ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ
ATHENS UNIVERSITY OF ECONOMICS AND BUSINESS
ΚΕΝΤΡΟ ΕΠΙΜΟΡΦΩΣΗΣ ΚΑΙ ΔΙΑ ΒΙΟΥ ΜΑΘΗΣΗΣ

Κληρονομικότητα και πολυμορφισμός

Αθ. Ανδρούτσος

Πυλώνες Ο-Ο Programming

- Τρεις είναι οι βασικοί πυλώνες στον Αντικειμενοστραφή Προγραμματισμό
  - **Ενθυλάκωση** (Encapsulation)
  - **Κληρονομικότητα** (Inheritance) και **ιδεατές μέθοδοι** (virtual methods – overridable methods)
  - **Πολυμορφισμός** (Polymorphism)

Ενθυλάκωση (1)

- Ο πιο σημαντικός παράγοντας που διακρίνει μία καλά σχεδιασμένη κλάση από μία φτωχά (poorly) σχεδιασμένη είναι ο βαθμός στον οποίο **η κλάση αποκρύβει τα εσωτερικά της δεδομένα και άλλες εσωτερικές λεπτομέρειες υλοποίησης** (γενικά δηλαδή τα class internals) από άλλες κλάσεις
- Μία **καλοσχεδιασμένη κλάση αποκρύβει όλες τις λεπτομέρειες υλοποίησής της**, διαχωρίζοντας το API της από την **εσωτερική υλοποίηση**

Ενθυλάκωση (2)

- Στη συνέχεια, τα instances επικοινωνούν μεταξύ τους μόνο μέσω των public API τους και αγνοούν το ένα τις εσωτερικές λειτουργίες του άλλου
- Η απόκρυψη πληροφοριών (information hiding) ή **ενθυλάκωση (encapsulation), είναι μια θεμελιώδης αρχή του σχεδιασμού λογισμικού**

Ενθυλάκωση (3)

- Η βασική αρχή της **Ενθυλάκωσης** είναι: "Make each class or class member as inaccessible as possible".
- Πρακτικά, αυτό σημαίνει:
  - Private fields
  - Διαχωρισμός το API (public & protected methods) από την εσωτερική υλοποίηση της κλάσης μας (private & package private methods)
  - Στα doc comments του API αναφέρουμε ΤΙ κάνει η μέθοδος και όχι ΠΩΣ το κάνει

Κληρονομικότητα (Inheritance)

- Οι κλάσεις μπορούν να ***κληρονομούν (derive)*** από άλλες (υπέρ)κλάσεις, που σημαίνει ότι **περιέχουν** τα πεδία και τις μεθόδους της υπερκλάσης (εκτός των constructors και όσων πεδίων και μεθόδων είναι private και package-private) και **μπορούν να ορίζουν επιπλέον** και τα δικά τους πεδία και μεθόδους
- Η κληρονομικότητα είναι επομένως μία μορφή **επαναχρησιμοποίησης** μία κλάσης

Κληρονομικότητα

- Η κληρονομικότητα έχει νόημα όταν η υπερκλάση είναι μία πιο **γενική κλάση** (more general class) ενώ η υποκλάση είναι μια πιο **ειδική κλάση** (more specific class)
- Οπότε τότε λέμε ότι **η σχέση μεταξύ υποκλάσης-υπερκλάσης είναι μία σχέση 'IS-A'** (για παράδειγμα Participant IS-A User)

Superclasses - Subclasses

- Υπερκλάσεις (Superclasses)
  - Περιέχουν **μεθόδους και πεδία που κληρονομούνται** από τις υποκλάσεις (subclasses ή derived classes)
- Υποκλάσεις (Subclasses)
  - Κληρονομούν πεδία και μεθόδους από τις υπερκλάσεις
  - Μπορεί να ορίζουν επιπλέον πεδία και μεθόδους
  - Μπορούν να **επαναορίζουν / υπερκαλύπτουν (override)** μεθόδους που κληρονομούν από τις υπερκλάσεις

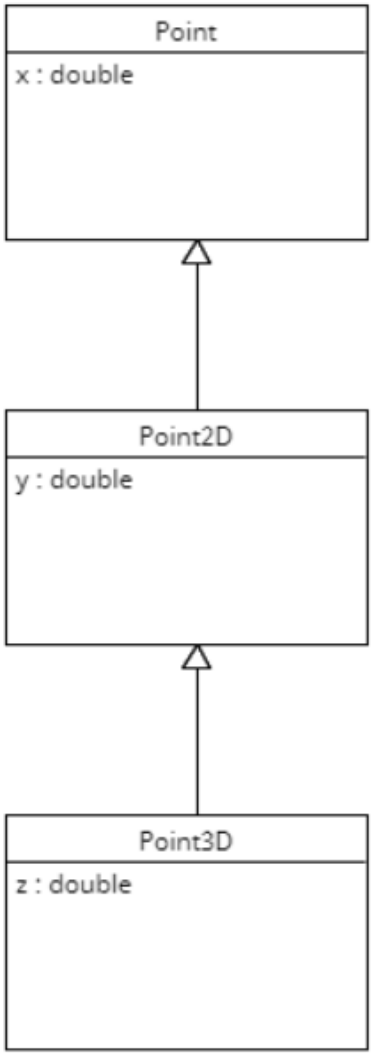
Υπερκάλυψη - Override

- Η **υπερκάλυψη (override)** τεχνικά σημαίνει ότι μία υποκλάση μπορεί να περιλαμβάνει μεθόδους της υπερκλάσης με την ίδια ακριβώς υπογραφή αλλά διαφορετικό σώμα (διαφορετική υλοποίηση)
- Είναι μία σημαντική ιδιότητα στην κληρονομικότητα που θα εξερευνήσουμε στα επόμενα

extends keyword

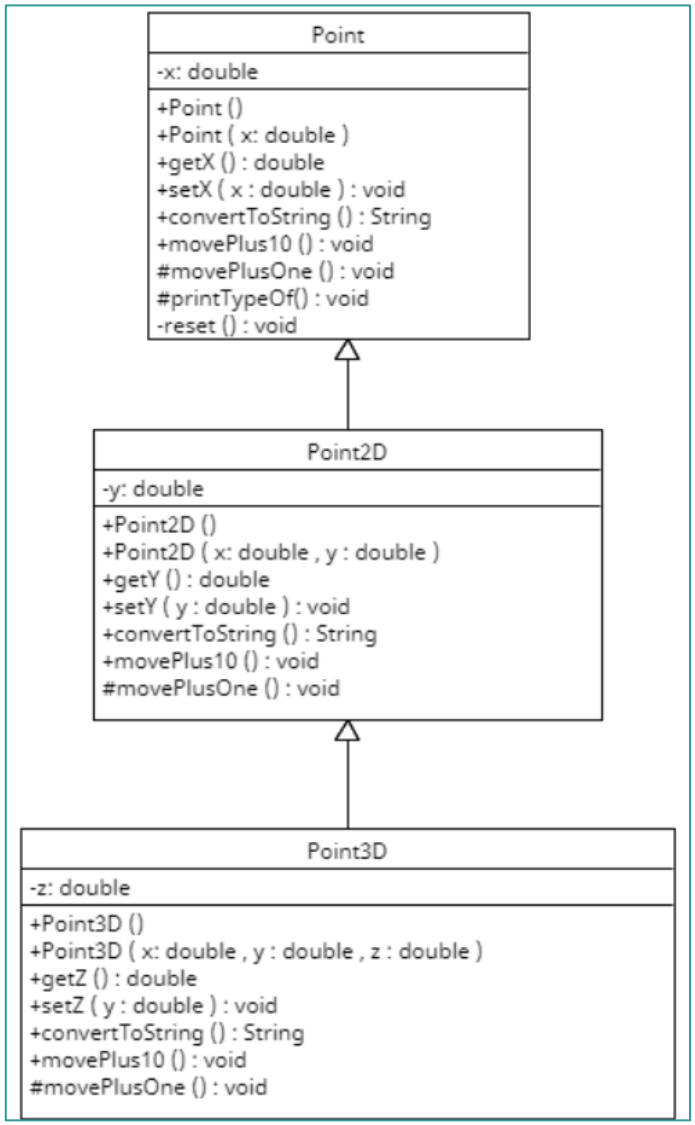
- Στην Java κληρονομούμε από μία κλάση χρησιμοποιώντας το keyword: **extends**
- Στη συνέχεια θα δημιουργήσουμε μία κλάση **Point** που ορίζει ένα σημείο στον άξονα x, και που θα λειτουργήσει ως root superclass για δύο κλάσεις Point2D και Point3D
- **public class** Point2D **extends** Point {
  // code
  }
- **public class** Point3D **extends** Point2D {
  // code
  }

