NARSIMHA REDDY ENGINEERING COLLEGE

UGC AUTONOMOUS INSTITUTION

Maisammaguda (V), Kompally - 500100, Secunderabad, Telangana State, India

UGC - Autonomous Institute
Accredited by NBA & NAAC with 'A' Grade
Approved by AICTE
Permanently affiliated to JNTUH

Object Oriented Programming Through JAVA (23CS305)

II B. Tech I Semester (NR23)

Prepared by

K.Anusha/Dr.N.Kavitha/P.Revathy/Gopal

UNIT-1
OBJECT-ORIENTED THINKING,JAVA BASICS

History of Java

Java team members (also known as Green Team), initiated a revolutionary task to develop a language for digital devices such as set-top boxes, televisions etc. It was best suited for internet programming. Later, Java technology as incorporated by Netscape.James Gosling, Mike Sheridan, and Patrick Naughton initiated the Java language project in June 1991. The small team of sun engineers called Green Team.

Firstly, it was called "Greentalk" by James Gosling and file extension as .gt.After
that, it was called OAK and was developed as a part of the Green project.

OAK is a symbol of strength and chosen as a national tree of many countries like U.S.A., France, Germany, Romania etc. In 1995, Oak was renamed as "Java" because it was already a trademark by Oak Technologies. Java is an island of Indonesia where first coffee was produced (called java coffee).

A Way of Viewing World

A way of viewing the world is an idea to illustrate the object-oriented programming concept with an example of a real-world situation.

Let us consider a situation, I am at my office and I wish to get food to my family members who are at my home from a hotel. Because of the distance from my office to home, there is no possibility of getting food from a hotel myself. So, how do we solve the issue?

To solve the problem, let me call zomato (an **agent** in food delevery community), tell them the variety and quantity of food and the hotel name from which I wish to delever the food to my family members. Look at the following image.

JAVA BUZZWORDS OR FEATURES OF JAVA

- Simple
- Secured
- Platform independent
- Robust
- Portable
- Architecture neutral
- Dynamic
- Interpreted
- High Performance
- Multithreaded
- Distributed

APPLICATIONS OF OBJECT ORIENTED PROGRAMMING

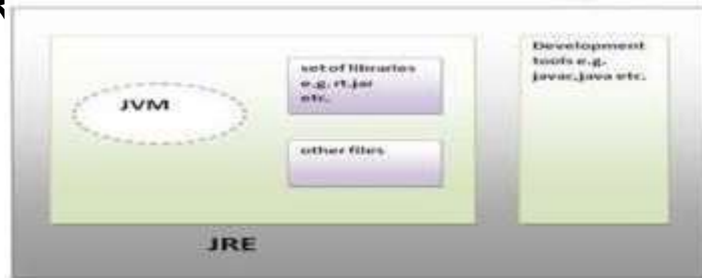
Main application areas of OOP are:

- User interface design such as windows, menu.
- Real Time Systems
- Simulation and Modeling
- Object oriented databases
- AI and Expert System
- Neural Networks and parallel programming
- Decision support and office automation systems etc.

JDK, JRE and JVM

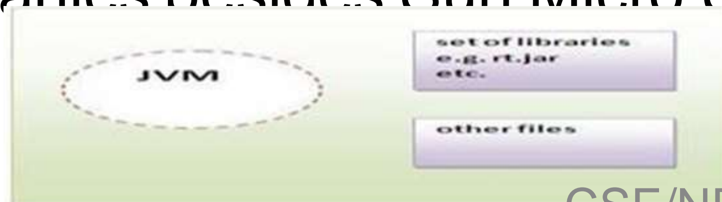
Java Development Kit (JDK):

JDK is an acronym for Java Development Kit. It physically exists. It contains JRE + development tools.



Java Runtime Environment (JRE):

JRE is an acronym for Java Runtime Environment. It is used to provide runtime environment. It is the implementation of JVM. It physically exists. It contains set of libraries + other files that JVM uses at runtime. Implementations of JVMs are also actively released by other companies besides Sun Micro Systems.



JDK, JRE and JVM

Java Virtual Machine (JVM):

JVM (Java Virtual Machine) acts as a run-time engine to run Java applications. JVM is the one that actually calls the main method present in a java code. JVM is a part of JRE (Java Runtime Environment).

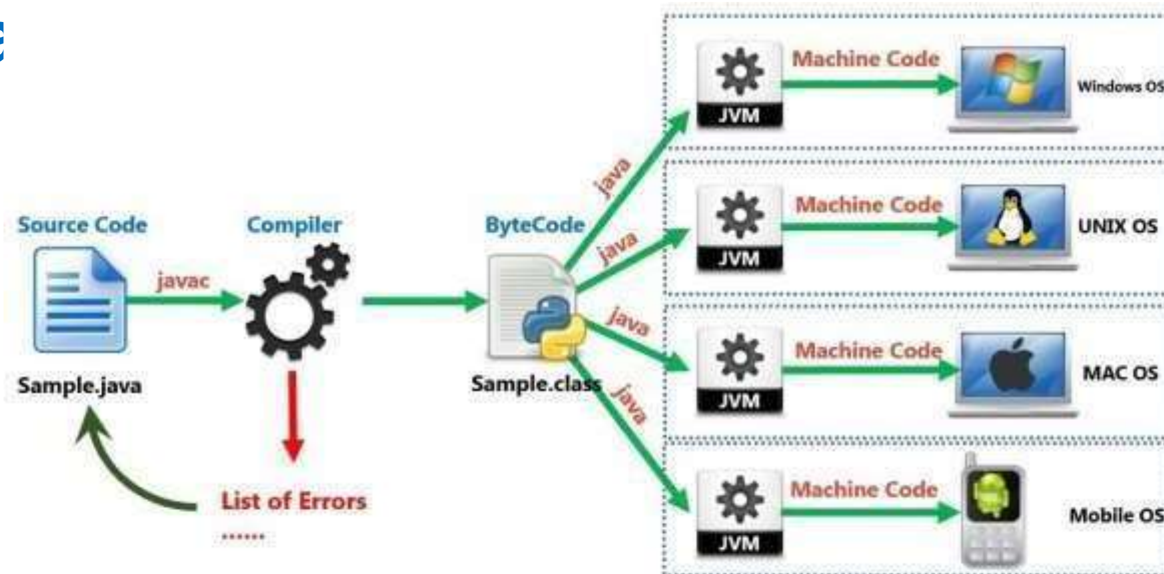
Java applications are called WORA (Write Once Run Anywhere). This means a programmer can develop Java code on one system and can expect it to run on any other Java enabled system without any adjustment. This is all possible because of JVM.

NOTE: JVM, JRE and JDK are platform dependent because configuration of each OS differs. But, Java is platform independent

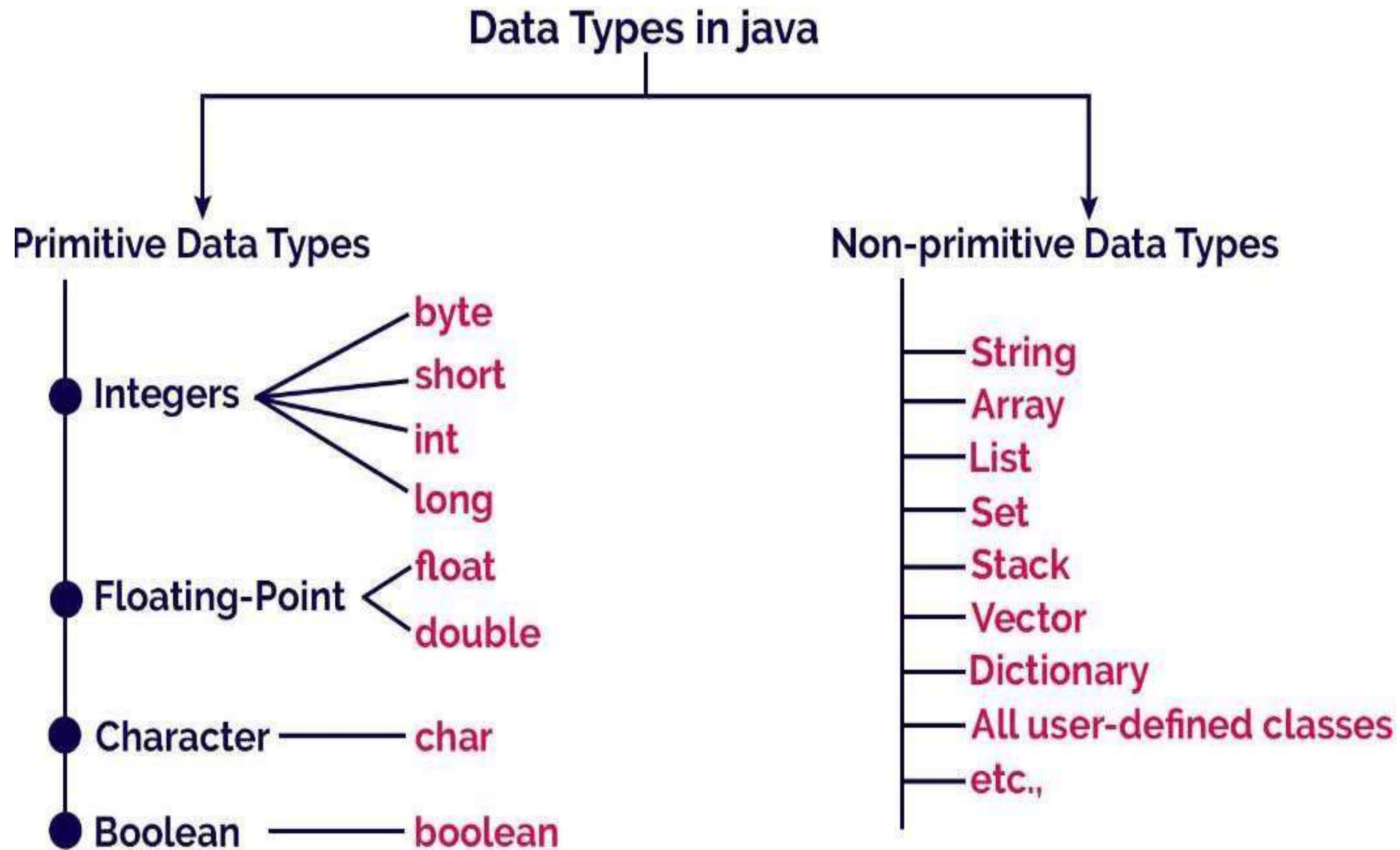
Execution Process of Java Program

The following three steps are used to create and execute a java program.

- Create a source code (.java file).
- Compile the source code using javac command.
- Run or execute .class file uisng java comma



JAVA DATA TYPES



JAVA VARIABLES

A variable is a named memory location used to store a data value. A variable can be defined as a container that holds a data value.

In java, we use the following syntax to create variables.

Syntax

data_type
variable_name; (or)
data_type variable_name_1,
variable_name_2,...; (or)
data_type variable_name =
value; (or)
data_type variable_name_1 = value,
variable_name_2 = value,...;

JAVA ARRAYS

Creating an array
In the java programming language, an array must be created using new operator and with a specific size. The size must be an integer value but not a byte, short, or long. We use the following syntax to create an array.

Syntax
data_type array_name[ ] = new data_type[size]; (or)
data_type[ ] array_name = new data_type[size];

In java, an array can also be initialized at the time of its declaration. When an array is initialized at the time of its declaration, it need not specify the size of the array and use of the new operator. Here, the size is automatically decided based on the number of values that are initialized.

Example: int list[ ] = {10, 20, 30, 40, 50};

Multidimensional Array

In java, we can create an array with multiple dimensions. We can create 2-
dimensional, 3-dimensional, or any dimensional array.
In Java, multidimensional arrays are arrays of arrays. To create a multidimensional array variable, specify each additional index using another set of square brackets. We use the following syntax to create two-dimensional array.

Syntax

data_type array_name[ ][ ] = new
data_type[rows][columns]; (or)
data_type[ ][ ] array_name = new data_type[rows][columns];

When we create a two-dimensional array, it created with a separate index for rows and columns. The individual element is accessed using the respective row index followed by the column index. A multidimensional array can be initialized while it has created using the following syntax.

JAVA OPERATORS

An operator is a symbol used to perform arithmetic and logical operations. Java provides a rich set of operators. In java, operators are classified into the following four types.

- Arithmetic Operators
- Relational (or) Comparison Operators
- Logical Operators
- Assignment Operators
- Bitwise Operators
- Conditional Operators

Arithmetic Operators

Operator	Meaning	Example
+	Addition	10 + 5 = 15
-	Subtraction	10 - 5 = 5
*	Multiplication	10 * 5 = 50
/	Division	10 / 5 = 2
%	Modulus - Remainder of the Division	5 % 2 = 1
++	Increment	a++
--	Decrement	a--

Relational Operators (<, >, <=, >=, ==, !=)

Operator	**Meaning**	**Example**
<	Returns TRUE if the first value is smaller than second value otherwise returns FALSE	10 < 5 is FALSE
>	Returns TRUE if the first value is larger than second value otherwise returns FALSE	10 > 5 is TRUE
<=	Returns TRUE if the first value is smaller than or equal to second value otherwise returns FALSE	10 <= 5 is FALSE
>=	Returns TRUE if the first value is larger than or equal to second value otherwise returns FALSE	10 >= 5 is TRUE
==	Returns TRUE if both values are equal otherwise returns FALSE	10 == 5 is FALSE
!=	Returns TRUE if both values are not equal otherwise returns FALSE	10 != 5 is TRUE

Logical Operators

Operator	Meaning	Example
&	**Logical AND** - Returns TRUE if all conditions are TRUE otherwise returns FALSE	false & true => false
|	**Logical OR** - Returns FALSE if all conditions are FALSE otherwise returns TRUE	false | true => true
^	**Logical XOR** - Returns FALSE if all conditions are same otherwise returns TRUE	true ^ true => false
!	**Logical NOT** - Returns TRUE if condition is FLASE and returns FALSE if it is TRUE	!false => true
&&	**short-circuit AND** - Similar to Logical AND (&), but once a decision is finalized it does not evaluate remianing.	false & true => false
||	**short-circuit OR** - Similar to Logical OR (|), but once a decision is finalized it does not evaluate remianing.	false | true => true

Assignment Operators

Operator	Meaning	Example
=	Assign the right-hand side value to left-hand side variable	A = 15
+=	Add both left and right-hand side values and store the result into left-hand side variable	A += 10
-=	Subtract right-hand side value from left-hand side variable value and store the result into left-hand side variable	A -= B
*=	Multiply right-hand side value with left-hand side variable value and store the result into left-hand side variable	A *= B
/=	Divide left-hand side variable value with right-hand side variable value and store the result into the left-hand side variable	A /= B
%=	Divide left-hand side variable value with right-hand side variable value and store the remainder into the left-hand side variable	A %= B
&=	Logical AND assignment	-
|=	Logical OR assignment	-
^=	Logical XOR assignment	-

Bitwise Operators

Operator	Meaning	Example
&	the result of Bitwise AND is 1 if all the bits are 1 otherwise it is 0	A & B ⇒ 16 (10000)
|	the result of Bitwise OR is 0 if all the bits are 0 otherwise it is 1	A | B ⇒ 29 (11101)
^	the result of Bitwise XOR is 0 if all the bits are same otherwise it is 1	A ^ B ⇒ 13 (01101)
~	the result of Bitwise once complement is negation of the bit (Flipping)	~A ⇒ 6 (00110)
<<	the Bitwise left shift operator shifts all the bits to the left by the specified number of positions	A << 2 ⇒ 100 (1100100)
>>	the Bitwise right shift operator shifts all the bits to the right by the specified number of positions	A >> 2 ⇒ 6 (00110)

Conditional Operators

The conditional operator is also called a **ternary operator** because it requires three operands.
This operator is used for decision making. In this operator, first, we verify a condition, then we perform one operation out of the two operations based on the condition result.
If the condition is TRUE the first option is performed, if the condition is FALSE the second option is performed.

Syntax
Condition ? TRUE Part : FALSE Part;

JAVA EXPRESSIONS

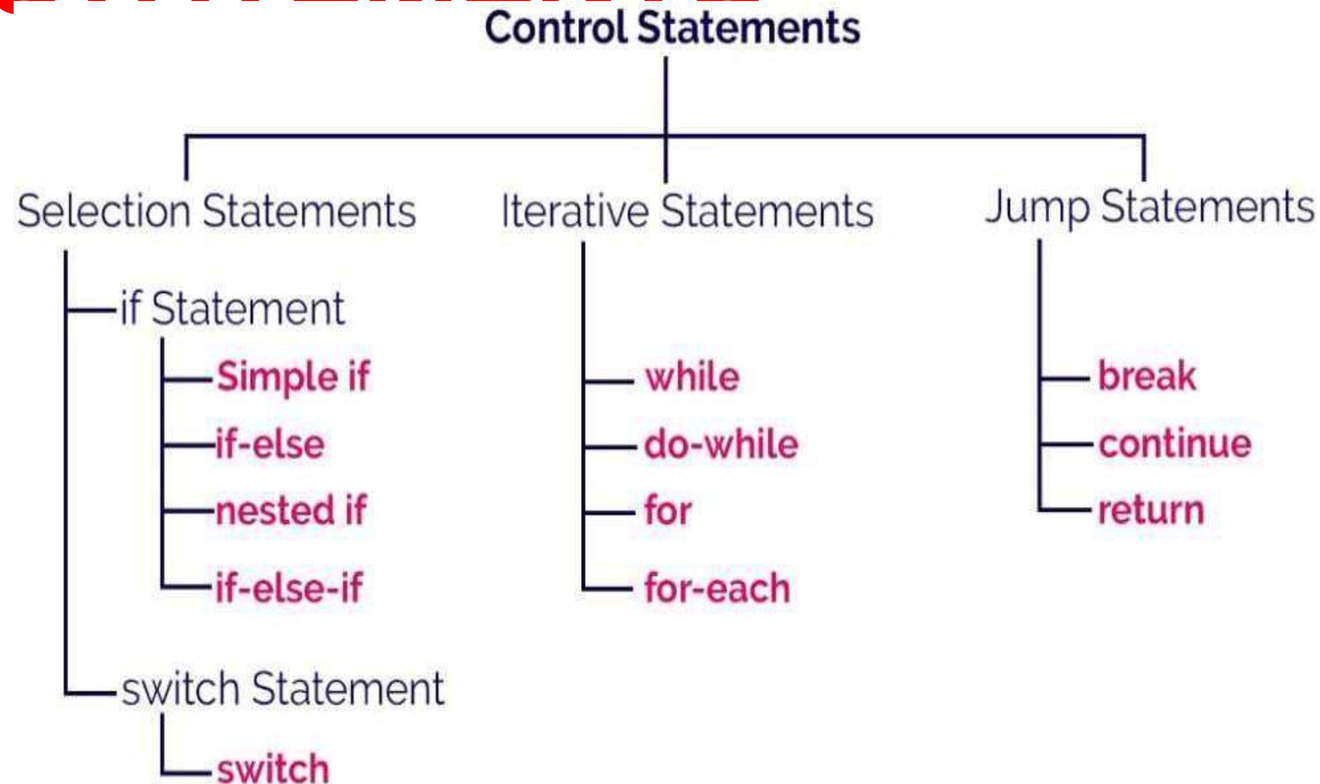
- In the java programming language, an expression is a collection of operators and operands that represents a specific value.

Expression Types

In the java programming language, expressions are divided into THREE types. They are as follows.

- Infix Expression
- Postfix Expression
- Prefix Expression

JAVA CONTROL STATEMENTS

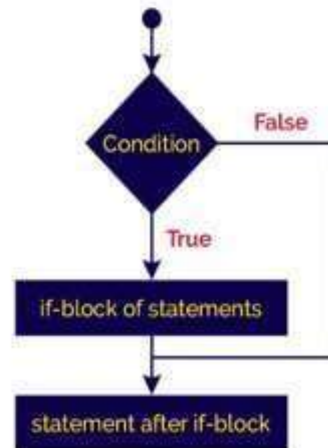


JAVA CONTROL STATEMENTS

if statement

Syntax

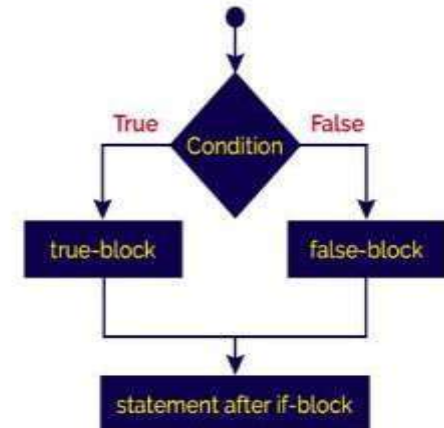
```
if(condition){
    if-block of statements;
    ...
}
statement after if-block;
...
```



if-else statement

Syntax

```
if(condition){
    true-block of statements;
    ...
}
else{
    false-block of statements;
    ...
}
statement after if-block;
...
```

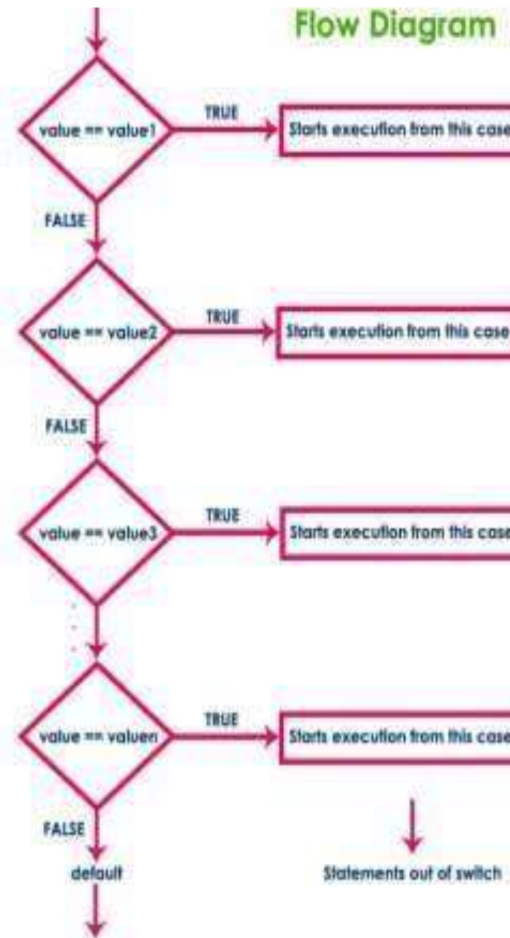


JAVA CONTROL STATEMENTS

Switch statement

Syntax

```
switch ( expression or value )
{
        case  value1:  set of statements;
                       ....
        case  value2:  set of statements;
                       ....
        case  value3:  set of statements;
                       ....
        case  value4:  set of statements;
                       ....
        case  value5:  set of statements;
                       ....
        .
        .
        default: set of statements;
}
```

