PRATHYUSHA

ENGINEERING COLLEGE

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

REGULATION 2021 – III SEMESTER

CS3391 – OBJECT ORIENTED PROGRAMMING

CS3391 OBJECT ORIENTED PROGRAMMING

COURSE OBJECTIVES:

- To understand Object Oriented Programming concepts and basics of Java programming language
- To know the principles of packages, inheritance and interfaces
- To develop a java application with threads and generics classes
- To define exceptions and use I/O streams
- To design and build Graphical User Interface Application using JAVAFX 66

UNIT I INTRODUCTION TO OOP AND JAVA

Overview of OOP – Object oriented programming paradigms – Features of Object Oriented Programming – Java Buzzwords – Overview of Java – Data Types, Variables and Arrays – Operators – Control Statements – Programming Structures in Java – Defining classes in Java – ConstructorsMethods -Access specifiers - Static members- Java Doc comments

UNIT II INHERITANCE, PACKAGES AND INTERFACES

Overloading Methods – Objects as Parameters – Returning Objects –Static, Nested and Inner Classes. Inheritance: Basics– Types of Inheritance -Super keyword -Method Overriding – Dynamic Method Dispatch –Abstract Classes – final with Inheritance. Packages and Interfaces: Packages – Packages and Member Access –Importing Packages – Interfaces.

UNIT III EXCEPTION HANDLING AND MULTITHREADING

Exception Handling basics – Multiple catch Clauses – Nested try Statements – Java's Built-in Exceptions – User defined Exception. Multithreaded Programming: Java Thread Model– Creating a Thread and Multiple Threads – Priorities – Synchronization – Inter Thread Communication- Suspending –Resuming, and Stopping Threads –Multithreading. Wrappers – Auto boxing.

UNIT IV I/O, GENERICS, STRING HANDLING

I/O Basics – Reading and Writing Console I/O – Reading and Writing Files. Generics: Generic Programming – Generic classes – Generic Methods – Bounded Types – Restrictions and Limitations. Strings: Basic String class, methods and String Buffer Class.

UNIT V JAVAFX EVENT HANDLING, CONTROLS AND COMPONENTS

JAVAFX Events and Controls: Event Basics – Handling Key and Mouse Events. Controls: Checkbox, ToggleButton – RadioButtons – ListView – ComboBox – ChoiceBox – Text Controls – ScrollPane. Layouts – FlowPane – HBox and VBox – BorderPane – StackPane – GridPane. Menus – Basics – Menu – Menu bars – MenuItem.

COURSE OUTCOMES:

On completion of this course, the students will be able to

CO1:Apply the concepts of classes and objects to solve simple problems

CO2:Develop programs using inheritance, packages and interfaces

CO3:Make use of exception handling mechanisms and multithreaded model to solve real world problems

CO4:Build Java applications with I/O packages, string classes, Collections and generics concepts

CO5:Integrate the concepts of event handling and JavaFX components and controls for developing GUI based applications

TEXT BOOKS:

1. Herbert Schildt, “Java: The Complete Reference”, 11 th Edition, McGraw Hill Education, New Delhi, 2019

2. Herbert Schildt, “Introducing JavaFX 8 Programming”, 1 st Edition, McGraw Hill Education, New Delhi, 2015 67

REFERENCE:

1. Cay S. Horstmann, “Core Java Fundamentals”, Volume 1, 11 th Edition, Prentice Hall, 2018.

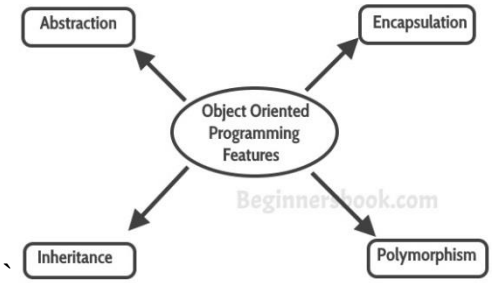
CS3391 OBJECT ORIENTED PROGRAMMING

UNIT I INTRODUCTION TO OOP AND JAVA

Overview of OOP – Object oriented programming paradigms – Features of Object Oriented Programming – Java Buzzwords – Overview of Java – Data Types, Variables and Arrays – Operators – Control Statements – Programming Structures in Java – Defining classes in Java – Constructors-Methods -Access specifiers - Static members- Java Doc comments

---

1. **Explain the features of object oriented programming.**



Object-oriented programming organizes a program around its data that is, objects) and a set of well-defined interfaces to that data. An object-oriented program can be characterized as *data controlling access to code*.

1) ABSTRACTION:

Abstraction is a process of showing only "relevant" data and "hide" unnecessary details of an object from the user. For example, when you login to your bank account online, you enter your user_id and password and press login, what happens when you press login, how the input data sent to server, how it gets verified is all abstracted away from the you.
Eg. Car, ATM machine

2) ENCAPSULATION:

- *Encapsulation* is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse.
- One way to think about encapsulation is as a protective wrapper that prevents the code and data from being arbitrarily accessed by other code defined outside the wrapper. Access to the code and data inside the wrapper is tightly controlled through a well-defined interface.

Class:

A *class* defines the structure and behavior (data and code) that will be shared by a set of objects. Each object of a given class contains the structure and behavior defined by the class

Object :

Object is an instance of a class. **Objects** have states and behaviors.

3) INHERITANCE:

Inheritance is the process by which one object acquires the properties of another object. This is important because it supports the concept of hierarchical classification.

4) POLYMORPHISM:

- Polymorphism refers to the ability of a variable, object or function to take on multiple forms. The concept of polymorphism is often expressed by the phrase "one interface, multiple methods." This means that it is possible to design a generic interface to a group of related activities. This helps reduce complexity by allowing the same interface to be used to specify a *general class of action*.

Types of Polymorphism
1) Static Polymorphism
2) Dynamic Polymorphism

Static Polymorphism:

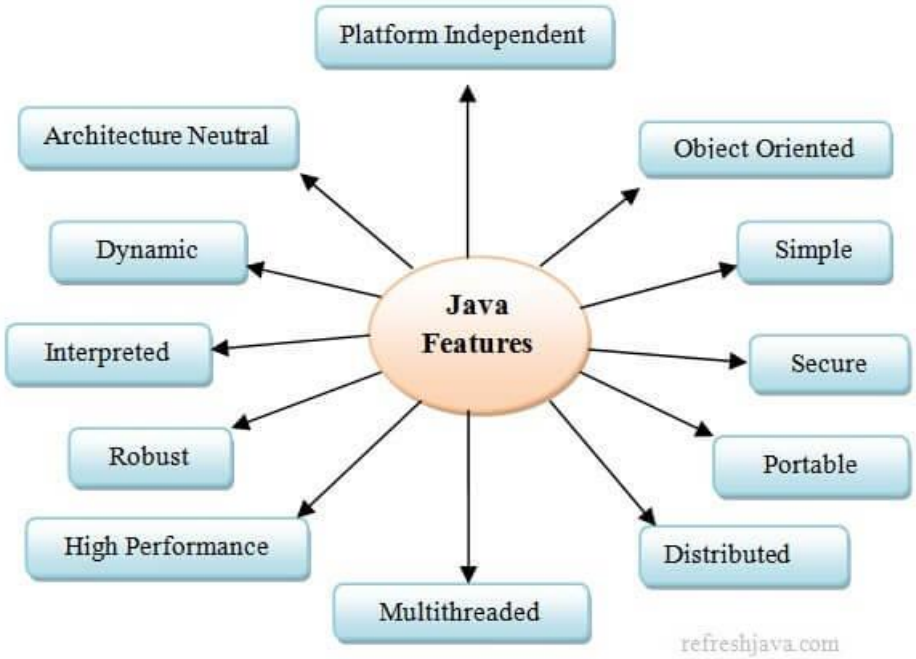
Polymorphism that is resolved during compiler time is known as static polymorphism. Method overloading can be considered as static polymorphism example.
Method Overloading: This allows us to have more than one methods with same name in a class that differs in signature.

Dynamic Polymorphism

- It is also known as Dynamic Method Dispatch. Dynamic polymorphism is a process in which a call to an overridden method is resolved at runtime rather, thats why it is called runtime polymorphism

2. Explain about the java buzzwords.

Java buzzwords are:
• Simple
• Secure
• Portable
• Object-oriented
• Robust
• Multithreaded
• Architecture-neutral
• Interpreted
• High performance
• Distributed
• Dynamic



1) Simple

Java was designed to be easy for the professional programmer to learn and use effectively. Assuming that you have some programming experience, you will not find Java hard to master. If you already understand the basic concepts of object oriented programming, learning Java will be even easier. Best of all, if you are an experienced C++ programmer, moving to Java will require very little effort. Because Java inherits the C/C++ syntax and many of the object-oriented features of C++, most programmers have little trouble learning Java.

2) **Secure:**

- As an Internet programming language, Java is used in a networked and distributed environment. If you download a Java applet (a special kind of program) and run it on your computer, it will not damage your system because Java implements several security mechanisms to protect the system against harm caused by stray programs. The security is based on the premise that nothing should be trusted.

3) **Portable**

- Because Java is architecture neutral, Java programs are portable. They can be run on any platform without being recompiled. Moreover, there are no platform-specific features in the Java language.

4) Object-Oriented

Java is an object-oriented programming language. Everything in Java is an object. Object-oriented means we organize our software as a combination of different types of objects that incorporate both data and behaviour. All code belong to some class. Classes are in turn arranged in a hierarchy or package structure

5) Robust

Java is simple. Java is not using pointers. Java support Exception handling. The try/catch/finally series allows for simplified error recovery. Java is Strongly typed language – many errors caught during compilation.

6) **Multithreaded**
Java was designed to meet the real-world requirement of creating interactive, networked programs. To accomplish this, Java supports multithreaded programming, which allows to write programs that do many things simultaneously. Java's easy-to-use approach to multithreading allows to think about the specific behaviour of your program, not the multitasking subsystem.

7) **Architecture-Neutral**
The Java designers designed the Java Virtual Machine(JVM). The goal of JVM is "write once; run anywhere, anytime, forever."

8) **Interpreted :**
Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java bytecode. This code can be executed on any system that implements the Java Virtual Machine.

9) **High Performance:**
Java is faster than other traditional interpreted programming languages because Java bytecode is "close" to native code. The Java bytecode was carefully designed so that it would be easy to translate directly into native machine code for very high performance by using a just-in-time compiler. Java run-time systems that provide this feature lose none of the benefits of the platform-independent code.

10) **Distributed:**
Java is designed for the distributed environment of the Internet because it handles TCP/IP protocols. In fact, accessing a resource using a URL is not much different from accessing a file. Java also supports Remote Method Invocation (RMI). This feature enables a program to invoke methods across a network.

11) **Dynamic:**
Java programs carry with them substantial amounts of run-time type information that is used to verify and resolve accesses to objects at run time. This makes it possible to dynamically link code in a safe and expedient manner. This is crucial to the robustness of the Java environment, in which small fragments of bytecode may be dynamically updated on a running system.

3. Explain about literals in java.

Integer Literals

Any whole number value is an integer literal. Examples are 1, 2, 3, and 42. These are all decimal values, meaning they are describing a base 10 number. Two other bases that can be used in integer literals are *octal* (base eight) and *hexadecimal* (base 16). Octal values are denoted in Java by a leading zero.Integer literals create an **int** value, which in Java is a 32-bit integer value.

When a literal value is assigned to a **byte** or **short** variable, no error is generated if the literal value is within the range of the target type. An integer literal can always be assigned to a **long** variable. However, to specify a **long** literal, you will need to explicitly tell the compiler that the literal value is of type **long**. You do this by appending an upper- or lowercase *L* to the literal. For example, 0x7fffffffffffffL or 9223372036854775807L is the largest **long**. An integer can also be assigned to a **char** as long as it is within range.

Floating-Point Literals

Floating-point numbers represent decimal values with a fractional component. They can be expressed in either standard or scientific notation. *Standard notation* consists of a whole number component followed by a decimal point followed by a fractional component.

Floating-point literals in Java default to **double** precision. To specify a **float** literal, you must append an *F* or *f* to the constant. You can also explicitly specify a **double** literal by appending

a *D* or *d*. Doing so is, of course, redundant. The default **double** type consumes 64 bits of storage, while the smaller **float** type requires only 32 bits.

Boolean Literals

Boolean literals are simple. There are only two logical values that a **Boolean** value can have, **true** and **false**. The values of **true** and **false** do not convert into any numerical representation. The **true** literal in Java does not equal 1, nor does the **false** literal equal 0. In Java, the Boolean literals can only be assigned to variables declared as **boolean** or used in expressions with Boolean operators.

Character Literals

Characters in Java are indices into the Unicode character set. They are 16-bit values that can be converted into integers and manipulated with the integer operators, such as the addition and subtraction operators. A literal character is represented inside a pair of single quotes. All of the visible ASCII characters can be directly entered inside the quotes, such as '*a*', 'z', and '@'.

String Literals

String literals in Java are specified like they are in most other languages—by enclosing a sequence of characters between a pair of double quotes. Examples of string literals are

"Hello World"

"two\nlines"

4. **Explain operators in operators.**

Java provides a rich operator environment. Most of its operators can be divided into the following four groups:

1. Arithmetic Operators
2. Bitwise Operators
3. Relational Operators
4. Logical Operators

Arithmetic Operators

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following table lists the arithmetic operators:

Operator	Result
+	Addition
–	Subtraction (also unary minus)
*	Multiplication
/	Division
%	Modulus
++	Increment
+=	Addition assignment
–=	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Modulus assignment
– –	Decrement

The Modulus Operator

The modulus operator, **%**, returns the remainder of a division operation. It can be applied to floating-point types as well as integer types. The following example program demonstrates the **%**:

Arithmetic Compound Assignment Operators

Java provides special operators that can be used to combine an arithmetic operation with an assignment. As you probably know, statements like the following are quite common in programming:

a = a + 4;

In Java, you can rewrite this statement as shown here:

a += 4;

The Bitwise Operators

Java defines several *bitwise operators* that can be applied to the integer types, **long**, **int**, **short**,
char, and **byte**. These operators act upon the individual bits of their operands. They are summarized in the following table:

Operator	Result
~	Bitwise unary NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
>>	Shift right
>>>	Shift right zero fill
<<	Shift left
&=	Bitwise AND assignment
|=	Bitwise OR assignment
^=	Bitwise exclusive OR assignment
>>=	Shift right assignment
>>>=	Shift right zero fill assignment
<<=	Shift left assignment

The bitwise operators manipulate the bits within an integer.

All of the integer types are represented by binary numbers of varying bit widths. For example, the **byte** value for 42 in binary is 00101010, where each position represents a power of two, starting with 20 at the rightmost bit. The next bit position to the left would be 21, or 2, continuing toward the left with 22, or 4, then 8, 16, 32, and so on. So 42 has 1 bits set at positions 1, 3, and 5 (counting from 0 at the right); thus, 42 is the sum of 21 + 23 + 25, which is 2 + 8 + 32.

All of the integer types (except **char**) are signed integers. This means that they can represent negative values as well as positive ones. Java uses an encoding known as *two's complement,* which means that negative numbers are represented by inverting (changing 1's to 0's and vice versa) all of the bits in a value, then adding 1 to the result.

The Bitwise Logical Operators

The bitwise logical operators are **&**, |, ^, and **~**. The following table shows the outcome of each operation. In the discussion that follows, keep in mind that the bitwise operators are applied to each individual bit within each operand.

A	B	A | B	A & B	A ^ B	~A
0	0	0	0	0	1
1	0	1	0	1	0
0	1	1	0	1	1
1	1	1	1	0	0

Relational Operators

The *relational operators* determine the relationship that one operand has to the other. Specifically, they determine equality and ordering. The relational operators are shown here:

Operator	Result
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

The outcome of these operations is a **boolean** value. The relational operators are most frequently used in the expressions that control the **if** statement and the various loop statements.

Boolean Logical Operators

Operator	Result
&	Logical AND
|	Logical OR
^	Logical XOR (exclusive OR)
||	Short-circuit OR
&&	Short-circuit AND
!	Logical unary NOT
&=	AND assignment
|=	OR assignment
^=	XOR assignment
==	Equal to
!=	Not equal to
?:	Ternary if-then-else

The logical Boolean operators, **&**, |, and ^, operate on **boolean** values in the same way that they operate on the bits of an integer. The logical **!** operator inverts the Boolean state: **!true == false** and **!false == true**. The following table shows the effect of each logical operation:

A	B	A | B	A & B	A ^ B	!A
False	False	False	False	False	True
True	False	True	False	True	False
False	True	True	False	True	True
True	True	True	True	False	False

The ? Operator

Java includes a special *ternary* (three-way) *operator* that can replace certain types of if-then-else statements. This operator is the **?**. It can seem somewhat confusing at first, but the **?** can be used very effectively once mastered. The **?** has this general form:
expression1 **?** *expression2* **:** *expression3*

Here, *expression1* can be any expression that evaluates to a **boolean** value. If *expression1* is **true**, then *expression2* is evaluated; otherwise, *expression3* is evaluated. The result of the **?** operation is that of the expression evaluated. Both *expression2* and *expression3* are required to return the same type, which can't be **void**.
Here is an example of the way that the **?** is employed:
ratio = denom == 0 ? 0 : num / denom;

Operator Precedence

Table shows the order of precedence for Java operators, from highest to lowest. The first row shows items that you may not normally think of as operators: parentheses, square brackets, and the dot operator. Technically, these are called *separators*, but they act like operators in an expression. Parentheses are used to alter the precedence of an operation. As you know from the previous chapter, the square brackets provide array indexing. The dot operator is used to dereference objects.

