Static Methods in Java

The Java construct for implementing functions is known as the *static method.*

Using and defining static methods.

The use of static methods is easy to understand. For example, when you write `Math.abs(a-b)` in a program, the effect is as if you were to replace that code with the *return value* that is produced by Java's `Math.abs()` method method when passed the expression `a-b` as an *argument.*

In the screen shot below substitute **System.out.println for StdOut.** These examples use some helpers that we don't have.

```
public class Harmonic
{

   public static double harmonic(int n)
   {
      double sum = 0.0;
      for (int i = 1; i <= n; i++)
         sum += 1.0/i;
      return sum;
   }


   public static void main(String[] args)
   {
      for (int i = 0; i < args.length; i++)
      {
         int arg = Integer.parseInt(args[i]);

         double value = harmonic(arg);

         StdOut.println(value);
      }
   }
}
```
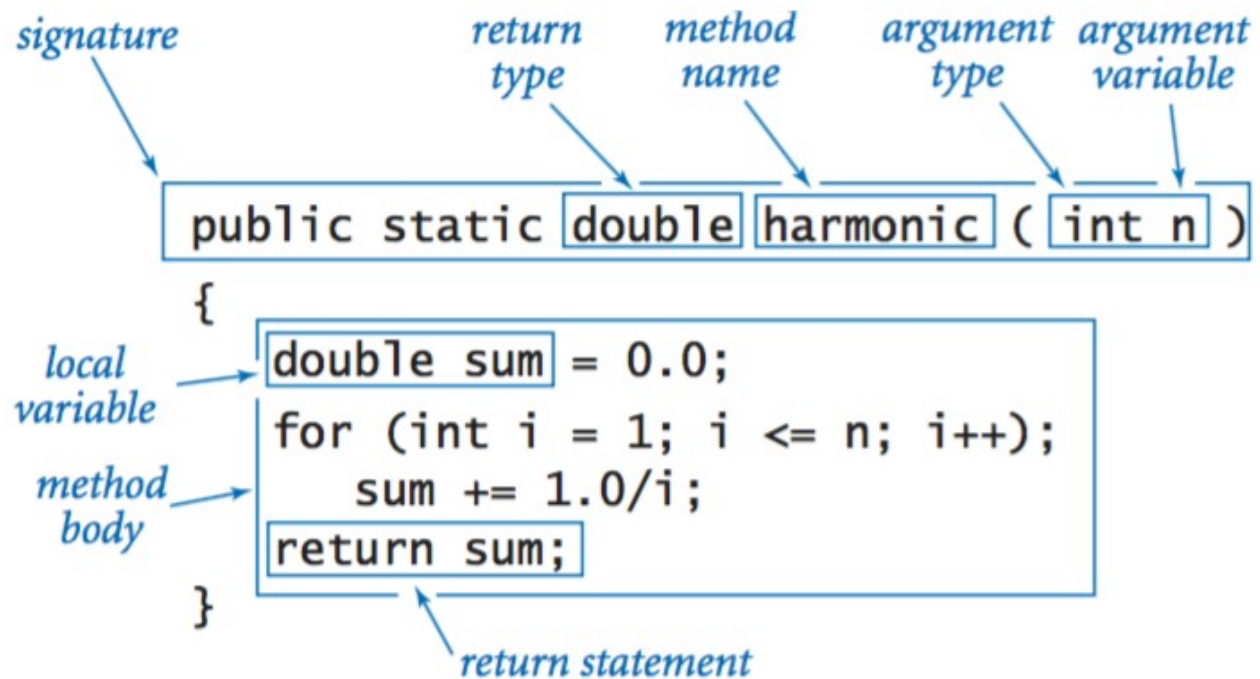
```
i = 1
arg = 1
harmonic(1)
    sum = 0.0
    sum = 1.0
    return 1.0
value = 1.0
i = 2
arg = 2
harmonic(2)
    sum = 0.0
    sum = 1.0
    sum = 1.5
    return 1.5
value = 1.5
i = 3
arg = 4
harmonic(4)
    sum = 0.0
    sum = 1.0
    sum = 1.5
    sum = 1.8333333333333333
    sum = 2.083333333333333
    return 2.083333333333333
value = 2.083333333333333
```
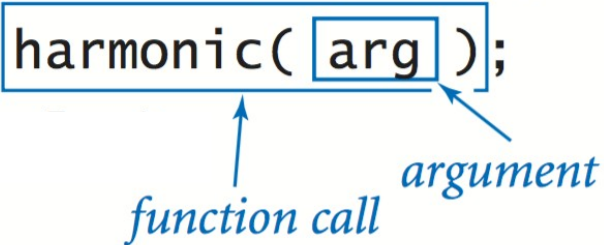
Function-call trace for
java Harmonic 1 2 4

