UNIT 1

History of Java

- Java started out as a research project.
- Research began in 1991 as the Green Project at Sun Microsystems, Inc.
- Research efforts birthed a new language, OAK. (A tree outside of the window of James Gosling's office at Sun).
- It was developed as an embedded programming language, which would enable embedded system application.
- It was not really created as web programming language.
- Java is available as jdk and it is an open source s/w.

Language was created with 5 main goals:

- It should be object oriented.
- A single representation of a program could be executed on multiple operating systems. (i.e. write once, run anywhere)
- It should fully support network programming.
- It should execute code from remote sources securely.
- It should be easy to use.
- Oak was renamed Java in 1994.

Java Platforms

There are three main platforms for Java:

- Java SE (Java Platform, Standard Edition) – runs on desktops and laptops.
- Java ME (Java Platform, Micro Edition) – runs on mobile devices such as cell phones.
- Java EE (Java Platform, Enterprise Edition) – runs on servers.

Java Terminology

1. **Java Development Kit:**

It contains one (or more) JRE's along with the various development tools like the Java source compilers, bundling and deployment tools, debuggers, development libraries, etc.

2. **Java Virtual Machine:**

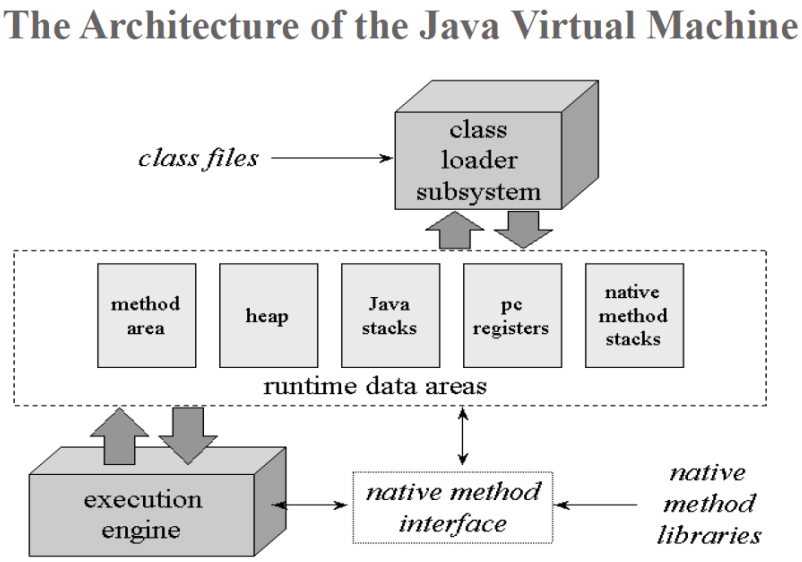
An abstract machine architecture specified by the Java Virtual Machine Specification.

It interprets the byte code into the machine code depending upon the underlying OS and hardware combination. JVM is platform dependent. (It uses the class libraries, and other supporting files provided in JRE)

3. Java Runtime Environment:

A runtime environment which implements Java Virtual Machine, and provides all class libraries and other facilities necessary to execute Java programs. This is the software on your computer that actually runs Java programs.

JRE = JVM + Java Packages Classes (like util, math, lang, awt, swing etc) +runtime libraries.



Java Virtual Machine

- Class loader subsystem: A mechanism for loading types (classes and interfaces) given fully qualified names.
- The Java virtual machine organizes the memory it needs to execute a program into several runtime data areas.
- Each Java virtual machine also has an execution engine: a mechanism responsible for executing the instructions contained in the methods of loaded classes.