Highest			
( )	[ ]	.	
++	– –	~	!
*	/	%	
+	–		
>>	>>>	<<	
>	>=	<	<=
==	!=		
&			
^			
|			
&&			
||			
?:			
=	op=		
Lowest			

5. **Explain about control statements.**

Java's program control statements can be put into the following categories:

1. Selection
2. Iteration
3. jump

Selection statements allow your program to choose different paths of execution based upon the outcome of an expression or the state of a variable. *Iteration* statements enable program execution to repeat one or more statements (that is, iteration statements form loops). *Jump* statements allow your program to execute in a nonlinear fashion.

Java's Selection Statements

Java supports two selection statements:

if and **switch**.

If:

the general form of the **if**
statement:
if (*condition*) *statement1*;
else *statement2*;
Here, each *statement* may be a single statement or a compound statement enclosed in curly braces (that is, a *block*). The *condition* is any expression that returns a **boolean** value. The **else** clause is optional.

Nested ifs

A *nested* **if** is an **if** statement that is the target of another **if** or **else**. Nested **if**s are very common in programming. When you nest **if**s, the main thing to remember is that an **else** statement always refers to the nearest **if** statement that is within the same block as the **else** and that is not already associated with an **else**.

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

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

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

switch

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

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


// statement sequence
break;
default:
// default statement sequence
}
```

Iteration Statements

Java's iteration statements are **for**, **while**, and **do-while**. These statements create what we commonly call *loops.* As you probably know, a loop repeatedly executes the same set of instructions until a termination condition is met. As you will see, Java has a loop to fit any programming need.

While

The **while** loop is Java's most fundamental loop statement. It repeats a statement or block while its controlling expression is true. Here is its general form:

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

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

do-while

The **do-while** loop always executes its body at least once, because its conditional expression is at the bottom of the loop. Its general form is

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

Each iteration of the **do-while** loop first executes the body of the loop and then evaluates the conditional expression. If this expression is true, the loop will repeat. Otherwise, the loop terminates. As with all of Java's loops, *condition* must be a Boolean expression.
Here is a reworked version of the "tick" program that demonstrates the **do-while** loop.
It generates the same output as before.

For:

Here is the general form of the traditional **for** statement:
for(*initialization*; *condition*; *iteration*) {
// body
}
If only one statement is being repeated, there is no need for the curly braces.
The **for** loop operates as follows. When the loop first starts, the *initialization* portion of the loop is executed. Generally, this is an expression that sets the value of the *loop control variable*, which acts as a counter that controls the loop. It is important to understand that the initialization expression is only executed once. Next, *condition* is evaluated. This must be a Boolean expression. It usually tests the loop control variable against a target value. If this expression is true, then the body of the loop is executed. If it is false, the loop terminates.
Next, the *iteration* portion of the loop is executed. This is usually an expression that increments or decrements the loop control variable. The loop then iterates, first evaluating the conditional expression, then executing the body of the loop, and then executing the iteration expression with each pass. This process repeats until the controlling expression is false.

Jump Statements
Java supports three jump statements: **break**, **continue**, and **return**. These statements transfer control to another part of your program.

Using break
In Java, the **break** statement has three uses. First, as you have seen, it terminates a statement sequence in a **switch** statement. Second, it can be used to exit a loop. Third, it can be used as a "civilized" form of goto.

Using break to Exit a Loop
By using **break**, you can force immediate termination of a loop, bypassing the conditional expression and any remaining code in the body of the loop. When a **break** statement is encountered inside a loop, the loop is terminated and program control resumes at the next statement following the loop

Using continue
Sometimes it is useful to force an early iteration of a loop. That is, you might want to continue running the loop but stop processing the remainder of the code in its body for this particular iteration. This is, in effect, a goto just past the body of the loop, to the loop's end. The **continue** statement performs such an action. In **while** and **do-while** loops, a **continue** statement causes control to be transferred directly to the conditional expression that controls the loop.
return
The last control statement is **return**. The **return** statement is used to explicitly return from a method. That is, it causes program control to transfer back to the caller of the method. As such, it is categorized as a jump statement.

6. **Explain about class, object, method and constructors in java.**

(or)

Explain about constructor overloading.

Class:

A *class* defines the structure and behaviour (data and code) that will be shared by a set of objects. Each object of a given class contains the structure and behaviour defined by the class

Object:

Object is an instance of a class. **Objects** have states and behaviours.

Method:

A **method** is a block of code or collection of statements or a set of code grouped together to perform a certain task or operation. It is used to achieve the **reusability** of code.

Constructors:

A constructor is a block of code. Constructor is called when an instance of the object is created, and memory is allocated for the object.

Constructor overloading is a technique that enables a single class to have more than one constructor that varies by the list of arguments passed.

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

Type of constructor:

1. Default constructor
2. Parameterized constructor

Program:

```
class Box
 {
double width;
double height;
double depth;
Box()
{
width = 5;
height = 4;
depth = 3;
}

 Box(double w, double h, double d)
{
width = w;
height = h;
depth = d;
}

  double volume()
 {
return width * height * depth;
}
```

```
}
class BoxDemo
 {
public static void main(String args[])
{
 Box b1 = new Box();
Box b2 = new Box(3, 6, 9);
double vol;
 vol = b1.volume();
  System.out.println("Volume of box1 is " + vol);
vol = b2.volume();
System.out.println("Volume of box2 is " + vol);
}
}
```

7. Explain about access control or access specifiers.

- Encapsulation links data with the code that manipulates it. However, encapsulation provides another important attribute: *access control*. Through encapsulation, you can control what parts of a program can access the members of a class.
- By controlling access, you can prevent misuse.
- Java supplies a rich set of access modifiers.

- Java's access modifiers are **public**, **private**, and **protected**. Java also defines a default access level.

	Private	**No Modifier (Default)**	**Protected**	**Public**
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

1) **protected** applies only when inheritance is involved.
2) When a member of a class is modified by **public**, then that member can be accessed by any other code.
3) When a member of a class i**s** specified as **private**, then that member can only be accessed by other members of its class.
4) When no access modifier is used, then **by default** the member of a class is public within its own package, but cannot be accessed outside of its package.

8. Explain about static members (static variable,static method,static block).

- When a member is declared **static**, it can be accessed before any objects of its class are created, and without reference to any object. You can declare both methods and variables to be **static**. The most common example of a **static** member is **main( )**. **main( )** is declared as **static** because it must be called before any objects exist.

1) Static variable:

- Instance variables declared as **static** are, essentially, global variables. When objects of its class are declared, no copy of a **static** variable is made. Instead, all instances of the class share the same **static** variable.

2) Static method:

- Methods declared as **static** have several restrictions:
  - They can only directly call other **static** methods.
  - They can only directly access **static** data.
  - They cannot refer to **this** or **super** in any way. (The keyword **super** relates to inheritance and is described in the next chapter.)

3) Static block:

- If you need to do computation in order to initialize your **static** variables, you can declare a **static** block that gets executed exactly once, when the class is first loaded. The following example shows a class that has a **static** method, some **static** variables, and a **static** initialization block:

Demonstrate static variables, methods, and blocks.

```
class student
 {
     int a;
     static int b;
     student(){
          b++;
     }
     static
     {
     System.out.println("First static block");
     }
     static
     {
     System.out.println("Second static block");
     }
     public void showData()
     {
      System.out.println("Value of a = "+a);
          System.out.println("Value of b = "+b);
     }

     static void display()
     {
     System.out.println("This is static method");
     }
}
public class studemo
{
   public static void main(String args[]){
```

```
    student s1 = new student();
    s1.showData();
    student s2 = new student();
    s2.showData();
    student.display();
  }
}
```

output:

```
First static block
Second static block
Value of a =0
Value of b =1
Value of a =0
Value of b =2
This is static method
```

9. Explain about javadoc comments.

Java supports three types of comments. The first two are the // and the /* */. The third type is called a *documentation comment.* It begins with the character sequence /**. It ends with */. Documentation comments allow you to embed information about your program into the program itself. You can then use the **javadoc** utility program (supplied with the JDK) to extract the information and put it into an HTML file. Documentation comments make it convenient to document your programs. You have almost certainly seen documentation that uses such comments because that is the way the Java API ibrary was documented. Beginning with JDK 9, **javadoc** includes support for modules.

The javadoc Tags

The **javadoc** utility recognizes several tags, including those shown here:

Tag	Meaning
@author	Identifies the author.
{@code}	Displays information as-is, without processing HTML styles, in code font.
@deprecated	Specifies that a program element is deprecated.
{@docRoot}	Specifies the path to the root directory of the current documentation.
@exception	Identifies an exception thrown by a method or constructor.
@hidden	Prevents an element from appearing in the documentation.
{@index}	Specifies a term for indexing.
{@inheritDoc}	Inherits a comment from the immediate superclass.
{@link}	Inserts an in-line link to another topic.
{@linkplain}	Inserts an in-line link to another topic, but the link is displayed in a plain-text font.
{@literal}	Displays information as-is, without processing HTML styles.
@param	Documents a parameter.

Tag	Meaning
@provides	Documents a service provided by a module.
@return	Documents a method's return value.
@see	Specifies a link to another topic.
@serial	Documents a default serializable field.
@serialData	Documents the data written by the **writeObject( )** or **writeExternal( )** methods.
@serialField	Documents an **ObjectStreamField** component.
@since	States the release when a specific change was introduced.
{@summary}	Documents a summary of an item. (Added by JDK 10.)
@throws	Same as **@exception**.
@uses	Documents a service needed by a module.
{@value}	Displays the value of a constant, which must be a **static** field.
@version	Specifies the version of a program element.

The General Form of a Documentation Comment

After the beginning **/****, the first line or lines become the main description of your class, interface, field, constructor, method, or module. After that, you can include one or more of the various **@** tags. Each **@** tag must start at the beginning of a new line or follow one or more asterisks (*****) that are at the start of a line. Multiple tags of the same type should be grouped together. For example, if you have three **@see** tags, put them one after the other. In-line tags (those that begin with a brace) can be used within any description. Here is an example of a documentation comment for a class:

javadoc Outputs

The **javadoc** program takes as input your Java program's source file and outputs several HTML files that contain the program's documentation. Information about each class will be in its own HTML file. **javadoc** will also output an index and a hierarchy tree. Other HTML files can be generated. Beginning with JDK 9, a search box feature is also included.

```
/**
* The AddNum program implements an application that
* simply adds two given integer numbers and Prints
* the output on the screen.
* @author  John
* @version 1.0
* @since   30-06-2019
*/

public class addnum {
   /**
   * This method is used to add two integers. This is
   * a the simplest form of a class method, just to
   * show the usage of various javadoc Tags.
   * @param numA This is the first paramter to addNum method
   * @param numB  This is the second parameter to addNum method
   * @return int This returns sum of numA and numB.
   */
```

```
  public int addNum(int numA, int numB) {
    return numA + numB;
  }

  /**
  * This is the main method which makes use of addNum method.
  * @param args Unused.
  * @return Nothing.
  */

  public static void main(String args[])
  {
    addnum obj = new addnum();
    int sum = obj.addNum(10, 20);
    System.out.println("Sum of 10 and 20 is :" + sum);
  }
}
```

Compile:
Z:\>javac addnum.java
Html file creation:
Z:\>javadoc addnum.java
Html file will be created in the same directory.

12. Develop program to sort numbers in ascending order.

```
class GFG {
    public static void main(String[] args)
    {
        int arr[] = { 4, 3, 2, 1 };
        for (int i = 0; i < arr.length; i++) {
            for (int j = i + 1; j < arr.length; j++) {
                int temp = 0;
                if (arr[j] < arr[i]) {
                    temp = arr[i];
                    arr[i] = arr[j];
                    arr[j] = temp;
                }
            }
            System.out.print(arr[i] + " ");
        }
    }
}
```

Output

```
1 2 3 4
```

CS8392 OBJECT ORIENTED PROGRAMMING
UNIT- I

Part-A

1. **Differentiate between Process-oriented programming and Object-oriented programming**

1) Process-oriented model :

Process-oriented model approach characterizes a program as a series of linear steps. The process-oriented model can be thought of as *code acting on data*. Procedural languages such as C employ this model to considerable success. Problems with this approach appear as programs grow larger and more complex.

2) Object-oriented programming :

To manage increasing complexity *object-oriented programming* was conceived. Object-oriented programming organizes a program around its data (that is, objects) and a set of well-defined interfaces to that data. An object-oriented program can be characterized as *data controlling access to code*.

2. **Define abstraction:**

Abstraction is a process where you show only "relevant" data and "hide" unnecessary details of an object from the user. For example, when you login to your Amazon account online, you enter your user_id and password and press login, what happens when you press login, how the input data sent to amazon server, how it gets verified is all abstracted away from the you.

3. **Define encapsulation.**

Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse.

4. **What is classes and an objects?**

A class defines the structure and behavior (data and code) that will be shared by a set of objects. Each object of a given class contains the structure and behavior defined by the class, as if it were stamped out by a mold in the shape of the class. For this reason, objects are sometimes referred to as *instances of a class*. Thus, a class is a logical construct; an object has physical reality.

5. **Define inheritance.**

Inheritance is the process by which one object acquires the properties of another object. This is important because it supports the concept of hierarchical classification. Most knowledge is made manageable by hierarchical (that is, top-down) classifications.

6. **What is meant by polymorphism? Specify its type.**

Polymorphism is the concept where an object behaves differently in different situations. There are two types of polymorphism – compile time polymorphism and runtime polymorphism. Polymorphism could be static and dynamic both. Method Overloading is static polymorphism while, Method overriding is dynamic polymorphism.

7. **Differentiate between method Overloading and method Overriding**

Method Overloading means more than one method having the same method name that behaves differently based on the arguments passed while calling the method. This called static because, which method to be invoked is decided at the time of compilation. **Method Overriding** means a derived class is implementing a method of its super class. The call to overridden method is resolved at runtime, thus called runtime polymorphism

8. List the java buzzwords. (Or) List the characteristics of java.

1) Simple
2) Secure
3) Portable
4) Object-oriented
5) Robust
6) Multithreaded
7) Architecture-neutral
8) Interpreted
9) High performance
10) Distributed
11) Dynamic

9. What is the use of constructor?

A *constructor* initializes an object immediately upon creation. It has the same name as the class in which it resides and is syntactically similar to a method. Once defined, the constructor is automatically called when the object is created, before the **new** operator completes.

Constructors look a little strange because they have no return type, not even **void**. This is because the implicit return type of a class' constructor is the class type itself.

Java's access modifiers are **public**, **private**, and **protected**. Java also defines a default access level. **protected** applies only when inheritance is involved.

10. State The use of this Keyword.

Sometimes a method will need to refer to the object that invoked it. To allow this, Java defines the **this** keyword. **this** can be used inside any method to refer to the *current* object. That is, **this** is always a reference to the object on which the method was invoked.

11. What is done in garbage collection?

When no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed by the garbage collection. Garbage collection only occurs at irregular intervals during the execution of your program.

12. What is the use of finalize method?

Sometimes an object will need to perform some action when it is destroyed. For example, if an object is holding some non-Java resource such as a file handle or character font, then you might want to make sure these resources are freed before an object is destroyed. To handle such situations, Java provides a mechanism called *finalization*. By using finalization, you can definespecific actions that will occur when an object is just about to be reclaimed by the garbage collector.

To add a finalizer to a class, you simply define the **finalize( )** method. The Java run time calls that method whenever it is about to recycle an object of that class. Inside the **finalize( )** method, you will specify those actions that must be performed before an object is destroyed.

13. Specify various access control supported by java.

Encapsulation provides another important attribute: *access control*. Through encapsulation, you can control what parts of a program can access the members of a class.

By controlling access, you can prevent misuse. For example, allowing access to data only through a well-defined set of methods, you can prevent the misuse of that data.

How a member can be accessed is determined by the *access modifier* attached to its declaration.

Java's access modifiers are **public**, **private**, and **protected**. Java also defines a default access level. **protected** applies only when inheritance is involved.

1) **Public:** When a member of a class is modified by **public**, then that member can be accessed by any other code.

2) **Private:** When a member of a class is specified as **private**, then that member can only be accessed by other members of its class.

3) **Protected:** protected members can be accessed by the same package and its derived classes.

4) **Default:** When no access modifier is used, then by default the member of a class is public within its own package, but cannot be accessed outside of its package.

14. State the situation where static members(static variable,static method, and static are used.

There will be times when you will want to define a class member that will be used independently of any object of that class. Normally, a class member must be accessed only in conjunction with an object of its class. However, it is possible to create a member that can be used by itself, without reference to a specific instance. To create such a member, precede its declaration with the keyword **static**.

When a member is declared **static**, it can be accessed before any objects of its class are created, and without reference to any object. You can declare both methods and variables to be **static**. The most common example of a **static** member is **main( )**. **main( )** is declared as **static** because it must be called before any objects exist.

Instance variables declared as **static** are, essentially, global variables. When objects of its class are declared, no copy of a **static** variable is made. Instead, all instances of the class share the same **static** variable.

Methods declared as **static** have several restrictions:

They can only directly call other **static** methods.

They can only directly access **static** data.

They cannot refer to **this** or **super** in any way.

If you need to do computation in order to initialize your **static** variables, you can declare a **static** block that gets executed exactly once, when the class is first loaded.

15. State the use of final keyword.

A field can be declared as **final**. Doing so prevents its contents from being modified, making it, essentially, a constant. This means that you must initialize a **final** field when it is declared.

Example:

final int FILE_NEW = 1;

16. **What is the use of break and continue statement?**
    The break statement is used to terminate the loop immediately. The continue statement is used to skip the current iteration of the loop.

Part-B

1. Discuss various features of OOP.
2. Elaborate the about various Java buzzwords.
3. Elaborate about various categories of operators in java.
4. Explain about one dimensional and two dimensional arrays in java with example programs.

5. Discuss the use of constructor and finalize() method in java with programming example. Show how garbage collection is achieved here.
6. Explain about class, method and constructor overloading in java.
7. Explain about selection statements in java.
8. Explain about various literals in java. (or) Explain about various datatypes in java.

9. Explain about the control statements in java with example programs.
10. Illustrate method overloading with example program.
11. Explain static field and static method with example.
12. Discuss about javaDoc. Explain the comments for classes, methods, fields.

13. Create a class to print the area of a square and a rectangle. The class has two methods with the same name but different number of parameters. The method for printing area of rectangle has two parameters which are length and breadth respectively while the other method for printing area of square has one parameter which is side of square.

14. Develop a program by creating an 'Employee' class having the following methods and print the final salary.

    'getInfo()' which takes the salary, number of hours of work per day of employee as parameter.
    'AddSal()' which adds Rs.100 to salary of the employee if it is less than Rs.5000.
    'AddWork()' which adds Rs.50 to salary of employee if the number of hours of work per day is more than 6 hours.

12. Suppose you have a Piggie Bank with an initial amount of Rs.500 and you have to add some more amount to it. Create a class 'AddAmount' with a data member named 'amount' with an initial value of Rs.500. Now make two constructors of this class as follows: 1 - without any parameter - no amount will be added to the Piggie Bank 2 - having a parameter which is the amount that will be added to Piggie Bank Create object of the 'AddAmount' class and display the final amount in Piggie Bank.

13. Develop a java program to find smallest number in an array.
14. Develop a program to print the area of a triangle by creating a class named 'Area' taking the values of its Breadth and Height as parameters of its constructor and having a

method named 'returnArea' which returns the area of the triangle. Breadth and Height of triangle are entered through keyboard.

15. Develop a java program that take 20 integer inputs from user and print the following: number of positive numbers

    number of negative
    numbers number of odd
    numbers number of even
    numbers number of 0.

16. Create a class named 'Programming'. While creating an object of the class, if nothing is passed to it, then the message "I like programming languages" should be printed. If some String is passed to it, then in place of "programming languages" the name of that String variable should be printed. For example, while creating object if we pass "Java", then "I like Java" should be printed.
17. Write a java program to find whether the number is odd or even numbers in an array.
18. Write a java program to perform the following functions using classes, objects, constructors and destructors where essential.
    i)Get as input the marks of 5 students in 5 subjects
    ii) Calculate the total and average
    iii)Print the formatted results on the screen
19. Develop a mini project for Mark sheet Preparation system using Java concepts.

20. Write a Java program that prompts the user for an integer and then prints out all prime numbers up to that Integer.

21. Write a Java Program that reads a line of integers, and then displays each integer, and the sum of all the integers

UNIT II INHERITANCE, PACKAGES AND INTERFACES

Overloading Methods – Objects as Parameters – Returning Objects –Static, Nested and Inner Classes. Inheritance: Basics– Types of Inheritance -Super keyword -Method Overriding – Dynamic Method Dispatch –Abstract Classes – final with Inheritance. Packages and Interfaces: Packages – Packages and Member Access –Importing Packages – Interfaces.

---

1. **Explain about overloading methods.**

Method Overloading is a feature that allows a class to have multiple methods with the same name but with different number of parameter and different type of parameters.

Method overloading is one of the ways that Java supports polymorphism.

When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call. Thus, overloaded methods must differ in the type and/or number of their parameters. While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method. When Java encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call.

Example:

```
class OverloadDemo
 {
void test()
 {
System.out.println("No parameters");
}
void test(int a)
{
System.out.println("a= " + a);
}

void test(int a, int b)
{
System.out.println("a and b: " + a + " " + b);
}
double test(double a)
 {
System.out.println("double a: " + a);
return a*a;
}
}

class Overload
{
public static void main(String args[])
{
OverloadDemo ob = new OverloadDemo();
double result;
```

```
ob.test();
ob.test(10);
ob.test(10, 20);
result = ob.test(123.25);
System.out.println("Result of ob.test(123.25): " + result);
}
}
```

This program generates the following output:

```
No parameters
a: 10
a and b: 10 20
double a: 123.25
Result of ob.test(123.25): 15190.5625
```

2. **Explain about overloading constructors or constructor overloading.**

Class:

A *class* defines the structure and behaviour (data and code) that will be shared by a set of objects. Each object of a given class contains the structure and behaviour defined by the class

Object:

Object is an instance of a class. *Objects* have states and behaviours.

Method:

A **method** is a block of code or collection of statements or a set of code grouped together to perform a certain task or operation. It is used to achieve the **reusability** of code.

Constructors:

A constructor is a block of code. Constructor is called when an instance of the object is created, and memory is allocated for the object.

Constructor overloading is a technique that enables a single class to have more than one constructor that varies by the list of arguments passed.

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

Type of constructor:

3. Default constructor
4. Parameterized constructor

```
class Box
{
double width;
double height;
double depth;
Box(double w, double h, double d)
{
width = w;
height = h;
depth = d;
}
Box()
```

```
{
width = -1;
height = -1;
depth = -1;
}
// constructor used when cube is created
Box(double len)
{
width = height = depth = len;
}
double volume()
 {
return width * height * depth;
}
}
class OverloadCons
 {
public static void main(String args[])
{

Box mybox1 = new Box(10, 20, 15);
Box mybox2 = new Box();
Box mycube = new Box(7);
double vol;
vol = mybox1.volume();
System.out.println("Volume of mybox1 is " + vol);
vol = mybox2.volume();
System.out.println("Volume of mybox2 is " + vol);
vol = mycube.volume();
System.out.println("Volume of mycube is " + vol);
}
}
```

Output:

```
Volume of mybox1 is 3000.0
Volume of mybox2 is -1.0
Volume of mycube is 343.0
```

5. Explain about using objects as parameters.

Objects can be passed as parameter to methods and constructors..
For example, the following version of **Box** allows one object to initialize another:

```
class Box
{
double width;
double height;
double depth;
```

```
Box(Box ob)
{
width = ob.width;
height = ob.height;
depth = ob.depth;
}
Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
Box() {
width = -1;
height = -1;
depth = -1;
}
Box(double len) {
width = height = depth = len;
}
double volume()
 {
return width * height * depth;
}
}
class OverloadCons2
 {
public static void main(String args[])
{
Box mybox1 = new Box(10, 20, 15);
Box mybox2 = new Box();
Box mycube = new Box(7);
Box myclone = new Box(mybox1); // create copy of mybox1
double vol;
vol = mybox1.volume();
System.out.println("Volume of mybox1 is " + vol);
vol = mybox2.volume();
System.out.println("Volume of mybox2 is " + vol);
vol = mycube.volume();
System.out.println("Volume of cube is " + vol);
vol = myclone.volume();
System.out.println("Volume of clone is " + vol);
}
}
```

6. Explain about returning objects.

A method can return any type of data, including class types that you create. For example, in the following program, the **increment( )** method returns an object in which the value of **a** is ten greater than it is in the invoking object.

```
class Test
```

```
{
int a;
Test(int i)
 {
a = i;   }
Test increment( )
{
Test temp = new Test(a+10);
return temp;
}
}
class testdemo
 {
public static void main(String args[])
{
Test ob1 = new Test(2);
Test ob2;
ob2 = ob1.increment();
System.out.println("ob1.a: " + ob1.a);
System.out.println("ob2.a: " + ob2.a);
ob2 = ob2.increment();
System.out.println("ob2.a after second increase: "+ ob2.a);
}
}
```

Output:

```
ob1.a: 2
ob2.a: 12
ob2.a after second increase: 22
```

7. Explain about Nested and Inner Classes.

Defining a class within another class is known as *nested classes*.
The scope of a nested class is bounded by the scope of its enclosing class. Thus, if class B is defined within class A, then B does not exist independently of A. A nested class has access to the members, including private members, of the class in which it is nested. However, the enclosing class does not have access to the members of the nested class. A nested class that is declared directly within its enclosing class scope is a member of its enclosing class. It is also possible to declare a nested class that is local to a block.
There are two types of nested classes:

1) *static*
2) *non-static*.

STATIC NESTED CLASSES:
A static nested class is one that has the **static** modifier applied. Because it is static, it must access the members of its enclosing class through an object. That is, it cannot refer to members of its enclosing class directly. Because of this restriction, static nested classes are rarely used.
NON-STATIC NESTED CLASSES:

An inner class is a non-static nested class. It has access to all of the variables and methods of its outer class and may refer to them directly in the same way that other non-static members of the outer class do.
The following program illustrates how to define and use an inner class. The class named **Outer** has one instance variable named **outer_x**, one instance method named **test( )**, and defines one inner class called **Inner**.

```
class Outer {
int x = 100;
void test() {
Inner in= new Inner();
in.display();
}
// this is an inner class
class Inner {
void display() {
System.out.println("display: outer_x = " + x);
}
}
}
class InnerClassDemo {
public static void main(String args[]) {
Outer out = new Outer();
out.test();
}
}
```

Output from this application is shown here:

```
display: outer_x = 100
```

In the program, an inner class named **Inner** is defined within the scope of class **Outer**. Therefore, any code in class **Inner** can directly access the variable **outer_x**. An instance method named **display( )** is defined inside **Inner**. This method displays **outer_x** on the standard output stream. The **main( )** method of **InnerClassDemo** creates an instance of class **Outer** and invokes its **test( )** method. That method creates an instance of class **Inner** and the **display( )** method is called.
It is important to realize that an instance of **Inner** can be created only within the scope of class **Outer**. The Java compiler generates an error message if any code outside of class **Outer** attempts to instantiate class **Inner**. (In general, an inner class instance must be created by an enclosing scope.) You can, however, create an instance of **Inner** outside of **Outer** by qualifying its name with **Outer**, as in **Outer.Inner**.

8. Explain types of inheritance in java.

- Inheritance is the mechanism in java by which one class is allow to inherit the features(fields and methods) of another class.
- Inheritance allows the creation of hierarchical classifications.
- Using inheritance, you can create a general class that defines behavior common to a set of related items. This class can then be inherited by other, more specific classes, each adding those things that are unique to it.

Superclass: A class that is inherited is called a *superclass*.

Subclass: The class that does the inheriting is called a *subclass*. Therefore, a subclass is a specialized version of a superclass. It inherits all of the members defined by the superclass and adds its own, unique elements.

Inheritance Basics:

- To inherit a class, you simply incorporate the definition of one class into another by using the **extends** keyword.
- Being a superclass for a subclass does not mean that the superclass cannot be used by itself. Further, a subclass can be a superclass for another subclass.

The general form of a **class** declaration that inherits a superclass is shown here:

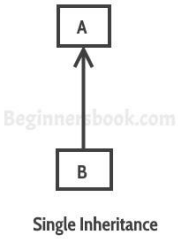
class *subclass-name* extends *superclass-name*
{
// body of class
}

You can only specify one superclass for any subclass that you create. Java does not support the inheritance of multiple superclasses into a single subclass. You can create a hierarchy of inheritance in which a subclass becomes a superclass of another subclass. However, no class can be a superclass of itself.

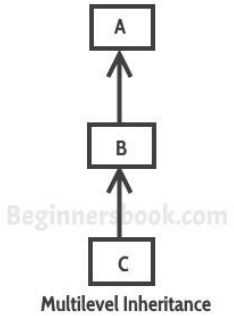
A reference variable of a superclass can be assigned a reference to any subclass derived from that superclass.

TYPES OF INHERITANCE IN JAVA:

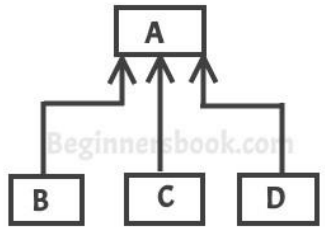
1) Single Inheritance: refers to a sub and super class relationship where a class extends the another class.



2) Multilevel inheritance: refers to a sub and super class relationship where a class extends the subclass. For example class C extends class B and class B extends class A.

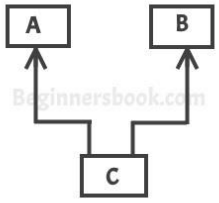


3) Hierarchical inheritance: refers to a sub and super class relationship where more than one classes extends the same class. For example, classes B, C & D extends the same class A.



Hierarchical Inheritance

4) Multiple Inheritance: refers to the concept of one class extending more than one classes, which means a sub class has two super classes. For example class C extends both classes A and B**. Java doesn't support multiple inheritance.**



Multiple Inheritance

MEMBER ACCESS AND INHERITANCE:

Although a subclass includes all of the members of its superclass, it cannot access those members of the superclass that have been declared as **private**.

Protected Member:

The private members of a class cannot be directly accessed outside the class. Only methods of that class can access the private members directly. Sometimes it may be necessary for a subclass to access a private member of a superclass. If you make a private member public, then anyone can access that member. So, if a member of a superclass needs to be accessed in a subclass and yet still prevent its direct access outside the class, you must declare that member **protected**.

Following table describes the difference

Modifier	**Class**	**Subclass**	**World**
Public	Y	Y	Y
Protected	Y	Y	N
Private	Y	N	N

Following program illustrates how the methods of a subclass can directly access a protected member of the superclass.

1) Single inheritance:

When a class inherits another class, it is known as a *single inheritance*.

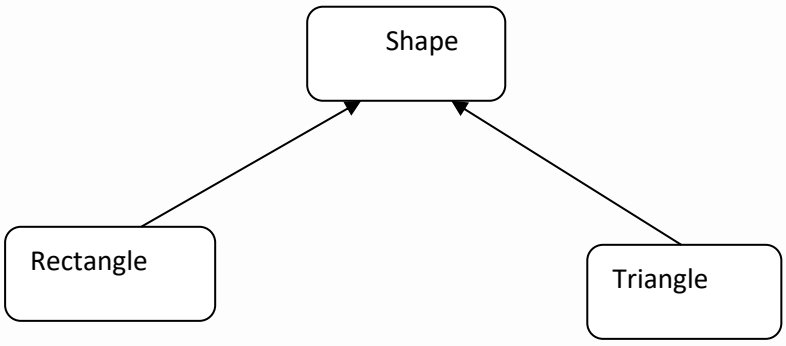
Example: A class Shapes is superclass and derive the class : Rectangle

```
class shape
{
   protected double length;
   protected double breadth;
  shape (double l, double b)
   {
      length = l;
      breadth = b;
   }
}
class rectangle extends shape
{
rectangle(double l,double b)
{
super(l,b);
}


double area()
{
return length*breadth;
}
}

class shapedemo1
{
public static void main(String args[])
{
rectangle r=new rectangle(5,6);
 System.out.println("Area of rectangle : " +r.area());
}
}
```

Output:
Area of rectangle : 30.0

2) Hierarchical inheritance:

When more than one classes inherit a same class then this is called hierarchical inheritance.



Example: A class Shapes is superclass and derive the two classes : Rectangle and Triangle

```
class shape
{
   protected double length;
   protected double breadth;
  shape (double l, double b)
   {
      length = l;
      breadth = b;
   }
}
class rectangle extends shape
{
rectangle(double l,double b)
{
super(l,b);
}


double area()
{
return length*breadth;
}
}

class triangle extends shape
{
triangle(double l,double b)
{
super(l,b);
}
double area()
{
return 0.5*length*breadth;
}
}
class shapedemo1
{
public static void main(String args[])
{
triangle t=new triangle(3,4);
rectangle r=new rectangle(5,6);
 System.out.println("Area of triangle : " + t.area());
 System.out.println("Area of rectangle : " +r.area());
}
}
```

Output:

```
Area of triangle : 6.0
Area of rectangle : 30.0
```

3) MULTILEVEL INHERITANCE:

When there is a chain of inheritance, it is known as *multilevel inheritance*.

Consider the following program. In it, the subclass **BoxWeight** is used as a superclass to create the subclass called **Shipment**. **Shipment** inherits all of the traits of **BoxWeight** and **Box**, and adds a field called **cost**, which holds the cost of shipping such a parcel.

