

METHODS

Methods let you group a series of statements together to perform a specific task. If different parts of a script repeat the same task, you can reuse the method (rather than repeating the same set of statements.) Consider the following program, which draws two text boxes on the console:

```
1 public class DrawBoxes {
2     public static void main(String[] args) {
3         System.out.println("+-----+");
4         System.out.println("|         |");
5         System.out.println("|         |");
6         System.out.println("+-----+");
7         System.out.println();
8         System.out.println("+-----+");
9         System.out.println("|         |");
10        System.out.println("|         |");
11        System.out.println("+-----+");
12    }
13 }
```

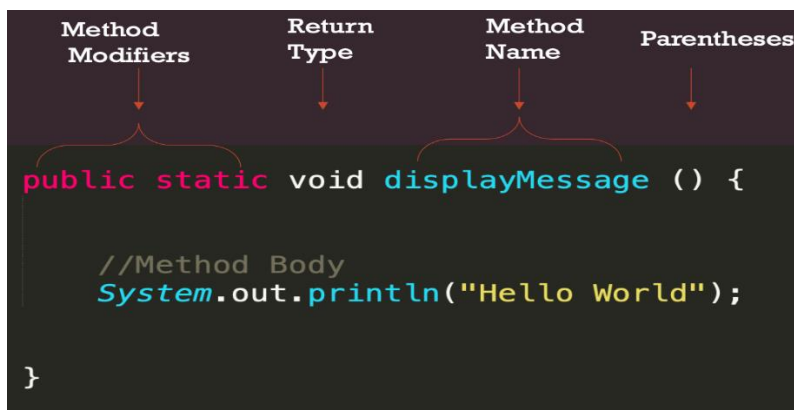
The program works correctly, but the four lines used to draw the box appear twice.

A preferable program would include a Java command that specifies how to draw the box and then executes that command twice. Java does not have a “draw a box” command, but you can create one. Such a named command is called a static method.

Declaring a Method

In the methods, the first line is known as the method header. You do not yet need to fully understand what each part of this header means in Java; for now, just remember that you will need to write `void`, followed by the `public static` name you wish to give the method, followed by a set of parentheses.

After the header in our sample method, a series of statements makes up the body of this static method. As in the method, the statements of this method are executed in order from first to last.



Method Modifiers: For now, every method we write will begin with `public static`.

Return Type: When the keyword `void` appears, it means that the method is a void method, and does not return a value. We will see value-returning methods later.

Method Name: Give each method a descriptive name. The same rules that apply to variable names also apply to method names. (check naming conventions)

Parentheses: The method name is always followed by a set of parentheses. Methods can be capable of receiving arguments. When this is the case, a list of one or more variable declarations will appear inside the parentheses. The method in this example does not receive any arguments, so the parentheses are empty.

Calling a Method

By defining the method `drawBox`, you give a simple name to this sequence of `println` statements. It is like saying to the Java compiler, "Whenever I tell you to 'drawBox,' I really mean that you should execute the `println` statements in the `drawBox` method." But the command will not actually be executed unless our method explicitly says that it wants to do so. The main act of executing a static method is called a method call.

```
2 public static void displayMessage(){
3     System.out.println("Hello");
4 }
//Code before hello...
1 displayMessage();
4 //Code after hello...
```

The methods can store the instructions for a specific task. When you need the script to perform that task, you call the method. The method executes the code in method code block. When it has finished, the code continues to run from the point where it was initially called.

To execute the `drawBox` command include this line in your program's main method:

`drawBox()`

Since we want to execute the `drawBox` command twice (to draw two boxes), the main method should contain two calls to the `drawBox` method.

```
1 public class DrawBoxes2 {
2     public static void main(String[] args) {
3         drawBox();
4         System.out.println();
5         drawBox();
6     }
7
8     public static void drawBox() {
9         System.out.println("+-----+");
10        System.out.println("|       |");
11        System.out.println("|       |");
12        System.out.println("+-----+");
13    }
14 }
```

Hierarchical Method Call

Method can also be called in a hierarchical, or layered fashion. In other words, method A can call method B, which can then call method C. When method C finishes, JVM returns to method B. When method B finishes, JVM returns to method A.

