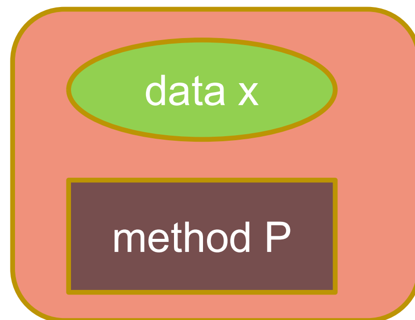
16. ΚΛΗΡΟΝΟΜΙΚΟΤΗΤΑ ΙΙ

Κληρονομικότητα

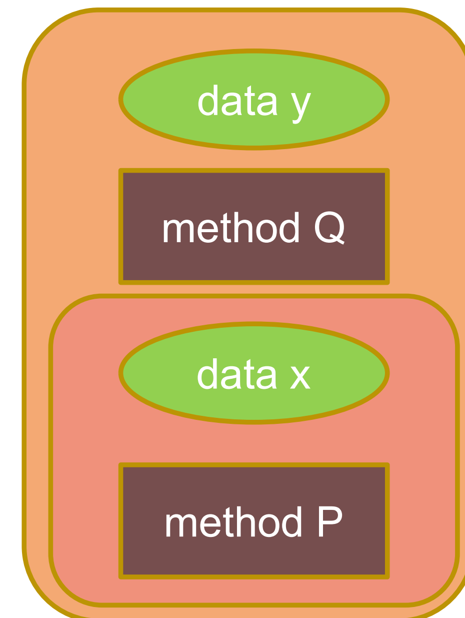
Έχουμε μια Βασική Κλάση (Base Class) B, με κάποια πεδία και μεθόδους.

Βασική Κλάση B



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

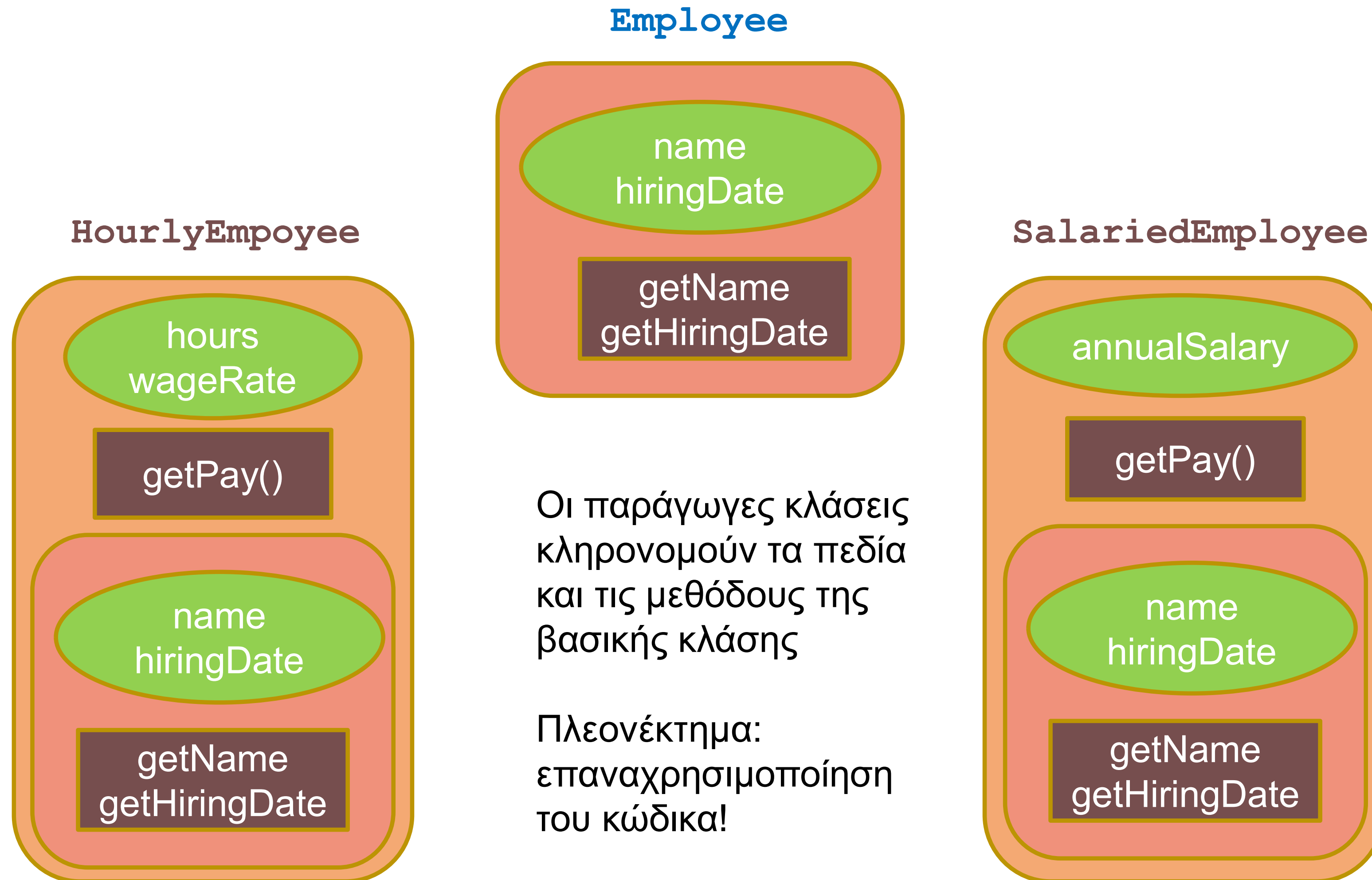
Παράγωγη Κλάση D



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

Αυτή διαδικασία λέγεται κληρονομικότητα

Παράδειγμα



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

- H 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

```
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

Σε τι μας χρειάζεται το downcasting?

```
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];
    }
}
```

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
```

```
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);
    }
}
```

Τι θα τυπώσει η **showEmployee** όταν την καλέσουμε με ορίσματα το alice και το bob?
Ποια μέθοδος **toString** θα κληθεί?

```java
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);
    }
}
```

Θα καλέσει την **toString** της κλάσης του αντικειμένου που περνάμε σαν όρισμα (**HourlyEmployee** ή **SalariedEmployee**) και όχι την κλάση που εμφανίζεται στον ορισμό της παραμέτρου (**Employee**).

Ο μηχανισμός αυτός ονομάζεται late binding (και/ή πολυμορφισμός)

Late Binding (καθυστερημένη δέσμευση)

- Η δέσμευση (binding) αναφέρεται στον συσχετισμό μεταξύ της κλήσης μιας μεθόδου και του ορισμού (κώδικα) της μεθόδου.

- Early binding: Η δέσμευση γίνεται κατά τη μεταγλώττιση του προγράμματος
  - Στην περίπτωση αυτή η μέθοδος toString() που θα κληθεί θα είναι η μέθοδος της κλάσης **`Employee`** μιας και όταν γίνεται η μεταγλώττιση ο compiler βλέπει το όρισμα ως αντικείμενο της κλάσης **`Employee`**.
- Late binding: Η δέσμευση γίνεται κατά τη εκτέλεση του προγράμματος
  - Το κάθε αντικείμενο έχει πληροφορία για την κλάση του και τον ορισμό (κώδικα) των μεθόδων του.
  - Στην περίπτωση αυτή η μέθοδος toString() που θα κληθεί εξαρτάται από την κλάση που περνάμε σαν όρισμα (**`Employee, HourlyEmployee`** ή **`SalariedEmployee`**). Ανάλογα με το αντικείμενο καλείται η ανάλογη μέθοδος.

- Στη Java εφαρμόζεται ο μηχανισμός του late binding για όλες τις μεθόδους (σε αντίθεση με άλλες γλώσσες προγραμματισμού).

Παράδειγμα

```
public class Example3
{
    public static void main(String[] args)
    {
        Employee employeeArray[] = new Employee[3];

        employeeArray[0] = new Employee("alice",
                                new Date(1,1,2010));

        employeeArray[1] = new HourlyEmployee("bob",
                            new Date(1,1,2011), 20, 160);

        employeeArray[2] = new SalariedEmployee("charlie",
                            new Date(1,1,2012), 24000);

       for (int i = 0; i < 3; i ++){
               System.out.println(employeeArray[i]);
        }
    }
}
```

Για κάθε στοιχείο του πίνακα καλείται διαφορετική μέθοδος toString ανάλογα με το αντικείμενο που τοποθετήσαμε σε εκείνη τη θέση

```java
public class mySale
{
    protected String name;
    protected double price;

    public mySale(String theName, double thePrice){
        name = theName;
        price = thePrice;
    }

    public String toString( ){
        return (name + " Price and total cost = $" + price);
    }

    public double bill( ){
        return price;
    }

    public boolean equalDeals(mySale otherSale){
        return (name.equals(otherSale.name)
                && this.bill( ) == otherSale.bill( ));
    }

    public boolean lessThan (mySale otherSale){
        return (this.bill( ) < otherSale.bill( ));
    }
}
```

Σύμφωνα με το βιβλίο δεν συνίσταται η χρήση της protected αλλά την χρησιμοποιούμε για απλότητα στο παράδειγμα

```java
public class myDiscountSale extends mySale
{
    private double discount;

    public myDiscountSale(String theName,
                                double thePrice, double theDiscount)
    {
        super(theName, thePrice);
        discount = theDiscount;
    }

    public double bill( )
    {
        double fraction = discount/100;
        return (1 - fraction)*price;
    }


    public String toString( )
    {
        return (name + " Price = $" + price
                + " Discount = " + discount + "%\n"
                + "   Total cost = $" + bill( ));
    }
}
```

Υπέρβαση της μεθόδου bill()

