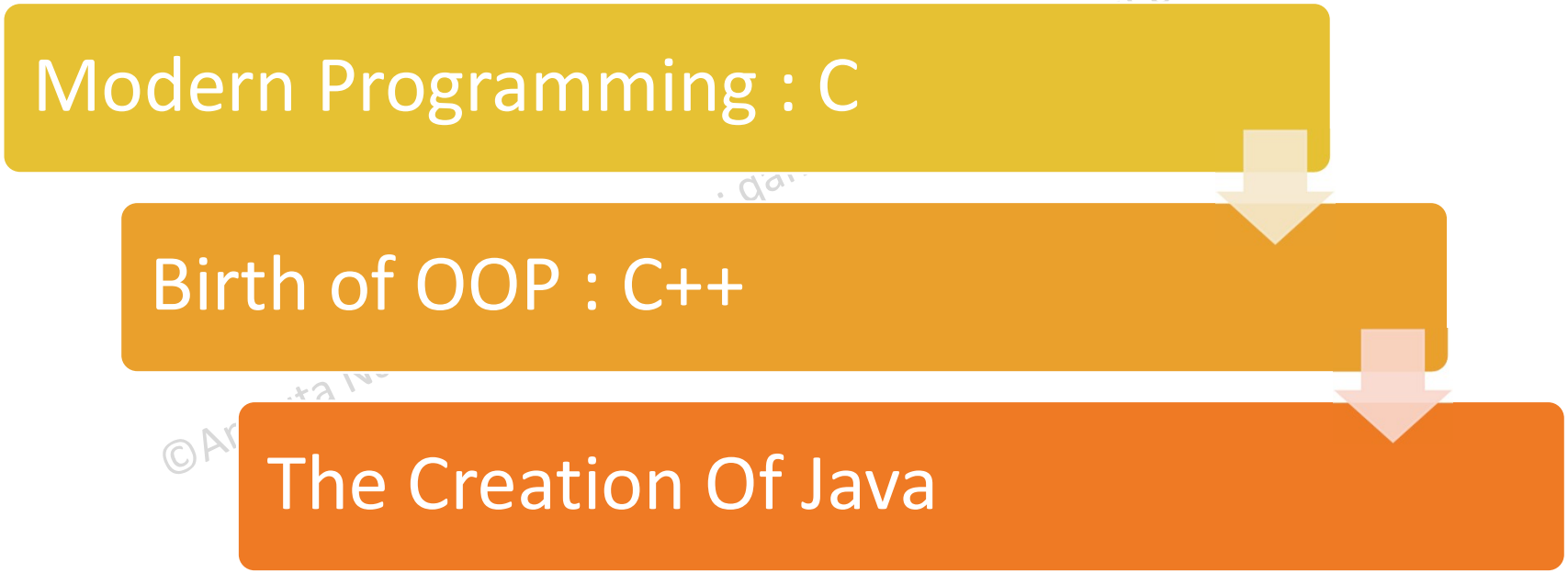
Core Java

-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

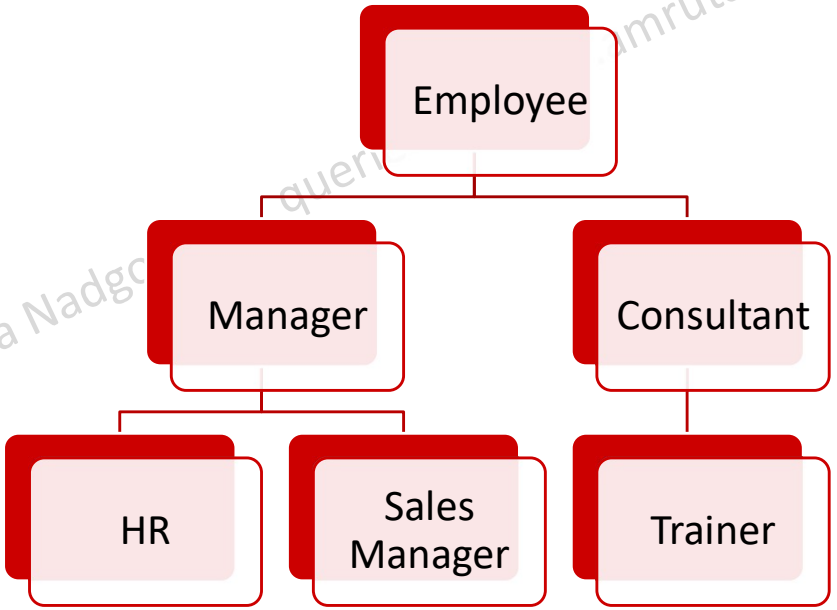
Encapsulation

- 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

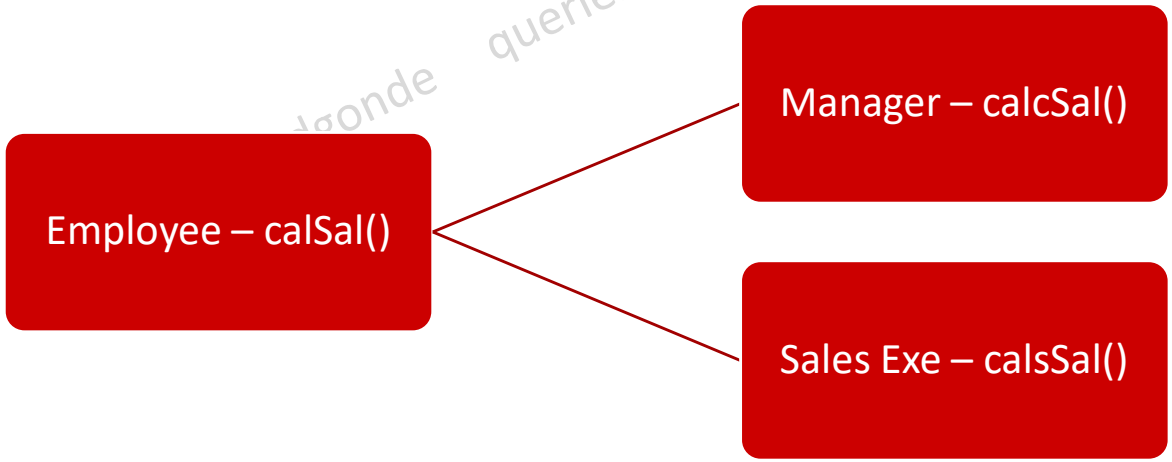
- Factoring out common elements within a set of categories into a more general category called super-class
- Moving up in the hierarchy is said to be generalization. The topmost class in the hierarchy is the most general class.
- Needs good skill of abstraction.

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

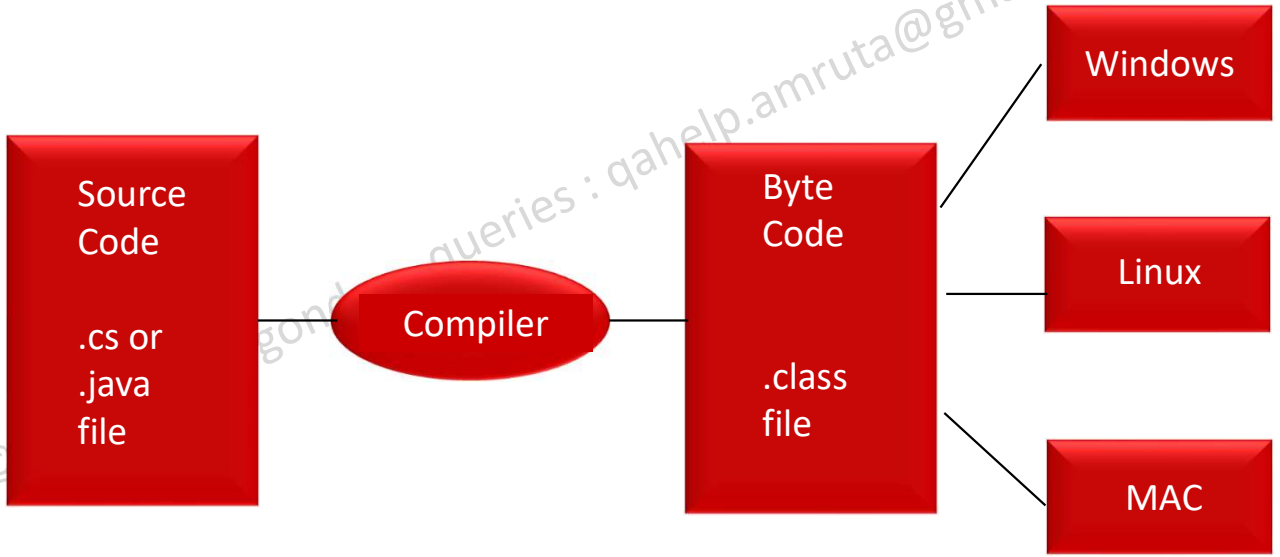
- The ability of different type of related objects to respond to the same message in their own ways.
- Poly -> Many and Morph -> Form
- One command can invoke different implementation for different related objects.



Features of Java

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

Platform Independence

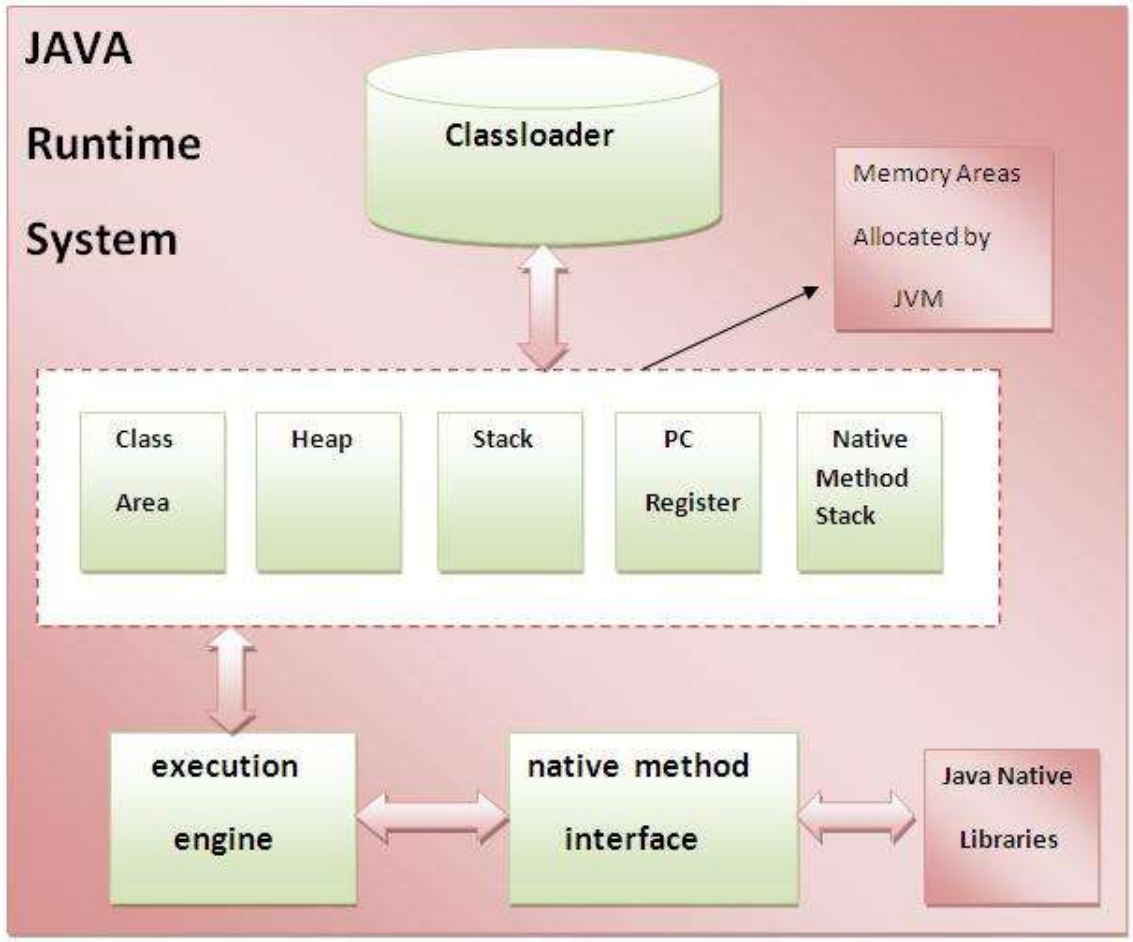


Platform Independence

- 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)