Ιεραρχία Κληρονομικότητας



- Το UML **Domain Model** αναπαριστά την ιεραρχία κληρονομικότητας, όπου η κλάση Point αναπαριστά ένα ***σημείο x σε μια ευθεία γραμμή***
- Η κλάση Point2D ορίζει ένα **σημείο (x, y) στο επίπεδο.**
- Η Point3D ορίζει ένα ***σημείο (x, y, z) στο χώρο***
- Το σύμβολο ⇧ ορίζει τη σχέση κληρονομικότητας μεταξύ των κλάσεων, δεδομένου ότι για να ορίσουμε την Point2D δεν χρειάζεται να το κάνουμε από την αρχή αλλά μπορούμε να κληρονομήσουμε την Point και να προσθέσουμε επιπλέον στοιχεία της κλάσης Point2D
- Το ίδιο ισχύει και για την κλάση Point3D η οποία δεν χρειάζεται να γραφεί από την αρχή αλλά κληρονομεί από την Point2D και μέσω της Point2D και από την Point και προσθέτει τα επιπλέον στοιχεία που χρειάζεται

UML Class Diagram



- Στο **Class Diagram** όπου σε κάθε class περιλαμβάνονται όλα τα class members με τους access modifiers (+, -, #, ~) οι protected μέθοδοι απεικονίζονται με #
- Στις derived κλάσεις δεν χρειάζεται να απεικονίζουμε τις μεθόδους που κληρονομούνται εκτός αν γίνονται override

Κλάση Point (1)

Point.java

```
1   package testbed.ch14;
2
3   /**
4    * Defines a one-dimension point.
5    */
6   public class Point {
7       private double x;
8
9       public Point() {}
10
11      public Point(double x) {
12          this.x = x;
13      }
14
15      public double getX() {
16          return x;
17      }
18
19      public void setX(double x) {
20          this.x = x;
21      }
22
23      public String convertToString() {
24          return "(" + x + ")";
25      }
```

- Η κλάση Point είναι η root superclass στην ιεραρχία που σχεδιάσαμε
- Περιλαμβάνει πεδία και μεθόδους ενός JavaBean
- Οι κλάσεις Point2D και Point3D θα κληρονομήσουν από την Point

Κλάση Point (2)

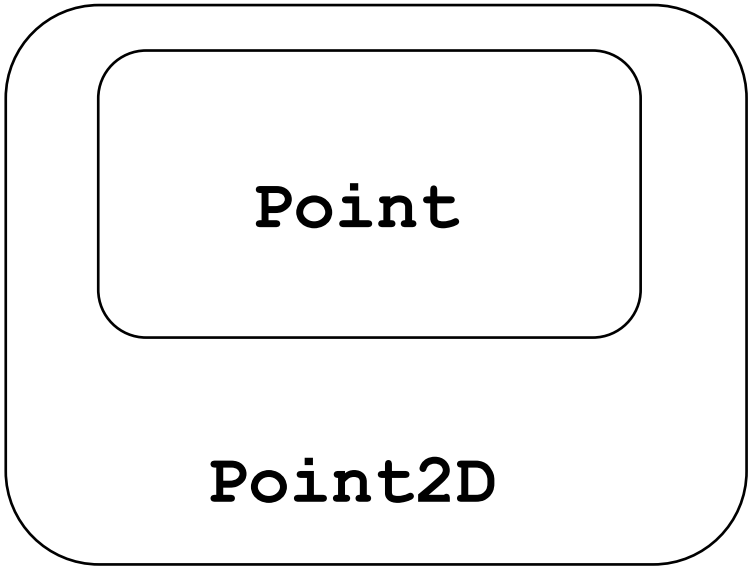

```
27      public void movePlus10() {
28          //    x += 10;
29
30          // Self-use of movePlusOne
31          for (int i = 1; i <= 10; i++) {
32              movePlusOne();
33          }
34      }
35
            1 override
36      protected void movePlusOne() {
37          x++;
38      }
39
40      protected void printTypeOf() {
41          System.out.println(this.getClass().getSimpleName());
42      }
43
44      private void reset() {
45          x = 0;
46      }
47  }
```

- Η movePlus10() είναι public και είναι μέρος του API της κλάσης
- Οι movePlusOne() και printTypeOf() είναι protected για να μπορούν να κληρονομηθούν (και επομένως να μπορούν να χρησιμοποιηθούν από τις derived κλάσεις, αλλιώς θα ήταν private). Είναι μέρος του API της κλάσης
- Η reset() είναι private και δεν μπορεί να κληρονομηθεί. Δεν είναι μέρος του API
- Η ***getClass().getSimpleName()*** επιστρέφει το όνομα της κλάσης του αντικειμένου

Κλάση Point2D (1)

- Έστω ότι θέλουμε να αναπτύξουμε μία κλάση Point2D που αναπαριστά ένα point στο δισδιάστατο επίπεδο
- Στο τεχνικό επίπεδο, αντί να την γράψουμε από την αρχή, μπορούμε να εκμεταλλευτούμε την κληρονομικότητα και να κληρονομήσουμε από την Point

Κλάση Point2D (2)



- Λέγοντας ότι η **`Point2D`** κληρονομεί την **`Point`** εννοούμε ότι η κλάση Point2D ***περιλαμβάνει και επεκτείνει (extends))*** την κλάση Point
- Έτσι όταν γίνεται new η Point2D, ***δημιουργείται ένα instance*** της Point2D που περιλαμβάνει ένα instance της Point
- Η **`Point2D`** μπορεί να προσθέσει νέα μέλη

Point2D (1)


```java
package testbed.ch15;

public class Point2D extends Point {
    private double y;

    public Point2D() {
        super();
        y = 0;
    }
    public Point2D(double x, double y) {
        super(x);
        this.y = y;
    }
    public double getY() {
        return y;
    }
    public void setY(double y) {
        this.y = y;
    }

```

- Η Point2D κάνει extends την Point. Με αυτό τον τρόπο δηλώνουμε ότι η κλάση Point2D κληρονομεί από την Point. Επίσης, ορίζει ένα επιπλέον πεδίο, το y
- Ο default constructor της **Point2D** πρέπει ***ως 1η εντολή*** να καλέσει τον default constructor του Point με την ***super()*** ώστε να **αρχικοποιήσει το Point** και **μετά να αρχικοποιήσει το δικό του μέρος** (το πεδίο y) (ο super είναι ένας δείκτης που δείχνει στο parent instance, όπως το this δείχνει στο τρέχον instance και το super() είναι μία ειδική μέθοδος που αναφέρεται στον default constructor του parent class)

Point2D (2)


```
1  package testbed.ch15;
2
3  public class Point2D extends Point {
4      private double y;
5
6      public Point2D() {
7          // super();
8          // y = 0;
9      }
10     public Point2D(double x, double y) {
11         super(x);
12         this.y = y;
13     }
14     public double getY() {
15         return y;
16     }
17     public void setY(double y) {
18         this.y = y;
19     }
```

- Οι γραμμές 7 και 8 παρέχονται και αυτόματα από το runtime
- Ο overloaded constructor ως 1η εντολή καλεί την super(x) δηλαδή τον overloaded constructor της parent class
- Θα μπορούσε και εδώ αντί της γραμμής 11 να καλούμε το super() (το οποίο καλείται και by default) και μετά να κάνουμε this.setX(x);
- Σε κάθε περίπτωση προϋπόθεση είναι να υπάρχει **public ή protected default ή overloaded constructor** στην superclass, αλλιώς δεν μπορούμε να κληρονομήσουμε

Point2D (3)


```
22      public String convertToString() {
23          return "(" + getX() + ", " + y + ")";
24      }
25
            1 override
26      @Override
27      public void movePlus10() {
28          super.movePlus10();
29          y += 10;
30
31          // self-use of movePlusOme()
32          /*for (int i = 1; i <= 1; i++) {
33              this.movePlusOne();
34          }*/
35      }
36
            1 override
37      @Override
38      protected void movePlusOne() {
39          super.movePlusOne();
40          y += 1;
41      }
```

- Στη συνέχεια μπορούμε να κάνουμε override όσες μεθόδους έχουμε κληρονομήσει και θέλουμε να αλλάξουμε τη λειτουργία τους διατηρώντας την ίδια υπογραφή
- Για παράδειγμα η convertToString(), η movePlus10() και η movePlusOne() γίνονται override
- Το annotation **@override** δεν είναι υποχρεωτικό αλλά είναι σημαντικό να υπάρχει ώστε ο μεταγλωττιστής να ελέγχει αν η μέθοδος που κάνει override έχει την ίδια υπογραφή με την overridable μέθοδο της superclass