```
class Box
{
private double width;
private double height;
private double depth;

Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
double volume() {
return width * height * depth;
}
}

class BoxWeight extends Box {
double weight;
BoxWeight(double w, double h, double d, double m) {
 super(w, h, d);
 weight = m;
}
}

class Shipment extends BoxWeight {
 double cost;
Shipment(double w, double h, double d,double m, double c) {
super(w, h, d, m);
 cost = c;
}
}

class shipmentdemo {
public static void main(String args[]) {
Shipment shipment1 =new Shipment(10, 20, 15, 10, 3.41);
double vol;
vol = shipment1.volume();
System.out.println("Volume of shipment1 is " + vol);
```

```
System.out.println("Weight of shipment1 is "+shipment1.weight);
System.out.println("Shipping cost: $" + shipment1.cost);
System.out.println();
}
}
```

Output:

```
Volume of shipment1 is 3000.0
Weight of shipment1 is 10.0
Shipping cost: $3.41
```

Because of inheritance, **Shipment** can make use of the previously defined classes of **Box** and **BoxWeight**, adding only the extra information it needs for its own, specific application. This is part of the value of inheritance; it allows the reuse of code.
This example illustrates one other important point: **super( )** always refers to the constructor in the closest superclass. The **super( )** in **Shipment** calls the constructor in **BoxWeight**. The **super( )** in **BoxWeight** calls the constructor in **Box**. In a class hierarchy, if a superclass constructor requires parameters, then all subclasses must pass those parameters "up the line." This is true whether or not a subclass needs parameters of its own.

4) MULTIPLE INHERITANCE

Java does not support Multiple inheritances

9. Explain the uses of super keyword.

Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword **super**.
super has two general forms.

1) The first calls the superclass' constructor.
2) The second is used to access a member of the superclass that has been hidden by a member of a subclass.

1) USING SUPER TO CALL SUPERCLASS CONSTRUCTORS:

A subclass can call a constructor defined by its superclass by use of the following form of **super**:

super(*arg-list*);

Here, *arg-list* specifies any arguments needed by the constructor in the superclass. **super( )** must always be the first statement executed inside a subclass' constructor.

Example:

```
class Box
{
   double width;
    double height;
   double depth;
    Box(double w, double h, double d)
    {
```

```
        width = w;
        height = h;
        depth = d;
      }
      double volume()
      {
        return width * height * depth;
      }
    }
class BoxWeight extends Box
{
double weight;
  BoxWeight(double w, double h, double d, double m)
  {
    super(w,h,d);
    weight = m;
  }
}
class DemoBoxWeight
{
  public static void main(String args[])
  {
    BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3); double
    vol;
    vol = mybox1.volume();
    System.out.println("Volume of mybox1 is " + vol);
    System.out.println("Weight of mybox1 is " + mybox1.weight);
  }
}
```

Output:
Volume of mybox1 is 3000.0
Weight of mybox1 is 34.3

2) A SECOND USE FOR SUPER:

The second form of **super** acts somewhat like **this**, except that it always refers to the superclass of the subclass in which it is used. This usage has the following general form:

super.*member*
Here, *member* can be either a method or an instance variable.

This second form of **super** is most applicable to situations in which member names of a subclass hide members by the same name in the superclass. Consider this simple class hierarchy:
Using super to overcome name hiding.

```
class A
{
int i;
}
class B extends A
```

```
{
int i; // this i hides the i in A
B(int a, int b)
 {
super.i = a;                                    // i in A
i = b;                                           // i in B
}
void show() {
System.out.println("i in superclass: " + super.i);
System.out.println("i in subclass: " + i);
}
}

class UseSuper {
public static void main(String args[]) {
B subOb = new B(1, 2);
subOb.show();
}
}
```

This program displays the following:
i in superclass: 1
i in subclass: 2

Although the instance variable **i** in **B** hides the **i** in **A**, **super** allows access to the **i** defined in the superclass. As you will see, **super** can also be used to call methods that are hidden by a subclass.

10. Explain about types of polymorphism.

- Polymorphism is the concept where an object behaves differently in different situations.
- There are two types of polymorphism – compile time polymorphism and runtime polymorphism. Polymorphism could be static and dynamic.
- Method Overloading is static polymorphism while, Method overriding is dynamic polymorphism.
  Polymorphism refers to the ability of a variable, object or function to take on multiple forms.

Types of Polymorphism

1) Static Polymorphism: (or) compile time polymorphism

Polymorphism that is resolved during compiler time is known as static polymorphism. Method overloading can be considered as static polymorphism example. **Method Overloading**: This allows us to have more than one methods with same name in a class that differs in signature.

Example program:

```
class addition
{
void add(int a,int b)
{
```

```
System.out.println("Sum = "+ (a+b));
}
void add(int a,int b,int c)
{
System.out.println("Sum = "+(a+b+c));
}
}

class demo
{
public static void main(String args[])
{
addition a1=new addition();
 addition a2=new addition();
a1.add(4,7);
a3.add(50,30,60);
}
}
```

2) DYNAMIC POLYMORPHISM :

METHOD OVERRIDING:

In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to *override* the method in the superclass. When an overridden method is called from within its subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the superclass will be hidden.

- A superclass reference variable can refer to a subclass object.

DYNAMIC METHOD DISPATCH:

Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time. Dynamic method dispatch is important because this is how Java implements run-time polymorphism.

Overridden methods are another way that Java implements the "one interface, multiple methods" aspect of polymorphism.

```
 class shape
 {
double dim1length;
 double breadth;

shape(double a, double b) {
length = a;
breadth = b;
}
double area() {
System.out.println("Area for shape is undefined.");
 return 0;
}
}
```

```
class Rectangle extends shape {
Rectangle(double a, double b) {
super(a, b);
}
double area() {
return (length * breadth);
}
}

class Triangle extends shape
{
Triangle(double a, double b)
 {
super(a, b);
}
 double area() {
return (length * breadth / 2);
}
}

class shapedemo {
public static void main(String args[]) {
shape s1 = new shape(10, 10);
Rectangle r = new Rectangle(9, 5);
Triangle t = new Triangle(10, 8);
Shape s2;
s2 = s1;
System.out.println("Area is  " + s2.area());
s2 = t;
System.out.println("Area of triangle is  " + s2.area());
s2 = r;
System.out.println("Area  of rectangle is  " + s2.area());
}
}
```

output:

```
Area for shape is undefined.
Area is 0
Area of triangle  is 45
Area of rectangle  is 40
```

11. Explain about abstract classes and abstract methods.

An *abstract class* is a class that is declared `abstract`—it may or may not include abstract methods. Abstract classes cannot be instantiated, but they can be subclassed.

Example:

```
abstract class shape
{
….
}
```

An *abstract method* is a method that is declared without an implementation (without braces, and followed by a semicolon), like this:

```
abstract void area(double x, double y);
```

program:

```
abstract class shape {
double  length;  double
breadth;
shape(double a, double b) {
length = a;
breadth = b;
}
 abstract double area();
}
class Rectangle extends shape
 {
Rectangle(double a, double b)
{
super(a, b);
}
double area()
{
System.out.println("Inside Area for Rectangle.");
 return length * breadth;
}
}

class Triangle extends shape {
Triangle(double a, double b) {
super(a, b);
}
 double area() {
System.out.println("Inside Area for Triangle.");
 return length * breadth / 2;
}
}


class shapedemo
 {
public static void main(String args[])
{
Rectangle r = new Rectangle(9, 5);
Triangle t = new Triangle(10, 8);
shape s2;
s2 = r;
System.out.println("Area is " + s2.area());
s2 = t;
System.out.println("Area is " + s2.area());
```

```
}
}
```

10. Explain the uses of final keyword in inheritance.

The keyword **final** has three uses.

1) final keyword can be used to create the equivalent of a named constant.
   Example: final double PI=3.14;

2) final keyword can be used to prevent overriding
3) final keyword can be used to prevent inheritance

1) final can be used to create the equivalent of a named constant.

Example:

```
final double PI=3.14;
```

2) using final to prevent overriding:

While method overriding is one of Java's most powerful features, there will be times when you will want to prevent it from occurring. To disallow a method from being overridden, specify **final** as a modifier at the start of its declaration. Methods declared as **final** cannot be overridden. The following fragment illustrates **final**:

```
class A {
final void meth() {
System.out.println("This is a final method.");
}
}
class B extends A {
void meth() { // ERROR! Can't override.
System.out.println("Illegal!");
}
}
```

Because **meth( )** is declared as **final**, it cannot be overridden in **B**. If you attempt to do so, a compile-time error will result.

Methods declared as **final** can sometimes provide a performance enhancement: The compiler is free to *inline* calls to them because it "knows" they will not be overridden by a subclass. When a small **final** method is called, often the Java compiler can copy the bytecode for the subroutine directly inline with the compiled code of the calling method, thus eliminating the costly overhead associated with a method call. Inlining is an option only with **final** methods. Normally, Java resolves calls to methods dynamically, at run time. This is called *late binding*. However, since **final** methods cannot be overridden, a call to one can be resolved at compile time. This is called early binding.

3) using final to prevent inheritance:

Sometimes you will want to prevent a class from being inherited. To do this, precede the class declaration with **final**. Declaring a class as **final** implicitly declares all of its methods as **final**, too. As you might expect, it is illegal to declare a class as both **abstract** and **final** since an

abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations.

Here is an example of a **final** class:

```
final class A {
//...
}
// The following class is illegal.

class B extends A { // ERROR! Can't subclass A
//...
}
```

As the comments imply, it is illegal for **B** to inherit **A** since **A** is declared as **final**.

12. Explain about packages in java.

A **java package** is a group of similar types of classes, interfaces and sub-packages. Package in java can be categorized in two form, built-in package and user-defined package. There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

Advantage of Java Package:

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.
2) Java package provides access protection.
3) Java package removes naming collision.

You can define classes inside a package that are not accessible by code outside that package. You can also define class members that are exposed only to other members of the same package. This allows your classes to have intimate knowledge of each other, but not expose that knowledge to the rest of the world.

Defining a Package:

To create a package is quite easy: simply include a **package** command as the first statement in a Java source file. Any classes declared within that file will belong to the specified package. The **package** statement defines a name space in which classes are stored. If you omit the **package** statement, the class names are put into the default package, which has no name. While the default package is fine for short, sample programs, it is inadequate for real applications. Most of the time, you will define a package for your code.

This is the general form of the **package** statement:

package *pkg*;

Here, *pkg* is the name of the package. For example, the following statement creates a package called **MyPackage**:

package MyPackage;

More than one file can include the same **package** statement. The **package** statement simply specifies to which package the classes defined in a file belong. It does not exclude other classes in other files from being part of that same package. Most real-world packages are spread across many files.

Example:

```
package pack;
public class dis
{
public  void display()
{
System.out.println("This is a simple package");
}
```

```
}
```

File name: dis.java
Compile: Z:\>javac –d . dis.java

IMPORTING PACKAGES:

In a Java source file, **import** statements occur immediately following the **package** statement (if it exists) and before any class definitions. This is the general form of the **import** statement:

```
import pkg1:
import pack.dis;
class demo
{
public static void main( String args[])
{
dis d=new dis();
d.display();
}
}
```

File name: demo.java
Compile:
Z:\>javac demo.java
Run:
Z:\>java demo

Output:
This is a simple package

13. What is meant by interface? How it is declared and implemented in java. Give example.

- An interfaces can have methods and variables but the methods declared in interface contain only method signature, not body.
- **An interface in java** is a blueprint of a class. It has static constants and abstract methods.
  The interface in java is **a mechanism to achieve abstraction**.
  - You cannot instantiate an interface.
  - An interface does not contain any constructors.
  - All of the methods in an interface are abstract.
  - An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.
  - An interface is not extended by a class; it is implemented by a class.
  - An interface can extend multiple interfaces.

One interface can extend another.

```
interface A
 {
   void method1();
   void method2();
}
 interface B extends A
 {
   void method3();
}
```

```
class inter implements B
{
   public void method1() {
      System.out.println("Implement method1().");
   }
   public void method2() {
      System.out.println("Implement method2().");
   }
   public void method3() {
      System.out.println("Implement method3().");
   }
}
class interdemo{
   public static void main(String arg[]) {
      inter  ob = new inter();
      ob.method1();
      ob.method2();
      ob.method3();
   }
}
```

Implement multiple interfaces :

Multiple inheritance (`extends`) is not allowed. Interfaces are not classes, however, and a class *can implement more than one interface.*

```
interface Inter1
{
  public void test(int i);
}
interface Inter2
{
  public void test(String s);
}

public class MultInterfaces implements Inter1, Inter2
 {
  public void test(int i)
{
    System.out.println("In MultInterfaces.I1.test");
  }
  public void test(String s)
{
    System.out.println("In MultInterfaces.I2.test");
  }
}
class multidemo
{
  public static void main(String  args[])
{
    MultInterfaces t = new MultInterfaces();
    t.test(42);
    t.test("Hello");
```

```
  }
}
```

CS3391 / OBJECT ORIENTED PROGRAMMING

Unit-2

PART-A

1. What is meant by inheritance?

Inheritance is the process by which one object acquires the properties of another object. Inheritance allows the creation of hierarchical classifications. Using inheritance, you can create a general class that defines behavior common to a set of related items. This class can then be inherited by other, more specific classes, each adding those things that are unique to it.

Superclass: A class that is inherited is called a *superclass*.

Subclass: The class that does the inheriting is called a *subclass*.

2. What is meant by polymorphism? Specify its type.

Polymorphism is the concept where an object behaves differently in different situations. There are two types of polymorphism – compile time polymorphism and runtime polymorphism.

Polymorphism could be static and dynamic both. Method Overloading is static polymorphism while, Method overriding is dynamic polymorphism.

3. Differentiate between method Overloading and method Overriding.

- **Method Overloading** means more than one method having the same method name that behaves differently based on the arguments passed while calling the method. This called static because, which method to be invoked is decided at the time of compilation
- **Method Overriding** means a sub class is implementing a method of its super class. The call to overridden method is resolved at runtime, thus called runtime polymorphism

4. Define Dynamic method dispatch. (run time polymorphism or dynamic polymorphism)

Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time. Dynamic method dispatch is important because this is how Java implements run-time polymorphism.

5. Define method overriding.

In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to *override* the method in the superclass. When an overridden method is called from within its subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the superclass will be hidden.

4. What are the uses of the final keyword?

1) final keyword can be used to create the equivalent of a named constant.

Example:

```
final double PI=3.14;
```

2) final keyword can be used to prevent overriding

3) final keyword can be used to prevent inheritance

5. What is an object class?

Object is a superclass of all other classes. This means that a reference variable of type Object can refer to an object of any other class. Also, since arrays are implemented as classes, a variable of type Object can also refer to any array.

6. What is an abstract class?

An *abstract class* is a class that is declared `abstract`—it may or may not include abstract methods. Abstract classes cannot be instantiated, but they can be subclassed.

An *abstract method* is a method that is declared without an implementation (without braces, and followed by a semicolon), like this:

abstract void area(double x, double y);

7. Define interface and write syntax of interface.

An interface in java is a blueprint of a class. It has static constants and abstract methods.

The interface in java is **a mechanism to achieve abstraction**. There can be only abstract methods in the java interface not method body. It is used to achieve abstraction and multiple inheritance in Java.

In other words, you can say that interfaces can have methods and variables but the methods declared in interface contain only method signature, not body.

Syntax:

```
access interface name {
return-type method-name1(parameter-list);
return-type method-name2(parameter-list);
type final-varname1 = value;
type final-varname2 = value;
//...
return-type method-nameN(parameter-list);
type final-varnameN = value;
}
```

8. Compare classes and interfaces.

BASIS FOR COMPARISON	CLASS	INTERFACE
Basic	A class is instantiated to create objects.	An interface can never be instantiated as the methods are unable to perform any action on invoking.
Keyword	Class	Interface
Access specifier	The members of a class can be private, public or protected.	The members of an interface are always public.
Methods	The methods of a class are defined to perform a specific action.	The methods in an interface are purely abstract.

BASIS FOR COMPARISON	CLASS	INTERFACE
Implement/Extend	A class can implement any number of interface and can extend only one class.	An interface can extend multiple interfaces but can not implement any interface.
Constructor	A class can have constructors to initialize the variables.	An interface can never have a constructor as there is hardly any variable to initialize.

9. **What is an inner class?**

Inner class is a class within another class. such classes are known as *nested classes*.

The scope of a nested class is bounded by the scope of its enclosing class. Thus, if class B is defined within class A, then B does not exist independently of A. A nested class has access to the members, including private members, of the class in which it is nested. However, the enclosing class does not have access to the members of the nested class. A nested class that is declared directly within its enclosing class scope is a member of its enclosing class. It is also possible to declare a nested class that is local to a block.

10. How do you implement multiple inheritance in java?

Java does not allow multiple inheritance:

To reduce the complexity and simplify the language To avoid the ambiguity caused by multiple inheritance

It can be implemented using Interfaces.

11. Define super class and subclass.

The process of deriving a new class from an existing class is inheritance. A class that is inherited is called a *superclass* and the class that does the inheriting is called a *subclass.*

12. State the use of keyword super.

It can be used to refer immediate parent class instance variable when both parent and child class have member with same name

It can be used to invoke immediate parent class method when child class has overridden that method.

super() can be used to invoke immediate parent class constructor.

13. Define abstract class?

Abstract classes are classes from which instances are usually not created. It is basically used to contain common characteristics of its derived classes. Abstract classes generally act as super classes. Methods can also be declared as abstract. This implies that non-abstract classes must implement these methods.

14. When to use abstract Methods & abstract class?

Abstract methods are usually declared where two or more subclasses are expected to do a similar thing in different ways through different implementations. These subclasses extend the same Abstract class and provide different implementations for the abstract methods.

Abstract classes are used to define generic types of behaviors at the top of an object-oriented programming class hierarchy, and use its subclasses to provide implementation details of the abstract class.

15. What's the difference between an interface and an abstract class?

An abstract class may contain code in method bodies, which is not allowed in an interface. With abstract classes, we have to inherit our class from it and Java does not allow multiple inheritance. On the other hand, we can implement multiple interfaces in your class.

16. State the situation where static members(static variable,static method, and static are used.

There will be times when you will want to define a class member that will be used independently of any object of that class. Normally, a class member must be accessed only in conjunction with an object of its class. However, it is possible to create a member that can be used by itself, without reference to a specific instance. To create such a member, precede its declaration with the keyword **static**.

When a member is declared **static**, it can be accessed before any objects of its class are created, and without reference to any object. You can declare both methods and variables to be **static**. The most common example of a **static** member is **main( )**. **main( )** is declared as **static** because it must be called before any objects exist.

Instance variables declared as **static** are, essentially, global variables. When objects of its class are declared, no copy of a **static** variable is made. Instead, all instances of the class share the same **static** variable.

Methods declared as **static** have several restrictions:

They can only directly call other **static** methods.

They can only directly access **static** data.

They cannot refer to **this** or **super** in any way.

If you need to do computation in order to initialize your **static** variables, you can declare a **static** block that gets executed exactly once, when the class is first loaded.

17. What are the uses of package?

A **java package** is a group of similar types of classes, interfaces and sub-packages. Package in java can be categorized in two form, built-in package and user-defined package.There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

Advantage of Java Package:

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.
2) Java package provides access protection.
3) Java package removes naming collision.

18. What are the difference between static variable and instance variable?

Instance variables	Static (class) variables
Instance variables are declared in a class, but outside a method, constructor or any block.	Class variables also known as static variables are declared with the static keyword in a class, but outside a method, constructor or a block.
Instance variables are created when an object is created with the use of the keyword 'new' and destroyed when the object is destroyed.	Static variables are created when the program starts and destroyed when the program stops.
Instance variables can be accessed directly by calling the variable name inside the class. However, within static methods (when instance	Static variables can be accessed by calling with the class name*ClassName.VariableName*.

variables are given accessibility), they should be called using the fully qualified name. *ObjectReference.VariableName.*

Instance variables hold values that must be referenced by more than one method, constructor or block, or essential parts of an object's state that must be present throughout the class.

There would only be one copy of each class variable per class, regardless of how many objects are created from it.

19. What's the difference between constructors and other methods?

Constructors are used to initialize the object's state. Like methods, a constructor also contains **collection of statements(i.e. instructions)** that are executed at time of Object creation. Each time an object is created using **new()** keyword at least one constructor (it could be default constructor) is invoked to assign initial values to the **data members** of the same class.

A method is a collection of statements that perform some specific task and return the result to the caller. A method can perform some specific task without returning anything. Methods allow us to reuse the code without retyping the code. In Java, every method must be part of some class.

Part- B

1. What is inheritance? Write a program for inheriting a class.
2. Explain multilevel inheritance with example program.
3. Can java support multiple inheritances? Illustrate your answer with example java program.
4. What does it mean that a class or method is abstract? Can we make an instance of abstract class? Explain it with example.
5. Explain in detail about constructor overloading with an example program.
6. Explain method overloading with an example program.
7. Discuss about object as parameter with an example program.
8. Discuss about object as return type with an example program.
9. Explain dynamic binding and final keyword with example.
10. What is polymorphism in java? Explain how polymorphism supported in java.
11. Define polymorphism. Show how compile time polymorphism is achieved in java program with example program.
12. Explain dynamic dispatch method Or method overriding.
13. Explain about types of packages with an example program.
14. Explain about abstract class and abstract method.
15. Write a program to create an interface for customer. In this keep the method called information(), show() and also maintain the tax rate. Implement the interface in employee class and calculate the tax of an employee based on their income.

Income	Tax percentage
<=Rs 2,50,000	Nil
Rs 2,50,000 – Rs 5,00,000	5%
Rs 5,00,000 – 10,00,000	20%
> Rs 10,00,000	30%

16. Define inner classes. How to access object state using inner classes. Give an example.
17. Explain nested interface with example program.
18. What is an interface? Show that how interface can be extended.

19. What is meant by interface? How it is declared and implemented in java. Give example.
20. Write a Java Program to create an abstract class named Shape that contains two integers and an empty method named printArea() and Circumference(). Provide three classes named Rectangle, Triangle and Circle such that each one of the classes extends the class Shape. Each one of the classes contains the method to printArea () and Circumference (). That prints the area and circumference of the given shape.
21. Create a java class shape with constructor to initialize the one parameter "dimension". Now create three subclasses of shape with the following methods.

i) "Circle" with method to calculate the area and circumference of the circle with dimension as radius.
ii) "Square" with method to calculate the area and length of diagonal of square.
iii)"Sphere" with method to calculate the volume and surface area.

CS3391 OBJECT ORIENTED PROGRAMMING

UNIT III EXCEPTION HANDLING AND MULTITHREADING

Exception Handling basics – Multiple catch Clauses – Nested try Statements – Java's Built-in Exceptions – User defined Exception. Multithreaded Programming: Java Thread Model–Creating a Thread and Multiple Threads – Priorities – Synchronization – Inter Thread Communication- Suspending –Resuming, and Stopping Threads – Multithreading. Wrappers – Auto boxing.

---

1. Explain about exception hierarchy.

An exception is a problem that arises during the execution of a program. When an **Exception** occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled.

An exception can occur for many different reasons. Following are some scenarios where an exception occurs.

- A user has entered an invalid data.
- A file that needs to be opened cannot be found.
- A network connection has been lost in the middle of communications or the JVM has run out of memory.

- A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code. When an exceptional condition arises, an object representing that exception is created and *thrown* in the method that caused the error. That method may choose to handle the exception itself, or pass it on. Either way, at some point, the exception is *caught* and processed.
- Exceptions can be generated by the Java run-time system, or they can be manually generated by your code. Exceptions thrown by Java relate to fundamental errors that violate the rules of the Java language or the constraints of the Java execution environment. Manually generated exceptions are typically used to report some error condition to the caller of a method.
  - Java exception handling is managed via five keywords:

    try,

    catch,

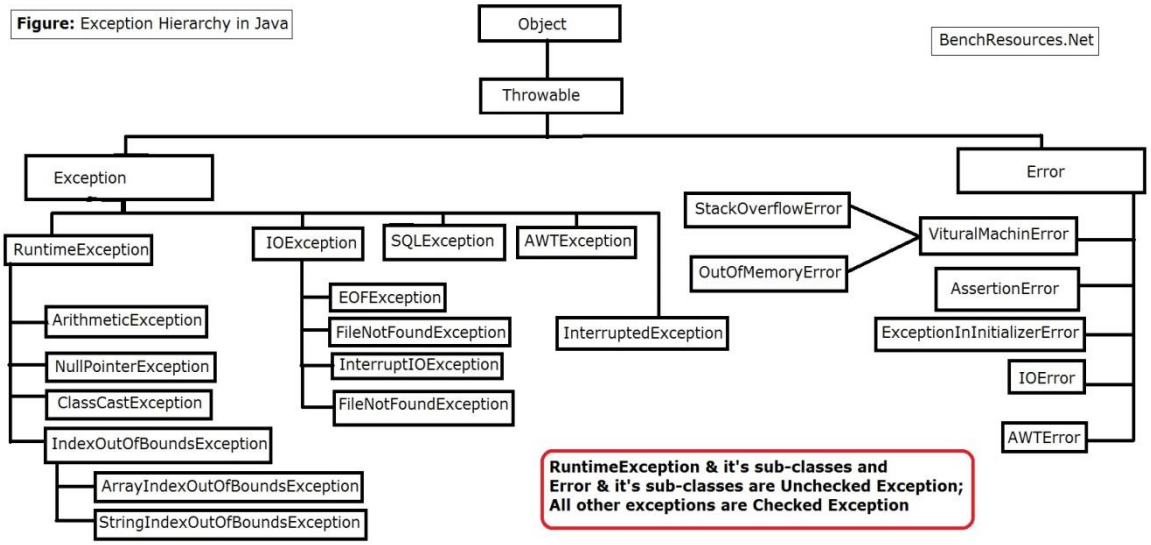
    throw,

    throws,

    finally.

- Program statements that you want to monitor for exceptions are contained within a **try** block. If an exception occurs within the **try** block, it is thrown. Your code can catch this exception (using **catch**) and handle it in some rational manner. System-generated exceptions are automatically thrown by the Java run-time system.
- To manually throw an exception, use the keyword **throw**. Any exception that is thrown out of a method must be specified as such by a **throws** clause. Any code that absolutely must be executed after a **try** block completes is put in a **finally** block.

EXCEPTION HIERARCHY :



- All exception types are subclasses of the built-in class **Throwable**. Thus, **Throwable** is at the top of the exception class hierarchy.
- Immediately below **Throwable** are two subclasses that partition exceptions into two distinct branches.
- One branch is headed by **Exception**. This class is used for exceptional conditions that user programs should catch. This is also the class that you will subclass to create your own custom exception types. There is an important subclass of **Exception**, called **RuntimeException**. Exceptions of this type are automatically defined for the programs that you write and include things such as division by zero and invalid array indexing.

- The other branch is topped by **Error**, which defines exceptions that are not expected to be caught under normal circumstances by your program. Exceptions of type **Error** are used by the Java run-time system to indicate errors having to do with the run-time environment, itself. Stack overflow is an example of such an error.

USING TRY AND CATCH:

Although the default exception handler provided by the Java run-time system is useful for debugging, you will usually want to handle an exception yourself.

Exception handler provides two benefits.

1) It allows you to fix the error.
2) It prevents the program from automatically terminating.

Program:

```
class Exc2
 {
public static void main(String args[])
{
int d, a;
try
 {
d = 0;
a = 42 / d;
```

```
System.out.println("This will not be printed.");
}
catch (ArithmeticException e)
{
 System.out.println("Division by zero.");
}

System.out.println("After catch statement.");
}
}
```

This program generates the following output: Division
by zero.
After catch statement.

2. Explain about multiple catch clauses.

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

```
class MultipleCatches
 {
public static void main(String args[])
 {
 try
{  int a = 0;
System.out.println("a = " + a);
int b = 42 / a;
int c[] = { 1 };
c[42] = 99;
}
catch(ArithmeticException e)
 {
System.out.println("Divide by 0: " + e);
}
catch(ArrayIndexOutOfBoundsException e)
 {
System.out.println("Array index oob: " + e);
}
System.out.println("After try/catch blocks.");
}
}
```

Output:

a = 0
Divide by 0: java.lang.ArithmeticException: / by zero
After try/catch blocks.

3. Explain about nested try statements.

The **try** statement can be nested. That is, a **try** statement can be inside the block of another **try**. Each

time a **try** statement is entered, the context of that exception is pushed on the stack. If an inner **try** statement does not have a **catch** handler for a particular exception, the stack is unwound and the next **try** statement's **catch** handlers are inspected for a match. This continues until one of the **catch** statements succeeds, or until all of the nested **try** statements are exhausted. If no **catch** statement matches, then the Java run-time system will handle the exception. Here is an example that uses nested **try** statements:

```
    An example of nested try statements. class
NestTry
{
public static void main(String args[])
{
try {
int a = 0;
 int b = 42 / a;
System.out.println("a = " + a);
try
{
if(a==1)
a = a/(a-a);
 if(a==2)
{
            int c[] = { 1 };
            c[42] = 99;
}
}
catch(ArrayIndexOutOfBoundsException e)
 {
 System.out.println("Array index out-of-bounds: " + e);
}
}
catch(ArithmeticException e)
{
System.out.println("Divide by 0: " + e);}}}
```

This program nests one **try** block within another. The program works as follows. When you execute the program with no command-line arguments, a divide-by-zero exception is generated by the outer **try** block. Execution of the program with one command-line argument generates a divide-by-zero exception from within the nested **try** block. Since the inner block does not catch this exception, it is passed on to the outer **try** block, where it is handled. If you execute the program with two command-line arguments, an array boundary exception is generated from within the inner **try** block.

Output:
Divide by 0: java.lang.ArithmeticException: / by zero
Output when a=1:
a = 1
Divide by 0: java.lang.ArithmeticException: / by zero C:\>java
NestTry One Two
Output when a=2:
Array index out-of-bounds:
java.lang.ArrayIndexOutOfBoundsException:42

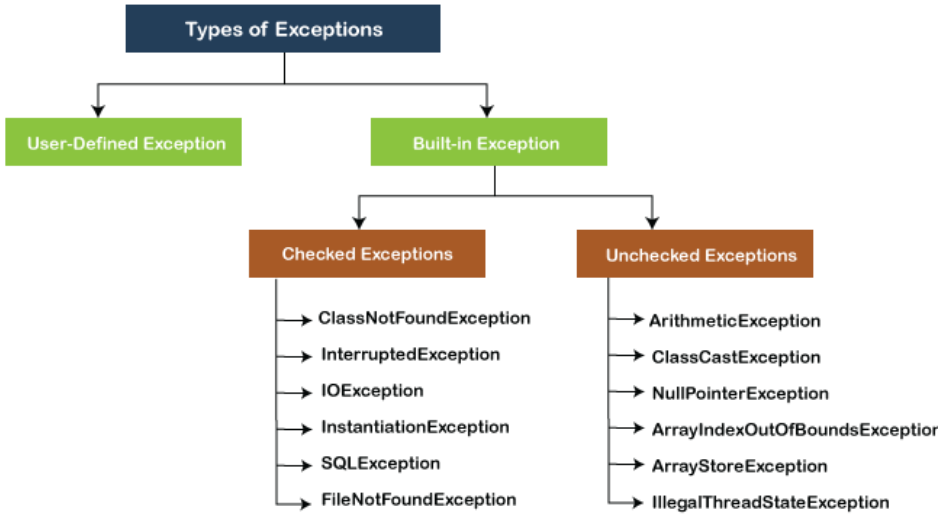
finally:

When exceptions are thrown, execution in a method takes a rather abrupt, nonlinear path that alters the normal flow through the method. Depending upon how the method is coded, it is even possible for an exception to cause the method to return prematurely. This could be a problem in some methods. For example, if a method opens a file upon entry and closes it upon exit, then you will not want the code that closes the file to be bypassed by the exception-handling mechanism. The **finally** keyword is designed to

address this contingency.
Finally creates a block of code that will be executed after a **try /catch** block has completed and before the code following the **try/catch** block. The **finally** block will execute whether or not an exception is thrown. If an exception is thrown, the **finally** block will execute even if no **catch** statement matches the exception.

4. Explain in detail about various types of exception.

An exception is a problem that arises during the execution of a program. When an **Exception** occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled.



Two types of exceptions are

- Built in exception
- User defined exception

1) BUILT-IN EXCEPTIONS:

Inside the standard package **java.lang**, Java defines several exception classes. The most general of exceptions are the subclasses of the standard type **RuntimeException**. These exceptions need not be included in any method's **throws** list.

i) Checked Exception

Checked exceptions are checked at compile-time. The classes which directly inherit Throwable class except RuntimeException and Error are known as checked exceptions.
e.g. IOException, SQLException etc. Checked exceptions are checked at compile- time.

ii) Unchecked Exception

Unchecked exceptions are not checked at compile-time, but they are checked at runtime. The classes which inherit RuntimeException are known as unchecked exceptions. These exceptions need not be included in any method's **throws** list.
ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc.

Program:

```
class MultipleCatches
{
public static void main(String args[])
{
try
{
int a = 0;
 System.out.println("a = " + a); int b =
42 / a;
int c[] = { 1 };
c[42] = 99;
}
catch(ArithmeticException e)
{
System.out.println("Divide by 0: " + e);
}
catch(ArrayIndexOutOfBoundsException e)
{
System.out.println("Array index oob: " + e);
}
System.out.println("After try/catch blocks.");
}
finally
{
System.out.println(" Finally block");
}
}
```

Output:

```
C:\>java MultipleCatches a = 0
Divide by 0: java.lang.ArithmeticException: / by zero
After try/catch blocks.
Finally block
```

2) USER DEFINED EXCEPTION:

User can create their own exception types to handle situations specific to user applications. This is quite easy to do: just define a subclass of **Exception .**

The **Exception** class does not define any methods of its own. It inherits the methods provided by **Throwable**. Thus, all exceptions, including those that you create, have the methods defined by **Throwable** available to them.

Exception defines four constructors. Two support chained exceptions. The other two are shown here:

```
Exception( )
Exception(String msg)
```

The first form creates an exception that has no description. The second form lets you specify a description of the exception.

Program:

```
class MyException extends Exception
{
int detail; MyException(int
a)
{
```

```
detail = a;
}
public String toString()
{
return "MyException:" + detail ;
}
}
class ExceptionDemo
{
static void compute(int a) throws MyException
{
System.out.println("Called compute”);
 if(a > 10)
throw new MyException(a);
System.out.println("Normal exit");
}
public static void main(String args[])
{
try
{
compute(1);
compute(20);
}
catch (MyException e)
{
System.out.println("Caught " + e);
}
}
}
```

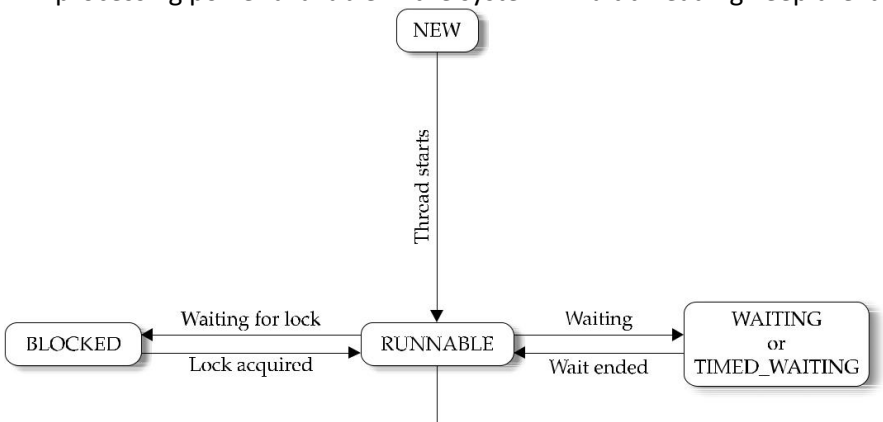
Output:

```
Called compute(1) Normal exit
    Called compute(20)
    Caught
    MyException[20]
```

5. Discuss the life cycle of threads with neat diagram.

Answer:

- Thread is the smallest unit of dispatchable code. This means that a single program can perform two or more tasks simultaneously. For instance, a text editor can format text at the same time that it is printing, as long as these two actions are being performed by two separate threads.
- Threads are lighter weight. They share the same address space and cooperatively share the same heavyweight process. Interthread communication is inexpensive, and context switching from one thread to the next is lower in cost.
- Multithreading enables to write efficient programs that make maximum use of the processing power available in the system. Multithreading keep the idle time to a minimum.



Threads exist in several states.

1) A thread can be ***running***. It can be *ready to run* as soon as it gets CPU time.
2) A running thread can be ***suspended***, which temporarily halts its activity.
3) A suspended thread can then be ***resumed***, allowing it to pick up where it left off.
4) A thread can be ***blocked*** when waiting for a resource.
5) At any time, a thread can be **terminated**, which halts its execution immediately. Once terminated, a thread cannot be resumed.

Obtaining A Thread's State

A thread can exist in a number of different states. You can obtain the current state of a thread by calling the **getState( )** method defined by **Thread**. It is shown here:

```
Thread.State getState( )
```

It returns a value of type **Thread.State** that indicates the state of the thread at the time at which the call was made. **State** is an enumeration defined by **Thread**. Here are the values that can be returned by **getState( )**:

Value	State
BLOCKED	A thread that has suspended execution because it is waiting to acquire a lock.

NEW	A thread that has not begun execution.
RUNNABLE	A thread that either is currently executing or will execute when it gains access to the CPU.
TERMINATED	A thread that has completed execution.
TIMED_WAITING	A thread that has suspended execution for a specified period of time, such as when it has called sleep( ). This state is also entered when a timeout version of wait( ) or join( ) is called.
WAITING	A thread that has suspended execution because it is waiting for some action to occur. For example, it is waiting because of a call to a non-timeout version of wait( ) or join( ).

Given a **Thread** instance, you can use **getState( )** to obtain the state of a thread. For example, the following sequence determines if a thread called **thrd** is in the **RUNNABLE** state at the time **getState( )** is called:

```
Thread.State ts = thrd.getState();

if(ts == Thread.State.RUNNABLE) // ...
```

It is important to understand that a thread's state may change after the call to **getState( )**. Thus, depending on the circumstances, the state obtained by calling **getState( )** may not reflect the actual state of the thread only a moment later. For this (and other) reasons, **getState( )** is not intended to provide a means of synchronizing threads. It's primarily used for debugging or for profiling a thread's run-time characteristics.

6. Develop a program for creating threads by using Thread class and Runnable interface.
   (or) Why do we need both start() and run() method both? can we achieve it with only run() method?

Answer:

We can call run() method if we want but then it would behave just like a normal method and we would not be able to take the advantage of multithreading. When the run method gets called though start() method then a new separate thread is being allocated to the execution of run method, so if more than one thread calls start() method that means their run method is being executed by separate threads (these threads run simultaneously).

Java defines two ways in which this can be accomplished:

1) By extending the **Thread** class.
2) By implement the **Runnable** interface.

1. Creating thread by using Thread class:

Create a new class that extends **Thread**, and then create an instance of that class. The extending class must override the **run( )** method, which is the entry point for the new thread. It must also call **start( )** to begin execution of the new thread.

```
class multithreading extends Thread
```

```
{
   public void run()
   {
      try
      {
          System.out.println ("Thread " +   Thread.currentThread().getId() + " is
                running");
       }
      catch (Exception e)
      {
           System.out.println ("Exception is caught");
      }
   }
}
 public class multithreadingdemo
{
   public static void main(String[] args)
   {
         multithreading m1=new multithreading();
         multithreading m2=new multithreading();
        m1.start();
       m2.start();
}
}
```

Output:

Thread 21 is running

Thread 22 is running

2. **<u>Creating thread by using Runnable interface:</u>**

The easiest way to create a thread is to create a class that implements the **Runnable** interface. To implement **Runnable**, a class need only implement a single method called **run( )**, which is declared like this:

public void run( )

Inside **run( )**, you will define the code that constitutes the new thread. It is important to understand that **run( )** can call other methods, use other classes, and declare variables, just like the main thread can. The only difference is that **run( )** establishes the entry point for another, concurrent thread of execution within your program. This thread will end when **run( )** returns.

After you create a class that implements **Runnable**, you will instantiate an object of type **Thread** from within that class. **Thread** defines several constructors. The one that we will use is shown here:

Thread(Runnable *threadOb)*

Thread(Runnable *threadOb*, String *threadName*)

Thread(String *threadName*)

In this constructor, *threadOb* is an instance of a class that implements the **Runnable** interface. This defines where execution of the thread will begin. The name of the new thread is specified by *threadName*.

After the new thread is created, it will not start running until you call its **start( )** method, which is declared within **Thread**. In essence, **start( )** executes a call to **run( )**. The **start( )** method is shown here:

void start( )

Here is an example that creates a new thread and starts it running:

```
class multithreading implements Runnable
{
  public void run()
   {
     try
     {
         System.out.println ("Thread " +   Thread.currentThread().getId() + " is
                  running");

     }
     catch (Exception e)
```

```
    {
        System.out.println ("Exception is caught");
    }
  }
}
 public class multithreadingdemo
{
   public static void main(String[] args)
   {
         multithreading m1=new multithreading();
             Thread t1 = new Thread(m1);
         multithreading m2=new multithreading();
        Thread t2 = new Thread(m2);
         t1.start();
         t2.start();
}
}
```

Output:

```
Thread 21 is running
Thread 22 is running
```

3. Develop a java program that provide synchronization for two threads deposit and withdraw in a bank application.
   (or) What is thread synchronization? Discuss with an example.
   Answer:

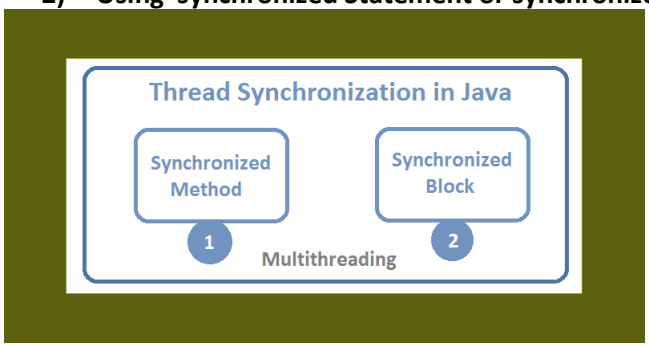
When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called *synchronization*. Java provides unique, language-level support for it.

The synchronization is mainly used to

- To prevent thread interference.
- To prevent consistency problem.

Synchronization can be done in two ways:

1) **Using Synchronized Methods**
2) **Using synchronized Statement or synchronized block**



1)Using Synchronized Methods:

Synchronization is easy in Java, because all objects have their own implicit monitor associated with them. To enter an object's monitor, just call a method that has been modified with the **synchronized** keyword. While a thread is inside a synchronized method, all other threads that try to call it (or any other synchronized method) on the same instance have to wait. To exit the monitor and

relinquish control of the object to the next waiting thread, the owner of the monitor simply returns from the synchronized method.

Program:

```
class bank{
   int amount=5000;
synchronized void transaction(int n,char c ){
  try
  {
  if (c=='d')
{
System.out.println("Before deposit Balance ="+amount);
    amount =amount+n;
System.out.println("After deposit Balance ="+amount);
}
   else if(c=='w')  {
     System.out.println("Before withdraw Balance ="+amount);
```

```
     amount = amount-n;
     System.out.println("After withdraw Balance ="+amount);
}
Thread.sleep(400);
  }
    catch(Exception e)
{
System.out.println(e);}
 }
}
 class deposit extends Thread
{
bank t;
deposit(bank x){
t=x;
}
public void run(){
t.transaction(4000,'d');
}
}
class withdraw extends Thread{
bank t;
withdraw(bank x){
t=x;
}
public void run(){
t.transaction(2000,'w');
}
}

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

```
bank obj = new bank();

deposit t1=new deposit(obj);

withdraw t2=new withdraw(obj);

t1.start();

t2.start();

}

}
```

Output:

```
Before deposit Balance =5000
After deposit Balance =9000
Before withdraw Balance =9000
After withdraw Balance =7000
```

2)The synchronized Statement: (synchronized block)

While creating **synchronized** methods within classes that you create is an easy and effective means of achieving synchronization, it will not work in all cases. To understand why, consider the following. Imagine that you want to synchronize access to objects of a class that was not designed for multithreaded access. That is, the class does not use **synchronized** methods.

Further, this class was not created by you, but by a third party, and you do not have access to the source code. Thus, you can't add **synchronized** to the appropriate methods within the class. How can access to an object of this class be synchronized? Fortunately, the solution to this problem is quite easy: You simply put calls to the methods defined by this class inside a **synchronized** block.

This is the general form of the **synchronized** statement:

```
synchronized(object) {
  // statements to be synchronized
}
```

Here, *object* is a reference to the object being synchronized. A synchronized block ensures that a call to a method that is a member of *object* occurs only after the current thread has successfully entered *object*'s monitor.

Program:

```
class bank{
   int amount=5000;
void transaction(int n,char c ){
```

```
  try
  {
  if (c=='d')
{
System.out.println("Before deposit Balance ="+amount);
    amount =amount+n;
System.out.println("After deposit Balance ="+amount);
}
   else if(c=='w')  {
     System.out.println("Before withdraw Balance ="+amount);
     amount = amount-n;
     System.out.println("After withdraw Balance ="+amount);
}
Thread.sleep(400);
  }
    catch(Exception e){System.out.println(e);}
 }
}

class deposit extends Thread{
bank t;
deposit(bank x){
t=x;
}
public void run(){
synchronized(t)
{
t.transaction(4000,'d');
}
}
}
```

```
class withdraw extends Thread{
bank t;
withdraw(bank x){
t=x;
}
public void run(){
synchronized(t){
t.transaction(2000,'w');
}
}
}

public class testsync{
public static void main(String args[]){
bank obj = new bank();
deposit t1=new deposit(obj);
withdraw t2=new withdraw(obj);
t1.start();
t2.start();
}
}
```

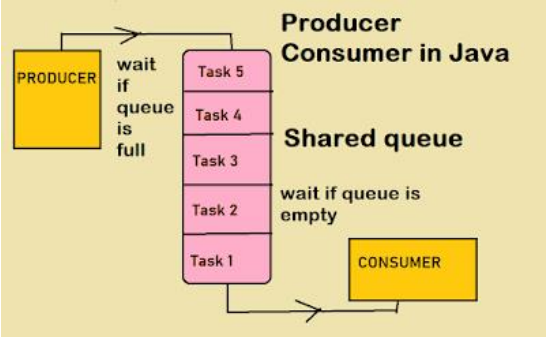
Output:

```
Before deposit Balance =5000
After deposit Balance =9000
Before withdraw Balance =9000
After withdraw Balance =7000
```

4. Write a java program for inventory problem to illustrate the usage of thread synchronized keyword and interthread communication process. They have three classes called consumer, producer and stock.
   (Or) Explain about interthread communication with example program.

   - **Java includes an elegant interprocess communication mechanism via the wait( ), notify( ), and notifyAll( ) methods. Inter-thread communication is all about allowing synchronized threads to communicate with each other.** These methods are implemented as **final** methods in **Object**, so all classes have them. All three methods can be called only from within a **synchronized** context.

wait( ) tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls **notify( )**.
notify( ) wakes up a thread that called **wait( )** on the same object.
notifyAll( ) wakes up all the threads that called **wait( )** on the same object. One of the threads will be granted access.



Consider the following sample program that implements a simple form of the producer/ consumer problem. It consists of four classes: **Queue**, the queue that you're trying to synchronize; **Producer**, the threaded object that is producing queue entries; **Consumer**, the threaded object that is consuming queue entries; and **PC**, the tiny class that creates the single **Q**, **Producer**, and **Consumer**.

```
class Queue

{

int n;

boolean valueSet = false;


synchronized int get()

{

while(!valueSet)

try {

wait();

}

catch(InterruptedException e)

{

System.out.println("InterruptedException caught");

}

System.out.println("Got: " + n);

valueSet = false;

notify();
```

```
return n;
}



synchronized void put(int x)
 {
while(valueSet)
try {
wait();
}
catch(InterruptedException e)
{
System.out.println("InterruptedException caught");
}
n = x;
valueSet = true;
System.out.println("Put: " + n);
notify();
}
}

class Producer extends Thread {
   Queue q;
Producer(Queue q1) {
q = q1;
}
public void run() {
int i = 0;
while(true) {
q.put(i++);
}
```

```
}
}

class Consumer extends Thread{
   Queue q;
Consumer(Queue q1) {
q = q1;
}

public void run() {
while(true) {
q.get();
}
}
}
```

```
public class PCFixed
{
public static void main(String args[])
{
Queue q = new Queue();
Producer p=new Producer(q);
Consumer c= new Consumer(q);
p.start();
c.start();
System.out.println("Press Control-C to stop.");
}
}
```

Output:

Put: 1

Got: 1

Put: 2

Got: 2

Put: 3

Got: 3

Put: 4

Got: 4

Put: 5

Got: 5

10. Explain about creating multiple threads and thread priorities.

The program can create as many threads as it needs.

```
class multithreading extends Thread
{
   public void run()
   {
      try
      {
          System.out.println ("Thread " +   Thread.currentThread().getId() + " is
                running");
       }
      catch (Exception e)
      {
           System.out.println ("Exception is caught");
      }
```

```
   }
}
 public class multithreadingdemo
{
   public static void main(String[] args)
   {
         multithreading m1=new multithreading();
         multithreading m2=new multithreading();
m1.start();
m2.start();
}
}
```

Output:

```
Thread 21 is running
Thread 22 is running
```

Using isAlive( ) and join( )

Often you will want the main thread to finish last. This can be accomplished by calling **sleep( )** within **main( )**, with a long enough delay to ensure that all child threads terminate prior to the main thread. However, this is hardly a satisfactory solution, and it also raises a larger question: How can one thread know when another thread has ended? Fortunately, **Thread** provides a means by which you can answer this question.

Two ways exist to determine whether a thread has finished.

1) First, you can call **isAlive( )** on the thread. This method is defined by **Thread**, and its general form is shown here:

```
final boolean isAlive( )
```

The **isAlive( )** method returns **true** if the thread upon which it is called is still running. It returns **false** otherwise. **isAlive( )** is occasionally useful.

2) Most commonly used method to wait for a thread to finish is called **join( )**, shown here:

final void join( ) throws InterruptedException

This method waits until the thread on which it is called terminates. Its name comes from the concept of the calling thread waiting until the specified thread *joins* it. Additional forms of **join( )** allow you to specify a maximum amount of time that you want to wait for the specified thread to terminate.

```
class jointhread extends Thread{
   public void run() {
       try {
         System.out.println(Thread.currentThread().getName());
            Thread.sleep(100);
         }
         catch (InterruptedException e) {
           System.out.println("Thread interrupted");
         }

   }
}
public class JoinExample {
   public static void main(String[] args) {
      jointhread t1 = new jointhread();
      jointhread t2 = new jointhread();

      t1.start();
      t2.start();

      System.out.println("t1 Alive - " + t1.isAlive());
      System.out.println("t2 Alive - " + t2.isAlive());

try {
```

```
            t1.join();
            t2.join();
        }
        catch (InterruptedException e) {
            System.out.println("Thread interrupted");
        }
        System.out.println("t1 Alive - " + t1.isAlive());
        System.out.println("t2 Alive - " + t2.isAlive());
        System.out.println("Processing finished");
    }
}
```

Output:

```
Thread-0
Thread-1
t1 Alive - true
t2 Alive - true
t1 Alive - false
t2 Alive - false
Processing finished
```

THREAD PRIORITIES:

- Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run. In theory, higher-priority threads get more CPU time than lower-priority threads. In practice, the amount of CPU time that a thread gets often depends on several factors besides its priority. (For example, how an operating system implements multitasking can affect the relative availability of CPU time.)
- A higher-priority thread can also preempt a lower-priority one. For instance, when a lower-priority thread is running and a higher-priority thread resumes (from sleeping or waiting on I/O, for example), it will preempt the lower-priority thread.

