

1_Hello_World_.pdf
2_Code_Explanation.pdf
3_Compiling_Your_First_Java_Program_.pdf
4_Challenge__Play_Around_With__print__Statement.pdf
5_Solution_Review__Play_Around_With__print__Statement.pdf
6_Quick_Quiz_.pdf
7_Variables_in_Java.pdf
8_Data_Types.pdf
9_Variable_Syntax.pdf
10_Taking_Variable_Value_From_User.pdf
11_Challenge__Declaring_Variables.pdf
12_Solution_Review__Declaring_Variables.pdf
13_Quick_Quiz_.pdf
14_Simple_Java_Maths.pdf
15_Mathematical_Functions.pdf
16_Logical_Expressions.pdf
17_Challenge_1__Compute_an_Expression_Using_Maths.pdf
18_Solution_Review__Compute_an_Expression_Using_Maths.pdf
19_Challenge_2__Compute_an_Expression_Using_Logical_Operators.pdf
20_Solution_Review__Compute_an_Expression_Using_Logic.pdf
21_Quick_Quiz_.pdf
22_Java.Strings.pdf
23_String_Methods.pdf
24_Challenge__Finding_the_Right_Words.pdf
25_Solution_Review__Finding_the_Right_Words.pdf
26_Quick_Quiz_.pdf
27_Conditional_Statement.pdf
28_if_Conditional_Statements.pdf
29_switch_Statement.pdf

30_Conditional_Expression.pdf
31_Challenge_1_Even_or_Odd.pdf
32_Solution_Review_Even_or_Odd.pdf
33_Challenge_2_What_Day_is_it_.pdf
34_Solution_Review_What_Day_Is_It_.pdf
35_Quick_Quiz_.pdf
36_while__do_while_Loops.pdf
37_for_Loop.pdf
38_Infinite_Loops.pdf
39_Challenge_1_Multiplication_Table_of_a_Number.pdf
40_Solution_Review_Multiplication_Table_of_a_Number.pdf
41_Challenge_2_Calculating_the_First_n_Fibonacci_Numbers.pdf
42_Solution_Review_Calculating_the_first_n_Fibonacci_numbers.pdf
43_Challenge_3_Pyramid_Printing_by_Using_for_Loop.pdf
44_Solution_Review_Pyramid_Printing_by_Using_for_Loop.pdf
45_Quick_Quiz_.pdf
46_Methods_in_Java.pdf
47_Parameters_and_Return_Types_in_Methods.pdf
48_Return_Parameters_in_Methods.pdf
49_Constructor.pdf
50_Static_Methods.pdf
51_Challenge_1_Method_to_Check_Sum.pdf
52_Solution_Review_Method_to_Check_Sum.pdf
53_Challenge_2_Letter_Grade_to_GPA.pdf
54_Solution_Review_Letter_Grade_to_GPA.pdf
55_Challenge_3_Sum_of_Digits_in_an_Integer.pdf
56_Solution_Review_Sum_of_Digits_in_an_Integer.pdf
57_Challenge_4_Playing_With_Strings.pdf
58_Solution_Review_Playing_With_Strings.pdf

59_Quick_Quiz_.pdf
60_What_Are_Arrays_.pdf
61_A_Bit_More_About_Arrays.pdf
62_Two_Dimensional_Arrays.pdf
63_Challenge_1_Find_the_Maximum_Value.pdf
64_Solution_Review_Find_the_Maximum_Value.pdf
65_Challenge_2_Sorting_an_Array.pdf
66_Solution_Review_Sorting_an_Array.pdf
67_Challenge_3_Print_a_Matrix.pdf
68_Solution_Review_Print_a_Matrix.pdf
69_Challenge_4_Pascal_s_Triangle.pdf
70_Solution_Review_Pascal_s_Triangle.pdf
71_Quick_Quiz_.pdf
72_Introduction_to_Classes.pdf
73_Constructors.pdf
74_Class_Member_Methods.pdf
75_Inheritance_in_Java.pdf
76_Challenge_1_Calculating_the_Area.pdf
77_Solution_Review_Calculating_the_Area.pdf
78_Challenge_2_Displaying_Message_Using_Inheritance.pdf
79_Solution_Review_Displaying_Message_Using_Inheritance.pdf
80_Quick_Quiz_.pdf
81_Introduction_to_Generics.pdf
82_Generic_Class.pdf
83_Challenge_1_Finding_Max_in_an_Array.pdf
84_Solution_Review_Finding_Max_in_an_Array.pdf
85_Quick_Quiz_.pdf
86_ArrayLists_in_Java.pdf
87_Creating_an_ArrayList_Object.pdf

88_Inbuilt_Methods.pdf

89_Challenge_1_Gathering_Zeros_to_the_Start.pdf

90_Solution_Review_Gathering_Zeros_to_the_Start.pdf

91_Challenge_2_Remove_Duplicates_From_an_ArrayList.pdf

92_Solution_Review_Remove_Duplicates_From_an_ArrayList.pdf

93_Quick_Quiz_.pdf

Hello World!

In this lesson, you will get acquainted with the Hello World program and a basic introduction to Java.

We'll cover the following

- Hello World: Example
- Java as a structural language
- 🌟 New features
- 💯 Bad practices

Hello World: Example

The first program that most aspiring programmers write is the classic “Hello World” program. The *purpose* of this program is to display the text “Hello World!” to the user.

The “Hello World” example is somewhat famous as it is often the first program presented when introducing a programming language.

```
<Hello World/>
```

Try running the code below!

```
class HelloWorld {  
    public static void main(String args[]) {  
        System.out.println("Hello World!");  
    }  
}
```





Did you know?

The `System.out.println()` displays text in the current line of the console and then move the cursor to the beginning of the next line.

Java as a structural language

Before discussing the particulars, it is useful to think of a computer program in terms of both its **structure** and its **meaning** simultaneously.

A `Java` program is structured in a specific and particular manner. `Java` is a language and therefore has grammar similar to a spoken language like *English*. The grammar of computer languages is usually much, much simpler than spoken languages but comes with the disadvantage of having stricter rules. Applying this structure or grammar to the language is what allows the computer to understand the program and what it is supposed to do.

The overall program has a **structure**, but it is also important to understand the purpose of part of that **structure**. By analogy, a textbook can be split into sections, chapters, paragraphs, sentences, and words (the structure), but it is also necessary to understand the overall meaning of the words, the sentences, and chapters to fully understand the content of the textbook. You can think of this as the semantics of the program.

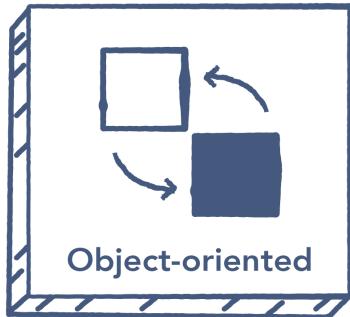
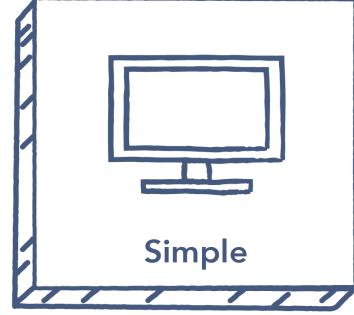
A line-by-line analysis of the program should give a better idea of both the **structure** and meaning of the classic “Hello World” program. Let’s take a look at it in the upcoming lessons.

New features

The new features and upgrades included in Java changed the face of the programming environment and gave a new definition to *Object-Oriented Programming* (OOP in short). But unlike its predecessors, Java needed to be bundled with standard functionality and be independent of the host platform.

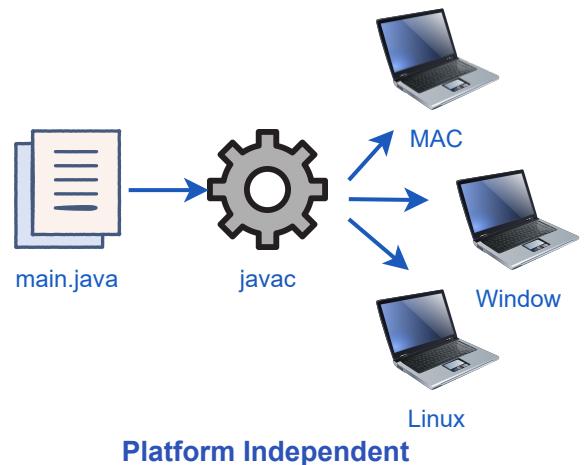
The primary goals in the creation of the Java language:

- It is **Simple & Portable**: Memory Management using Pointers is not allowed.



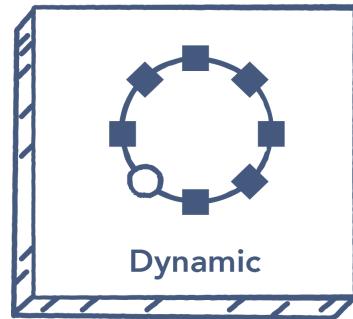
- It is **Object-Oriented**.

- It is **Independent** of the host platform.



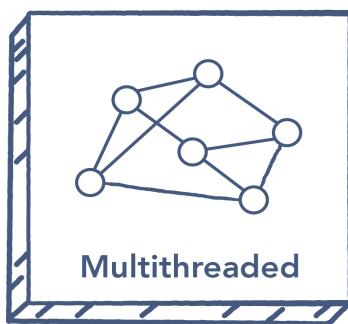
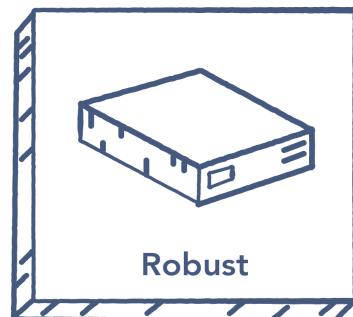
- It is **Secured & Dynamic**: Designed to execute code from remote sources securely.

- It contains language facilities and libraries for **Networking**.



- **High Performance:** With the use of JIT (*Just-In-Time*) compilers, Java achieves high performance through the use of **byte-code** that can be easily translated into native machine code.

- It is **Robust**: Java has its own strong **memory management** system. This helps to eliminate errors as it checks the code during compile and runtime.



- Java is **Multithreaded**: It supports multiple executions of threads (i.e., *lightweight processes*), including a set of synchronization primitives.

The Java language introduces some new features that did not exist in other

Bad practices

Over the years, some features in C/C++ programming became abused by the programmers. Although the language allows it, it was known as bad practices. So the creators of Java have disabled them:

- Use of Pointers
 - Operator overloading
 - Multiple inheritance
 - Friend classes (access another object's private members)
 - Restrictions of explicit type casting (related to memory management)
-

Let's give you a detailed breakdown of the above code in the next lesson!

Code Explanation

In this lesson, we will explore how the "Hello World!" program works in Java.

We'll cover the following ^

- The given code
- Detailed explanation
 - class HelloWorld
 - void main()
 - {}
 - Print statement
 - Semicolons

The given code

```
class HelloWorld {  
    public static void main(String args[]) {  
        System.out.println("Hello World!");  
    }  
}
```



Detailed explanation

Now let's look at the **Hello World!** sequentially.

class HelloWorld #

It is written as:

```
class HelloWorld
```



Java is an object-oriented programming language. Therefore, every line that needs to be executed should be inside a class. This line declares a class **HelloWorld**.

- **Declarations and Definitions:** A class declaration provides the name and visibility of a class. In our example, `class HelloWorld` is the class declaration. It consists (in this case) of one keyword, `class` followed by the identifier `HelloWorld`.
 - The `class` declaration is then followed by a block (surrounded by curly braces `{}`), which provides the class's definition.

This means that we are defining a class named `HelloWorld`. Other classes, or in our case, the command line, can refer to the class by this name.

```
void main() #
```

The `main` is the point from where all the Java programs start its execution. In order to execute your program, you must follow the valid signature of `main()` method which is given below:

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

Don't worry about the details of classes, objects, and the signature of the `main()` method. We will cover it in detail in the upcoming chapters. For now, just remember that we will always write our code inside the class, and this class must contain the `main` method.

Note: In object-oriented programming languages, the program consists of objects. Objects are comprised of methods and the associated data. All objects of the same kind are instances of the same class, which provides a template for all of its objects. For a Java program to run, it must have at least one object, which must define a method called `main()`.

```
{ } #
```

A block of code is defined with the `{ }` tokens. `{` signifies the start of a block of code, and `}` signifies the end.

NOTE: The `{ }` tokens have other uses as well.

Print statement

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



- `System.out.println("Hello World!");`

Gives processor the command to **output** the literal specified in the double quotes ("") and adds a new line to the console. In this case, **Hello World!** is printed.

Did you know? `System.out.print()` prints the content and does not add the new line.

Semicolons

Statements in Java must be terminated with a **semicolon**,

- Just as sentences in English must be terminated with a period.
- Just as sentences in English can span several lines, so can the statements in Java.

In fact, you can use as many spaces and new lines between the words of a Java program as you wish to beautify your code just as spaces are used to justify the text printed on the pages of a book.

Now that we have learned the basics of our code. Let's learn how to compile our code in Java in the next lesson!

Compiling Your First Java Program!

In this lesson, an explanation of how to compile your code in Java, which compilers to use and the most suitable ones for different Operating Systems, is provided.

We'll cover the following

- What does the compiler do?
 -  Did you Know?
- Running the compiler
- Compilation procedure

For the computer to execute the *Hello World* code you have written, it first needs to be compiled by a Java compiler. The compiler translates the computer code written in a particular programming language into the machine code.

What does the compiler do?

In very broad terms:

Did you Know?

The compiler is a translator that acts as an intermediary between the programmer and the CPU on the computer.

A high-level language like Java is actually a sort of '*compromise*' language between the native language of the CPU (generally referred to as machine language) and the native language of the programmer (say English).

Computers do not natively understand human languages. Therefore, the compiler is reading the code written by the programmer and translating it into machine language code that the computer can execute directly.

In **Java**, programs are **not** compiled into **executable files**; they are compiled into **bytecode**, which the **JVM** (Java Virtual Machine) then executes at runtime. Java

source code is compiled into **bytecode** when we use the [javac](#) compiler. The **bytecode** gets saved on the disk with the file extension [.class](#). When the program is to be run, the bytecode is converted using the just-in-time ([JIT](#)) compiler. The result is machine code, which is then fed to the memory and is executed.

In short, we follow the following two steps to execute the Java program:

1. Java programs need to be compiled to **bytecode**.
2. Then we need to convert the **bytecode** into **machine code**.

The Java classes/bytecode are compiled to machine code and loaded into memory by the JVM when needed the first time. This is different from other languages like C/C++, where programs are to be compiled to machine code and linked to create an executable file before it can be executed.

Running the compiler

The code needs to be compiled with a compiler to finish the process.

What if you don't have one?

Well, the good news is, there are *several* good compilers that are available for free.

- The [Eclipse](#) has various versions available for most systems and does a good job of implementing the **Java** standard libraries.
- The [NetBeans](#) has complete support for Java used for multiple purposes, including **WebApps** and **Web-Services**. It also provides Git Repositories to its users.
- Some Mac Users prefer using [Xcode](#) for running Java on their machine.

There are multiple compilers for one language, and picking out one is totally a personal preference.

Compilation procedure

We are providing a Java execution environment in our course, so you don't have to worry about this. But, if you prefer to install the environment on your own machine for other uses, you can follow the instructions given in this [link](#).

We hope you've successfully installed and run your first Java program.

That's it for now, let's play around with some print statements in the next lesson to get *'you'* used to it.

Challenge: Play Around With 'print' Statement

In this challenge, we will display particular text on the console.

We'll cover the following ^

- Try yourself

Try yourself

Print this line using `System.out.print()`:

The movie quote is: My name is Bond. James Bond!

Important Note: When you write the above line, make sure there is only a single space between each word, and no space between `is:`. Otherwise, your code won't pass the test case.

It would be better if you try using the `System.out.println()` statement by yourself first and then see the solution.

```
class HelloWorld {  
    public static void print() {  
        // write your code here  
        System.out.println("write your code here");  
    }  
}
```



For a step by step break down of the solution, let's go to the solution review in the upcoming lesson.

Solution Review: Play Around With 'print' Statement

In this review, a quick analysis of the Print Statement Challenge is provided.

We'll cover the following



- Given solution
- Explanation

Given solution

```
class HelloWorld {  
    public static void main( String args[] ) {  
        System.out.println("The movie quote is: My name is Bond. James Bond!");  
    }  
}
```



Explanation

The above program is quite self-explanatory. We put the required piece of text in double quotation marks inside `System.out.println()`, and it will automatically display the required text on the console.

Let's wrap up this chapter by solving a quiz in the upcoming lesson.

Quick Quiz!

A short quiz to test the basic understanding of the concepts explained in this chapter!

1

A compiler translates the machine-language program into a high-level language.

2

A function can return any type of value in Java regardless of the type specified in declaration.

3

The `int main` function is called by the operating system when the program is executed.

4

A block of code is defined with the `()` tokens.

5

What is the purpose of a semicolon(`;`) at the end of each line in Java?

Retake Quiz

In the next lesson, you will be introduced to variables and user input in Java.

Variables in Java

In this lesson, an introduction to variables, their naming conventions, where they are stored in memory, and how much space they take, will be explained.

We'll cover the following ^

- Introduction
- Naming a variable
- Where are the variables stored?
- Size of a variable?

Introduction

Variables are just like containers which hold the values while the program is executed.

A **variable** is a storage location paired with an associated symbolic name (an identifier), which contains some known or unknown quantity of information referred to as a value.

Variables in Java are **strongly typed**; thus they all must have a **data type** declared with their **identifier**. In this lesson, we will discuss *identifiers*, and we will leave data types for later.

Naming a variable

Just as every person has a name that helps identify them, every variable has a name associated with it. There are certain naming conventions that one must follow in order to decide a name for the variable. This name is called the variable's *identifier*. Variable names should be descriptive based on what data item they are declared to store.

There are three rules to create an identifier:

1. Characters from **A to Z**, as well as their lowercase counterparts, can be used.
2. Numbers from **0-9** can be used.
3. Special characters that can be used are **\$** (the dollar sign) and **_** (underscore).

Note:

1. A variable must never start with a number
2. The identifier can not have spaces in it

While you can give a variable any name, the identifier must describe or be related to the data being stored inside it. The same way when you name a book, it is indicative of the story or characters inside it, the identifier must also be related to the data it represents.

Think of it this way, a book about wizards and magic being called **“Facts and Figures of the Modern World”**. Doesn’t make sense, does it? Rather, if we called it, **“A Magical Journey”**, it would make much more sense. Below are a few examples of the identifiers we can use.

Let’s say we have a book, a book has multiple details- the name of the book, the number of copies published, the author and the publishing company. Imagine if we gave the identifiers as ‘a’, ‘b’, ‘c’, and ‘d’. Wouldn’t that be confusing? Now we will give them identifiers in line with the naming conventions that we have described above:

```
//Declaring variables in java
char author_name;
char book_name;
char publishing$company;
int copies_published_number;
```

If we had written **Book name**, this would not have been a valid identifier because of the spaces in the title. Similarly, **#OfCopies** would have been an invalid identifier as well because it uses a special character that is not allowed. Neither will be **1_BookName** as it starts with a number.

Where are the variables stored?

Variables, once declared in the program, are allocated space in memory. Memory itself is organized as a long collection of one byte after another. These bytes are

sequentially numbered, with the number being referred to as a memory address for the corresponding location. It is customary to use hexadecimal numbers to show the addresses of memory locations.

The value of a given variable is stored in some memory location. Referring to a piece of data using its address is tedious, that is why programming languages provide the abstraction of referring to each piece of data with a variable name.

16 bit Data Memory (RAM)	Memory Address
	0x0800
	0x0802
"Hello"	0x0804
	0x0806
	0x080A

Storing "Hello" in memory at 0x0804 address

1 of 3

16 bit Data Memory (RAM)	Memory Address
	0x0800
	0x0802
"Hello"	0x0804
	0x0806
	0x080A

Is there an easier way to read the text stored at address 0x804?

2 of 3

16 bit Data Memory (RAM)	Memory Address
	0x0800
	0x0802
	0x0804
	0x0806
	0x080A

greeting represents variable name. It refers to memory location 0x0804

3 of 3



Now that we have placed this variable in the machine's memory, one can raise the question of how much space does it take?

Size of a variable?

The number of locations occupied by a given variable gives the size of the variable in memory. For example, a shorter piece of text like "Hello!" occupies less memory than a longer one, like "Hello world!". Java programs have different **data types** that define the space that a variable takes. The smallest space that a variable can take is a data type called **byte** that takes **one byte**.

Now that we know some basic information about variables and how they are stored in memory let's look at the different data types that can be assigned to a variable in Java.

Data Types

In this lesson, we will discuss primitive data types in Java.

We'll cover the following ^

- Numeric data types
 - Integer
 - Floating points
- Textual data types
 - Char
- Boolean data type
- Null data type
- Size of data types

Numeric data types

This classification contains all data types that are numeric.



Did you know?

Variable Types are more often referred to as “**Data Types**”.

Integer

An *integer* is a whole number; that is, it does not have any decimal places.

Examples of an integer include **4**, **5**, **2000**, whereas numbers that are not integers include **4.3**, **4.55**.

Within integers, there are different data types that are based on the space they take within memory. The types include **short**, **int**, and **long**.

```
// A short type is a 16 bit signed integer  
short age; // Stores from +32,767 to -32,768
```

```
// An int type is a 32 bit signed integer  
int age; // Stores from +2,147,483,647 to -2,147,483,648  
  
// A long type is a 64 bit signed integer  
long age; // Stores from +9,223,372,036,854,775,807 to -9,223,372,036,854,775,808
```

Note: Data types in Java are all signed. This means that they can store positive as well as negative values of integers.

Floating points

This data type stores numbers that include a *decimal point*. Hence they will include numbers like **4.34** or even **12233.333922**. Keep in mind that floating points will only store an approximation of the decimal value and not the *exact* value.

Floating points have two types, **floats** and **double**. The primary difference between the two is that **double** is able to store more places after the decimal point more accurately than **float**. The **float** data type is fine for most applications. But, scientific applications that deal in really small values like the radius of an atom or really large values like the distance between two galaxies are more amenable to using **double**.

```
// Assigning a float to a radius of a circle  
  
public class Type_Floats {  
    public static void main(String[] args) {  
        float radius = 8.56f;  
        System.out.println(radius);  
    }  
}
```



Note: If we have to assign a value to float, we must add an 'f' at the end of the number

Now let's look at the assignment of data type double.

```
// Assigning a double to a radius of a circle  
  
public class Type_Double {  
    public static void main(String[] args) {  
        double radius = 8.56d;
```



```
    float radius1 = 0.0000000000863872078f;
    System.out.println(radius1);
    double radius = 0.0000000000863872078;

    System.out.println(radius);
}
}
```



In the above code, you can see `double` data type stores very small values from precisely than the `float` data type.

Textual data types

This data type class deals with data that holds characters. The data may even have single-digit stored; however, this digit can not be used in arithmetic calculations when used in this context.

Char

A `char` is a 16-bit data type that contains either a single digit or a character. Each character is represented by a unique number according to a standard code like Unicode. Let us look at a few examples below.

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

        char number_as_text = '6';
        System.out.println(number_as_text);

        char number = 65;
        System.out.println(number);
    }
}
```



Note: Letters in char type must always be in single inverted commas

Note: A character literal is always enclosed in single quotation marks.

Boolean data type

This data type refers only to two values, *true* and *false*. The default value is false. To understand the use of this data type, let's consider an example. Suppose we have an employee, and we have to store details like 'Does this employee have a car', 'Is this employee part-time?' and so on. Here we can simply store the values as 'true' or 'false', making the variable easier to understand and store. This example is illustrated in the code below.

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



Null data type

Null is a very interesting data type. This basically refers to a value that does not refer to any object. Confused? Basically, we attempt to represent an uninitialized state by using the null value. If we try to read the value of a variable that hasn't been assigned anything yet in our program, we will get the value **Null**.

The following code will generate **Error**, that **line** has not been initialized and hence has a null value.

```
public class Type_Null  
{  
    public static void main(String[] args)  
    {  
        String line;  
        System.out.println(line);  
  
    }  
}
```



Reading compilation errors: The line that shows an error has a specific structure. It shows the name of the file containing the erroneous line of code, followed by a line number and a description of the error.

Size of data types

The following table describes the different data types we covered in this lesson, along with their corresponding memory requirements.

Data Type	Size in bytes
Integer	4 bytes
Float	4 bytes
Double	8 bytes
Character	2 bytes
Boolean	1 byte

Now that we have an idea about most primitive data types let us look at how to declare a variable in Java.

Variable Syntax

In this lesson, we will learn how to use data types and identifiers to create a variable and assign it a value.

We'll cover the following

- Declaring variables
- Initializing variables
- Declaring multiple variables in a single line
- Initializing multiple variables in a single line

Declaring variables

The format followed to declare a variable is simple:

```
<variable type> <variable identifier>;
```

Let's look at an example of declaring a variable without a value.

```
// Creating an age int  
int age;
```

Initializing variables

Each variable that we define must hold some data that it describes. This data is the value of the variable. In order to initialize a variable, we must equate it to a value. Let's look at how this is done by the example of a code snippet.

```
// We have initialized the variable my_age with the value of 13  
int my_age = 13;
```

We can initialize and declare multiple variables in a single line. This helps keep the code precise and compact.

Declaring multiple variables in a single line

It allows variables of the same data type to be created in the same line, as shown in the example below.

```
// Group Declarations  
  
int age, height;  
char letter, alpha;  
double radium, area;
```

Initializing multiple variables in a single line

You can combine multiple variable initializations on the same line as well.

```
// Group initializations  
int age = 10, height = 5;  
double radius = 2.34, area = 4.55;
```

Now that you are familiar with the concept of variables in Java, in the next lesson, we will look into getting a variable's value from the user.

Taking Variable Value From User

In this lesson, an explanation of how to store a value in a variable by using input from the user through the keyboard is provided.

We'll cover the following ^

- Taking user input
- Understanding the code
 - Line 1
 - Line 5
 - Line 8

Taking user input

Up to this point, we have only seen values assigned to variables by means of constants. Sometimes, we would like the user to input a value for a variable from the keyboard.

See the code given below!

Note: Press the [>_STDIN](#) button & type your input before running the widget; otherwise, it'll give an error.

```
import java.util.Scanner;

class take_input {
    public static void main(String[] args) {
        Scanner scanner_one = new Scanner(System.in);

        System.out.println("Enter your name: ");
        String name = scanner_one.nextLine();
        System.out.println("Your name is: " + name);
    }
}
```



Understanding the code

One way to take keyboard input in Java is to use the Scanner class, which is used by first importing the class' definition as in line 1 and then by creating an object of this class as on line 5.

Line 1

The first step is to import the Scanner class so that it can be used in the code below. The *Java* Scanner class is from [java.util](#) package. It is easy to use, but it must be imported for the class to work. The snippet below shows how to import the class.

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



The Scanner class allows the user to take input from the user through the keyboard. *Input* is when a message is received by the system from the user directly.

Line 5

- We declare data type as **Scanner** and give it the following identifier **scanner_one**
- Assign this to the expression **new Scanner(System.in)**
- This expression means that we want to create a new Scanner object which will take **input** from the user

Line 8

- We declare a *variable* of data type **String** with the identifier **name**
- Assign this to the expression **scanner_one.nextLine()**
- This indicates that the Scanner created has a method called **nextLine()**
- This method takes **keyboard input** from the user until the user presses the **Enter** key

In the next lesson, you will solve a simple challenge related to variables.

Challenge: Declaring Variables

In this lesson, we will do a short coding exercise to test the understanding of data types and creating variables in Java

We'll cover the following ^

- Problem statement
- Expected output

Problem statement

- Declare an `int` type variable, name it **int_number** and assign it a value of `1000`.
- Declare a `float` type variable, name it **float_number** and assign it a value of `10.292`.
- Declare a `double` type variable, name it **double_number** and assign it a value of `0.000000000512365123`.
- Declare a `char` type variable, name it **char_name** and assign it a value of `N`.
- Declare a `boolean` type variable, name it **bool_accept** and assign it a value of `true`.

Expected output

Your code should print the following output:

```
Value of Int is:  
1000  
Value of Float is:  
10.292  
Value of Double is:  
5.12365123E-12  
Value of Char is:  
N  
Value of Bool is:
```

Write your code below. We highly recommend that you write the code yourself before checking the solution provided.

Best of Luck!

```
public class Exercise {  
    public static void print() {  
        // Define your variables below this line  
  
        System.out.println("Value of Int is: ");  
        // print your int variable here  
  
        System.out.println("Value of Float is: ");  
        // print your float variable here  
  
        System.out.println("Value of Double is: ");  
        // print your double variable here  
  
        System.out.println("Value of Char is: ");  
        // print your char variable here  
  
        System.out.println("Value of Bool is: ");  
        // print bool double variable here  
    }  
}
```



Note: \n in the test result indicates the next line.

For a step by step break down of the solution, let's go to the solution review in the upcoming lesson.

Solution Review: Declaring Variables

In this review, the solution of the challenge 'Declaring Variables' is provided.

We'll cover the following



- Given solution
- Explanation

Given solution

```
class HelloWorld {  
    public static void main( String args[] ) {  
        //Defining all variables  
        int int_number = 1000;  
        char char_name = 'N';  
        float float_number = 10.292f;  
        double double_number = 0.000000000512365123;  
        boolean bool_accept = true;  
  
        // Printing values of variables  
        System.out.println("Value of Int is: ");  
        System.out.println(String.valueOf(int_number));  
  
        System.out.println("Value of Float is: ");  
        System.out.println(String.valueOf(float_number));  
  
        System.out.println("Value of Double is: ");  
        System.out.println(String.valueOf(double_number));  
  
        System.out.println("Value of Char is: ");  
        System.out.println(String.valueOf(char_name));  
  
        System.out.println("Value of Bool is: ");  
        System.out.println(String.valueOf(bool_accept));  
    }  
}
```



Explanation

From lines 4 to 8, all the variables were declared with their respective data types and values.



Then after every variable was printed using the `System.out.println` command.

Let's go through a *quick quiz* to test your understanding of variables.

Quick Quiz!

Let's take a short quiz to test your understanding of variables in Java

1

Which of the following is not a primitive data type?

2

Which of the following is not a floating-point type value?

3 !

Which variable declaration is correct?

4 !

What is the data type of the value '5'?

[Retake Quiz](#)

In the next chapter, you will be introduced to arithmetic and logical operators in Java.

Simple Java Maths

In this lesson, an introduction of the basic operators used in Java. For example, subtraction, addition, division, and multiplication are explained.

We'll cover the following ^

- Introduction
- Operations in Java
- Example
- Operator precedence
- Operator associativity

Introduction

Math in Java is very simple. Keep in mind that Java mathematical operations follow a particular order much the same as high school math.

For example, multiplication and division take precedence over addition and subtraction. The *order* in which these operations are evaluated can be changed using *parentheses*.

Operations in Java

The arithmetic operators in Java are listed below.

Symbols	Arithmetic operators
+	add
-	subtract
/	divide
*	multiply

%

modulus division

++

post and pre-increment

--

post and pre-decrement

Example

Let's take a look at how to use these operations while coding in Java.

```
public class Operators {
    public static void main(String[] args) {
        int answer;

        System.out.println("ADDITION");
        int add = 20;

        System.out.println("Initial value: " + add);
        answer = add + 2;
        System.out.println("add + 2 = " + answer);
        answer = add;
        System.out.println("add = " + answer);
        System.out.println();

        System.out.println("SUBTRACTION");
        int sub = 15;

        System.out.println("Initial value: " + sub);
        System.out.println("sub - 4 = " + (sub - 4));
        System.out.println("sub = " + sub);
        System.out.println();

        System.out.println("MULTIPLICATION");
        int mult = 25;

        System.out.println("Initial value: " + mult);
        answer = mult * 3;
        System.out.println("mult * 3 = " + answer);
        answer = mult;
        System.out.println("mult = " + mult);
        System.out.println();

        System.out.println("DIVISION (INT)");
        int div_int = 15;

        System.out.println("Initial value: " + div_int);
        System.out.println("div_int / 2 = " + (div_int / 2));
        System.out.println("div_int = " + div_int);
        System.out.println();

        System.out.println("MODULO (REMINDER)");
        int mod_int = 15;
```

```

int rem = 5;

System.out.println("Initial value: " + rem);

answer = rem % 2;
System.out.println("rem % 2 = " + answer);
answer = rem;
System.out.println("rem = " + answer);
System.out.println();

System.out.println("PREINCREMENT BY ONE");
int pre_inc = 5;

System.out.println("Initial value: " + pre_inc);
System.out.println("++pre_inc = " + (++pre_inc));
System.out.println("pre_inc = " + pre_inc);
System.out.println();

System.out.println("PREDECREMENT BY ONE");
int pre_dec = 5;

System.out.println("Initial value: " + pre_dec);
answer = --pre_dec;
System.out.println("--pre_dec = " + answer);
answer = pre_dec;
System.out.println("pre_dec = " + answer);
System.out.println();

System.out.println("POST INCREMENT BY ONE");
int post_inc = 5;

System.out.println("Initial value: " + post_inc);
System.out.println("post_inc++ = " + (post_inc++));
System.out.println("post_inc = " + post_inc);
System.out.println();

System.out.println("POSTDECREMENT BY ONE");
int post_dec = 5;

System.out.println("Initial value: " + post_dec);
answer = post_dec--;
System.out.println("post_dec-- = " + answer);
answer = post_dec;
System.out.println("post_dec = " + answer);
System.out.println();
}
}

```



Operator precedence

Operator precedence specifies the order in which operations will execute provided that the expression contains more than one operator. With mathematical operations, the precedence follows the rule of BODMAS, which says; Brackets, Order, Division, Multiplication, Addition, and then Subtraction. Let's look at an example below to check this.

example below to check this:

For the purpose of these examples, we will set the following variables; $x=5$, $y=10$, $z=7$, and $w=6$. Now let's evaluate the expressions below using these variables.

- $(x - z) + y * y$
 - The part that will be evaluated first is $(x - z)$ as it is in the brackets and will come out to be -2
 - The second part that will be calculated is $y * y$ as BODMAS ensures that multiplication is done before addition, giving us the answer 100
 - The final step will be $-2 + 100$ giving us the answer 98
- $w / y + (x * z)$
 - The first step to be calculated is $x * z$, giving us the answer, 35
 - The second step is the division, w / y . Keep in mind that since the variables are assumed to be integers, the result will also be an integer. Hence the answer will be 0 (an int data type rounds to the lower number)
 - The last step will be the addition of the two answers $0 + 35$ giving us the result 35

Now, let's look at the code below to see if this actually works!

