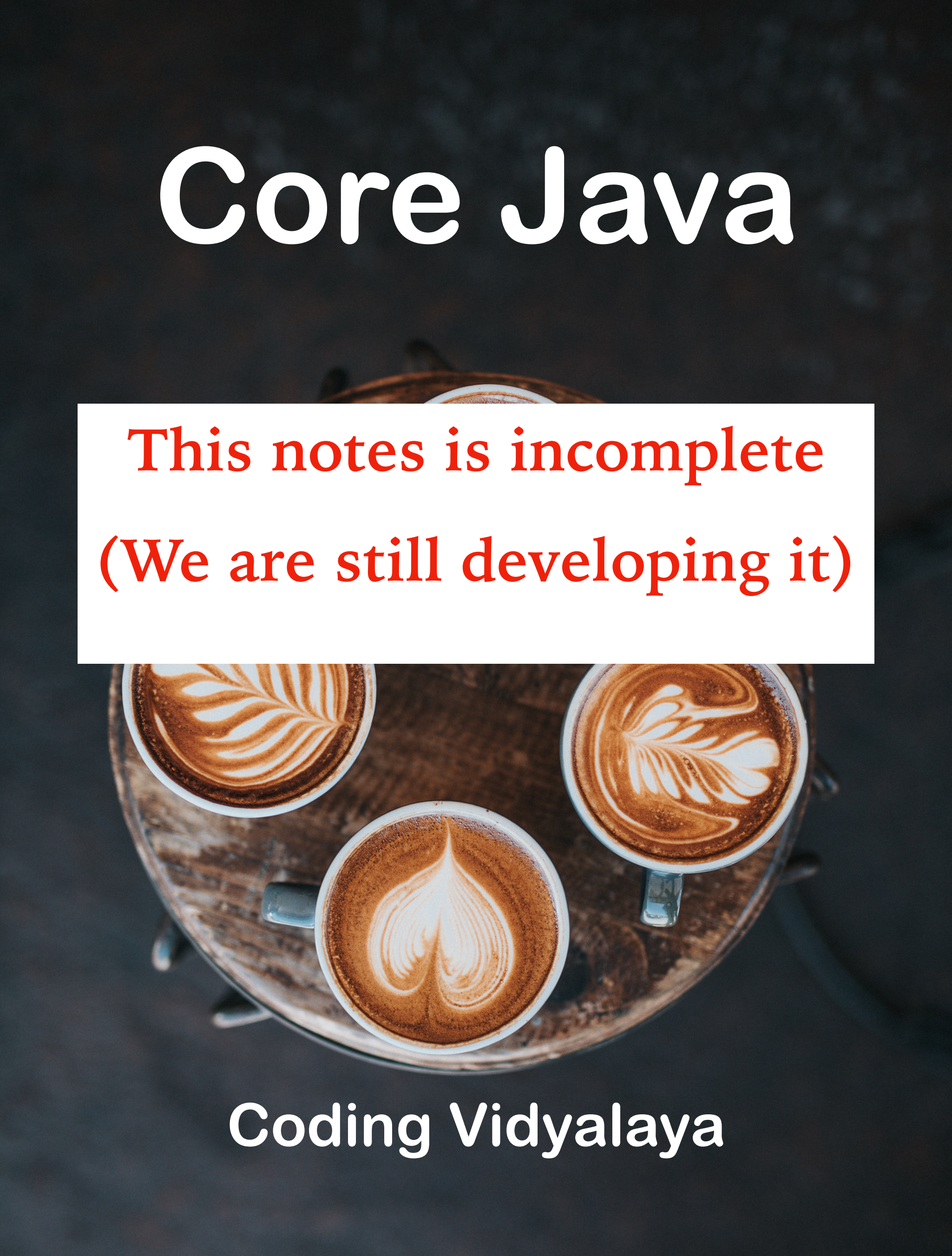
Core Java

This notes is incomplete

(We are still developing it)



Coding Vidyalaya

1 Fundamentals of Java

Java is the most widely used language of the world. This is not a co-incidence. This language was designed keeping in mind the learnings of other good languages such as C++. Java has also been able to keep pace with the industry in keeping itself updated and providing latest features.

Introduction of Java

Java was conceived in 1991 at Sun Microsystem. This was initially called “Oak”, but was later renamed to “Java" in 1995.

Most popular language at the time, C++, used to be compiled for each platform on which it was run. C++ program compiled on a windows machine with one particular processor architecture cannot be run on another windows machine with a different processor architecture.

Java designer's main aim was to build a language that can be run anywhere. If you write a Java program on your Linux machine, and share the bytecode(explained in next section) with your friend who has a windows machine or a Mac, h/she should be able to run it directly without having to compile it again. This gives Java, portability and platform independence.

Rise of internet coupled with Java’s platform independence, let to explosion in Java’s popularity and demand.

Reason for Popularity: Bytecode and JVM

We know that Java is platform independent. But, how does it achieve this feature?

When you compile Java, you don’t get executable, you get bytecode. And this bytecode is run by Java Virtual Machine(JVM). Once you have a JVM available for a given platform, you can run Java bytecode generated on any platform. It must be noted that, JVM itself is not same for all the platforms, it is implemented separately for separate platforms.

You can think of JVM as a cord, with one end that gets connected to bytecode. This end always remains same. The second end of this cord gets connected to platforms, this end depends on the platform. Since the end connected to bytecode remains the same, the same bytecode can be run on any platform that has a JVM available.

In C++ or many other popular languages, when you compile your program, you get executable that is run directly on the platform. Unlike Java's JVM, no such glue is available for C++ that understand the same bytecode and runs on every platform. So, C++ is not portable.

