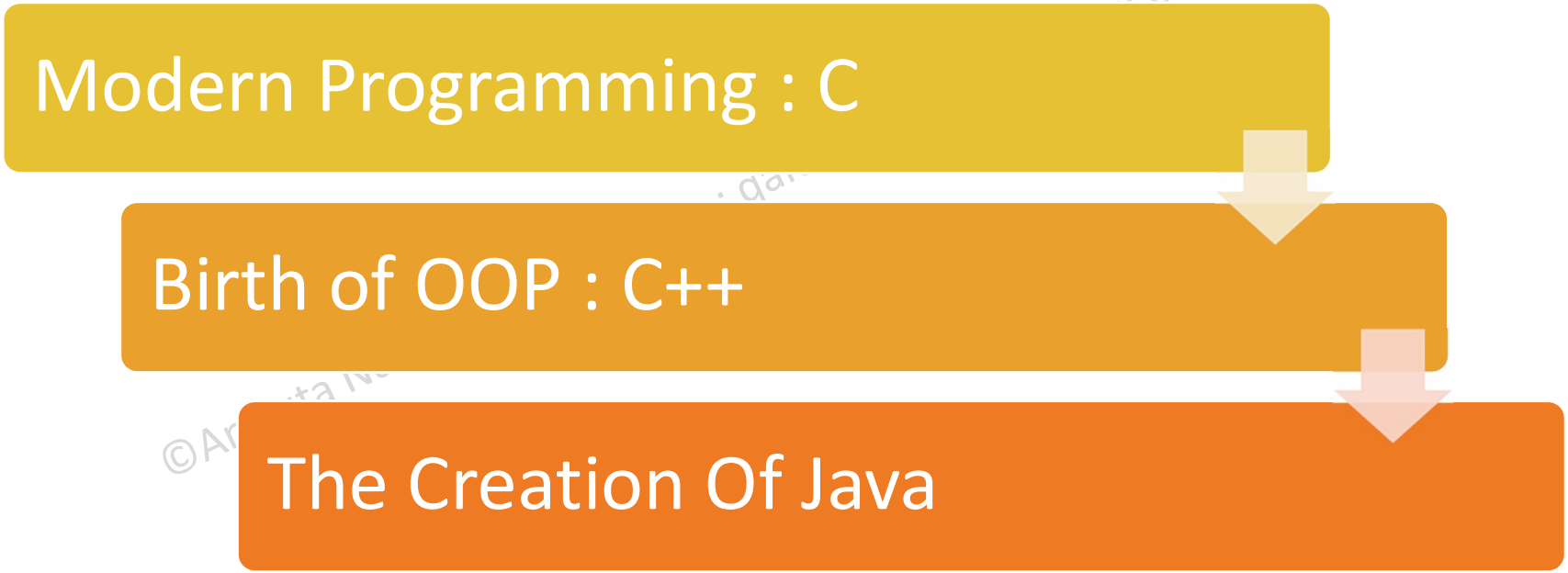
Core Java

-AMRUTA NADGONDE

Chapter 1

INTRODUCTION TO JAVA

History Of Java



Object-Oriented Programming

An approach to problem-solving where all computations are carried out using objects

What is an Object?

- Object is the basic unit of object-oriented programming
- Component of program that performs certain actions and interact with other elements
- Has well defined structure (properties) and behavior (methods)

What is a Class?

- A class is a blueprint of an object.

Examples - C#, Java, Perl and Python.

What is an Object

A real world entity with well-defined structure and behavior.

Definition:

- An object represents an individual, identifiable item, unit or entity, either real or abstract, with well-defined role in the problem domain.

Characteristics:

- State
- Behavior
- Identity
- Responsibility

Examples

- Person, car, pen, sound, instrument, back account, student, signal, transaction, medical investigation etc.

Java Class

- Class is a way of representing real world entity in the software domain.
- Class is a template for creation of objects.
- There is no memory associated with the class hence it cannot hold any values.
- An object is an instance of class. - One instance of a class can hold values for a specific entity.

Syntax

```
public class Class_Name {
        variable declaration;

        method declaration;

        public static void main(String [] args)
        {
                ...
        }
}
```

Key Concepts of OOP

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

Abstraction

- Process of identifying key aspects of an entity and ignoring the rest.
- Process of extracting essential/relevant details of an entity from the real world.
- Refers to both attributes (data) as well as behavioral (functions) abstraction.

- Consider ‘person’ as on object; abstraction of person would be different in different programming paradigm.

Social Survey	Health Care	Employment
Name	Name	Name
Age	Age	Age
Birth date	Birth date	Birth date
Marital Status		
income group		
Address	Address	Address
Occupation	Occupation	Occupation
	Blood Group	
	Weight	
	Prev records	
		Qualification
		Department
		Experience

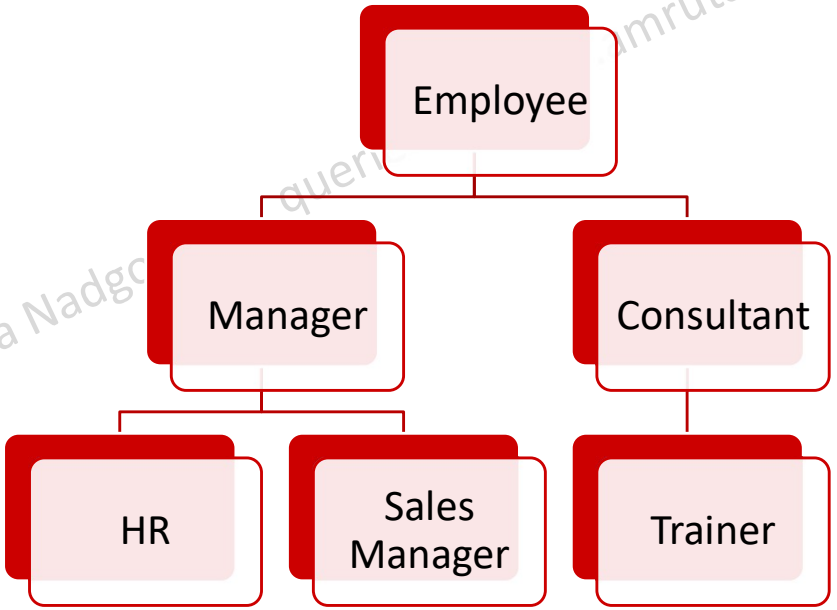
Encapsulation

- The mechanism used to hide the data, internal structure and implementation details of an object.
- Internal data can be accessed only through a public interface – Methods
- Ensures security of Data
- Separates interface of abstraction from its implementation
- Even if implementation is changed user need not to worry as long as interface remains unchanged.

Example – TV remote

Inheritance

- Process by which an object can acquire properties of other object.
- “is-a” a kind of hierarchy.



