Προγραμματισμός ΙΙ

Η γλώσσα αντικειμενοστραφούς προγραμματισμού Java

Διδάσκων –
Δημήτριος Κατσαρός, Ph.D.

@ Τμ. Μηχανικών Η/Υ, Τηλεπικοινωνιών & Δικτύων
Πανεπιστήμιο Θεσσαλίας

Διάλεξη 5η: 29/03/2006 & 30/03/2006

Πολυμορφισμός και Αφηρημένες κλάσεις

Polymorphism and Abstract Classes

Εισαγωγή στον Πολυμορφισμό

- Υπάρχουν τρεις προγραμματιστικοί μηχανισμοί που συνιστούν στον Αντικειμενοστραφή Προγραμματισμό (OOP)
  - Ενθυλάκωση - Encapsulation
  - Κληρονομικότητα - Inheritance
  - Πολυμορφισμός - Polymorphism
- Πολυμορφισμός είναι η ικανότητα να συσχετίζουμε πολλές έννοιες στο όνομα μιας μεθόδου
  - Αυτό γίνεται διαμέσου ενός ειδικού μηχανισμού που είναι γνωστός ως *καθυστερημένη σύνδεση* ((*late binding*) ή *δυναμική σύνδεση* (*dynamic binding*)

Εισαγωγή στον Πολυμορφισμό

- Με την κληρονομικότητα μπορούμε να ορίσουμε μια βασική κλάση, και κατόπιν να παράξουμε νέες από αυτή
  - Ο κώδικας της βασικής κλάσης μπορεί να χρησιμοποιηθεί για τα δικά της αντικείμενα, καθώς και για τα αντικείμενα των παραγόμενων κλάσεων
- Με τον πολυμορφισμό μπορούμε να κάνουμε αλλαγές στους ορισμούς των μεθόδων στις παραγόμενες κλάσεις, και *να πετύχουμε οι αλλαγές αυτές να επιδρούν στο λογισμικό που έχει γραφεί για τη βασική κλάση*

Late Binding

- Η διαδικασία συσχέτισης του ορισμού μιας μεθόδου με την ενεργοποίηση μιας μεθόδου ονομάζεται *σύνδεση* (*binding*)
- Εάν ο ορισμός της μεθόδου συσχετιστεί με την ενεργοποίηση όταν μεταγλωττίζεται ο κώδικας, τότε έχουμε την *πρώιμη σύνδεση* (*early binding*)
- Εάν ο ορισμός της μεθόδου συσχετίζεται με την ενεργοποίηση όταν ενεργοποιηθεί η μέθοδος, τότε έχουμε *καθυστερημένη σύνδεση* (*late binding*) ή *δυναμική σύνδεση* (*dynamic binding*)

Late Binding

- Η Java χρησιμοποιεί την τεχνική late binding για όλες τις μεθόδους (εκτός από τις private, `final`, και static μεθόδους)
- Εξαιτίας του late binding, μια μέθοδος μπορεί να γραφεί σε μια βασική κλάση για να εκτελεί μια ενέργεια, ακόμα και εάν τμήματα αυτής της ενέργειας δεν έχουν ακόμα οριστεί
- Για παράδειγμα, θα εξετάσουμε τη σχέση μιας βασικής κλάσης με όνομα **`Sale`** και μιας παραγόμενής της με όνομα **`DiscountSale`**

Οι κλάσεις **`Sale`** και **`DiscountSale`**

- Η κλάση **`Sale`** περιέχει δυο instance variables
  - **`name`**: το όνομα ενός στοιχείου (**`String`**)
  - **`price`**: η τιμή ενός στοιχείου (**`double`**)
- Περιέχει τρεις constructors
  - Έναν no-argument constructor που θέτει το **`name`** στην τιμή **`"No name yet"`**, και την τιμή σε **`0.0`**
  - Έναν constructor με δυο παραμέτρους που δέχεται ένα **`String`** (για το **`name`**) και έναν **`double`** (για την **`price`**)
  - Έναν copy constructor που δέχεται ως παράμετρο ένα αντικείμενο **`Sale`**