To set a thread's priority, use the **setPriority( )** method, which is a member of **Thread**.

```
final void setPriority(int level)
```

Here, *level* specifies the new priority setting for the calling thread. The value of *level* must be within the range **MIN_PRIORITY** and **MAX_PRIORITY**. Currently, these values are 1 and 10, respectively. To return a thread to default priority, specify **NORM_PRIORITY**, which is currently 5. These priorities are defined as **static final** variables within **Thread**.

You can obtain the current priority setting by calling the **getPriority( )** method of **Thread**, shown here:

```
final int getPriority( )
```

12. Explain about deadlock .

- Deadlock occurs when two threads have a circular dependency on a pair of synchronized objects.
- For example, suppose one thread enters the monitor on object X and another thread enters the monitor on object Y.
- If the thread in X tries to call any synchronized method on Y, it will block as expected. However, if the thread in Y, in turn, tries to call any synchronized method on X, the thread waits forever, because to access X, it would have to release its own lock on Y so that the first thread could complete. Deadlock is a difficult error to debug for two reasons:

1) In general, it occurs only rarely, when the two threads time-slice in just the right way.
2) It may involve more than two threads and two synchronized objects.

```
// Online Java Compiler

// Use this editor to write, compile and run your Java code online


public class TestThread

{

  static String r1 = "java";

  static String r2 = "program";


  public static void main(String args[]) {

    ThreadDemo1 T1 = new ThreadDemo1();

    ThreadDemo2 T2 = new ThreadDemo2();

    T1.start();

    T2.start();

  }
```

```
static class ThreadDemo1 extends Thread
{
    public void run()
    {
        synchronized (r1)
        {
          System.out.println("Thread 1: Holding lock 1...");

          System.out.println("Thread 1: Waiting for lock 2...");

          synchronized (r2)
          {
            System.out.println("Thread 1: Holding lock 1 & 2...");
          }
        }
     }
  }
  static class ThreadDemo2 extends Thread
  {
     public void run()
     {
        synchronized (r2)
        {
          System.out.println("Thread 2: Holding lock 2...");

          System.out.println("Thread 2: Waiting for lock 1...");

          synchronized (r1)
          {
            System.out.println("Thread 2: Holding lock 1 & 2...");
          }
```

```
    }
  }
 }}
```

Output

Thread 1: Holding lock 1...

Thread 2: Holding lock 2...

Thread 1: Waiting for lock 2...

Thread 2: Waiting for lock 1...

As this example illustrates, if your multithreaded program locks up occasionally, deadlock is one of the first conditions that you should check for.

13. Explain about suspending, resuming, and stopping threads.

- ✓ **stop**() method in **Thread** – This method terminates the **thread** execution.
- ✓ **suspend**() method in **Thread** – If you want to **stop** the **thread** execution and start it again when a certain event occurs. ...
- ✓ **resume**() method in **Thread** – **resume**() method works with **suspend**() method.

- The mechanisms to suspend, stop, and resume threads differ between early versions of Java, such as Java 1.0, and more modern versions, beginning with Java 2. Prior to Java 2, a program used **suspend( )**, **resume( )**, and **stop( )**, which are methods defined by **Thread**, to pause, restart, and stop the execution of a thread. Although these methods seem to be a perfectly reasonable and convenient approach to managing the execution of threads, they must not be used for new Java programs.

- Here's why. The **suspend( )** method of the **Thread** class was deprecated by Java 2 several years ago. This was done because **suspend( )** can sometimes cause serious system failures. Assume that a thread as obtained locks on critical data structures. If that thread is suspended at that point, those locks are not relinquished. Other threads that may be waiting for those resources can be deadlocked.
- The **resume( )** method is also deprecated. It does not cause problems, but cannot be used without the **suspend( )** method as its counterpart. The **stop( )** method of the **Thread** class, too, was deprecated by Java 2. This was done because this method can sometimes cause serious system failures.
- Assume that a thread is writing to a critically important data structure and has completed only part of its changes. If that thread is stopped at that point, that data structure might be left in a corrupted state. The trouble is that **stop( )** causes any lock the calling thread holds to be released. Thus, the corrupted data might be used by another thread that is waiting on the same lock.
- Because you can't now use the **suspend( )**, **resume( )**, or **stop( )** methods to control a thread, you might be thinking that no way exists to pause, restart, or terminate a thread.
- But, fortunately, this is not true. Instead, a thread must be designed so that the **run( )** method periodically checks to determine whether that thread should suspend, resume, or stop its own execution. This is accomplished by establishing a flag variable that indicates the execution state of the thread. As long as this flag is set to "running," the **run( )** method must continue to let the thread execute. If this variable is set to "suspend," the thread must pause. If it is set to "stop," the thread must terminate. Of course, a variety of ways exist in which to write such code, but the central theme will be the same for all programs.

- The following example illustrates how the **wait( )** and **notify( )** methods that are inherited from **Object** can be used to control the execution of a thread. Let us consider its operation. The **NewThread** class contains a **boolean** instance variable named **suspendFlag**, which is used to control the execution of the thread. It is initialized to **false** by the constructor. The **run( )** method contains a **synchronized** statement block that checks **suspendFlag**. If that variable is **true**, the **wait( )** method is invoked to suspend the execution of the thread. The **mysuspend( )** method sets **suspendFlag** to **true**. The **myresume( )** method sets **suspendFlag** to **false** and invokes **notify( )** to wake up the thread. Finally, the **main( )** method has been modified to invoke the **mysuspend( )** and **myresume( )** methods.

```
class NewThread implements Runnable

{

String name;

Thread t;

boolean suspendFlag;

NewThread(String threadname)

 {

name = threadname;

t = new Thread(this, name);

System.out.println("New thread: " + t);

suspendFlag = false;

t.start();

}


public void run()

{

try

 {

for(int i = 3; i > 0; i--)

{

System.out.println(name + ": " + i);

Thread.sleep(200);

synchronized(this)

 {

while(suspendFlag)

{
```

```
wait();
}
}
}
}
 catch (InterruptedException e)
{
System.out.println(name + " interrupted.");
}
System.out.println(name + " exiting.");
}
void mysuspend()
{
suspendFlag = true;
}
synchronized void myresume()
{
suspendFlag = false;
notify();
}
}

class SuspendResume
 {
public static void main(String args[])
{
NewThread ob1 = new NewThread("One");
NewThread ob2 = new NewThread("Two");
try
 {
Thread.sleep(1000);
ob1.mysuspend();
```

```
System.out.println("Suspending thread One");
Thread.sleep(1000);
ob1.myresume();
System.out.println("Resuming thread One");
ob2.mysuspend();
System.out.println("Suspending thread Two");
Thread.sleep(1000);
ob2.myresume();
System.out.println("Resuming thread Two");
}
catch (InterruptedException e)
 {
System.out.println("Main thread Interrupted");
}
// wait for threads to finish
try
{
System.out.println("Waiting for threads to finish.");
ob1.t.join();
ob2.t.join();
}
 catch (InterruptedException e)
 {
System.out.println("Main thread Interrupted");
}
System.out.println("Main thread exiting.");
}
}
```

Output:

New thread: Thread[One,5,main]

New thread: Thread[Two,5,main]

One: 3

Two: 3

One: 2

Two: 2

One: 1Two: 1

One exiting.

Two exiting.

Suspending thread One

Resuming thread One

Suspending thread Two

Resuming thread Two

Waiting for threads to finish.

Main thread exiting.

14. Explain about type wrappers in java.

Java uses primitive types (also called simple types), such as **int** or **double**, to hold the basic data types supported by the language. Primitive types, rather than objects, are used for these quantities for the sake of performance. Using objects for these values would add an unacceptable overhead to even the simplest of calculations. Thus, the primitive types are not part of the object hierarchy, and they do not inherit **Object**.

Despite the performance benefit offered by the primitive types, there are times when you will need an object representation. For example, you can't pass a primitive type by reference to a method. Also, many of the standard data structures implemented by Java operate on objects, which means that you can't use these data structures to store primitive types. To handle these situations, Java provides *type wrappers,* which are classes that encapsulate a primitive type within an object.

The type wrappers are **Double**, **Float**, **Long**, **Integer**, **Short**, **Byte**, **Character**, and **Boolean**. These classes offer a wide array of methods that allow you to fully integrate the primitive types into Java's object hierarchy.

Character:

Character is a wrapper around a **char**. The constructor for **Character** is Character(char *ch*)

Here, *ch* specifies the character that will be wrapped by the **Character** object being created.

To obtain the **char** value contained in a **Character** object, call **charValue( )**, shown here:

char charValue( )

It returns the encapsulated character.

Boolean

Boolean is a wrapper around **boolean** values. It defines these constructors:

Boolean(boolean *boolValue*)

Boolean(String *boolString*)

In the first version, *boolValue* must be either **true** or **false**. In the second version, if *boolString* contains the string "true" (in uppercase or lowercase), then the new **Boolean** object will be true. Otherwise, it will be false.

To obtain a **boolean** value from a **Boolean** object, use **booleanValue( )**, shown here:

boolean booleanValue( )

It returns the **boolean** equivalent of the invoking object.

The Numeric Type Wrappers

By far, the most commonly used type wrappers are those that represent numeric values. These are **Byte**, **Short**, **Integer**, **Long**, **Float**, and **Double**. All of the numeric type wrappers inherit the abstract class **Number**. **Number** declares methods that return the value of an object in each of the different number formats. These methods are shown here:

byte byteValue( )

double doubleValue( )

float floatValue( )

int intValue( )

long longValue( )

short shortValue( )

For example, **doubleValue( )** returns the value of an object as a **double**, **floatValue( )** returns the value as a **float**, and so on. These methods are implemented by each of the numeric type wrappers.

All of the numeric type wrappers define constructors that allow an object to be constructed from a given value, or a string representation of that value. For example, here are the constructors defined for **Integer**:

Integer(int *num*)

Integer(String *str*)

If *str* does not contain a valid numeric value, then a **NumberFormatException** is thrown. All of the type wrappers override **toString( )**. It returns the human-readable form of the value contained within the wrapper. This allows you to output the value by passing a type wrapper object to **println( )**, for example, without having to convert it into its primitive type.The following program demonstrates how to use a numeric type wrapper to encapsulate a value and then extract that value.

```
// Demonstrate a type wrapper.

class Wrap {

public static void main(String args[]) {

Integer iOb = new Integer(100);

int i = iOb.intValue();

System.out.println(i + " " + iOb); // displays 100 100

}

}
```

This program wraps the integer value 100 inside an **Integer** object called **iOb**. The program then obtains this value by calling **intValue( )** and stores the result in **i**.

The process of encapsulating a value within an object is called *boxing.* Thus, in the program,

this line boxes the value 100 into an **Integer**:

```
Integer iOb = new Integer(100);
```

The process of extracting a value from a type wrapper is called *unboxing*. For example, the

program unboxes the value in **iOb** with this statement:

```
int i = iOb.intValue();
```

The same general procedure used by the preceding program to box and unbox values has been employed since the original version of Java. However, with the release of JDK 5, Java

fundamentally improved on this through the addition of autoboxing, described next.

15. Explain about autoboxing and unboxing.

Beginning with JDK 5, Java added two important features:

autoboxing and *auto-unboxing.*

Autoboxing is the process by which a primitive type is automatically encapsulated (boxed) into its equivalent type wrapper whenever an object of that type is needed. There is no need to explicitly construct an object.

Auto-unboxing is the process by which the value of a boxed object is automatically extracted (unboxed) from a type wrapper when its value is needed. There is no need to call a method such as **intValue( )** or **doubleValue( )**.

The addition of autoboxing and auto-unboxing greatly streamlines the coding of several algorithms, removing the tedium of manually boxing and unboxing values. It also helps prevent errors. Moreover, it is very important to generics, which operates only on objects. Finally, autoboxing makes working with the Collections Framework much easier.

With autoboxing it is no longer necessary to manually construct an object in order to wrap a primitive type. You need only assign that value to a type-wrapper reference. Java automatically constructs the object for you. For example, here is the modern way to construct an **Integer** object that has the value 100:

```
Integer iOb = 100;                                    // autobox an int
```

Notice that no object is explicitly created through the use of **new**. Java handles this for you, automatically.

To unbox an object, simply assign that object reference to a primitive-type variable. For example, to unbox **iOb**, you can use this line:

```
int i = iOb;                                          // auto-unbox
```

Java handles the details for you.

Here is the preceding program rewritten to use autoboxing/unboxing:

```
// Demonstrate autoboxing/unboxing.

class AutoBox
```

```
 {
public static void main(String args[])
{
Integer iOb = 100; // autobox an int
int i = iOb; // auto-unbox
System.out.println(i + " " + iOb);                    // displays 100 100
}
}
```

Autoboxing/Unboxing Boolean and Character Values

Java also supplies wrappers for **boolean** and **char**. These are **Boolean** and **Character**. Autoboxing/unboxing applies to these wrappers, too. For example, consider the following program:

```
// Autoboxing/unboxing a Boolean and Character.

class AutoBox5
{
public static void main(String args[])
{
Boolean b = true;
if(b)
 System.out.println("b is true");
Character ch = 'x';                                          // box a char
char ch2 = ch;                                               // unbox a char
System.out.println("ch2 is " + ch2);
}
}
```

The output is shown here:

```
b is true
ch2 is x
```

CS3391/OBJECT ORIENTED PROGRAMMING

Unit-III

Part-A

1. What is an exception?

Exception is an abnormal condition which occurs during the execution of a program and disrupts normal flow of the program. This exception must be handled properly. If it is not handled, program will be terminated abruptly.

2. How the exceptions are handled in java? OR Explain exception handling mechanism in java?

Exceptions in java are handled using try, catch and finally blocks.

try block : The code or set of statements which are to be monitored for exception are kept in this block.

catch block : This block catches the exceptions occurred in the try block.

finally block : This block is always executed whether exception is occurred in the try block or not and occurred exception is caught in the catch block or not.

3. What is the difference between error and exception in java?

Errors are mainly caused by the environment in which an application is running. For example, OutOfMemoryError happens when JVM runs out of memory. Where as exceptions are mainly caused by the application itself. For example, NullPointerException occurs when an application tries to access null object.

4. Can we write only try block without catch and finally blocks?

No, It shows compilation error. The try block must be followed by either catch or finally block.

You can remove either catch block or finally block but not both.

5. What is unreachable catch block error?

When you are keeping multiple catch blocks, the order of catch blocks must be from most specific to most general ones. i.e sub classes of Exception must come first and super classes later. If you keep super classes first and sub classes later, compiler will show unreachable catch block error.

6. Describe the hierarchy of exceptions in java?

All exception types are subclasses of the built-in class Throwable. Thus, Throwable is at the top of the exception class hierarchy.

Immediately below Throwable are two subclasses that partition exceptions into two distinct branches.

One branch is headed by Exception. This class is used for exceptional conditions that user programs should catch. This is also the class that you will subclass to create your own custom exception types. There is an important subclass of Exception, called RuntimeException. Exceptions of this type are automatically defined for the programs that you write and include things such as division by zero and invalid array indexing.

The other branch is topped by Error, which defines exceptions that are not expected to be caught under normal circumstances by your program. Exceptions of type Error are used by the Java run-time system to indicate errors having to do with the run-time environment, itself. Stack overflow is an example of such an error.

7. What are run time exceptions in java. Give example?

The exceptions which occur at run time are called as run time exceptions. These exceptions are unknown to compiler. All sub classes of java.lang.RunTimeException and java.lang.Error are run time exceptions. These exceptions are unchecked type of exceptions. For example, NumberFormatException, NullPointerException, ClassCastException,

8. What are checked and unchecked exceptions in java?

Checked exceptions are the exceptions which are known to compiler. These exceptions are checked at compile time only. Hence the name checked exceptions. These exceptions are also called compile time exceptions. Because, these exceptions will be known during compile time.

Unchecked exceptions are those exceptions which are not at all known to compiler. These exceptions occur only at run time. These exceptions are also called as run time exceptions. All sub classes of java.lang.RunTimeException and java.lang.Error are unchecked exceptions.

9. What is Re-throwing an exception in java?

Exceptions raised in the try block are handled in the catch block. If it is unable to handle that exception, it can re-throw that exception using throw keyword. It is called re-throwing an exception.

10. What is the use of throws keyword in java?

If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception. You do this by including a throws clause in the method's declaration. A throws clause lists the types of exceptions that a method might throw. This is necessary for all exceptions, except those of type Error or RuntimeException, or any of their subclasses. All other exceptions that a method can throw must be declared in the throws clause. If they are not, a compile-time error will result.