Δεν έχουμε υπέρβαση των μεθόδων equalDeals και lessThan

```
public class myLateBindingDemo
{
    public static void main(String[] args)
    {
        mySale simple = new mySale("floor mat", 10.00);//One item at $10.00.
        myDiscountSale discount = new myDiscountSale("floor mat", 11.00, 10);
                                  //One item at $11.00 with a 10% discount.

        System.out.println(simple);
        System.out.println(discount);

        if (discount.lessThan(simple))
            System.out.println("Discounted item is cheaper.");
        else
            System.out.println("Discounted item is not cheaper.");

        mySale regularPrice = new mySale("cup holder", 9.90);//One item at $9.90.
        myDiscountSale specialPrice = new myDiscountSale("cup holder", 11.00, 10);
                                  //One item at $11.00 with a 10% discount.

        System.out.println(regularPrice);
        System.out.println(specialPrice);

        if (specialPrice.equalDeals(regularPrice))
            System.out.println("Deals are equal.");
        else
            System.out.println("Deals are not equal.");
    }
}
```

Οι lessThan και equalDeals κληρονομούνται από την mySale

Με το μηχανισμό του late binding στην κλήση τους ξέρουμε ότι το αντικείμενο που τις καλεί είναι τύπου myDiscountSale

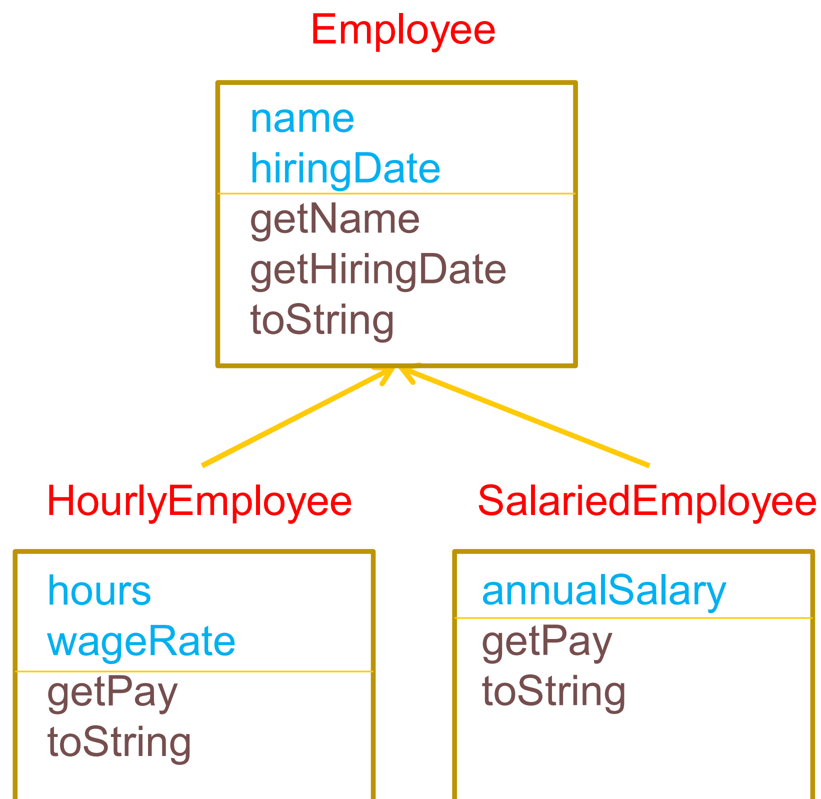
Ξέρουμε λοιπόν ότι όταν εκτελούμε τον κώδικα της lessThan και equalDeals η μέθοδος bill() που θα πρέπει να καλέσουμε είναι αυτή της myDiscountSale ενώ για το otherSale.bill() είναι αυτή της mySale

17. ΚΛΗΡΟΝΟΜΙΚΟΤΗΤΑ ΙΙΙ

Αφηρημένες κλάσεις

Interfaces – διεπαφές

Κληρονομικότητα



Οι παράγωγες κλάσεις κληρονομούν τα πεδία και τις μεθόδους της βασικής κλάσης και έχουν και δικά τους πεδία και μεθόδους.

Επίσης μπορούμε να υπερβαίνουμε (override) κάποιες μεθόδους (toString)

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

    public String toString(){
        return (name + " " + hireDate.toString( ));
    }
}
```

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

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

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

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

```java
public class IsADemo
{
    public static void main(String[] args)
    {
        SalariedEmployee sam = new SalariedEmployee("Josephine",
                                   new Date("January", 1, 2004), 100000);
        HourlyEmployee han = new HourlyEmployee("Sam",
                               new Date("February", 1, 2003), 50.50, 40);

        System.out.println("showEmployee(sam) invoked:");
        showEmployee(sam);

        System.out.println("showEmployee(han) invoked:");
        showEmployee(han);

    }

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

Τι θα τυπώσει η **showEmployee** όταν την καλέσουμε με ορίσματα το sam και το **han**?
Ποια μέθοδος **toString** θα κληθεί?

```java
public class IsADemo
{
    public static void main(String[] args)
    {
        SalariedEmployee sam = new SalariedEmployee("Josephine",
                                   new Date("January", 1, 2004), 100000);
        HourlyEmployee han = new HourlyEmployee("Sam",
                               new Date("February", 1, 2003), 50.50, 40);

        System.out.println("showEmployee(sam) invoked:");
        showEmployee(sam);

        System.out.println("showEmployee(han) invoked:");
        showEmployee(han);

    }

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

Θα καλέσει την **toString** της κλάσης του αντικειμένου που περνάμε σαν όρισμα (**HourlyEmployee** ή **SalariedEmployee**) και όχι την κλάση που εμφανίζεται στον ορισμό της παραμέτρου (**Employee**).

Ο μηχανισμός αυτός ονομάζεται late binding (και/ή πολυμορφισμός)

Late Binding (καθυστερημένη δέσμευση)

- Η δέσμευση (binding) αναφέρεται στον συσχετισμό μεταξύ της κλήσης μιας μεθόδου και του ορισμού (κώδικα) της μεθόδου.

- Early binding: Η δέσμευση γίνεται κατά τη μεταγλώττιση του προγράμματος
  - Στην περίπτωση αυτή η μέθοδος toString() που θα κληθεί θα είναι η μέθοδος της κλάσης `Employee` μιας και όταν γίνεται η μεταγλώττιση ο compiler βλέπει το όρισμα ως αντικείμενο της κλάσης `Employee`.
- Late binding: Η δέσμευση γίνεται κατά τη εκτέλεση του προγράμματος
  - Το κάθε αντικείμενο έχει πληροφορία για την κλάση του και τον ορισμό (κώδικα) των μεθόδων του.
  - Στην περίπτωση αυτή η μέθοδος toString() που θα κληθεί εξαρτάται από την κλάση που περνάμε σαν όρισμα (`Employee`, `HourlyEmployee` ή `SalariedEmployee`). Ανάλογα με το αντικείμενο καλείται η ανάλογη μέθοδος.

- Στη Java εφαρμόζεται ο μηχανισμός του late binding για όλες τις μεθόδους (σε αντίθεση με άλλες γλώσσες προγραμματισμού).

Παράδειγμα

```
public class Example3
{
    public static void main(String[] args)
    {
        Employee employeeArray[] = new Employee[3];

        employeeArray[0] = new Employee("alice",
                                        new Date(1,1,2010));

        employeeArray[1] = new HourlyEmployee("bob",
                                  new Date(1,1,2011), 20, 160);

        employeeArray[2] = new SalariedEmployee("charlie",
                                  new Date(1,1,2012), 24000);

       for (int i = 0; i < 3; i ++){
                System.out.println(employeeArray[i]);
        }
    }
}
```

Για κάθε στοιχείο του πίνακα καλείται διαφορετική μέθοδος toString ανάλογα με το αντικείμενο που τοποθετήσαμε σε εκείνη τη θέση

```
public class Sale
{
    protected String name;
    protected double price;

    public Sale(String theName, double thePrice){
        name = theName;
        price = thePrice;
    }

    public String toString( ){
        return (name + " Price and total cost = $" + price);
    }

    public double bill( ){
        return price;
    }

    public boolean equalDeals(Sale otherSale){
        return (name.equals(otherSale.name)
                && this.bill( ) == otherSale.bill( ));
    }

    public boolean lessThan (Sale otherSale){
        return (this.bill( ) < otherSale.bill( ));
    }
}
```

Σύμφωνα με το βιβλίο δεν συνίσταται η χρήση της protected αλλά την χρησιμοποιούμε για απλότητα στο παράδειγμα

```java
public class DiscountSale extends Sale
{
    private double discount;

    public DiscountSale(String theName,
                        double thePrice, double theDiscount)
    {
        super(theName, thePrice);
        discount = theDiscount;
    }

    public double bill( )
    {
        double fraction = discount/100;
        return (1 - fraction)*price;
    }


    public String toString( )
    {
        return (name + " Price = $" + price
                + " Discount = " + discount + "%\n"
                + "   Total cost = $" + bill( ));
    }
}
```

Υπέρβαση της μεθόδου bill()

Δεν έχουμε υπέρβαση των μεθόδων equalDeals και lessThan

```java
public class LateBindingDemo
{
    public static void main(String[] args)
    {
        Sale simple = new Sale("floor mat", 10.00);//One item at $10.00.
        DiscountSale discount = new DiscountSale("floor mat", 11.00, 10);
                                  //One item at $11.00 with a 10% discount.

        System.out.println(simple);
        System.out.println(discount);

        if (discount.lessThan(simple))
            System.out.println("Discounted item is cheaper.");
        else
            System.out.println("Discounted item is not cheaper.");

        Sale regularPrice = new Sale("cup holder", 9.90);//One item at $9.90.
        DiscountSale specialPrice = new DiscountSale("cup holder", 11.00, 10);
                                  //One item at $11.00 with a 10% discount.