Liskov substitution principle

- Επίσης ο μεταγλωττιστής ελέγχει αν η μέθοδος που κάνει override έχει το ίδιο ή υψηλότερο επίπεδο access modifier
- Μία μέθοδος που κάνει override μία superclass method δεν μπορεί να έχει πιο restrictive access level από ότι η superclass method ώστε να διασφαλίσουμε ότι ένα instance της subclass είναι το ίδιο χρήσιμο όσο και ένα instance της superclass (Liskov substitution principle)

UML - Point και Point2D

Point
- x: int
+Point() +Point(x: double) +getX (): double +setX (x : double): void +convertToString () : String +movePlus10 () : void #movePlusOne () : void #printTypeOf() : void -reset () : void

Point2D
- y: int
Point2D() **Point2D(x : double,** **y : double)** +getX (): double +setX (x : double): void **+getY () : double** **+setY (y : double) : void** **+convertToString () : String** **+movePlus10 () : void** **#movePlusOne () : void** #printTypeOf() : void

- Κλάσεις ***Point*** και ***Point2D*** μετά την κληρονομικότητα
  - Αριστερά στην Point, με μπλε είναι τα μέλη της Point που έχουν κληρονομηθεί
  - Δεξιά στην Point2D,
    - Με **σκούρο κόκκινο** είναι τα μέλη που έχει προσθέσει η Point2D
    - Με μπλε είναι τα κληρονομημένα μέλη που δεν έχουν γίνει override
    - Με **μπλε** είναι τα κληρονομημένα μέλη που έχουν γίνει override

Derived Κλάσεις

- Παρατηρούμε πως στο πλαίσιο της κληρονομικότητας, τα **private (και package-private) members και οι Constructors δεν κληρονομούνται**
- Όλα τα άλλα μέλη κληρονομούνται

Χαρακτηρισμός protected

- Μπορούμε επομένως να χαρακτηρίζουμε τα μέλη μιας κλάσης ως protected όταν σκοπεύουμε να τα χρησιμοποιήσουμε για κληρονομικότητα
- Ενώ τα private μέλη μιας κλάσης δεν μπορούν να κληρονομηθούν, τα *protected μέλη κληρονομούνται ενώ ταυτόχρονα θεωρούνται private έξω από το package στο οποίο ορίζονται*

Δημιουργοί της subclass (1)

- Οι υποκλάσεις όπως είπαμε δεν κληρονομούν τους constructors της superclass
- **Οι δημιουργοί της subclass θα πρέπει πάντα να έχουν ως πρώτη εντολή την κλήση στον δημιουργό της superclass με τη χρήση της super**
- Διαφορετικά, αυτόματα το runtime καλεί ως πρώτη εντολή την **super()**. Αν υπάρχει μία ιεραρχία κληρονομικότητας τότε ο κάθε constructor της subclass καλεί τον constructor της superclass (*constructor chaining*)
- Επομένως θα πρέπει στο πλαίσιο της κληρονομικότητας οι constructors της superclass να μην είναι private

Ο δείκτης super

- Ο super είναι μία αναφορά (ένας δείκτης) που ***δείχνει στο parent class (parent instance)*** και μπορεί να χρησιμοποιηθεί για να κληθούν οι μέθοδοι της parent class που δεν είναι private.
- Η κλήση έχει τη μορφή super.movePlus10() για παράδειγμα
- Μία ειδική μορφή της super είναι η μέθοδος ***super()*** που καλεί τον default constructor της parent class, ενώ η ***super(x)*** καλεί τον υπερφορτωμένο constructor της parent class (στο παράδειγμα της κλάσης Point)

Δημιουργοί της subclass (2)

- Όταν μέσα σε ένα constructor μίας υποκλάσης δεν ορίσουμε κλήση προς τον constructor της superclass, ***ο μεταγλωττιστής της Java εισάγει αυτόματα ως πρώτη εντολή μία κλήση προς τον default constructor της υπερκλάσης***
- Αν δεν υπάρχει τέτοιος constructor στην υπερκλάση, τότε δημιουργείται compile-time error

Κλάση Point2D default constructor


```
6       public Point2D() {
7           super();
8           y = 0;
9       }
```

```
6       public Point2D() {
7           super();
8       }
```

```
6       public Point2D() {
7           y = 0;
8       }
```

```
6       public Point2D() {
7
8       }
```

- Όλοι οι ορισμοί αριστερά του default constructor είναι ισοδύναμοι
- Αν δεν δοθούν οι κλήσεις super() και y = 0, παρέχονται αυτόματα από τον JDK
- Η τελευταία μορφή είναι η λιγότερο verbose και πιο συνηθισμένη, αρκεί να γνωρίζουμε τι σημαίνει

Κλάση Point2D overloaded constructor

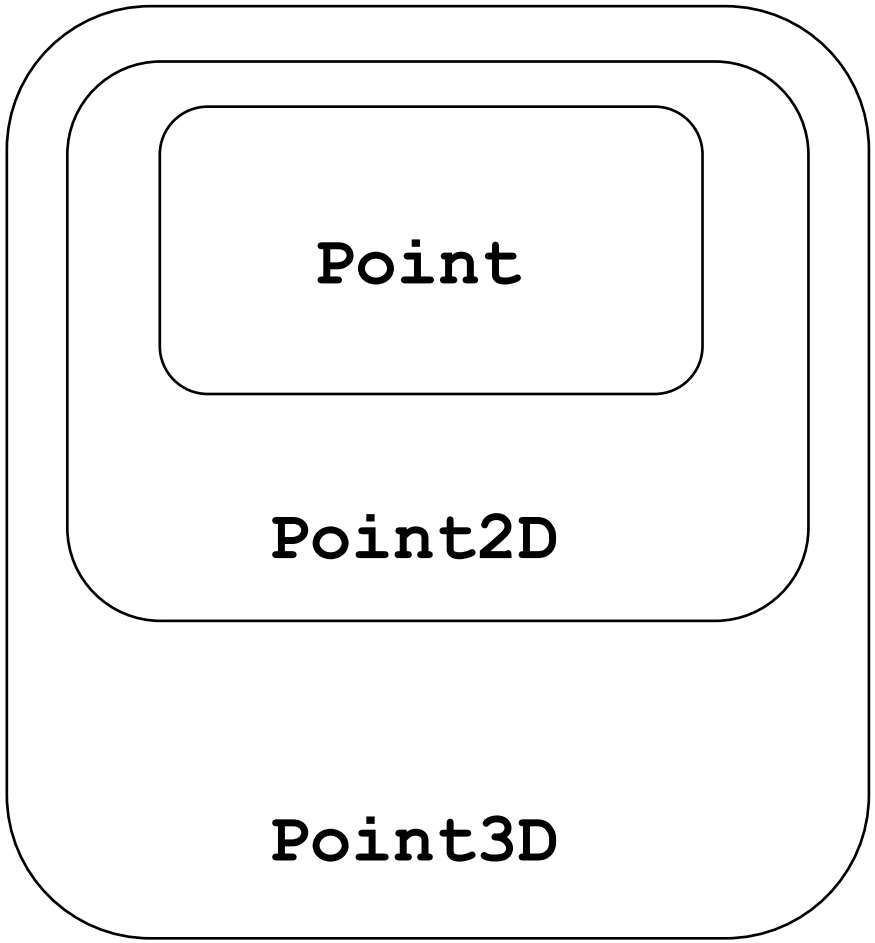

```
10      public Point2D(double x, double y) {
11          super(x);
12          this.y = y;
13      }
```

```
10      public Point2D(double x, double y) {
11          super();
12          setX(x);
13          this.y = y;
14      }
```

```
10      public Point2D(double x, double y) {
11          setX(x);
12          this.y = y;
13      }
```

- Το ίδιο ισχύει και για τον overloaded constructor
- Όλες οι κλήσεις αριστερά είναι ισοδύναμες
- Αν δεν δοθεί η κλήση super() ή super(x) καλείται αυτόματα ο default constructor της superclass, δηλαδή super()

Κλάση Point3D (1)



- Ορίζουμε στη συνέχεια την κλάση Point3D που κάνει extends την Point2D, άρα και έμμεσα και την κλάση Point

Κλάση Point3D (2)


```java
package testbed.ch14;

public class Point3D extends Point2D {
    private double z;

    public Point3D() {}

    public Point3D(double x, double y, double z) {
        super(x, y);
        this.z = z;
    }

    public double getZ() {
        return z;
    }

    public void setZ(double z) {
        this.z = z;
    }
```

- Ο default constructor της Point3D καλεί έμμεσα την super() δηλαδή την Point2D(), η οποία καλεί με τη σειρά της τη δική της super(), την Point() (**constructor chaining**)
- Επίσης, στον default constructor το z γίνεται 0.0 από το JVM

