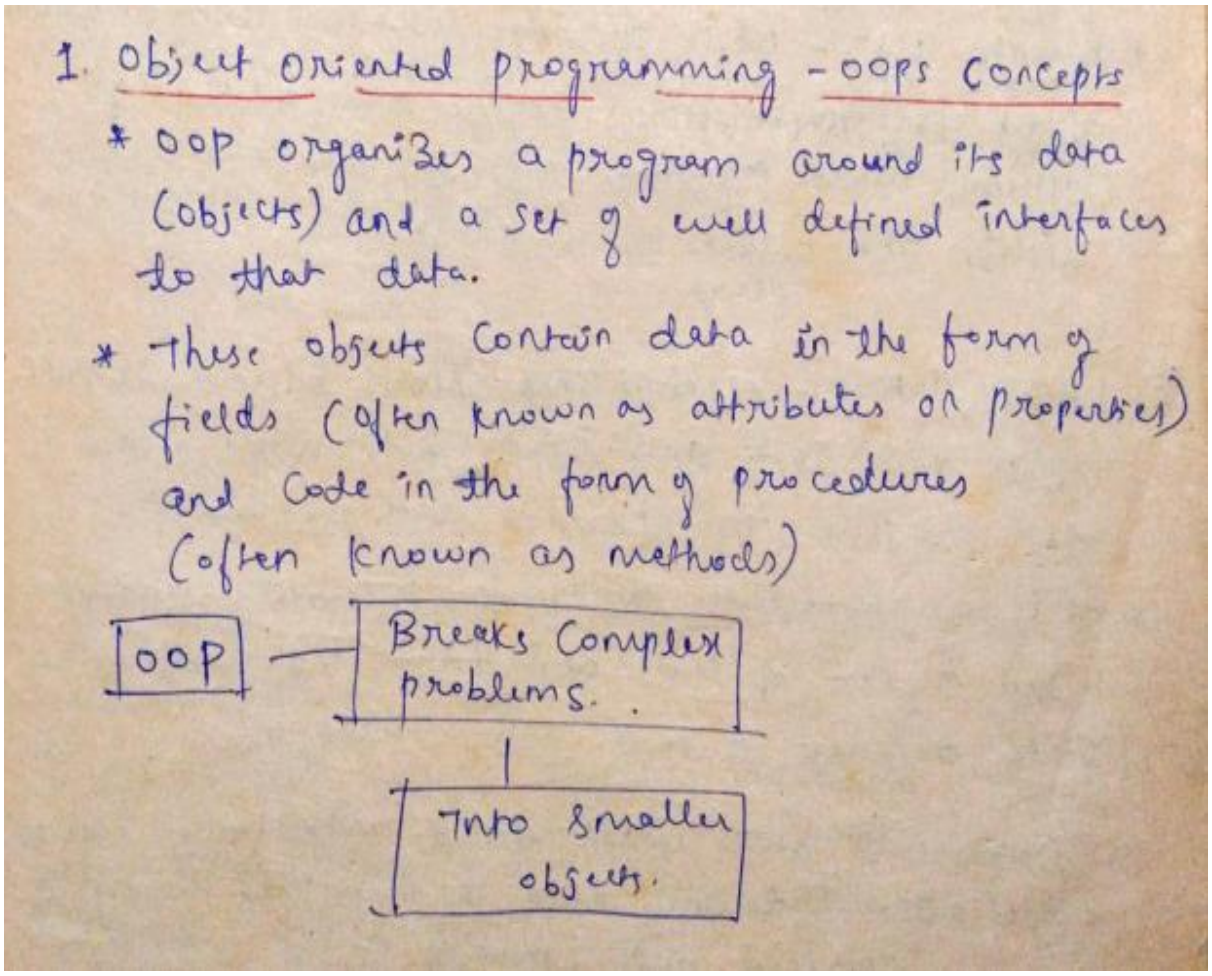
MODULE 1 NOTES

SYLLABUS- MODULE 1

Module 1- Introduction
Fundamentals: An overview of Java: Object-Oriented Programming concepts, The Java virtual machine, Features of Java, Structure of a Java program, Data Types and Variables, Type conversion and casting, Arrays. Classes: Fundamentals, Declaring Objects, Assigning Object Reference Variables, Methods, Constructors, this Keyword, Garbage Collection. Methods and Classes: Overloading Methods, Using Objects as Parameters, Argument Passing, Returning Objects, Access Control, static, final, Command-Line Arguments



OOPs Concepts – (4)

ⓐ Abstraction
ⓑ Encapsulation.
ⓒ Inheritance
ⓓ Polymorphism

ⓐ Abstraction - It is a key concept in oop lang to handle complexity by hiding unnessary details from the user. and showing only ess -tial Information to the user.

* Abstraction can be achieved in 2 ways.
  ⓐ Abstract Class
  ⓑ Interfaces

Example - Car - while driving, we are not con about its how transmission, Engine and braking systems work together. Instead we are free utilize the object as a whole. (Car)

ⓑ Encapsulation - Mechanisms, that binds toge code and data and keeps both safe from the outside interference and misuse.

* It is a protective wrapper, prevents unauth -rized access of code and data by another Code defined outside this wrapper.
* There will be a well defined interface wi which the code and data inside the wrapp Can be accessed and utilized.

* Encapsulation is achieved by declaring the ~~inste~~ instance variables of a class as private, so that it can be accessed only within the class. (3)
* Classes are the basis of Encapsulation
* Class defines structure (data) and behaviour (Code) that will be shared by set of objects
* objects – instance of class.

Class → are → logical construct

an object → has → physical reality

* Purpose of class — is to Encapsulate Complexity
* Example (Generic)

```
Class
{
  Member Variables (also called Instance Variables)
  Member Methods
}
```

Member Variables + Member methods = members of class.

Data is defined in member variables.
Code that operates on this data is defined in member functions.

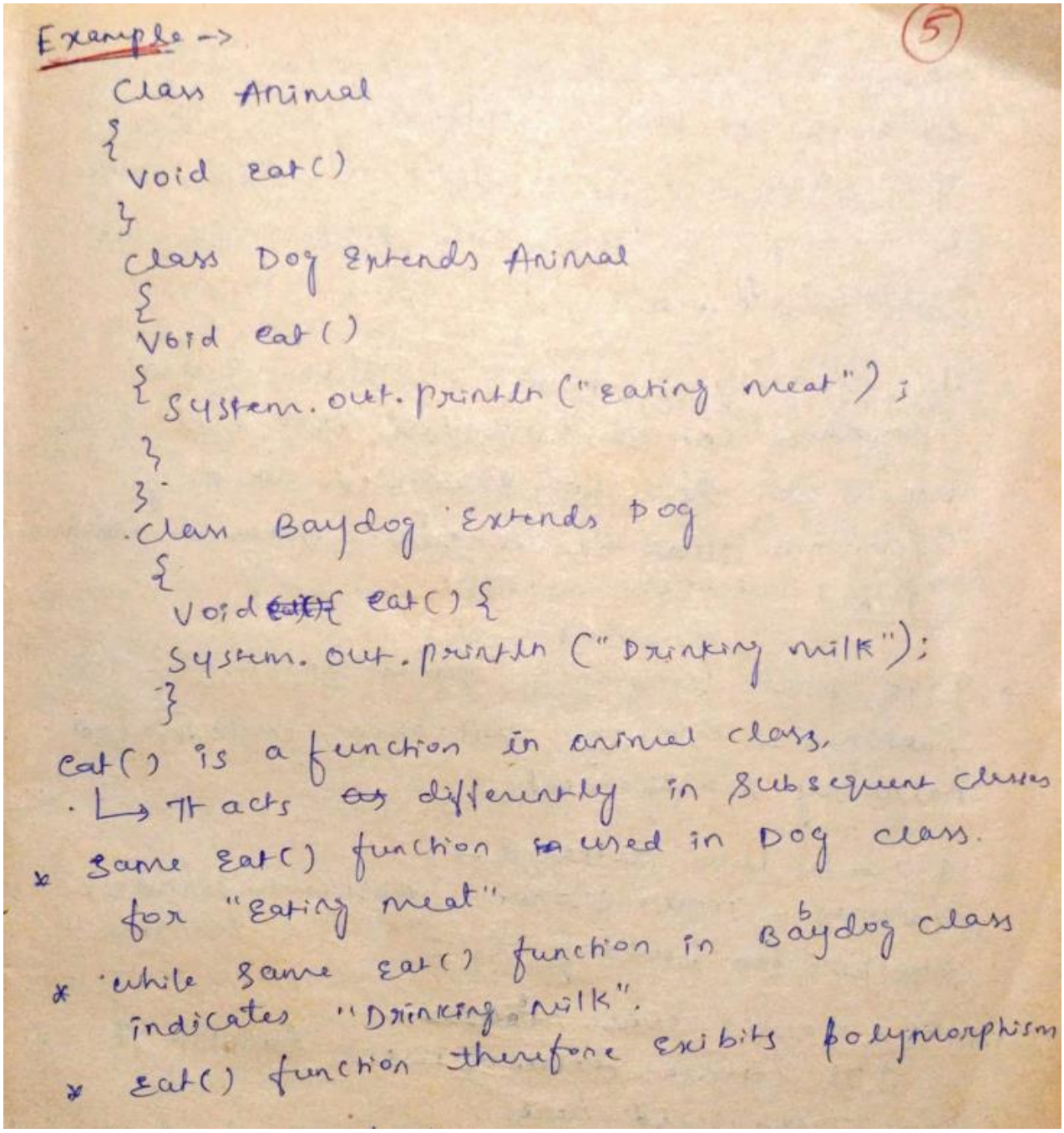
Inheritance - Mechanism in Java by which one class aquire the features (variables and methods) of another class. (4)

* It is Creating new classes based on Existing classes.
* New class are called → Child class
* Existing classes are called → parent class.
* A child class can reuse the methods and fields (variables and functions) of parent Cla

Example - Syntax

```
Class A
{
// Member Variable A
// Member function A
}
Class B extends A
{
// Can utilize (reuse) member function A
// "  "  "  Variable A
}// Member variables & functions of class B.
```

Polymorphism :- An oops feature that allows same entity (variable or function) to behave differently in different scenarios.



FEATURES OF JAVA

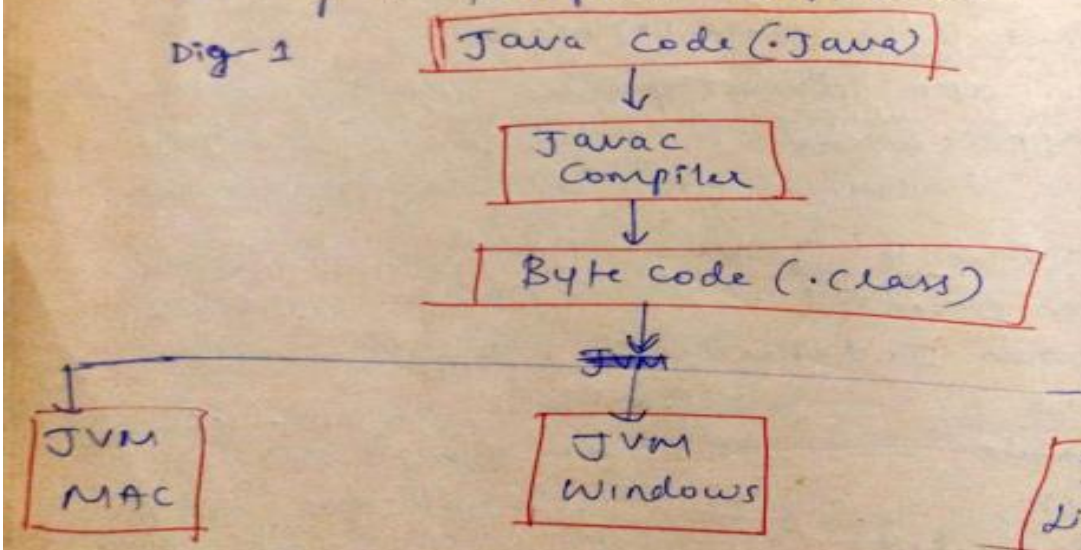
① **object oriented** —

* Java is object oriented prg. Lang. with features like classes, objects, inheritance, polymorphism, encapsulation promoting modular & reusable code.