        System.out.println(regularPrice);
        System.out.println(specialPrice);

        if (specialPrice.equalDeals(regularPrice))
            System.out.println("Deals are equal.");
        else
            System.out.println("Deals are not equal.");
    }
}
```

Οι `lessThan` και `equalDeals` κληρονομούνται από την `Sale`

Με το μηχανισμό του late binding στην κλήση τους ξέρουμε ότι το αντικείμενο που τις καλεί είναι τύπου `DiscountSale`

Ξέρουμε λοιπόν ότι όταν εκτελούμε τον κώδικα της `lessThan` και `equalDeals` η μέθοδος `bill()` που θα πρέπει να καλέσουμε είναι αυτή της `DiscountSale` ενώ για το `otherSale.bill()` είναι αυτή της `Sale`

Ένα διαφορετικό πρόβλημα

- Ας υποθέσουμε ότι στην Employee θέλουμε να προσθέσουμε μια μέθοδο που ελέγχει αν δύο υπάλληλοι έχουν τον ίδιο μισθό (ανεξάρτητα αν είναι ωρομίσθιοι, ή πλήρους απασχόλησης)
- Η συνάρτηση είναι απλή:

```
public boolean sameSalary(Employee other)
{
        if (this.getPay() == other.getPay()){
                return true;
        }
        return false
}
```

- Το πρόβλημα: Που θα την ορίσουμε?
  - Ιδανικά στην Employee, αλλά η Employee δεν έχει συνάρτηση **getPay()**
  - Αν την ορίσουμε στην HourlyEmployee, ή στην SalariedEmployee, δεν μπορούμε να περάσουμε όρισμα Employee εφόσον δεν έχει μέθοδο **getPay()**

Αφηρημένες μέθοδοι

- Η λύση είναι να ορίσουμε την `getPay()` ως αφηρημένη μέθοδο (abstract method) της Employee.
  - `public abstract double getPay();`
  - Μια αφηρημένη μέθοδος δηλώνεται σε μία κλάση αλλά ορίζεται στις παράγωγες κλάσεις.
  - Χρησιμοποιούμε τη δεσμευμένη λέξη abstract για να δηλώσουμε ότι μια μέθοδος είναι αφηρημένη.
  - Η δήλωση μιας αφηρημένης μεθόδου δεν έχει κώδικα οπότε η εντολή τερματίζει με το ;
  - Οι αφηρημένες μέθοδοι πρέπει να είναι public (ή protected), όχι private.

Αφηρημένες κλάσεις

- Οι κλάσεις που περιέχουν μια αφηρημένη μέθοδο ορίζονται υποχρεωτικά ως αφηρημένες κλάσεις (abstract classes)
  - `public abstract class Employee { … }`
- Δεν μπορούμε να δημιουργήσουμε αντικείμενα μιας αφηρημένης κλάσης
  - Μια αφηρημένη κλάση χρησιμοποιείται μόνο για να δημιουργούμε παράγωγες κλάσεις.
  - Στην περίπτωση μας δεν χρειαζόμαστε αντικείμενα τύπου Employee. Ένας υπάλληλος θα είναι είτε ωρομίσθιος, είτε μόνιμος.
- Οι παράγωγες κλάσεις μιας αφηρημένης κλάσης θα πρέπει πάντα να ορίζουν τις αφηρημένες μεθόδους
  - Εκτός αν είναι και αυτές αφηρημένες.
- Μια κλάση (ή μέθοδος) που δεν είναι αφηρημένη λέγεται ενυπόστατη (concrete)

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

    public abstract double getPay();

    public boolean samePay(Employee other){
        return (this.getPay() == other.getPay());
    }

    public Employee( ) { … }

    public Employee(String theName, Date theDate) { … }

    public Employee(Employee originalObject) { … }

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

    public Date getHireDate( ) { … }
    public void setHireDate(Date newDate) { … }

    public String toString()

}
```

Ορισμός της αφηρημένης κλάσης

Ορισμός της αφηρημένης μεθόδου

Χρήση της αφηρημένης μεθόδου και της αφηρημένης κλάσης

Όταν καλέσουμε την **samePay** θα την καλέσουμε με ένα αντικείμενο μιας από τις παράγωγες κλάσεις.

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

    public HourlyEmployee( ) { … }

    public HourlyEmployee(String theName, Date theDate,
                    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( ){ … }
}
```

Εφόσον η κλάση HourlyEmployee παράγεται από αφηρημένη κλάση και η ίδια δεν είναι αφηρημένη, πρέπει υποχρεωτικά να ορίσει την αφηρημένη μέθοδο getPay

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

    public SalariedEmployee( ) { … }

    public SalariedEmployee(String theName,
                            Date theDate, double theSalary) { … }

    public SalariedEmployee(SalariedEmployee originalObject ) { … }

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

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

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

Εφόσον η κλάση SalariedEmployee παράγεται από αφηρημένη κλάση και η ίδια δεν είναι αφηρημένη, πρέπει υποχρεωτικά να ορίσει την αφηρημένη μέθοδο getPay

```
public class Example
{
  public static void main(String args[]){
        HourlyEmployee A = new HourlyEmployee("Alice",
                new Date(4,18,2013), 10, 100);
        SalariedEmployee B = new SalariedEmployee("Bob",
                new Date(4,17,2013), 12000);
        if (A.samePay(B)){
             System.out.println("The two employees
                        earn the same amount per month");
        }
        else{
             System.out.println("The two employees do NOT
                        earn the same amount per month");
        }
  }
}
```

```
public class Example
{
  public static void main(String args[]){
        Employee A = new HourlyEmployee(“Alice",
                new Date(4,18,2013), 10, 100);
        Employee B = new SalariedEmployee(“Bob",
                new Date(4,17,2013), 12000);
        if (A.samePay(B)){
             System.out.println("The two employees
                         earn the same amount per month");
        }
        else{
             System.out.println("The two employees do NOT
                         earn the same amount per month");
        }
  }
}
```

Μπορούμε να ορίσουμε μεταβλητές αφηρημένης κλάσης. Θα πρέπει να όμως να τους αναθέσουμε αντικείμενα μιας από τις παράγωγες ενυπόστατες κλάσεις. Δεν μπορούμε να ορίσουμε ένα αντικείμενο της αφηρημένης κλάσης.

```
public class Example
{
  public static void main(String args[]){
        HourlyEmployee A = new HourlyEmployee("Alice",
               new Date(4,18,2013), 10, 100);
        SalariedEmployee B = new SalariedEmployee("Bob",
               new Date(4,17,2013), 12000);
        compareAndPrint(A,B)
  }

  private static void compareAndPrint(Employee A, Employee B){
        if (A.samePay(B)){
            System.out.println("The two employees
                      earn the same amount per month");
        }
        else{
            System.out.println("The two employees do NOT
                      earn the same amount per month");
        }
  }
}
```

Αφηρημένες κλάσεις

- Αφηρημένες κλάσεις είναι οι κλάσεις που περιέχουν αφηρημένες μεθόδους
  - Η υλοποίηση των αφηρημένων μεθόδων μετατίθεται στις μη αφηρημένες (ενυπόστατες – concrete) κλάσεις που είναι απόγονοι μιας αφηρημένης κλάσης.
  - Η υλοποίηση είναι υποχρεωτική. Άρα έτσι εξασφαλίζουμε ότι μια concrete κλάση θα έχει την μέθοδο που θέλουμε.
- Οι αφηρημένες κλάσεις εκτός από αφηρημένες μεθόδους έχουν και πεδία και ενυπόστατες μεθόδους.
  - Κληρονομούν επιπλέον χαρακτηριστικά στους απογόνους τους, όχι μόνο τις αφηρημένες μεθόδους.

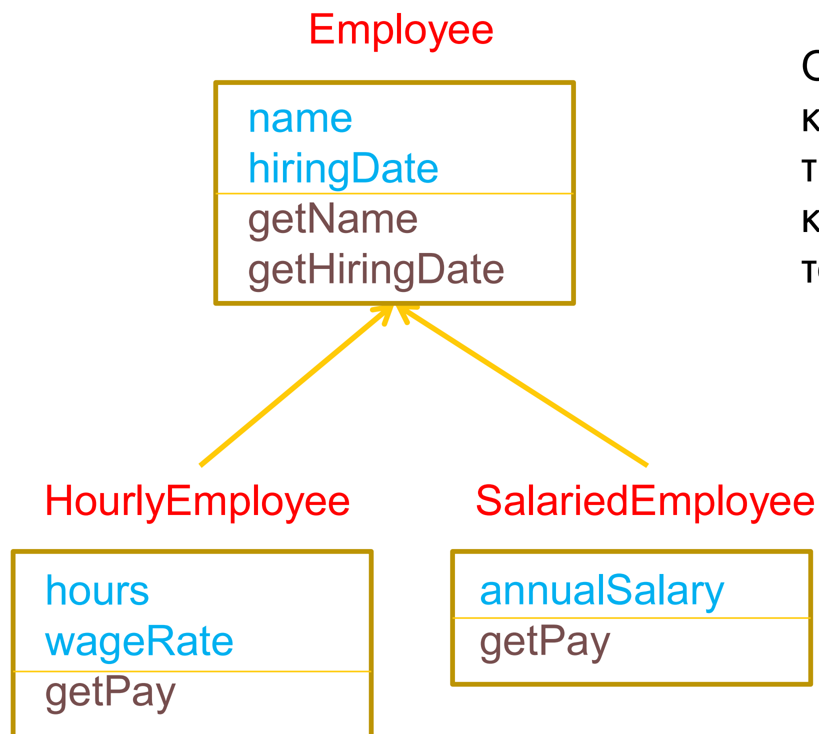
18. ΚΛΗΡΟΝΟΜΙΚΟΤΗΤΑ IV

Πολυμορφισμός – Αφηρημένες κλάσεις

Interfaces (διεπαφές)

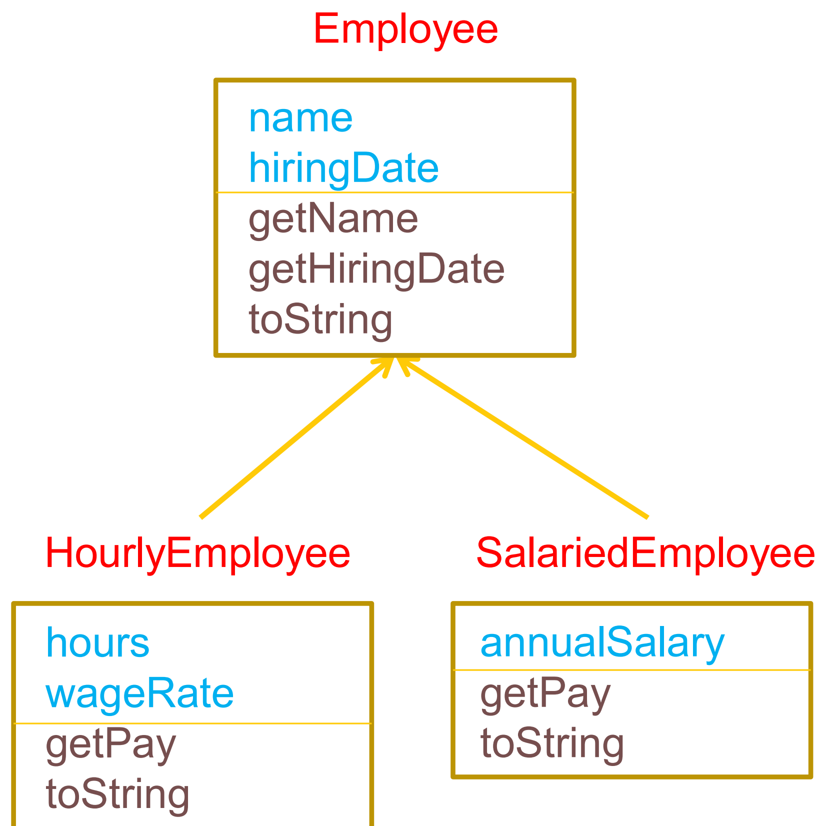
Παραδείγματα

Κληρονομικότητα



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

Late Binding

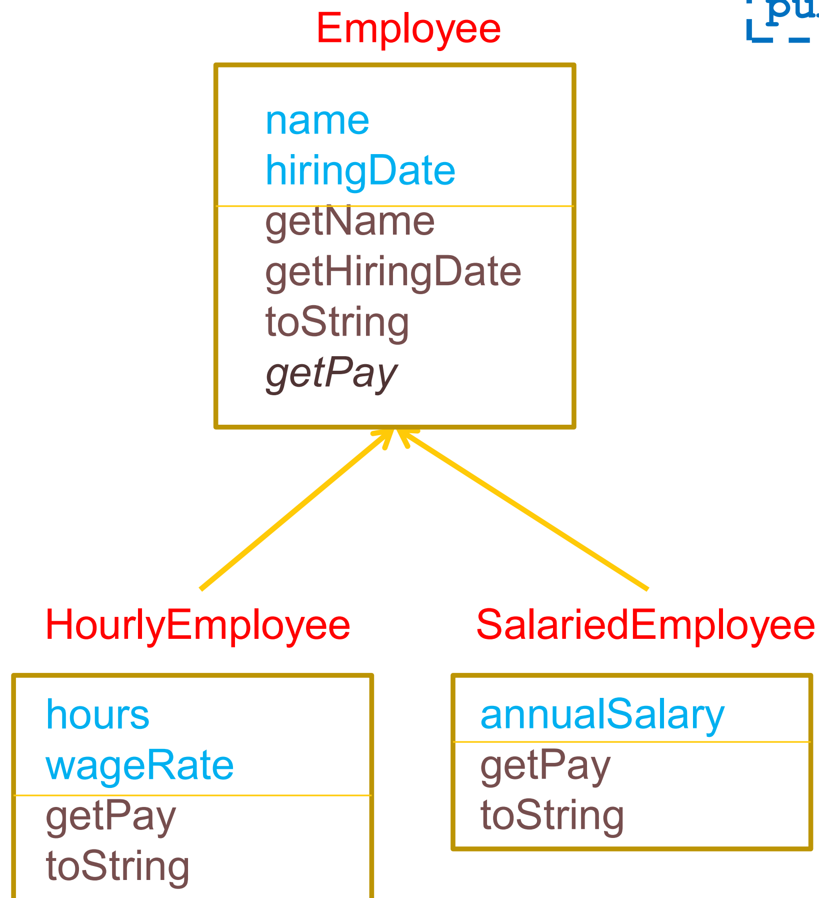