```
public class Oper_Prec {  
    public static void main(String[] args) {  
        int x = 5;  
        int y = 10;  
        int z = 7;  
        int w = 6;  
        int answer;  
  
        System.out.println("x: " + x);  
        System.out.println("y: " + y);  
        System.out.println("z: " + z);  
        System.out.println("w: " + w);  
  
        System.out.println("Calculating Expressions with Multiple Operators");  
  
        System.out.println("(x-z) + y*y = " + ((x - z) + y * y));  
        answer = (w / y + (x * z));  
  
        System.out.println("w/y + (x*z) = " + answer);  
    }  
}
```

Operator associativity

Operator associativity determines whether, in an expression, if there are multiple operators like (1 + 2 - 5), how will they be evaluated if they are of the same precedence. The addition + and subtraction - have left associativity.

- $x + y - z$
 - The part $x + y$ will be evaluated first, which will give us 15
 - Then it will calculate $15 - 7$ and hence give the final answer 8

RUN the code given below and see the output!

```
public class Oper_Prec {  
    public static void main(String[] args) {  
        int x = 5;  
        int y = 10;  
        int z = 7;  
        int w = 6;  
        int answer;  
  
        System.out.println("x: " + x);  
        System.out.println("y: " + y);  
        System.out.println("z: " + z);  
  
        System.out.println("Calculating Expression containing operators the with same precedence")  
  
        answer = x+y-z;  
  
        System.out.println("x+y-z = " + answer);  
    }  
}
```



The following table shows the associativity of **Java** operators (from highest to lowest precedence).

Operators	Description	Associativity
+ , -	Unary Plus and minus	Right to left
! , ~	Logical NOT and bitwise NOT	Right to left

=	Direct assignment	Right to left
+= , -=	Assignment by sum and difference	Right to left
*= , /= , %=	Assignment by product, quotient, and remainder	Right to left
<<= , >>=	Assignment by bitwise left shift and right shift	Right to left
&= , ^= , =	Assignment by bitwise AND, XOR, and OR	Right to left
++ , -	Suffix/postfix increment and decrement	Right to left
* , / , %	Multiplication, division, and remainder	Left to Right
+ , -	Addition and subtraction	Left to Right
<< , >>	Bitwise left shift and right shift	Left to Right
< , <=	For relational operators	Left to Right
> , >=	For relational operators	Left to Right
== , !=	For relational	Left to Right
&	Bitwise AND	Left to Right

<code>^</code>	Bitwise XOR (exclusive or)	Left to Right
<code>&&</code>	Logical AND	Left to Right
<code> </code>	Logical OR	Left to Right
<code> </code>	Bitwise OR (inclusive or)	Left to Right

Going good so far? Then let's move on to the next lesson for more interesting operations that you can play around within Java!

Mathematical Functions

In this lesson, an explanation of the `java.lang.Math` class and the various methods we can find in it are provided.

We'll cover the following ^

- Java math class
- Exponentiation
- Logarithms
- Trigonometry
- Absolute value
- Maximum and minimum values

Java math class

The **Java Math class** is from the `java.lang` package. It is easy to use, and since it is a part of the `java.lang` package and everything from the `java.lang` package is automatically imported and available, so it does not need to be explicitly imported.

Now, let's look at the different functions it has to offer!

Exponentiation

1. The **pow** method takes two arguments and returns the value of the first argument raised to the power of the second one. `double Math.pow(2,3) returns 8`
2. The **exp** method takes one argument and returns the value of e raised to the power of the given argument. `double Math.exp(2) returns e^2`
3. The Math class also has a **sqrt** and **cbrt** methods, which returns the square root and cube root of the number specified. `double Math.sqrt(16) returns 4` and `double Math.cbrt(27) returns 3`.

```
public class exponent {  
    public static void main(String[] args) {  
        System.out.println(Math.sqrt(16));  
        System.out.println(Math.cbrt(27));  
    }  
}
```

```
public static void main(String[] args) {  
    System.out.println("2 raised to the power 3 is " + Math.pow(2, 3));  
    System.out.println("Exponent squared is " + Math.exp(2));  
    System.out.println("The square root of 16 is " + Math.sqrt(16));  
    System.out.println("The cube root of 27 is " + Math.cbrt(27));  
}  
}
```



Logarithms

1. The **log** method takes one argument and returns the natural log of the given argument. `double Math.log(2) returns 0.69`
2. The **log10** method is a useful method that takes in one argument and returns the value of the *common log* of that argument. `double Math.log10(100) returns 2`

```
public class logs {  
    public static void main(String[] args) {  
        System.out.println("log of 2 is " + Math.log(2));  
        System.out.println("log to the base 10 of 100 is " + Math.log10(100));  
    }  
}
```



Trigonometry

To perform *trigonometric* operations, Java provides us the following methods: **sin**, **cos**, and **tan**. Each of the *three* takes only *one argument*, of the data type *double*, on which these operations need to be applied.

Note: Trigonometric methods in `java.lang.Math` takes an angle in *RADIANS*.

```
public class trig {  
    public static void main(String[] args) {  
        System.out.println("tan(45) =" + Math.tan(Math.toRadians(45)));  
        System.out.println("sin(45) =" + Math.sin(Math.toRadians(45)));  
        System.out.println("cos(45) =" + Math.cos(Math.toRadians(45)));  
    }  
}
```



 Hide Hint

The method `Math.toRadians()` converts a degree number to a radian number and `Math.toDegrees()` does vice versa.

Absolute value

The **abs** method is used to return the absolute, i.e., the positive value of the given parameter. This method is compatible with the following types: `int`, `long`, `float`, and `double`.

```
public class absolute {
    public static void main(String[] args) {
        System.out.println("Absolute value of -2: " + Math.abs(-2));
        System.out.println("Absolute value of 2: " + Math.abs(2));
    }
}
```



Maximum and minimum values

The **max** and **min** methods return the maximum and minimum of the two arguments given to the functions, respectively. This method is compatible with the following data types: `int`, `long`, `float`, and `double`.

```
public class max_min {
    public static void main(String[] args) {
        System.out.println("Maximum between 2.04 and 2.05: " + Math.max(2.04, 2.05));
        System.out.println("Minimum between 2 and 23: " + Math.min(2, 23));
    }
}
```



Let's move on to the key concepts of logical expressions in the next lesson.

Logical Expressions

In this lesson, an explanation of the use of logical expressions in Java programs is provided.

We'll cover the following



- Introduction
- Comparative operators
- Boolean operators

Introduction

Logical expressions are also known as **Boolean expressions**. It will always evaluate to a value of either **true** or **false**. Therefore, they will be represented by the data type **boolean**. While they may seem similar to mathematical operators, the difference lies in how they are used with **comparative** or **boolean operators**. Let's look at both types of operators in detail.

Comparative operators

Java has several operators that can be used to *compare* value. Comparison implies knowing which value is greater than the other, or equal to it, and so on. The table below shows the entire list of operators available in Java.

Symbols	Comparative operators
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

==

Equal to

!=

Not equal to

Note here that these comparative operators can be used on any **primitive data type** except for boolean.

Now that we have an idea of what these operators are let's see how we can use them in code.

```
class log_op {  
    public static void main(String[] args) {  
        int x = 5;  
        int y = 10;  
  
        System.out.println("x is equal to: " + x);  
        System.out.println("y is equal to: " + y);  
  
        System.out.println("x is greater than y");  
        System.out.println(x > y);  
  
        System.out.println("x is less than y");  
        System.out.println(x < y);  
  
        System.out.println("x is greater than or equal to y");  
        System.out.println(x >= y);  
  
        System.out.println("y is less than or equal to y");  
        System.out.println(y <= y);  
  
        System.out.println("x is equal to y");  
        System.out.println(x == y);  
  
        System.out.println("x is not equal to y");  
        System.out.println(x != y);  
    }  
}
```



Boolean operators

These operators rely on **boolean algebra**. Hence, boolean operators will work directly on boolean values. There are **four** types of boolean operators. Let's first look at their symbols and what they are in the table below before we discuss what functionality they perform.

!

Boolean NOT

&&

Boolean AND

||

Boolean OR

^

Boolean exclusive XOR

- The **boolean NOT** inverts the value of the boolean expression that follows it.
- The **boolean AND** will return **true** if and only if, expressions on both sides of the operator are **true**.
- The **boolean OR** will return **true** if the expression on either or both sides of the operator is **true**.
- The **boolean exclusive XOR** will return **true** if one of the expressions evaluates to **true** and the other to **false**.

Now that we have some understanding of these operators let's see how to use them in code.

```
class bool_ops {  
    public static void main(String[] args) {  
        boolean x = true;  
        boolean y = false;  
  
        System.out.println("Value of x: " + x);  
        System.out.println("Value of y: " + y);  
  
        System.out.println("Boolean NOT of x");  
        System.out.println(!x);  
  
        System.out.println("Boolean AND of x and y");  
        System.out.println(x && y);  
  
        System.out.println("Boolean OR of x and y");  
        System.out.println(x || y);  
  
        System.out.println("Boolean exclusive XOR of x and y");  
        System.out.println(x ^ y);  
    }  
}
```



Now that we have understood the fundamentals of Maths and Logic in **Java**, we will do some practice before moving on to the next lesson!

Challenge 1: Compute an Expression Using Maths

In this exercise, you need to compute the expression.

We'll cover the following



- Problem statement
- Coding exercise

Problem statement

After all this number crunching, we will move onto more complex expressions. Just like a Pokemon evolves, so do the expressions you work with. The mathematical expression given below is one that is slightly more difficult than the ones we have worked with, and hence, it is a challenge for you to solve!

Below is an *illustration* showing the **expression** you need to compute:

$$\sqrt[3]{x^2 + y^2 - |z|}$$

Coding exercise

Write a code in which you:

- First, compute the **respective powers** of **two floating-point** variables `x` and `y`.
- Then **Add** them after taking **powers**.
- Then, compute the **absolute value** of **floating-point** `z`.
- Subtract this from the above-computed addition value.
- Now take **Cube Root** of the answer.
- Use `inbuilt functions` to calculate this expression

Only write the code where instructed in the snippet below. You need to calculate

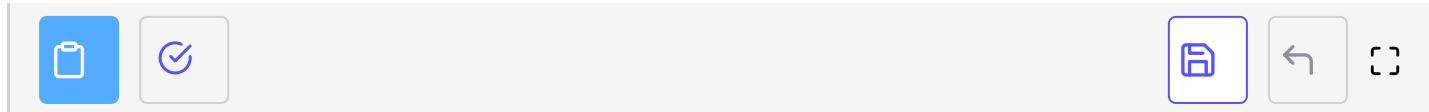
the value of the expression, and then store your final result in the variable `answer`. The **return** statement and the **variable** to be returned are already mentioned for you. Don't worry too much about the return statement for the moment, and just set the value of the `answer` correctly.

Test your code against our cases and see if you can pass them.

The solution is given in case you get stuck, and the next lesson will include a review of the solution, but it is **highly recommended** that you try it yourself first!

Good Luck!

```
class exercise {  
    public static double exercise_one(double x, double y, double z) {  
        double answer = 0;  
        // Enter your code here  
        // Calculate the value of an expression and store the final value in the answer  
  
        /* You do not need to worry too much about the return statement for the  
        moment and just set the value of “answer” correctly*/  
        return answer;  
    }  
}
```



In the next lesson, we will review the solution to the above challenge.

Solution Review: Compute an Expression Using Maths

In this review, the solution of the challenge 'Compute an expression using maths' from the previous lesson is provided.

We'll cover the following ^

- Solution
- Understanding the Code

Solution

```
class HelloWorld {  
    public static void main( String args[] ) {  
        double x = 55.0;  
        double y = 18.0;  
        double z = 51.0;  
        double answer;  
        double sum = Math.pow(x, 2) + Math.pow(y, 2);  
        double sub = sum - Math.abs(z);  
        answer = Math.cbrt(sub);  
        System.out.println(answer);  
    }  
}
```



Understanding the Code

Line 7

- The first step is to create a **double** type variable called **sum**.
- This variable stores the *addition result* of **two** expressions.
- The first of these expressions is the **square of x** calculated using the **pow()** method.
- The second is the same but with variable **y**.
- The sum of the two expressions is assigned to the variable **sum**.

Line 8

- The first step is to create a **double** type variable called `sub`.
- This variable stores the *subtraction result* of **two** expressions.
- The variable from which the expression is to be subtracted is the `sum`.
- The second is the **absolute** value of `z` calculated using the method `abs()`.
- This expression is **subtracted** from `sum`, and the answer is stored in `sub`.

Line 9

- The **cube root** of the variable `sub` is calculated in the final step.
- This is done using the `cbrt()` method.
- The final result is stored in the `answer` variable.

In the next lesson, we will solve the challenge related to logical expression.

Challenge 2: Compute an Expression Using Logical Operators

In this exercise, you need to compute an expression using logical operators.

We'll cover the following ^

- Problem statement
- Coding exercise

Problem statement

There's always some logic in whatever we do, whether it seems like that or not. This time, we are going to give you the logic, and all you need to do is implement it. The challenge is to create the expression explained below and find what result it gives!

Coding exercise

The first step is done for you, which is the method. The method takes in two variables and computes a logical expression using them. For explanation's sake, the parameters are called `x` and `y`.

Now you must do the following:

- Find the **Boolean NOT** of `x`
- **Boolean XOR** the result of above with `x` itself
- Find the **Boolean AND** of the above answer with `y`
- Return the **Boolean NOT** of the entire expression

Only write the code where instructed in the snippet below. You need to calculate the value of the expression, and then store your final result in the variable `answer`. The **return** statement and the **variable** to be returned are already mentioned for you. Don't worry too much about the return statement for the moment and just set the value of the `answer` correctly.

Test your code against our cases and see if you can pass them.

The solution is given in case you get stuck, and the next lesson will include a review of the solution, but it is **highly recommended** that you try it yourself first!

Good Luck!

Your solution will be considered correct only when all the **four** test cases are passed in the below widget.

```
class exercise {  
    public static boolean exercise_two(boolean x, boolean y) {  
        boolean answer = false;  
  
        // Enter your code here  
        // Calculate the value of an expression and store the final value in the answer  
  
        /* You do not need to worry too much about the return statement for the  
        moment and just set the value of "answer" correctly*/  
        return answer;  
    }  
}
```



In the next lesson, we will review the solution to the above challenge.

Solution Review: Compute an Expression Using Logic

In this review, the solution of the challenge 'Compute an expression using logic' from the previous lesson is provided.

We'll cover the following ^

- Solution: Is it true or false?
- Understanding the code

Solution: Is it true or false?

```
class HelloWorld {  
    public static void main( String args[] ) {  
        boolean x = true;  
        boolean y = true;  
        boolean answer;  
  
        boolean not_x = !x;  
        boolean xor_x = not_x ^ x;  
        boolean and_xy = xor_x && y;  
        answer = !and_xy;  
  
        System.out.println(answer);  
    }  
}
```



Understanding the code

Line 7

- The variable, `not_x` stores the result of the **Boolean NOT** of `x`.
- The value of `x` is **true**. Hence, it stores a value of *false* in the example above

Line 8

- The variable `xor_x` stores the result of the **Boolean XOR** of `x` with `not_x`.
- Hence, it stores a value of *true* as the **XOR** of *true* with *false* is **true**.

Line 9

- The variable, `and_xy` stores the result of the **Boolean AND** of `y` with `xor_x`.
- Hence, it stores a value of `true` as the **AND** of `true` with `true` is **true**.

Line 10

- This stores the result of the **Boolean NOT** of `and_xy`.
- Hence, it stores a value of `true` as the **NOT** of `true` is **false**.

Let's go through a *quick quiz* to test your understanding.

Quick Quiz!

Let's take a short quiz to test your understanding of operators in Java.

1

What is the answer of the following expression?

```
(2+3)-10 * 5;
```

2

Which of the following methods does NOT take one input?

3

What is the result of the following logical expression?

```
( 5>= Math.pow(2,3))
```

4

What is the result of the following expression?

```
((true && true) || false) ^ false
```

[Retake Quiz](#)

In the next chapter, we will cover Java strings.

Java Strings

In this lesson, the basic string data type in Java and the various inbuilt functions that the language offers will be explained.

We'll cover the following ^

- Introduction
- String literals
- Immutability

Introduction

String is a class that is built into the **java.lang** package. This represents **character** strings, i.e., text type of data is stored in the *String* type in Java.

String literals

String literals are a means of expressing text type of data in Java. They start with **double-quotation marks** and then contain zero or more characters, including Unicode characters. The String literal must end with another double-quotation marks **"**. This would be a String in Java, **"Hello World!"**.

In String literals, the backslash character **** helps incorporate special characters into the line of text. The table below shows the syntax for these special characters.

Name	Character
Backspace	\b
Tab	\t
NULL character	\0
Newline	\n

Carriage Control

\r

Double Quote

\"

Single Quote

\'

Backslash

\\"

The code below shows an example of *how to use* these special characters!

```
class special_characters {  
  
    public static void main(String[] args) {  
        String special_char = "Line one\n" + "Line two\n";  
        System.out.println(special_char);  
    }  
}
```



Immutability

Keep in mind that **String** objects are *immutable*, i.e, they cannot be modified once created. Hence, when a String object is modified, it is actually a **new** string that is being created. Hence, to save changes to the String, we must assign it to the new String. Look at the example below!

```
class immutability {  
    public static void main(String[] args) {  
        String text = "      Cut string";  
        // The trim function is meant to eliminate leading & trailing spaces  
        text.trim();  
        // Without assigning the text variable to the trimmed string  
        System.out.println(text);  
  
        // Assigning trimmed string to the variable  
        text = text.trim();  
        System.out.println(text);  
    }  
}
```



Now that you have the basic idea of **Strings** in Java. Let's look at the different functions that this class has to offer in the next lesson!

String Methods

In this lesson, we will see the functionality of inbuilt methods in String Class.

We'll cover the following

- Concatenation
- Comparing strings
- Splitting a string
- Substrings
 - Understanding the two implementations
- String cases
- Length of a string

Concatenation

Java provides special support for the concatenation of multiple *Strings*.

Concatenation is referred to as the joining of two or more Strings. This is done by the use of the `+` operator. The code below shows an example of both.

Did you know? The interesting thing is that the `+` operator can be used to **not only** join a String with other Strings but also join Strings with **other types** of objects.



Concatenation

```
class concat {  
    public static void main(String[] args) {  
        String one = "Hello";  
        String two = " World";  
        int number = 10;  
  
        // concatenating two strings  
        System.out.println(one + two);  
  
        //concatenating a number and string  
        System.out.println(one + " " + number);  
  
        //saving concatenated string and printing  
        String new_string = one + two + " " + number;  
        System.out.println(new_string);  
    }  
}
```

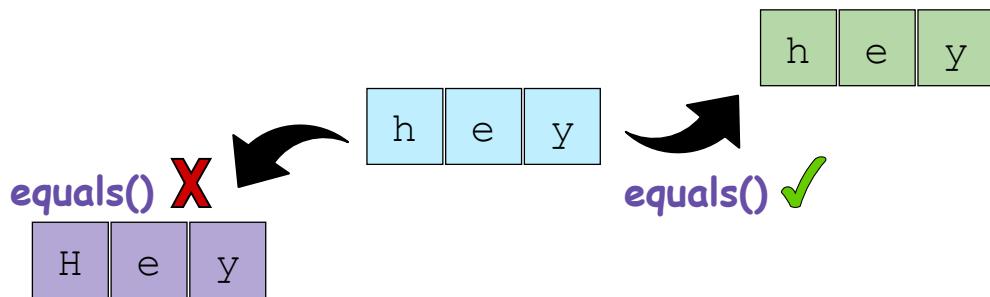


Note: Keep in mind that using the + operator will first convert the number or other objects to String type and then do the concatenation!

Comparing strings

The **String** class has an in-built function called **equals()** for this operation. The method returns **true** if the two Strings are identical and **false** if they aren't. The function is case-sensitive, as can be seen in the code snippet below.

```
class concat {  
    public static void main(String[] args) {  
        String one = "Hello";  
        String two = "World";  
        String lower = "hello";  
        String same = "Hello";  
  
        System.out.println(one.equals(two));  
  
        System.out.println(one.equals(lower));  
  
        System.out.println(one.equals(same));  
    }  
}
```



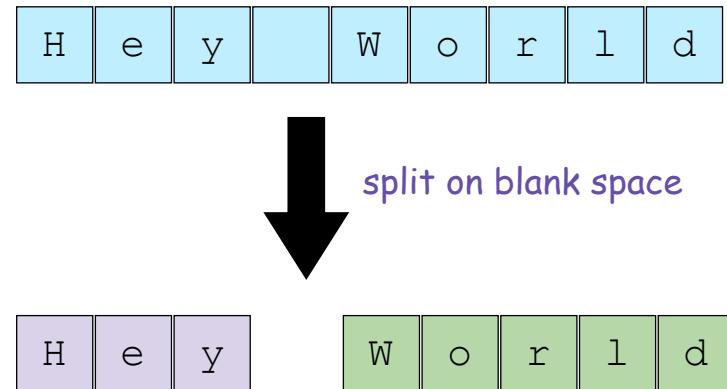
Comparing strings

Splitting a string

This allows the programmer to split the **String** on the basis of a *regular expression*.

This is similar to `split()` method of the **String** class, which splits a string into an array of substrings.

This, in simple terms, means that the **String** will be split on a particular pattern that we can give to the **split()** function built into the **String** class. The function will return a **String array** with elements separated on the basis of the *expression* given. Let's see how it works in the snippet below.



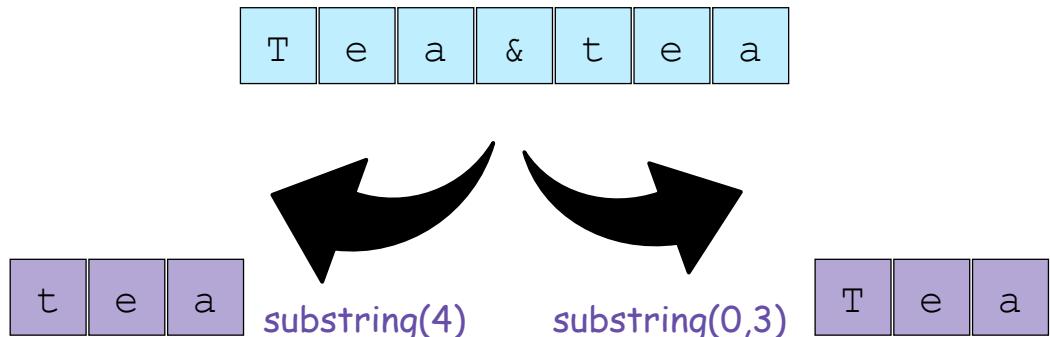
Splitting a String

```
class split_string {
    public static void main(String[] args) {
        String greet = "Hello World,My name is Waldo,How are you?";
        String[] greetings = greet.split(",");
        System.out.println(greetings[0]);
        System.out.println(greetings[1]);
        System.out.println(greetings[2]);
    }
}
```



Substrings

This method allows the programmer to extract **substrings** from given Strings, i.e., you can take out a part of an existing String as a new String. The method that allows this functionality is called **substring**, and it works in two ways. The code snippet below shows both ways of using this method.



```
class substring_ {
    public static void main(String[] args) {
        String choice = "CoffeeOrTea";
        //First: Only one argument
        System.out.println(choice.substring(8));

        //Second: Two arguments
        System.out.println(choice.substring(0, 6));
    }
}
```



Understanding the two implementations



The image above shows how a String is stored with each letter in a box. The counting of the boxes starts with **0**. Hence the letter **H** will be at **index 0** of the string. Now that this is understood let's understand the implementation of the two methods.

One Argument: This method takes only one input for the `substring()` function. This argument given signifies the index at which the *extraction of substring* should

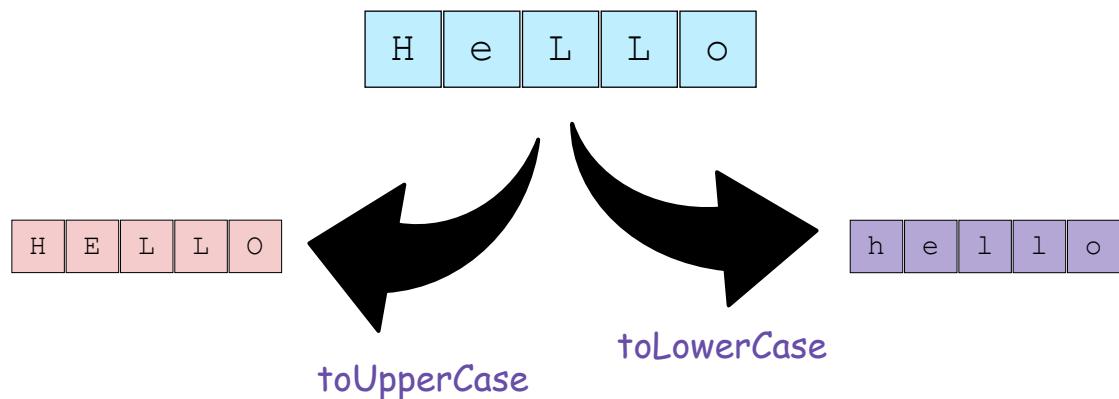
begin. Hence, in the snippet above, the index given is **8**, and so it starts extraction from the **index 8** till the **end** of the String.

Two Arguments: This method takes in two input arguments, one for the index at which the *substring extraction* would begin and the second for where the substring should *end*.

Note: The end index is not *inclusive*, and hence the last character in the substring will be from the *end index given -1* of the original String.

String cases

There are two in-built functions that take a String in its input and return the new String that contains all the Upper or Lower case characters. The code snippet below shows how to use the methods, **toUpperCase()** and **toLowerCase()**, respectively.



String Cases

```
class split_string {
    public static void main(String[] args) {
        String greet = "HeLlo WoRld";

        //Returns new string in which all characters are converted to upper case
        System.out.println(greet.toUpperCase());
```

```
//Returns new string in which all characters are converted to lower case  
System.out.println(greet.toLowerCase());
```

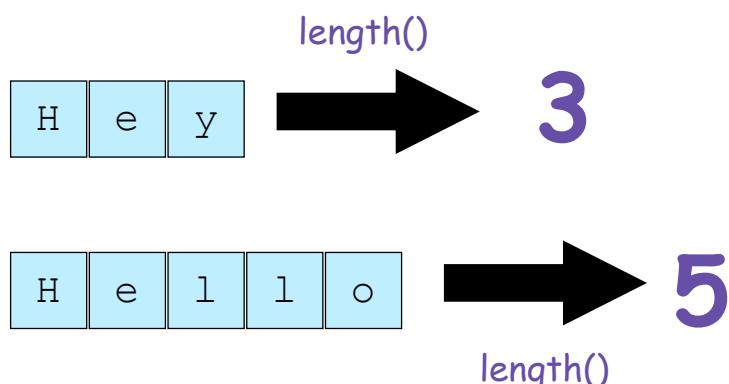
```
}
```

```
}
```



Length of a string

There is an in-built method in Java that returns the total length of a String. This will include any *white spaces* within the String as well. The method is called **length()**. The code snippet below shows how to use this method.



Length of a String

```
class length_of_String {  
    public static void main(String[] args) {  
        String greeting = "Hello World";  
        System.out.println("The length of greeting is: " + greeting.length());  
    }  
}
```



In the next lesson, we will solve a challenge related to strings.

Challenge: Finding the Right Words

In this exercise, you need to figure out how to find the right words in a String according to the given conditions.

We'll cover the following ^

- Problem statement
- Coding exercise

Problem statement

Sometimes we can never find the right words to say. At other times, you find too many words, and you have to find the right phrase. That is exactly what we will do in this challenge. We will take a garbled set of sentences and extract what we want from that set.

It's time to get started! You're going to be working with a string of text that contains several sentences. Your task is to find specific words or phrases within those sentences. To make things more interesting, some of the words may be capitalized or appear in different contexts. You'll need to use your coding skills to identify the correct words and extract them from the string. Good luck!

Coding exercise

This challenge will test your understanding of all *String* knowledge that you have learned. The following steps must be done in that order and on the result from the previous step.

1- Remove all the spaces from the given string.

2- Take the result of step 1 and extract all characters between index `0` and `5` inclusive.

3- Take the result of step 2 and convert all letters to uppercase

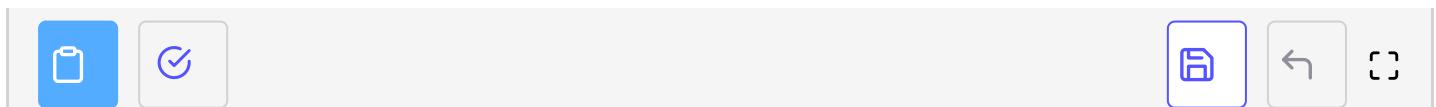
Only write the code where instructed in the snippet below. You need to store your final result in the variable `answer`. The **return** statement and the **variable** to be returned are already mentioned for you. Don't worry too much about the return statement for the moment, and just set the value of the `answer` correctly.

Test your code against our cases, and see if you can pass them.

The solution is given in case you get stuck, and the next lesson will include a review of the solution, but it is highly recommended that you try it yourself first!

Good Luck!

```
class Stringchallenge {  
    public static String sc(String text) {  
  
        String answer = "";  
        // Enter your code here  
        // Store your final result in the variable answer  
  
        /* You do not need to worry too much about the return statement for the  
        moment and just set the value of "answer" correctly*/  
        return answer;  
    }  
}
```



In the next lesson, we will see the solution review of this challenge.

Solution Review: Finding the Right Words

In this review, the solution of the challenge 'Finding the Right Words' from the previous lesson is provided.

We'll cover the following

- Solution: did you find the right words?
- Understanding the solution

Solution: did you find the right words?

```
class HelloWorld {  
    public static void main( String args[] ) {  
        String text = "    Names! Doe    ";  
        String answer = "";  
  
        answer = text.trim();  
        answer = answer.substring(0, 6);  
        answer = answer.toUpperCase();  
  
        System.out.println(answer);  
    }  
}
```



Understanding the solution

The first step is to analyze the given String. The String has three **leading** spaces and three **trailing** spaces. There are only two capitalized letters, and the String is longer than six characters; hence the **required** String must be *extracted*.

Line 6

- Using the **trim()** method, the leading and trailing *spaces* are removed from the given String.
- Assign **answer** to this String to **save** the changes made to argument **text**.

Line 7

- This expression assigns an answer to the modified version of the *answer*.
- The **substring()** method extracts the required String from the given one.
- The starting parameter is **0**, as it is inclusive. However, the end parameter is **6** as that is **not inclusive**.

Line 8

- This expression assigns an answer to the modified version of the *answer*.
- This uses the **toUpperCase()** method to convert all characters to upper case ones in the String chosen.

Let's go through a *quick quiz* to test your understanding.

Quick Quiz!

Let's take a quick quiz to test your understanding of what we have learned so far!

1

Which special character is used to add a new line in Strings?

2

What will be the output of the following code?

```
String name="    John Doe    ";
name.trim();
System.out.println(name);
```

Note: The spaces in the name are signified by - in the answers

3

What is the output of the following code?

```
String text="Hi my name is";
String name="John";
System.out.println(text+name);
```

4

What is the substring extracted by this code?