② **platform Independence** —

* Java prog. can run on any platform that has a JVM. installed. JVM interprets the Compiled Java byte code and translates it into Machine Code for specific platform, making it platform Independent.

Dig 1



③ **security** → It has builtin security features such as class loader and byte code verifier

Class loader – dynamically loads Java Classes in JVM during time, locates and loads only necessary class files, needed to Execute Java Prg

Bytecode verifier – part of JVM, Ensures the safety and security of Java prg. (9)

* It checks the validity and integrity of bytecode before it is Executed.

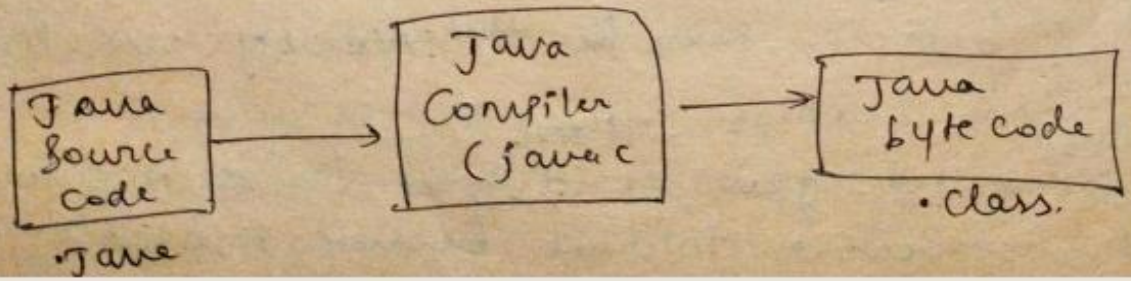
(4) Portability – Java is a portable programming language, it can be compiled once and run on different platforms, (write about JVM).

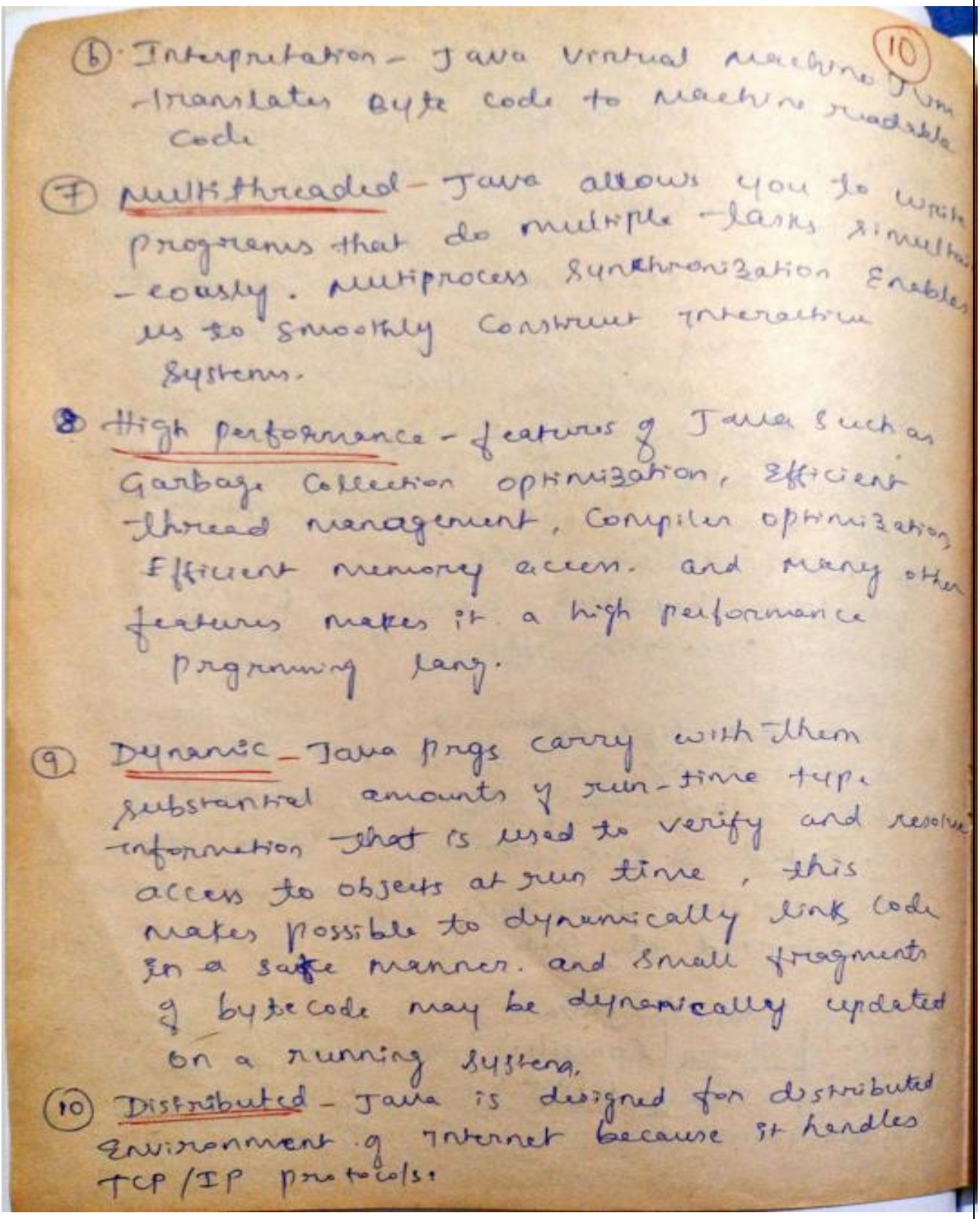
(5) Robust – Java prgs must be Executed in variety of platforms, thus robustness is Given high priority in the design of Java

* Java's Strict compile-time checking catches many Errors before run time, resulting in more reliable code
* Exception handling allows recovery from Errors preventing the Entire prog from crashing
* Java's Extensive Standard library provides a wide range of robust API for tasks such as networking, I/O, cryptography.

(6) Interpreted – 2 Step process → Compilation → Interpretation

(a) Compilation – Java code is first
→ Compiled into bytecode




(10)

(6) Interpretation - Java Virtual Machine (JVM) translates Byte code to Machine readable code

(7) Multithreaded - Java allows you to write programs that do multiple tasks simultaneously. Multiprocess Synchronization Enables us to smoothly Construct Interactive Systems.

(8) High performance - features of Java such as Garbage Collection optimization, Efficient thread management, Compiler optimizations Efficient memory access. and many other features makes it a high performance prgrnning lang.

(9) Dynamic - Java prgs carry with them substantial amounts of run-time type information that is used to verify and resolve access to objects at run time, this makes possible to dynamically link code in a safe manner. and small fragments of bytecode may be dynamically updated on a running system.

(10) Distributed - Java is designed for distributed Environment of Internet because it handles TCP/IP protocols.

11. Java is a Strongly Typed Language A strongly-typed programming language is one in which each type of data (such as integer, character, hexadecimal, packed decimal, and so forth) is predefined as part of the programming language and all

constants or variables defined for a given program must be described with one of the data types. Certain operations may be allowable only with certain data types.

In other words, every variable has a type, every expression has a type, and every type is strictly defined. And, all assignments, whether explicit or via parameter passing in method calls, are checked for type compatibility. There are no automatic coercions or conversions of conflicting types as in some languages. The Java compiler checks all expressions and parameters to ensure that the types are compatible. Any type mismatches are errors that must be corrected before the compiler will finish compiling the class. These features of Java make it a strongly typed language.

POP vs. OOP

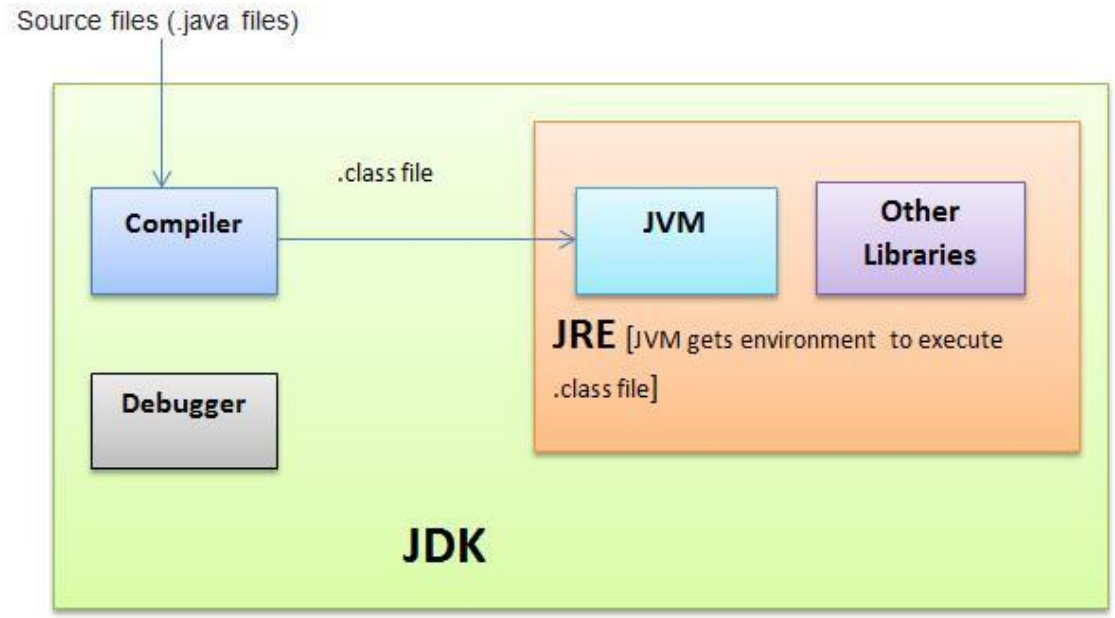
	Procedure-Oriented Programming	**Object Oriented Programming**
Divided Into	In POP, program is divided into small parts called functions.	In OOP, program is divided into parts called **objects**.
Importance	In POP, Importance is not given to **data** but to functions as well as **sequence** of actions to be done.	In OOP, Importance is given to the data rather than procedures or functions because it works as a **real world**.
Approach	POP follows **Top Down approach**.	OOP follows **Bottom Up approach**.
Access Specifiers	POP does not have any access specifier.	OOP has access specifiers named Public, Private, Protected,
Data Moving	In POP, Data can move freely from function to function in the system.	In OOP, objects can move and communicate with each other through member functions.

