Chapter 6 Static Functions

Java AP

It's All About Functions

Some of the first statements made during this course were these:

- A Java program is essentially a collection of one or more classes.
    - Which are collections of functions and data.
- There must be a function named **main** and it must be in all lowercase.
- **main** is the program's starting point; it can call other functions by name.
- There must be *exactly* one function named **main** in every program.
- Functions, can also be called methods.

And the general form of a Java program was shown. (The parts in **bold** are the subject of this topic)

```
package imports

class declaration
{
     data declaration (member fields)

     function definitions

     main function definition
}
```

Although we've actually only written one function so far (**`main`**), we've used others: System.out.println , Scanner.nextInt() These functions are in the Java standard library. These functions unlike main are not static, in this chapter we are going to cover static functions, non-static functions will be covered later.

A static function has the form:

```
modifier static return_type function_name(formal_parameters)
{
  function_body
}
```

Explanations:

Function Part	Description
modifier	The modifier specifies the accessibility of a function (public, private, or protected). For the sake of simplicity, all our functions will be public. all types of modifiers will be covered in more detail in a future chapter. **NOTE:** If you do not specify a modifier it will default to private which is probably not the behavior you want most of the time. It is good practice to always specify a modifier as it makes your code clearer.
return_type	This describes the type of the data that the function will return. Almost all data types are allowed to be returned from a function. If a function returns data, then the function can be used in an expression, otherwise, it can't.
function_name	The name of the function. The name must start with either a _ or a letter, and can only contain a-z, A-Z, 0-9, _
formal_parameters	This is an optional comma-separated list of values that will be passed to the function.
function_body	The body consists of all of the declarations and executable code for the function. If the function is defined to return a value, there must be at least one return statement in the body. (There can be multiple return statements.)

Here's an example of a *user-defined* static function:

```
public static float average(float a, float b)
{
  return (a + b) / 2.0f;
}
```

The function above meets the requirements:
1. The function is named *average*.
2. It will return a value of type **`float`**.
3. It expects to be passed two values, both of type **`float`**. (Note that you can't do this for the parameters: **`float`** a, b)
4. Has a modifier

A complete program using our new function:

```
public class Main
{
    public static float average(float a, float b)
    {
           return (a + b) / 2.0f;
    }

   public static void main(String[] args)
   {
       float x = 10.0f;
       float y = 20.0f;

       float ave = average(x,y);
       System.out.println("Average of " + x + " and " + y + " is " + ave);

       x = 7.0f;
       y = 10.0f;
       ave = average(x,y);
       System.out.println("Average of " + x + " and " + y + " is " + ave);
   }
}
```

Output:	Average of 10.0 and 20.0 is 15.0 Average of 7.0 and 10.0 is 8.5

Both the return value and the parameters are optional:

No Return and No Parameters	No Return and One Parameter	Return and No Parameters

public static void foo()
{
System.out.println(“Foo”);
}
	Public static void bar(int one)
	{
	System.out.println(one);
	}
		public static float baz()
		{
		return 0.0f;
		}

For simplicity we are going to define all of our functions in our Main class (where our main function resides).

Calling the functions:

```
  public static void main(String[] args)
  {
     foo();           /* No arguments, needs parentheses */
     bar(5);          /* One argument*/
     float p = baz(); /* No arguments */
     p = baz;         /* Error, parentheses are required */
  }
```

When calling these functions we are only using the name of the function. This is possible because all of the functions (main, Foo, Bar and Baz) are defined in the Main class.

Full Code

```
public class Main
{
    public static void foo()
    {
       System.out.println("Foo");
    }

    public static void bar(int one)
    {
        System.out.println(one);
    }

    public static float baz()
    {
        return 0.0f;
    }

    public static void main(String[] args)
    {
        foo();           /*no arguments, needs parentheses */
        bar(5);          /* One argument*/
        float p = baz(); /* No arguments */
    }
}
```

Common problems when first using functions

Example

```
public class Main
{
      public static float average(float a, float b)
      {
        return (a + b) / 2.0f;
      }

      public static void main(String[] args)
      {
          /* Define some variables */
           int i;
           float f1, f2, f3, f4;
           double d1, d2;

           /* Set some values */
           f1 = 3.14F;  f2 = 5.893F;  f3 = 8.5F;
           d1 = 3.14;  d2 = 5.893;

           /* Fatal errors when these execute */
           i = average(f1, f2);
           f4 = average(d1, d2);
           f4 = average(3.14, 9.1F);
           f4 = average(f2);
           f4 = average(f1, f2, f3);
      }
}
```