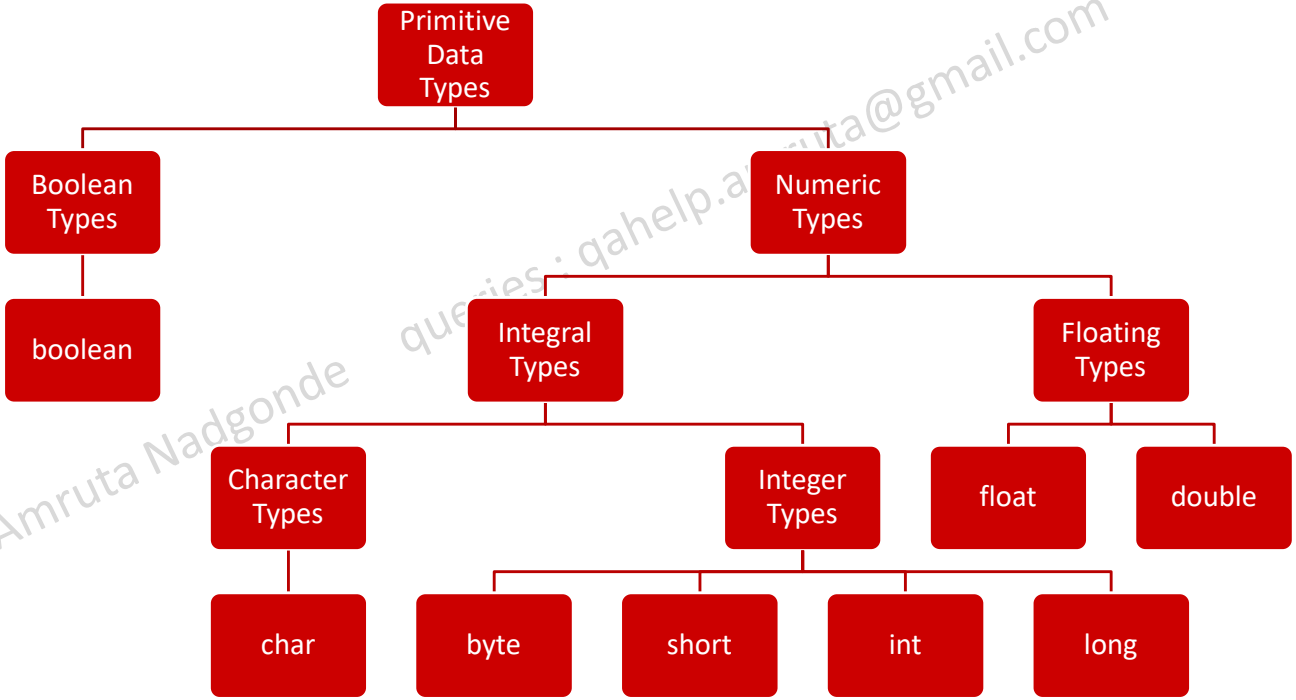
- A piece of software that is implemented on non-virtual hardware and on standard OS.
- Provides runtime environment in which java byte code can be executed.
- JVMs are available for many hardware and software platforms.
- Distributed along with a set of standard class libraries that implement the Java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

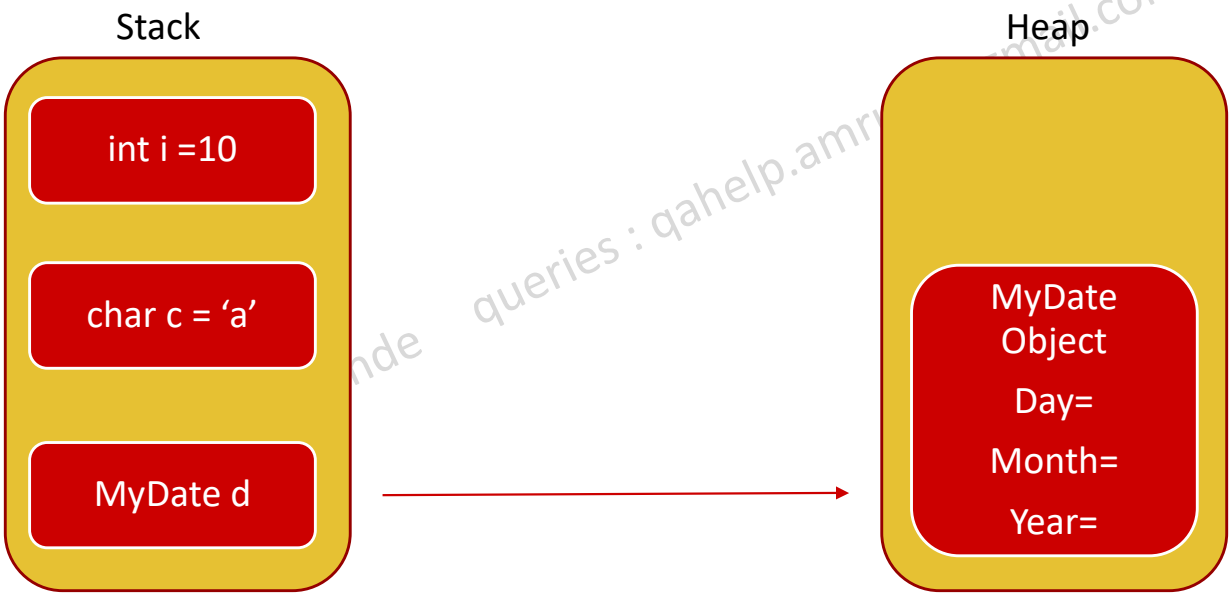
- The Java programming language is statically-typed.
- Variables must be declared before used
  - int i =1;
- Not mandatory to initialized the variable. But a good practice
- Compiler will set default value for such variables.
- Default is NULL or Zero ‘0’

Primitive Data Types



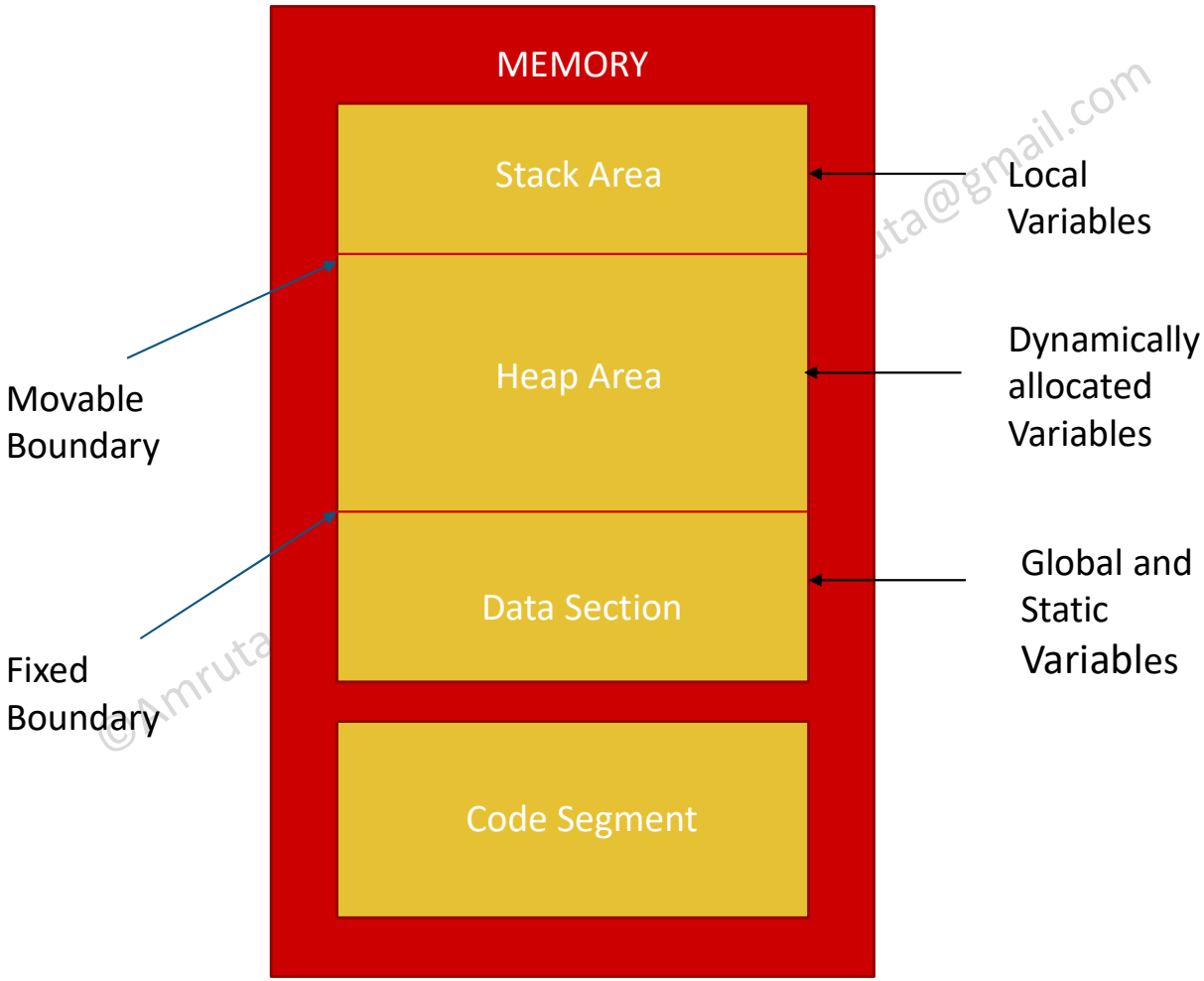
Type	Memory requirement	Example
Byte	1 byte	byte i =2
Short	2bytes	short r = 100
Int	4 bytes	int i= 5000
Long	8 bytes	long l= 10000L
Float	4 bytes	float f= 10.5f
Double	8 bytes	double d= 123.45d
Char	2 bytes	char a = 's'
Boolean	1 bit	boolean flag = true

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

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
}
```

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

Import and Classpath

- To make other classes available to the class you are writing import statement is used
  - import package.*;
  - Import package.classname;
- The .class files are kept in the subdirectories for package
- The Java Virtual Machine will know where to find your compiled class looking at the import statement, however the root directory must be a part of classpath
- Classpath should contains:
  - JAR files, and
  - Paths to the top of package hierarchies.

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)
```

Chapter 4

OOPS CONCEPTS

Inheritance

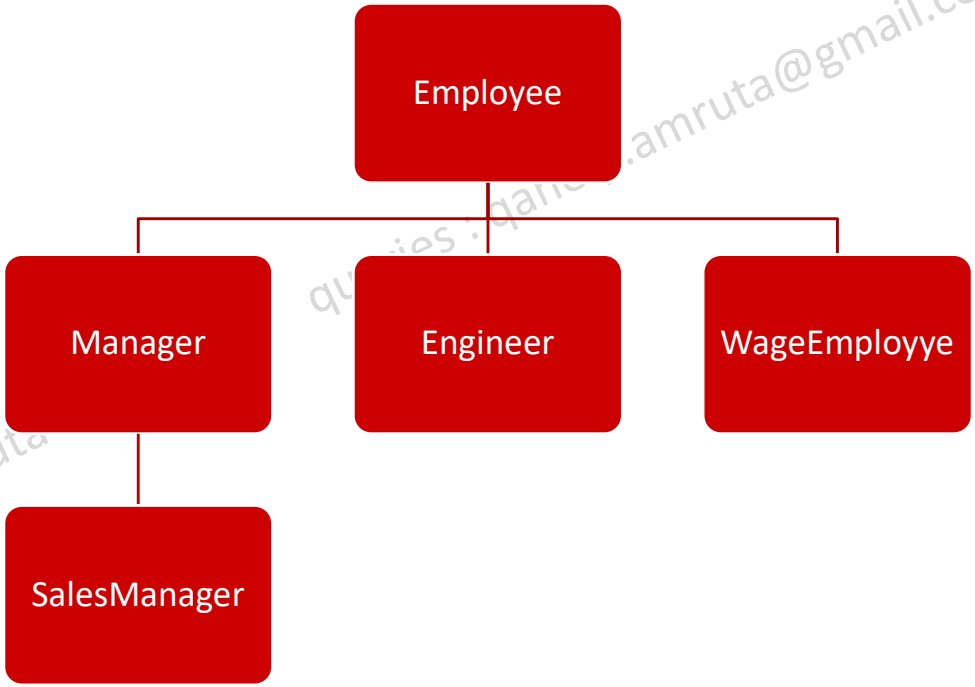
- One of the powerful features of object oriented programming.
- It is an “is-a” kind of a relationship.
- To inherit means to receive properties of already existing class.
- All data members and methods are inherited except the private fields.
- Hierarchy of classes is created using inheritance; new class (derived) is derived from an existing(base) class.

Advantages:

- Reusability: Once a class written and tested, it can be used to create new classes.
- Extensibility : As new classes can be derived from existing class ;this provides the extensibility of adding and removing classes in a hierarchy as and when needed.

Inheritance

is-a

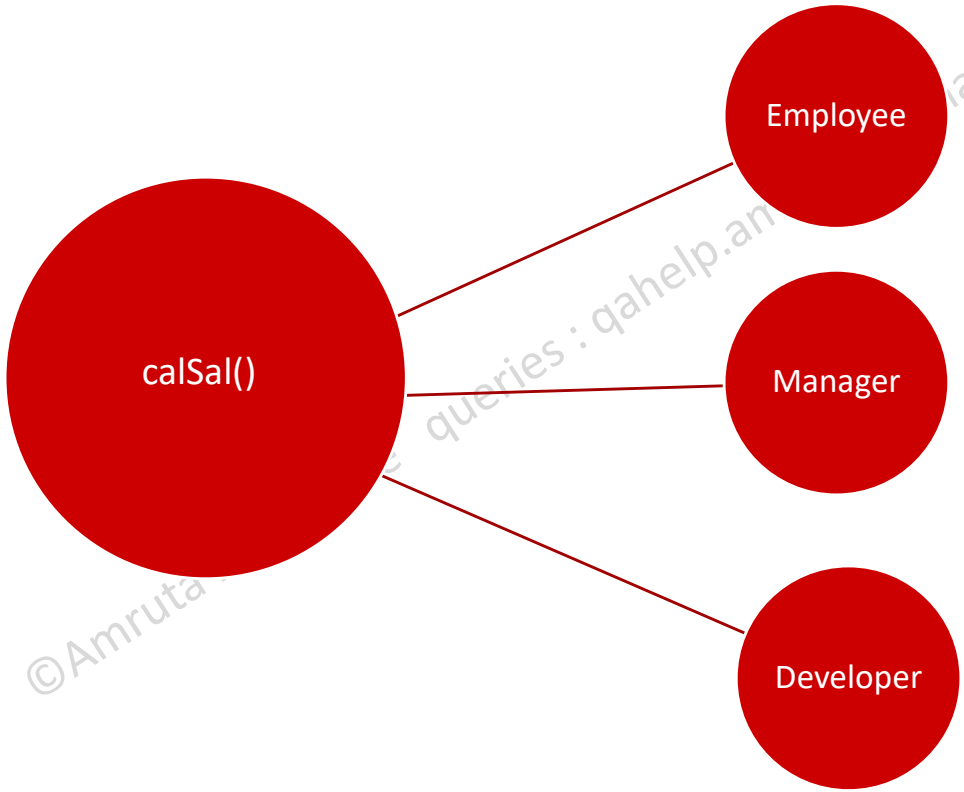


‘super’ Keyword

- When a class is derived from another class; member initialization of base class in derived class is achieved by using ‘***super’*** keyword.
- Constructor of the derived class first invokes the constructor of super class.
- If explicit call to parameterized constructor is not given default no argument constructor is called.
- ‘***super’*** keyword is used to call any non static method of the base class from the derived class.
- When ‘***super’*** *is used* for method invocation, it explicitly means base class implementation of the method.