Οι κλάσεις **`Sale`** και **`DiscountSale`**

- Η κλάση **`Sale`** έχει επίσης ένα σύνολο μεθόδων accessors μεθόδων (**`getName`**, **`getPrice`**), μεθόδων mutators (**`setName`**, **`setPrice`**), overridden μεθόδων **`equals`** και **`toString`**, και μια static **`announcement`** μέθοδο
- Η κλάση **`Sale`** έχει μια μέθοδο **`bill`**, που καθορίζει τη χρέωση για μια πώληση, η οποία απλά επιστρέφει την τιμή του στοιχείου
- Έχει δυο μεθόδους, **`equalDeals`** και **`lessThan`**, κάθε μια εκ των οποίων συγκρίνει δυο αντικείμενα *με σύγκριση των χρεώσεών τους* (*bills*) και επιστρέφει μια τιμή **`boolean`**

Οι κλάσεις **`Sale`** και **`DiscountSale`**

- Η κλάση **`DiscountSale`** κληρονομεί τις instance variables και μεθόδους της κλάσης **`Sale`**
- Επιπρόσθετα, έχει τη δική της instance variable, **`discount`** (ποσοστό επί της **`price`**), και τους δικούς της constructor methods, μέθοδο accessor (**`getDiscount`**), μέθοδο mutator (**`setDiscount`**), overriden μέθοδο **`toString`**, και τη static μέθοδο **`announcement`**
- Η κλάση **`DiscountSale`** έχει τη δική της μέθοδο **`bill`** που υπολογίζει το bill ως συνάρτηση της **`discount`** και της **`price`**

Οι κλάσεις **Sale** και **DiscountSale**

- Η μέθοδος **lessThan** της κλάσης **Sale**
  - Σημειώστε τις ενεργοποιήσεις της μεθόδου **bill()** :

```
public boolean lessThan (Sale otherSale)
{
  if (otherSale == null)
  {
    System.out.println("Error: null object");
    System.exit(0);
  }
  return (bill( ) < otherSale.bill( ));
}
```

Οι κλάσεις **Sale** και **DiscountSale**

- Η μέθοδος **bill()** της κλάσης **Sale** :

```
public double bill( )
{
  return price;
}
```

- Η μέθοδος **bill()** της κλάσης **DiscountSale** :

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

Οι κλάσεις **Sale** και **DiscountSale**

- Δεδομένου του ακόλουθου προγράμματος:

```
  . . .
Sale simple = new sale("floor mat", 10.00);
DiscountSale discount = new
          DiscountSale("floor mat", 11.00, 10);
  . . .
if (discount.lessThan(simple))
  System.out.println("$" + discount.bill() +
                " < " + "$" + simple.bill() +
                " because late-binding works!");
  . . .
```

  - Η έξοδος θα είναι:

```
$9.90 < $10 because late-binding works!
```

Οι κλάσεις `Sale` και `DiscountSale`

- Στο προηγούμενο παράδειγμα, η `boolean` έκφραση στη δήλωση `if` επιστρέφει `true`
- Όπως δείχνει η έξοδος, όταν εκτελείται η μέθοδος `lessThan` στην κλάση `Sale`, γνωρίζει ποια μέθοδο `bill()` να ενεργοποιήσει
  - Τη μέθοδος `bill()` της κλάσης `DiscountSale` για `discount`, και τη μέθοδο `bill()` της κλάσης `Sale` για `simple`
- Σημειώστε ότι δημιουργήθηκε και μεταγλωττίστηκε η κλάση `Sale`, η κλάση `DiscountSale` και η μέθοδός της `bill()` δεν υπήρχαν
  - Αυτά τα αποτελέσματα είναι εφικτά με το μηχανισμό late-binding