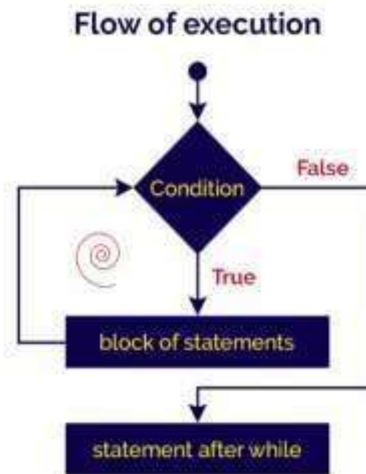


JAVA CONTROL STATEMENTS

while statement

Syntax

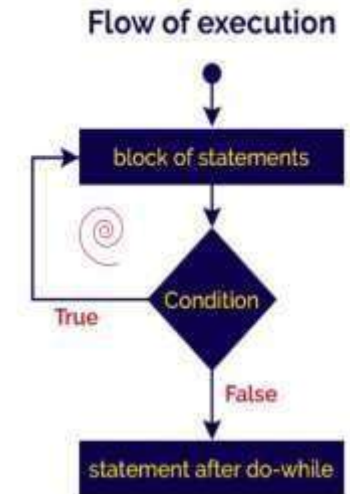
```
while(boolean-expression){
      block of statements;
      ...
}
statement after while;
...
```



do-while statement

Syntax

```
do{
      block of statements;
      ...
}while(boolean-expression);
statement after do-while;
...
```



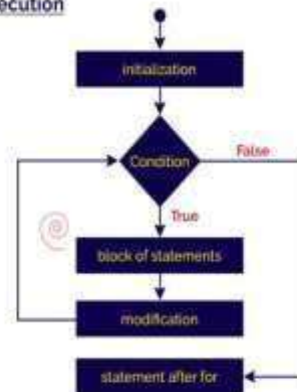
JAVA CONTROL STATEMENTS

for statement

Syntax

```
for(initialization; boolean-expression; modofication){
      block of statements;
      ...
}
statement after for;
...
```

Flow of execution

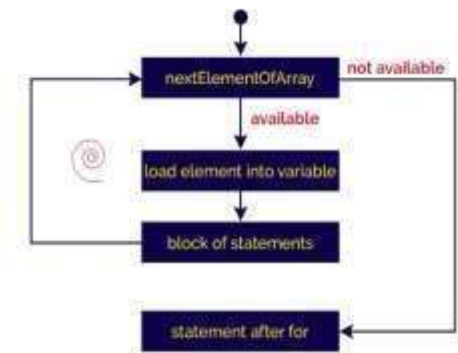


for-each statement

Syntax

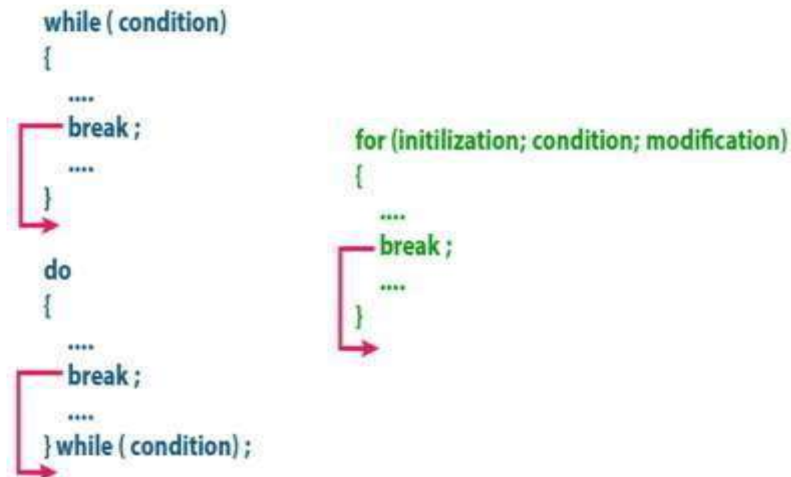
```
for( dataType  variableName : Array ){
      block of statements;
      ...
}
statement after for;
...
```

Flow of execution

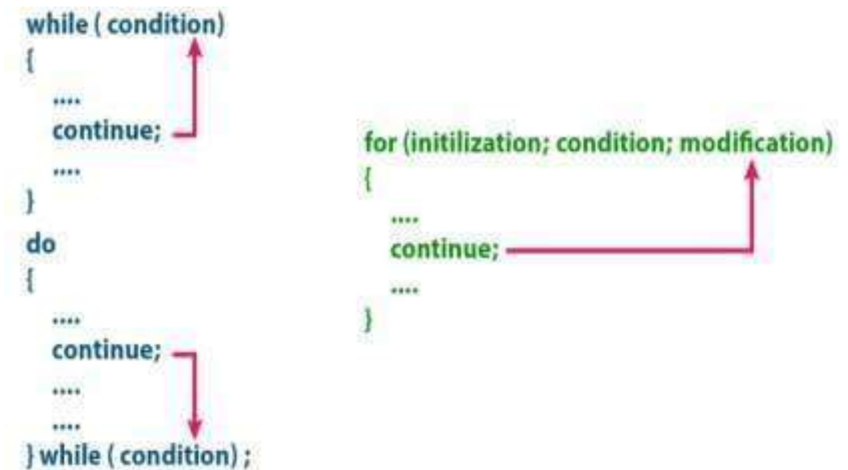


JAVA CONTROL STATEMENTS

break statement



continue statement



JAVA CLASSES

- Java is an object-oriented programming language, so everything in java
  program must be based on the object concept.
- In a java programming language, the class concept defines the skeleton
  of an object.
- The java class is a template of an object. The class defines the blueprint
  of an object. Every class in java forms a new data type.
- Once a class got created, we can generate as many objects as we want.
  Every class defines the properties and behaviors of an object.
- All the objects of a class have the same properties and behaviors that
  were defined in the class.

Every class of java programming language has the following

JAVA CLASSES

Creating a Class

In java, we use the keyword class to create a class. A class in java contains properties as variables and behaviors as methods. Following is the syntax of class in the java.

Syntax

```
class <ClassName>{
   data members declaration;
   methods defination;
}
```

The ClassName must begin with an alphabet, and the Upper-case letter is
preferred.
The ClassName must follow all naming rules.

JAVA CLASSES

Creating an Object

In java, an object is an instance of a class. When an object of a class is created, the class is said to be instantiated. All the objects that are created using a single class have the same properties and methods. But the value of properties is different for every object. Following is the syntax of class in the java.

Syntax

<ClassName> <objectName> = new <ClassName>( );

The objectName must begin with an alphabet, and a Lower-case letter is preferred.

The objectName must follow all naming rules.

JAVA METHODS

A method is a block of statements under a name that gets executes only when it is called. Every method is used to perform a specific task. The major advantage of methods is code re-usability (define the code once, and use it many times).

In a java programming language, a method defined as a behavior of an
object. That means, every method in java must belong to a class. Every method in java must be declared inside a class.

Every method declaration has the following characteristics. **returnType** - Specifies the data type of a return value. **name** - Specifies a unique name to identify it.

parameters - The data values it may accept or recieve.

{ } - Defienes the block belongs to the method.

JAVA METHODS

Creating a method

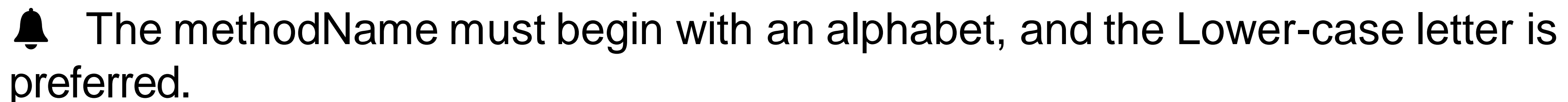
A method is created inside the class and it may be created with any access specifier.

However, specifying access specifier is optional.

Following is the syntax for creating methods in java.

Syntax

```
class <ClassName>{
   <accessSpecifier> <returnType> <methodName>( parameters ){
      ...
      block of statements;
      ...
   }
}
```


The methodName must begin with an alphabet, and the Lower-case letter is preferred.

The methodName must follow all naming rules.

If you don't want to pass parameters, we ignore it.

If a method defined with return type other than void, it must contain the return statement, otherwise, it may be ignored.

CSE/NRCM

JAVA METHODS

Calling a method

In java, a method call precedes with the object name of the class to which it belongs and a dot operator. It may call directly if the method defined with the static modifier. Every method call must be made, as to the method name with parentheses (), and it must terminate with a semicolon.

Syntax

<objectName>.<methodName>(actualArguments);

- The method call must pass the values to parameters if it has.
- If the method has a return type, we must provide the receiver.
- The objectName must begin with an alphabet, and a Lower-case letter is preferred.
- The objectName must follow all naming rules.

Variable arguments of a method

In java, a method can be defined with a variable number of arguments. That means creating a method that receives any number of arguments of the same data type.

Syntax

<returnType> <methodName>(dataType...parameterName);

When a method has both the normal parameter and variable-argument, then the variable

Constructor

A constructor is a special method of a class that has the same name as the class name. The constructor gets executes automatically on object creation. It does not require the explicit method call. A constructor may have parameters and access specifiers too. In java, if you do not provide any constructor the compiler automatically creates a default constructor. **A constructor cannot have return value.**

Let's look at the following example java code.

Example

```
public class ConstructorExample {

        ConstructorExample() {
                System.out.println("Object created!");
        }
        public static void main(String[] args) {

                ConstructorExample obj1 = new ConstructorExample();
                ConstructorExample obj2 = new ConstructorExample();
        }
}
```

JAVA STRING HANDLING

A string is a sequence of characters surrounded by double quotations. In a java programming language, a string is the object of a built-in class **String**.

In the background, the string values are organized as an array of a character data type. The string created using a character array cannot be extended. It does not allow to append more characters after its definition, but it can be modified.

Let's look at the following example java code.

Example

```
char[] name = {'J', 'a', 'v', 'a', ' ', 'T', 'u', 't', 'o', 'r', 'i', 'a', 'l', 's'};
//name[14] = '@';          //ArrayIndexOutOfBoundsException
name[5] = '-';
System.out.println(name);
```

The **String** class defined in the package **java.lang** package.

Creating String object in java

Creating String object in java

In java, we can use the following two ways to create a string object.

Using string literal

Using String constructor

Let's look at the following example java code.

Example

String title = "Java Tutorials";   // Using literals

String siteName = new String("www.btechsmartclass.com");   // Using constructor

The String class constructor accepts both string and character array as an argument.

String handling methods

Method	Description	Return Value
charAt(int)	Finds the character at given index	char
length()	Finds the length of given string	int
compareTo(String)	Compares two strings	int
compareToIgnoreCase(String)	Compares two strings, ignoring case	int
concat(String)	Concatenates the object string with argument string.	String
contains(String)	Checks whether a string contains sub-string	boolean
contentEquals(String)	Checks whether two strings are same	boolean
equals(String)	Checks whether two strings are same	boolean
equalsIgnoreCase(String)	Checks whether two strings are same, ignoring case	boolean
startsWith(String)	Checks whether a string starts with the specified string	boolean
endsWith(String)	Checks whether a string ends with the specified string	boolean
getBytes()	Converts string value to bytes	byte[]
hashCode()	Finds the hash code of a string	int
indexOf(String)	Finds the first index of argument string in object string	int
lastIndexOf(String)	Finds the last index of argument string in object string	int

String handling methods

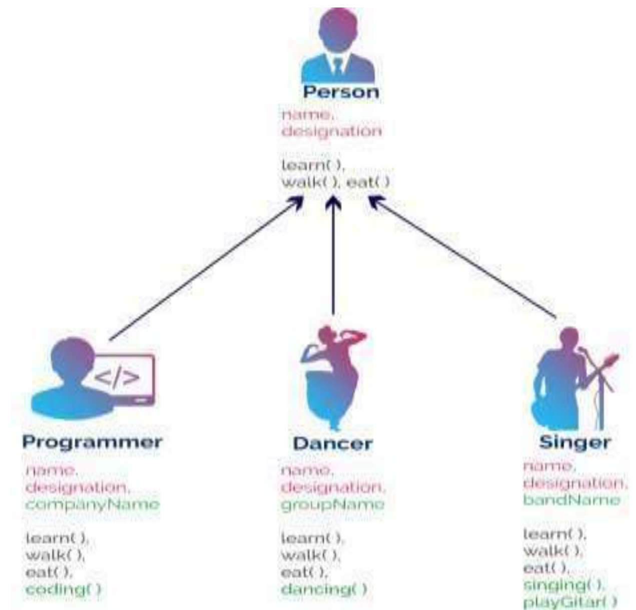
Method	Description	Return Value
isEmpty()	Checks whether a string is empty or not	boolean
replace(String, String)	Replaces the first string with second string	String
replaceAll(String, String)	Replaces the first string with second string at all occurrences.	String
substring(int, int)	Extracts a sub-string from specified start and end index values	String
toLowerCase()	Converts a string to lower case letters	String
toUpperCase()	Converts a string to upper case letters	String
trim()	Removes whitespace from both ends	String
toString(int)	Converts the value to a String object	String
split(String)	splits the string matching argument string	String[]
intern()	returns string from the pool	String
join(String, String, ...)	Joins all strings, first string as delimiter.	String

UNIT – II
INHERITANCE, PACKAGES, INTERFACES

JAVA INHERITANCE

The inheritance can be defined as follows. **The inheritance is the acquiring process of properties of one th class to another class. Inheritance e Basics**

In inheritance, we use the terms like parent class, child class, base class, derived class, superclass, and subclass. The **Parent class** is the class which provides features to another class. The parent class is also known as **Base class** or **Superclass**.



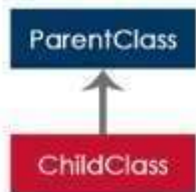
The **Child class** is the class which receives features from another class. The child class is also known as the **Derived Class** or **Subclass**.

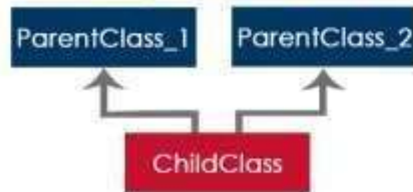
In the inheritance, the child class acquires the features from its parent class. But the parent class never acquires the features from its child class.

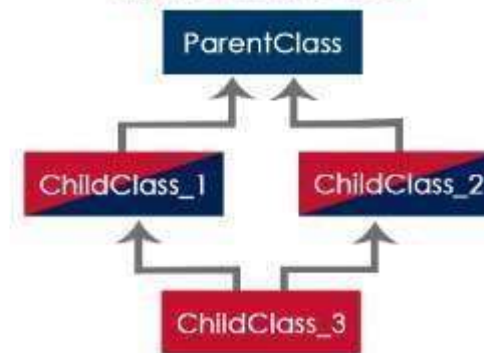
TYPES OF INHERITANCES

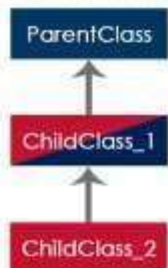
There are five types of inheritances, and they are as follows.

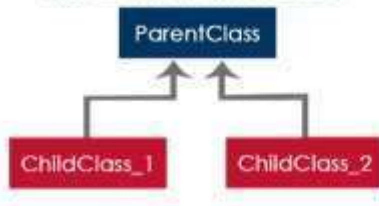
- **Simple Inheritance (or) Single Inheritance**
- **Multiple Inheritance**
- **Multi-Level Inheritance**
- **Hierarchical Inheritance**
- **Hybrid Inheritance**











Creating Child Class in java

In java, we use the keyword **extends** to create a child class. The following syntax used to
create a child class in java.

Syntax

```
class <ChildClassName> extends <ParentClassName>{
   ...
   //Implementation of child class
   ...
}
```

In a java programming language, a class extends only one class. Extending multiple classes is not allowed in java.

JAVA ACCESS MODIFIERS

In Java, the access specifiers (also known as access modifiers) used to restrict the scope or accessibility of a class, constructor, variable, method or data member of class and interface. There are four access specifiers, and their list is below.

default (or) no modifier
public
protecte
d
Private

Access control for members of class and interface in java

Access Specifier \ Accessibility Location	Same Class	Same Package		Other Package	
		Child class	Non-child class	Child class	Non-child class
Public	Yes	Yes	Yes	Yes	Yes
Protected	Yes	Yes	Yes	Yes	No
Default	Yes	Yes	Yes	No	No
Private	Yes	No	No	No	No

The **public** members can be accessed everywhere.

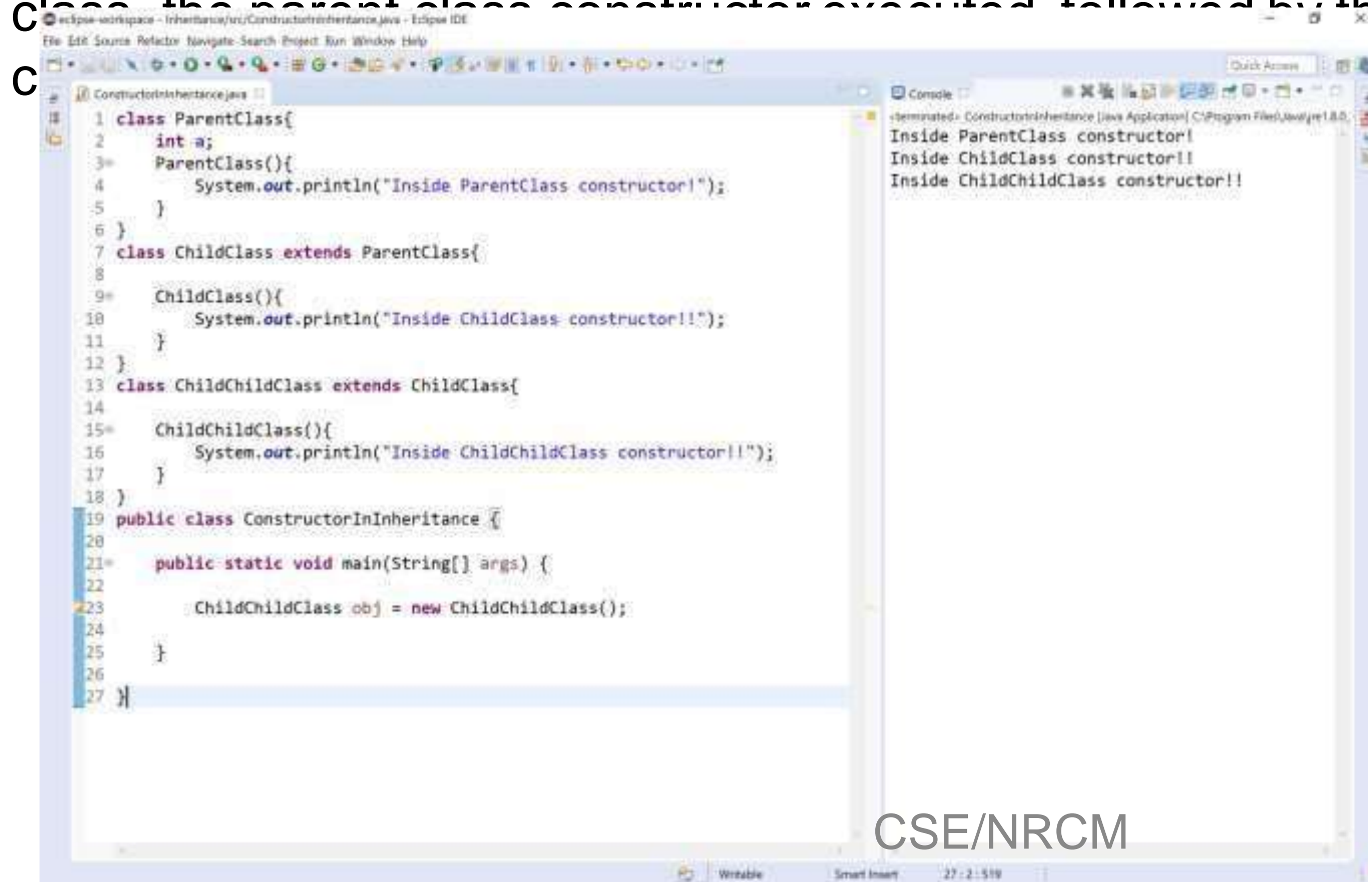
- The **private** members can be accessed only inside the same class.
- The **protected** members are accessible to every child class (same package or other
packages).
- The **default** members are accessible within the same package but not outside the
package.

JAVA CONSTRUCTORS IN INHERITANCE

It is very important to understand how the constructors get executed in the inheritance
concept.

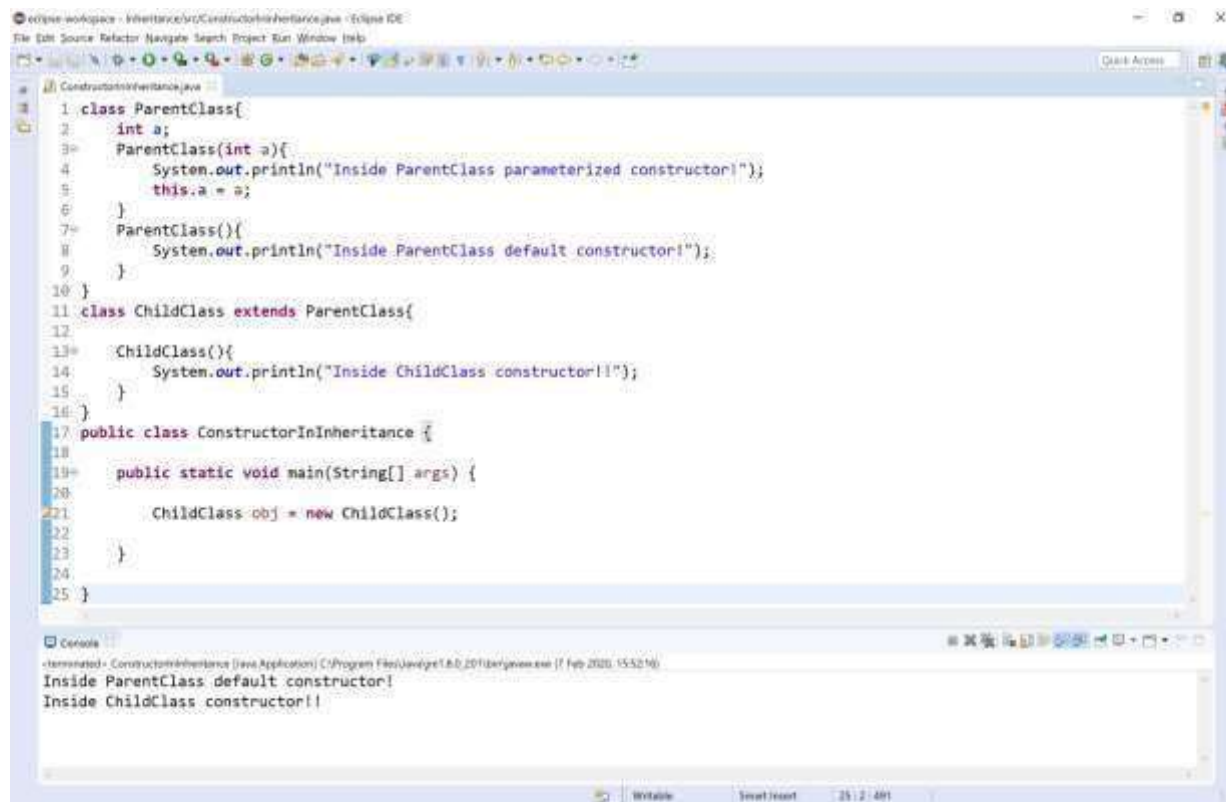
In the inheritance, the constructors never get inherited to any child class.