Polymorphism

- "Poly" means many and "morph" means form.
- Ability to make more than one form is called polymorphism.
- In programming terminology, one command may invoke different implementation for different related objects.
- Allows different objects to share the same external interface although the implementation may be different.
- Can be achieved in two ways – Compile time polymorphism and run time polymorphism.



Compile time and Late Binding

- Binding is the process of associating a function call to an object
- **Compile time polymorphism (early binding)** – binding occurs at compile time. The argument passed to the method decides the method to be invoked.
- Achieved using **method overloading**.
- **Run time polymorphism** (late binding) - binding occurs at run time. The method to be invoked is decided based on the object used to invoke it.
- Achieved using **method overriding**

Method overriding

- Overriding is directly related to sub classing.
- Overriding is required to modify super class methods as required in sub class.
- Overridden methods should have same argument list of identical type and order else they are considered as overloaded methods.
- Return type of overridden method must be same else compiler generates an error.

Method Overloading vs Overriding

	Overloading	**Overriding**
Scope	In the same class	In the inherited classes
Purpose	Handy for program design	Specific implementation of Derived class
Signature of Methods	Different for each method	Has to be same in all the classes
Return Type	Can be same or different	Has to be same in all the classes

Up-casting and down-casting

- Up-casting: In inheritance super class reference can refer to a derived class object without needing a cast.

```
Employee e = new WageEmployee();

int sal = e.calSal(); //invokes derived class implementation
```

- Down-casting : In situations where specific methods of derived class need to be invoked using base class reference explicit casting is required; called as down-casting

```
WageEmployee we = (WageEmployee) e

int wages = we.calWages();          //specific method of WageEmployee
```

Covariant Return Types

- In method overriding, overridden method in subclass should have signature same as that of the super class.
- In Java 5 return type of a overridden method can be changed as long as the new return type is the subtype of the declared type.

```
class Employee{
        public Number getEmployees()
        {
                return EmployeeCount();
}

class SalesExe extend Employee{
        public Integer getEmployees()
        {
                return SalesExeCount();
        }  }
```

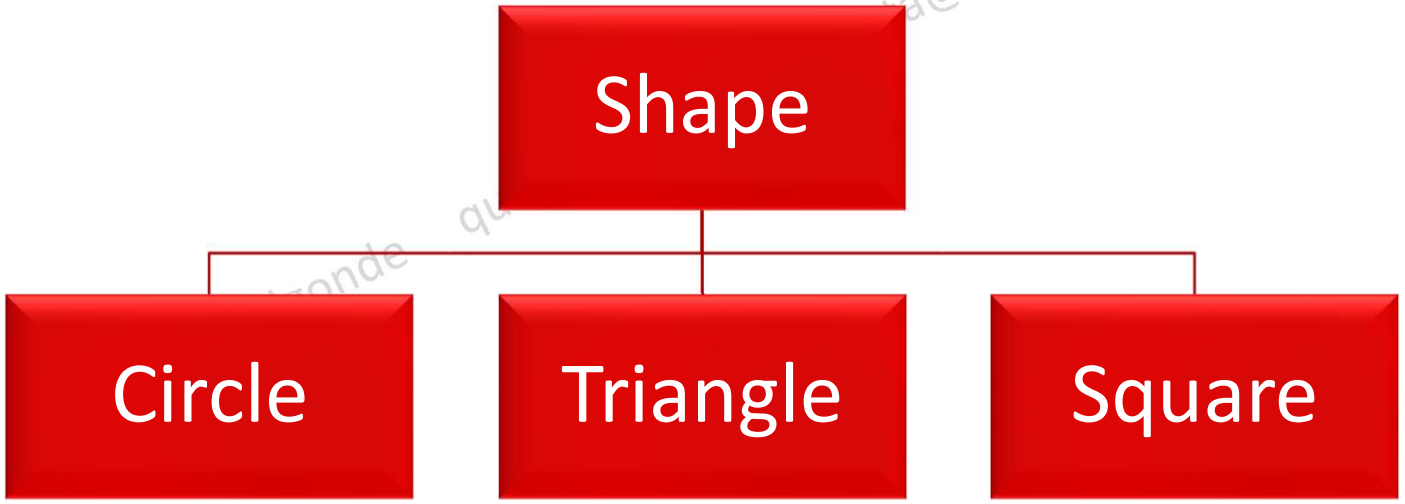
‘final’ Methods and Classes

- The main purpose of a final class/method is to take away inheritance feature from user so the class cannot be derived or the method cannot be overridden.
- ‘***final’*** keyword is used to make a class /method final
- Final method cannot be overridden in a subclass
- All methods in a final class are final by default. A final class cannot be sub classed.
- ‘private’ methods are final by default.
- In Java ‘String’ class is a final class.

Abstract Class

- Abstract Class: A class which contain generic/common features that multiple derived classes can share.
- An abstract method is a method that is declared without an implementation (without braces, and followed by a semicolon) :
- A class which has at least one abstract method is called abstract class. Other methods of the class can be abstract / non abstract.
- Object of abstract class can not be created but reference can be.
- When an abstract class is sub-classed, the subclass usually provides implementations for all of the abstract methods in its parent class. However, if it does not, then the subclass must also be declared abstract.

Abstract Class



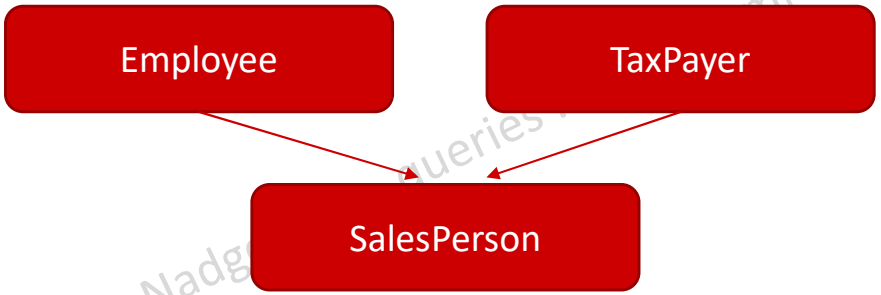
Abstract Class

- Abstract class cannot be instantiated
- Abstract class must contain at least one abstract method.
- Abstract methods do not have any implementation.
- One can create references so these classes support polymorphism
- Abstract classes are very useful during component creation because they allow specifying an invariant level of functionality in some method but leave the implementation of the other methods until a specific implementation of that class is needed.
- A class inheriting from an abstract class must provide implementation to all the abstract methods, else the class should be declared as abstract.
- Abstract modifier cannot be used for constructor

Interface

Need :

- It is possible for entities belonging to two different hierarchy sets to belong to same role:



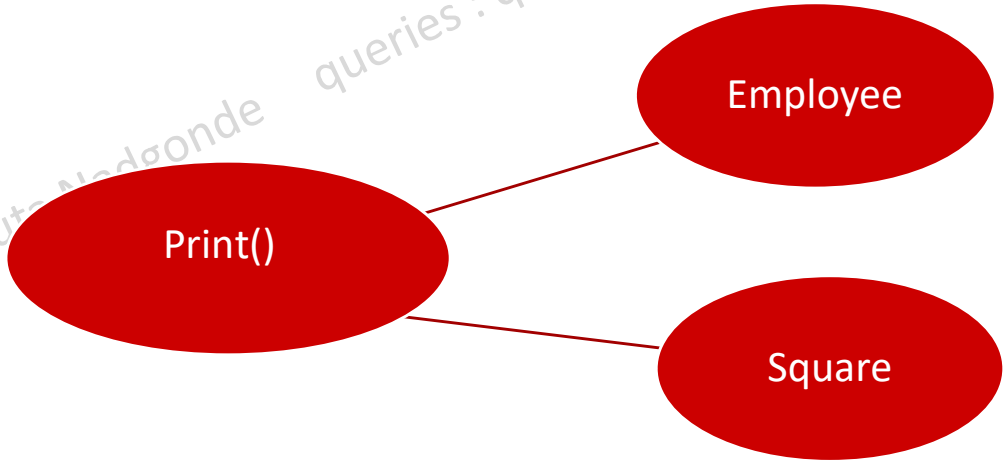
- Allows role-based inheritance.

Role → Interface

- To create loosely coupled applications

Interface

- Interface is a contract between the consumer and provider.
- It defines a standard set of properties, methods and events.
- Any class which implements the interface has to provide the implementation to all the members of the interface.



Interface

- All the methods declared in an interface are implicitly *abstract.*
- Methods in an interface are always *public*, they cannot be *static*. (static methods are always class specific).
- All fields in an interface are always *static* and *final*.
- A variable can be declared as an interface type, and all the constants and methods declared in the interface can be accessed using this variable.
- An interface can extend another interface just like a class can extend another class. However unlike inheritance classes can implement multiple interfaces.
- A class that implements the interface should give the implementation of all the methods declared in an interface otherwise, must declare as an abstract.
- '*implements'* is the keyword used to implement an interface. If class implements multiple interface, then it can be done using comma-delimited list.