JAVA VIRTUAL MACHINE(JVM)

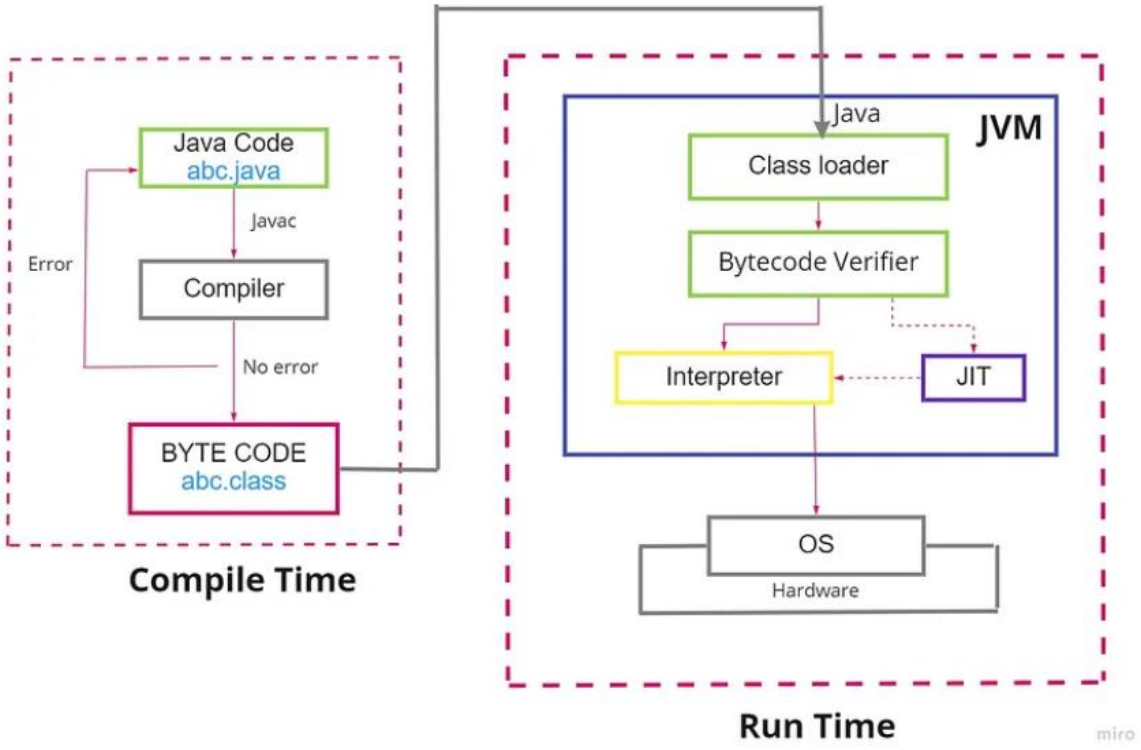
- The JVM is the Java run-time system and is the main component of making the java a platform independent language.
- For building and running a java application we need JDK(Java Development Kit) which comes bundled with Java runtime environment(JRE) and JVM.
- With the help of JDK the user compiles and runs his java program. As the compilation of java program starts the Java Bytecode is created i.e. a .class file is created by JRE.
- Bytecode is a highly optimized set of instructions designed to be executed by JVM. Now the JVM comes into play, which is made to read and execute this bytecode.
- The JVM is linked with operating system and runs the bytecode to execute the code depending upon operating system. Therefore, a user can take this class file(Bytecode

file) formed to any operating system which is having a JVM installed and can run his program easily without even touching the syntax of a program and without actually having the source code.

- The .class file which consists of bytecode is not user-understandable and can be interpreted by JVM only to build it into the machine code.
- Remember, although the details of the JVM will differ from platform to platform, all understand the same Java bytecode.
- Thus, the execution of bytecode by the JVM is the easiest way to create truly portable programs and this makes Java PLATFORM INDEPENDENT



These features make the java as a portable (platform-independent) language



A First Simple Program:

Here, we will discuss the working of a Java program by taking an example –

Program 1.1 Illustration of First Java Program

```
class Prg1
{
      public static void main(String args[ ])
      {
            System.out.println("Hello World!!!");
      }

}
```

Save this program as Prg1.java. A java program source code is a text file containing one or more class definitions is called as compilation unit and the extension of this file name should be .java.

To compile above program, use the following statement in the command prompt –

javac Prg1.java (Note: You have to store the file Prg1.java in the same location as that of javac compiler or you should set the Environment PATH variable suitably.)

Now, the javac compiler creates a file Prg1.class containing bytecode version of the program, which can be understandable by JVM. To run the program, we have to use Java application launcher called java. That is, use the command – java Prg1

The output of the program will now be displayed as – Hello World!!!

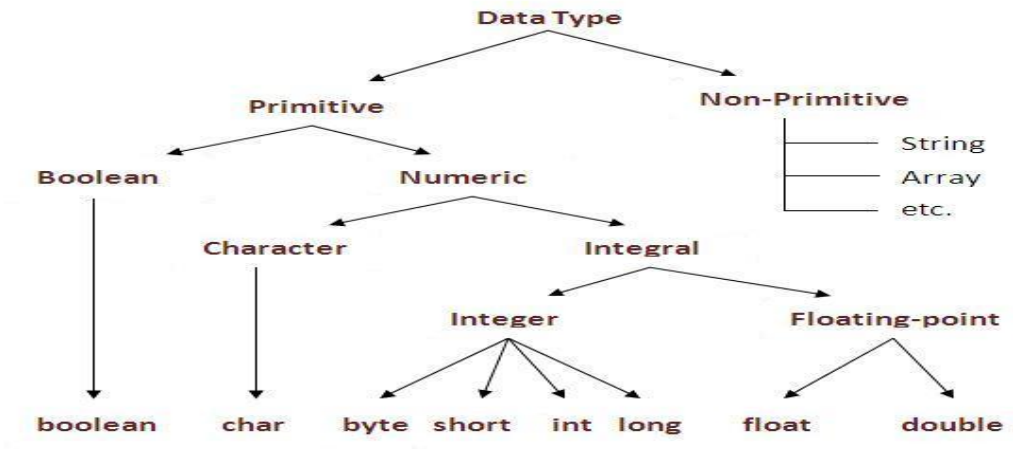
Note: When java source code is compiled, each class in that file will be put into separate output file having the same name as of the respective class and with the extension of .class. To run a java code, we need a class file containing main() function

(Though, we can write java program without main(), for the time-being you assume that we need a main() function!!!). Hence, it is a tradition to give the name of the java source code file as the name of the class containing main() function.

Let us have closer look at the terminologies used in the above program now –

class	is the keyword to declare a class.
Prg1	is the name of the class. You can use any valid identifier for a class name.
main()	is name of the method from which the program execution starts.
public	is a keyword indicating the access specifier of the method. The *public* members can be accessed from outside the class in which they have been declared. The *main()* function must be declared as *public* as it needs to be called from outside the class.
static	The keyword *static* allows *main()* to be called without having to instantiate a particular instance of the class. This is necessary since *main()* is called by the Java Virtual Machine before any objects are made.
void	indicates that *main()* method is not returning anything.
String args[ ]	The *main()* method takes an array of *String* objects as a command-line argument.
System	is a predefined class (present in java.lang package) which gives access to the system. It contains pre-defined methods and fields, which provides facilities like standard input, output, etc.
out	is a static final (means not inheritable) field (ie, variable)in System class which is of the type PrintStream (a built-in class, contains methods to print the different data values). Static fields and methods must be accessed by using the class name, so we need to use *System.out*.
println	is a public method in PrintStream class to print the data values. After printing the data, the cursor will be pushed to the next line (or we can say that, the data is followed by a *new line*).

DATA TYPES:



The Primitive Types Java defines eight primitive (or simple) data types viz. • byte, short, int, long :

belonging to Integers group involving whole-valued signed numbers.

• char : belonging to Character group representing symbols in character set like alphabets, digits, special characters etc.

• float, double : belonging to Floating-point group involving numbers with fractional part.

• boolean : belonging to Boolean group, a special way to represent true/false values.

These types can be used as primitive types, derived types (arrays) and as member of user-defined types (classes). All these types have specific range of values irrespective of the platform in which the program

Integers Java defines four integer types viz. byte, short, int and long. All these are signed numbers and Java does not support unsigned numbers. The width of an integer type should not be thought of as the amount of storage it consumes, but rather as the behaviour it defines for variables and expressions of that type. The Java run-time environment is free to use whatever size it wants, as long as the types behave as you declared them. The width and ranges of these integer types vary widely, as shown in this table:

Name	Width (in bits)	Range
long	64	-2^{63} to $+2^{63}-1$
int	32	-2^{31} to $+2^{31}-1$
short	16	-2^{15} to $+2^{15}-1$ (-32768 to +32767)
byte	8	-2^{7} to $+2^{7}-1$ (-128 to +127)

byte : This is the smallest integer type. Variables of type **byte** are especially useful when you are working with a stream of data from a network or file. They are also useful when you are working with raw binary data that may not be directly compatible with Java's other built-in types. Byte variables are declared by use of the ***byte*** keyword. For example,

```
byte b, c;
```

short : It is probably the least-used Java type. Here are some examples of **short** variable declarations:

```
short s;
short t;
```

int : The most commonly used integer type is **int**. In addition to other uses, variables of type **int** are commonly employed to control loops and to index arrays. Although you might think that using a **byte** or **short** would be more efficient than using an **int** in situations in which the larger range of an **int** is not needed, this may not be the case. The reason is that when **byte** and **short** values are used in an expression they are *promoted* to **int** when the expression is evaluated. (Type promotion is described later in this chapter.) Therefore, **int** is often the best choice when an integer is needed.

long : It is useful for those occasions where an **int** type is not large enough to hold the desired value. The range of a **long** is quite large. This makes it useful when big, whole numbers are needed.

Program 1.5: Program to illustrate need for *long* data type

```
class Light
{
      public static void main(String args[ ])
      {
            int lightspeed;
            long days, seconds, distance;


            // approximate speed of light in miles per second
            lightspeed = 186000;
            days = 1000;                  // specify number of days here

            seconds = days * 24 * 60 * 60;   // convert to seconds
            distance = lightspeed * seconds;      // compute distance

            System.out.print("In " + days);
            System.out.print(" days light will travel about ");
            System.out.println(distance + " miles.");
      }
}
```

The output will be –

In 1000 days light will travel about 16070400000000 miles.

Floating –Point Types

Floating-point (or real) numbers are used when evaluating expressions that require fractional precision. Java implements the standard (IEEE–754) set of floating-point types and operators. There are two kinds of floating-point types, **float** and **double**, which represent single- and double-precision numbers, respectively. Their width and ranges are shown here:

Name	Width (in bits)	Range
double	64	4.9e–324 to 1.8e+308
float	32	1.4e–045 to 3.4e+038

float : The type **float** specifies a *single-precision* value that uses 32 bits of storage. Single precision is faster on some processors and takes half as much space as double precision, but will become imprecise when the values are either very large or very small. Variables of type **float** are useful when you need a fractional component, but don't require a large degree of precision. For example, **float** can be useful when representing currencies, temperature etc. Here are some example **float** variable declarations:

```
float hightemp, lowtemp;
```

double : Double precision is actually faster than single precision on some modern processors that have been optimized for high-speed mathematical calculations. All transcendental math functions, such as **sin( )**, **cos( )**, and **sqrt( )**, return **double** values. When you need to maintain accuracy over many iterative calculations, or are manipulating large-valued numbers, **double** is the best choice.