Κλάση Point3D (3)


```
21      @Override
22      public String convertToString() {
23          return "(" + this.getX() + ", " + this.getY()
24                  + ", " + this.getZ() + ")";
25      }
26
27      @Override
28      public void movePlus10() {
29          super.movePlus10();
30          z += 10;
31      }
32
33      @Override
34      protected void movePlusOne() {
35          super.movePlusOne();
36          z += 1;
37      }
```

- Οι μέθοδοι που έχουν κληρονομηθεί από την Point2D, αν θέλουμε και χρειάζεται, τις κάνουμε override, όπως εδώ

Χαρακτηρισμός visibility (1)

- Όπως είπαμε **τα private members μία κλάσης καθώς και οι constructors δεν κληρονομούνται**
- Ωστόσο, μπορούμε να κάνουμε instantiate την superclass με την super()
- Επίσης, μπορούμε να έχουμε πρόσβαση σε private πεδία, μέσω των setters και getters αν αυτοί παρέχονται ως public ή protected

Overridable – virtual methods


```
2 overrides
23  public String convertToString() {
24      return "(" + x + ")";
25  }
26
    2 overrides
27  public void movePlus10() {
28      //    x += 10;
29
30      // Self-use of movePlusOne
31      for (int i = 1; i <= 10; i++) {
32          movePlusOne();
33      }
34  }
35
    2 overrides
36  protected void movePlusOne() {
37      x++;
38  }
39
40  protected void printTypeOf() {
41      System.out.println(this.getClass().getSimpleName());
42  }
43
44  private void reset() { x = 0; }
```

- Η κλάση Point ορίζει μεθόδους public και protected οι οποίες κληρονομούνται
- Οι μέθοδοι αυτές είναι **overridable (ή αλλιώς virtual)**, δηλαδή μπορεί η κάθε υποκλάση να αλλάξει τη λειτουργικότητά τους
- Και οι package-private μέθοδοι (default access) μπορούν να κληρονομηθούν αλλά μόνο μέσα στο ίδιο package

Virtual μέθοδοι

- Όπως έχουμε πει η υπερκάλυψη (override) είναι η ιδιότητα με βάση την οποία σε μια ιεραρχία κληρονομικότητας μπορούμε να δηλώνουμε μεθόδους με ίδια υπογραφή αλλά διαφορετική λειτουργία
- Οι μέθοδοι που γίνονται override (overridable) ονομάζονται virtual
- **Overridable μέθοδοι σημαίνει: non-static και non-final, public ή protected μέθοδοι**
- Η υπερκάλυψη είναι από τα σημαντικότερα χαρακτηριστικά του αντικειμενοστραφούς προγραμματισμού

Επισήμανση @Override

- Το **annotation @Override** είναι καλό να το βάζουμε πριν τις μεθόδους που υπερκαλύπτουμε
- Δεν είναι υποχρεωτικό αλλά καλή πρακτική. Μας βοηθάει να μπορεί ο compiler να βρίσκει τυχόν λάθη που μπορεί να συμβούν όπως να κάνουμε λάθος στις παραμέτρους ή το όνομα της μεθόδου που κάνουμε override ή αν η μέθοδος αυτή διαγραφεί από την superclass πάλι ο compiler μας ειδοποιεί
- Επίσης, ελέγχει την τήρηση του Liskov substitution principle

Στατικές μέθοδοι - Hiding

- Οι ***static μέθοδοι μπορούν να κληρονομηθούν*** σύμφωνα πάντα με τα access modifiers που τις προσδιορίζουν (π.χ. private static δεν κληρονομούνται) αλλά δεν μπορούν να γίνουν override, παρά μόνο ***hide***
- Η λογική του hiding φαίνεται παρόμοια με το override, δηλαδή ίδια υπογραφή – διαφορετική λειτουργία, ωστόσο υπάρχει μία πολλή σημαντική διαφορά: ***οι static μέθοδοι δεν μπορούν να λειτουργήσουν στο πλαίσιο του Πολυμορφισμού***, όπως θα θα δούμε αργότερα, γιατί ελέγχονται @compile-time ενώ οι virtual μέθοδοι @runtime

Χαρακτηρισμός protected (1)

- Τον access modifier **protected**, όπως αναφέραμε τον χρησιμοποιούμε μόνο κατά την κληρονομικότητα. Τα protected πεδία και μέθοδοι κληρονομούνται από τις derived κλάσεις (όπως και τα public πεδία και μέθοδοι)
- Στα protected μέλη μίας κλάσης έχουν πρόσβαση και οι κλάσεις του package
- Για όλες τις άλλες κλάσεις που είτε δεν κληρονομούν ή δεν είναι στο ίδιο package, τα protected members θεωρούνται private

Χαρακτηρισμός protected (2)

- Στόχος της κληρονομικότητας είναι να δώσει στις derived κλάσεις, πρόσβαση, πέρα από το Public API της το οποίο ούτως ή άλλως είναι διαθέσιμο προς όλους και στην **εσωτερική λειτουργικότητα της (που διαφορετικά θα ήταν private ή package-private μεθόδους)** ή σε μέρος αυτής
- Διαφορετικά δεν έχει νόημα η κληρονομικότητα με την έννοια της επαναχρησιμοποίησης του κώδικα της superclass

Χαρακτηρισμός protected (3)

- Εφόσον ούτως ή άλλως όλες οι κλάσεις έχουν πρόσβαση στο Public API άλλων κλάσεων, **ο χαρακτηρισμός protected αφορά μεθόδους και πεδία που θα ήταν κανονικά private αλλά εφόσον πρόκειται να κληρονομηθούν, χαρακτηρίζονται ως protected**
- Η λογική, όπως αναφέραμε, είναι να δώσουμε ένα μέρος της λειτουργικότητας ή όλη τη λειτουργικότητα μίας κλάσης στις derived κλάσης

Superclass

- Επομένως, μία superclass για να είναι σωστά σχεδιασμένη για κληρονομικότητα θα πρέπει να **δίνει πρόσβαση (hooks) στην εσωτερική της λειτουργικότητα, δηλαδή σε protected μεθόδους** (private μέθοδοι που έχουν επιλεγεί και μετατραπεί σε protected ώστε να κληρονομηθούν) **και πολύ σπάνια σε protected πεδία**
- Επομένως **το API που κάνει export η superclass προς τις subclasses είναι ευρύτερο από το public API και περιλαμβάνει και τις protected μεθόδους**

Access Modifiers & implications

- Ότι είναι από package-private και κάτω (default access και private modifier) είναι μέρος του εσωτερικού implementation μίας κλάσης και μπορεί να αλλάξει χωρίς φόβο γιατί δεν επηρεάζει τους clients
- Ότι είναι από protected και πάνω (protected και public) είναι μέρος του exported API και πρέπει να υποστηρίζεται για πάντα (δεν αλλάζει) γιατί επηρεάζει τους clients

