**Static vs. Non-static**

There are two types of methods in Java, static and non-static. You have repeatedly used non-static, or instance methods, within your class definitions (e.g., calcTotal, reverseInput), invoking them after constructing an object. For example, if you wrote the Mystery class, which contained a public instance method called someMethod, you would have to do the following to invoke the method:

Mystery theMystery = new Mystery();  
theMystery.someMethod();

This is a well-established syntactical pattern used in almost every program: construct an object (i.e., theMystery) then invoke a method (i.e., someMethod) using dot notation.

Imagine what would happen if the main method were a non-static instance method. Since program execution begins at the main method, nothing can happen prior to that point in time.

* If nothing can happen before the main method executes, no object can be constructed.
* If an object does not exist, an instance variable cannot be invoked.
* Therefore, the main method cannot be an *instance* method of an object because no objects exist before the main method executes.
* There are some things you never see, such as baby pigeons and statements like someObject.main.

So why is the main method static? Static methods can be executed **without** the prior existence of an object. Consequently, without a static main method, your program cannot run. If you doubt this, remove static from the main method statement of any program you have written, compile it, and run it. Nothing, nada, zip!

### Part 1

Static methods, also called class methods, provide some unique capabilities not available with instance methods. Be sure to carefully study the eIMACS labs in order to gain a better understanding and appreciation of static methods. The interactive lab exercises are very important, so use them to your advantage.

When we first introduced [static methods](https://www.eimacs.com/eimacs/mainpage?epid=E205325272&cid=162149#StaticMethod), it was before we had made any mention of [class definitions](https://www.eimacs.com/eimacs/mainpage?epid=E205325330&cid=162149). Later on, when we told you a little about how the [Java compiler](https://www.eimacs.com/eimacs/mainpage?epid=E2296832518&cid=162149#MethodInClass) operates, we revealed that Java requires the definition of *every* method — including the static main method — to appear within a class definition.

In fact, prior to revealing this information, we had secretly been arranging things so that the definitions of all the static methods that we discussed or that you created appeared in the same class that included the definition of the main method. When you completed the definition of the factorial static method in [Exercise 91](https://www.eimacs.com/eimacs/mainpage?epid=E2001138599&cid=162149#Exe084), for example, your code probably looked like this:

  public static int factorial( int n )   
  {   
    if ( n == 0 )   
      return 1;   
  
    return n \* factorial( n - 1 );   
  } z  
  
  public static void main( String[] args )   
  {   
    System.out.println( factorial( 5 ) );   
  }

If you click on the *Show program details »* link above, you will see that in fact the definitions of both factorial and main are included within the definition of the same MainClass class.

Static methods — that is, methods such as factorial whose headers include the modifier keyword static — are sometimes also referred to as *class methods*. (The two phrases "static method" and "class method" are synonymous; they may be used interchangeably.) Methods whose headers do not include the static modifier are *instance methods* of the class within whose definition they are defined.

There are important practical implications of the presence or absence of the static modifier. A public class method may be invoked and executed as soon as Java processes the definition of the class to which it belongs. That is not the case for an instance method. Before a public instance method may be invoked and executed, an instance must be created of the class to which it belongs. To use a public class method, you just need the class. On the other hand, before you can use a public instance method you must have an *instance* of the class.

The manner in which a static method is invoked within the definition of another method varies according to whether or not the two methods belong to the same class. In the example above, factorial and main are both methods of the MainClass class. As a result, the invocation of factorial in the definition of main simply references the method name, "factorial".

It is perfectly possible, however, for factorial and main to belong to different classes, as they do here:

public class MathMethods

{

public static int factorial( int n )

{

if ( n == 0 )

return 1;

return n \* factorial( n - 1 );

}

}

public class MainClass

{

public static void main( String[] args )

{

System.out.println( factorial( 5 ) );

}

}

By separating the definitions into different classes, though, we have broken the program. If you click the **Run** button above, the program will generate an error message. Try it and see!

This happens because, when Java is asked to execute a static method using just the method name, it looks for that method in the current class. In this case, of course, there is no factorial method to be found in the class to which main belongs, so Java complains that it "cannot resolve" the symbol factorial. To get around this problem, we must tell Java where to look for the factorial method. We do this by referencing the class name using *dot notation*. In the main method above, replace "factorial" by "MathMethods.factorial" and run the program to verify that this fixes the problem.

The rules we gave on the previous page for referencing a class method within the definition of another method apply in particular when that other method is an instance method. Consequently, within the definition of an instance method we may reference any class method that belongs to the same class without using dot notation. It follows that the class methods of a class are directly available to all instances of that class. Consider the following class definition, for example. Notice in particular that the instance method printPrimary refers to the class method isPrimary without using dot notation. On the other hand, since the main method belongs to a different class, it must use dot notation to reference the isPrimary method.

public class Color   
{   
  private String myColorName;   
  
  public Color( String colorName )   
  {   
    myColorName = colorName;   
  }   
  
  public void printPrimary()   
  {   
    if ( isPrimary( myColorName ) )   
      System.out.println( myColorName + " is primary" );   
    else   
      System.out.println( myColorName + " is NOT primary" );   
  }   
  
  public static boolean isPrimary( String colorName )   
  {   
    return colorName.equals( "red" )   
                  || colorName.equals( "yellow" )   
                  || colorName.equals( "blue" );   
  }   
}   
  
public class MainClass   
{   
  public static void main( String[] args )   
  {   
    Color b = new Color( "blue" );   
    Color g = new Color( "green" );   
  
    b.printPrimary();   
    g.printPrimary();   
  
    System.out.println( Color.isPrimary( "red" ) );   
  }   
}

We remark in passing that there are many ways to encode the printPrimary method. We chose to decompose part of the operation into a subsidiary operation implemented by the class method isPrimary. The decomposition of operations into subsidiary operations is sometimes called functional decomposition, and is an essential part of good design practice.

