

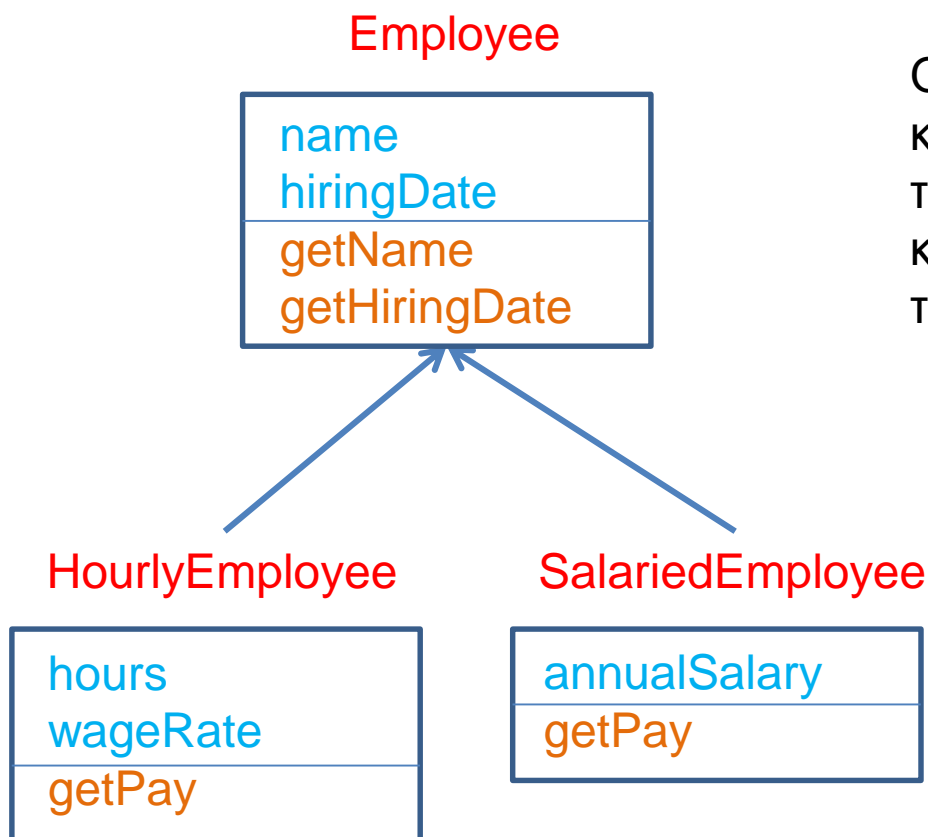
# ΤΕΧΝΙΚΕΣ ΑΝΤΙΚΕΙΜΕΝΟΣΤΡΑΦΟΥΣ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΥ

---

Κληρονομικότητα

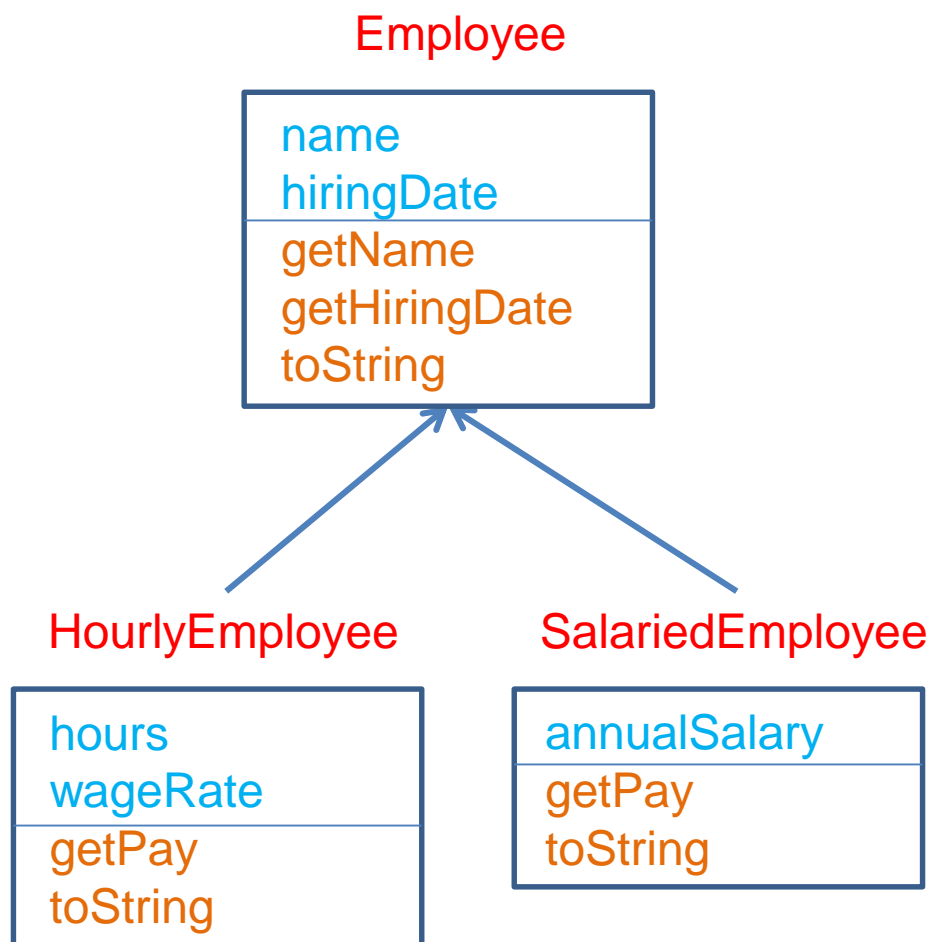
Γενικευμένες κλάσεις

# Κληρονομικότητα



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

# Late Binding

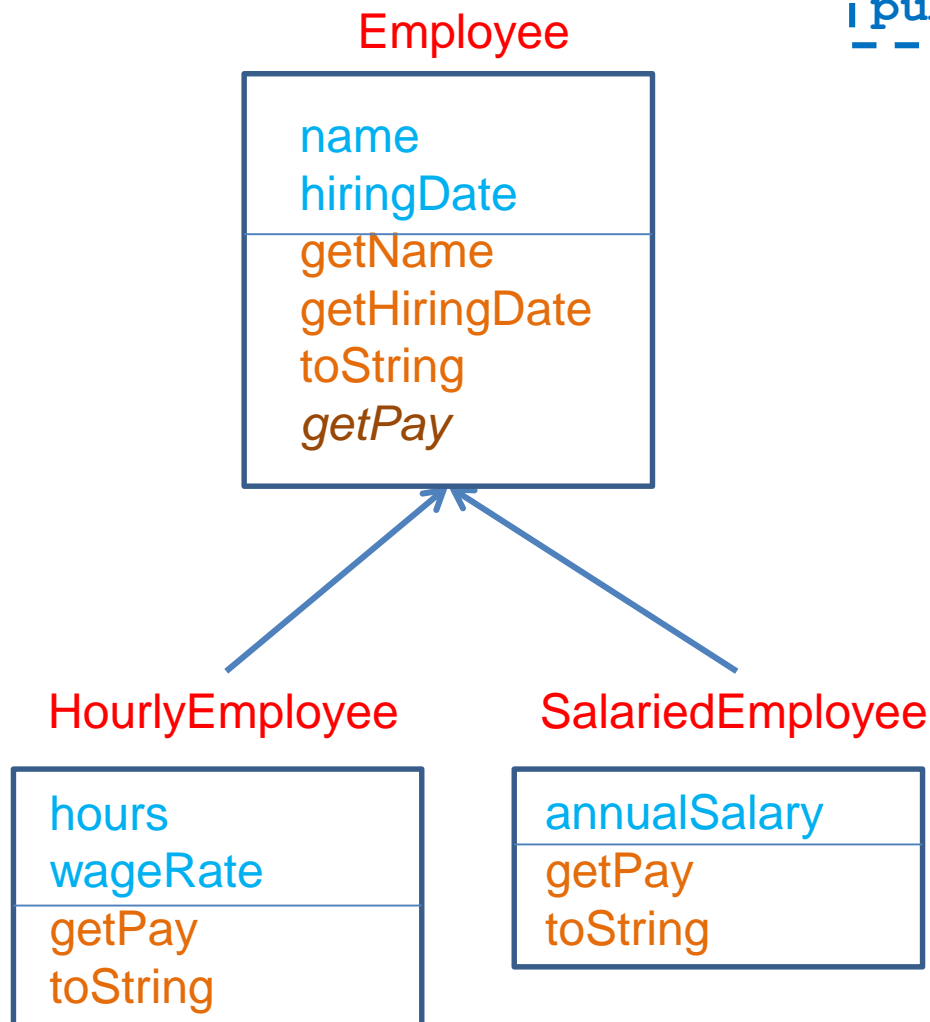


```
Employee e;  
e = new HourlyEmployee();  
System.out.println(e);  
e = new SalariedEmployee();  
System.out.println(e);
```