Passing Arguments to a Method

```
public static void displayValue(int num){
    System.out.println("The value is " + num);
}
```

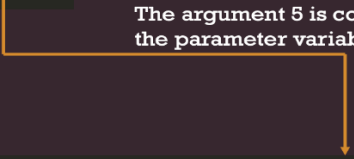
Sometimes a method needs specific information to perform its task. In such cases, when you declare the method, you give it parameters. Inside the method, the parameters act like variables. The parameter is a local variable, but it gets its initial value from the number call.

Calling Method That Need Information

```
displayValue(5);
```

The argument 5 is copied into the parameter variable num

```
public static void displayValue(int num){  
    System.out.println("The value is " + num);  
}
```



Argument and Parameter Data Type Compatibility

- When you pass an argument to a method, be sure that the argument's data type is compatible with the parameter variable's data type.
- Java will automatically perform a widening conversion if the argument's data type is ranked lower than the parameter variable's data type.
- Java will not automatically convert an argument to a lower-ranking data type. This means that a long, float, or double value cannot be passed a method that has an int parameter variable.
- Computer scientists use the word "parameter" broadly to mean both what appears in the method header (the formal parameter) and what appears in the method call (the actual parameter).

```
short s = 1;  
displayValue(s); //Converts short to int
```


```
byte b = 2;  
displayValue(b); //Converts byte to int
```

```
double d = 1.0;  
displayValue(d); //Error! Can not convert double to int
```

```
double d = 1.0;  
displayValue((int)d); // This will work
```

Passing Multiple Arguments

```
showSum(5,10);
```



```
public static void showSum(double num1, double num2){  
    double sum;  
    sum = num1 + num2;  
    System.out.println("The sum is " + sum);  
}
```

```
displayValue(int x); //Error  
displayValue(x); //Correct
```

```
public static void showSum(double num1, num2); //Error
```

```
public static void showSum(double num1, double num2); //Correct
```

When passing a variable as an argument, simply write the variable name inside the parentheses of the method call. Do not write the data type of the argument variable in the method call. Each parameter variable in a parameter list must have a data type listed before its name.

When you are writing methods that accept many parameters, the method header can become very long. It is common to wrap long lines (ones that exceed roughly 80 characters in length) by inserting a line break after an operator or parameter and indenting the line that follows by twice the normal indentation width:

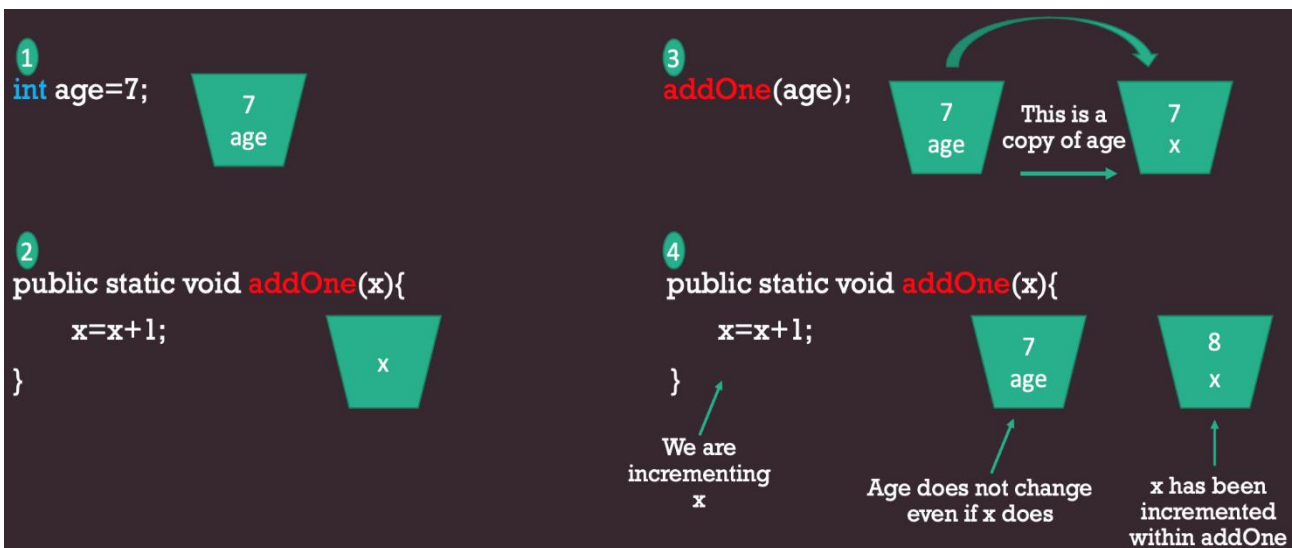
```
// this method's header is too long, so we'll wrap it
public static void printTriangle(int xCoord1, int yCoord1,
    int xCoord2, int yCoord2, int xCoord3, int yCoord3) {
    ...
}
```