Generalization

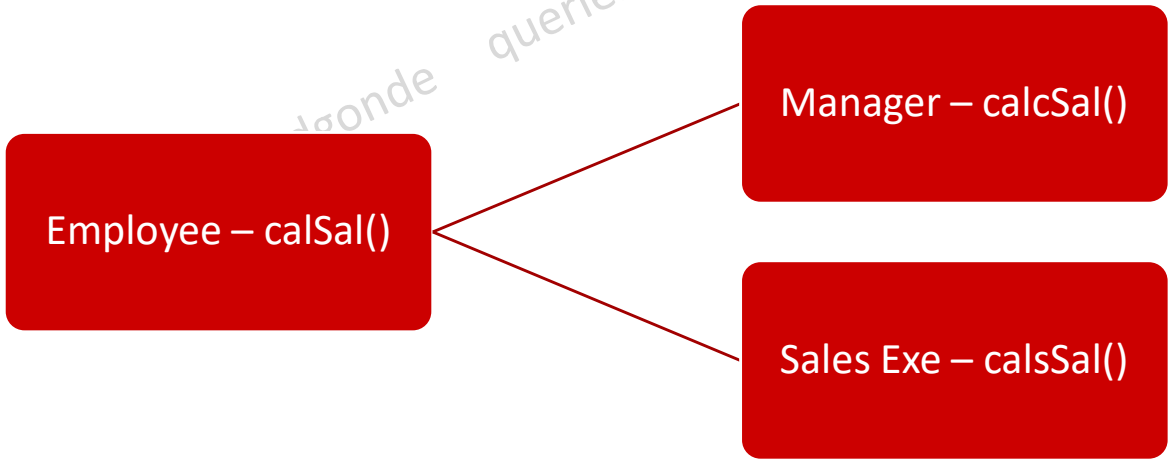
- Factoring out common elements within a set of categories into a more general category called super-class
- Moving up in the hierarchy is said to be generalization. The topmost class in the hierarchy is the most general class.
- Needs good skill of abstraction.

Specialization

- Allows capturing of specific features of a set of object.
- While moving down in the hierarchy more and more specific features are added in each sub-category that is formed.

Polymorphism

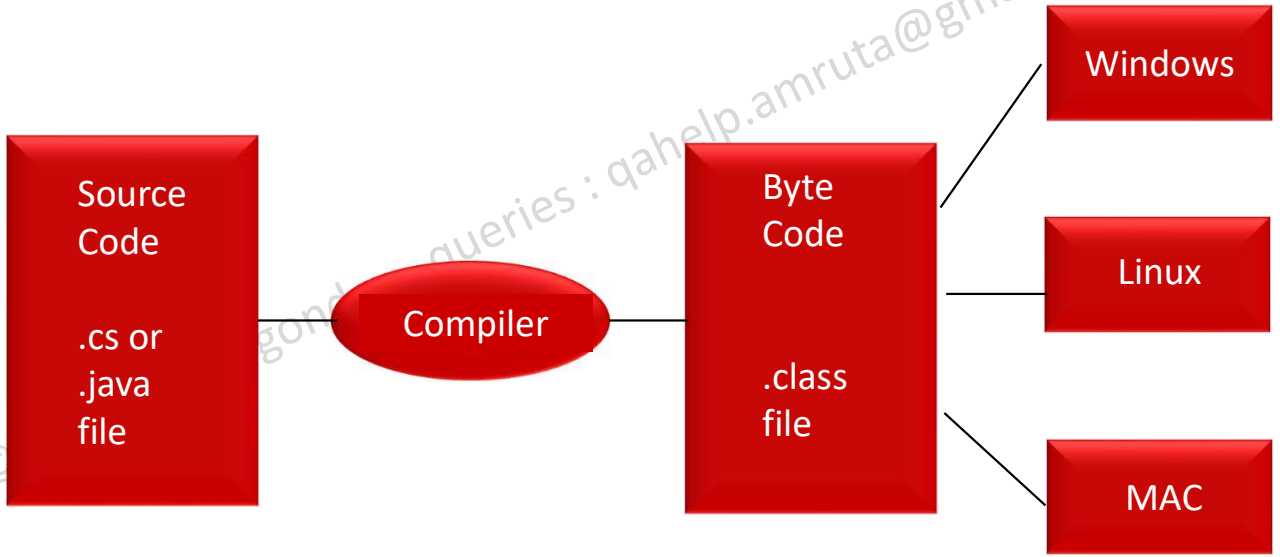
- The ability of different type of related objects to respond to the same message in their own ways.
- Poly -> Many and Morph -> Form
- One command can invoke different implementation for different related objects.



Features of Java

- Object Oriented
- Simple
- Robust
- Distributed
- Secure
- Architecture Neutral
- Interpreted
- Multithreaded

Platform Independence

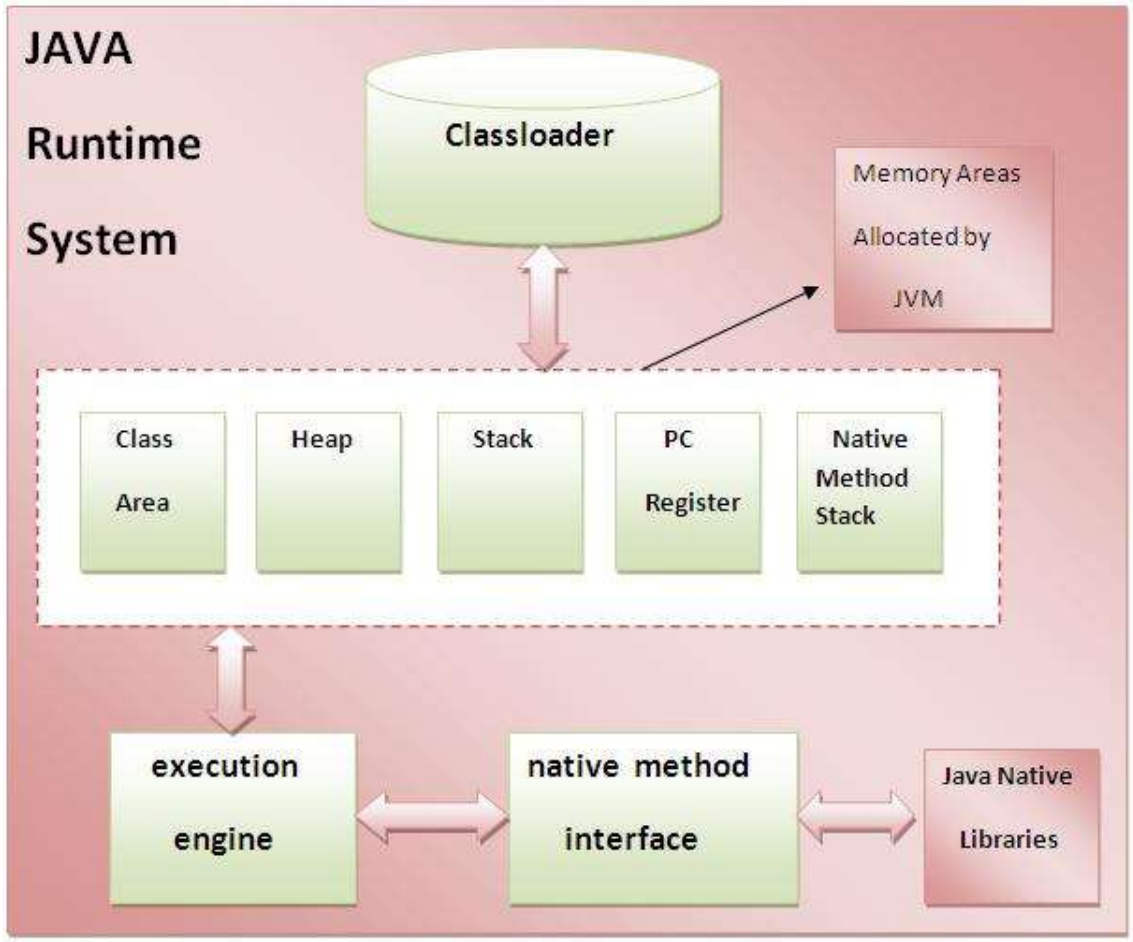


Platform Independence

- Software application created in many different languages.
- Compilers convert high level language source code in machine understandable machine code
- Every platform requires specific machine language makes it difficult to port application.
- Java and .Net provides solution:
  - Language specific compiler translates source code in some specific pattern independent of any platform.
  - Second component then translate this non executable code into executable instruction set specific to platform.
- Allows Java to be described as “Compile once, run anywhere” programming language.

