ΕΛΛΗΝΙΚΗ ΔΗΜΟΚΡΑΤΙΑ

Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών

ΠΛΗΡΟΦΟΡΙΚΗ ΙΙ

Ενότητα 5: Κατασκευαστές (Constructors)

Μιχάλης Δρακόπουλος

Σχολή Θετικών επιστημών

Τμήμα Μαθηματικών

ΠΛΗΡΟΦΟΡΙΚΗ ΙΙ (Java)
Ενότητα 5

Κατασκευαστές (Constructors)

Ειδικός τύπος μεθόδων, οι οποίες:

- είναι `public` και έχουν το ίδιο όνομα με αυτό της κλάσης
- χρησιμοποιούνται για να αρχικοποιήσουν κάποιες μεταβλητές των αντικειμένων που δημιουργούν
- καλούνται αυτόματα όταν αρχικοποιούμε ένα αντικείμενο κάποιας κλάσης
- δεν επιστρέφουν κάποια τιμή (αλλά παρόλα αυτά *δεν* δηλώνονται σαν `void`)

Σε μία κλάση με το όνομα `ClassName` μπορούμε να έχουμε:

```
public class ClassName
{
   private int a;
   private String b;

     // default κατασκευαστής:
   public ClassName()
   {

   }

     // ένας άλλος κατασκευαστής:
   public ClassName(int x, String s)
   {
      a = x;
      b = s;
   }

   // άλλες δηλώσεις, μέθοδοι, κτλ.

}
```

- Κάθε κλάση περιέχει (αυτόματα από τη Java) τον default (προεπιλεγμένο) κατασκευαστή, παρόλο που δεν φαίνεται στον κώδικά της.

- Ο default κατασκευαστής παύει να δίνεται αυτόματα από τη στιγμή που δημιουργηθεί έστω και ένας κατασκευαστής από τον προγραμματιστή. Ο λόγος που γίνεται αυτό είναι ότι όταν ο προγραμματιστής δημιουργεί δικούς του κατασκευαστές που αρχικοποιούν κάποιες μεταβλητές, πιθανότατα να μην θέλει να δίνει το δικαίωμα κατασκευής αντικειμένων χωρίς την αρχικοποίηση αυτών των μεταβλητών. Εάν θέλουμε να μπορούμε να φτιάχνουμε αντικείμενα της κλάσης με τον απλό τρόπο (χωρίς παραμέτρους εισόδου για αυτόματη αρχικοποίηση μεταβλητών) θα πρέπει να προσθέσουμε στον κώδικα της κλάσης τον default κατασκευαστή, έστω και κενό.

Παράδειγμα κατασκευαστών στην κλάση `Circle`:

```
public class Circle
{
   private double x, y;  // συντεταγμένες κέντρου
   private double r;     // ακτίνα

      // μέθοδος για υπολογισμό εμβαδού κύκλου
   public double area()
   {
      return Math.PI*Math.pow(r,2);
   }

      // μέθοδος για υπολογισμό περιφέρειας κύκλου
   public double circumf()
   {
      return 2*Math.PI*r;
   }
}
```

Οι μεταβλητές `x`, `y` και `r` είναι `private`. Ένας τρόπος για να τους δώσουμε τιμές θα ήταν η ύπαρξη αντίστοιχων τροποποιητικών (modifier) μεθόδων στην κλάση `Circle`. Επειδή όμως **κάθε** αντικείμενο («κύκλος») που θα δημιουργούμε αναγκαστικά θα πρέπει να έχει τουλάχιστον κάποια δεδομένη ακτίνα, μπορούμε να αυτοματοποιήσουμε την απόδοση τιμών σε (κάποιες από) αυτές τις μεταβλητές, με τη δημιουργία κάποιων κατασκευαστών:

```
public Circle(double a, double b, double c)
{
   x = a;
   y = b;
   r = c;
}

public Circle(double r)
{
   x = 0;
   y = 0;
   this.r = r;     // το this. αναφέρεται στη μεταβλητή της κλάσης
                   // ενώ πλέον το r είναι η τοπική μετ/τή της μεθόδου
}

public Circle()
{
   x = 0;
   y = 0;
   r = 1;
}
```

[Οι κατασκευαστές αυτοί τοποθετούνται φυσικά μέσα στην κλάση Circle, μετά τις δηλώσεις των μεταβλητών και συνήθως πριν τις μεθόδους της κλάσης]

