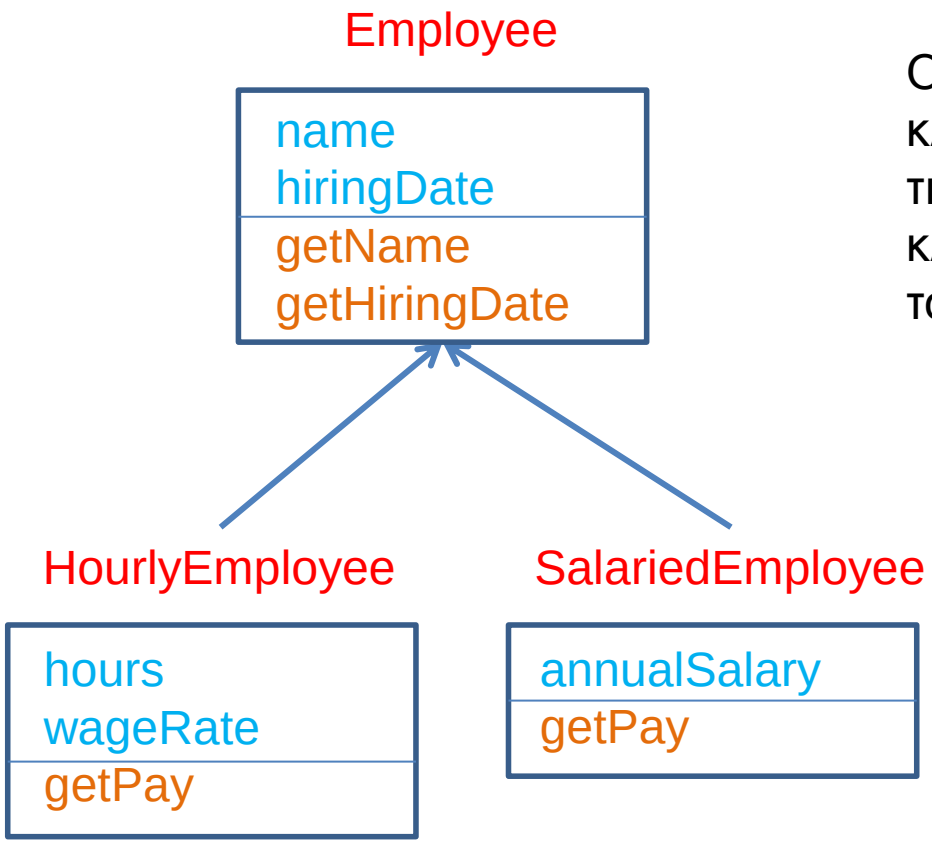
# ΤΕΧΝΙΚΕΣ ΑΝΤΙΚΕΙΜΕΝΟΣΤΡΑΦΟΥΣ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΥ

Πολυμορφισμός – Αφηρημένες κλάσεις

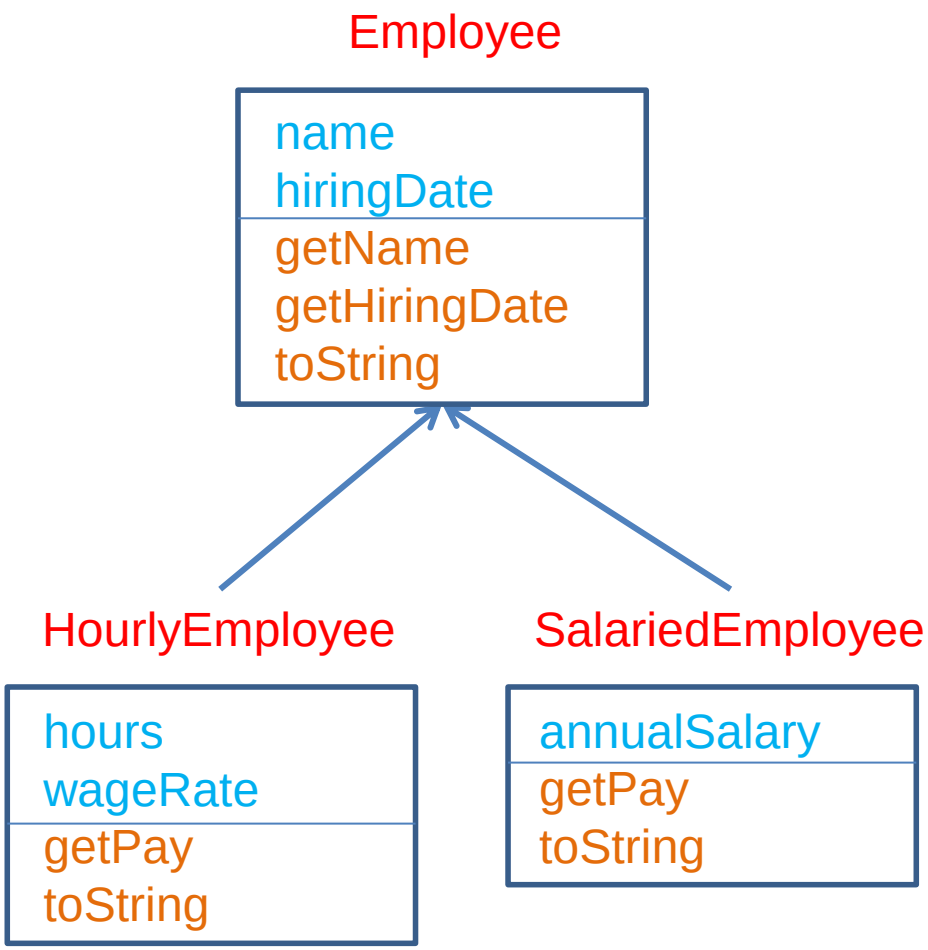
Interfaces (διεπαφές)

# Κληρονομικότητα




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

# Late Binding

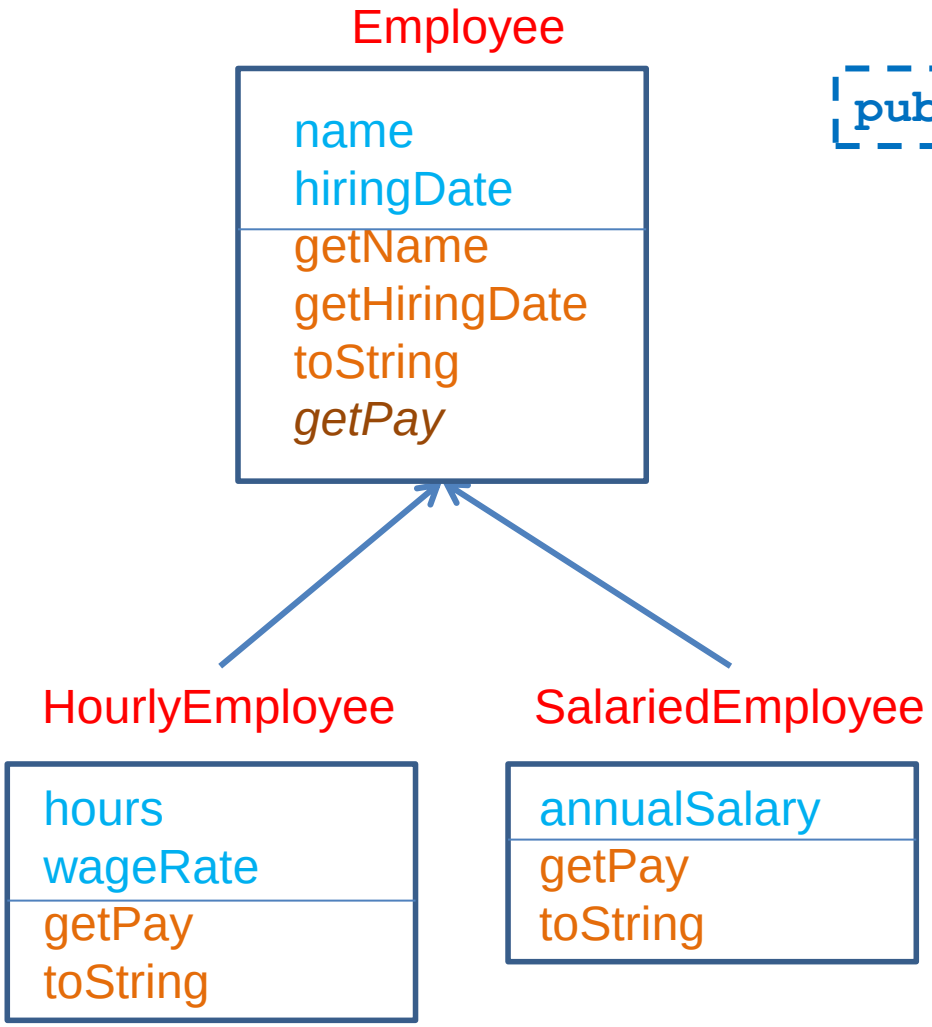



```
Employee e;
e = new HourlyEmployee();
System.out.println(e);
e = new SalariedEmployee();
System.out.println(e);
```