Παγίδα: Όχι late binding για static μεθόδους

- Όταν γίνεται η απόφαση κατά τη μεταγλώττιση για το ποια μέθοδο θα χρησιμοποιηθεί, αυτό αποκαλείται *στατική σύνδεση* (*static binding*)
  - Αυτή η απόφαση γίνεται με βάση *τον τύπο της μεταβλητής που ονοματίζει το αντικείμενο*
- Η Java χρησιμοποιεί static binding, όχι late binding για τις private, **`final`**, και static μεθόδους
  - Στην περίπτωση των **`private`** και των **`final`** μεθόδων, το late binding δεν θα εξυπηρετούσε κανένα σκοπό
  - Όμως, στην περίπτωση μιας static μεθόδου που ενεργοποιείται με χρήση ενός καλούντος αντικειμένου, δεν έχει καμία διαφορά

Παγίδα: Όχι late binding για static μεθόδους

- Η μέθοδος **`announcement()`** της κλάσης **`Sale`** :

```
public static void announcement( )
{
  System.out.println("Sale class");
}
```

- Η μέθοδος **`announcement()`** της κλάσης **`DiscountSale`** :

```
public static void announcement( )
{
  System.out.println("DiscountSale class");
}
```

Παγίδα: Όχι late binding για static μεθόδους

- Στο προηγούμενο παράδειγμα, δημιουργήθηκαν τα αντικείμενα **`simple`** (της κλάσης **`Sale`**) και **`discount`** (της κλάσης **`DiscountClass`**)
- Δεδομένης της ακόλουθης ανάθεσης:

```
simple = discount;
```

  - Οι δυο μεταβλητές δείχνουν στο ίδιο αντικείμενο
  - Ειδικότερα, η μεταβλητή τύπου κλάσης **`Sale`** ονοματίζει ένα αντικείμενο **`DiscountClass`**

Παγίδα: Όχι late binding για static μεθόδους

- Δεδομένης της ενεργοποίησης:

```
simple.announcement();
```

  - Η έξοδος είναι:

```
Sale class
```

- Σημειώστε ότι εδώ,η μέθοδος **`announcement`** είναι static που ενεργοποιείται από ένα καλούν αντικείμενο (αντί για το όνομα της κλάσης της)
  - Επομένως ο τύπος της **`simple`** προσδιορίζεται από το όνομα της μεταβλητής, όχι από από το αντικείμενο στο οποίο αναφέρεται

Παγίδα: Όχι late binding για static μεθόδους

- Υπάρχουν άλλες περιπτώσεις όπου μια static μέθοδος έχει καλούν αντικείμενο
- Για παράδειγμα, μια static μέθοδος μπορεί να ενεργοποιηθεί μέσα στον ορισμό μιας non-static μεθόδου, αλλά χωρίς να υπάρχει ρητά το όνομα κλάσης ή το καλούν αντικείμενο
- Στην περίπτωση αυτή, το καλούν αντικείμενο είναι το **`this`**

Ο modifier **`final`**

- Μια *μέθοδος* που σημειώνεται ως **`final`** σημαίνει ότι δεν μπορεί να γίνει override με νέο ορισμό σε μια παραγόμενη κλάση
  - Εάν είναι **`final`**, ο compiler μπορεί να χρησιμοποιήσει early binding με τη μέθοδο αυτή

```
public final void someMethod() { . . . }
```

- Μια *κλάση* που σημειώνεται ως **`final`** σημαίνει ότι δεν μπορεί να χρησιμοποιηθεί ως βασική κλάση από την οποία να παράξουμε άλλες κλάσεις

Late binding με τη μέθοδο `toString`

- Εάν μια κατάλληλη μέθοδος **`toString`** οριστεί για μια κλάση, τότε ένα αντικείμενο αυτής της κλάσης μπορεί να τυπωθεί με τη **`System.out.println`**

```
Sale aSale = new Sale("tire gauge",
  9.95);
System.out.println(aSale);
```