Java Virtual Machine (JVM)

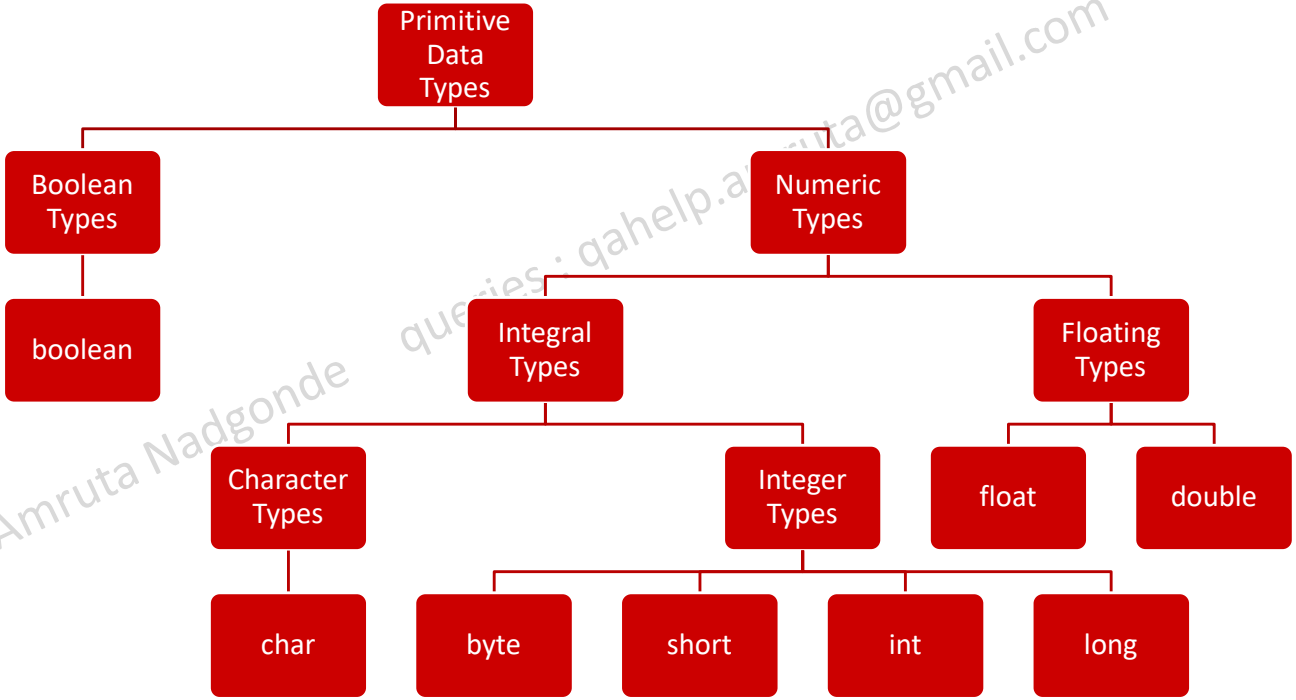
- A piece of software that is implemented on non-virtual hardware and on standard OS.
- Provides runtime environment in which java byte code can be executed.
- JVMs are available for many hardware and software platforms.
- Distributed along with a set of standard class libraries that implement the Java API
- The JVM performs following operation:
  - Loads code
  - Verifies code
  - Executes code
  - Provides runtime environment


JAVA
Runtime
System
Classloader
Memory Areas
Allocated by
JVM
Class
Area
Heap
Stack
PC
Register
Native
Method
Stack
execution
engine
native method
interface
Java Native
Libraries

Java Primitive data types

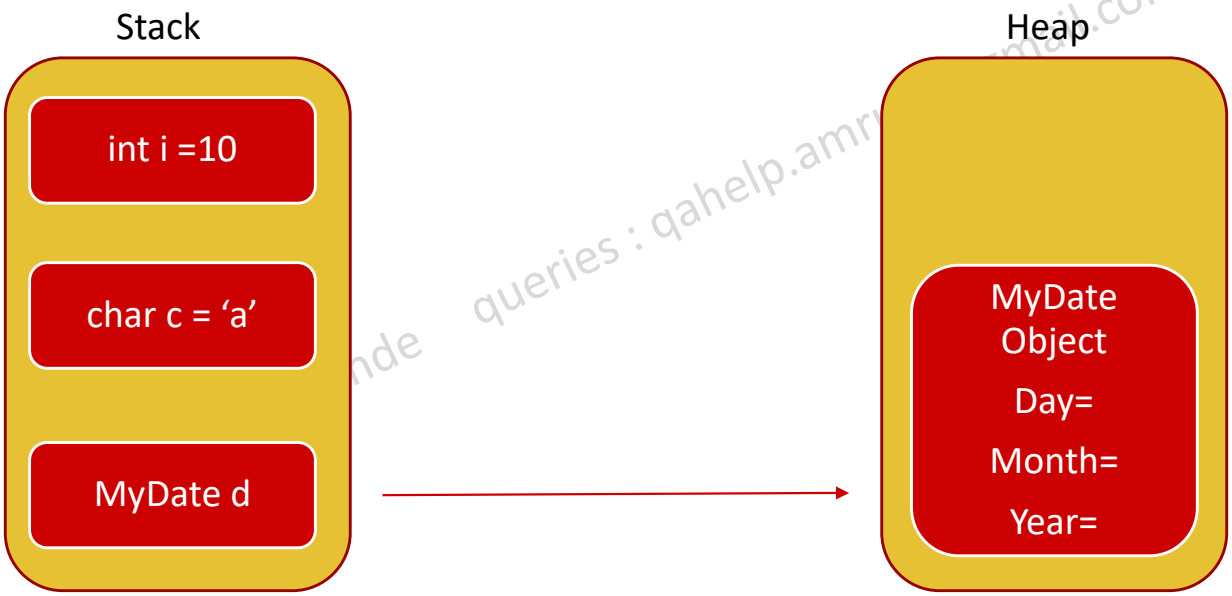
- The Java programming language is statically-typed.
- Variables must be declared before used
  - int i =1;
- Not mandatory to initialized the variable. But a good practice
- Compiler will set default value for such variables.
- Default is NULL or Zero ‘0’