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();
```

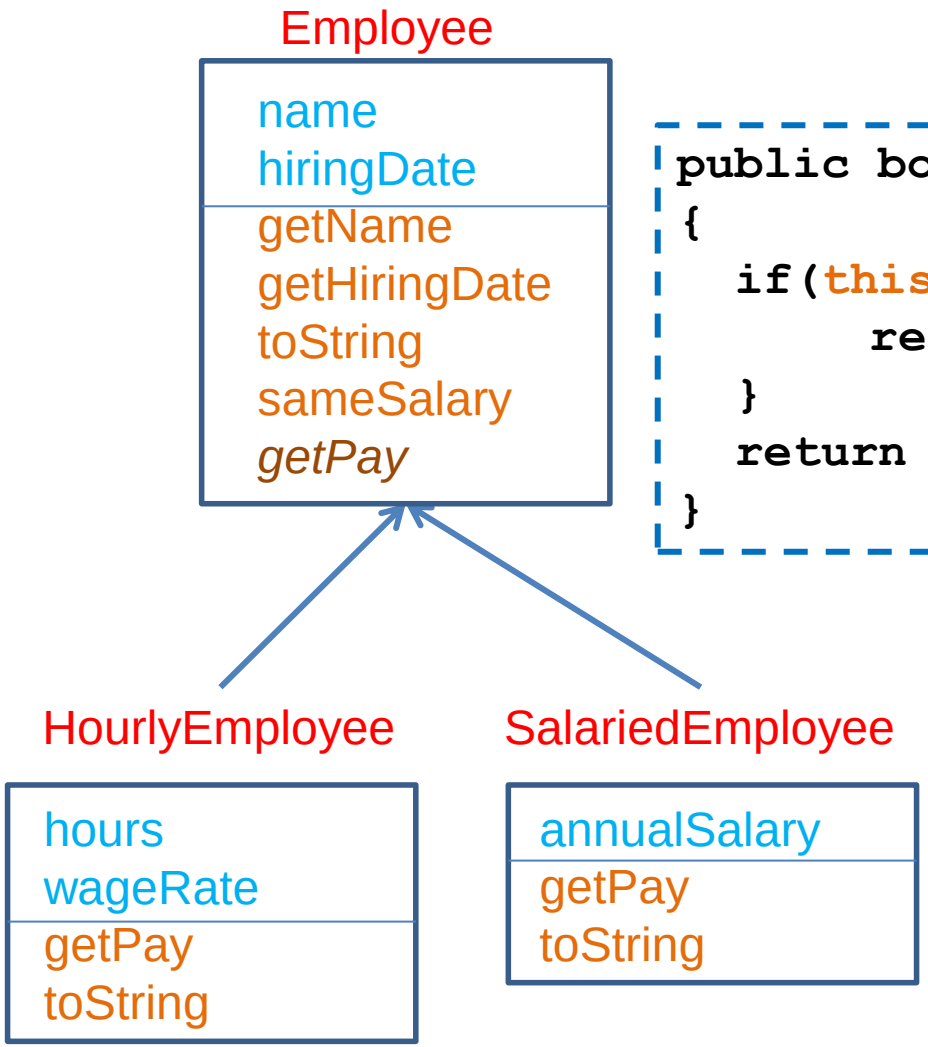
Μια αφηρημένη μέθοδος δηλώνεται σε μια γενική κλάση και ορίζεται σε μια πιο εξειδικευμένη κλάση

Οι κλάσεις με αφηρημένες μεθόδους είναι αφηρημένες κλάσεις.

Δεν μπορούμε να ορίσουμε αντικείμενα αφηρημένων κλάσεων.