## 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**.

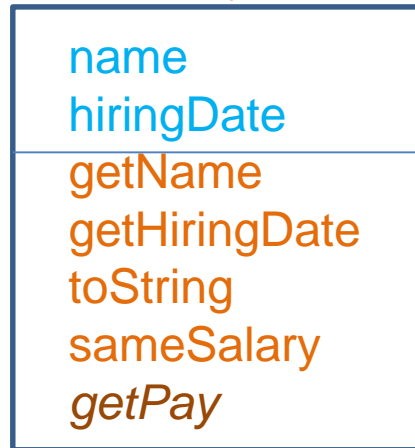
Δεν μπορούμε να **δημιουργήσουμε** αντικείμενα αφηρημένων κλάσεων.

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

Οι παράγωγες **ενυπόστατες** κλάσεις πρέπει να **υλοποιούν** τις αφηρημένες μεθόδους.

# Αφηρημένες κλάσεις

## Employee



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

## HourlyEmployee

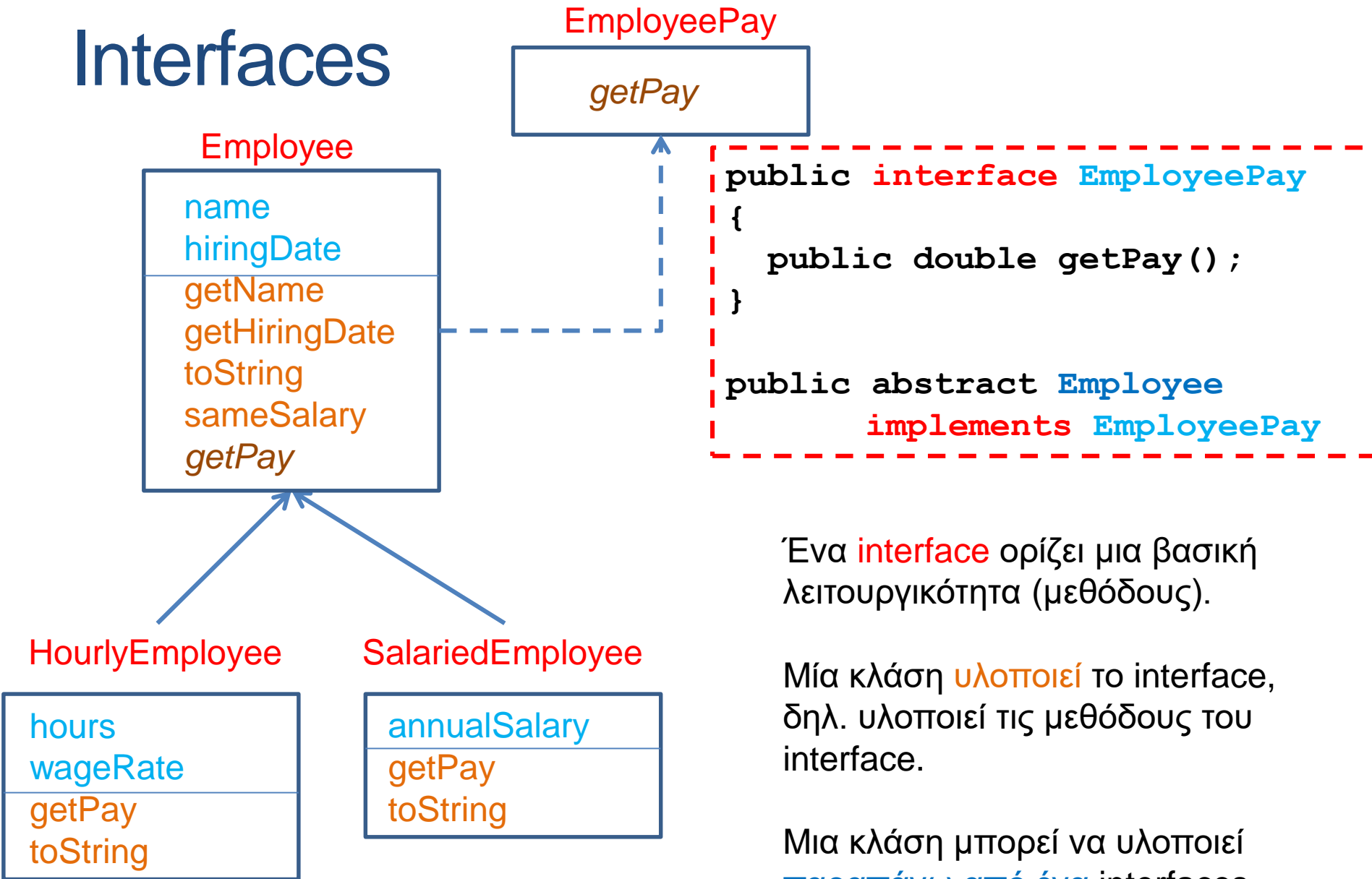
hours wageRate
getPay toString

## SalariedEmployee

annualSalary
getPay toString

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

# Interfaces



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?

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

```
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);
    }
}
```

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

```
