ΤΕΧΝΙΚΕΣ ΑΝΤΙΚΕΙΜΕΝΟΣΤΡΑΦΟΥΣ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΥ

Κληρονομικότητα

Παράδειγμα

- Στο παράδειγμα με το τμήμα πανεπιστημίου οι φοιτητές και οι καθηγητές είχαν κάποια κοινά στοιχεία
  - Και οι δύο είχαν όνομα
  - Και οι δύο είχαν κάποιο χαρακτηριστικό αριθμό
- και κάποιες διαφορές
  - Οι καθηγητές δίδασκαν μαθήματα
  - Οι φοιτητές έπαιρναν μαθήματα, βαθμούς και μονάδες
- Παρομοίως μπορούμε να έχουμε πολλούς διαφορετικούς τύπους φοιτητών ανάλογα με το έτος τους ή τις απαιτήσεις του μαθήματος

- Δεν θα ήταν βολικό αν είχαμε μεθόδους που να χειρίζονταν με κοινό τρόπο τις ομοιότητες (π.χ. εκτύπωση των βασικών στοιχείων) και να έχουν ξεχωριστές μεθόδους για τις διαφορές?
  - Έτσι δεν θα έπρεπε να γράφουμε τον ίδιο κώδικα πολλές φορές και οι αλλαγές θα έπρεπε να γίνουν μόνο μια φορά.

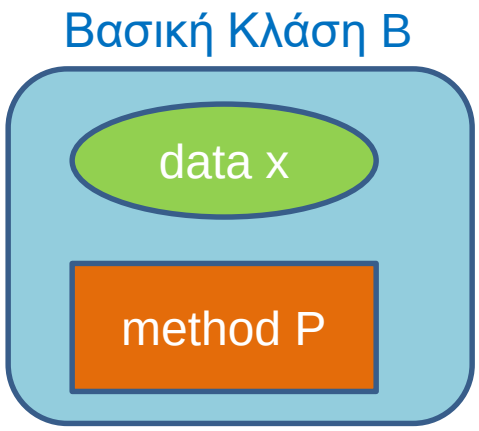
- Αυτό το καταφέρνουμε με την κληρονομικότητα!

Κληρονομικότητα

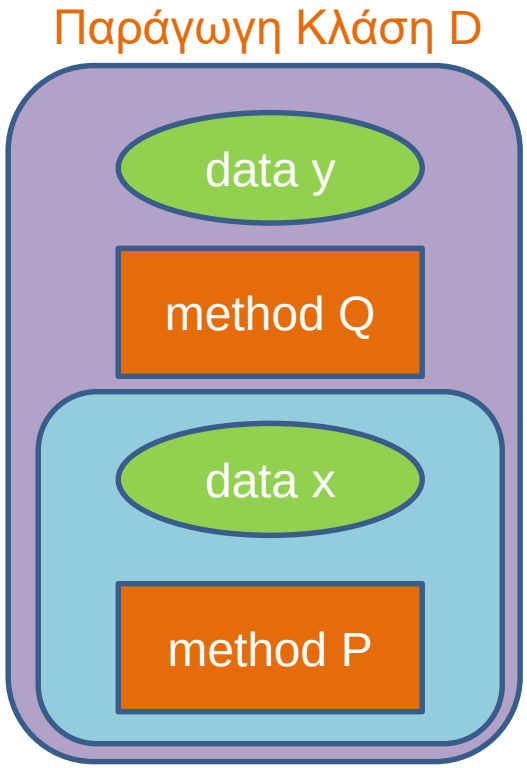
- Η κληρονομικότητα είναι κεντρική έννοια στον αντικειμενοστραφή προγραμματισμό.
- Η ιδέα είναι να ορίσουμε μια γενική κλάση που έχει κάποια χαρακτηριστικά (πεδία και μεθόδους) που θέλουμε και μετά να ορίσουμε εξειδικευμένες παραλλαγές της κλάσης αυτής στις οποίες προσθέτουμε ειδικότερα χαρακτηριστικά.
  - Οι εξειδικευμένες κλάσεις λέμε ότι κληρονομούν τα χαρακτηριστικά της γενικής κλάσης

Κληρονομικότητα

Έχουμε μια Βασική Κλάση (Base Class) Β, με κάποια πεδία και μεθόδους.



Θέλουμε να δημιουργήσουμε μια νέα κλάση D η οποία να έχει όλα τα χαρακτηριστικά της Β, αλλά και κάποια επιπλέον.



Αντί να ξαναγράψουμε τον ίδιο κώδικα δημιουργούμε μια Παράγωγη Κλάση (Derived Class) D, η οποία κληρονομεί όλη τη λειτουργικότητα της Βασικής Κλάσης Β και στην οποία προσθέτουμε τα νέα πεδία και μεθόδους.

