When we first introduced [static methods](https://www.eimacs.com/eimacs/mainpage?epid=E2395204352&cid=162149#StaticMethod), it was before we had made any mention of [class definitions](https://www.eimacs.com/eimacs/mainpage?epid=E2212927970&cid=162149). Later on, when we told you a little about how the [Java compiler](https://www.eimacs.com/eimacs/mainpage?epid=E2203090901&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=E1942114541&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 );   
  }   
  
  public static void main( String[] args )   
  {   
    System.out.println( factorial( 5 ) );   
  }

[Show program details »](https://www.eimacs.com/eimacs/mainpage?cid=162149&epid=E1961790107)

120

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( MathMethods.factorial( 5 ) );

  }

}

120

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.

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 );   
  }   
}

IntClass.java:20: error: non-static method getN() cannot be referenced from a static context   
  
    return getN() + m;   
  
           ^

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 );   
  }   
}

8

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.