Ο πρώτος κατασκευαστής αρχικοποιεί όλες τις μεταβλητές, ο δεύτερος αρχικοποιεί μόνο την ακτίνα (θέτοντας συγχρόνως τη θέση του κύκλου στο σημείο (0,0)), ενώ ο τρίτος είναι παραλλαγή του default constructor και δημιουργεί «μοναδιαίους κύκλους» (αντικείμενα με r = 1).
Έτσι, μπορούμε να έχουμε τις εξής δηλώσεις για τη δημιουργία αντικειμένων της κλάσης `Circle` (μέσα σε κάποια άλλη κλάση φυσικά):

```
   Circle c1 = new Circle(2.5, 3.5, 6);
```

(δημιουργεί «κύκλο» με κέντρο στο (2.5,3.5) και ακτίνα 6)

```
Circle c2 = new Circle(4.3);
```

(δημιουργεί «κύκλο» με κέντρο στο (0,0) και ακτίνα 4.3)

```
Circle c3 = new Circle();
```

(δημιουργεί μοναδιαίο «κύκλο» στο (0,0))

~~`Circle c4 = new Circle(1.3, 4.5);`~~ → ΛΑΘΟΣ! Δεν έχουμε φτιάξει κατασκευαστή που να δέχεται **δύο** παραμέτρους.

Όταν σε μια κλάση υπάρχουν περισσότεροι του ενός κατασκευαστές, τότε παρατηρείται το φαινόμενο του **overloading κατασκευαστών** (υπερφόρτωση). Αυτό συμβαίνει όταν στην ίδια κλάση βρίσκονται μέθοδοι με το ίδιο όνομα (στην προκειμένη περίπτωση κατασκευαστές, που αναγκαστικά έχουν όλοι το ίδιο όνομα) με διαφορετικό πλήθος ή/και είδος παραμέτρων εισόδου. Το φαινόμενο της υπερφόρτωσης θα αναλυθεί αργότερα σε αυτή την ενότητα, στην *υπερφόρτωση μεθόδων*.

Η λέξη `static`:

Όλα τα στοιχεία μιας κλάσης ανήκουν στα αντικείμενά της. Δηλαδή, μια μεταβλητή `x` μιας κλάσης `MyClass`, έχει τόσα αντίγραφα (τόσες υποστάσεις) όσα (-ες) και τα αντικείμενα της κλάσης `MyClass`. Το ίδιο ισχύει και για τις μεθόδους. Δηλαδή, κάθε αντικείμενο έχει τις δικές του μεταβλητές και μεθόδους, όπως αυτές ορίζονται στην κλάση του (στο «καλούπι» του).

Για τον περιορισμό αυτής της «ποικιλομορφίας» του αντικειμενοστραφούς προγραμματισμού, υπάρχει η λέξη `static`. Εξαίρεση λοιπόν στα προηγούμενα αποτελούν οι μεταβλητές και μέθοδοι που δηλώνονται ως **`static`**. Αυτές είναι κοινές για όλα τα αντικείμενα μιας κλάσης και δεν χρειάζεται η δημιουργία αντικειμένου για τη χρήση τους.

→ Οι `static` μέθοδοι καλούνται με τη μορφή:

```
<Κλάση>.<μέθοδος>();
```

αντί της κανονικής μορφής `<αντικείμενο>.<μέθοδος>();`

→ Μέθοδοι που έχουν δηλωθεί `static` έχουν πρόσβαση μόνο σε `static` μεταβλητές και `static` μεθόδους της κλάσης τους.

Άρα, η `main` (που είναι `static`) διαχειρίζεται μόνο `static` μεταβλητές κλάσης (και φυσικά τις δικές της τοπικές μεταβλητές, δηλαδή αυτές που έχουν δηλωθεί μέσα στην ίδια τη `main`) και μπορεί να καλέσει άμεσα μόνο `static` μεθόδους της κλάσης εφαρμογής. Υπάρχει τρόπος να ξεπεραστεί αυτό το "πρόβλημα". Ας δούμε το ακόλουθο παράδειγμα, που αποτελεί τροποποίηση μιας άσκησης της Ενότητας 3 που κάνει χρήση του `switch`:

- Να γραφεί μέθοδος που να δέχεται τον αριθμό του μήνα (1-12) και να επιστρέφει το πλήθος των ημερών του μήνα αυτού, υποθέτοντας ότι το έτος δεν είναι δίσεκτο.

```
public class Ex2
{
   public int daysInMonth(int month)
   {
      switch (month)
      {
         case 4:
         case 6:
         case 9:
         case 11: return 30;
         case 2: return 28;
         default: return 31;
      }
   }  // end method
}   // end class
```

Μέχρι το σημείο αυτό έχουμε ολοκληρώσει τη μέθοδο που ζητάει άσκηση. Αν θέλαμε να ολοκληρώσουμε το πρόγραμμα ώστε να μπορεί να εκτελεστεί, θα πρέπει να προσθέσουμε τη μέθοδο `main` στην κλάση μας (`Ex2`), ως εξής (πριν ή μετά τη μέθοδο `daysInMonth`, δεν έχει σημασία):

```
   public static void main(String [] args)
   {
      Scanner input = new Scanner(System.in);
      Ex2 obj = new Ex2();
      System.out.println("Δώσε τον αριθμό του μήνα (1-12)");
      int m = input.nextInt();
         // κλήση της μεθόδου daysInMonth για υπολογισμό ημερών
      int d = obj.daysInMonth(m);
      System.out.println("Ο μήνας έχει " + d + " ημέρες.");
   }
```

και φυσικά δεν πρέπει να ξεχάσουμε να κάνουμε `import java.util.*;` στην κλάση μας (`Ex2`), αφού πλέον χρησιμοποιούμε τη `Scanner` για είσοδο από το πληκτρολόγιο.

ΣΗΜΑΝΤΙΚΟ: Σε αυτή τη μέθοδο `main` βλέπουμε ότι δημιουργούμε αντικείμενο της ίδιας της κλάσης μέσα στην οποία βρισκόμαστε (`Ex2`) και μέσω αυτού καλούμε τη μέθοδο `daysInMonth`. Ο λόγος είναι ότι ενώ η `main` είναι `static` μέθοδος, η `daysInMonth` δεν είναι `static`. Οι `static` μέθοδοι έχουν άμεση πρόσβαση μόνο σε `static` μεταβλητές και μεθόδους. Για να μπορέσει η `main` να καλέσει μια μη-`static` μέθοδο, πρέπει να την καλέσει μέσω κάποιου συγκεκριμένου αντικειμένου. Όταν καλεί μεθόδους άλλων κλάσεων (υποστήριξης), αυτό είναι αυτονόητο. Όταν όμως καλεί μη-`static` μεθόδους της κλάσης στην οποία βρίσκεται, δεν είναι προφανές ότι πρέπει να δημιουργήσει αντικείμενο της ίδιας της κλάσης της, γι αυτό και επισημαίνεται εδώ.

Μέθοδοι που δέχονται ή/και επιστρέφουν αντικείμενα

Τα αντικείμενα είναι σύνθετοι τύποι μεταβλητών, οπότε οι μέθοδοι μπορούν να τα διαχειριστούν (δηλαδή, να τα δεχθούν ή να τα επιστρέψουν) όπως και τις μεταβλητές. Όσον αφορά μάλιστα την επιστροφή μιας μεθόδου, με τη χρήση επιστροφής αντικειμένου (αντί για επιστροφή μιας απλής μεταβλητής) μια μέθοδος μπορεί ουσιαστικά να επιστρέφει περισσότερες από μία μεταβλητές, επιστρέφοντας στην ουσία όλες τις "μεταβλητές κλάσης" που μπορεί να περιέχει το συγκεκριμένο αντικείμενο που επιστρέφει. Ας δούμε το ακόλουθο παράδειγμα:

```
public class ClassA
{
  private int x, y;

   // κατασκευαστές
  public ClassA(int x, int y)
  {
    this.x = x;
    this.y = y;
  }

  public ClassA()
  { }

   // μέθοδος απόδοσης τιμών στις μεταβλητές κλάσης
  public void setValues(int a, int b)
  {
    x = a;
    y = b;
  }

    // μέθοδοι πρόσβασης για τις δύο private μεταβλητές της κλάσης
  public int getX()
  {
    return x;
  }

  public int getY()
  {
    return y;
  }

    // μέθοδος που αυξάνει τις τιμές των x και y κατά 1
  public ClassA increaseXY(ClassA a)
  {
    ClassA ca = new ClassA(a.getX()+1, a.getY()+1);
    return ca;
  }
}
```

