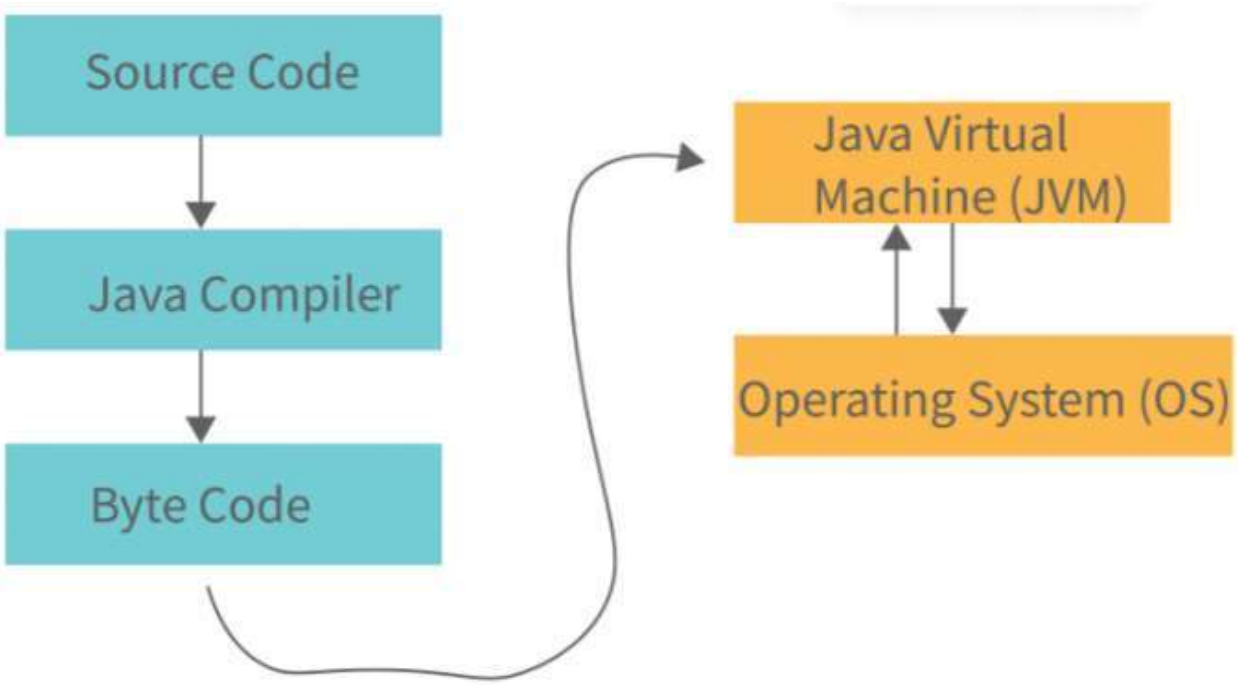
Unit 1

Basics of Programming in Java 7hrs

1.1 Java Architecture, Class Paths, Simple Program

Java Architecture

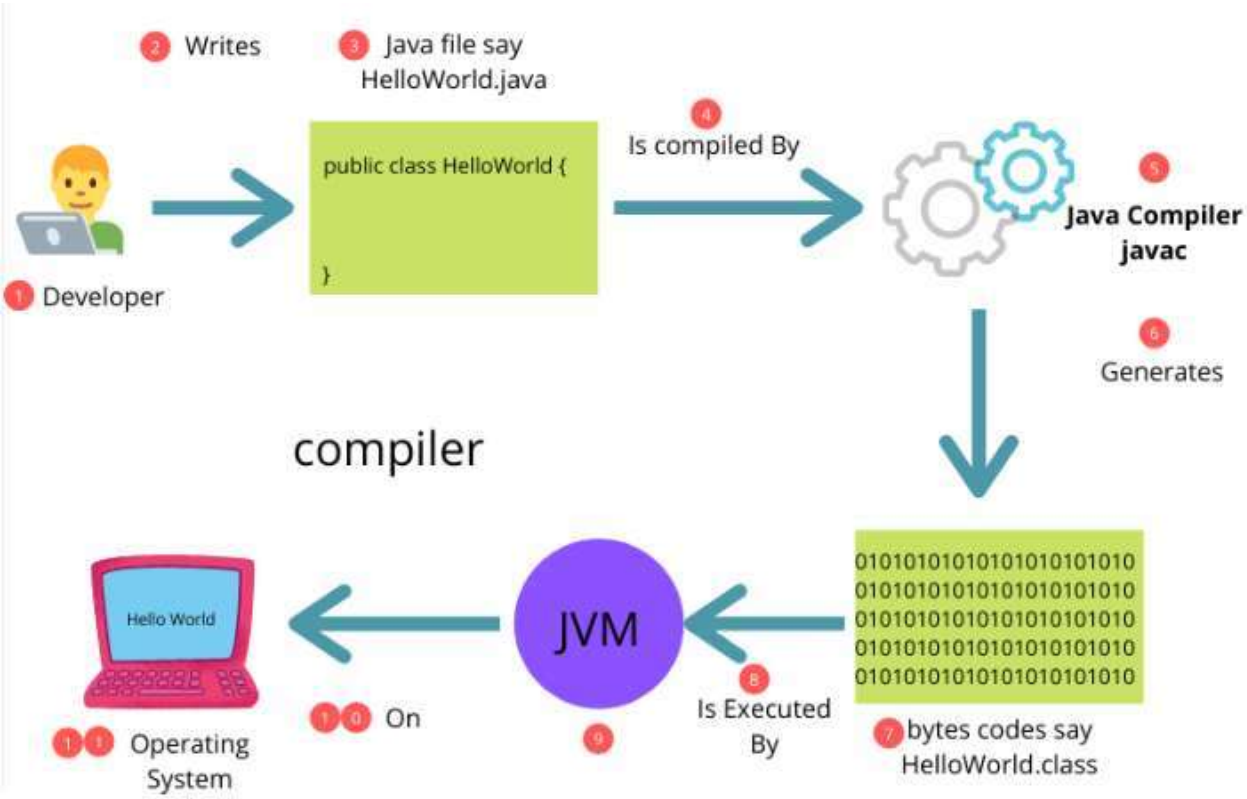
The JVM, JRE, and JDK are all components of Java Architecture. It combines the interpretation and compilation processes. It enumerates all of the steps involved in writing a Java program. The Java Architecture describes each step in the compilation and execution of a program. The below-mentioned points and diagram will simply illustrate Java architecture:



The four steps on the writing and running the java program:

1. There is a compilation and interpretation process in Java.
2. After the JAVA code is written, the JAVA compiler comes into the picture that converts this code into byte code that can be understood by the machine.
3. After the creation of bytecode JAVA virtual machine (JVM) converts it to the machine code, i.e. (.class file)
4. And finally, the machine runs that machine code.

Let's see the detail working of the java compilation process



Implementation of Simple java programs:

- **Creating a java program (source code):** Type the java code using editor such as notepad, netbeans, eclipse, and intellij etc. and then save it on a secondary storage device with .java extension.
- **Compiling a java program into bytecodes:** Use javac command to compile a program. The compiler converts the java source code into bytecodes. The bytecodes are platform independent and hence are portable. The bytecodes are .class files.
- **Loading a program into memory:** The class loader loads the .class files from disk to primary memory before it executes.
- **Bytecode verification and execution:** The bytecode verifier examines the bytecodes and then JVM interprets the bytecode and generates machine code which runs in specific machine.

Example:

```
public class Simple{
      public static void main(String args[]) {
            System.out.println("hello world");
      }
}
```

- To run this program, we need to save it as Simple.java and compile it as below; **javac Simple.java**
- It generates Simple.class (bytecode) file. To interpret this we use, **java Simple**
- Finally, this program gives the output

hello world

Another Simple Program:

```
public class SimpleProgram {
      public static void main(String[] args) {
            System.out.println("Hello Learners, Welcome to Java Programming");
      }
}
```

1.2Class, Object, Constructors

Classes

A class is a collection of objects of similar type. For example manager, peon, secretary, clerk are member of the class employee and class vehicle includes objects car, bus, etc. It defines a data type, much like a **struct** in C programming language and built in data type (int char, float etc). It specifies what data and functions will be included in objects of that class. Defining class does not create attributes and behaviors. Person Class may have attributes: Name, Age, Sex etc. and behaviors: Speak(), Listen(), Walk(), etc.

Vehicle Class may have attributes: name, model, color, height etc. and behaviors: start(), stop(), accelerate() etc.

When class is defined, objects are created as
<classname> <objectname> ;

If employee has been defined as a class, then the statement

```
Employee manager;
```

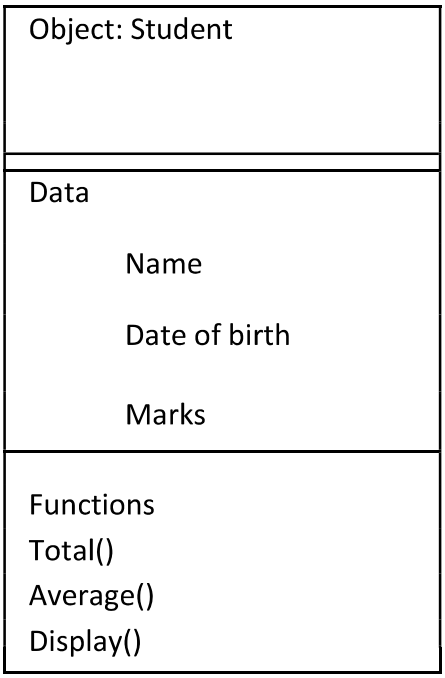
Will create an object manager belonging to the class employee.
Each class describes a possibly infinite set of individual objects, each object is said to be an instance of its class and each instance of the class has its own value for each attribute but shares the attribute name and operations with other instances of the class.

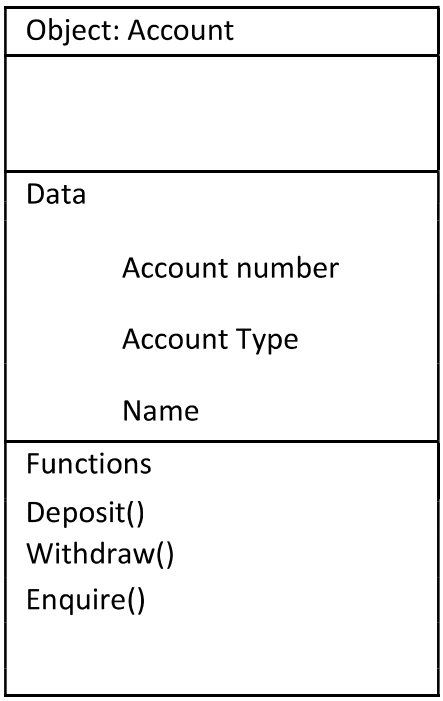
The following point gives the idea of class:

- A class is a template that unites data and operations.
- A class is an abstraction of the real-world entities with similar properties.
- Ideally, the class is an implementation of abstract data type.

Objects

Objects are the entities in an object-oriented system through which we perceive the world around us. We naturally see our environment as being composed of things which have recognizable identities & behavior. The entities are then represented as objects in the program. They may represent a person, a place, a bank account, or any item that the program must handle. For example Automobiles are objects as they have size, weight, color etc as attributes (ie data) and starting, pressing the brake, turning the wheel, pressing accelerator pedal etc as operation (that is functions).





Some of the example of objects are as follows:

Physical Objects:

- Automobiles in traffic flow
- Countries in Economic model
- Air craft in traffic control

system. Computer user

environment objects.

- Window, menus,

icons etc Data storage

constructs.