Arguments are Passed by Value

In Java, all arguments of the primitive data types are passed by value, which means that only a copy of an argument's value is passed into a parameter variable.

A method's parameter variables are separate and distinct from the arguments that are listed inside the parentheses of a method call. If a parameter variable is changed inside a method, it has no effect on the original argument.

The local manipulations of the parameter do not change these variables outside the method. The fact that variables are copied is an important aspect of parameters. On the positive side, we know that the variables are protected from change because the parameters are copies of the originals. On the negative side, it means that although parameters will allow us to send values into a method, they will not allow us to get values back out of a method.



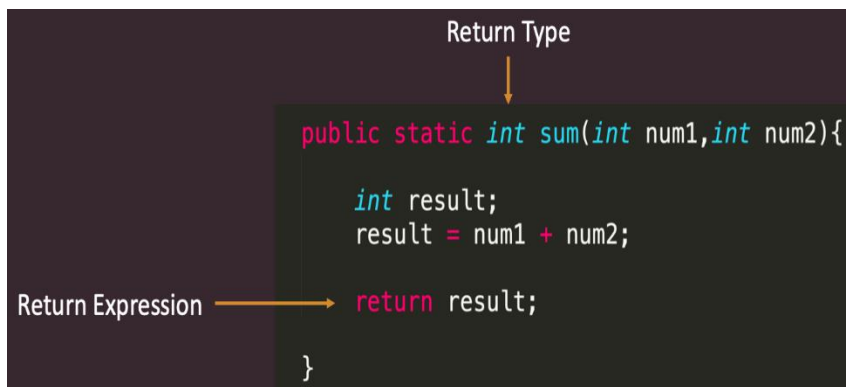
Local Variable

A local variable is declared inside a method and is not accessible to statements outside the method. Different methods can have local variables with the same names because the methods cannot see each other's local variables.

A method's local variables exist only while the method is executing. This is known as the lifetime of a local variable. When the method begins, its local variables and its parameter variables are created in memory, and when the method ends, the local variables and parameter variables are destroyed.

How to Return a Value from a Method

A method can return a value by using the return statement followed by an expression or a value.



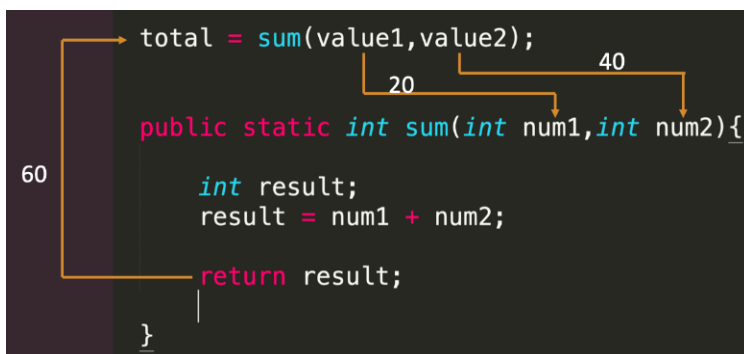
Java methods can return only one value. To return multiple values in Java, we should instead combine all the values we want to return in a compound object, whose instance variables include all the values we want to return, and then return a reference to that compound object. In addition,

we can change the internal state of an object that is passed to a method as another way of "returning" multiple results.

Notice once again that in the header for the method the familiar word `void` (indicating no return value) has been replaced with the word `int`. Remember that when you declare a method that returns a value, you must tell Java what kind of value it will return.

When Java encounters a statement, it evaluates the given expression `return` and immediately terminates the method, returning the value it obtained from the expression. As a result, it is not legal to have any other statements after `return` statement; the `return` must be the last statement in your method. It also returns an error for a `void` return type.