Στο παράδειγμα αυτό, η μέθοδος `increaseXY` της κλάσης `ClassA` δέχεται μια "μεταβλητή" τύπου `ClassA`, δηλαδή ένα αντικείμενο της κλάσης `ClassA` και επιστρέφει αντικείμενο της κλάσης `ClassA`. Αυτό που κάνει είναι να δημιουργεί και να επιστρέφει ένα νέο αντικείμενο της `ClassA` με τιμές των `x` και `y` αυτού του νέου αντικειμένου τις τιμές που έχουν οι μεταβλητές για το αντικείμενο `a` που δέχεται η μέθοδος αυξημένες κατά 1. Άρα, ουσιαστικά η συγκεκριμένη μέθοδος επιστρέφει *δύο* μεταβλητές και όχι μία, αφού κάθε αντικείμενο της κλάσης `ClassA` περιέχει δύο μεταβλητές.
Η δημιουργία του νέου αντικειμένου `ca` μέσα στη μέθοδο `increaseXY` αντί για:

```
ClassA ca = new ClassA(a.getX()+1, a.getY()+1);
```

θα μπορούσε να γραφεί και ως εξής:

```
ClassA ca = new ClassA(a.x+1, a.y+1);
```

Παρόλο που οι μεταβλητές `x` και `y` είναι `private`, αφού βρισκόμαστε μέσα στην κλάση `ClassA`, μπορούμε να έχουμε πρόσβαση στις τιμές τους απλά γράφοντας `<αντικείμενο>.<μεταβλητή>`, κάτι το οποίο δεν το χρησιμοποιούμε σχεδόν ποτέ, αφού πάντα ορίζουμε τις μεταβλητές κλάσης

`private`, οπότε ο τρόπος αυτός πρόσβασης σε μια μεταβλητή ενός αντικειμένου *από κάποια άλλη κλάση* δεν είναι δυνατός.

ΠΡΟΣΟΧΗ: Εάν η δημιουργία αντικειμένου μέσα στην `increaseXY` γινόταν ως εξής:

```
ClassA ca = new ClassA(x+1, y+1);
```

η μέθοδος δεν θα έκανε αυτό το οποίο θα θέλαμε να κάνει. Αυτό που θα έκανε θα ήταν να αυξάνει την τιμή των `x` και `y` του αντικειμένου της `ClassA` *με το οποίο θα την καλούσαμε* και όχι του συγκεκριμένου αντικειμένου που δέχεται η μέθοδος (`a`).

Σε μία άλλη κλάση, η οποία χρησιμοποιεί την `ClassA`, θα μπορούσαν να υπάρχουν τα εξής:

```
public class Program
{
   public static void main (String[] args)
   {
      ClassA obj1 = new ClassA(5,10);
      System.out.println("Τιμές των x και y για το obj1:");
      System.out.println(obj1.getX());
      System.out.println(obj1.getY());

      ClassA obj2 = new ClassA(15,20);
      ClassA obj3 = obj1.increaseXY(obj2);

      System.out.println("Τιμές των x και y για το obj3:");
      System.out.println(obj3.getX());
      System.out.println(obj3.getY());
   }
}
```

Το πρόγραμμα θα εκτύπωνε τα εξής:

```
Τιμές των x και y για το obj1:
5
10
Τιμές των x και y για το obj3:
16
21
```

Οι τιμές 16 και 21 προκύπτουν από την αύξηση κατά ένα των τιμών των μεταβλητών του αντικειμένου `obj2`, που επιτυγχάνει η κλήση της μεθόδου `increaseXY` στην οποία αποστέλλεται (περνιέται) το αντικείμενο `obj2`. Προφανώς στο συγκεκριμένο παράδειγμα δεν έχει σημασία το αντικείμενο το οποίο χρησιμοποιείται για να γίνει η κλήση της μεθόδου (δηλαδή το `obj1`). Αν όμως η μέθοδος `increaseXY` περιείχε την εντολή που αναφέρθηκε προηγουμένως:

```
ClassA ca = new ClassA(x+1, y+1);
```