Primitive Data Types



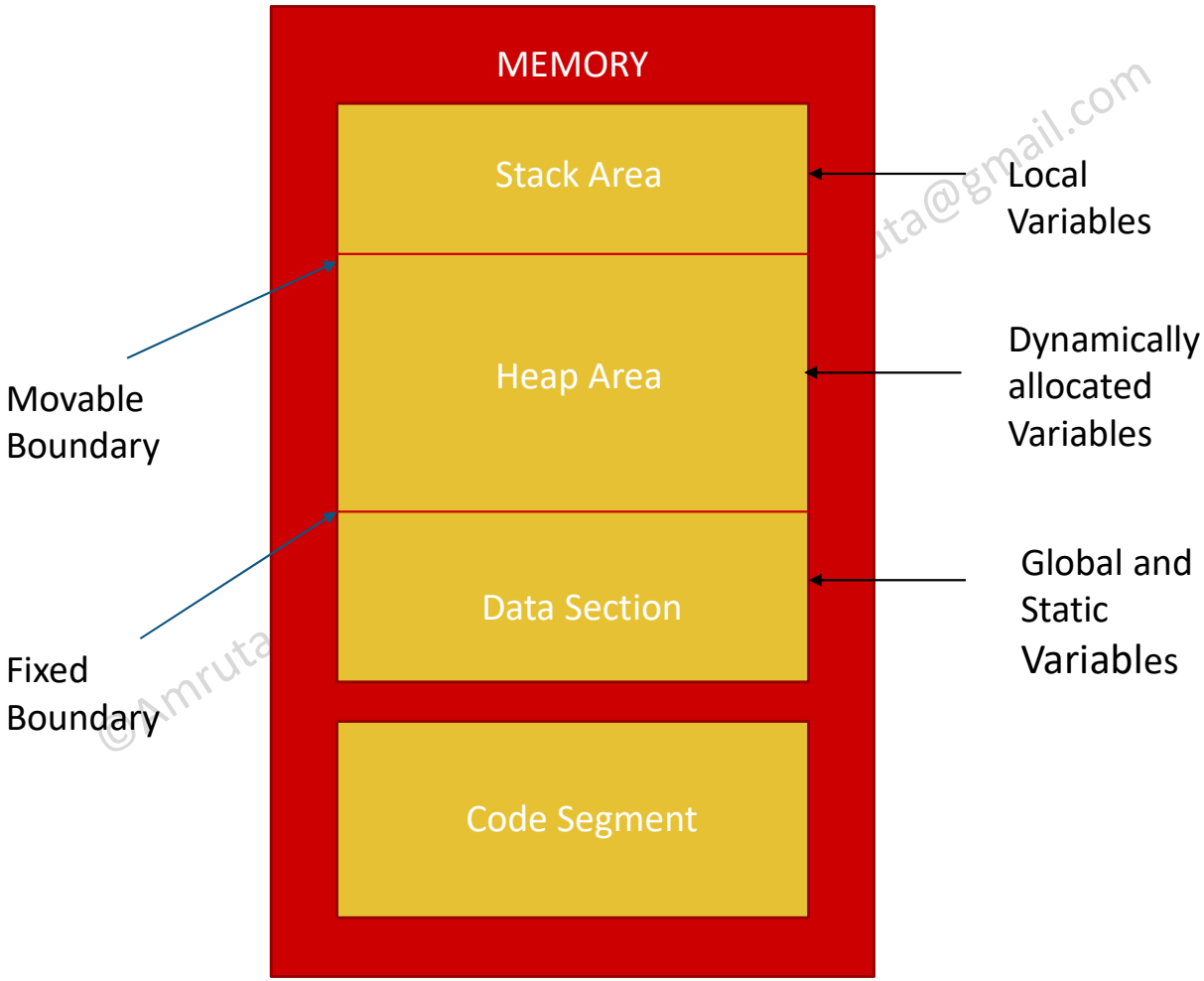
Type	Memory requirement	Example
Byte	1 byte	byte i =2
Short	2bytes	short r = 100
Int	4 bytes	int i= 5000
Long	8 bytes	long l= 10000L
Float	4 bytes	float f= 10.5f
Double	8 bytes	double d= 123.45d
Char	2 bytes	char a = 's'
Boolean	1 bit	boolean flag = true

Primitives vs non primitives



Java Memory Management

- The process of allocating new objects and removing unused objects to make space for those new object allocations.
- Exe loaded in memory is organized in 2 parts- Data segment and Code segment.
- Java objects reside in an area called the *heap*.
- Heap is divided into two parts – Nursery(young generation) and old generation.
- When the heap becomes full, *garbage* is collected.


MEMORY
Stack Area
Heap Area
Data Section
Code Segment
Local Variables
Dynamically allocated Variables
Global and Static Variables
Movable Boundary
Fixed Boundary

Garbage Collection

- Heap is divided into two parts – Nursery(young generation) and old generation.
- Nursery is for allocation of new objects.
- Young Collection - where all objects that have lived long enough in the nursery are *promoted*(moved) to the old space.
- Young collection frees up nursery for new objects.
- Old Collection – When old space becomes full garbage is collected there.
- This frees up memory faster than garbage collection of a single-generational heap (a heap without a nursery).

Mark and Sweep/Compact Method

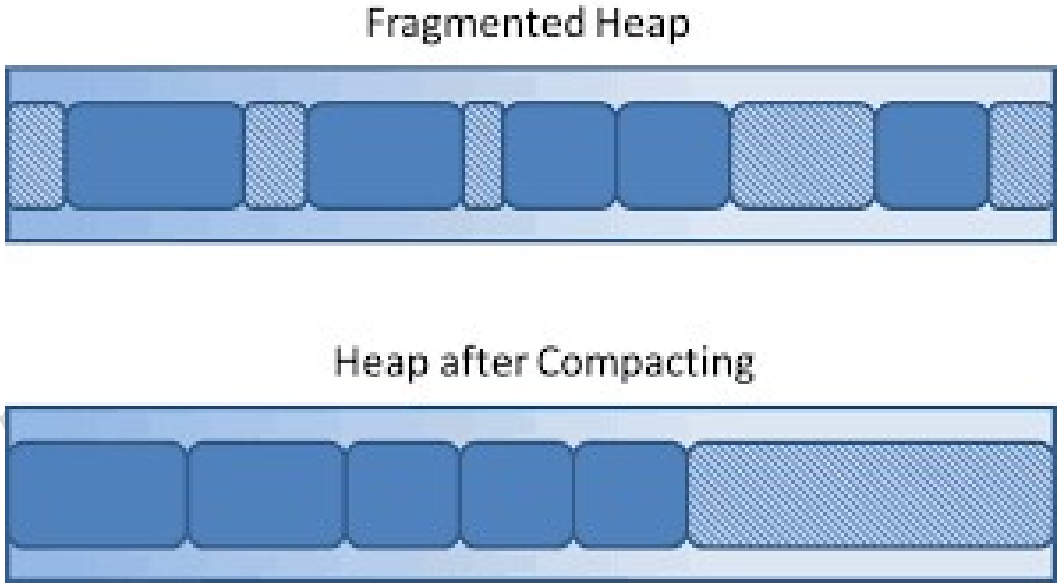
Mark Phase:

- Identify all objects that are reachable from Java threads, native handles and other root sources as well as the objects that are reachable from these objects and so forth.
- Rest of the objects can be considered garbage.
- Finds the memory that can be reclaimed.

Sweep/Compact Phase:

- Heap is traversed to find the gaps between the live objects. These gaps are recorded in a *free list.*
- Move all the live objects to the bottom of the heap, leaving free space at top.