Αυτή διαδικασία λέγεται κληρονομικότητα

Κληρονομικότητα

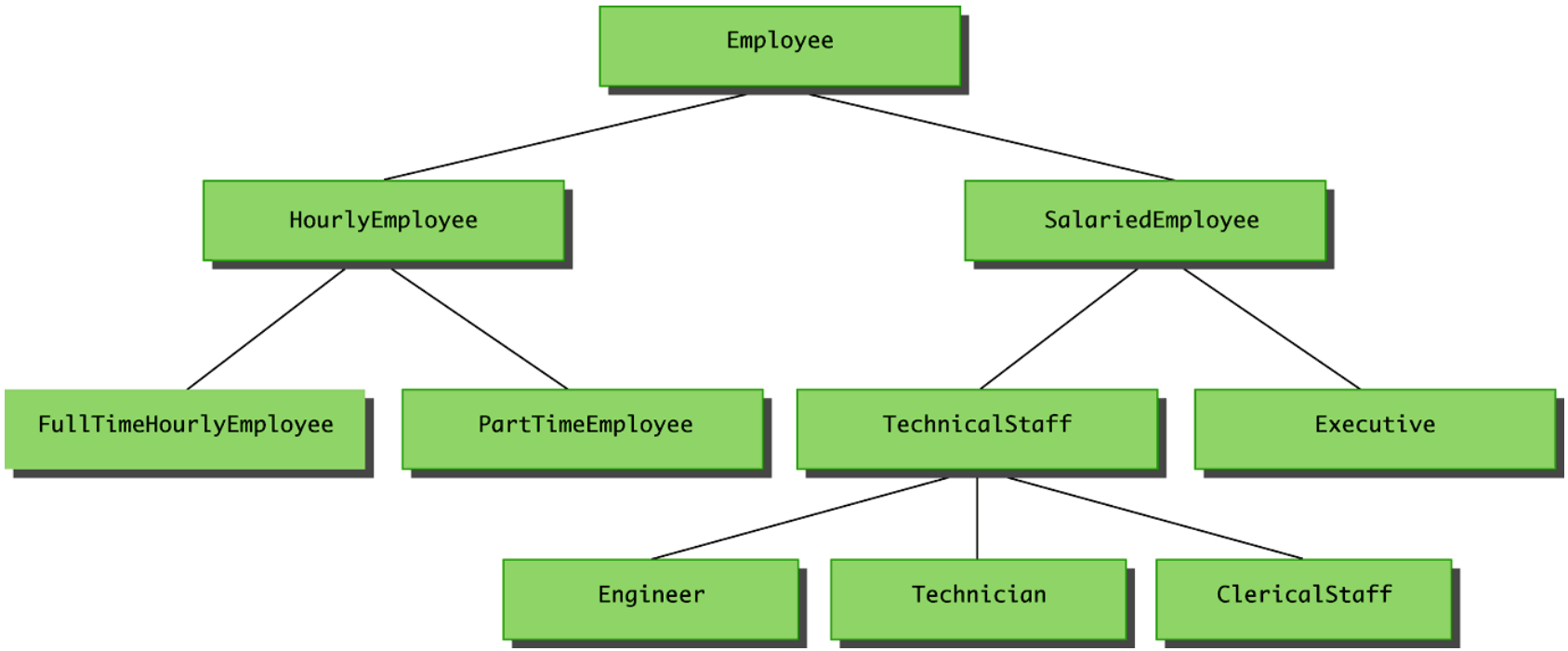
- Η κληρονομικότητα είναι χρήσιμη όταν
  - Θέλουμε να έχουμε αντικείμενα και της κλάσης B και της κλάσης D.
  - Θέλουμε να ορίσουμε πολλαπλές παράγωγες κλάσεις D1, D2, … που η κάθε μία επεκτείνει την B με διαφορετικό τρόπο.
- Μπορούμε να ορίσουμε παράγωγες κλάσεις των παράγωγων κλάσεων.
  - Με αυτό τον τρόπο ορίζεται μια ιεραρχία κλάσεων.

Ιεραρχία κλάσεων (Class Hierarchy)

- Παράδειγμα: Έχουμε ένα πρόγραμμα που διαχειρίζεται τους Εργαζόμενους μιας εταιρίας.
  - Όλοι οι εργαζόμενοι έχουν κοινά χαρακτηριστικά το όνομα τους και το ΑΦΜ τους.
- Οι εργαζόμενοι χωρίζονται σε Ωρομίσθιους και Έμμισθους
  - Διαφορετικά χαρακτηριστικά θα κρατάμε όσον αφορά το μισθό για τον καθένα
- Οι Ωρομίσθιοι χωρίζονται σε Πλήρους και Μερικής απασχόλησης
- Οι Έμμισθοι χωρίζονται σε Τεχνικό Προσωπικό και Διευθυντικό προσωπικό
- Κ.ο.κ.…