τότε θα συνέβαινε το ακριβώς αντίθετο, δηλαδή η αύξηση κατά ένα θα γινόταν στις μεταβλητές του αντικειμένου με το οποίο θα καλούνταν η μέθοδος (δηλαδή το `obj1`) και όχι στου αντικειμένου που αποστέλλεται σαν όρισμα στη μέθοδο (`obj2`). Άρα το πρόγραμμα θα εμφάνιζε το "λανθασμένο" αποτέλεσμα:

```
Τιμές των x και y για το obj1:
5
10
Τιμές των x και y για το obj3:
6
11
```

Άρα, τελικά η συγκεκριμένη μέθοδος ίσως να ήταν προτιμότερο να είχε δηλωθεί σαν `static` ώστε να καλείται γενικά και όχι για συγκεκριμένο αντικείμενο.

Παράδειγμα με την κλάση Complex

Στην προηγούμενη ενότητα είχαμε ξεκινήσει τη δημιουργία ενός "σύνθετου τύπου" μεταβλητών (δηλαδή μιας κλάσης) για τους μιγαδικούς αριθμούς (`Complex`). Το περιεχόμενο της κλάσης `Complex` ήταν το εξής:

```
public class Complex
{

  private double real, imag; // το πραγματικό και το φανταστικό μέρος

  public void setValues(double x, double y)
  {
    real = x;
    imag = y;
  }

  public double getReal()
  {
    return real;
  }

  public double getImaginary()
  {
    return imag;
  }

  public String printComplex()
  {
    if (imag>0)
      return (real + " + " + imag + "i");
    else if (imag<0)
      return (real + " - " + (-imag) + "i");
    else
     return (real + ""); //προσθέτουμε κενό String για να μετατραπεί όλο σε String
  }
}
```

→ Κάποιες βελτιώσεις και επεκτάσεις που μπορούν να γίνουν σε αυτή την κλάση είναι οι εξής:

1. Η διαδικασία απόδοσης τιμής στο πραγματικό και φανταστικό μέρος ενός μιγαδικού (μεταβλητές `real` και `imag` αντίστοιχα) μπορεί να αυτοματοποιηθεί μέσω ενός κατάλληλου κατασκευαστή (οπότε η ύπαρξη της μεθόδου `setValues` δεν θα είναι πλέον απαραίτητη):

```
  // Constructor
public Complex(double x, double y)
{
   real = x;
   imag = y;
}
```

i) Σε ποια περίπτωση θα παρέμενε απαραίτητη η ύπαρξη της μεθόδου `setValues`; → Στην περίπτωση που εκτός από τον παραπάνω κατασκευαστή, προσθέταμε και τον default constructor, οπότε κάποιος θα μπορούσε να φτιάξει αντικείμενο με τον κλασικό τρόπο χωρίς να στείλει κατευθείαν τιμές στις μεταβλητές.

ii) Σε ποια άλλη περίπτωση θα εξακολουθούσε να είναι απαραίτητη η ύπαρξη της μεθόδου `setValues`; → Στην περίπτωση που θα θέλαμε να δώσουμε τη δυνατότητα δημιουργίας μεθόδων που δέχονται αντικείμενο τύπου `Complex` και μεταβάλουν τις τιμές των μεταβλητών του αντικειμένου αυτού. Μια τέτοια μέθοδος θα δεχόταν ένα ήδη δημιουργημένο αντικείμενο, οπότε θα χρειαζόταν την ύπαρξη μιας τέτοιας μεθόδου για να μεταβάλει τα δεδομένα του αντικειμένου αυτού. (Το συγκεκριμένο παράδειγμα θα χρησιμοποιηθεί στην επόμενη ενότητα, που αναφέρεται στο "pass-by-value").

2. Μπορούν να προστεθούν μέθοδοι που επιτελούν πράξεις μεταξύ μιγαδικών αριθμών. Π.χ., για την πρόσθεση μιγαδικών θα μπορούσαν να ορισθούν οι ακόλουθες πιθανές εκδοχές μιας μεθόδου `add`:

```
  public Complex add(Complex a)
  {
     return new Complex(real+a.getReal(), imag+a.getImaginary());
       // αντί για real και imag: this.real και this.imag
  }

  // Η add(Complex a) θα μπορούσε να είναι και ως εξής, κάνοντας
  // χρήση της add(Complex a, Complex b) που ακολουθεί:
  //
  // public Complex add(Complex a)
  // {
  //   return add(this, a); // Το this αναφέρεται στο  αντικείμενο
  //                        // με το οποίο καλείται η μέθοδος
  // }


public static Complex add(Complex a, Complex b)
{
  return new Complex(a.getReal()+b.getReal(),a.getImaginary()+b.getImaginary());
}
```