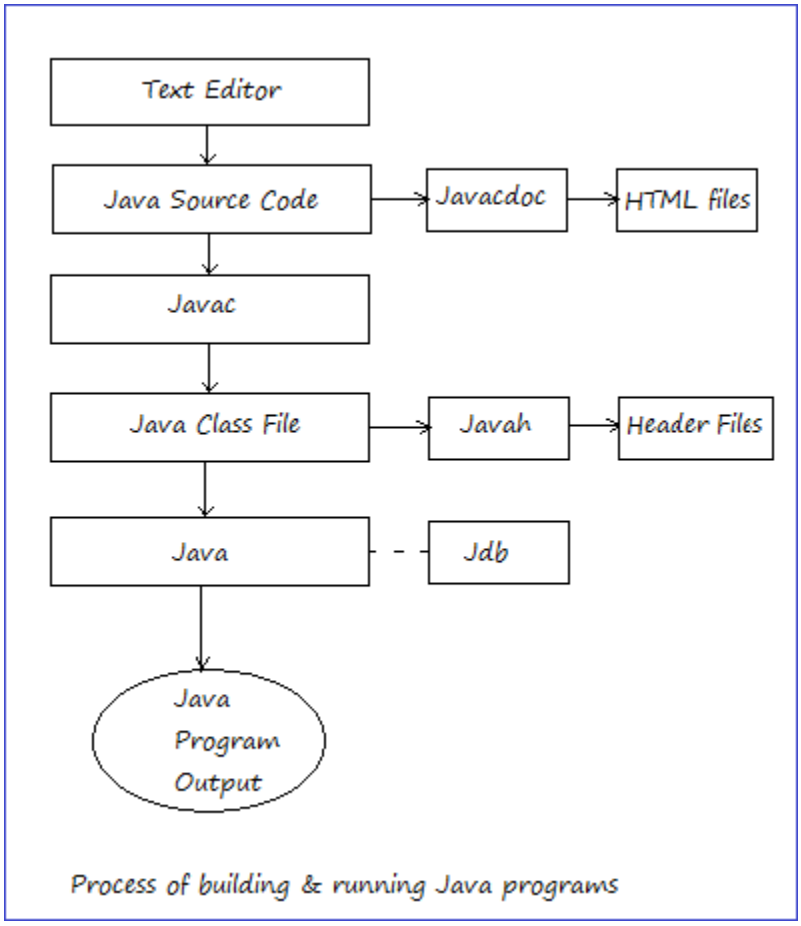
Class loader subsystem

- The Java virtual machine contains two kinds of class loaders: a bootstrap class loader and user-defined class loaders.
- The bootstrap class loader is a part of the virtual machine implementation, and user-defined class loaders are part of the running Java application.
- **Loading**: finding and importing the binary data for a type
- **Linking**: performing verification, preparation, and (optionally) resolution
- **Verification**: ensuring the correctness of the imported type
- **Preparation**: allocating memory for class variables and initializing the memory to default values
- **Resolution**: transforming symbolic references from the type into direct references.
- **Initialization**: invoking Java code that initializes class variables to their proper starting values.

When the virtual machine loads a class file, it parses information about a type from the binary data contained in the class file.

It places this type information into the method area. As the program runs, the virtual machine places all objects the program instantiates onto the heap. As each new thread comes into existence, it gets its own pc register (program counter) and Java stack.

Byte code is a highly optimized set of instructions designed to be executed by the Java run-time system, which is called the Java Virtual Machine (JVM). JVM is an interpreter for byte code.



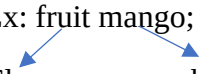
Process of building & running Java programs

Object Oriented Programming Concepts

- Objects
- Classes
- Data abstraction and Encapsulation
- Inheritance
- Polymorphism
- Dynamic Binding
- Interface

1. A **class** is collection of objects of similar type or it is a template.

Ex: fruit mango;



2. **Objects** are instances of the type class.
3. **Encapsulation**
   - Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse.
   - One way to think about encapsulation is as a protective wrapper that prevents the code and data from being arbitrarily accessed by other code defined outside the wrapper.
   - Access to the code and data inside the wrapper is tightly controlled through a well-defined interface.
   - To relate this to the real world, consider the automatic transmission on an automobile.
   - It encapsulates hundreds of bits of information about your engine, such as how much we are accelerating, the pitch of the surface we are on, and the position of the shift.
   - The power of encapsulated code is that everyone knows how to access it and thus can use it regardless of the implementation details—and without fear of unexpected side effects.
4. **Abstraction**
   - Abstraction in Java or Object-oriented programming is a way to segregate implementation from interface and one of the five fundamentals along with Encapsulation, Inheritance, Polymorphism, Class and Object.
   - An essential component of object-oriented programming is Abstraction
   - Humans manage complexity through abstraction.
   - For example, people do not think a car as a set of tens and thousands of individual parts.
   - They think of it as a well-defined object with its own unique behavior.
   - This abstraction allows people to use a car ignoring all details of how the engine, transmission and braking systems work.
   - In computer programs the data from a traditional process-oriented program can be transformed by abstraction into its component objects.
   - A sequence of process steps can become a collection of messages between these objects. Thus, each object describes its own behavior.
5. **Inheritance**
   - Object-oriented programming allows classes to *inherit* commonly used state and behaviour from other classes. Different kinds of objects often have a certain amount in common with each other.
   - In the Java programming language, each class is allowed to have one direct superclass, and each superclass has the potential for an unlimited number of *subclasses*

- The derivation of one class from another so that the attributes and methods of one class are part of the definition of another class. The first class is often referred to the base or parent class. The child is often referred to as a derived or sub-class. Derived classes are always a kind of their base classes. Derived classes generally add to the attributes and/or behaviour of the base class. Inheritance is one form of object-oriented code reuse.
- Mountain bikes, road bikes, and tandem bikes, for example, all share the characteristics of bicycles (current speed, current pedal cadence, current gear). Yet each also defines additional features that make them different: tandem bicycles have two seats and two sets of handlebars; road bikes have drop handlebars; some mountain bikes have an additional chain ring, giving them a lower gear ratio. In this example, Bicycle now becomes the *super class* of Mountain Bike, Road Bike, and Tandem Bike.

