Κληρονομικότητα

Inheritance

- Κληρονομικότητα: Μηχανισμός στις αντικειμενοστραφείς γλώσσες για τον εμπλουτισμό μιας υπάρχουσας κλάσης
- Η κληρονομικότητα επεκτείνει μία κλάση προσθέτοντας ή επαναορίζοντας μεθόδους και προσθέτοντας ιδιότητες
- Παράδειγμα: Savings account = bank account με επιτόκιο

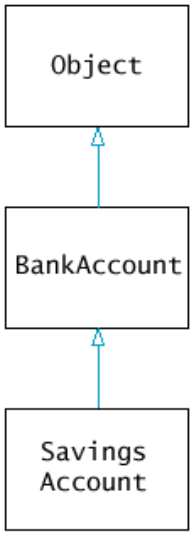
```
class BankAccount {
    public BankAccount()
    {
       balance = 0;
    }
    public void deposit(double amount)
    {
       double newBalance = balance + amount;
       balance = newBalance;
    }
    public double getBalance()
    {
       return balance;
    }
    private double balance;
}
```

```
class SavingsAccount extends BankAccount
{
   new methods
   new instance fields

}
```

- Όλες οι μέθοδοι της `BankAccount` κληρονομούνται αυτόματα
- Μπορούν κάλλιστα να κληθούν οι `deposit`, `getBalance` σε ένα αντικείμενο `SavingsAccount`
- Κλάση που επεκτείνεται = *υπερκλάση*, κλάση που επεκτείνει = *υποκλάση*

Ένα διάγραμμα Κληρονομικότητας



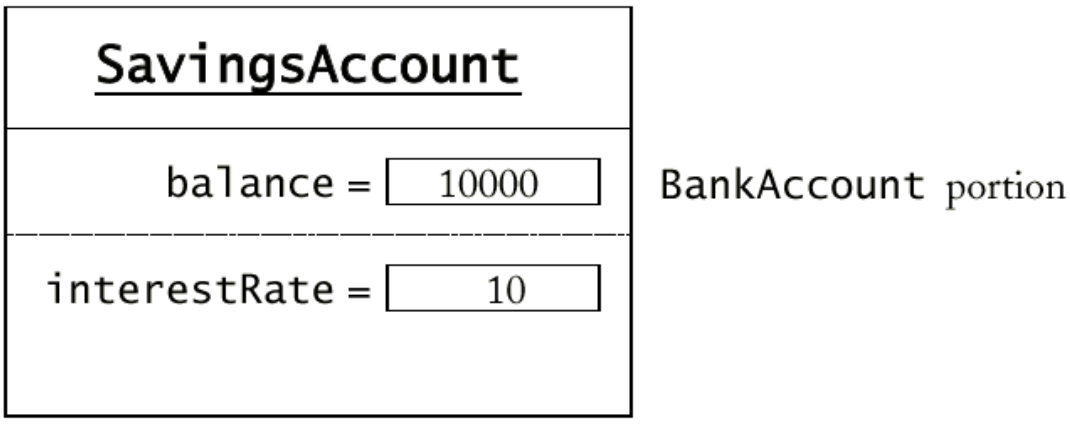
Προσθέτοντας μεθόδους στις υποκλάσεις

```
public class SavingsAccount extends BankAccount
{
   public SavingsAccount(double rate)
   {
      interestRate = rate;
   }

   public void addInterest()
   {
      double interest = getBalance() * interestRate / 100;
      deposit(interest);
   }

   private double interestRate;
}
```

Δομή ενός αντικειμένου της υποκλάσης



Σύνταξη: Κληρονομικότητα

```
class SubclassName extends
SuperclassName
{
   methods
```

```
   instance fields
}
```

Παράδειγμα:

```
public class SavingsAccount
extends BankAccount
{
   public SavingsAccount(double
rate)
   {
      interestRate = rate;
   }

   public void addInterest()
   {
      double interest =
getBalance() * interestRate / 100;

      deposit(interest);
   }

   private double interestRate;
}
```