```
String text="Please bring my coffee";
System.out.println(text.substring(6,11));
```

[Retake Quiz](#)

Now that we know how *Strings* in Java work and how they can be used, the next chapter will teach you about **Conditional Statements** in Java.

Conditional Statement

In this lesson, you will be introduced to conditional statement.

We'll cover the following



- Introduction
- Comparison operators

Introduction

As the name implies, *conditional statements* specify whether another *statement* or *block* of statements should be executed or not. These are often called “**selection constructs**”. The two general types are:

- “if-then-else”
- the “switch-case” construct

Comparison operators

The conditions tested are specified using **comparison operators**. These *operators* cause the **immediate** statement in which they are contained to return a **Boolean** value of either `true` or `false`.

Note: In certain circumstances, `true` and `false` may be assigned to `1` and `0`, respectively. So, be careful while combining *conditional* statements with arithmetic.

Let's look at the *if conditional* statement in the next lesson.

if Conditional Statements

In this lesson, an explanation of if statements and how to write them using an example is provided.

We'll cover the following ^

- The if statement
 - Example
 - Explanation
- The if-else statement
 - Example
 - Explanation
- The if-else if statement
 - Example
 - Explanation

The **if** statement

The **if statement** is used when a statement or group of statements are to be executed if a condition is `true`. The condition will be checked, and if the condition is true, then the certain operation will be executed, and if the condition is false, then nothing will happen. The program will move to the next executable statement. Let's understand this with the below example:

Example

```
class conditional {  
    public static void main(String[] args) {  
        int x = 10;  
  
        if (x > 4) {  
            System.out.println("x is greater than 4");  
        }  
    }  
}
```



Explanation

Line 5:

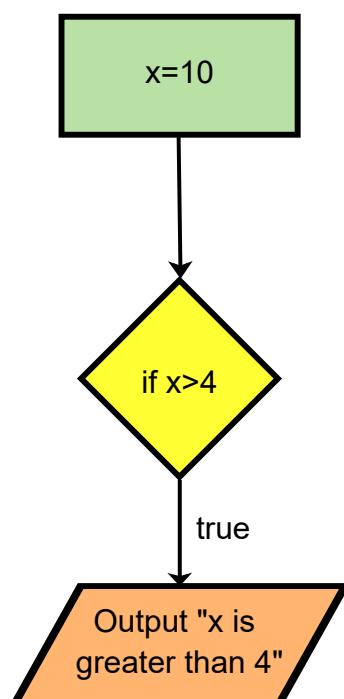
- Inside the parentheses is the **condition**; in this case, it is a *test for greater than*.
- It is a good practice to compare only the same types of data (for example, do not compare *floating-point* values to *characters*).
- Also, note the **left curly brace** `{` after the closing paratheses. This symbol denotes a block of *multiple* lines of code. Without it, the conditional would only refer to the statement immediately following it.
- It is a good practice to always use braces.

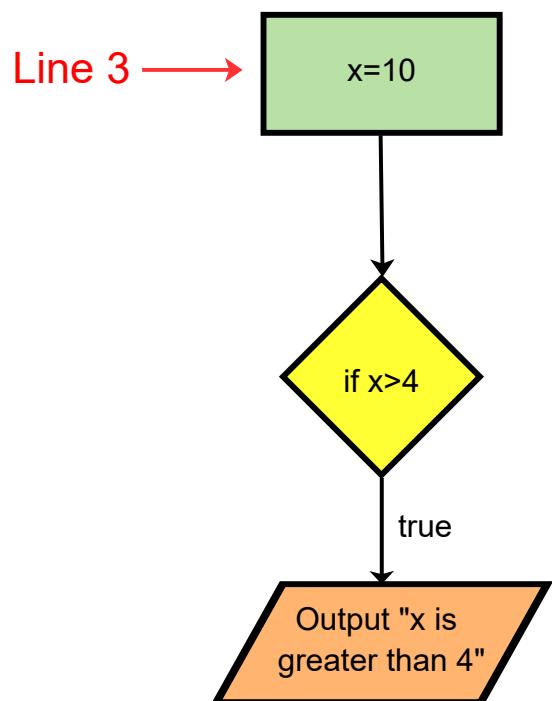
Line 6:

- This statement represents the **body** of the conditional statement.

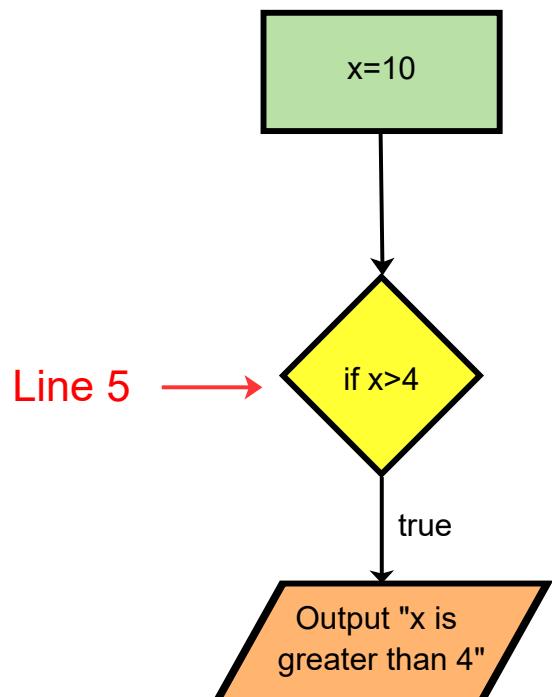
Line 7:

- The **right curly brace** is **essential**; it matches the *opening brace* on line 5 and signals the *end* of the `if` body.

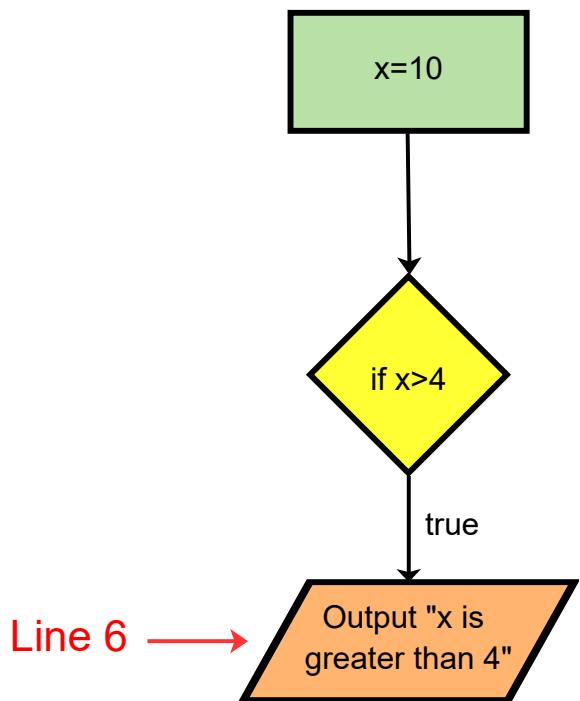




2 of 4



3 of 4



4 of 4



The `if-else` statement

The `if` clause executes statements only when a certain condition is true. If some commands also need to be executed when the condition is false, then the `if-else` clause comes into play. The `if` clause will be the same, and an `else` clause will be added to execute statements when the condition under `if` fails.

Note: There can be multiple conditions for which `if` clause fails, so use commands under `else` carefully.

Let's understand this with the below example:

Example

```

class conditional {
    public static void main(String[] args) {
        int x = 1;

        if (x > 4) {
            System.out.println("x is greater than 4");
        }
    }
}
  
```



```

        }
    else {
        System.out.println("x is less than 4");
    }
}

```



Explanation

The details for the `if` clause is the same. The `else` part is described below:

Line 8:

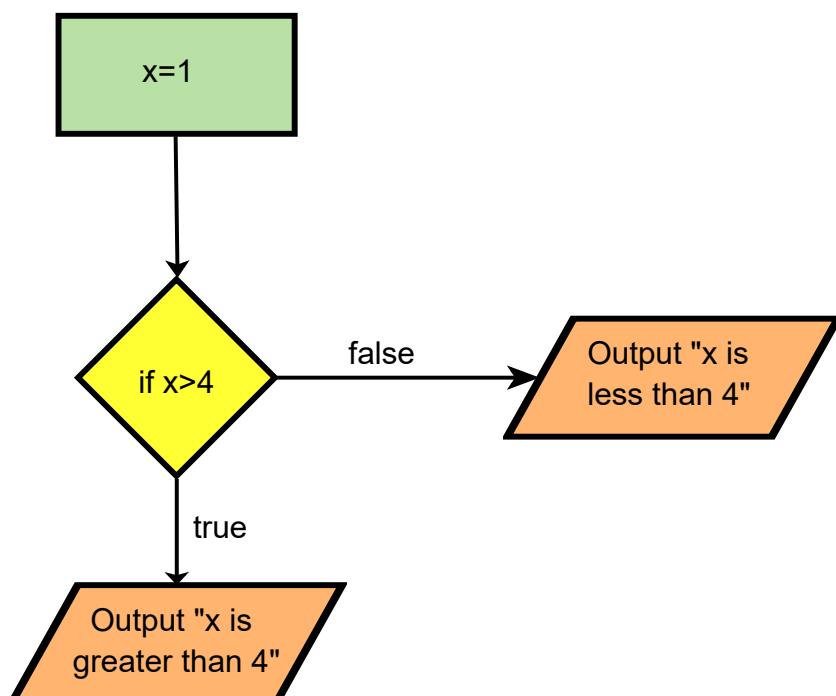
- An `else` statement is defined. The `else` clause does not belong by itself, only directly following an `if` clause.

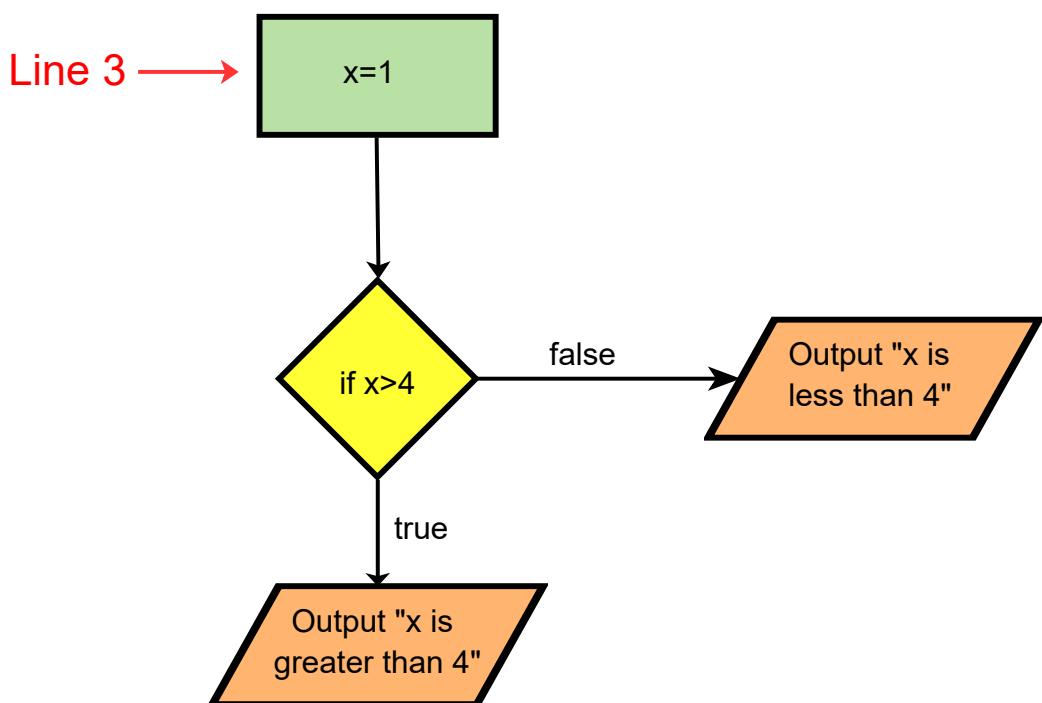
Line 9:

- This is the body of the `else` clause.

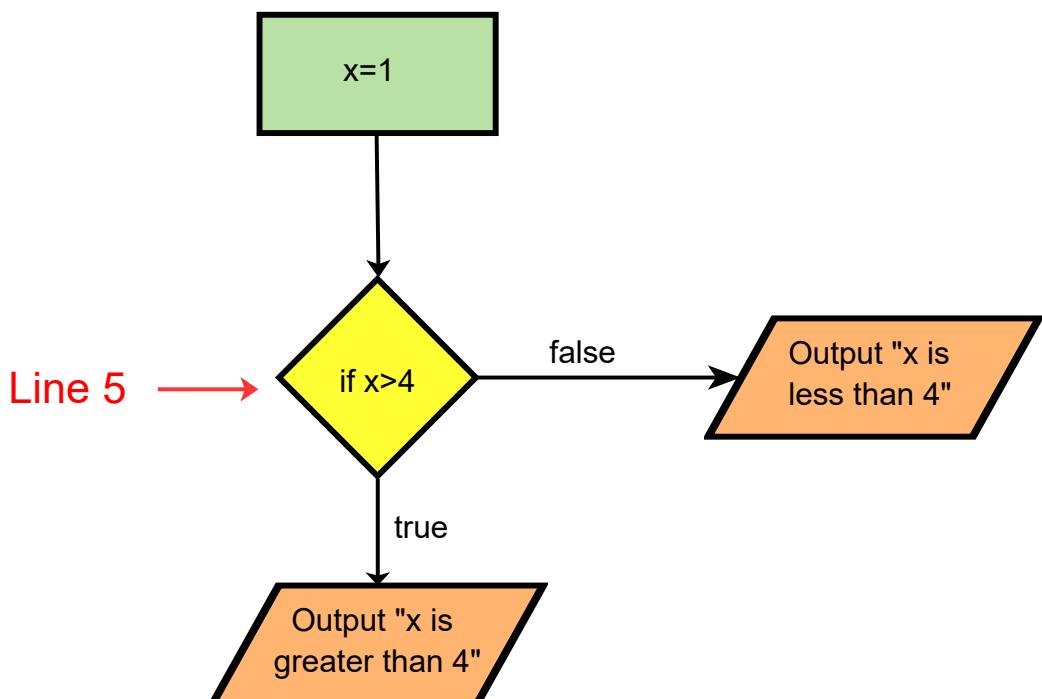
Line 10:

- This *curly brace* is also **essential**; it matches the *opening brace* on line 8 and signals the end of the `else` body.

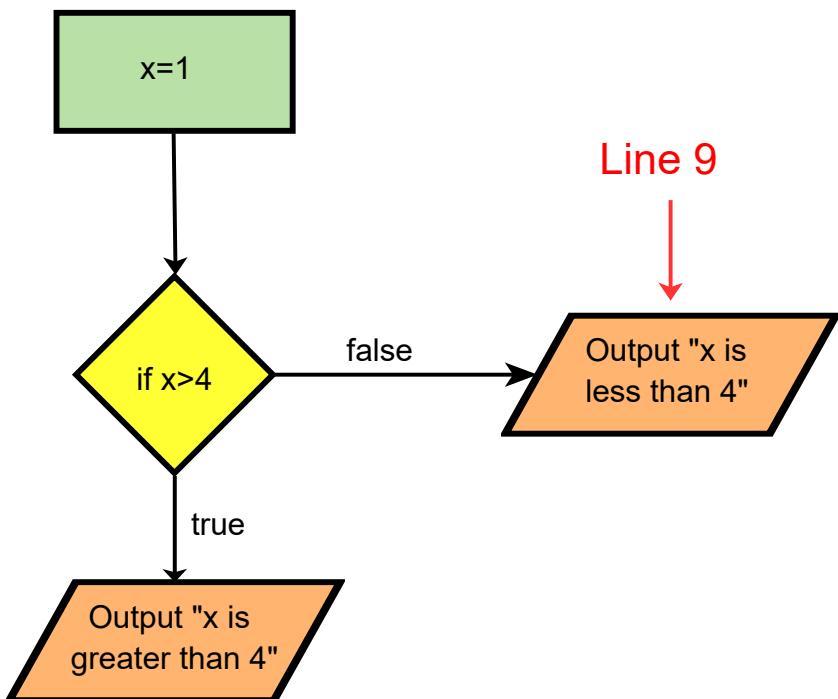




2 of 4



3 of 4



4 of 4



The **if-else if** statement

When multiple conditions need to be checked, and certain operations are related to every condition being true, then the **if-else if** clause comes into play. It will step by step check all conditions until one is true. Then the statements under that condition will be executed, and rest will be ignored. Let's understand this with the below example:

Example

```

class conditional {
    public static void main(String[] args) {
        int x = 4;

        if (x > 4) {
            System.out.println("x is greater than 4");
        }
        else if (x == 4) {
            System.out.println("x is equal to 4");
        }
        else {
            System.out.println("x is less than 4");
        }
    }
}

```

}



Explanation

The detail for the `if` and `else` clause is the same. The `else if` part is described below:

Line 8:

- An `else if` statement is defined. The `else if` clause also does not belong by itself, only directly following an `if` clause.

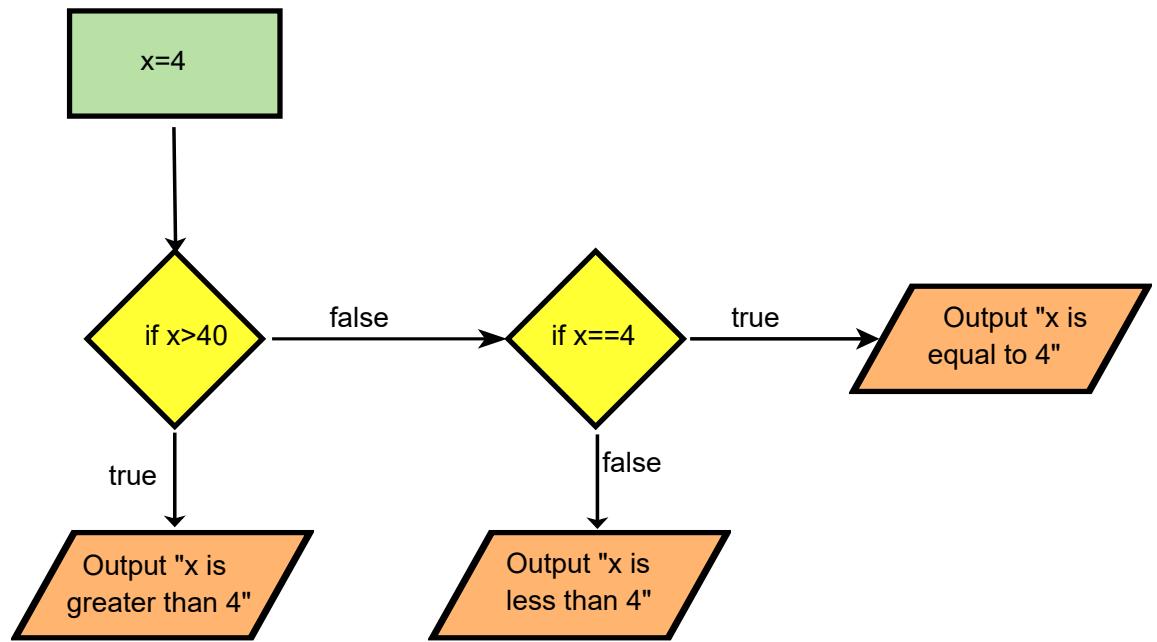
Line 9:

- This is the body of the `else if` clause.

Line 10:

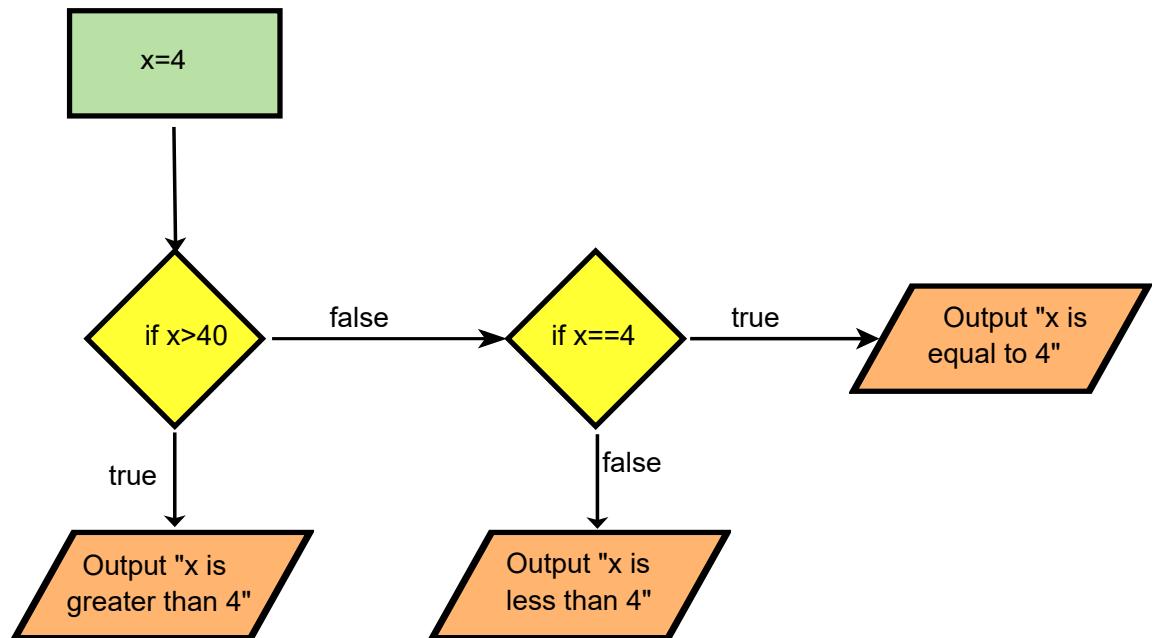
- This *curly brace* is also **essential**; it matches the *opening brace* on line 11 and signals the end of the `else if` body.

Note: It is not mandatory that an `else` clause always follows `else if`. It depends on the problem and the programmer's choice.



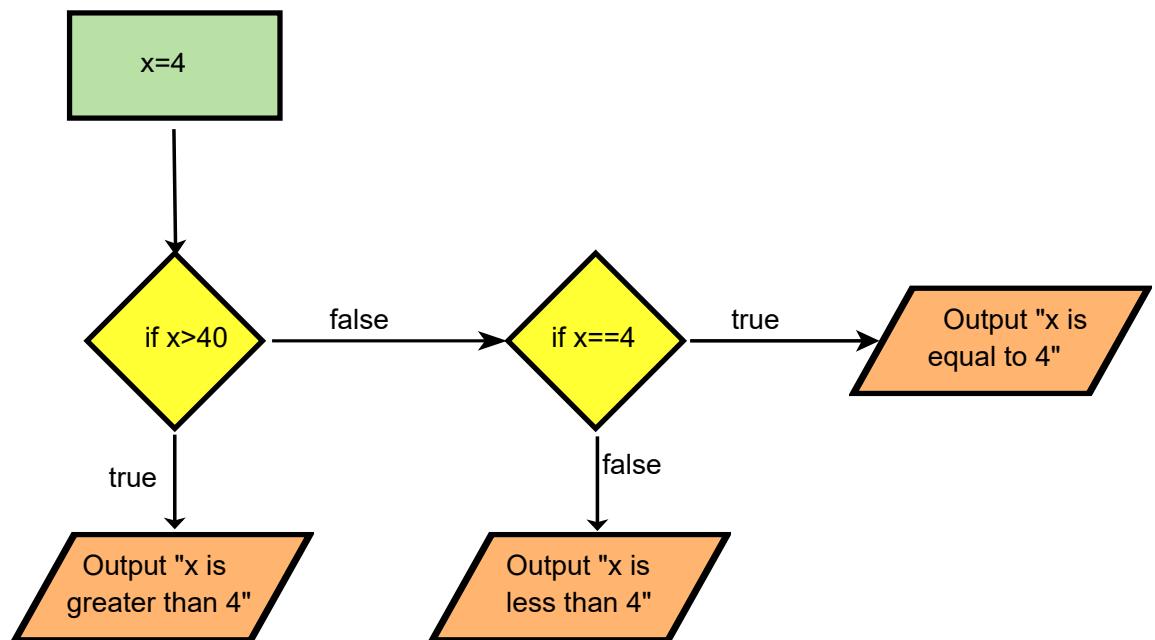
1 of 5

Line 3



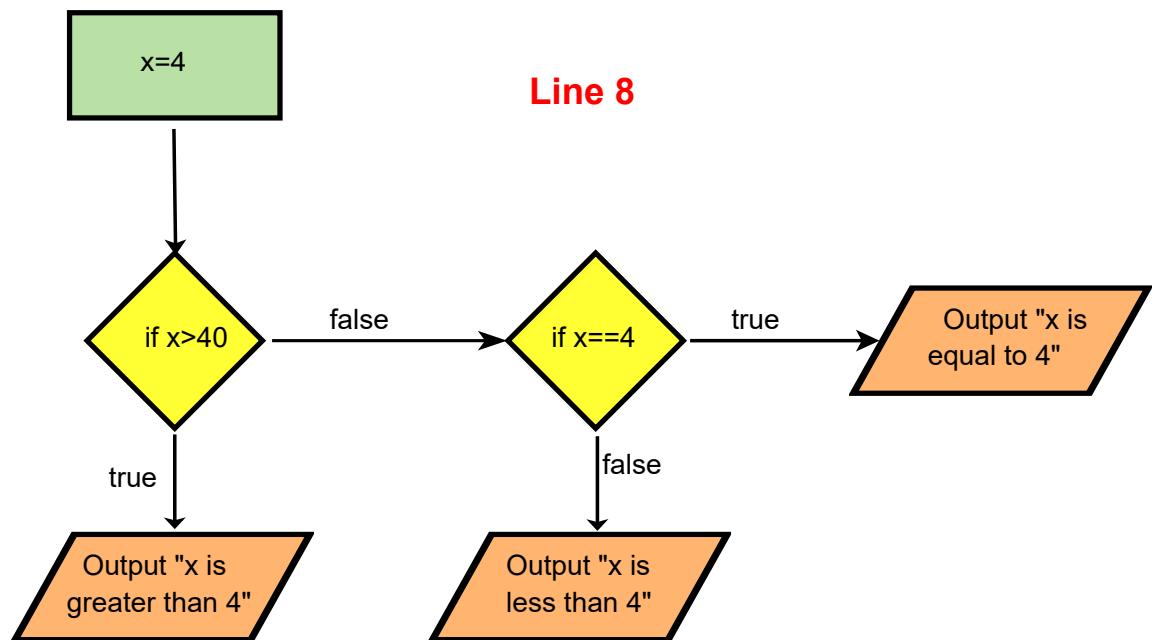
2 of 5

Line 5



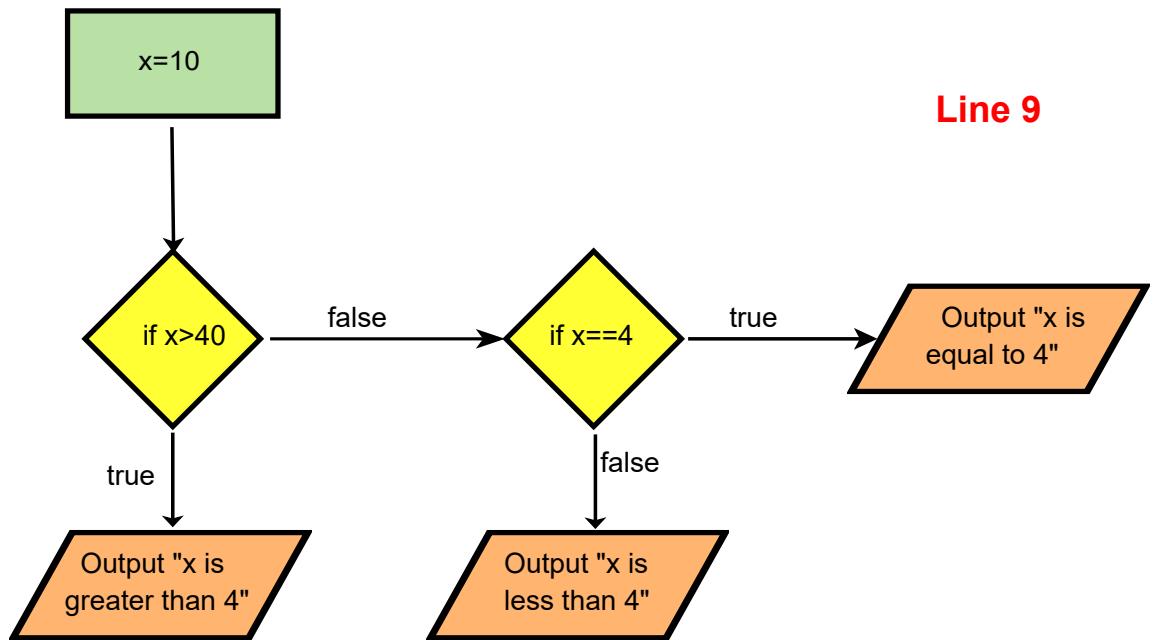
3 of 5

Line 8



4 of 5

Line 9



5 of 5



Now let's take a look at the `switch` statements in the upcoming lesson.

switch Statement

In this lesson, an introduction of the switch statement, its basic syntax, and how it is written using an example is provided.

We'll cover the following ^

- The switch case construct
 - Explanation

The switch case construct

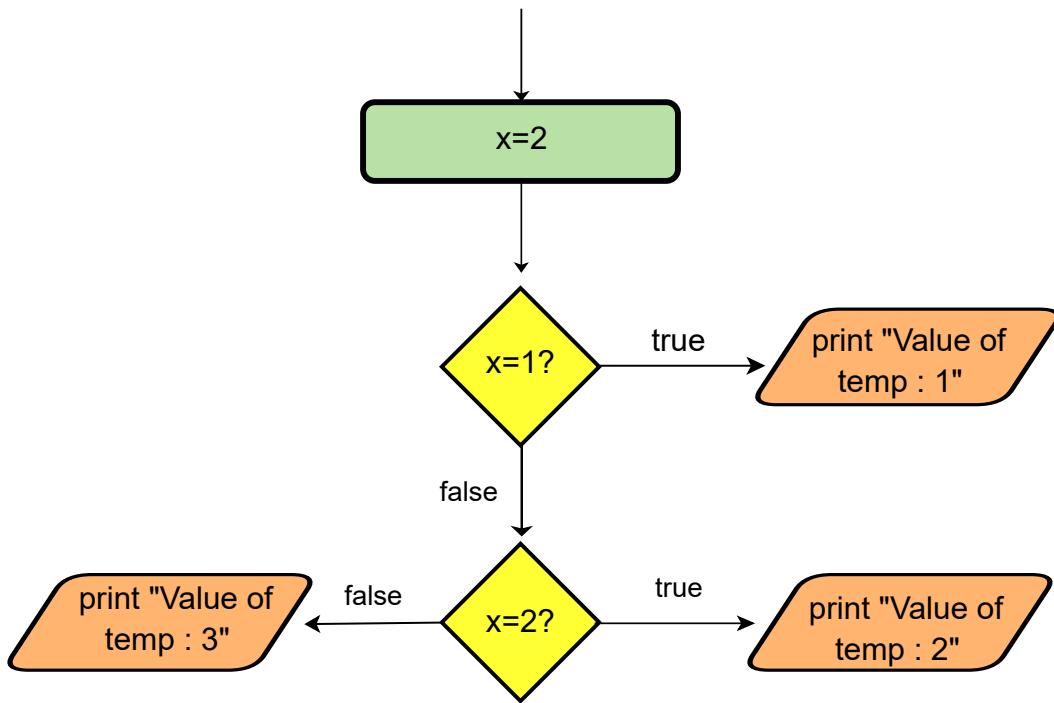
This `switch` clause tests an input variable for equality with any number of cases and then executes the corresponding code.

The `switch` statement is very similar to the `if-else if` statement, described in the previous lesson. The only difference is that the `switch` case construct **only** uses `int`, `short` values, and character *constants* or character *literals*. For this reason, the **switch-case** can come in handy, but it is limited to these circumstances.

```
class switch_statement {
    public static void main(String[] args) {
        int x = 2;
        int temp;

        switch (x) {
            case 1:
                temp = 1;
                break;
            case 2:
                temp = 2;
                break;
            default:
                temp = 3;
                break;
        }
        System.out.println("Value of temp: " + temp);
    }
}
```





Explanation

Here is a line-by-line explanation:

- **Line 6:** The `switch` will take the value of `x` in its input and then compare this value with each case value. If the value of `x` matches the value of the case, then the statements under this case will be executed.
- **Line 7:** Since `x` is not equal to 1. Therefore, statements under this case won't be executed.
- **Line 10:** Since `x` is equal to 2. Therefore, statements under this case will be executed.
- **Line 12:** `break` allow us to come out of the `switch` block immediately. If you don't use the `break`, then all the statements following the correct case will be executed.

Note: `break` is used to break the normal execution of code, skipping everything in a conditional or a loop block and moving to the statement right after the conditional or loop block.

- **Line 13:** The `default` block is similar to the `else` block in a normal conditional. It is chosen if **none** of the *previous* cases matched the value of the *condition* variable.

Sometimes the same statement(s) should be executed in multiple conditions. The following example shows how such a scenario can be implemented using a `switch` case statement.

```
class switch_statement {  
    public static void main(String[] args) {  
        int x = 2;  
        int temp;  
        switch (x) {  
  
            case 1:  
            case 2:  
            case 3:  
                temp = 0;  
                break;  
            case 4:  
                temp = 4;  
                break;  
            default:  
                temp = 5;  
                break;  
        }  
        System.out.println("Value of temp: " + temp);  
    }  
}
```



Try changing the value of `x` above to **1, 3, 4, or any other number** and then see the value of `temp` in output.

Now let's look at the *conditional expressions* in the next lesson.

Conditional Expression

In this lesson, an explanation of what conditional expressions are, how to use them, and their basic syntax is provided.

We'll cover the following

- Introduction
- Syntax
- Sample code

Introduction

A **conditional expression** is a concise way to write the equivalent of an **if-else** statement.

This kind of expression can help to produce highly readable assignment statements fitting onto *one* line of the source code.

Syntax

This is the syntax:

```
( condition ) ? expressionIfTrue : expressionIfFalse;
```



1. The condition is evaluated to see whether it holds true or not
2. If the result is **true**, then only the `expressionIfTrue` is evaluated, i.e., the code block between the `?` and the `:` is executed.
3. If the condition evaluates to **false**, then the resulting value is given by the evaluation of the `expressionIfFalse` branch of the *conditional expression*, i.e., the code block between `:` and the semi-colon.

A common use of conditional expression is to assign values using conditional blocks in a simple and concise manner. Let's look at the code below to understand how this is useful.

Sample code

See the sample code below:

```
class conditional_exp {  
    public static void main(String[] args) {  
        int x = 10;  
        int y = 5;  
        int answer;  
        // Using conditional expression  
        answer = (x > y) ? x : y;  
        System.out.println("Answer using conditional: " + answer);  
        // The above code is equivalent to:  
        // Using the if-else method  
        if (x > y) {  
            answer = x;  
        } else {  
            answer = y;  
        }  
        System.out.println("Answer using if-else: " + answer);  
    }  
}
```



The code given above stores the maximum of two values in the variable `answer`. As you can see, this makes simple conditionals all the easier.

Note: Use the conditional expression only if you feel that it really enhances the readability.

Extra Task:

See if you can come up with a few uses of the conditional constructs you have just learned.

In the next lesson, we will solve a challenge related to the conditional statement.

Challenge 1: Even or Odd

In this challenge, you will implement a Java program to find out if a number is even or odd.

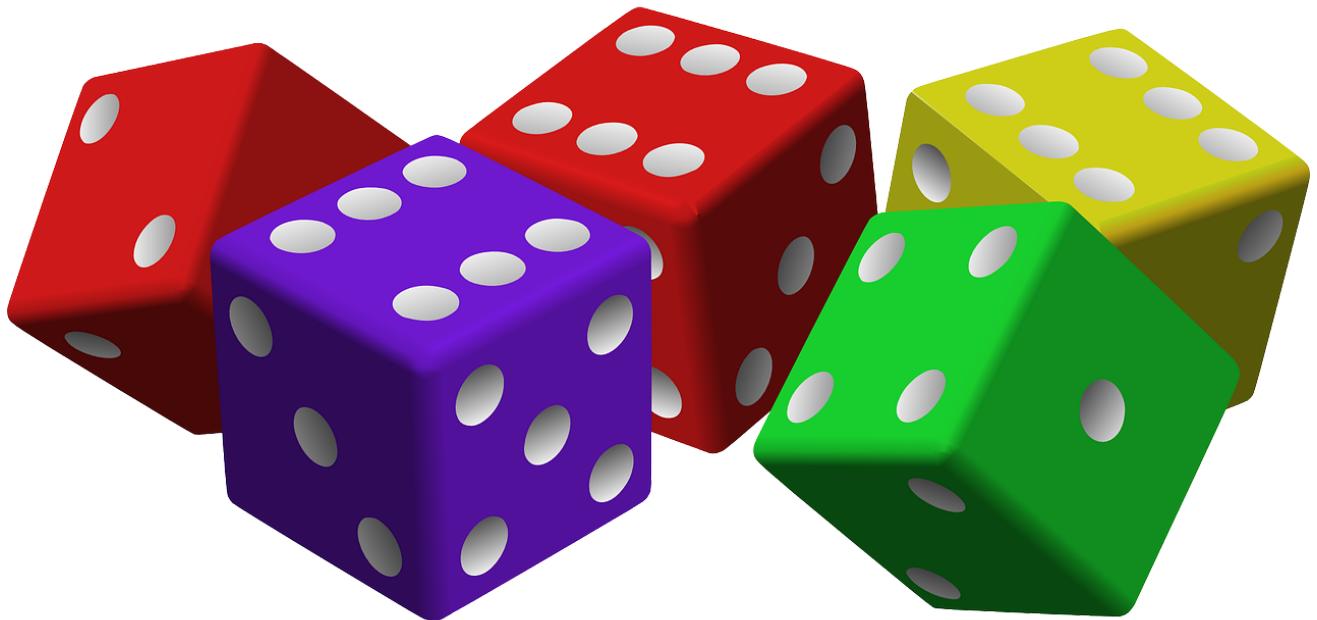
We'll cover the following



- Problem statement
- Coding exercise

Problem statement

A group of friends rolls a dice, and they want to figure out whether the number that comes is an **even** number or an **odd** number. They employ your coding skills to figure it out.



Any given side of a dice can be either 'even' or 'odd'

Coding exercise

Given a number `x`, you should check whether it is even or odd.

- If it is **even**, then store `"even"` in `answer`.

- If it is **odd**, then store "odd" in **answer**.

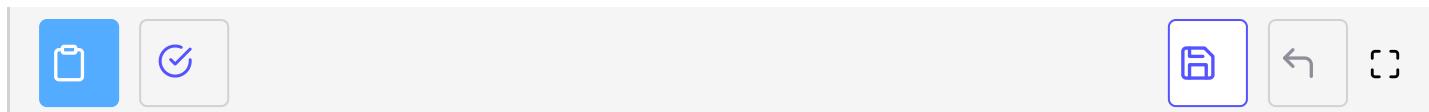
Only write the code where instructed in the snippet below. The **return** statement and the **variable** to be returned are already mentioned for you. Don't worry too much about the return statement for the moment, and just set the value of the **answer** correctly.

Test your code against our cases and see if you can pass them.

The solution is given in case you get stuck, and the next lesson will include a review of the solution, but it is **highly recommended** that you try it yourself first!

Good Luck!