6. **Polymorphism**
   - *Polymorphism* (from the Greek, meaning ―many forms‖) is a feature that allows one interface to be used for a general class of actions.
   - The specific action is determined by the exact nature of the situation. Consider a stack (which is a last-in, first-out list). We might have a program that requires three types of stacks. One stack is used for integer values, one for floating-point values, and one for characters. The algorithm that implements each stack is the same, even though the data being stored differs.
   - In Java we can specify a general set of stack routines that all share the same names.
   - More generally, the concept of polymorphism is often expressed by the phrase ―one interface, multiple methods. This means that it is possible to design a generic interface to a group of related activities.
   - This helps reduce complexity by allowing the same interface to be used to specify a *general class of action.*
   - Polymorphism allows us to create clean, sensible, readable, and resilient code.
7. **Dynamic Binding**
   - When a method is called within a program, it associated with the program at run time rather than at compile time is called dynamic binding.
8. **Interface**
   - The behaviour that a class exposes to the outside world; its public face. Also called its contract. In Java interface is also a keyword similar to class. However, a Java interface contains no implementation: it simply describes the behaviour expected of a particular type of object, it doesn't so how that behaviour should be implemented

Benefits of OOP

- Through inheritance, we can eliminate redundant code and extend the use of existing classes.
- The principle of data hiding helps the programmer to build secure programs.
- It is easy to partition the work in a project based on objects.
- Object oriented system easily upgraded from small to large systems.

- Software complexity can be easily managed.

Differences b/w C++ and Java (FYI)

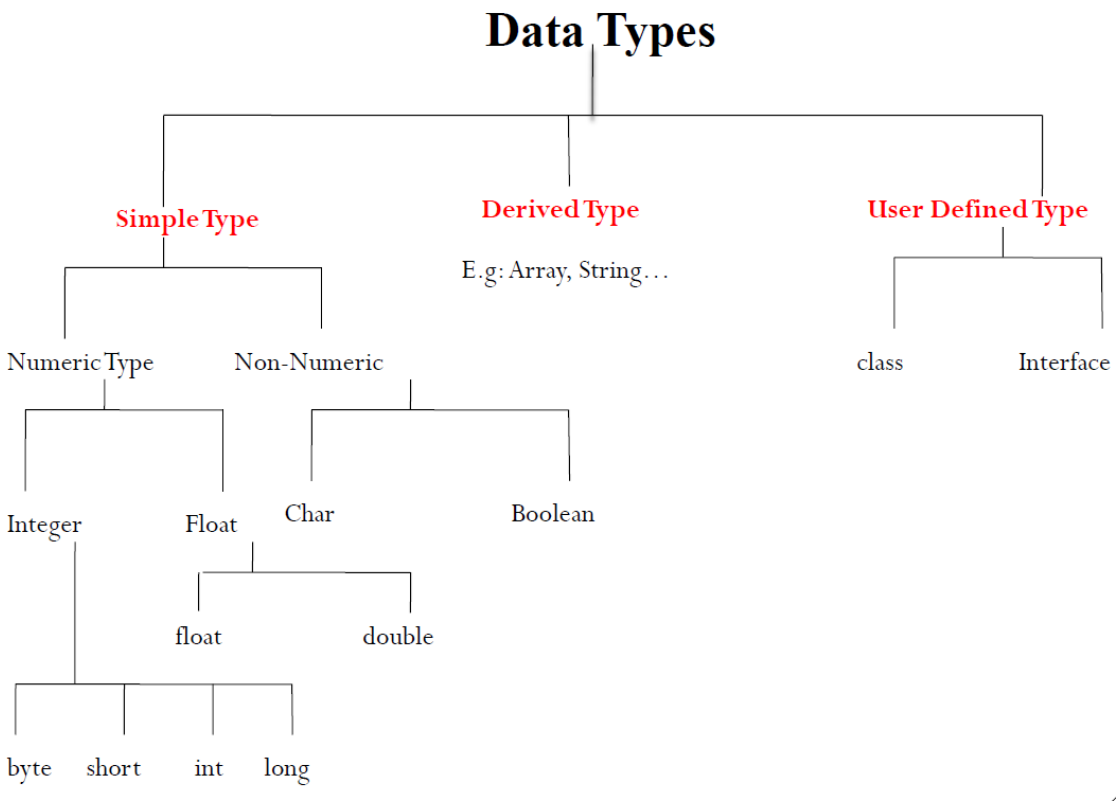
C++	Java
1. Global variable are supported.	1. No Global variables. Everything must be inside the class only.
2.Multiple inheritance is supported.	2. No direct multiple Inheritance.
3.Constructors and Destructors supported.	3.Java supporting constructors only & instead of destructors garbage collection is supported.
4.In c++ pointers are supported.	4.No pointer arithmetic in Java.
5.C++ supporting ASCII character set.	5. Java supports Uni code Character set.

Features of Java (Java Buzz Words)

- Simple
- Object Oriented
- Compile, Interpreted and High Performance
- Portable
- Reliable
- Secure
- Multithreaded
- Dynamic
- Distributed
- Architecture-Neutral

1. **Simple**
   - Java is very easy to learn, and its syntax is simple, clean and easy to understand
   - Java has removed many complicated and rarely-used features, for example, explicit pointers, operator overloading, etc.
   - There is no need to remove unreferenced objects because there is an Automatic Garbage Collection in Java
   - Rich pre-defined class library