Οι παράγωγες ενυπόστατες κλάσεις πρέπει να υλοποιούν τις αφηρημένες μεθόδους.

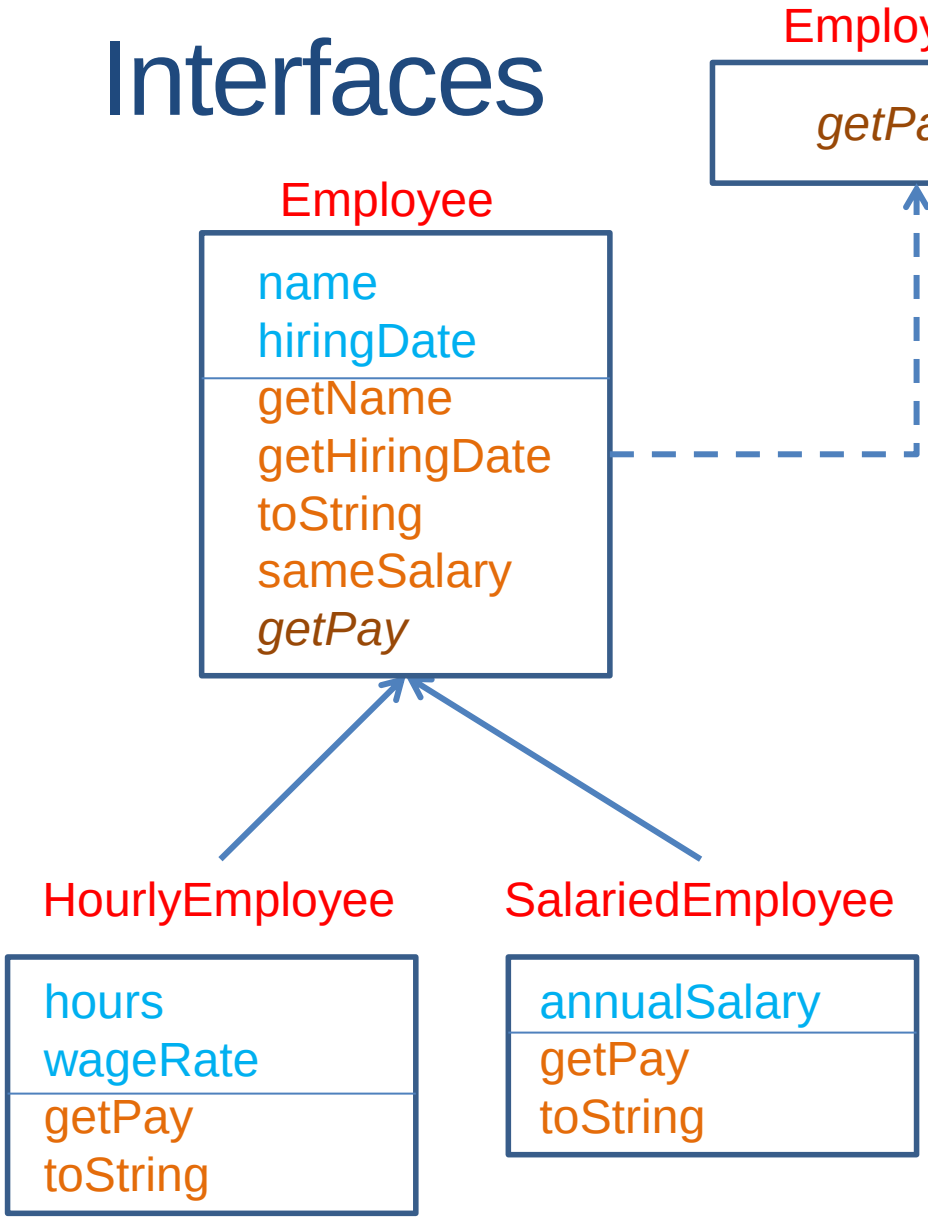
# Αφηρημένες κλάσεις




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

Μια αφηρημένη μέθοδος μπορεί να χρησιμοποιηθεί μέσα σε άλλες μεθόδους της αφηρημένης κλάσης

# Interfaces