```
class even_odd {  
    public static String evenodd(int x) {  
        String answer = "";  
        // Enter your code here  
        // Store your final result in the variable answer  
  
        /* You do not need to worry too much about the return statement for the  
        moment and just set the value of "answer" correctly*/  
  
        return answer;  
    }  
}
```



In the next lesson, we will review the solution to the above challenge.

Solution Review: Even or Odd

In this review, the solution of the challenge 'Even or Odd' from the previous lesson is provided.

We'll cover the following ^

- Solution: is it even or odd?
- Understanding the Solution

Solution: is it even or odd?

```
class HelloWorld {  
    public static void main( String args[] ) {  
        int x = 3;  
        String answer;  
        if (x % 2 == 0) {  
            answer = "even";  
        }  
        else {  
            answer = "odd";  
        }  
        System.out.println(answer);  
    }  
}
```



Understanding the Solution

Reasoning: An **even** number, when divided by 2, will give a remainder **0**, whereas an **odd** number will give a remainder **1**.

• Line 5

- The evaluation condition for the **if** statement is in this line.
- The **modulus** of **x** is taken with **2** in the **if** condition.
- The **result** of this modulus is compared for **equality** with **0**.
- **Remember:** The expression must be enclosed in round brackets **()**.

• Line 6

- This is the **if-condition-is-true** block.
- Hence this block says that *if the modulus of x, with 2, is equal to 0* then set **answer** to the string **even**.

- **Line 8-10**

- This **else** is followed by curly brackets, **{}**.
 - This is the **if-condition-is-false** block.
 - Hence this block says that *if the modulus of x, with 2, is not equal to 0*, then set **answer** to the string **odd**.
-

In the upcoming lesson, we will solve another challenge related to the conditional statement.

Challenge 2: What Day is it?

In this challenge, you will find out the respective weekday depending upon the given number.

We'll cover the following



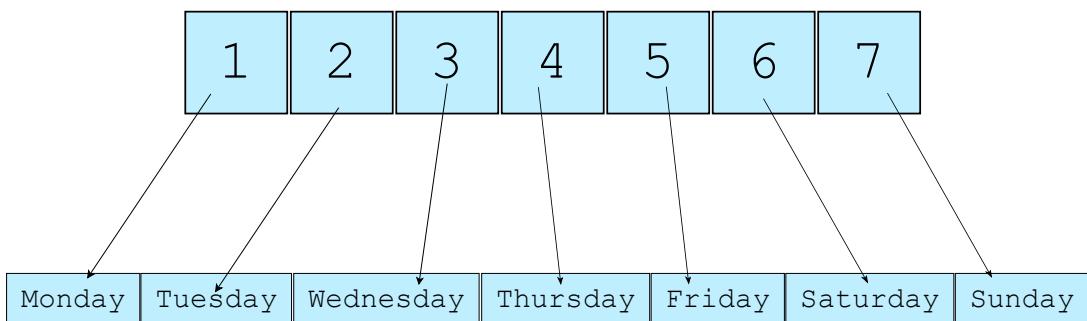
- Problem statement?
- Coding exercise

Problem statement?

There are so many formats in the days of the week. Some people say the first day, some say Monday. Today we will define a format for the day of the week!

Coding exercise

In this coding exercise, you are given a number `x`, and you have to set the corresponding day in the variable `answer`. Which number corresponds to which day is given in the picture below! If the number given does not correspond to any of the numbers in the picture, simply set the `answer` to an "`"invalid input"`".



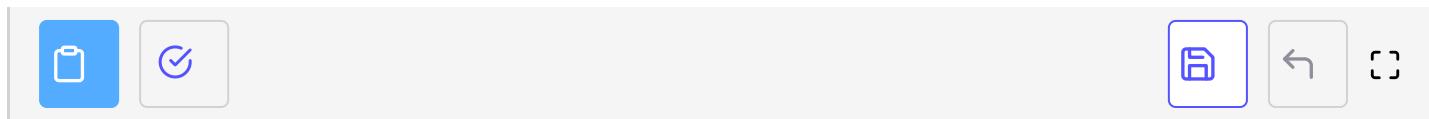
Only write the code where instructed in the snippet below. The **return** statement and the **variable** to be returned are already mentioned for you.

Test your code against our cases and see if you can pass them.

The solution is given in case you get stuck, and the next lesson will include a review of the solution, but it is **highly recommended** that you try it yourself first!

Good Luck!

```
class weekday {  
    public static String week_day(int x) {  
        String answer = "";  
        // Enter your code here  
        // Store your final result in the variable answer  
  
        /* You do not need to worry too much about the return statement for the  
        moment and just set the value of “answer” correctly*/  
  
        return answer;  
    }  
}
```



In the next lesson, we will review the solution to the above challenge.

Solution Review: What Day Is It?

In this review, the solution of the challenge What Day Is It? from the previous lesson is provided.

We'll cover the following

- Solution: did you find the right day?
- Understanding the code

Solution: did you find the right day?

```
class HelloWorld {  
    public static void main( String args[] ) {  
        int x =3;  
        String answer = "";  
        switch (x) {  
            case 1:  
                answer = "Monday";  
                break;  
            case 2:  
                answer = "Tuesday";  
                break;  
            case 3:  
                answer = "Wednesday";  
                break;  
            case 4:  
                answer = "Thursday";  
                break;  
            case 5:  
                answer = "Friday";  
                break;  
            case 6:  
                answer = "Saturday";  
                break;  
            case 7:  
                answer = "Sunday";  
                break;  
            default:  
                answer = "invalid input";  
        }  
        System.out.println(answer);  
    }  
}
```



Understanding the code

Understanding the code

1. Line 5:

- The `switch` statement is used. This switch statement will take the variable `x` in its input.

2. Line 6-8:

- The first `case` is written.
- This states that if the **value of x** is **1**, then set `answer` to the String, **Monday**.
- We used the `break` statement to come out of the `switch` block immediately.

3. Line 9-26:

- Similarly, we have written the cases to the set value of `answer` for the rest of the values of `x` (2 to 7).

4. Line 27-28:

- We have to cater the case when the value of the day number is invalid.
- This is the **default case** in which we have simply set the `answer` to **invalid input**, as is specified in the problem statement.

Let's go through a *quick quiz* to test your understanding.

Quick Quiz!

Quiz to test your understanding of conditional statements in Java.

1

Which of the following evaluates to true?

2

If `x=10`, `y=20`, what will the following code display?

```
if (x>y){  
    System.out.println("x is greater than y");  
}  
else if(x==y){  
    System.out.println("x is equal to y");  
}  
else{  
    System.out.println("x is less than y");  
}
```

3

What will the following code display?

```
int x = 2;
switch (x) {
    case 1:
    case 2:
        System.out.println("First statement");
        break;
    case 3:
        System.out.println("Second statement");
        break;
    case 4:
        System.out.println("Third statement");
    default:
        System.out.println("Default statement");
        break;
}
```

4

What is the output of the following code?

```
int answer;
int x=10, y=15;
answer= (x>=y)?(x+y):(x-y);
System.out.println("Answer: " + answer);
```

[Retake Quiz](#)

This marks the end of the chapter on *conditional statements*. In the next chapter, we'll discuss the interesting concept of *loops* in Java.

while & do-while Loops

In this lesson, an introduction of the while and do-while loops in Java is provided. It uses coding examples to show their implementation and explain their workings.

We'll cover the following ^

- The while loop
- The do...while loop
- When is do-while used?

Loops allow a programmer to execute the same block of code repeatedly.

The while loop

The `while` loop is really the only necessary repetition construct. It will keep running the statements inside its body until some condition is met.

The syntax is as follows:

```
while ( condition ) {  
    //body  
}
```

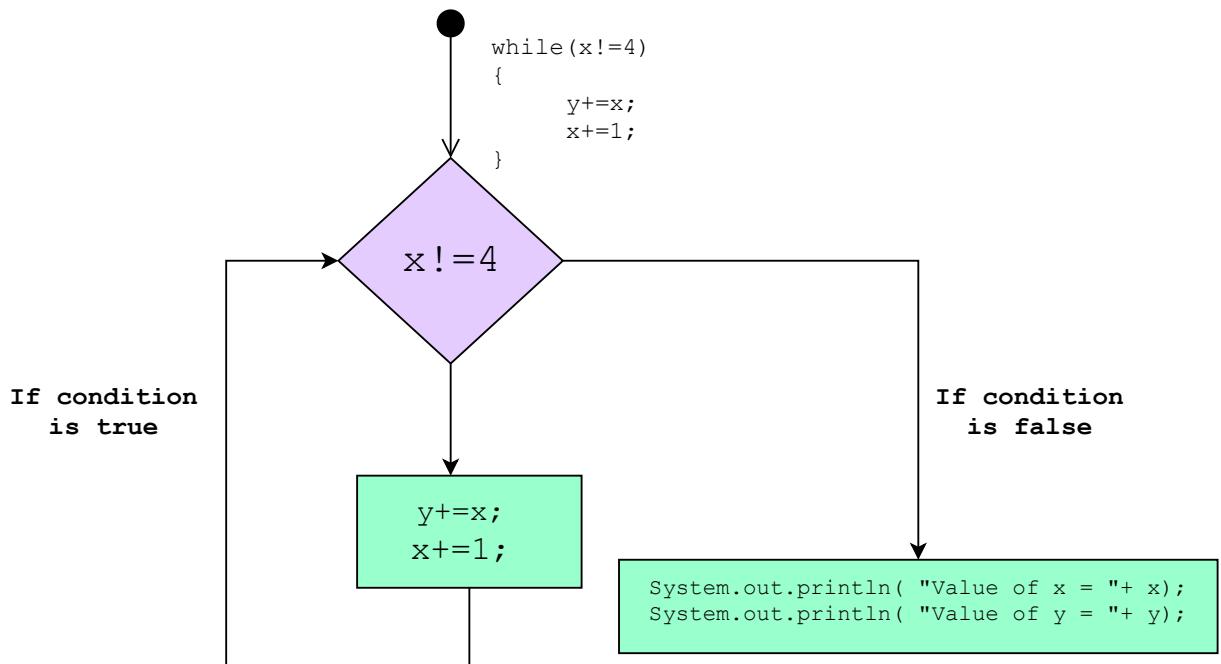
Again, the **curly braces** surrounding the *body* of the `while` loop indicate that *multiple* statements will be executed as part of this *loop*.

Here's a look at the `while` loop code:

```
class HelloWorld {  
    public static void main(String args[]) {  
        int x = 0;  
        int y = 0;  
        while (x != 4) { // the while loop will run as long as x==4 condition is being met  
            y += x;  
            x += 1;  
        }  
        System.out.println("Value of y = " + y);  
        System.out.println("Value of x = " + x);  
    }  
}
```



Below is an *illustration* of the code above to help you better understand the logic.



Flow Chart For While Loop Code

If the `while` loop code looked like this instead, There would be a problem.

You will witness an Execution Timed Out Exception.

```
class Loops {  
    public static void main(String args[]) {  
        int x = 0, y = 0;  
        while (x != 4) //since x is not being changed inside the while loop you will get stuck  
        { // in an infinite loop as the condition will always be met  
            y += x;  
        }  
        x += 1;  
    }  
}
```



In the code above, **line number 8** is still there. It will only be run after the while loop ends. But the while loop doesn't end because the terminating condition is never met.

This is a huge problem because the variable involved in the condition (`x`) does

not change, so the condition will always evaluate to *true*, making this an **infinite loop**.

Note: Be careful with the `while` loop as it has the potential of being endless.

The do...while loop

The `do-while` loop is nearly identical to the `while` loop, but instead of checking the *conditional* statement before the loop starts, the `do-while` loop checks the *conditional* statement **after** the *first* run, then continuing onto another iteration if the condition is `true`.

The *syntax* is as follows:

```
do {  
    //body  
} while (condition);
```

As you can see, it will run the loop **at least** once before checking the conditional statement.

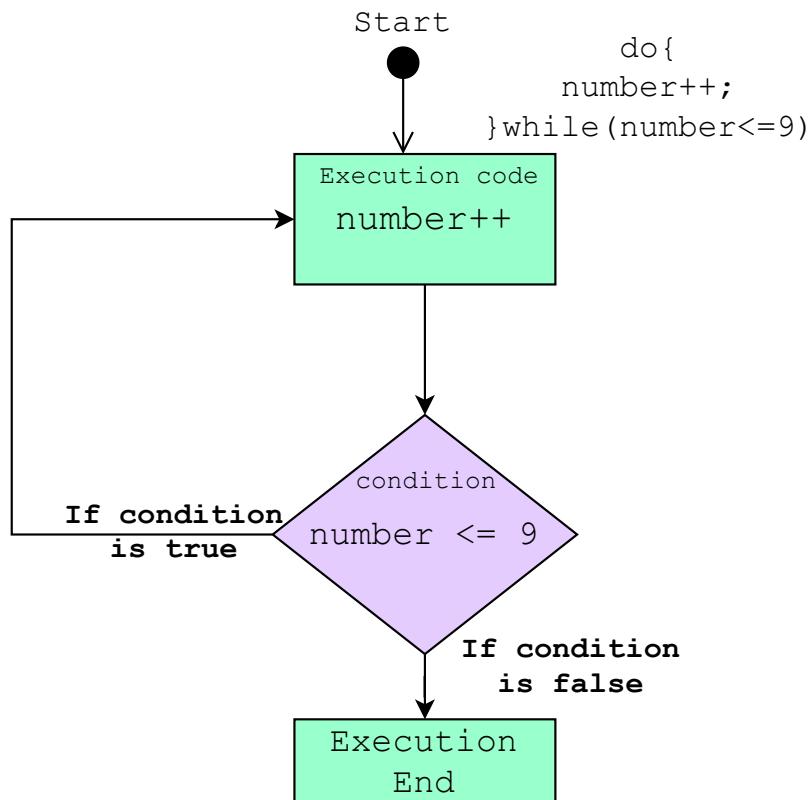
Note: The `do-while` loop is still haunted by *infinite loops*, so exercise the same caution with the `do-while` loop as you would with the `while` loop. Its usefulness is much more limited than the `while` loop, so use this only when necessary.

Below is an example showing how to implement the `do...while` loop in Java.

```
class HelloWorld {  
    public static void main(String args[]) {  
        int number = 5;  
        do {  
            System.out.println("Value of number is: " + number);  
            number++;  
        } while (number <= 9); // the condition is being checked after the first run  
    }  
}
```



Below is an *illustration* of the code above to help you better understand the logic.



Flow Chart For Do While Loop Code

When is `do-while` used?

A `do-while` loop is used where your loop should execute **at least one** time even if the given condition is `false`.

For example, let's consider a scenario where we want to take an *integer* input from the user until the user has entered a **positive** number. In this case, we will use a `do-while` as we have to run loop **at-least-once** because we want input from the user at least once. This loop will continue running until the user enters a **positive** number.

That's all the major stuff you needed to know about the workings of `while` and `do..while` loops in Java. Let's learn about `for` loops in the next lesson.

for Loop

In this lesson, the concept and implementation of for loops and nested for loops in Java is explained.

We'll cover the following

- for loop: syntax
 - What does the for loop do?
 - Nested for loops
 - Example of nested for loop
 - How does nested for loop work?

for loop: syntax

The `for` loop is a loop that lets a programmer control exactly how many times a loop will *iterate*.

The *syntax* is as follows:

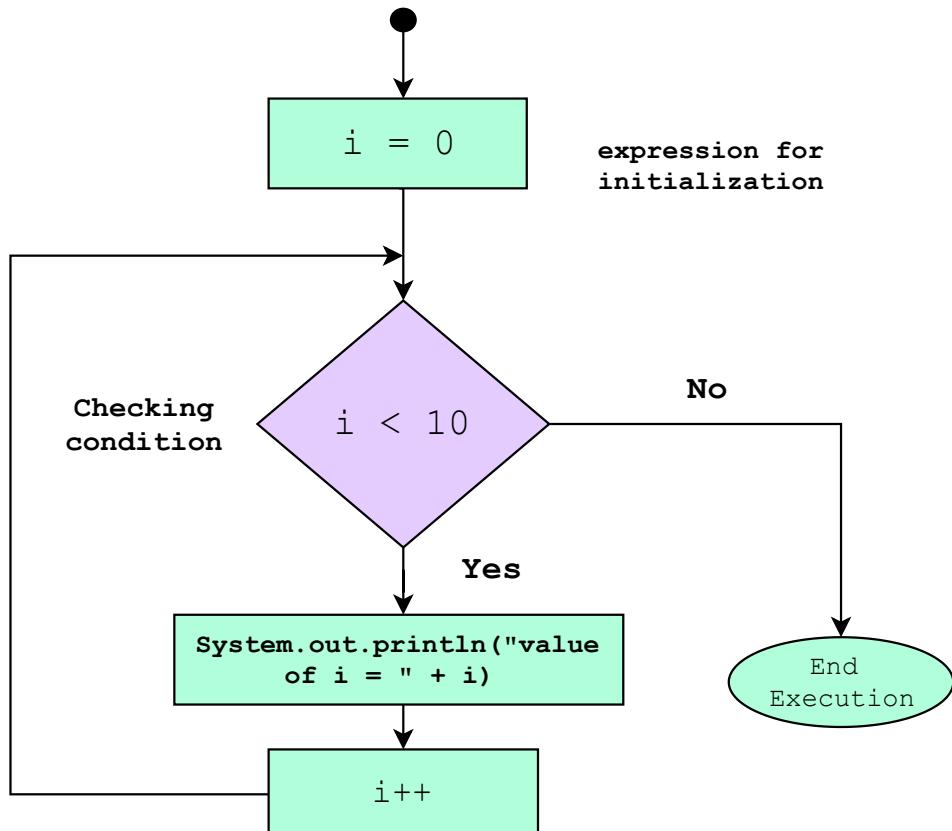
```
for (expression for initialization ; expression for testing ; expression for updating) {  
    //body  
}
```

Here is an example of how the `for` loop works:

```
class Loops {  
    public static void main(String args[]) {  
        for (int i = 0; i < 10; ++i) {  
            // for loop iterates 10 times  
            System.out.println("value of i = " + i);  
        }  
    }  
}
```



Take a look at the illustration below to understand the code above more clearly.



Flow Chart for For Loop

The `for` loop code above and the `while` below are more or less equivalent.

```

class Loops {
    public static void main(String args[]) {
        int i = 0;
        while (i < 10) // while loop runs till i is less than 10 just like in for loop
        {
            System.out.println("value of i = " + i); //prints value of i
            i++;
        }
    }
}

```



What does the for loop do?

- Prior to the *first* iteration, it sets the value of `i` to **0**.
- Next, it tests (like a normal `while` loop) if `i` is *less* than **10**.
- If the conditional statement evaluates to `true`, the body of the loop is

executed, and the program will *print* the value of `i` to the console.

- Once all the statements in loop body are executed, `i` is incremented (by 1), as specified in the update statement, and the conditional is tested again.

So, this loop will run a total of **10** times, printing the “`i`” value each time. You’ve just taught your program to count! **Wow!**

Nested for loops

It is possible to *nest* `for` loops. *Nesting* means including one `for` loop in another `for` loop.

The syntax for a **nested** `for` loop is as follows:

```
for (expression for initialization ; expression for testing ; expression for updating ) {  
    for (expression for initialization ; expression for testing ; expression for updating) {  
        //body  
    }  
    //body  
}
```

Example of nested for loop

Let’s take a look at an example code to understand *nesting* of `for` loops better.

```
class HelloWorld {  
    public static void main(String args[]) {  
        int input = 5;  
  
        System.out.println("How many missiles will you fire?");  
        System.out.println("I will fire: " + input + " missiles");  
  
        for (int i = 0; i < input; i++) { // outer for loop  
            for (int j = 3; j > 0; j--) { // inner for loop  
                System.out.println(j + " ");  
            }  
            System.out.println("Missile " + (i + 1) + " has launched.");  
        }  
  
        System.out.println("All missiles have been launched.");  
    }  
}
```



In a nested `for` loop, for a single value of the **outer** loop, in this case, `i`, the inner (**nested**) `for` loop will iterate over all its values, that is, for example for `i=0` the

inner (*nested*) loop will run from `j = 3` to `j=1`. After this is done, `i` will be

incremented to **1**, and the inner loop will again iterate over all its values against this value of `i`. The process continues until all values of `i` are iterated over.

Look at the **illustration** below, which will help you visualize this and help you understand this concept more clearly.

How does nested `for` loop work?



main

1 of 25



main

input = 5

2 of 25

main

input = 5

How many missiles will you fire?



3 of 25

main

input = 5

How many missiles will you fire?

I will fire: 5 missiles



4 of 25

main

input = 5

How many missiles will you fire?

I will fire: 5 missiles

i = ?



```
main  
input = 5  
How many missiles will you fire?  
I will fire: 5 missiles  
i = 0  
j = ?
```



6 of 25

```
main  
input = 5  
How many missiles will you fire?  
I will fire: 5 missiles  
i = 0  
j = 3
```



7 of 25

```
main  
input = 5  
How many missiles will you fire?  
I will fire: 5 missiles  
i = 0  
j = 3
```

3



8 of 25

```
main  
input = 5  
How many missiles will you fire?  
I will fire: 5 missiles  
i = 0  
j = 3
```

3



9 of 25

```
main  
input = 5  
How many missiles will you fire?  
I will fire: 5 missiles  
i = 0  
j = 2
```

3

10 of 25

```
main  
input = 5  
How many missiles will you fire?  
I will fire: 5 missiles  
i = 0  
j = 2
```

3 2

11 of 25

```
main
```

```
input = 5
```

```
How many missiles will you fire?  
I will fire: 5 missiles
```

```
i = 0
```

```
j = 1
```

3 2

12 of 25

```
main
```

```
input = 5
```

```
How many missiles will you fire?  
I will fire: 5 missiles
```

```
i = 0
```

```
j = 1
```

3 2 1

13 of 25

```
main
```

```
input = 5
```

```
How many missiles will you fire?  
I will fire: 5 missiles
```

```
i = 0
```



3 2 1

14 of 25

```
main
```

```
input = 5
```

```
How many missiles will you fire?  
I will fire: 5 missiles
```

```
i = 0
```

3 2 1



Missile 1 has launced.

15 of 25

```
main  
input = 5  
How many missiles will you fire?  
I will fire: 5 missiles  
i = 1
```



16 of 25

```
main  
input = 5  
How many missiles will you fire?  
I will fire: 5 missiles  
i = 1  
j = 3
```



17 of 25

```
main
```

```
input = 5
```

```
How many missiles will you fire?  
I will fire: 5 missiles
```

```
i = 1
```

```
j = 3
```

3



18 of 25

```
main
```

```
input = 5
```

```
How many missiles will you fire?  
I will fire: 5 missiles
```

```
i = 1
```

```
j = 2
```

3



19 of 25

```
main  
input = 5  
How many missiles will you fire?  
I will fire: 5 missiles  
i = 1  
j = 2
```

3 2



20 of 25

```
main  
input = 5  
How many missiles will you fire?  
I will fire: 5 missiles  
i = 1  
j = 1
```

3 2



21 of 25

```
main  
input = 5  
How many missiles will you fire?  
I will fire: 5 missiles  
i = 1  
j = 1
```

3 2 1



22 of 25

```
main  
input = 5  
How many missiles will you fire?  
I will fire: 5 missiles  
i = 1
```

3 2 1



Missile 2 has launched.

23 of 25

```
main  
input = 5  
How many missiles will you fire?  
I will fire: 5 missiles  
i = 1
```



These Iterations will run for all the values of i till i = 4

24 of 25

```
main  
input = 5  
How many missiles will you fire?  
I will fire: 5 missiles
```

3 2 1 Missile 1 has launched.
3 2 1 Missile 2 has launched.
3 2 1 Missile 3 has launched.
3 2 1 Missile 4 has launched.
3 2 1 Missile 5 has launched.

All missiles have been launched.



The final result after the program has run for all values

25 of 25



Note: Just like the `for` loop, any other loop can be nested in the same way.

Exciting, right? Now that the concept of `for` loops and *nested for* loops are clear, let's look at infinite loops in the next lesson.

Let's look at infinite loops in the next lesson.

Infinite Loops

In this lesson, an explanation of how infinite loops might emerge in a loop is provided.

We'll cover the following



- Emergence of infinite loops
 - Uses of infinite loops
 - Example of infinite loop

Emergence of infinite loops

One common programming mistake is to create an **infinite** loop. An **infinite** loop refers to a loop, which under certain valid (or at least plausible) input, will never **exit**.

Note: Beginning programmers should be careful to examine all the *possible* inputs into a *loop* to ensure that for each such set of inputs, there is an **exit** condition that will eventually be reached.

Compilers, debuggers, and other programming tools can only help the programmer so far in detecting **infinite** loops.

Note: In the fully general case, it is not possible to automatically detect an infinite loop. This is known as the **halting** problem.

While the halting problem is not solvable in the fully general case, it is possible to determine whether a loop will **halt** for some *specific* cases.

Uses of infinite loops

The conditions when infinite loops can be extremely useful are the following:

- When continuous input is required to the program or system, and it has to

respond accordingly without stopping.

- When we do not know the exit condition, some calculations need to be performed inside the loop, which will decide when to exit or terminate the loop.

Example of infinite loop

Below is an example of an *infinite* loop.

Note: You will get an **error** when you try to run the code below because an infinite number of print statements cannot be shown here.

```
class HelloWorld {  
    public static void main(String args[]) {  
        boolean a = true;  
        while (a) { // the while condition will always be met as will always return true  
            System.out.println("Infinite loop");  
        }  
    }  
}
```



In a normal case, the code above will print “**Infinite loop**” without stopping because the condition statement in the **while** loop will always evaluate to **true**. There is no point at which it’ll return **false** hence we’ll get stuck in an *infinite* loop, and the code will execute forever.

Note: The generation of infinite loops is not constrained to only **while** loops. It can very well occur in other looping structures too.

Next up, some challenges are provided to test the understanding of loops.

Challenge 1: Multiplication Table of a Number

In this challenge, you have to print the multiplication table of a number up to 10.

We'll cover the following



- Problem statement
- Example
- Coding Challenge

Problem statement

In this challenge, you will store all the values computed from a multiplication in the string `answer`. You can use the `valueOf()` method of string to convert `int` value into a string.

Example

Input: `int num = 5`

Here's an illustration showing what the string `answer` should have stored in it at the end of the `while` loop for the above-mentioned `input`.

5 10 15 20 25 30 35 40 45 50

Multiplication Values Stored in String 'answer'

Your output should be in the format given above. Otherwise, your code will not pass the test cases.

Note: Just like it is shown in the picture above, the values in the `string` should have spaces in between them. You can add space by simply adding quotation marks with space in between them to your string, like this: " ".

marks with space in between them to your string, like this: . . .

Coding Challenge

Only write the code where instructed in the snippet below. You need to store your final result in the variable `answer`. The **return** statement and the **variable** to be returned are already mentioned for you. Don't worry too much about the return statement for the moment, and just set the value of the `answer` correctly.

Write your code below. It is recommended that you try solving the exercise yourself before viewing the solution.

Good Luck!

```
class MultiplicationTable {  
    public static String test(int num) {  
        String answer = "";  
        // Enter your code here  
        // Store your final result in the variable answer  
  
        /* You do not need to worry too much about the return statement for the  
        moment and just set the value of “answer” correctly*/  
  
        return answer;  
    }  
}
```



In the next lesson, we will review the solution to the above challenge.

Solution Review: Multiplication Table of a Number

In this review, the solution of the challenge 'Multiplication Table of a Number' from the previous lesson is provided.

We'll cover the following ^

- Solution
- Explanation

Solution

```
class HelloWorld {  
    public static void main( String args[] ) {  
        int num = 5;  
        int i = 1;  
        int table;  
        while (i <= 10) {  
            table = num * i;  
            answer += String.valueOf(table) + " ";  
            System.out.println(num + " x " + i + " = " + table);  
            i++;  
        }  
        System.out.println(answer);  
    }  
}
```



Explanation

To calculate the table of the given number, we will multiply the given number with **1, 2, up to 10**.

- `int i = 1;` `i` is initialized to 1 as we will first multiply the given number with **1**.
- `int table;` stores the result, which is calculated at each step.
- `while(i <= 10)` while loop will iterate from `i = 1` to `i = 10` as we have to multiply the given number with the values from **1 to 10**. We don't need to

multiply the given number with **11**. Therefore, `while` loop will stop when the `i` reaches **11**.

- `table = num*i;` Let's say `num = 5`, and `i = 1`, so the answer will store `5 x 1`, i.e., `5`
 - `i++;` Increment `i` for the next iteration. The values will range `1,2,3,...,10`.
-

In the next lesson, we will solve another challenge related to loops.

Challenge 2: Calculating the First 'n' Fibonacci Numbers

In this challenge, you have to calculate the first 'n' fibonacci numbers.

We'll cover the following ^

- Problem statement
 - Example

Problem statement

In this exercise, you have to calculate the first 'n' fibonacci numbers. The fibonacci series is:

0, 1, 1, 2, 3, 5, 8, 13,

The first two fibonacci numbers are 0 and 1. Every subsequent number in the fibonacci sequence is the sum of the previous two.

You will be given a number `n`, and your code should store first `n` fibonacci values in the string `fib`. You can use the `valueOf()` method of string to convert `int` value into a string.

Example

If the value of `n` is 6. Then the `fib` should store the following sequence of numbers:

```
0 1 1 2 3 5
```

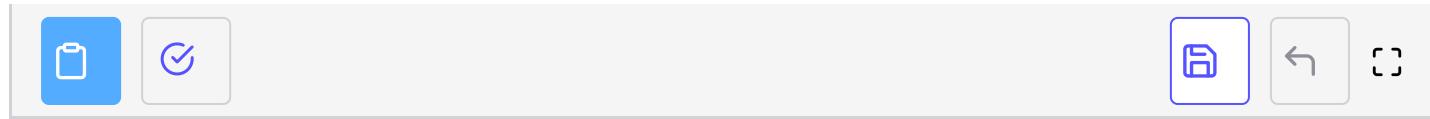
Note Just like it is shown above, the values in the **string** should have spaces in between them. You can add space by simply adding quotation marks with space in between them to your string, like this: " ".

Only write the code where instructed in the snippet below. You need to store your final result in the variable `fib`. The **return** statement and the **variable** to be returned are already mentioned for you. Don't worry too much about the return statement for the moment, and just set the value of the `fib` correctly.

Write your code below. It is recommended that you try solving the exercise yourself before viewing the solution.

Good Luck!

```
class Fibonacci {  
    public static String test(int n) {  
        String fib = "";  
  
        // Enter your code here  
        // Store your final result in the variable fib  
  
        /* You do not need to worry too much about the return statement for the  
        moment and just set the value of "fib" correctly*/  
  
        return fib;  
    }  
}
```



In the next lesson, we will review the solution to the above challenge.

Solution Review: Calculating the first 'n' Fibonacci numbers

In this review, the solution of the challenge " Calculating the First 'n' Fibonacci Numbers" from the previous lesson is provided.

We'll cover the following

- Solution
- Explanation

Solution

```
class HelloWorld {  
    public static void main( String args[] ) {  
        String fib = "";  
        int n = 6;  
        int first = 0, second = 1, fibonacci = 0;  
        System.out.println("Fibonacci Series upto " + n + " Terms ");  
  
        for (int c = 0; c < n; c++) {  
            if (c <= 1) {  
                fibonacci = c;  
                fib += String.valueOf(fibonacci) + " ";  
            } else {  
                fibonacci = first + second;  
                first = second;  
                second = fibonacci;  
                fib += String.valueOf(fibonacci) + " ";  
            }  
            System.out.println(fibonacci + " ");  
        }  
    }  
}
```



Explanation

Fibonacci is famous and simplest of the series that you have learned to calculate along with the implementation of loops. In the fibonacci series, each number is the sum of the previous two numbers. Therefore, to find the next number, you must keep track of the last two numbers. Let's break down the code above for you:

keep track of the last two numbers. Let's break down the code above for your better understanding.

- `int first = 0, second = 1, fibonacci = 0;` Declare variables to store the second last, last and the recently calculated *fibonacci* number. Since the first two fibonacci numbers are 0 and 1. Therefore, we have set the `first` and `second` to **0** and **1** respectively.
 - `for (int c = 0; c < n; c++)` since we have to calculate the `n` fibonacci numbers. Therefore, the loop will iterate from **0 to n-1**
 - `if (c <= 1)` Boundary Check to restrict extra calculations. If `c` is less than equal to 1, it will simply set the `fibonacci` to `n` as the first two numbers are 0 and 1, respectively.
 - `fibonacci = first + second;` For `c >= 2`, we will apply the series formula, as the current number is equal to the sum of the previous two elements.
 - `first = second; second = fibonacci;` Update the first and second values for the next iteration.
-

Let's solve another mind-crunching exercise in the next lesson.

Challenge 3: Pyramid Printing by Using 'for' Loop

In this challenge, you have to print half a pyramid for a given number of rows.

We'll cover the following



- Problem statement
 - Example

Problem statement

Write code that draws half a **pyramid** using the `#` character.

You are given an **integer** variable `rows` as **input**, and you have to *print* the pyramid with that number of *rows* displaying `#`.

Hint: Use `for` loops to *implement* the solution.

Example

Input

- `rows` is equal to 5.

Then:

Output

Number of Rows = 5

```
#  
# #  
# # #  
# # # #  
# # # # #
```

Expected Output

From the above figure, it is obvious that there should be a space after every hash character.

Note: `System.out.print()` prints the content and does not add the new line. Whereas `System.out.println()` prints content and add a new line after it. We can also use `\n` to add a new line on the console.

Write your code below. It is recommended that you try solving the exercise yourself before viewing the solution.

Good Luck!