Program 1.6 Finding area of a cirlce

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

          double pi, r, a;
          r = 10.8;
          pi = 3.1416;
          a = pi * r * r;
          System.out.println("Area of circle is " + a);
      }
}
```

The output would be –

Area of circle is 366.436224

Characters

In Java, **char** is the data type used to store characters. In C or C++, ***char*** is of 8 bits, whereas in Java it requires 16 bits. Java uses Unicode to represent characters. **Unicode** is a computing industry standard for the consistent encoding, representation and handling of text expressed in many languages of the world. Unicode has a collection of more than 109,000 characters covering 93 different languages like Latin, Greek, Arabic, Hebrew etc. That is why, it requires 16 bits. The range of a **char** is 0 to 65,536. The standard set of characters known as ASCII still ranges from 0 to 127 as always, and the extended 8-bit character set, ISO-Latin-1, ranges from 0 to 255. Since Java is designed to allow programs to be written for worldwide use, it makes sense that it would use Unicode to represent characters. Though it seems to be wastage of memory as the languages like English, German etc. can accommodate their character set in 8 bits, for a global usage point of view, 16-bits are necessary.

Though, ***char*** is designed to store Unicode characters, we can perform arithmetic operations on them. For example, we can add two characters (but, not char variables!!), increment/decrement character variable etc. Consider the following example for the demonstration of characters.

Program 1.7 Demonstration of *char* data type

```
class CharDemo
{
      public static void main(String args[])
      {
            char ch1=88, ch2='Y';

            System.out.print("ch1 and ch2: ");
            System.out.println(ch1 + " " + ch2);

            ch1++;      //increment in ASCII (even Unicode) value
            System.out.println("ch1 now contains "+ch1);

            --ch2;           //decrement in ASCII (even Unicode) value
            System.out.println("ch2 now contains "+ch2);

            /*    ch1=35;
                  ch2=30;
                  char ch3;

                   ch3=ch1+ch2;    //Error
              */
```

```
            ch2='6'+'A';    //valid
            System.out.println("ch2 now contains "+ch2);
        }
    }
```

The output would be –

ch1 and ch2: X Y
ch1 now contains Y
ch2 now contains X
ch2 now contains w

Booleans

For storing logical values (***true*** and ***false***), Java provides this primitive data type. Boolean is the output of any expression involving relational operators. For control structures (like if, for, while etc.) we need to give boolean type. In C or C++, false and true values are indicated by zero and a non-zero numbers respectively. And the output of relational operators will be 0 or 1. But, in Java, this is not the case. Consider the following program as an illustration.

Program 1.8 Demonstration of *Boolean* data type

```
class BoolDemo
{
    public static void main(String args[])
    {
        boolean b = false;

        System.out.println("b is " + b);
        b = true;
        System.out.println("b is " + b);

        if(b)
            System.out.println("True block");

        b = false;
        if(b)
            System.out.println("False Block will not be executed");

        b=(3<5);
        System.out.println("3<5 is " +b);
    }
}
```

The output would be –

b is false
b is true
True block
3<5 is true

NOTE: Size of a Boolean data type is JVM dependent. But, when Boolean variable appears in an expression, Java uses 32-bit space (as int) for Boolean to evaluate expression.

A Closer Look at Literals:

A literal is the source code representation of a fixed value. In other words, by literal we mean any number, text, or other information that represents a value. Literals are represented directly in our code without requiring computation. Here we will discuss Java literals in detail.

Integer Literals: Integers are the most commonly used type in the typical program. Any whole number value is an integer literal. For example, 1, 25, 33 etc.

Floating-Point Literals: Floating-point numbers represent decimal values with a fractional component. They can be expressed in either standard or scientific notation. Standard notation consists of a whole number component followed by a decimal point followed by a fractional component. For example, 2.0, 3.14159, and 0.6667 represent valid standard-notation floating-point numbers.

Floating-point literals in Java default to double precision. To specify a float literal, you must append an F or f to the constant. You can also explicitly specify a double literal by appending a D or d. Doing so is, of course, redundant. The default double type consumes 64 bits of storage, while the less-accurate float type requires only 32 bits.

Boolean Literals Boolean literals are simple. There are only two logical values that a boolean value can have, true and false. The values of true and false do not convert into any numerical representation. The true literal in Java does not equal 1, nor does the false literal equal 0. In Java, they can only be assigned to variables declared as boolean, or used in expressions with Boolean operators.

Character Literals Characters in Java are indices into the Unicode character set. They are 16-bit values that can be converted into integers and manipulated with the integer operators, such as the addition and subtraction operators. A literal character is represented inside a pair of single quotes. All of the visible ASCII characters can be directly entered inside the quotes, such as 'a', 'z', and '@'. For characters that are impossible to enter directly, there are several escape sequences that allow you to enter the character you need, such as '\'' for the single-quote character itself and '\n' for the new-line character. There is also a mechanism for directly entering the value of a character in octal or hexadecimal. For octal notation, use the backslash followed by the three-digit number. For example, '\141' is the letter 'a'. For hexadecimal, you enter a backslash-u (\u), then exactly four hexadecimal digits. Following table shows the character escape sequences.

Escape Sequence	Description
\ddd	Octal character (ddd)
\uxxxx	Hexadecimal Unicode character (xxxx)
\'	Single quote
\"	Double quote
\\	Back slash
\r	Carriage return (Enter key)
\n	New line (also known as line feed)
\f	Form feed
\t	Tab
\b	Back space

String Literals

String literals are a sequence of characters enclosed within a pair of double quotes. Examples of string literals are

```
"Hello World"
"two\nlines"
"\"This is in quotes\""
```

Java strings must begin and end on the same line. There is no line-continuation escape sequence as there is in some other languages. In Java, strings are actually objects and are discussed later in detail.

VARIABLES

The variable is the basic unit of storage. A variable is defined by the combination of an identifier, a type, and an optional initializer. In addition, all variables have a scope, which defines their visibility, and a lifetime.

Declaring a Variable

In Java, all variables must be declared before they can be used. The basic form of a variable declaration is shown here:

***type identifier* [= *value*][, *identifier* [= *value*] ...] ;**

The *type* is any of primitive data type or class or interface. The *identifier* is the name of the variable. We can initialize the variable at the time of variable declaration. To declare more than one variable of the specified type, use a comma-separated list. Here are several examples of variable declarations of various types. Note that some include an initialization.

```
int a, b=5, c;
byte z = 22;
double pi = 3.1416;
char x = '$';
```

Dynamic Initialization

Although the preceding examples have used only constants as initializers, Java allows variables to be initialized dynamically, using any expression valid at the time the variable is declared. For example,

```
int a=5, b=4;
int c=a*2+b;     //variable declaration and dynamic initialization
```

The key point here is that the initialization expression may use any element valid at the time of the initialization, including calls to methods, other variables, or literals.

The Scope and Lifetime of Variables

A variable in Java can be declared within a block. A block is begun with an opening curly brace and ended by a closing curly brace. A block defines a scope which determines the accessibility of variables and/or objects defined within it. It also determines the lifetime of those objects.

- Java has two scopes viz.
- class level scope
- and method (or function) level scope.

Class level scope is discussed later and we will discuss method scope here.

- The scope defined by a method begins with its opening curly brace. However, if that method has parameters, they too are included within the method's scope.
- As a general rule, variables declared inside a scope are not visible (that is, accessible) to code that is defined outside that scope.
- Thus, when you declare a variable within a scope, you are localizing that variable and protecting it from unauthorized access and/or modification. Indeed, the scope rules provide the foundation for encapsulation. Scopes can be nested.
- For example, each time you create a block of code, you are creating a new, nested scope. When this occurs, the outer scope encloses the inner scope.
- This means that objects declared in the outer scope will be visible to code within the inner scope. However, the reverse is not true.
- Objects declared within the inner scope will not be visible outside it. ***Variables are created when their scope is entered, and destroyed when their scope is left.***
- This means that a variable will not hold its value once it has gone out of scope. Therefore, variables declared within a method will not hold their values between calls to that method.
- Also, a variable declared within a block will lose its value when the block is left. Thus, the lifetime of a variable is confined to its scope.

Program 1.9 Demonstration of *scope* of variables

```
class Scope
{
      public static void main(String args[])
      {
            int x=10, i;          // x and i are local to main()

            if(x == 10)
            {
                  int y = 20;     // y is local to this block
                  System.out.println("x and y: " + x + " " + y);
                  x = y * 2;
            }

            // y = 100;      //y cannot be accessed here
```

```
                System.out.println("x is " + x);

                for(i=0;i<3;i++)
                {
                      int a=3;          // a is local to this block
                      System.out.println("a is " + a);
                      a++;
                }
          }
    }
```

The output would be –

x and y: 10 20
x is 40
a is 3
a is 3
a is 3

Note that, variable ***a*** is declared within the scope of ***for*** loop. Hence, each time the loop gets executed, variable ***a*** is created newly and there is no effect of ***a++*** for next iteration.

1.13 Type Conversion and Casting

It is quite common in a program to assign value of one type to a variable of another type. If two types are compatible, Java performs ***implicit type conversion***. For example, *int* to *long* is always possible. But, whenever the types at two sides of an assignment operator are not compatible, then Java will not do the conversion implicitly. For that, we need to go for ***explicit type conversion*** or ***type casting.***

Java's Automatic Conversions

When one type of data is assigned to another type of variable, an *automatic type conversion* will take place if the following two conditions are met:

- The two types are compatible.
- The destination type is larger than the source type.

When these two conditions are met, a ***widening conversion*** takes place. For example, the **int** type is always large enough to hold all valid **byte** values, so no explicit cast statement is required. For widening conversions, the numeric types, including integer and floating-point types, are compatible with each other. However, there are no automatic conversions from the numeric types to **char** or **boolean**. Also, **char** and **boolean** are not compatible with each other. As mentioned earlier, Java also performs an automatic type conversion when storing a literal integer constant into variables of type **byte**, **short**, **long**, or **char**.

Casting Incompatible Types

Although the automatic type conversions are helpful, they will not fulfill all needs. For example, what if you want to assign an **int** value to a **byte** variable? This conversion will not be performed automatically, because a **byte** is smaller than an **int**. This kind of conversion is sometimes called a ***narrowing conversion***, since you are explicitly making the value narrower so that it will fit into the target type. To create a conversion between two incompatible types, we must use a cast. A *cast* is simply an explicit type conversion. It has this general form:

(target-type) value

Here, *target-type* specifies the desired type to convert the specified value to. For example,

```
int a;
byte b;
b = (byte) a;
```

When a floating-point value is assigned to an integer type, the fractional component is lost. And such conversion is called as ***truncation*** (***narrowing***)***.*** If the size of the whole number component is too large to fit into the target integer type, then that value will be reduced modulo the target type's range. Following program illustrates various situations of explicit casting.

Program 1.10 Illustration of type conversion

```
class Conversion
{
      public static void main(String args[])
      {
            byte b;
            int i = 257;
            double d = 323.142;

            System.out.println("\nConversion of int to byte.");
            b = (byte) i;
            System.out.println("i and b " + i + " " + b);

            System.out.println("\nConversion of double to int.");
            i = (int) d;
            System.out.println("d and i " + d + " " + i);

            System.out.println("\nConversion of double to byte.");
            b = (byte) d;


           System.out.println("d and b " + d + " " + b);
      }
}
```

The output would be –

```
Conversion of int to byte:      i = 257        b = 1
Conversion of double to int:   d = 323.142     i = 323
Conversion of double to byte: d = 323.142      b = 67
```

Here, when the value 257 is cast into a **byte** variable, the result is the remainder of the division of 257 by 256 (the range of a **byte**), which is 1 in this case. When the **d** is converted to an **int**, its fractional component is lost. When **d** is converted to a **byte**, its fractional component is lost, *and* the value is reduced modulo 256, which in this case is 67.

1.14 Automatic Type promotion in Expression

Apart from assignments, type conversion may happen in expressions also. In an arithmetic expression involving more than one operator, some intermediate operation may exceed the size of either of the operands. For example,

```
byte x=25, y=80, z=50;
int p= x*y/z ;
```

Here, the result of operation x*y is 4000 and it exceeds the range of both the operands i.e. byte (-128 to +127). In such a situation, Java promotes byte, short and char operands to int That is, the operation x*y is performed using int but not byte and hence, the result 4000 is valid.

On the other hand, the automatic type conversions may cause error. For example,