A Class Hierarchy

Display 7.1 **A Class Hierarchy**

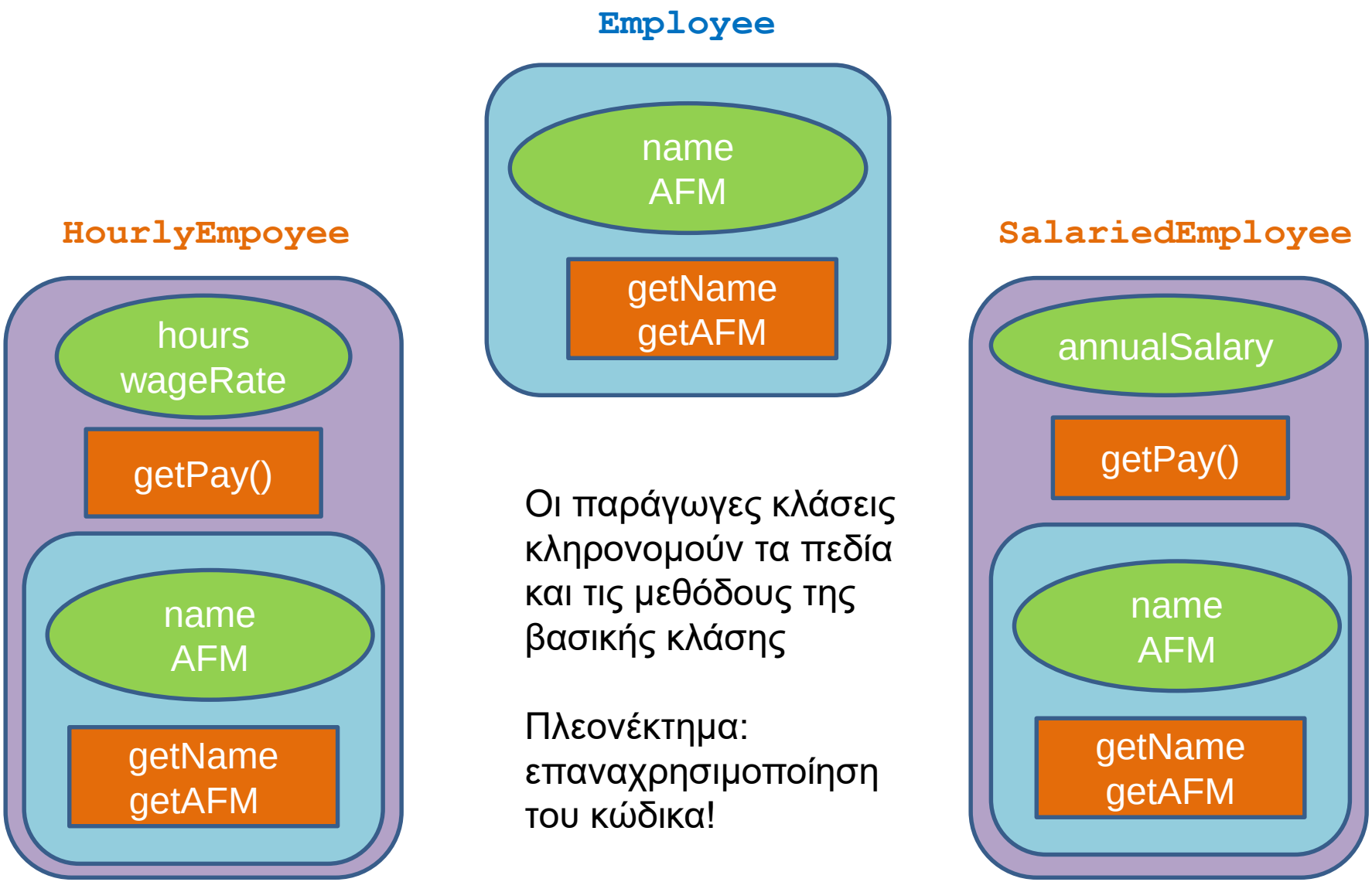


Ιεραρχία κλάσεων

- Η ιεραρχία από κλάσεις ορίζει κάτι σαν γενεαλογικό δέντρο κλάσεων από πιο γενικές προς πιο ειδικές κλάσεις.

- Στη Java όλες οι κλάσεις ανήκουν στην ίδια ιεραρχία.
  - Στην κορυφή της ιεραρχίας είναι η κλάση Object.

Παράδειγμα



Οι παράγωγες κλάσεις κληρονομούν τα πεδία και τις μεθόδους της βασικής κλάσης

Πλεονέκτημα: επαναχρησιμοποίηση του κώδικα!

Ορολογία

- Η βασική κλάση συχνά λέγεται και υπέρ-κλάση (superclass) και η παραγόμενη κλάση υπό-κλάση (subclass).
- Επίσης η βασική κλάση λέμε ότι είναι ο γονέας της παραγόμενης κλάσης, και η παράγωγη κλάση το παιδί της βασικής.
  - Αν έχουμε παραπάνω από ένα επίπεδο κληρονομικότητας στην ιεραρχία, τότε έχουμε πρόγονο και απόγονο κλάση.

Συντακτικό

- Ας πούμε ότι έχουμε την βασική κλάση **Employee** και τις παραγόμενες κλάσεις **HourlyEmployee** και **SalariedEmployee**.
- Για να ορίσουμε τις παραγόμενες κλάσεις χρησιμοποιούμε το εξής συντακτικό στη δήλωση της κλάσης:

```
public class HourlyEmployee extends Employee
public class SalariedEmployee extends Employee
```

Η βασική κλάση

```
public class Employee
{
    private String name;
    private int AFM;

    public Employee( ) { … }

    public Employee(String theName, int theAFM) { … }

    public Employee(Employee originalObject) { … }

    public String getName( ) { … }
    public void setName(String newName) { … }

    public int getAFM( ) { … }
    public void setAFM (int newAFM) { … }

    public String toString() { … }

}
```

Η παράγωγη κλάση HourlyEmployee

```
public class HourlyEmployee extends Employee
{
    private double wageRate;
    private double hours; //for the month

    public HourlyEmployee( ) { … }

    public HourlyEmployee(String theName, int theAFM,
                          double theWageRate, double theHours) { … }

    public HourlyEmployee(HourlyEmployee originalObject) { … }

    public double getRate( ) { … }
    public void setRate(double newWageRate) { … }

    public double getHours( ) { … }
    public void setHours(double hoursWorked) { … }

    public double getPay( ) {
        return wageRate*hours;
    }

    public String toString( ){ … }
}
```

Νέα πεδία για την HourlyEmployee

Μέθοδος getPay υπολογίζει το μηνιαίο μισθό

Η παράγωγη κλάση SalariedEmployee

```
public class SalariedEmployee extends Employee
{
    private double salary; //annual

    public SalariedEmployee( ) { … }

    public SalariedEmployee(String theName,
                            int theAFM, double theSalary) { … }

    public SalariedEmployee(SalariedEmployee originalObject ) { … }

    public double getSalary( ) { … }
    public void setSalary(double newSalary) { … }

    public double getPay( )
    {
        return salary/12;
    }

    public String toString( ) { … }
}
```