HotSpot Technology

Originally, JVM was designed to be an interpreter for bytecode. But, soon after Java's early release, HotSpot technology was introduced. It provides a Just In Time(JIT) compiler, that compiles a selected piece of the Java bytecode and converts them into executable. It is done real time, piece-by-piece, on demand basis.

There are two types of languages, interpreted and compiled. In simple terms, interpreted languages are run line by line whereas compiled languages are compiled once and the generated executable is run. Since, compiled languages compile everything at one go, they are faster than interpreted language. But, Java is able to take advantage of compilation using HotSpot Technology and JIT, they have significantly improved in terms of speed.

Security

Since, Java bytecode is run inside JVM, the a malicious program cannot access system's resources without JVM's permission.

Since, JVM controls the access to resources available to programs, Java is considered secure. Moreover, since JVM is not under control of software vendors, there is very little chance, that it can be exploited.

JAVA BUZZWORDS

Key consideration by Java design team for modelling the language is contained in this table.

SImple	Secure	Portable	Object Oriented
Robust	Multi-threaded	Architecture-neutral	Interpreted
High Performance	Distributed	Dynamic	

Fundamentals of Object Oriented Programming

Multiple paradigms of programming exist, The most popular being

- Imperative
    - Procedural
    - Object Oriented
- Declarative
    - Functional

Languages supporting OOP has three common traits: Encapsulation, Polymorphism, and Inheritance.

Encapsulation is the mechanism by which closely related code and data is bind together, and some sort of mechanism is employed to protect it from outside inference.

Polymorphism is the mechanism which gives multiple forms/behaviour to same function.

Inheritance allows objects of a class to inherit properties of its parent class.

Hello World

```java
public class HelloWorld {
  public static void main(String[] args) {
      // Prints "Hello, World" to the terminal window.
      System.out.println("Hello, World");
  }
}
```

Code Editors

There are many high quality IDE(Integrated development environment) and code editors available. For example: IntelliJ IDEA, eclipse, Netbeans etc. In our lectures, we would primarily be using VS Code.

Compile and Debug

If you are using IDE, follow the documentation provided on the IDE's portal. We will outline the process of compile and debugging using command line.

Download Java JDK from oracle site. And install it be ticking yes to all the defaults. Make sure you install JDK 11 or above for the purpose of running programs in this book. Majority of the programs given in this book should run on earlier versions also, but they are tested to work on JDK version 11.

Copy the above Hello World program in a file named “HelloWorld.java”.

From the directory where “HelloWorld.java” is located, run the command, “javac HelloWorld.java”. This will create the file “HelloWorld.class”

Now, run the command “java HelloWorld” to finally execute the program. This will print the output “Hello, World” on the terminal.

2 Data Types and Operators

At the core, all programming languages exist to handle data and to support some operations on them. In this chapter, we will look at primitive data types of Java and common operations available for them.

What Is Data and Why We Need Data Types

Everything in the world of computer science is 0 and 1. Your instagram username, youtube videos, facebook passwords, all of them are stored somewhere in the form of 0s and 1s. But, human beings will have very hard time understanding information available in bits. So, there exist programming languages, that lets you write programs and informations in a language that is human readable, and internally saves them in whatever format the computer understands. So, programming languages gives us the convenience in interacting with computer. But again, whatever is convenient for one use case, may not be convenient in some other use case, so, there exist many different programming languages, that provide multiple choices to users.

Primitive Data Types

Java supports two types of data. Object Oriented and Non-Object Oriented. Object oriented data types are defined by classes, which we will discuss later. Java supports eight Non-Object Oriented or primitive data types. They are listed in the table below.

Type	Meaning
boolean	Represents true false values
byte	8 bit integer
char	Character
double	Double precision floating point
float	Single precision floating point
int	Integer
long	Long integer
short	Short integer

Integers

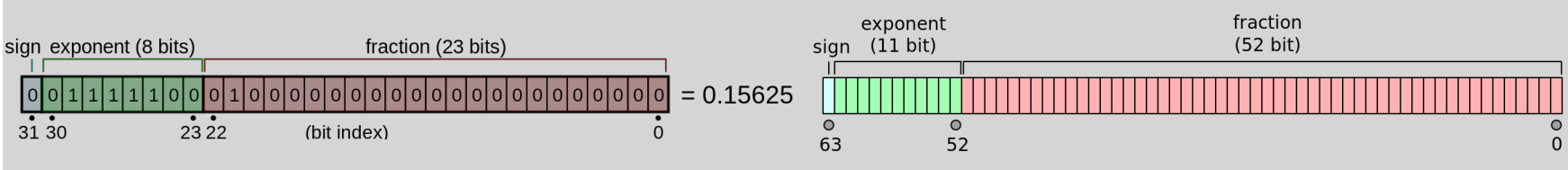
Java supports four integer types. Their range and width are mentioned in the table below.

Type	Width	Range
byte	8	-128 to 127
short	16	-32,768 to 32,767
int	32	-2,147,483,648 to 2,147,483, 647
long	64	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

Java doesn't support unsigned integers types.

Floating Point Types

Java supports two types of floating point numbers. They are float and double. Float is a single precision number whereas double is double precision number.



IEEE standard defines single precision and double precision in terms of bits in significand and exponent.

Double is more widely used, and many of the libraries support double unless specified otherwise.

Characters

In Java, char is an unsigned 16-bit type having range from 0 to 65,536. Unlike many other programming languages such as C which uses ASCII encoding, Java uses unicode for character encoding. ASCII, whose range is 0 to 127 is a subset of Unicode.

The reason why Java adopted Unicode is because Java was intended to be portable, therefore it needed an encoding that could support all the characters used in all human languages in the world.

