**The Development Cycle**

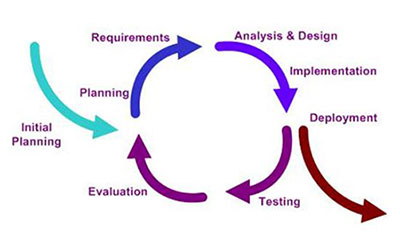
Linus Torvalds, the inventor of the Linux Operating System, said this in an essay called “ [Just for Fun: The Story of an Accidental Revolutionary](https://l.flvsgl.com/GSL03b9af71d41bec13b2adb502c6873f7fc).”

A great mathematician doesn't solve a problem the long and boring way because he sees what the real pattern is behind the question, and applies that pattern to find the answer in a much better way. The same is definitely true in computer science, too. Sure, you can just write a program that calculates the sum. On today's computers that would be a snap. But a great programmer would know what the answer is simply by being clever. He would know to write a beautiful program that attacks the problem in a new way that, in the end, is the right way. It's still hard to explain what can be so fascinating about beating your head against the wall for three days, not knowing how to solve something the better way, the beautiful way. But once you find that way, it's the greatest feeling in the world.

Linus Torvald

Writing a program and learning to program follow similar development cycles: incremental and iterative. New material is introduced in small amounts, so you have the opportunity to practice each new concept and skill, and integrate them with what you have already learned. And, to improve your programming ability, it is necessary to write more and more programs of increasing difficulty. However, too much information at one time or presented too rapidly disrupts the growth process necessary for productive learning. Would you water a house plant with a fire hose? Too much water too quickly will not nurture the plant; better to water the plant in small amounts at regular intervals. Students, like plants, can only absorb so much at any one time.

Every programming project begins with planning and understanding the requirements. This is followed by a closer analysis of the details of the problem and design of a solution. The prototype design is implemented, deployed, and tested to determine if you are on the right track, but it rarely represents the completed program. Further testing helps eliminate bugs and evaluation validates the design; then the process starts again.

© Creative Commons image CC-BY-SA

Following this development cycle allows you to build the program incrementally, making small changes each time through the loop until *finally* you have a bug-free, fully-functioning program. At least that is the way it is supposed to work!

Let's dive deeper into class definitions with multiple constructors and overloaded methods. Practice within the eIMACS interactive exercises to discover and test various scenarios. This practice will strengthen your programming muscle as you expand your skills.

We mentioned [earlier](https://www.eimacs.com/eimacs/mainpage?epid=E1932277324&cid=162149#MultiConstructors) that Java allows a class to have more than one constructor provided that all the constructors have the same name as the class and no two of them have the same signature. Recall from [Lab 2](https://www.eimacs.com/eimacs/mainpage?epid=E2011026054&cid=162149#Task02-2) that the original definition of the APRectangle class that you produced had the following form:

public class APRectangle   
{   
  private APPoint myTopLeft;   
  private double myWidth;   
  private double myHeight;   
  
  public APRectangle( APPoint topLeft, double width, double height )   
  {   
    /\* Initialization code \*/   
  }   
  
  /\*   
     Accessor methods getTopLeft, getWidth, getHeight and   
     modifier methods setTopLeft, setWidth, setHeight   
   \*/   
}

None of the later work in that lab altered the fact that the APRectangle class had just one constructor. But there are other ways to describe a rectangle whose sides are parallel to the coordinate axes. One of the most common is to specify a pair of diagonally-opposite corners. It is true that, given a pair of diagonally-opposite corners, we could calculate the height and the width and then feed the results to the constructor above. Why should we have to do that, however, when the computer is perfectly capable of performing the calculations for itself? Consequently, Java allows us to add a second constructor to the APRectangle class, as follows:

public class APRectangle   
{   
  private APPoint myTopLeft;   
  private double myWidth;   
  private double myHeight;   
  
  public APRectangle( APPoint topLeft, double width, double height )   
  {   
    myTopLeft = topLeft;   
    myWidth = width;   
    myHeight = height;   
  }   
  
  public APRectangle( APPoint topLeft, APPoint bottomRight )   
  {   
    myTopLeft = topLeft;   
    myWidth = bottomRight.getX() - topLeft.getX();   
    myHeight = bottomRight.getY() - topLeft.getY();   
  }   
  
  /\* Various methods \*/   
}

The class knows which constructor to use when any particular instance is being created by matching the arguments provided to the constructor against the two distinctive constructor signatures. (The act of creating an instance of a class is called *instantiation*. By creating the instance we are said to *instantiate* the class. Furthermore, the object created is also said to *instantiate* the class.)