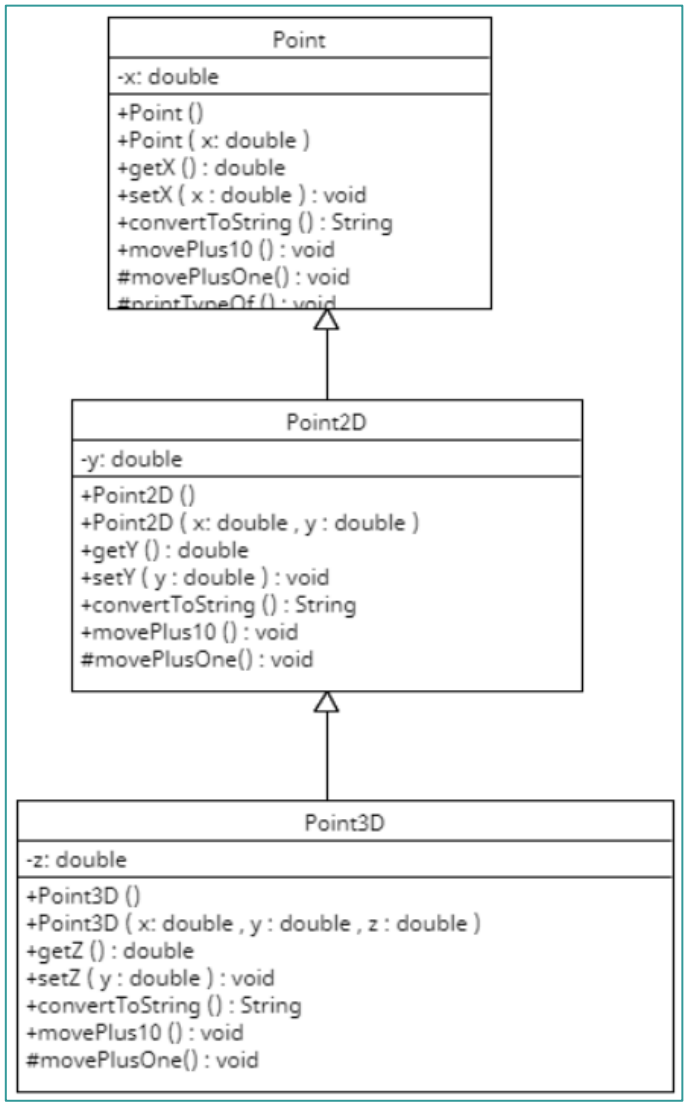
Protected μέλη

- Επομένως, ένα protected member μίας κλάσης είναι μέρος του API της κλάσης
- Αυτό σημαίνει ότι πρέπει να υποστηρίζεται για πάντα (δεν μπορεί να αλλάξει για επηρεάζει άλλες κλάσεις και clients που το χρησιμοποιούν)
- Από την άλλη πλευρά θα πρέπει να αντιληφθούμε ότι **τα protected members ενός exported class αποτελούν public commitments των implementation details μίας κλάσης κάτι που αντιβαίνει στην έννοια της ενθυλάκωσης**

Inheritance vs Encapsulation

- Στόχος της κληρονομικότητας είναι να δώσει πρόσβαση στη λειτουργικότητα της ή σε μέρος αυτής στις derived κλάσεις
- Κάτι τέτοιο όμως καταστρατηγεί την ιδέα της προστασίας της εσωτερικής υλοποίησης της κλάσης
- Επομένως: **inheritance violates encapsulation**
- Θα δούμε στα επόμενα κεφάλαια, πότε έχει νόημα να χρησιμοποιούμε κληρονομικότητα και πότε όχι

Κληρονομικότητα Point



- Μέχρι στιγμής ορίσαμε μία ιεραρχία κληρονομικότητας Point, Point2D, Point3D
- Η ***κληρονομικότητα παρέχει επιπλέον δυνατότητες***
- Μία σημαντική δυνατότητα που *παρέχεται λόγω της κληρονομικότητας και της υπερκάλυψης* ονομάζεται ***πολυμορφισμός***

Πολυμορφικές Μέθοδοι


```
15      /**
16       * Its is a polymorphic method. That is a method
17       * that may get many forms of input, not only Point
18       * but also Point2D & Point3D.
19       *<br>
20       * It is agnostic of the type of Point. It accepts
21       * any point in the inheritance hierarchy.
22       *
23       * @param point a Point instance or any type of instance
24       *              in the inheritance hierarchy of Point
25       */
26  @   public static void doMovePlus10(Point point) {
27          point.movePlus10();
28      }
```

- Θα δημιουργήσουμε μια γενική μέθοδο που να παίρνει ως παράμετρο ένα οποιοδήποτε αντικείμενο τύπου Point / Point2D / Point3D και να καλεί την κατάλληλη ***movePlus10()***

Πολυμορφισμός (1)

- Το γεγονός ότι η doMovePlus10(Point point) παίρνει ως παράμετρο ένα αντικείμενο τύπου Point σημαίνει ότι μπορούμε να περάσουμε ως πραγματική παράμετρο αντικείμενα όχι μόνο τύπου Point αλλά και οποιοδήποτε άλλο αντικείμενο στην ιεραρχία της κληρονομικότητας, όπως αντικείμενα τύπου *Point2D* και *Point3D* αφού **Point2D IS-A Point** και **Point3D IS-A Point**

Πολυμορφισμός (2)

- Επειδή η παράμετρος εισόδου point της doMovePlus10(Point point) μπορεί να πάρει ***πολλές μορφές*** ονομάζεται πολυμορφική παράμετρος
- Και αντίστοιχα η doShowPoint(Point p) ονομάζεται ***πολυμορφική μέθοδος***

Πολυμορφισμός (3)


```
5   public static void main(String[] args) {
6       Point p1 = new Point();
7       Point2D p2 = new Point2D();
8       Point3D p3 = new Point3D();
9
10      doMovePlus10(p1);
11      doMovePlus10(p2);
12      doMovePlus10(p3);
13  }
```

- Το γεγονός ότι η doMovePlus10(Point point) παίρνει ως παράμετρο ένα αντικείμενο τύπου Point σημαίνει ότι μπορούμε να περάσουμε ως πραγματική παράμετρο αντικείμενα όχι μόνο τύπου Point αλλά και οποιοδήποτε άλλο αντικείμενο στην ιεραρχία της κληρονομικότητας, όπως αντικείμενα τύπου Point2D και Point3D

Overriding & Πολυμορφισμός

- Μέσα στο σώμα της **doMovePlus10(Point point)** καλούμε την **point.movePlus10()** και ***καλείται εκείνη η movePlus10() ανάλογα τι αντικείμενο περνάει ως πραγματική παράμετρος στη θέση του point***
- Αν περάσουμε ***ως πραγματική παράμετρο*** στη θέση του point, ένα αντικείμενο τύπου **Point, όπως p1** θα κληθεί η μέθοδος **movePlus10()** της **Point**, αν καλέσουμε με αντικείμενο τύπου **Point2D, όπως p2** θα κληθεί η **movePlus10()** της **Point2D**, ενώ αν περάσουμε ως πραγματική παράμετρο αντικείμενο τύπου **Point3D, όπως p3** θα κληθεί η **movePlus10()** της **Point3D**
- Η ιδιότητα αυτή ονομάζεται καθυστερημένη δέσμευση (***late binding)*** γιατί η επιλογή της κατάλληλης **`movePlus10()`** γίνεται κατά το χρόνο εκτέλεσης του προγράμματος και όχι κατά το χρόνο μεταγλώττισης

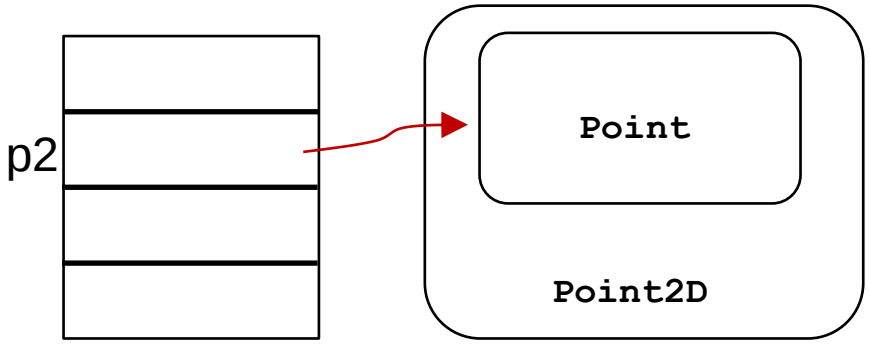
Πολυμορφισμός υποτύπων


```
1   package testbed.ch15;
2
3   public class Main {
4
5       public static void main(String[] args) {
6           Point p1 = new Point();
7           Point p2 = new Point2D();
8           Point p3 = new Point3D();
9
```

- Εφόσον τα Point2D και Point3D IS-A Point μπορούμε να κάνουμε δηλώσεις της παραπάνω μορφής, που είναι πολύ ευέλικτες γιατί **διαχωρίζουν τον τύπο του αντικειμένου που εδώ και στις τρεις περιπτώσεις είναι Point, από την υλοποίηση** που είναι Point, Point2D, Point3D αντίστοιχα

Τύπος Αναφοράς και πραγματικός τύπος


```
5   public static void main(String[] args) {
6       Point p1 = new Point();
7       Point p2 = new Point2D();
8       Point p3 = new Point3D();
```



- Στη γραμμή 7, ο τύπος αναφοράς είναι Point αλλά η υλοποίησης (το instance) είναι Point2D. Αυτό καταρχάς σημαίνει πως με την p2 μπορούμε να καλούμε όλες τις virtual/overridable μεθόδους της root κλάσης Point. Αυτό είναι πολύ σημαντικό γιατί προσδίδει ομοιομορφία στο API που μπορούν να καλούν τα p1, p2, p3 και γενικά οποιοδήποτε Point κληρονομεί από την Point
- Ωστόσο δεν μπορούν να κληθούν άμεσα από την p2 οι μέθοδοι του Point2D που έχουν οριστεί επιπλέον και δεν ανήκουν στην Point, όπως οι getY, setY. Αυτό μπορεί να γίνει μόνο ρητά με typecast (downcast), όπως: ((Point2D) p2).getY()