```
Employee e;
e = new HourlyEmployee();
System.out.println(e);
e = new SalariedEmployee();
System.out.println(e);
```

Late Binding:

Ο κώδικας που εκτελείται για την `toString()` εξαρτάται από την κλάση του αντικειμένου την ώρα της κλήσης (HourlyEmployee ή SalariedEmployee) και όχι την ώρα της δήλωσης (Employee)

Αφηρημένες κλάσεις




```
public abstract double getPay();
```

Μια αφηρημένη μέθοδος δηλώνεται σε μια γενική κλάση και ορίζεται σε μια πιο εξειδικευμένη κλάση

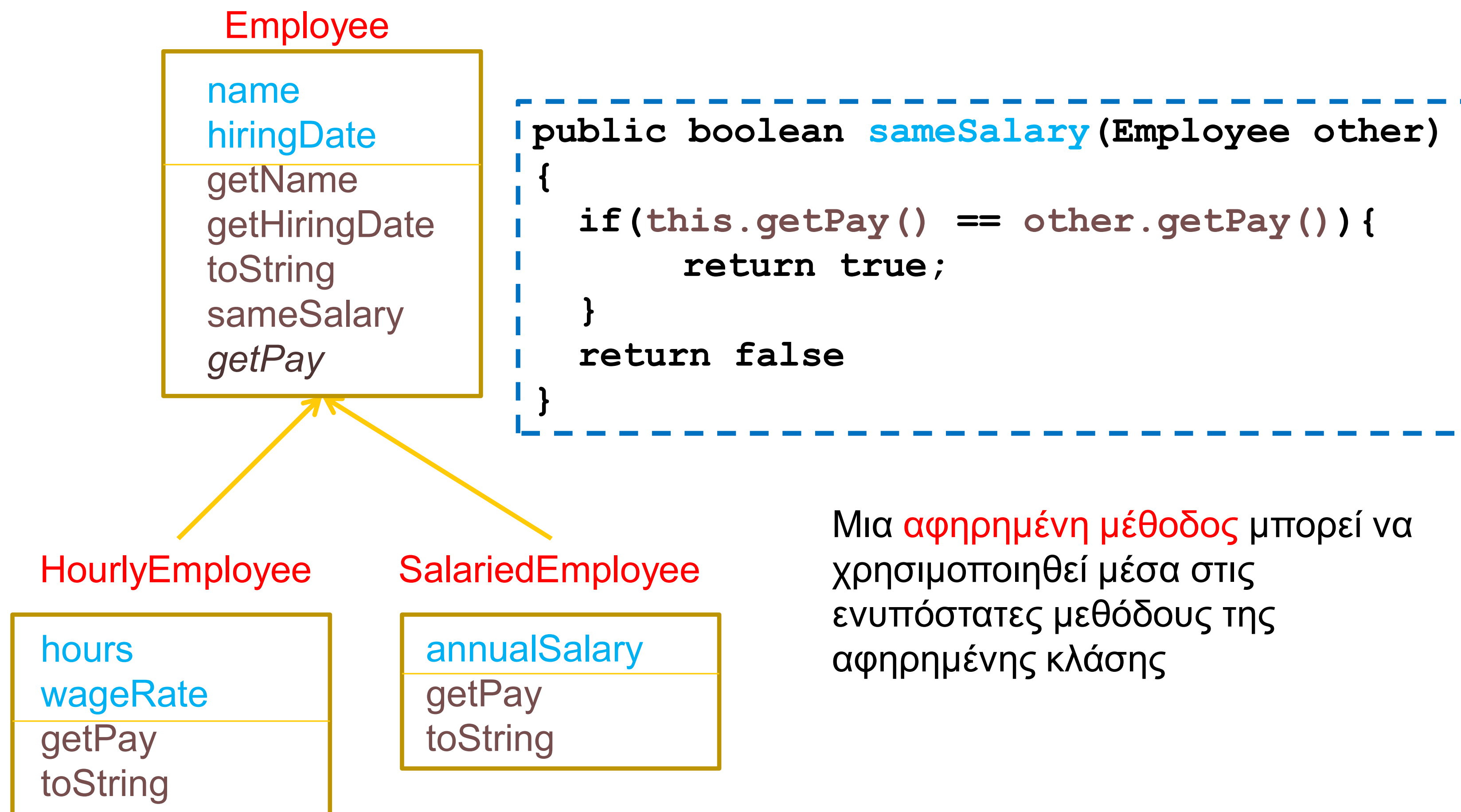
Οι κλάσεις με αφηρημένες μεθόδους είναι αφηρημένες κλάσεις.

Δεν μπορούμε να δημιουργήσουμε αντικείμενα αφηρημένων κλάσεων.

- Δηλαδή δεν μπορούμε να κάνουμε `new Employee()` εφόσον η Employee είναι αφηρημένη

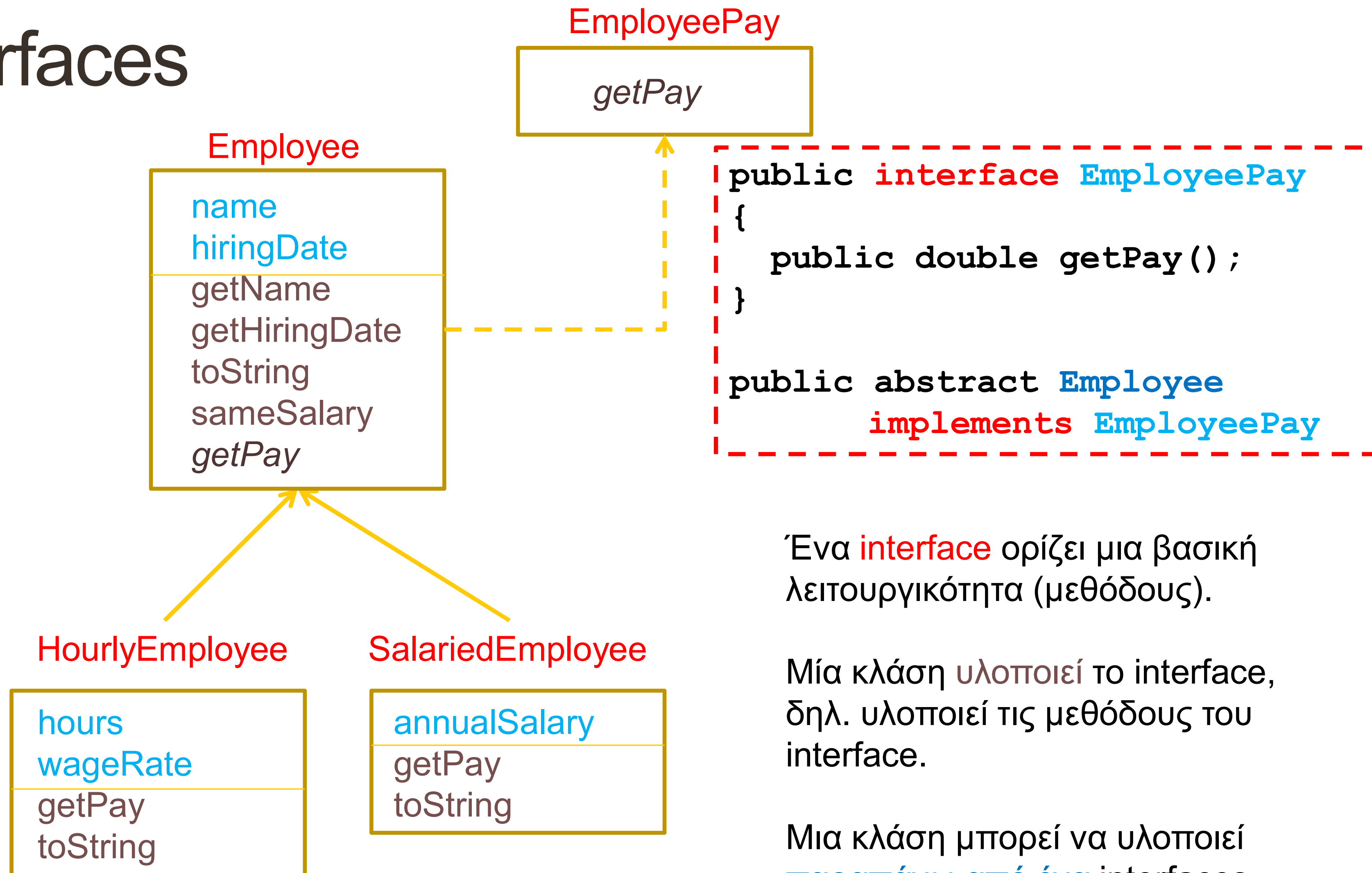
Οι παράγωγες ενυπόστατες κλάσεις πρέπει να υλοποιούν τις αφηρημένες μεθόδους.

Αφηρημένες κλάσεις



Μια αφηρημένη μέθοδος μπορεί να χρησιμοποιηθεί μέσα στις ενυπόστατες μεθόδους της αφηρημένης κλάσης

Interfaces



Βρείτε τα λάθη

- Στο πρόγραμμα στην επόμενη διαφάνεια υπάρχουν διάφορα λάθη
  - Ποια είναι?