Calling a Value Returning Method



You will also want to be able to write methods that compute values. These methods are more like questions, as in "What is the square root of 2.5?" or "What do you get when you carry 2.3 to the 4th power?" Consider, for example, a method called `sqrt` that would compute the square root of a number.

It might seem that the way to write such a method would be to have it accept a parameter of type `double` and `println` its square root to the console. But you may want to use the square root as part of a larger expression or computation, such as solving a quadratic equation or computing the distance between points on an `x/y` plane.

A better solution would be a square root command that passes the number of interests as a parameter and returns its square root back to the program as a result. You could then use the result as part of an expression, store it in a variable, or print it to the console. Such a command is a new type of method that is said to return a value.

Common Programming Error - Ignoring the Returned Value

When you call a method that returns a value, the expectation is that you will do something with the value that is returned. You can print it, store it in a variable, or use it as part of a larger expression. It is legal (but unwise) to simply call the method and ignore the value being returned from it:

```
sum(1000); // doesn't do anything
```

The preceding call does not print the sum or have any noticeable effect. If you want the value printed, you must include a `println` statement:

```
int answer = sum(1000); // better
System.out.println("Sum up to 1000 is " + answer);
```

A shorter form of the fixed code would be the following:

```
System.out.println("Sum up to 1000 is " + sum(1000));
```

Common Programming Error - Statement after Return

It is illegal to place other statements immediately after a `return` statement because those statements can never be reached or executed. New programmers often accidentally do this when trying to print the value of a variable after returning. Say you have written the `hypotenuse` method but have accidentally written the parameters to `Math.pow` in the wrong order, so the method is not producing the right answer. You would try to debug this by printing the value of `c` that is being returned. Here is the faulty code:

```
// trying to find the bug in this buggy version of hypotenuse
public static double hypotenuse(double a, double b) {
    double c = Math.sqrt(Math.pow(2, a) + Math.pow(2, b));
    return c;
    System.out.println(c); // this doesn't work
}
```

The compiler complains about the `println` statement being unreachable since it follows a `return` statement. The compiler error output looks something like this:

```
Triangles.java:10: error: unreachable statement
    System.out.println(c);
    ^
Triangles.java:11: error: missing return statement
    }
    ^
2 errors
```

Reasoning about Paths

The combination of `if/else` and `return` is powerful. It allows you to solve many complex problems in the form of a method that accepts some input and computes a result. But you must be careful to think about the different paths that exist in the code that you write. At first this process might seem annoying, but when you get the hang of it, you will find that it allows you to simplify your code.

For example, suppose that we want to convert scores on the SAT into a rating to be used for college admission. Each of the three components of the SAT ranges from 200 to 800, so the overall total ranges from 600 to 2400. Suppose that a hypothetical college breaks up this range into three subranges with totals below 1200 considered not competitive, scores of at least 1200 but less than 1800 considered competitive, and scores of 1800 to 2400 considered highly competitive.

Let us write a method called `rating` that will take the total SAT score as a parameter and will return a string with the appropriate text. We can use the AND operator described earlier to write an if/else construct that has tests for each of these ranges:

```
public static String rating(int totalSAT) {
    if (totalSAT >= 600 && totalSAT < 1200) {
        return "not competitive";
    } else if (totalSAT >= 1200 && totalSAT < 1800) {
        return "competitive";
    } else if (totalSAT >= 1800 && totalSAT <= 2400) {
        return "highly competitive";
    }
}
```

This method has been written in a logical manner with specific tests for each of the three cases, but it does not compile. The compiler indicates at the end of the method that there was a “missing return statement.” That seems odd because there are three different return statements in this method. We have included a return for each of the different cases, so why is there a compiler error?

When the compiler encounters a method that is supposed to return a value, it computes every possible path through the method and makes sure that each path ends with a call on return. The method we have written has four paths through it. If the first test succeeds, then the method returns "not competitive". Otherwise, if the second test succeeds, then the method returns "competitive". If both of those tests fail but the third test succeeds, then the method returns "highly competitive".

But what if all three tests fail? That case would constitute a fourth path that does not have a return statement associated with it. Instead, we would reach the end of the method without having returned a value. That is not acceptable, which is why the compiler produces an error message.

