Introduction to Java

This simple, beginner-friendly Java course requires no previous coding knowledge. All you need is a mobile phone or desktop computer and 5 minutes a day! You'll learn all about the key concepts of Java, and will be writing clear, working code right from your first lesson.

Basic Concepts

Getting started with Java

Welcome to Java!

Java is one of the most popular programming languages.
Java's slogan is "Write once, run anywhere". Java programs can run on different platforms, including mobile, desktop, and other portable systems. You can use Java to build apps, games, banking applications, web apps, and much more!

Coding

Humans use computer programs to communicate with machines. Without computer programs, we wouldn't have smartphones, websites, or even exploration in outer space.

Learning some coding can help you innovate and create different solutions to problems, giving you a competitive edge in this technology-driven world.

What's a programming language?
A language used by machines to communicate with aliens
A language used by humans to communicate with machines

Output

Most computer programs are designed to produce outputs. Here are some examples:
- "You've got a new message" notifications
- "Game Over" displayed on the screen when playing video games
- Your account balance when checking your online banking app.

The simplest output consists of displaying a message on the screen.

Notifications and text displayed on a screen are examples of outputs from computer programs
False
True

Coders use outputs all the time to check that the computer is following the given instructions and fix problems with code.
The following line of code displays Java's slogan on the screen as an output:
System.out.println("Write once, run anywhere!");

The println instruction needs to be followed by parentheses.

Lesson Takeaways

Awesome! You completed your first lesson . Remember the following important points:
 You can write code that generates outputs with the System.out.println() statement
 The println instruction needs to be followed by parentheses

What's next?

In the next lesson, you will create code with multiple lines and different types of data.

Multiple Statements

More Complex Programs

Real computer programs can include thousands of lines of code.
In this lesson, you'll start building more complex programs.

Statement

A line of code is called a statement. A statement performs a specific task.
The output command is an example of a statement.
System.out.println("Write once, run anywhere!");

Each statement needs to end with a semicolon ;.

You can add as many statements (or lines of code) as you need.
The following piece of code consists of 2 statements. It outputs two messages in different lines.

```
public class Program
{
   public static void main(String[] args) {
      System.out.println("Name:");
      System.out.println("Surname:");
   }
}
```

OUTPUT
Name:
Surname:

Remember your code will result in an error if you forget the semicolons ; at the end of the statements.

When you give the computer many statements, the instructions will be executed line by line, from top to bottom.

Text vs Numbers

Computers treat text and numbers differently. When printing text outputs, you need to enclose the text in double quotes. You don't need quotes when outputting numbers.

```
public class Program{
   public static void main(String[] args){
      System.out.println("Points:");
      System.out.println(500);
   }
}
```

OUTPUT
Points:
500

Your code will result in an error if you forget the quotes around the text.

Java is a case-sensitive language. This means that you need to pay attention to the correct input of uppercase and lowercase letters.

Lesson Takeaways

Great work ! You completed the lesson. You learned that:
 You can add multiple statements to your programs
 Text needs to be enclosed in quotes
 Java is a case-sensitive language.
In the next lesson, you'll learn about the structure of programs in Java.

Program Structure

Java Program

In this lesson we will break down the structure of Java programs and understand how they work.
The whole code to create a valid Java program that outputs a simple text is the following:

```
public class Demo{
   public static void main(String[] args){
      System.out.println("Hi there");
   }
}
```

OUTPUT
Hi there

We are already familiar with the println method and the statement used to create output, so let's learn what the other parts of the code are.

The first line of the code defines a class called Demo.

```
class Demo {
```

In Java, every line of code that can actually run needs to be inside a class.
You can call the class anything you want.

We will learn more about classes in more advanced modules. For now, just remember, that any Java program needs to be inside a class.

The class opens and closes using curly brackets, like this:

```
class Demo {

}
```

Any code that we want to include in the class needs to go inside the brackets.

The opening bracket can also be written below the class name, but it's generally common to write it on the same line to save space.

Our program includes one more thing that we need to cover:

As you can see, the Demo class includes the following line:

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

In Java, each application has an entry point, or a starting point, which is a method called main.

We will cover public, static and void keywords definition in later lessons, when learning about methods. For now, remember that the main method needs to be declared identically to the code above.

The starting point of every Java program is:
the main method
the first line of the code

Notice that the main method, similar to the class, opens and closes with curly brackets.

```
public static void main(String[] args) {
  System.out.println("Hi there");
}
```

The main method contains the code that executes when we run our program.
In this case, the println method will be executed, when we run our program.

The main method can contain multiple statements, for example:

```
public class Demo{
   public static void main(String[] args){
      System.out.println("Welcome");
      System.out.println("This is a Demo");
      System.out.println("Bye");
   }
}
```

OUTPUT
```
Welcome
This is a Demo
Bye
```

This is now a fully functioning Java program.

Lesson Takeaways

Great job □! You now know how to create a valid Java program structure!
Remember the following important points:
□ You need to start your program by creating a class.
□ The class needs to include a main method, which is the starting point of the program.
□ The main method includes the statements that need to be executed when the program runs.
□ The class, as well as the main method opens and closes using opening and closing curly brackets.

Variables

Every program works with values.
A variable lets you store a value by assigning it to a name. The name can be used to refer to the value later in the program.

For example, in game development, you would use a variable to store how many points the player has scored.