Mark and Sweep/Compact Method



Chapter 2

JAVA BASICS

Operators in Java

- Arithmetic Operators
- Relational Operators
- Bitwise Operators
- Logical Operators
- Assignment Operators
- Misc Operators

Arithmetic Operators

Operator	Operation
+ (Addition)	Adds values on either side of the operator
- (Subtraction)	Subtracts right hand operand from left hand operand
* (Multiplication)	Multiplies values on either side of the operator
/ (Division)	Divides left hand operand by right hand operand
% (Modulus)	Divides left hand operand by right hand operand and returns remainder
++ (Increment)	Increases the value of operand by 1
-- (Decrement)	Decreases the value of operand by 1

Relational Operators

Operator	Operation
== (equal to)	Checks if the values of two operands are equal or not, if yes then condition becomes true.
!= (not equal to)	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.
> (greater than)	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.
< (less than)	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true
>= (greater than or equal to)	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.
<= (less than or equal to)	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.

Bitwise Operators

- These are applied on long, int, short, char, and byte.
- These operators work on bits and performs bit-by-bit operation.

- Assume a= 60, b= 13 then

  a = 0011 1100

  b = 0000 1101

  ----

  a&b = 0000 1100

  a|b = 0011 1101

  a^b = 0011 0001

  ~a = 1100 0011

Logical Operators

- These operators perform operations on Boolean variables/conditions.

Operator	Operation
&& (logical and)	If both the conditions are true, then the condition becomes true
|| (logical or)	If any of the two conditions is true, then the condition becomes true.
! (logical not)	Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.

Assignment Operators

Operator	Operation
='	Simple assignment operator, Assigns values from right side operands to left side operand.
+='	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand. Example: C += A is equivalent to C = C + A
-='	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand.
*='	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operan
/='	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand
%='	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand

Unary Operator

Operator	Operation
minus(-)	Used to convert a positive value to a negative one
NOT (!)	Used to convert true to false or vice versa
post increment (val++)	Increments and then uses the variable
pre increment (++val)	Uses and then increments the variable.
post decrement (val--)	Decrements and then uses the variable
pre decrement (--val)	Uses and then decrements the variable.

Java Loop Controls

- Used when a statement or group of statements need to be executed repeatedly.
- Control structures allow for more complicated execution paths.
- Loop statement allows execution of code multiple times depending on the result of a condition.
- Java loop controls:
  - While loop
  - Do…while loop
  - For loop

While loop

- Repeatedly executes a target statement as long as a given condition is true.

```
while(Boolean_expression)
{
        //Statements
        Update;
}
```


- Loop will be executed as long as expression result it True.
- When the condition becomes false, program control passes to the line immediately following the loop
- If condition is false at the beginning then loop will not run at all

Do...While loop

- Similar to While loop except that a do...while loop is guaranteed to execute at least one time.

```
do
{
        /Statements
}       while(Boolean_expression);
```


- Loop is executed as long as condition is True.
- Even if condition is false at the beginning then loop, the loop will be executed once before passing the control to the line immediately following the loop

For Loop

- Repetition control structure used to efficiently write a loop that executes specific number of times.
- Useful when you know how many times a task is to be repeated.

```
for(initialization; Boolean_expression; update)
{
        //Statements
}
```

Decision Making

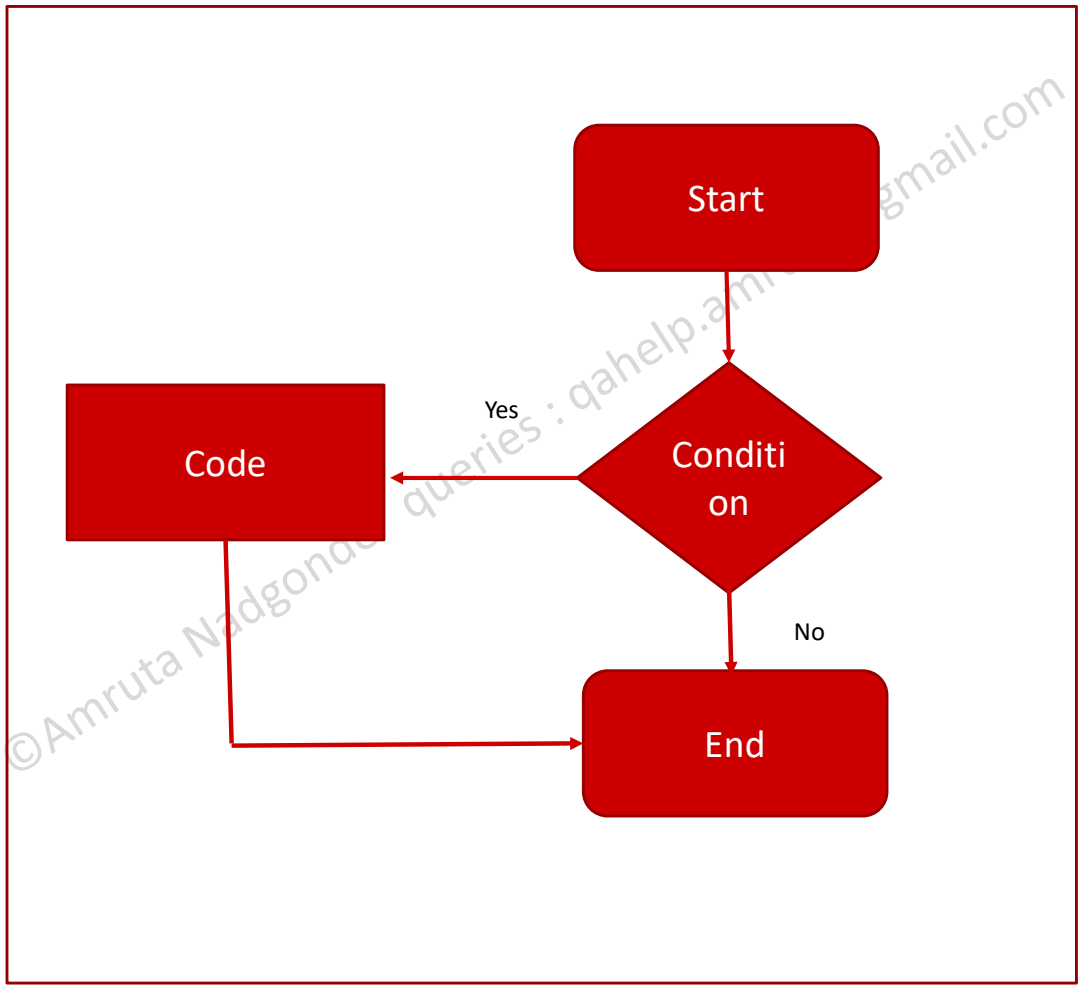
- It involves evaluation of a condition and execution of statement or group of statements depending on result True/False
- Java conditional statements:
  - if statement
  - if…else statement
  - nested if…else statement
  - switch statement

if Statement

- Consists of a Boolean expression followed by one or more statements.

```
if(Boolean_expression)
{
   //Statements will execute if the Boolean expression is true
}
```