  - Η έξοδος που παράγεται:

```
tire gauge Price and total cost = $9.95
```

- Αυτή δουλεύει εξαιτίας του late binding

Late binding με τη μέθοδο `toString`

- Ένας ορισμός της μεθόδου **`println`** δέχεται ως όρισμα ένα αντικείμενο τύπου **`Object`**:

```
public void println(Object theObject)
{
  System.out.println(theObject.toString());
}
```

  - Με τη σειρά της, ενεργοποιεί τη **`println`** που δέχεται ένα όρισμα τύπου **`String`**
- Σημειώστε ότι η μέθοδος **`println`** ορίστηκε πριν αναπτυχθεί η κλάση **`Sale`**
- Όμως, εξαιτίας του late binding, χρησιμοποιείται η μέθοδος **`toString`** της κλάσης **`Sale`**, και όχι η μέθοδος **`toString`** της κλάσης **`Object`**

Κάθε αντικείμενο γνωρίζει τους ορισμούς των μεθόδων του

- Ο τύπος μιας μεταβλητής κλάσης καθορίζει ποια ονόματα μεθόδων μπορούν να χρησιμοποιηθούν με τη μεταβλητή
  - Όμως, το αντικείμενο που ονοματίζεται με τη μεταβλητή καθορίζει ποιος ορισμός με το ίδιο όνομα μεθόδου θα χρησιμοποιηθεί
- Μια ειδική περίπτωση αυτού του κανόνα έχει ως εξής:
  - Ο τύπος μια παραμέτρου τύπου κλάσης καθορίζει ποια ονόματα μεθόδων μπορούν να χρησιμοποιηθούν με την παράμετρο
  - Το όρισμα καθορίζει ποιος ορισμός της μεθόδου θα χρησιμοποιηθεί

Upcasting και Downcasting

- *Upcasting* είναι όταν ένα αντικείμενο μιας παραγόμενης κλάσης ανατίθεται σε μια μεταβλητή τύπου της βασικής κλάσης (ή οποιασδήποτε κλάσης προγόνου)

```
Sale saleVariable; //Base class
DiscountSale discountVariable = new
   DiscountSale("paint", 15,10); //Derived class
saleVariable = discountVariable; //Upcasting
System.out.println(saleVariable.toString());
```

- Εξαιτίας του late binding, η **`toString`** χρησιμοποιεί τον ορισμό που παρέχεται στην κλάση **`DiscountSale`**

Upcasting και Downcasting

- *Downcasting* είναι όταν εκτελείται μια προσαρμογή τύπου (type cast) από μια βασική κλάση σε μια παραγόμενη (ή από μια κλάση πρόγονο σε μια οποιαδήποτε κλάση απόγονο)
  - Το downcasting πρέπει να γίνεται πολύ προσεκτικά
  - Σε αρκετές περιπτώσεις δεν έχει νόημα, ή δεν είναι έγκυρο:

```
discountVariable =                    //will produce
     (DiscountSale)saleVariable;//run-time error
discountVariable = saleVariable //will produce
                                      //compiler error
```

  - Υπάρχουν περιπτώσεις όμως, όπου το downcasting είναι απαραίτητο, π.χ., μέσα στη μέθοδο **`equals`** για μια κλάση:

```
Sale otherSale = (Sale)otherObject;//downcasting
```

Παγίδα: Downcasting

- Είναι στην ευθύνη του προγραμματιστή να χρησιμοποιήσει το downcasting μόνο στις περιπτώσεις όπου έχει νόημα
  - Ο compiler δεν ελέγχει εάν το downcasting έχει νόημα να εκτελεστεί
- Χρησιμοποιώντας downcasting σε μια περίπτωση όπου δεν έχει νόημα δεν παράγει run-time error