Every variable has a type, which defines the type of the value it holds.
A variable can hold a text value, a number, a decimal, etc.
We are already familiar with text values - they are created using quotes:
"this is some text"

Text in quotes is called a String.

Let's create a variable of type String:

```
String name;
```

This creates a variable called name of type String.
Now, our variable name can hold String values.

In programming terms, the process of creating a variable is called declaration.

After declaring our variable, we can assign it a value using the assignment = operator:

```
String name;
name = "James";
```

Now, name holds the value "James".

Note, that the type String should start with a capital letter S. A lowercase version will cause an error.

We can use our variable in our program.
For example, let's output its value using println():

```
public class Demo{
   public static void main(String[] args){
      String name;
      name = "James";
      System.out.println(name);
   }
}
```

OUTPUT
James

We can combine the declaration and assignment into one statement, like this:

```
public class Demo{
   public static void main(String[] args){
      String name = "James";
      System.out.println(name);
   }
}
```

OUTPUT
James

This is handy when we already know the value for our variable and makes the code shorter and more readable.

A variable can change its value during the program multiple times.
For example, the player of a game can change his name:

```
public class Demo{
   public static void main(String[] args){
      String name = "James";
      name = "David";
      System.out.println(name);
   }
}
```

OUTPUT
David

Lesson Takeaways

Awesome! Here are some key takeaways:
- A variable has a name and a type of the value it holds.
- To declare a variable use the type followed by the name of the variable.
- You can assign a value to the declared variable using the = operator.
- A variable can change its value during the program, by being assigned to a new value.

We will learn about more variable types in the next lesson!

Variable Types

There are other types that you can use for variables.
The int type is used to store whole numbers (or integers, as we call them in programming).

```
public class Demo{
   public static void main(String[] args){
      int age = 42;
      System.out.println(age);
   }
}
```

OUTPUT
42

Now, the age variable of type int holds the value 42.

Decimals

To work with decimal numbers, use the type double:

```
public class Demo{
   public static void main(String[] args){
      double weight = 12.5;
      System.out.println(weight);
   }
}
```

OUTPUT
12.5

Java has another type for decimals called float.
When using the float type, you need to use an f postfix after the value:

```
public class Demo{
   public static void main(String[] args){
      float height = 1.94f;
      System.out.println(height);
   }
}
```

OUTPUT
1.94

This tells Java to use the value as a float, instead of double.

float vs double

By default, decimal values are of type double.
float is using less storage in the memory, but is not as precise as the double type.
This means that the calculations that use floats are faster than the ones that use double, however, the result is less accurate in terms of the decimal digits.

As a general rule: use float instead of double when memory usage is critical. If you need more precise computations, for example, when dealing with currency, use double.

Characters

The char type is used to hold a single character.
It is created similar to Strings, however it uses single quotes for the value:

```
public class Demo{
   public static void main(String[] args){
      char letter = 'B';
      System.out.println(letter);
   }
}
```

OUTPUT
B

A char value must be enclosed in:
commas
parentheses
double quotes
single quotes

Boolean

Another important type is boolean.
It can hold only the values true or false.
This is handy when we work with states or conditions, for example:

```
public class Demo{
   public static void main(String[] args){
      boolean isOpen = false;
      System.out.println(isOpen);
   }
}
```

OUTPUT
false

For example, the boolean above can show whether a shop is open or closed.

Lesson Takeaways

Great job! Here are some key takeaways:
- int holds integers (whole numbers).
- double holds decimal numbers.
- float is similar to double, but has less precision and requires less memory.
- You need to use an f postfix after the value to make it a float (for example: 3.14f)
- char holds a single character.
- boolean can have one of the following values: true or false.

We will learn how to make calculations in the next lesson.

Doing Math

You can use common math operators to perform calculations.
For example:

```
public class Demo{
   public static void main(String[] args){
      int x = 6;
      int y = 3;
      System.out.println(x + y);
   }
}
```

OUTPUT
9

This will output the sum of the two variables.

The result can be assigned to another variable, like this:

```
public class Demo{
   public static void main(String[] args){
      double price1 = 24.99;
      double price2 = 19.45;
      System.out.println(price1 + price2);
   }
}
```

OUTPUT
44.44

Similarly, you can use - for subtraction:

```
public class Demo{
   public static void main(String[] args){
      int salary = 90000;
      int tax = 15000;
      int result = salary - tax;
      System.out.println(result);
   }
}
```

OUTPUT
75000

Multiplication

The * operator multiplies two values.
Let's use it to find how much is 20% of the given value:

```
public class Demo{
   public static void main(String[] args){
      int price = 1200;
      System.out.println(price * 0.2);
   }
}
```

OUTPUT
240.0

0.2 represents 20%

Division

The / operator divides one value by another.
The following program calculates how many products with the given price you can buy with the given amount:

```
public class Demo{
   public static void main(String[] args){
      int amount = 9000;
      int price = 49;
      int result = amount / price;
      System.out.println(result);
   }
}
```

OUTPUT
183

In the example above, the result of the division equation will be a whole number, as int is used as the data type.