```
byte x=10;
byte y= x *3;    //causes error!!!
```

Here, the result of x *3 is 30, and is well within the range of byte. But, for performing this operation, the operands are automatically converted to byte and the value 30 is treated as of int type. Thus, assigning an int to byte is not possible, which generates an error. To avoid such problems, we should use type casting. That is,

```
byte x=10;
byte y=(byte) (x *3);    //results 30
```

Type Promotion Rules

Java defines several *type promotion* rules that apply to expressions. They are as follows:

- All **byte**, **short**, and **char** values are promoted to **int.**
- If one operand is a **long**, the whole expression is promoted to **long**.
- If one operand is a **float,** the entire expression is promoted to **float**.
- If any of the operands is **double**, the result is **double**.

Program 1.11 Demonstration of type promotions

```
class TypePromo
{
   public static void main(String args[])
   {
         byte b = 42;


          char c = 'a';
          short s = 1024;
          int i = 50000;
          float f = 5.67f;
          double d = .1234;
          double result = (f * b) + (i / c) - (d * s);

          System.out.println("result = " + result);
    }
 }
```

The output would be –

result = 626.7784146484375

Let's look closely at the type promotions that occur in this line from the program:

```
double result = (f * b) + (i / c) - (d * s);
```

In the first sub-expression, **f * b**, **b** is promoted to a **float** and the result of the sub-expression is **float**. Next, in the sub-expression **i / c**, **c** is promoted to **int**, and the result is of type **int**. Then, in **d * s**, the value of **s** is promoted to **double**, and the type of the sub-expression is **double**. Finally, these three intermediate values, **float**, **int**, and **double**, are considered. The outcome of **float** plus an **int** is a **float**. Then the resultant **float** minus the last **double** is promoted to **double**, which is the type for the final result of the expression.

Arrays

Definition/meaning:

In Java, an array is a data structure that allows you to store multiple values of the same data type in a contiguous block of memory.

- Arrays have a fixed size.
- A specific element in an array is accessed by its index.
- Arrays offer a convenient means of grouping related information.

TYPES OF ARRAYS

1. One-Dimensional Arrays
2. Multidimensional Arrays

One-Dimensional Arrays

A *one-dimensional array* is, essentially, a list of like-typed variables.

ARRAY SYNTAX:

type var-name[ ];

Here type determines the data type of each element that comprises the array.

Example: the following declares an array named **month_days** with the type "array of int":

EXAMPLE FOR ARRAY DECLARATION

```
int month_days[];
```

- The value of **month_days** is set to **null**, which represents an array with no value.
- To link **month_days** with an actual, physical array of integers, you must allocate one using **new** and assign it to **month_days**.
- **new** is a special operator that allocates memory.
- The general form of **new** as it applies to one-dimensional arrays appears as follows:

array-var = new *type*[*size*];

- *Type* specifies the data type of data being allocated,
- *size* specifies the number of elements in the array,
- and *array-var* is the array variable that is linked to the array.

This example allocates a 12-element array of integers and links them to **month_days**.

```
month_days = new int[12];
```

After this statement executes, month_days will refer to an array of 12 integers
Summary:

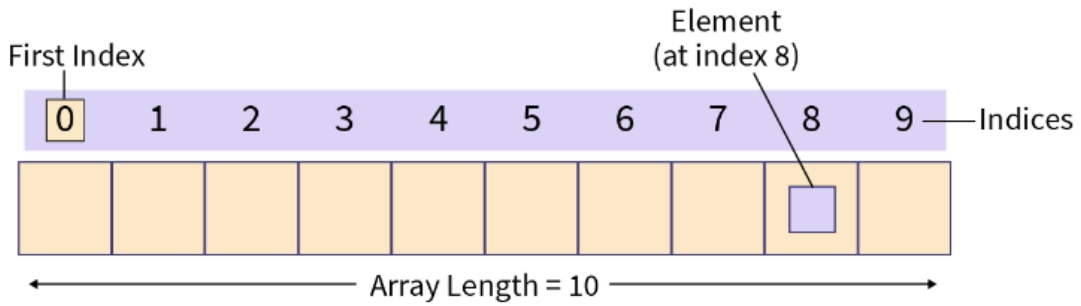
Obtaining an array is a two-step process.

- you must declare a variable of the desired array type.
- you must allocate the memory that will hold the array, using **new**, and assign it to the array variable.

For example, following statement create an array of 10 integers – int arr[ ]=new int[10];

- Array index starts with 0 and we can assign values to array elements as –
- arr[0]=25;
- arr[1]=32;

and so on.

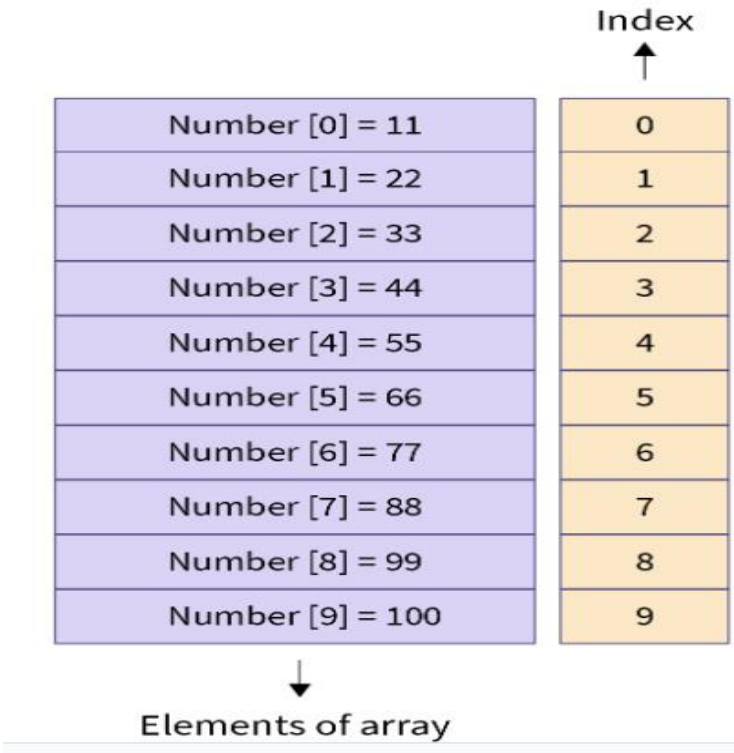


An *array initializer*
It is a list of comma-separated expressions surrounded by curly braces. The commas separate the values of the array elements. The array will automatically be created large enough to hold the number of elements you specify in the array initializer. There is no need to use **new**. For

example, to store the number of days in each month, the following code creates an initialized array of integers:

```
// An improved version of the previous program.
class AutoArray {
public static void main(String args[]) {
int month_days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31,
30, 31 };
System.out.println("April has " + month_days[3] + " days.");
}
}
```

```java
public class AssignValues {

  public static void main(String args[]) {
    int number[]; // array declared
    number = new int[10]; // allocating memory, initialization
    number[0] = 11;
    number[1] = 22;
    number[2] = 33;
    number[3] = 44;
    number[4] = 55;
    number[5] = 66;
    number[6] = 77;
    number[7] = 88;
    number[8] = 99;
    number[9] = 100;
  }
}
```



EXAMPLE PROGRAMS FOR 1D ARRAY

Example 1:

public class Example {

```
  public static void main(String args[]) {
    int arr[] = { 1, 5, 10, 15, 20 }; // initializing array
    for (int i = 0; i < arr.length; i++) {
      System.out.println(arr[i] + " "); // printing array elements
    }
  }
}
```

This array program initializes and print the array elements.

Example 2:

```
import java.util.Scanner;
public class OneDimensionalArrayInput {
  public static void main(String args[]) {
    // creating object of Scanner class
    Scanner scan = new Scanner(System.in);
    System.out.println("Enter length of Array: ");
    int arrLength = scan.nextInt();
    int[] anArray = new int[arrLength];
    System.out.println("Enter the elements of the Array");
    for (int i = 0; i < arrLength; i++) {
      // taking array input
      anArray[i] = scan.nextInt();
    }
    System.out.println("One dimensional array elements are:");
    for (int i = 0; i < arrLength; i++) {
      // printing array elements
      System.out.print(anArray[i] + " ");
    }
  }
}
```

EXAMPLE 3: A String Array which is initialized and its elements are printed in the output

```
public class StringArray {
  public static void main(String args[]) {
    String[] array = { "Learning", "One-dimensional array", "in java" };
    System.out.println("String array elements are:");
    for (int i = 0; i < array.length; i++) {
      System.out.println(array[i]);
    }
  }
}
```

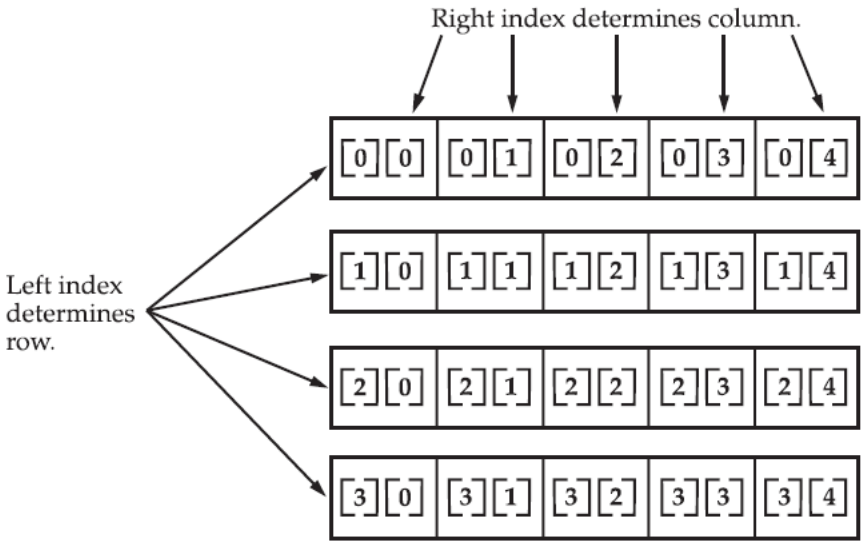
Multidimensional Arrays

In Java, *multidimensional arrays* are actually arrays of arrays.

For example, the following declares a twodimensional array variable called **twoD**.

```
int twoD[][] = new int[4][5];
```

This allocates a 4 by 5 array and assigns it to **twoD as shown in the diagram below**



Given: int twoD [ ] [ ] = new int [4] [5] ;

EXAMPLE: A Java program to find the row, column position of a specified number (row, column position) in a given 2- dimensional array.