Downcasting


```
5   public static void main(String[] args) {
6       Point p1 = new Point();
7       Point p2 = new Point2D();
8       Point p3 = new Point3D();
9
10      doMovePlus10(p1);
11      doMovePlus10(p2);
12      doMovePlus10(p3);
13
14      ((Point2D) p2).setY(12);
15  }
```

- Όπως είπαμε ***η p2 ως δείκτης σε Point έχει πρόσβαση μόνο στις μεθόδους της Point*** και όχι στις επιπλέον μεθόδους των Point2D. Για να έχουμε πρόσβαση από την p2 στις μεθόδους που δεν έχουν κληρονομηθεί ή υπερκαλυφθεί, αλλά έχουν οριστεί στην Point2D, θα πρέπει να κάνουμε ρητό downcast (γρ. 14)

Static μέθοδοι και πολυμορφισμός

- Οι static μέθοδοι **δεν μπορούν να λειτουργήσουν σε πολυμορφικές μεθόδους** δεδομένου ότι οι static μέθοδοι δεν ανήκουν στα instances αλλά κατανέμονται από τον JVM κατά το χρόνο μεταγλώττισης

Απλή Κληρονομικότητα

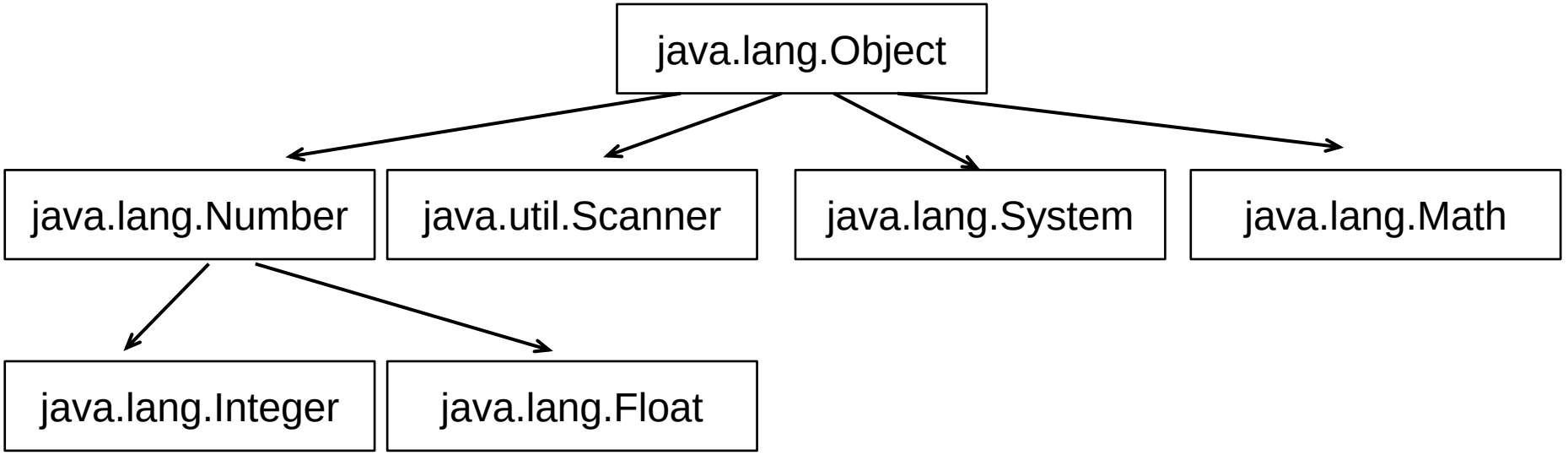
- Η Java **δεν επιτρέπει πολλαπλή κληρονομικότητα** παρά μόνο **απλή κληρονομικότητα** δηλαδή κάνουμε extend μόνο μία κλάση
- Κάτι τέτοιο είναι περιοριστικό όταν θέλουμε μία κλάση να αποτελέσει συνένωση δύο ή περισσοτέρων άλλων κλάσεων

Κλάση Object

- Η ίδια η Java οργανώνει τις κλάσεις της σε μια ιεραρχία κληρονομικότητας
- Στο πιο ψηλό σημείο, δηλαδή ***στη ρίζα (root) της ιεραρχίας βρίσκεται η κλάση Object*** από όπου κληρονομούν όλες οι κλάσεις της Java
- Όχι μόνο οι κλάσεις της Java, αλλά και όλες οι κλάσεις που γράφουμε εμείς έμμεσα κληρονομούν από την κλάση Object, δηλαδή ο μεταγλωττιστής της Java, προσθέτει αυτόματα (δεν φαίνεται) τη φράση ***`extends java.lang.Object`*** σε κάθε κλάση που ορίζουμε

Ιεραρχία κληρονομικότητας Java

- Ένα μέρος της ιεραρχίας κληρονομικότητας της Java απεικονίζεται παρακάτω



Μέθοδος toString() (1)

- Η ***toString()*** είναι μία μέθοδος της κλάσης **Object** την οποία κληρονομούν όλες οι κλάσεις στην Java, αφού όλες οι κλάσεις κληρονομούν από την Object
- Σκοπός της ***toString()*** είναι να επιστρέψει ένα **String** με τα πεδία του αντικειμένου (το state του αντικειμένου), ώστε να μπορούμε οπουδήποτε θέλουμε, να χειριστούμε ένα αντικείμενο ως String

Μέθοδος toString() (2)

- Για παράδειγμα, αν θέλουμε να εκτυπώσουμε με την **println()** ένα αντικείμενο (δηλαδή τα πεδία/state του αντικειμένου), τότε το αντικείμενο εκτυπώνεται κατά αυτό τον τρόπο που ορίζουμε εμείς στην ***toString()***
- Η ***Object.toString()*** δεν κάνει τίποτα, αλλά επιστρέφει απλά το όνομα της κλάσης από το αντικείμενο της οποίας καλείται και την διεύθυνση μνήμης του αντικειμένου
- Γιαυτό (αφού κληρονομούμε την toString() από την κλάση Object) μπορούμε να την υπερκαλύπτουμε (override) ώστε να επιστρέφει αυτό που ορίζουμε εμείς

Point και toString()

- Έτσι, ***η κλάση Point αντί να ορίζει την convertToString()*** για να εκτυπώνει τα πεδία των αντικειμένων της κλάσης Point και των υποκλάσεων Point2D και Point3D ***μπορεί να υπερκαλύπτει την toString() τόσο αυτή όσο και οι κλάσεις Point2D και Point3D***
- Έτσι η println(point) όταν καλεί την println(point) --θα καλεί την point.toString() και μάλιστα τη σωστή toString() που έχουμε κάνει override
- Γενικά σε οποιοδήποτε 'String context' η Java για τα instances ψάχνει μία toString() και με βάση αυτή – εφόσον έχει υπερκαλυφθεί- μετατρέπει το instance σε string

Κλάση Point


```
48      @Override
49      public String toString() {
50          return "(" + x + ")";
51      }
52  }
```

- Στην κλάση Point ***υπερκαλύπτουμε την toString()***
- Η toString() επιστρέφει ένα String με τα πεδία (state) της κλάσης

Κλάση Point2D


```
45      @Override
46      public String toString() {
47          return "(" + this.getX() + ", " + y + ")";
48      }
49  }
```

- Στην κλάση ***Point2D υπερκαλύπτουμε την toString()***
- Η toString() επιστρέφει ένα String με τα x και y πεδία της κλάσης

Κλάση Point3D


```
46      @Override
47      public String toString() {
48          return "(" + this.getX() + ", " + this.getY()
49                  + ", " + z + ")";
50      }
51  }
```

- Η κλάση ***Point3D*** είναι ίδια με την Point3D που είδαμε αλλά αντί της convertToString() ***υπερκαλύπτουμε την toString()***
- Η toString() επιστρέφει ένα String με τα x, y και z πεδία της κλάσης

Η μέθοδος println()

- Θυμίζουμε ότι η ***System.out.println()*** μετατρέπει σε String όλα τα ορίσματα που της περνάμε
- Αν για παράδειγμα καλέσουμε με ***println(i), όπου int i 10;*** η println(i) θα μετατρέψει αυτόματα το i σε String και θα το εκτυπώσει
- Το ίδιο κάνει και αν περάσουμε ως όρισμα ένα αντικείμενο, το μετατρέπει σε String και το εκτυπώνει. Το πώς όμως το μετατρέπει σε String και επομένως τι εκτυπώνει ***το ορίζουμε εμείς στην toString()***

Πολυμορφισμός (1)