Υπόδειξη: Έλεγχος εάν το downcasting έχει νόημα

- Το downcasting σε έναν τύπο έχει νόημα εάν το αντικείμενο όπου το εφαρμόζουμε είναι μια instance αυτού του τύπου
  - Αυτό ακριβώς ελέγχει ο τελεστής **`instanceof`** :
    `object` **`instanceof`** ***`ClassName`***
  - Θα επιστρέψει true εάν το ***`object`*** είναι τύπου ***`ClassName`***
  - Ειδικότερα, θα επιστρέψει true εάν το ***`object`*** είναι instance μιας οποιαδήποτε κλάσης απογόνου της ***`ClassName`***

Πρώτη ματιά στη μέθοδο `clone`

- Κάθε αντικείμενο κληρονομεί μια μέθοδο με όνομα `clone` από την κλάση `Object`
  - Η μέθοδος `clone` δεν δέχεται παραμέτρους
  - Υποτίθεται ότι επιστρέφει ένα deep copy του καλούντος αντικειμένου
- Όμως, η κληρονομούμενη έκδοση της μεθόδου δεν σχεδιάστηκε για να χρησιμοποιηθεί έτσι από όλες τις κλάσεις
  - Κάθε κλάση πρέπει να την κάνει override με μια πιο κατάλληλη έκδοση

Πρώτη ματιά στη μέθοδο `clone`

- Η κεφαλίδα της μεθόδου **`clone`** που ορίζεται στην κλάση **`Object`** είναι ως εξής:
  `protected Object clone()`
- Η κεφαλίδα για μια μέθοδο **`clone`** που θα κάνει override τη μέθοδο **`clone`** της κλάσης **`Object`** μπορεί να διαφέρει ελαφρά από την παραπάνω κεφαλίδα
  - Μια αλλαγή σε μια έκδοση με περισσότερες ελευθερίες, όπως από protected σε public, είναι επιτρεπτή
  - Αλλάζοντας τον επιστρεφόμενο τύπο από **`Object`** σε τύπο της κλάσης που κλωνοποιείται, επιτρέπεται επειδή κάθε κλάση στη Java είναι κλάση-απόγονος της κλάσης **`Object`**
  - Αυτό είναι ένα παράδειγμα του *covariant return type*

Πρώτη ματιά στη μέθοδο `clone`

- Εάν μια κλάση έχει έναν copy constructor, η μέθοδος **`clone`** για την κλάση αυτή μπορεί να χρησιμοποιήσει τον *copy constructor* για να δημιουργήσει ένα αντίγραφο που θα επιστρέφεται από τη μέθοδο **`clone`**

```
public Sale clone()
{
  return new Sale(this);
}
```

κι άλλο παράδειγμα:

```
public DiscountSale clone()
{
  return new DiscountSale(this);
}
```

Παγίδα: Μερικές φορές ο επιστρεφόμενος τύπος της μεθόδου `clone` είναι `Object`

- Πριν από την έκδοση 5.0, η Java δεν επέτρεπε covariant return types
- Επομένως, η μέθοδος `clone` όλων των κλάσεων έχουν το `Object` ως επιστρεφόμενο τύπο
  - Αφού ο επιστρεφόμενος τύπος της μεθόδου clone για την κλάση `Object` είναι ένα αντικείμενο `Object`, ο επιστρεφόμενος τύπος για την overriding clone μέθοδο οποιασδήποτε άλλης κλάσης είναι επίσης `Object`

Παγίδα: Μερικές φορές ο επιστρεφόμενος τύπος της μεθόδου `clone` είναι `Object`

- Πριν από την έκδοση Java 5.0, η μέθοδος **`clone`** για την κλάση **`Sale`** θα ήταν ως εξής:

```
public Object clone()
{
  return new Sale(this);
}
```

- Επομένως, στο αποτέλεσμα πρέπει πάντα να γίνεται type cast όταν χρησιμοποιούμε τη μέθοδο **`clone`** γραμμένη για μια παλιότερη έκδοση της Java

```
Sale copy = (Sale)original.clone();
```