In java, the default constructor of a parent class called automatically by the constructor of its child class. That means when we create an object of the child class, the parent class constructor executed, followed by the child class
c



JAVA CONSTRUCTORS IN INHERITANCE

However, if the parent class contains default and parameterized constructor, then only the default constructor called automatically by the child class constructor.
Let's look at the following example java code.
The parameterized constructor of parent class must be called explicitly using the **super** keyword.



JAVA SUPER KEYWORD

In java, super is a keyword used to refers to the parent class object. The super keyword came into existence to solve the naming conflicts in the inheritance. When both parent class and child class have members with the same name, then the super keyword is used to refer to the parent class version.
In java, the super keyword is used for the following purposes.

- **To refer parent class data members**
- **To refer parent class methods**
- **To call parent class constructor**

🔔 The **super** keyword is used inside the child class only.

super to refer parent class data members
When both parent class and child class have data members with the same name, then the super keyword is used to refer to the parent class data member from child class. **super to refer parent class method**
When both parent class and child class have method with the same name, then the
super keyword is used to refer to the parent class method from child class.
super to call parent class constructor
When an object of child class is created, it automatically calls the parent class default- constructor before it's own. But, the parameterized constructor of parent class must be called explicitly using the **super** keyword inside the child class

JAVA FINAL KEYWORD

In java, the final is a keyword and it is used with the following things.

- With variable (to create constant)
- With method (to avoid method overriding)
- With class (to avoid inheritance)

final with variables

When a variable defined with the **final** keyword, it becomes a constant, and it does not allow us to modify the value. The variable defined with the final keyword allows only a one-time assignment, once a value assigned to it, never allows us to change it again.

final with methods

When a method defined with the **final** keyword, it does not allow it to override. The final method extends to the child class, but the child class can not override or re-define it. It must be used as it has implemented in the parent class.

final with class

When a class defined with final keyword, it can not be extended by any other class.

JAVA PYLYMORPHISM

The polymorphism is the process of defining same method with different implementation. That
means creating multiple methods with different behaviors.
In java, polymorphism implemented using method overloading and method overriding.

Ad hoc polymorphism

The ad hoc polymorphism is a technique used to define the same method with different implementations and different arguments. In a java programming language, ad hoc polymorphism carried out with a method overloading concept.
In ad hoc polymorphism the method binding happens at the time of compilation. Ad hoc polymorphism is also known as compile-time polymorphism. Every function call binded with the respective overloaded method based on the arguments. The ad hoc polymorphism implemented within the class only.

Pure polymorphism

The pure polymorphism is a technique used to define the same method with the same arguments but different implementations. In a java programming language, pure polymorphism carried out with a method overriding concept. In pure polymorphism, the method binding happens at run time. Pure polymorphism is also known as run-time polymorphism. Every function call binding with the respective overridden method based on the object reference.

When a child class has a definition for a member function of the parent class, the parent class

JAVA METHOD OVERRIDING

The method overriding is the process of re-defining a method in a child class that is already defined in the parent class. When both parent and child classes have the same method, then that method is said to be the overriding method.
The method overriding enables the child class to change the implementation of the method
which aquired from parent class according to its requirement.
In the case of the method overriding, the method binding happens at run time. The method binding which happens at run time is known as late binding. So, the method overriding follows late binding.
The method overriding is also known as **dynamic method dispatch** or **run time polymorphism** or **pure polymorphism**.

JAVA METHOD OVERRIDING

Rules for method overriding

- While overriding a method, we must follow the below list of rules.
- Static methods can not be overridden.
- Final methods can not be overridden.
- Private methods can not be overridden.
- Constructor can not be overridden.
- An abstract method must be overridden.
- Use super keyword to invoke overridden method from child class.
- The return type of the overriding method must be same as the parent has it.
- The access specifier of the overriding method can be changed, but the visibility must increase but not decrease. For example, a protected method in the parent class can be made public, but not private, in the child class.
- If the overridden method does not throw an exception in the parent class, then the child class overriding method can only throw the unchecked exception, throwing a checked exception is not allowed.
- If the parent class overridden method does throw an exception, then the child class overriding method can only throw the same, or subclass exception, or it may not throw any exception.

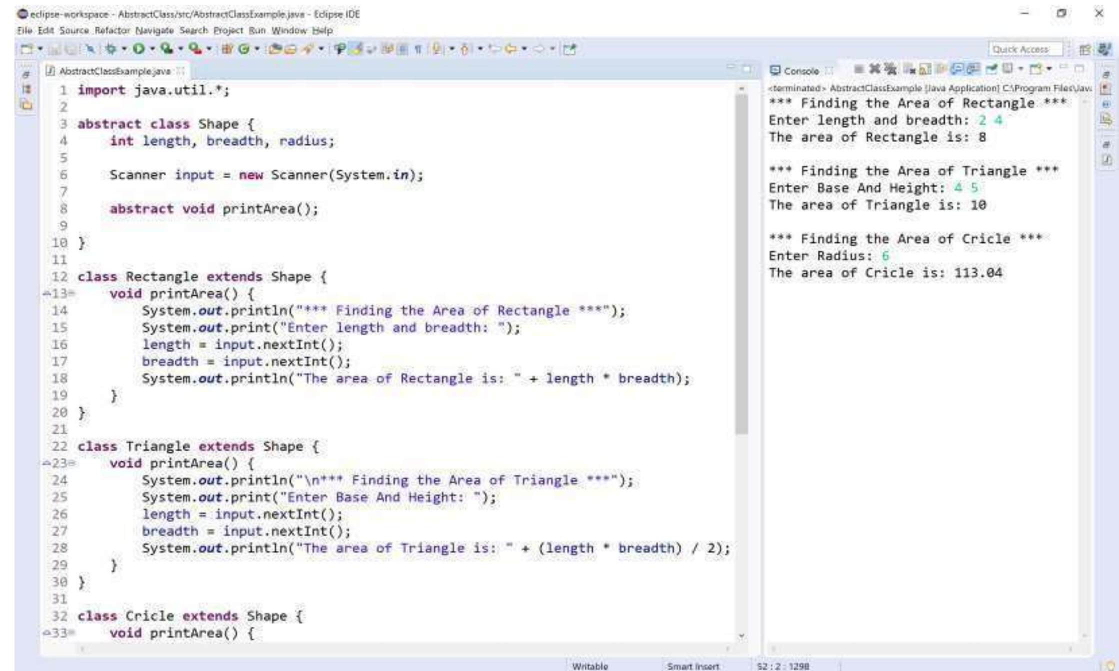
JAVA ABSTRACT CLASS

An abstract class is a class that created using abstract keyword. In other words, a class prefixed
with abstract keyword is known as an abstract class.
In java, an abstract class may contain abstract methods (methods without implementation) and
also non-abstract methods (methods with implementation).
We use the following syntax to create an abstract class.

Syntax

```
abstract class <ClassName>{
    ...
}
```

An abstract class cannot be instantiated but can be referenced. That means we can not create an object of an abstract class, but base reference can be created.



JAVA ABSTRACT CLASS

Rules for method overriding

- An abstract class must follow the below list of rules.
- An abstract class must be created with abstract keyword.
- An abstract class can be created without any abstract method.
- An abstract class may contain abstract methods and non-abstract methods.
- An abstract class may contain final methods that can not be overridden.
- An abstract class may contain static methods, but the abstract method can not be static.
- An abstract class may have a constructor that gets executed when the child class object created.
- An abstract method must be overridden by the child class, otherwise, it must be defined as an abstract class.
- An abstract class can not be instantiated but can be referenced.

Java Object Class

In java, the Object class is the super most class of any class hierarchy. The Object class in the
java programming language is present inside the **java.lang** package.
Every class in the java programming language is a subclass of Object class by default.
The Object class is useful when you want to refer to any object whose type you don't know.
Because it is the superclass of all other classes in java, it can refer to any type of object.

Methods of Object class

The following table depicts all built-in methods of Object class in java.

Method	Description	Return Value
getClass()	Returns Class class object	object
equals(Object obj)	compares the argument object to calling object.	boolean
concat(String)	Creates copy of invoking object	object
toString()	eturns the string representation of invoking object.	String
notify()	wakes up a thread, waiting on invoking object's monitor.	void
notifyAll()	wakes up all the threads, waiting on invoking object's monitor.	void
wait()	causes the current thread to wait, until another thread notifies.	void
wait(long,int)	causes the current thread to wait for the specified milliseconds and nanoseconds, until another thread notifies.	void
finalize()	It is invoked by the garbage collector before an object is being garbage collected.	void

JAVA FORMS OF INHERITANCE

The inheritance concept used for the number of purposes in the java programming language. One of the main purposes is substitutability. The substitutability means that when a child class acquires properties from its parent class, the object of the parent class may be substituted with the child class object. For example, if B is a child class of A, anywhere we expect an instance of A we can use an instance of B.
The substitutability can achieve using inheritance, whether using extends or implements keywords.
The following are the different forms of inheritance in java.

- Specialization
- Specification
- Construction
- Extension
- Limitation
- Combination

JAVA FORMS OF INHERITANCE

Specialization
It is the most ideal form of inheritance. The subclass is a special case of the parent class. It
holds the principle of substitutability.
Specification
This is another commonly used form of inheritance. In this form of inheritance, the parent class just specifies which methods should be available to the child class but doesn't implement them. The java provides concepts like abstract and interfaces to support this form of inheritance. It holds the principle of substitutability.
Construction
This is another form of inheritance where the child class may change the behavior defined by the parent class (overriding). It does not hold the principle of substitutability.
Extension
This is another form of inheritance where the child class may add its new properties. It holds
the principle of substitutability.
Limitation
This is another form of inheritance where the subclass restricts the inherited behavior. It does
not hold the principle of substitutability.

JAVA FORMS OF INHERITANCE

Specialization
It is the most ideal form of inheritance. The subclass is a special case of the parent class. It
holds the principle of substitutability.
Specification
This is another commonly used form of inheritance. In this form of inheritance, the parent class just specifies which methods should be available to the child class but doesn't implement them. The java provides concepts like abstract and interfaces to support this form of inheritance. It holds the principle of substitutability.
Construction
This is another form of inheritance where the child class may change the behavior defined by the parent class (overriding). It does not hold the principle of substitutability.
Extension
This is another form of inheritance where the child class may add its new properties. It holds
the principle of substitutability.
Limitation
This is another form of inheritance where the subclass restricts the inherited behavior. It does
not hold the principle of substitutability.

Benefits and Costs of Inheritance in java

Benefits of Inheritance

- Inheritance helps in code reuse. The child class may use the code defined in the parent
  class without re-writing it.
- Inheritance can save time and effort as the main code need not be written again.
- Inheritance provides a clear model structure which is easy to understand.
- An inheritance leads to less development and maintenance costs.
- With inheritance, we will be able to override the methods of the base class so that the meaningful implementation of the base class method can be designed in the derived class. An inheritance leads to less development and maintenance costs.
- In inheritance base class can decide to keep some data private so that it cannot be altered by the derived class.

Costs of Inheritance

- Inheritance decreases the execution speed due to the increased time and effort it takes,
  the program to jump through all the levels of overloaded classes.
- Inheritance makes the two classes (base and inherited class) get tightly coupled. This
  means one cannot be used independently of each other.
- The changes made in the parent class will affect the behavior of child class too.
- The overuse of inheritance makes the program more complex.

DEFINING PACKAGES

In java, a package is a container of classes, interfaces, and sub-packages. We may think of it as a
folder in a file directory.
We use the packages to avoid naming conflicts and to organize project-related classes, interfaces, and sub-packages into a bundle.
In java, the packages have divided into two
types. Built-in Packages
User-defined Packages

Built-in Packages
The built-in packages are the packages from java API. The Java API is a library of pre-defined classes, interfaces, and sub-packages. The built-in packages were included in the JDK. There are many built-in packages in java, few of them are as java, lang, io, util, awt, javax, swing, net, sql, etc. We need to import the built-in packages to use them in our program. To import a package, we use the import statement.

User-defined Packages
The user-defined packages are the packages created by the user. User is free to create their own packages.

Definig/Creating a Package in java

We use the ***package*** keyword to create or define a package in java programming language.

Syntax

```
package packageName;
```

The package statement must be the first statement in the program. The package name must be a single word.
The package name must use Camel case notation.
Let's consider the following code to create a user-defined package myPackage.

Example

```
package myPackage;
public class DefiningPackage {
        public static void main(String[] args) {
                System.out.println("This class belongs to myPackage.");
        }
}
```

Now, save the above code in a file **DefiningPackage.java**, and compile it using the following command.

javac -d . DefiningPackage.java

The above command creates a directory with the package name ***myPackage***, and the ***DefiningPackage.class*** is saved into it. Run the program use the following command.

java myPackage.DefiningPackage

When we use IDE like Eclipse, Netbeans, etc. the package structure is created automatically.

ACCESS PROTECTION IN JAVA PACKAGES

In java, the access modifiers define the accessibility of the class and its members. For example, private members are accessible within the same class members only. Java has four access modifiers, and they are default, private, protected, and public.
In java, the package is a container of classes, sub-classes, interfaces, and sub-packages. The class acts as a container of data and methods. So, the access modifier decides the accessibility of class members across the different packages.
In java, the accessibility of the members of a class or interface depends on its access specifiers. The
following table provides information about the visibility of both data members and methods. The **public** members can be accessed everywhere.
The **private** members can be accessed only inside the same class.
The **protected** members are accessible to every child class (same package or other packages).
The **default** members are accessible within the same package but not outside the package.

Access control for members of class and interface in java

Access Specifier \ Accessibility Location	Same Class	Same Package		Other Package	
		Child class	Non-child class	Child class	Non-child class
Public	Yes	Yes	Yes	Yes	Yes
Protected	Yes	Yes	Yes	Yes	No
Default	Yes	Yes	Yes	No	No
Private	Yes	No	No	No	No

www.btechsmartclass.com

IMPORTING PACKAGES IN JAVA

In java, the ***import*** keyword used to import built-in and user-defined packages. When a package has
imported, we can refer to all the classes of that package using their name directly.
The import statement must be after the package statement, and before any other
statement. Using an import statement, we may import a specific class or all the
classes from a package.
Using one import statement, we may import only one package or a class.
Using an import statement, we cannot import a class directly, but it must be a part of a
package. A program may contain any number of import statements.

Importing specific class

Using an importing statement, we can import a specific class. The following syntax is employed to import a
specific class.

Syntax : import packageName.ClassName;

Let's look at an import statement to import a built-in package and Scanner class.

Example

```
package myPackage;
import
java.util.Scanner;
public class ImportingExample {
        public static void main(String[] args) {
                Scanner read = new
                Scanner(System.in); int i
```

IMPORTING PACKAGES IN JAVA

Importing all the classes

Using an importing statement, we can import all the classes of a package. To import all the classes of the
package, we use * symbol. The following syntax is employed to import all the classes of a package.

Syntax

```
import packageName.*;
```

Let's look at an import statement to import a built-in package.

Example

```
package myPackage;
import java.util.*;
public class ImportingExample {
        public static void main(String[] args) {
                Scanner read = new Scanner(System.in);
                int i = read.nextInt();
                System.out.println("You have entered a number "
                + i); Random rand = new Random();

                int num = rand.nextInt(100);
                System.out.println("Randomly generated number " + num);     }}
```

In the above code, the class **ImportingExample** belongs to **myPackage** package, and it also importing all
the classes like Scanner, Random, Stack, Vector, ArrayList, HashSet, etc. from the **java.util**

DEFINING AN INTERFACE IN JAVA

In java, an ***interface*** is similar to a class, but it contains abstract methods and static final variables only.
The interface in Java is another mechanism to achieve abstraction.
We may think of an interface as a completely abstract class. None of the methods in the interface has an implementation, and all the variables in the interface are constants.
All the methods of an interface, implemented by the class that implements it.
The interface in java enables java to support multiple-inheritance. An interface may extend only one interface, but a class may implement any number of interfaces.

- ❖ An interface is a container of abstract methods and static final variables.
- ❖ An interface, implemented by a class. (**class implements interface**).
- ❖ An interface may extend another interface. (**Interface extends Interface**).
- ❖ An interface never implements another interface, or class.
- ❖ A class may implement any number of interfaces.
- ❖ We can not instantiate an interface.
- ❖ Specifying the keyword abstract for interface methods is optional, it automatically added.
- ❖ All the members of an interface are public by default.

IMPLEMENTING AN INTERFACE IN JAVA

In java, an **interface** is implemented by a class. The class that implements an interface must provide code for all the methods defined in the interface; otherwise, it must be defined as an abstract class.
The class uses a keyword ***implements*** to implement an interface. A class can implement any
number of interfaces. When a class wants to implement more than one interface, we use
the ***implements*** keyword is followed by a comma-separated list of the interfaces implemented
by the class.

The following is the syntax for defineing a class that implements an interface.

Syntax

```
class className implements InterfaceName{
   ...
   boby-of-the-class
   ...
}
```

Implementing multiple Interfaces

Implementing multiple Interfaces

When a class wants to implement more than one interface, we use the ***implements*** keyword is followed by a comma-separated list of the interfaces implemented by the class.

The following is the syntax for defineing a class that implements multiple interfaces.

Syntax

```
class className implements InterfaceName1, InterfaceName2, ...{
   ...
   boby-of-the-class
   ...
}
```

NESTED INTERFACES IN JAVA

In java, an interface may be defined inside another interface, and also inside a class. The interface that defined inside another interface or a class is konwn as nested interface. The nested interface is also refered as inner interface.
The nested interface declared within an interface is public by default.
The nested interface declared within a class can be with any access modifier.
Every nested interface is static by default.
The nested interface cannot be accessed directly. We can only access the nested interface by using outer interface or outer class name followed by dot(.), followed by the nested interface name.

Nested interface inside another interface

The nested interface that defined inside another interface must be accessed as **OuterInterface.InnerInterface**.
Let's look at an example code to illustrate nested interfaces inside another interface.

Example

```
interface OuterInterface{
        void outerMethod();

        interface InnerInterface{
                void
                innerMethod();
}       }
```

VARIABLES IN JAVA INTERFACES

In java, an interface is a completely abstract class. An interface is a container of abstract methods and static final variables. The interface contains the static final variables. The variables defined in an interface cannot be modified by the class that implements the interface, but it may use as it defined in the interface.

The variable in an interface is public, static, and final by default.

If any variable in an interface is defined without public, static, and final keywords then, the compiler automatically adds the same.

No access modifier is allowed except the public for interface variables.

Every variable of an interface must be initialized in the interface itself.

The class that implements an interface can not modify the interface variable, but it may use as it defined
in the interface.

Example

```
interface SampleInterface{
        int UPPER_LIMIT = 100;
        //int LOWER_LIMIT; // Error - must be initialised
}

public class InterfaceVariablesExample implements SampleInterface{
        public static void main(String[] args) {
                System.out.println("UPPER LIMIT = " +
        }       UPPER_LIMIT);
        }       // UPPER_LIMIT = 150; // Can not be modified
```

EXTENDING AN INTERFACE IN JAVA

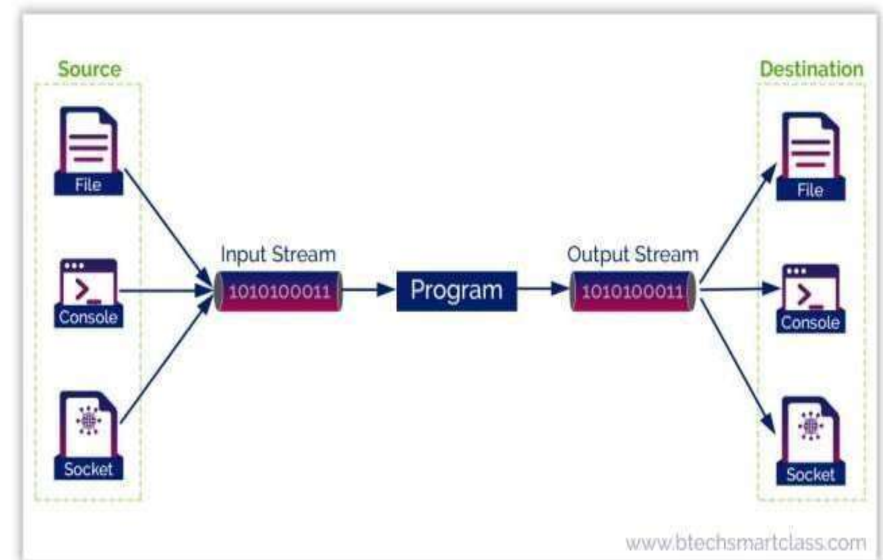
In java, an interface can extend another interface. When an interface wants to extend another interface, it uses the keyword ***extends***. The interface that extends another interface has its own members and all the members defined in its parent interface too. The class which implements a child interface needs to provide code for the methods defined in both child and parent interfaces, otherwise, it needs to be defined as abstract class.

- An interface can extend another interface.
- An interface cannot extend multiple interfaces.
- An interface can implement neither an interface nor a class.
- The class that implements child interface needs to provide code for all the methods defined in both
  child and parent interfaces.

STREAM IN JAVA

In java, the IO operations are performed using the concept of streams. Generally, a stream means a continuous flow of data. In java, a stream is a logical container of data that allows us to read from and write to it. A stream can be linked to a data source, or data destination, like a console, file or network connection by java IO system. The stream-based IO operations are faster than normal IO operations.
The Stream is defined in the java.io package.
To understand the functionality of java streams, look at the following picture.

In Java, every program creates 3 streams automatically, and these streams are attached to the console.
System.out: standard output stream for console output operations.
System.in: standard input stream for console input
operations.
System.err: standard error stream for console error
output operations.
The Java streams support many different kinds of data, including simple bytes, primitive data types, localized characters, and objects.



TYPES OF STREAMS

Java provides two types of streams, and they are as follows.

Byte Stream

Character Stream

BYTE STREAM IN JAVA

In java, the byte stream is an 8 bits carrier. The byte stream in java allows us to transmit 8 bits of data.

In Java 1.0 version all IO operations were byte oriented, there was no other stream (character stream). The java byte stream is defined by two abstract classes, **InputStream** and **OutputStream**. The InputStream class used for byte stream based input operations, and the OutputStream class used for byte stream based output operations.

The InputStream and OutputStream classes have several concreate classes to perform various IO operations based on the byte stream.

The following picture shows the classes used for byte stream operations

BYTE STREAM IN JAVA

InputStream class

The InputStream class has defined as an abstract class, and it has the following methods which have implemented by its concrete classes.