You can use double to get the result of the division with a decimal point.

```
public class Demo{
   public static void main(String[] args){
      double amount = 9000;
      double price = 49;
      double result = amount / price;
      System.out.println(result);
   }
}
```

OUTPUT
183.6734693877551

The result of dividing two doubles is:
an int
a double
zero

Modulo

The modulo % (or remainder) math operation returns the remainder of a division.
For example, let's use it to find how many of the given items will be left over if we place them in boxes of 5:

```
public class Demo{
   public static void main(String[] args){
      int items = 23;
      int res = items % 5;
      System.out.println(res);
   }
}
```

OUTPUT
3

Strings

Java allows to add together strings, using the + operator. The result is the combination of the strings.
For example, let's combine the first and last names to result in the full name of a user:

```
public class Demo{
   public static void main(String[] args){
      String firstname = "James";
      String lastname = "Smith";
      String fullname = firstname + lastname;
      System.out.println(fullname);
   }
}
```

OUTPUT
JamesSmith

Note, that the result will combine the given strings without any separator. We can add a space " " between them:

```
public class Demo{
   public static void main(String[] args){
      String firstname = "James";
      String lastname = "Smith";
      String fullname = firstname + " " + lastname;
      System.out.println(fullname);
   }
}
```

OUTPUT
James Smith

The process of adding strings together is called concatenation.
You can also concatenate strings with other types, such as integers and doubles.

Lesson Takeaways

Math is fun! Here are some key takeaways:
You can use basic math operators to perform calculations with values and variables.
+ is addition
- is subtraction
/ is division
% finds the remainder of a division.
You can add Strings using the + operator, in a process called concatenation.

Java Comments

Comments

One more thing to learn before wrapping up the first module!
Comments are annotations in the code that explain what the code is doing.
Code is for computers, while comments are for humans who read and work with the code.

A single-line comment starts with two forward slashes and continues until it reaches the end of the line. For example:

```
public class Demo{
   public static void main(String[] args){
      // storing the age of the user
      int age = 23; // this is just a demo value
      System.out.println(age);
   }
}
```

OUTPUT
23

Adding comments as you write code is a good practice, because they provide clarification and understanding when you need to refer back to it, as well as for others who might need to read it.

You can also comment out lines of code, in case they are work-in-progress or you don't want to delete it yet:

```
public class Demo{
   public static void main(String[] args){
      int age = 23;
      //int height = 122;
      System.out.println(age);
      //System.out.println(height);
   }
}
```

OUTPUT
23

The commented lines of code will get ignored when you run the program.

Multi-Line Comments

If you need to comment out multiple lines, or write a multi-line comment, you can use the /* */ symbols, like this:

```
public class Demo{
   public static void main(String[] args){
      /*This is just a
      demo program
      that outputs a number*/
      int age = 23;
      System.out.println(age);
   }
}
```

OUTPUT
23

Anything between the /* and */ symbols becomes a comment.

You can also use multi-line comments to comment out multiple lines of code.

Lesson Takeaways

Now you know how to add comments to your code! Here are some key takeaways:
Comments are explanatory statements that explain what the code is doing.
They can contain notes, todos as well as code that is work-in-progress.
// starts a single line comment.
/* */ is used for multi-line comments.

Control Flow

Taking User Input

Input

Some programs need user input. For example, a game may ask the user for a nickname and show it in the game, or a converter can ask for a value that you want to convert.

There are a number of ways to take input in Java.

To take input from the user, we first must import the corresponding class.
That is done using the following line:

```
import java.util.Scanner;
```

This line should be written at the very top of your code, before the class declaration.

This imports the Scanner class, which we will use for taking input.

After importing the Scanner class, we need to create a Scanner object:

Scanner sc = new Scanner(System.in);

Confused about the new terminology? Classes, objects, import... Don't worry, you will learn about these in more advanced lessons. For now, just remember the syntax for creating the Scanner object.

Now we are ready to take input from the user and assign it to a variable.
For example, to take a String input, we need to use the following:

```
import java.util.Scanner;
public class Demo{
   public static void main(String[] args){
      Scanner sc = new Scanner(System.in);
      String name = sc.nextLine();
      System.out.println("Name:-" + name);
      sc.close();
   }
}
```

OUTPUT
Thirueswaran V
Name:-Thirueswaran V

Similarly, we can take a integer as input using nextInt():

```
import java.util.Scanner;
public class Demo{
   public static void main(String[] args){
      Scanner sc = new Scanner(System.in);
      int age = sc.nextInt();
      System.out.println("Age:-" + age);
      sc.close();
   }
}
```

OUTPUT
21
Age:-21

This will accept an integer input from the user and assign it to the age variable.

There are similar methods available to take other types as input: nextDouble(), nextFloat(), nextBoolean().

Multiple Inputs

You can use the same Scanner to take multiple inputs.

For example, let's take the name and age as input and output them

```
import java.util.Scanner;
public class Program{
   public static void main(String[] args){
      Scanner sc = new Scanner(System.in);
      String name = sc.nextLine();
      int age = sc.nextInt();
      System.out.println(name + " " + age);
      sc.close();
   }
}
```

OUTPUT
Thiru
21
Thiru 21

Lesson Takeaways

That's how you take input from the user in Java!
Here are the steps:
1. import the java.util.Scanner class.
2. create a Scanner object:
Scanner sc = new Scanner(System.in);
3. Use the corresponding method of the Scanner to take input, for example:

```
import java.util.Scanner;
public class Demo{
   public static void main(String[] args){
      Scanner sc = new Scanner(System.in);
      int num = sc.nextInt();
      System.out.println(num);
      sc.close();
   }
}
```

OUTPUT
8
8

Conditionals

Decision Making

Conditional statements are used to perform different actions based on different conditions.
For example, a billing program can apply a discount to the total only if the amount is greater than a threshold.