[Earlier](https://www.eimacs.com/eimacs/mainpage?epid=E1991301432&cid=162149#DefaultCons), when we told you a little bit about how the Java compiler works, we mentioned that if you forget to include (or if you deliberately omit) a class constructor in a class definition, then Java operates as if you had specified a constructor that takes no arguments and that has no body. It turns out that it is sometimes useful deliberately to include a no-argument constructor in a class definition. Unlike the no-argument, no-body constructor supplied by Java, however, a deliberately included no-argument constructor typically has a body that assigns sensible default values to the instance variables. Any such no-argument constructor — whether the no-body version automatically supplied by Java or a default-body version supplied by the programmer — is known as a *default constructor*.

One of the reasons why you might want to include a default constructor that assigns default values to the instance variables is that you might want to instantiate a class before the specific characteristics of the object to be created are known. Your default constructor will then create a default instance whose characteristics can be set later using the object's modifier instance methods. In the code below, for example, we have added a default constructor to the APRectangle class. In this case, the default rectangle created by the default constructor is a rectangle with its top left corner at the origin and zero area.

public class APRectangle   
{   
  private APPoint myTopLeft;   
  private double myWidth, myHeight;   
  
  public APRectangle()   
  {   
    myTopLeft = new APPoint( 0.0, 0.0 );   
    myWidth = 0.0;   
    myHeight = 0.0;   
  }   
  
  public APRectangle( APPoint topLeft, double width, double height )   
  {   
    myTopLeft = topLeft;   
    myWidth = width;   
    myHeight = height;   
  }   
  
  public APRectangle( APPoint topLeft, APPoint bottomRight )   
  {   
    myTopLeft = topLeft;   
    myWidth = bottomRight.getX() - topLeft.getX();   
    myHeight = bottomRight.getY() - topLeft.getY();   
  }   
  
  // instance methods omitted   
}

[A little while ago](https://www.eimacs.com/eimacs/mainpage?epid=E2365695970&cid=162149), we saw that the keywords public and private may be used to control the visibility of class variables and constants. In particular, a class variable that is declared as private is only visible to the methods of its own class. On the other hand, a public class variable is visible and may be used anywhere in the program in which it is declared.

In exactly the same way, the keywords public and private can be used to control the visibility of instance variables. However, it is the convention in the Advanced Placement examination — and it is often considered good programming practice — to declare instance variables as private, thereby restricting access to their values to the data accessor methods of the class in which they are declared.

Similarly, the keywords public and private may also be used to control the visibility of both the instance and the class methods of a class. A public instance or class method is visible throughout the program in which it is defined, whereas a private instance or class method may only be referenced within the class in which it is defined.

In general, it is considered good practice to declare as public only those methods that are specifically designed to be used outside the class, and to declare all other methods to be private. A particularly striking example of this occurs in the context of multiple constructors, where it is quite common for the various constructors to differ solely in the extent to which they assign default values to instance variables. In such circumstances, it is common in many programming languages — including Java — to use a single private method — often called initialize — to perform instance variable initialization and to call that method from each of the different constructors. Consider the following class, for example:

public class FastFood   
{   
  private String myType;   
  private String myFriesSize;   
  private String myDrink;   
  
  public FastFood()   
  {   
    initialize( "hamburger", "no", "no drink" );   
  }   
  
  public FastFood( String type )   
  {   
    initialize( type, "no", "no drink" );   
  }   
  
  public FastFood( String type, String friesSize )   
  {   
    initialize( type, friesSize, "no drink" );   
  }   
  
  public FastFood( String type, String friesSize, String drink )   
  {   
    initialize( type, friesSize, drink );   
  }   
  
  private void initialize( String type, String friesSize, String drink )   
  {   
    myType = type;   
    myFriesSize = friesSize;   
    myDrink = drink;   
  }   
  
  /\* other methods \*/   
  
  public String toString()   
  {   
    return myType + " with " + myFriesSize + " fries and " + myDrink;   
  }   
}   
    
public class MainClass   
{   
  public static void main( String[] args )   
  {   
    FastFood orderA = new FastFood();   
    FastFood orderB = new FastFood( "fish sandwich" );   
    FastFood orderC = new FastFood( "cheeseburger", "medium" );   
    FastFood orderD = new FastFood( "garden salad", "no", "small lemonade" );   
    System.out.println( orderA );   
    System.out.println( orderB );   
    System.out.println( orderC );   
    System.out.println( orderD );   
  }   
}

In a case like this, it is clear that the sole purpose of the initialize method is to avoid the need for duplicating several similar code statements in each of the constructors. Its usefulness is strictly internal to this class, and therefore it is natural to designate it as a private method.