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);
    }
}

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 = 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);
    }
}
```

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 υλοποιεί 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[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

# ΠΑΡΑΔΕΙΓΜΑ ΚΛΗΡΟΝΟΜΙΚΟΤΗΤΑΣ

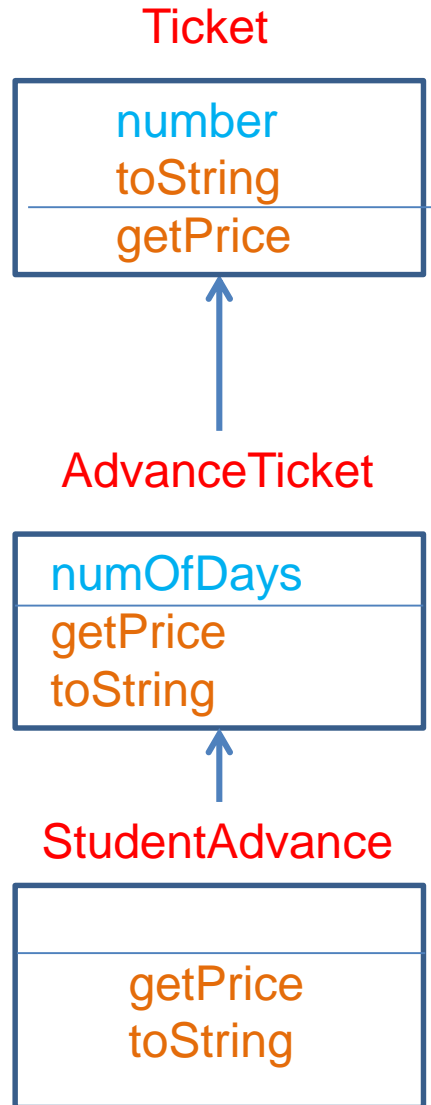
---

# Παράδειγμα κληρονομικότητας

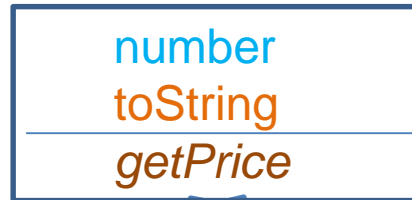
- Έχουμε ένα σύστημα διαχείρισης **εισιτηρίων** μιας συναυλίας. Το κάθε εισιτήριο έχει ένα **νούμερο** και **τιμή**. Η τιμή του εισιτηρίου εξαρτάται αν θα αγοραστεί στην **είσοδο** (50 ευρώ), ή θα αγοραστεί μέχρι και **10 μέρες πριν την συναυλία** (40 ευρώ), ή **πάνω από 10 μέρες πριν την συναυλία** (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