Chapter 5

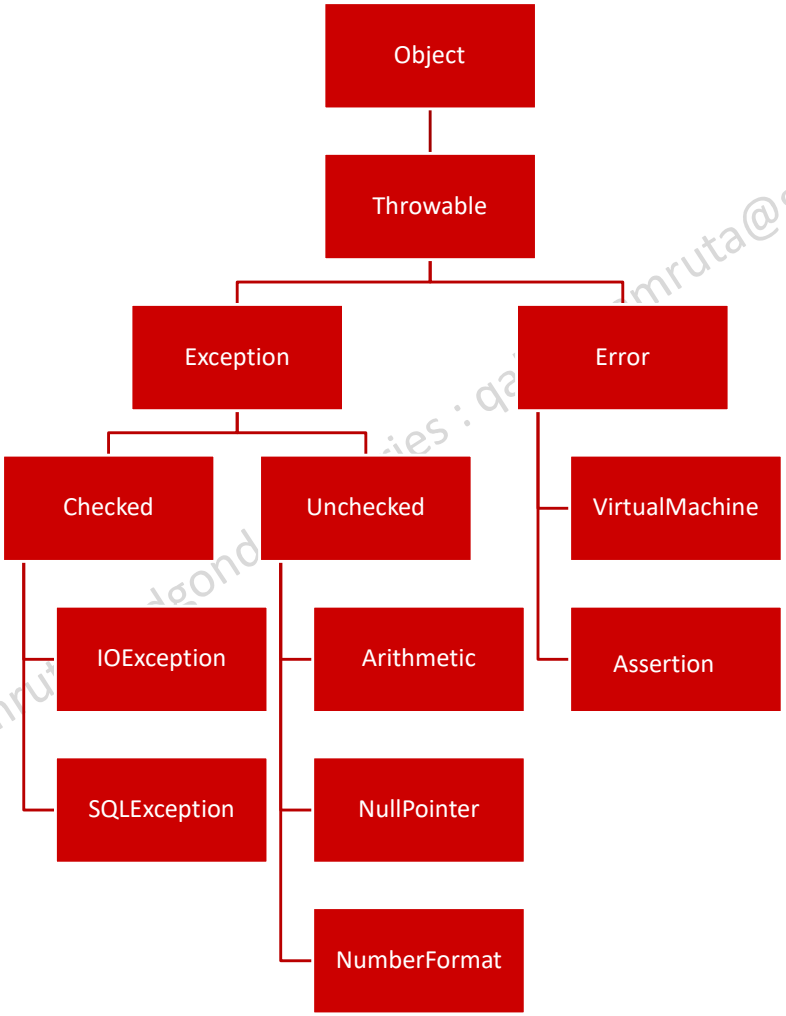
EXCEPTION HANDLING

Errors

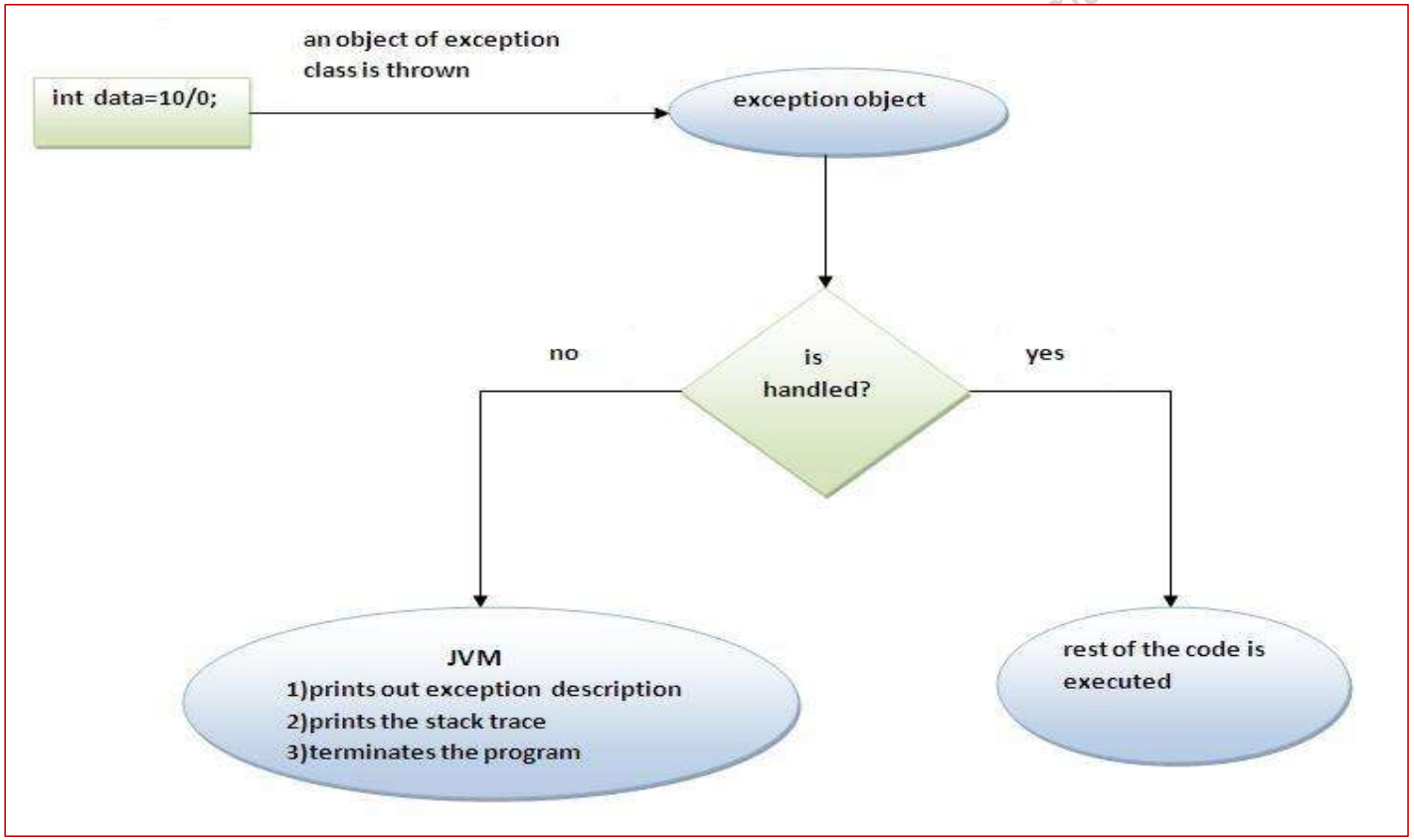
- Programs cannot be 100% error-free
- Logical errors:
  - Occur when there is something wrong with the logic used by the programmer.
  - Can be corrected by debugging and dry runs.
- Run Time errors
  - Occur at run-time, when the program is executed.
  - Can be controlled by predicting the error and checking appropriate condition.

Limitations of C-Style Error Handling

- No separate mechanism for error handling.
- Error values are checked using if..else/switch statements to take appropriate action and generation of error messages.
- Business logic gets secondary importance whereas error handling is given primary importance
- Java - structured way of handling these runtime errors.
- Exception handling - mechanism that provides a way to respond to run time errors by transferring control to special code called handler.
- Achieved using ‘try-catch’ blocks.


Object
Throwable
Exception
Error
Checked
Unchecked
VirtualMachine
IOException
Arithmetic
Assertion
SQLException
NullPointer
NumberFormat

Internal working of ‘try-catch’



Multiple ‘try-catch’ Blocks

- Single try-Multiple catch blocks to handle individual errors.
- ‘try’ block executes normally unless exception occurs.
- When exception occurs, the system searches for the nearest catch block which matches the type of exception and handles it.
- Object of *Exception* class catches all exceptions that are not handled by any catch block.
- All catch blocks must be ordered from most specific to most general.

‘finally’ Block

- Exceptions occurring in the application before cleaning up (such as closing connection, stream etc) must be handled.
- Clean-up code cannot be written in ‘try’ block.
- Each ‘catch’ block needs to have own copy of clean-up.
- The solution is ‘finally’ block.
- A finally block is always executed where there is an exception or not

‘throw’ Keyword

- Used to explicitly throw an exception.
- Used for both checked or unchecked exceptions.
- Mainly used to throw custom exception

```
throw new IOException ("sorry device error);
```

‘throws’ Keyword

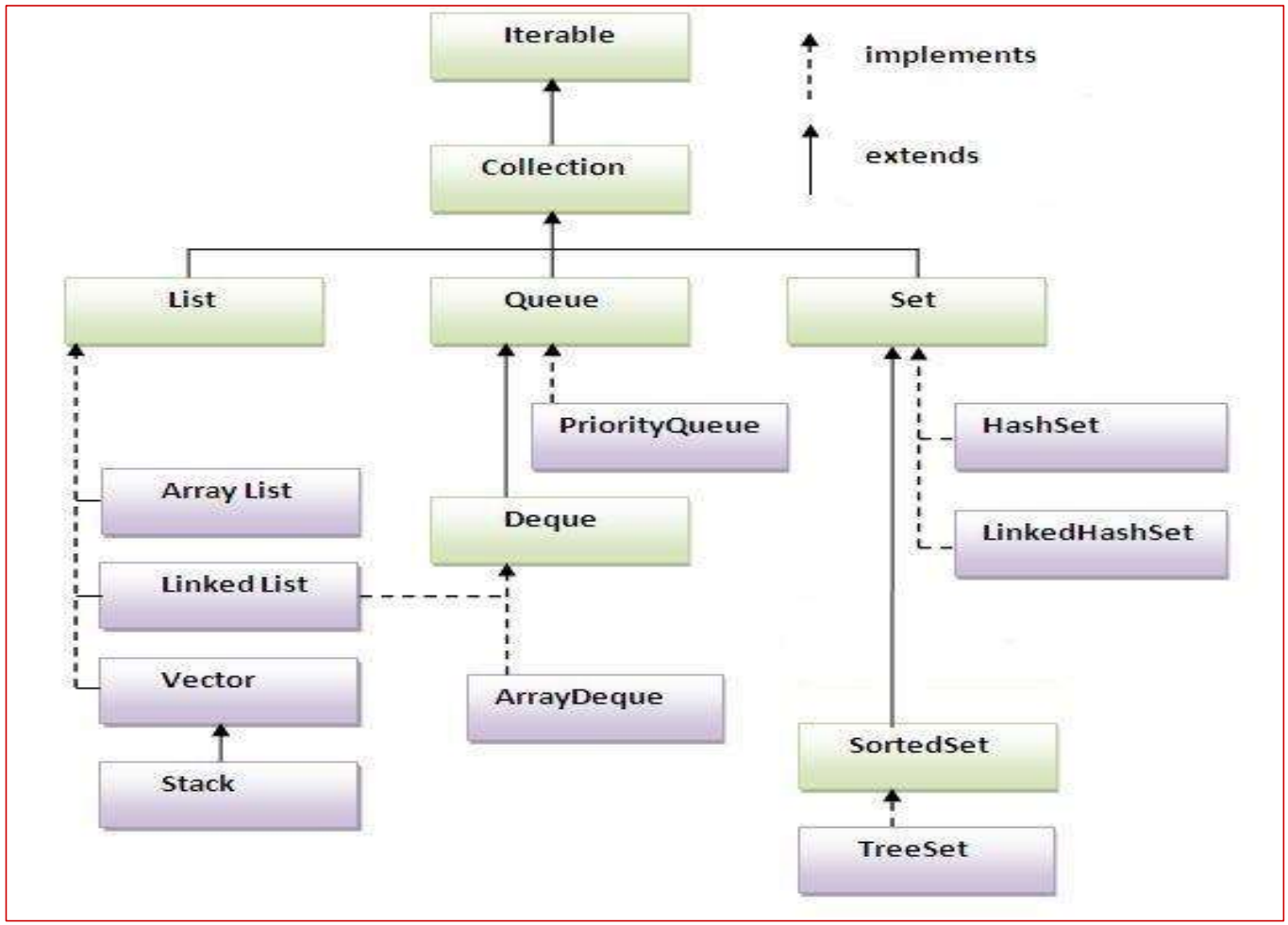
- Exception propagation – Exception is thrown from the top, drops down the call stack until it is caught.
- If a method that is implementing a particular business logic doesn’t want to handle the exception it throws it to the calling method.
- ‘***throws’*** is the keyword used to declare an exception.
- ***‘throws’*** is mandatory in case of checked exceptions otherwise compiler will give an error.
- In case of ‘**unchecked**’ exception, exceptions can be handled anywhere in the call stack.

Chapter 6

JAVA COLLECTION

Collections

- Why Collections?
  - Arrays are used to store a set of data, but their size is fixed.
  - insertion and deletion in the array is also costly in terms of performance.
  - In Actual development scenario, objects need to processed – added, deleted, sorted dynamically.
- Collections in java is a framework that provides an architecture to store and manipulate the group of objects.
- Operations such as searching, sorting, insertion, manipulation, deletion etc. can be performed by Java Collections


Iterable
implements
extends
Collection
List
Queue
Set
PriorityQueue
HashSet
Array List
Deque
LinkedHashSet
Linked List
Vector
ArrayDeque
SortedSet
Stack
TreeSet

Collection Interface

- Basic operations that are possible on any collection:
  - Add an object dynamically
  - Remove an object dynamically
  - List the objects by Iterating through the collection
  - Search specific object from collection
  - Retrieve an object from the collection
  - Sort the objects of the collection
- These common operations are defined in the form of interface.

List, List<E> interface

- Represents a collection of objects that can be accessed by an index.
- **Classes**: ArrayList, LinkedList, Vector.
- In List interface
  - Duplicates are allowed
  - Order is fixed
- **ArrayList** : Extends AbstractList class and implements List interface. Uses a dynamic array for storing the elements.
- **LinkedList**: Another Class that implements List interface, internally uses **doubly linked list** to store the elements.

Set, Set<E> interface

- Represents a collection that contains no duplicates.
- In Set interface order is not fixed.
- **Classes**: HashSet (unordered), LinkedHashSet (ordered), TreeSet(ascending order)
- **HashSet**: Extends Abstract Set class and implements Set interface
- Uses hashtable to store the elements.
- **TreeSet**: implements NavigableSet interface that extends the SortedSet interface. Maintains ascending order.

Map Interface

- Contains values based on the key i.e. key and value pair. (known as Entry)
- Map contains only unique elements.
- **Classes**: HashMap (unordered), LinkedHashMap (ordered), TreeMap (ascending order)

- **HashMap**: Extends AbstractMap class and implements Map interface
- Contains values based on the key.
- May have one null key and multiple null values.
- Maintains no order.

Iterator Interface

- Allows user to visit the elements in collection one by one
- Contains three methods:
  - Object next()
  - boolean hasNext()
  - void remove()


```
Iterator i = li.iterator();
while(i.hasNext())
{
        System.out.println(i.next());
}
```

Sorting of Collection Objects

- Sorting can be applied to:
  - String Objects
  - Wrapper class Objects
  - User-defined class Objects

- **Collections** class provides static methods for sorting the elements of collection.
- **public void sort (List list)** is the method used to sort List elements.
- If collection elements are of Set type, we can use TreeSet.

Comparable Interface

- Collections.sort() method provides sorting of List elements that are comparable.
- String class and Wrapper classes implement Comparable interface so sort () method can be directly used for these class objects.
- Comparable interface allows sorting of objects of user-defined class.
- Available in java.lang package.
- Provides single sorting sequence i.e. objects can be sorted based on only one data member.
- Provides **public int compareTo (Object obj)** method to compare objects of user-defined class.

Comparator Interface

- Used to order the objects of user-defined class.
- Available in java.util package.
- It provides multiple sorting sequence i.e. you can sort the elements based on any data member.
- Contains two methods to compare objects of user-defined class:
  - **public int compare(Object obj1,Object obj2):** compares the first object with second object.
  - **public void sort(List list,Comparator c):** is used to sort the elements of List by the given comparator.

Why Generics?

- All objects in non-generic collections are stored as Object type.
- Adding and retrieving values in such case requires essential type-cast.
- There is no guarantee of type safety with non-generic collections.
- Generics, forces the java programmer to store specific type of objects.