```
public interface EmployeePay
{
   public double getPay();
}

public abstract Employee
        implements EmployeePay
```

Ένα interface ορίζει μια βασική λειτουργικότητα (μεθόδους).

Μία κλάση υλοποιεί το interface, δηλ. υλοποιεί τις μεθόδους του interface.

Μια κλάση μπορεί να υλοποιεί παραπάνω από ένα 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 Example
{
  public static void main(String[] args){
    Vehicle[] V = new Vehicle[3];
    V[0] = new Car(0,100);
    V[1] = new Bike();
    V[0].drive(); V[0].print();
    V[1].move(); V[1].print();
    V[2] = new Vehicle(0);
  }
}
```

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

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

```
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);
   }
}
```

```
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[0].drive(); V[0].print();
     V[1].move(); V[1].print();
     V[2] = new Vehicle(0);
   }
}
```

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

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

```
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.

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

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

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

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

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

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

Η Car πρέπει να υλοποίει την μέθοδο move

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

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

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

```
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[0].move(); V[0].print();
    V[1].move(); V[1].print();
    // V[2] = new Vehicle(0);
  }
}
```

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

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;
      }
}
```

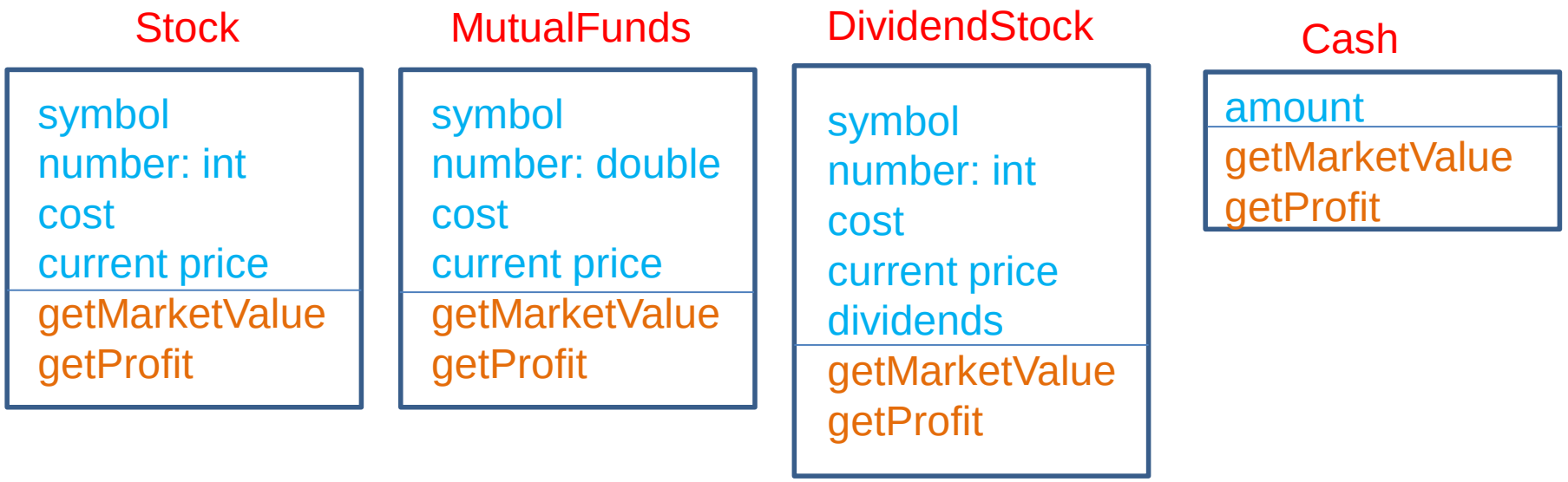
Όχι. Εφόσον η EngineVehicle είναι αφηρημένη δεν χρειάζεται να ορίσουμε την αφηρημένη μέθοδο move