Java provides an alternative method of avoiding code duplication in a multiple constructor context such as the FastFood class. It uses the keyword this to refer to the FastFood class itself, as follows:

public class FastFood   
{   
  private String myType;   
  private String myFriesSize;   
  private String myDrink;   
  
  public FastFood()   
  {   
    this( "hamburger", "no", "no drink" );   
  }   
  
  public FastFood( String type )   
  {   
    this( type, "no", "no drink" );   
  }   
  
  public FastFood( String type, String friesSize )   
  {   
    this( type, friesSize, "no drink" );   
  }   
  
  public FastFood( String type, String friesSize, String drink )   
  {   
    myType = type;   
    myFriesSize = friesSize;   
    myDrink = drink;   
  }   
  
  /\* other methods \*/   
  
  public String toString()   
  {   
    return myType + " with " + myFriesSize + " fries and " + myDrink;   
  }   
}   
    
public class MainClass   
{   
  public static void main( String[] args )   
  {   
    FastFood orderA = new FastFood();   
    FastFood orderB = new FastFood( "fish sandwich" );   
    FastFood orderC = new FastFood( "cheeseburger", "medium" );   
    FastFood orderD = new FastFood( "garden salad", "no", "small lemonade" );   
    System.out.println( orderA );   
    System.out.println( orderB );   
    System.out.println( orderC );   
    System.out.println( orderD );   
  }   
}

This use of the keyword this is not part of the Advanced Placement Java subset. However, there are other uses of the keyword that are in the subset. We meet them [in a little while](https://www.eimacs.com/eimacs/mainpage?epid=E2124394331&cid=162149#This).

So far in this course, we have only seen public constructors. It is possible, however, to declare a constructor as private. This is useful in circumstances that occur in certain kinds of practical applications where it is important that only one instance of a class exists while a program is executing. Here is a trivial example:

public class OnlyOne   
{   
  private static OnlyOne myInstance;   
  private static String myName;   
  
  private OnlyOne()   
  {   
  }    
  
  public static OnlyOne getInstance( String name )   
  {   
    if ( myInstance == null)   
    {   
      myInstance = new OnlyOne();   
      myName = name;   
    }   
    return myInstance;   
  }   
  
  public String toString()   
  {   
    return "OnlyOne instance: " + myName;   
  }   
}   
  
public class MainClass   
{   
  public static void main( String[] args )    
  {   
    OnlyOne oo1 = OnlyOne.getInstance( "Adam" );   
    OnlyOne oo2 = OnlyOne.getInstance( "Eve" );     
    System.out.println( oo1 );   
    System.out.println( oo2 );   
  }   
}

Here, by declaring the default constructor to be private we make it invisible and inaccessible outside the OnlyOne class. In fact, the only way to create OnlyOne objects outside the OnlyOne class is to use the getInstance class method, and the conditional statement in that method prevents a new instance being created if one already exists. Thus, the second statement in the above main method does not create a new OnlyOne object with name "Eve"; instead, it creates oo2 as an "alias" for the already-in-existence object oo1. (We have more to say about [object aliasing](https://www.eimacs.com/eimacs/mainpage?epid=E2365696530&cid=162149) later on.)

Using the keyword this in the way that we remarked on the previous page is not in the Advanced Placement Java subset, we can also combine public and private constructors to enforce the provision of arguments when class instances are created. Here is a trivial example:

public class Trivial   
{   
  private int myIndex;   
  
  private Trivial()   
  {   
    System.out.println( "One more triviality!" );   
  }   
  
  public Trivial( int index )   
  {   
    this();   
    myIndex = index;   
  }   
  
  public String toString()   
  {   
    return "Trivial #" + myIndex;   
  }   
}   
  
public class MainClass   
{   
  public static void main( String[] args )   
  {   
    Trivial t1 = new Trivial( 1 );   
    Trivial t2 = new Trivial( 2 );   
    System.out.println( t1 );   
    System.out.println( t2 );   
  }   
}

Here, because the default constructor is private, the only way to create instances of the Trivial class is to use the public constructor, which requires the provision of an integer argument.

It is important to note that, just as in the case of super when creating subclasses of a class, the call to this in this context must be the first statement in the constructor code block.

The keywords public and private are the only access modifiers in the Advanced Placement Java subset. In addition to these, Java also provides two other types of visibility: protected and package.