Νέα πεδία για την SalariedEmployee

Μέθοδος getPay υπολογίζει το μηνιαίο μισθό. Διαφορετική από την προηγούμενη

```
public class Example1
{
    public static void main(String[] args)
    {
        HourlyEmployee alice = new HourlyEmployee("Alice",
                                                  100, 50.5, 40);

        SalariedEmployee bob = new SalariedEmployee("Bob",
                                                    200, 100000);

        System.out.println("Alice: "
                           + alice.getName() + " "
                           + alice.getAFM() + " "
                           + alice.getPay());

        System.out.println("Bob: "
                           + bob.getName() + " "
                           + bob.getAFM() + " "
                           + bob.getPay());

    }
}
```

Μέθοδοι της Employee

Μέθοδοι των παράγωγων κλάσεων

Constructor

```
public class Employee
{
    private String name;
    private int AFM;


    public Employee()
    {
        name = "no name";
        AFM = 0;
    }

    public Employee(String theName, int theAFM)
    {
        if (theName == null || theAFM <= 0)
        {
            System.out.println("Fatal Error creating employee.");
            System.exit(0);
        }
        name = theName;
        AFM = theAFM;
    }
}
```

```
public class HourlyEmployee extends Employee
{
    private double wageRate;
    private double hours; //for the month


    public HourlyEmployee(String theName, int theAFM,
                          double theWageRate, double theHours)
    {
        super(theName, theAFM);
        if ((theWageRate >= 0) && (theHours >= 0))
        {
            wageRate = theWageRate;
            hours = theHours;
        }
        else
        {
           System.out.println(
                     "Fatal Error: creating an illegal hourly employee.");
           System.exit(0);
        }
    }
}
```

Με τη λέξη κλειδί super αναφερόμαστε στην βασική κλάση.

Εδώ καλούμε τον constructor της Employee με ορίσματα το όνομα και το ΑΦΜ

Ο constructor super μπορεί να κληθεί μόνο στην αρχή της μεθόδου.

```
public class SalariedEmployee extends Employee
{
    private double salary; //annual

    public SalariedEmployee(String theName,
                    int theAFM, double theSalary)
    {
        super(theName, theAFM);
        if (theSalary >= 0)
            salary = theSalary;
        else
        {
            System.out.println(
                 "Fatal Error: Negative salary.");
            System.exit(0);
        }
    }
}
```

```
public class SalariedEmployee extends Employee
{
    private double salary; //annual

    public SalariedEmployee()
    {
        super();
        salary = 0;
    }
}
```

Καλεί τον default constructor της Employee

Η εντολή δεν είναι απαραίτητη σε αυτή την περίπτωση. Αν δεν έχουμε κάποια κλήση προς τον constructor της γονικής κλάσης, τότε καλείται εξ ορισμού ο default constructor της Employee.

```
public class SalariedEmployee extends Employee
{
    private double salary; //annual

    public SalariedEmployee(String theName,int theAFM)
    {
        salary = 0;
    }
}
```

Πως θα αρχικοποιηθεί το αντικείμενο στην περίπτωση που κληθεί αυτός ο constructor?

Εφόσον δεν καλούμε εμείς κάποιο constructor της γονικής κλάσης θα κληθεί ο default constructor ο οποίος θα αρχικοποιήσει το όνομα στο “no name” και το ΑΦΜ στο μηδέν.

```
public class SalariedEmployee extends Employee
{
    private double salary; //annual

    public SalariedEmployee(String theName,int theAFM)
    {
        super(theName, theAFM);
        salary = 0;
    }
}
```

Αν θέλουμε να αρχικοποιήσουμε το όνομα και το ΑΦΜ θα πρέπει να καλέσουμε τον αντίστοιχο constructor της γονικής κλάσης.

Constructor this

- Όπως καλείται ο constructor super της γονικής κλάσης μπορούμε να καλέσουμε και τον constructor this της ίδιας κλάσης.

```
public class SalariedEmployee extends Employee
{
    private double salary; //annual

    public SalariedEmployee(String theName, int theAFM, double theSalary)
    {
        super(theName, theAFM);
        if (theSalary >= 0)
            salary = theSalary;
        else{
           System.out.println("Fatal Error: Negative salary.");
            System.exit(0);
        }
    }

    public SalariedEmployee(){
        this("no name", 0, 0);
    }
}
```

Καλεί ένα άλλο constructor της ίδιας κλάσης

Γιατί να μην κάνουμε κάτι πιο απλό? Κατευθείαν ανάθεση των πεδίων

```
public class SalariedEmployee extends Employee
{
    private double salary; //annual

    public SalariedEmployee(String theName,
                            int theAFM, double theSalary)
    {
        name = theName;
        AFM = theAFM;
        salary = theSalary;
    }
}
```

ΛΑΘΟΣ!

Οι παραγόμενες κλάσεις δεν έχουν πρόσβαση στα private πεδία και τις private μεθόδους της βασικής κλάσεις.

Κληρονομικότητα και ενθυλάκωση

- Οι παραγόμενες κλάσεις κληρονομούν την πληροφορία που έχει και η γονική κλάση
  - Ένα αντικείμενο SalariedEmployee έχει πληροφορία για το όνομα και το ΑΦΜ του υπαλλήλου.

- Δεν έχουν όμως πρόσβαση να διαβάσουν και να αλλάξουν ότι είναι private μέσα στην γονική κλάση.
  - Στην περίπτωση του SalariedEmployee, δεν μπορούμε να αλλάξουμε ή να διαβάσουμε το όνομα. Θα πρέπει να χρησιμοποιήσουμε τις public μεθόδους setName, getName.
  - Για τον constructor πρέπει να καλέσουμε την super.
- Με αυτό τον τρόπο προστατεύουμε τα δεδομένα της γονικής κλάσης από κώδικα εκτός της κλάσης.

