# 07 Methods in Self-contained Programs



## What is a method?

Methods, sometimes called 'procedures' or 'functions' in other programming languages, are programmer-defined names (identifiers) for reusable blocks of code. By giving a name to a sequence of instructions—that is, by defining a method—we can reuse that functionality without having to actually repeat those instructions in our program. And by defining a method so that it has parameters (requires input values when called) we can make its behaviour customisable, giving us flexible reuse of code.

This section of notes deals with the creation of methods within a standalone program, like those you have been creating up until now. In a few week's time you will define methods that belong to new types of object when you learn how to define your own data types. The syntax is only a little different in the two cases.

**Tip:** If you find yourself repeating the same sequence of statements in multiple parts of your program the consider defining a method that contains them. Then you can write just one statement to call that method in place you had repeated code.

## Pattern for defining a method

You have already been defining a single method in each program you write: main(), which is the *entry point* for any Java program, and which has the form:

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

The methods you define will have a similar form. In general, the components of a method declaration in a self-contained program are:

```
/**
 * commenta describing the method.
 */
public static [return_type] [identifier] ([parameter_list]) {
    [local variable declarations]
    [code to do the work]
    [return statement (if return_type is not void)]
}
```

where you have *control over* the components in boxes; the comment, the *return type*, the identifier (name of the method), the list of parameters, and what code you place in the method's body. Later you will see that you also have control over the keywords `public` `static` in the method header, but for the time being treat those as fixed.

### Common variations on that pattern

The key parts of the pattern above are: does the method need *'external' input (from another part of the program, not necessarily the user);* and does the method calculate a value or just do something?

| Does it need additional information to achieve its goal? | No | Yes |
|---|---|---|
| No | (a parameterless method)<br>`public static void someName() {`<br>`  //Statements`<br>`}` | (a method)<br>`public static void someName(int param) {`<br>`  //Statements that use param`<br>`  //The type, number and names of parameters will vary`<br>`}` |
| Yes | (a parameterless 'function')<br>`public static int someName() {`<br>`  //Perform type will vary`<br>`  //Statements that calculate some value`<br>`  return x; //x could be a variable or expression`<br>`}` | (a 'function')<br>`public static int someName(int param) {`<br>`  //Perform type will vary`<br>`  //Statements that use param1 and param2 and that calculate some value`<br>`  //The type, number and names of parameters will vary`<br>`  return x; //x could be a variable or expression`<br>`}` |

**Tip:** You don't need to remember these four alternatives, they are just natural outcomes of the decisions you make then defining a method.

### A checklist for writing a method declaration

When you have decided that you want to group a set of statements into a reusable method as part of your program keep in mind the following general form:

```
/**
 * commenta describing the method.
 */
public static [return_type] [identifier] ([parameter_list]) {
    [local variable declarations]
    [code to do the work]
    [return statement (if return_type is not void)]
}
```

where the *parameter_list* is a comma-separated list of variable declarations: *type identifier, type identifier,* etc.

And then answer the following questions in order to decide what to put in each part you can modify:

1. What does the method achieve and how would you describe that in 1–2 short, English-language sentences?
2. What value, if anything, does the method return?
   - Can be `void` if it doesn't return anything, or a type name (primitive or class name) if it does return a value
3. What name (identifier) is suitable to describe the method's function or purpose?
   - Remember the convention is to start with a lower case letter and, if the name is made up of multiple words, start each subsequent word with an upper case letter, as in `drawInCircle` or `makeSlot`.
4. What external data does it need to do its work? That is, what parameters does it have?
5. And then what code should we write inside the method block to achieve its goals?

## Examples

The following three examples assume this starting point for a program, which greets the user enthusiastically, prompts for and reads the radius of a circle, calculates and displays the area of that circle (according to $\pi * radius^2$), and displays another enthusiastic greeting:

```
import java.util.Scanner;

/**
 * A program with some methods (eventually).
 * @author James Montgomery
 */
public class MethodProgram {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int r; //radius of a circle
        double area; //area of that circle

        System.out.println("Hello!");
        System.out.println("Hello!");

        System.out.print("Enter the circle's radius: ");
        r = sc.nextInt();

        area = Math.PI * r * r;
        System.out.println("The area of the circle is " + area);

        System.out.println("Hello!");
        System.out.println("Hello!");
    }

}
```

We're going to modify this main() by taking some of its code and putting in other methods (and then calling them from main() as needed).

### A parameterless method: happyGreeting

Since we use the greeting code twice it's a good candidate for being converted into a method. Working through the *steps from above*:

1. **What does the method achieve?** It prints *Hello!* twice, so a good description would be *Prints the greeting Hello! twice, over two lines.*
2. **What value, if anything, does the method return?** Nothing, it just performs an action, so its return type is `void`.
3. **What name (identifier) is suitable?** One option would be *happyGreeting* (although you would be equally correct to name it *annoyingGreeting*).
4. **What external data does it need to do its work?** Nothing, since it knows what message to display already.
5. **And then what code should we write inside?** The two `println()` statements.