It seems annoying that we must deal with a fourth case because we know that the total SAT score will always be in the range of 600 to 2400. Our code covers all the cases that we expect for this method, but that isn't good enough. **Java insists that we cover every possible case.**

Understanding this idea can simplify the code you write. If you think in terms of paths and cases, you can often eliminate unnecessary code. For our method, if we really want to return just one of three different values, then we do not need a third test. We can make the final branch of the nested if/else be a simple else:

```
public static String rating(int totalSAT) {
    if (totalSAT >= 600 && totalSAT < 1200) {
        return "not competitive";
    } else if (totalSAT >= 1200 && totalSAT < 1800) {
        return "competitive";
    } else {           // totalSAT >= 1800
        return "highly competitive";
    }
}
```

This version of the method compiles and returns the appropriate string for each different case. We were able to eliminate the final test because we know that we want only three paths through the method. Once we have specified two of the paths, then everything else must be part of the third path.

We can carry this idea one step further. We have written a method that compiles and computes the right answer, but we can make it even simpler. Consider the first test, for example. Why should we test for the total being greater than or equal to 600? If we expect that it will always be in the range of 600 to 2400, then we can simply test whether the total is less than 1200. Similarly, to test for the highly competitive range, we can simply test whether the score is at least 1800. Of the three ranges, these are the two simplest to test for. So, we can simplify this method even further by including tests for the first and third subranges and assume that all other totals are in the middle range:

```
public static String rating(int totalSAT) {
    if (totalSAT < 1200) {
        return "not competitive";
    } else if (totalSAT >= 1800) {
        return "highly competitive";
    } else { // 1200 <= totalSAT < 1800
        return "competitive";
    }
}
```

Whenever you write a method like this, you should think about the different cases and figure out which ones are the simplest to test for. This will allow you to avoid writing an explicit test for the most complex case. As in these examples, it is a good idea to include a comment on the final else branch to describe that case in English.

Before we leave this example, it is worth thinking about what happens when the method is passed an illegal SAT total. If it is passed a total less than 600, then it classifies it as not competitive and if it passed a total greater than 2400, it would classify it as highly competitive. Those are not bad answers for the program to give, but the right thing to do is to document the fact that there is a precondition on the total. In addition, we can add an extra test for this case and throw an exception if the precondition is violated. Testing for the illegal values is a case in which the logical OR is appropriate because illegal values will either be too low or too high (but not both):

```
// pre: 600 <= totalSAT <= 2400 (throws IllegalArgumentException if not)
public static String rating(int totalSAT) {
    if (totalSAT < 600 || totalSAT > 2400) {
        throw new IllegalArgumentException("total: " + totalSAT);
    } else if (totalSAT < 1200) {
        return "not competitive";
    } else if (totalSAT >= 1800) {
        return "highly competitive";
    } else { // 1200 <= totalSAT < 1800
        return "competitive";
    }
}
```

Method Overloading

- Method overloading is a feature that allows us to have more than one method with the same name, so long as we use different parameters.
- It is the ability to create multiple methods of the same name with different implementations.
- It improves the code readability and re-usability.
- It is easier to remember one method name instead of remembering multiple names.
- Overloaded methods give coder the flexibility to call similar method with different types of data.


```
public static int sumTwoNumbers(int a, int b) {
    return a + b;
}
```

BAD PRACTICE!

```
public static int sumThreeNumbers(int a, int b, int c) {
    return a + b + c;
}
```

```
public static int sumFourNumbers(int a, int b, int c, int d) {
    return a + b + c + d;
}
```

```
public static int sum(int a, int b) {
    return a + b;
}
```

GOOD PRACTICE!

```
public static int sum(int a, int b, int c) {
    return a + b + c;
}
```

```
public static int sum(int a, int b, int c, int d) {
    return a + b + c + d;
}
```

Rules:

1-Number of parameters

add(int,int)

add(int,int,int)

2-Data Type of parameters

add(int,int)

add(int,float)

3-Sequence of data type parameters

add(int,float)

add(float,int)

4-Invalid Case

int add(int,int)

float add(int,int)

5-Main Method

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

public static void main(String args){}

public static void main{}

DESIGNING METHODS

Procedural Design Heuristics

There are often many ways to divide (decompose) a problem into methods, but some sets of methods are better than others. Decomposition is often vague and challenging, especially for larger programs that have complex behavior. But the rewards are worth the effort because a well-designed program is more understandable and more modular. These features are important when programmers work together or when revisiting a program written earlier to add new behavior or modify existing code. There is no single perfect design, but in this section, we will discuss several heuristics (guiding principles) for effectively decomposing large programs into methods.