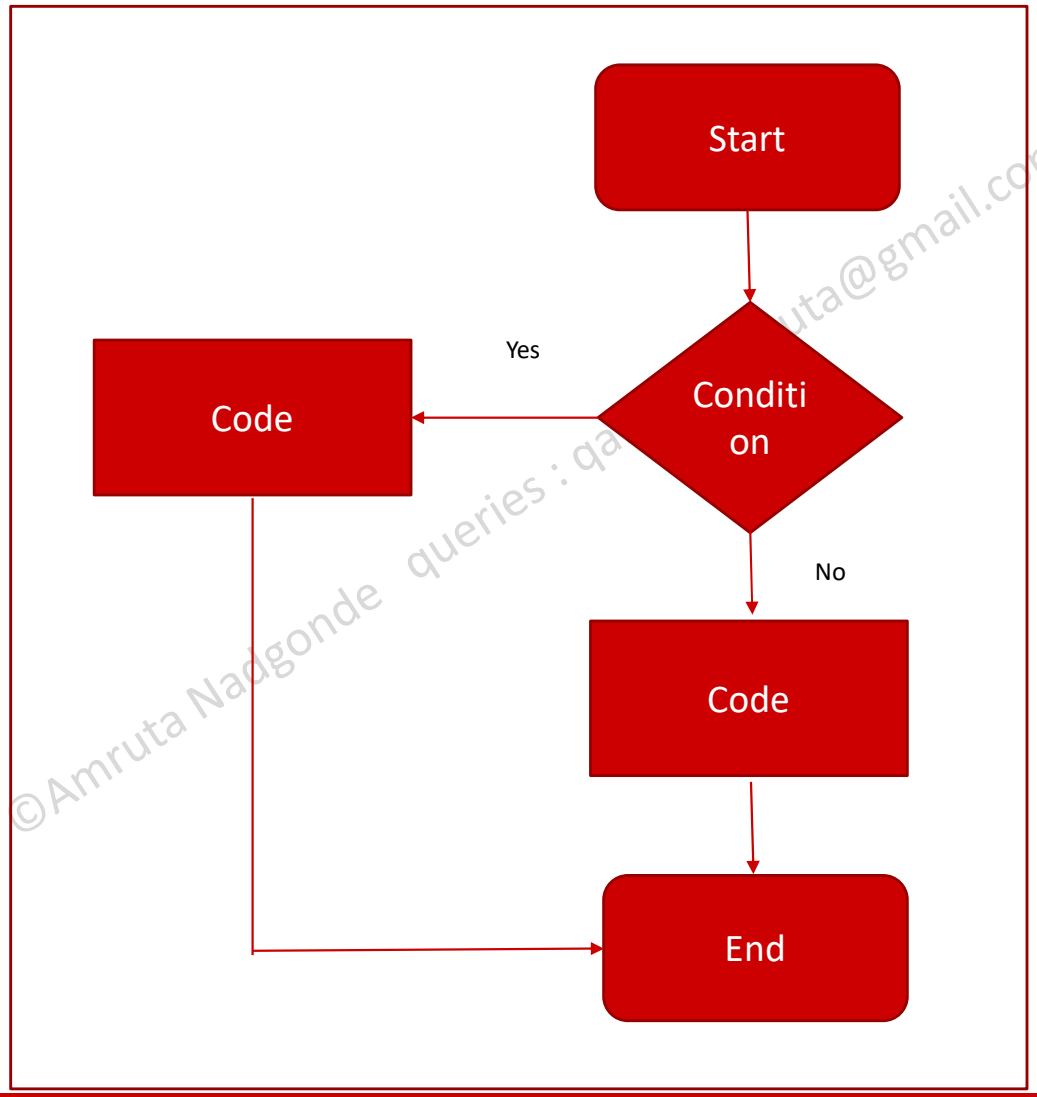
- Code block will be executed if Boolean_expression evaluates to True
- Control transfers to immediate next line if result is False.



If...else Statement

- An **if** statement can be followed by an optional **else** statement, which executes when the Boolean expression is false.

```
if(Boolean_expression)
{
  //Executes when the Boolean expression is true
}
Else
{
  //Executes when the Boolean expression is false
}
```

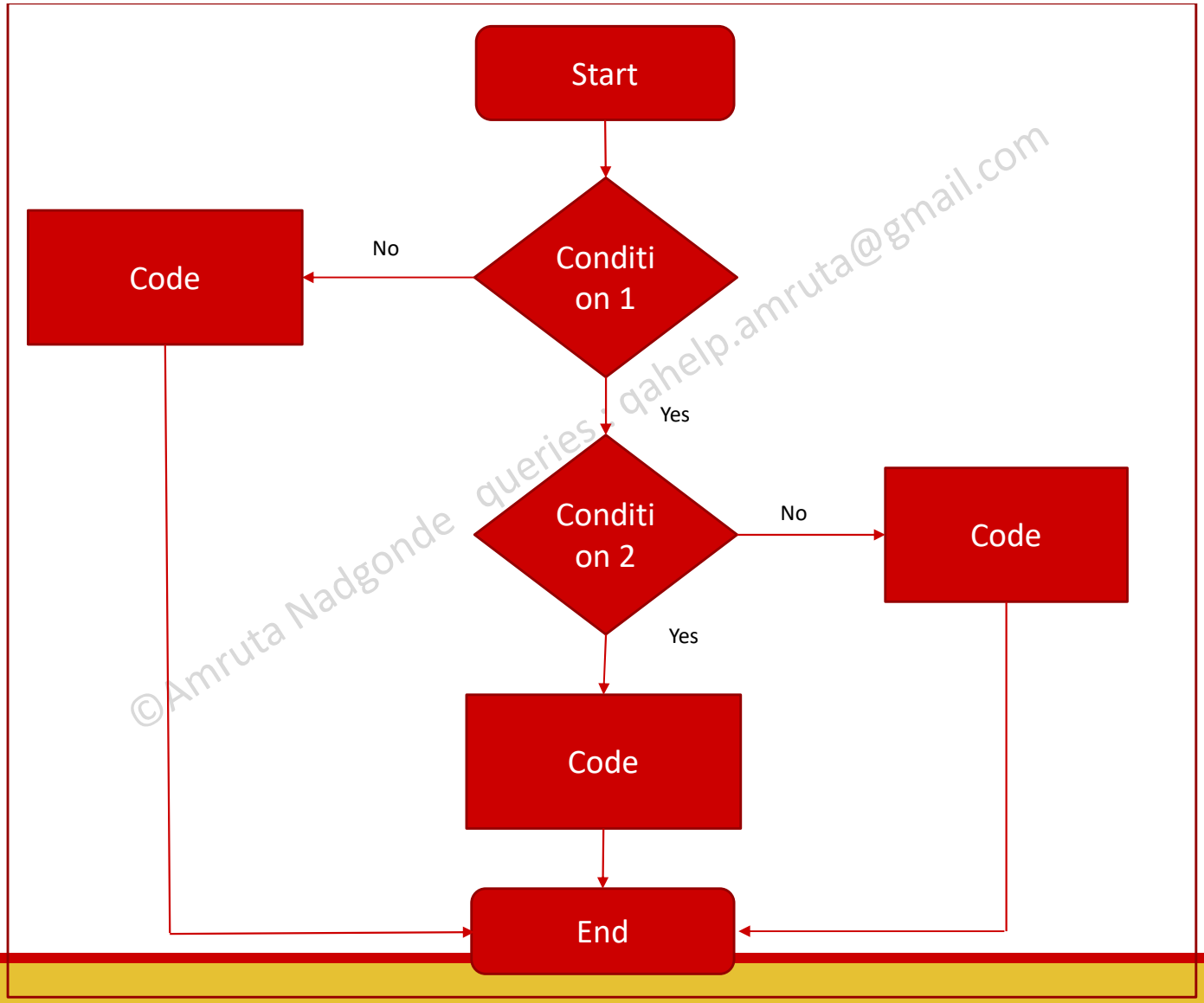


Nested if...else Statement

- One if or else if statement can be used inside another if or else if statement.

```
if(Boolean_expression 1)
{
  //Executes when the Boolean expression 1 is true
  if(Boolean_expression 2)
  {
    //Executes when the Boolean expression 2 is true
  }
}
```


- **else if...else** can be nested in the similar way.