Let's learn how to create such programs.

The if statement is one of the most frequently used conditional statements.

If the if statement's condition is true, the block of code inside the if statement is executed.

Syntax:

```
if (condition) {
  //some code
}
```

if Statement

Any of the following comparison operators may be used to form the condition:
< less than
> greater than
!= not equal to
== equal to
<= less than or equal to
>= greater than or equal to

For example:

```
public class Demo{
   public static void main(String[] args){
      int age = 24;
      if(age >= 18){
         System.out.println("Welcome");
      }
   }
}
```

OUTPUT
Welcome

This code will output the message only if the age value is greater or equal to 18.

Remember that you need to use two equal signs (==) to test for equality, since a single equal sign is the assignment operator.

For example:

```
public class Demo{
   public static void main(String[] args){
      int number = 8;
      if(number == 8){
         System.out.println("Bingo");
      }
   }
}
```

OUTPUT
Bingo

else Statement

An if statement can be followed by an optional else statement, which executes when the condition evaluates to false.

For example:

```
public class Demo{
   public static void main(String[] args){
      int age = 30;
      if(age < 16){
         System.out.println("Too Young");
      }
      else{
         System.out.println("Welcome!");
      }
   }
}
```

OUTPUT
Welcome

As age equals 30, the condition in the if statement evaluates to false and the else statement is executed.

Nested if Statements

You can use one if-else statement inside another if or else statement.

For example:

```
public class Demo {
   public static void main(String[] args) {
      int age = 25;
      if (age > 0) {
         if (age > 16) {
            System.out.println("Welcome!");
         } else {
            System.out.println("Too Young");
         }
      } else {
         System.out.println("Error");
      }
   }
}
```

OUTPUT
Welcome!

You can nest as many if-else statements as you want, however the code will become harder to read and understand.

else if Statements

Instead of using nested if-else statements, you can use the else if statement to check multiple conditions.

For example:

```
public class Demo {
   public static void main(String[] args) {
      int age = 25;
      if (age <= 0) {
         System.out.println("Error");
      } else if (age <= 16) {
         System.out.println("Too Young");
      } else if (age < 100) {
         System.out.println("Welcome!");
      } else {
         System.out.println("Really?");
      }
   }
}
```

OUTPUT
Welcome!

You can include as many else if statements as you need.

Lesson Takeaways

Now you know how to make decisions in your code! Here are some key takeaways:
You can check for a condition using the if statement.
In case the condition is false, the code in an else statement can be executed.
Here is a generic structure of if-else statements:

```
if(condition) {
  //some code
} else if(condition) {
//some other code
} else {
  //some other code
}
```

We will learn about another decision making statement in the next lesson.

The switch Statement

Conditionals

Consider a program that takes a day number as input and outputs the corresponding weekday:

```
public class Demo {
   public static void main(String[] args) {
      int day = 2;
      if (day == 1) {
         System.out.println("Monday");
      } else if (day == 2) {
         System.out.println("Tuesday");
      } else if(day == 3){
         System.out.println("Wednesday");
      }
   }
}
```

OUTPUT
Tuesday

For a shorter code in our demo, we have checked only for the first 3 values.
You can continue the code and check for all 7 day numbers.

switch

Instead of many if else statements, which become hard to read, we can use a switch statement.
A switch statement tests a variable for equality against a list of values.
Here is the previous example using a switch statement:

```
public class Demo {
   public static void main(String[] args) {
      int day = 2;
      switch(day){
         case 1:
            System.out.println("Monday");
            break;
         case 2:
            System.out.println("Tuesday");
            break;
         case 3:
            System.out.println("Wednesday");
            break;
      }
   }
}
```

OUTPUT
Tuesday

Let's look at the code again:

When the variable being switched on is equal to a case, the statements following that case will execute until a break statement is reached.

You can have any number of case statements within a switch. Each case is followed by the comparison value and a colon.

It is important to have a break statement for each case.
If no break appears, the program will continue to execute the next case in the switch, even if the value does not match the variable that is switched on.
Run this example to see what happens when there is no break in the case:

```
public class Demo {
   public static void main(String[] args) {
      int day = 2;
      switch(day){
         case 1:
            System.out.println("Monday");
         case 2:
            System.out.println("Tuesday");
         case 3:
            System.out.println("Wednesday");
      }
   }
}
```

OUTPUT
Tuesday
Wednesday

The default Case

A switch statement can have an optional default case.
The default case can be used for performing a task when none of the cases is matched.

For example:

```
public class Demo {
   public static void main(String[] args) {
      int day = 5;
      switch(day){
         case 1:
            System.out.println("Monday");
         case 2:
            System.out.println("Tuesday");
         case 3:
            System.out.println("Wednesday");
         default:
            System.out.println("Another day");
      }
   }
}
```

OUTPUT
Another day

No break is needed in the default case, as it is always the last statement in the switch.

Lesson Takeaways

The switch statement is a handy way to check for multiple values and run code.
- Remember, that each case is followed by a value and a colon.
- Each case needs a break statement, or the code of the other cases will continue to get executed.
- The default case can be used to run code if none of the cases match.
In the next lesson, we will learn how to check for multiple conditions.