```java
public abstract class Vehicle
{
  private int position = 0;

  public Vehicle(int pos){
        position = pos;
  }

  public abstract void move();

  public void print()
  {
    System.out.println("position = "
                           + position);
  }
}
```

```java
public class Car extends Vehicle
{
  private int gas;

  public Car(int pos, int gas){
    position = pos;
    this.gas = gas;
  }

  public void drive(){
    position += 10;
    gas -= 10;
  }

  public int getGas(){
    return gas;
  }

  public void print(){
    super.print();
    System.out.println("gas =" + gas);
  }
}
```

```java
public class Example
{
  public static void main(String[] args){
    Vehicle[] V = new Vehicle[3];
    V[0] = new Car(0,100);
    V[1] = new Bike();
    V[2] = new Vehicle(0);
    V[0].drive(); V[0].print();
    V[1].move(); V[1].print();
    int gas = V[0].getGas();
  }
}
```

```java
public class Bike extends Vehicle
{
  public void move(){
        position ++;
  }
}
```

```java
public abstract class Vehicle
{
  private int position = 0;

  public Vehicle(int pos){
        position = pos;
  }

  public abstract void move();

  public void print()
  {
    System.out.println("position = "
                              + position);
  }
}
```

```java
public class Example
{
  public static void main(String[] args){
    Vehicle[] V = new Vehicle[3];
    V[0] = new Car(0,100);
    V[1] = new Bike();
    V[2] = new Vehicle(0);
    V[0].drive(); V[0].print();
    V[1].move(); V[1].print();
    int gas = V[0].getGas();
  }
}
```

```java
public class Car extends Vehicle
{
  private int gas;

  public Car(int pos, int gas){
    position = pos;
    this.gas = gas;
  }

  public void drive(){
    position += 10;
    gas -= 10;
  }

  public int getGas(){
    return gas;
  }

  public void print(){
    super.print();
    System.out.println("gas =" + gas);
  }
}
```

```java
public class Bike extends Vehicle
{
  public void move(){
        position ++;
  }
}
```

```java
public abstract class Vehicle
{
   protected int position = 0;

   public Vehicle(){
   }

   public Vehicle(int pos){
         position = pos;
   }

   public int getPosition{
         return position
   }

   public void setPosition(int pos){
         position = pos;
   }

   public abstract void move();

   public void print()
   {
     System.out.println("position = "
                              + position);
   }
}
```

Το πεδίο position πρέπει να είναι protected εφόσον το χρησιμοποιούν και οι παράγωγες κλάσεις ή να ορίσουμε getPosition και setPosition μεθόδους

Πρέπει να ορίσουμε και ένα κενό constructor, ή να καλούμε την super μέσα στις παράγωγες κλάσεις.

```java
public class Car extends Vehicle
{
  private int gas;

  public Car(int pos, int gas){
    position = setPosition(pos);
    this.gas = gas;
  }


  public void move(){
    setPosition(getPosition() + 10);
    gas -= 10;
  }

  public int getGas(){
    return gas;
  }

  public void print(){
    super.print();
    System.out.println("gas =" + gas);
  }

}
```

O constructor δουλεύει μόνο αν έχουμε constructor χωρίς ορίσματα στην Vehicle. Αλλιώς χρειαζόμαστε αυτό τον constructor:

```java
public Car(int pos, int gas){
    super(pos);
    this.gas = gas;
 }
```

H Car πρέπει να υλοποίει την μέθοδο move

```
public class Bike extends Vehicle
{
  public void move(){
      position ++;
   }
}
```

O constructor (ή μάλλον η έλλειψη του) δουλεύει μόνο αν έχουμε constructor χωρίς ορίσματα στην Vehicle. Αλλιώς χρειαζόμαστε αυτό τον constructor:

```
 public Bike(){
     super(0);
}
```

```java
public class Example
{
  public static void main(String[] args){
    Vehicle[] V = new Vehicle[2];
    V[0] = new Car(0,100);
    V[1] = new Bike();
    //V[2] = new Vehicle(0);
    V[0].move(); V[0].print();
    V[1].move(); V[1].print();
    int gas = ((Car)V[0]).getGas()
  }
}
```

Δεν μπορούμε να δημιουργήσουμε αντικείμενο τύπου Vehicle γιατί είναι αφηρημένη κλάση.

Η Vehicle δεν έχει μέθοδο getGas.
Για να την καλέσουμε θα πρέπει να κάνουμε downcast το αντικείμενο V[0] σε Car.

Ερωτήσεις:

- Υπάρχει πρόβλημα με την εντολή `Vehicle[] V = new Vehicle[2];` ?
- Ποια print καλείται για το αντικείμενο V[0]? Ποια για το V[1]? Γιατί?
- Τι θα τυπώσει το πρόγραμμα?

Υπάρχει κάποιο λάθος σε αυτό τον ορισμό?

```
public abstract class EngineVehicle extends Vehicle
{
	protected int gas;