- Ο περιορισμός ισχύει και για μεθόδους που είναι private στην γονική κλάση.

```
public class Employee
{
      private void doSomething(){
            System.out.println("doSomething");
      }
}
```

```
public class SalariedEmployee extends Employee
{
      public void doSomethingMore(){
            doSomething();
            System.out.println("and more");
      }
}
```

ΛΑΘΟΣ!

Protected μέλη

- Οι παράγωγες κλάσεις έχουν πρόσβαση σε όλα τα public πεδία και μεθόδους της γενικής κλάσης.
- ΔΕΝ έχουν πρόσβαση στα private πεδία και μεθόδους.
  - Μόνο μέσω public μεθόδων set* και get*

- Protected: αν κάποια πεδία και μέθοδοι είναι protected μπορούν να τα δουν όλοι οι απόγονοι της κλάσης.
  - Το βιβλίο δεν το συνιστά.

- Package access: αν δεν προσδιορίσετε public, private, ή protected access τότε η default συμπεριφορά είναι ότι η μεταβλητή είναι προσβάσιμη από άλλες κλάσεις μέσα στο ίδιο πακέτο.

Employee

```
public class Employee
{
    private String name = "default";
    private Date hireDate = new Date(11,4,2013);


    public Employee()
    {
       System.out.println("empty contructor");
    }

    public Employee(String theName, Date theDate)
    {
       name = theName;
       hireDate = new Date(theDate);
    }
}
```

```
public class HourlyEmployee extends Employee
{
    private double wageRate;
    private double hours; //for the month


    public HourlyEmployee(String theName, Date theDate,
                          double theWageRate, double theHours)
    {
        name = theName;
        hireDate = new Date(theDate);
        wageRate = theWageRate;
        hours = theHours;
    }
}
```

Χτυπάει λάθος η πρόσβαση σε private πεδία.

Employee

```
public class Employee
{
    protected String name = "default";
    protected Date hireDate = new Date(11,4,2013);


    public Employee()
    {
       System.out.println("empty contructor");
    }

    public Employee(String theName, Date theDate)
    {
       name = theName;
       hireDate = new Date(theDate);
    }
}
```

```
public class HourlyEmployee extends Employee
{
    private double wageRate;
    private double hours; //for the month


    public HourlyEmployee(String theName, Date theDate,
                          double theWageRate, double theHours)
    {
        name = theName;
        Date = new (theDate);
        wageRate = theWageRate;
        hours = theHours;
    }
}
```

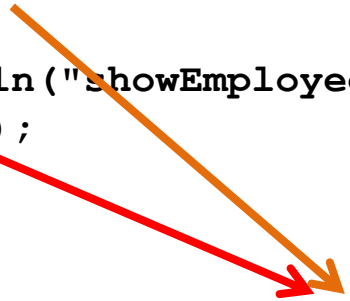
ΟΚ η πρόσβαση σε protected πεδία.

Πολλαπλοί τύποι

- Ένα αντικείμενο της παράγωγης κλάσης έχει και τον τύπο της βασικής κλάσης
  - Ένας HourlyEmployee είναι και Employee
  - Υπάρχει μία is-a σχέση μεταξύ των κλάσεων.

- Αυτό μπορούμε να το εκμεταλλευτούμε χρησιμοποιώντας την βασική κλάση όταν θέλουμε να χρησιμοποιήσουμε κάποια από τις παράγωγες.

```
public class IsADemo
{
    public static void main(String[] args)
    {
        HourlyEmployee alice = new HourlyEmployee("Alice",
                                                  100, 50.5, 40);
        SalariedEmployee bob = new SalariedEmployee("Bob",
                                                    200, 100000);

        System.out.println("showEmployee(alice):");
        showEmployee(alice);

        System.out.println("showEmployee(bob):");
        showEmployee(bob);

    }

    public static void showEmployee(Employee employeeObject)
    {
            System.out.println(employeeObject.getName( ));
            System.out.println(employeeObject.getAFM( ));
    }
}
```



Μπορούμε να καλέσουμε τη μέθοδο και με HourlyEmployee και με SalariedEmployee γιατί και οι δύο είναι και Employee.

