μεθόδων που υπερκαλύπτουν κάποια μέθοδο, ορίζεται από την ορατότητα της υπερκαλυμμένης μεθόδου στην περίπτωση αντικειμένων της υποκλάσης "τύπου υπερκλάσης" (εδώ, της `Student` τύπου `Person`).*

Αλλαγή τύπου υπερκαλυμμένης (overriden) μεθόδου

Από την έκδοση 5.0 της Java και μετά, εκτός από αλλαγή ορατότητας, μια υπερκαλυμμένη μέθοδος μπορεί να υποστεί και αλλαγή τύπου (επιστρεφόμενης τιμής). Σύμφωνα με το γενικό κανόνα της υπερκάλυψης (override), η μέθοδος της υποκλάσης που κάνει override τη μέθοδο της υπερκλάσης, πρέπει να είναι του ίδιου τύπου με τη μέθοδο που υπερκαλύπτεται. Στον κανόνα αυτό υπάρχει η

εξής εξαίρεση: εάν η μέθοδος που υπερκαλύπτεται είναι τύπου κλάσης, τότε η μέθοδος που την υπερκαλύπτει μπορεί να είναι τύπου οποιασδήποτε κλάσης απογόνου της αρχικά επιστρεφόμενης κλάσης. Για παράδειγμα, εάν σε μια κλάση `Parent` υπάρχει η μέθοδος `getUnivMember` που επιστρέφει αντικείμενο τύπου `Student` (κάποιον φοιτητή):

```
public class Parent
{
     ...
   public Student getUnivMember(int id)
   {
      ...
   }
     ...
}
```

σε μια υποκλάση της `Parent` θα μπορούσε να υπάρχει μια `getUnivMember` που την υπερκαλύπτει, ως εξής:

```
public class Child extends Parent
{
     ...
   public Undergrad getUnivMember(int id)
   {
      ...
   }
     ...
}
```

Η υπερκαλυμμένη έκδοση της `getUnivMember` επιστρέφει αντικείμενο τύπου `Undergrad` (δηλαδή κάποιον προπτυχιακό φοιτητή). Αυτό επιτρέπεται γιατί η `Undergrad` κληρονομεί τη `Student`.

Στην πραγματικότητα, αυτό που συμβαίνει στη μέθοδο που υπερκαλύπτει την αρχική, δεν είναι μια απλή αλλαγή τύπου, αλλά η προσθήκη επιπλέον περιορισμών στον επιστρεφόμενο τύπο της. Κάθε αντικείμενο της `Undergrad` είναι στην ουσία ένα αντικείμενο της `Student` με κάποιες επιπρόσθετες ιδιότητες, οι οποίες ενώ είναι ιδιότητες κάθε προπτυχιακού φοιτητή, δεν είναι ιδιότητες που έχει ο κάθε φοιτητής γενικά.

Βασικός λόγος που τα `private` πεδία δεν είναι ορατά στους απογόνους

Όπως ήδη αναφέρθηκε, τα `private` στοιχεία μιας υπερκλάσης, κληρονομούνται μεν στις υποκλάσεις της, αλλά δεν είναι ορατά από αυτές. Π.χ., στο αρχικό παράδειγμα με τις κλάσεις `Person` και `Student`, η μεταβλητή `name` της `Person` είναι `private`. Κληρονομείται φυσικά στη `Student`, δηλαδή κάθε αντικείμενο της `Student` έχει μια μεταβλητή `name` (ο κάθε φοιτητής έχει όνομα), όμως η `Student` για να την προσπελάσει χρησιμοποιεί τις ανάλογες `public` μεθόδους της `Person` (την "τροποποιητική" `setName` και την "πρόσβασης" `accessName`). Ο βασικός λόγος που συμβαίνει αυτό είναι γιατί σε διαφορετική περίπτωση, *κάποιος που θα ήθελε να αποκτήσει άμεση πρόσβαση σε οποιαδήποτε `private` μεταβλητή/μέθοδο μιας κλάσης, θα μπορούσε να το κάνει απλά φτιάχνοντας μια υποκλάση της!* Θα καταστρατηγούνταν δηλαδή η έννοια της *ενθυλάκωσης* (encapsulation), μιας από τις πολύ βασικές αρχές του αντικειμενοστραφούς προγραμματισμού που αναφέρεται στην απόκρυψη των "ιδιωτικών" στοιχείων μιας κλάσης από τις υπόλοιπες κλάσεις του προγράμματος.