S.No.	Method with Description
1	**int available()** It returns the number of bytes that can be read from the input stream.
2	**int read()** It reads the next byte from the input stream.
3	**int read(byte[] b)** It reads a chunk of bytes from the input stream and store them in its byte array, b.
4	**void close()** It closes the input stream and also frees any resources connected with this input stream.

OutputStream class

The OutputStream class has defined as an abstract class, and it has the following methods which have implemented by its concrete classes.

S.No.	Method with Description
1	**void write(int n)** It writes byte(contained in an int) to the output stream.
2	**void write(byte[] b)** It writes a whole byte array(b) to the output stream.
3	**void flush()** It flushes the output steam by forcing out buffered bytes to be written out.
4	**void close()** It closes the output stream and also frees any resources connected with this output stream.

CHARACTER STREAM IN JAVA

In java, when the IO stream manages 16-bit Unicode characters, it is called a character stream. The unicode set is basically a type of character set where each character corresponds to a specific numeric value within the given character set, and every programming language has a character set.

In java, the character stream is a 16 bits carrier. The character stream in java allows us to transmit 16 bits of data.

The character stream was introduced in Java 1.1 version. The charater stream

The java character stream is defined by two abstract classes, **Reader** and **Writer**. The Reader class used for character stream based input operations, and the Writer class used for charater stream based output operations.

The Reader and Writer classes have several concreate classes to perform various IO operations based on

the character stream.

CHARACTER STREAM IN JAVA

Reader class

The Reader class has defined as an abstract class, and it has the following methods which have
implemented by its concrete classes.

S.No.	Method with Description
1	**int read()** It reads the next character from the input stream.
2	**int read(char[] cbuffer)** It reads a chunk of charaters from the input stream and store them in its byte array, cbuffer.
3	**int read(char[] cbuf, int off, int len)** It reads charaters into a portion of an array.
4	**int read(CharBuffer target)** It reads charaters into into the specified character buffer.
5	**String readLine()** It reads a line of text. A line is considered to be terminated by any oneof a line feed ('\n'), a carriage return ('\r'), or a carriage returnfollowed immediately by a linefeed.
6	**boolean ready()** It tells whether the stream is ready to be read.
7	**void close()** It closes the input stream and also frees any resources connected with this input stream.

CHARACTER STREAM IN JAVA

Writer class

The Writer class has defined as an abstract class, and it has the following methods which have
implemented by its concrete classes.

S.No.	Method with Description
1	**void flush()** It flushes the output steam by forcing out buffered bytes to be written out.
2	**void write(char[] cbuf)** It writes a whole array(cbuf) to the output stream.
3	**void write(char[] cbuf, int off, int len)** It writes a portion of an array of characters.
4	**void write(int c)** It writes single character.
5	**void write(String str)** It writes a string.
6	**void write(String str, int off, int len)** It writes a portion of a string.
7	**Writer append(char c)** It appends the specified character to the writer.
8	**Writer append(CharSequence csq)** It appends the specified character sequence to the writer
9	**Writer append(CharSequence csq, int start, int end)** It appends a subsequence of the specified character sequence to the writer.
10	**void close()** It closes the output stream and also frees any resources connected with this output stream.

CONSOLE I/O OPERATIONS IN JAVA

Reading console input in java

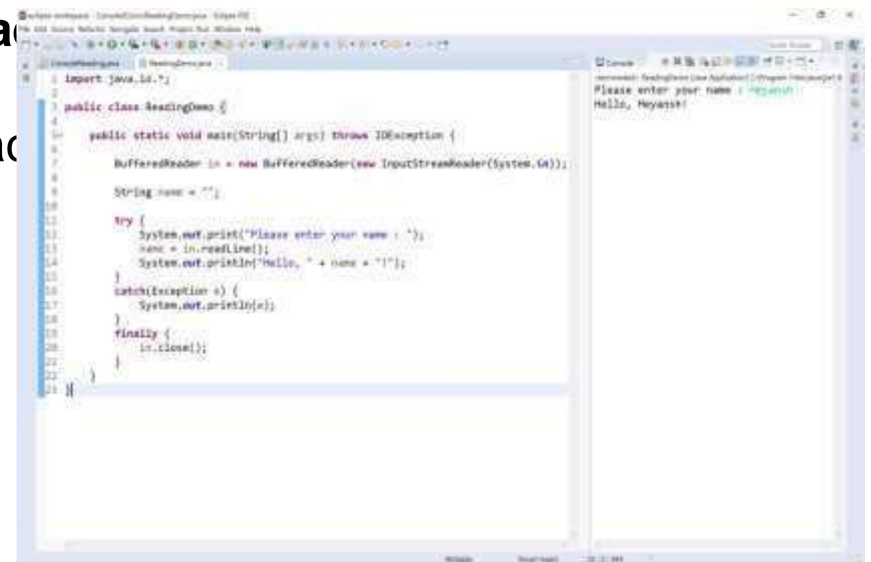
In java, there are three ways to read console input. Using the 3 following ways, we can read input data
from the console.

- Using BufferedReader class
- Using Scanner class
- Using Console class

Let's explore the each method to read data with example.

1. Reading console input using BufferedReader class in java

Reading input data using the **BufferedReader** class is the traditional technique. This way of the reading method is used by wrapping the **System.in** (standard input stream) in an **InputStreamReader** which is wrapped in a **BufferedRea**
console.
The **BufferedReader** class has defined in the **java.io** pac



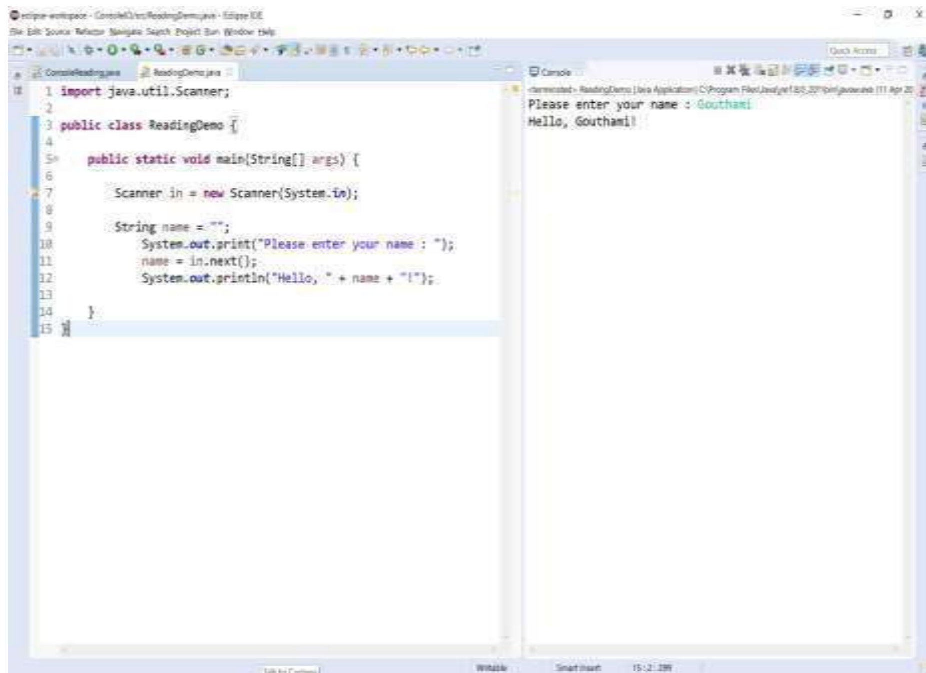
CONSOLE I/O OPERATIONS IN JAVA

2. Reading console input using Scanner class in java

Reading input data using the **Scanner** class is the most commonly used method. This way of the reading method is used by wrapping the **System.in** (standard input stream) which is wrapped in a **Scanner**, we can read input from the console.

The **Scanner** class has defined in the **java.util** package.

Consider the following example code to understand how to read console input using Scanner class.



CONSOLE I/O OPERATIONS IN JAVA

3. Reading console input using Console class in java

Reading input data using the **Console** class is the most commonly used method. This class was introduced
in Java 1.6 version.
The **Console** class has defined in the **java.io** package.
Consider the following example code to understand how to read console input using Console class.

Example

```
import java.io.*;
public class ReadingDemo {
		public static void main(String[] args) {
				String name;
	Console con = System.console();
				if(con != null) {
						name = con.readLine("Please enter your name : ");
						System.out.println("Hello, " + name + "!!");
				}
				else {
						System.out.println("Console not available.");
				}
		}
}
```

CONSOLE I/O OPERATIONS IN JAVA

3. Reading console input using Console class in java

Reading input data using the **Console** class is the most commonly used method. This class was introduced
in Java 1.6 version.

The **Console** class has defined in the **java.io** package.

Consider the following example code to understand how to read console input using Console class.

Example

```
import java.io.*;
public class ReadingDemo {
        public static void main(String[] args) {
                String name;
    Console con = System.console();
                if(con != null) {
                        name = con.readLine("Please enter your name : ");
                        System.out.println("Hello, " + name + "!!");
                }
                else {
                        System.out.println("Console not available.");
                }
        }
}
```

CONSOLE I/O OPERATIONS IN JAVA

Writing console output in java

In java, there are two methods to write console output. Using the 2 following methods, we can write
output data to the console.

- Using print() and println() methods
- Using write() method

1. Writing console output using print() and println() methods

The PrintStream is a bult-in class that provides two methods print() and println() to write console output.

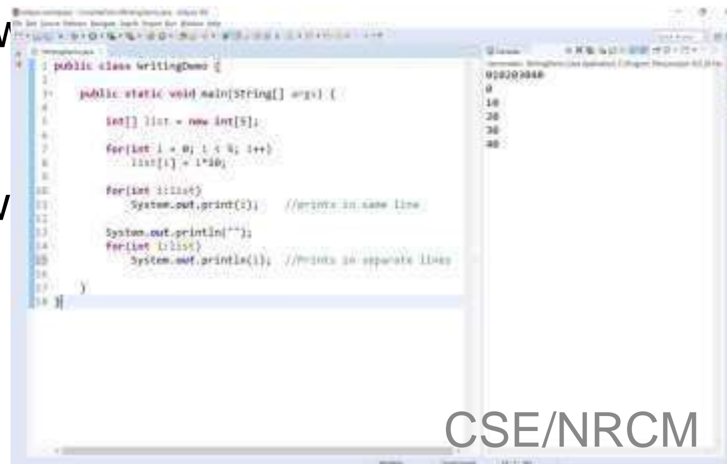
The print() and println() methods are the most widely used methods for console
output. Both print() and println() methods are used with System.out stream.

The print() method writes console output in the same line. This method can be used with console output
only.

The println() method w line (new line). This method can be used with
console ans also with

Let's look at the follow ntln() methods.

Example



CONSOLE I/O OPERATIONS IN JAVA

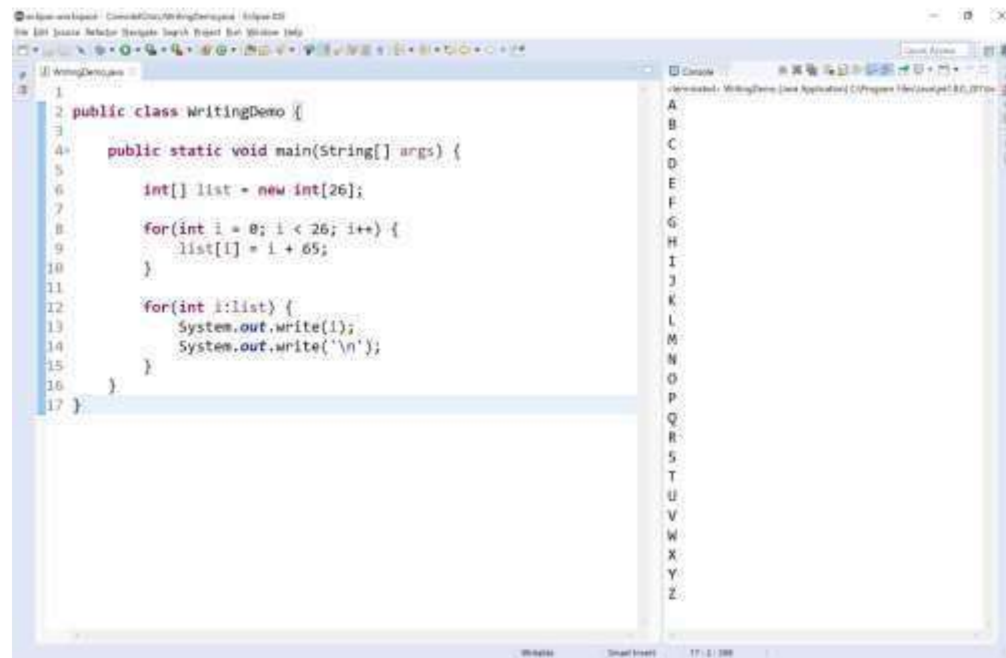
Writing console output using write() method

Alternatively, the PrintStream class provides a method write() to write console output.

The write() method take integer as argument, and writes its ASCII equalent character on to the console, it also acept escape sequences.

Let's look at the following code to illustrate write() method.

Example



FILE CLASS IN JAVA

The **File** is a built-in class in Java. In java, the File class has been defined in the **java.io** package. The File class represents a reference to a file or directory. The File class has various methods to perform operations like creating a file or directory, reading from a file, updating file content, and deleting a file or directory.

The File class in java has the following constructors.

S.No.	Constructor with Description
1	**File(String pathname)** It creates a new File instance by converting the givenpathname string into an abstract pathname. If the given string isthe empty string, then the result is the empty abstract pathname.
2	**File(String parent, String child)** It Creates a new File instance from a parent abstractpathname and a child pathname string. If parent is null then the new File instance is created as if by invoking thesingle-argument File constructor on the given child pathname string.
3	**File(File parent, String child)** It creates a new File instance from a parent abstractpathname and a child pathname string. If parent is null then the new File instance is created as if by invoking thesingle-argument File constructor on the given child pathname string.
4	**File(URI uri)** It creates a new File instance by converting the given file: URI into an abstract pathname.

FILE CLASS IN JAVA

The File class in java has the following methods.

S.No.	Methods with Description
1	**String getName()** It returns the name of the file or directory that referenced by the current File object.
2	**String getParent()** It returns the pathname of the pathname's parent, or null if the pathname does not name a parent directory.
3	**String getPath()** It returns the path of curent File.
4	**File getParentFile()** It returns the path of the current file's parent; or null if it does not exist.
5	**String getAbsolutePath()** It returns the current file or directory path from the root.
6	**boolean isAbsolute()** It returns true if the current file is absolute, false otherwise.
7	**boolean isDirectory()** It returns true, if the current file is a directory; otherwise returns false.
8	**boolean isFile()** It returns true, if the current file is a file; otherwise returns false.
9	**boolean exists()** It returns true if the current file or directory exist; otherwise returns false.
10	**boolean canRead()** It returns true if and only if the file specified exists and can be read by the application; false otherwise.
11	**boolean canWrite()** It returns true if and only if the file specified exists and the application is allowed to write to the file; false otherwise.
12	**long length()** It returns the length of the current file.
13	**long lastModified()** It returns the time that specifies the file was last modified.
14	**boolean createNewFile()** It returns true if the named file does not exist and was successfully created; false if the named file already exists.
15	**boolean delete()** It deletes the file or directory. And returns true if and only if the file or directory is successfully deleted; false otherwise.

FILE READING & WRITING IN JAVA

In java, there multiple ways to read data from a file and to write data to a file. The most commonly used
ways are as follows.
Using Byte Stream (FileInputStream and FileOutputStream) Using Character Stream (FileReader and FileWriter)
Let's look each of these ways.

File Handling using Byte Stream
In java, we can use a byte stream to handle files. The byte stream has the following built-in classes to perform various operations on a file.
FileInputStream - It is a built-in class in java that allows reading data from a file. This class has implemented based on the byte stream. The FileInputStream class provides a method **read()** to read data from a file byte by byte.
FileOutputStream - It is a built-in class in java that allows writing data to a file. This class has implemented based on the byte stream. The FileOutputStream class provides a method **write()** to write data to a file byte by byte.

FILE READING & WRITING IN JAVA

File Handling using Character Stream

In java, we can use a character stream to handle files. The character stream has the following built-in
classes to perform various operations on a file.

FileReader - It is a built-in class in java that allows reading data from a file. This class has implemented based on the character stream. The FileReader class provides a method **read()** to read data from a file character by character.

FileWriter - It is a built-in class in java that allows writing data to a file. This class has implemented based on the character stream. The FileWriter class provides a method **write()** to write data to a file character by character.

Let's look at the following example program that reads data from a file and writes the same to another file
using FIleReader and FileWriter classes.

RANDOMACCESSFILE IN JAVA

In java, the **java.io** package has a built-in class **RandomAccessFile** that enables a file to be accessed randomly. The RandomAccessFile class has several methods used to move the cursor position in a file. A random access file behaves like a large array of bytes stored in a file.

RandomAccessFile Constructors

The RandomAccessFile class in java has the following constructors.

S.No.	Constructor with Description
1	**RandomAccessFile(File fileName, String mode)** It creates a random access file stream to read from, and optionally to write to, the file specified by the File argument.
2	**RandomAccessFile(String fileName, String mode)** It creates a random access file stream to read from, and optionally to write to, a file with the specified fileName.

Access Modes

Using the RandomAccessFile, a file may created in th following modes.

r - Creates the file with read mode; Calling write methods will result in an IOException.

rw - Creates the file with read and write mode.

rwd - Creates the file with read and write mode - synchronously. All updates to file content is written to
the disk synchronously.

rws - Creates the file with read and write mode - synchronously. All updates to file content or meta data is
written to the disk synchronously

RandomAccessFile methods

S.No.	Methods with Description
1	**int read()** It reads byte of data from a file. The byte is returned as an integer in the range 0-255.
2	**int read(byte[] b)** It reads byte of data from file upto b.length, -1 if end of file is reached.
3	**int read(byte[] b, int offset, int len)** It reads bytes initialising from offset position upto b.length from the buffer.
4	**boolean readBoolean()** It reads a boolean value from from the file.
5	**byte readByte()** It reads signed eight-bit value from file.
6	**char readChar()** It reads a character value from file.
7	**double readDouble()** It reads a double value from file.
8	**float readFloat()** It reads a float value from file.
9	**long readLong()** It reads a long value from file.
10	**int readInt()** It reads a integer value from file.
11	**void readFully(byte[] b)** It reads bytes initialising from offset position upto b.length from the buffer.
12	**void readFully(byte[] b, int offset, int len)** It reads bytes initialising from offset position upto b.length from the buffer.
13	**String readUTF()** t reads in a string from the file.
14	**void seek(long pos)** It sets the file-pointer(cursor) measured from the beginning of the file, at which the next read or write occurs.
15	**long length()** It returns the length of the file.
16	**void write(int b)** It writes the specified byte to the file from the current cursor position.
17	**void writeFloat(float v)** It converts the float argument to an int using the floatToIntBits method in class Float, and then writes that int value to the file as a four-byte quantity, high byte first.
18	**void writeDouble(double v)** It converts the double argument to a long using the doubleToLongBits method in class Double, and then writes that long value to the file as an eight-byte quantity, high byte first.

CONSOLE CLASS IN JAVA

In java, the **java.io** package has a built-in class **Console** used to read from and write to the console, if one
exists. This class was added to the Java SE 6. The Console class implements teh **Flushable** interface. In java, most the input functionalities of Console class available through **System.in**, and the output functionalities available through **System.out**.

Console class Constructors

The Console class does not have any constructor. We can obtain the Console class object by calling **System.console()**.

Console class methods

The Console class in java has the following methods

S.No.	Methods with Description
1	**void flush( )** It causes buffered output to be written physically to the console.
2	**String readLine( )** It reads a string value from the keyboard, the input is terminated on pressing enter key.
3	**String readLine(String promptingString, Object...args)** It displays the given promptingString, and reads a string fron the keyboard; input is terminated on pressng Enter key.
4	**char[ ] readPassword( )** It reads a string value from the keyboard, the string is not displayed; the input is terminated on pressing enter key.
5	**char[ ] readPassword(String promptingString, Object...args)** It displays the given promptingString, and reads a string value from the keyboard, the string is not displayed; the input is terminated on pressing enter key.
6	**Console printf(String str, Object....args)** It writes the given string to the console.
7	**Console format(String str, Object....args)** It writes the given string to the console.
8	**Reader reader( )** It returns a reference to a Reader connected to the console.
9	**PrintWriter writer( )** It returns a reference to a Writer connected to the console.

SERIALIZATION AND DESERIALIZATION IN JAVA

In java, the **Serialization** is the process of converting an object into a byte stream so that it can be stored on to a file, or memory, or a database for future access. The deserialization is reverse of serialization. The deserialization is the process of reconstructing the object from the serialized state.

Using serialization and deserialization, we can transfer the Object Code from one Java Virtual machine to another.

Serialization in Java

In a java programming language, the Serialization is achieved with the help of interface **Serializable**. The
class whose object needs to be serialized must implement the Serializable interface.

We use the **ObjectOutputStream** class to write a serialized object to write to a destination. The ObjectOutputStream class provides a method **writeObject()** to serializing an object.

We use the following steps to serialize an object.

Step 1 - Define the class whose object needs to be serialized; it must implement Serializable interface.

Step 2 - Create a file reference with file path using FileOutputStream class.

Step 3 - Create reference to ObjectOutputStream object with file reference.

Step 4 - Use writeObject(object) method by passing the object that wants to be serialized.

Step 5 - Close the FileOutputStream and ObjectOutputStream.

SERIALIZATION AND DESERIALIZATION IN JAVA

Deserialization in Java

In a java programming language, the Deserialization is achieved with the help of class **ObjectInputStream**.

This class provides a method **readObject()** to deserializing an object. We use the following steps to serialize an object.

Step 1 - Create a file reference with file path in which serialized object is available using FileInputStream class.

Step 2 - Create reference to ObjectInputStream object with file reference.

Step 3 - Use readObject() method to access serialized object, and typecaste it to destination type.

Step 4 - Close the FileInputStream and ObjectInputStream.

ENUM IN JAVA

In java, an **Enumeration** is a list of named constants. The enum concept was introduced in Java SE 5 version. The enum in Java programming the concept of enumeration is greatly expanded with lot more new features compared to the other languages like C, and C++.
In java, the enumeration concept was defined based on the class concept. When we create an enum in java, it converts into a class type. This concept enables the java enum to have constructors, methods, and instance variables.
All the constants of an enum are ***public***, ***static***, and ***final***. As they are static, we can access directly using
enum name.
The main objective of enum is to define our own data types in Java, and they are said to be enumeration
data types.

Creating enum in Java
To create enum in Java, we use the keyword ***enum***. The syntax for creating enum is similar to that of class. In java, an enum can be defined outside a class, inside a class, but not inside a method.

- Every enum is converted to a class that extends the built-in class **Enum**.
- Every constant of an enum is defined as an object.
- As an enum represents a class, it can have methods, constructors. It also gets a few extra methods
  from the Enum class, and one of them is the **values()** method.