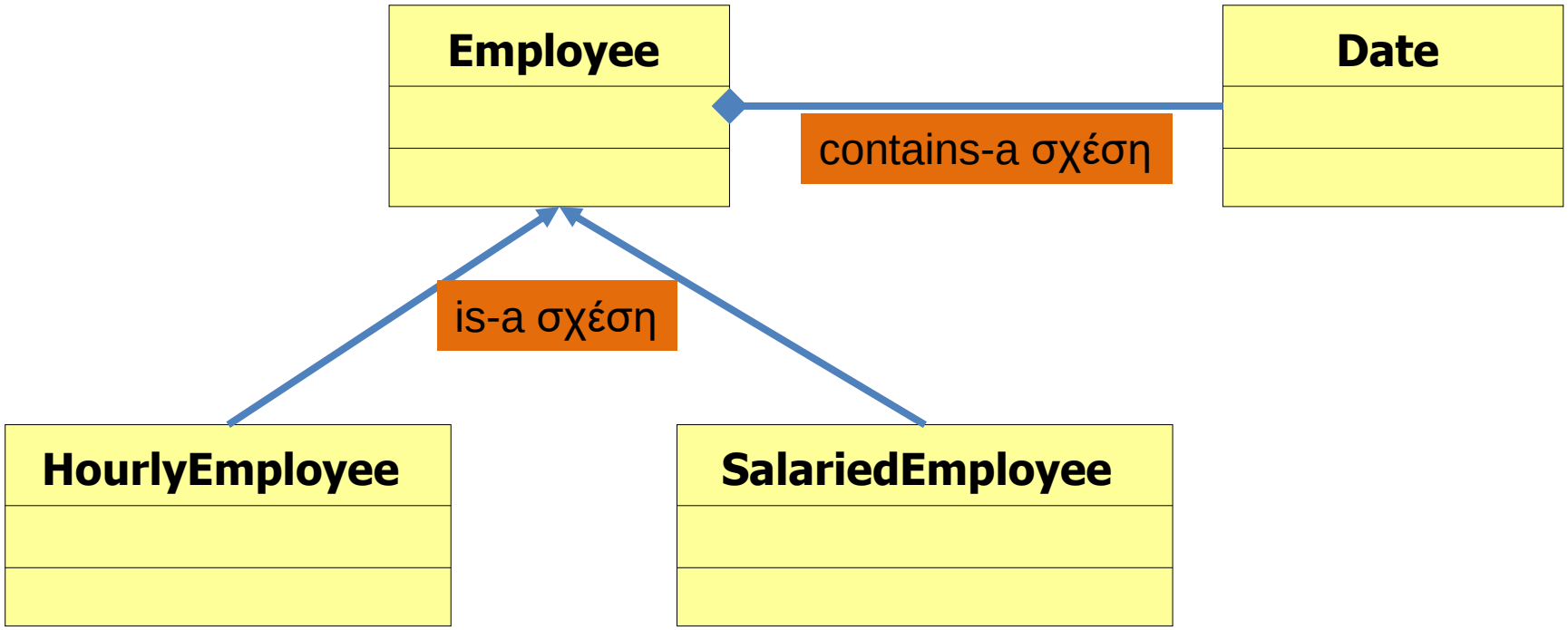
```
public class Employee
{
        private String name;
        private int AFM;

        public Employee(Employee other){
                this.name = other.name;
                this.AFM = other.AFM;
        }
}
```

```
public class SalariedEmployee extends Employee
{
        public SalariedEmployee(SalariedEmployee other){
                super(other);
                this.salay = other.salary;        }
}
```

Η κλήση του copy constructor της Employee (μέσω της super(other)) γίνεται με ένα αντικείμενο τύπου SalariedEmployee. Αυτό γίνεται γιατί SalariedEmployee is a Employee και το αντικείμενο other έχει και τους δύο τύπους.

UML διάγραμμα

- Αναπαράσταση κληρονομικότητας



METHOD OVERRIDING
ΥΠΕΡΒΑΣΗ ΜΕΘΟΔΩΝ

Υπέρβαση μεθόδων (method overriding)

- Μία μέθοδος που ορίζεται στην βασική κλάση μπορούμε να την ξανα-ορίσουμε στην παράγωγη κλάση με διαφορετικό τρόπο
  - Παράδειγμα: η μέθοδος `toString()`. Την ξανα-ορίζουμε για κάθε παραγόμενη κλάση ώστε να παράγει αυτό που θέλουμε
  - Αυτό λέγετε υπέρβαση της μεθόδου (method overriding).
- Η υπέρβαση των μεθόδων είναι διαφορετική από την υπερφόρτωση.
  - Στην υπερφόρτωση αλλάζουμε την υπογραφή της μεθόδου.
  - Εδώ έχουμε την ίδια υπογραφή, απλά αλλάζει ο ορισμός στην παραγόμενη κλάση.

```
public class Employee
{
    private String name;
    private int AFM;

    public Employee( ) { … }

    public Employee(String theName, int theAFM) { … }

    public Employee(Employee originalObject) { … }

    public String getName( ) { … }
    public void setName(String newName) { … }

    public Date getAFM( ) { … }
    public void setAFM(int AFM) { … }

    public String toString()
    {
        return (name + " " + AFM);
    }

}
```

```
public class HourlyEmployee extends Employee
{
    private double wageRate;
    private double hours; //for the month

    public HourlyEmployee( ) { … }

    public HourlyEmployee(String theName, int theAFM,
                          double theWageRate, double theHours) { … }

    public HourlyEmployee(HourlyEmployee originalObject) { … }

    public double getRate( ) { … }
    public void setRate(double newWageRate) { … }

    public double getHours( ) { … }
    public void setHours(double hoursWorked) { … }

    public double getPay( ) {
        return wageRate*hours;
    }

    public String toString( ){
       return (getName( ) + " " + getAFM( )
            + "\n$" + wageRate + " per hour for " + hours + " hours");
    }
}
```

```
public class SalariedEmployee extends Employee
{
    private double salary; //annual

    public SalariedEmployee( ) { … }

    public SalariedEmployee(String theName,
                            int theAFM, double theSalary) { … }

    public SalariedEmployee(SalariedEmployee originalObject ) { … }

    public double getSalary( ) { … }
    public void setSalary(double newSalary) { … }

    public double getPay( )
    {
        return salary/12;
    }

    public String toString( ) { … }
    {
        return (getName( ) + " " + getAFM( )
                                  + "\n$" + salary + " per year");
    }
}
```

```
public class SalariedEmployee extends Employee
{
    private double salary; //annual

    public SalariedEmployee( ) { … }

    public SalariedEmployee(String theName,
                            int theAFM, double theSalary) { … }

    public SalariedEmployee(SalariedEmployee originalObject ) { … }

    public double getSalary( ) { … }
    public void setSalary(double newSalary) { … }

    public double getPay( )
    {
        return salary/12;
    }

    public String toString( ) { … }
    {
        return (super.toString( ) + "\n$" + salary + " per year");
    }
}
```

Έτσι καλούμε την toString της βασικής κλάσης
Πιο καλή υλοποίηση, μπορεί να έχει φωλιασμένες κλήσεις από προγονικές κλάσεις

super

- Το keyword super χρησιμοποιείται σαν αντικείμενο κλήσης για να καλέσουμε μια μέθοδο της γονικής κλάσης την οποία έχουμε κάνει override.
  - Π.χ., super.toString() για να καλέσουμε την toString της Employee.