- Μπορούμε να χρησιμοποιήσουμε μια πολυμορφική μέθοδο doPrint() για να εκτυπώνουμε οποιοδήποτε Point (οποιοδήποτε instance στην ιεραρχία της Point) γίνεται inject στην doPrint(PointX point)
- Για να δουλέψει σωστά θα πρέπει να έχουμε υπερκαλύψει την toString σε όλες τις κλάσεις τύπου Point

```
32      public static void doPrint(Point point) {
33          System.out.println(point);
34      }
```

Πολυμορφισμός (2)

- Η doPrint(Point point) είναι πολυμορφική μέθοδος όπου ως πραγματική παράμετρο μπορούμε να περάσουμε οποιοδήποτε αντικείμενο που ανήκει στην ιεραρχία κληρονομικότητας της Point

```
5    public static void main(String[] args) {
6        Point p1 = new Point();
7        Point p2 = new Point2D();
8        Point p3 = new Point3D();
9
10       doMovePlus10(p1);
11       doMovePlus10(p2);
12       doMovePlus10(p3);
13
14       doPrint(p1);
15       doPrint(p2);
16       doPrint(p3);
```

Συλλογή από Point

- Έστω ότι θέλουμε να αποθηκεύσουμε σε ένα πίνακα αντικείμενα τύπου Point (και όλων των υποκλάσεων)
- Τότε μπορούμε να ορίσουμε μια δομή πίνακα τύπου Point (βλ. επόμενη διαφάνεια)

Point Array


```
2
3   public class PointsDemo {
4
5       private final static Point[] points;
6       private static int counter1d = 0;
7       private static int counter2d = 0;
8       private static int counter3d = 0;
9
10      static {
11          points = new Point[] {
12                  new Point(1), new Point2D(2, 2), new Point3D(1, 2, 3),
13                  new Point3D(4, 5, 6)
14          };
15      }
```

- Ο πίνακας points αρχικοποιείται στο static block με διάφορα instances Point, Point2D, Point3D

instanceof

- Αν θέλουμε να ελέγξουμε τον τύπο του αντικειμένου σε μια ιεραρχία κληρονομικότητας μπορούμε να χρησιμοποιούμε την ***instanceof,*** *που χρησιμοποιείται ως* ***object instanceof type,*** όπου *object* είναι η αναφορά στο αντικείμενο και *type* ο τύπος της κλάσης ή υπερκλάσης, και επιστρέφει true αν το object IS-A type, αλλιώς false
- Αν στη θέση του type βάλουμε την κλάση/υπερκλάση και στη θέση του αντικειμένου, αναφορές σε αντικείμενα υποκλάσεων, τότε επιστρέφει true

Παράδειγμα Instanceof


```
17      public static void main(String[] args) {
18
19          for (Point point : points) {
20              point.printTypeOf();
21
22              if (point instanceof Point3D) {
23                  counter3d++;
24              } else if (point instanceof Point2D) {
25                  counter2d++;
26              } else {
27                  counter1d++;
28              }
29          }
30
31          System.out.println("Point instances: " + counter1d);
32          System.out.println("Point2D instances: " + counter2d);
33          System.out.println("Point3D instances: " + counter3d);
34      }
35  }
```

- Με την instanceof ελέγχουμε τον τύπο ενός αντικειμένου σε μια ιεραρχία κληρονομικότητας

Advanced Inheritance

- Η κληρονομικότητα είναι ένας δυνατός μηχανισμός για code reuse αλλά δεν είναι πάντα ο καλύτερος και ο πιο αποτελεσματικός μηχανισμός

Safe Inheritance

- Είναι σχετικά ασφαλές να χρησιμοποιούμε κληρονομικότητα **μέσα στο ίδιο package όταν το subclass και superclass implementation είναι κάτω από τον έλεγχο μας**
- Είναι επίσης safe να χρησιμοποιούμε κληρονομικότητα όταν κάνουμε extend **κλάσεις που έχουν σχεδιαστεί και τεκμηριωθεί για κάτι τέτοιο**
- **Δεν είναι όμως ασφαλές** να κάνουμε extends από μία οποιαδήποτε κλάση έξω από τα package boundaries

Inheritance Design

- Ο λόγος είναι πως μία subclass εξαρτάται από τα implementation details της superclass
- Αν στη διάρκεια ζωής μίας εφαρμογής η superclass αλλάξει υλοποίηση, η subclass μπορεί να γίνει break ακόμα κι αν ο κώδικάς της δεν έχει αλλάξει καθόλου
- Επομένως, αφού οι δύο κλάσεις πρέπει να εξελίσσονται παράλληλα, ο μόνος ασφαλής τρόπος είναι η σούπερ-κλάση να έχει σχεδιαστεί και τεκμηριωθεί κατάλληλα

Παράδειγμα


```
3   public class Point2D extends Point {
4       private double y;
5       private int moves = 0;
6
7       public Point2D() {
8       }
9       public Point2D(double x, double y) {...}
13      public double getY() { return y; }
16      public void setY(double y) { this.y = y; }
19
        1 override
20      @Override
21      public void movePlus10() {
22          super.movePlus10();
23          y += 10;
24
25      }
26
        1 override
27      @Override
28      protected void movePlusOne() {
29          super.movePlusOne();
30          y += 1;
31
32      }
```

- Αυτή η κλάση φαίνεται λογική, αλλά δεν δουλεύει σωστά
- Κάθε φορά που καλούμε την movePlus10() το y αυξάνεται κατά 20
- Αυτό γίνεται γιατί η movePlus10() της superclass έχει υλοποιηθεί on top of movePlusOne() και επομένως όταν καλείται καλεί με την σειρά της την overridden movePlusOne() της παρούσας κλάσης η οποία με τη σειρά της αυξάνει επίσης κατά ένα το y
- Επομένως καλείται 10 φορές η movePlusOne() αυξάνοντας το y κατά 10, ενώ στην συνέχεια η ίδια η movePlus10() αυξάνει κατά άλλες 10 μονάδες το y

Πιθανές λύσεις (1)

- Θα μπορούσαμε βέβαια να κάνουμε eliminate το override της movePlus10 και να μην κάνουμε move += 10 ή y += 10
- Παρότι η κλάση μας θα δούλευε, θα βάσιζε την ορθή λειτουργία της στο ότι η movePlus10() της superclass είναι υλοποιημένη on top of movePlusOne()
- Το 'self-use' είναι ένας τρόπος υλοποίησης που μπορεί να αλλάξει στο μέλλον στην superclass, επομένως η Point2D είναι πολύ εύθραυστη και μπορεί στο μέλλον να γίνει break

Πιθανές λύσεις (2)

- Θα μπορούσαμε επίσης να ξαναγράψουμε την υλοποίηση της movePlus10(), ώστε να αποφύγουμε την υλοποίηση της movePlus10() της superclass, κάτι το οποίο είναι δύσκολο και χρονοβόρο ενώ σε κάποιες περιπτώσεις θα απαιτούσε και πρόσβαση σε δομές δεδομένων της superclass που πιθανά θα ήταν private και inaccessible στην derived class

Overriding

- Όλα τα προβλήματα αυτής της μορφής πηγάζουν από το **overriding**
- Ο μόνος τρόπος να μην έχουμε αυτά τα προβλήματα είναι αντί να χρησιμοποιήσουμε κληρονομικότητα, να δημιουργήσουμε μία Point2D που να περιέχει ως private field ένα αντικείμενο Point και να καλούμε επιλεκτικά τις μεθόδους του (όπως είχαμε κάνει στο προηγούμενο κεφάλαιο)

Composition & Forwarding

- Αυτό είναι το μοντέλο **Composition and Forwarding**
- Στο Composition & Forwarding, οποιαδήποτε αλλαγή στην κλάση Point δεν επηρεάζει την κλάση Point2D

```
3   public class Point {
4       private Double x;
5
6       public Point() {
7
8       }
9
10      public Point(Double x) {
11          this.x = x;
12      }
```

```
3   public class Point2D {
4       private Point point;
5       private double y;
6
7       public Point2D() {
8
9       }
10
11      public Point2D(Point point, double y) {
12          this.point = point;
13          this.y = y;
14      }
```

Σχέση IS-A

- Η κληρονομικότητα είναι κατάλληλη μόνο αν η υποκλάση είναι πράγματι υποτύπος της superclass. **Αν υπάρχει δηλαδή σχέση "is-a"**
- Αν δεν υπάρχει τέτοια σχέση ή δεν είμαστε σίγουροι ότι υπάρχει δεν πρέπει να χρησιμοποιούμε κληρονομικότητα, αλλά composition

Composition & Forwarding

- Ακόμα όμως και αν υπάρχει σχέση 'is-a' πάλι θα υπάρχουν προβλήματα αν οι κλάσεις δεν είναι στο ίδιο package ή η superclass δεν είναι σχεδιασμένη για κληρονομικότητα
- Για να αποφύγουμε αυτά τα προβλήματα καλό είναι να χρησιμοποιούμε composition και forwarding αντί κληρονομικότητας