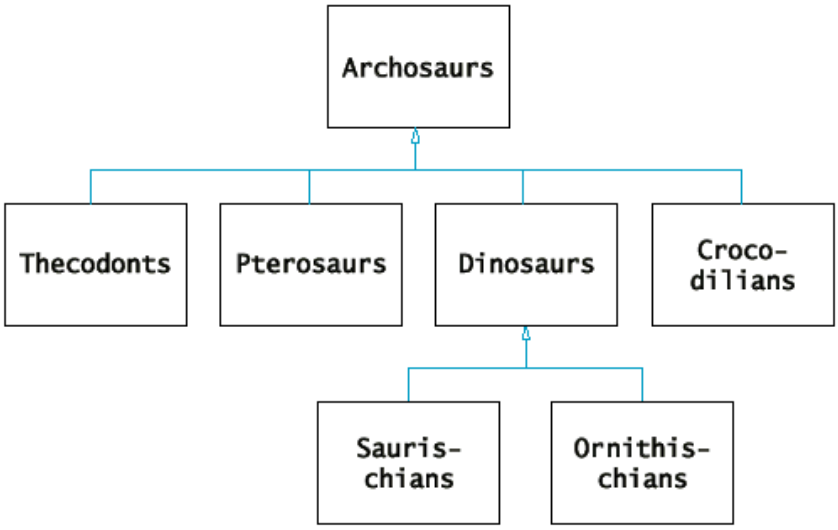
Σκοπός:

Να οριστεί μία νέα κλάση η οποία κληρονομεί από μία υπάρχουσα κλάση και να οριστούν οι μέθοδοι και τα μέλη δεδομένων που προστίθενται στη νέα κλάση.

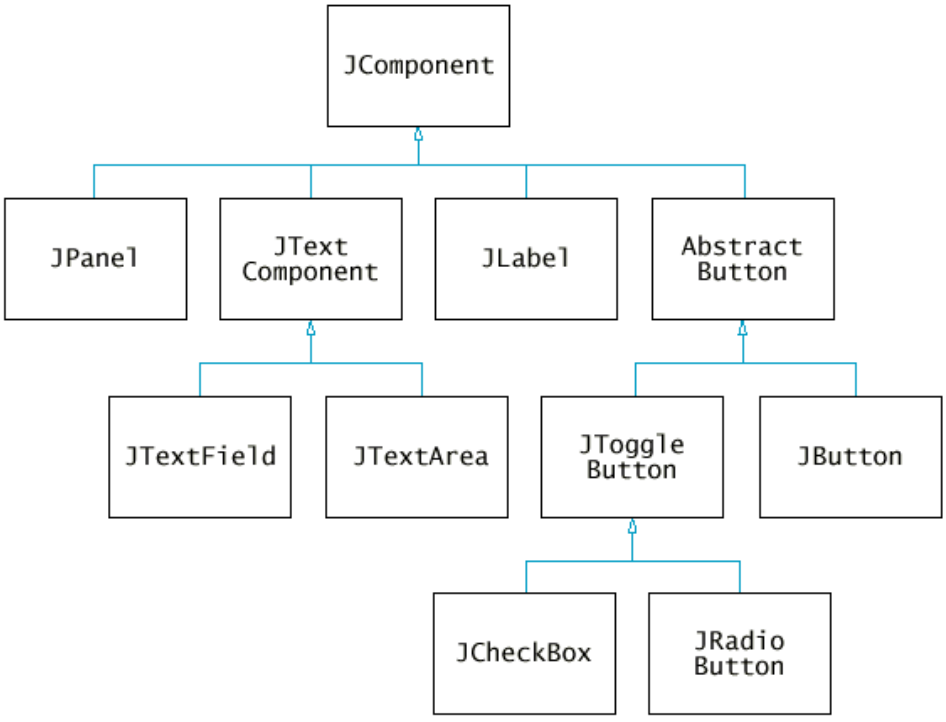
Ιεραρχίες Κληρονομικότητας

- Δενδροειδείς ιεραρχίες κλάσεων, υποκλάσεων και υπερκλάσεων απαντούνται στην πραγματικότητα συχνά
- Παράδειγμα: Ιεραρχία συστατικών Swing
- Η υπερκλάση `JComponent` έχει μεθόδους `getWidth`, `getHeight`
- Η κλάση `AbstractButton` έχει μεθόδους για να θέτει/λαμβάνει το κείμενο του πλήκτρου
- Στη συνέχεια θα μελετηθεί μία απλή ιεραρχία για λογαριασμούς τραπεζών

Ένα τμήμα της ιεραρχίας αρχαίων ερπετών !!!