# 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

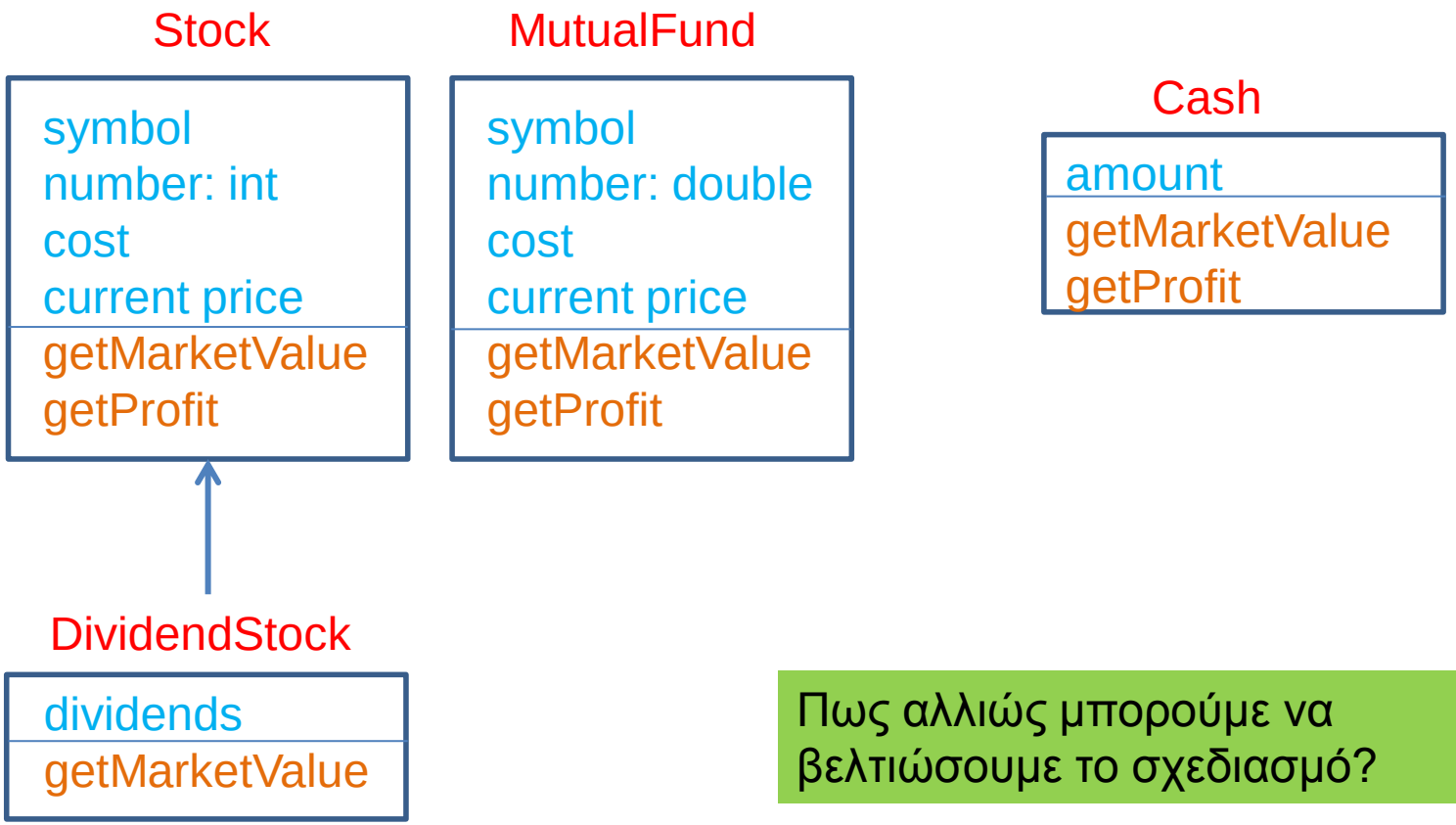



Πως μπορούμε να βελτιώσουμε το σχεδιασμό των κλάσεων?