Ticket

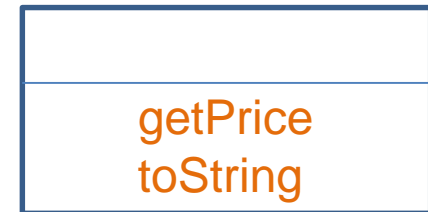


Ένας άλλος σχεδιασμός

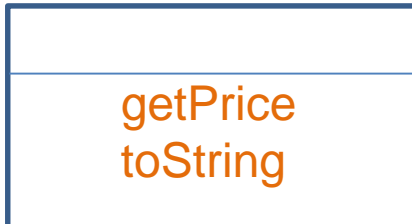
AdvanceTicket



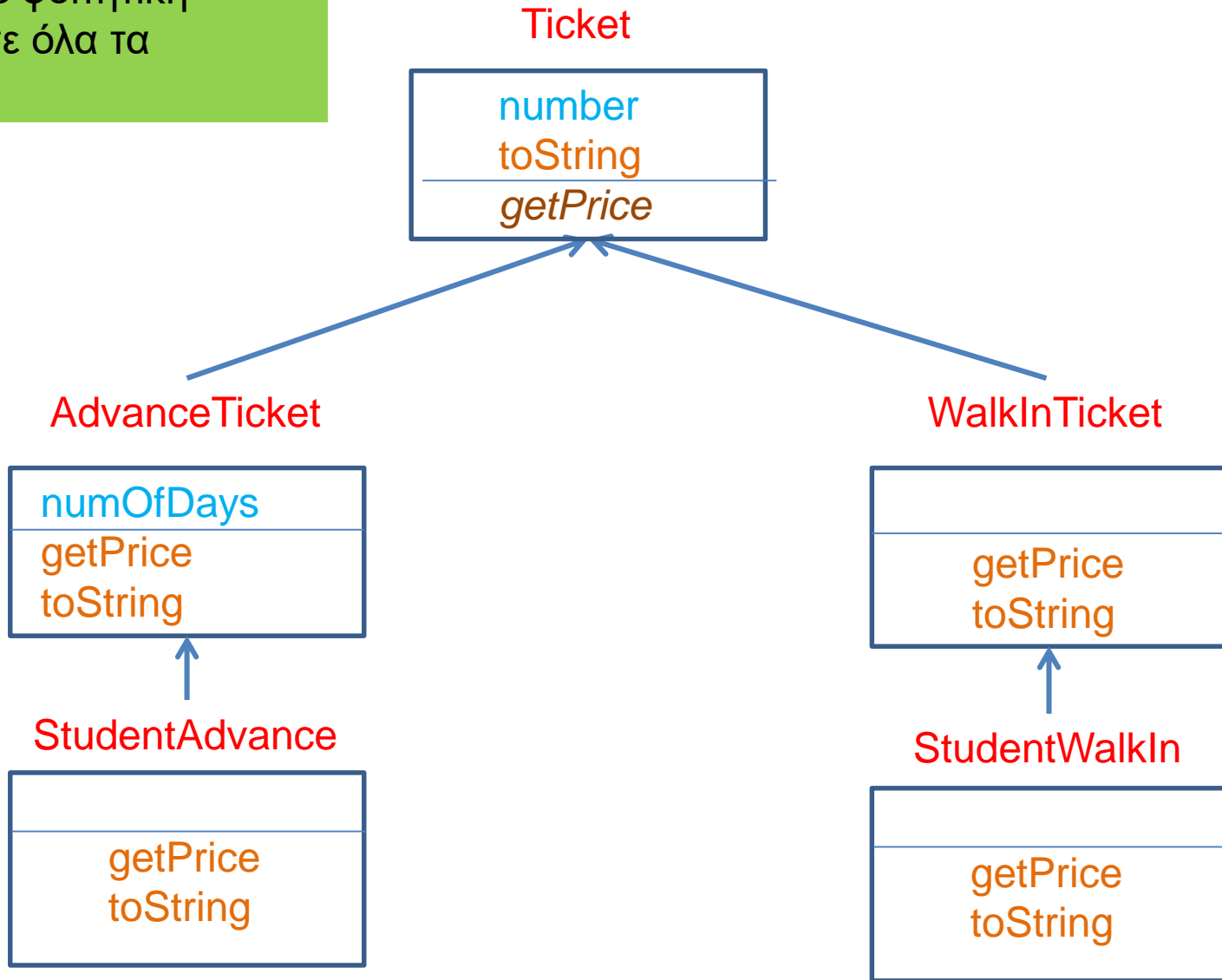
WalkInTicket



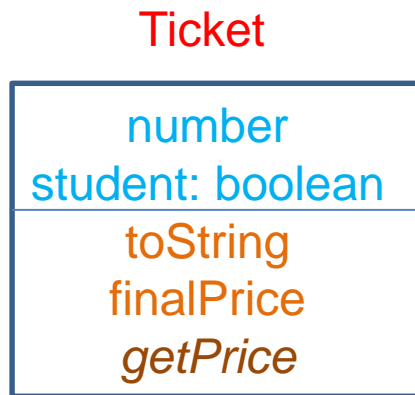
StudentAdvance



Αν θέλουμε φοιτητική έκπτωση σε όλα τα εισιτήρια?