Boolean

The boolean type represents true and false values. Java does it using the reserved keyword, true and false.

Literals

In general programming scenarios, we call it constants. Literals can be of any primitive type. For example, 151 is a literal. Generally literals are used to assign values to a variable. They

can also be used directly in situations where you don't want to use variable such as looping or checking equality etc.

Variable Declaration and Initialisation

There are two parts to using a variable. Declaration and initialisation. Declaration means declaring a variable with its type. Example:

```
int myAge;
char firstCharOfMyName;
```

Initialisation is when you assign a given variable some value/literal. Example:

```
myAge = 24;
firstCharOfMyName = 'R';
```

Declaration and initialisation can happen together also. Example:

```
int myGraduationYear = 2016;
```

A variable in Java can be initialised dynamically using other already initialised variable also. Example:

```
int myGraduationYear = 2016;
int graduationDuration = 4;
int graduationStart = myGraduationYear - graduationDuration;
```

Just, to remind you, it is possible to re-assign a value to a variable in Java after being initialised. Example:

```
int myGraduationYear = 2016;
myGraduationYear = 2017;
```

Variable Lifetime and Scope

Variables in Java is block scoped. Every time you start an opening curly brace, you are starting a scope, and this scope ends when you close the curly brace. Variables defined outside the scope is available inside the scope, but the variables defined inside the scope is not available outside. In case of functions, the variables defined in the scope of its starting and ending brace and the parameters is considered local to that function. Example:

```
class ScopeDemo {
  public static void main(String args[]) {
    int x = 10;
```

```
    if ( x == 10 ) {
      int y = 20;
      x = y * 2;
      System.out.println("x and y are: " + x + " " + y);
    }
    //System.out.println(“y is: ” + y); //This line will cause error
    System.out.println("x: " + x); // This line will work fine
  }
}
```

In the above example, you can see that variable x is available inside the scope of if block, but the variable y, which is defined inside the scope of if block is not available outside. Also, not that the lifetime of variable y is over once the closing curly brace of the if block is encountered.

Arithmetic Operators

Operator	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus
++	Increment
–	Decrement

The operators work the same way they work in other programming languages

Relational Operators

Operator	Meaning
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Logical Operators

Operator	Meaning
&	AND

Operator	Meaning
|	OR
^	XOR (exclusive OR)
||	Short circuit OR
&&	Short circuit AND
!	NOT

Short Circuiting Logical Operators

If you implement normal OR or AND operator, expression on both the ends of the operator is evaluated. Whereas, if you apply short circuit AND or short circuit OR, the expression on the left side is evaluated first, if they are not able to give the final output, then only the expression on the right is evaluated.

Assignment Operators

The assignment operator, single = sign, is used to assign literal to a variable. This is same as in other common programming languages. One thing worth noting is that:

```
int z, x, y;
x = y = z = 100;
```

Is same as

```
int x, y, z;
z = 100;
y = z;
x = y;
```

This is also worth noting that, shorthand assignment is available in Java and more often than not, they are more preferred than regular assignment, because they make the code easier to understand while debugging. Example;

```
x = x + 10;
```

This is same as:

```
x += 10;
```

Type Conversion and Cast

In many programming languages, it is common to assign a literal of one type to a variable of another type. In those cases, an automatic type conversion takes place. In Java, this functionality is available with strict condition. The variable to which you are assigning a new value, should be compatible with the value being assigned. Also, the cast should be of widening nature. Byte and int types are compatible because they are integer types. But, an int cannot be assigned to a variable of byte type, because this is not a widening conversion. Whereas, byte can be assigned to a variable of type int. Similar is the case for integer and float, and float and double.

In the cases, where automatic cast is not allowed and you need to perform a cast, you will need to explicitly cast the value to the type of variable. Example;

```
double d = 12.3;
int myInt = (int) d;
```

Operator Precedence

Following is the table for operator precedence:

Highest						
[]	()					
++(postfix)	- - (postfix)					
++(prefix)	- - (prefix)	~	!	+(unary)	- (unary)	Type-cast
*	/	%				
+	-					
>>	>>>	<<				
>	>=	<	<=			
“=="	!=					
&						
^						
|						
&&						
||						
?:						
- >						
“="						
Lowest						

3 Control Statements

Many people want to drive Ferrari, not all of them can afford it. They drive Ferrari, only if they can afford it. This is just one situation, but real life is full of situations which are driven by conditions. Since computer programs exist to solve real life problems, they have to understand the conditions and act accordingly. We will learn in this chapter how it does that.

Getting Input

Before we get into conditionals, let's learn how to get input from users. Until now, we have giving output to the users, but not getting any input from them. We will learn I/O in detail later in this book, right now we will learn just the basics which are required to learn other topics. To understand Java I/O in detail requires knowledge of Class and methods, so, we have decided to teach it after required chapters are complete. Let's see a program that takes a character as an input.

```
class InpurChar {
  public static void main(String args[])
    throws java.io.IOException {
    char ch;
    System.out.print("Press any key followed by an enter: ");
    ch = (char) System.in.read();
    System.out.println("Your key is: " + ch);
  }
}
```

We have already seen System.out, System.in is a complement to that. It is the input object attached to keyboard. The read() method waits for the user to press a key and then return the result. Since the character is returned as an integer, it has to be cast to char variable.

One point to be noted is that, by default console is line buffered, so, you will have to press enter after entering the character.

One another point to be noted is that, our main function is different this time. Whenever a method or a function has the potential to throw error, we have to explicitly declare that this methods throws error. Here, since System.in can throw exception, and it is used inside main, so, potentially main can also throw that exception and hence Java requires us to explicitly mention them. Now, we know how to take input, let's move on to learn conditional statements.