Advantages:

- Type-safety
- Type casting is not required
- Compile time checking

Generics Class

- A class that can refer to any type is known as generic class.
- **T** type parameter is used to create the generic class of specific type.

```
public class DemoGenerics <T> {
        T obj;
        public String getType(){
        return (obj.getClass().getName());
        }
public static void main(String[] args) {
        DemoGenerics<String> demo = new      DemoGenerics<>("Demo");
        System.out.println(demo.getType());  //returns     java.lang.String
}
```

Wildcards in Generics

- In generics programming , for types S and T, where S is a subtype of T, a generic type G<S> is not a valid subtype of G<T>
- Wildcards in Generic programming represents unknown type – represented by <?>
- Can be used in a variety of situations such as the type of a parameter, field, or local variable; sometimes as a return type
- Unbounded wildcards can be used when the code is using methods in the generic class that doesn't depend on the type parameter
- To implement generics with hierarchical class objects the remedy could be wildcards
- One can use upper bounded wildcards or lower bounded wildcards depending on the need

Chapter 7

FUNCTIONAL PROGRAMMING AND STREAMS

Lambda Expression

- Streams and Lambdas are two key factors in functional programming
- Functional programming helps in creating certain programs that are simpler, faster and have fewer bugs
- Such programs are easier to parallelize providing user the advantage of multi-core architecture to enhance performance
- With lambda Java introduced a new syntax and an operator -> lambda operator
- Lambda is essentially an anonymous method, where left hand side is parameter list and right hand side looks like method body [single statement or block of code]
- Lambda expressions allow us to create methods that can be treated as data

Functional Interface

- A functional interface is an interface that contains exactly one abstract method [may contain default/static methods] , also called as Single Abstract Method[SAM] interface
- Functional interface contains a single abstract methods, which defines the function of the interface
- A function is said to be a pure function if it –
  - Depends only on its parameters
  - Has no side effects
  - Do not maintain any state
- In java, methods that implement functional interfaces are pure functions– typically defined as lambdas

Functional Interface and lambdas

- Lambda expressions can be used anywhere functional interfaces are expected
- Lambda can be specified only in the context where a Target type is defined
- The java compiler can infer the types of lambda parameters and the type returned by the lambda from the context in which lambda is used
- This is determined by the lambda's **target-type –** i.e. type of functional interface type that the lambda implements

```
NumTest ispositive = (n) -> n >= 0 ;
```

Here compiler determines, the target type is interface NumTest. Lambda is assigned to a variable that can later be used to invoke abstract method of NumTest

Passing Lambda as argument

- Lambda can be used in any context that provides a target type
- One of the context is when lambda is passed as an argument to another method.
- This is most common use of lambda as it gives us a way to pass executable code as an argument to a method
- The methods that accept lambdas should have parameter type as functional interface. This type should be compatible with the lambda expression passed

```
map((x) -> x*2)
```

Here lambda is passed as a argument to map method. The parameter type of map method is IntUnaryOperator. The applyAsInt method of IntUnaryOperator accepts an int and return an int which is compatible with the lambda expression

Method Reference

- Method reference : provides a way to refer to a method without executing it.
- It is Java's way of sending a method reference inside another method
- The method that accepts another method as reference must have functional interface as parameter
- The lambda expression is written in place of actual method code. The lambda must be compatible with function interface
- To create a static method reference

```
Classname :: method_name
```

- To create an instance method reference

```
objRef::method_name
```

Imperative vs Declarative Programming

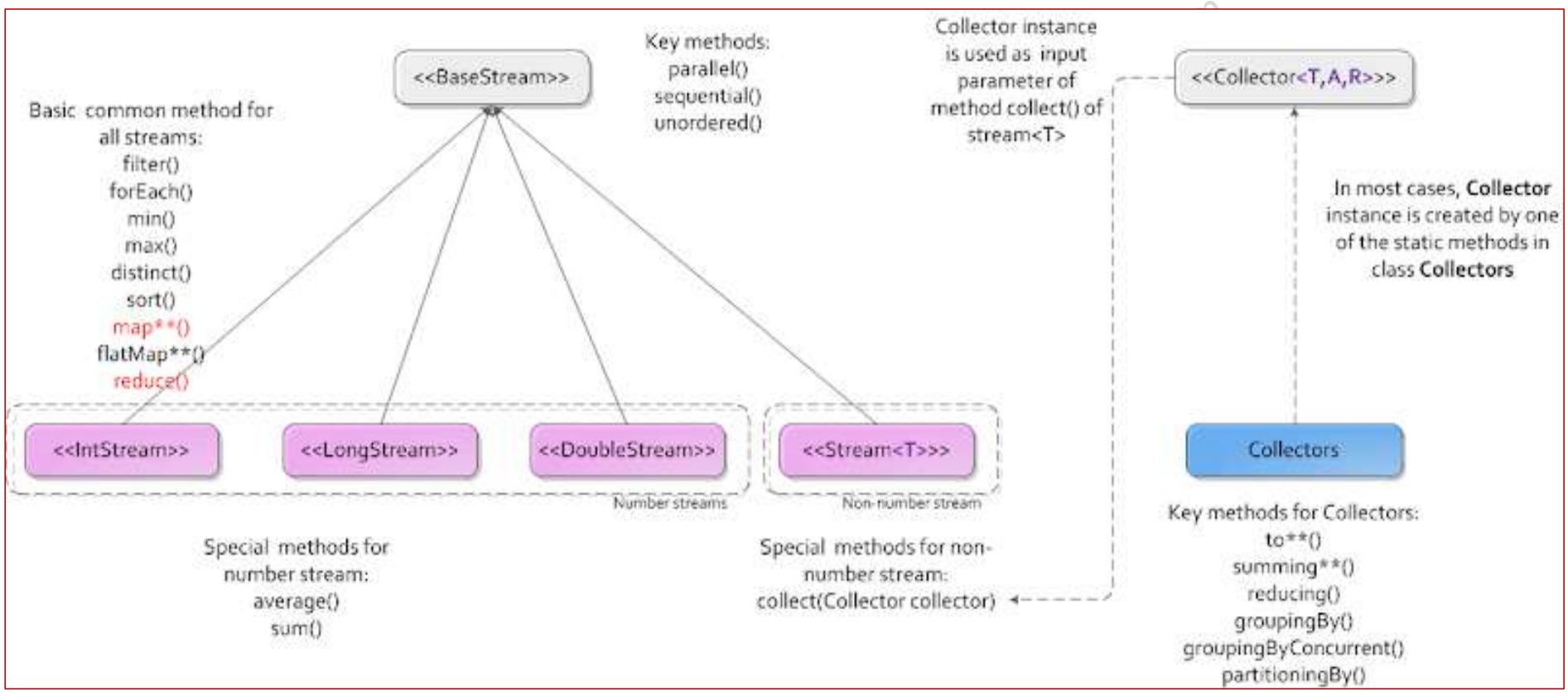
- Counter-controlled iteration : Operations where we typically specify ‘what’ do we want to accomplish and ‘how’ to accomplish the task using a loop (e.g. for)
- Task is performed in each iteration of the loop – external iteration
- Such loops are error prone leading to errors related to initialization , increment and bounds
- Imperative Programming : where we specify what do we want to achieve followed by how the task can be achieved
- Declarative Programming : We specify ‘what’ we want the task to accomplish (rather than how)
- Internal iteration – Iteration over the elements without specifying ‘how’ to iterate or declaring any mutable variable

Streams

- Introduced in Java 8, in java.util.stream package
- Stream : sequence of elements on which we perform a specific task
- A stream operates on data source such as an array or collection
- Stream classes support declarative programming – specify what to do rather than how to do it
- Stream pipelines - Chained method calls on streams create a stream pipeline
- Stream pipelines start with a method call that creates a stream [e.g. range()]
- We can perform many intermediate operations on the stream before a terminal operation is performed
- Terminal operation terminates the stream and produces final result

Stream API

- The stream API defines several stream interfaces, packaged in java.util.streams
- BaseStream interface is at the Base of all streams and defines basic functionalities for all streams
- Several types of streams are derived from the base stream like – most generic is Stream
- Stream is a stream of reference types [objects]
- Stream API also provides streams to operate on primitive data – IntStream, LongStrem and DoubleStream
- Collections are used in java as data structures to store objects, streams on the other hand are used to operate on the data stored in the collections or arrays


<<BaseStream>>
Key methods:
parallel()
sequential()
unordered()
Basic common method for all streams:
filter()
forEach()
min()
max()
distinct()
sort()
map**()
flatMap**()
reduce()
Collector instance is used as input parameter of method collect() of stream<T>
<<Collector<T,A,R>>>
In most cases, Collector instance is created by one of the static methods in class Collectors
<<IntStream>>
<<LongStream>>
<<DoubleStream>>
<<Stream<T>>>
Number streams
Non-number stream
Collectors
Special methods for number stream:
average()
sum()
Special methods for non-number stream:
collect(Collector collector)
Key methods for Collectors:
to**()
summing**()
reducing()
groupingBy()
groupingByConcurrent()
partitioningBy()

Obtaining a Stream

- Stream can be obtained for a collection as well as for an array
- Two new methods are added Collection interface which can obtain a stream from the collection [JDK 8]

  collection_obj.stream()

- This obtains a sequential stream. A parallel stream can be obtained using parallelStream method

  collection_obj.parallelStream()

- Streams can also be obtained for arrays using static method stream()

  Arrays.stream(array_obj)

Collecting

- It is common to obtain a stream from collection, but it is equally desirable to obtain a collection from the stream
- This can be viewed as the terminal operation
- To perform this stream provides collect() method

  stream.collect(Collector<?super T,A,R > collectFunc)

- The Collector interface provides several method to perform the collection function
- Collectors [implements Collector] class methods toList() and toSet() are most commonly used

Performing Stream Operations

- Intermediate operations use lazy evaluation
- Lazy evaluation : each intermediate operation results into a new stream object but does not preform any operation on it immediately until terminal operation is performed to produce a result
- Terminal operations use eager evaluation
- Eager evaluation : Terminal operations perform the requested operation when they are called
- This enhances the performance to a great extend
- Also some intermediate operations are stateless while other are stateful
- In stateless operation each element is process independently of other, in stateful operation such processing might depend on other element values

Typical Operations

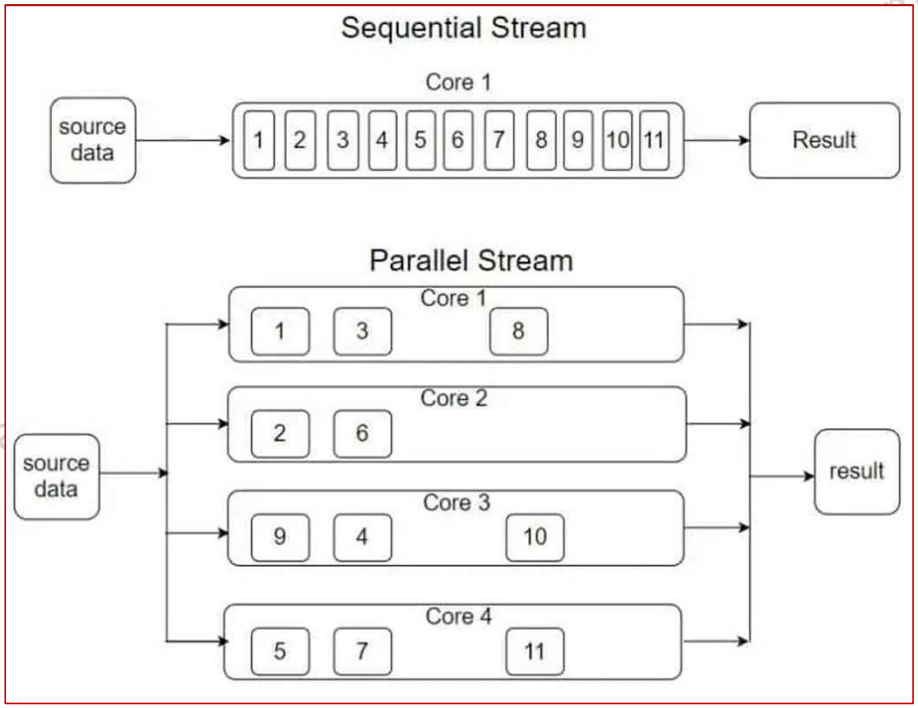
Intermidiate Operations	Terminal Operations
Filter	forEach
Map	reduce
Distinct	count
limit	min
sorted	max
	collect

Functional Interfaces

- java.util.functions contain several functional interfaces. Few of the basic ones are explained below:

Interface	Description
Binary Operator<T>	Represents an operation upon two operands of the same type, producing a result of the same type as the operands. E.g. the lambda passed to reduce method implements this interface
Consumer<T>	Represents an operation that accepts a single input argument and returns void. E.g. the lambda passed to forEach method implements this interface
Function <T, R>	Represents a function that accepts one argument and produces a result. E.g. the lambda passed to map method implements this interface
Predicate <T>	Represents a one parameter method that returns boolean value.E.g. the lambda passed to filter method implements this interface
Supplier<T>	Represents a supplier of results - a one parameter method that returns a result. Often used to create a collection object in which a stream operation's result are passed

Parallel Stream



Chapter 8

JAVA DATETIME

Introduction

- Working with date and time is one of the challenges in programming language
- Java DateTime API helps to deal effectively with time zones, daylight saving time, and different written date formats
- The old Java Date API was not thread-safe, It depended on java.text classes to format the date
- The Objects of new DateTime API classes are immutable [hence thread safe]
- Computers count time from an instant called the Unix epoch [January 1, 1970, at 00:00:00 UTC]
- An 'Instant' is a point on the timeline
- Duration is difference between two instants

LocalDate Class

- Gives date and/or time of a day without any time zone information

  LocalDate today = LocalDate.now()

  LocalDate adate = LocalDate.of(2023, Month.AUGUST, 10)

- The difference between two dates is Period
- Using plus/minus methods we can add/subtract period
- plusDays(), plusWeeks(), plusMonths(), plusYears() methods can be used to add specific value to the date

  LocalDate nextweek = today.plusWeeks(1)

Method	Description
now(), of()	Construct a LocalDate from the current time/ from given year,month and day
plusDays(),plusWeeks(), plusMonths(), plusYears()	Adds a number of days, weeks, months or years to this LocalDate
minusDays(),minusWeeks() ,minusMonths(), minusYears()	Subtracts a number of days, weeks, months or years from this LocalDate
plus(), minus()	Adds or substrats a Duration of Period
getDayOfMonth()	Gets the day-of-month field
getDayOfWeek()	Gets the day-of-week field, which is an enum DayOfWeek.
getDayOfYear()	Gets the day-of-year field.
getMonth()	Gets the month-of-year field using the Month enum.
getYear()	Gets the year field.
unitl()	Gets the period, or the number of the given Chronounits, between the two dates
isBefore, isAfter	Compares the LocalDate with another