- Stacks,

Trees etc Human

entities:

- employees, student,

teacher etc. Geometric objects:

- point, line, Triangle etc.

Objects mainly serve the following purposes:

- Understanding the real world and a practical base for designers
- Decomposition of a problem into objects depends on the nature of problem.

Constructor

Constructor is a block of codes similar to the method. It is called when an instance of the class is created. At the time of calling constructor, memory for the object is allocated in the memory. Every time an object is created using the new keyword, at least one constructor is called.

It calls a default constructor if there is no constructor available in the class. In such case, Java compiler provides a default constructor by default.

There are two types of constructors in Java:

- no-arg constructor, and
- parameterized constructor.

Rules for creating constructor

There are two rules defined for the constructor.

- Constructor name must be the same as its class name
- A Constructor must have no explicit return type
- A constructor cannot be abstract, static, final, and synchronized

Example,

Creating class and Constructors

```
public class MyClass {
	private Long id;
	private String firstName;
	private String lastName;
	//creating default constructor
	public MyClass() {
	}

	//creating parameterized constructor
	public MyClass(Long id, String firstName, String lastName) {
		this.id = id;
		this.firstName = firstName;
		this.lastName = lastName;
	}

	//creating getter/setter methods
	public Long getId() {
		return id;
	}
	public void setId(Long id) {
		this.id = id;
	}
	public String getFirstName() {
		return firstName;
	}
	public void setFirstName(String firstName) {
		this.firstName = firstName;
	}
	public String getLastName() {
		return lastName;
	}
	public void setLastName(String lastName) {
		this.lastName = lastName;
	}
}
```

Creating object

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

```
        //to create an object of Default constructor of MyClass
        MyClass obj = new MyClass();

        //to create an object of parameterized constructor
        MyClass obj2= new MyClass(1L, "Sunil", "Bist");

        //call member variable if not private
        //Long id = obj.id;

        //call methods
        String firstName = obj.getFirstName();

        //call method to get parameterized constructor
        String firstName2=obj2.getFirstName();
}
}
```

1.3 Package and Data Types

Packages: The package is both a naming and a visibility control mechanism. We can define classes inside a package that are not accessible by code outside that package. Packages are the Java's way of grouping a variety of classes together. The main reason for using packages is to guarantee the uniqueness of class names.

Defining a Package:

To create a package, we use the '**package** command as the first statement in a Java source file. Any classes declared within that file will belong to the specified package. If we omit the **package** statement, the class names are put into the default package, which has no name.
The general form of the **package** statement:

package **packagename;**

e.g package nast;

Importing Packages:

Java includes the **import** statement to bring certain classes, or entire packages, into visibility. The general form of the **import** statement:

```
import packagename.*;
or
import packagename.ClassName;
```

example,

```
package mypack;
public class Balance{
        String name;
```

```
	double bal;
	public Balance(String n, double b){
		name = n;
		bal = b;
	}
	public void show(){
		System.out.println(name + ":" + bal);
	}
}
```

//import statement goes here

```
import mypack.*;
class TestBalance{
	public static void main(String args[]){
		Balance test = new Balance("Sunil", 1000.3);
		test.show();
	}
}
```

Creating subpackage:

To create a subpackage the syntax is:

```
	package packagename.subpackagename;
```

Data Types

Data types are divided into two groups:

Primitive data types - includes byte, short, int, long, float, double, boolean and char

Non-primitive data types - such as String, Arrays and Classes etc.

Primitive Data Types

A primitive data type specifies the size and type of variable values, and it has no additional methods.

There are eight primitive data types in Java:

Data Type	Size	Description
Byte	1 byte	Stores whole numbers from -128 to 127
Short	2 bytes	Stores whole numbers from -32,768 to 32,767
Int	4 bytes	Stores whole numbers from -2,147,483,648 to 2,147,483,647
Long	8 bytes	Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
Float	4 bytes	Stores fractional numbers. Sufficient for storing 6 to 7 decimal digits

Double	8 bytes	Stores fractional numbers. Sufficient for storing 15 decimal digits
Boolean	1 bit	Stores true or false values
Char	2 bytes	Stores a single character/letter or ASCII values

- **Integers:** Java defines four integer types; byte, short, int and long. Byte is the smallest singed integer type of 8-bit, short is a signed 16-bit type and is least used, int is signed 32-bit type and most commonly used and long is a signed 64-bit type.
- **Floating-point types:** Floating point numbers, also known as real numbers are used when evaluating expression that require fractional portion. It includes „float" and „double".
- **Characters:** The datatype that is used to store characters is „char" which is 16-bittype.
- **Boolean:** Boolean is simple type & it is represented by datatype "Boolean". It contains only two possible values true or false.

Non-Primitive Data Types

Non-primitive data types are called reference types because they refer to objects.

The main difference between primitive and non-primitive data types are:

- Primitive types are predefined (already defined) in Java. Non-primitive types are created by the programmer and is not defined by Java (except for String).
- Non-primitive types can be used to call methods to perform certain operations, while primitive types cannot.
- A primitive type has always a value, while non-primitive types can be null.
- A primitive type starts with a lowercase letter, while non-primitive types start with an uppercase letter.

Variables:

Variables are the name of memory space used to store values during program execution. A variable is defined by a combination of a data type, an identifier (variable name) and an optional initializer that contains value.

It"s syntax is: datatype variable=value;

To declare more than one variable of the specified type, we can use a comma.

Example:

```
//declaration of variable a
int a;

//initialization of variable b & c with values 2 and 3 respectively
int b=2,c=3;
```

Dynamic initialization of variable: Dynamic initialization of variable means that we can initialize the variables using expression. The value of the dynamic variable changes during compile time if the values in the expression changes.