If Statement

The basic construct of if statement is

```
if (condition) {
  statements;
} else {
  statements;
}
```

If the number of statements is just one, then the braces are optional.

```
if (condition)
  statement;
else
  statement;
```

Well, even the else part is optional, In this case, the program would look like

```
if (condition) {
      statements;
}
```

Let's write a program utilising if statement

```
class GuessLetter {
  public static void main(String args[])
    throws java.io.IOException {
    char ch, answer = 'A';
    System.out.println("Guess a character");
    ch = (char) System.in.read();
    if (ch == answer) {
      System.out.println("You guessed it right.");
    }
  }
}
```

In the above program, if the user guessed the right character, he will be informed that he guessed the right character. But, if he has guessed the wrong character, then, the program will simple end. We can inform the user that his guess was not correct by exploiting else clause. Let's see it in the next program.

```
class GuessLetterBetter {
  public static void main(String args[])
    throws java.io.IOException {
```

```
    char ch, answer = 'A';
    System.out.println("Guess a character");
    ch = (char) System.in.read();
    if (ch == answer) {
      System.out.println("You guessed it right.");
    } else {
      System.out.println("Tough luck this time!");
    }
  }
}
```

Nested if-Else

I have a friend, who doesn't eat non-veg in the month of shravan. Even in other months, he doesn't eat non-veg on days other than Thursday and Tuesday. He feels bad when his wife cooks non-veg on the days when he doesn't eat. The poor man finds it so difficult to control. Seeing his condition, his wife decided to write a program that can tell her if the given day is suitable for her to cook chicken or not. Whenever she wants to cook non-veg, she will first run this program, if the program outputs "suitable" then only she will cook otherwise she will postpone her plans for some other day. Lets help her write this program.

```
import java.util.Scanner;
class SuitableDayForNonVeg {
  public static void main(String args[]) {
    Scanner sc = new Scanner(System.in);
    System.out.println("Enter month:  ");
    String month = sc.nextLine();
    System.out.println("Enter day:  ");
    String day = sc.nextLine();
    if (month != "shravan") {
      if ( day != "Tuesday" || day != "Thursday") {
        System.out.println("suitable");
      } else {
        System.out.println("Not Suitable");
      }
    } else {
      System.out.println("Not Suitable");
    }
  }
}
```

Here you can see that, we have nested if condition to handle real life situations which are more complicated than simple yes and no.

Also note that, this time we are using Java Util's Scanner class to take string input. Students are advised to follow this pattern until we discuss I/O in detail. Everything will be clear after that.

Else-if Loop

Nested If-else, is not always easy to read and understand. It takes a bit of effort to go inside one if/else block, and then go one step deeper. The common understanding among modern day top developers is that any software program which is not easy to understand is a bad program. So, we as an industry professional put in a lot of effort in writing easy to read and understand program. So, nested if-else conditional is used a bit less frequently.

Let's write a program for deciding the contract of batsman in Indian Cricket Team. If the batting average is more than 42, then the contract will be A-List, otherwise if the batting average is more than 35, then contract will be B-List, otherwise contract will be C-List.

```
class BatsmanContract {
  public static void main(String args[]) {
    float battingAverage = 34;
    if (battingAverage > 42) {
      System.out.println("A-List");
    } else if (battingAverage > 35) {
      System.out.println("B-List");
    } else {
      System.out.println("C-List");
    }
  }
}
```

As you can probably guess, this program is just a rearrangement of nested if-else conditional, but much more readable.

Also note that, instead of taking input from user, we have just hardcoded the battingAverage. We did this to make program shorter for demonstration purpose.

Switch

Now, we are at a position where we are fairly comfortable in writing programs for handling complicated situations. We also understand conditionals to a decent extent. If you have noticed, all the conditionals that we used till now, can be used for a variable that can hold a range of values. Example, the batsman average could be any number.

But there are some cases, where the variable can only hold a few predefined values. In those situations, switch cases are preferred. Else-if can be also used, but switch makes the program more easy to understand, and are also more performant(due to some internal optimisation). Let's see a case where switch case is the most ideal conditional.

Assume, you have to write a greeting to the user of a website, depending upon the access right of this user. If he is superadmin, you have to greet:, "Hello Sir, thanks for logging-in". If he is an admin, you have to greet, "Hey admin, good to see you.". If he is a normal user, greet: "Welcome to our website". If he is a guest user, greet: "Please create an account or sign-in to avail sites full functionality". Let's see the program.

```
class GreetUser {
  public static void main(String args[]) {
    String userType = "admin";
    switch(userType) {
      case "guest":
        System.out.println("Please create an account or sign-in to avail sites full
functionality");
        break;
      case "user":
        System.out.println("Welcome to our website");
        break;
      case "admin":
        System.out.println("Hey admin, good to see you.");
        break;
      case "superadmin":
        System.out.println("Hello Sir, thanks for logging-in");
        break;
      default:
        System.out.println("Please create an account or sign-in to avail sites full
functionality");
        break;
    }
  }
}
```

We are using break, because, once the control goes to a case, it executes every other subsequent cases. To avoid it from doing so, we have to use break statement. Did you notice that, case default and case guest have same message? We can use fall-through nature of switch to make program a but shorter.

```
class GreetUser2 {
  public static void main(String args[]) {
    String userType = "admin";
    switch(userType) {
```

```
      case "user":
        System.out.println("Welcome to our website");
        break;
      case "admin":
        System.out.println("Hey admin, good to see you.");
        break;
      case "superadmin":
        System.out.println("Hello Sir, thanks for logging-in");
        break;
      case "guest":
      default:
        System.out.println("Please create an account or sign-in to avail sites full
functionality");
        break;
    }
  }
}
```

Nested Switch

Just like nesting in if and if-else, switch also support nesting. But, this is rarely used.