	public EngineVehicle(int pos, int gas){
	   super(pos);
	   this.gas = gas;
	}
}
```

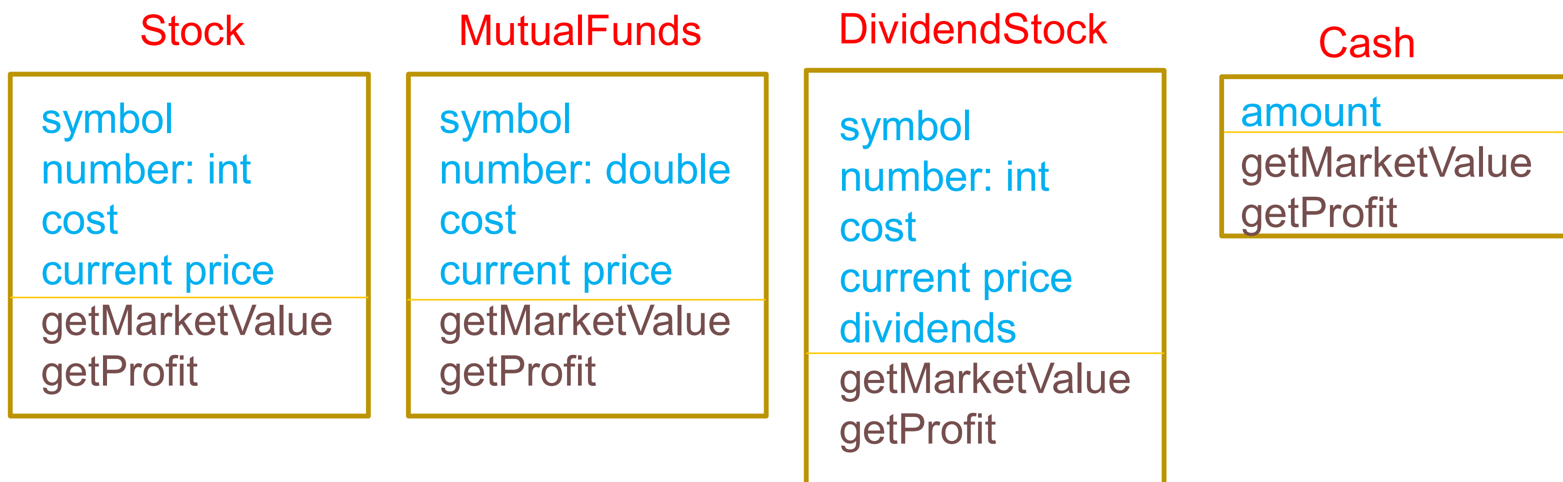
Όχι. Εφόσον η EngineVehicle είναι αφηρημένη δεν χρειάζεται να ορίσουμε την αφηρημένη μέθοδο move

Ένα μεγάλο παράδειγμα

- Θέλουμε να φτιάξουμε ένα πρόγραμμα που διαχειρίζεται το πορτοφόλιο (portofolio) ενός χρηματιστή. Το portofolio έχει μετοχές (stocks), μετοχές που δίνουν μέρισμα (divident stocks), αμοιβαία κεφάλαια (mutual funds), και χρήματα (cash). Για κάθε μια από αυτές τις αξίες (assets) θέλουμε να υπολογίζουμε την τωρινή της αποτίμηση (market value) και το κέρδος (profit) που μας δίνει. Μετά θέλουμε να υπολογίσουμε τη συνολική αξία του πορτοφόλιου και το συνολικό κέρδος

Λεπτομέρειες

- Cash: Δεν μεταβάλλεται η αξία του, δεν έχει κέρδος
- Stocks: Η αξία του είναι ίση με τον αριθμό των μετοχών επί την αξία της μετοχής. Το κέρδος είναι η διαφορά της τωρινής αποτίμησης με το κόστος αγοράς
- Mutual Funds: Παρόμοια με τα Stocks αλλά ο αριθμός των μετοχών που μπορούμε να έχουμε είναι πραγματικός αριθμός αντί για ακέραιος
- Dividend Stocks: Όμοια με τα Stocks αλλά στο κέρδος προσθέτουμε και τα μερίσματα

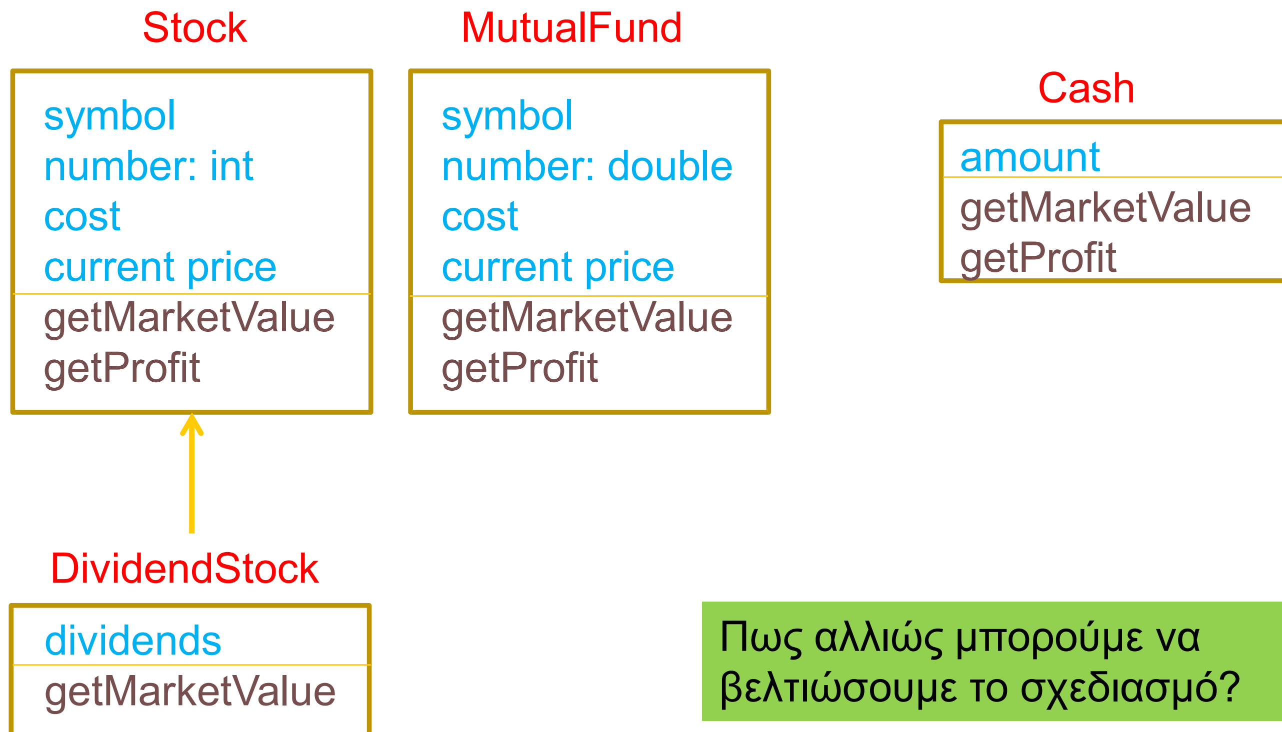


Πως μπορούμε να βελτιώσουμε το σχεδιασμό των κλάσεων?

Σχεδιασμός

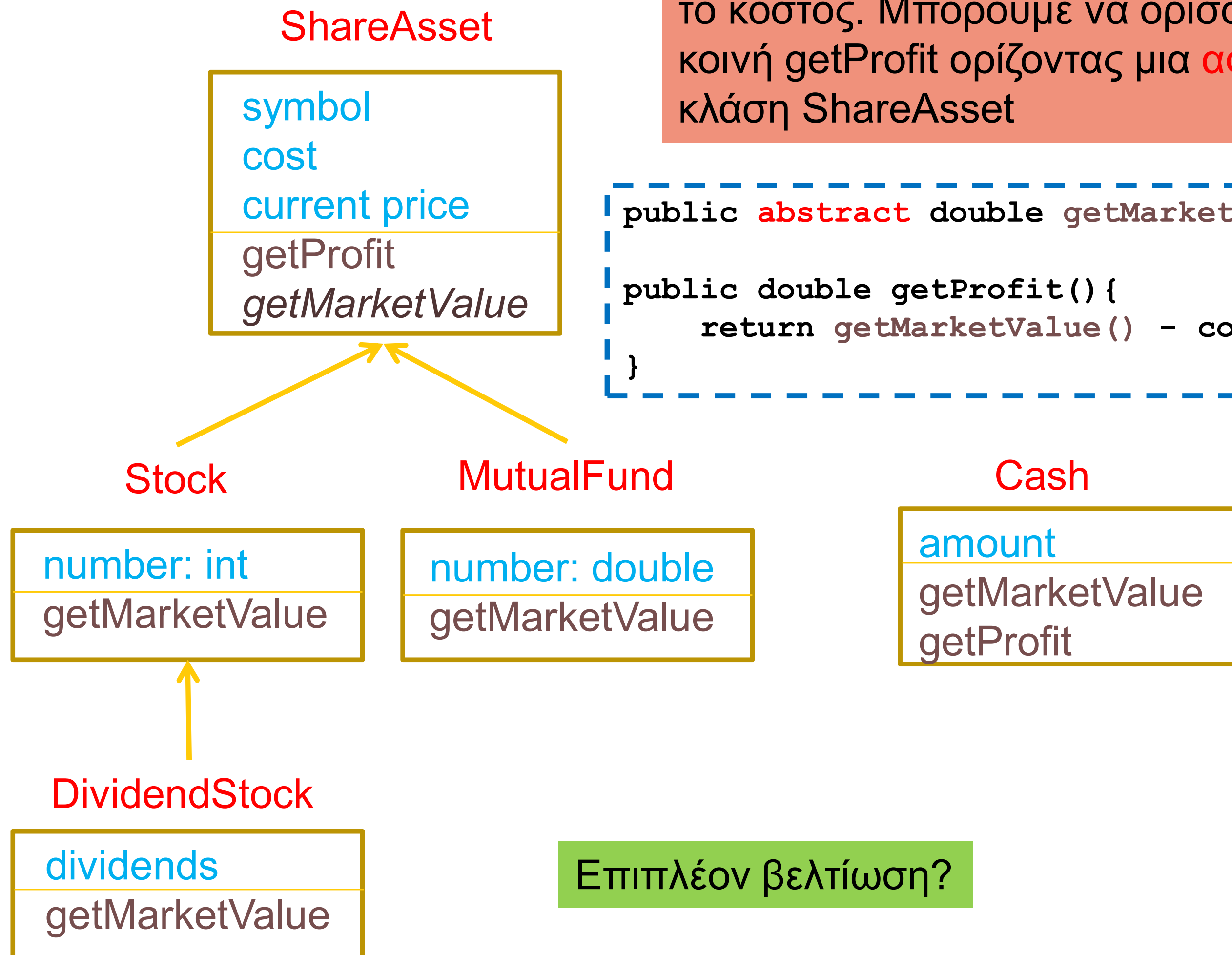
- Βλέπουμε ότι υπάρχουν διάφορα κοινά στοιχεία μεταξύ των διαφόρων οντοτήτων που μας ενδιαφέρουν
  - Χρειαζόμαστε για κάθε asset μια συνάρτηση που να μας δίνει το market value και μία που να υπολογίζει το profit
  - Για τα share assets (stocks, dividend stocks, mutual funds) το κέρδος είναι η διαφορά της τωρινής τιμής με το κόστος
  - Η τιμή των dividend stocks υπολογίζεται όπως αυτή την απλών stocks απλά προσθέτουμε και το μέρισμα

Η DividentStock έχει τα ίδια χαρακτηριστικά με την Stock και απλά αλλάζει ο τρόπος που υπολογίζεται η αποτίμηση ώστε να προσθέτει τα dividents



Πως αλλιώς μπορούμε να βελτιώσουμε το σχεδιασμό?

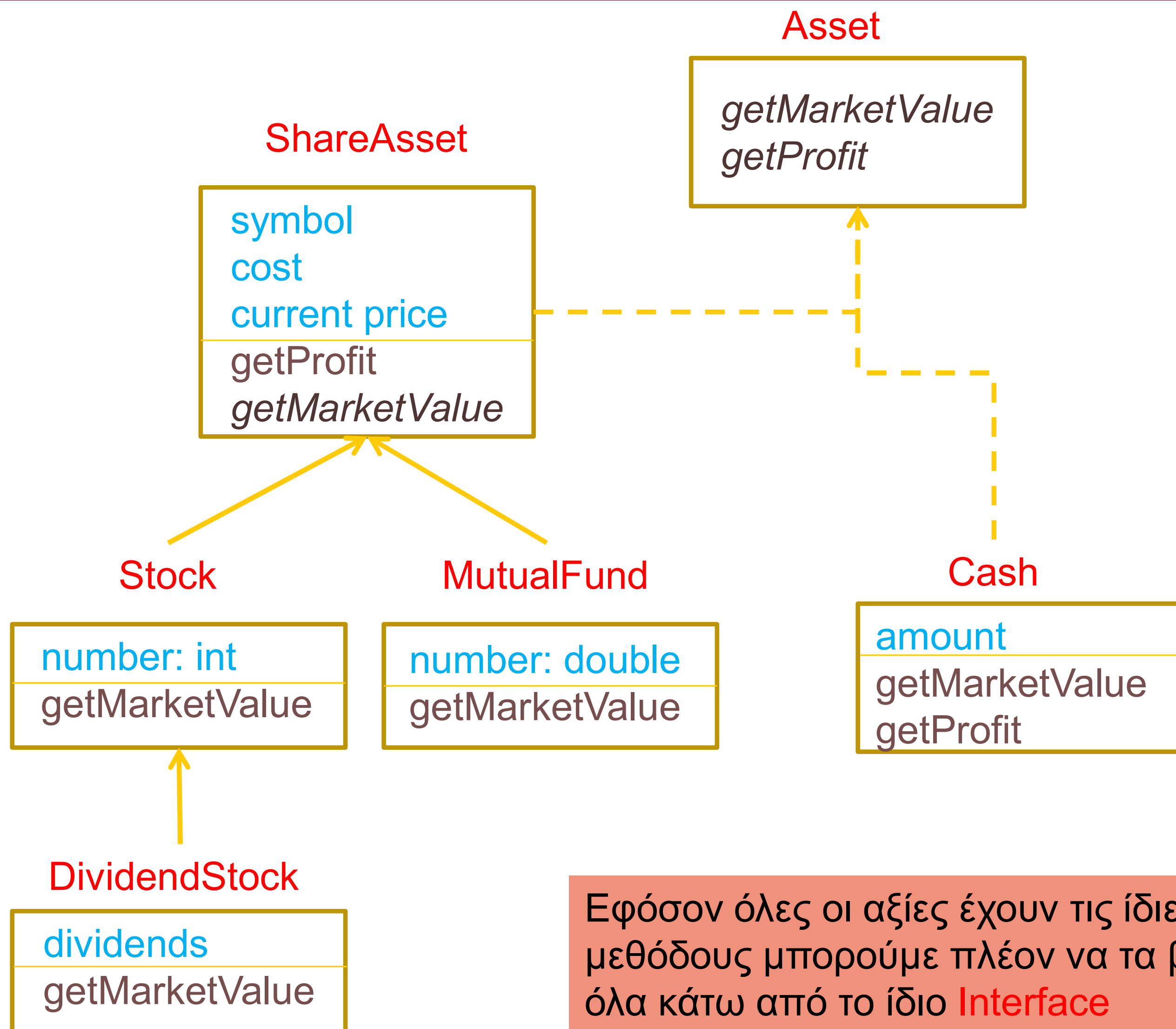
Η getProfit είναι ουσιαστικά η ίδια για όλα τα shares: τωρινή αποτίμηση μείον το κόστος. Μπορούμε να ορίσουμε μια κοινή getProfit ορίζοντας μια αφηρημένη κλάση ShareAsset