AUTOBOXING AND UNBOXING IN JAVA

In java, all the primitive data types have defined using the class concept, these classes known as **wrapper**
classes. In java, every primitive type has its corresponding wrapper class.
All the wrapper classes in Java were defined in the **java.lang** package.
The following table shows the primitive type and its corresponding wrapper class.

S.No.	Primitive Type	Wrapper class
1	byte	Byte
2	short	Short
3	int	Interger
4	long	Long
5	float	Float
6	double	Double
7	char	Character
8	boolean	Boolean

The Java 1.5 version introduced a concept that converts primitive type to corresponding wrapper type and
reverses of it.

AUTOBOXING AND UNBOXING IN JAVA

Autoboxing in Java

In java, the process of converting a primitive type value into its corresponding wrapper class object is
called autoboxing or simply boxing. For example, converting an int value to an Integer class object. The compiler automatically performs the autoboxing when a primitive type value has assigned to an object of the corresponding wrapper class.

🔔 We can also perform autoboxing manually using the method **valueOf( )**, which is provided by every
wrapper class.

Auto un-boxing in Java

In java, the process of converting an object of a wrapper class type to a primitive type value is called auto
un-boxing or simply unboxing. For example, converting an Integer object to an int value.
The compiler automatically performs the auto un-boxing when a wrapper class object has assigned to
aprimitive type.

🔔 We can also perform auto un-boxing manually using the method **intValue( )**, which is provided by
Integer wrapper class. Similarly every wrapper class has a method for auto un-boxing.

GENERICS IN JAVA

The java generics is a language feature that allows creating methods and class which can handle any type of data values. The generic programming is a way to write generalized programs, java supports it by java generics.

The java generics is similar to the templates in the C++ programming language. Most of the collection framework classes are generic classes.

The java generics allows only non-primitive type, it does not support primitive types like int, float, char,
etc.

The java generics feature was introduced in Java 1.5 version. In java, generics used angular brackets “< >”. In java, the generics feature implemented using the following.

- **Generic Method**
- **Generic Classe**

Generic methods in Java

The java generics allows creating generic methods which can work with a different type of data values. Using a generic method, we can create a single method that can be called with arguments of different types. Based on the types of the arguments passed to the generic method, the compiler handles each method call appropriately.

Generic Class in Java

In java, a class can be defined as a generic class that allows creating a class that can work with different
types.

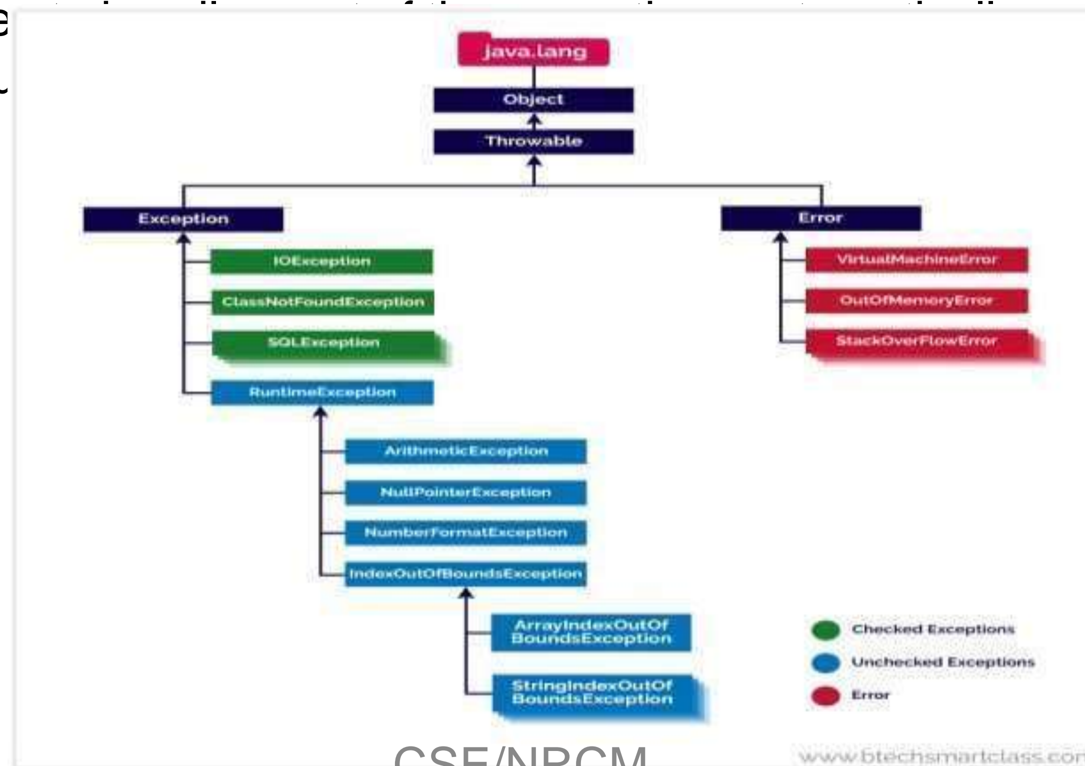
A generic class declaration looks like a non-generic class declaration, except that the class name is

UNIT – III

EXCEPTION HANDLING & MULTITHREADIN G

EXCEPTION HANDLING IN JAVA

An exception in java programming is an abnormal situation that is araised during the program execution.
In simple words, an exception is a problem that arises at the time of program execution.
When an exception occurs, it disrupts the program execution flow. When an exception occurs, the program execution gets terminated, and the system generates an error. We use the exception handling mechanism to avoid abnormal termination of program execution.
Java programming language has a very powerful and efficient exception handling mechanism with a large
number of built-in classe
Java programming langu exception
handling mechanism.



EXCEPTION HANDLING IN JAVA

Reasons for Exception Occurrence

Several reasons lead to the occurrence of an exception. A few of them are as follows.
When we try to open a file that does not exist may lead to an
exception. When the user enters invalid input data, it may lead to an
exception.
When a network connection has lost during the program execution may lead to an
exception. When we try to access the memory beyond the allocated range may lead to an
exception.
The physical device problems may also lead to an exception**.**

Types of Exception

In java, exceptions have categorized into two types, and they are as follows.
Checked Exception - An exception that is checked by the compiler at the time of compilation is
called a
checked exception.
Unchecked Exception - An exception that can not be caught by the compiler but occurrs at the
time of
program execution is called an unchecked exception.

How exceptions handled in Java?

In java, the exception handling mechanism uses five keywords namely *try*, *catch*, *finally*, *throw*,
and *throws*.
We will learn all these concepts in this series of tutorials.

EXCEPTION TYPES IN JAVA

In java, exceptions are mainly categorized into two types, and they are as follows.
Checked Exceptions
Unchecked Exceptions

Checked Exceptions
The checked exception is an exception that is checked by the compiler during the compilation process to confirm whether the exception is handled by the programmer or not. If it is not handled, the compiler displays a compilation error using built-in classes.
The checked exceptions are generally caused by faults outside of the code itself like missing resources,
networking errors, and problems with threads come to mind.
The following are a few built-in classes used to handle checked exceptions in java.

- IOException
- FileNotFoundException
- ClassNotFoundException
- SQLException
- DataAccessException
- InstantiationException
- UnknownHostException

In the exception class hierarchy, the checked exception classes are the direct children of the Exception
class.
The checked exception is also known as a compile-time exception

Unchecked Exceptions

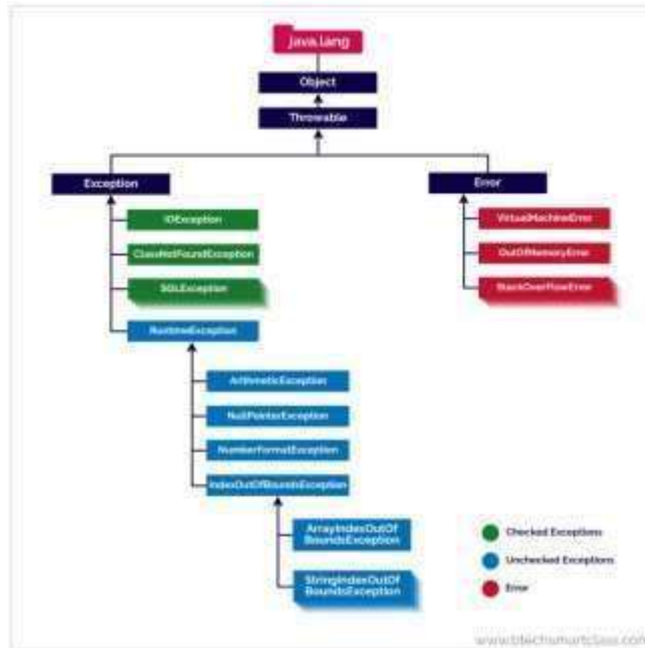
The unchecked exception is an exception that occurs at the time of program execution. The unchecked
exceptions are not caught by the compiler at the time of compilation.
The unchecked exceptions are generally caused due to bugs such as logic errors, improper use of resources, etc.
The following are a few built-in classes used to handle unchecked exceptions in java.

- ArithmeticException
- NullPointerException
- NumberFormatException
- ArrayIndexOutOfBoundsException
- StringIndexOutOfBoundsException

In the exception class hierarchy, the unchecked exception classes are the children of RuntimeException
class, which is a child class of Exception class.
The unchecked except ception.
Let's look at the followi ecked exceptions.

Exception class hierarchy

In java, the built-in classes used to handle exceptions have the following class hierarchy.



EXCEPTION MODELS IN JAVA

In java, there are two exception models. Java programming language has two models of exception handling. The exception models that java suports are as follows.

Termination Model

Resumptive Model

Let's look into details of each exception model.

Termination Model

In the termination model, when a method encounters an exception, further processing in that method is terminated and control is transferred to the nearest catch block that can handle the type of exception encountered.

In other words we can say that in termination model the error is so critical there is no way to get back to
where the exception occurred.

Resumptive Model

The alternative of termination model is resumptive model. In resumptive model, the exception handler is expected to do something to stable the situation, and then the faulting method is retried. In resumptive model we hope to continue the execution after the exception is handled.

In resumptive model we may use a method call that want resumption like behavior. We may also place the
try block in a while loop that keeps re-entering the try block util the result is satisfactory.

UNCAUGHT EXCEPTIONS IN JAVA

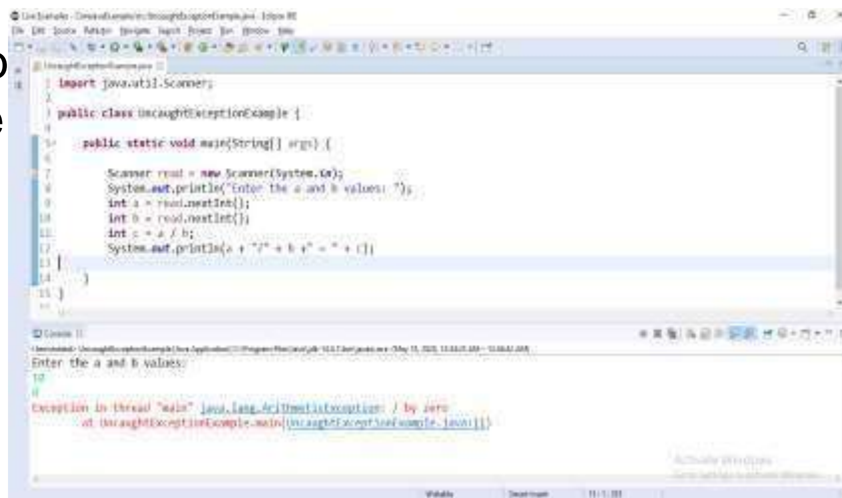
In java, assume that, if we do not handle the exceptions in a program. In this case, when an exception occurs in a particular function, then Java prints a exception message with the help of uncaught exception handler.

The uncaught exceptions are the exceptions that are not caught by the compiler but automatically caught and handled by the Java built-in exception handler.

Java programming language has a very strong exception handling mechanism. It allow us to handle the exception use the keywords like try, catch, finally, throw, and throws.

When an uncaught exception occurs, the JVM calls a special private method known **dispatchUncaughtException( )**, on the Thread class in which the exception occurs and terminates the thread.

The Division b ... or uncaught exceptions. Look at the following code



try AND catch IN JAVA

In java, the **try** and **catch**, both are the keywords used for exception handling.
The keyword try is used to define a block of code that will be tests the occurence of an exception. The keyword catch is used to define a block of code that handles the exception occured in the respective try block.
The uncaught exceptions are the exceptions that are not caught by the compiler but automatically caught and handled by the Java built-in exception handler.
Both try and catch are used as a pair. Every try block must have one or more catch blocks. We can not use
try without atleast one catch, and catch alone can be used (catch without try is not allowed). The following is the syntax of try and catch blocks.

Syntax

```
try{
   ...
   code to be tested
   ...
}
catch(ExceptionType object){
   ...
   code for handling the exception
   ...
}
```

try AND catch IN JAVA

Multiple catch clauses

In java programming language, a try block may has one or more number of catch blocks. That means a
single try statement can have multiple catch clauses.

When a try block has more than one catch block, each catch block must contain a different exception type to be handled.

The multiple catch clauses are defined when the try block contains the code that may lead to different
type of exceptions.

The try block generates only one exception at a time, and at a time only one catch block is executed. When there are multiple catch blocks, the order of catch blocks must be from the most specific exception handler to most general.

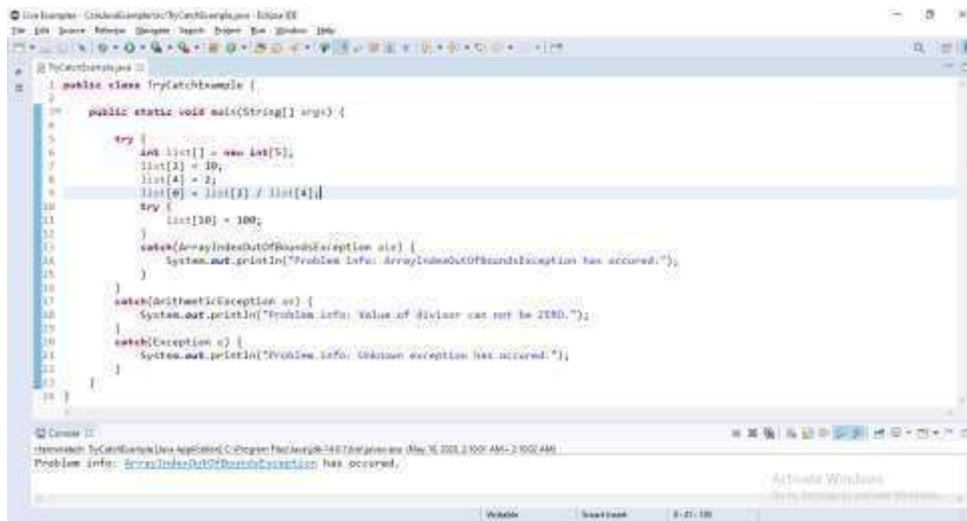
The ca e defined at the last.

try AND catch IN JAVA

Nested try statements

The java allows to write a try statement inside another try statement. A try block within another try block
is known as nested try block.

When there are nested try blocks, each try block must have one or more seperate catch blocks.



In case of nested try blocks, if an exception occured in the inner try block and it's catch blocks are unable
to handle it then it transfers the control to the outer try's catch block to handle it.

throw, throws, AND finally KEYWORDS IN JAVA

In java, the keywords throw, throws, and finally are used in the exception handling concept. Let's look at
each of these keywords.

throw keyword in Java
The throw keyword is used to throw an exception instance explicitly from a try block to corresponding catch block. That means it is used to transfer the control from try block to corresponding catch block.
The throw keyword must be used inside the try blcok. When JVM encounters the throw keyword, it stops
the execution of try block and jump to the corresponding catch block.
Using throw keyword only object of Throwable class or its sub classes can be thrown.
Using throw keyword only one exception can be thrown.
The throw keyword must followed by an throwable instance.

The following is the general syntax for using throw keyword in a try block.
Syntax
throw instance;

Here the instace must be throwable instance and it can be created dynamically using new operator.

throw, throws, AND finally KEYWORDS IN JAVA

throws keyword in Java

The throws keyword specifies the exceptions that a method can throw to the default handler and does not handle itself. That means when we need a method to throw an exception automatically, we use throws keyword followed by method declaration

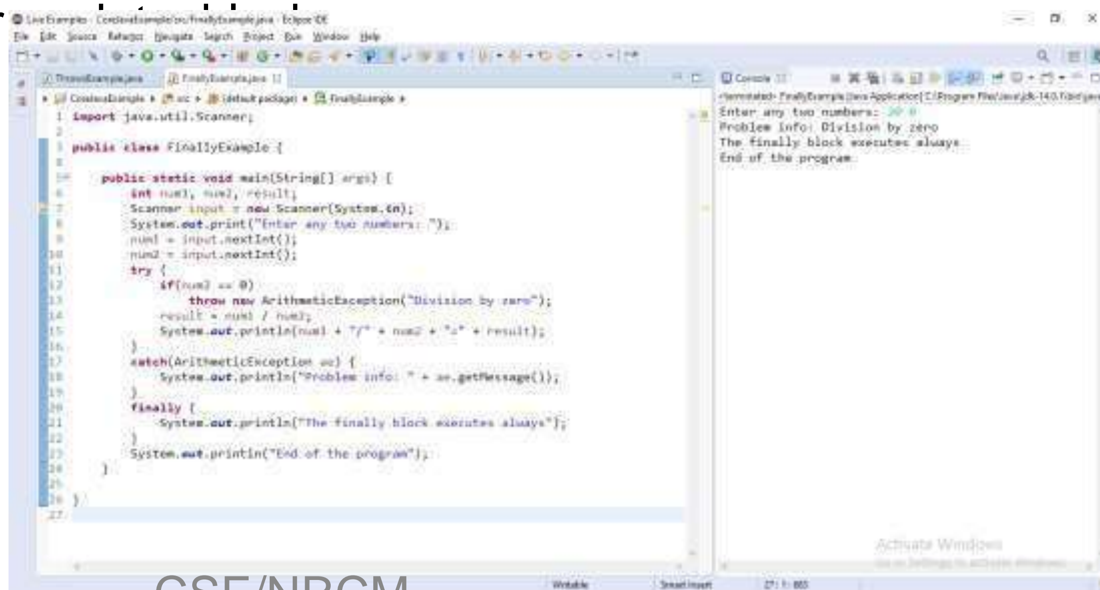
When a method throws an exception, we must put the calling statement of method in try-catch block.

finally keyword in Java

The finally keyword used to define a block that must be executed irrespective of exception occurence. The basic purpose of finally keyword is to cleanup resources allocated by try block, such as closing file, closing database connection, etc.

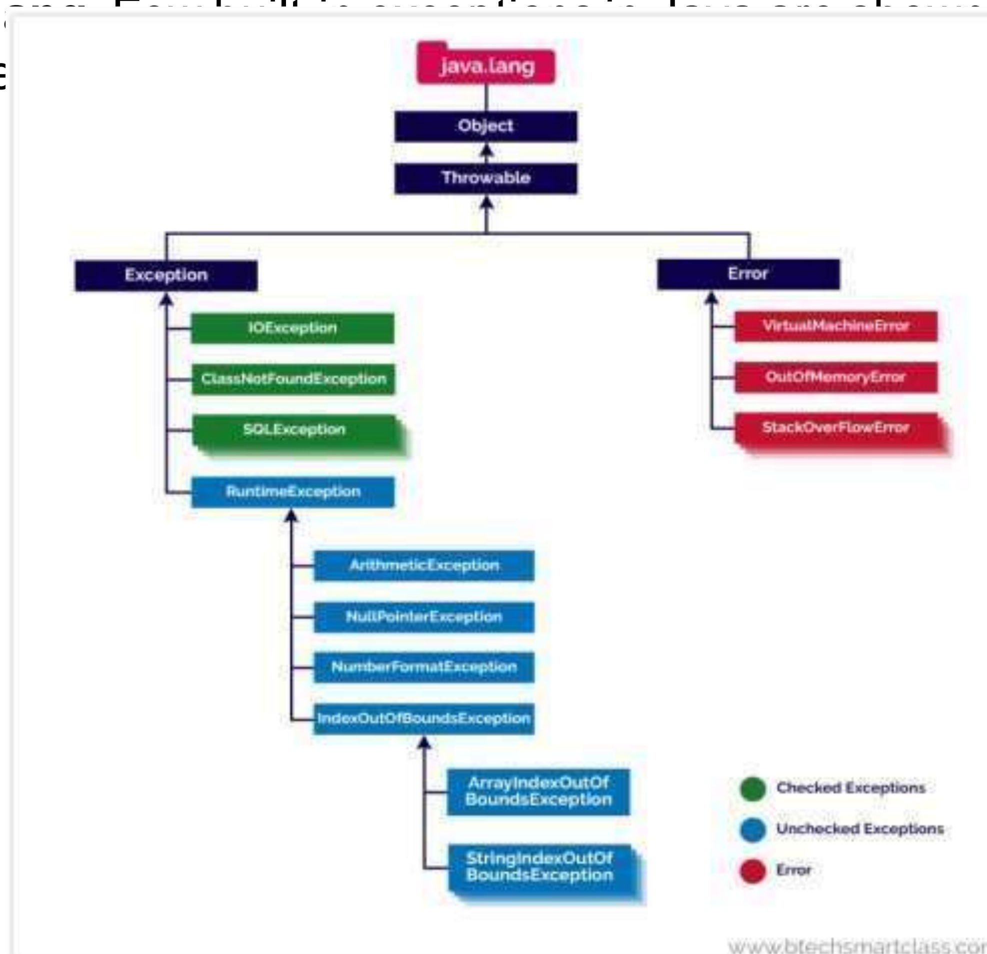
Only one finally block is allowed for

Use of finally block is optional.