For Loop

For loop is one of the most popular loop in the world. It iterates over an array of elements and perform some operation if specified. Let's write a program that calculates the average marks of a student.

```
class AverageMarks {
  public static void main(String args[]){
    int[] marksInDiffSubjects = {90, 46, 50, 67, 100};
    int totalMarks = 0;
    int numOfSubject = marksInDiffSubjects.length;
    float average = 0;
    for(int i = 0; i < numOfSubject; i++) {
      totalMarks += marksInDiffSubjects[i];
    }
    average = (float) totalMarks / numOfSubject;
    System.out.println("Your average marks is: " + average);
  }
}
```

Note that we had to first cast totalMarks to float, otherwise the output would be wrong. It must be noted that arrays in Java are 0 indexed. So, counting starts from 0. Also, you can see

that inside the parenthesis of for loop, there are three statements separated by semi-colon. First one is initialisation, second one is conditional, and third one is update section.

```
for(initialisation; conditional; update) {
  //statements;
}
```

All the three parts of for-loop, i.e. initialisation, conditional, and update are optional. But, the semi-colon between them are mandatory.

The program sequence is as follows

intialisaton

conditional

update

conditional

update

conditional

.

.

.

This goes on until the program encounters break statement inside the for loop, or conditionals statement returns false. So, if the conditional is not there, of if it never returns false, and the program doesn't encounter break statement inside its body, it will become an infinite loop. Example:

```
for(; ;){
}
```

While Loop

If you don't want to do initialisation inside for loop, you can use while loop. Many people in IT industry, including me, prefer while loop over for loop because it makes program easier to understand at a single glance. But, there is no consensus in the industry as to which one is better. Different people have different opinion.

Let's re-write the average marks program that we wrote above using while loop.

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

```
    int[] marksInDiffSubjects = { 90,46, 50, 67, 100 };
    int totalMarks = 0;
    int numOfSubject = marksInDiffSubjects.length;
    float average = 0;
    int index = 0;
    while(index < numOfSubject) {
      totalMarks += marksInDiffSubjects[index];
      index += 1;
    }
    average = (float) totalMarks / numOfSubject;
    System.out.println("Your average marks is: " + average);
  }
}
```

Do-While Loop

In do-while loop, the loop body will be executed at least once, even if the conditional fails the first time. The structure of do-while is:

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

do-while is very little used.

Break

At any point inside the loop body, if the program encounters a break, the loop stops executing. Example:

```
class BreakExample {
  public static void  main(String args[]) {
    for( int i = 0; i < 20; i++) {
      System.out.println(i);
      if ( i == 10) {
        break;
      }
    }
  }
}
```

Continue

If the encounters continue, it starts the next iteration of the loop. Example:

```
class BreakExample {
  public static void  main(String args[]) {
    for( int i = 0; i < 20; i++) {
      if ( i % 2 == 1) {
        continue;
      }
      System.out.println(i);
    }
  }
}
```

Break as a Form of Goto.

Java doesn't support goto statement, as it produces code that are hard to understand. But, we if we use break with a label, then break acts as goto. Note that, this break statement with label works only inside the label to which it can break. Example:

```
class BreakLabel {
  public static void main(String args[]) {
    for(int i = 0; i < 2; i ++) {
      one: {
        for (int j = 0; i < 20; j ++) {
          if (j > 5) {
            break one;
          }
          System.out.println( "i and j are: " + i + " "+ j);
        }
      }
    }
  }
}
```

Nested Loops

Java supports nested loops and there is no limit as to which loop can be used inside which loop. But, the user should be careful while nesting loop. Anything less than two level deep, is not appreciable by industry experts.

4 OOP in Java

OOP is a programming paradigm that is centred around data. Until a few years ago, OOP was thought to be the best programming paradigm. But, now a days, functional paradigm is gaining good sentiments. Nonetheless, OOP will remain the most popular and widely used paradigm for a very long time

Class

Grouping together similar stuffs is called classifying. And the group itself is called class.

Example: When you say that everything that ever moved and breathed is animal, you have basically created a class called animal, and you have put everything that moves and breathes into that class. You can do further grouping inside animal class, ie herbivores, carnivores etc.

Class in Java is no different. In Java, every individual member of the class is called an object. And an object is said to be the instance of the class. Let's create a class called animal and an object of the this class in Java

```
class Animal {
  // Animal properties to be mentioned here
}
Animal myAnimal = new Animal()
```

Here, Animal is the class that we talked about, and myAnimal is an individual member of this class.

One thing to note is that, class is also a data type just like integer and float. The only difference being, class is a user defined data type. Notice the similarity of defining variable.

```
int x;
Animal y;
```

Well, before we discuss any further concepts, let's look at the general form of the class.

```
class classname {
  //declare instance variable
  type var1;
  type var2;
  .
  .
  type varN;

  //declare methods
  type method1(parameters) {
```

```
    //body
  }
  type method2(parameters) {
    //body
  }
  .
  .
  type methodN(parameters) {
    //body
  }
}
```

Although, the language doesn't force it on users, but a single class should represent one single entity only.

Notice, the classes that we have been using till now, they used to contain main function, but, the general structure doesn't contain main. Well, main is required only in those cases where the given program is starting point of the application. Example: Java applets doesn't require a main().