WAP that demonstrate the concept of dynamic initialization of variables

```
class MainClass {
	public static void main(String args[]) {
		double a = 3.0, b = 4.0;
		// c is dynamically initialized
		double c = Math.sqrt(a * a + b * b);
		System.out.println("Hypotenuse is " + c);
	}
}
```

This program consists of three local variables a, b and c. The first two a and b are initialized by constants. However, c is initialized dynamically to the result of the expression calculated by the **sqrt** function defined inside **Math** class.

Java Naming Conventions

Naming conventions make programs more understandable by making them easier to read. They can also give information about the function of the identifier-for example, whether it's a constant, package, or class-which can be helpful in understanding the code.

Identifier Type	Rules for Naming	Examples
Packages	The prefix of a unique package name is always written in all-lowercase ASCII letters and should be one of the top-level domain names, currently com, edu, gov, mil, net, org, or one of the English two-letter codes identifying countries as specified in ISO Standard 3166, 1981. Subsequent components of the package name vary according to an organization's own internal naming conventions. Such conventions might specify that certain directory name components be division, department, project, machine, or login names.	com.sun.eng com.apple.quicktime.v2 edu.cmu.cs.bovik.cheese
Classes	Class names should be nouns, in mixed case with the first letter of each internal word capitalized. Try to keep your class names simple and descriptive. Use whole words-avoid acronyms and abbreviations (unless the abbreviation is much more widely used than the long form, such as URL or HTML).	class Raster; class ImageSprite;

Identifier Type	Rules for Naming	Examples
Interfaces	Interface names should be capitalized like class names.	interface RasterDelegate; interface Storing;

Methods	Methods should be verbs, in mixed case with the first letter lowercase, with the first letter of each internal word capitalized.	run();
		runFast();
		getBackground();

Variables	Except for variables, all instance, class, and class constants are in mixed case with a lowercase first letter. Internal words start with capital letters. Variable names should not start with underscore _ or dollar sign $ characters, even though both are allowed. Variable names should be short yet meaningful. The choice of a variable name should be mnemonic- that is, designed to indicate to the casual observer the intent of its use. One-character variable names should be avoided except for temporary "throwaway" variables. Common names for temporary variables are `i`, `j`, `k`, `m`, and `n` for integers; `c`, `d`, and `e` for characters.	int i; char c; float myWidth;

Constants	The names of variables declared class constants and of ANSI constants should be all uppercase with words separated by underscores ("_"). (ANSI constants should be avoided, for ease of debugging.)	static final int MIN_WIDTH = 4;
		static final int MAX_WIDTH = 999;
		static final int GET_THE_CPU = 1;

1.4Conditional Statements

The control statements are used to control the flow of execution of the program. This execution order depends on the supplied data values and the conditional logic of the program. Java's program control statements can be put into the following categories: ***Selection***, ***Iteration***, and ***Branching*** or ***Jump*** Statements.

Java's Selection Statements: Java supports two selection statements: ***if and switch***. These statements allow you to control the flow of your program's execution based upon conditions

known only during run time. Java selection statements are also known as control or branching or decision-making statements. Java supports following selection statements:

1. If statement
2. Switch statement

1. The if statement: The if statement is a powerful decision-making statement and it is used to control the flow of execution of statements. It is the two-way statement and is used in conjunction with a test condition. The general form of if statement is:

Here, if is a keyword and the condition following the keyword if is always enclosed within the pair of parentheses. The value of test condition may be true (non zero) or false (zero). The if statement allow the computer to test the condition first and depending on whether the value of the condition is true or false, it transfers the control to a particular statement. The if statement has two paths: one for the true condition and other for false condition. The if statements may be of different forms depending upon the complexity of conditions to be tested.

1. Simple if statement
2. if-else statement
3. Nested if-else statement
4. else-if ladder statement

```
 SYNTAX:
SYNTAX:
if(condition){
    //This block of
    //Code executed
    //if the given
    //condition is true
}
-----
    //This block of
    //Code executed
    //if the given
    //condition is false
}
```

```
SYNTAX:
if(condition){
    //This block of
    //Code executed
    //if the first given
    //condition is true
}
else if(condition){
    //This block of
    //Code executed
    //if the second given
    //condition is true
}
else if(condition){
    //This block of
    //Code executed
    //if the third given
    //condition is true
}
.
.
.
else{
    //This block of
    //Code executed
    //if the given all
    //condition is false
}
```

Nested if: A nested if is an if statement that is (placed or put inside another if or else)the target of another if or else. Nested ifs are very common in programming. When you nest ifs, the main thing to remember is that an else statement always refers to the nearest if statement that is within the same block as the else and that is not already associated with an else. Here is an example:

```
if(i == 10) {
   if(j < 20)
        a = b;
   if(k > 100)
        c = d; // this if is
   else
        a = c; // associated with this else
}
else
a = d; // this else refers to if(i == 10)
```

Switch Case Statement

The switch case statement, also called a case statement is a multi-way branch with several choices. It is easier to implement than a series of if/else statements. The switch statement begins with a keyword switch followed by an expression that evaluates to a no long integral value. Following the controlling expression is a code block that contains zero or more labeled cases. Each label must equate to an integer constant and each must be unique. When the switch statement executes, it compares the value of the controlling expression to the values of each case label. The program will select the value of the case label that equals the value of the controlling expression and branch down that path to the end of the code block. If none of the case label values match, then none of the codes within the switch statement code block will be executed. Java includes a default label to use in cases where there are no matches. We can have a nested switch within a case block of an outer switch.

Its general form is as follows:

```
SYNTAX:
switch(expression){
    case constant1:
                    statement 1;
                    ..
                    ..
                    break;
    case constant2:
                    statement 1;
                    ..
                    ..
                    break;
    .
    .
    .
    case constantn:
                    statement 1;
                    ..
                    ..
                    break;
    default:
            statement 1;
            ..
            ..
            break;

}
```

When executing a switch statement, the program falls through to the next case. Therefore, if you want to exit in the middle of the switch statement code block, you must insert a break statement, which causes the program to continue executing after the current code block.