Compiler Errors

```
Exception in thread "main" java.lang.Error: Unresolved compilation
problems:
      Type mismatch: cannot convert from float to int
      The method average(float, float) in the type Main is not
applicable for the arguments (double, double)
      The method average(float, float) in the type Main is not
applicable for the arguments (double, float)
      The method average(float, float) in the type Main is not
applicable for the arguments (float)
      The method average(float, float) in the type Main is not
applicable for the arguments (float, float, float)

      at Main.main(Main.java:20)
```

Tracing Function Calls

What is the sequence of function calls made by the program below. Specify the function being called and its parameters.

```
public class Main
{
    public static void fnB()
    {
        System.out.print("tied the room " );
    }

    public static void fnC()
    {
        System.out.println("together");
    }

    public static void fnA()
    {
        System.out.print("The rug really " );
        fnB();
    }

    public static void fnD()
    {
        fnA();
        fnC();
    }

    public static void main(String[] args)
    {
        fnD();
    }
}
```

The sequence is this:

```
main();
fnD();
fnA();
System.out.print("That rug really ");
fnB();
System.out.print("tied the room ");
fnC();
System.out.println("together");
```

What is the output of the program?

Pass By Value

In Java, all function arguments that are **primitive** types (byte, short, int, long, float, double, boolean and char) are passed by value. In a nutshell, this means that any changes made to the parameters in the body of the function will not affect the values at the call site. Non-integral types are subtly different, and will be covered later. An example will clarify:

Example:

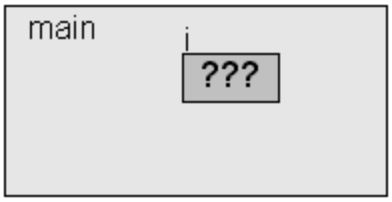
```
public class Main
{
    public static void fn(int x)
    {
       System.out.println("In fn, before assignments, x is " + x);

        x = 10;
       System.out.println("In fn, after assignments, x is " + x);
    }

    public static void main(String[] args)
    {
        int i;
        i = 5;

        System.out.println("Before call: i is " + i);

        fn(i);

        System.out.println("After call: i is " + i);
    }
}
```

Output:

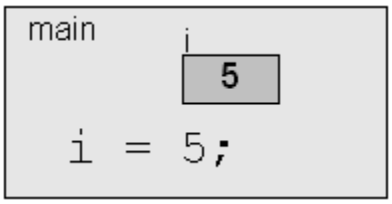
```
Before call: i is 5
In fn, before assignment, x is 5
In fn, after assignment, x is 10
After call: i is 5
```

A *copy* of the value is passed to the function, so any changes made are made to the copy, not the original value. Visually, the process looks something like this:

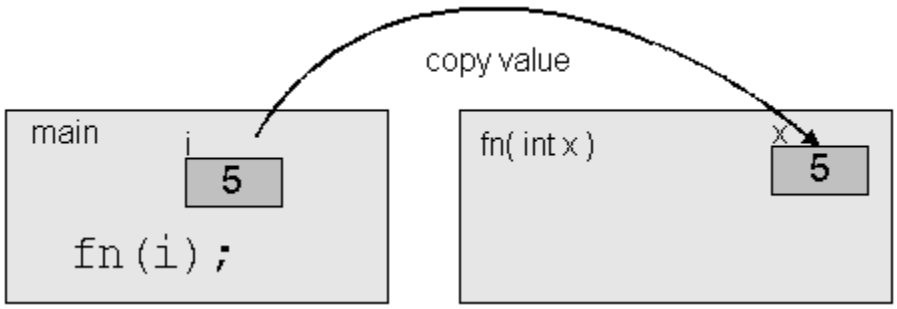
When `main` begins, `i` is undefined:



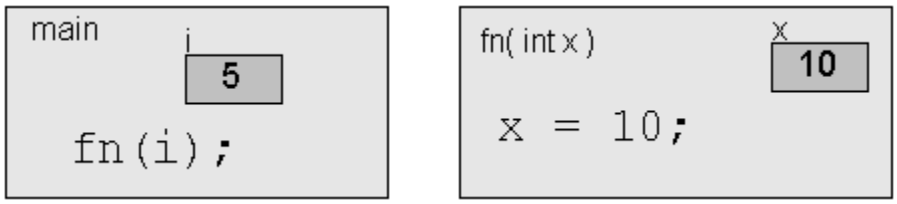
Next statement, `i` is assigned a value:



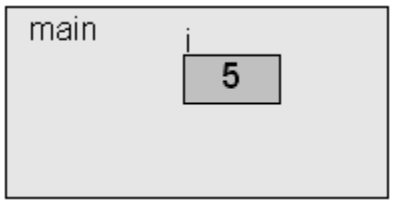
The function is called and a copy of the argument is made:



The parameter, `x`, is assigned a value:



The function returns back to `main` and the original value is preserved:



Thus we can modify the integral parameter without worrying that we will be changing something elsewhere in code.

while loop (with extra variable)	while loop (no extra variable)

public static void sayHelloALot(int count)
{
int i = 0;
while(i < count)
{
System.out.println("Hello");
++i;
}
}
	public static void sayHelloALot(int count)
	{
	while(count > 0)
	{
	System.out.println("Hello");
	count--;
	}
	}

List Of Primitive Types:
byte short int
long
float double boolean char

We will cover String, and other Objects(instances of a class) later, their behavior is subtly different.

Functions and Scope

What happens when we have a variable named `A` in two different functions? Does the compiler or program get confused?

```
void fn1(void)
{
  int A;
  /* statements */
}
```

```
void fn2(void)
{
  int A;
  /* statements */
}
```

Declaring/defining variables inside of a function makes them *local* to that function. No other function can see them or access them.

Example:

Each function contains its own "a" variable

```
public class Main
{
   public static int add3(int x, int y, int z)
   {
      int a  = x + y + z;
      return a;
   }

   public static int mult3(int x, int y, int z)
   {
      int a = x * y * z;
      return a;
   }

   public static void main(String[] args)
   {
      int a = 2;
      int b = 3;
      int c = 4;
      int d = add3(a,b,c);
      int e = add3(a,b,c);
   }
}
```

Technically, curly braces define a scope. This means that any time you have a pair of curly braces, a new scope is created. So, yes, this means that compound statements within loops (`for`, `while`, etc.) and conditionals (`if`, etc.) are in a different scope than the statements outside of the curly braces. More on scopes later...

The return Statement

We've already seen the `return` statement many times so far. It is used when we want to leave (i.e. *return* from) a function.

The general form is:

```
return expression ;
```

- If the function is defined to return **`void`**, then *expression* is not provided.
- If the function is defined to return some value, then the type of *expression* must match (or be compatible with) the type specified.
- Must not make code inaccessible.
- A function can have several `return` statements. These functions all do the same thing:

These 3 functions below are equivalent:

Function 1:

```
public static int compare1(int a, int b)
{
        int value;
        if(a > b)
        {
            value = 1;
        }
        else if( a < b)
        {
            value = -1;
        }
        else
        {
            value = 0;
        }
        return value;
}
```

Function 2:

```
public static int compare2(int a, int b)
{
        if(a > b)
        {
            return 1;
        }
        else if(a < b)
        {
            return -1;
        }
        else
        {
            return 0;
        }
}
```

Function 3:	public static int compare3(int a, int b)
	{
	if(a > b)
	{
	return 1;
	}
	
	if(a < b)
	{
	return -1;
	}
	
	return 0;
	}

These functions both have illegal `return` statements:

Function	Compiler Error

public static int f1(int a, int b)
{
/* statements */
 	
/* Illegal, must return an integer */
return;
}
	Main.java:8: error: missing return value
	return;
	^

public static void f2(int a, int b)
{
/* statements */
 	
/* Illegal, can't return anything */
return 0;
}
	Main.java:16: error: cannot return a value from function whose result type is void
	return 0;
	^

Additionally, unlike some other languages, you cannot hide code with a return statement. The following code is illegal:

Function	public static void Foo()
	{
	System.out.println(“Foo”);
	
	return;//Illegal there is code after this point, in the same block.
	