Defining Classes

Let's define a class for vehicle. This class will contain three informations about the vehicle, fuel capacity, number of passengers it can carry, and the mileage. We will make this data-only class, meaning this will not contain any methods.

```
class Vehicle {
  int passengers;
  int kpl;
  int fuelCapacity;
}
```

Class definition creates a new data type. So, we will have a new data type, "Vehicle", and we will use this name to declare objects of this class. *Remember, class declaration is only a type definition, and it doesn't create an actual object*. To create an actual object, you have to create an instance of the class. Now the question arises, what kind of variable will store the instance of the class. The answer is, the class is itself a data type, and we will require variable of this type only.

```
Vehicle motorCycle = new Vehicle();
```

This statement will create an object, a real entity called motorCycle which is an instance of Vehicle. Every time we create an instance of a class, we create an object that contains all the instance variables of that class. And to access those variable, we have to use the dot (.) operator.

```
//object.member
motorCycle.kpl = 50;
```

Note that, each object has its own copy of the instance variable. And that the value of instance variable can differ from one object to other.

Discussion on Objects

In the preceding sections, we create object with following syntax:

```
Vehicle motorCycle = new Vehicle();
```

This can be broken down into two parts:

```
Vehicle motorCycle;
motorCycle = new Vehicle();
```

In the first line, we create a variable of type Vehicle, which can hold the reference of an object that is an instance of the class Vehicle.

While the second class, creates an instance of the class Vehicle using new operator and assigns its' reference to the variable, motorCycle, declared in the first line.

Reference Variable and Assignment

Object reference variables are different from variables of primitive types. The variables of primitive types store the value of the data which is assigned to it, whereas object reference variables stores the reference of the data which is assigned to it.

The following examples will make it clear.

```
Vehicle myCar = new Vehicle();
Vehicle myNewCar = myCar;
myNewCar.kpl = 25;
System.out.println(myCar.kpl);
System.out.println(myNewCar.kpl);
```

The output for both the printing statements will be 25. It may sound trivial that myCar and myNewCar refers to different objects, but they are not.

Since myCar stores the reference to an object, the same reference is assigned to another variable named myNewCar. Now, two reference exist for the same object. Note that the references are two, but the object is only one. Since both the variables refers to the same

object, changes made to the original object by any reference will reflect in objects referred by both the references.

Methods

Just like instance variables of a class, a class can also have methods. And the method is referenced by using a (.) dot operator after the instance name. Let's add a method to Vehicle class that we have, this method will return the kilometre that this vehicle will go after the tank has been filled to full capacity.

```
class Vehicle {
  int passengers;
  int kpl;
  int fuelCapacity;
  int totalKm() {
    return fuelCapacity * kpl;
  }
}
```

Let's see an example of how we will use this class method;

```
Vehicle myBike = new Vehicle();
myBike.kpl = 50;
myBike.fuelCapacity = 24;
System.out.println("Bike will travel: " + myBike.totalKm() + " when tank is filled.");
```

Method Paramater

Let's first get clear between value and parameter. The value that we pass to a method is called argument. And the variable inside a method that receives the argument is called parameter. Let's see an example:

```
int square(int num) {
  return num * num;
}
System.out.println("Square of given num is: " + square(6));
```

Here, 6 is argument, and num is parameter.

Well, we can also have parameters to a class method. Let's consider a situation where, due to bad road, mileage of the vehicle drops. We are given a number depending upon the quality of road, by which the mileage of the vehicle drops. In this case, the new method for calculating totalKm would be.

```
class Vehicle {
```

```
  int passengers;
  int kpl;
  int fuelCapacity;
  int totalKm( int mileageDrop) {
    return fuelCapacity * (kpl - mileageDrop);
  }
}
```

Constructors

If you look at the way we have been using class, you will see that we first create an instance and then, manually assign the default variables such as fuelCapacity etc. Java and most other programming language provide an automated way of doing that.

Constructors are just like other methods, with a difference that, it doesn't specify any return type, and its name is identical to that of the class. Let's see an example:

```
class Vehicle {
  int passengers;
  int kpl;
  int fuelCapacity;
  Vehicle() {
    kpl = 25;
    fuelCapacity = 20;
    passengers = 20;
  }
  int totalKm( int mileageDrop) {
    return fuelCapacity * (kpl - mileageDrop);
  }
}
```

All classes in Java have constructors, whether you define it or not. If you don't define the constructors, Java will call the default constructor of the class, which will initialise instance variable to its default value, ie zero, null and false.

Parametrised Constructors

The constructor that we wrote in last section is rarely useful in real life. In real life, every time you create an instance of a class, you would like to set the defaults all by yourself. Therefore, Java supports parametrised constructors also. Let's see.

```
class Vehicle {
  int passengers;
  int kpl;
  int fuelCapacity;
  Vehicle(kplExt, fuelCapacityExt, passengersExt) {
```

```
    kpl = kplExt;
    fuelCapacity = fuelCapacityExt;
    passengers = passengersExt;
  }
  int totalKm( int mileageDrop) {
    return fuelCapacity * (kpl - mileageDrop);
  }
}
```

If you have a parametrised constructor, then you have to pass the value of constructor parameters as arguments while instantiating the class. Example:

```
Vehicle myBike = new Vehicle(20, 30, 25);
```

New Operator

We have been using new operator for quite some time, let's understand it before moving any further.

new operator can be used to create a new instance of a class. It the class provides a constructor, new will use the given constructor, otherwise it will use the default constructor of the class. When new creates the the instance, it returns its reference.

One thing to note is that, new may fail in creating an instance of the class. In that case, it will raise an exception rather than returning a reference. In real world programming, this needs to be taken care of.

Garbage Collection

Every time you create an instance of a class, you utilise some memory. When you create a lot of objects, you might run out of memory and you will start getting error. To avoid those scenarios, Java has garbage collection mechanism, that runs from time to time or when required to clean up the objects that is no longer required. In some languages, this process is manual, i.e., developer has to identify the variables no longer used and delete them all by himself/herself. But, Java provides this functionality automatically.