```
class PrintPyramid {  
    public static void test(int rows) {  
  
        //write the code for making and printing the pyramid here  
        //use " " to add space between # in pyramid  
    }  
}
```



In the next lesson, we will review the solution to the above challenge.

Solution Review: Pyramid Printing by Using 'for' Loop

In this review, the solution of the challenge 'Pyramid Printing by Using 'for' Loop' from the previous lesson is provided.

We'll cover the following

- Solution
- Explanation

Solution

```
class HelloWorld {  
    public static void main( String args[] ) {  
        int rows = 5;  
        for (int i = 1; i <= rows; ++i) {  
            for (int j = 1; j <= i; ++j) {  
                System.out.print("# ");  
            }  
            System.out.println();  
        }  
    }  
}
```



Explanation

Here is a breakdown of the above code for your thorough understanding.

- `for(int i = 1; i <= rows; ++i)` Outer for loop to iterate till the total number of rows. The outer for loop will terminate when the condition `i <= rows` is no longer true.
- `for(int j = 1; j <= i; ++j)` Inner for loop to print number of *hash* characters in each row equal to the row number!
- `System.out.print("# ");` Prints each *hash* character.
- `System.out.println();` Adds new line after every row.

Let's wrap up this chapter by solving a quiz in the upcoming lesson.

Quick Quiz!

Here is a quick quiz to check your understanding of loops.

1

Which of the following is not a nested loop?

2

The `break` statement is used to exit a loop.

3

Which of the following statements about the `while` loop is NOT true?

4

Which of the following `for` loop is infinite?

5

What is the final value of `i` when the following code is run?

```
for(int i = 0; i < 5; i++) {  
    System.out.println(i);  
}
```

6

Which of the following best describes when will the line `i=i+1` execute?

```
while(i<=10){  
    i = i+1;  
}
```

7

A `for` loop can be converted to a `while` loop in usual cases

8

Infinite loops terminate upon meeting the boundary condition.

9

Which of the Java loop is not a pretest loop?

10!

`while` loops can not be nested.

[Retake Quiz](#)

Well, this marks the end of the chapter on **loops**. Let's discuss methods in java in the upcoming chapter.

Methods in Java

In this lesson, the basics of Methods in Java are explained.

We'll cover the following



- Understanding syntax

Methods are how we **communicate** with objects in Java. We *invoke* or *call* a method, i.e.; we are asking the object to carry out a task *through* the method. Hence, a **method** is a mechanism that allows us to *implement functionality* or *attribute behavior* to objects. Methods follow a particular syntax, as shown in the snippet below.

```
ClassOne{  
    visibility ReturnType Methodname(parameter_one, parameter_two){  
        //method body  
        return returnVariable;  
    }  
}
```



Understanding syntax

Visibility

Visibility is the scope of a Method, i.e., how accessible it is from different places in the programs. There are **four types** within visibility:

- public: The method can be called *anywhere* in the application.

For now, we will consider only public methods, and later on, when we do understand classes and packages, we'll talk about other options.

ReturnType

This defines whether the method **returns** a value or not. If it does, then this is where the **data type** of the returning variable is mentioned in the declaration. If the method does not return any value, then we will skip this part.

the method *does not return* anything then the type is set to **void**. Otherwise, it can return **primitive** as well as special data types.

Methodname

This is where you define the name for your method. Like a variable identifier, this name **should** be descriptive of the functionality of the method. For instance, the **substring** method for Strings that we have learned in this [lesson](#). This method has an explanatory name that identifies what this method does. Imagine if it had a vague name like **partOfString** or **smallString** which could be interpreted to multiple meanings, it would be so confusing!

Parameters

The ‘Methodname’ is followed by two round brackets, `()`. Within these round brackets, the parameters are passed to the method. These parameters are used within the method body, which is enclosed in curly brackets, `{ }`. Parameters within the brackets `()` are separated by commas and can have multiple data types.

returnVariable

The ‘returnVariable’ is the variable whose value is to be returned outside the method body. This variable must have the *data type* of the returnType signified in the method declaration line.

Let’s look in detail at parameters being passed to a method in the next lesson.

Parameters and Return Types in Methods

In this lesson, an explanation of how to pass parameters to the methods is explained.

We'll cover the following ^

- Primitive type
- Reference type
 - Understanding the code

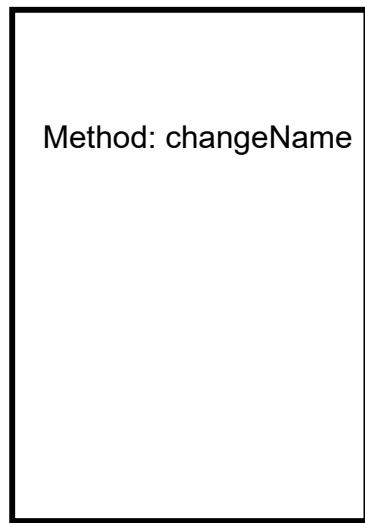
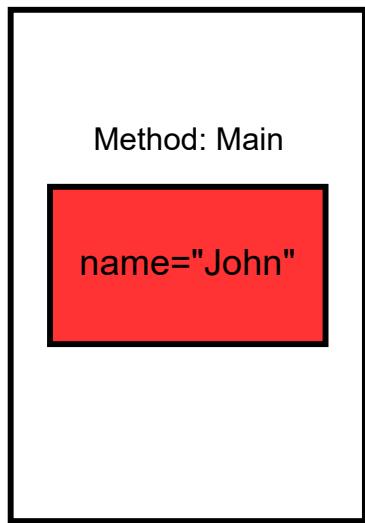
Methods take in **two** types of parameters. One is the **primitive data type** and the other is the **reference data type**. We will be looking in detail at both types of parameters and how each of them is used for a different purpose.

Primitive type

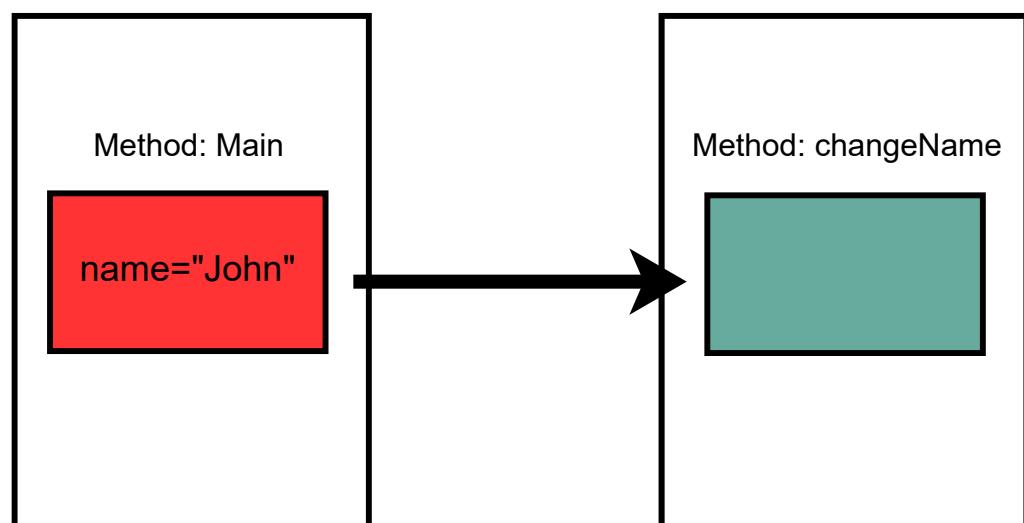
These types of parameters are said to be *passed by value*. The value from the calling method IS copied to a variable in the called method. What this signifies is that once a parameter is passed or given to a method as an argument, there is no relation between the object i.e. the parameter that is present outside the method body, and the modification that is happening inside the method to that parameter. This can be a bit hard to understand.

The code and diagram below explain this concept.

```
class prim_type {  
    public static void main(String[] args) {  
  
        String name = "John";  
        changeName(name);  
        System.out.print(name);  
    }  
  
    public static void changeName(String n) {  
  
        n = "Doe";  
        System.out.println(n);  
    }  
}
```

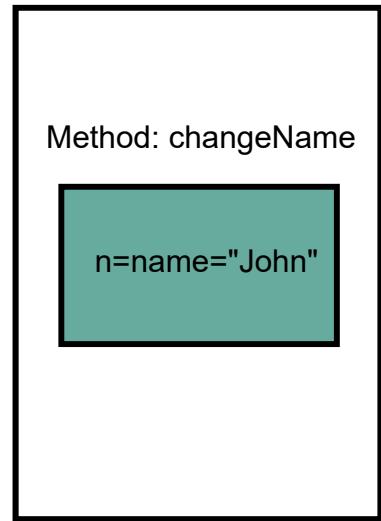
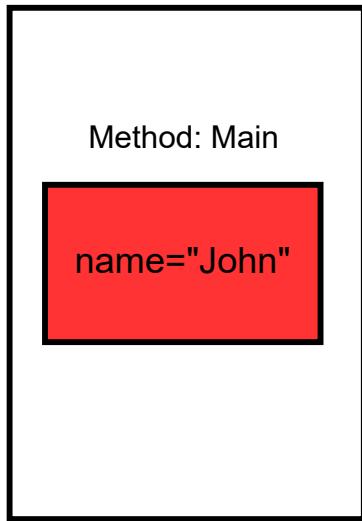


1 of 4



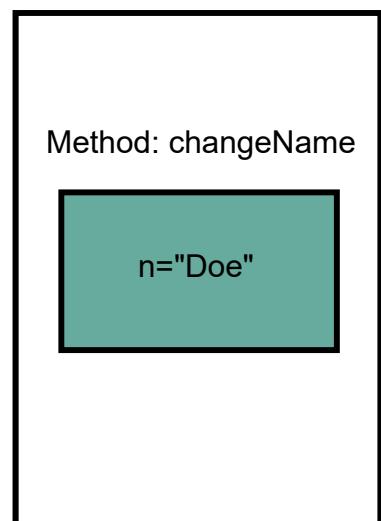
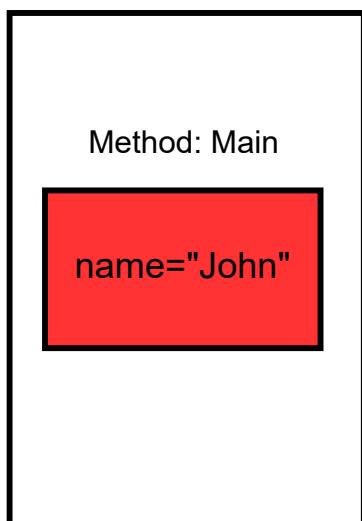
Giving 'name' as parameter to changeName

2 of 4



The variable within the method is n, this is a variable only relevant to the method.

3 of 4



n="Doe" only within the method, it has no link to the variable "name"

4 of 4



In the above code, we could have passed a string literal instead of a string variable in the method call.

Hence, we see that printing the variable **n** in the method gives the result as **Doe**, whereas outside the defined method, the variable value is retained as **John**.

Note: The method call must match the method definition.

Reference type

In this type of parameter passing, the **reference to the object** is passed by value. This **does not** mean that the object itself is passed rather a **link** between the variable name and the object instance is passed. An **object instance** is a variation of an object within the method. This link is called an **object reference**, which simply points to the address where the object instance is kept in memory.

One analogy that we can use with value and reference parameters is a one-way and a two-way pipe between the calling and the called method. With value types, there is a one-way of passing value between them. With reference types, changes can be propagated back to the calling method. With reference types actually it is one variable that both are accessing.

```
class car {  
    public String colour;  
    public car() {};  
    public car(String col) {  
        this.colour = col;  
    }  
    public String getColour() {  
        return this.colour;  
    }  
    public static void main(String[] args) {  
        car newCar = new car("Pink");  
        System.out.print("Car colour is: " + newCar.getColour() + "\n");  
        setColourBlue(newCar);  
        System.out.print("Car colour is: " + newCar.getColour() + "\n");  
    }  
  
    public static void setColourBlue(car c) {  
        c.colour = "Blue";  
    }  
}
```



Understanding the code

This code snippet is slightly different from what we have seen so far, so let's look at it in a little bit more detail.

Line 3: A `String colour` is created which will store the car colour.

Lines 5-10: Helper methods are created within the class to `set` and `get` the car colour.

Lines 11-16:

- A `main` method creates a new car object and the initial colour of the car is *set* as `Pink`.
- The method that returns the car colour is called.
- Then the car colour is set to `Blue` using an external method which is explained below (line 18-20).
- The car colour is then printed again, to see if the changes within the above-mentioned method are **applicable** to the **actual object**.

Lines 18-20

- A new method is declared that takes in the `car` object as an argument.
- Then uses the **colour String** belonging to the `car class` and *modifies* its value to `Blue`.
- Note that the function **does not** return anything.

In the next lesson, we will discuss return parameters in methods.

Return Parameters in Methods

In this lesson, we will discuss return parameters in methods.

We'll cover the following

- Return parameters
 - Returning multiple values
 - Modify object by calling methods
 - Create a custom object

Return parameters

Return parameters are ones that the method will **send back** to the class from where it was called. To understand, let's look at the diagram below which shows a function that changes a *String* value and returns the new **modified** String.

```
class car {  
  
    public String colour;  
    public car() {};  
    public car(String col) {  
        this.colour = col;  
    }  
    public String getColour() {  
        return this.colour;  
    }  
    public void changeColourToBlue() {  
        this.colour = "Blue";  
    }  
    public static void main(String[] args) {  
        car newCar = new car("Pink");  
        System.out.print("Car colour is: " + newCar.getColour() + "\n");  
        newCar.changeColourToBlue();  
        System.out.print("Car colour is: " + newCar.getColour() + "\n");  
    }  
}
```



This defines the return type of the variable

```
public String changeColorToBlue(String col)
```

```
c="Blue";  
return c;
```

This is the variable to be returned

The data type of the two circled objects must be same

Another interesting aspect is that a method *may not* return anything. In this case, the **return type** in the method declaration is written as **void**. This is seen in the **car class** example above in the method that changes car colour to “Blue” (line 11-13).

Returning multiple values

A method can only return one value. However, if the user wants to return multiple values, then two ways can be adopted.

Modify object by calling methods

The first way is by defining a class and calling methods that modify the objects themselves.

Create a custom object

Another way is to create *custom* objects which means that you specify a special object that can store multiple types of data or values, as per requirement and **return** that!

In the next lesson, we will discuss constructors in detail.

Constructor

Let's discuss constructor in this lesson.

We'll cover the following

- Introduction to a constructor
 - Explanation

Introduction to a constructor

Constructor is a special method that is automatically called when an object is created. There are certain syntactical rules when creating a constructor.

- The constructor has the **same name** as that of the class.
- Every class **must have** a constructor. Provided that a constructor is **not** defined, the compiler will create a **default** constructor which will be *empty*!

Let's look at an example of a constructor in the code snippet below.

```
class car {  
  
    private String model;  
    private int horsepower;  
  
    //This constructor takes in parameters and sets  
    // the variables in the class  
  
    public car(String m, int hp) {  
  
        this.model = m;  
        this.horsepower = hp;  
    }  
  
    // This method shows the horsepower of the car  
    public void showDetails() {  
        System.out.println("Car horsepower is: " + this.horsepower);  
    }  
}  
  
class check {  
    public static void main(String[] args) {  
        //Creating the car object with details  
        car newCar = new car("New", 1000);  
        //Showing that the car details have been saved  
    }  
}
```

```
//Showing that the car details have been saved  
newCar.showDetails();  
}  
}
```



Explanation

On *lines 9-13* we have defined a constructor that takes two parameters in its input and sets the values of class variables.

Let's discuss static methods in the upcoming lesson.

Static Methods

In this lesson, we will discuss static methods.

We'll cover the following

- Introduction to static methods
 - main is static method

Introduction to static methods

The keyword `static` is a useful tool in Java. It basically means that a **static** method is one that can be called without any object instance. There are a few key points to remember with **static methods**.

- These methods **do not** belong to any particular object but rather to the whole class.
- This method can be invoked **without creating** an object instance.
- Keep in mind that **non-static** data members cannot be used and neither can **non-static** methods be called directly in *static methods*.

To understand by example, we look at the `System.out.println()` method. This method is a static method of the `System` class. Making this method static has its own benefit- you can use **one** output stream for **all** printing of outputs.

With that, let's look at another example of a static method in the snippet below.

```
class car {  
  
    public String colour;  
    public car() {};  
    public car(String col) {  
        this.colour = col;  
    }  
    public String getColour() {  
        return this.colour;  
    }  
    public static void main(String[] args) {  
        car newCar = new car("Pink");  
        System.out.print("Car colour is: " + newCar.getColour() + "\n");  
    }  
}
```

```
        System.out.print("Car colour is: " + newCar.getColour() + "\n");
        setColourBlue(newCar);
        System.out.print("Car colour is: " + newCar.getColour() + "\n");
    }

    public static void setColourBlue(car c) {
        c.colour = "Blue";
    }
}
```



In the above code `setColorBlue()` is a static method. That's why on *line 14* we called this method without creating an object.

main is static method

See the code given below!

```
class static_methods {
    public static void main(String[] args) {
        int my_variable = 0;
        System.out.println("Printing variable: " + my_variable);
    }
}
```



Note: `main()` is a Static method

In the code above, `main()` is a static method. Why does it **need** to be static? This method is called even when no object is created in the code. It does not need an object to be present for its functioning. Hence, this method has the `static` keyword attached to it.

Well, this is all about **methods**. Next up, some challenges are provided to test the understanding of methods in Java.

Challenge 1: Method to Check Sum

In this challenge, you will implement a method which checks the Sum of two integers.

We'll cover the following



- Problem statement
 - Example

Problem statement

Write a static method `checkSum` that adds two integer arguments and returns 0, 1 or 2.

- Takes two integers `one` and `two` in its input. Arguments are *pass by value* to the method.
- Has a `check` variable whose value gets updated as explained below.
- **Adds** `num1` and `num2`, and *checks* if their **sum** is **less** than **100** in which case
 - Sets the value of the `check` variable to **0**.
- If **sum** is **greater** than **100**
 - Sets the value of the `check` variable to **1**.
- If **sum** is **equal** to **100**
 - Sets the value of the `check` variable to **2**.
- In the end, it will `return` the `check` variable.

Example

Input: 4, 80

The console should display the following:

Output

84 is less than 100

Note: In the above case the method should set `check` to **0**.

Write your code below. It is recommended that you try solving the exercise yourself before viewing the solution.

Good Luck!

```
class challenge_one {  
    public static int checkSum(int one, int two) {  
        //Write your code here  
        //Declare the necessary variable  
  
        //Change the return variable as well  
        return -1;  
    }  
}
```



In the next lesson, we will review the solution to the above challenge.

Solution Review: Method to Check Sum

In this review, solution of the challenge 'Method to Check Sum' from the previous lesson is provided.

We'll cover the following

- Solution: Do you know your maths?
- Understanding the code

Solution: Do you know your maths?

```
class challenge_one{
    public static int checkSum(int one, int two){
        //Write your code here
        //Declare the necessary variable
        int check;
        int sum= one+two;
        if(sum<100)
            check = 0;
        else if(sum>100)
            check=1;
        else
            check=2;
        //Change the return variable as well
        return check;
    }

    public static void main(String[] args){
        int answer=checkSum(100,110);
        System.out.println("The value of check is: "+answer);

        answer=checkSum(100,0);
        System.out.println("The value of check is: "+answer);

        answer=checkSum(100,-110);
        System.out.println("The value of check is: "+answer);
    }
}
```



Understanding the code

Line 5: Declare a variable called **check** which will store the **integer** value **0** or **1** or **2**.

Line 5: Declare a variable called **check** which will store the **Integer** value **0**, **1** or **2**, depending on the condition.

Line 6: Declare an **int** variable called **sum**. In this variable, calculate and store, the sum of the two parameters given, **one** and **two**.

Line 7:

- This marks the start of the conditional block
- The **if** condition evaluates whether the **value** of **sum** is *less than 100*.
- If the condition is met, it proceeds to Line 8, otherwise it skips down to Line 9.

Line 8: As per problem statement, the value of **check** is set to **0**.

Line 9:

- Provided that the first condition is not met on Line 7, this condition is checked.
- This conditional block checks if the value of **sum** is **greater than 100**, then proceed to Line 10, otherwise skip down to Line 11.

Line 10: As per problem statement, the value of **check** is set to **1**.

Line 11: This statement is an **else** block which basically means that provided that both conditions are unfulfilled, then this block of code between the two curly brackets **{}** must be executed.

Line 12: The value of **check** is set to **2** as per question guidelines.

Line 14: The **check** variable is *returned* to the method calling the **checkSum()** method.

Let's solve another challenge in the upcoming lesson.

Challenge 2: Letter Grade to GPA

In this challenge, you will implement a method which converts the Letter grade to GPA.

We'll cover the following ^

- Problem statement
- Coding Exercise

Problem statement

There are many school systems that give standardized GPA points instead of letter grades and vice versa. To bring all these institutions on the same grade scale, we have hired you! Your job is to take a **letter grade** and change it to the required **GPA point**.

Coding Exercise

Write a method called `** letterToGPA**` that takes in a **String** value and returns the correct **decimal GPA point**, with respect to the scale given below:

- A+: 4
- A: 4
- A-: 3.7
- B+: 3.3
- B: 3
- B-: 2.8
- C+: 2.5
- C: 2
- C-: 1.8
- D: 1.5
- F: 0

If any other grade is given, simply return a `-1`.

Hint: Revise conditionals and see what helps keep your code short and brief

Only write the code where instructed in the snippet below.

Test your code against our cases and see if you can pass them.

The solution is given in case you get stuck and the next lesson will include a review of the solution, but it is **highly recommended** that you try it yourself first!

Good Luck!

```
class gpaHelper{  
    public static double letterToGPA(String grade) {  
        //Write your code here  
        //Change the return statement according to what should be returned  
        return -10.0;  
    }  
}
```



If you need some help with understanding this code, turn to the next lesson for the *Solution Review* of this challenge!

Solution Review: Letter Grade to GPA

In this review, solution of the challenge 'Letter Grade to GPA' from the previous lesson is provided.

We'll cover the following



- Solution: Is your GPA correct?
- Understanding your Code

Solution: Is your GPA correct?

```
class gpaHelper{  
  
    public static double letterToGPA (String grade) {  
        double answer;  
  
        switch (grade) {  
            case "A+":  
            case "A":  
                answer = 4;  
                break;  
  
            case "A-":  
                answer = 3.7;  
                break;  
  
            case "B+":  
                answer = 3.3;  
                break;  
  
            case "B":  
                answer = 3;  
                break;  
  
            case "B-":  
                answer = 2.8;  
                break;  
  
            case "C+":  
                answer = 2.5;  
                break;  
  
            case "C":  
                answer = 2.0;  
                break;  
  
            case "C-":  
                answer = 1.8;  
                break;  
        }  
    }  
}
```

```

        case "D":
            answer = 1.5;
            break;

        case "F":
            answer = 0;
            break;

        default:
            answer = -1;
    }

    return answer;
}
public static void main( String args[] ) {
    System.out.println("Grade A: " + letterToGPA("A"));
    System.out.println("Grade D: " + letterToGPA("D"));
    System.out.println("Grade L: " + letterToGPA("L"));
}
}

```



Understanding your Code

Line 3

- The method `letterToGPA` is declared **static** so it can be called without creating an object.
- The method takes in a **single** argument which is of type *String* and return *double* value in the output.

Line 4: A **double** type variable `answer` is declared which will store the final GPA point.

Note: We have chosen to use `switch` statements instead of an if condition for code brevity. We can also use `if-then-else` block.

Lines 6-46:

- This statement is the start of the **switch** statement block.
- The argument in the round brackets, `()`, is the variable which will be equated in the following *cases*. We check the value of `grade` against each case until the `grade` matches the particular case value. Once the value matches, we will set

the value of `answer` accordingly and come out of the loop immediately.

Lines 48-49

- This is a special *case*.
- The **default** case states that provided that **no other** condition is met, this is what should be done.
- In the problem statement above, we are told that if **any other** grade is given, the output should be `-1` so that an incorrect grade can be detected.
- This can be seen in Line 49 where the variable **answer** is equated to `-1`.

Line 53

- The `answer` variable is returned to the main method calling it.
-

In the next lesson, we will solve one more challenge related to methods.

Challenge 3: Sum of Digits in an Integer

In this challenge, you will implement a method which will return the sum of digits in an integer.

We'll cover the following ^

- Problem statement
 - Method Prototype
 - Sample input
 - Sample output
 - Pictorial Representation

Problem statement

Implement a *method* in Java called *sumOfDig* which will **return** the sum of the *digits* in an `int` variable passed to it.

Method Prototype

```
int sumOfDig(int var)
```

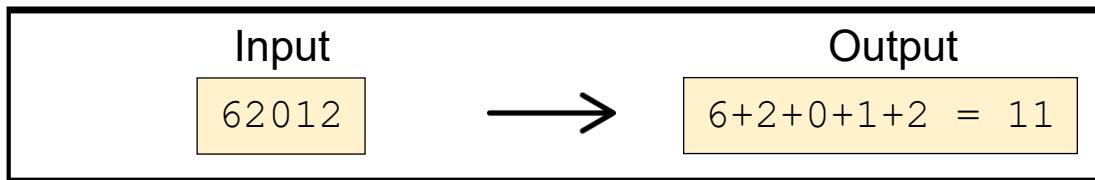
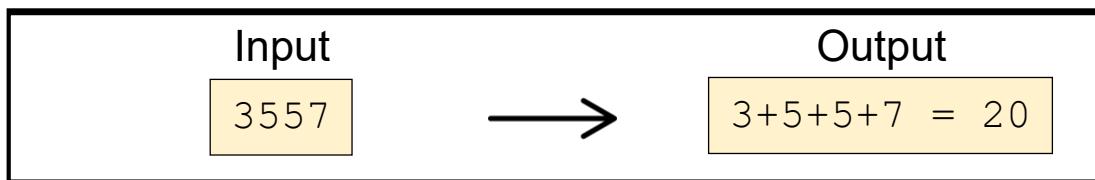
Sample input

```
var = 3557
```

Sample output

```
sum = 3 + 5 + 5 + 7 = 20
```

Pictorial Representation

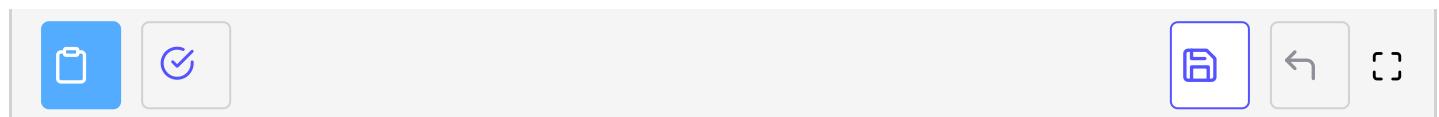


Sum of digits

Try to implement the solution by *yourself* once you have understood the *problem statement* clearly. Referring to the solution part should be your last resort. **Good luck!**

```
class SumDigits {
    public static int sumOfDig(int
        var) {

        // your code goes here
        return var; // return the resultant sum
    }
}
```



Sum of Digits

Let's discuss the solution of the above challenge in the next lesson.

Solution Review: Sum of Digits in an Integer

In this review, solution of the challenge 'Sum of Digits in an Integer' from the previous lesson is provided.

We'll cover the following ^

- Solution
- Understanding the code

Solution

```
class SumDigits {  
    public static int sumOfDig(int var) {  
        int result = 0; //variable for resultant sum  
        int lastDigit = 0;  
        while (var > 0) {  
            //seclude & keep adding the last digit into result  
            lastDigit = var % 10;  
            result = result + lastDigit;  
            System.out.println("Last Digit: " + lastDigit);  
            System.out.println("Sum: " + result);  
            var /= 10; //update the new value of var  
            System.out.println("Number: " + var);  
        }  
        return result;  
    }  
    public static void main( String args[] ) {  
        int number = 1745;  
        System.out.println("Number: " + number);  
        System.out.println( "Sum of digits in 1024 is: "+ sumOfDig(number) );  
    }  
}
```



Understanding the code

- **Line 3:** We start by declaring an `int result` variable to store the *sum*.
- **Line 7-8:** To add the digits one by one to the result we will take the remainder from division of `var` by 10 to get the last digit and add it to the `result`.

- **Line 11:** After separating and adding the last digit of the `var` to the `result`, we update the `var` by dividing it by 10. In this way, the value in `var` gets lesser and lesser after each iteration and we use the condition `var > 0` in the `while` loop.
 - **Line 14:** At the end we have the resultant sum of digits in the `result` variable and *return* it from the method.
-

In the next solve we will solve a challenge related to Strings.

Challenge 4: Playing With Strings

In this challenge, you will implement a method which will return the String in upper or lower case.

We'll cover the following



- Problem statement
- Coding exercise

Problem statement

In this exercise, you have to write a method `test()`, that **first** checks the length of the String given to the method as input.

If the length of the String is an **even** number then return the entire String in **upper case** letters. On the other hand, if the length is **odd**, then return the entire String in **lower case**.

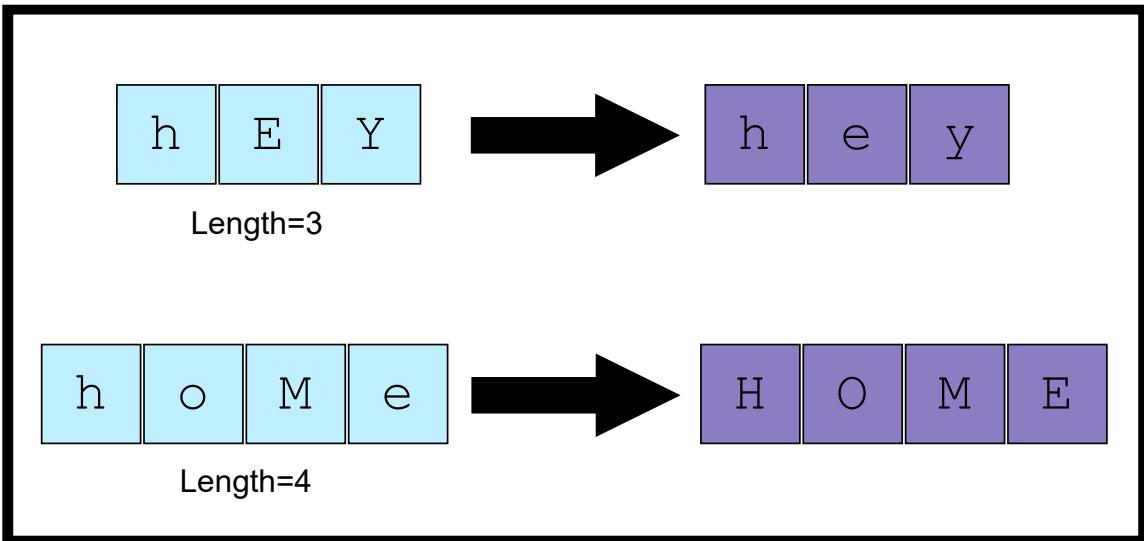
Coding exercise

Only write the code where instructed in the snippet below.

Test your code against our cases and see if you can pass them.

The solution is given in case you get stuck and the next lesson will include a review of the solution, but it is **highly recommended** that you try it yourself first!

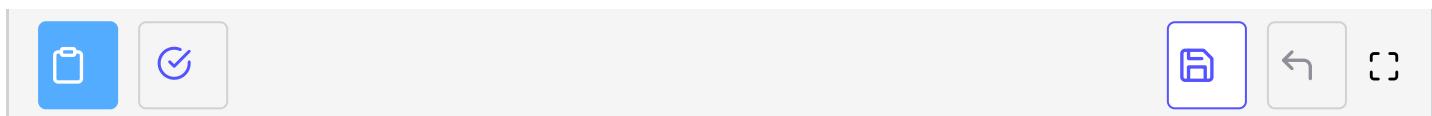
Good Luck!



The purple Strings show what the method should return on the basis of the length

```
class challenge_four {
    public static String test(String x) {
        //Write your code here
        //Declare any variables that are needed

        //Change the return statement accordingly
        return "";
    }
}
```



In the next lesson, we will review the solution to the above challenge.

Solution Review: Playing With Strings

In this review, solution of the challenge 'Playing With Strings' from the previous lesson is provided.

We'll cover the following ^

- Solution: To be upper case or not to be?
- Understanding the code

Solution: To be upper case or not to be?