```
public abstract double getMarketValue();

public double getProfit(){
    return getMarketValue() - cost;
}
```

Επιπλέον βελτίωση?



Εφόσον όλες οι αξίες έχουν τις ίδιες μεθόδους μπορούμε πλέον να τα βάλουμε όλα κάτω από το ίδιο Interface

```java
public interface Asset
{
        public double getMarketValue();

        public double getProfit();
}
```

```
public class DividendStock extends Stock
{
        private double dividends = 0;

        public DividendStock(String symbol, int number, double costPrice){
                super(symbol,number, costPrice);
        }

        public DividendStock(String symbol, double costPrice){
                super(symbol,costPrice);
        }

        public void payDividends(double amountPerShare){
                dividends = amountPerShare*getNumber();
        }

        public double getMarketValue(){
                return super.getMarketValue() + dividends;
        }

        public String toString(){
                return super.toString() +"\nDividends: " + dividends;
        }
}
```

```java
public abstract class ShareAsset implements Asset
{
            private String symbol;
            private double cost = 0;
            private double currentPrice;

            public ShareAsset(String symbol, double price){
                        this.symbol = symbol;
                        currentPrice = price;
            }

            public abstract double getMarketValue();

            public double getProfit(){
                        return getMarketValue() - cost;
            }

            public void addCost(double cost){
                        this.cost += cost;
            }

            public void setCurrentPrice(double price){
                        currentPrice = price;
            }

            public double getCost(){
                        return cost;
            }

            public double getCurrentPrice(){
                        return currentPrice;
            }

            public String getSymbol(){
                        return symbol;
            }
}
```

```java
public class Stock extends ShareAsset
{
	private int number = 0;

	public Stock(String symbol, int number, double costPrice){
		super(symbol,costPrice);
		this.number = number;
		addCost(number*costPrice);
	}

	public Stock(String symbol, double costPrice){
		super(symbol,costPrice);
	}

	public void purchase(int number, double price){
		this.number += number;
		addCost(number*price);
	}

	public double getMarketValue(){
		return number*getCurrentPrice();
	}

	public int getNumber(){
		return number;
	}

	public String toString(){
		return getSymbol() +": " + number + " cost:" + getCost()
			+ "\nCurrent price: " + getCurrentPrice()
			+ "\nMarket Value: " + getMarketValue()
			+ "\nProfit: " + getProfit();
	}
}
```

```
public class MutualFund extends ShareAsset
{
        private double number = 0;

        public MutualFund(String symbol, double number, double costPrice){
                super(symbol,costPrice);
                this.number = number;
                addCost(number*costPrice);
        }

        public MutualFund(String symbol, double costPrice){
                super(symbol,costPrice);
        }

        public void purchase(double number, double price){
                this.number += number;
                addCost(number*price);
        }

        public double getMarketValue(){
                return number*getCurrentPrice();
        }

        public double getNumber(){
                return number;
        }

        public String toString(){
                return getSymbol() +": " + number + " cost:" + getCost()
                        + "\nCurrent price: " + getCurrentPrice()
                        + "\nMarket Value: " + getMarketValue()
                        + "\nProfit: " + getProfit();
        }
}
```

```
public class DividendStock extends Stock
{
	private double dividends = 0;

	public DividendStock(String symbol, int number, double costPrice){
		super(symbol,number, costPrice);
	}

	public DividendStock(String symbol, double costPrice){
		super(symbol,costPrice);
	}

	public void payDividends(double amountPerShare){
		dividends = amountPerShare*getNumber();
	}

	public double getMarketValue(){
		return super.getMarketValue() + dividends;
	}

	public String toString(){
		return super.toString() +"\nDividends: " + dividends;
	}
}
```

```java
public class Cash implements Asset
{
        private double amount = 0;

        public Cash(double amount)
        {
                this.amount = amount;
        }

        public double getMarketValue(){
                return amount;
        }

        public double getProfit(){
                return 0;
        }

        public String toString(){
                return "Cash: " + amount;
        }
}
```

```java
import java.util.*;

public class Portofolio
{
    public static void main(String[] args){
        ArrayList<Asset> myPortofolio = new ArrayList<Asset>();
        myPortofolio.add(new Cash(1000));
        Stock msft = new DividendStock("STOCK", 100, 39.5);
        myPortofolio.add(msft);
        MutualFund fund = new MutualFund("FUND", 10.5, 30);
        myPortofolio.add(fund);
        fund.setCurrentPrice(40);
        fund.purchase(3.5, 40);
        msft.setCurrentPrice(40);
        Stock appl = new Stock("APPL", 10, 100);
        myPortofolio.add(appl);
        appl.setCurrentPrice(97);