```
class Search
{
int m, n;
Scanner s = new Scanner(System.in);
Boolean calculate(int arr[][])
{
int row = 0;
int num;
int col = arr[row].length - 1;
System.out.println("Enter the number to find: ");
num = s.nextInt();
while (row < arr.length && col >= 0)
{
if (arr[row][col] == num)
{
System.out.println("The number has been found at array index: [" + row + "][" + col + "]");
return true;
}
if (arr[row][col] < num)
{
row++;
}
else
{
col--;
}
}
```

```
System.out.println("The number is not found in the entered array.");
return false;
}
}

public static void main(String args[])
{
int m, n;
System.out.println("Enter the size of the 2D Array:");
Scanner s = new Scanner(System.in);
m = s.nextInt();
n = s.nextInt();
int arr[][] = new int[m][n];
System.out.println("Enter the numbers for the array:");
for (int i = 0; i < m; i++)
for (int j = 0; j < n; j++)
arr[i][j] = s.nextInt();
Search ss = new Search();
ss.calculate(arr);
}
}
```

Program 1.12 Demonstration of 2-d array

```
class TwoDArray
{
      public static void main(String args[])
      {
            int twoD[][]= new int[3][4];
            int i, j;

            for(i=0; i<3; i++)
                  for(j=0; j<4; j++)
                        twoD[i][j] = i+j;

            for(i=0; i<3; i++)
            {
                  for(j=0; j<4; j++)
                        System.out.print(twoD[i][j] + " ");

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

The output would be –

```
0 1 2 3
1 2 3 4
2 3 4 5
```

Alternative Array Declaration Syntax:

There is a second form that may be used to declare an array:

type*[ ] *var-name;

Here, the square brackets follow the type specifier, and not the name of the array variable.
For example, the following two declarations are equivalent:

```
int a1[] = new int[3];
int[] a2 = new int[3];
```

The following declarations are also equivalent:

```
char twod1[][] = new char[3][4];
char[][] twod2 = new char[3][4];
```

This alternative declaration form offers convenience when declaring several arrays at the same time. For example,

```
int[] nums, nums2, nums3; // create three arrays
```

creates three array variables of type **int**.
It is the same as writing

```
int nums[], nums2[], nums3[]; // create three arrays
```

ArrayIndexOutOfBoundsException in Java

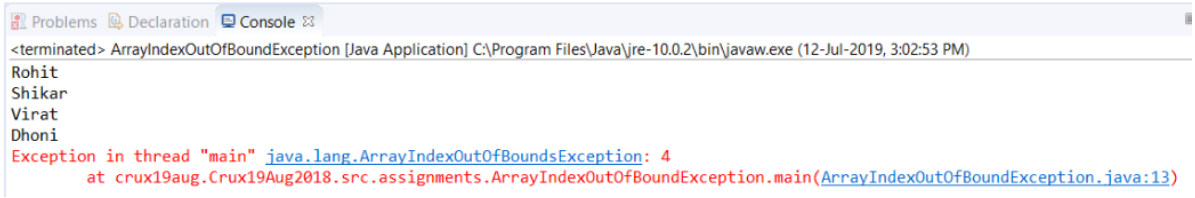
The ArrayIndexOutOfBoundsException occurs whenever we are trying to access any item of an array at an index which is not present in the array. In other words, the index may be negative or exceed the size of an array.

The ArrayIndexOutOfBoundsException is a subclass of IndexOutOfBoundsException, and it implements the Serializable interface.

Example of ArrayIndexOutOfBoundsException

```
public class ArrayIndexOutOfBoundException {

   public static void main(String[] args) {
String[] arr = {"Rohit","Shikar","Virat","Dhoni"};
//Declaring 4 elements in the String array

      for(int i=0;i<=arr.length;i++) {

//Here, no element is present at the iteration number arr.length, i.e 4
         System.out.println(arr[i]);
//So it will throw ArrayIndexOutOfBoundException at iteration 4
      } } }
```

Output:



How to avoid ArrayIndexOutOfBoundsException

One of the biggest issues due to which ArrayIndexOutOfBoundsException occurs is that the indexing of array starts with 0 and not 1 and the last element is at the array length -1 index, due to which, users commonly make a mistake while accessing the elements of the array.

Always take care while making the starting and end conditions of the loop.

Using an enhanced for loop

An enhanced for loop is the one which iterates over contiguous memory allocation and ***iterates only on legal indexes.***
For example:-

```
public class ArrayIndexOutOfBoundException {
  public static void main(String[] args) {
    ArrayList<String> listOfPlayers = new ArrayList<>();
    listOfPlayers.add("Rohit");
    listOfPlayers.add("Shikhar");
    listOfPlayers.add("Virat");
    listOfPlayers.add("Dhoni");
    for(String val : listOfPlayers) // enhanced for loops
{
    System.out.println("Player Name: "+val);
    } }}
```

CLASS: FUNDAMENTALS

Class Fundamentals Class can be thought of as a user-defined data type. We can create objects of that data type. So, we can say that class is a template for an object and an object is an instance of a class. Most of the times, the terms object and instance are used interchangeably.

- Any thing you wish to implement in a Java program must be encapsulated within a class
- It's a user defined data-type
- When we create a class, we declare its exact form and nature
- Once defined it is used to create objects
- Object is instance of a class and Class acts as a template for the object

class *classname {*

type instance-variable1;

type instance-variable2;

// ...

type instance-variableN;

type methodname1(parameter-list) {

// body of method

}

type methodname2(parameter-list) {

// body of method

}

// ...

type methodnameN(parameter-list) {

// body of method

}

}

A Simple Class Here we will consider a simple example for creation of class, creating objects and using members of the class. One can store the following program in a single file called BoxDemo.java.

```
class Box
{
        double w, h, d;
}
class BoxDemo
{
        public static void main(String args[])
        {
                Box b1=new Box();
                Box b2=new Box();
                double vol;

                b1.w=2;
                b1.h=4;
                b1.d=3;

                b2.w=5;
                b2.h=6;
                b2.d=2;

                vol=b1.w*b1.h*b1.d;
                System.out.println("Volume of Box1 is " + vol);

                vol=b2.w*b2.h*b2.d;
                System.out.println("Volume of Box2 is " + vol);
        }
}
```

The output would be –

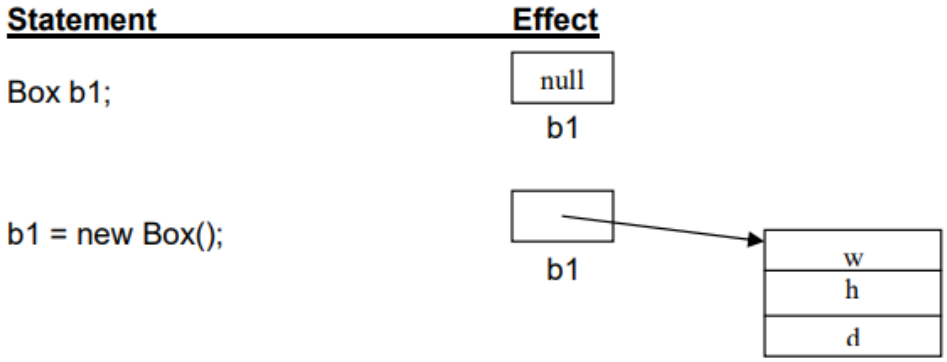
```
Volume of Box1 is 24.0
Volume of Box1 is 60.0
```

With the term dynamic memory allocation, we can understand that the keyword "**new**" allocates memory for the object during runtime. So, depending on the user's requirement memory will be utilized. This will avoid the problems with static memory allocation (either shortage or wastage of memory during runtime). If there is no enough memory in the heap when we use new for memory allocation, it will throw a run-time exception.

DECLARING OBJECTS

- Declaring Objects Creating a class means having a user-defined data type. To have a variable of this new data type, we should create an object.
- Consider the following declaration: **Box b1**;
- This statement will not actually create any physical object, but the object name b1 can just refer to the actual object on the heap after memory allocation as follows
  b1 = new Box ();
- We can even declare an object and allocate memory using a single statement
  Box b1=new Box();
- Without the usage of new, the object contains null. Once memory is allocated dynamically, the object b1 contains the address of real object created on the heap. The memory map is as shown in the following diagram.

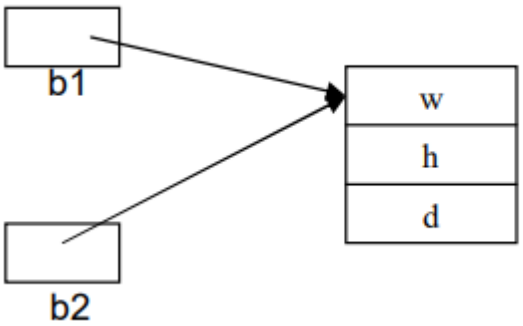
- ***The general form for object creation is – obj_name = new class_name();***
- With the term dynamic memory allocation, we can understand that the keyword new allocates memory for the object during runtime. So, depending on the user's requirement memory will be utilized.



Assigning Object Reference Variables.

When an object is assigned to another object, no separate memory will be allocated. Instead, the second object refers to the same location as that of first object. Consider the following declaration – Box b1= new Box(); Box b2= b1

Now both b1 and b2 refer to same object on the heap. The memory representation for two objects can be shown as.



Thus, any change made for the instance variables of one object affects the other object also. Although b1 and b2 both refer to the same object, they are not linked in any other way. For example, a subsequent assignment to b1 will simply unhook b1 from the original object without affecting the object or affecting b2.

For example:

Box b1 = new Box();

Box b2 = b1; // ... b1 = null;

Here, b1 has been set to null, but b2 still points to the original object. NOTE that when you assign one object reference variable to another object reference variable, you are not creating a copy of the object, you are only making a copy of the reference.

INTRODUCING METHODS

A class can consist of instance variables and methods. We have seen declaration and usage of instance variables in Program 2.1. Now, we will discuss about methods. The general form of a method is –

```
ret_type method_name(para_list)
{
        //body of the method
        return value;
}
```

Here, ***ret_type*** specifies the data type of the variable returned by the method. It may be any primitive type or any other derived type including name of the same class. If the method does not return any value, the ***ret_type*** should be specified as ***void***.

method_name is any valid name given to the method

para_list is the list of parameters (along with their respective types) taken the method. It may be even empty also.

body of method is a code segment written to carryout some process for which the method is meant for.

return is a keyword used to send ***value*** to the calling method. This line will be absent if the ***ret_type*** is void.

Adding Methods to Box class:

- It is advisable to have methods to operate on those data.
- Because, methods acts as interface to the classes.
- This allows the class implementer to hide the specific layout of internal data structures behind cleaner method abstractions.
- In addition to defining methods that provide access to data, you can also define methods that are used internally by the class itself.
- Consider the following example –

```
class Box
{
        double w, h, d;

        void volume()
        {
                System.out.println("The volume is " + w*h*d);
        }
}

class BoxDemo
{
        public static void main(String args[])
        {
                Box b1=new Box();
                Box b2=new Box();

                b1.w=2;
                b1.h=4;
                b1.d=3;

                b2.w=5;
                b2.h=6;
                b2.d=2;

                b1.volume();
                b2.volume();
        }
}
```

The output would be –

```
The volume is 24.0
The volume is 60.0
```

In the above program, the Box objects b1 and b2 are invoking the member method volume() of the Box class to display the volume. To attach an object name and a method name, we use dot (.) operator. Once the program control enters the method volume(), we need not refer to object name to use the instance variables w, h and d.

Returning a value:

In the previous example, we have seen a method which does not return anything. Now we will modify the above program so as to return the value of volume to main() method.

```
class Box
{
        double w, h, d;

        double volume()
        {
                return w*h*d;
        }
}

class BoxDemo
{
        public static void main(String args[])
        {
                Box b1=new Box();
                Box b2=new Box();
                double vol;

                b1.w=2;
                b1.h=4;
                b1.d=3;

                b2.w=5;
                b2.h=6;
                b2.d=2;

                vol = b1.volume();
                System.out.println("The volume is " + vol);
                System.out.println("The volume is " + b2.volume());
        }
}
```

The output would be –

```
The volume is 24.0
The volume is 60.0
```

There are two important things to understand about returning values:

• The type of data returned by a method must be compatible with the return type specified by the method. For example, if the return type of some method is boolean, you could not return an integer.

• The variable receiving the value returned by a method (such as vol, in this case) must also be compatible with the return type specified for the method.

Constructors

- Constructor is a block of codes similar to the method which is invoked automatically when the object gets created.
- Constructors are used for object initialization.
- They have same name as that of the class.

- Since they are called automatically, there is no return type for them.
- Constructors may or may not take parameters.