Finalise()

Garbage collector is generally called only when required. It is considered required when the two conditions are met: there are objects to be recycled, and there is a need to recycle objects.

Well, before the garbage collector could destroy the object, if you have defined finalise method, it will be called. Note that, it is not called simply when the objects are no longer required or if it has gone out of scope, it is called when Java is about to recycle/destroy the object.

finalise() ensures that the method destruction happens cleanly. If you are opening some files inside the class, then you should check on file descriptor whether the file is open, and off it is open then it should be closed before the object is being recycled.

```
protected void finalise() {
  // finalise code here
}
```

5 Arrays

With the chapter on arrays, we have come back to the topic of data types in Java. In this chapter, we will learn ways to declare arrays and some methods available on Arrays.

Introduction

Arrays can be thought of as a list of variables with common name, which are of same type. Arrays in Java have one difference with arrays in many other languages: Arrays in Java is implemented as objects. This offers advantages, the most important of them is that unused arrays can be garbage collected. This also has downside, implementing arrays as objects is not so good at memory.

One Dimensional Arrays

One dimensional arrays are the most popular type of array. This stores data of various entities of same type, and these data can be assessed by same name. For example, we can store batting averages of entire Indian team on one array.

Declaration syntax is as follows:

```
type array-name[] = new type[size];
```

Example:

```
int battingAverages[] = new int[11];
```

This part can also be broken down into two parts:

```
int battingAverages[];
battingAverages = int[11];
```

The first part, declares a variable that will store the reference of the array of type int. The second part will actually create an array of type int and size 11, and it will assign the reference of this array to battingAverages variable.

An individual element inside the array is accessed using an array. The array in Java is 0 indexed, means, the index of first element inside the array is 0.

Let's look at the following program to understand index of an array.

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

```
    int sample[] = new int[5];
    int index;
    for (index = 0; index < 5; index = index + 1) {
      sample[index] = index;
    }
    for (index = 0; index < 5; index = index + 1) {
      System.out.println("sample[" + index + "] = " + sample[index]);
    }
  }
}
```

The output will be:

sample[0] = 0

sample[1] = 1

sample[2] = 2

sample[3] = 3

sample[4] = 4

Just to get a bit more familiar with the concepts of array, let's try to find out the min and max values stored in an int array of size 10.

```
class ArrayExample {
  public static void main(String args[]) {
    int myArray[] = { 3, 2, 67, 32, -5, 89, 35, 89, 3, 23 };
    int min = myArray[0];
    int max = myArray[0];
    for (int index = 0; index < 10; index += 1) {
      if (myArray[index] > max) {
        max = myArray[index];
      }
      if (myArray[index] < min) {
        min = myArray[index];
      }
    }
    System.out.println("Min and max are: " + min + ", " + max);
  }
}
```

Notice that we have initialised the array using a different construct than we have been doing till now.

Multidimensional Arrays

Multidimensional arrays are not so popular as single dimensional arrays, but they are still very useful.

Let's initialise a 3x4 array.

```
int myArray[][] = new int[3][4];
```

The resulting array will be like this.

0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	2,1	2,2	2,3

Multidimensional arrays are used just like normal arrays, only difference is that instead of specifying one index, we have to specify multiple indexes for accessing or assigning the data. Example:

```
class ArrayExampleTwo {
  public static void main(String args[]) {
    int myArray[][] = new int[2][4];
    for (int i = 0; i < 2; i += 1) {
      for(int j = 0 ; j < 4; j += 1) {
        myArray[i][j] = i + j;
      }
    }
    for (int i = 0; i < 2; i += 1) {
      for(int j = 0 ; j < 4; j += 1) {
        System.out.println( "Data at " + i + ", " + j + " is: " + myArray[i][j] );
      }
    }
  }
}
```

Irregular Arrays

While declaring multi-dimensional arrays, we need to specify only the size of first dimension, so, it is possible to have variable length arrays in 2nd dimension onwards. This will be more clear, if we look at the example.

```
int irregularArrays[][] = new int[2][];
irregularArrays[0] = new int[5];
irregularArrays[1] = new int[2];
```

Arrays With More Than Two Dimension

The way, we have declared and used two dimensional arrays, we can also have three or more dimensional arrays. Let's look at the general syntax for these type of arrays

type name[][]….[] = new int[size1][size2]….[sizeN];

Example:

```
int mySpecialArray[][][][] = new int[2][2][3][7];
```

This will create a four dimensional array of size 2 x 2 x 3 x 7.

Something that I would like to tell you from my experience, don't go deeper than 2 levels. And consider level deeper than three as forbidden.

Alternate Array Declaration Syntaxes

Till now, we have been declaring arrays by writing square boxes after variable name, but we can declare arrays by writing square boxes around array type also. Example:

```
int[][] myFirstArray;
```

Well, the advantage of this approach is that you can declare multiple arrays in one line. Example:

```
int[][] myFirstArray, mySecondArray;
```

Note that, declaring multiple variables in one line is not considered a good practise my many professionals.

Array References

As we have already discussed, arrays in Java is implemented as objects. And since, variables that are assigned objects contain the reference to the object and not the actual object itself, the array variables also contain the reference of the arrays. Let's see it through an example.

```
class ArrayRef {
  public static void main(String args[]){
    int[] firstArray = {2, 3, 4};
    int[] secondArray = firstArray;
    for(int i = 0; i < 3; i++) {
      secondArray[i] = i;
    }

    for(int i = 0; i < 3; i++) {
      System.out.println("Now data in index " + i + " of firstArray is: " + firstArray[i]);
    }
```

```
  }
}

The output is:
Now data in index 0 of firstArray is: 0
Now data in index 1 of firstArray is: 1
Now data in index 2 of firstArray is: 2
```

We can see that changes made to second array is reflected in first array, because both of them contain the reference to the same array.