```
class challenge_four {  
    public static String test(String x) {  
  
        if (x.length() % 2 == 0) {  
            return x.toUpperCase();  
        }  
  
        return x.toLowerCase();  
    }  
    public static void main( String args[] ) {  
        String odd = "Hello";  
        String even = "John";  
        System.out.println( "Hello:" + test(odd));  
        System.out.println( "John:" + test(even));  
    }  
}
```



Understanding the code

- **Line 4:** The **if condition** first finds the length of the input String. This is done by using the method, `x.length()`. Then it checks whether the length, when **divided by 2** gives a zero remainder or not. Given that the **remainder** is `0`, the input variable, `x` is **even** so *line 5* will be executed. If not, then jump to *line 8*.
- **Line 5:** `returns` the `upper_case` String `x`

• Line 5: Returns the upper case string `x`.

- **Line 8:** Provided this if condition is not met, on **line 4** the lower case version of String `x` is returned.
-

Let's go through a *quick quiz* to test your understanding.

Quick Quiz!

To test your understanding of methods, let's take a short quiz.

1

Which visibility type fulfills this requirement: “It can only be called by the class it is defined in”?

2

The returnVariable and the returnType must have what feature in common?

3

If a method is made to be static, which of the following is true?

4

Method A creates an object and passes it to method B as argument, which modifies it. The modification changes the object defined in method B.

Retake Quiz

That marks the end of this chapter. In the next chapter, we will discuss arrays.

What Are Arrays?

In this lesson, an introduction to Arrays data structure is provided.

We'll cover the following



- Definition
- Declaration
 - Default values
- Initializing & accessing array elements
 - Using loops

Definition

An *array* is a collection of *similar* data types in a sequenced format. The position of each data member in this sequence is represented with a number known as its *index*.

Elements of an Array

The members of this data collection are referred to as *array elements*.

Length of an Array

The *total number* of elements in an array is called the length of an array. This length is set once at the time of creation of an array and *can't be changed later* in a program. We can check the length of an array using an inbuilt functionality of Java as `arrayName.length`.

Indexing

The indexes in an array always range from **zero** to **length-1**. For example, an array containing the first 100 natural numbers will have a *length of 100* and *indexes from 0 to 99*.

Declaration

In Java, at the time of declaration an array we must specify the following:

- Datatype
- Name
- Size

Syntactically we can declare an array of an integer data type in the following ways:

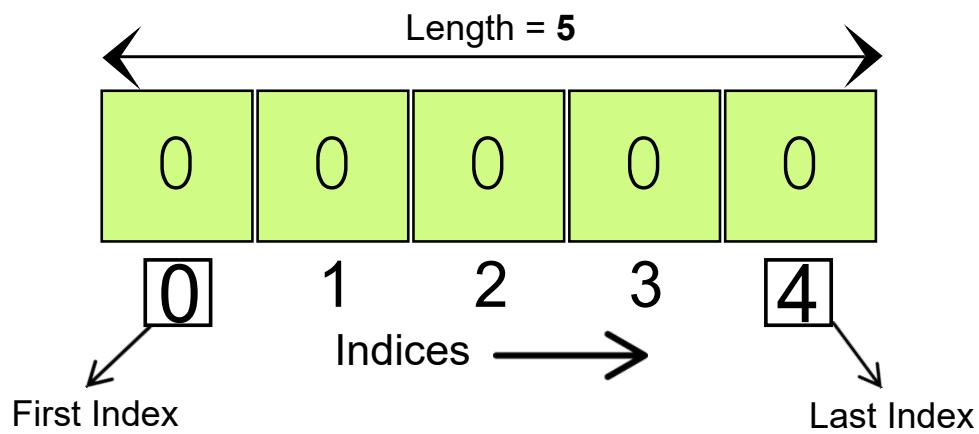
```
int[] myArray;  
int []myArray;  
int myArray[];
```

In Java an array needs to be instantiated i.e. we need to specify the size of the array which is to be allocated to it in the memory. Instantiation is done using the keyword `new` as follows:

```
int[] myArray = new int[5] // Instantiation of an Array of size 5;
```

Array instantiation using keyword new

Lets see what the above line of code actually does:



Array Illustration

It creates an array to store 5 int values and sets them all to 0.

Note: The starting index of an array is always **0** not **1**

Default values

If the user has not assigned any values to the array elements Java puts some *default* values in there as follows:

Array data Type	Default Value
All numeric data types (<code>int</code> , <code>short</code> , <code>double</code> etc.)	<code>0</code>
<code>boolean</code>	<code>false</code>
<code>char</code>	<i>null character</i>

Initializing & accessing array elements

Initialization means assigning specific values to array elements and defining the array in the same line of code.

One can *declare*, *instantiate* and *initialize* an array in a single line using curly braces containing the values we want to store in that array. The number of values in the curly braces decide the length of that array. The syntax is as follows:

```
int[] myArray1 = {1,2,3,4,5} //This declares,instantiates and initializes
                           // an array with 5 elements 1,2,3,4 and 5
int[] myArray2 = {2,4,6}    //This declares,instantiates and initializes
                           // an array with 3 elements 2,4 and 6
```

The above approach is normally used when the values to be stored are already known.

The array elements can be declared and assigned individually using the *index values*. To learn how to access the elements individually one can refer to the below example:

```
class Arrays {
    public static void main(String args[]) {
        int[] myArray = new int[5]; //Declaration and Instantiation of Array with length 5
```

```

System.out.println("Printing the size of myArray: " + myArray.length);

myArray[0] = 2; //accessing 1st element and assigning value 2 to it
myArray[1] = 4; //accessing 1st element and assigning value 4 to it
myArray[2] = 6; //accessing 1st element and assigning value 6 to it
myArray[3] = 8; //accessing 1st element and assigning value 8 to it
myArray[4] = 10; //accessing 1st element and assigning value 10 to it

/*Printing the stored values*/

System.out.println("First element of myArray is: " + myArray[0]);
System.out.println("Second element of myArray is: " + myArray[1]);
System.out.println("Third element of myArray is: " + myArray[2]);
System.out.println("Fourth element of myArray is: " + myArray[3]);
System.out.println("Fifth element of myArray is: " + myArray[4]);
}
}

```



Initializing an Array

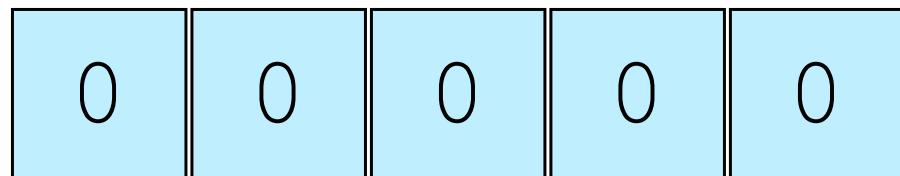
The square bracket can be used to:

- Indicate that a variable is an array when declaring it.
- Read from or writing to a specific array element.

Note: A positive integer index inside square brackets is used to access a certain element as `arrayName[index]` .

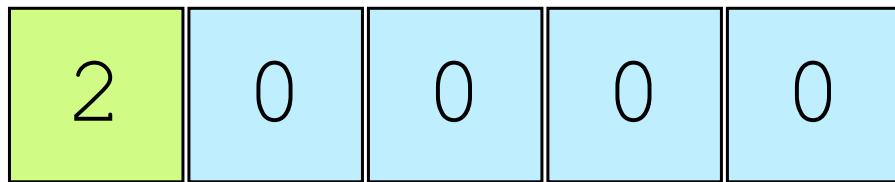
A graphical representation of the above is:

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



myArray[0] myArray[1] myArray[2] myArray[3] myArray[4]

myArray[0] = 2;

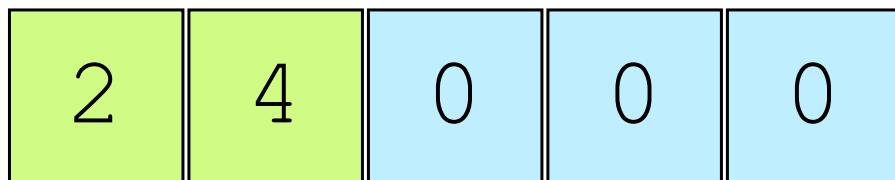


myArray[0]

Index = 0

2 of 6

myArray[1] = 4;

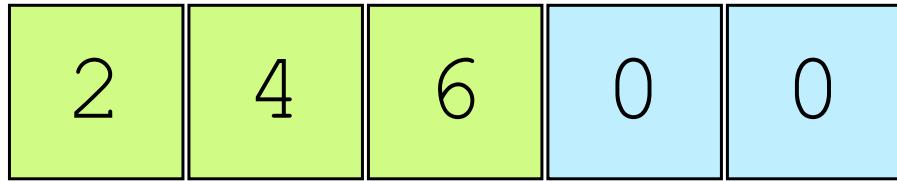


myArray[1]

Index = 1

3 of 6

myArray[2] = 6;

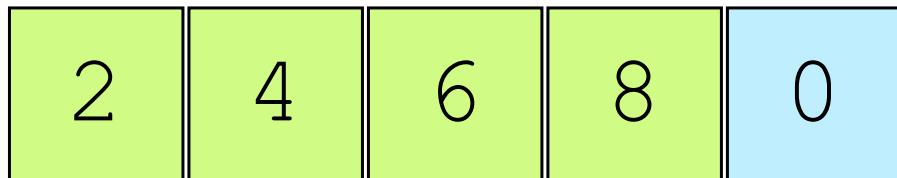


myArray[2]

Index = 2

4 of 6

myArray[3] = 8;

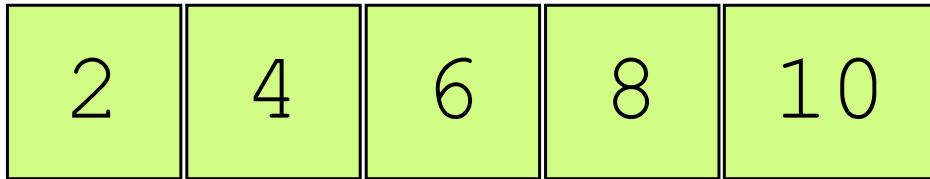


myArray[3]

Index = 3

5 of 6

```
myArray[4] = 10;
```



myArray[4]

Index = 4

6 of 6



Quick question: What do you think about setting a negative number as the length of an array?

Hide Hint

An array cannot have a negative length!

Using the above methods, one can notice that assigning values to an array with a large number of elements will require a lot of effort and code duplication. It will be significantly inefficient to do so. Hence, to populate an array with values in an efficient way, we can use the following method.

Using loops

Let's learn how can we use *loops* to initialize arrays:

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



```
int[] myArray = new int[10]; // Declaration and Instantiation of Array with length 10

for (int i = 0; i < myArray.length; i++) // Iterate through indexes 0-9
{
    myArray[i] = i + 1; // Initialize values to 1-10
}

for (int i = 0; i < myArray.length; i++) {
    System.out.println("The value at myArray[" + i + "] is: " + myArray[i]);
    // Printing all values to console
}
}
```



Initialization using loops

In the next lesson, we will discuss arrays in more detail.

A Bit More About Arrays

In this lesson, an explanation of Character Arrays, String Arrays, Arrays & Methods is provided.

We'll cover the following

- Character Arrays
- String Arrays
 - Index out of bound exception
- Arrays & Methods



As we have covered the basics of Arrays in the previous lesson, let's discuss how to declare and use arrays of `char` & `String` data types.

Character Arrays

We can declare an Array of `char` in which we can store characters. Let's write a code to declare and initialize a character array:

```
class CharArr {  
    public static void main(String args[]) {  
        char[] chArray1 = {  
            'a',  
            'b',  
            'c'  
        }; //initialization of first char array  
        char[] chArray2 = new char[5]; //instantiation of second char array  
        int index = 0;  
        for (char i = 'v'; i <= 'z'; i++) { //Assiging char values to second array using loop  
  
            chArray2[index] = i;  
            index++;  
        }  
        //printing out the stored values in the arrays  
        System.out.print("The Values stored in ChArray1 are: ");  
  
        for (int i = 0; i < chArray1.length; i++) {  
  
            System.out.print(chArray1[i] + " ");  
        }  
  
        System.out.println();  
        System.out.print("The Values stored in ChArray2 are: ");  
  
        for (int i = 0; i < chArray2.length; i++) {  
  
            System.out.print(chArray2[i] + " ");  
        }  
    }  
}
```

```
        System.out.print(chArray2[i] + " ");
    }
}
```



Character Arrays

The above example has a loop that uses `char` variable. Each `char` value is represented by a number, so `i++` actually increments the internal code, which makes us hop from `v` to `w` and so on.

Note that while storing a value in `char` array the literal should be inside pair of single quote like `chArray = 'a';`

String Arrays

We can also store *Strings* in an Array, let's get familiar to `String` arrays through the below code:

```
class StrArray {
    public static void main(String args[]) {
        String[] strArray = {
            "A",
            "Quick",
            "Brown",
            "Fox",
            "Jumps",
            "Over",
            "a",
            "Lazy",
            "Dog"
        }; //initialization of first char array

        System.out.print("The Values stored in strArray are: ");

        for (int i = 0; i < strArray.length; i++) {
            System.out.print(strArray[i] + " ");
        }
    }
}
```



String Array

Note that while storing a value in `String` array the literal should be inside pair of double quote like `strArray = "a";`

Index out of bound exception

If we try to index beyond `array.length-1`, the runtime throws an exception and ends the program.

Warning: The program given below will generate an exception.

```
class Arrays {  
    public static void main(String args[]) {  
  
        int[] myArray = new int[10]; // Declaration and Instantiation of Array with length 10  
  
        for (int i = 0; i < myArray.length; i++) // Iterate through indexes 0-9  
        {  
            myArray[i] = i + 1; // Initialize values to 1-10  
        }  
  
        System.out.println("The value at myArray[11] is:" + myArray[11]);  
    }  
}
```



Initialization using loops

Arrays & Methods

An array can be declared in a calling function and passed to a function as an argument. Any changes made to the argument array in the called function are actually performed directly on the array defined in the calling function. Let's see this in action.

```
class ArrayToMethod {  
    public static void main(String args[]) {  
        int[] arr = {  
            1,  
            2,  
            3,  
            4  
        }  
    }  
}
```

```

        4,
        5
    }; //initialize
    System.out.println("The Values before calling the method are:");
    for (int i = 0; i < arr.length; i++) {
        System.out.print(arr[i] + " "); //printing the array before calling method

    }
    int[] returnedArr = multiply(arr, 3); //storing the returned array
    System.out.println();
    System.out.println("The Values from the returned Array are:");
    for (int i = 0; i < arr.length; i++) {
        System.out.print(returnedArr[i] + " "); //printing the returned array

    }
}
} //end of main()

static int[] multiply(int[] arr, int num) {
    for (int i = 0; i < arr.length; i++) {
        arr[i] = arr[i] * num;
    }
    return arr; //returning Array
}
}

```



Arrays & Methods

In the above program, the *called* method is `multiply(int[] arr, int num)` and the *calling* method is `main()`.

In Java whenever **primitive** datatypes are passed as an argument to a method a copy of the variable is made and passed to that method which means if *called method* makes any changes to the passed values it is not visible to the *calling method*. As an array is a data structure rather than a primitive data type, it is passed by reference to a method which means that if the *called method* alters any value of an array this alteration will be visible to the *calling method* also.

Let's experience the above said through making some changes to the coding example given before:

```

class ArrayToMethod {
    public static void main(String args[]) {
        int[] arr = {
            1,
            2,
            3,
            4,
            5
        }; //initialize

        System.out.println("The Values before calling the method are:");

```



```
for (int i = 0; i < arr.length; i++) {
    System.out.print("arr[" + i + "] = " + arr[i] + " ");
}
System.out.println();
multiply(arr, 3); //nothing is being returned
System.out.println("The Values after calling the method are:");
for (int i = 0; i < arr.length; i++) {
    System.out.print("arr[" + i + "] = " + arr[i] + " ");
}
}

} //end of main()

static void multiply(int[] arr, int num) { //will change the values of passed array
    for (int i = 0; i < arr.length; i++) {
        arr[i] = arr[i] * num;
    }
    //Not returning anything
}
}
```



Arrays by Reference

We can notice in the above code that the *called method* has changed the values of the array passed to it and this change is visible to the *calling method*.

In the next lesson, we will discuss another type of Arrays known as two-dimensional Arrays.

Two Dimensional Arrays

In this lesson, an introduction of another type of Arrays known as two-dimensional Array is provided.

We'll cover the following



- What are two-dimensional arrays?
 - Declaration
 - Instantiation
 - Graphical Representation
 - Example program

In the previous lesson, we got familiar with the basic concepts of Arrays in Java. All we have been learning till now was about **linear arrays**.

Another type of arrays is a **two dimensional array**, let's discuss briefly about it.

What are two-dimensional arrays?

Unlike linear arrays, two-dimensional arrays are just like an $m \times n$ matrix with **m** number of *rows* and **n** number of *columns*.

Declaration

The declaration of *2-d arrays* is as follows:

```
Datatype[m][n] name;
```

The elements in a two-dimensional array are arranged in the form of rows and columns.

Instantiation

The instantiation of a character array with 2 rows and 2 columns will be as:

```
int[][] twoDimArray = new int[2][2];
```



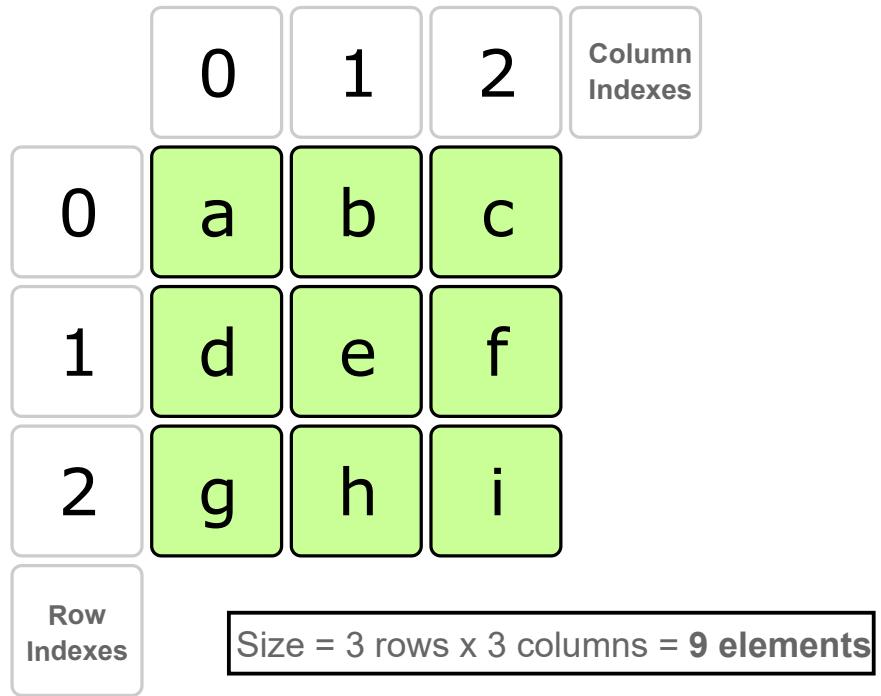
What should be the index range of two-dimensional arrays?

 Hide Hint

The index range of a two-dimensional array of size $m \times n$ will be: row indexes: 0 to $m-1$ column indexes: 0 to $n-1$

Graphical Representation

Graphically we can portray the 2D array as:



Two-dimensional Arrays

Each element in a 2-d array is indexed by means of row and column index using square bracket notation. To access the **b** in the above array we will use the following notation:

arrayName[0][1]

Example program

Let's initialize this two-dimensional character array with some characters and get them printed to the console:

```
class HelloWorld {  
    public static void main(String args[]) {  
  
        char[][] twoDimArray = new char[3][3]; //instantiating a character array of size 3*3 = 9 e  
  
        twoDimArray[0][0] = 'a'; //stores a at row:0, column:0 position  
        twoDimArray[0][1] = 'b'; //stores b at row:0, column:1 position  
        twoDimArray[0][2] = 'c'; //stores c at row:0, column:2 position  
        twoDimArray[1][0] = 'd'; //stores d at row:1, column:0 position  
        twoDimArray[1][1] = 'e'; //stores e at row:1, column:1 position  
        twoDimArray[1][2] = 'f'; //stores f at row:1, column:2 position  
        twoDimArray[2][0] = 'g'; //stores g at row:2, column:0 position  
        twoDimArray[2][1] = 'h'; //stores h at row:2, column:1 position  
        twoDimArray[2][2] = 'i'; //stores i at row:2, column:2 position  
  
        //Printing out the stored values  
  
        System.out.print(twoDimArray[0][0] + " " + twoDimArray[0][1] + " " + twoDimArray[0][2] +  
        System.out.print(twoDimArray[1][0] + " " + twoDimArray[1][1] + " " + twoDimArray[1][2] +  
        System.out.print(twoDimArray[2][0] + " " + twoDimArray[2][1] + " " + twoDimArray[2][2]);  
    }  
}
```



Note: The dimensions of an **array** is not limited to just **2**. An **array** can be of **n** number of dimensions depending on the problem.

This was pretty much about the Arrays, now let's test our understanding through solving some coding challenges.

Challenge 1: Find the Maximum Value

In this challenge, you will figure out how to find out the maximum value stored in an Integer Array.

We'll cover the following ^

- Problem statement
- Input
- Output
- Sample input & output
- Pictorial representation

Problem statement

Write a method named *findMaxVal()* to find the maximum integer value stored in an Array of any size.

Input

- The input to the method will be an array passed as a parameter.

Output

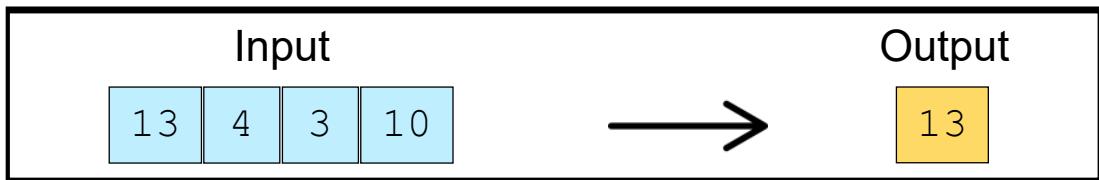
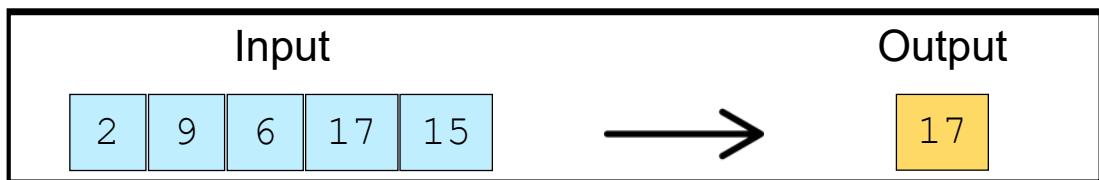
- As an output the method will return the maximum value found in the array.

Sample input & output

Input: {4, 10, 12, 17, 11}

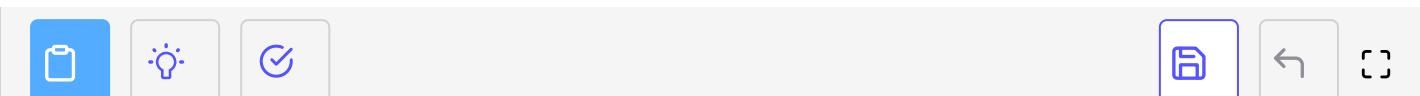
Output: 17

Pictorial representation



Think over the problem carefully and write your code below. It's recommended that you try solving the problem by yourself before referring to the solution part.
Good luck!

```
class CheckMax {
    public static int findMaxVal(int[] arr) {
        // Write your code here
        return -1;
    }
}
```



Hint 1 of 1

Traverse over the array element by element and update the variable containing the maximum value if next value is greater than the previous.

Find the Maximum Value

Let's discuss the solution of the above challenge in the next lesson.

Solution Review: Find the Maximum Value

In this review, solution of the challenge 'Find the Maximum Value' from the previous lesson is provided.

We'll cover the following

- Solution
- How does the above code work?

Solution

```
class CheckMax {  
    //Returns maximum value from Array passed as parameter  
    public static int findMaxVal(int[] arr) {  
  
        int max = arr[0];  
  
        for (int i = 1; i < arr.length; i++) { //iterate over all the array elements  
  
            if (arr[i] > max) { //check if current element is greater than the already  
                //stored max value  
                max = arr[i]; // if yes then update the max value to current element  
            }  
        }  
  
        return max; //return the maximum value  
  
    } //end of findMaxValue()  
//end of CheckMax  
    public static void main( String args[] ) {  
        int array [] = {78, 89, 32, 90, 21};  
        System.out.println( "The maximum value in an array is: "+findMaxVal(array));  
    }  
}
```



How does the above code work?

- **Line 5:** We start by declaring an **max** variable and storing the first element of the array in it.
- **Line 7:** We are using the *for loop* to traverse through the array using the control variable **i** as the index.

control variable `i` as the *index*.

- **Line 9-11:** The `if` condition is then comparing the current element with the value stored in the `max` variable. The `max` variable will be updated only if the previously stored value is lesser than the currently compared array element. Otherwise, `i` is incremented for the next repetition.
- **Line 15:** In the last statement the `max` variable is returned to the calling point.

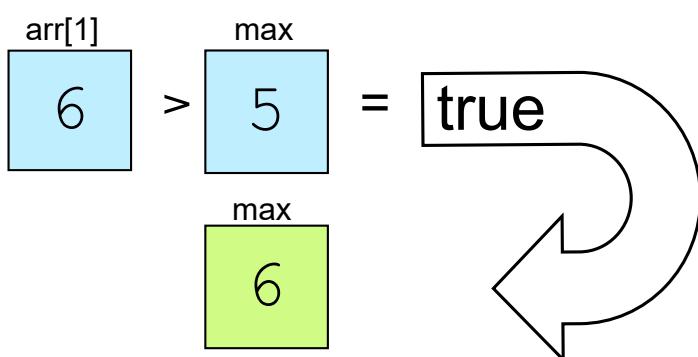
int max = arr[0] = 

max = 5

Assigning Value to max

1 of 5

i = 1



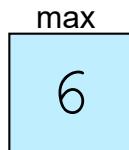
First Iteration

2 of 5

i = 2



$$\begin{array}{c} \text{arr}[2] \\ 3 \end{array} > \begin{array}{c} \text{max} \\ 6 \end{array} = \text{false}$$



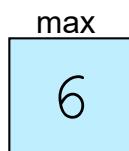
Second Iteration

3 of 5

i = 3



$$\begin{array}{c} \text{arr}[3] \\ 1 \end{array} > \begin{array}{c} \text{max} \\ 6 \end{array} = \text{false}$$



Third Iteration

4 of 5

i = 4



arr[4]

10

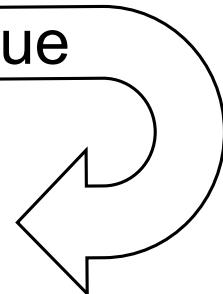
max

6

= true

max

10



Last Iteration

5 of 5



In the next challenge, we will sort the values stored in an array.

Challenge 2: Sorting an Array

In this challenge, you will figure out how to sort an Integer Array.

We'll cover the following ^

- Problem statement
 - Function prototype
 - Sample input
 - Sample output
 - Pictorial representation

Problem statement

Implement a *method* in Java called `sortAsc()` which will **sort** an *array* of any *size* having *integer* values in an **ascending** order and return the sorted array in the output.

Note: The array does not need to be returned, we are doing it for ease of testing

Function prototype

```
int[] sortAsc(int[] arr)
```

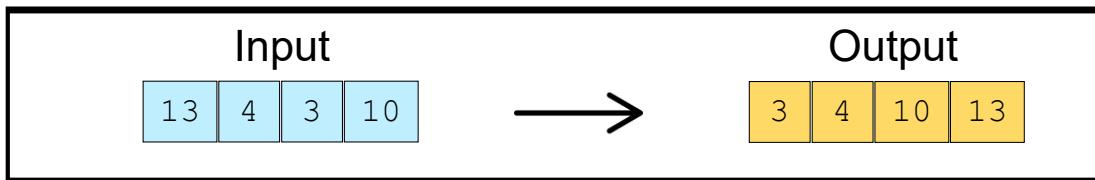
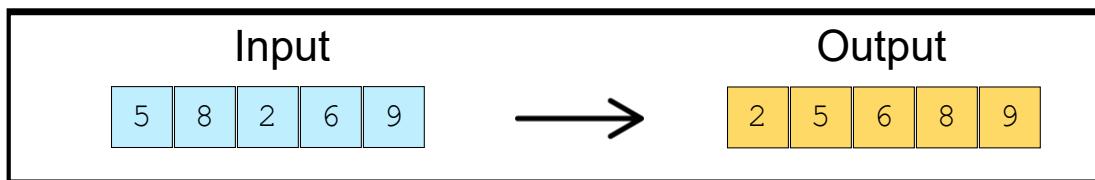
Sample input

```
arr = {5,8,2,6,9}
```

Sample output

```
arr = {2,5,6,8,9}
```

Pictorial representation

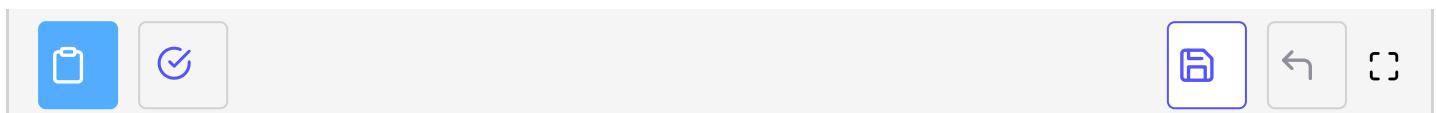


Sorting an Array

Try to implement the solution by *yourself* once you have understood the *problem statement* clearly. Referring to the solution part should be your last resort. **Good luck!**

```
class SortArr {
    public static int[] sortAsc(int[] arr) {

        // your code goes here
        return arr; // change this and return the correct result array
    }
}
```



Sorting an Array in Ascending Order

Let's discuss the solution of the above challenge in the next lesson.

Solution Review: Sorting an Array

In this review, solution of the challenge 'Sorting an Array' from the previous lesson is provided.

We'll cover the following

- Solution
- How does the above code work?

Solution

```
class SortArr {  
    public static void sortAsc(int[] arr) {  
        int temp = 0; //a variable to store temporary value while swapping  
  
        for (int i = 0; i < arr.length-1; i++) //for loop to hold the current element to be compared  
        {  
            for (int j = i + 1; j < arr.length; j++) //for loop to iterate over the other elements  
            { //to get them compared with the current element  
                if (arr[i] > arr[j]) //if any of the higher index element is smaller than  
                { //the current element  
                    temp = arr[i]; //store the current element to temp  
                    arr[i] = arr[j]; //store the smaller element to the lower index position  
                    arr[j] = temp; //store the current element to greater index position  
                }  
            }  
        }  
    }  
    public static void main( String args[] ) {  
        int array[] = {56, 9, 45, 108, 567, 21};  
        System.out.println( "Array values before sorting:" );  
        for (int i =0 ; i < array.length; i++){  
            System.out.print(array[i]+ " ");  
        }  
        System.out.println();  
        sortAsc(array);  
        System.out.println( "Array values after sorting:" );  
        for (int i =0 ; i < array.length; i++){  
            System.out.print(array[i]+ " ");  
        }  
    }  
}
```



How does the above code work?

In the above code, we have implemented nested `for` loops and an `int` variable named `temp` to store a value which is going to be swapped.

- **Line 5 - 8:**

The *control variable* `i`, from the outer `for` loop is used as an index to access the array elements one by one. The currently accessed element `arr[i]` (using the outer for loop) is then compared with all the elements from index `j=i+1` onwards using the inner loop's control variable `j` as an index to access the next elements of the array.

Notice that for each value of `i` the inner loop iterates from `i+1` to the end of the array.

- **Line 9:** At each iteration of the inner loop, a condition is being checked i.e.

```
if(arr[i] > arr[j])
```

- **Line 11 - 13:** The above condition checks:

- if any of the *higher index* element has **lesser** integer value than the *lower index* element.
- If it is so, the element with **greater** value is stored into `temp` and then the element with **greater** value at the *lower index* `arr[i]` will be replaced with the element with **lesser** value at the *higher index* `arr[j]`.
- After this, the **greater** value in the `temp` stored at the *higher index*. This *swapping* is repeated till all the elements are sorted in ascending order i.e. the element with the least `int` value at the start of the array and the element with the maximum value at the end of the array.

In the next lesson, we will try to print the values stored in a matrix.

Challenge 3: Print a Matrix

In this challenge, you will figure out how to print a Matrix using a two-dimensional Array.

We'll cover the following



- Problem statement
 - Function prototype
 - Sample input
 - Sample output pictorial representation

Problem statement

As we have already discussed that a *two-dimensional array* can also be thought of a *matrix*.

Implement a Java method to **initialize & print** a two-dimensional array of size **n x n** in the form of a matrix. You should initialize and Print an array of size 3 x 3 as:

- The *diagonal* of the *array* should be filled with **0**.
- The *lower side* should be filled will **-1s**.
- *upper side* should be filled with **1s**.

Note: In order to move a value to the next line you can use **\n**.

Function prototype

```
void printMat(int n)
```

Sample input

```
int n = 3;
```

Sample output pictorial representation

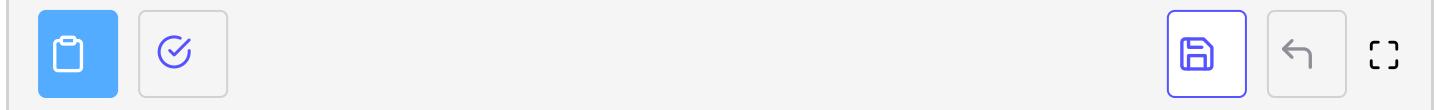
0	1	1
-1	0	1
-1	-1	0

Sample Output Array

Try to implement the solution by *yourself* once you have understood the *problem statement* clearly. Referring to the solution part should be your last resort.

Good luck!

```
class TwoDimArr {
    public static void printMat(int n) {
        //write the codefor making and printing the matrix here
        //use can use \n to move numers to next line in the matrix
        //use " " to add space between numbers in matrix
        System.out.println(); //comment out this line when you start writing code
    }
}
```



Printing the Matrix

Let's discuss the solution of the above challenge in the next lesson.

Solution Review: Print a Matrix

In this review, solution of the challenge 'Print a Matrix' from the previous lesson is provided.

We'll cover the following



- Solution
- How does the above code work?

Solution

```
class TwoDimArr {  
    public static void main( String args[] ) {  
        int n = 3;  
        int[][] arr = new int[n][n];  
        for (int i = 0; i < arr.length; i++) { //assign values to the arr  
            for (int j = 0; j < arr.length; j++) {  
                if (i == j) { //if row=column=> fill the matrix with 0  
                    arr[i][j] = 0;  
                } else if (i > j) { //if row>columns=> fill matrix with -1  
                    arr[i][j] = -1;  
                } else { //if row<columns=> fill matrix with 1  
                    arr[i][j] = 1;  
                }  
            }  
        }  
        for (int i = 0; i < arr.length; i++) { //print the array  
            for (int j = 0; j < arr.length; j++) {  
                System.out.print(arr[i][j] + " ");  
            }  
            System.out.println();  
        }  
    }  
}
```



How does the above code work?

In the above solution, we have to initialize an array using nested loops having:

- All the diagonal elements as `0` which means: when the *row index: i* is **equal** to the *column index: j* we store zeros in the array.
- All the elements on the lower side of the diagonal as `-1` which means: when

All the elements on the lower side of the diagonal as **-1** which means: when the *row index: i* is **greater** than the *column index: j* we store **-1** in the array.

- All the elements on the upper side of the diagonal as **1** which means: when the *row index: i* is **lesser** than the *column index: j* we store **1** in the array.
- After initialization, we have implemented nested loops once again to print the array.

For clearer understanding refer to the below picture:

0	1	1
-1	0	1
-1	-1	0

Legend:
Light Blue Square: Columns > Rows
Pink Square: Rows > Columns
Green Square: Columns = Rows

Let's solve another challenge related to two-dimensional arrays in the upcoming lesson.

Challenge 4: Pascal's Triangle

In this challenge, you have to implement Pascal's triangle using a two-dimensional Array.

We'll cover the following ^

- Problem statement
 - Function prototype
 - Sample input
 - Sample output
 - Pictorial representation

Problem statement

Implement a Java method that takes an *integer* size as input and **displays** a table that represents a [Pascal's triangle](#) using a *two-dimensional* array.

Function prototype

```
void printPascalTri(int size)
```

Sample input

```
int size = 5;
```

Sample output

Print Pascal's triangle of the size 5.

Pascal's triangle is filled from the top towards the bottom. In *Pascal's triangle*:

- **first** and the **second** rows are set to **1**.
- Each *element* of the *triangle* (from the **third** row downward) is the **sum** of the element directly above it and the *element* to the **left** of the *element* directly **above** it.

Note: In order to move a value to the next line you can use `\n`.

Pictorial representation

See the example *Pascal triangle(size=5)* below:

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

Try to implement the solution by *yourself* once you have understood the *problem statement* clearly. Referring to the solution part should be your last resort.

Good luck!