Multiple Conditions

In some cases we need to combine multiple conditions, for example, let's say we want to check if the age value is greater than 18 and less than 50.
This can be done using the && operator.

```
public class Demo {
   public static void main(String[] args) {
      int age = 42;
      if(age > 18 && age < 50){
         System.out.println("Welcome!");
      }
   }
}
```

OUTPUT
Welcome!

The && operator is also referred to as the logical AND operator.

The OR Operator

The OR operator (||) checks if any one of the conditions is true.

For example:

```
public class Demo {
   public static void main(String[] args) {
      int age = 25;
      int height = 100;
      if(age > 18 || height > 150){
         System.out.println("Welcome!");
      }
   }
}
```

OUTPUT
Welcome!

The code above will print Welcome! if age is greater than 18 or if height is greater than 150.

NOT

The NOT (!) logical operator is used to reverse the condition.
If a condition is true, the NOT logical operator will make it false, and vice versa.

Example:

```
public class Demo {
   public static void main(String[] args) {
      int age = 25;
      if (!(age > 18)) {
         System.out.println("Too Young");
      } else {
         System.out.println("Welcome");
      }
   }
}
```

OUTPUT
Welcome!

if !(age > 18) reads as "if age is NOT greater than 18".

You can chain multiple conditions using parentheses and the logical operators.

For example:

```
public class Demo {
   public static void main(String[] args) {
      String country = "US";
      int age = 42;
      if((country == "US" || country == "GB") && (age > 0 && age < 100)){
         System.out.println("Allowed");
      }
   }
}
```

OUTPUT
Allowed

Lesson Takeaways

Logical operators allow to combine multiple conditions.
- The AND operator &&combines two conditions and checks if both of them are true.
- The OR operator || check if any of the conditions if true.
- The NOT operator ! reverses the condition.

Next lesson will be fun! We will cover loops, which allow to repeat a block of code multiple times.

While Loops

Loops

Loops allow you to repeat a block of code multiple times.

For example, a banking app can loop over all bank transactions and check for some conditions.

A while loop statement repeatedly executes a target statement as long as a given condition is true.

Example:

```
public class Demo {
   public static void main(String[] args) {
      int x = 3;
      while(x > 0){
         System.out.println(x);
         x = x - 1;
      }
   }
}
```

OUTPUT

```
3
2
1
```

The while loops check for the condition x > 0. If it evaluates to true, it executes the statements within its body. Then it checks for the statement again and repeats.

The code above will output the numbers 3 to 1, and then stop, as the condition will become false, after x reaches 0.

Let's look at the code again:

The line x = x-1; is important, as without it the condition would never become false and the loop would run forever.

Each time the loop runs, 1 is subtracted from x.

As it is common to decrease or increase a variable by 1 during loops, Java provides increment and decrement operators.

These are a shorter way to increase or decrease the value of a variable by one.

For example, the statement x=x-1; can be simplified to x--;

```
public class Demo {
   public static void main(String[] args) {
      int x = 3;
      while(x > 0){
         System.out.println(x);
         x--;
      }
   }
}
```

OUTPUT

```
3
2
1
```

Similarly, the increment operator ++ is used to increase the value of a variable by one. Here is a loop that outputs the numbers 1 to 10:

```
public class Demo {
   public static void main(String[] args) {
      int x = 1;
      while(x <= 10){
         System.out.println(x);
         x++;
      }
   }
}
```

OUTPUT

```
1
2
3
4
5
6
7
8
9
10
```

Some loops require to increase or decrease the value of a variable by a different number.
For example, lets output only the even numbers from 0 to 10.

```
public class Demo {
   public static void main(String[] args) {
      int x = 0;
      while(x <= 10){
         System.out.println(x);
         x = x + 2;
      }
   }
}
```

OUTPUT
0
2
4
6
8
10

Surprise! There is also a shorter way for x = x+2; It can be written as x+=2;

Similarly, there are shorthand operators for other mathematical operations, such as -= for subtraction, *= for multiplication, etc.

You can perform calculations and other operations in loops.
For example, let's calculate the sum of the numbers from 1 to 100 and output it:

```
public class Demo {
   public static void main(String[] args) {
      int sum = 0;
      int num = 0;
      while(num <= 100){
         sum += num;
         num++;
      }
      System.out.println(sum);
   }
}
```

OUTPUT
5050

We add the value of num to sum each time the loop runs, and then increase the num value by 1.
At the end of the loop, sum holds the result of our calculation.

Notice that the last print statement is out of the while scope.

do-while Loops

Another variation of the while loop is do-while.

For example:

```
public class Demo {
   public static void main(String[] args) {
      int x = 1;
      do {
         System.out.println(x);
         x++;
      } while (x < 5);
   }
}
```

OUTPUT

```
1
2
3
4
```

Notice that the condition appears at the end of the loop, so the statements in the loop execute once before it is tested.
Even with a false condition, the code will run once.
Also, note the semicolon after the while condition.

The difference between while and do-while is that do-while is guaranteed to run at least once, even with a false condition.

Lesson Takeaways

Here is a summary about the while loop:
- The code in the while loop runs as long as the condition is true.
- The ++ and -- operators are used to increase and decrease the value of a variable by one.
- Java provides shorthand operators to perform mathematical operations on a variable, for example x = x * 9; can be written as x * = 9.
- The do-while loop is similar to a while loop, but it is guaranteed to run at least once.