Αν θέλουμε φοιτητική  
έκπτωση σε όλα τα  
εισιτήρια?

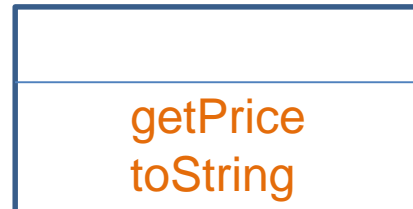


```
public abstract double getPrice();  
  
public double finalPrice()  
{  
    if (student){  
        return getPrice()*0.5;  
    }  
    return getPrice();  
}
```

**AdvanceTicket**



**WalkInTicket**



# ΓΕΝΙΚΕΥΜΕΝΕΣ ΚΛΑΣΕΙΣ

---

# Stack

- Θυμηθείτε πως ορίσαμε μια **στοίβα ακεραίων**

```
public class IntStackElement
{
    private int value;
    private IntStackElement next = null;

    public IntStackElement(int value) {
        this.value = value;
    }

    public int getValue() {
        return value;
    }

    public IntStackElement getNext() {
        return next;
    }

    public void setNext(IntStackElement element) {
        next = element;
    }
}
```

```
public class IntStack
{
    private IntStackElement head;
    private int size = 0;

    public int pop(){
        if (size == 0){ // head == null
            System.out.println("Pop from empty stack");
            System.exit(-1);
        }
        int value = head.getValue();
        head = head.getNext();
        size--;
        return value;
    }

    public void push(int value){
        IntStackElement element = new IntStackElement(value);
        element.setNext(head);
        head = element;
        size++;
    }
}
```



# Stack

- Αν θέλουμε η **στοίβα** μας να αποθηκεύει αντικείμενα της κλάσης **Person** θα πρέπει να ορίσουμε μια διαφορετική **Stack** και διαφορετική **StackElement**.

```
class PersonStackElement
{
    private Person value;
    private PersonStackElement next;

    public PersonStackElement(Person val) {
        value = val;
    }

    public void setNext(PersonStackElement element) {
        next = element;
    }

    public PersonStackElement getNext() {
        return next;
    }

    public Person getValue() {
        return value;
    }
}
```

```
public class PersonStack
{
    private PersonStackElement head;
    private int size = 0;

    public Person pop() {
        if (size == 0) { // head == null
            System.out.println("Pop from empty stack");
            System.exit(-1);
        }
        Person value = head.getValue();
        head = head.getNext();
        size--;
        return value;
    }

    public void push(Person value) {
        PersonStackElement element = new PersonStackElement(value);
        element.setNext(head);
        head = element;
        size++;
    }
}
```

# Stack

- Θα ήταν πιο βολικό αν μπορούσαμε να ορίσουμε **μία μόνο** κλάση **Stack** που να μπορεί να αποθηκεύει αντικείμενα οποιουδήποτε τύπου.
  - **Πώς** μπορούμε να το κάνουμε αυτό?
- Μια λύση: Η **ObjectStack** που κρατάει αντικείμενα **Object**, την πιο γενική κλάση
- Τι πρόβλημα μπορεί να έχει αυτό?

```
class ObjectStackElement
{
    private Object value;
    private ObjectStackElement next;

    public ObjectStackElement(Object val) {
        value = val;
    }

    public void setNext(ObjectStackElement element) {
        next = element;
    }

    public ObjectStackElement getNext() {
        return next;
    }

    public Object getValue() {
        return value;
    }
}
```

```
public class ObjectStack
{
    private ObjectStackElement head;
    private int size = 0;

    public Object pop() {
        if (size == 0) { // head == null
            System.out.println("Pop from empty stack");
            System.exit(-1);
        }
        Object value = head.getValue();
        head = head.getNext();
        size--;
        return value;
    }

    public void push(Object value) {
        ObjectStackElement element = new ObjectStackElement(value);
        element.setNext(head);
        head = element;
        size++;
    }
}
```

```
public class ObjectStackTest
{
    public static void main(String[] args){
        ObjectStack stack = new ObjectStack();

        Person p = new Person("Alice", 1);
        Integer i = new Integer(10);
        String s = "a random string";

        stack.push(p);
        stack.push(i);
        stack.push(s);
    }
}
```

Δεν μπορούμε να **ελέγξουμε** τι αντικείμενα μπαίνουν στην στοίβα. Κατά την εξαγωγή θα πρέπει να γίνει **μετατροπή (downcasting)** και θέλει προσοχή να μετατρέπουμε το σωστό αντικείμενο στον σωστό τύπο.