2. **Object Oriented**
   - Focus on the data (objects) and methods manipulating the data
   - All methods are associated with objects
   - Potentially better code organization and reuse
3. **Compile, Interpreted and High Performance**
   - Java compiler generate byte-codes, not native machine code
   - The compiled byte-codes are platform-independent
   - Java byte codes are translated on the fly to machine readable instructions in runtime (Java Virtual Machine)
   - Easy to translate directly into native machine code by using a just-in-time compiler.
4. **Portable**
   - Same application runs on all platforms
   - The sizes of the primitive data types are always the same
   - The libraries define portable interfaces
5. **Reliable/Robust**
   - Extensive compile-time and runtime error checking
   - No pointers but real arrays. Memory corruptions or unauthorized memory accesses are impossible
   - Automatic garbage collection tracks objects usage over time
   - Secure: Java Programs run inside a virtual machine sandbox
   - Java's robustness feature makes java secure.
   - Access restrictions are forced (private, public)
6. **Multithreaded**
   - It supports multithreaded programming.
   - Need not wait for the application to finish one task before beginning another one.
7. **Dynamic**
   - Libraries can freely add new methods and instance variables without any effect on their clients
   - Interfaces promote flexibility and reusability in code by specifying a set of methods an object can perform, but leaves open how these methods should be implemented.
8. **Distributed**
   - Java is designed for the distributed environment of the Internet, because it handles TCP/IP protocols.
   - Allows objects on two different computers to execute procedures remotely by using package called Remote Method Invocation (RMI).
9. **Architecture-Neutral**
   - Java is architecture neutral because there are no implementation dependent features, for example, the size of primitive types is fixed.



Java is a **Strongly Typed Language**

- Every variable has a type, every expression has a type, and every type is strictly defined.
- All assignments, whether explicit or via parameter passing in method calls, are checked for type compatibility.
- There are no automatic conversions of conflicting types as in some languages.

For example, in C/C++ you can assign a floating-point value to an integer. In Java, you cannot.

1. **Integer Data Types**
   - Java does not support unsigned, positive-only integers.
   - All are signed, positive and negative values.
     Ex: int a=2, b= -1;

2. **Byte Data Types**
   - The smallest integer type is byte.
   - Variables of type byte are especially useful while working with a stream of data from a network or file.
   - Byte variables are declared by use of the byte keyword.
     Ex: byte b, c;

3. **Floating Point Types**
   - There are two kinds of floating-point types.
   - All math functions, such as sin( ), cos( ), and sqrt( ), return double values.

Ex: float a=3.5F;
double b=6.70;

4. **Boolean Data Types**
   - It can have only one of two possible values, **true** or **false**.
   - This is the type, returned by all relational operators, such as a < b.
     Ex: bool c=true;

5. **Character Data Type**

- char in Java is not the same as char in C or C++.
- Java uses Unicode to represent characters.
- Unicode defines a fully international character set that can represent all of the characters found in all human languages.
- It is a unification of dozens of characters sets, such as Latin, Greek, Arabic, Cyrillic, Hebrew, Katakana, Hangul, and many more.
- Hence it requires 16 bits.
- The range of a char in java is 0 to 65,536.
- There are no negative chars.

Data Types

Name	Width in bits	Range
long	64	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
int	32	–2,147,483,648 to 2,147,483,647
short	16	–32,768 to 32,767
byte	8	–128 to 127
double	64	4.9e–324 to 1.8e+308
float	32	1.4e−045 to 3.4e+038
char	16	0 to 65,536.

Type Conversion and Casting

- We can assign a value of one type to a variable of another type.
- If the two types are compatible, then Java will perform the conversion automatically.
- For example, it is always possible to assign an int value to a long variable. However, not all types are compatible, and thus, not all type conversions are implicitly allowed.
- But it is possible for conversion between incompatible types. To do so, you must use a cast, which performs an explicit conversion between incompatible types.

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 satisfied:

- 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, the numeric types are not compatible with char or boolean. Also, char and boolean are not compatible with each other.

Casting Incompatible Types

- When assigning an int value to a byte variable, 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, you 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.

Variables

- The variable is the basic unit of storage in a Java program.
- A variable is defined by the combination of an identifier, a type, and an optional initializer.

Declaring a Variable

- In Java, all variables must be declared before they can be used.

Syntax:

data_type var1, var2, var3,…….varn;

Initialization (Compile time):

var1=value;

var2=value;

Types

- Instance Variable
- Static Variable
- Local Variable
- Parameters

1. **Local variables:**
   - Local variables are declared in methods, constructors, or blocks.

- Local variables are created when the method, constructor or block is entered and the variable will be destroyed once it exits the method, constructor or block.
- Access modifiers cannot be used for local variables.
- Local variables are visible only within the declared method, constructor or block.
- There is no default value for local variables so local variables should be declared and an initial value should be assigned before the first use.

2. **Instance variables:**
   - Instance variables are declared in a class, but outside a method, constructor or any block.
   - Instance variables are created when an object is created with the use of the key word 'new' and destroyed when the object is destroyed.
   - Each object gets its own copy of the instance variables.
   - Access modifiers can be given for instance variables.
   - The instance variables are visible for all methods, constructors and block in the class.
   - Instance variables have **default values (usually 0).**
   - Instance variables can be accessed directly by calling the variable name inside the class.
   - However, within static methods and different class (when instance variables are given accessibility) that should be called using the fully qualified name **ObjectReference.VariableName**
3. **Static variables:**
   - Class variables also known as static variables are declared with the *static* keyword in a class, but outside a method, constructor or a block.
   - There would only be one copy of each class variable per class, regardless of how many objects are created from it.
   - Static variables are stored in static memory.
   - Static variables are created when the program starts and destroyed when the program stops.
   - Visibility is similar to instance variables.
   - Default values are same as instance variables.
   - Static variables can be accessed by calling with the class name **ClassName.variableName**