- Αν θέλουμε να το ξεχωρίσουμε από την κλήση της toString της SalariedEmployee, μπορούμε να χρησιμοποιήσουμε το this. Μέσα στην SalariedEmployee:
  - super.toString() καλεί την toString της Employee
  - this.toString() καλεί την toString της SalariedEmployee

- Προσοχή: Δεν μπορούμε να έχουμε αλυσιδωτές κλήσεις του super.
  - super.super.toString() είναι λάθος!

Παράδειγμα

```
public class InheritanceDemo
{
    public static void main(String[] args)
    {
       SalariedEmployee sam = new SalariedEmployee("Sam",
                                                   100, 100000);
       HourlyEmployee han = new HourlyEmployee("Han",
                                                 200, 50.5, 40);
       Employee eve = new Employee("Eve",300);

       System.out.println(eve);

       System.out.println(sam);

       System.out.println(han);
       }
}
```

Καλεί τη μέθοδο της Employee

Καλεί τη μέθοδο της SalariedEmployee

Καλεί τη μέθοδο της HourlyEmployee

Υπέρβαση και αλλαγή επιστρεφόμενου τύπου

- Μια αλλαγή που μπορούμε να κάνουμε στην υπογραφή της κλάσης που υπερβαίνουμε είναι να αλλάξουμε τον επιστρεφόμενο τύπο σε αυτόν μιας παράγωγης κλάσης
  - Ουσιαστικά δεν είναι αλλαγή αφού η παράγωγη κλάση έχει και τον τύπο της γονικής κλάσης.

```
public class Employee
{
        private String name;
        private Date hireDate;

        public Employee createCopy()
        {
          return new Employee(this);
        }
}
```

```
public class HourlyEmployee extends Employee
{
    private double wageRate;
    private double hours; //for the month

    public HourlyEmployee createCopy()
    {
        return new HourlyEmployee(this);
    }
}
```

Ο επιστρεφόμενος τύπος αλλάζει από Employee σε HourlyEmployee στην υπέρβαση. Ουσιαστικά όμως δεν υπάρχει αλλαγή μιας και κάθε αντικείμενο HourlyEmployee είναι και Employee

```
public class SalariedEmployee extends
Employee
{
    private double salary; //annual

    public SalariedEmployee createCopy()
    {
        return new SalariedEmployee(this);
    }

}
```

Ο επιστρεφόμενος τύπος αλλάζει από Employee σε SalariedEmployee στην υπέρβαση. Ουσιαστικά όμως δεν υπάρχει αλλαγή μιας και κάθε αντικείμενο SalariedEmployee είναι και Employee

toString και equals

- Είπαμε ότι η Java για κάθε αντικείμενο «περιμένει» να δει τις μεθόδους toString και equals
  - Αυτό σημαίνει ότι οι μέθοδοι αυτές ορίζονται στην κλάση Object που είναι ο πρόγονος όλων το κλάσεων και κάθε νέα κλάση μπορεί να τις υπερβεί (override).
  - Είδαμε παραδείγματα πως υπερβήκαμε την μέθοδο toString.

equals

- Η equals στην κλάση Object ορίζεται ως:
  - `public boolean equals(Object other)`
- Για την κλάση Employee θα την ορίσουμε ως:
  - `public boolean equals(Employee other)`
- Αλλάζουμε την υπογραφή της κλάσης, άρα δεν κάνουμε υπέρβαση, αλλά υπερφόρτωση της equals
  - Πως θα την ορίσουμε ώστε να κάνουμε υπέρβαση?

Overriding equals

```
public class Employee
{
    private String name;
    private Date hireDate;

    public boolean equals(Object otherObject)
    {
        if (otherObject == null)
            return false;
        else if (getClass( ) != otherObject.getClass( ))
            return false;
        else
        {
            Employee otherEmployee = (Employee)otherObject;
            return (name.equals(otherEmployee.name)
               && hireDate.equals(otherEmployee.hireDate));
        }
    }
}
```

getClass: μέθοδος της Object, επιστρέφει μια αναπαράσταση της κλάσης του αντικειμένου

Downcasting: μετατροπή ενός αντικειμένου από μια υψηλότερη σε μία χαμηλότερη κλάση

Το downcasting δεν είναι πάντα δυνατόν και αν δεν γίνει σωστά μπορεί να προκαλέσει λάθη κατά την εκτέλεση του προγράμματος

DOWNCASTING

Downcasting

```
public class DowncastingExample
{
    public static void main(String[] args)
    {
                SalariedEmployee sam = new SalariedEmployee("Sam",
                                new Date(1, 1, 2010), 100000);
                Employee eve = new Employee("Eve", new Date(1,1,2012));

                SalariedEmployee eve2 = eve;
                if (sam.getHireDate().equals(eve2.getHireDate())){
                        System.out.println("Same hire date");
                }else{
                        System.out.println("Different hire date");
                }
        }
}
```

Στην περίπτωση αυτή προσπαθούμε να κάνουμε το downcasting έμμεσα, αναθέτοντας μια μεταβλητή Employee σε μια μεταβλητή SalariedEmployee.
Θα μας χτυπήσει λάθος κατά την μεταγλώτιση.