Let us summarize what we have learned so far about one method calling another.

* Any class method and any instance method of one class may call any public class method of another class, but in order to do so it must use dot notation and reference the name of the second class.
* Any class method and any instance method of one class may call any public instance method of another class, but in order to do so it must use dot notation and reference the name of an *instance* of the second class.
* Any instance method of a class may call any class method of the same class, and may do so without using dot notation.

That leaves one combination unaccounted for: how can a class method call an instance method of the same class? It might be expected that, since they belong to the same class, there is no question about which instance method is intended, so dot notation is unnecessary. According to this way of thinking, in the following example we might expect the add class method to be able to access the getN instance method without benefit of dot notation.

public class IntClass   
{   
  private int myN;   
  
  public IntClass( int n )   
  {   
    myN = n;   
  }   
  
  public int getN()   
  {   
    return myN;   
  }   
  
  public static int add( int m )   
  {   
    return getN() + m;   
  }   
}   
  
public class MainClass   
{   
  public static void main( String[] args )   
  {   
    IntClass t = new IntClass( 5 );   
    int y = IntClass.add( 3 );   
    System.out.println( y );   
  }   
}

If you click the **Run** button, however, you will discover that an error message is generated indicating that the instance method getN cannot be referenced directly within the definition of a class method (in this case, the add method).

The governing principle is that an instance method does not become available until an instance of the class has been created. So within the definition of the add class method we must ensure that the call to the getN instance method is referenced to an instance. The easiest way to guarantee this is to insist that an instance be provided to the class method in the form of an argument, as in the following code (which does *not* produce an error when run):

public class IntClass   
{   
  private int myN;   
  
  public IntClass( int n )   
  {   
    myN = n;   
  }   
  
  public int getN()   
  {   
    return myN;   
  }   
  
  public static int add( int m, IntClass i )   
  {   
    return i.getN() + m;   
  }   
}   
  
public class MainClass   
{   
  public static void main( String[] args )   
  {   
    IntClass t = new IntClass( 5 );   
    int y = IntClass.add( 3, t );   
    System.out.println( y );   
  }   
}

The reason why this technique is not necessary when one instance method calls another instance method in the same class is that the calling method is not available until an instance has been created. In such a context, both the calling and the called methods are of course assumed to be methods of that instance.

### Part 2

Where have you seen static methods? Consider the following code segments. Which of the following is correct?

int x = 4;  
int root = x.sqrt();

int x = 4;  
int root = Math.sqrt(x);

While the statement on the left looks like it might be valid, there is a fundamental flaw: methods operate on objects, not on primitive data types. The **x** variable cannot be used in the object position in dot notation because it is of type int (i.e., a primitive data type).

Likewise, the statement on the right looks valid, although no Math object has been declared; however, you know from past experience that this is a perfectly valid statement.

Take a quick look at the Java [Math class](https://l.flvsgl.com/GSL0978f93819e481697621edf45b45af7b3). You will see that Math class methods are all static. Static methods **are not** invoked on an object. Since classes always begin with a capital letter, you might deduce that the class name can be used in dot notation when invoking a static method. It is all very logical when you stop and think about it.

## Answer Key

Top of Form

## Question 1 (Worth 4 points)

c  
  
public class MyClass  
{  
    public static void someMethod()  
    {  
        // implementation not shown  
    }  
  
    public void someOtherMethod()  
    {  
        // implementation not shown  
    }  
  
    private void yetAnotherMethod()  
    {  
        // implementation not shown  
    }  
}  
  
The method yetAnotherMethod is defined as private. This means

 the method is accessible outside MyClass

 the method is not accessible outside MyClass

 the method is accessible only by a static method of MyClass

 the method is accessible without instantiating a MyClass object

 the method is accessible only by using a previously instantiated MyClass object

Points earned on this question: **4**

## Question 2 (Worth 4 points)

Consider the following client code and assume that it compiles correctly:  
  
public class MyTesterClass  
{  
    public static void main(String[] args)  
    {  
        MyClass myObject = new MyClass (12.4, 20);  
  
        int value1 = MyClass.SOME\_VALUE;  
        int value2 = myObject.method1();  
        int value3 = MyClass.method2(20);  
    }  
}  
  
Which of the following is an instance method?

 method1

 method2

 MyClass

 SOME\_VALUE

 This cannot be determined by examining the above code

Points earned on this question: **0**

## Question 3 (Worth 4 points)

When one method calls another, which of the following statements are true?

1. The class methods and instance methods of one class may call the public class methods of another class using dot notation and referencing the name of the other class.
2. The class methods and instance methods of one class may call the public instance methods of another class by using dot notation to invoke the method on an instance of the other class.
3. The instance methods of a class may call, only using dot notation, any class method of the same class.

 I only

 III only

 I and II only

 I and III only

 I, II, and III

Points earned on this question: **0**

## Question 4 (Worth 4 points)

A method defined within a class and whose header does NOT include the static modifier is known as which of the following?

 class method

 instance method

 super method

 private method

 public method

Points earned on this question: **4**

## Question 5 (Worth 4 points)

Which of the following describes an overridden method?

 Any method that invokes super();

 A method that contains a call to itself

 One of many methods in a class with the same name but different return values

 One of many methods in a class with the same name but different parameter lists

 A method in a subclass with the same method heading as a method in a parent class

Points earned on this question: **0**

Bottom of Form