```
class PrintTri {
    public static void printPascalTri(int size) { //define your function

        //write your code here for making and displaying pascals triangle
        //use can use \n to move numbers to next line in the triangle
        //use " " to add space between numbers in triangle
        System.out.println(); //comment out this line when you write your code
    }
}
```



Printing the Pascal's triangle

Let's discuss the solution of the above challenge in the next lesson.

Solution Review: Pascal's Triangle

In this review, solution of the challenge 'Pascal's Triangle' from the previous lesson is provided.

We'll cover the following



- Solution

Solution

```
class HelloWorld {  
    public static void main( String args[] ) {  
        int size = 5;  
        int[][] pascalTr = new int[size][size];  
        int row, col;  
        //assign zero to every array element  
        for (row = 0; row < size; row++)  
            for (col = 0; col < size; col++)  
                pascalTr[row][col] = 0;  
        //first and second rows are set to 1s  
        pascalTr[0][0] = 1;  
        pascalTr[1][0] = 1;  
        pascalTr[1][1] = 1;  
  
        for (row = 2; row < size; row++) {  
            pascalTr[row][0] = 1;  
            for (col = 1; col <= row; col++) {  
                pascalTr[row][col] = pascalTr[row - 1][col - 1] + pascalTr[row - 1][col];  
            }  
        }  
  
        //display the Pascal Triangle  
        for (row = 0; row < size; row++) {  
            for (col = 0; col <= row; col++) {  
                System.out.print(pascalTr[row][col] + " ");  
            }  
            System.out.print("\n");  
        }  
    }  
}
```



Let's assume that we have to print a triangle of **size = 5** as follows:

```
1 1  
1 2 1
```

```
1 3 3 1  
1 4 6 4 1
```

In any case, our starting point will be to initialize the first two rows to `1` like:

```
1  
1 1
```

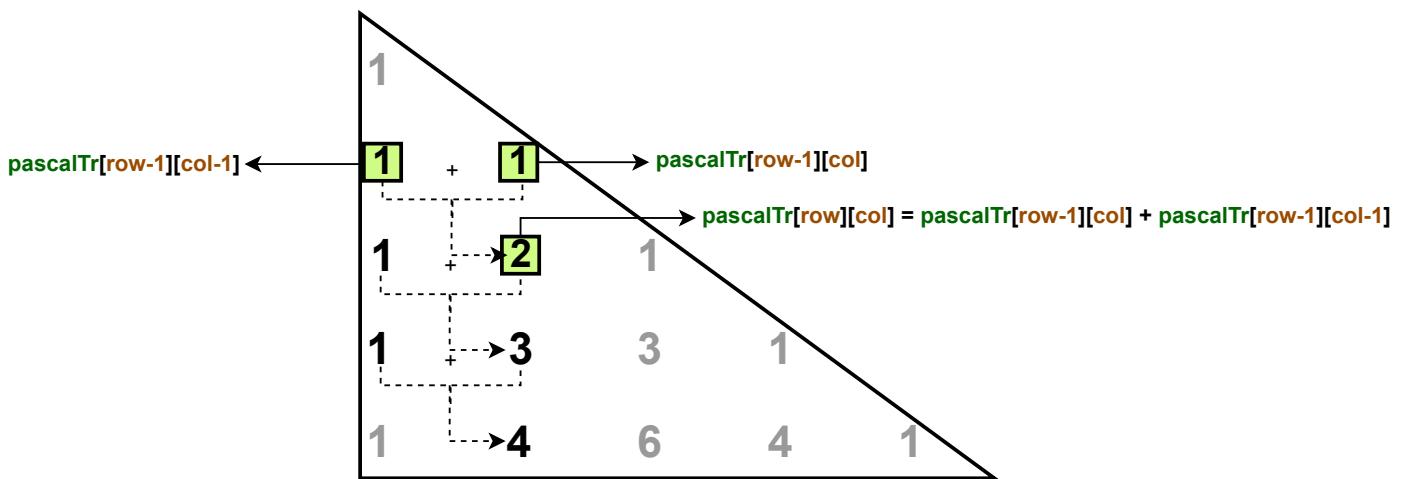
After this, we are fixing the first element of each row to `1` inside the loop by writing:

```
pascalTr[row][0]=1;
```

The logic behind the next line of code

```
pascalTr[row][col]=pascalTr[row-1][col-1]+pascalTr[row-1][col];
```

is depicted below:



Understanding the Logic

At the end of the solution, we have implemented nested for loops to print this triangle to console.

Well, let's test our understanding of array through a quick quiz.

Quick Quiz!

Let's test your understanding by solving a quiz in this lesson.

1

The legal indexes of the character array `arr={'a','r','b','e'}` are:

2

What is the output of the following code fragment?

```
boolean[] arr =new boolean[3];
arr[0]= (arr.length==2);
arr[1]= (arr.length==3);
System.out.print(arr[0]+" "+arr[1]+" "+arr[2]);
```

3

Arrays can hold?

4

A two-dimensional array can be thought of as a

5

In the instantiation of a two-dimensional array:

```
int[][] twoDimArr= new int[m][n];
```

what does m and n represent respectively?

[Retake Quiz](#)

That marks the end of our discussion on arrays. Let's discuss classes and inheritance in the upcoming lesson.

Introduction to Classes

In this lesson, an explanation of the basics about classes in Java- what are they and how to define them is provided.

We'll cover the following ^

- Definition
- Body of class
- Fundamentals
- Example of a pet class
 - Explanation
 - Private members
 - Public members

Definition

Classes are the building blocks of programs built using the object-oriented methodology. Objects have certain similar traits - state and behavior. This commonality is provided in a blueprint or template for the instantiation of all similar objects. This blueprint is known as a *class*.

Body of class

```
class className {  
    int integer_one;  
    String string_one;  
}
```

A *Keyword* **class** is used with every *declaration* of class followed by the name of the class. You can use any *className* as you want.

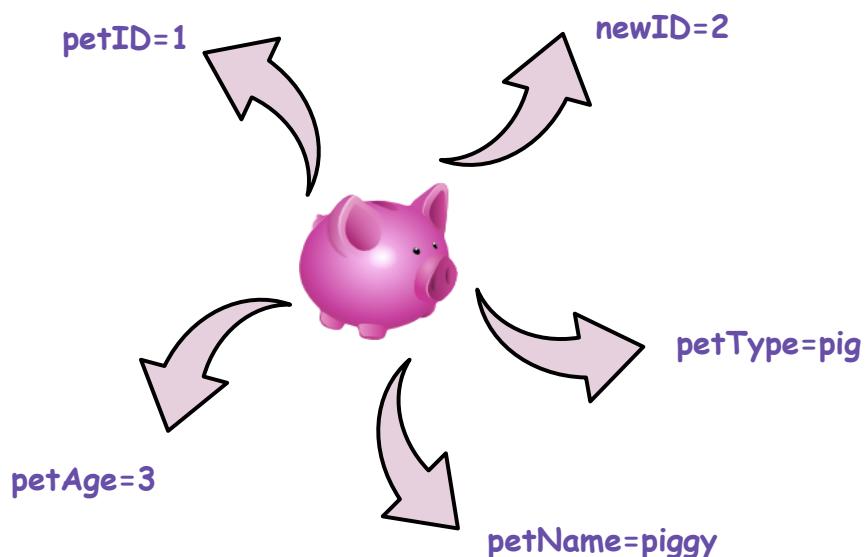
Fundamentals

Every class in Java can be composed of the following elements:

- **fields, member variables or instance variables** - These are variables that hold data specific to a particular object. For each object of a class, there is one field.
- **member methods or instance methods** - These perform operations on objects and are defined within the class.
- **static or class fields** - These fields are common to any object within the same class. They can only exist once in the class irrespective of the total number of objects in the class.
- **static or class methods** - These methods do not affect a specific object in the class.
- **inner classes** - At times a class will be defined within a class if it is a subset of another class. The class contained within another is the **inner** class.

Now that we understand these basic concepts within classes, let's look at an example of a class and see if we can understand them better.

Example of a pet class



```

class Pet {

    private static int newID;
    private int petID;
    private String petType;
    private String petName;
    private int petAge;

    public Pet(String type, String name, int age) {
        petType = type;
        petID = newID;
        petName = name;
        petAge = age;

        newID = newID + 1;
    }

    public String getName() {
        return petType;
    }

    public static int getNewID() {
        return newID;
    }

    public void printPetDetails() {
        System.out.println("Pet ID: " + petID);
        System.out.println("Pet Type: " + petType);
        System.out.println("Pet Name: " + petName);
        System.out.println("Pet Age: " + petAge);
    }
}

class PetList {
    public static void main(String[] args) {

        Pet myDog = new Pet("dog", "Ruffy", 3);
        myDog.printPetDetails();

        Pet newcat = new Pet("cat", "Princess", 2);
        newcat.printPetDetails();
    }
}

```



Explanation

For this example we have taken a **Pet class**. In this class we have five variables:

- **petType** : This determines the type of animal that the particular pet is
- **petID** : This is the identification number given to an individual pet

- `petName` : This is the name that the pet will be assigned
- `petAge` : This is the integer value of the age of the pet
- `newID` : This is a **static variable**, hence only one copy of this field will exist, no matter how many Pets are created.

An instance of a *pet*, say, a **dog** named **Ruffy**, would be an **object**. So would a **cat** named **Princess**. Hence, you can have *multiple* instances of a *class*, just like you can have *multiple* pets.

Properties, that is the variables defined in the class, are like “**inner variables**” of each *object* made of type `Pet`.

To **declare** objects of a class we should follow the format on Lines 38 and 41.

We used the **dot** operator to access members of a class *object*. See how in the above example, the methods are called in Lines 39 and 41.

Private members

As you can see above, we have used the word `private` before *declaring* the class variables. The *private members* can only be referenced within the definitions of member methods. If a program tried to access `private` *variables* directly it will get a **compiler error**.

Note: Private members can be *variables* or *methods*.

Try running the code below. It will give an error when you try to compile it, as in the code, private members of the class are being accessed directly.

```
class Pet {
    private String petName;
    private String petType;

    public Pet() {};
}

class PetList {

    public static void main(String[] args) {
        Pet myDog = new Pet();
        System.out.println(myDog.petType);
    }
}
```



The code within the same class can access private members. See the code given below!

```
class Pet {  
    private String petName;  
    private String petType;  
  
    public Pet() {};  
  
    public static void main(String[] args) {  
        Pet myDog = new Pet();  
        System.out.println(myDog.petType);  
    }  
}
```



A method can refer to a private variable of the class using the same syntax as if it were a local variable defined in the method.

Quick Question: Why we need to keep the members private?

Hide Hint

The Object Oriented paradigm takes inspiration from the fact that the state of a real world object is changed by sending messages to it, instead of directly fiddling with an object's state. Hence the motivation to keep instance variables private.

Public members

The keyword `public` identifies members of a class that can be accessed from outside of the class. Look at the **methods** in the `Pet Class` example. See how these are being called in the `main()` method without any compilation errors.

Now that we have understood classes and the basic types of elements they can

Now that we have understood classes and the basic types of elements they can have, the next lesson will delve deeper into class member methods.

Constructors

In this lesson, an explanation of what are constructors in classes and the different types of constructors that can be created is provided.

We'll cover the following

- Introduction
- Understanding constructors
- Invoking a constructor

Introduction

A *constructor* is automatically called when an *object* of the *class* is declared.

- A *constructor* is a *member* method that is usually `public`.
- A *constructor* can be used to initialize *member* variables when an *object* is declared.

Note: A *constructor's name* must be the **same** as the *name* of the *class* it is declared in.

A constructor cannot return a value.

Note: No *return type*, not even `void` can be used while declaring a *constructor*

```
class Pet {  
    private int petAge;  
    private String petType;  
    private String petName;  
  
    //This is the constructor without any parameters  
    public Pet() {  
        petAge = 0;  
        petName = "";
```



```

    petType = "";
}

//This is the constructor with parameters
public Pet(String name, String type, int age) {
    petAge = age;
    petType = type;
    petName = name;
}

//This is the copy constructor
public Pet(Pet copyThisPet) {
    petName = copyThisPet.petName;
    petType = copyThisPet.petType;
    petAge = copyThisPet.petAge;
}
}

```

Understanding constructors

As you can see the constructor syntax is pretty close to that of declaring a method. There are **three** types of constructors.

- Line 7: **Default constructor** takes no parameters and is simply used to create an object of class **Pet** without any value assigned to the variables within it.
- Line 14: **Constructor Overloading** - this constructor takes in *parameters* which are then stored in the respective variables of the newly created object.
- Line 21: **Copy constructor** - this has an object of the same class as a parameter. It then assigns the values of new objects variables to those of the input object. The purpose is to create a copy of an existing object into a new one.

Note: In copy constructor, we “usually” assign all members of one object to the other. It isn’t necessary that all members are copied. It depends on the situation.

```

class Pet {
    private int petAge;
    private String petType;
    private String petName;

    //This is the constructor without any parameters
    public Pet() {
        petAge = 0;
        petName = "";
        petType = "";
    }
}

```

```

//This is the constructor with parameters
public Pet(String name, String type, int age) {
    petAge = age;

    petType = type;
    petName = name;
}

//This is the copy constructor
public Pet(Pet copyThisPet) {
    petName = copyThisPet.petName;
    petType = copyThisPet.petType;
    petAge = copyThisPet.petAge;
}

public void print() {
    System.out.println("Pet Name: " + petName);
    System.out.println("Pet Type: " + petType);
    System.out.println("Pet Age: " + petAge);
}

}

class pet_list {
    public static void main(String[] args) {
        Pet dog = new Pet();
        dog.print();

        Pet cat = new Pet("Princess", "cat", 45);
        cat.print();

        Pet cat_copy = new Pet(cat);
        cat_copy.print();
    }
}

```



Invoking a constructor

As you can see above in **lines 36, 39, and 42**, the way to call a *constructor* is not like a normal *member* function.

- It is called in *object* declaration.
- It creates a **Pet** object.
- Then *calls* the *constructor* to initialize *variables*, this is preceded by the keyword **new**.

In the example, we also *declare* and use the **default constructor** which takes no parameters and just *initializes* **petAge** to **0** and **petName** and **petType** to empty **Strings**. As you can see in **line 36** a *default constructor* is automatically called when you create an *object*, no *parameters* are needed.

Note: It's a good practice to use default constructors even if you don't want to initialize any variables.

Similarly, on **line 39** an **overloaded constructor** is called, such that the constructor is given parameters which are then stored in the objects respective variables. This can be seen when we print the `cat` object.

Lastly, the third implementation, the **copy constructor** can be seen on **line 42**. This shows that an object of type `Pet` is passed as a parameter and the variable values of that object are copied into the new object that we create called `cat_copy`.

Now that we have a basic understanding of constructors in Java, the next lesson will be about member methods in Java.

Class Member Methods

In this lesson, an explanation of how methods can be made within classes is provided.

We'll cover the following ^

- Defining member methods
- Setters and Getters
- Example explanation

Defining member methods

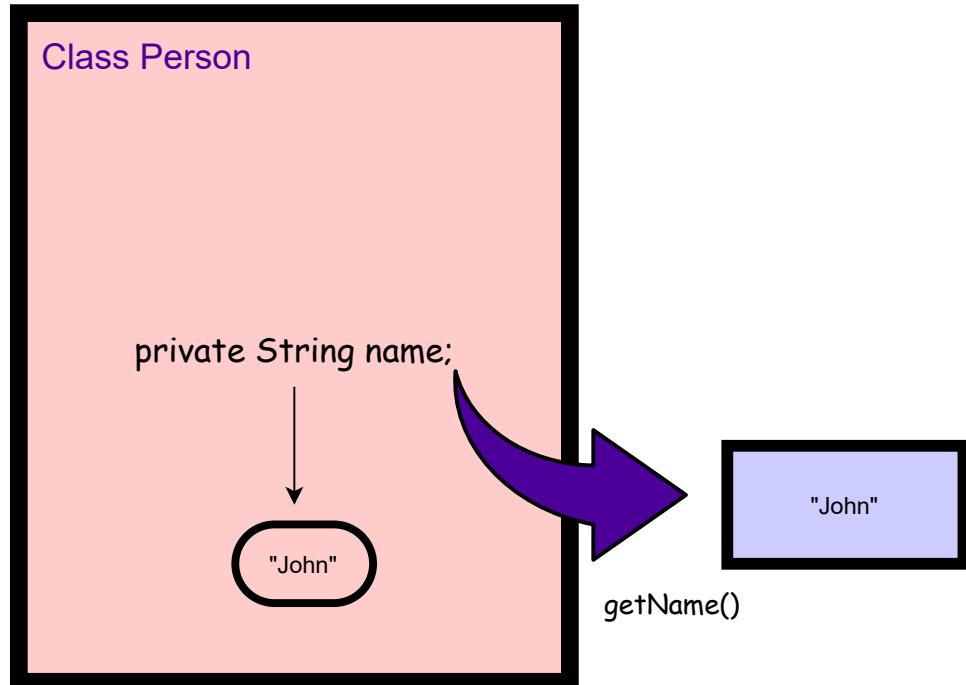
Member methods are *declared* where the class is defined. The methods are a **member** of the class within which they are written. These methods are also used to access the private parts of the class and help us perform various actions on an object through them.

Setters and Getters

As we discussed, in the previous lesson, **private** *member* variables cannot be accessed directly outside the class.

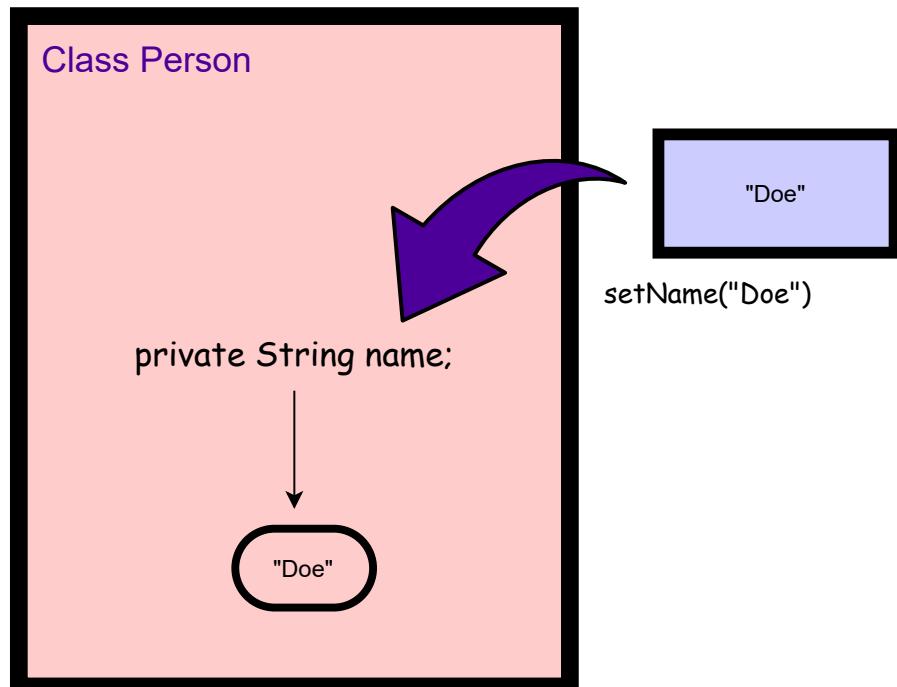
In order to *access* or *change* their values, we need to *define* **public** *member* methods. These *methods* can be used to *set* the values of the **private** variables as well as to *get* their values since being **private** these members cannot be accessed directly.

Take a look at the example below to understand this better.



Getter Method

1 of 2



Setter Method

2 of 2



```
class Pet {  
  
    private static int newID;  
    private int petID;  
    private String petType;  
    private String petName;  
    private int petAge;  
  
    public Pet(String type, String name, int age) {  
        petType = type;  
        petID = newID;  
        petName = name;  
        petAge = age;  
  
        newID = newID + 1;  
    }  
  
    public String getName() {  
        return petType;  
    }  
  
    public static int getNewID() {  
        return newID;  
    }  
  
    public void setName(String new_name) {  
        petName = new_name;  
    }  
  
    public void printPetDetails() {  
        System.out.println("Pet ID: " + petID);  
        System.out.println("Pet Type: " + petType);  
        System.out.println("Pet Name: " + petName);  
        System.out.println("Pet Age: " + petAge);  
    }  
}  
  
class PetList {  
    public static void main(String[] args) {  
  
        Pet myDog = new Pet("dog", "Ruffy", 45);  
        myDog.printPetDetails();  
        myDog.setName("Scooby");  
        myDog.printPetDetails();  
  
        Pet newcat = new Pet("cat", "Princess", 2);  
        newcat.printPetDetails();  
    }  
}
```



Example explanation

- First we make a class named `Pet` and declare the `public` and `private members`.
- `Public` members include:
 - *Methods:* `getName`, `setName` and `printPetDetails`
- `Private` members include:
 - *Variables:* `petID`, `petType`, `petName` and `petAge`

Let's take a look at all the *methods* one by one.

`getName()`

- It simply returns the `petName` variable, hence allowing the program to access a *private* variable. This is an example of a **getter** method.

`getNewID()`

- It simply returns the `newID` variable, hence allowing the program to access a *private* variable. This is also an example of a **getter** method.

`setName(String new_name)`

- Since the `petName` variable is **private** it cannot be changed directly by the `main()` method outside of the Pet class as we saw in the last lesson.
- Hence, we have a **setter** method which takes a parameter i.e. the new value of the name of the pet and *sets* the `current petName` to the `new_name`.

`printPetDetails():`

- Displays the various properties within the Pet class i.e the `name`, `age`, `type` and the `ID`.

`main(String[] args)`

- Declares **two** objects `myDog` and `newCat` for class `Pet`.
- Calls a `constructor` method to create these objects.
- *Calls* the `printPetDetails` method to show the values for each variable within object `myDog`.
- *Calls* `set` function to *update* the value for `petName` for `myDog`.

- Calls the `get` method to show the change for `petName`.
-

Now that we know how to write member methods, we move on to inheritance in Java in the next lesson.

Inheritance in Java

In this lesson, you'll learn about an important concept of Object Oriented programming known as Inheritance.

We'll cover the following ^

- What is inheritance?
- Terminology
- Notation
- What does a child have?
- Class access specifiers
- The super keyword
- Understanding the example

What is inheritance?

Inheritance provides a way to create a **new** class from an **existing** class. The **new** class is a *specialized* version of the **existing** class such that it **inherits** elements of the existing class.

Terminology

- **Superclass(Parent Class)**: This class allows the re-use of its members in another class.
- **Derived Class(Child Class or Subclass)**: This class is the one that inherits from the superclass.

Hence, we can see that classes can be built by using previously created classes!

Notation

Let's take a look at the notation necessary for the creation of **subclasses**. This is done by using the keyword **extends**.

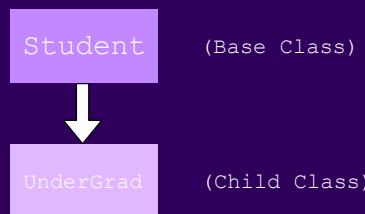
```

public String name;
public int age;
}

class Undergrad extends Student{
    String major;
    public Undergrad(){
        this.major = "Computer"
        this.name="John Doe";
        this.age=50;
    }
}

```

Note: In line 6, the derived class is declared but is followed by the word "extends" and then the name of the parent class



Inheritance establishes an "is an" relationship between *classes*.

An *object* of the *derived class* “is an” object of the *base class* or the *parent class*. For example:

- An **UnderGrad** is a **Student**.

Important Note: A *derived* object has **all** characteristics of the *parent* class.

What does a child have?

An *object* of **child** class has:

- All *members* defined in the **child** class.
- All *members* declared in the **parent** class.

An object of a child class can use:

- All **public** members defined in the **child** class.
- All **public** members defined in the **parent** class.

Note: Some classes cannot be inherited. Such classes are defined with the keyword, `final`. An example of such a class is `Integer` - this class cannot have derived classes.

Another thing to be kept in mind is that a class can only inherit from *one class*! Hence, a subclass cannot be created as a child of two classes- this creates ambiguity and would require complicated rules. Hence, **to keep it simple**, inheritance is only applicable if derived from a single class.

Class access specifiers

- `public` : the *object* of the **derived** class can be treated as an object of the **super** class.
- `protected` : more restrictive than `public`, but allows **derived** classes to know details of the *super* class.
- `private` : prevents objects of the **derived** class to be treated as objects of the **super** class.

Student

```
public String name;  
public int age;  
  
protected String grade;  
  
private String enrolled;
```

Student

```
private String enrolled;
```

```
UnderGrad
```

```
public String name;  
public int age;
```

```
Protected String grade;
```

2 of 2



The **super** keyword

This keyword allows the *derived* class to access members of its *superclass* because the `this` keyword is used to access the members of its own class. Confused? Let us look at the example below, and its explanation for a better understanding.

```
class Student {  
    public String name;  
    public int age;  
  
    public void setAge(int a) {  
        age = a;  
    }  
}  
  
class UnderGrad extends Student {  
    public UnderGrad() {  
        this.age = 10;  
        this.name = "John Doe";  
    }  
    public void set_age(int a) {  
        if (a < 50) {  
            age = 0;  
        } else {  
            super.setAge(a);  
        }  
    }  
}  
  
class example {  
    public static void main(String[] args) {  
        UnderGrad one = new UnderGrad();  
        System.out.println("Age without any method called, only constructor: " + one.age);  
        one.set_age(50);  
    }  
}
```

```
System.out.println("Age after set_age(50) is called: " + one.age);
one.set_age(10);
System.out.println("Age after set_age(10) is called: " + one.age);

}
```



Understanding the example

Lines 1 - 8:

- The **superclass** is declared in these lines. The name of the **superclass** is set as **Student**.
- The class has two variables, **name** which is of type **String** and **age** which is of type **int**.
- This class also has a **public** method called **setAge()**. This method simply takes in an **integer** as an argument and sets the age to the given input.

Lines 11 - 21:

- The **derived class** is declared in these lines. The derived class is given the name **UnderGrad** and see that on **line 11** it **extends** from the **Student** class- this indicates that it is a **child class** of this particular class.
- This class has a **constructor** which sets the **name** of the **Undergrad** student initially as **John Doe** and the **age** is set to **10**.
- The class also has a **public** method called **set_age()**. This method is interesting as it uses the keyword **super** in its body. The method takes in an **integer** parameter. It then **checks** that if the parameter is **less than 50** then, it sets the **age** as **0**. However, if this is not the case, then it simply calls the **setAge()** method of the *parent class* by using the keyword **super**. We already know what this parent class method does!

Now that we have an understanding of classes and inheritance in Java, let's look at some challenges!

Challenge 1: Calculating the Area

In this challenge, you will implement a class for calculating the area of the right angled triangle.

We'll cover the following



- Problem statement
- Coding exercise

Problem statement

Basic maths is something we all learned in our early years. Areas of shapes, their perimeters and so on! Today, we want you to calculate the area of the **right-angled triangle** using a class.

Coding exercise

Write a *class* having **two** `double` type variables for `length` and `height`, an **overloaded constructor** and **one member function** called `area` which will *return* the **area** of the right-angled **triangle**.

Only write the code where instructed in the snippet below.

Test your code against our cases and see if you can pass them.

The solution is given in case you get stuck and the next lesson will include a review of the solution, but it is highly recommended that you try it yourself first!

Good Luck!

```
class rightAngleTriangle {  
    //Define the member variables, constructor and  
    // relevant area method  
  
    //The constructor below is just to help you  
    //You can make your own or modify the one below  
    public rightAngleTriangle(int b, int a) {}  
  
}  
  
class challenge_one {  
    public static double test(rightAngleTriangle rt) {  
        //Your code here  
    }  
}
```



```
//decide what should be returned in the statement below;  
return -10;
```

```
}
```



In the next lesson, we will review the solution to the above challenge.

Solution Review: Calculating the Area

In this review, solution of the challenge 'Calculating the Area' from the previous lesson is provided.

We'll cover the following



- Solution: Did you find the 'right' area?
- Understanding the code

Solution: Did you find the 'right' area?

```
class rightAngleTriangle {  
    //Define the member variables, constructor and  
    // relevant area method  
    private int length;  
    private int height;  
  
    public rightAngleTriangle(int l, int h) {  
        length = l;  
        height = h;  
    }  
  
    public double area() {  
        return (length * height / 2.0);  
    }  
}  
  
class challenge_one {  
    public static double test(rightAngleTriangle rt) {  
        return rt.area();  
    }  
    public static void main( String args[] ) {  
        rightAngleTriangle one= new rightAngleTriangle(3,5);  
        System.out.println("Area of right Angle traingle:" + test(one));  
    }  
}
```



Understanding the code

- Lines 4-5: The two **int** type variables `length` and `height` are created. Both

variables are **private** to ensure they can only be accessed by **member** methods.

- Lines 7-9: The **overloaded constructor** is created. It takes as parameters, two **int** type variables. The **first** argument is assigned to the **length** and the **second** to the **height**. The constructor is **public** so that objects of type **rightAngleTriangle** can be made from any other class.
- Lines 12-14: This is the **method** for calculating the **area** of the triangle. Note that the method **does not** take in any parameters as it calculates the area using the *private members*. It calculates the area of the **right angle triangle** by the formula **0.5* (length * height)**. The **return type** for the method is **double** as it returns the **area** calculated.
- Lines 18-20: This creates a **static** method which simply takes in as a parameter, the **rightAngleTriangle** object and returns the **answer** of the **area** method.
 - The method was made **static** so it will belong to the whole class it is created in and not for every object of that class.

Let's display the message using the concepts of inheritance.

Challenge 2: Displaying Message Using Inheritance

In this challenge, can you extend a parent class with one function overridden for each child?

We'll cover the following



- Problem statement
- Coding exercise
- Example

Problem statement

Whenever you visit a zoo, there are many types of animals in it. However, for each animal some things never change- they all have a name and an age. Also a country of origin. This time, we want you to use the concept of classes and inheritance to solve the exercise below!

Coding exercise

The code below has:

- A **parent class** named `Animal`.
 - Inside it *define*:
 - `name`
 - `age`
 - `set_value(int a, string b):`
 - sets `age` and `name` parameters and to the given values.
 - `default constructor`
 - Then there are **two derived classes**
 - `Zebra`
 - `Dolphin`
 - The **derived classes** should

- Return a string containing a *message* telling the `age` and the `name` as well as information about *place of origin* of that *animal*.
 - Hint:** You have to create **two separate message functions** for both the **base classes**.

Example

- The `animal_type` named `name` is `age` years old. The `animal_type` comes from `origin`.
- If we have an animal of class `Zebra`, whose name is `Z` with age, `2` and the country name `Africa`. The output should be as follows:

The zebra named Z is 2 years old. The zebra comes from Africa

Only write the code where instructed in the snippet below.

Test your code against our cases and see if you can pass them.

The solution is given in case you get stuck and the next lesson will include a review of the solution, but it is highly recommended that you try it yourself first!

Good Luck!

```
class Animal {
    //declare private members here

    void set_data(int a, String b) {
        //initialize members here
    }

    //implement getters here
}

//define derived class named "Zebra" here
class Zebra extends Animal {
    String message_zebra(String str) {
        //define here
        str = "No code added yet"; //update this when you write your code
        return str;
    }
}

//define derived class named "Dolphin" here
class Dolphin extends Animal {
    String message_dolphin(String str) {

```

```
//define here  
str = "No code added yet"; //update this when you write your code  
return str;  
}  
}
```



Let's go over the solution review in the upcoming lesson.

Solution Review: Displaying Message Using Inheritance

In this review, solution of the challenge 'Displaying Message Using Inheritance' from the previous lesson is provided.

We'll cover the following

- Solution: Displaying message using inheritance
- Understanding the Code
 - *
 - Lines 1 - 19: Creating the Animal class
 - Lines 21 - 36: Creating the Zebra & Dolphin class

Solution: Displaying message using inheritance

```
class Animal {  
    private int age;  
    private String name;  
  
    void set_data(int a, String b) {  
  
        age = a;  
        name = b;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public int getAge() {  
        return age;  
    }  
}  
  
class Zebra extends Animal {  
  
    String message_zebra(String str) {  
        str = "The zebra named " + getName() + " is " + getAge() + " years old. The zebra comes fro  
        return str;  
    }  
}  
  
class Dolphin extends Animal {
```

```
String message_dolphin(String str) {  
    str = "The dolphin named " + getName() + " is " + getAge() + " years old. The dolphin come  
    return str;  
}  
}
```



Understanding the Code

Lines 1 - 19: Creating the Animal class

- On lines 3,4 the member variables are declared that are **private** to the student class. These include an **int** type variable **age** and a second variable is also declared, **name** which is of type **String**.
- From lines 6-10 is the *method* named **set_data**. This according to the requirements, takes in parameters and sets both **age** and **name** to the passed parameters.
- From lines 12-14, is the method **getName()** which returns a **String** type of value.
- Lines 16-18 show the method **getAge()** which returns a **int** type of value.

Lines 21 - 36: Creating the Zebra & Dolphin class

- The **Zebra** & **Dolphin** class **inherit** from the **Animal** class, each contain only one method named **method_zebra** and **method_dolphin** respectively. A variable **str** is being passed to the methods to store the output **String** according to the respective classes. Getters are used to showcase the inheritance because we are using them without instantiating the **Animal** class.

Let's wrap up this chapter by solving a quiz in the upcoming lesson.

Quick Quiz!

1

The following access modifier prevents the member of a class to be accessed outside of the class.

2

What is the type of the following method and what does it do? Choose the best answer.

```
public int age_student()
{
    System.out.println(this.age);
    return this.age;
}
```

3

What kind of constructor for class **Student** is shown?