LocalTime Class

- A LocalTime represents a time of the day
- A time Instance can be created using methods now or of

```
LocalTime tcurrent = LocalTime.now();
LocalTime later = LocalTime.of(22, 00);
```

- Interval can be given using Duration

```
LocalTime plusone = wakeuptime.plus(Duration.ofHours(1))
```

- getHour, getMinute, getSecond can be used to retrieve time values

Method	Description
now(), of()	Construct a LocalTime from the current time/ from given hours, minutes and seconds
plusHours(),plusMinutes(), plusSeconds(), plusNanos()	Adds a number of hours, minutes, seconds or nanoseconds to this LocalTime
minusHours(),minusMinutes(), minusSeconds(), minusNanos()	Subtracts a number of hours, minutes, seconds or nanoseconds to this LocalTime
plus(), minus()	Adds or substrats a Duration of Period
getHour()	Gets the Hour
getMinute()	Gets the Minutes
getSeonds()	Gets the Seconds
getNano()	Gets the Nano Seconds
isBefore, isAfter	Compares the Local Time with another
toEpochSecond	Given a Local Date and Zone Offset, yields the number of seconds from the epoch to the specified point in time

Zoned Time

- Zone in which a date occurs is an important aspect of ensuring your code is correct
- IANA – Internet Assigned Number Authority keeps database of all known time zones around the world
- Each time zone has an ID, which can be found out by ZoneId.getAvailableIds()
- Using a static method ZoneId.of("Europe/Berlin") one can get a ZoneId object, which can be used to convert a LocalDate to ZonedDateTime

```
ZonedDateTime now = ZonedDateTime.of(LocalDate.now(),
                                     LocalTime.now(), ZoneId.of("Europe/Berlin"));
```

Method	Description
now(), of()	Construct a LocalDate from the current time/ from given year,month and day
plusDays(),plusWeeks(), plusMonths(), plusYears()	Adds a number of days, weeks, months or years to this LocalDate
minusDays(),minusWeeks(), minusMonths(), minusYears()	Subtracts a number of days, weeks, months or years to this LocalDate
plus(), minus()	Adds or substrats a Duration of Period
getDayOfMonth()	Gets the day-of-month field
getDayOfWeek()	Gets the day-of-week field, which is an enum DayOfWeek.
getDayOfYear()	Gets the day-of-year field.
getMonth()	Gets the month-of-year field using the Month enum.
getYear()	Gets the year field.
unitl()	Gets the period, or the number of the given Chronounits, between the two dates
isBefore, isAfter	Compares the LocalDate with another

Formatting and Parsing

- The DateTimeFormatter class provides three kinds of formatters to print a date/time value
  - Predefined standard formatters
  - Local-specific formatters
  - Formatters with custom patterns
- The format() method can be used to give standard formatter
- The standard formatters are mostly intended for machine readable timestamps, For human readable dates locale specific formatters are used
- Custom patterns can be given using a user defined pattern with the help of ofPattern() method

Symbol	Meaning	Example	Symbol	Meaning	Example
G	Era Designation	AD	S	Millisecond	968
Y	Year	2022	E	Day in week	Monday
M	Month In A Year	March; Mar; 03	D	Day in year	180
d	Day in a Month	5	w	Week in year	25
h	hour in am/pm (1-12)	10	W	Week in month	3
H	hour in day (0-23)	March; Mar; 03	a	Am/Pm marker	PM
m	Minute in hour	40	k	Hour in day (1-24)	20
s	Second in minute	50	K	Hour in am/pm (0-11)	2

Chapter 9

MULTITHREADING

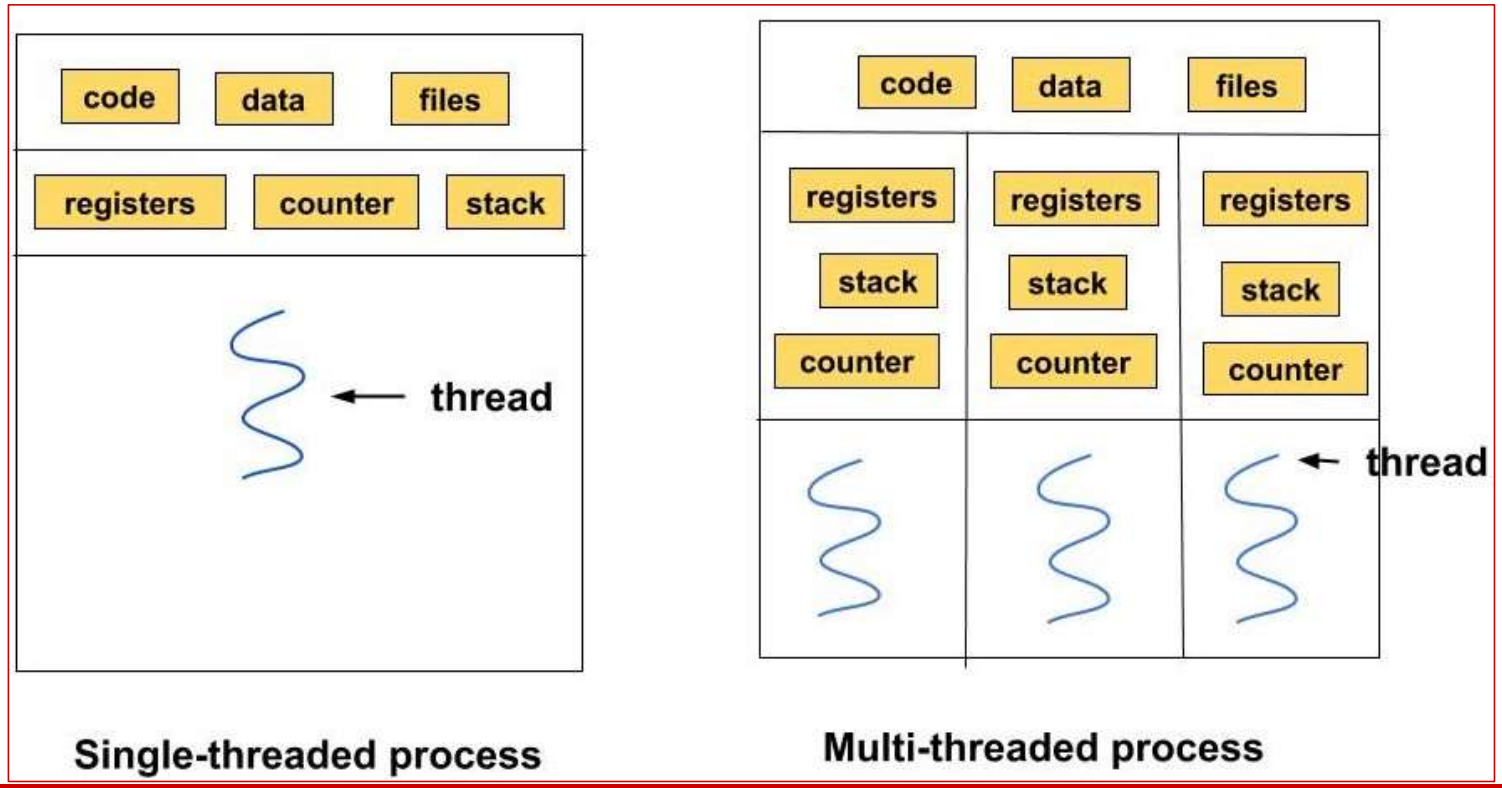
Multithreading

- In built support for Multithreading programming
- Multithreaded program - two or more parts of the program can run concurrently, each part is called a thread
- Enables programmers to write efficient programs that can make maximum use of processing power
- Most important benefit is minimum CPU idle time
- In single threaded environment program has to wait for each task to get processed (e.g reading from a file), most of the time program is idle waiting for input
- In multi threaded environment idle time is minimized because another thread can run when one is waiting

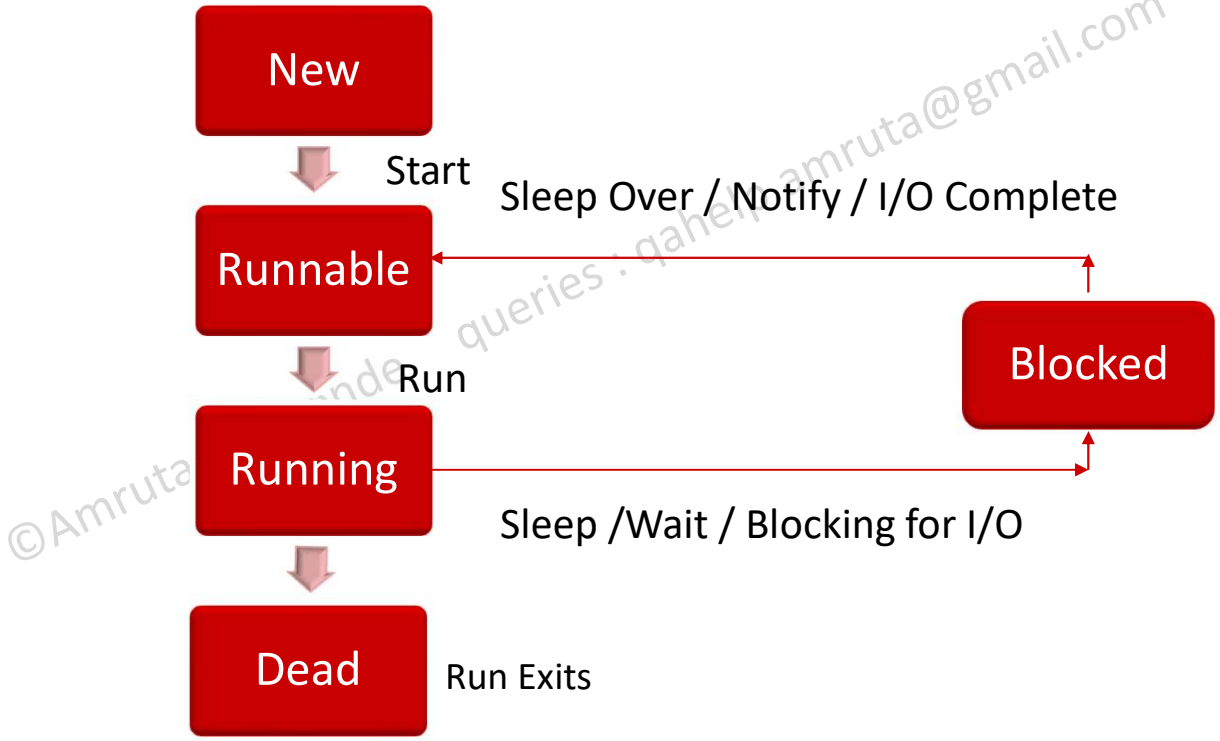
Multi processing vs Multi Threading

Multi Processing	Multi Threading
Alllows multiple programs to run concurrently	Multiple parts of a program can be run concurrently
Each process requires its own separate address space	Threads share same address space
Context switching from one process to another is costly	Context switching from one thread to another is not costly
Inter-process communication is expensive	Inter thread communication is inexpensive

Single threaded vs Multithreaded



Java Thread Model



CPU Scheduling

- CPU Scheduler : Whenever the CPU becomes idle the operating system must select one of the processes in the ready queue to be executed. The selection process is carried out by the Short-Term scheduler (CPU scheduler)
- Scheduler selects the process from the processes that are ready to execute and allocates CPU to it
- Dispatcher : Module that gives control of the CPU to the process selected by the CPU scheduler
- CPU scheduling can take place when:
  - Process switches from running state to waiting state (e.g for I/O complete)
  - Process switches from running to ready (e.g. on interrupt)
  - Process switches from waiting to ready (on I/O complete)
  - Process terminates

Non -preemptive and preemptive scheduling

- When the process is in waiting state (#1) or it has terminated (#4) , the CPU must be given to another process, there is no choice
- When scheduling takes place under only situation 1 & 4, it is called as non- preemptive (cooperative) scheduling
- In such scheduling scheme the CPU is assigned to another process only when the current process has either terminated or blocked [waiting]
- In other two situation the process was in running state when the CPU can be given to another ready process – preemptive scheduling
- In preemptive scheduling a lower priority thread that does not yield the processor can be preempted by higher priority thread

Thread Methods

sleep	Causes currently executing thread to sleep for specified number of milliseconds
yeild	Causes currently executing thread to pause and allow other thread to execute
join	Waits for a thread to die
Wait (From Object)	Allows a thread to wait inside a synchronized method
Notify (From Object)	Wakes up a single thread that is waiting
notifyAll (From Object)	Wakes up all threads those are waiting

Synchronization

- Synchronization in computer program means making one instruction run before another
- When Threads share a common resource, they need to make sure that resources will be used by one thread at a time
- Synchronization is implemented using concept of Monitor – An object used as a mutually exclusive lock
- Only one thread can own a monitor at a time, when one thread enters the monitor all other threads are suspended (said to be waiting for the monitor)
- Synchronized method – When one thread is inside the synchronized method, all other threads trying to call it on the same instance have to wait for the first one to return
- Another way is to put the calls to any method inside a synchronized block, and pass the object reference of the object being synchronized

Producer Consumer Problem

- Multithreaded programs have a division of labor between threads, some threads produce while others consume
- For instance, Event-driven programming

  An “event” is something that happens that requires the program to respond. Whenever event occurs, a producer thread creates an event object and adds it to the event buffer. Concurrently, consumer threads take events out of the buffer and process them
- Here when an item is added/removed from the buffer the buffer is in inconsistent state
- Threads must have exclusive access to the buffer.
- If a consumer thread arrives while the buffer is empty, it blocks until a producer adds a new item.