Repetition Statements

while loop statements: This is a looping or repeating statement. It executes a block of code or a statement till the given condition is true. The expression must be evaluated to a Boolean value. It continues testing the condition and executes the block of code. When the expression results to false control comes out of loop. The while loop check the condition at the entry level so it is also called entry controlled loop.

```
SYNTAX:
Initialization
while(condition){
    //while loop block
    Increment/Decrement
}
```

do-while loop statements: This is another looping statement that tests the given condition at the end so you can say that the do-while looping statement is a exit control looping statement. First the **do** block statements are executed then the condition given in **while** statement is checked. So in this case, even the condition is false in the first attempt, do block of code is executed at least once.

```
SYNTAX:
Initialization
do{
    ..........
    //while loop block
    Increment/Decrement
}while(condition);
```

for loop statements: This is also a loop statement that provides a compact way to iterate over a range of values. From a user point of view, this is reliable because it executes the statements within this block repeatedly till the specified conditions are true.

WHERE **initialization:** The loop is started with the value specified. **Condition:** It evaluates to either 'true' or 'false'. If it is false then the loop is terminated. **Increment or decrement:** After each iteration, value increments or decrements.

```
SYNTAX:
for(initialization; condition; increment/decrement){
    //This block of code is executed deupnding upon
    //the condition provided by the programmer
}
```

The foreach Version of the for Loop: Beginning with JDK 5, a second form of **for** was defined that implements a "for-each" style loop. As you may know, contemporary language theory has embraced the for-each concept, and it is quickly becoming a standard feature that programmers have come to expect. A **foreach** style loop is designed to cycle through a collection of objects, such as an array, in strictly sequential fashion, from start to finish. Unlike some languages, such as C#, that implement a for-each loop by using the keyword **foreach**, Java adds the for-each capability by enhancing the **for** statement. The advantage of this approach is that no new keyword is required, and no preexisting code is broken. The for-each style of **for** is also referred to as the *enhanced* **for** loop. The general form of the for-each version of the **for** is shown here:

```
SYNTAX:
for(data type variableName: arrayVariable){
    //This block of code is executed until
    //each of the items of an arrayVariable
    //are not finished.
}
```

Here, *data type* specifies the type and *variableName* specifies the name of an *iteration variable* that will receive the elements from a collection, one at a time, from beginning to end.

Nested Loops: Like all other programming languages, Java allows loops to be nested. That is, one loop may be inside another. For example, here is a program that nests **for** loops:

```
for(int i=0; i<10; i++) {
    for(int j=i; j<10; j++){
        System.out.print("*");
    }
    System.out.println();
}
```

Branching Statements

Break statements: The break statement is a branching statement that contains two forms: labeled and unlabeled. The break statement is used for breaking the execution of a loop (while, do-while and for). It also terminates the switch statements.

```
SYNTAX:
break; // breaks the innermost loop or switch statement.
break label; // breaks the outermost loop in a series of nested loops.
```

For example,

```
class BreakExample{
    public static void main(String [] args){
        for(int i=0; i<100; i++) {
            if(i == 10) break; // terminate loop if i is 10
            System.out.println("i: " + i);
        }
    System.out.println("Loop complete.");
    }
}
```

Continue statements: This is a branching statement that is used in the looping statements (while, do-while and for) to skip the current iteration of the loop and resume the next iteration.
Here in this example the program will print 0, 1, 2, 3, 4, 5, …., 9, and skip 10 and print 11, 12, 13, 14, …., continue all the other numbers up to 19.

Return statements: It is a special branching statement that transfer the control to the caller of the method. This statement is used to return a value to the caller method and terminates execution of

```
class ReturnExample{
    public static void main(String[] args){
    RExample e= new RExample();
        System.out.println("HELLO WORLD Before Return");
        System.out.println(getMessage());
    }
    String getMessage()
    {
        return "Hello WORLD After Return";
    }
}
```

method. This has two forms: one that returns a value and the other that cannot return. The returned value type must match the return type of method.

```
SYNTAX:
 return;
 return values;
```

For example,

```
class ContinueExample{
        public static void main(String args[]){
            for(int a=0; a<20; a++){
                if(a==10){
                    continue;
                }
                System.out.println(a + "\t");
            }
        }
}
```

1.5 Access Modifiers

Access modifiers help to restrict the scope of a class, constructor, variable, method, or data member. It provides security, accessibility, etc. to the user depending upon the access modifier used with the element.

There are four types of access modifiers in java

- Default – No keyword required
- Private
- Protected
- Public

Default Access Modifier

When no access modifier is specified for a class, method, or data member it is said to be having the default access modifier by default. The data members, classes, or methods that are not declared using any access modifiers i.e. having default access modifiers are accessible only within the same package.

Private Access Modifier

The private access modifier is specified using the keyword private. The methods or data members declared as private are accessible only within the class in which they are declared.

- Any other class of the same package will not be able to access these members.
- Top-level classes or interfaces cannot be declared as private because
    - private means “only visible within the enclosing class”.
    - protected means “only visible within the enclosing class and any subclasses”

Hence these modifiers in terms of application to classes, apply only to nested classes and not on top-level classes

Protected Access Modifier

The protected access modifier is specified using the keyword protected. The methods or data members declared as protected are accessible within the same package or subclasses in different packages.

Public Access modifier

The public access modifier is specified using the keyword public.

- The public access modifier has the widest scope among all other access modifiers.
- Classes, methods, or data members that are declared as public are accessible from everywhere in the program. There is no restriction on the scope of public data members.

Important Points:

- If other programmers use your class, try to use the most restrictive access level that makes sense for a particular member. Use private unless you have a good reason not to.
- Avoid public fields except for constants.

Access Modifiers

Modifier	Class	Package	Subclass	Global
Public	✓	✓	✓	✓
Protected	✓	✓	✓	✗
Default	✓	✓	✗	✗
Private	✓	✗	✗	✗

1.6Exception Handling

Error

It is common to make mistakes while developing as well as typing a program. A mistake might lead to an error causing the program to produce unexpected results. An error may produce an incorrect output or may terminate the execution of the program abruptly of even may cause the system to crash. It is therefore important to detect and manage properly all the possible error conditions in the program so that the program will not terminate or crash during execution.