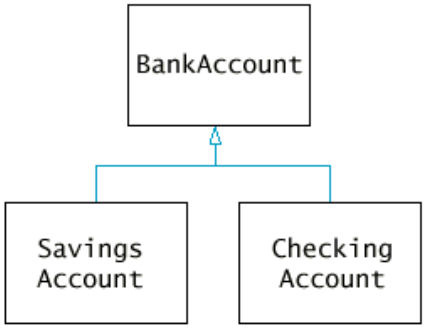


Ένα τμήμα της ιεραρχίας των συστατικών του Swing UI



Ιεραρχία Λογαριασμών Τράπεζας (Bank Account Hierarchy)

Checking Account: Όχι επιτόκιο, ορισμένες δωρεάν συναλλαγές, χρέωση για επιπλέον συναλλαγές
Savings Account: Μηνιαίο επιτόκιο



Κληρονομικότητα και Μέθοδοι

- Επικάλυψη Μεθόδου: Διαφορετική υλοποίηση για μία μέθοδο που υφίσταται στην υπερκλάση
- Κληρονόμηση μεθόδου: Όταν δεν παρέχεται μία νέα υλοποίηση για μία μέθοδο που υφίσταται στην υπερκλάση
- Προσθήκη μεθόδου: Παροχή μιας νέας μεθόδου που δεν υφίσταται στην υπερκλάση

Κληρονομικότητα και Ιδιότητες

- Κληρονόμηση ιδιοτήτων: Όλα τα μέλη δεδομένων της υπερκλάσης κληρονομούνται αυτόματα
- Προσθήκη ιδιότητας: Παροχή ενός νέου μέλους δεδομένων που δεν υφίσταται στην υπερκλάση
- Δεν είναι δυνατή η επικάλυψη ιδιοτήτων (δεν συνίσταται ο ορισμός στην υποκλάση ιδιοτήτων με ίδιο όνομα όπως η υπερκλάση)

CheckingAccount Class

- Τρεις πρώτες συναλλαγές δωρεάν
- Χρέωση 2€ για κάθε επιπλέον συναλλαγή
- Επικάλυψη της `deposit` ώστε να αυξάνει τον αριθμό των συναλλαγών
- Υλοποίηση μεθόδου `deductFees` για την αφαίρεση ποσού χρέωσης και επαναφορά του μετρητή συναλλαγών

Οι ιδιότητες που κληρονομούνται είναι ιδιωτικές (Private)

- Έστω η μέθοδος `deposit` της κλάσης `CheckingAccount`
```
public void deposit(double amount)
{
   transactionCount++;
   // now add amount to balance
   ...
}
```
- Δεν μπορούμε απλά να προσθέσουμε την μεταβλητή `amount` στην `balance`
- Η `balance` είναι ιδιωτικό μέλος δεδομένων (*private*) της υπερκλάσης
- Οι υποκλάσεις πρέπει να χρησιμοποιούν τις δημόσιες μεθόδους της υπερκλάσης όπως οποιαδήποτε άλλη κλάση

Κλήση μιας μεθόδου της υπερκλάσης

- Δεν μπορούμε απλά να καλέσουμε
  `deposit(amount)`
  στην μέθοδο `deposit` της κλάσης `CheckingAccount`

- διότι είναι ισοδύναμο προς
  `this.deposit(amount)`
- Καλεί την ίδια μέθοδο (αέναη αναδρομή)
- Αντίθετα, καλούμε την μέθοδο της υπερκλάσης (*superclass method*)
  `super.``deposit(amount)`
- Τώρα πλέον καλείται η μέθοδος `deposit` της κλάσης `BankAccount`
- Πλήρης μέθοδος:
```
public void deposit(double amount)
{
   transactionCount++;
   super.deposit(amount);
}
```