We will learn about another loop type in the next lesson!

For Loops

Another kind of loop is the for loop.
It looks like this:

```
public class Demo {
   public static void main(String[] args) {
      for (int x = 1; x < 5; x++) {
         System.out.println(x);
      }
   }
}
```

OUTPUT

```
1
2
3
4
```

This will output the numbers 1 through 4.

The for loop has 3 components:

The first part runs once when we enter the loop and initializes the variable.
The second part is the condition of the loop.
The third part runs every time the loop runs.

Notice the semicolons (;) after the parts of the loop.

You can have any type of condition and any type of increment statements in the for loop.
The example below prints only the even values between 0 and 10:

```
public class Demo {
   public static void main(String[] args) {
      for (int x = 0; x < 10; x += 2) {
         System.out.println(x);
      }
   }
}
```

OUTPUT

```
0
2
4
6
8
```

The for loop is best when we know the number of times we need to run the loop.

Loop Control

Remember the break statement from the switch?
It can also be used to terminate loops.

Example:

```
public class Demo {
   public static void main(String[] args) {
      int x = 1;
      while (x < 10) {
         System.out.println(x);
         if (x == 4) {
            break;
         }
         x++;
      }
   }
}
```

OUTPUT

```
1
2
3
4
```

This will end the loop when x reaches the value 4.

break

It also works in the for loop:

```
public class Demo {
   public static void main(String[] args) {
      for (int x = 1; x < 10; x++) {
         System.out.println(x);
         if (x == 4) {
            break;
         }
      }
   }
}
```

OUTPUT

```
1
2
3
4
```

Here is one example use case of break:
For example, you are making a calculator and need to take numbers from the user to add together and stop when the user enters "stop".
In this case, the break statement can be used to end the loop when the user input equals "stop".

continue

Another control statement is continue.
It makes the loop skip the current iteration and go to the next one.

Example:

```
public class Demo {
   public static void main(String[] args) {
      for (int x = 10; x <= 40; x += 10) {
         if (x == 30) {
            break;
         }
         System.out.println(x);
      }
   }
}
```

OUTPUT
10
20

The above code skips the value of 30, as directed by the continue statement.

An example use case of continue:
An airline ticketing system needs to calculate the total cost for all tickets purchased. Tickets for children under the age of 3 are free. We can use a while loop to iterate through the list of passengers and calculate the total cost of their tickets. Here, the continue statement can be used to skip the children.

Lesson Takeaways

Great progress! Here is a summary:
- The for loop has the following syntax:

```
for(init; condition; increment) {
  //code
}
```

- The break statement can be used to stop a loop.
- The continue statement can be used to skip the current iteration of the loop and jump to the next one.

Arrays

Imagine a program that needs to store the ages of 10 users.
With variables, you would need to create 10 separate variables for each of the users. That would be really not effective and repetitive.
Arrays can help in these situations!
An array stores multiple values in a single variable.

An array needs to be declared, just like a variable, with the type of the items it will hold.
To declare an array, you need to define the type of the elements with square brackets.

```
int[ ] ages;
```

The name of the array is ages. The type of elements it will hold is int.

Now, to create the array, we need to specify the number of items it will hold using the new keyword:

```
int[ ] ages;
ages = new int[5];
```

The code above creates an array of 5 integers.
We can combine the above code into one line:

```
int[] ages = new int[5];
```

The items in an array are accessed using their position, also called their index.
Here is an example, setting the item with the index 2 to the value 25:

```
ages[2] = 25;
```

The index is specified in square brackets, next to the array name.

The item with index 2 is actually the 3rd item of the array.
That's because array indexes start from 0, meaning that the first element's index is 0 rather than one. So, the maximum index of the array int[5] is 4.
Let's set the first items value:

```
ages[0] = 18;
```

Similar to setting values, we can also access values of the array, using square brackets and the index of the item we want to access:

```
public class Demo {
   public static void main(String[] args) {
      int[] ages = new int[5];
      ages[0] = 18;
      ages[2] = 25;
      System.out.println(ages[2]);
   }
}
```

OUTPUT
25

This will output the value of the 3rd item.

If you already know what values to store in the array, instead of assigning them one by one, you can use the following syntax:

```
public class Demo {
   public static void main(String[] args) {
      String[] names = { "A", "B", "C", "D" };
      System.out.println(names[2]);
   }
}
```

Place the values in a comma-separated list, enclosed in curly braces.
The code above automatically creates an array containing 4 items, and stores the provided values.

Sometimes you might see the square brackets placed after the array name, which also works, but the preferred way is to place the brackets after the array's data type.

Lesson Takeaways

Awesome! Here are some key points about arrays:
Arrays allow you to store multiple values in a variable.
When creating an array, we need to provide the type of the items and the size of the array, like this:

```
int[] nums = new int[4];
```

Array items are accessed using their indexes, placed in square brackets. The first item has the index 0.
You can also create an array with values using the following syntax:

```
int[] nums = {4, 6, 2, 1};
```

In the next lesson we will learn how to loop over the values of an array and make calculations.

Looping Over Arrays

Let's learn how to output the items of an array using a loop.
To use a loop, we first need to find out how many items the array stores.
For that, the array has a length property, which is accessed like this:

```
public class Demo {
   public static void main(String[] args) {
      int[] ages = { 18, 33, 24, 64, 45 };
      System.out.println(ages.length);
   }
}
```

OUTPUT
5

This will output the number of items stored in the array.