```
public Student(Student s){  
    this.age=s.age;  
    this.name=s.name;  
}
```

4

There is no difference in a protected and public variable of a mother class in terms of accessibility and use for a derived class.

Is this statement correct?

5

The keyword `super` is used to access members of a derived class in a mother class.

Is this statement correct?

6

Which keyword signifies that a class is being derived from another class?

Retake Quiz

In the upcoming chapter, we will discuss generics in Java.

Introduction to Generics

In this lesson, an explanation is provided to get started with Generics Methods (functions) in Java.

We'll cover the following

-  Definition
- Generic methods
 - Syntax
 - Generic methods with multiple type parameters

Definition

"Generics allow the reusability of code, where one single method can be used for different **data-types** of variables or objects."

The idea is to allow different types like `Integer`, `String`, ... etc and user-defined types to be a parameter to methods, classes, and interfaces.

For example, classes like `HashSet`, `ArrayList`, `HashMap`, etc use generics very well. We can use them for any type.

The following example illustrates three **non-generic** (type-sensitive) functions for finding maximum out of 3 inputs:

```
class Main {  
    public static void main(String[] args) {  
        System.out.printf("Max of %d, %d and %d is %d\n\n", 3, 4, 5,  
            MaximumTest.maximum(3, 4, 5));  
        System.out.printf("Max of %.1f,%.1f and %.1f is %.1f\n\n", 6.6, 8.8, 7.7,  
            MaximumTest.maximum(6.6, 8.8, 7.7));  
        System.out.printf("Max of %s, %s and %s is %s\n", "pear", "apple", "orange",  
            MaximumTest.maximum("pear", "apple", "orange"));  
    }  
}  
class MaximumTest {  
    // determines the largest of three Comparable objects  
    public static int maximum(int x, int y, int z) {  
        int max = x; // assume x is initially the largest
```

```

if (y > max) {
    max = y; // y is the largest so far
}

if (z > max) {
    max = z; // z is the largest now
}
return max; // returns the largest object
}

public static double maximum(double x, double y, double z) {
    double max = x; // assume x is initially the largest

    if (y > max) {
        max = y; // y is the largest so far
    }

    if (z > max) {
        max = z; // z is the largest now
    }
    return max; // returns the largest object
}
public static String maximum(String x, String y, String z) {
    String max = x; // assume x is initially the largest

    if (y.compareTo(max) > 0) {
        max = y; // y is the largest so far
    }

    if (z.compareTo(max) > 0) {
        max = z; // z is the largest now
    }
    return max; // returns the largest object
}
}

```



Three *methods* that do exactly the same thing, but cannot be defined as a single method because they use *different* data types. (`int`, `double` & `String`)

Generic methods

To use generic methods, we use the following syntax.

Syntax

```
<T> void MyFirstGenericMethod(T element)
```

`T` is our *generic* data type's name, and when the method is to be called, it would be the same as if `T` was a `typedef` for your datatype.

The following example now illustrates how the `multiply` method would be written

The following example now illustrates how the `maximum()` interacts with and its variation using a *template*:

```
class Main {  
    public static void main(String[] args) {  
        System.out.printf("Max of %d, %d and %d is %d\n\n", 3, 4, 5,  
            MaximumTest.maximum(3, 4, 5));  
        System.out.printf("Max of %.1f,%.1f and %.1f is %.1f\n\n", 6.6, 8.8, 7.7,  
            MaximumTest.maximum(6.6, 8.8, 7.7));  
        System.out.printf("Max of %s, %s and %s is %s\n", "pear", "apple", "orange",  
            MaximumTest.maximum("pear", "apple", "orange"));  
    }  
}  
  
class MaximumTest {  
    // determines the largest of three Comparable objects  
    public static < T extends Comparable < T >> T maximum(T x, T y, T z) {  
        T max = x; // assume x is initially the largest  
  
        if (y.compareTo(max) > 0) {  
            max = y; // y is the largest so far  
        }  
  
        if (z.compareTo(max) > 0) {  
            max = z; // z is the largest now  
        }  
        return max; // returns the largest object  
    }  
}
```



Generic methods with multiple type parameters

In the above code, all of the arguments to `maximum()` must be the same type.

Optionally, a template can have more type options, and the syntax is pretty simple.

For a template with **three** types, called `T1`, `T2` and `T3`, we have:

```
class Generics {  
    public static < T1, T2, T3 > void temp(T1 x, T2 y, T3 z) {  
        System.out.println("This is x =" + x);  
        System.out.println("This is y =" + y);  
        System.out.println("This is z =" + z);  
    }  
    public static void main(String args[]) {  
        temp(1, 2, 3);  
    }  
}
```



In the next lesson, we'll take a look at templates in some more detail!

Generic Class

In this lesson, we explain how to create and use Generic Class Objects & functions of the Generic Class.

We'll cover the following ^

- Generic objects & type members
- Example
 - Explanation
 - Syntax to instantiate object

Generic objects & type members

As another powerful feature of Java, you can also make *Generic classes*, which are classes that can have *members* of the **generic** type.

Example

```
class Test<T>
{
    T obj;      // An object of type T is declared
    Test(T obj) // parameterized constructor
    {
        this.obj = obj;
    }
    public T getObject() // get method
    {
        return this.obj;
    }
}
```

Explanation

- We have declared a class `Test` that can keep the `obj` of any type.
- The constructor `Test(T obj)` assigns the value passed as a parameter to data member `obj` of type `T`.
- The get method `T getObject()` returns the `obj` of type `T`.

Syntax to instantiate object

To create objects of the generic class, we use the following syntax.

```
// To create an instance of generic class  
Test <DataType> obj = new Test <DataType>()
```

Have a look at the detailed implementation of Generic Class and its methods.

```
// We use <> to specify Parameter type  
class Test < T > {  
    T obj;  
    Test(T obj) {  
        this.obj = obj;  
    }  
    public T getObject() {  
        return this.obj;  
    }  
}  
class Main {  
    public static void main(String[] args) {  
        // Test for Integer type  
        Test < Integer > obj1 = new Test < Integer > (5);  
        System.out.println(obj1.getObject());  
  
        // Test for double type  
        Test < Double > obj2 = new Test < Double > (15.777755);  
        System.out.println(obj2.getObject());  
  
        // Test for String type  
        Test < String > obj3 = new Test < String > ("Yayy! That's my first Generic Class.");  
        System.out.println(obj3.getObject());  
    }  
}
```



As you can see we create 3 different `Integer`, `Double` and `String` type variables for three different generic class objects.

You can play around with more datatypes to test the above class to check your understanding.

This marks the end of our discussion on *Generics*. Next up we'll look at some more coding challenges and will step through them in detail.

Challenge 1: Finding Max in an Array

In this exercise you have to use template type to generalize the method used to find max element in an array.

We'll cover the following ^

- Problem statement
 - Expected input
 - Expected output
- Coding challenge

Problem statement

In the code widget below, **two** methods both called `array_max` are declared. One finds **max** value for `int` type inputs and the other for `double` type.

In this exercise, you need to define a **Generic Class type** method called `array_max` that will generalize the method such that it finds **maximum** value for both `int` and `double` type array input values.

IMPORTANT NOTE: Remove both the `int` and `double` type `array_max` methods and then write the code for the **Generic Class type** `array_max` method there.

Your generic class type `array_max` method code should find the **maximum** element in an array using generics.

- Method will take the *array* and *array size* as parameters.

Expected input

Input 1:

If input is:

```
int arr[] = {2,8,20,3,4};
```

Input 2:

If input is:

```
double arr[] = {2.7,3.8,5.5,6.7,9.7}
```

Expected output

Expected Output 1:

```
20
```

Expected Output 2:

```
9.7
```

Coding challenge

You have to write a function that uses the Generic Class type to write a single function that works for both `int` and `double` inputs. Remove the two functions below in the code widget and instead write a Generic class type `array_max` function that finds max value for either of the two types.

Write your code below. It is recommended that you try solving the exercise yourself before viewing the solution.

Good Luck!

```
class FindMax {  
    public static Integer array_max(Integer data[], int n) {  
        //body of code  
        return 0;  
    }  
  
    public static Double array_max(Double data[], int n) {  
        //body of code  
        return 0.0;  
    }  
}
```



Hint 1 of 1

< Think of reusing the solution for finding max out of 3 inputs in
the previous lesson. >

Let's go over the solution review in the upcoming lesson.

Solution Review: Finding Max in an Array

In this review, solution of the challenge 'Finding Max in an Array' from the previous lesson is provided.

We'll cover the following ^

- Given solution
- Explanation

Given solution

```
class FindMax {  
    public static < T extends Comparable < T >> T array_max(T data[], int n) {  
        T max = data[0];  
        int i;  
        for (i = 1; i < n; i++) {  
            if (max.compareTo(data[i]) < 0) {  
                max = data[i];  
            }  
        }  
        return max;  
    }  
    public static void main( String args[] ) {  
        Integer[] inputs1 = {2,8,20,3,4};  
        Double[] inputs2 = {2.7,3.8,5.5,6.7,9.7};  
        System.out.println(array_max(inputs1,5));  
        System.out.println(array_max(inputs2,5));  
    }  
}
```



Explanation

- `public static <T extends Comparable<T>> T array_max(T data[], int n)`

A static function is defined that extends the Comparable class (already implemented in Java) since we use a comparison function later in the program. `T` defines the generic user data-type. An array of generic type `T` is also defined as `T data[]`, where `n` is the size of `data[]`

- `I max = data[0];`

Declares a variable `max` of generic type and store the element at 0th index in it.

- `for(int i = 1; i < n; i++)`

for loop to traverse through the array `data[]`

- `if(max.compareTo(data[i]) < 0)`

check if the `max` is smaller than the element at the *i*th index. If yes, then the built-in function `.compareTo()` returns -1 in this case.

- `max = data[i];`

If the element at `ith` index is greater than `max` then replace the `max` with the element at the *i*th position.

- At the end return `max` to the calling point.

Let's wrap up this chapter by solving a quiz in the upcoming lesson.

Quick Quiz!

Here is a quick quiz to check your understanding of Generics.

1

Which of the following is true about Generics?

2

Which keyword can be used in Generics?

3

What is the output of the following program?

```
class GetMax {  
    public static <T extends Comparable<T>> T maximum (T a, T b)  
    {  
        if(a.compareTo(b) > 0)  
            return a;  
        else  
            return b;  
    }  
    public static void main( String args[ ] )  
    {  
        System.out.println(maximum(4, 6));  
        System.out.println(maximum(4.0, 6.1));  
        System.out.println(maximum(4, 6.7));  
    }  
}
```

4

What is the output of the following program?

```
class Score <T>
{
    T value;
    public int counter = 0;
    public Score() {counter++;}
    public Score(T v) { value = v; counter++;}

class Main
{
    public static void main(String args[])
    {
        Score <Integer> x = new Score <Integer>();
        Score <Integer> y = new Score <Integer>();
        Score <Double> z = new Score <Double> ();

        System.out.println(x.counter);
        System.out.println(y.counter);
        System.out.println(z.counter);

    }
}
```

5

A generic class declaration looks like a non-generic class declaration, except that the class name is followed by a **type** parameter section.

6

With generic methods, the type parameter section of a generic class can NOT have one or more type parameters separated by commas.

7



What are the benefits of Generics in Java?

Generics can be used for more than one types defined in the form of comma separated list in the class / function header.

```
public static <A,B,C> void MultipleTypesFunction(A x, B y, C z)
{
    System.out.println( x + " is Type A");
    System.out.println( y + " is Type B");
    System.out.println( z + " is Type C");
}
```

Retake Quiz

In the next chapter, we will discuss array lists.

ArrayLists in Java

In this lesson, an explanation of all the basics about the inbuilt ArrayList class in Java is provided.

We'll cover the following



- What is an ArrayList?
- Why to use ArrayLists?
- How does it work?

What is an ArrayList?

ArrayList is a `class` in Java which extends the `AbstractList` class and implements the `List` interface.

Why to use ArrayLists?

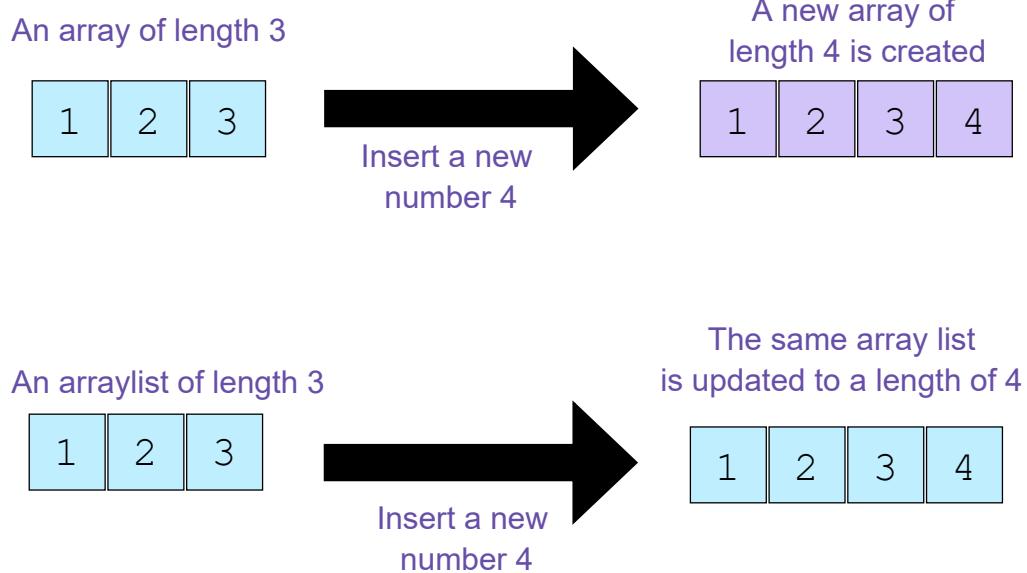
We have already discussed the Arrays in Java. As it has been mentioned earlier that the length of an Array is set at the time of its declaration and cannot be varied later at any point in the program.

For example, if the length of an array is set to 5 at the start of a program it cannot store **more than 5** elements throughout the program. To overcome this, we will then have to declare a `new` array with the length set to some *greater number* and copy the contents of the previous array to this `new` array. Similarly, if this array is used to store **less than 5** elements then the excess allocated space is useless and is a wastage of the *memory resource*.

More often, we are not sure of the total number of data items we are going to store in an array later in the program. So, to overcome the said issue we need a *dynamic* variation in the length of an array according to the required space for the storage of elements. This can be easily achieved using the `ArrayList`.

How does it work?

An ArrayList actually uses an array to store data. Whenever the number of elements to be stored exceeds the `length` of the current array, the contents of this array are copied and pasted into a `new` array of greater size. Hence, the size of an `arrayList` varies dynamically i.e. during the *runtime* of a program.



Going good so far? In the next lesson we will discuss how we can create an Object of ArrayList class.

Creating an ArrayList Object

In this lesson, you'll learn how to create an ArrayList object in Java.

We'll cover the following

- Wrapper classes
- Generics in action
 - Instantiation
 - Empty ArrayList
 - ArrayList with initially defined Capacity

Let's practically use an *ArrayList Object* in our code. As we already know, the `ArrayList` is a class in Java and to use a class's functionality we have to *instantiate* it i.e. to construct its *object*.

To construct an `object` of a `class` we use a constructor. Before we jump to the constructor part let's discuss a bit about the `wrapper classes`.

Wrapper classes

In Java, we cannot *directly* instantiate an `ArrayList` of the primitive data types like `int`, `char`, `boolean`. The reason for this is that primitive data types are not *objects*. For this purpose, Java has inbuilt `wrapper classes` which just *wrap* these primitive data types in a `class`. Below given is the list of *inbuilt wrapper classes*:

Primitive Data Type	Wrapper Class
<code>boolean</code>	<code>Boolean</code>
<code>byte</code>	<code>Byte</code>
<code>char</code>	<code>Character</code>
<code>double</code>	<code>Double</code>

float

Float

int

Integer

long

Long

short

Short

Note: `String` is not a primitive data type. These are objects instantiated from a `String` base class in Java, so they **don't need** any wrapper class.

Generics in action

The above wrapper classes are passed to the `ArrayList` class as *type parameters* inside the *pair* of angle brackets `<>`. This is exactly the concept of [Generics](#). Syntactically we can generalize the declaration of an `ArrayList` as:

```
ArrayList<Type> name;
```

Instantiation

The `ArrayList` objects can be instantiated using the *keyword* `new` and the `ArrayList` `class` are **three** types of constructors which are discussed below:

Empty ArrayList

The following is the basic and the most used way to instantiate an `ArrayList` of `Integer` data type.

```
ArrayList<Integer> myarrList = new ArrayList<>();
```

Instantiation of an Empty Integer ArrayList

The above line of code will instantiate an empty `Integer` `ArrayList`. The array on which this `ArrayList` is based has a `length` of `10` by-default at the time of instantiation and this size grows dynamically (*during runtime*) according to the required number of memory locations to store the elements.

ArrayList with initially defined Capacity

We can instantiate an ArrayList with an initially defined capacity to ensure the space allocation at the time of instantiation.

The **capacity** is the number of elements the list can potentially accommodate without reallocating its internal structures.

Let's instantiate an ArrayList of **Char** using the respective *Wrapper Class* with an initial capacity of **20** elements.

```
ArrayList<Character> chArrList = new ArrayList<Character>(20);
```

Instantiation of an ArrayList with Initial Capacity = 20

We can notice that the concept of constructor overloading is being implemented in the above line of code i.e. rather than calling an *empty* constructor, we are passing *initial capacity = 20* to it.

The above declared ArrayList will reallocate its resources and grow dynamically once it runs out of these **20** spaces.

In the next lesson, let's check out how ArrayLists come in handy by using their in-built methods.

Inbuilt Methods

In this lesson, the mostly used inbuilt methods of ArrayList class are explained.

We'll cover the following ^

- Adding an element
- Getting or Setting an element
- Size of an ArrayList
- Some more inbuilt methods

ArrayLists have plenty of inbuilt methods that can be accessed by the programmers to store and manage data. We will frequently come across method overloading in these inbuilt methods below.

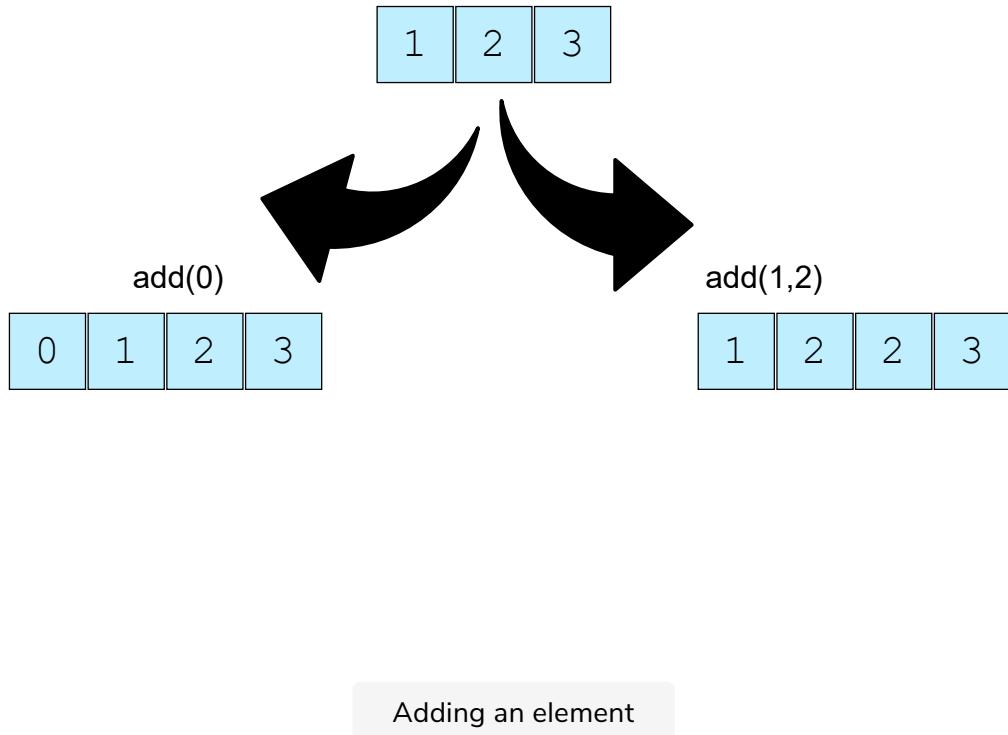
Adding an element

Addition of the elements to ArrayLists can be achieved using `add()` method:

Method Name	Description	Return Type
<code>add(object)</code>	The item is added to the end of the current ArrayList <code>boolean</code>	
<code>add(index, object)</code>	Adds a single object of the respective type at the specified <i>index</i> to the ArrayList	<code>void</code>

Note: `add(index, object)` method pushes the element on the current index to the ArrayList.

Note. `add(index, object)` method pushes the element on the current index to the next index.

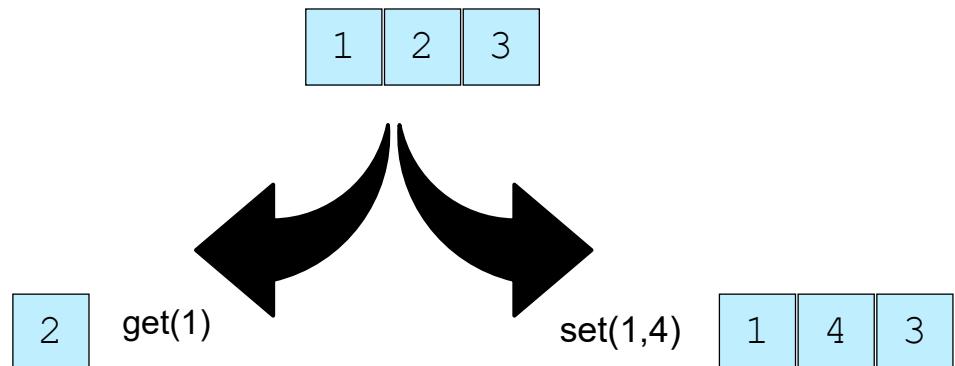


Adding an element

Getting or Setting an element

Access to the stored elements in an ArrayList can be achieved using `get()` method. We can also *set* the element at a specific position using the method `set()`.

Method Name	Description	Return Type
<code>get(index)</code>	Gets the object stored at the specified <i>index</i> in the ArrayList	<code>Object</code>
<code>set(index, object)</code>	Replaces the object at the specified <i>index</i> in the ArrayList with the <i>object</i> passed as parameter	<code>Object</code>



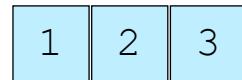
Getting or Setting an element

Size of an ArrayList

We can get the number of elements stored in an ArrayList using the `size()` method.

Method Name	Description	Return Type
<code>size()</code>	Returns the No. of elements stored in an ArrayList	<code>int</code>

Let's use the above methods in a coding example to strengthen our understanding.



size() → **3**

Finding the length/size of the array list

```
class ArrList {  
    public static void main(String args[]) {  
        ArrayList < String > emails = new ArrayList < String > (); //Instantiation  
        emails.add("user1@abc.com"); //adds at index 0  
        emails.add("user3@abc.com"); //adds at index 1  
  
        System.out.println("The two added elements are:");  
        System.out.println("1. " + emails.get(0));  
        System.out.println("2. " + emails.get(1));  
        System.out.println("The current size of the ArrayList is: " + emails.size() + "\n");  
  
        emails.add(1, "user2@abc.com"); //adds at index 1 pushes the index 1  
        //element to index 2  
        System.out.println("After adding an element to index 1:");  
        System.out.println("1. " + emails.get(0));  
        System.out.println("2. " + emails.get(1));  
        System.out.println("3. " + emails.get(2));  
        System.out.println("The current size of the ArrayList is: " + emails.size());  
    }  
}
```

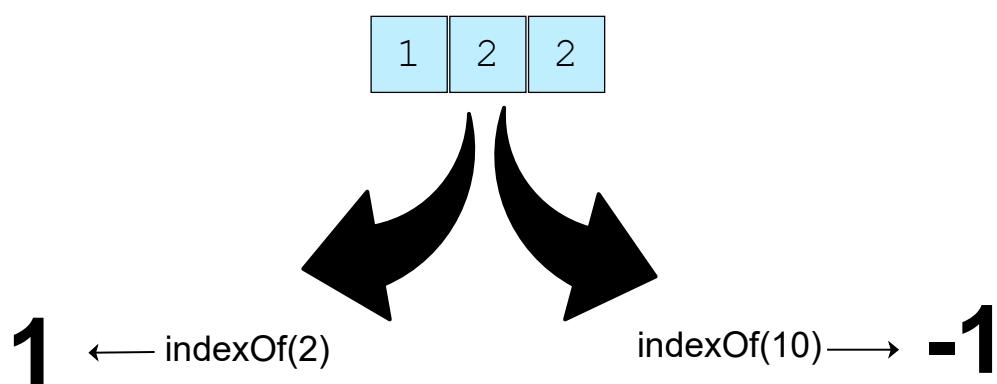


Inbuilt ArrayList Methods

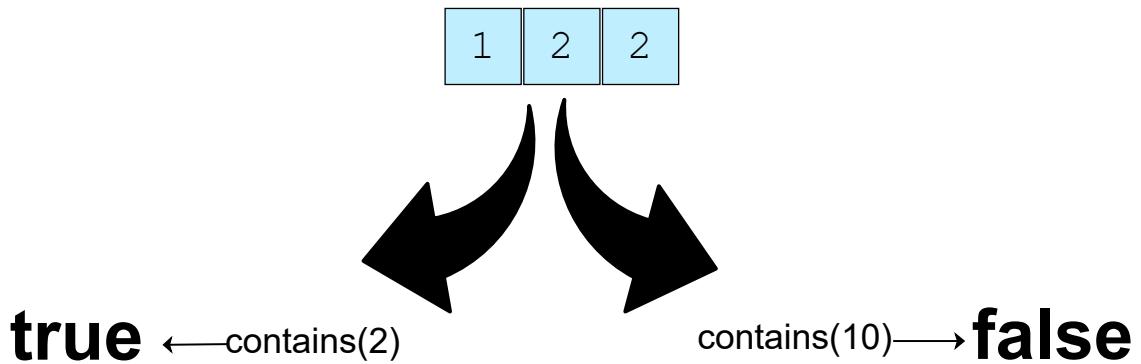
Some more inbuilt methods

Method Name	Description	Return Type
<code>indexOf(object)</code>	Returns the index of the first occurrence of the specified element, or -1 if the List does not contain this element	<code>int</code>
<code>contains(object)</code>	Returns true if this list contains the specified element	<code>boolean</code>
<code>remove(object)</code>	Returns true after removing the specified element	<code>boolean</code>

Duplicate elements are allowed in the ArrayLists.

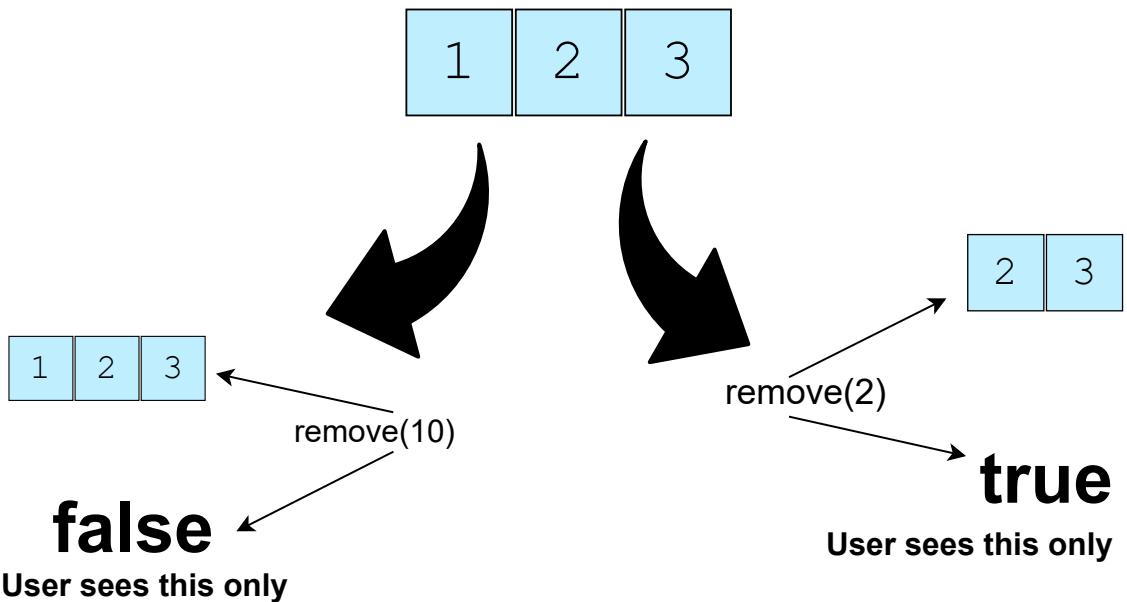


Finding the index of an element



Does the array list contain this value

2 of 3



Removing an element from the arraylist

3 of 3



A list of all the inbuilt methods can be found [here](#).

Let's put our understanding of ArrayLists into practice by taking up some coding challenges.

Challenge 1: Gathering Zeros to the Start

In this challenge, you'll implement the solution to gather all the zeros stored in an ArrayList to the starting indices.

We'll cover the following ^

- Problem statement
 - Function prototype
 - Output
 - Sample input
 - Sample output

Problem statement

In this problem, you have to implement the `zerosToStart()` function which will sort the elements of an `Integer` ArrayList such that all the `zeros` appear at left and other elements appear at the right.

Function prototype

```
void zerosToStart(ArrayList<Integer> arrList)
```

Output

A sorted ArrayList with zeros at the left i.e. starting indices and positive elements at the right.

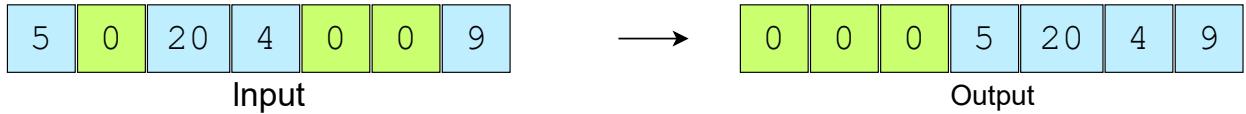
Sample input

```
arrList = [5,0,20,4,0,0,9]
```

Sample output

```
arrList = [0,0,0,5,20,4,9]
```

Note: All non zero elements must appear in the original order.



```
class ArrList {  
    public static void zerosToStart(ArrayList<Integer> arrList) {  
        //write your code here  
    }  
}
```



Gathering Zeros to the Start

In the next lesson, we will review the solution of the above challenge.

Solution Review: Gathering Zeros to the Start

In this review, solution of the challenge 'Gathering Zeros to the Start' from the previous lesson is provided.

We'll cover the following

- Solution
 - How does the above code work?

Solution

```
class ArrList {  
    public static void zerosToStart(ArrayList < Integer > arrList) {  
        ArrayList < Integer > newArrayList = new ArrayList < Integer > ();  
        int newArray_index = 0;  
  
        //Fill newArrayList with Zeros first.  
        //Then Fill it with non-zero Values.  
        //In the end, insert every element of newArrayList back into origional arrList.  
        for (int i = 0; i < arrList.size(); i++) {  
  
            if (arrList.get(i) == 0)  
                newArrayList.add(newArray_index++, arrList.get(i));  
        }  
  
        for (int i = 0; i < arrList.size(); i++) {  
  
            if (arrList.get(i) != 0)  
                newArrayList.add(newArray_index++, arrList.get(i));  
        }  
  
        for (int j = 0; j < newArrayList.size(); j++) {  
            arrList.set(j, newArrayList.get(j));  
        }  
    }  
    public static void main( String args[] ) {  
        ArrayList<Integer> input = new ArrayList<Integer>(Arrays.asList(5, 0, 20, 4, 0, 0, 9));  
        System.out.println("Array List before calling zerosToStart");  
        for (int i = 0; i < input.size(); i++){  
            System.out.print(input.get(i)+ " ");  
        }  
        System.out.println();  
        ArrList.zerosToStart(input);  
        System.out.println("Array List after calling zerosToStart");  
        for (int i = 0; i < input.size(); i++){  
            System.out.print(input.get(i)+ " ");  
        }  
        System.out.println();  
    }  
}
```

```
}
```



...

How does the above code work?

In the above solution code, we have created a new `Integer` `ArrayList`. Using the for loop, first, we have copied the zeros from the original `ArrayList` i.e. `arrList` to the `newArrayList` meanwhile keeping track of the index via using an `int` variable.

In the next loop, we have copied the *non-zero* elements to the `newArrayList` in a similar way. Now that, all the elements have been arranged in the given order and added to the `newArrayList` we have implemented a for loop to copy the sorted elements back to the `arrList`.

In the next lesson, there will be another coding challenge to test your understanding of the `ArrayLists`.

Challenge 2: Remove Duplicates From an ArrayList

In this challenge, you'll have to code for removing the duplicates from an ArrayList.

We'll cover the following ^

- Problem statement
- Function prototype
- Output
- Sample input
- Sample output

Problem statement

Given a **Character** ArrayList, you have to implement a Java method to remove all the *duplicate elements* from it.

Function prototype

```
void removeDuplicates(ArrayList<Character> arrList)
```

Output

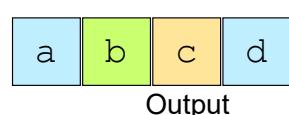
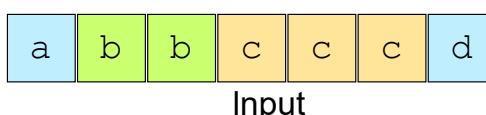
An ArrayList with all the unique elements.

Sample input

```
arrList = [a,b,b,c,c,c,d]
```

Sample output

```
arrList = [a,b,c,d]
```



```
class DuplicatesRemover {  
    public static void removeDuplicates(ArrayList<Character> arrList) {  
        //write your code here  
    }  
}
```



Removing Duplicates

Let's go over the solution review in the upcoming lesson.

Solution Review: Remove Duplicates From an ArrayList

In this review, solution of the challenge 'Remove Duplicates From an ArrayList' from the previous lesson is provided.

We'll cover the following

- Solution
- How does the above code work?

Solution

```
class DuplicatesRemover {  
    public static void removeDuplicates(ArrayList < Character > arrList) {  
        for (int i = 0; i < arrList.size(); i++) {  
            for (int j = i + 1; j < arrList.size(); j++) {  
                if (arrList.get(i).equals(arrList.get(j))) { //check if there is any duplicate  
                    arrList.remove(j); //remove duplicate  
                    j--; // j is decremented  
                }  
            }  
        }  
    }  
    public static void main( String args[] ) {  
        ArrayList<Character> input = new ArrayList<Character>(Arrays.asList('d', 'c','a', 'b', 'b',  
        System.out.println("Array List before calling removeDuplicates");  
        for (int i = 0; i < input.size(); i++){  
            System.out.print(input.get(i)+ " ");  
        }  
        System.out.println();  
        removeDuplicates(input);  
        System.out.println("Array List after calling removeDuplicates");  
        for (int i = 0; i < input.size(); i++){  
            System.out.print(input.get(i)+ " ");  
        }  
        System.out.println();  
    }  
}
```



How does the above code work?

- In the above solution code, we have implemented a nested `for loop` for comparing a single element at a time with the entire ArrayList's elements

iteratively.

- If any of the duplicates are found, we use the inbuilt `remove()` method to remove that duplicate element. `j--` decrements the inner for loop's iteration value so that we may tackle three or more consecutive duplicates.
-

In the next lesson, there will be a quick quiz to test your understanding of the ArrayLists.

Quick Quiz!

Let's test your understanding by solving a quiz in this lesson.

1

The Size of an ArrayList

2

The wrapper classes are used to convert primitive data types into

3 !

We can check the number of elements in an ArrayList using

4 !

Duplicates in an ArrayList are not allowed.

Retake Quiz