BUILT-IN EXCEPTIONS IN JAVA

The Java programming language has several built-in exception class that support exception handling. Every
exception class is suitable to explain certain error situations at run
time. All the built-in exception classes in Java were defined a
package **java.l** n the
following image



List of checked exceptions in Java

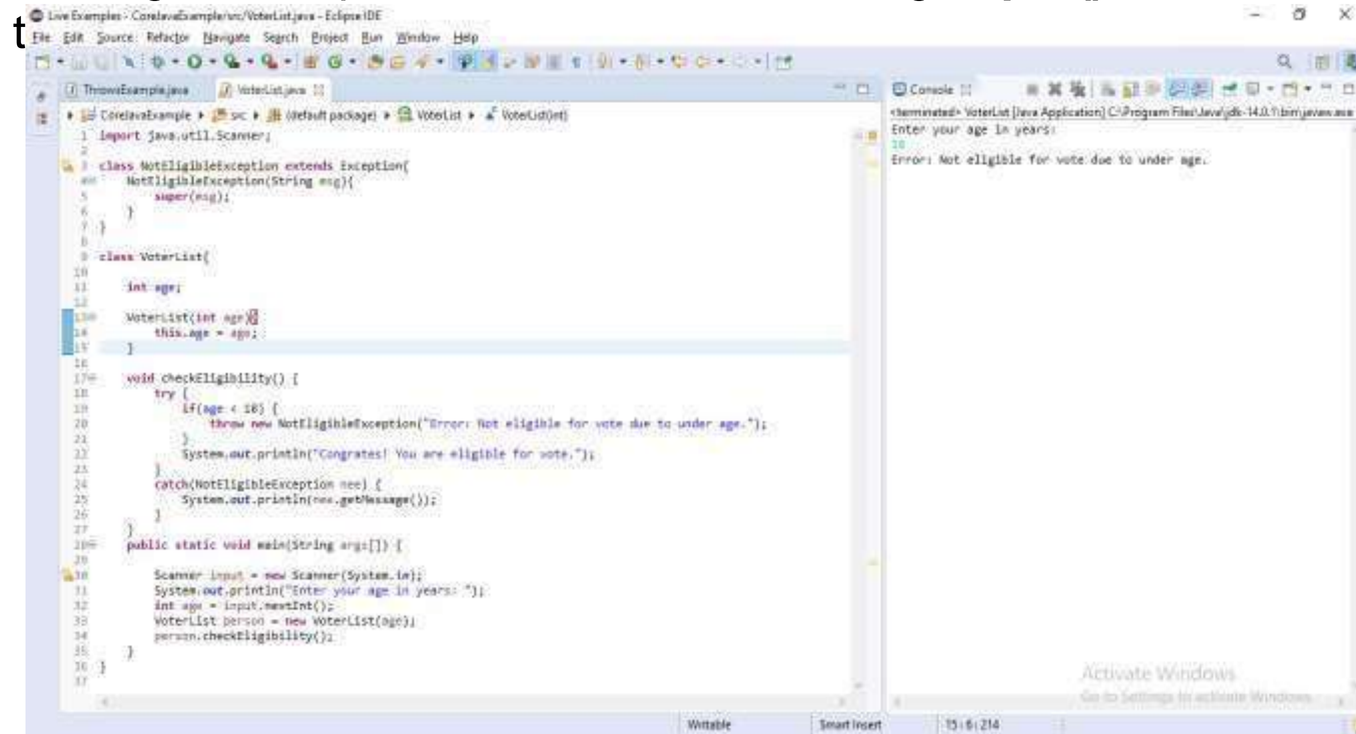
S. No.	Exception Class with Description
1	**ClassNotFoundException** It is thrown when the Java Virtual Machine (JVM) tries to load a particular class and the specified class cannot be found in the classpath.
2	**CloneNotSupportedException** Used to indicate that the clone method in class Object has been called to clone an object, but that the object's class does not implement the Cloneable interface.
3	**IllegalAccessException** It is thrown when one attempts to access a method or member that visibility qualifiers do not allow.
4	**InstantiationException** It is thrown when an application tries to create an instance of a class using the newInstance method in class Class , but the specified class object cannot be instantiated because it is an interface or is an abstract class.
5	**InterruptedException** It is thrown when a thread that is sleeping, waiting, or is occupied is interrupted.
6	**NoSuchFieldException** It indicates that the class doesn't have a field of a specified name.
7	**NoSuchMethodException** It is thrown when some JAR file has a different version at runtime that it had at compile time, a NoSuchMethodException occurs during reflection when we try to access a method that does not exist.

List of unchecked exceptions in Java

S. No.	Exception Class with Description
1	**ArithmeticException** It handles the arithmetic exceptions like dividion by zero
2	**ArrayIndexOutOfBoundsException** It handles the situations like an array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the array.
3	**ArrayStoreException** It handles the situations like when an attempt has been made to store the wrong type of object into an array of objects
4	**AssertionError** It is used to indicate that an assertion has failed
5	**ClassCastException** It handles the situation when we try to improperly cast a class from one type to another.
6	**IllegalArgumentException** This exception is thrown in order to indicate that a method has been passed an illegal or inappropriate argument.
7	**IllegalMonitorStateException** This indicates that the calling thread has attempted to wait on an object's monitor, or has attempted to notify other threads that wait on an object's monitor, without owning the specified monitor.
8	**IllegalStateException** It signals that a method has been invoked at an illegal or inappropriate time.
9	**IllegalThreadStateException** It is thrown by the Java runtime environment, when the programmer is trying to modify the state of the thread when it is illegal.
10	**IndexOutOfBoundsException** It is thrown when attempting to access an invalid index within a collection, such as an array, vector , string , and so forth.
11	**NegativeArraySizeException** It is thrown if an applet tries to create an array with negative size.
12	**NullPointerException** it is thrown when program attempts to use an object reference that has the null value.
13	**NumberFormatException** It is thrown when we try to convert a string into a numeric value such as float or integer, but the format of the input string is not appropriate or illegal.
14	**SecurityException** It is thrown by the Java Card Virtual Machine to indicate a security violation.
15	**StringIndexOutOfBounds** It is thrown by the methods of the String class, in order to indicate that an index is either negative, or greater than the size of the string itself.
16	**UnsupportedOperationException** It is thrown to indicate that the requested operation is not supported.

CREATING OWN EXCEPTIONS IN JAVA

The Java programming language allows us to create our own exception classes which are basically
subclasses built-in class **Exception**.
To create our own exception class simply creates a class as a subclass of built-in Exception class. We may create constructor in the user-defined exception class and pass a string to Exception class constructor using **super()**. We can use **getMessage()** method t



MULTITHREADING IN JAVA

The java programming language allows us to create a program that contains one or more parts that can run simultaneously at the same time. This type of program is known as a multithreading program. Each part of this program is called a thread. Every thread defines a separate path of execution in java. A thread is explained in different ways, and a few of them are as specified below.

A thread is a light wieght process.

A thread may also be defined as follows.

A thread is a subpart of a process that can run individually.

In java, multiple threads can run at a time, which enables the java to write multitasking programs. The multithreading is a specialized form of multitasking. All modern operating systems support multitasking. There are two types of multitasking, and they are as follows.

Process-based multitasking

Thread-based multitasking

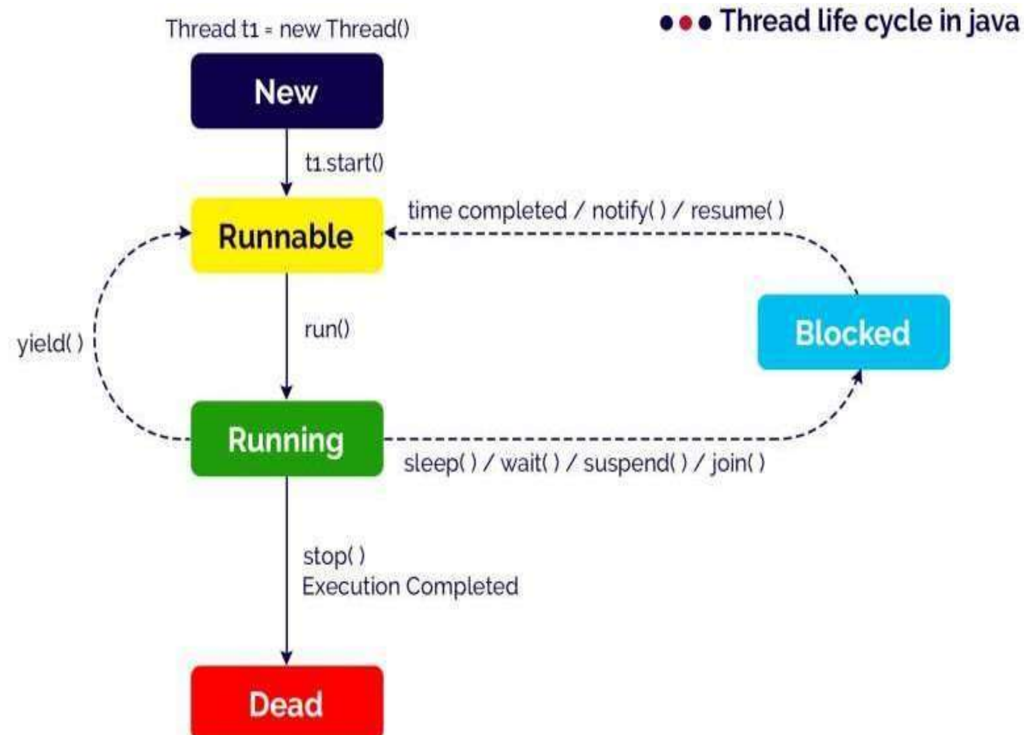
It is important to know the difference between process-based and thread-based multitasking. Let's distinguish both.

Process-based multitasking Vs Thread-based multitasking

Process-based multitasking	Thread-based multitasking
It allows the computer to run two or more programs concurrently	It allows the computer to run two or more threads concurrently
In this process is the smallest unit.	In this thread is the smallest unit.
Process is a larger unit.	Thread is a part of process.
Process is heavy weight.	Thread is light weight.
Process requires seperate address space for each.	Threads share same address space.
Process never gain access over idle time of CPU.	Thread gain access over idle time of CPU.
Inter process communication is expensive.	Inter thread communication is not expensive.

JAVA THREAD MODEL

In java, a thread goes through different states throughout its execution. These stages are called thread life cycle states or phases. A thread may in any of the states like new, ready or runnable, running, blocked or wait, and dead or terminated state. The life cycle of a thread in java is shown in the following figure.



CREATING THREADS IN JAVA

In java, a thread is a lightweight process. Every java program executes by a thread called the main thread. When a java program gets executed, the main thread created automatically. All other threads called from the main thread.

The java programming language provides two methods to create threads, and they are listed below.

- **Using Thread class (by extending Thread class)**
- **Uisng Runnable interface (by implementing Runnable interface)**

Extending Thread class

The java contains a built-in class Thread inside the java.lang package. The Thread class contains all the
methods that are related to the threads.

To create a thread using Thread class, follow the step given below.

Step-1: Create a class as a child of Thread class. That means, create a class that extends Thread class. **Step-2**: Override the run( ) method with the code that is to be executed by the thread. The run( ) method must be public while overriding.

Step-3: Create the object of the newly created class in the main( ) method.

Step-4: Call the start( ) method on the object created in the above step.

CREATING THREADS IN JAVA

Implementing Runnable interface

The java contains a built-in interface Runnable inside the java.lang package. The Runnable interface
implemented by the Thread class that contains all the methods that are related to the threads. To create a thread using Runnable interface, follow the step given below.

Step-1: Create a class that implements Runnable interface.

Step-2: Override the run( ) method with the code that is to be executed by the thread. The run( ) method
must be public while overriding.

Step-3: Create the object of the newly created class in the main( ) method.

Step-4: Create the Thread class object by passing above created object as parameter to the Thread class
constructor.

Step-5: Call the start( ) method on the Thread class object created in the above step.

More about Thread class

The Thread class in java is a subclass of Object class and it implements Runnable interface. The Thread
class is available inside the java.lang package. The Thread class has the following syntax.

```
class Thread extends Object implements Runnable{
...
}
```

The Thread class has the following consructors.

- **Thread( )**
- **Thread(String threadName)**
- **Thread(Runnable objectName)**
- **Thread(Runnable objectName, String threadName)**

Thread class methods.

Method	Description	Return Value
run( )	Defines actual task of the thread.	void
start( )	It moves thre thread from Ready state to Running state by calling run( ) method.	void
setName(String)	Assigns a name to the thread.	void
getName( )	Returns the name of the thread.	String
setPriority(int)	Assigns priority to the thread.	void
getPriority( )	Returns the priority of the thread.	int
getId( )	Returns the ID of the thread.	long
activeCount()	Returns total number of thread under active.	int
currentThread( )	Returns the reference of the thread that currently in running state.	void
sleep(long)	moves the thread to blocked state till the specified number of milliseconds.	void
isAlive( )	Tests if the thread is alive.	boolean
yield( )	Tells to the scheduler that the current thread is willing to yield its current use of a processor.	void
join( )	Waits for the thread to end.	void

JAVA THREAD PRIORITY

In a java programming language, every thread has a property called priority. Most of the scheduling algorithms use the thread priority to schedule the execution sequence. In java, the thread priority range from 1 to 10. Priority 1 is considered as the lowest priority, and priority 10 is considered as the highest priority. The thread with more priority allocates the processor first.
The java programming language Thread class provides two methods **setPriority(int)**, and **getPriority( )** to handle thread priorities.
The Thread class also contains three constants that are used to set the thread priority, and they are listed
below.
MAX_PRIORITY - It has the value 10 and indicates highest priority. ***NORM_PRIORITY*** - It has the value 5 and indicates normal priority. ***MIN_PRIORITY*** - It has the value 1 and indicates lowest priority.

🔔 The default priority of any thread is 5 (i.e. NORM_PRIORITY).

setPriority( ) method
The setPriority( ) method of Thread class used to set the priority of a thread. It takes an integer range from 1 to 10 as an argument and returns nothing (void).
The regular use of the setPriority( ) method is as follows.

JAVA THREAD SYNCHRONISATION

The java programming language supports multithreading. The problem of shared resources occurs when two or more threads get execute at the same time. In such a situation, we need some way to ensure that the shared resource will be accessed by only one thread at a time, and this is performed by using the concept called synchronization.

🔔 The synchronization is the process of allowing only one thread to access a shared resource at a time.

In java, the synchronization is achieved using the following concepts.

- ❑ Mutual Exclusion
- ❑ Inter thread communication

Mutual Exclusion

Using the mutual exclusion process, we keep threads from interfering with one another while they accessing the shared resource. In java, mutual exclusion is achieved using the following concepts. Synchronized method
Synchronized block

Synchronized method

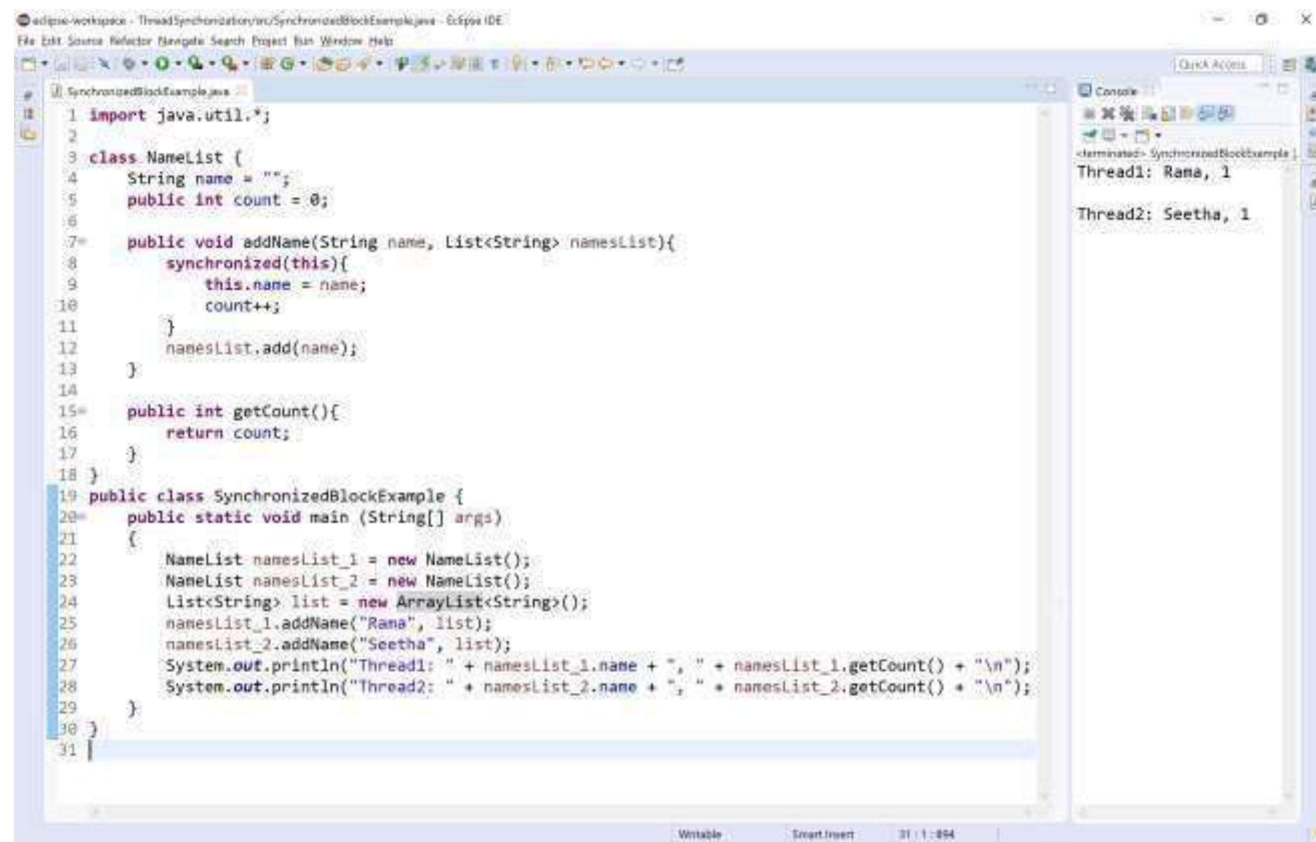
When a method created using a synchronized keyword, it allows only one object to access it at a time. When an object calls a synchronized method, it put a lock on that method so that other objects or thread that are trying to call the same method must wait, until the lock is released. Once the lock is released on the shared resource, one of the threads among the waiting threads will be allocated to the shared resource.

Synchronized block

The synchronized block is used when we want to synchronize only a specific sequence of lines in a method. For example, let's consider a method with 20 lines of code where we want to synchronize only a sequence of 5 lines code, we use the synchronized block.
The folllowing syntax is used to define a synchronized block.

Syntax

```
synchronized(object){
   ...
   block code
   ...
}
```



JAVA INTER THREAD COMMUNICATION

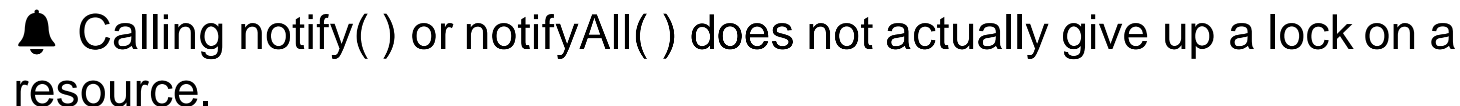
Inter thread communication is the concept where two or more threads communicate to solve the problem of **polling**. In java, polling is the situation to check some condition repeatedly, to take appropriate action, once the condition is true. That means, in inter-thread communication, a thread waits until a condition becomes true such that other threads can execute its task. The inter-thread communication allows the synchronized threads to communicate with each other.

Java provides the following methods to achieve inter thread communication.

- wait( )
- notify( )
- notifyAll( )

The following table gives detailed description about the above methods.

Method	Description
void wait( )	It makes the current thread to pause its execution until other thread in the same monitor calls notify( )
void notify( )	It wakes up the thread that called wait( ) on the same object.
void notifyAll()	It wakes up all the threads that called wait( ) on the same object.



Calling notify( ) or notifyAll( ) does not actually give up a lock on a resource.

UNIT – IV

EVENT HANDLING

EVENT HANDLING

Changing the state of an object is known as an event. For example, click on button, dragging mouse etc. The
java.awt.event package provides many event classes and Listener interfaces for event handling.

Java Event classes and Listener interfaces

Event Classes	Listener Interfaces
ActionEvent	ActionListener
MouseEvent	MouseListener and MouseMotionListener
MouseWheelEven t	MouseWheelListener
KeyEvent	KeyListener
ItemEvent	ItemListener
TextEvent	TextListener
AdjustmentEvent	AdjustmentListener
WindowEvent	WindowListener
ComponentEvent	ComponentListener
ContainerEvent	ContainerListener
FocusEvent	FocusListener

EVENT HANDLING

To perform Event Handling, we need to register the source with the listener. For registering the component
with the Listener, many classes provide the registration methods. For example:

Button

public void addActionListener(ActionListener a){}

MenuItem

public void addActionListener(ActionListener a){}

TextField

public void addActionListener(ActionListener a){} public void addTextListener(TextListener a){}

TextArea

public void addTextListener(TextListener a){}

Checkbox

public void addItemListener(ItemListener a){}

Choice

public void addItemListener(ItemListener a){}

List

public void addActionListener(ActionListener a){}
public void addItemListener(ItemListener a){}

Delegation Event Model in Java

The Delegation Event model is defined to handle events in GUI programming languages. The GUI stands for
Graphical User Interface, where a user graphically/visually interacts with the system.

The GUI programming is inherently event-driven; whenever a user initiates an activity such as a mouse activity, clicks, scrolling, etc., each is known as an event that is mapped to a code to respond to functionality to the user. This is known as event handling.

Event Processing in Java
Java support event processing since Java 1.0. It provides support for AWT (Abstract Window Toolkit), which is an API used to develop the Desktop application. In Java 1.0, the AWT was based on inheritance. To catch and process GUI events for a program, it should hold subclass GUI components and override action() or handleEvent() methods. The below image demonstrates the event processing

Delegation Event Model in Java

The key advantage of the Delegation Event Model is that the application logic is completely separated from
the interface logic.

In this model, the listener must be connected with a source to receive the event notifications. Thus, the events will only be received by the listeners who wish to receive them. So, this approach is more convenient than the inheritance-based event model (in Java 1.0).

In the older model, an event was propagated up the containment until a component was handled. This needed components to receive events that were not processed, and it took lots of time. The Delegation Event model overcame this issue.

Basically, an Event Model is based on the following three components:

Events
Events Sources
Events Listeners

Delegation Event Model in Java

Events
The Events are the objects that define state change in a source. An event can be generated as a reaction of a user while interacting with GUI elements. Some of the event generation activities are moving the mouse pointer, clicking on a button, pressing the keyboard key, selecting an item from the list, and so on. We can also consider many other user operations as events.

The Events may also occur that may be not related to user interaction, such as a timer expires, counter
exceeded, system failures, or a task is completed, etc. We can define events for any of the applied actions.

Event Sources
A source is an object that causes and generates an event. It generates an event when the internal state of
the object is changed. The sources are allowed to generate several different types of events.

A source must register a listener to receive notifications for a specific event. Each event contains its registration method. Below is an example:
public void addTypeListener (TypeListener e1)

Delegation Event Model in Java

Event Listeners

An event listener is an object that is invoked when an event triggers. The listeners require two things; first, it must be registered with a source; however, it can be registered with several resources to receive notification about the events. Second, it must implement the methods to receive and process the received notifications. The methods that deal with the events are defined in a set of interfaces. These interfaces can be found in the java.awt.event package.

For example, the MouseMotionListener interface provides two methods when the mouse is dragged and moved. Any object can receive and process these events if it implements the MouseMotionListener interface.

Types of Events

The events are categories into the following two categories:

The Foreground Events:

The foreground events are those events that require direct interaction of the user. These types of events are generated as a result of user interaction with the GUI component. For example, clicking on a button, mouse movement, pressing a keyboard key, selecting an option from the list, etc.

The Background Events :

The Background events are those events that result from the interaction of the end-user. For example, an

Operating system interrupts system failure (Hardware or Software).