Overridable

- Τι σημαίνει όμως μία κλάση να είναι σχεδιασμένη για κληρονομικότητα
- Η κλάση πρέπει να τεκμηριώνει σε doc comments τις ΄self-use΄ overridable μεθόδους της (Για παράδειγμα η Point θα έπρεπε να τεκμηριώνει αναλυτικά την movePlus10)

@implSpec

- Η τεκμηρίωση self-use μεθόδων θα πρέπει να γίνεται με ένα ειδικό tag στα doc comments, το **@implSpec** που να περιγράφονται τα implementation requirements αναλυτικά και να αναφέρονται τα self-use invocations στο τέλος των implementation requirements

Βασικός κανόνας doc comments

- Μία τέτοια αναλυτική περιγραφή της υλοποίησης όμως σε doc comments παραβιάζει τον βασικό κανόνα για API documentation
- Ο βασικός κανόνας είναι να περιγράφουμε **ΤΙ κάνει η μέθοδος και όχι ΠΩΣ το κάνει** γιατί τότε εκθέτουμε την υλοποίησή της και παραβιάζουμε την αρχή του encapsulation

Inheritance breaks encapsulation

- Επομένως ο σωστός τρόπος σχεδιασμού της κληρονομικότητας παραβιάζει το encapsulation!

Χαρακτηρισμός Protected (1)

- Πέρα όμως από την **αποφυγή του self-use**, όπως έχουμε αναφέρει μία **superclass για να είναι σωστά σχεδιασμένη για κληρονομικότητα θα πρέπει να δίνει πρόσβαση (hooks) στην εσωτερική της λειτουργικότητα, δηλαδή σε protected μεθόδους** (private μέθοδοι που έχουν επιλεγεί και μετατραπεί σε protected ώστε να κληρονομηθούν) και **πολύ σπάνια σε protected πεδία**

Χαρακτηρισμός Protected (2)

- Οι protected μέθοδοι δεν έχουν ενδιαφέρον για τους clients (δεν είναι μέρος του public API) αλλά παρέχονται μόνο για ευκολία των προγραμματιστών στο κομμάτι του subclassing
- Οι protected μέθοδοι λοιπόν είναι μέρος του API για τις subclasses αλλά όχι μέρος του public API

Χαρακτηρισμός Protected (3)

- Όσο περισσότερες μεθόδους κάνουμε protected τόσο περισσότερο εκθέτουμε την υλοποίησή μας
- Όσο πιο λίγες, τόσο δυσκολεύουμε την κληρονομικότητα και το reuse
- Το πόσες μέθοδοι είναι κατάλληλες να γίνουν protected φαίνεται όταν πάμε να κάνουμε subclassing test
- Αν δυσκολευόμαστε, τότε χρειαζόμαστε περισσότερες protected, αν δε χρησιμοποιούμε τις protected, τότε τις μετατρέπουμε σε private
- Τρεις subclasses είναι αρκετές για test, όπου η μία να έχει γραφεί από κάποιον τρίτο και όχι τον συγγραφέα της superclass

Περιορισμοί Σχεδιασμού

- Άλλος περιορισμός στον σχεδιασμό της superclass είναι ότι οι Constructors δεν πρέπει να καλούν overridable methods
- Αυτό γιατί οι constructors της subclass πρώτα καλούνε τους constructors της superclass και επομένως αν κληθεί εκείνη τη στιγμή μία μέθοδος της subclass που έχει γίνει override θα αποτύχει γιατί δεν θα έχει δημιουργηθεί ακόμα το instance της subclass

Inheritance Use cases

- Καταλαβαίνουμε λοιπόν πως το να σχεδιάσουμε για κληρονομικότητα δεν είναι κάτι εύκολο και θέτει πολλούς περιορισμούς στην κλάση μας
- Υπάρχουν περιπτώσεις που **η κληρονομικότητα είναι η σωστή επιλογή όπως όταν έχουμε abstract κλάσεις οι οποίες υλοποιούν skeletal implementation και κληρονομούνται από concrete κλάσεις**

Inheritance και Immutability

- Υπάρχουν επίσης περιπτώσεις όπου η κληρονομικότητα είναι κακή επιλογή, όπως όταν έχουμε immutable κλάσεις
- Σε αυτή την περίπτωση αν κληρονομήσουμε από immutable κλάση και εισάγουμε στη συνέχεια non-final πεδίο/α στην derived κλάση ή δεν λάβουμε μέριμνα για immutability στην derived κλάση, τότε παρόλο που θα θεωρείται ότι και η derived κλάση είναι immutable αφού θα έχει μία σχέση 'is-a' με την immutable κλάση, δεν θα είναι στην πραγματικότητα immutable
- Οπότε για αυστηρό immutability, οι immutable κλάσεις θα πρέπει να γίνονται final ή να παρέχουν private ή package-private constructors ώστε να μην επιτρέπεται να κληρονομούνται

Κληρονομικότητα απλών κλάσεων

- Τι γίνεται όμως με τις κανονικές κλάσεις, που κατά κανόνα δεν είναι ούτε final, ούτε έχουν σχεδιαστεί προσεκτικά για κληρονομικότητα;
- Κάθε φορά κατά τη διάρκεια του κύκλου ζωής της superclass που θα γίνεται μία αλλαγή στην superclass, θα υπάρχει η πιθανότητα οι subclasses να γίνουν break

Non-Subclassing approaches (1)

- Καλύτερα λοιπόν να απαγορεύουμε το subclassing για κλάσεις που δεν έχουν σχεδιαστεί για κληρονομικότητα
- Αυτό μπορεί να γίνει με δύο τρόπους
  1. Να κάνουμε την κλάση **final**
  2. Να κάνουμε τους **constructors private ή package-private** και να παρέχουμε public static factories

Non-Subclassing approaches (2)

- Αυτή η προσέγγιση ίσως να φαίνεται περίεργη γιατί πολλοί προγραμματιστές έχουν ΄μεγαλώσει΄ και σπουδάσει και συνηθίσει να κληρονομούν απλές κλάσεις και να προσθέτουν λειτουργικότητα

Self-use

- Συμπερασματικά, αν θέλουμε μία κλάση να κληρονομείται πρέπει να είναι σωστά σχεδιασμένη και τουλάχιστον να αποφεύγει το self-use
- Αυτό μπορεί να γίνει τεχνικά αν μεταφέρουμε τον κώδικα των overridable methods σε private helper methods οι οποίες να καλούνται από τις overridable methods
- Και στη συνέχεια να αντικαταστήσουμε τα self-use invocations με invocation στις private helper methods

Use Cases

- Σε κάθε περίπτωση **κληρονομικότητα μπορούμε να εφαρμόζουμε σε τρεις περιπτώσεις**:
  - Αν μία superclass είναι σχεδιασμένη για κληρονομικότητα και υπάρχει σχέση 'is-a' και πάλι με επιφύλαξη γιατί το life-cycle της κλάσης μας θα εξαρτάται από το life-cycle της superclass
  - **Αν ελέγχουμε εμείς μέσα στο package μας όλη την ιεραρχία κληρονομικότητας** και υπάρχει σχέση 'is-a'
  - **Αν κληρονομούμε από abstract κλάσεις** στο πλαίσιο **skeletal implementations**

Άσκηση (1)

- Στο παράδειγμα του Point προσθέστε μία μέθοδο public double getDistanceFromOrigin() που να επιστρέφει την απόσταση του σημείου από την αρχή των αξόνων
- Για παράδειγμα αν ένα Point είναι το 5.5 τότε το distance είναι 5.5 – 0 = 5.5
- Υπερκαλύψτε (override) για Point2D και Point3D (βλ. επόμενες διαφάνειες)

Άσκηση (2)

- Για το Point2D το distance είναι η τετραγωνική ρίζα του αθροίσματος των τετραγώνων των διαφορών του x και y από την αρχή των αξόνων, όπως στο πυθαγόρειο θεώρημα των ορθογωνίων τριγώνων
- Για το Point3D είναι ακριβώς το ίδιο όπως παραπάνω (η τετραγωνική ρίζα του αθροίσματος των τετραγώνων των διαφορών), απλά προσθέτουμε στο άθροισμα και τη διαφορά της διάστασης z από την αρχή των αξόνων

Άσκηση (3)

- Στη μία Utility class με όνομα PointUtil αναπτύξτε μία πολυμορφική public static double μέθοδο distanceFromOrigin(Point point) που επιστρέφει την απόσταση από την αρχή των αξόνων (origin) οποιουδήποτε σημείου Point, Point2D, Point3D