	System.out.println(“You cannot reach me!”);
	}

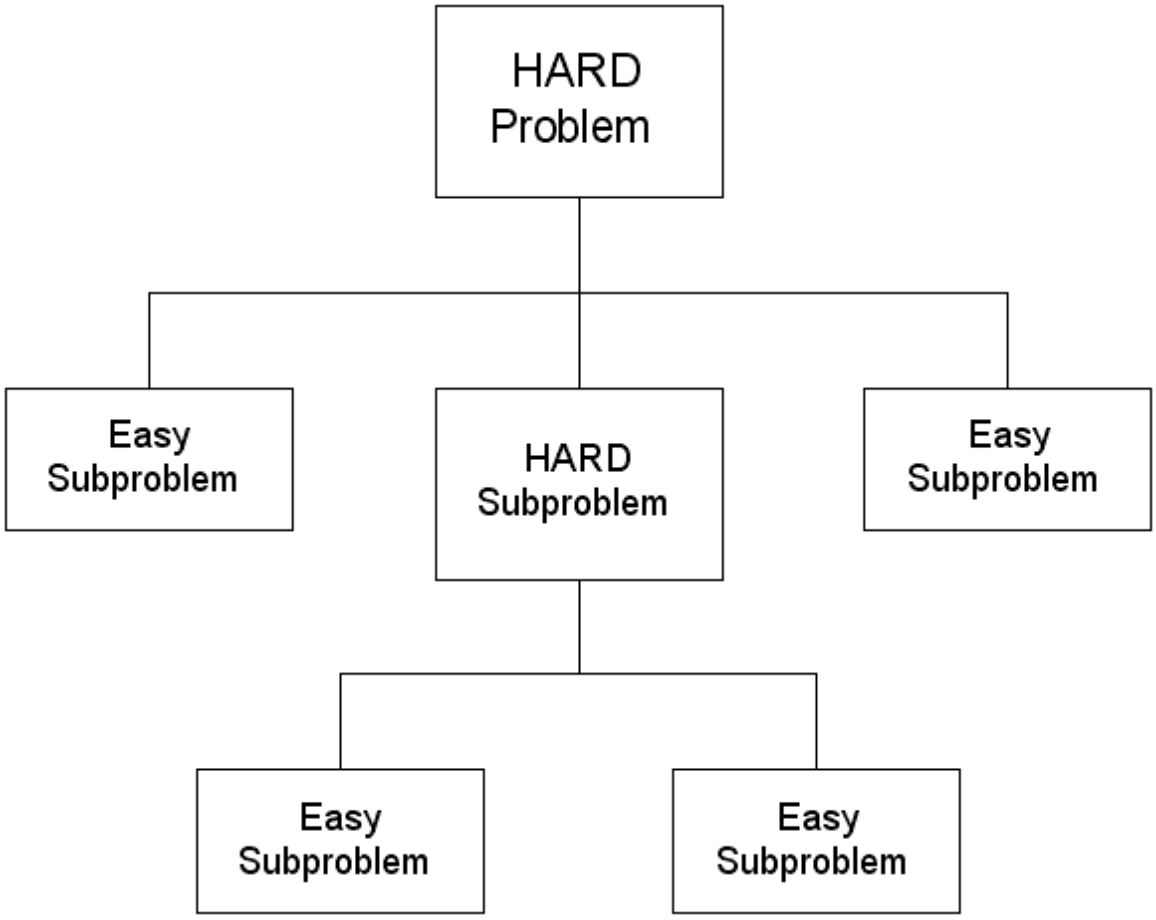
Compiler Error	**Main.java:12: error: unreachable statement** **System.out.println("You cannot reach me!");**

This limitation works on block scope thus the following is legal:

```
public static void Foo()
{
   System.out.println("Foo");

   int x = 4;

   if(x > 2)
   {
         return;//Legal there is no code after this point in the block
   }

   System.out.println("You can reach me!");
}
```

Divide and Conquer

The main point of using functions is to break the program into smaller, more manageable pieces. The technique is called *Divide and Conquer* and can be visualized like this:



The idea is that it will be easier to work individually on smaller parts of the whole problem, rather than try to solve the entire problem at once.

More on Scope

The *scope* or *visibility* of an identifier is what determines in which parts of the program the identifier can be seen.

- We've seen function scope, where variables declared in function can be seen anywhere within the function that follows that point of declaration.

Example:

```
public static foo()
{
   int a = 0;

   System.out.println(a);/* Legal a was defined before we tried to use it */
   System.out.println(b);/* Illegal b was defined after this point */

   int b = 2;

   System.out.println(b);/* Legal b was defined before this point */
}
```

- This is a form of *block scope*, and is delimited by the curly braces.
- An identifier with block scope is visible from its point of declaration to the end of the block that encloses the declaration.
- Block scope is delimited by curly braces (e.g. in functions, compound statements, etc.)

This contrived example shows three different scopes. (The curly braces for each scope are highlighted.)