1. Each method should have a coherent set of responsibilities. In a company analogy, each group of employees must have a clear idea of what work he/she is to perform. If any of the groups does not have clear responsibilities, it is difficult for the company director to keep track of who is working on what task. When a new job comes in, two departments might both try to claim it, or a job might go unclaimed by any department. The analogous concept in programming is that each method should have a clear purpose and set of responsibilities. This characteristic of computer programs is called cohesion.

A desirable quality in which the responsibilities of a method or process are closely related to each other. **A good rule of thumb is that you should be able to summarize each of your methods in a single sentence such as “The purpose of this method is to”** Writing a sentence like this is a good way to develop a comment for a method's header. **It is a bad sign when you have trouble describing the method in a single sentence or when the sentence is long and uses the word “and” several times.** Those indications can mean that the method is too large, too small, or does not perform a cohesive set of tasks.

A subtler application of this first heuristic is that not every method must produce output. Sometimes a method is more reusable if it simply computes a complex result and returns it rather than printing the result that was computed. This format leaves the caller free to choose whether to print the result or to use it to perform further computations.

2. No one method should do too large a share of the overall task. One subdivision of a company cannot be expected to design and build the entire product line for the year. This system would overwork that subdivision and would leave the other divisions without enough work to do. It would also make it difficult for the subdivisions to communicate effectively, since so much important information and responsibility would be concentrated among so few people.

Similarly, one method should not be expected to comprise the bulk of a program. **This principle follows naturally from our first heuristic regarding cohesion, because a method that does too much cannot be cohesive. We sometimes refer to methods like these as “do-everything” methods because they do nearly everything involved in solving the problem.** You may have written a “do-everything” method if one of your methods is much longer than the others, hoards most of the variables and data, or contains most of the logic and loops.

3. Coupling and dependencies between methods should be minimized. A company is more productive if each of its subdivisions can largely operate independently when completing small work tasks. Subdivisions of the company do need to communicate and depend on each other, but such communication comes at a cost. Interdepartmental interactions are often

minimized and kept to meetings at specific times and places. When we are programming, we try to avoid methods that have tight coupling.

Coupling means an undesirable state in which two methods or processes rigidly depend on each other. Methods are coupled if one cannot easily be called without the other. One way to determine how tightly coupled two methods are to look at the set of parameters one passes to the other. A method should accept a parameter only if that piece of data needs to be provided from outside and only if that data is necessary to complete the method's task. In other words, if a piece of data could be computed or gathered inside the method, or if the data is not used by the method, it should not be declared as a parameter to the method.

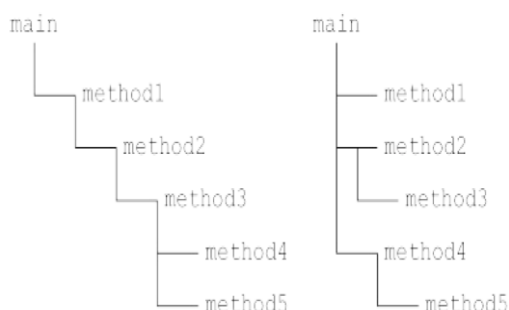
An important way to reduce coupling between methods is to use return statements to send information back to the caller. A method should return a result value if it computes something that may be useful to later parts of the program. Because it is desirable for methods to be cohesive and self-contained, it is often better for the program to return a result than to call further methods and pass the result as a parameter to them.

4. The main method should be a concise summary of the overall program. The top person in each major group or department of our hypothetical company reports to the group's director. If you look at the groups that are directly connected to the director at the top level of the company diagram, you can see a summary of the overall work: design, engineering, and marketing. This structure helps the director stay aware of what each group is doing. Looking at the top-level structure can also help the employees get a quick overview of the company's goals.