[Some time ago](https://www.eimacs.com/eimacs/mainpage?epid=E2267319295&cid=162149), we introduced the idea of overloading a static method, that is, providing a variety of definitions for methods with the same name but different signatures. Since this was before we had told you about class definitions, we did not at that time mention that such a collection of overloaded class methods may be (and almost always will be) included within one and the same class definition. For the sake of completeness, we should also remark that it is also possible to overload instance methods in exactly the same way. For example, in the following code the Test class contains several variations of the pick method:

public class Test   
{   
  public Test()   
  {   
  }   
  
  public int pick()   
  {     
     return 0;   
  }   
  
  public int pick( int t )   
  {   
    return t;   
  }   
  
  public int pick( int t, int q )   
  {   
    return (t + q) / 2;   
  }   
}

So far in this course, we have come into contact with the wrapper classes Integer and Double on two occasions. The one you probably remember occurred when we [explained](https://www.eimacs.com/eimacs/mainpage?epid=E2193253980&cid=162149#Wrappers) by what means it is possible to store the primitive data types int and double in ArrayLists. At the time, we limited ourselves to stating that such classes exist, explaining how to declare ArrayLists with the corresponding type parameter, and assuring you that, having declared an ArrayList appropriately, you could add an int (or a double) to the ArrayList and get it back out of the ArrayList without having to be concerned about the conversions between ints and Integers (and between doubles and Doubles) that are going on behind the scenes. Let's take a moment now to consider those conversions.

Instances of the Integer class are created using that class's constructor, whose signature is

Integer( int n )

To retrieve the value of an Integer as an int we may use the accessor method intValue. The constructor and this method provide the tools with which the behind-the-scenes conversions between ints and Integers are actually performed. Similarly, the constructor for the Double class has the following signature:

Double( double d )

and the value of a Double may be retrieved as a double using the accessor method doubleValue.

Run the following code to verify that the behavior when storing ints in an ArrayList<Integer> and then retrieving the ints from the ArrayList is the same whether you perform the conversions yourself by hand or you allow them to be performed automatically.

    int[] t = { 1, 2, 3, 6, 10, 15, 21 };   
    ArrayList<Integer> a = new ArrayList<Integer>();   
    ArrayList<Integer> b = new ArrayList<Integer>();   
  
    // make two ArrayLists from array t   
    for ( int n : t )   
    {   
      a.add ( new Integer( n ) );   
      b.add ( n );   
    }   
  
    // print the contents of ArrayList a   
    for ( Integer x : a )   
      System.out.print( x.intValue() + " " );   
  
    // newline   
    System.out.println();   
  
    // print the contents of ArrayList b   
    for ( int n : b )   
      System.out.print( n + " " );

The program below is intended to find the arithmetic mean of the numbers stored in the array q in two ways: once by storing the numbers in an ArrayList d, where you allow all the necessary conversions to be performed automatically; and once by storing them in an ArrayList e, where you perform all the conversions by hand. Complete the program.

    double[] q = { 0.5, 2.4, 7.4, 2.8, -6.2 };   
    ArrayList<Double> d = new ArrayList<Double>();   
    ArrayList<Double> e = new ArrayList<Double>();   
  
    for ( double x : q )   
    {   
      d.add( x );   
      e.add ( new Double ( x ) );   
    }   
  
    double dTotal = 0.0,   
           eTotal = 0.0;

The following code performs the asked-for calculation:

  for ( double x : d )  
    dTotal += x;  
  
  for ( Double y : e )  
    eTotal += y.doubleValue();  
  
  System.out.println( "Mean of d is " + (dTotal / d.size()) );  
  System.out.println( "Mean of e is " + (eTotal / e.size()) );

The output that results from this code is:

Mean of d is 1.3800000000000003  
Mean of e is 1.3800000000000003

The other contact we have had with the Integer class and the Double class was rather more surreptitious. It occurred much earlier in this course, when we [introduced](https://www.eimacs.com/eimacs/mainpage?epid=E2212926314&cid=162149#WrapperMethods) the methods Integer.parseInt, Integer.toString, Double.parseDouble, and Double.toString. Now that you have more experience, you will readily recognize that the dot notation used to name these methods is a sure indication that they are methods of an Integer class and a Double class. We remark that these are just a few of the methods that are provided by these wrapper classes. As far as the Advanced Placement examination is concerned, however, you are only required to be familiar with a small number of them. For a complete listing, click [here](javascript:secWindow('mainpage?epid=E1922451555&cid=162149&s=2','WMethPop',540,460,50,50,'menubar,scrollbars,resizable')).