All statements are legal:

```
public static void f1(int param)/*Scope of param starts here */
{
      int a = 0;/* Scope of a starts here */
      int b = 0;/* Scope of b starts here */
      int c = 0;/* Scope of c starts here */

      while(a < 10)
      {
         int x = 0;/* Scope of x starts here */

         if(b == 5)
         {
            int p;/* Scope of p starts here */
            int q;/* Scope of q starts here */

            p = a; /* Ok, both in scope */
            q = x + param; /*Ok, both in scope */
         }/* Scope of p and q ends here */

         int y = 0;/* Scope of y starts here */

         x = a; /*Ok, both in scope */
         y = b; /*Ok, both in scope */

      }/* Scope of x and y ends here */

}/*Scope for a,b,c, and param ends here */
```

Some statements are not legal:

```
 public static void f1(int param)/*Scope of param starts here */
{
      int a = 0;/* Scope of a starts here */
      int b = 0;/* Scope of b starts here */
      int c = 0;/* Scope of c starts here */

      while(a < 10)
      {
         int x = 0;/* Scope of x starts here */

         if(b == 5)
         {
            int p;/* Scope of p starts here */
            int q;/* Scope of q starts here */

            p = a; /* Ok, both in scope */
            q = x + param; /*Ok, both in scope */
         }/* Scope of p and q ends here */

         int y = 0;/* Scope of y starts here */

         x = p; /*Error, p is not in scope */
         y = q; /*Error, q is not in scope */

      }/* Scope of x and y ends here */

      a = x; /*Error, x is not in scope */
      b = p; /* Error, p is not in scope */

}/*Scope for a,b,c, and param ends here */
```

Notes about variables in block scope: (local variables)

- Variables that are not initialized by the programmer have undefined values.
- Variables declared within a block have *automatic storage*.
- This means that the memory required to hold the variable is allocated when the declaration/definition statement is encountered and is released (deallocated) when the variable goes out of scope (i.e. at the end of its block).
- Since memory is being allocated automatically for local variables, the declarations are also *definitions*.
- You cannot define a variable more than once (same name) in a block scope, or any scope started inside of that scope, the first block scope is the enclosing scope.

```
public static void f3(int param) /* Scope of param starts here */
{
 int a;    /* Scope of a starts here                  */
 int param; /* Error: param already defined in this scope */
 int a;    /* Error: a already defined in this scope    */

 while (a < 10)
 {
    int a;    /* Illegal, overwrites a from enclosing scope*/
    int param; /* Illegal, overwrites pram from enclosing scope */
    int a;    /* Illegal, a already defined in this scope, and  enclosing scope */

    if (a == 2)
    {
       int a;    /* Illegal, overwrites a from enclosing scope*/
       int param; /* Illegal, overwrites pram from enclosing scope */
    }

 }

 for(int i = 0; i < 10; ++i)
 {
    a += i;
 }/* Scope of i ends here */

 int i = 0;/* Fine, the scope for the i in for loop was that loop*/

} /* Scope of a and param ends here */
```

Function Overloading

Remember back to **System.out.println()**, we could pass any variable type and it would output its value to the console. How could this work?
From what we have seen we define a function that takes a certain set of parameters and only that set of parameters. However what happened if we had more than one version of the function differing by what values they take?

Example:

```
public class Main
{
	public static void printSquare(int i)
	{
		System.out.println("Squaring an int: " + i);
		System.out.println(i * i);
	}

	public static void printSquare(double d)
	{
		System.out.println("Squaring a double: " + d);
		System.out.println(d * d);
	}

	public static void main(String[] args)
	{
		printSquare(2);
		printSquare(3.0);
	}
}
```

In this example two methods were defined both named **printSquare()**, however one takes an int and the other a double. This is possible because of a concept known as overloading. But first let's look at the output of the program:

Output:

```
Squaring an int: 2
4
Squaring a double: 3.0
9.0
```

We can see that indeed the int version was called when we passed an integer (1) and the double version was called when we passed a double(3.0). How is this possible?

Both functions share the same name however they do not have the same function signature.
A function signature is made of 3 parts, the return type, the function name, and the parameters. The access modifier is not part of the function signature.

The fact that the two functions differ in the type of parameters they take allows the compiler to tell them apart. Of course, you cannot have two methods that are named the same, with the same number and type of parameters.

Note: The return type does not give the compiler enough information to differentiate between functions, that is why the following would be an error:

Example:

```
public class Main
{
        public static void square(int i )
        {
            i * i;
        }

        public static int square(int i)
        {
            return i * i;
        }

        public static void main(String[] args)
        {
            square(2);    /*Which Version?*/
        }
}
```

Compiler Error:

```
Exception in thread "main" java.lang.Error: Unresolved compilation problems:
        Duplicate method square(int) in type Main
        Syntax error on token "*", invalid AssignmentOperator

        at Main.square(Main.java:4)
        at Main.main(Main.java:16)
```