Array Methods

Since arrays are internally objects, they contain some methods and members. The most important being length member. All the arrays contain a member called length that gives you the length of the array. Let's see it through an example:

```
class ArrayLength {
  public static void main(String args[]) {
    int[] demoArray = {2, 3, 4, 5, 6};
    for(int i = 0; i < demoArray.length; i++) {
      System.out.println("Value at " + i + " is: " + demoArray[i]);
    }
  }
}
```

Foreach Loop

Sometimes, all we want is to iterate over an entire array. But, we have to maintain logic for counter and array length. There exist a simple way to iterate over an entire array in Java. This loop is called foreach loop. Let's see an example of this loop.

```
class ArrayForeach {
  public static void main(String args[]) {
    int[] demoArray = { 2, 3, 4, 5 };
    for (int val : demoArray) {
      System.out.println("Value is: " + val);
    }
  }
}
```

6 Strings and Operators

String is a very widely used data type of Java. In many other programming languages, string is an array of characters, but in Java, string is an object. We have been using strings from the very beginning of this book, without realising. For example, in the following statement:

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

the string, "Hello World" is automatically made into a string object by Java. In this chapter, we will discuss some important things about string.

How To Construct Strings

The easiest way to construct string is by inclosing a few text in double quotes. Well, we can also construct string using the new operator. Let's look at them.

```
String str = "I am learning string";
String str2 = new String("I am also learning String");
```

Strings can also be constructed by passing another string to the String constructor. Example:

```
String str = "I am learning string";
String str2 = new String(str);
```

String Operations

The String class contains some very useful methods. Let's look at them in the following table.

Method	Description
boolean equals(String)	Returns true the invoking string is same as the argument being passed character-wise.
int length()	Returns the length of a string.
char charAt(int)	Returns the character at the index provided as argument
int compareTo(String)	Returns less than zero if the invoking string is less than the argument being passes. Returns zero, if both are equal. If the invoking string is greater than the argument then it returns a positive value.
int indexOf(String)	Returns the index of the substring provided as argument in the invoking string. If found, returns the first index of the substring otherwise returns -1.
int lastIndexOf(String)	Same as indexOf, but here if substring occurs multiple times in the invoking string, it will return the last index.

Let's see an example of one such method and the use of other methods will become self evident.

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

```
    String str = "I am good at Java!";
    String str2 = "I am good at Java!";
    System.out.println("Are both strings equal? " + str.equals(str2));
  }
}
```

Note that you cannot do '==', because Strings are objects, and String variables contain object references and not actual string. So, '==' will check if both the string references point to the same object or not. If they point to different objects, then it will return false, even if the value of both the strings are same.

Array of Strings

Just like an array of int, we can have an array of strings also.

```
class test{
  public static void main(String args[]){
    String strArr[] = {"Apna", "Vidyalaya"};
    for (String str : strArr) {
      System.out.println(str);
    }
  }
}
```

Immutability of Strings

If you talk about Java related interviews, this topic seems to be very popular. As the topic suggests, strings are immutable, which means it cannot be changed. If you define a string like this

```
String str = "Apna Vidyalaya";
```

Two things are happening in the above statement. First a String object is being created with the value, "Apna Vidyalaya". And then, the reference of this object is assigned to str, which is a variable of type String.

When I say that String is immutable, I mean that the string object that was being created to contain, "Apna Vidyalaya", will never change in its lifetime. But, the value of the variable str can change. Let's see one more example.

```
String myStr = "Apna Vidyalaya";
myStr = "Best Training Institute";
```

In this example, first statement will create an object and its reference will be assigned to str. In the second statement, a new object will be created and str will now contain reference to the new object. Since the first object is not being referred anymore, it will be garbage collected later on.

But, if you want a string that can change/mutate. You can use StringBuilder or StringBuffer classes.

Strings in Switch Statements

Unlike many other languages, switch statement in Java supports string also.

```
class test{
  public static void main(String args[]){
    String str = "vidyalaya";
    switch (str) {
      case "vidyalaya":
        System.out.println("Beautiful Hindi word!");
        break;
      case "school":
        System.out.println("English word");
        break;
    }
  }
}
```

Accepting Command Line Arguments

Now that we are aware of string arrays, the parameters of the main function that we have been writing since the beginning of this book, might have become clear to you. While running a program we can pass some arguments and the main function receives them as string arrays. These are called command line arguments.

Bitwise Operations

Ternary Operators

7 Revisiting Class and Methods

Access to Class Members

Passing Objects to Methods

Returning Objects

Method Overloading

Constructor Overloading

Recursion

Static Blocks

Nested Class

Variable Number of Arguments

8 Inheritance

Basics

Member Access Inheritance

Constructors and Inheritance

Super

Multilevel Hierarchy

When Are Constructors Executed

Superclass References

Subclass Objects

Method Overriding and Polymorphism

Advantage of Method Override

Abstract Class

Final

The Object Class

9 Packages

Defining Package

10 Modules

Introduction

11 Exception Handling

Exception Hierarchy

12 I/0

Streams

Multithreading

Fundamentals

13 Enumerations, Auto Boxing, Static Import and Annotations

Enumerations

14 Generics and Collections

Fundamentals

15 Lambda Expressions and Method References

Introduction to Lambda

16 JDBC

17 Networking

18 Reflection

19 Internalisation

20 Jshell