Η κλάση `Object`

Στη Java, κάθε κλάση θεωρείται απόγονος της κλάσης `Object`. Έτσι, κάθε αντικείμενο οποιασδήποτε κλάσης είναι και τύπου `Object` (αλλά και τύπου όλων των προγόνων της, εάν η συγκεκριμένη κλάση κληρονομεί κάποια άλλη κλάση). Αυτό επιτρέπει τη δημιουργία μεθόδων με παράμετρο εισόδου τύπου `Object`, η οποία είσοδος κατά την κλήση της μεθόδου μπορεί να αντικατασταθεί από ένα αντικείμενο οποιασδήποτε κλάσης.

Μία χρήσιμη μέθοδος της κλάσης `Object` είναι η `toString`. Η μέθοδος αυτή επιστρέφει το περιεχόμενο των μεταβλητών ενός αντικειμένου (οποιασδήποτε κλάσης, αφού όλες οι κλάσεις την κληρονομούν από την `Object`) σε μορφή `String`. Όμως, παρόλο που η μέθοδος αυτή κληρονομείται σε οποιαδήποτε κλάση, δεν μπορεί να χρησιμοποιηθεί άμεσα, γιατί δεν καταφέρνει να κωδικοποιήσει σωστά τις υπάρχουσες κάθε φορά μεταβλητές ενός αντικειμένου σε `String`. Θα πρέπει να υπάρχει μέθοδος `toString` που να την υπερκαλύπτει. Στο παράδειγμα με τις κλάσεις `Person -> Student -> Undergrad` κτλ., αντί των μεθόδων `writeOutput`, θα ήταν καλύτερα να είχαμε μεθόδους `toString` που να κάνουν ακριβώς τα ίδια πράγματα (δηλ. να περιέχουν τον ίδιο κώδικα με τις `writeOutput`) και να επιστρέφουν το επιθυμητό `String` αντί να το εκτυπώνουν. Ο βασικός λόγος είναι ότι οι μέθοδοι `toString` που κάνουν override την `toString` της `Object`, αποκτούν μια πολύ "βολική" ιδιότητα αυτής της `toString`: Καλούνται αυτόματα (όταν υπάρχουν σε μια κλάση φυσικά) όταν κάποιος προσπαθήσει να εμφανίσει τα περιεχόμενα ενός αντικειμένου μιας κλάσης, καλώντας το αντικείμενο άμεσα μέσα σε μια `System.out.println`. Π.χ., στο αρχικό παράδειγμα, εάν στην κλάση `Student` η `writeOutput` μετατρεπώνταν στην αντίστοιχη `toString` (οπότε φυσικά η εντολή `s.writeOutput();` της `main` θα ήταν λάθος), η εντολή που κανονικά θα έπρεπε να υπάρχει στη `main`:

```
System.out.println(s.toString());
```

θα μπορούσε να αντικατασταθεί από την:

```
System.out.println(s);
```

(υπενθυμίζεται ότι το `s` είναι ένα αντικείμενο της `Student`)

Σημείωση: εάν δεν υπήρχε η έκδοση της `toString` στη `Student` που υπερκαλύπτει την `toString` της `Object`, τότε η εντολή δεν θα ήταν λάθος, αλλά δεν θα εκτύπωνε το επιθυμητό αποτέλεσμα.

Ο τελεστής `instanceof` και η μέθοδος `getClass()`

Ο τελεστής `instanceof` συντάσσεται ως εξής:

```
αντικείμενο instanceof Κλάση
```

και επιστρέφει `true` εάν το αντικείμενο είναι τύπου της συγκεκριμένης Κλάσης και `false` εάν δεν είναι. Προσοχή χρειάζεται στο ότι, όπως προαναφέρθηκε, τα αντικείμενα μιας κλάσης που κληρονομεί κάποια άλλη είναι τύπου της κλάσης αυτής αλλά και τύπου των προγόνων της. Άρα σε κάθε τέτοια περίπτωση αντικειμένου, ο τελεστής `instanceof` θα επιστρέφει `true`.