Η πρώτη μέθοδος (`public Complex add(Complex a)`) δέχεται ένα αντικείμενο τύπου `Complex` και δημιουργεί ένα νέο αντικείμενο της `Complex` (το οποίο και επιστρέφει) με τιμές των μεταβλητών του τα αθροίσματα των τιμών του αντικειμένου που δέχεται (`a`) με τις τιμές του αντικειμένου με το οποίο καλεί κάποιος τη μέθοδο (μεταβλητές `real` και `imag`). [Παράδειγμα κλήσης ακολουθεί παρακάτω]

Η δεύτερη μέθοδος (`public static Complex add(Complex a, Complex b)`) ακολουθεί διαφορετική προσέγγιση στην πρόσθεση δύο αντικειμένων. Δέχεται και τα δύο αντικείμενα σαν παραμέτρους εισόδου και επιστρέφει νέο αντικείμενο με τιμές μεταβλητών τα αθροίσματα των τιμών των δύο αντικειμένων που δέχεται. Σε αυτή την περίπτωση, δεν έχει νόημα η μέθοδος να καλείται για κάποιο συγκεκριμένο αντικείμενο (αφού και τα δύο αντικείμενα που την ενδιαφέρουν τα δέχεται σαν ορίσματα), γι αυτό και δηλώνεται σαν `static`, ώστε να καλείται γενικά.

Η δεύτερη εκδοχή της πρώτης μεθόδου (που φαίνεται σε σχόλιο στον παραπάνω κώδικα), στην ουσία χρησιμοποιεί τη δεύτερη μέθοδο (με τις δύο παραμέτρους εισόδου), στην οποία στέλνει σαν πρώτο όρισμα το αντικείμενο με το οποίο κάποιος την καλεί (αυτό αντιπροσωπεύει η λέξη `this`) και σαν δεύτερο όρισμα το δικό της όρισμα (`a`).

Με αντίστοιχο τρόπο θα μπορούσαν να προστεθούν στην κλάση `Complex` και άλλες μέθοδοι, για τις υπόλοιπες πράξεις μιγαδικών αριθμών.

Η κλάση εφαρμογής ενός προγράμματος που χρησιμοποιεί τη βελτιωμένη έκδοση της κλάσης `Complex` θα μπορούσε να είναι η ακόλουθη:

```
import java.util.*;

public class TestComplex
{
  public static void main(String [] args)
  {
    Scanner input = new Scanner(System.in);
    System.out.print("Δώσε το πραγματικό και το φανταστικό μέρος του πρώτου μιγαδικού:
");
    double r1 = input.nextDouble();
    double i1 = input.nextDouble();
    System.out.print("Δώσε το πραγματικό και το φανταστικό μέρος του δεύτερου μιγαδικού:
");
    double r2 = input.nextDouble();
    double i2 = input.nextDouble();

    Complex c1 = new Complex(r1,i1);
    Complex c2 = new Complex(r2,i2);

    System.out.print("Ο πρώτος μιγαδικός αριθμός είναι ο: ");
    System.out.println(c1.printComplex());
    System.out.print("Ο δεύτερος μιγαδικός αριθμός είναι ο: ");
    System.out.println(c2.printComplex());

    Complex c3 = c1.add(c2);
    // ή εναλλακτικά, με κλήση της static εκδοχής της μεθόδου add:
    // Complex c3 = Complex.add(c1,c2);
    System.out.print("Το άθροισμα των δύο μιγαδικών ισούται με: ");
    System.out.println(c3.printComplex());
  }
}
```

Ένα παράδειγμα εκτέλεσης του προγράμματος:

```
Δώσε το πραγματικό και το φανταστικό μέρος του πρώτου μιγαδικού: 3 7
Δώσε το πραγματικό και το φανταστικό μέρος του δεύτερου μιγαδικού: 2 -1
Ο πρώτος μιγαδικός αριθμός είναι ο: 3.0 + 7.0 i
Ο δεύτερος μιγαδικός αριθμός είναι ο: 2.0 - 1.0 i
Το άθροισμα των δύο μιγαδικών ισούται με: 5.0 + 6.0 i
```