Reader Writer Problem

- Pertains to any situation where a data structure, database, or file system is read and modified by concurrent threads
- When data is being written/modified it is essential to bar other threads from reading the data storage
- Readers and writers should execute different code before entering the critical section
- Synchronization constrains would be:
  - Any number of readers can be in the critical section simultaneously
  - Writers must have exclusive access to the critical section

Interthread Communication

- Implicit monitors are powerful, but for better control inter-process communication is useful
- To avoid polling, Java implements inter process communication using wait, notify and notifyAll methods
- These methods can be called only from within a synchronized context
- **wait** – Tells the calling thread to give up monitor and go to sleep until another thread enters the same monitor and calls notify or notifyAll method
- **notify** – wakes up the thread that called wait on the same object
- **notifyAll** - wakes up all the threads that called wait on the same object. One of the threads will be granted access

Java Concurrency Package

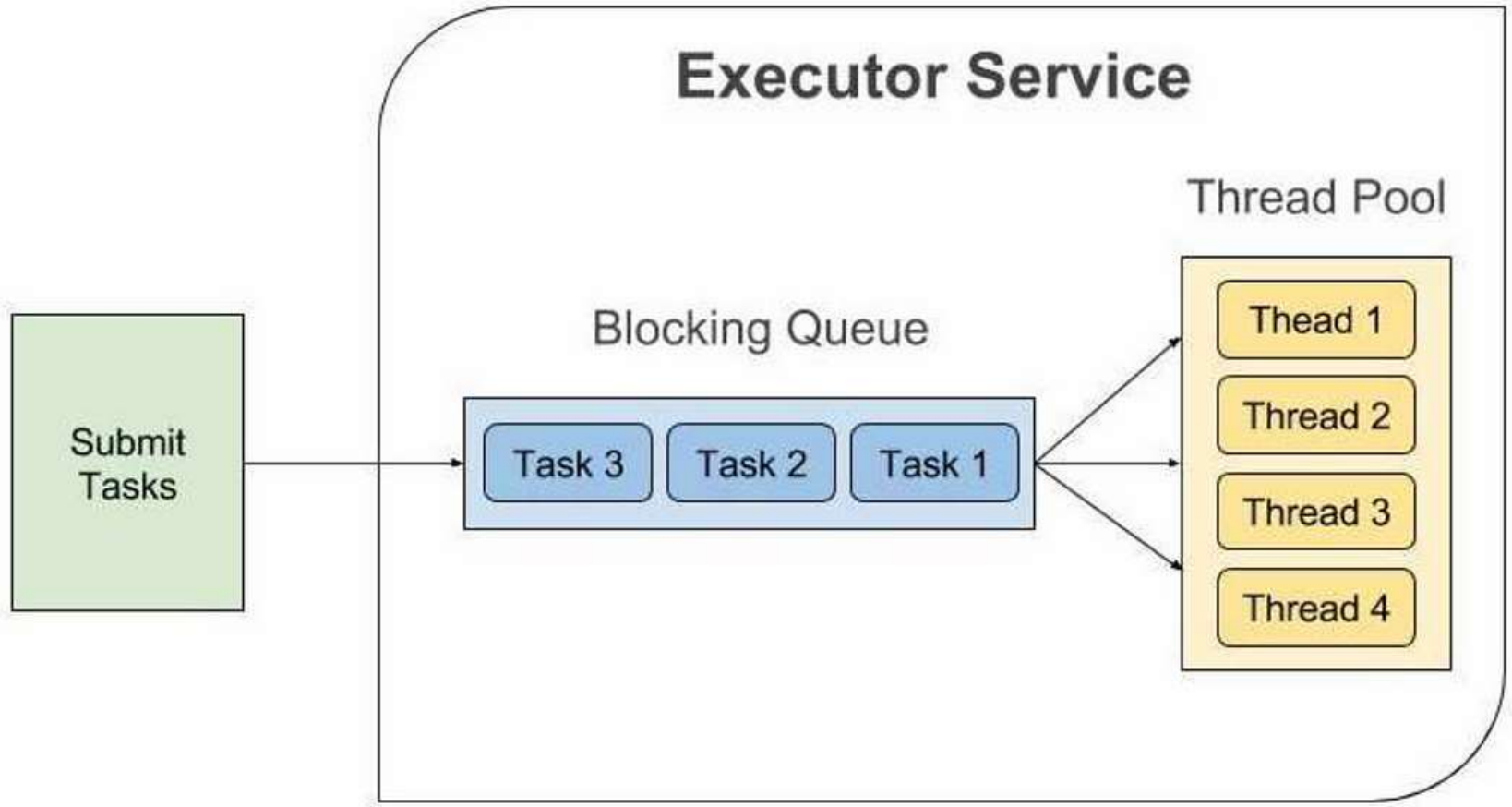
- Although java supports concurrent application development in a thread safe way using synchronization keyword and interthread communication, it is not ideal for application that extensively use multiple threads
- In JDK 5, java introduced util.concurrent package, commonly referred as concurrent API
- This supplies many features to support development of concurrent applications
- The key features include :
  - Synchronizers
  - Executors
  - Concurrent Collections
  - The fork/join framework

Concurrent Collections

- Concurrent collection represent set of classes that allow multiple threads to access and modify a collection concurrently
- For concurrent modifications no explicit synchronization is needed
- These collections provide thread-safe implementations of the traditional collection interfaces like List, Set, and Map
- Include following interface and classes:
  - Blocking queue implemented by ArrayBlockingQueue, LinkedBlockingQueue
  - ConcurrentMap implemented by ConcurrentHashMap
  - Classes like CopyOnWriteArrayList, CopyOnWriteArraySet

Executors

- The executors manage thread execution
- ExecutorService extends Executor, and it provides methods that manage the execution of threads
- Creating a java thread is much like creating a OS thread, which is an expensive operations
- When we need to run multiple tasks concurrently, an alternative is to use an Executor service
- The executor framework creates a thread pool with fixed no of thread which will execute the tasks concurrently
- So instead of creating a new thread every time, we submit multiple tasks which will be executed by the threads in the fixed thread pool
- The thread pool uses a thread safe data structure - blocked queue to store the tasks,


Executor Service
Thread Pool
Blocking Queue
Submit Tasks
Task 3
Task 2
Task 1
Thead 1
Thread 2
Thread 3
Thread 4

Executor Framework

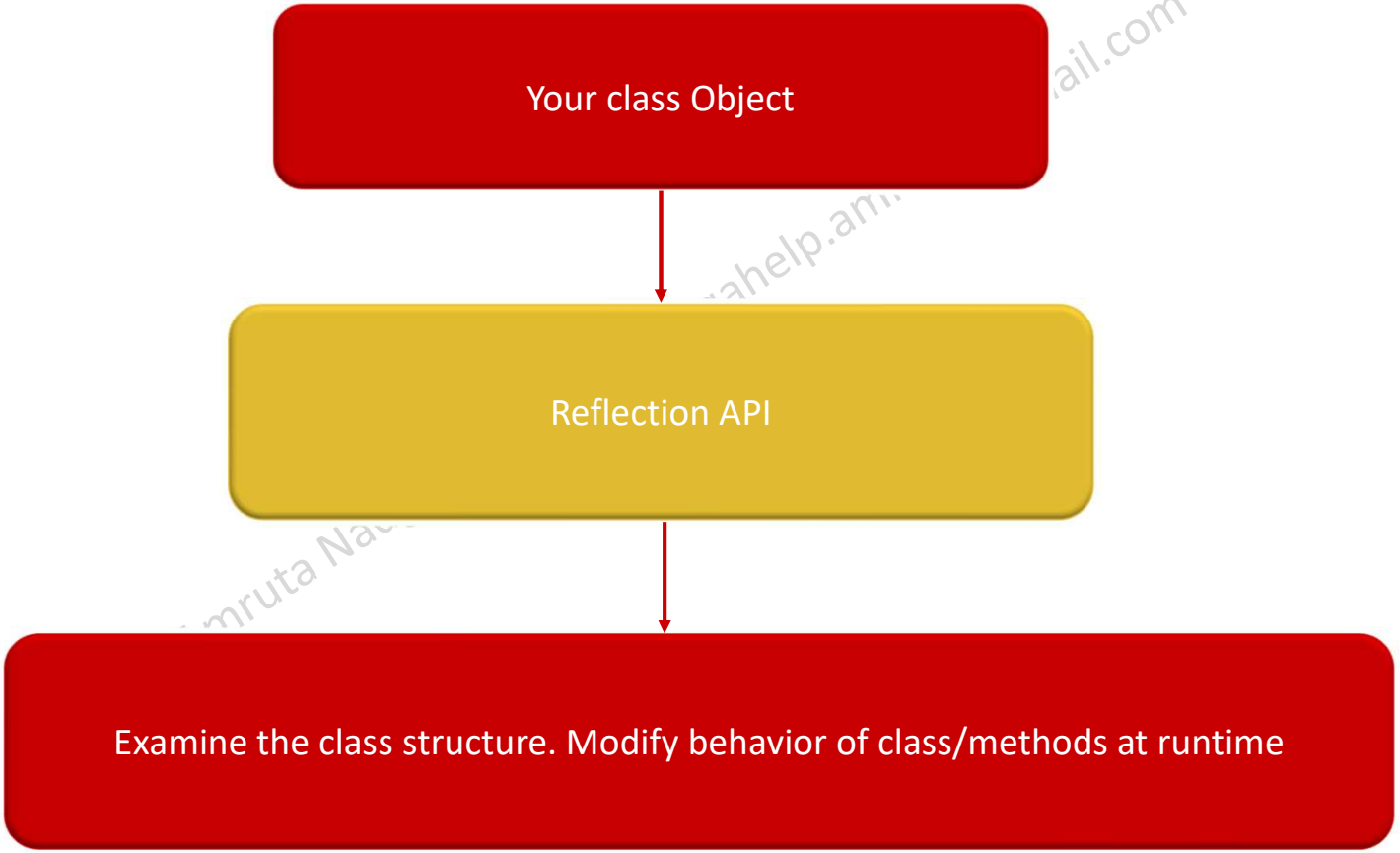
- The executor framework support different types of thread pools
  - Fixed Thread Pool
  - Cached Thread Pool
  - Scheduled Thread Pool
  - Single Thread Executor
- A cached thread pool uses a synchronous queue to hold only 1 task. It searches for a thread that can execute the task, and if all threads are busy it creates a new thread for execution
- Since a new thread is created for each task , if all threads are busy, that may lead to too many threads
- To avoid this cached thread pool kills the threads after 60 seconds of idle time

Chapter 10

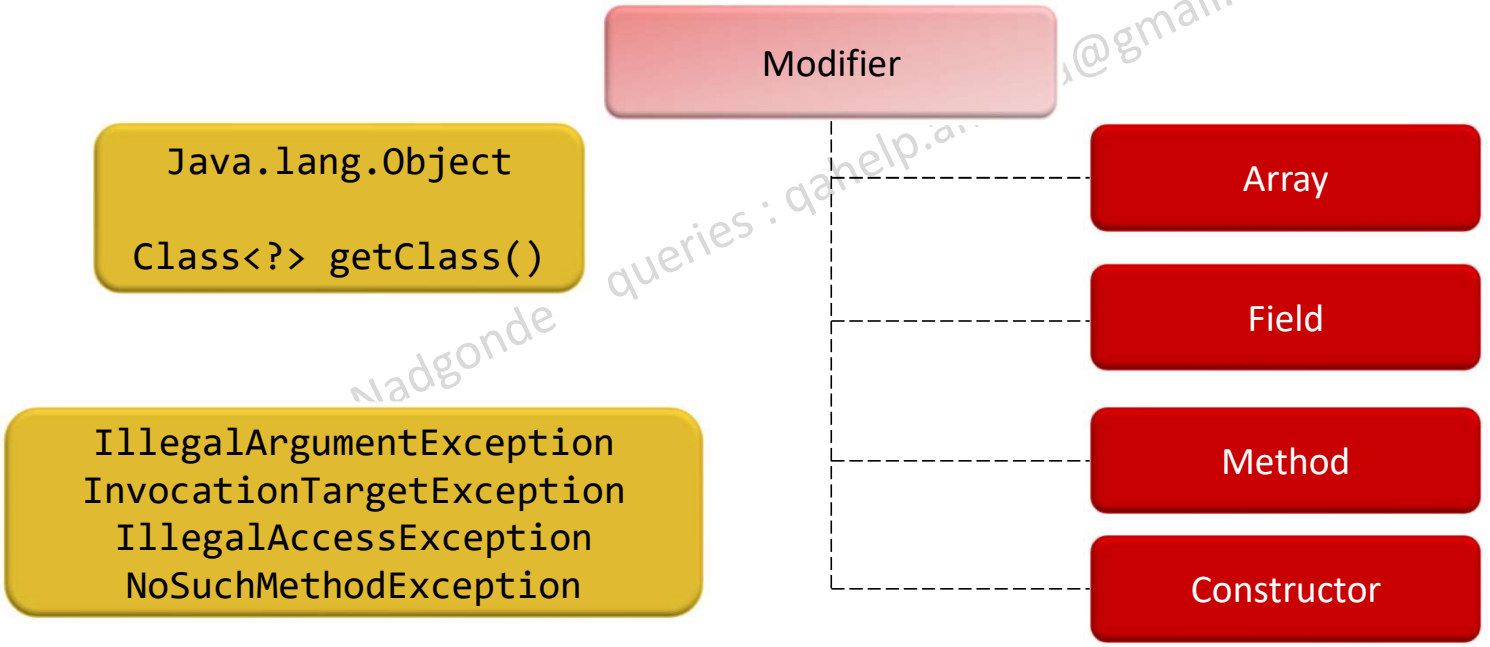
REFLECTION AND ANNOTATIONS

Java Reflection

- Reflection is a java API that allows us examine and /or modify behavior of methods, classes and interfaces at run time
- The classes that support reflection are available in java.lang.reflect package
- Using reflection, we can get more information about fields, constructors, methods as well as annotations used in a class
- Reflection gives us information about the class to which an object belongs and also the methods of that class that can be executed by using the object.
- Through reflection, we can invoke methods at runtime irrespective of the access specifier used with them.
- The getClass() method of Object class is used to return the runtime class of any Object