Sample Program

```
class HelloWorld {

    public static void main (String args []) {

        System.out.println ("Welcome to Java Programming…..");

    }

}
```

- ***public*** allows the program to control the visibility of class members. When a class member is preceded by ***public***, then that member may be accessed by code outside the class in which it is declared.
- In this case, ***main( )*** must be declared as public, since it must be called by code outside of its class when the program is started.
- ***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 interpreter before any objects are made.
- ***void*** states that the ***main( )*** method will not return any value.
- ***main( )*** is called when a Java application begins. In order to run a class, the class must have a ***main( )*** method.
- ***String args[]*** declares a parameter named ***args***, which is an array of ***String*** class. In this case, ***args*** receives any command-line arguments present when the program is executed.
- ***System*** is a class which is present in **java.lang** package.
- ***out*** is a static field which is defined in ***System*** class which returns a **PrintStream** object. As ***out*** is a static field it can be referenced directly with classname.
- ***println( )*** is a method which presents in **PrintStream** class which can be called through the **PrintStream** object returned by static field ***out*** presented in System class to print a line to console.

```
class Sample{
public static void main(String args[]){
        System.out.println("sample:main");
        Sample s=new Sample();
        s.display();
}
void display(){
        System.out.println("display:main");
        }
}
```

The Scope and Lifetime of Variables

Scope

- The scope of a declared element is the portion of the program where the element is visible.

Lifetime

- The lifetime of a declared element is the period of time during which it is alive.
- The lifetime of the variable can be determined by looking at the context in which they're defined.

- Java allows variables to be declared within any block.
- A block begins with an opening curly brace and ends by a closing curly brace.
- Variables declared inside a scope are not accessible to code outside.
- Scopes can be nested. The outer scope encloses the inner scope.
- Variables declared in the outer scope are visible to the inner scope.
- Variables declared in the inner scope are not visible to the outside scope.

```
public class Scope
{
    public static void main(String args[]){
        int x; //know to all code within main
        x=10;
        if(x==10){ // starts new scope
            int y=20; //Known only to this block
            //x and y both known here
            System.out.println("x and y: "+x+" "+y);
            x=y+2;
        }
        // y=100; // error ! y not known here
        //x is still known here
        System.out.println("x is "+x);
    }
}
```

Control Statements

- **Selection Statements**: if, if-else-if ladder, nested ifs & switch
- **Iteration Statements**: for, while and do-while
- **Jump Statements**: break, continue and return

1. **Selection Statements**

```
if (condition)
    statement1;
else
    statement2;
```

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

```
switch (expression)
 {
    case value1:
     // statement sequence
       break;
    case value2:
    // statement sequence
       break;
    …
    case valueN:
    // statement sequence
        break;
    default:
  // default statement sequence
}
```

- The **condition** is any expression that returns a **boolean** value.
- The **expression** must be of type byte, short, int, or char;
- Each of the values specified in the case statements must be of a **type compatible** with the **expression**.

a) If Statement:

- The if statement executes a block of code only if the specified expression is true. If the value is false, then the if block is skipped and execution continues with the rest of the program. The simple if statement has the following syntax:

if (<conditional expression>)
        <statement action>

The following program explains the if statement.

```
public class programIF{

public static void main(String[] args){

        int a = 10, b = 20;

        if (a > b)
```

```
            System.out.println("a > b");
        if (a < b)
            System.out.println("b < a");
        }
}
```

b) If-else Statement:

- The if/else statement is an extension of the if statement. If the condition in the if statement fails, the statements in the else block are executed. The if-else statement has the following

Syntax:

```
if (<conditional expression>)
        <statement action>
else
        <statement action>
```

The following program explains the if-else statement.

```
public class ProgramIfElse{
public static void main(String[] args){
    int a = 10, b = 20;
    if (a > b)
    {
        System.out.println("a > b");
    }
    else
    {
        System.out.println("b < a");
    }
}
}
```

c) If-else-if ladder

- A common programming construct that is based upon a sequence of nested ifs is the if-else-if ladder.
- It looks like this:

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

statement;

```
class Javaapp {

    public static void main(String[] args) {

        int mark = 85;
        if(mark<=40)
        {
            System.out.println("fail");
        }
        else if(mark<=60)
        {
            System.out.println("C Grade");
        }
        else if(mark<=80)
        {
            System.out.println("B Grade");
        }
        else if(mark<=100)
        {
            System.out.println("A Grade");
        }
    }
}
```

d) switch statement

- The switch statement is Java's multiway branch statement. It provides an easy way to dispatch execution to different parts of your code based on the value of an expression. As such, it often provides a better alternative than a large series of if-else-if statements.

2. **Iteration Statements**

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

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

```
for(initialization; condition; iteration)
{
  // body
}
```

a. while loop

- It repeats a statement or block while its controlling expression is true. Here is its general form:

  while (condition) {

  // body of loop

}

- The condition can be any Boolean expression. The body of the loop will be executed as long as the conditional expression is true. When condition becomes false, control passes to the next line of code immediately following the loop.
- The curly braces are unnecessary if only a single statement is being repeated.