Types of errors

Errors are broadly classified into two categories:

- Compile time errors
- Run-time errors

Compile time errors

Those errors that encounter at compilation time are known as compile time errors. All syntax errors will be detected and displayed by the java compiler and therefore these errors are known as compile time errors. Whenever the compiler displays an error, it will not create the .class file. It is therefore necessary that all the errors should be fixed before successfully compile and run the program. Most of the compile time errors are due to typing mistakes. The most common compile time errors are:

- Missing semicolon
- Missing (or mismatch of) brackets in classes and methods
- Misspelling of identifiers and keywords
- Missing double quotes in string
- Use of undeclared variables
- Incompatible types in assignment/initialization
- Bad reference to objects
- And so on.

Run time errors

All the errors that encounter at run time are known as run time errors. During runtime errors a program may compile successfully creating the .class file but may not run properly. Such programs may produce wrong results due to wrong logic or may terminate due to errors. Most common runtime errors are:

- Dividing an integer by zero.
- Accessing an element that is out of the bounds of an array.
- Trying to store a value into an array of an incompatible class or type.
- Trying to cast an instance of a class to one of its subclasses

- Passing a parameter that is not in a valid range or value for a method
- Trying to illegally change the state of a thread
- Attempting to use a negative size for an array
- Converting invalid string to a number
- Accessing a character that is out of bounds of a string
- And so on.

Exceptions

An exception is a condition that is caused by a runtime error in the program. When the Java interpreter encounters an error such as dividing an integer by zero, it creates an exception object and throws (i.e., inform us that an error has occurred).

If the exception object is not caught and handled properly, the interpreter will display an error message and will terminate the program. To continue the program execution, we should try to catch the exception object thrown by the error condition and then display an appropriate message for taking corrective actions. This task is known as exception handling. The purpose of exception handling is to provide a means to detect and report an exceptional circumstance so that appropriate action can be taken.

Java exception handling is managed by five keywords: **try, catch, throw, throws and finally**

Program statements that may generate exception are contained within **try** block. If an exception occurs within the try block, it is thrown. Those statements that are used to catch those exceptions are kept in **catch** block and handled in some relational manner. System generated exceptions are automatically thrown by the java runtime system. To manually throw an exception, use the keyword **throw**. Any exception that is thrown out of a method must be specified as such by a **throws** clause. Any code that absolutely must be executed before a method return is put in a **finally** block.

The general syntax of exception handling block is:

```
try{
      // block of code to monitor for errors
}
catch( ExceptionType1 exOb){
      //exception handler for exception type1
}catch( ExceptionType2 exOb){
      //exception handler for exception type2
}finally{
      //block of code to be executed before try block ends
}
```

Here, exception type is the type of exception that has occurred.

Exception type

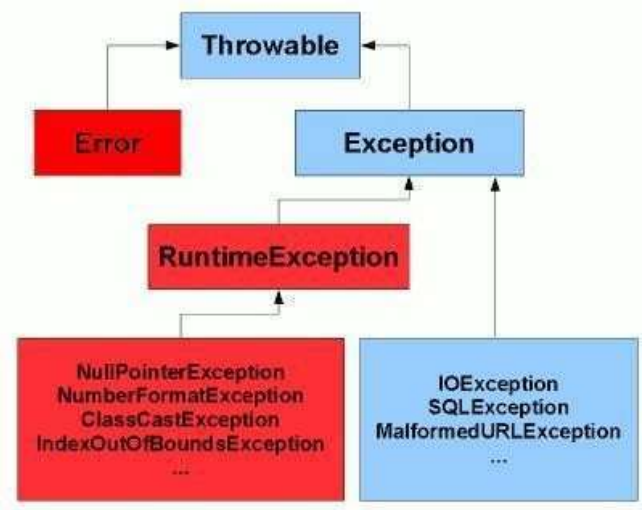


Fig: exception hierarchy

All exception types are subclasses of the built-in class **Throwable.** Thus Throwable is at the top of the exception class hierarchy. Immediately below Throwable are two subclasses that partition exceptions into two distinct branches: error and exception. There is an important subclass of exception, called **RuntimeException**. The other branch called error, which define exceptions that are not expected to be caught under normal circumstances by the program.

Using try and catch

The code that may generate exception is enclosed within try block. Immediately following try block, include a catch clause that specifies the exception type that you wish to catch.

Program which includes try and catch block that processes the ArithmeticException generated by the division-by-zero error.

```
public class ArithmeticExceptionDemo{
      public static void main(String args[])    {
            int a,b;
            try         {
                  b = 0;
                  a = 12/b;
                  System.out.println(" Inside try block");
                  // this will not be printed
            }
            catch(ArithmeticException e){ //catch division by zero error

                  System.out.println("Division by zero.");
            }
```

```
            System.out.println("Outside try and catch block");
      }
}
```

Output:

Division by zero.

Outside try and catch block

Multiple catch clauses

In some cases, more than one exception could be raised by a single piece of code. To handle this type of situation, we can specify two or more **catch** clauses, each catching a different type of exception. When an exception is thrown, each catch statement is inspected in order, and the first one whose type matches that of the exception is executed. After one **catch** statement executes, the others are bypassed and execution continues after the **try/catch** block.