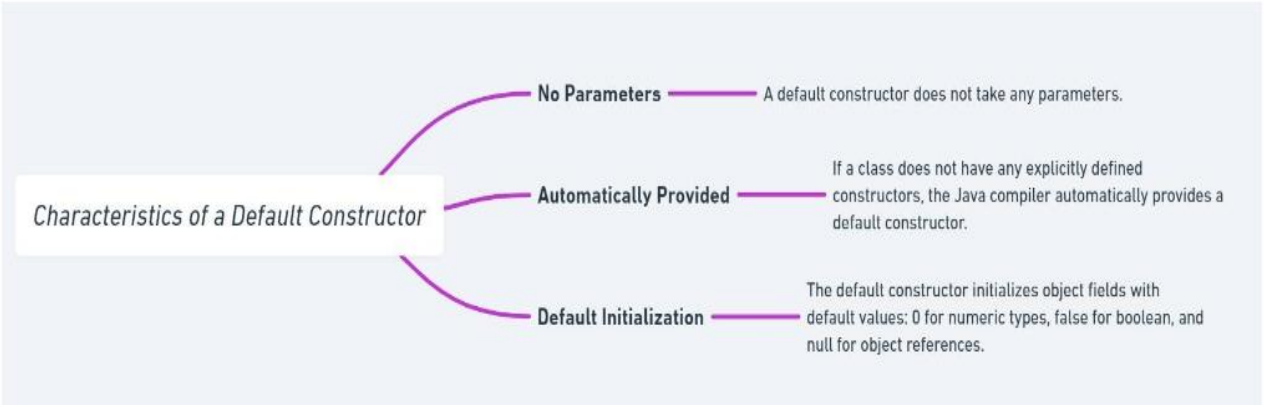
How Java Constructors are Different From Java Methods?

- Constructors must have the same name as the class within which it is defined it is not necessary for the method in Java.
- Constructors do not return any type while method(s) have the return type or **void** if does not return any value.
- Constructors are called only once at the time of Object creation while method(s) can be called any number of times.

Example for constructors

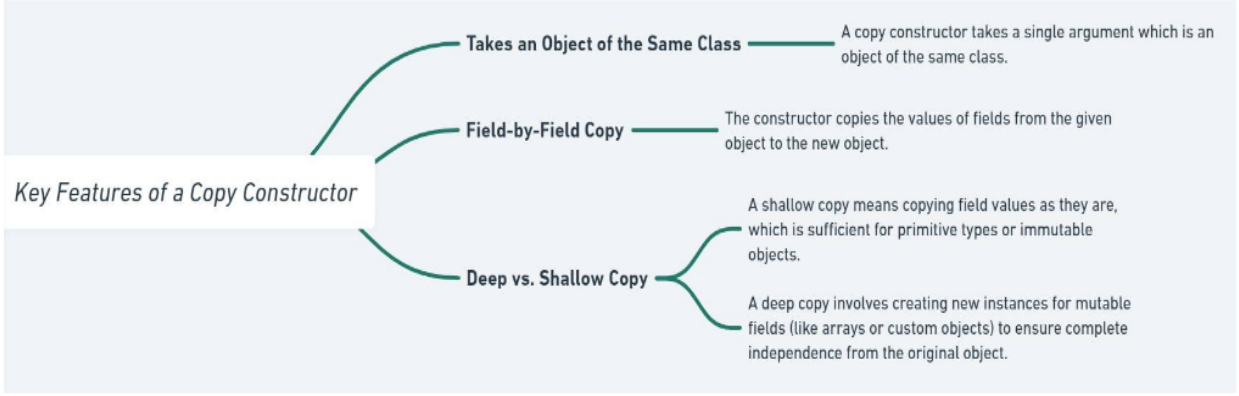
```
public class Person {
   private String name;
   private int age;

   // Default constructor
   public Person() {
      name = "Unknown";
      age = 0;
   }
   // Parameterized constructor
   public Person(String name, int age) {
      this.name = name;
      this.age = age;
   }

   // Getter methods (not related to constructors, just for illustration)
   public String getName() {
      return name;
   }

   public int getAge() {
      return age;
   }
}
```

Characteristics of a Default Constructor



Key Features of Parameterized Constructors

Key Features of a Copy Constructor



Comparison Table of Default Constructor, Parameterized Constructor and Copy Constructor

Feature	Default Constructor	Parameterized Constructor	Copy Constructor
Definition	A no-argument constructor provided by the compiler if no other constructors are defined.	A constructor that takes one or more parameters to initialize an object with specific values.	A constructor that creates a new object as a copy of an existing object of the same class.

Parameters	None	One or more parameters	One parameter of the same class type
Initialization	Initializes fields with default values (e.g., 0 for numbers, false for boolean, null for objects).	Initializes fields with specific values provided by the arguments.	Initializes fields with the values from an existing object.
Usage	Used when no specific initialization is required or to ensure objects have default values.	Used to initialize objects with specific values at the time of creation.	Used to create a new object with the same state as an existing object.
Automatically Provided	Yes, if no other constructors are defined.	No, it must be explicitly defined by the programmer.	No, it must be explicitly defined by the programmer.
Benefits	Ensures objects have default values; simple and straightforward.	Allows objects to be initialized with specific values; flexible and convenient.	Allows for easy duplication of objects; ensures new object has the same state as the original.
Common Use Cases	Placeholder objects, simple initializations.	Creating objects with specific configurations or initial states.	Cloning objects, ensuring new objects start with the same state as an existing object.

What is this keyword in Java?

- **this** keyword in Java represents the current instance variable of a class.
- If in case, the formal parameter and data members (instance variables) of the class are the same, then it leads to ambiguity.
- So, in order to differentiate between formal parameter and data member (instance variables) of the class, the data member (instance variables) of the class must be preceded by the "**this**" keyword.

Syntax:
this. data member of the current class

Note: Class variables, Instance variables and data members all are same.

Understanding the problem without this keyword

Let's understand the problem if we don't use this keyword by the example given below:

```
class Student{
int rollno;
String name;
float fee;
Student(int rollno,String name,float fee){
rollno=rollno;
name=name;
fee=fee;
}
void display(){System.out.println(rollno+" "+name+" "+fee);}
}
class TestThis1{
public static void main(String args[]){
Student s1=new Student(111,"ankit",5000f);
Student s2=new Student(112,"sumit",6000f);
s1.display();
s2.display();
}}
```

Output:

```
0 null 0.0
0 null 0.0
```

In the above example, parameters (formal arguments) and instance variables are same. So, we are using this keyword to distinguish local variable and instance variable.

Solution of the above problem by this keyword

```
class Student{
int rollno;
String name;
float fee;
Student(int rollno,String name,float fee){
this.rollno=rollno;
this.name=name;
this.fee=fee;
}
void display(){System.out.println(rollno+" "+name+" "+fee);}
}

class TestThis2{
public static void main(String args[]){
Student s1=new Student(111,"ankit",5000f);
Student s2=new Student(112,"sumit",6000f);
s1.display();
s2.display();
}}
```

Output:

```
111 ankit 5000.0
112 sumit 6000.0
```

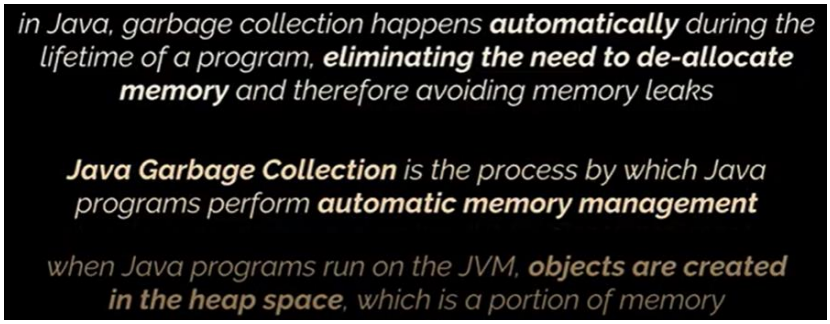
If local variables(formal arguments) and instance variables are different, there is no need to use this keyword.

Garbage Collection:

- In C++, dynamically allocated objects must be manually released by use of a **delete** operator.

- *The technique that accomplishes deallocation of memory for the user automatically is called Garbage Collection*.

- It works like this: when no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed. There is no explicit need to destroy objects as in C++.



Heap memory stores 2 types of objects

GENERATION GARBAGE COLLECTION IN JAVA

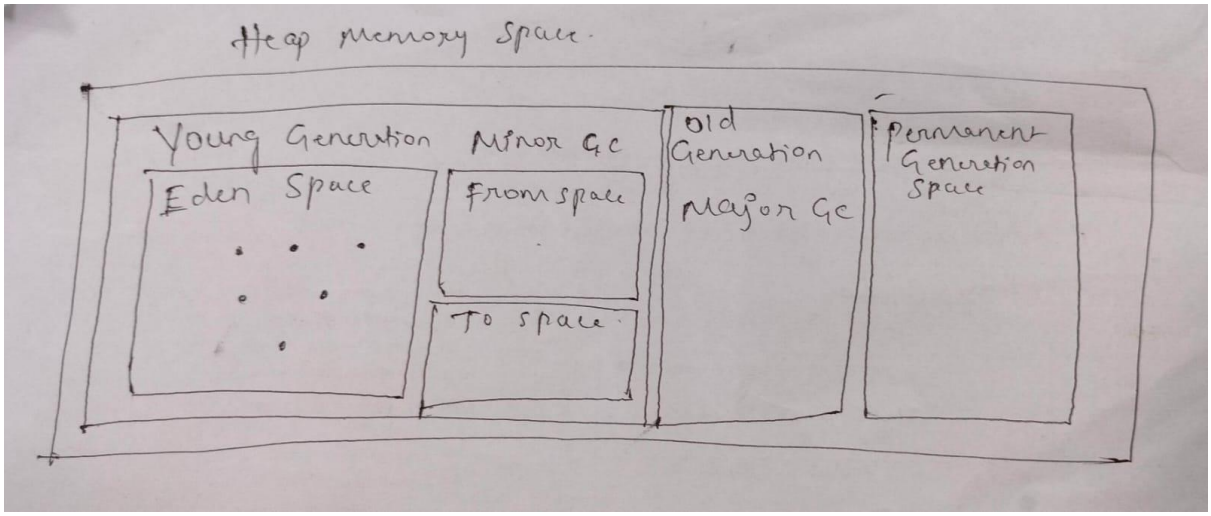
- Java Garbage collector implements generational garbage collection strategy that categorizes the objects by their age. As more and more objects are allocated, the list of objects grows leading to longer garbage collection time and infact several analysis have shown that most objects have very short lifespan. So leverage these findings, the heap memory is divided into 3 sections.

- **The Young Generation:** consists of newly created objects and is further subdivided into the eden space and the survivor space, all the new objects that start in the eden space, initial memory is allocated to them, if an object survives 1 GC (Garbage collection) cycle then they are moved to survivor space, when the objects are garbage collected from young generation, it is called minor garbage collection event. A minor GC is performed when the Eden space is filled with the objects either dead or alive. All the dead objects are removed from eden space and all alive objects are moved to the from space, Eden and To spaces are now empty, new objects are created and added to the eden. Some objects in the from space becomes dead as well, minor GC occurs and all dead objects are removed from eden and from space, all the alive objects are moved to To space, eden and From space are empty now. So at any time one of the survivor spaces is always empty.

- When the surviving objects reach a certain threshold of moving around the survivor spaces, they are moved to the old Generation

- **The Old Generation** is the second space out of three sections of heap memory space. It consists of objects that have remained in the survivor space for long time. When the objects are garbage collected from the older generation, it is known as major garbage collection event.