- *Function-call trace.* One simple approach to following the flow of control through function calls is to imagine that each function prints its name and argument value when it is called and its return value just before returning, with indentation added on calls and subtracted on returns.
- *Debugger.* A better way to understand control flow through methods, loops and decision structures is to set break points and step through running code in debug mode.
- *Static method definition.* The first line of a static method definition, known as the *signature,* gives a name to the method and to each *parameter variable.* It also specifies the *type* of each parameter variable and the *return type* of the method. Following the signature is the *body* of the method, enclosed in curly braces. The body consists of the kinds of statements we discussed in

Chapter 1. It also can contain a *return statement,* which transfers control back to the point where the static method was called and returns the result of the computation or *return value.* The body may declare *local variables,* which are variables that are available only inside the method in which they are declared.

```
public static double harmonic ( int n )
{
    double sum = 0.0;
    for (int i = 1; i <= n; i++);
        sum += 1.0/i;
    return sum;
}
```

signature — *return type* — *method name* — *argument type* — *argument variable*

local variable → `double sum`

method body → *return statement* — `return sum;`

Function calls. A static method call is nothing more than the method name followed by its arguments, separated by commas and enclosed in parentheses. A method call is an expression, so you can use it to build up more complicated expressions. Similarly, an argument is an expression—Java evaluates the expression and passes the resulting value to the method. So, you can write code like `Math.exp(-x*x/2) / Math.sqrt(2*Math.PI)` and Java knows what you mean.
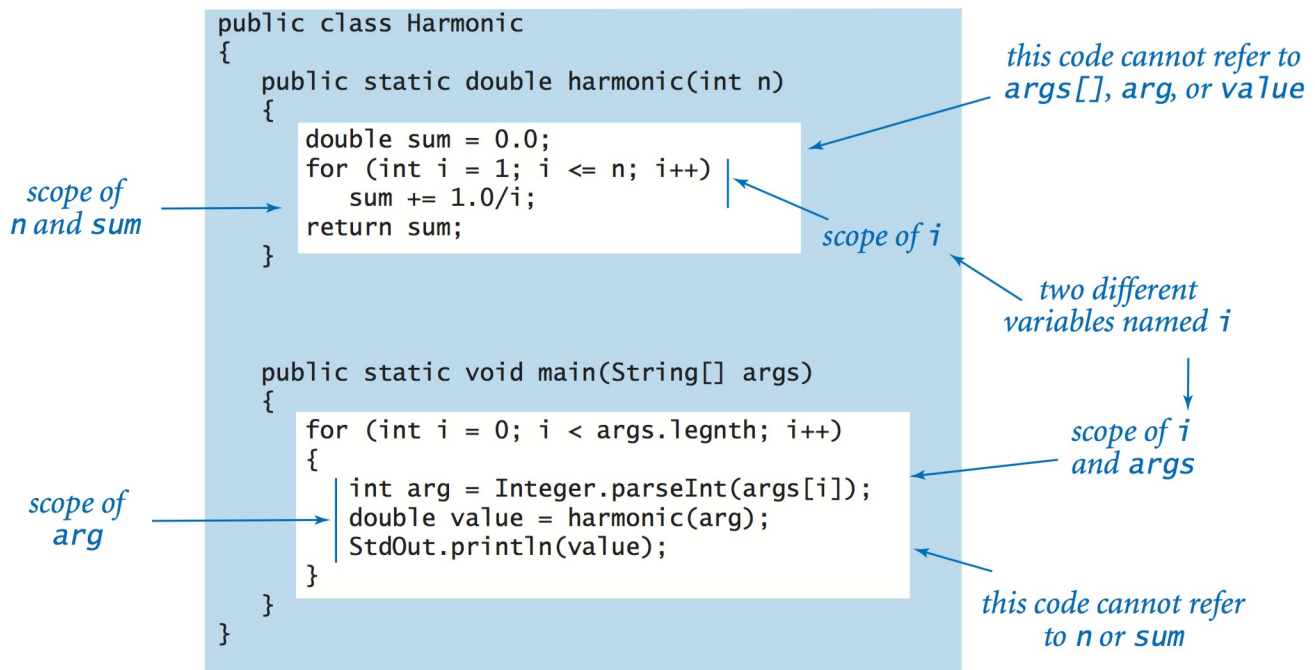
```
for (int i = 0; i < args.length; i++)
{
    arg = Integer.parseInt(args[i]);
    double value = harmonic( arg );

}
```
function call   *argument*

Properties of static methods.