Παγίδα: Μερικές φορές ο επιστρεφόμενος τύπος της μεθόδου **`clone`** είναι **`Object`**

- Είναι έγκυρο να χρησιμοποιούμε το Object ως επιστρεφόμενο τύπο για μια μέθοδο clone, ακόμα και κλάσεις που ορίζονται στην έκδοση 5.0 της Java
  - Όταν υπάρχουν αμφιβολίες, δεν είναι εσφαλμένο να περιλαμβάνουμε το type cast
  - Για παράδειγμα, το επόμενο είναι έγκυρο για τη μέθοδο clone για την κλάση Sale:

```
Sale copy = original.clone();
```

  - Όμως, προσθέτοντας το επόμενο type cast δεν δημιουργεί προβλήματα:

```
Sale copy = (Sale)original.clone();
```

Παγίδα: Περιορισμοί των Copy Constructors

- Παρόλο που ο copy constructor και η μέθοδος **`clone`** για μια κλάση εμφανίζονται να κάνουν το ίδιο πράγμα, υπάρχουν περιπτώσεις όπου μόνο η **`clone`** θα δουλέψει
- Για παράδειγμα, δεδομένης μια μεθόδου **`badcopy`** στην κλάση **`Sale`** που αντιγράφει έναν πίνακα από sales
  - Εάν αυτός ο πίνακας από sales περιέχει αντικείμενο από μια παραγόμενη κλάση της **`Sale`** (δηλ., **`DiscountSale`**), τότε το αντίγραφο θα είναι απλώς sale, όχι ένα πραγματικό αντίγραφο

```
b[i] = new Sale(a[i]); //plain Sale object
```

Παγίδα: Περιορισμοί των Copy Constructors

- Όμως, εάν χρησιμοποιηθεί η μέθοδος **`clone`** αντί για τον copy constructor, τότε (εξαιτίας του late binding) δημιουργείται ένα πραγματικό αντίγραφο, ακόμα και για αντικείμενα μιας παραγόμενης κλάσης (δηλ., **`DiscountSale`**):

```
b[i] = (a[i].clone());//DiscountSale object
```

  - Ο λόγος που αυτό δουλεύει είναι εξαιτίας του ότι η μέθοδος **`clone`** έχει το ίδιο όνομα σε όλες τις κλάσεις, και ο πολυμορφισμός δουλεύει με ονόματα μεθόδων
  - Οι copy constructors με όνομα **`Sale`** και **`DiscountSale`** έχουν διαφορετικά ονόματα, και ο πολυμορφισμός δεν δουλεύει με μεθόδους με διαφορετικά ονόματα

Εισαγωγή στις Αφηρημένες κλάσεις (Abstract classes)

- Θυμηθείτε, η βασική κλάση **`Employee`** έχει δυο παραγόμενες κλάσεις, **`HourlyEmployee`** και **`SalariedEmployee`**
- Η επόμενη μέθοδος προστίθεται στην κλάση **`Employee`**
  - Συγκρίνει δυο employees εάν έχουν το ίδιο μισθό:

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

Εισαγωγή στις Αφηρημένες κλάσεις

- Υπάρχουν διάφορα προβλήματα για τη μέθοδο αυτή:
  - Η μέθοδος **`getPay`** ενεργοποιείται στη μέθοδος **`samePay`**
  - Υπάρχουν μέθοδοι **`getPay`** σε κάθε μια από τις παραγόμενες κλάσεις
  - Δεν υπάρχει μέθοδος **`getPay`** στην κλάση **`Employee`**, ούτε υπάρχει τρόπος να την ορίσουμε με κάποιο λογικό τρόπο χωρίς να ξέρουμε εάν ένας employee είναι hourly ή salaried