Θέλουμε να δημιουργούμε στοίβες **συγκεκριμένου τύπου**.

# Γενικευμένες (Generic) κλάσεις

- Οι γενικευμένες κλάσεις περιέχουν ένα τύπο δεδομένων **T** που ορίζεται **παραμετρικά**
- Όταν χρησιμοποιούμε την κλάση αντικαθιστούμε την παράμετρο **T** με τον **τύπο δεδομένων** (την κλάση) που θέλουμε
- ΣΥΝΤΑΚΤΙΚΟ:
  - `public class Example<T> {...}`
- Ορίζει την γενικευμένη κλάση `Example` με παράμετρο τον τύπο **T**
  - Μέσα στην κλάση ο τύπος **T** χρησιμοποιείται σαν **τύπος δεδομένων**
- Όταν χρησιμοποιούμε την κλάση `Example` αντικαθιστούμε το **T** με κάποια συγκεκριμένη **κλάση**
  - `Example<String> ex = new Example<String>();`



# Ένα πολύ απλό παράδειγμα

```
public class Example<T>{
    private T data;

    public Example(T data){
        this.data = data;
    }

    public void setData(T data){
        this.data = data;
    }

    public T getData(){
        return data;
    }

    public static void main(String[] args){
        Example<String> ex = new Example<String>("hello world");
        System.out.println(ex.getData());
    }
}
```

Όταν ορίζουμε το αντικείμενο `ex` η κλάση `String` αντικαθιστά τις εμφανίσεις του `T` στον κώδικα

Ο ορισμός του constructor γίνεται χωρίς το `<T>` παρότι στην δημιουργία του αντικειμένου χρησιμοποιούμε το `<String>`

# Γενικευμένη Στοίβα

- Μπορούμε τώρα να φτιάξουμε μια στοίβα για οποιοδήποτε τύπο δεδομένων

```
class StackElement<T>
{
    private T value;
    private StackElement<T> next;

    public StackElement(T val) {
        value = val;
    }

    public void setNext(StackElement<T> element) {
        next = element;
    }

    public StackElement<T> getNext() {
        return next;
    }

    public T getValue() {
        return value;
    }
}
```

```
public class Stack<T>
{
    private StackElement<T> head;
    private int size = 0;

    public T pop(){
        if (size == 0){ // head == null
            System.out.println("Pop from empty stack");
            System.exit(-1);
        }
        T value = head.getValue();
        head = head.getNext();
        size --;
        return value;
    }

    public void push(T value){
        StackElement<T> element = new StackElement<T>(value);
        element.setNext(head);
        head = element;
        size ++;
    }
}
```

```
public class StackTest
{
    public static void main(String[] args){
        Stack<Person> personStack = new Stack<Person>();

        personStack.push(new Person("Alice", 1));
        personStack.push(new Person("Bob", 2));
        System.out.println(personStack.pop());
        System.out.println(personStack.pop());

        Stack<Integer> intStack = new Stack<Integer>();
        intStack.push(new Integer(10));
        intStack.push(new Integer(20));
        System.out.println(intStack.pop());
        System.out.println(intStack.pop());

        Stack<String> stringStack = new Stack<String>();
        stringStack.push("string 1");
        stringStack.push("string 2");
        System.out.println(stringStack.pop());
        System.out.println(stringStack.pop());
    }
}
```

Δημιουργούμε στοίβες **συγκεκριμένου τύπου**.

# Πολλαπλές παράμετροι

- Μπορούμε να έχουμε πάνω από ένα παραμετρικούς τύπους

```
public class KeyValuePair<K, V>{  
    private K key;  
    private V value;  
    ...  
}
```

# Παγίδες

1. Ο τύπος **T** **δεν** μπορεί να αντικατασταθεί από ένα **πρωταρχικό τύπο δεδομένων** (π.χ. int, double, boolean – πρέπει να χρησιμοποιήσουμε τα **wrapper classes** γι αυτά, Integer, Boolean, Double)
2. **Δεν** μπορούμε να ορίσουμε ένα **πίνακα** από αντικείμενα γενικευμένης κλάσης.