# Σχεδιασμός

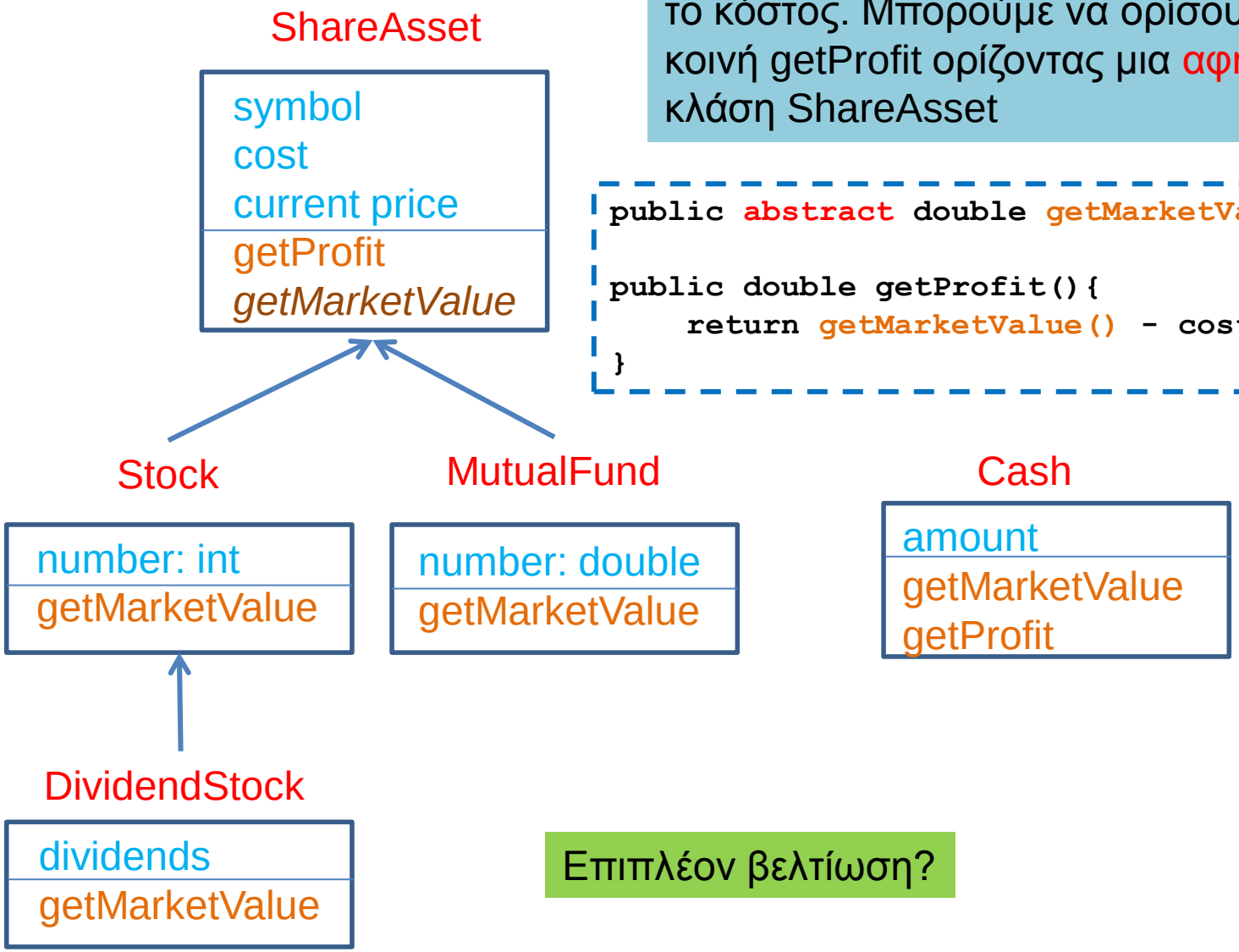
- Βλέπουμε ότι υπάρχουν διάφορα κοινά στοιχεία μεταξύ των διαφόρων οντοτήτων που μας ενδιαφέρουν
  - Χρειαζόμαστε για κάθε asset μια συνάρτηση που να μας δίνει το market value και μία που να υπολογίζει το profit
  - Για τα share assets (stocks, dividend stocks, mutual funds) το κέρδος είναι η διαφορά 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?