Now, when we know the number of items, we can use a for loop and output all the items of the array:

```
public class Demo {
   public static void main(String[] args) {
      int[] ages = { 18, 33, 24, 64, 45 };
      for (int x = 0; x < ages.length; x++) {
         System.out.println(ages[x]);
      }
   }
}
```

OUTPUT
18
33
24
64
45

We used the x variable of the loop as the index for our array. During each iteration of the loop, the next item of the array is accessed.

We can also use a for loop to make calculations using array values. For example, let's calculate the sum of all values in an array:

```
public class Demo {
   public static void main(String[] args) {
      int[] ages = { 18, 33, 24, 64, 45 };
      int sum = 0;
      for (int x = 0; x < ages.length; x++) {
         sum += ages[x];
      }
      System.out.println(sum);
   }
}
```

OUTPUT
184

In the code above, we declared a variable sum to store the result and assigned it 0.
Then we used a for loop to iterate through the array, and added each item's value to the variable.

for-each Loop

Java provides another version of the for loop, called the for-each loop, to loop over arrays, making the code shorter and easier to read.

Here it is:

```
public class Demo {
   public static void main(String[] args) {
      int[] nums = { 2, 3, 5, 7 };
      for (int x : nums) {
         System.out.println(x);
      }
   }
}
```

OUTPUT
2
3
5
7

The loop goes through each item in the array and assigns the current item's value to the new variable during each iteration of the loop.
You can call the variable anything you want: we called it x in our example.

Notice the colon after the variable - it reads as "for each x in nums".

Let's use a for-each loop to calculate the sum of all values of an array:

```
public class Demo {
   public static void main(String[] args) {
      int[] ages = { 18, 33, 24, 64, 45 };
      int sum = 0;
      for (int x : ages) {
         sum += x;
      }
      System.out.println(sum);
   }
}
```

OUTPUT
184

Note, that in this case we do not have the index, we have the value of each item of the array.

Lesson Takeaways

Looping over arrays is fun!
You can use a for loop to loop over an array.
The length property is used to get the number of items of the array.

```
for(int x=0;x<arr.length; x++) {
//current item is arr[x]
}
```

Another way to loop over arrays is the for-each loop:

```
for(int x: arr) {
// current item is x
}
```

We will learn about multidimensional arrays in the next lesson!

Multidimensional Arrays

Arrays can have multiple dimensions (or number of indices).
For example, imagine a ticketing program that is storing seat numbers in a stadium, which have a row and column number.
Or a chess board, where each square has 2 coordinates: a letter and a number

The arrays in these examples have 2 dimensions.

To create multidimensional arrays, place each array within its own set of square brackets:

```
int[ ][ ] sample = { {1, 2, 3}, {4, 5, 6} };
```

Note that the array is created using two square brackets, specifying the two-dimensionality.

To access an element in the two-dimensional array, provide two indexes, one for the array, and another for the element inside that array.
The following example accesses the first element in the second array of sample:

```
public class Demo {
   public static void main(String[] args) {
      int[][] sample = { { 1, 2, 3 }, { 4, 5, 6 } };
      int x = sample[1][0];
      System.out.println(x);
   }
}
```

<u>OUTPUT</u>
4

The array's two indexes are called row index and column index.
Here is how we can visualize it:

```
public class Demo {
   public static void main(String[] args) {
      int[][] sample = {
         { 1, 2, 3 },
         { 4, 5, 6 }
      };
      int x = sample[1][0];
      System.out.println(x);
   }
}
```

OUTPUT

```
4
```

Each row is an item, which is an array. So, to access a value, we provide the row index, then the column index.

To loop over a 2-dimensional array, we need nested for loops:

```
public class Demo {
   public static void main(String[] args) {
      int sample[][] = {
            { 1, 2, 3 },
            { 4, 5, 6, }
      };
      for (int x = 0; x < sample.length; x++) {
         for (int y = 0; y < sample[x].length; y++) {
            System.out.println(sample[x][y]);
         }
      }
   }
}
```

OUTPUT

```
1
2
3
4
5
6
```

The first loop iterates over the rows, and the second one over their items.

Lesson Takeaways

Arrays with multiple dimensions are simply arrays that contain other arrays.
The number of square brackets match the dimension of the array, for example [][] denotes a 2-dimensional array.
To access the items of the array, specify the row index in the first square brackets, followed by the column index in the second.

Methods

A method is a block of code designed to perform a particular task.
For example, our app can have methods like login(), logout(), convert(), etc.
The println() that we use for output is also a method.

The purpose of a method is to create it once and call it multiple times when needed to perform particular tasks.

You can define your own methods to perform your desired tasks.
Here is an example:

```
static void welcome() {
  System.out.println("Welcome");
  System.out.println("I am a method");
  System.out.println("End of method");
}
```

The code above declares a method called welcome, which prints 3 lines of text.

Note that the name of the method is followed by parentheses (). The statements of the method are inside curly braces.

Let's look at the code again:

static is needed to be able to use the method in main. You will learn about the static keyword in more advanced lessons.
void means that this method does not have a return value. You will learn more about return values later in this module.
welcome is the name of the method.

Calling a Method

After defining the method, we can use it in our program by "calling" it.
To call a method, type its name followed by a set of parentheses.