// Demonstrate the while loop.

```
public class JavaApplication18 {
    public static void main(String[] args) {
        int i=5;
        while (i<5)
        {
            System.out.println("Hello");
        }
    }
}
```

b. do-while

- As you just saw, if the conditional expression controlling a while loop is initially false, then the body of the loop will not be executed at all.
- However, sometimes it is desirable to execute the body of a while loop at least once, even if the conditional expression is false to begin with.
- Syntax:

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

- Each iteration of the do-while loop first executes the body of the loop and then evaluates the conditional expression. If this expression is true, the loop will repeat. Otherwise, the loop terminates.

```
// Demonstrate the do-while loop.
class DoWhile {
public static void main(String args[]) {
        int n = 10;
        do {
                System.out.println("tick " + n);
                n--;

        } while(n > 0);
```

```
        }
    }
```

c. for loop

- Here is the general form of the for statement:

```
for(initialization; condition; iteration) {
        // body
}
```

- The for loop operates as follows.
- When the loop first starts, the initialization portion of the loop is executed.
- Generally, this is an expression that sets the value of the loop control variable, which acts as a counter that controls the loop.
- Next, condition is evaluated. This must be a Boolean expression.
- It usually tests the loop control variable against a target value.
- If this expression is true, then the body of the loop is executed. If it is false, the loop terminates. Next, the iteration portion of the loop is executed. This is usually an expression that increments or decrements the loop control variable**.**

```
// Demonstrate the for loop.

class ForTick {

public static void main(String args[]) {

        int n;

        for(n=10; n>0; n--)

                System.out.println("tick " + n);

}

}
```

d. foreach loop

- It starts with the keyword for like a normal for-loop.
- Instead of declaring and initializing a loop counter variable, you declare a variable that is the same type as the base type of the array, followed by a colon, which is then followed by the array name.

- In the loop body, you can use the loop variable you created rather than using an indexed array element.

Syntax:

```
for (type var : array)
{
    statements using var;
}
```

is equivalent to:

```
for (int i=0; i<arr.length; i++)
{
    type var = arr[i];
    statements using var;
}
```

//Demonstrate foreach loop

```
class Loop {
  public static void main(String args[]) {
    // declaring an array
    int arr[] = { 18, 19, 21, 29, 46 };
    // traversing the array with for-each loop
    for (int i : arr) {
      System.out.println(i);
    }
  }
}
```

3. **Jump Statements (FYI)**

```
continue;     //bypass the followed instructions

break;    //exit from the loop

label:
- - - -
- - - -
break label;          //it's like goto
statement

return;      //control returns to the caller
```

ARRAYS

- An array is a group of like-typed variables that are referred to by a common name. Arrays of any type can be created and may have one or more dimensions. A specific element in an array is accessed by its index. Arrays offer a convenient means of grouping related information.

1. **One-Dimensional Arrays**
    - A one-dimensional array is, essentially, a list of like-typed variables.
    - To create an array, you first must create an array variable of the desired type.
    - The general form of a one-dimensional array declaration is

      type var-name[ ];
    - Here, type declares the base type of the array. The base type determines the data type of each element that comprises the array.
    - For example, the following declares an array named month with the type "array of int":

      Ex: int month [];
    - Although month is an array variable, no array actually exists.
    - The value of month is set to null, which represents an array with no value.
    - To link month with an actual, physical array of integers, you must allocate one using **new** and assign it to month.
    - **new** is a special operator that allocates memory dynamically.
    - The general form of new is as follows:

      array-var = new type[size];
    - The elements in the array allocated by new will automatically be initialized to zero.

Ex: month = new int[12];

- Another way to declare an array in single step is

  type array-var =new type[size];

- Arrays can be initialized when they are declared. There is no need to use **new** operator.
- The array will automatically be created large enough to hold the number of elements you specify in the array initializer.

  Ex: int month[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31,30, 31 };

Demonstrate 1D array

```
import java.util.Scanner;
class ArrayEx{
   public static void main(String args[]){
        Scanner input=new Scanner(System.in);
        int a[]={10,20,30,40,50};
        char []c={'a','b','c','d','e'};
        int b[]=new int[5];
        for(int i=0;i<5;i++){
                System.out.print(a[i]+" ");
                System.out.println(c[i]+" ");
        }
        for(int i=0;i<5;i++){
                b[i]=input.nextInt();
        }
        for(int i=0;i<5;i++){
                System.out.print(b[i]+" ");
        }
   }
}
```

2. Multidimensional Arrays

- In Java, multidimensional arrays are actually arrays of arrays.
- To declare a multidimensional array variable, specify each additional index using another set of square brackets.
- For example, the following declares a two-dimensional array variable called **twoD**.

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

- This allocates a 4 by 5 array and assigns it to twoD.

// Demonstrate a two-dimensional array.

class TwoDArray {

```
public static void main(String args[]) {
        int twoD[][]= new int[4][5];
        int i, j, k = 0;
        for(i=0; i<4; i++)
                for(j=0; j<5; j++) {
                        twoD[i][j] = k;
                        k++;
                }
        for(i=0; i<4; i++) {
                for(j=0; j<5; j++)
                        System.out.print(twoD[i][j] + " ");
                System.out.println();
                }
        }
}
```

Output:

```
0 1 2 3 4
5 6 7 8 9
10 11 12 13 14
15 16 17 18 19
```

- When you allocate memory for a multidimensional array, you need only specify the memory for the first (leftmost) dimension.
- You can allocate the remaining dimensions separately.