Αντιθέτως, η μέθοδος `getClass()` της κλάσης `Object` επιστρέφει την κλάση της οποίας

αντικείμενο είναι το αντικείμενο με το οποίο καλείται. Στην ουσία δεν επιστρέφει απαραίτητα τον τύπο του αντικειμένου, αλλά την κλάση με τον κατασκευαστή της οποίας δημιουργήθηκε ένα αντικείμενο. Χαρακτηριστικό είναι το ακόλουθο παράδειγμα. Εάν έχουν δημιουργηθεί τα αντικείμενα:

```
Student s = new Student();
Person ps = new Student();
Person p = new Person();
```

τότε οι ακόλουθες εντολές εκτυπώνουν αυτά που φαίνονται δίπλα σε σχόλια:

```
System.out.println(s instanceof Student);    // -> true
System.out.println(s instanceof Person);     // -> true
System.out.println(ps instanceof Student);   // -> true
System.out.println(ps instanceof Person);    // -> true
System.out.println(p instanceof Person);     // -> true
System.out.println(p instanceof Student);    // -> false
System.out.println(ps.getClass() == s.getClass());  // -> true
System.out.println(ps.getClass() == p.getClass());  // -> false
```

Η λέξη `final`:

α) Σε μεταβλητές: Δεν αλλάζει η τιμή τους μετά την αρχικοποίησή τους (σταθερές).

π.χ.
```
private final int x = 5;
x = 6;  // → ΛΑΘΟΣ!
```

β) Σε μεθόδους: Δεν γίνονται override.

π.χ.
```
public final void specialMethod()
{
  ...
}
```

γ) Σε κλάσεις: Δεν κληρονομούνται.

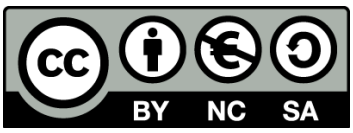
π.χ.
```
public final class MyClass
{
  ...
}
------------------
public class ClassName extends MyClass   // → ΛΑΘΟΣ!
```

Σημειώματα

Σημείωμα Αναφοράς

Μιχάλης Δρακόπουλος. «Πληροφορική ΙΙ. Ενότητα 9: Κληρονομικότητα (Inheritance)». Έκδοση: 1.0. Αθήνα 2014. Διαθέσιμο από τη δικτυακή διεύθυνση: http://opencourses.uoa.gr/courses/MATH106/.

Σημείωμα Αδειοδότησης



[1] http://creativecommons.org/licenses/by-nc-sa/4.0/

Ως **Μη Εμπορική** ορίζεται η χρήση:

- που δεν περιλαμβάνει άμεσο ή έμμεσο οικονομικό όφελος από την χρήση του έργου, για το διανομέα του έργου και αδειοδόχο
- που δεν περιλαμβάνει οικονομική συναλλαγή ως προϋπόθεση για τη χρήση ή πρόσβαση στο έργο
- που δεν προσπορίζει στο διανομέα του έργου και αδειοδόχο έμμεσο οικονομικό όφελος (π.χ. διαφημίσεις) από την προβολή του έργου σε διαδικτυακό τόπο

Ο δικαιούχος μπορεί να παρέχει στον αδειοδόχο ξεχωριστή άδεια να χρησιμοποιεί το έργο για εμπορική χρήση, εφόσον αυτό του ζητηθεί.

Διατήρηση Σημειωμάτων

- Οποιαδήποτε αναπαραγωγή ή διασκευή του υλικού θα πρέπει να συμπεριλαμβάνει:
- το Σημείωμα Αναφοράς
- το Σημείωμα Αδειοδότησης
- τη δήλωση Διατήρησης Σημειωμάτων
- το Σημείωμα Χρήσης Έργων Τρίτων (εφόσον υπάρχει)

μαζί με τους συνοδευόμενους υπερσυνδέσμους.

Χρηματοδότηση

- Το παρόν εκπαιδευτικό υλικό έχει αναπτυχθεί στο πλαίσιο του εκπαιδευτικού έργου του διδάσκοντα.
- Το έργο «**Ανοικτά Ακαδημαϊκά Μαθήματα στο Πανεπιστήμιο Αθηνών**» έχει χρηματοδοτήσει μόνο τη αναδιαμόρφωση του εκπαιδευτικού υλικού.
- Το έργο υλοποιείται στο πλαίσιο του Επιχειρησιακού Προγράμματος «Εκπαίδευση και Δια Βίου Μάθηση» και συγχρηματοδοτείται από την Ευρωπαϊκή Ένωση (Ευρωπαϊκό Κοινωνικό Ταμείο) και από εθνικούς πόρους.