To handle these events, we need an event handling mechanism that provides control over the events and

responses.

HANDLING MOUSE EVENTS

Java MouseListener Interface

The Java MouseListener is notified whenever you change the state of mouse. It is notified against
MouseEvent. The MouseListener interface is found in java.awt.event package. It has five methods.

Methods of MouseListener interface

The signature of 5 methods found in MouseListener interface are given below:
public abstract void mouseClicked(MouseEvent
e); public abstract void
mouseEntered(MouseEvent e); public abstract
void mouseExited(MouseEvent e); public
abstract void mousePressed(MouseEvent e);
public abstract void
mouseReleased(MouseEvent e);

HANDLING OF KEYBOARD EVENTS

Java KeyListener Interface
The Java KeyListener is notified whenever you change the state of key. It is notified against KeyEvent. The
KeyListener interface is found in java.awt.event package, and it has three methods.

Interface declaration
Following is the declaration for java.awt.event.KeyListener interface:

public interface KeyListener extends EventListener

Methods of KeyListener interface
The signature of 3 methods found in KeyListener interface are given below:

Sr. no.	Method name	Description
1.	public abstract void keyPressed (KeyEvent e);	It is invoked when a key has been pressed.
2.	public abstract void keyReleased (KeyEvent e);	It is invoked when a key has been released.
3.	public abstract void keyTyped (KeyEvent e);	It is invoked when a key has been typed.

Methods inherited
This interface inherits methods from the following interface:
java.awt.EventListener

Jbutton

Jbutton

The JButton class is used to create a labeled button that has platform independent implementation. The
application result in some action when the button is pushed. It inherits AbstractButton class.

JButton class declaration

public class JButton extends AbstractButton implements Accessible

Commonly used Constructors:

Constructor	Description
JButton()	It creates a button with no text and icon.
JButton(String s)	It creates a button with the specified text.
JButton(Icon i)	It creates a button with the specified icon object.

Commonly used Methods of AbstractButton class:

Methods	Description
void setText(String s)	It is used to set specified text on button
String getText()	It is used to return the text of the button.
void setEnabled(boolean b)	It is used to enable or disable the button.
void setIcon(Icon b)	It is used to set the specified Icon on the button.
Icon getIcon()	It is used to get the Icon of the button.
void setMnemonic(int a)	It is used to set the mnemonic on the button.
void addActionListener(ActionListener a)	It is used to add the action listener to this object.

JCheckBox

The JCheckBox class is used to create a checkbox. It is used to turn an option on (true) or off (false). Clicking
on a CheckBox changes its state from "on" to "off" or from "off" to "on ".It inherits JToggleButton class.

JCheckBox class declaration

Let's see the declaration for javax.swing.JCheckBox class.

public class JCheckBox extends JToggleButton implements Accessible

Commonly used Constructors:

Constructor	Description
JCheckBox()	Creates an initially unselected check box button with no text, no icon.
JChechBox(String s)	Creates an initially unselected check box with text.
JCheckBox(String text, boolean selected)	Creates a check box with text and specifies whether or not it is initially selected.
JCheckBox(Action a)	Creates a check box where properties are taken from the Action supplied.

Commonly used Methods:

Methods	Description
AccessibleContext getAccessibleContext()	It is used to get the AccessibleContext associated with this JCheckBox.
protected String paramString()	It returns a string representation of this JCheckBox.

JRadioButton

The JRadioButton class is used to create a radio button. It is used to choose one option from multiple
options. It is widely used in exam systems or quiz.
It should be added in ButtonGroup to select one radio button only.

JRadioButton class declaration
Let's see the declaration for javax.swing.JRadioButton class.
public class JRadioButton extends JToggleButton implements Accessible

Commonly used Constructors:

Constructor	Description
JRadioButton()	Creates an unselected radio button with no text.
JRadioButton(String s)	Creates an unselected radio button with specified text.
JRadioButton(String s, boolean selected)	Creates a radio button with the specified text and selected status.

Commonly used Methods:

Methods	Description
void setText(String s)	It is used to set specified text on button.
String getText()	It is used to return the text of the button.
void setEnabled(boolean b)	It is used to enable or disable the button.
void setIcon(Icon b)	It is used to set the specified Icon on the button.
Icon getIcon()	It is used to get the Icon of the button.
void setMnemonic(int a)	It is used to set the mnemonic on the button.
void addActionListener(ActionListener a)	It is used to add the action listener to this object.

JOptionPane (Dialogs)

The JOptionPane class is used to provide standard dialog boxes such as message dialog box, confirm dialog box and input dialog box. These dialog boxes are used to display information or get input from the user. The JOptionPane class inherits JComponent class.

JOptionPane class declaration

public class JOptionPane extends JComponent implements Accessible

Common Constructors of JOptionPane class

Constructor	Description
JOptionPane()	It is used to create a JOptionPane with a test message.
JOptionPane(Object message)	It is used to create an instance of JOptionPane to display a message.
JOptionPane(Object message, int messageType	It is used to create an instance of JOptionPane to display a message with specified message type and default options.

Common Methods of JOptionPane class

Methods	Description
JDialog createDialog(String title)	It is used to create and return a new parentless JDialog with the specified title.
static void showMessageDialog(Component parentComponent, Object message)	It is used to create an information-message dialog titled "Message".
static void showMessageDialog(Component parentComponent, Object message, String title, int messageType)	It is used to create a message dialog with given title and messageType.
static int showConfirmDialog(Component parentComponent, Object message)	It is used to create a dialog with the options Yes, No and Cancel; with the title, Select an Option.
static String showInputDialog(Component parentComponent, Object message)	It is used to show a question-message dialog requesting input from the user parented to parentComponent.
void setInputValue(Object newValue)	It is used to set the input value that was selected or input by the user.

JAVA LIST INTERFACE

The **List** interface is a child interface of the Collection interface. The List interface is available inside the **java.util** package. It defines the methods that are commonly used by classes like ArrayList, LinkedList, Vector, and Stack.

- The List interface extends **Collection** interface.
- The List interface allows duplicate elements.
- The List interface preserves the order of insertion.
- The List allows to access the elements based on the index value that starts with zero.

Methods of List interface in java

java.util.List

- void add(int *index*, E *obj*)
  - Inserts the **obj** at **index** in the invoking List. Any elements at or beyond the **index** are shifted.
- boolean addAll(int *index*, Collection *c*)
  - Adds all the elements of **c** to the invoking List from **index** position.
- <E> remove(int *index*)
  - Removes the element at **index** from the invoking List. The resulting list is compacted.
- <E> get(int *index*)
  - Returns the element at **index** from the invoking List.
- List<E> subList(int *startIndex*, int *endIndex*)
  - Returns a list containing elements from **startIndex** to **endIndex-1** from the invoking List.
- <E> set(int *index*, E *obj*)
  - Assigns **obj** to the position **index** in the invoking List.
- int indexOf(Object *obj*)
  - Returns the index value of **obj** first occurrence in the invoking List. Returns **-1** if **obj** not found.
- int lastIndexOf(Object *obj*)
  - Returns the index value of **obj** last occurrence in the invoking List. Returns **-1** if **obj** not found.
- ListIterator<E> listIterator( )
  - Returns an iterator to the start of the invoking List.
- ListIterator<E> listIterator(int *index*)
  - Returns an iterator that begins at **index** of the invoking List.

www.btechsmartclass.com

JAVA QUEUE INTERFACE

The **Queue** interface is a child interface of the Collection interface. The Queue interface is available inside the **java.util** package. It defines the methods that are commonly used by classes like PriorityQueue and ArrayDeque.

The Queue is used to organize a sequence of
elements prior to the actual operation.

- The Queue interface extends **Collection** interface.
- The Queue interface allows duplicate elements.
- The Queue interface preserves the order of insertion.

The Queue interface defines the following methods.

JAVA DEQUE INTERFACE

The **Deque** interface is a child interface of the Queue interface. The Deque interface is available inside the **java.util** package. It defines the methods that are used by class ArrayDeque.

🔔 The Deque interface extends **Queue** interface.

🔔 The Deque interface allows duplicate elements.

🔔 The Deque interface preserves the order of insertion.

The Deque interface defines the following methods.

Methods of Deque interface in java

java.util.Deque

- **void addFirst(E *obj*)** - Adds the **obj** to head of the invoking deque. Throws **IllegalStateException** if out of space
- **void addLast(E *obj*)** - Adds the **obj** to tail of the invoking deque. Throws **IllegalStateException** if out of space
- **boolean offerFirst(E *obj*)** - Attempts to add **obj** the head. Returns true if added otherwise returns false
- **boolean offerLast(E *obj*)** - Attempts to add **obj** the tail. Returns true if added otherwise returns false
- **void push(E *obj*)** - Adds the **obj** to head of the invoking deque. Throws **IllegalStateException** if out of space
- **<E> getFirst( )** - Returns the first element from the invoking dequeu. Throws **NoSuchElementException**
- **<E> getLast( )** - Returns the last element from the invoking dequeu. Throws **NoSuchElementException**
- **<E> peekFirst( )** - Returns the first element from the invoking dequeu. Returns **null** if deque is empty
- **<E> peekLast( )** - Returns the last element from the invoking dequeu. Returns **null** if deque is empty
- **<E> pollFirst( )** - Removes first element and returns the same. Returns **null** if deque is empty
- **<E> pollLast( )** - Removes last element and returns the same. Returns **null** if deque is empty
- **<E> pop( )** - Returns the first element and removes it from dequeu. Throws **NoSuchElementException**
- **<E> removeFirst( )** - Removes first element and returns the same. Throws **NoSuchElementException**
- **<E> removeLast( )** - Removes last element and returns the same. Throws **NoSuchElementException**
- **boolean removeFirstOccurrence(Object *obj*)** - Removes the first occurrence of **obj** from the invoking deque
- **boolean removeLastOccurrence(Object *obj*)** - Removes the last occurrence of **obj** from the invoking deque
- **Iterator<E> descendingIterator( )** - Returns an iterator that moves from tail to head of the deque

www.btechsmartclass.com

JAVA SORTEDSET INTERFACE

Set Interface

The Set interface is a child interface of Collection interface. It does not defines any additional methods of it, it has all the methods that are inherited from Collection interface. The Set interface does not allow duplicates. Set is a generic interface.

SortedSet Interface

The **SortedSet** interface is a child interface of the Set interface. The SortedSet interface is available inside the **java.util** package. It defines the methods that are used by classes HashSet, LinkedHashSet, and TreeSet.

- 🔔 The SortedSet interface extends **Set** interface.
- 🔔 The SortedSet interface does not allow duplicate elements.
- 🔔 The SortedSet interface organise the elements based on the ascending order.

The SortedSet interface defines the following methods.

JAVA NAVIGABLESET INTERFACE

The **NavigableSet** interface is a child interface of the SortedSet interface. The NavigableSet interface is available inside the **java.util** package. It defines the methods that are used by class TreeSet.

🔔 The NavigableSet interface extends **SortedSet** interface.
🔔 The SortedSet interface does not allow duplicate elements.
🔔 The SortedSet interface organise the elements based on the ascending order.
The NavigableSet interface defines several utility methods that are used in the TreeSet class and they are as follows.

JAVA ARRAYLIST CLASS

The **ArrayList** class is a part of java collection framework. It is available inside the **java.util** package. The
ArrayList class extends AbstractList class and implements List interface.
The elements of ArrayList are organized as an array internally. The default size of an ArrayList is 10.

The ArrayList class is used to create a dynamic array that can grow or shrunk as needed.
- The ArrayList is a child class of **AbstractList**
- The ArrayList implements interfaces like **List**, **Serializable**, **Cloneable**, and **RandomAccess**.
- The ArrayList allows to store duplicate data values.
- The ArrayList allows to access elements randomly using index-based accessing.
- The ArrayList maintains the order of insertion.

ArrayList class declaration
The ArrayList class has the following declaration.
Example
public class ArrayList<E> extends AbstractList<E> implements List<E>, RandomAccess,
Cloneable,
Serializable

ArrayList class constructors
The ArrayList class has the following constructors.

JAVA ARRAYLIST CLASS

Operations on ArrayList
The ArrayList class allow us to perform several operations like adding, accesing, deleting, updating,
looping, etc. Let's look at each operation with examples.
Adding Items
The ArrayList class has the following methods to add items.
boolean add(E element) - Appends given element to the ArrayList.
boolean addAll(Collection c) - Appends given collection of elements to the ArrayList.
void add(int index, E element) - Inserts the given element at specified index.
boolean addAll(int index, Collection c) - Inserts the given collection of elements at specified index.

Accessing Items
The ArrayList class has the following methods to access items.
E get(int index) - Returns element at specified index from the ArrayList.
ArrayList subList(int startIndex, int lastIndex) - Returns an ArrayList that contails elements from
specified startIndex to lastIndex-1 from the invoking ArrayList.
int indexOf(E element) - Returns the index value of given element first occurence in the ArrayList.
int lastIndexOf(E element) - Returns the index value of given element last occurence in the ArrayList.

JAVA ARRAYLIST CLASS

Updating Items
The ArrayList class has the following methods to update or change items.
E set(int index, E newElement) - Replace the element at specified index with newElement in the invoking ArrayList.
ArrayList replaceAll(UnaryOperator e) - Replaces each element of invoking ArrayList with the result of applying the operator to that element.

Removing Items
The ArrayList class has the following methods to remove items.
E remove(int index) - Removes the element at specified index in the invoking ArrayList.
boolean remove(Object element) - Removes the first occurence of the given element from the invoking
ArrayList.
boolean removeAll(Collection c) - Removes the given collection of elements from the invoking ArrayList.
void retainAll(Collection c) - Removes all the elements except the given collection of elements from the
invoking ArrayList.
boolean removeIf(Predicate filter) - Removes all the elements from the ArrayList that satisfies the given
predicate.
void clear( ) - Removes all the elements from the ArrayList.

CSE/NRCM

JAVA LINKEDLIST CLASS

The **LinkedList** class is a part of java collection framework. It is available inside the **java.util** package. The LinkedList class extends **AbstractSequentialList** class and implements **List** and **Deque** interface. The elements of LinkedList are organized as the elements of linked list data structure.
The LinkedList class is used to create a dynamic list of elements that can grow or shrunk as needed.

- The LinkedList is a child class of **AbstractSequentialList**
- The LinkedList implements interfaces like **List**, **Deque**, **Cloneable**, and **Serializable**.
- The LinkedList allows to store duplicate data values.
- The LinkedList maintains the order of insertion.

LinkedList class declaration
The LinkedList class has the following declaration.
Example
public class LinkedList<E> extends AbstractSequentialList<E> implements List<E>, Deque<E>, Cloneable, Serializable

LinkedList class constructors
The LinkedList class has the following constructors.
LinkedList( ) - Creates an empty List.
LinkedList(Collection c) - Creates a List with given collection of elements.

CSE/NRCM

Operations on LinkedList

The LinkedList class allow us to perform several operations like adding, accesing, deleting, updating,
looping, etc. Let's look at each operation with examples.

Adding Items
The LinkedList class has the following methods to add items.
boolean add(E element) - Appends given element to the List.
boolean addAll(Collection c) - Appends given collection of elements to the List.
void add(int position, E element) - Inserts the given element at specified position.
boolean addAll(int position, Collection c) - Inserts the given collection of elements at specified
position.
void addFirst(E element) - Inserts the given element at beggining of
the list. **void addLast(E element)** - Inserts the given element at end
of the list. **boolean offer(E element)** - Inserts the given element at
end of the list.
boolean offerFirst(E element) - Inserts the given element at beggining of the list.
boolean offerLast(E element) - Inserts the given element at end of the list.
void push(E element) - Inserts the given element at beggining of the list.

Operations on LinkedList

Accessing Items
The LinkedList class has the following methods to access items.
E get(int position) - Returns element at specified position from the LinkedList.
E element( ) - Returns the first element from the invoking
LinkedList. **E getFirst( )** - Returns the first element from the
invoking LinkedList. **E getLast( )** - Returns the last element
from the invoking LinkedList.
E peek( ) - Returns the first element from the invoking LinkedList.
E peekFirst( ) - Returns the first element from the invoking LinkedList, and returns null if list is
empty. **E peekLast( )** - Returns the last element from the invoking LinkedList, and returns null
if list is empty. **int indexOf(E element)** - Returns the index value of given element first
occurence in the LinkedList. **int lastIndexOf(E element)** - Returns the index value of given
element last occurence in the LinkedList. **E pop( )** - Returns the first element from the
invoking LinkedList.

Updating Items
The LinkedList class has the following methods to update or change items.
E set(int index, E newElement) - Replace the element at specified index with
newElement in the invoking LinkedList.

Operations on LinkedList

Removing Items
The LinkedList class has the following methods to remove items.
E remove( ) - Removes the first element from the invoking LinkedList.
E remove(int index) - Removes the element at specified index in the invoking
LinkedList. **boolean remove(Object element)** - Removes the first occurrence of the
given element from the invoking LinkedList.
E removeFirst( ) - Removes the first element from the invoking LinkedList.
E removeLast( ) - Removes the last element from the invoking LinkedList.
boolean removeFirstOccurrence(Object element) - Removes from the first occurrence of
the given
element from the invoking LinkedList.
boolean removeLastOccurrence(Object element) - Removes from the last occurrence of
the given
element from the invoking LinkedList.
E poll( ) - Removes the first element from the LinkedList, and returns null if the list is
empty.
E pollFirst( ) - Removes the first element from the LinkedList, and returns null if the list
is empty. **E pollLast( )** - Removes the last element from the LinkedList, and returns null
if the list is empty. **E pop( )** - Removes the first element from the LinkedList.
void clear( ) - Removes all the elements from the LinkedList.

JAVA PRIORITYQUEUE CLASS

The **PriorityQueue** class is a part of java collection framework. It is available inside the **java.util** package. The PriorityQueue class extends **AbstractQueue** class and implements **Serializable** interface.

The elements of PriorityQueue are organized as the elements of queue data structure, but it does not follow FIFO principle. The PriorityQueue elements are organized based on the priority heap.

The PriorityQueue class is used to create a dynamic queue of elements that can grow or shrunk as needed.

- The PriorityQueue is a child class of **AbstractQueue**
- The PriorityQueue implements interface **Serializable**.
- The PriorityQueue allows to store duplicate data values, but not null values.
- The PriorityQueue maintains the order of insertion.
- The PriorityQueue used priority heap to organize its elements.

PriorityQueue class declaration

The PriorityQueue class has the following declaration.

Example

```
public class PriorityQueue<E> extends AbstractQueue<E> implements Serializable
```

JAVA PRIORITYQUEUE CLASS

PriorityQueue class constructors
The PriorityQueue class has the following constructors.
PriorityQueue( ) - Creates an empty PriorityQueue with the default initial capacity (11) that orders its elements according to their natural ordering.
PriorityQueue(Collection c) - Creates a PriorityQueue with given collection of elements.
PriorityQueue(int initialCapacity) - Creates an empty PriorityQueue with the specified initial capacity. **PriorityQueue(int initialCapacity, Comparator comparator)** - Creates an empty PriorityQueue with the specified initial capacity that orders its elements according to the specified comparator.
PriorityQueue(PriorityQueue pq) - Creates a PriorityQueue with the elements in the specified priority
queue.
PriorityQueue(SortedSet ss) - Creates a PriorityQueue with the elements in the specified SortedSet.

Operations on PriorityQueue
The PriorityQueue class allow us to perform several operations like adding, accesing, deleting, updating,
looping, etc. Let's look at each operation with examples.
Adding Items
The PriorityQueue class has the following methods to add items.
boolean add(E element) - Appends given element to the PriorityQueue.
boolean addAll(Collection c) - Appends given collection of elements to the PriorityQueue.

JAVA PRIORITYQUEUE CLASS

Accessing Items
The PriorityQueue class has the following methods to access items.
E element( ) - Returns the first element from the invoking PriorityQueue.
E peek( ) - Returns the first element from the invoking PriorityQueue, returns null if this queue is empty.

Updating Items
The PriorityQueue class has no methods to update or change items.
Removing Items
The PriorityQueue class has the following methods to remove items.
E remove( ) - Removes the first element from the invoking PriorityQueue.
boolean remove(Object element) - Removes the first occurrence of the given element from the invoking PriorityQueue.
boolean removeAll(Collection c) - Removes all the elements of specified collection from the invoking PriorityQueue.
boolean removeIf(Predicate p) - Removes all of the elements of this collection that satisfy the given predicate.
boolean retainAll(Collection c) - Removes all the elements except those are in the specified collection from the invoking PriorityQueue.
E poll( ) - Removes the first element from the PriorityQueue, and returns null if the list is empty.
void clear( ) - Removes all the elements from the PriorityQueue.

JAVA ARRAYDEQUE CLASS

The **ArrayDeque** class is a part of java collection framework. It is available inside the **java.util** package.
The ArrayDeque class extends **AbstractCollection** class and implements **Deque**, **Cloneable**, and **Serializable** interfaces.
The elements of ArrayDeque are organized as the elements of double ended queue data structure. The ArrayDeque is a special kind of array that grows and allows users to add or remove an element from both the sides of the queue.
The ArrayDeque class is used to create a dynamic double ended queue of elements that can grow or
shrunk as needed.

- The ArrayDeque is a child class of **AbstractCollection**
- The ArrayDeque implements interfaces like **Deque**, **Cloneable**, and **Serializable**.
- The ArrayDeque allows to store duplicate data values, but not null values.
- The ArrayDeque maintains the order of insertion.
- The ArrayDeque allows to add and remove elements at both the ends.
- The ArrayDeque is faster than LinkedList and Stack.

ArrayDeque class declaration

The ArrayDeque class has the following declaration.

Example

public class ArrayDeque<E> extends AbstractCollection<E> implements Deque<E>, Cloneable,
Serializable

JAVA ARRAYDEQUE CLASS

ArrayDeque class constructors
The PriorityQueue class has the following constructors.
ArrayDeque( ) - Creates an empty ArrayDeque with the default initial capacity (16).
ArrayDeque(Collection c) - Creates a ArrayDeque with given collection of elements.
ArrayDeque(int initialCapacity) - Creates an empty ArrayDeque with the specified initial capacity.

Operations on ArrayDeque
The ArrayDeque class allow us to perform several operations like adding, accesing, deleting, updating, looping, etc. Let's look at each operation with examples.

Adding Items
The ArrayDeque class has the following methods to add items.
boolean add(E element) - Appends given element to the ArrayDeque.
boolean addAll(Collection c) - Appends given collection of elements to the ArrayDeque.
void addFirst(E element) - Adds given element at front of the ArrayDeque. **void addLast(E element)** - Adds given element at end of the ArrayDeque. **boolean offer(E element)** - Adds given element at end of the ArrayDeque.
boolean offerFirst(E element) - Adds given element at front of the ArrayDeque. **boolean offerLast(E element)** - Adds given element at end of the ArrayDeque. **void push(E element)** - Adds given element at front of the ArrayDeque.