Syntax;

```
try{
      Statements;
}catch(ExceptionType1 e){
      Statements;
}catch(ExceptionType2 e){
      Statements;
}
…….
……..
catch(ExceptionTypeN e){
      Statement;
}
```

Write a program that demonstrate the concept of multiple catch blocks

```
public class MultipleCatchDemo{
      public static void main(String args[])    {
            int a[] = {5,10}; int b = 5;
            try{
                  int x = a[2]/(b-a[1]);
            }catch(ArithmeticException e){
                  System.out.println("Division by zero");
            }catch(ArrayIndexOutOfBoundsException e){
                  System.out.println("Array index error");
            }catch(ArrayStoreException e){
                  System.out.println("Wrong data type");
            }
```

```
            int y = a[1]/a[0];

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

Output:

Array index error Y=2

finally statement

Java supports another statement known as **finally** statement that can be used to handle an exception that is not caught by any of the previous **catch** statements. **finally** block can be used to handle any exception generated within a **try** block. It may be added immediately after the **try** block or after the last **catch** block.

```
try{


}finally{


}
```

```
Or
try{
      ……..
      ……..
      ……..
      ……..
      ……..
      ……..
}catch(……..){
      ……..
      ……..
}
catch(      ){


}finally{
      ……..
      ……..
      ……..
      ……..
}
```

throw

When we would like to throw our own exceptions, we can do this by using the keyword throw. The general syntax of throw is as:

throw new ThrowableInstance();

Here, ThrowableInstance must be an object of type Throwable or a subclass of throwable. The flow of execution stops immediately after the throw statement, any subsequent statements are not executed.

Program that demonstrate the concept of throw

```
public class MyException extends Exception{
	public MyException(String message){
		super(message);
	}
}
```

=>Save MyException.java

```
public class TestMyException{
	public static void main(String args[])
	{
		int x = 5, y = 1000;
		try	{
			float z = (float) x / (float) y;
			if(z < 0.01) {
				throw new MyException("Number is too small");
			}
		}catch(MyException e){
			System.out.println("Caught	my	Exception");
			System.out.println(e.getMessage());
		}finally{
			System.out.println("I am always here");
		}
	}
}
```

TestMyException.java

Output

Caught my Exception Number is too small

I am always here

throws clause

The throws keyword appears at the end of a method's signature. A **throws** clause lists the types of exceptions that a method might throw.

The general form of a method declaration that includes a **throws** clause:

```
type method-name(parameter-list) throws exception-list
```

```
{
	// body of method
}
```

Example showing the throws clause:

```
class ThrowsDemo {
	static void throwOne() throws IllegalAccessException {
		System.out.println("Inside throwOne.");
		throw new IllegalAccessException("demo");
	}
	public static void main(String args[]) {
		try {
			throwOne();
		} catch (IllegalAccessException e) {
			System.out.println("Caught " + e);
		}
	}
}
```

Output:

```
Inside throwOne
caught java.lang.IllegalAccessException: demo
```

1.7Java Collections

Any group of individual objects that are represented as a single unit is known as a Java Collection of Objects. In Java, a separate framework named the "Collection Framework" has been defined in JDK 1.2 which holds all the Java Collection Classes and Interface in it.

In Java, the Collection interface (java.util.Collection) and Map interface (java.util.Map) are the two main "root" interfaces of Java collection classes.

A framework is a set of classes and interfaces which provide a ready-made architecture.

Before the Collection Framework (or before JDK 1.2) was introduced, the standard methods for grouping Java objects (or collections) were Arrays or Vectors, or Hashtables. All of these collections had no common interface. Therefore, though the main aim of all the collections is the same, the implementation of all these collections was defined independently and had no correlation among them. And also, it is very difficult for the users to remember all the different methods, syntax, and constructors present in every collection class.

```
// Java program to demonstrate why collection framework was needed
import java.util.Hashtable;
import java.util.Vector;
```

```
class CollectionDemo {

	public static void main(String[] args)
	{
		// Creating instances of the array, vector and hashtable
		int arr[] = new int[] { 1, 2, 3, 4 };
		Vector<Integer> v = new Vector();
		Hashtable<Integer, String> h = new Hashtable();

		// Adding the elements into the vector
		v.addElement(1);
		v.addElement(2);

		// Adding the element into the hashtable
		h.put(1, "nast");
		h.put(2, "dhangadhi");

		// Array instance creation requires [],
		// while Vector and hastable require ()
		// Vector element insertion requires addElement(),
		// but hashtable element insertion requires put()

		// Accessing the first element of the array, vector and hashtable
		System.out.println(arr[0]);
		System.out.println(v.elementAt(0));
		System.out.println(h.get(1));