Εισαγωγή στις Αφηρημένες κλάσεις

- Η ιδανική κατάσταση θα ήταν εάν υπήρχε τρόπος να:
  - Αναβάλλουμε τον ορισμό της μεθόδου **`getPay`** μέχρι ο τύπος του employee να γίνει γνωστός (δηλ., στις παραγόμενες κλάσεις)
  - Αφήσουμε κάποιο είδος σημείωσης στην κλάση **`Employee`** για να σηματοδοτήσουμε αυτό
- Η Java το επιτρέπει αυτό με τη χρήση των αφηρημένων κλάσεων (abstract classes) και αφηρημένων μεθόδων (abstract methods)

Εισαγωγή στις Αφηρημένες κλάσεις

- Για να αναβάλλουμε τον ορισμό της μεθόδου, η Java επιτρέπει να δηλωθεί μια *αφηρημένη μέθοδος* (*abstract method*)
  - Μια abstract method έχει κεφαλίδα, αλλά δεν έχει σώμα
  - Το σώμα της μεθόδου ορίζεται στις παραγόμενες κλάσεις
- Η κλάση που περιέχει μια abstract method αποκαλείται *αφηρημένη κλάση* (*abstract class*)

Η έννοια της Abstract μεθόδου

- Μια abstract method είναι όπως ένας placeholder για μια μέθοδο που θα οριστεί πλήρως σε μια κλάση απόγονο
- Έχει πλήρη κεφαλίδα, στην οποία έχει προστεθεί ο modifier **`abstract`**
- Δεν μπορεί να είναι private
- Δεν έχει σώμα, και τελειώνει με ένα semicolon αντί για το σώμα της

```
public abstract double getPay();
public abstract void doIt(int count);
```

Η έννοια της Abstract κλάσης

- Μια κλάση που περιλαμβάνει μια τουλάχιστον abstract method αποκαλείται *abstract class*
  - Μια abstract class πρέπει να περιλαμβάνει τον modifier **`abstract`** στην κεφαλίδα της:

```
public abstract class Employee
{
  private instanceVariables;
  . . .
  public abstract double getPay();
  . . .
}
```

Η έννοια της Abstract κλάσης

- Μια abstract μπορεί να περιλαμβάνει οσεσδήποτε abstract και/ή πλήρως ορισμένες μεθόδους
- Εάν μια παραγόμενη κλάση μιας abstract class προσθέσει στις ή δεν ορίσει όλες τις abstract methods, τότε είναι και αυτή abstract, και πρέπει να προσθέτουμε τον modifier **`abstract`**

- Μια κλάση που δεν έχει abstract μεθόδους αποκαλείται *concrete class*

Παγίδα: Δεν επιτρέπεται η δημιουργία αντικειμένων μιας αφηρημένης κλάσης

- Μια abstract κλάση μπορεί να χρησιμοποιηθεί μόνο για να παράξουμε άλλες πιο εξειδικευμένες κλάσεις
- Ένας constructor μιας abstract class δεν μπορεί να χρησιμοποιηθεί για να δημιουργήσουμε ένα αντικείμενο μιας abstract class
  - Όμως, ο constructor μιας παραγόμενης κλάσης θα πρέπει να περιλαμβάνει μια ενεργοποίηση του constructor της abstract class με τη μορφή του **super**

Υπόδειξη: Μια αφηρημένη κλάση είναι ένας Τύπος δεδομένων

- Παρόλο που ένα αντικείμενο μιας abstract class δεν μπορεί να δημιουργηθεί, είναι απολύτως έγκυρο να έχουμε μια παράμετρο τύπου abstract class
  - Αυτό καθιστά δυνατό να ενσωματώσουμε ένα αντικείμενο που ανήκει σε μια οποιαδήποτε κλάση απόγονο της abstract κλάσης
- Είναι επίσης έγκυρο να χρησιμοποιήσουμε μια μεταβλητή τύπου abstract κλάσης, καθόσον ονοματίζει αντικείμενα των concrete κλάσεων απογόνων

Στην επόμενη διάλεξη …

- Εξαιρέσεις (Exceptions)