Π.χ., `StackElement<String>[] A =` **Δεν είναι αποδεκτό!**  
`new StackElement<String>[2];`

3. **Δεν** μπορούμε να χρησιμοποιούμε τον τύπο **T** όπως οποιαδήποτε άλλη κλάση.

Π.χ., `T obj = new T();`  
`T[] a = new T[10];`

**Δεν είναι αποδεκτό!**

# Γενικευμένες κλάσεις με περιορισμούς

- Ας υποθέσουμε ότι θέλουμε να ορίσουμε μία γενικευμένη κλάση `Pair` η οποία κρατάει ένα ζεύγος από δυο αντικείμενα οποιουδήποτε τύπου.

```
public class Pair<T>{  
    private T first;  
    private T second;  
    ...  
}
```



# Γενικευμένες κλάσεις με περιορισμούς

- Θέλουμε επίσης να μπορούμε να **διατάσουμε** τα ζεύγη
  - Για να γίνει αυτό θα πρέπει να υπάρχει τρόπος να **συγκρίνουμε** τα στοιχεία first και second.
  - **Περιορίζουμε** την T να **υλοποιεί** το **interface myComparable**

```
public class Pair<T extends myComparable>{  
    private T first;  
    private T second;  
  
    public void order(){  
        if (first.compareTo(second) > 0){  
            T temp = first; first = second; second = temp;  
        }  
    }  
}
```

extends όχι implements

# Γενικευμένες κλάσεις με περιορισμούς

- Θέλουμε επίσης να μπορούμε να **διατάσουμε** τα ζεύγη
  - Για να γίνει αυτό θα πρέπει να υπάρχει τρόπος να **συγκρίνουμε** τα στοιχεία first και second.
  - **Περιορίζουμε** την T να **υλοποιεί** το **interface Comparable**

```
public class Pair<T extends Comparable<T>>>{  
    private T first;  
    private T second;  
  
    public void order(){  
        if (first.compareTo(second) > 0){  
            T temp = first; first = second; second = temp;  
        }  
    }  
}
```

Η **Comparable<T>** της Java  
Το **T** είναι ο τύπος με τον οποίο  
μπορούμε να συγκρίνουμε

# Γενικευμένες κλάσεις με περιορισμούς

- Μπορούμε να περιορίσουμε τον παραμετρικό τύπο να **κληρονομεί** οποιαδήποτε **κλάση**, ή οποιοδήποτε **interface** ή συνδυασμό από τα παραπάνω.

- `public class SomeClass`

- `<T extends Employee> { ... }`

Δέχεται μόνο απογόνους της Employee

- `public class SomeClass`

- `<T extends Employee & Comparable<T>>`

- `{ ... }`

Δέχεται μόνο απογόνους της Employee που υλοποιούν το interface Comparable

Μπορούμε να έχουμε πολλά διαφορετικά interfaces στους περιορισμούς, αλλά μόνο μία κλάση και αυτή θα πρέπει να προηγείται στον ορισμό

# Γενικευμένες κλάσεις και κληρονομικότητα

- Μια γενικευμένη κλάση μπορεί να έχει απογόνους άλλες γενικευμένες κλάσεις.
  - Οι απόγονοι κληρονομούν και τον τύπο T.
  - `public class OrderedPair<T> extends Pair<T> { ... }`
- Δεν ορίζεται κληρονομικότητα ως προς τον παραμετρικό τύπο T
  - Δεν υπάρχει καμία σχέση μεταξύ των κλάσεων `Pair<Employee>` και `Pair<HourlyEmployee>`

# Wildcard

- Αν θέλουμε να ορίσουμε ένα γενικό παραμετρικό τύπο χρησιμοποιούμε την παράμετρο μπαλαντέρ ?, η οποία αναπαριστά ένα οποιοδήποτε τύπο T.
  - Προσέξτε ότι αυτό είναι κατά τη χρήση της γενικευμένης κλάσης
- `public void someMethod(Pair<?>) { ... }`
  - Με αυτή τη δήλωση ορίζουμε μία μέθοδο που παίρνει σαν όρισμα ένα αντικείμενο Pair με τύπο T οτιδήποτε.
- Μπορούμε να περιοριστούμε σε ένα τύπο που είναι απόγονος της Employee.
- `public void someMethod(Pair<? extends Employee>) { ... }`