Your class Object
Reflection API
Examine the class structure. Modify behavior of class/methods at runtime

The Reflection API



Runtime Class of an object

- The getClass() method of Object class returns the runtime Class of given object

```
Employee emp = new Employee();

Class c = emp.getClass()
```

- The static method of 'Class' can also be used to get the runtime class of the given object

```
Class c = Class.forName("Employee");
```

- A new instance of a class can be created using newInstance() method

```
Class c = Class.forName("Employee");
Employee e = (Employee)c.newInstance();
```

Class class

- Class objects are constructed automatically by the Java Virtual Machine as classes are loaded

Annotation[] getDeclaredAnnotations()	Returns annotations that are directly present on this element.
Constructor<?>[] getDeclaredConstructors	Returns an array of Constructor objects reflecting all the constructors declared by the class represented by this Class object.
Field[] getDeclaredFields()	Returns an array of Field objects reflecting all the fields declared by the class or interface represented by this Class object.
Method [] getDeclaredMethods()	Returns an array containing Method objects reflecting all the declared methods of the class or interface represented by this Class object
Type getGenericSuperclass()	Returns the Type representing the direct superclass of the entity represented by this Class
Class <?> [] getInterfaces()	Determines the interfaces implemented by the class or interface represented by this object.
String getName()	Returns the name of the entity epresented by this Class object
boolean isAnnotationPresent()	Returns true if an annotation for the specified type is present on this element, else false.

Runtime Method invocation

- To invoke a method through reflection we should know its name and parameter types

```
Method method = emp.getClass().getDeclaredMethod("setEmpid",
      int.class);
 method.invoke(emp, 112);
```

```
Method methods [] = emp.getClass().getDeclaredMethods();
 if(method.getName().equals("simplePrivateMethod"))
  {
      method.setAccessible(true);
      method.invoke(emp);
  }
```

Annotations

- Introduced in JDK 1.5, a feature that allows to add supplemental information into a source file
- This information is used by various tools during development and deployment [e.g. by a source code generator]
- Annotations are created using a @ that precedes a keyword interface
- Annotations consist only method declaration, no body
- The method implementation is provided by Java
- Marker annotations are annotations without any methods

Annotations at Runtime

- The annotations which are retained till runtime [`@Retention(RetentionPolicy.RUNTIME)]`
- The annotations can be used to modify behaviour of the code at runtime
- The runtime annotations can be obtained by

```
method.isAnnotationPresent(Test.class)
```

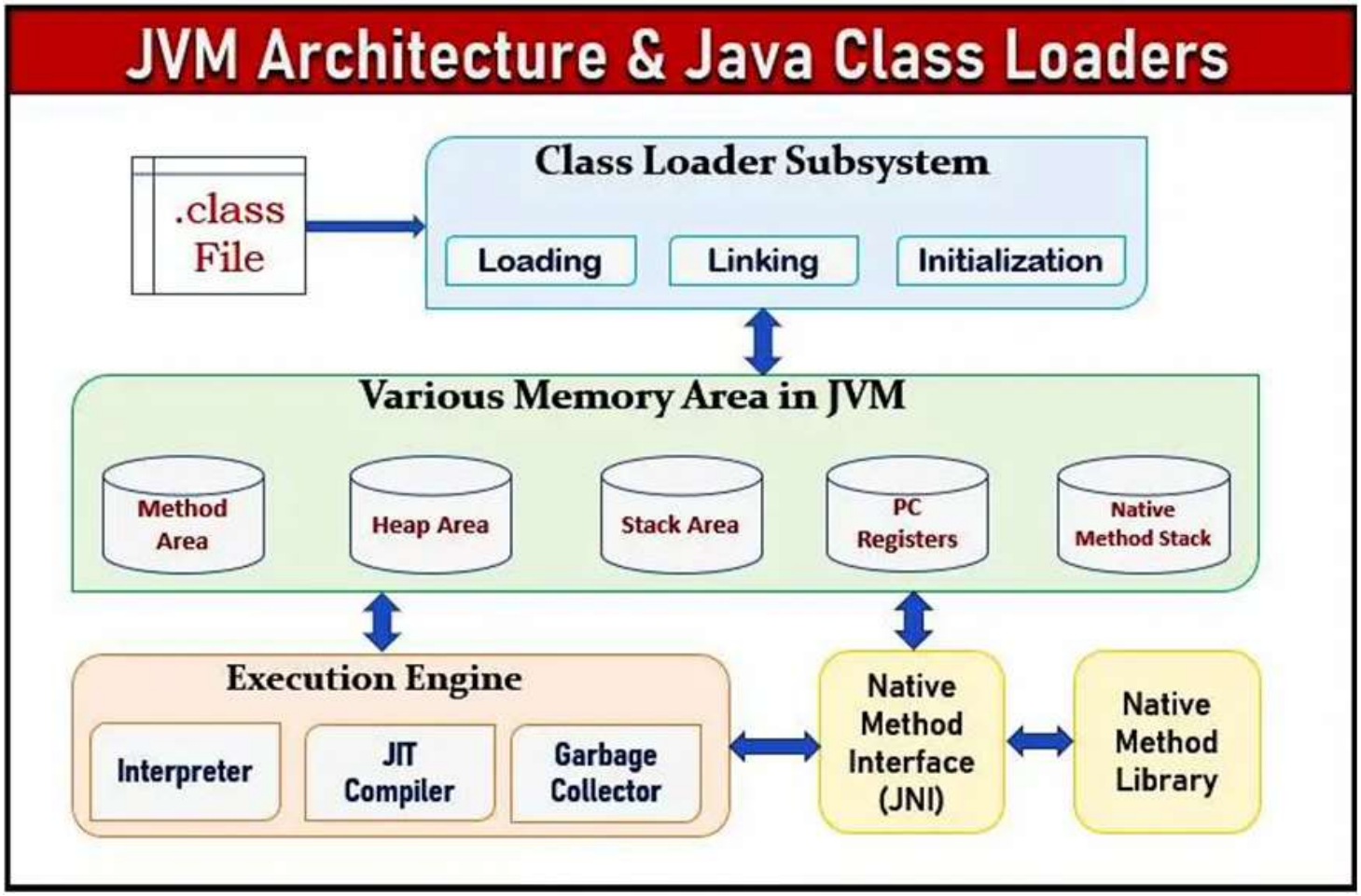
- The Annotation object is returned by using getAnnotation() method

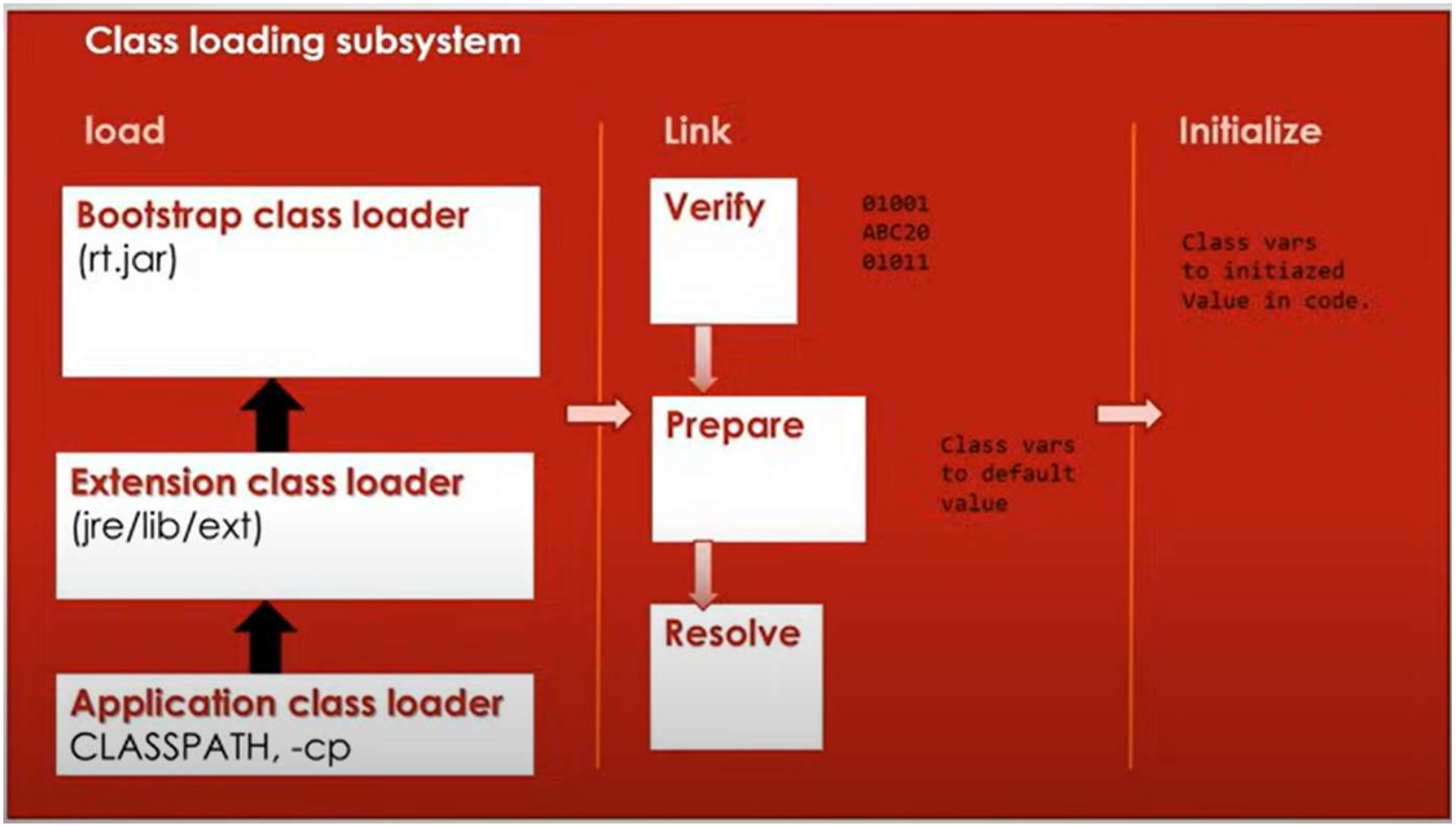
```
Test anno = method.getAnnotation(Test.class);
```

- Once the annotation object is obtained , it can be used to check annotation information and invoke methods / set field values etc

JVM

- JVM is responsible for loading and executing class file
- JVM instance is created when run command is given
- JVM contains class loader sub system and execution engine
- Class loader is responsible for loading the application .class file and in-built java classes
- The loaded classes [bytecode instructions] are fed to execution engine


JVM Architecture & Java Class Loaders
.class File
Class Loader Subsystem
Loading
Linking
Initialization
Various Memory Area in JVM
Method Area
Heap Area
Stack Area
PC Registers
Native Method Stack
Execution Engine
Interpreter
JIT Compiler
Garbage Collector
Native Method Interface (JNI)
Native Method Library


Class loading subsystem
load
Bootstrap class loader
(rt.jar)
Extension class loader
(jre/lib/ext)
Application class loader
CLASSPATH, -cp
Link
Verify
01001
ABC20
01011
Prepare
Class vars
to default
value
Resolve
Initialize
Class vars
to initiazed
Value in code.

Class Loader

- Consists of three phases : Load, Link, Initialize
- Load phase is responsible for loading the class file from physical memory / network
- Class loader types :
  - Bootstrap class loader : Loads java's internal classes residing in rt.jar which is distributed with JVM
  - Extension class loader : Loads class files for additional jars present in jre/lib/ext folder
  - Application class loader : Loads application class file from the value specified in the classpath variable

Class Loader – Link/Initialize

- The Link phase of class loader is further divided as verify, prepare and resolve
- The verify phase checks if the given byte code is as per the JVM specification [valid byte code]
- In the prepare phase static variables are assigned the memory [with default value]
- In the resolve phase all symbolic references in the class are resolved
- References to other classes, constant pool are changed from symbolic names to actual reference
- In the initialize phase the static initialization blocks get executed
- The static variables which were assigned with default values are now assigned with their actual values

Runtime Data [Memory] Area

- Method area : Method data corresponding to a class is stored here
- Many API in Java reflection require the data in method area. Static variables, class level constant pool etc. are saved here
- Heap: Heap is used to store the Java objects
- PC register : Contains the program counter for the next instruction per thread
- Java Stack : Contains method stack for the currently running threads. Each thread maintains its own method stack
- Native Method Stack : When native methods are invoked from one of the methods , native method stack is created

Execution Engine

- Interpreter is responsible for converting the byte code instruction into the native machine language
- Interpreter finds which native operation needs to be performed and executes that using native method interface (JNI)
- The JNI is a set of interfaces that permit a native method to interoperate with the Java virtual machine
- JIT Compiler : Responsible for compiling the repeated instructions on the fly and keep them ready as target code. Such code is executed directly without any repeated interpretation
- Hotspot Profiler : Helps the JIT to locate the repeated code (Hotspots)
- Garbage Collector : Responsible for memory management of dead objects