Σύνταξη: Κλήση μιας Μεθόδου της υπερκλάσης

```
super.methodName(parameters)
```

Παράδειγμα:

```
public void deposit(double amount)
{
   transactionCount++;
   super.deposit(amount);
}
```

Σκοπός:

Κλήση μιας μεθόδου της υπερκλάσης αντί της μεθόδου της τρέχουσας κλάσης

Κλήση του κατασκευαστή της υπερκλάσης

```
public class CheckingAccount extends BankAccount
{
   public CheckingAccount(double initialBalance)
   {
      // construct superclass
      super(initialBalance);

      // initialize transaction count
      transactionCount =0;
   }

   ...
}
```

- Πέρασμα παραμέτρων στον κατασκευαστή της υπερκλάσης
- Πρέπει να είναι η *πρώτη* εντολή στον κατασκευαστή της υποκλάσης

Σύνταξη: Κλήση του κατασκευαστή υπερκλάσης

```
ClassName(parameters)
{
   super(parameters);
   . . .
}
```

Παράδειγμα:

```
public CheckingAccount(double
initialBalance)
{
   super(initialBalance);

   transactionCount =0;
}
```

Σκοπός:

Κλήση του κατασκευαστή της υπερκλάσης. Η εντολή πρέπει να είναι η πρώτη εντολή στον κατασκευαστή της υποκλάσης.

Μετατροπή από υποκλάσεις σε υπερκλάσεις

- Δεν υπάρχει πρόβλημα για τη μετατροπή αναφορών υποκλάσεων σε αναφορές υπερκλάσεων
- ```
  SavingsAccount collegeFund = new SavingsAccount(10);
  BankAccount anAccount = collegeFund;
  Object anObject = collegeFund;
  ```
- Οι αναφορές των υπερκλάσεων δεν γνωρίζουν όλη την πληροφορία:
  ```
  anAccount.addInterest(); // ERROR
   anAccount.deposit(100);  // OK
  ```

- Γιατί κάποιος να θέλει να γνωρίζει *λιγότερα* για ένα αντικείμενο?

Πολυμορφισμός

- Έστω η γενική μέθοδος μεταφοράς ποσών:
  ```
  public void transfer(double amount, BankAccount other)
  {
     withdraw(amount);
     other.deposit(amount);
  }
  ```
- Δουλεύει με οποιοδήποτε είδος λογαριασμού (plain, checking, savings)
  ```
  BankAccount momsAccount = . . .
  BankAccount harrysAccount = . . .
  momsAccount.transfer(1000, harrysAccount);
  ```

- Μπορεί κάλιστα να χρησιμοποιηθεί για να μεταφερθεί ποσό σε ένα CheckingAccount.
  ```
  CheckingAccount babisChecking = . . .
  momsAccount.transfer(1000, babisChecking);
  ```
  Εδώ, μία αναφορά υποκλάσης (babisChecking) μετατρέπεται σε αναφορά υπερκλάσης (other)

- Πολυμορφισμός: Αν καλέσουμε
  `other.deposit(amount)`
  θα κληθεί η `CheckingAccount.deposit`
  διότι οι μέθοδοι που τελικά καλούνται, καθορίζονται από τον τύπο του πραγματικού αντικειμένου, όχι από τον τύπο της αναφοράς
  (late binding)

File AccountTest.java

```
1 /**
2    This program tests the BankAccount class and
```
```

```
   their subclasses.
*/
 public class AccountTest
{
   public static void main(String[] args)
   {
      SavingsAccount momsSavings
         = new SavingsAccount(0.5);

      CheckingAccount harrysChecking
         = new CheckingAccount(100);

      momsSavings.deposit(10000);

      momsSavings.transfer(harrysChecking, 2000);
      harrysChecking.withdraw(1500);
      harrysChecking.withdraw(80);

      momsSavings.transfer(harrysChecking, 1000);
      harrysChecking.withdraw(400);

      // simulate end of month
      momsSavings.addInterest();
      harrysChecking.deductFees();