A program's main method is like the director in that it begins the overall task and executes the various subtasks. A main method should read as a summary of the overall program's behavior. Programmers can understand each other's code by looking at main to get a sense of what the program is doing.

A common mistake that prevents main from being a good program summary is the inclusion of a “do-everything” method. When the main method calls it, the do-everything method proceeds to do most or all the real work.

Another mistake is setting up a program in such a way that it suffers from chaining. Chaining means an undesirable design in which a “chain” of several methods calls each other without returning the overall flow of control to main.



A program suffers from chaining if the end of each method simply calls the next method. Chaining often occurs when a new programmer does not fully understand returns and tries to avoid using them by passing more and more parameters down to the rest of the program. Figure shows a hypothetical program with two designs. The flow of calls in a badly chained program might look like the diagram on the left.

One method should not call another simply as a way of moving on to the next task. A more desirable flow of control is to let main manage the overall execution of tasks in the program, as shown on the right side of Figure.

This guideline does not mean that it is always bad for one method to call another method; it is okay for one method to call another when the second is a subtask within the overall task of the first.

5. Data should be “owned” at the lowest level possible. Decisions in a company should be made at the lowest possible level in the organizational hierarchy. For example, a low-level administrator can decide how to perform his or her own work without needing to constantly consult a manager for approval. But the administrator does not have enough information or expertise to design the entire product line; this design task goes to a higher authority such as the manager. The key principle is that each work task should be given to the lowest person in the hierarchy who can correctly handle it.

This principle has two applications in computer programs.

The first is that the main method should avoid performing low-level tasks as much as possible. For example, in an interactive program main should not read most of the user input or contain lots of `println` statements.

The second application is that variables should be declared and initialized in the narrowest possible scope. A poor design is for main (or another high-level method) to read all the input, perform heavy computations, and then pass the resulting data as parameters to the various low-level methods. A better design uses low-level methods to read and process the data and return data to main only if they are needed by a later subtask in the program.

It is a sign of poor data ownership when the same parameter must be passed down several method calls. If you are passing the same parameter down several levels of calls, perhaps that piece of data should instead be read and initialized by one of the lower-level methods (unless it is a shared object such as a `Scanner`).

Readability

Here are some simple rules to follow that will make your programs more readable:

- Put **no more than one statement on each line**.
- **Indent your program properly.** When an opening brace appears, increase the indentation of the lines that follow it. When a closing brace appears, reduce the indentation. Indent statements inside curly braces by a consistent number of spaces (a common choice is **four spaces per level** of indentation).
- **Use blank lines to separate parts of the program** (e.g., methods).
- Well-written Java programs can be quite readable, but often you will want to include some explanations that are not part of the program itself. You can **annotate programs by putting notes called comments** in them.

Comments

Text that programmers include in a program to explain their code. The compiler ignores comments.

```
//Java program to show single line comments
class Comment
{
    public static void main(String args[])
    {
        // Single line comment here
        System.out.println("Single line comment above");
    }
}

//Java program to show multi line comments
class Scomment
{
    public static void main(String args[])
    {
        System.out.println("Multi line comments below");
        /*Comment line 1
        Comment line 2
        Comment line 3*/
    }
}
```

Block comments that begin with `/**` (note the second asterisk) have a special purpose, allowing a program, called **Javadoc**, to read these comments and automatically generate software documentation.

The only things you are not allowed to put inside a comment are the comment end characters.

It is a good idea to include comments at the beginning of each class file to indicate what the class does.

You should also comment each method to indicate what it does.

Commenting becomes more useful in larger and more complicated programs, as well as in programs that will be viewed or modified by more than one programmer. Clear comments are extremely helpful to explain to another person, or to yourself later, what your program is doing and why it is doing it.

Statement

An executable snippet of code that represents a complete command.

Each statement is terminated by a semicolon.

A Java program is stored in a class.

Within the class, there are methods. At a minimum, a complete program requires a special method called `main`.

Inside a method like `main`, there is a series of statements, each of which represents a single command for the computer to execute.

Procedural Decomposition

“Controlling complexity is the essence of computer programming.”

A separation into discernible parts, each of which is simpler than the whole.

Dividing up the overall action into a series of smaller actions. This technique is called **procedural decomposition**.