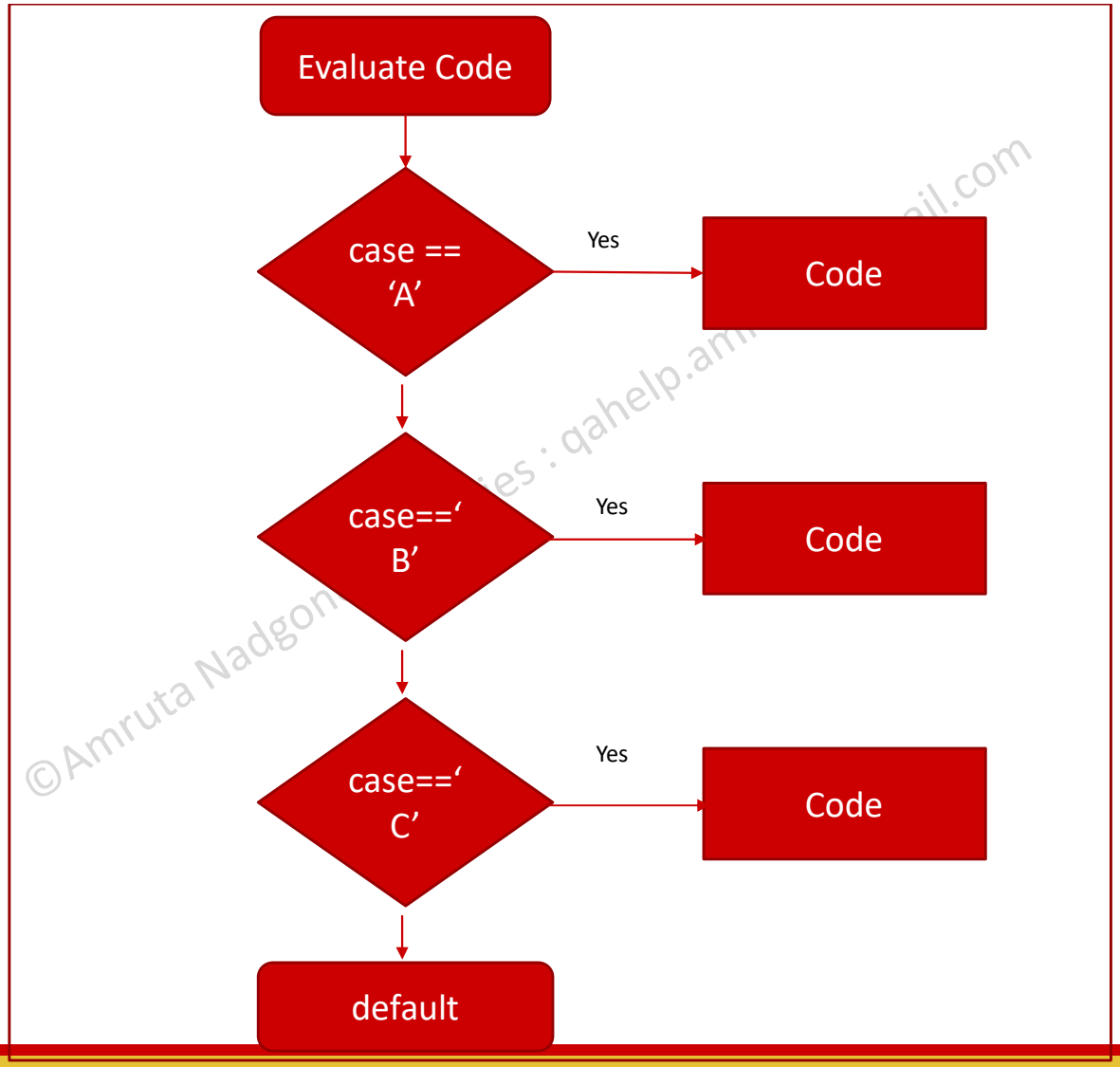


switch case statement

- A variable to be tested for equality against a list of values – case

```
switch(expression)
{
   case value :
        //Statements
        break; //optional
   case value :
        //Statements
        break; //optional
        //You can have any number of case statements.

   default : //Optional
       //Statements
}
```


Evaluate Code
case == 'A'
Yes
Code
case=='B'
Yes
Code
case=='C'
Yes
Code
default

Switch Case

- The variable used in a switch statement can only be integers, convertible integers (byte, short, char), strings and enums
- You can have any number of case statements each case is followed by the value to be compared to and a colon.
- The value for a case must be the same data type as the variable in the switch.
- When the variable being switched on is equal to a case, the statements following that case will execute until a break statement is reached.
- When a break statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
- Not every case needs to contain a break. If no break appears, the flow of control will fall through to subsequent cases until a break is reached.
- A switch statement can have an optional default case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.

Access Specifiers

- Define scope of the class members – data as well as method.
- Data Encapsulation is achieved using access specifiers.
- It is recommended that data should be kept private as far as possible.

Specifier	Scope
Public	Visible to all
Default	All classes within a package
Protected	Classes in hierarchy regardless of package
Private	Only within the same class

Class Object

- An object is an instance (occurrence/example) of a class.
- Object is created using *'new'* keyword.

```
public static void main(String []args)
{
        MyDate d = new MyDate()
        //Or
        MyDate d;
        d = new MyDate();
}
```

Java Methods

- A collection of statements that are grouped together to perform an operation.
- Java provides number of in built methods.
- User can create own methods – user defined methods.
- Method may or may not have a return value.
- Method may have zero or more parameters.
- The void keyword allows us to create methods which do not return a value

Java Methods

```
modifier returnType  nameOfMethod (Parameter List)
{
 // method body
}
```


- **modifier:** It defines the access type of the method.
- **returnType:** Method may return a value.
- **nameOfMethod:** This is the method name.
- **Parameter List:** The list of parameters, it is the type, order, and number of parameters of a method.
- **method body:** The method body defines what the method does with statements.

Method Calling

- Method that returns a value and that doesn't return a value is called differently.
- When a program invokes a method, the program control gets transferred to the called method.
- Called method then returns control to the caller in two conditions, when:
  - Return statement is executed.
  - Reaches the method ending closing brace.
- Method that returns no value (void) is considered as a statement

Accessor and Mutator methods

- In Java privacy of Data is achieved by one of the key features - Data encapsulation.
- The way to force data encapsulation is through the use of accessor and mutators.
- Accessor method is used to return the value of a private field.
- A mutator method is used to set a value of a private field.
- Also called setter and getter methods.