      System.out.println("Mom's savings balance = $"
         + momsSavings.getBalance());

      System.out.println("Harry's checking balance = $"
         + harrysChecking.getBalance());
   }
}
```

File BankAccount.java

```
/**
   A bank account has a balance that can be changed by
   deposits and withdrawals.
*/
public class BankAccount
{
   /**
      Constructs a bank account with a zero balance
   */
   public BankAccount()
   {
      balance = 0;
   }
```

```
14
15  /**
16     Constructs a bank account with a given balance
17     @param initialBalance the initial balance
18  */
19  public BankAccount(double initialBalance)
20  {
21     balance = initialBalance;
22  }
23
24  /**
25     Deposits money into the bank account.
26     @param amount the amount to deposit
27  */
28  public void deposit(double amount)
29  {
30     balance = balance + amount;
31  }
32
33  /**
34     Withdraws money from the bank account.
35     @param amount the amount to withdraw
36  */
37  public void withdraw(double amount)
38  {
39     balance = balance - amount;
40  }
41
42  /**
43     Gets the current balance of the bank account.
44     @return the current balance
45  */
46  public double getBalance()
47  {
48     return balance;
49  }
50
51  /**
52     Transfers money from the bank account to another account
53     @param other the other account
54     @param amount the amount to transfer
55  */
56  public void transfer(BankAccount other, double amount)
57  {
58     withdraw(amount);
59     other.deposit(amount);
60  }
61
```

```
62    private double balance;
63 }
```

File CheckingAccount.java

```
1  /**
2     A checking account that charges transaction fees.
3  */
4  public class CheckingAccount extends BankAccount
5  {
6     /**
7        Constructs a checking account with a given balance
8        @param initialBalance the initial balance
9     */
10    public CheckingAccount(int initialBalance)
11    {
12       // construct superclass
13       super(initialBalance);
14
15       // initialize transaction count
16       transactionCount = 0;
17    }
18
19    public void deposit(double amount)
20    {
21       transactionCount++;
22       // now add amount to balance
23       super.deposit(amount);
24    }
25
26    public void withdraw(double amount)
27    {
28       transactionCount++;
29       // now subtract amount from balance
30       super.withdraw(amount);
31    }
32
33    /**
34       Deducts the accumulated fees and resets the
35       transaction count.
36    */
37    public void deductFees()
38    {
39       if (transactionCount > FREE_TRANSACTIONS)
40       {
41          double fees = TRANSACTION_FEE *
42             (transactionCount - FREE_TRANSACTIONS);
43          super.withdraw(fees);
```

```
44         }
45         transactionCount = 0;
46      }
47
48      private int transactionCount;
49
50      private static final int FREE_TRANSACTIONS = 3;
51      private static final double TRANSACTION_FEE = 2.0;
52   }
```

File SavingsAccount.java