For example:

```
public class Demo {
   static void welcome() {
      System.out.println("Welcome");
      System.out.println("I am a method");
      System.out.println("End of method");
   }
   public static void main(String[] args) {
      welcome();
   }
}
```

OUTPUT
```
Welcome
I am a method
End of method
```

You can call a method as many times as necessary.

Example:

```
public class Demo {
   static void welcome() {
      System.out.println("Welcome");
      System.out.println("I am a method");
      System.out.println("End of method");
   }
   public static void main(String[] args) {
      welcome();
      // some code
      welcome();
      welcome();
   }
}
```

OUTPUT
```
Welcome
I am a method
End of method
Welcome
I am a method
End of method
Welcome
I am a method
End of method
```

Lesson Takeaways

Great job! Here are the main points to remember when defining your own methods:
- Methods are reusable, we define them once and can call them multiple times.
- To call a method, use its name, followed by parentheses.
- The void keyword means that the method does not return a value.

We will learn about return values in the next lessons, so stay tuned!

Method Parameters

Methods can have parameters that can be used in their code.
Parameters are defined in parentheses and can act as variables in a method.
For example, let's add a String parameter called name to our welcome() function:

```
static void welcome(String name) {
  System.out.println("Welcome, " + name);
}
```

The above method takes a String called name as its parameter, which is used in the method.

Now, when calling the method, we need to pass it a value for the name parameter inside the parentheses:

```
public class Demo {
    static void welcome(String name) {
        System.out.println("Welcome, " + name);
    }
    public static void main(String[] args) {
        welcome("James");
        welcome("Amy");
    }
}
```

OUTPUT
Welcome, James
Welcome, Amy

This way, we can call our method with different parameters and generate different results based on them.

The values passed as parameters are called arguments.

Multiple Parameters

Methods can take multiple parameters. For that, we simply need to separate them using commas, for example:

```
static void welcome(String name, int age) {
  System.out.println("Welcome, "+name);
  System.out.println("Your age: "+age);
}
```

Now, our welcome() method takes a String and an integer as its parameters.

Now, when calling the function, we need to provide all the parameters:

```
public class Demo {
   static void welcome(String name, int age) {
      System.out.println("Welcome, " + name);
      System.out.println("Your age: " + age);
   }
   public static void main(String[] args) {
      welcome("James", 42);
      welcome("Amy", 25);
   }
}
```

OUTPUT
Welcome, James
Your age: 42
Welcome, Amy
Your age: 25

Note that the arguments need to match the parameters and must be passed in the same order.

Method parameters are really handy! They allow our method to work with different values and produce results.
For example, we can create a method to calculate a given percentage of a number and output it:

```
public class Demo {
   static void perc(double num, int percentage) {
      double res = num * percentage / 100;
      System.out.println(res);
   }
   public static void main(String[] args) {
      perc(530, 23);
   }
}
```

OUTPUT
121.9

Lesson Takeaways

Method parameters are awesome!
Here is a summary:
- You can define parameters in the parentheses.
- Multiple parameters need to be separated by commas.
- The parameters are available in the method, like variables of the given names.
- When calling a method, you need to provide its parameters in the same order, as defined.

You will learn how to return values in the next lesson.

Returning From Methods

Return Values

The methods we have seen so far output their result.
In some cases we do not need to output the result, but need to assign it to a variable, to work with it in our program.
In these cases, we need our method to return the result value.

The Return Type

Consider a method from our previous lesson, that was used to output a percentage of the given value:

```
static void perc(double num, int percentage) {
  double res = num*percentage/100;
  System.out.println(res);
}
```

The void keyword in the definition specifies that the method does not return any value.

Here is the same function definition, specifying the return type to be a double:

```
static double perc(double num, int percentage) {
  ...
}
```

This means that our perc method will return a value of type double.

Returning a Value

Now, we can return our result using the return keyword:

```
static double perc(double num, int percentage) {
  double res = num*percentage/100;
  return res;
}
```

The return keyword stops the method from executing. If there are any statements after return, they won't run.

After we have created our method that returns a value, we can call it in our code and assign the result to a variable:

```
public class Demo {
   static double perc(double num, int percentage) {
      double res = num * percentage / 100;
      return res;
   }
   public static void main(String[] args) {
      double x = perc(530, 23);
      System.out.println("Result is: " + x);
   }
}
```

OUTPUT

```
Result is: 121.9
```

Returning is useful when you don't need to print the result of the method, but need to use it in your code. For example, a bank account's withdraw() method could return the remaining balance of the account.

Let's create a method that takes integer parameter, checks if the grade is over 70 and returns a booleanresult.
Then, let's use it in main:

```
public class Demo {
   static boolean check(int grade) {
      if (grade >= 70) {
         return true;
      } else {
         return false;
      }
   }
   public static void main(String[] args) {
      int x = 89;
      if (check(x) == true) {
         System.out.println("Congrats!");
      }
   }
}
```

OUTPUT

```
Congrats!
```

As you can see, we can use the method in an if statement, because it returns a boolean value.

The method can be used anywhere in our program to check if the grade is passing or not. In case anything changes in the logic of the check, we will need to modify the method only, without touching the rest of the program.

Lesson Takeaways

You did it! This was the last lesson of this course.
Here is a quick recap for returning from methods:

- Use the return statement to return a value from your method.
- The method needs to have its return type specified before its name.
- The returned value can be assigned to a variable when calling the method.