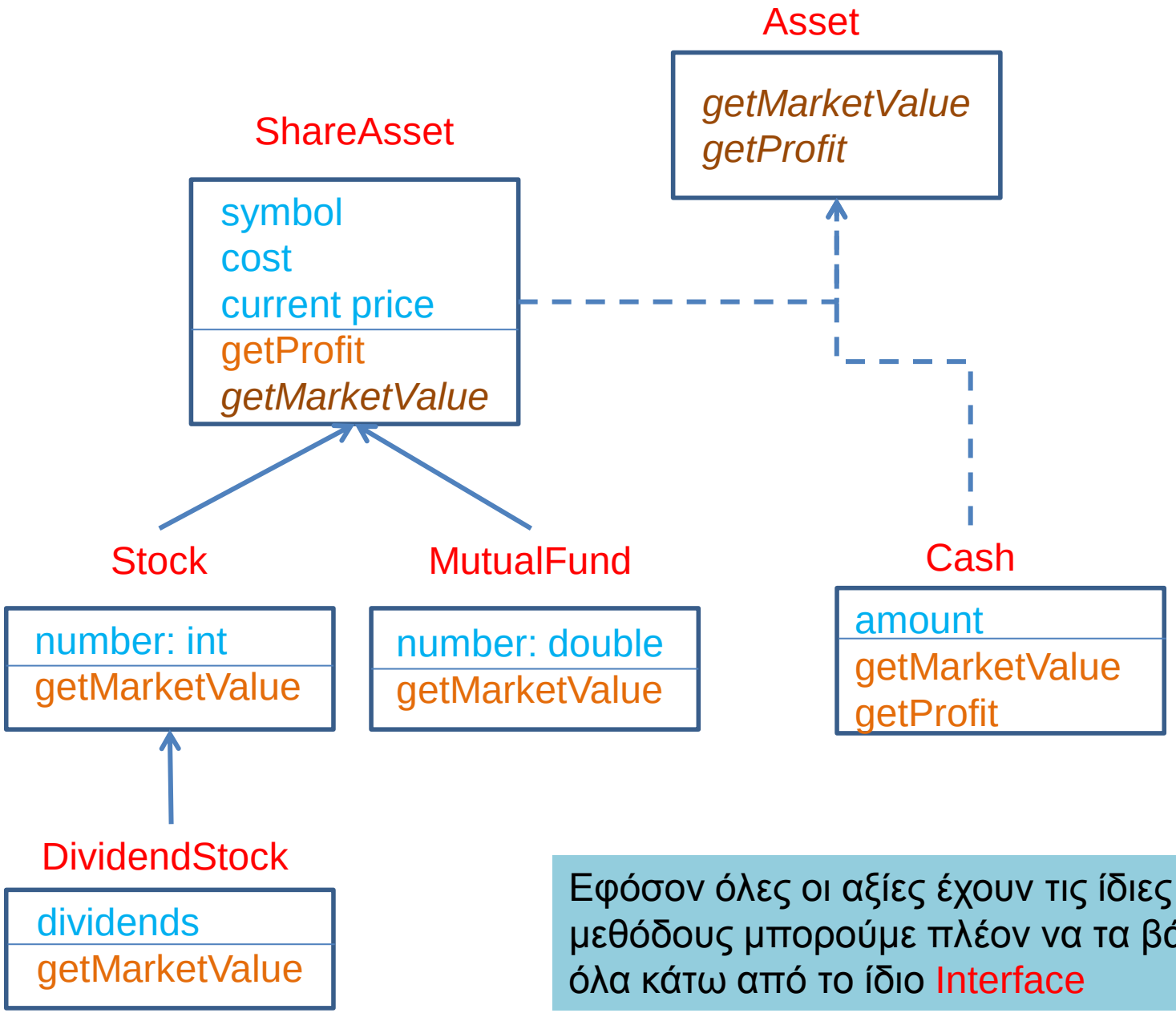
Η getProfit είναι ουσιαστικά η ίδια για όλα τα shares: τωρινή αποτίμηση μείον το κόστος. Μπορούμε να ορίσουμε μια κοινή getProfit ορίζοντας μια αφηρημένη κλάση ShareAsset




```
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

```
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("MSFT", 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