- ***Multiple arguments.*** Like a mathematical function, a Java static method can take on more than one argument, and therefore can have more than one parameter variable.

- ***Multiple methods.*** You can define as many static methods as you want in a `.java` file. These methods are independent and can appear in any order in the file. A static method can call any other static method in the same file or any static method in a Java library such as `Math`.

- ***Overloading.*** Static methods with different signatures are different methods. whose signatures differ are different static methods. Using the same name for two static methods whose signatures differ is known as *overloading*.

- ***Multiple return statements.*** You can put `return` statements in a method wherever you need them: control goes back to the calling program as soon as the first `return` statement is reached.

- ***Single return value.*** A Java method provides only one return value to the caller, of the type declared in the method signature.

- ***Scope.*** The *scope* of a variable is the part of the program that can refer to that variable by name. The general rule in Java is that the scope of the variables declared in a block of statements is limited to the statements in that block. In particular, the scope of a variable declared in a static method is limited to that method's body. Therefore, you cannot refer to a variable in one static method that is declared in another.

```
public class Harmonic
{
   public static double harmonic(int n)
   {
      double sum = 0.0;
      for (int i = 1; i <= n; i++)
         sum += 1.0/i;
      return sum;
   }


   public static void main(String[] args)
   {
      for (int i = 0; i < args.legnth; i++)
      {
         int arg = Integer.parseInt(args[i]);
         double value = harmonic(arg);
         StdOut.println(value);
      }
   }
}
```

scope of n and sum

this code cannot refer to args[], arg, or value

scope of i

two different variables named i

scope of i and args

scope of arg

this code cannot refer to n or sum

Side effects*. A* pure function *is a function that, given the same arguments, always returns the same value, without producing any observable* side effects, *such as consuming input, producing output, or otherwise changing the state of the system. The function* `harmonic()` *is an example of a pure function. However, in computer programming, we often define functions whose* only *purpose is to produce side effects. In Java, a static method may use the keyword* `void` *as its return type, to indicate that it has no return value.*

Using static methods to organize code.

With the ability to define functions, we can better organize our programs by defining functions within them when appropriate. Check the file **CouponCollector.java** to study a program broken into methods.

- Given *n*, compute a random coupon value.
- Given *n*, do the coupon collection experiment.
- Get *n* from the command line, and then compute and print the result.

```
16  public class Coupon {
17
18      // return a random coupon between 0 and n-1
19      public static int getCoupon(int n) {
20          return (int) (Math.random() * n);
21      }
22
23      // return number of cards you collect before obtaining one of each of the n types
24      public static int collect(int n) {
25          boolean[] isCollected = new boolean[n];  // isCollected[i] = true if card type i already collected
26          int count = 0;                           // number of cards collected
27          int distinct  = 0;                       // number of distinct card types collected
28
29          // repeat until you've collected all n card types
30          while (distinct < n) {
31              int value = getCoupon(n);            // pick a random card
32              count++;                             // one more card
33              if (!isCollected[value]) {           // discovered a new card type
34                  distinct++;
35                  isCollected[value] = true;
36              }
37          }
38          return count;
39      }
40
41      // test client
42      public static void main(String[] args) {
43          int n = Integer.parseInt(args[0]);
44          int count = collect(n);
45          System.out.println(count);
46      }
47  }
```

Whenever you can clearly separate tasks in programs, you should do so.

Passing arguments and returning values.

Next, we examine the specifics of Java's mechanisms for passing arguments to and returning values from functions. Study the file **ArrayFunctionExamples.java** to see illustrations of the following concepts.

- *Pass by value.* You can use parameter variables anywhere in the body of the function; the same way you use local variables. The only difference between a parameter variable and a local variable is that Java evaluates the argument provided by the calling code and initializes the parameter variable with the resulting value. This approach is known as *pass by value*.
- *Arrays as arguments.* When a static method takes an array as an argument, it implements a function that operates on an arbitrary number of values of the same type. Methods that accept arrays as arguments can modify the original array because it is the address of the array that is passed into the method. The array itself is not copied.
- *Side effects with arrays.* It is often the case that the purpose of a static method that takes an array as argument is to produce a side effect (change values of array elements). Methods that produce side effects on arrays are often void as the array itself is modified.
- *Arrays as return values.* A method can also provide an array as a return value.

See the examples provided in the accompanying file.