Overloading μεθόδων (υπερφόρτωση)

Στο προηγούμενο παράδειγμα της κλάσης `Complex`, στη μέθοδο `add`, παρατηρούμε το φαινόμενο της *υπερφόρτωσης μεθόδων*, σε αντιστοιχία με την υπερφόρτωση κατασκευαστών που αναφέρθηκε νωρίτερα. Όταν σε μία κλάση υπάρχουν μέθοδοι *με το ίδιο όνομα* αλλά με διαφορετικό πλήθος παραμέτρων εισόδου ή με διαφορετικούς τύπους παραμέτρων εισόδου, παρατηρείται το φαινόμενο της υπερφόρτωσης μεθόδου. Ο τύπος της μεθόδου δεν είναι από μόνος του ικανό στοιχείο διαφοροποίησης. Π.χ., στο παρακάτω παράδειγμα παρατηρείται υπερφόρτωση της μεθόδου `myMethod`:

```
public class MyClass
{
  public int myMethod(int x)
  {
     return 2*x;
  }

  public int myMethod(int x, int y)
  {
     return x*y;
  }
}
```

ενώ το παρακάτω παράδειγμα είναι λάθος:

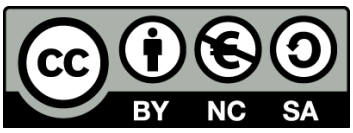
```
public class MyClass
{
  public int myMethod(int x)
  {
     return 2*x;
  }

  public double myMethod(int x)
  {
     return Math.sqrt(x);
  }
}
```

Σημειώματα

Σημείωμα Αναφοράς

Μιχάλης Δρακόπουλος. «Πληροφορική ΙΙ. Ενότητα 5: Κατασκευαστές (Constructors)». Έκδοση: 1.0. Αθήνα 2014. Διαθέσιμο από τη δικτυακή διεύθυνση: http://opencourses.uoa.gr/courses/MATH106/.

Σημείωμα Αδειοδότησης



[1] http://creativecommons.org/licenses/by-nc-sa/4.0/

Ως **Μη Εμπορική** ορίζεται η χρήση:

- που δεν περιλαμβάνει άμεσο ή έμμεσο οικονομικό όφελος από την χρήση του έργου, για το διανομέα του έργου και αδειοδόχο
- που δεν περιλαμβάνει οικονομική συναλλαγή ως προϋπόθεση για τη χρήση ή πρόσβαση στο έργο
- που δεν προσπορίζει στο διανομέα του έργου και αδειοδόχο έμμεσο οικονομικό όφελος (π.χ. διαφημίσεις) από την προβολή του έργου σε διαδικτυακό τόπο

Ο δικαιούχος μπορεί να παρέχει στον αδειοδόχο ξεχωριστή άδεια να χρησιμοποιεί το έργο για εμπορική χρήση, εφόσον αυτό του ζητηθεί.

Διατήρηση Σημειωμάτων

- Οποιαδήποτε αναπαραγωγή ή διασκευή του υλικού θα πρέπει να συμπεριλαμβάνει:
- το Σημείωμα Αναφοράς
- το Σημείωμα Αδειοδότησης
- τη δήλωση Διατήρησης Σημειωμάτων
- το Σημείωμα Χρήσης Έργων Τρίτων (εφόσον υπάρχει)

μαζί με τους συνοδευόμενους υπερσυνδέσμους.

Χρηματοδότηση

- Το παρόν εκπαιδευτικό υλικό έχει αναπτυχθεί στο πλαίσιο του εκπαιδευτικού έργου του διδάσκοντα.
- Το έργο «**Ανοικτά Ακαδημαϊκά Μαθήματα στο Πανεπιστήμιο Αθηνών**» έχει χρηματοδοτήσει μόνο τη αναδιαμόρφωση του εκπαιδευτικού υλικού.
- Το έργο υλοποιείται στο πλαίσιο του Επιχειρησιακού Προγράμματος «Εκπαίδευση και Δια Βίου Μάθηση» και συγχρηματοδοτείται από την Ευρωπαϊκή Ένωση (Ευρωπαϊκό Κοινωνικό Ταμείο) και από εθνικούς πόρους.