This leads us to the following modified program (given in full so the changes to main() are also visible):

```
import java.util.Scanner;

/**
 * A program with some methods.
 * @author James Montgomery
 */
public class MethodProgram {

    /** Prints the greeting Hello! twice, over two lines. */
    public static void happyGreeting() {
        System.out.println("Hello!");
        System.out.println("Hello!");
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int r; //radius of a circle
        double area; //area of that circle

        happyGreeting();

        System.out.print("Enter the circle's radius: ");
        r = sc.nextInt();

        area = Math.PI * r * r;
        System.out.println("The area of the circle is " + area);

        happyGreeting();
    }

}
```

### A parameterised method: happyGreeting (revised)

Saying *Hello!* to the user at the end of the program is a bit strange. If we consider the important functionality of `happyGreeting` to be that it says the same *thing twice* then we can revise its implementation to make it more flexible. Working through the *steps from above* again:

1. **What does the method achieve?** It prints some message twice, so a good description would be *Prints the greeting twice, over two lines.*
2. **What value, if anything, does the method return?** Still nothing, it just performs an action, so its return type is `void`.
3. **What name (identifier) is suitable?** Its name can remain *happyGreeting*, although others would also be suitable, such as *doublePrint*ln.
4. **What external data does it need to do its work?** It needs to know what String to display, so its parameter list (of one item) should be `String greeting` (where `greeting` could be any valid identifier).
5. **And then what code should we write inside?** The two `println()` statements will now print the value of the `greeting` parameter.

This leads us to the following modified program:

```
import java.util.Scanner;

/**
 * A program with some methods.
 * @author James Montgomery
 */
public class MethodProgram {

    /** Prints the greeting twice, over two lines. */
    public static void happyGreeting(String greeting) {
        System.out.println(greeting);
        System.out.println(greeting);
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int r; //radius of a circle
        double area; //area of that circle

        happyGreeting("Hello!");

        System.out.print("Enter the circle's radius: ");
        r = sc.nextInt();

        area = Math.PI * r * r;
        System.out.println("The area of the circle is " + area);

        happyGreeting("Goodbye!");
    }

}
```

### A 'function': circleArea

Calculating the area of a circle is something we might need to do more than once (even if we only do it once in this program), so it's also suitable for turning into a method. Working through the *steps from above*:

1. **What does the method achieve?** It calculates (and returns) the area of a circle, given its radius, so a suitable description would be *Calculates the area of a circle from its radius.*
2. **What value, if anything, does the method return?** Because the formula for a circle's area includes the irrational number π the result will be a real number, so make it `double`.
3. **What name (identifier) is suitable?** *circleArea* is short but meaningful.
4. **What external data does it need to do its work?** It needs to know the radius of the circle which, in our original program, was an integer, so its *parameter list* could be `int radius`.
5. **And then what code should we write inside?** While this method *could* be done in one line of code, we'll spread it out so that each step is clear (see below). The last line of the method will `return` the calculated result.

So the completed program is now:

```
import java.util.Scanner;

/**
 * A program with some methods.
 * @author James Montgomery
 */
public class MethodProgram {

    /** Prints the greeting twice, over two lines. */
    public static void happyGreeting(String greeting) {
        System.out.println(greeting);
        System.out.println(greeting);
    }

    /** Calculates the area of a circle from its radius. */
    public static double circleArea(int radius) {
        double area; //calculated area

        area = Math.PI * radius * radius;

        return area;
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int r; //radius of a circle

        happyGreeting("Hello!");

        System.out.print("Enter the circle's radius: ");
        r = sc.nextInt();

        System.out.println("The area of the circle is " + circleArea(r));

        happyGreeting("Goodbye!");
    }

}
```

**Note:** There are number of changes to main() above. We've moved the `area` variable to `circleArea` and, instead of storing it inside main() we immediately use the returned value as part of the println() statement. If we were going to use the value of the circle's area more than once then we would still declare the `area` variable inside main() and store the result of the method call.

## Parameters versus arguments

A **parameter**, also called a *formal parameter*, is the variable declared in the method header. It specifies the type of the variable and gives it a name to be used by the code inside the method declaration. In the worked example above we defined `circleArea` as:

```
public static double circleArea(int radius)
```

where `radius` is a parameter, of type `int`.

An **argument**, also called the *'actual parameter'*, is the value passed to a method at the time the method is called (i.e., when its code is executed). The argument can be a value literal, like 28, or the name of a variable that is local to the caller *(the name does not need to match the name of the parameter as only the value is passed).* So in the call to `circleArea`:

```
circleArea(r)
```

*r* (which has type `int`) is *the argument,* and its value (whatever the user typed in) is copied into `circleArea`'s `radius` parameter.

## Flow of Control and Data in Method Calls

When a method call is reached, flow of control transfers to that method (i.e., the code inside that method's block is executed), *after* which control reverts back to the calling location.

When the call happens, the values of any arguments are copied in the parameters of that method. This is what happens in more detail when a method is called:

- memory locations are set aside for the parameters (defined in the method declaration)
- the values of the arguments are copied in these memory locations
  - Note: if the argument is an object, the *reference* is copied, not the object

Then when the method's code has finished executing the memory locations for the parameters (and any locally declared variables) are 'destroyed'.

### A word about scope

A variable's *scope* is the region of a program where it is visible. It is largely determined by the location of its declaration (i.e., which block of { } it is defined in). So variables declared inside main() are *not* accessible from inside other methods, like `happyGreeting` or `circleArea`. Similarly the `area` variable declared inside `circleArea` cannot be accessed from inside main().

### Method calls and stack memory

*(Not assessable, but interesting and useful to know)*

Recall that the memory available to your program is divided into the stack and the heap, the heap being where objects and their data are stored. The stack is where memory for methods' parameters and local data is allocated.

When a method is called, memory is allocated for its parameters and local variables *on top of* the stack. Each of these 'stack frames' also records which statement in the next to be executed. If a method calls another method then another 'stack frame' is added for that method's data, while the caller's stack frame remembers where it was up to in its code.

When a method reaches its end, its stack frame is removed from the top of the stack (which is why its data is deleted).

A good metaphor for the stack is the history feature of a web browser. Links are like method calls, in that they take the user forward to new pages while the browser remembers where the user has been previously. When a user is done in a page they can press Back and be taken back not only to the previous page but the same location on that page. Unlike a web browser though, there is no *forward* option; once the method ends the only way to return to it is to call it again.