Constructor

- A special member function used to initialize the values of the attributes of an object.
- This function is implicitly called when object is created
- Constructor name must be same as class name.
- Defining a constructor is not mandatory, in this case compiler provides a default parameter less constructor.
- There is no return type given in a constructor signature not even void
- There is no return statement in the body of the constructor.
- Constructors can be overloaded.

‘this’ Reference

- Whenever an object invokes any property or method *‘this’* reference is passed implicitly to them.
- ‘*this*’ points to the current object. It holds the reference of the current object
- In constructor ‘*this*’ keyword is used to differentiate between instance (class) and local (constructor) members.
- Use of ‘*this*’
  - Access the attributes of current object
  - Remove shadowing of instance members
  - Call the constructor of the same class.

Static variables

- Every object of class has it's own copy of data members.
- Some attribute/characteristic belong to class rather than to a specific instance/object.
- Value of such attribute/characteristic remain same for all the objects of a class.
- Such data members are declared a 'static'.
- Since only one copy is maintained for all the objects of the class it is also called as 'class' variable.
- Static data can be accessed using class name; it doesn't require an object to access it.
- E.g. *interstRate* in Accounts class or *count* in Employee class.

Static methods

- Java supports static methods to access static members.
- Static methods don't require object of class to invoke them. They can be invoked using following syntax:

  ClassName.methodName(args)

- Static method can be invoked directly in the same class where it is defined without any class name.
- Reference '*this*' is never passed to a static method.
- Class methods ***cannot*** access instance variables but instance methods can static variables.

The ‘main ()’ Method

- main()’ method is a ‘static’ method.
- It is called before instantiation of class.
- The class loader loads the class so it is Classname.main()
- This method has array of String as an argument called as command line argument.
- Any initial information can be passed to a class through this array.

‘toString()’Method

- toString() returns a string representation of the object.
- Generalized method, can be used when only class file is available.

```
public class MyDate
{       int day, month, year;
        public String toString()
        {
                return("Date is :" +day + "-" +month+"-"
+year);
        }
}
```

Method Overloading

Methods with same name but different parameters (in numbers, types) and the sequence in which they are passed to the methods is called method overloading.

The method calls are resolved at compile time using method signature.

Rules:

- The two methods should have different number of parameters or arguments passed.
- If the two methods have same number of parameters then sequence / type of parameters must be different.
- It is not sufficient for two methods to differ only in their return type.

varArgs

- Introduced in Java 5 to overcome a situation where number of parameter values are not fixed or they are going to increase.
- varArgs means variable arguments.
- To represent variable arguments use operator called as ellipses (…) 3 dots after data type.

```
public static void add(int...num) //can add any no of parameters
 {
                int sum =0;
                for(int i=0;i<num.length;i++)
                {
                        sum = sum +num[i];
                }
}
```

Chapter 3

LANGUAGE FUNDAMENTALS

Arrays

- Arrays are first class objects in Java.
- Array references are stored on stack where as actual array is stored on heap.
- Declaration

```
int arr[];
arr = new int[10];
        or
int [] num = new int[5];
        or
int[] num = {10,20,30,40,50};
```

Java Packages

- Is a group of similar types of classes, interfaces and sub-packages.
- Can be built-in or user-defined.

Advantages:

- Used to categorize the classes and interfaces so that they can be easily maintained.
- Provides access protection
- Removes naming collision

Importing java packages

Accessing the package from outside the package.

- import package.*;
- Import package.classname;
- fully qualified name.

Immutable Classes

- Immutability is the ability to keep the same state even after modifications. Once it is defined no one can alter it
- The objects of immutable class cannot be changed
- In order to make a class immutable
  - Declare class as final.
  - Make all properties as private final.
  - Do not declare setters. Only getters.
  - Declare all args constructor.
  - If there are custom nested objects in the class as properties, implement clone.
  - If there are other types of nested objects as properties, perform a deep copy.

Wrapper Classes

- Java views everything as an object.
- The primitive data types are not objects; they do not belong to any class.
- Sometimes, it is required to convert data types into objects in Java language.
- Wrapper classes are used to convert corresponding data type into an object.
- Wrapper classes provide a home for methods and variables related to the type.
- A wrapper class wraps (encloses) around a data type and gives it an object appearance.
- Wrapper classes include methods to unwrap the object and give back the data type

```
int x = 20;
Integer i = Integer.valueOf(x);
```

Primitive to Wrapper

```
int y = i.intValue()
```

Wrapper to Primitive

```
String S = "30";
int i = Integer.parseInt(s);
```

String to Primitive

```
String s = "30";
Integer i = Integer.valueOf(s);
```

String to Wrapper

Advantages

- To convert simple data types into objects, that is to give object form to a data type; here constructors are used.
- To convert strings into data types (known as parsing operations), here methods of type parseXXX() are used.
- They provide mechanism to 'wrap' primitive values in an object so that primitives can do activities reserved for the objects like being added to ArrayList, Hashset, HashMap etc. collection.

Auto-Boxing and Unboxing

- Auto boxing is a feature of J2SE 5.0
- Before 5.0 working with a primitive data types to wrapper required conversions and vice-versa.

```
int x= 10;
Integer i = x;
```

Auto boxing -> Primitive to Wrapper

```
int y = i;
```

Auto unboxing -> Wrapper to primitive

- Applies when passing parameters to methods that expects an object/primitive type or assigning variables to wrapper/primitive types

String Class

- ‘String’ is a predefined class of Java available in *java.lang* package.
- Generally, string is a sequence of characters. But in java, string is an object that represents a sequence of characters.
- String can be created:
  - By string literal
  - By new keyword

```
String s = “Welcome";
String s = new String("Welcome”);
```

String Class

- Strings are immutable; the content of a String cannot be changed once it is instantiated.

```
s = “Welcome”;
s = “back”;
```

- String Constant pool: JVM checks if string already exists in the pool, a reference to the pooled instance is returned; else a new string instance is created and placed in the pool.
- New keyword: JVM will create a new string object in normal heap memory

String Class Methods

- char charAt(int index)
- int compareTo(String str)
- int compareToIgnoreCase(String str)
- String concat(String str)
- boolean endsWith(String suffix)
- boolean equals(Object anObject)
- boolean equalsIgnoreCase(String anotherString)
- int hashCode()
- String toLowerCase()
- String toUpperCase()

String Class Methods

- String trim()
- int length()
- String replace(char oldChar, char newChar)
- String replaceAll(String regex, String replacement)
- String replaceFirst(String regex, String replacement)
- String[] split(String regex, [int limit])
- boolean startsWith(String prefix)
- String substring(int beginIndex)
- String substring(int beginIndex, int endIndex)
- char[] toCharArray()
- String toString()

StringBuffer Class

- Java StringBuffer class is used to created mutable (modifiable) string.
- StringBuffer class is thread-safe. It avoids data conflicts but decreases performance.
- Constructors:

```
StringBuffer sb = new StringBuffer()
StringBuffer sb = new StringBuffer(String str)
StringBuffer sb = new StringBuffer(int capacity)
```

- The buffer grows automatically as characters are added.

StringBuilder Class

- Java StringBuilder class is used to create mutable (modifiable) string.
- It is same as StringBuffer class except that it is non-synchronized.
- It is available since JDK 1.5.

Constructors:

```
StringBuilder  sb = new StringBuilder()
StringBuilder  sb = new StringBuilder(String str)
StringBuilder  sb = new StringBuilder(int length)
```