Downcasting

```
public class DowncastingExample
{
    public static void main(String[] args)
    {
                SalariedEmployee sam = new SalariedEmployee("Sam",
                                    new Date(1, 1, 2010), 100000);
                Employee eve = new Employee("Eve", new Date(1,1,2012));

                SalariedEmployee eve2 = (SalariedEmployee)eve;
                if (sam.getHireDate().equals(eve2.getHireDate())){
                        System.out.println("Same hire date");
                }else{
                        System.out.println("Different hire date");
                }
        }
}
```

Στην περίπτωση αυτή θα μας χτυπήσει λάθος στο τρέξιμο παρότι χρησιμοποιούμε μόνο την κοινή μέθοδο getHireDate(). Το πρόγραμμα προβλέπει ότι μπορεί να υπάρχει πρόβλημα. Δεν γίνεται να μετατρέψουμε έναν Employee σε SalariedEmployee (ο Employee δεν έχει όλα τα πεδία που χρειάζεται ένας SalariedEmployee)

Downcasting

```
public class DowncastingExample
{
    public static void main(String[] args)
    {
                SalariedEmployee sam = new SalariedEmployee("Sam",
                                    new Date(1, 1, 2010), 100000);
                Employee eve = new Employee("Eve", new Date(1,1,2012));

                method(sam,sam);

        }

        private static void method(SalariedEmployee sEmp, Employee emp){
                SalariedEmployee sEmp2 = (SalariedEmployee) emp;

                if (sEmp.getHireDate().equals(sEmp2.getSalary())){
                        System.out.println("Same Salary");
                }else{
                        System.out.println("Different salary");
                }
        }
}
```

Στην περίπτωση αυτή το downcasting δεν χτυπάει λάθος γιατί υπάρχει η δυνατότητα να καλέσουμε σωστά την μέθοδο με SalariedEmployee αντικείμενο

Downcasting

```
public class DowncastingExample
{
    public static void main(String[] args)
    {
                SalariedEmployee sam = new SalariedEmployee("Sam",
                                    new Date(1, 1, 2010), 100000);
                Employee eve = new Employee("Eve", new Date(1,1,2012));

                method(sam,eve);

        }

        private static void method(SalariedEmployee sEmp, Employee emp){
                SalariedEmployee sEmp2 = (SalariedEmployee) emp;

                if (sEmp.getHireDate().equals(sEmp2.getSalary())){
                        System.out.println("Same Salary");
                }else{
                        System.out.println("Different salary");
                }
        }
}
```

Αν όμως την καλέσουμε με αντικείμενο Employee θα πάρουμε λάθος

```
import java.util.Random;

public class DowncastingExample2
{
    public static void main(String[] args)
    {
        SalariedEmployee[] sEmployees = new SalariedEmployee[4];
        sEmployees[0] = new SalariedEmployee("employee 100",new Date(1,1,2015),1000);
        sEmployees[1] = new SalariedEmployee("employee 101",new Date(2,1,2015),2000);
        sEmployees[2] = new SalariedEmployee("employee 102",new Date(3,1,2015),3000);
        sEmployees[3] = new SalariedEmployee("employee 103",new Date(4,1,2015),4000);

        SalariedEmployee rand = (SalariedEmployee)randomSelection(sEmployees);
        System.out.println(rand);
        System.out.println("Salary per month " + rand.getPay());
    }

    private static Employee randomSelection(Employee[] employees){
        Random rndGen = new Random();
        int r = rndGen.nextInt(employees.length);
        return employees[r];
    }
}
```

Σε τι μας χρειάζεται το downcasting?

Θέλουμε να καλέσουμε την μέθοδο getPay για τυπώσουμε τον μηνιαίο μισθό. Χρειαζόμαστε downcasting

Έχουμε μια γενική μέθοδο randomSelection που επιλέγει ένα τυχαίο στοιχείο από ένα πίνακα με Employee. Θέλουμε να την χρησιμοποιήσουμε σε ένα πίνακα με SalariedEmployee

Upcasting

- Η ανάθεση στην αντίθετη κατεύθυνση (upcasting) μπορεί να γίνει χωρίς να χρειάζεται casting
  - Μπορούμε να κάνουμε μια ανάθεση x = y δύο αντικειμένων αν:
    - τα δύο αντικείμενα να είναι της ίδιας κλάσης ή
    - η κλάση του αντικειμένου που ανατίθεται (y) είναι απόγονος της κλάσης του αντικειμένου στο οποίο γίνεται η ανάθεση (x)

- Για παράδειγμα, ο παρακάτω κώδικας δουλεύει χωρίς πρόβλημα:
  - `Employee anEmployee;`
  - `Hourly Employee hEmployee = new HourlyEmployee();`
  - `anEmployee = hEmployee;`

```
public class IsADemo
{
    public static void main(String[] args)
    {
        HourlyEmployee alice = new HourlyEmployee("Alice",
                                                  100, 50.5, 40);
        SalariedEmployee bob = new SalariedEmployee("Bob",
                                                    200, 100000);

        System.out.println("showEmployee(alice):");
        showEmployee(alice);

        System.out.println("showEmployee(bob):");
        showEmployee(bob);

    }

    public static void showEmployee(Employee employeeObject)
    {
            System.out.println(employeeObject.getName( ));
            System.out.println(employeeObject.getAFM( ));
    }
}
```

Όταν καλούμε την **showEmployee** έμμεσα κάνουμε τις αναθέσεις:

employeeObject = alice

employeeObject = bob