- If the object is still used after this point, the next GC Cycle will move them to the **permanent generation space**. Meta data of classes and methods are stored in the parament generation



The finalize( ) Method.
If an object is holding some non-Java resource such as a file handle or character font, then you might want to make sure these resources are freed before an object is destroyed. To handle such situations, Java provides a mechanism called *finalization.* By using finalization, you can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector.

Inside the **finalize( )** method, you will specify those actions that must be performed before an object is destroyed.

The **finalize( )** method has this general form:
protected void finalize( )
{
// finalization code here
}

Here, the keyword **protected** is a specifier that prevents access to **finalize( )** by code defined
outside its class.

It is important to understand that **finalize( )** is only called just prior to garbage collection.
It is not called when an object goes out-of-scope

Overloading of Methods:

- Defining two or more methods within the same class that share the **same name**, as long as their **parameter declarations are different**. When this is the case, the methods are said to be *overloaded,* and the process is referred to as *method overloading.*

- ***Method overloading is one of the ways that Java supports polymorphism.***

- overloaded methods must differ in the type and/or number of their parameters as the name of the method will be same.

  Example:

```
// Demonstrate method overloading.
class OverloadDemo {
void test() //method name is test here
{
System.out.println("No parameters");
}

// Overload test for one integer parameter.
void test(int a)  //method name is again test here
{
System.out.println("a: " + a);
}
// Overload test for two integer parameters.
void test(int a, int b) {
System.out.println("a and b: " + a + " " + b);
}

// overload test for a double parameter
double test(double a) {
System.out.println("double a: " + a);
return a*a;
}
}
```

As you can see, **test( )** is overloaded four times.

- The first version takes no parameters,
- the second takes one integer parameter,
- the third takes two integer parameters,
- and the fourth takes one **double** parameter.

Overloading Constructors

- In addition to overloading normal methods, you can also overload constructor methods.
- Constructor overloading in Java refers to the concept of having multiple constructors in a class with different parameter lists.
- When you define multiple constructors with different parameter lists (number or types of parameters), Java determines which constructor to call based on the arguments passed during object creation.

The key points to understand about constructor overloading are:

- Different parameter lists: Constructor overloading requires constructors to have different numbers or types of parameters. Java uses this information to distinguish between different constructors.

- Constructor naming: Constructors have the same name as the class, but they differ in their parameter lists.

- Object initialization: The purpose of constructor overloading is to provide multiple ways to initialize an object of the class, depending on the data provided during object creation.

Example:

```
/* Here, Box defines three constructors to initialize
the dimensions of a box various ways.
*/
class Box {
double width;
double height;
double depth;
// constructor used when all dimensions specified
Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
// constructor used when no dimensions specified
Box() {
width = -1; // use -1 to indicate
height = -1; // an uninitialized
depth = -1; // box
}
// constructor used when cube is created
Box(double len) {
width = height = depth = len;
}
// compute and return volume
double volume() {
return width * height * depth;
}
}
class OverloadCons {
public static void main(String args[]) {
// create boxes using the various constructors
Box mybox1 = new Box(10, 20, 15);
Box mybox2 = new Box();
Box mycube = new Box(7);
double vol;

// get volume of first box
vol = mybox1.volume();
System.out.println("Volume of mybox1 is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume of mybox2 is " + vol);
// get volume of cube
vol = mycube.volume();
System.out.println("Volume of mycube is " + vol);
}
}
```

The output produced by this program is shown here:

```
Volume of mybox1 is 3000.0
Volume of mybox2 is -1.0
Volume of mycube is 343.0
```

Using Objects as Parameters

- When you pass an object as a parameter, you are actually passing a reference to the object, not the object itself.
- Here's a simple example demonstrating how to use objects as parameters:
- Let's say we have a Person class representing a person with a name and age:

```
public class Person {
   private String name;
   private int age;
   public Person(String name, int age) {
```

```
        this.name = name;
        this.age = age;
    }
    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }
}
```

Now, let's create a method called printPersonDetails that takes a Person object as a parameter and prints their name and age:

```
public class ObjectParameterExample {
    public static void main(String[] args) {
        Person person = new Person("John Doe", 30);
        printPersonDetails(person);
    }

    public static void printPersonDetails(Person person) {
        System.out.println("Name: " + person.getName());
        System.out.println("Age: " + person.getAge());
    }
}
```

Output:
Name: John Doe
Age: 30

- In this example, we create a Person object named person with the name "John Doe" and age 30. Then, we call the printPersonDetails method and pass the person object as an argument.
- The printPersonDetails method takes a Person object as a parameter, and within the method, we use the object to access the name and age properties using the getter methods. The method then prints the name and age of the Person object.

3.14 Returning Objects

In Java, a method can return an object of user defined class.

```
class Test
{
      int a;
      Test(int i)
      {
            a = i;
      }

      Test incrByTen()
      {
            Test temp = new Test(a+10);
            return temp;
      }
}

class RetOb
{
  public static void main(String args[])
  {
    Test ob1 = new Test(2);
    Test ob2;

    ob2 = ob1.incrByTen();
    System.out.println("ob1.a: " + ob1.a);
    System.out.println("ob2.a: " + ob2.a);

    ob2 = ob2.incrByTen();
    System.out.println("ob2.a after second increase: " + ob2.a);
  }
}
```

Output:

```
ob1.a: 2
ob2.a: 12
ob2.a after second increase: 22
```

3.16 Introducing Access Control

Encapsulation feature of Java provides a safety measure viz. ***access control.*** Using ***access specifiers***, we can restrict the member variables of a class from outside manipulation. Java provides following access specifiers:

- public
- private
- protected

Along with above access specifiers, Java defines a *default access level.*

Some aspects of access control are related to inheritance and package (a collection of related classes). The *protected* specifier is applied only when inheritance is involved. So, we will now discuss about only *private* and *public.*

When a member of a class is modified by the public specifier, then that member can be accessed by any other code. When a member of a class is specified as private, then that member can only be accessed by other members of its class. When no access specifier is used, then by default the member of a class is public within its own package, but cannot be accessed outside of its package. Usually, you will want to restrict access to the data members of a class—allowing access only through methods. Also, there will be times when you will want to define methods that are private to a class. An access specifier precedes the rest of a member's type specification. For example,

```
public int x;
private char ch;
```

Consider a program given below –

```
class Test
{     int a;
      public int b;
      private int c;

      void setc(int i)
      {
            c = i;
      }

      int getc()
      {
            return c;
      }
}

class AccessTest
{
      public static void main(String args[])
      {
            Test ob = new Test();
            ob.a = 10;
            ob.b = 20;
            // ob.c = 100;        // inclusion of this line is Error!
            ob.setc(100);
            System.out.println("a, b, and c: " + ob.a + " " + ob.b + " "
                                                + ob.getc());
      }
}
```

The error is because, we are trying to access the private variables "c" of class "Test" in another class called "AccessTest".

Access control in Java refers to the mechanism that restricts access to classes, methods, and variables. Java provides four access modifiers to define different levels of access control, which help encapsulate data and prevent unintended usage of code components. The four access modifiers are:

1. private:

- **Scope**: The field, method, or constructor is only accessible within the same class.
- **Use Case**: It is used to encapsulate details of a class and hide its implementation from other classes.

private int age; // Only accessible within this class

2. default (no modifier):

- **Scope**: Accessible within the same package but not outside the package. If no access modifier is specified, this is the default access level.
- **Use Case**: Used when you want to restrict access to the package level.

int salary; // Accessible only within the same package

3. protected:

- **Scope**: Accessible within the same package and by subclasses (even if they are in different packages).
- **Use Case**: Provides access to subclasses, supporting inheritance while keeping some level of restriction.

```
protected String name;  // Accessible within the package and subclasses
```

4. public:

- **Scope**: Accessible from anywhere in the program, i.e., from any class or package.
- **Use Case**: Used for classes, methods, or variables that should be accessible by all other classes.

```
public void display() {  // Accessible from anywhere in the program
   System.out.println("Displaying details");
}
```

Summary of Access Levels:

Access Modifier	Class	Package	Subclass	World (outside package)
private	Yes	No	No	No
Default (none)	Yes	Yes	No	No
protected	Yes	Yes	Yes	No
public	Yes	Yes	Yes	Yes

Access control is essential for data encapsulation, protecting internal data and methods, and enforcing a modular structure in Java.

3.17 Understanding *static*

When a member is declared **static**, it can be accessed before any objects of its class are created, and without reference to any object. Instance variables declared as **static** are global variables. When objects of its class are declared, no copy of a **static** variable is made. Instead, all instances of the class share the same **static** variable.

Methods declared as **static** have several restrictions:

- They can only call other **static** methods.
- They must only access **static** data.
- They cannot refer to **this** or **super** in any way.

If you need to do computation in order to initialize your **static** variables, you can declare a **static** block that gets executed exactly once, when the class is first loaded.

```
class UseStatic
{
      static int a = 3;
      static int b;

      static void meth(int x)          //static method
      {
            System.out.println("x = " + x);
            System.out.println("a = " + a);
            System.out.println("b = " + b);
      }

      static     //static block
      {
            System.out.println("Static block initialized.");
            b = a * 4;
      }

      public static void main(String args[])
      {
            meth(42);
      }
}
```

Output:

Static block initialized.
x = 42
a = 3
b = 12

Outside of the class in which they are defined, **static** methods and variables can be used independently of any object. To do so, you need only specify the name of their class followed by the dot operator. The general form is –

classname.method();

FINAL

3.26 Using *final*

The keyword *final* can be used in three situations in Java:

- To create the equivalent of a named constant.
- To prevent method overriding
- To prevent Inheritance

SITUATION 1:

To create the equivalent of a named constant: A variable can be declared as final. Doing so prevents its contents from being modified. This means that you must initialize a final variable when it is declared. For example:

```
final int FILE_NEW = 1;
final int FILE_OPEN = 2;
final int FILE_SAVE = 3;
final int FILE_SAVEAS = 4;
final int FILE_QUIT = 5;
```

It is a common coding convention to choose all uppercase identifiers for final variables. Variables declared as final do not occupy memory on a per-instance basis. Thus, a final variable is essentially a constant.

SITUATION 2

To prevent method overriding: Sometimes, we do not want a superclass method to be overridden in the subclass. Instead, the same superclass method definition has to be used by every subclass. In such situation, we can prefix a method with the keyword *final* as shown below –

```
class A
{
      final void meth()
      {
            System.out.println("This is a final method.");
      }
}
class B extends A
{
      void meth()  // ERROR! Can't override.
      {
            System.out.println("Illegal!");
      }
}
```

SITUATION 3

To prevent Inheritance: As we have discussed earlier, the subclass is treated as a specialized class and superclass is most generalized class. During multi-level inheritance, the bottom most class will be with all the features of real-time and hence it should not be inherited further. In such situations, we can prevent a particular class from inheriting further, using the keyword *final*. For example –

```
final class A
{
      // ...
}
class B extends A      // ERROR! Can't subclass A
{
      // ...
}
```

Command Line Arguments in Java

- **Java command-line argument** is an argument i.e. passed at the time of running the Java program.
- In Java, the command line arguments passed from the console can be received in the Java program and they can be used as input.
- The users can pass the arguments during the execution bypassing the command-line arguments inside the main() method.

```
// Java Program to Check for Command Line Arguments
// Class
class CLA {
    // Main driver method
    public static void main(String[] args)
    {
            // Checking if length of args array is greater than 0
            if (args.length > 0) {
```

```
                // Print statements
                System.out.println("The command line"+ " arguments are:");
                // Iterating the args array
                // using for each loop
                for (String val : args) {
                        // Printing command line arguments
                        System.out.println(val);
            }
            else
                // Print statements
            System.out.println("No command line "+ "arguments found.");
        }
    }
```

OUTPUT