        double totalValue = 0;
        double totalProfit = 0;
        for (Asset a:myPortofolio){
                System.out.println(a+"\n");
                totalValue += a.getMarketValue();
                totalProfit += a.getProfit();
        }
        System.out.println("\nTotal value = "+ totalValue);
        System.out.println("Total profit = "+ totalProfit);
    }
}
```

Χρήση του Interface Asset

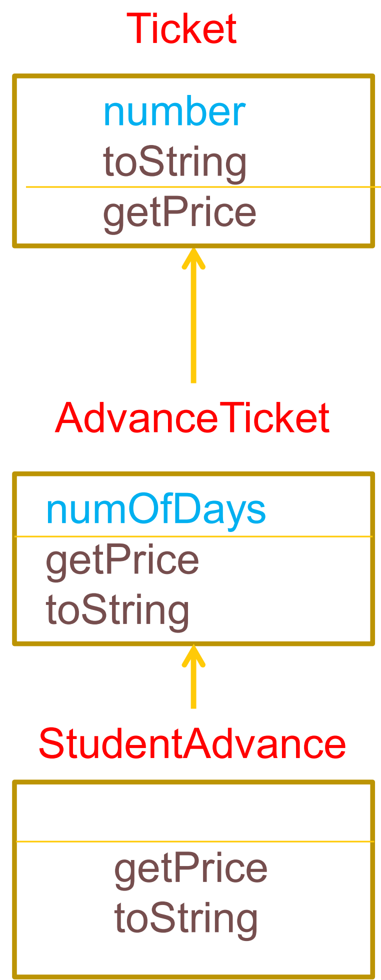
Χρήση των μεθόδων του Interface

Παράδειγμα κληρονομικότητας

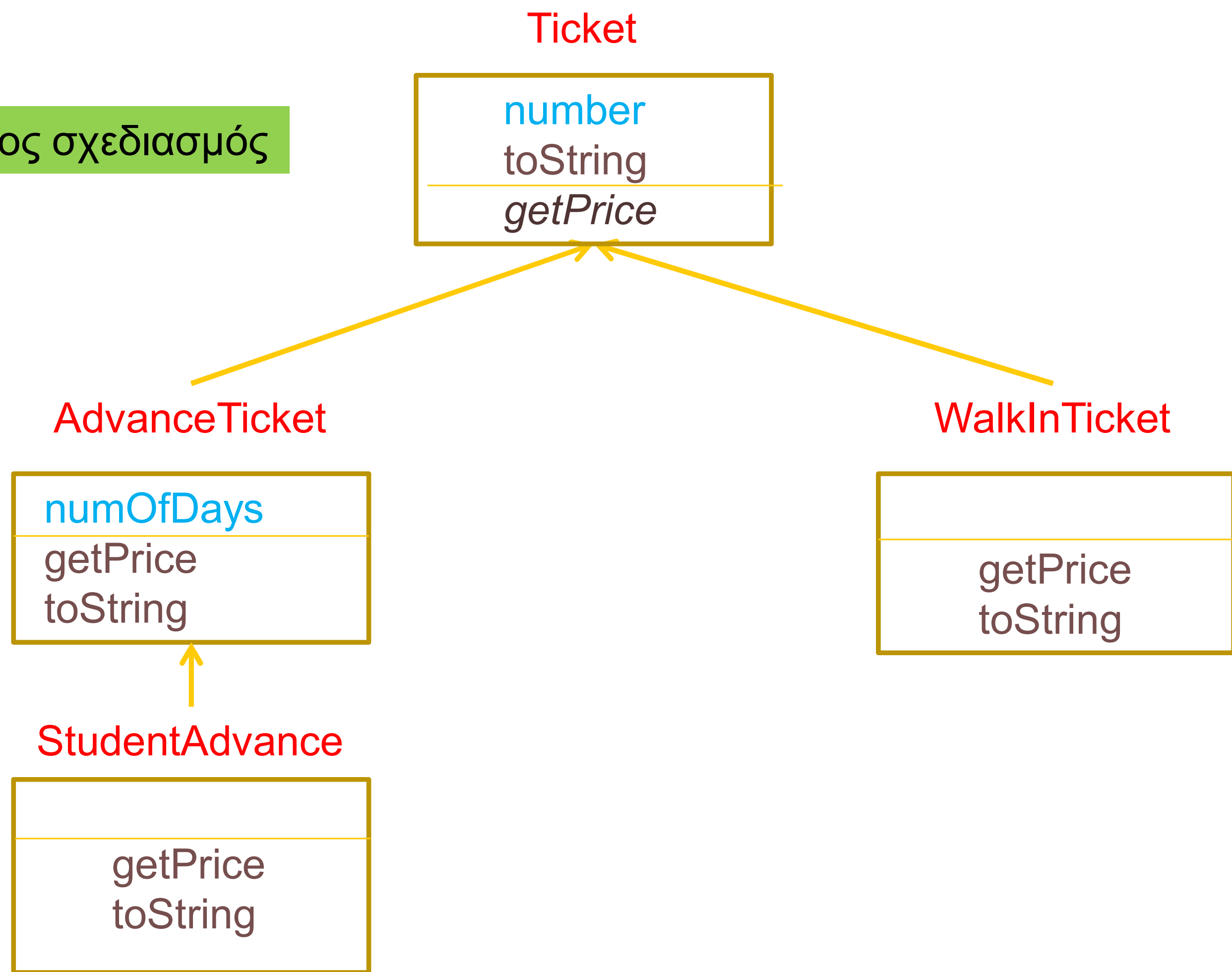
- Έχουμε ένα σύστημα διαχείρισης εισιτηρίων μιας συναυλίας. Το κάθε εισιτήριο έχει ένα νούμερο και τιμή. Η τιμή του εισιτηρίου εξαρτάται αν θα αγοραστεί στην είσοδο (50 ευρώ), ή θα αγοραστεί μέχρι και 10 μέρες πριν την συναυλία (40 ευρώ), ή πάνω από 10 μέρες πριν την συναυλία (30 ευρώ). Τα εισιτήρια εκ των προτέρων έχουν φοιτητική έκπτωση 50%.

- Θέλουμε να τυπώσουμε τα εισιτήρια και να υπολογίσουμε τα συνολικά έσοδα της συναυλίας.

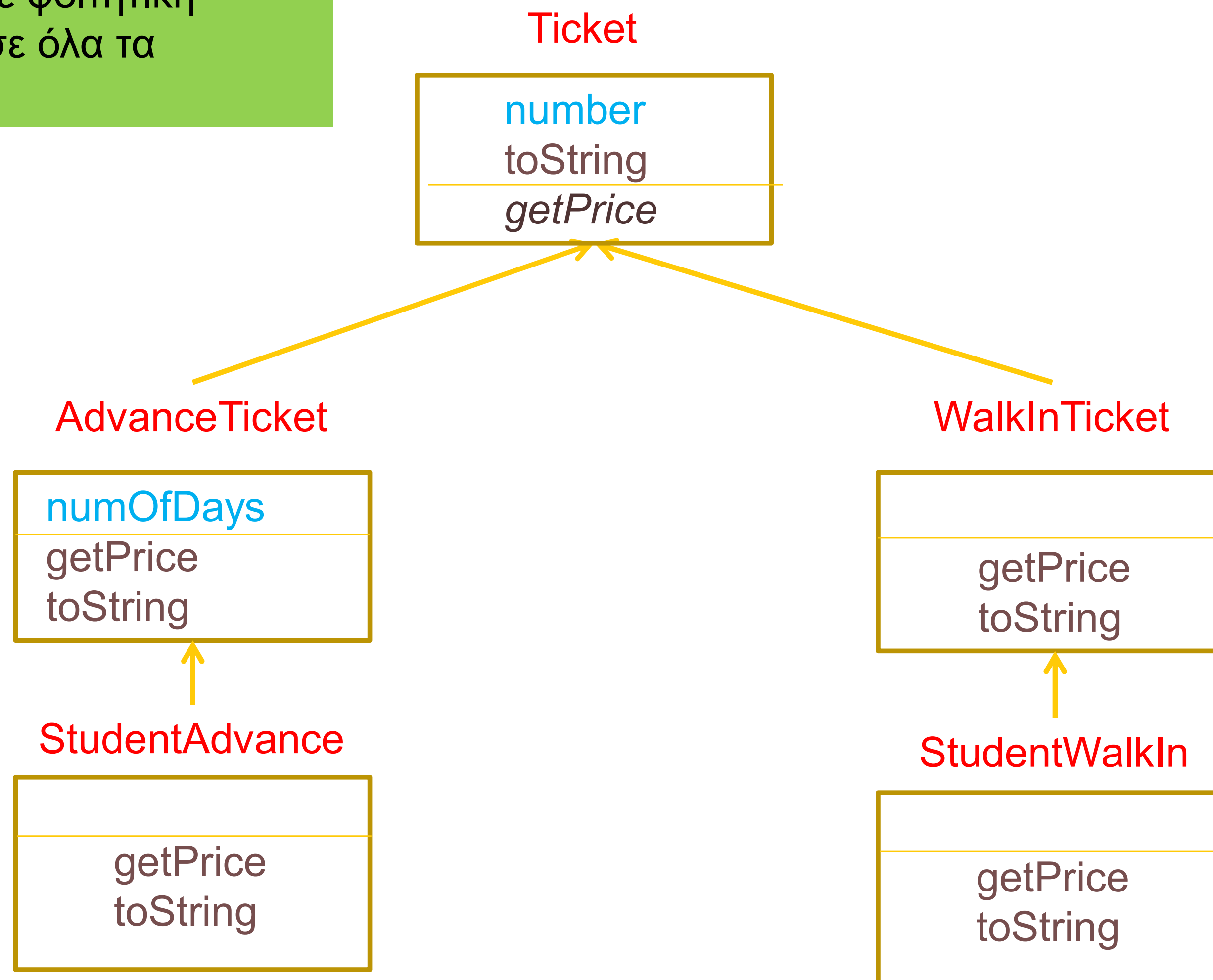
Ένας σχεδιασμός



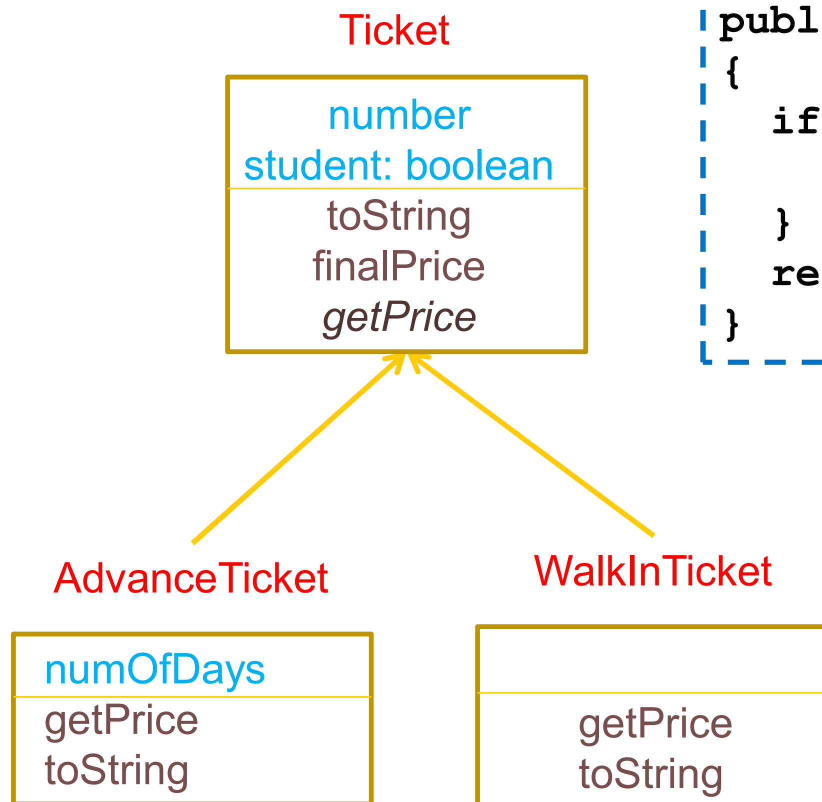
Ένας άλλος σχεδιασμός



Αν θέλουμε φοιτητική έκπτωση σε όλα τα εισιτήρια?



Αν θέλουμε φοιτητική έκπτωση σε όλα τα εισιτήρια?




```
public abstract double getPrice();

public double finalPrice()
{
   if (student){
         return getPrice()*0.5;
   }
   return getPrice();
}
```