		// Array elements are accessed using [],
		// vector elements using elementAt()
		// and hashtable elements using get()
	}
}
```

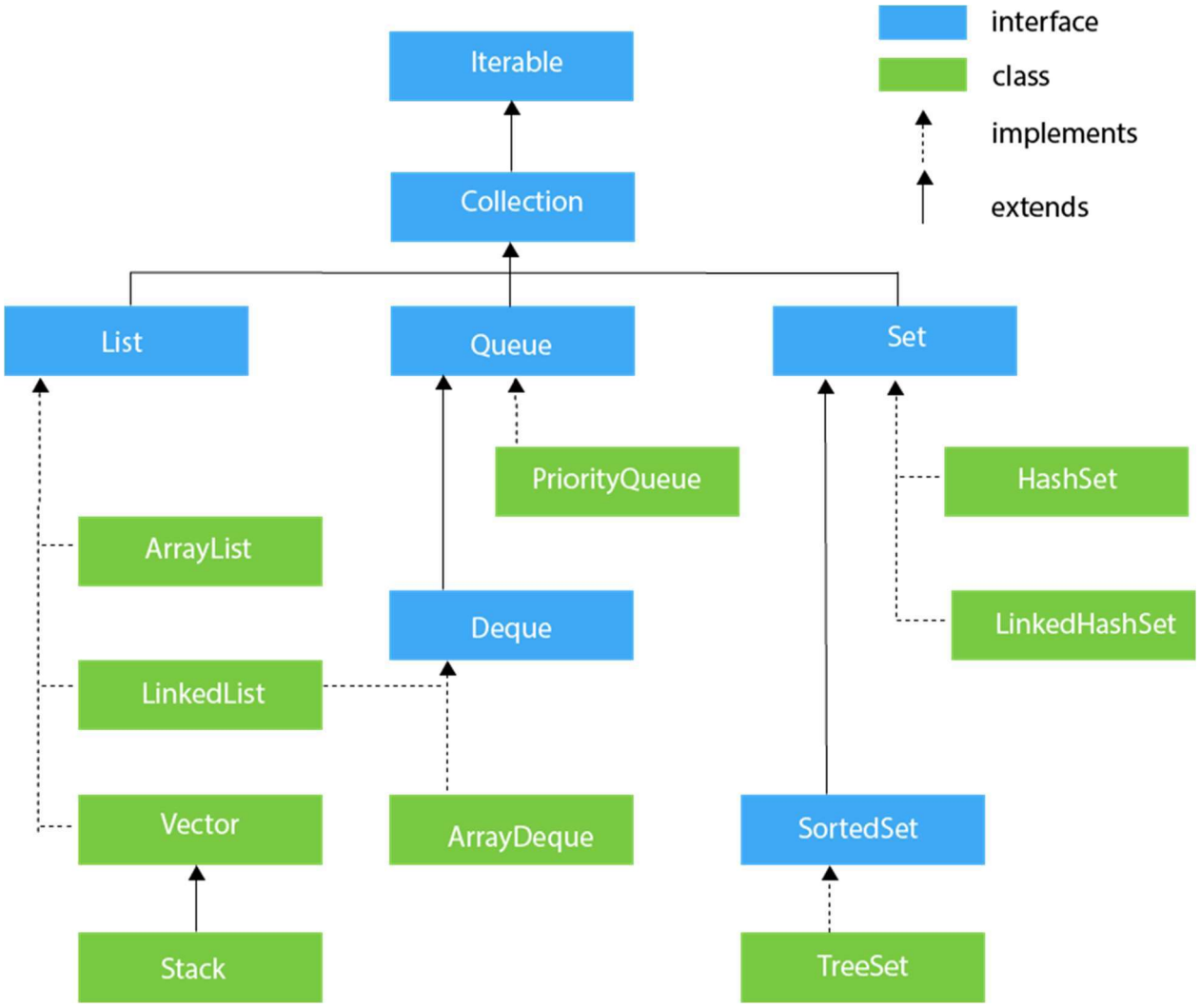
As we can observe, none of these collections (Array, Vector, or Hashtable) implements a standard member access interface, it was very difficult for programmers to write algorithms that can work for all kinds of Collections. Another drawback is that most of the 'Vector' methods are final, meaning we cannot extend the 'Vector' class to implement a similar kind of Collection. Therefore, Java developers decided to come up with a common interface to deal with the above-mentioned problems and introduced the Collection Framework in JDK 1.2 post which both, legacy Vectors and Hashtables were modified to conform to the Collection Framework.

Collection Framework

The Collection in Java is a framework that provides an architecture to store and manipulate the group of objects. Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.

Java Collection means a single unit of objects. Java Collection framework provides many interfaces (Set, List, Queue, Deque) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet).

Let us see the hierarchy of Collection framework. The java.util package contains all the classes and interfaces for the Collection framework.



Methods of Collection interface

add(Object)
addAll(Collection c)
clear()
contains(Object o)
containsAll(Collection c)

equals(Object o)
isEmpty()
iterator()
max()
remove(Object o)
removeAll(Collection c)
size()
toArray()

List Interface

For example:
List <T> al = new ArrayList<> ();
List <T> ll = new LinkedList<> ();
List <T> v = new Vector<> ();
Where T is the type of the object

ArrayList

```
// Java program to demonstrate the working of ArrayList
import java.util.ArrayList;

class ArrayListDemo {

	// Main Method
	public static void main(String[] args)
	{

		// Declaring the ArrayList with initial size n
		ArrayList<Integer> al = new ArrayList<Integer>();

		// Appending new elements at the end of the list
		for (int i = 1; i <= 5; i++)
			al.add(i);

		// Printing elements
		System.out.println(al);

		// Remove element at index 3
		al.remove(3);

		// Displaying the ArrayList after deletion
		System.out.println(al);

		// Printing elements one by one
		for (int i = 0; i < al.size(); i++)
			System.out.print(al.get(i) + " ");
	}
}
```

Set Interface

For example:
Set<T> hs = new HashSet<> ();
Set<T> lhs = new LinkedHashSet<> ();
Set<T> ts = new TreeSet<> ();
Where T is the type of the object.

```
// Java program to demonstrate the working of a HashSet
import java.util.HashSet;
import java.util.Iterator;

public class HashSetDemo {

	// Main Method
	public static void main(String args[])
	{
		// Creating HashSet and adding elements
		HashSet<String> hs = new HashSet<String>();

		hs.add("NAST");
		hs.add("For");
		hs.add("BE Computer");
		hs.add("Is");
		hs.add("Very helpful");

		// Traversing elements
		Iterator<String> itr = hs.iterator();
		while (itr.hasNext()) {
			System.out.println(itr.next());
		}
	}
}
```

Map Interface

For example:
Map<T> hm = new HashMap<> ();
Map<T> tm = new TreeMap<> ();
Where T is the type of the object.

HashMap

```
// Java program to demonstrate the working of a HashMap
import java.util.*;

public class HashMapDemo {

	// Main Method
	public static void main(String args[])
	{
		// Creating HashMap and adding elements
```

```
            HashMap<Integer, String> hm
            = new HashMap<Integer, String>();

            hm.put(1, "NAST");
            hm.put(2, "For");
            hm.put(3, "BE Computer");

            // Finding the value for a key
            System.out.println("Value for 1 is " + hm.get(1));

            // Traversing through the HashMap
            for (Map.Entry<Integer, String> e : hm.entrySet())
                  System.out.println(e.getKey() + " "
                              + e.getValue());
      }
}
```

[Note: Practice all the collection framework classes]