```
1  /**
2     An account that earns interest at a fixed rate.
3  */
4  public class SavingsAccount extends BankAccount
5  {
6     /**
7        Constructs a bank account with a given interest rate
8        @param rate the interest rate
9     */
10    public SavingsAccount(double rate)
11    {
12       interestRate = rate;
13    }
14
15    /**
16       Adds the earned interest to the account balance.
17    */
18    public void addInterest()
19    {
20       double interest = getBalance() * interestRate / 100;
21       deposit(interest);
22    }
23
24    private double interestRate;
25 }
```

Επίπεδο ελέγχου πρόσβασης

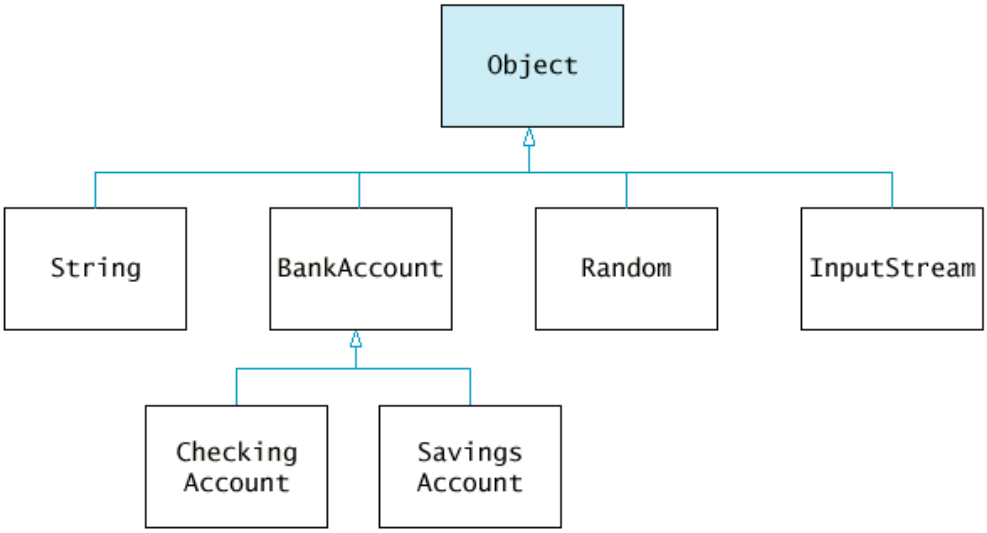
- `public`
- `private` (πρόσβαση μόνο από τις μεθόδους της ίδιας κλάσης)
- `protected` (πρόσβαση από την ίδια την κλάση αλλά και από τις υποκλάσεις)
- package access (εξ' ορισμού περίπτωση όταν δεν ορίζεται προσδιοριστικό)

Object: The Cosmic Superclass

- Όλες οι κλάσεις στην Java κληρονομούν την `Object`
- Πλέον χρήσιμες μέθοδοι:

- `String toString()` (επιστρέφει αλφαριθμητική αναπαράσταση του αντικειμένου)
- `boolean equals(Object otherObject)` (ελέγχει αν το αντικείμενο είναι το ίδιο με ένα άλλο)
- `Object clone()` (δημιουργεί ένα πλήρες αντίγραφο του αντικειμένου)

Η κλάση `Object` είναι η υπερκλάση κάθε Java κλάσης



Overriding the `clone` Method

- Η αντιγραφή μιας αναφοράς καταλήγει σε δύο αναφορές προς το ίδιο αντικείμενο
  `BankAccount account2 = account1;`
- Ορισμένες φορές, απαιτείται η αντιγραφή του αντικειμένου (δημιουργία ενός νέου αντικειμένου με την ίδια κατάσταση)
- Χρήση της `clone`:
  `BankAccount account2 = (BankAccount)account1.clone();`
- Πρέπει να γίνει ρητή μετατροπή τύπου (casting) γιατί η επιστρεφόμενη τιμή είναι τύπου `Object`
- Ορισμός της μεθόδου `clone` για τη δημιουργία νέου αντικειμένου:

```
public Object clone()
{
   BankAccount cloned = new BankAccount();
   cloned.balance = balance;
   return cloned;
}
```

Αφηρημένες (Abstract) Κλάσεις

- Μία αφηρημένη (abstract) μέθοδος, είναι μία μέθοδος για την οποία δεν προσδιορίζεται υλοποίηση
- `public abstract void dothat();`
- Δεν είναι δυνατόν να κατασκευαστούν αντικείμενα κλάσεων που έχουν έστω και μία abstract μέθοδο
- Μία κλάση για την οποία δεν μπορούν να κατασκευαστούν αντικείμενα είναι μία **αφηρημένη κλάση** (σε αντιδιαστολή προς τις συγκεκριμένες – concrete κλάσεις)
- `public abstract class BankAccount`
  `{`

```
    public abstract void deposit();
}
```

- Παρόλο που δεν μπορούν να κατασκευαστούν αντικείμενα μιας αφηρημένης κλάσης, **είναι απολύτως έγκυρο να υπάρχουν αναφορές ενός τύπου αφηρημένης κλάσης (βλ. Πολυμορφισμό)**

```
BankAccount anAccount;              //OK
anAccount = new BankAccount();      //Error, η BankAccount είναι αφηρημένη
anAccount = new SavingsAccount();   //OK
```