CSE/NRCM

JAVA ARRAYDEQUE CLASS

Accessing Items
The ArrayDeque class has the following methods to access items.
E element( ) - Returns the first element from the invoking
ArrayDeque. **E getFirst( )** - Returns the first element from the
invoking ArrayDeque. **E getLast( )** - Returns the last element
from the invoking ArrayDeque.
E peek( ) - Returns the first element from the invoking ArrayDeque, returns null if this queue is empty. **E peekFirst( )** - Returns the first element from the invoking ArrayDeque, returns null if this queue is empty.
E peekLast( ) - Returns the last element from the invoking ArrayDeque, returns null if this queue is
empty.

JAVA ARRAYDEQUE CLASS

Updating Items
The ArrayDeque class has no methods to update or change items.

Removing Items
The ArrayDeque class has the following methods to remove items.
E remove( ) - Removes the first element from the invoking ArrayDeque.
E removeFirst( ) - Removes the first element from the invoking ArrayDeque.
E removeLast( ) - Removes the last element from the invoking ArrayDeque.
boolean remove(Object o) - Removes the specified element from the invoking ArrayDeque.
boolean removeFirstOccurrence(Object o) - Removes the first occurrence of the specified element in
this ArrayDeque.
boolean removeLastOccurrence(Object o) - Removes the last occurrence of the specified element in
this ArrayDeque.
boolean removeIf(Predicate p) - Removes all of the elements of ArrayDeque collection that satisfy the
given predicate.
boolean retainAll(Collection c) - Removes all of the elements of ArrayDeque collection except specified collection of elements.
E poll( ) - Removes the first element from the ArrayDeque, and returns null if the list is empty.
E pollFirst( ) - Removes the first element from the ArrayDeque, and returns null if the list is empty. **E pollLast( )** - Removes the last element from the ArrayDeque, and returns null

JAVA HASHSET CLASS

The **HashSet** class is a part of java collection framework. It is available inside the **java.util** package. The
HashSet class extends **AbstractSet** class and implements **Set** interface.
The elements of HashSet are organized using a mechanism called hashing. The HashSet is used to create hash table for storing set of elements.
The HashSet class is used to create a collection that uses a hash table for storing set of elements.

- The HashSet is a child class of **AbstractSet**
- The HashSet implements interfaces like **Set**, **Cloneable**, and **Serializable**.
- The HashSet does not allows to store duplicate data values, but null values are allowed.
- The HashSet does not maintains the order of insertion.
- The HashSet initial capacity is 16 elements.
- The HashSet is best suitable for search operations.

HashSet class declaration

The HashSet class has the following declaration.

Example

public class HashSet<E> extends AbstractSet<E> implements Set<E>, Cloneable, Serializable

JAVA HASHSET CLASS

HashSet class constructors
The HashSet class has the following constructors.
HashSet( ) - Creates an empty HashSet with the default initial capacity (16).
HashSet(Collection c) - Creates a HashSet with given collection of elements.
HashSet(int initialCapacity) - Creates an empty HashSet with the specified initial capacity. **HashSet(int initialCapacity, float loadFactor)** - Creates an empty HashSet with the specified initial capacity and loadFactor.
Operations on HashSet
The HashSet class allow us to perform several operations like adding, accesing, deleting, updating,
looping, etc. Let's look at each operation with examples.
Adding Items
The HashSet class has the following methods to add items.
boolean add(E element) - Inserts given element to the HashSet.
boolean addAll(Collection c) - Inserts given collection of elements to the HashSet.

MAP INTERFACE CLASSES IN JAVA

The java collection framework has an interface **Map** that is available inside the **java.util** package. The
Map interface is not a subtype of Collection interface.
The **Map** interface has the following three classes.

Class	Description
HashMap	It implements the Map interface, but it doesn't maintain any order.
LinkedHashMap	It implements the Map interface, it also extends HashMap class. It maintains the insertion order.
TreeMap	It implements the Map and SortedMap interfaces. It maintains the ascending order.

Commonly used methods defined by Map interface

Method	Description
Object put(Object k, Object v)	It performs an entry into the Map.
Object putAll(Map m)	It inserts all the entries of m into invoking Map.
Object get(Object k)	It returns the value associated with given key.
boolean containsKey(Object k)	It returns true if map contain k as key. Otherwise false.
Set keySet()	It returns a set that contains all the keys from the invoking Map.
Set valueSet()	It returns a set that contains all the values from the invoking Map.
Set entrySet()	It returns a set that contains all the entries from the invoking Map.

HashMap Class

The HashMap class is a child class of AbstractMap, and it implements the Map interface. The HashMap
is used to store the data in the form of key, value pair using hash table concept.

Key Properties of HashMap

HashMap is a child class of AbstractMap class.
HashMap implements the interfeaces Map, Cloneable, and Serializable. HashMap stores data as a pair of key and value.
HashMap uses Hash table concept to store the data.
HashMap does not allow duplicate keys, but values may be repeated. HashMap allows only one null key and multiple null values.
HashMap does not follow any oreder.
HashMap has the default capacity 16 entries.

UNIT – V
APPLETS

The Basic GUI Application

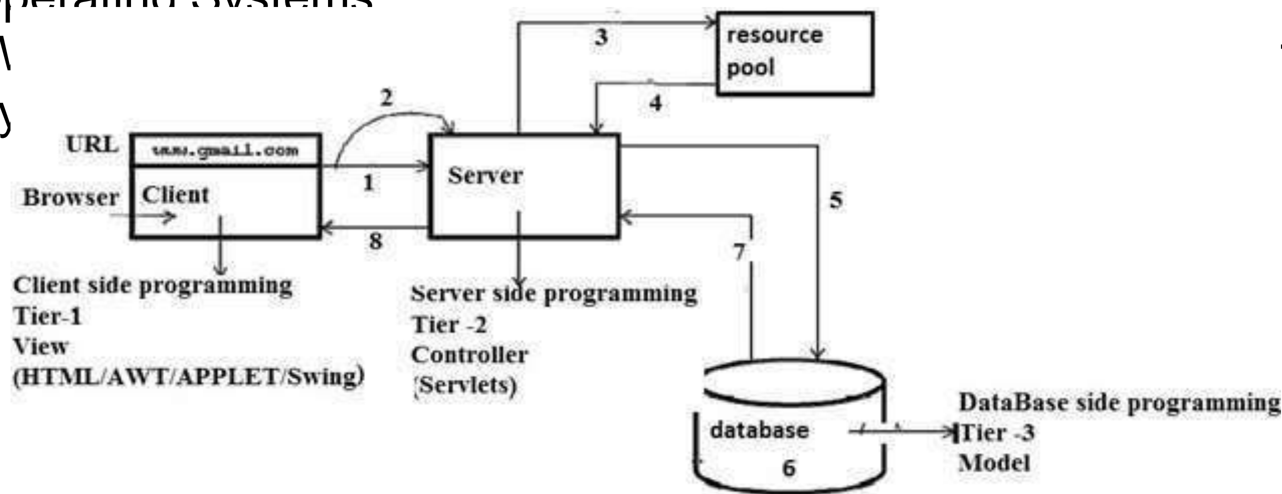
```
import javax.swing.JOptionPane;
public class HelloWorldGUI1
{
public static void main(String[] args) {
JOptionPane.showMessageDialog( null, "Hello
World!" );
}

}
```

When this program is run, a window appears on the screen that contains the message “Hello World!”. The window also contains an “OK” button for the user to click after reading the message. When the user clicks this button, the window closes and the program ends. By the way, this program can be placed in a file named HelloWorldGUI1.java, compiled, and run just like any other Java program.

LIMITATIONS OF AWT

Summary on limitations of AWT

AWT supports limited number of GUI
components AWT component are Heavy
weight components
AWT components are developed by using platform specific
code AWT components behave differently in different
Operating Systems
A\ ... ating
Sy



MVC ARCHITECTURE

In real time applications, in the case of server side programming one must follow the architecture to develop a distributed application.To develop any distributed application, it is always recommended to follow either 3-tier architecture or 2-tier architecture or n-tier architecture.

3-tier architecture is also known as MVC architecture.

M stands for Model (database programming),

V stands for View (client side programming, HTML/AWT/APPLET/Swing/JSP) C stands for Controller (server side programming, Servlets).

Model :

This is the data layer which consists of the business logic of the system. It consists of all the data of the application
It also represents the state of the application.
It consists of classes which have the connection to the database.
The controller connects with model and fetches the data and sends to the view layer.
The model connects with the database as well and stores the data into a database which is connected
to it.

MVC ARCHITECTURE

Model :

This is the data layer which consists of the business logic of the system.
It consists of all the data of the application
It also represents the state of the application.
It consists of classes which have the connection to the database.
The controller connects with model and fetches the data and sends to the view layer.
The model connects with the database as well and stores the data into a database which is connected to it.

View :

This is a presentation layer.
It consists of HTML, JSP, etc. into it.
It normally presents the UI of the application.
It is used to display the data which is fetched from the controller which in turn fetching data from model
layer classes.
This view layer shows the data on UI of the application.

Controller:

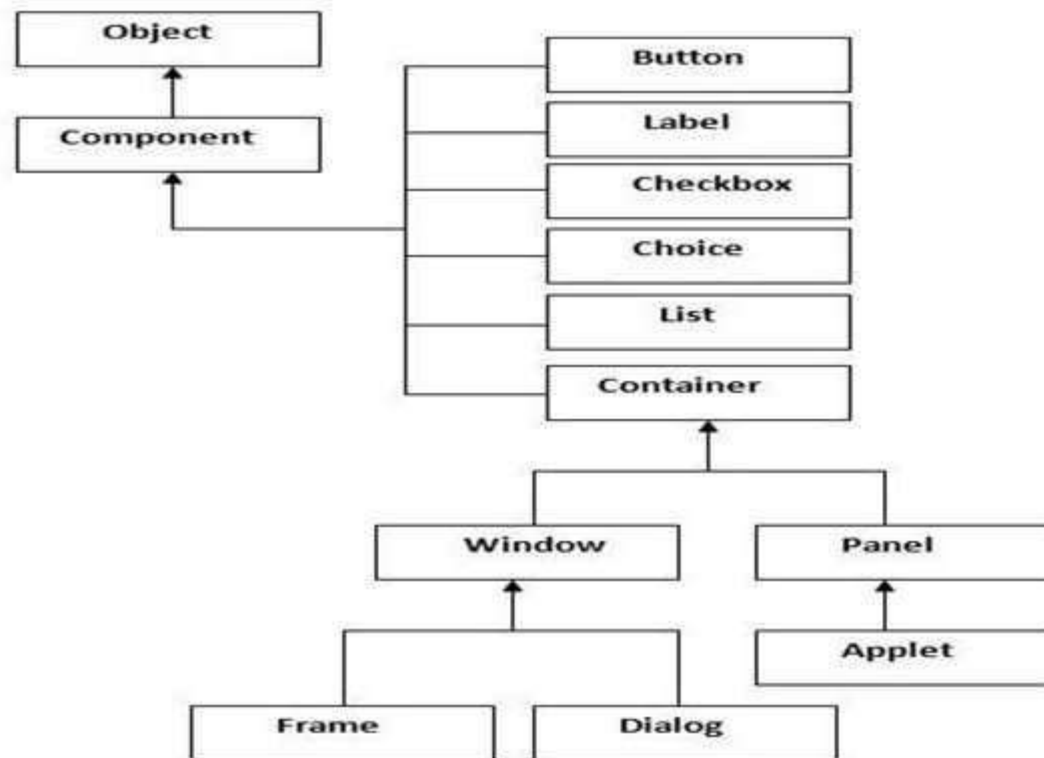
It acts as an interface between View and Model.
It intercepts all the requests which are coming from the view layer.
It receives the requests from the view layer and processes the requests and does the necessary validation for
the request.

This requests is further sent to model layer for data processing, and once the request is processed, it

COMPONENTS

Component is an object having a graphical representation that can be displayed on the screen and that can
interact with the user. For examples buttons, checkboxes, list and scrollbars of a graphical user interface.

A Component is an abstract super class for GUI controls and it represents an object with graphical representati



COMPONENTS

Every AWT controls inherits properties from Component class

Component	Description
Label	The easiest control to use is a label. A label is an object of type Label, and it contains a string,which it displays. Labels are passive controls that do not support any interaction with theuser. Label defines the following constructors
Button	This class creates a labeled button.
Check Box	A check box is a graphical component that can be in either an on (true) or off (false) state.
Check Box Group	The CheckboxGroup class is used to group the set of checkbox.
List	The List component presents the user with a scrolling list of text items.
Text Field	A TextField object is a text component that allows for the editing of a single line of text.
Text Area	A TextArea object is a text component that allows for the editing of a multiple lines of text.
Choice	A Choice control is used to show pop up menu of choices. Selected choice is shown on the top of the menu.
Canvas	A Canvas control represents a rectangular area where application can draw something or can receive inputs created by user.
Image	An Image control is superclass for all image classes representing graphical images.
Scroll Bar	A Scrollbar control represents a scroll bar component in order to enable user to select from range of values.
Dialog	A Dialog control represents a top-level window with a title and a border used to take some form of input from the user.
File Dialog	A FileDialog control represents a dialog window from which the user can select a file.

COMPONENTS

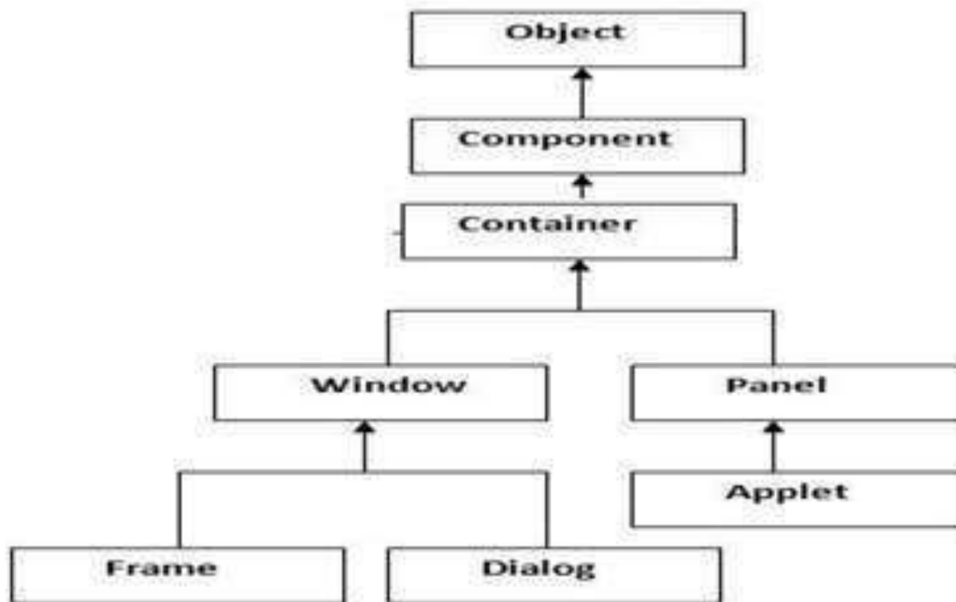
Commonly used Methods of Component class:

Method	Description
public void add(Component c)	inserts a component on this component.
public void setSize(intwidth,int height)	sets the size (width and height) of the component.
public void setLayout(LayoutManager m)	defines the layout manager for the component.
public void setVisible(boolean status)	changes the visibility of the component, by default false.
void remove(Component obj)	Here, obj is a reference to the control you want to remove.
void removeAll( ).	You can remove all controls by

CONTAINERS

Abstract Windowing Toolkit (AWT): Abstract Windowing Toolkit (AWT) is used for GUI programming in java.

AWT Container Hierarchy:



CONTAINERS

Container:
The Container is a component in AWT that can contain another components like buttons, textfields, labels
etc. The classes that extends Container class are known as container.

Window:
The window is the container that have no borders and menubars. You must use frame, dialog or another
window for creating a window.

Panel:
The Panel is the container that doesn't contain title bar and MenuBars. It can have other components like
button, textfield etc.

Frame:
The Frame is the container that contain title bar and can have MenuBars. It can have other components like
button, textfield etc.

Frame

There are two ways to create a frame:

By extending Frame class (inheritance)

By creating the object of Frame class (association)

Example program to create a frame by extending Frame class (inheritance) import java.awt.*;

```
class First extends Frame
{
First()
{
Button b=new Button("click me");
b.setBounds(30,100,80,30);/*setting button position public void setBounds(int xaxis, int yaxis, int
width, int
height); have been used in the above example that sets the position of the
button.*/ add(b);//adding button into frame
setSize(300,300);//frame size 300 width and 300 height setLayout(null);//no layout now bydefault
BorderLayout setVisible(true);//now frame willbe visible, bydefault not visible
}
public static void main(String args[])
{
First f=new First();
}
```

Frame

2.Example program to create a frame by creating the object of Frame class

```
import java.awt.*; class First2{
First2(){ Frame f=new Frame();
Button b=new Button("click me"); b.setBounds(30,50,80,30);
f.add(b); f.setSize(300,300); f.setLayout(null); f.setVisible(true);
}
public static void main(String args[]){ First2 f=new First2();
}
}
```

LAYOUT MANAGERS

The LayoutManagers are used to arrange components in a particular manner. The Java LayoutManagers facilitates us to control the positioning and size of the components in GUI forms. LayoutManager is an interface that is implemented by all the classes of layout managers. There are the following classes that represent the layout managers:

- java.awt.BorderLayout
- java.awt.FlowLayout
- java.awt.GridLayout
- java.awt.CardLayout
- java.awt.GridBagLayout

BorderLayout

The BorderLayout is used to arrange the components in five regions: north, south, east, west, and center. Each region (area) may contain one component only. It is the default layout of a frame or window. The BorderLayout provides five constants for each region:
public static final int
NORTH public static final
int SOUTH public static
final int EAST public static
final int WEST public
static final int CENTER

Constructors of BorderLayout class:
BorderLayout():
creates a border layout but with no gaps between the components.

BorderLayout(int hgap, int vgap):
creates a border layout with the given horizontal and vertical gaps between the components.

FlowLayout

The Java FlowLayout class is used to arrange the components in a line, one after another (in a flow). It is the
default layout of the applet or panel.

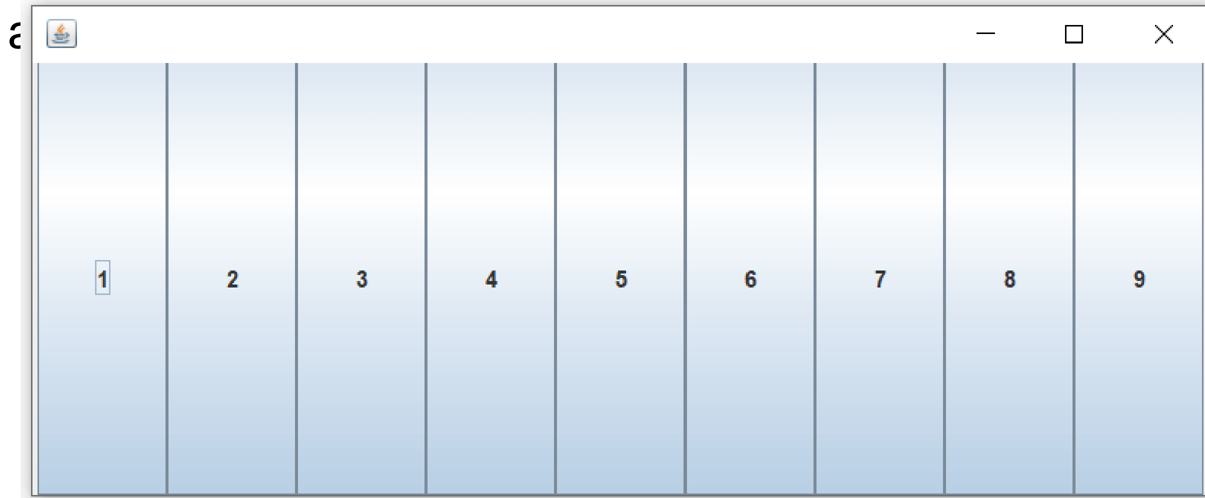
Fields of FlowLayout
class public static final int
LEFT public static final int
RIGHT public static final int
CENTER public static final
int LEADING public static
final int TRAILING

Constructors of FlowLayout class
FlowLayout(): creates a flow layout with centered alignment and a default 5 unit horizontal and vertical gap. **FlowLayout(int align):** creates a flow layout with the given alignment and a default 5 unit horizontal and vertical gap.
FlowLayout(int align, int hgap, int vgap): creates a flow layout with the given alignment and the given
horizontal and vertical gap.

GridLayout

The Java GridLayout class is used to arrange the components in a rectangular grid. One component is
displayed in each rectangle.

Constructors of GridLayout class
GridLayout(): creates a grid layout with one column per component in a row.
GridLayout(int rows, int columns): creates a grid layout with the given rows and columns but no gaps
between the components.
GridLayout(int rows, int columns, int hgap, int vgap): creates a grid layout with the given rows
a .



CardLayout

The Java CardLayout class manages the components in such a manner that only one component is visible at
a time. It treats each component as a card that is why it is known as CardLayout.

Constructors of CardLayout Class
CardLayout(): creates a card layout with zero horizontal and vertical gap.
CardLayout(int hgap, int vgap): creates a card layout with the given horizontal and vertical gap.

Commonly Used Methods of CardLayout Class

public void next(Container parent): is used to flip to the next card of the given container.
public void previous(Container parent): is used to flip to the previous card of the given container.
public void first(Container parent): is used to flip to the first card of the given container.
public void last(Container parent): is used to flip to the last card of the given container.
public void show(Container parent, String name): is used to flip to the specified card with the given name.

GridBagLayout

The Java GridBagLayout class is used to align components vertically, horizontally or along their baseline. The components may not be of the same size. Each GridBagLayout object maintains a dynamic, rectangular grid of cells. Each component occupies one or more cells known as its display area. Each component associates an instance of GridBagConstraints. With the help of the constraints object, we arrange the component's display area on the grid. The GridBagLayout manages each component's minimum and
preferred sizes in order to determine the component's size. GridBagLayout components are also arranged in
the rectangular grid but can have many different sizes and can occupy multiple rows or columns.

Constructor

GridBagLayout(): The parameterless constructor is used to create a grid bag layout manager.

Modifier and Type	Field	Description
double[]	columnWeights	It is used to hold the overrides to the column weights.
int[]	columnWidths	It is used to hold the overrides to the column minimum width.
protected Hashtable<Component,GridBagConstraints>	comptable	It is used to maintains the association between a component and its gridbag constraints.
protected GridBagConstraints	defaultConstraints	It is used to hold a gridbag constraints instance containing the default values.
protected GridBagLayoutInfo	layoutInfo	It is used to hold the layout information for the gridbag.
protected static int	MAXGRIDSIZE	No longer in use just for backward compatibility
protected static int	MINSIZE	It is smallest grid that can be laid out by the grid bag layout.
protected static int	PREFERREDSIZE	It is preferred grid size that can be laid out by the grid bag layout.
int[]	rowHeights	It is used to hold the overrides to the row minimum heights.
double[]	rowWeights	It is used to hold the overrides to the row weights.

Thank you