Ex:

```
int twoD[][] = new int[4][];
twoD[0] = new int[5];
twoD[1] = new int[5];
twoD[2] = new int[5];
twoD[3] = new int[5];
```

- We can create a two-dimensional array in which the sizes of the second dimension are unequal. Such type of arrays are called Irregular Arrays.

```
// Manually allocate differing size second dimensions.
class TwoDAgain {
public static void main(String args[]) {
        int twoD[][] = new int[4][];
        twoD[0] = new int[1];
        twoD[1] = new int[2];
        twoD[2] = new int[3];
        twoD[3] = new int[4];
        int i, j, k = 0;
        for(i=0; i<4; i++)
                for(j=0; j<i+1; j++) {
                        twoD[i][j] = k;
                        k++;
```

```
                }
        for(i=0; i<4; i++) {
                for(j=0; j<i+1; j++)
                        System.out.print(twoD[i][j] + " ");
                System.out.println();
                }
        }
}
```

Output:

```
0
1 2
3 4 5
6 7 8
```

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 al[] = 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];

Using length member:

- Because arrays are implemented as objects, each array has associated with it a length instance variable that contains the number of elements that the array can hold.
- In other words, length contains the size of the array.
- The length property can be invoked by using the dot (.) operator
- The general form is
  array_name.length;
- **For example,**
  int[] arr=new int[5];
  int arrayLength=arr.length;
- In this example, **arr.length** will return the size of the array **arr,** i.e. 5

```
//Demonstrate the use of length
public class ArrayLengthExample1 {
public static void main(String[] args)  {
    int[] num = new int[10];
    int arrayLength=num.length;
    System.out.println("The length of the array is: "+ arrayLength);
```

```
    }
}
```

Output:

The length of the array is: 10

Classes and Objects

Introduction to Class

- A class is a template that defines the form of an object which contains instance variables and methods.
- A class is the blueprint from which individual objects are created.
- When you define a class, you declare its exact form and nature. A class is created by using the keyword **class**.
- The general form of a class is

```
class className {
        //declare instance variables
        type var1;
        type var2
        //…….
        type varN;
        //declare methods
        type methodname1(parameter-list) {
                // body of method
        }
        // .....
        type methodnameN(parameter-list) {
                // body of method
        }
}
```

- The data, or variables, defined within a class but outside the methods are called instance variables

Defining a Class

- Let's define a class called Box

```
class Rectangle {
        int length, width;
}
```

- A class definition creates a new data type. In this case, the new data type is **Rectangle**
- We can use this name to declare objects of type **Rectangle**
- A class declaration is only a type description, it does not create an actual object

Declaring an Object

- Object is an instance of a class
- When you create a class, you are creating a new data type. You can use this type to declare objects of that type.
- However, obtaining objects of a class is a two-step process
    a. Declaring a variable of the class type. This variable does not define an object. Instead, it is simply a variable that can refer to an object.
    b. Acquiring an actual, physical copy of the object and assign it to that variable. This can be done using the **new** operator. The **new** operator dynamically allocates (i.e., allocates at run time) memory for an object and returns a reference to it. This reference is, more or less, the address in memory of the object allocated by **new**. This reference is then stored in the variable
- Each time you create an instance of a class, you are creating an object that contains its own copy of each instance variable defined by the class.
- In the above programs to declare an object of type Rectangle:

  Rectangle r = new Rectangle( );
- This statement combines the two steps just described. It can be rewritten like this to show each step more clearly:

  Rectangle r; // declares reference to object
  r = new Rectangle(); // allocate a Box object
- The first line declares **r** as a reference to an object of type **Rectangle**. After this line executes, **r** contains the value null, which indicates that it does not yet point to an actual object.
- Any attempt to use **r** at this point will result in a compile-time error.
- The next line allocates an actual object and assigns a reference to it to **r**. After the second line executes; you can use **r** as if it were a **Rectangle** object

Accessing class members

- To access instance variables, **dot(.)** operator is used. It links the name of an object with the name of a member.
- The general form to access an instance variable is

  object**.**instance_variable;

```
//Demonstrate class and object creation
class Box {
    double width;
    double height;
    double depth;
    }
class Demo {
    public static void main(String[] args){
        Box b1 = new Box();  //1st object of class Box
```

```
        Box b2 = new Box(); //2nd object of class Box
// Objects b1 and b2 will get their own copy of width, height and depth

        b1.width=20; //setting width of b1
        b1.height=21; //setting height of b1
        b1.depth=20; //setting depth of b1

        b2.width=30; //setting width of b2
        b2.height=20; //setting height of b2
        b2.depth=25; //setting depth of b2
        System.out.println("Width, height and depth of b1: "+b1.width+" "+b1.height+" "+b1.depth);
        System.out.println("Width, height and depth of b2: "+b2.width+" "+b2.height+" "+b2.depth);
    }
}
```

Output:

Width, height and depth of b1: 20 21 20

Width, height and depth of b2: 30 20 25

b1 →

width	20
height	21
depth	20

b2 →

width	30
height	20
depth	25

Reference Variables and Assignment

- When one primitive-type is assigned to another, i.e., the statements int x=y means that **x** receives a copy of the value contained in **y.**
- Hence, after the assignment, both x and y will contain their own, independent copies of the value. Changing one does not affect the other.
- But, when one object variable is assigned to another, the situation is bit different because you are assigning references.
- This means that you are changing the object that the reference variable refers to, not making a copy of that object
- For example,

```
Box b1=new Box();

Box b2=b1;
```

- In this case b1and b2 will both refer to the same object. Therefore, object can be changed through b1 or b2.

```
b1.width = 12;
System.out.println(b1.width);
System.out.println(b2.width);
```

- After executing both of these println() statements, we get the **same value for both** the statements as 12.

String Class in Java

- In many other programming languages, a string is defined as an array of characters. But is not the same in Java
- In Java, a string is defined as an object

Constructing Strings

1. Using new operator

Ex:

```
String str = new String("Hello");
```

- This creates a String object str which contains the character string "Hello"

2. Constructing a String from another String

Ex:

```
String str1 = new String("Hello")';
String str2 = new String("Hello");
```

- In this example, str2 is created from str1. Therefore, str2 will also contain the same characters as str1
- Though str1 and str2 have the same character sequence, they are referring to two different objects

3. Using String literal

Ex:

```
String str = "Welcome to Java";
```

- In this case, the string literal "Welcome to Java" is automatically converted to a String object by JVM and is assigned to str.

Ex:

```
class Demo{
    public static void main(String[] args) {
        String str1 = new String("Hello"); //Constructed using new operator
        String str2 = new String(str1); //Using str1
        String str3 = "Welcome to Java"; //Using string literal

        System.out.println(str1);
        System.out.println(str2+"\n"+str3);
    }
}
```

Output:

```
Hello
Hello
Welcome to Java
```

Operating on Strings

General forms of few methods that operate on strings

boolean equals(str)	Returns true if the invoking string contains the same character sequence as str
int length( )	Returns the number of characters (size) in the string
char charAt(index)	Returns the character at the place specified by "index"
int compareTo(str)	Returns negative value if the invoking string is less than str, positive value if the invoking string is greater than str, and zero if the strings are equal
int indexOf(str)	Searches the invoking string for the substring specified by str and returns the index of the first match or -1 on failure
int lastIndexOf(str)	Searches the invoking string for the substring specified by str and returns the index of the last match or -1 on failure
String toUpperCase()	Returns a string in upper case letters
String toLowerCase()	Ronverts and returns a string in lower case letters
char[] toCharArray()	Returns a string in a new character array
String substring(int start, int end)	Returns a new string that contains a specified portion of the invoking string

Consider the below example

String str1 = "Java";

String str2 = "JAVA";

String str3 =JavaJava;

1. boolean equals(str)
   - str1.equals(str2) returns false, as invoking string str1 does not have same character sequence as str2.

2. int length( )
   - str1.length( ) returns 4

3. char charAt(index)
   - str1.charAt(2) returns character “v” as index starts from 0

4. int compareTo(str)
   - str1. compareTo(str2) returns 32, as ASCII value of “a” is 32 times greater than “A”

5. int indexOf(str)
   - str3.indexOf(“Java”) returns 0, as it returns the index of the first match

6. int lastIndexOf(str)
   - str3.lastIndexOf(“Java”) returns 4, as it returns the index of the last match

Ex:

```
class Demo{
    public static void main(String[] args) {
        String str1 = "Java";
        String str2 = "JAVA";
        String str3 = "JavaJava";

        System.out.println("Is str1 = str2 ? "+str1.equals(str2));
        System.out.println("Length "+str1.length());
        System.out.println("Character "+str1.charAt(2));
        System.out.println("Compare "+str1.compareTo(str2));
        System.out.println("Firstindex1 "+str3.indexOf("Java"));
        System.out.println("Firstindex2 "+str3.indexOf("JAVA"));
        System.out.println("Lastindex "+str3.lastIndexOf(str1));
        System.out.println("Uppercase "+str3.toUpperCase());
        System.out.println("Lowercase "+str3.toLowerCase());
        System.out.println("Substring "+str3.substring(1,4));
    }
}
```

Output:

```
Is str1 = str2 ? false
Length 4
Character v
Compare 32
Firstindex1 0
Firstindex2 -1
Lastindex 4
Uppercase JAVAJAVA
Lowercase javajava
Substring ava
```

Arrays of Strings

- Like other types, its possible to create array of strings

Ex:

```
class Demo{
    public static void main(String[] args) {
        String[] str = {"This", "is", "an","array", "of", "strings"};
        System.out.println("Array elements are");
        for(String s:str)   //foreach loop
                System.out.print(s+" ");
        System.out.println();
        str[1] = "was";
        System.out.println("\nModified array elements are");
        for(String s:str)
            System.out.print(s+" ");
    }
}
```

Output:

```
Array elements are
This is an array of strings

Modified array elements are
This was an array of strings
```

Strings are Immutable

- The contents of a String object are immutable. That means, once created the character sequence that makes up the string cannot be altered
- Therefore, all the methods that are defined above don’t alter the original String object