11. What is the difference between final, finally and finalize in java?

final keyword :

final is a keyword which is used to make a variable or a method or a class as "unchangeable".

finally Block :

finally is a block which is used for exception handling along with try and catch blocks. finally block is always executed whether exception is raised or not and raised exception is handled or not. Most of time, this block is used to close the resources like database connection, I/O resources etc.

finalize() Method :

finalize() method is a protected method of java.lang.Object class. It is inherited to every class you create in java. This method is called by garbage collector thread before an object is removed from the memory. finalize() method is used to perform some clean up operations on an object before it is removed from the memory.

12. How do you create customized exceptions in java?

In java, we can define our own exception classes as per our requirements. These exceptions are called user defined exceptions in java OR Customized exceptions. User defined exceptions must extend any one of the classes in the hierarchy of exceptions.

13. What is the difference between throw, throws and throwable in java?

throw In Java :

throw is a keyword in java which is used to throw an exception manually. Using throw keyword, you can throw an exception from any method or block. But, that exception must be of type java.lang.Throwable class or it's sub classes. Below example shows how to throw an exception using throw keyword.

throws In Java :

throws is also a keyword in java which is used in the method signature to indicate that this method may throw mentioned exceptions. The caller to such methods must handle the mentioned exceptions either using try-catch blocks or using throws keyword. Below is the syntax for using throws keyword.

Throwable In Java :

Throwable is a super class for all types of errors and exceptions in java. This class is a member of java.langpackage. Only instances of this class or it's sub classes are thrown by the java virtual machine or by the throw statement. The only argument of catch block must be of this type or it's sub classes. If you want to create your own customized exceptions, then your class must extend this class. Click here to see the hierarchy of exception classes in java.

15. Which class is the super class for all types of errors and exceptions in java?

java.lang.Throwable is the super class for all types of errors and exceptions in java.

16. Define Process-based multitasking: (Multiprocessing)

A process is a program that is executing. Thus, process-based multitasking is the feature that allows your computer to run two or more programs concurrently. For example, process-based multitasking enables you to run the Java compiler at the same time that you are using a text editor or visiting a web site. In process-based multitasking, a program is the smallest unit of code that can be dispatched by the scheduler.

17. Define Thread-based multitasking: (Multithreading)

In a thread-based multitasking environment, the thread is the smallest unit of dispatchable code. This means that a single program can perform two or more tasks simultaneously. For instance, a text editor can format text at the same time that it is printing, as long as these two actions are being performed by two separate threads. Thus, process-based multitasking deals with the "big picture," and thread-based multitasking handles the details.

18. Differentiate between multitasking and multithreading.

Sl.no.	Multitasking	Multithreading
1	It is a process of executing many processes running simultaneously.	It is a process of executing multiple threads(sub-process).
2	Process is program in execution. They are Heavy weight.	**Thread** is basically a lightweight sub-process. It is a smallest unit of processing.

3	In multi tasking separate memory is allocated to each process.	Threads(sub-process) are sharing common memory.
4	Interprocess communication is expensive	Inter thread communication is inexpensive .
5	Context switching from one process to another is costly.	Context switching from one thread to another is inexpensive.

19. Define Multithreading.

Multithreading enables to write efficient programs that make maximum use of the processing power available in the system. One important way multithreading achieves this is by keeping idle time to a minimum.

20. **List are the States of thread.**

- A thread can be **running**. It can be ready to run as soon as it gets CPU time.
- A running thread can be **suspended**, which temporarily halts its activity.
- A suspended thread can then be **resumed**, allowing it to pick up where it left off.
- A thread can be **blocked** when waiting for a resource.
- At any time, a thread can be **terminated**, which halts its execution immediately. Once terminated, a thread cannot be resumed.

21. How can you create thread in Java?

Java defines two ways for creating thread.

1) By extending the **Thread** class.

2) By implement the **Runnable** interface.

22. What do you mean by synchronization?

When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called synchronization. Java provides unique, language-level support for it.

Synchronization can be done in two ways:

1) Using Synchronized Methods
2) Using synchronized Statement or synchronized block

23. What is the need for synchronization?

The synchronization is mainly used to

- To prevent thread interference.
- To prevent consistency problem.

24. What is meant by monitor?

A **monitor** is an object that is used as a mutually exclusive lock. Only one thread can own a monitor at a given time. When a thread acquires a lock, it is said to have entered the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor. These other threads are said to be waiting for the monitor. A thread that owns a monitor can reenter the same monitor if it so desires.

25. How will you set the priority of the thread?

Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run. In theory, higher-priority threads get more CPU time than lower-priority threads.

To set a thread's priority, use the **setPriority( )** method.

final void setPriority(int *level*)

The value of *level* must be within the range **MIN_PRIORITY** and **MAX_PRIORITY**. Currently, these values are 1 and 10, respectively. To return a thread to default priority, specify **NORM_PRIORITY**, which is currently 5.

Part-B

1. Develop a java program for handling divide by zero exception with multiple catches.
2. Develop a java program for handling ArrayIndexOutOfBoundsException exception with finally block.
3. Develop a java program for exception handling with throw and throws keyword.
4. Explain nested try/catch with example program.
5. Develop a java program with multiple catch clauses.
6. Develop a java program with multicatch that can handle both ArithmeticException and IndexOutOfBoundsException.
7. Discuss about exception handling in nested try/catch.
8. Explain about types of exception with an example.
9. Develop a Java program to implement user defined exception handling.
10. Discuss the life cycle of threads with neat diagram.
11. Develop a program for creating threads by using Thread class and Runnable interface.
12. Why do we need both start() and run() method both? can we achieve it with only run() method?
13. Write a java program that synchronize three threads of the same program and display the content the text supplied through these threads.
14. Develop a java program that provide synchronization for two threads deposit and withdraw in a bank application.
15. Define thread. Explain the state of threads. State reason for synchronization in thread. Write simple concurrent programming to create, sleep and delete thread.
16. What is thread synchronization? Discuss with an example.
17. Write a java program for inventory problem to illustrate the usage of thread synchronized keyword and interthread communication process. They have three classes called consumer, producer and stock.
18. Write a java program that implements a multi-threaded application that has three threads. First thread generates a random integer every 1 second and if the value is even, second thread computes the square of the number and prints. If the value is odd, the third thread will print the value of cube of the number.
19. Develop a java application program for generating four threads to perform the following operation.
    i) Getting N numbers as input
    ii) Printing even numbers
    iii) Printing odd numbers.
    iv) Computing the average
20. Explain about wrapper classes.
21. Elaborate in detail about autoboxing.
22. Discuss about Suspending, Resuming, and Stopping Threads.

CS3391 OBJECT ORIENTED PROGRAMMING

UNIT III EXCEPTION HANDLING AND MULTITHREADING

Exception Handling basics – Multiple catch Clauses – Nested try Statements – Java's Built-in Exceptions – User defined Exception. Multithreaded Programming: Java Thread Model–Creating a Thread and Multiple Threads – Priorities – Synchronization – Inter Thread Communication- Suspending –Resuming, and Stopping Threads – Multithreading. Wrappers – Auto boxing.

---

7. Explain about exception hierarchy.

An exception is a problem that arises during the execution of a program. When an **Exception** occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled.

An exception can occur for many different reasons. Following are some scenarios where an exception occurs.

- A user has entered an invalid data.
- A file that needs to be opened cannot be found.
- A network connection has been lost in the middle of communications or the JVM has run out of memory.

- A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code. When an exceptional condition arises, an object representing that exception is created and *thrown* in the method that caused the error. That method may choose to handle the exception itself, or pass it on. Either way, at some point, the exception is *caught* and processed.
- Exceptions can be generated by the Java run-time system, or they can be manually generated by your code. Exceptions thrown by Java relate to fundamental errors that violate the rules of the Java language or the constraints of the Java execution environment. Manually generated exceptions are typically used to report some error condition to the caller of a method.
  - Java exception handling is managed via five keywords:

    try,

    catch,

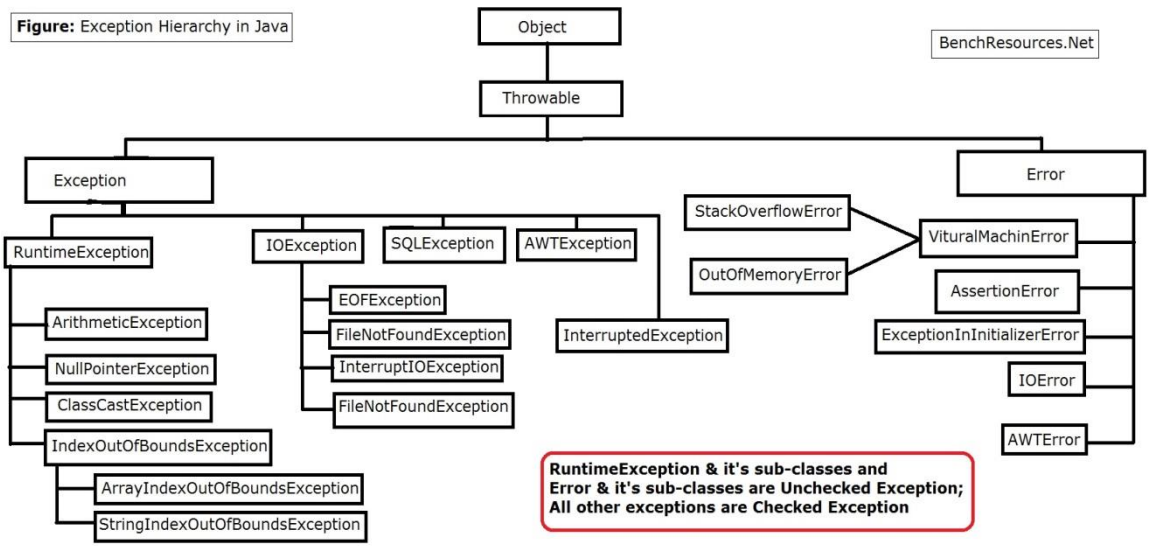
    throw,

    throws,

    finally.

- Program statements that you want to monitor for exceptions are contained within a **try** block. If an exception occurs within the **try** block, it is thrown. Your code can catch this exception (using **catch**) and handle it in some rational manner. System-generated exceptions are automatically thrown by the Java run-time system.
- To manually throw an exception, use the keyword **throw**. Any exception that is thrown out of a method must be specified as such by a **throws** clause. Any code that absolutely must be executed after a **try** block completes is put in a **finally** block.

EXCEPTION HIERARCHY :



- All exception types are subclasses of the built-in class **Throwable**. Thus, **Throwable** is at the top of the exception class hierarchy.
- Immediately below **Throwable** are two subclasses that partition exceptions into two distinct branches.
- One branch is headed by **Exception**. This class is used for exceptional conditions that user programs should catch. This is also the class that you will subclass to create your own custom exception types. There is an important subclass of **Exception**, called **RuntimeException**. Exceptions of this type are automatically defined for the programs that you write and include things such as division by zero and invalid array indexing.

- The other branch is topped by **Error**, which defines exceptions that are not expected to be caught under normal circumstances by your program. Exceptions of type **Error** are used by the Java run-time system to indicate errors having to do with the run-time environment, itself. Stack overflow is an example of such an error.

USING TRY AND CATCH:

Although the default exception handler provided by the Java run-time system is useful for debugging, you will usually want to handle an exception yourself.

Exception handler provides two benefits.

3) It allows you to fix the error.
4) It prevents the program from automatically terminating.

Program:

```
class Exc2
 {
public static void main(String args[])
{
int d, a;
try
 {
d = 0;
a = 42 / d;
System.out.println("This will not be printed.");
```

```
}
catch (ArithmeticException e)
{
 System.out.println("Division by zero.");
}

System.out.println("After catch statement.");
}
}
```

This program generates the following output: Division
by zero.
After catch statement.

8. Explain about multiple catch clauses.

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

```
class MultipleCatches
 {
public static void main(String args[])
 {
 try
{  int a = 0;
System.out.println("a = " + a);
int b = 42 / a;
int c[] = { 1 };
c[42] = 99;
}
catch(ArithmeticException e)
 {
System.out.println("Divide by 0: " + e);
}
catch(ArrayIndexOutOfBoundsException e)
 {
System.out.println("Array index oob: " + e);
}
System.out.println("After try/catch blocks.");
}
}
```

Output:

a = 0
Divide by 0: java.lang.ArithmeticException: / by zero
After try/catch blocks.

9. Explain about nested try statements.

The **try** statement can be nested. That is, a **try** statement can be inside the block of another **try**. Each time a **try** statement is entered, the context of that exception is pushed on the stack. If an inner **try**

statement does not have a **catch** handler for a particular exception, the stack is unwound and the next **try** statement's **catch** handlers are inspected for a match. This continues until one of the **catch** statements succeeds, or until all of the nested **try** statements are exhausted. If no **catch** statement matches, then the Java run-time system will handle the exception. Here is an example that uses nested **try** statements:

```
    An example of nested try statements. class
NestTry
{
public static void main(String args[])
{
try {
int a = 0;
 int b = 42 / a;
System.out.println("a = " + a);
try
{
if(a==1)
a = a/(a-a);
 if(a==2)
{
            int c[] = { 1 };
            c[42] = 99;
}
}
catch(ArrayIndexOutOfBoundsException e)
 {
 System.out.println("Array index out-of-bounds: " + e);
}
}
catch(ArithmeticException e)
{
System.out.println("Divide by 0: " + e);}}}
```

This program nests one **try** block within another. The program works as follows. When you execute the program with no command-line arguments, a divide-by-zero exception is generated by the outer **try** block. Execution of the program with one command-line argument generates a divide-by-zero exception from within the nested **try** block. Since the inner block does not catch this exception, it is passed on to the outer **try** block, where it is handled. If you execute the program with two command-line arguments, an array boundary exception is generated from within the inner **try** block.

Output:
Divide by 0: java.lang.ArithmeticException: / by zero
Output when a=1:
a = 1
Divide by 0: java.lang.ArithmeticException: / by zero C:\>java
NestTry One Two
Output when a=2:
Array index out-of-bounds:
java.lang.ArrayIndexOutOfBoundsException:42

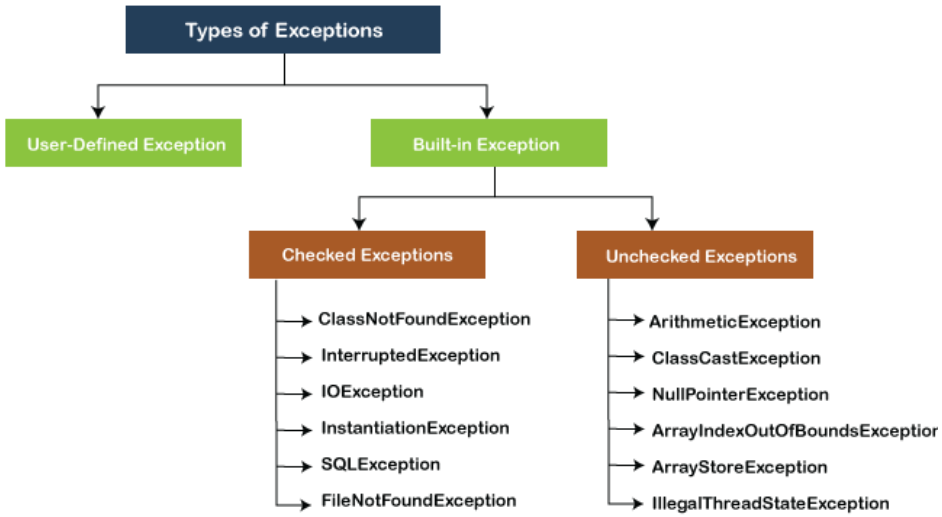
finally:

When exceptions are thrown, execution in a method takes a rather abrupt, nonlinear path that alters the normal flow through the method. Depending upon how the method is coded, it is even possible for an exception to cause the method to return prematurely. This could be a problem in some methods. For example, if a method opens a file upon entry and closes it upon exit, then you will not want the code that closes the file to be bypassed by the exception-handling mechanism. The **finally** keyword is designed to address this contingency.

Finally creates a block of code that will be executed after a **try /catch** block has completed and before the code following the **try/catch** block. The **finally** block will execute whether or not an exception is thrown. If an exception is thrown, the **finally** block will execute even if no **catch** statement matches the exception.

10. Explain in detail about various types of exception.

An exception is a problem that arises during the execution of a program. When an **Exception** occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled.



Two types of exceptions are

- Built in exception
- User defined exception

3) BUILT-IN EXCEPTIONS:

Inside the standard package **java.lang**, Java defines several exception classes. The most general of exceptions are the subclasses of the standard type **RuntimeException**. These exceptions need not be included in any method's **throws** list.

iii) Checked Exception

Checked exceptions are checked at compile-time. The classes which directly inherit Throwable class except RuntimeException and Error are known as checked exceptions.
e.g. IOException, SQLException etc. Checked exceptions are checked at compile- time.

iv) Unchecked Exception

Unchecked exceptions are not checked at compile-time, but they are checked at runtime. The classes which inherit RuntimeException are known as unchecked exceptions. These exceptions need not be included in any method's **throws** list.
ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc.

Program:

```
class MultipleCatches
{
public static void main(String args[])
{
try
{
int a = 0;
 System.out.println("a = " + a); int b =
42 / a;
int c[] = { 1 };
c[42] = 99;
}
catch(ArithmeticException e)
{
System.out.println("Divide by 0: " + e);
}
catch(ArrayIndexOutOfBoundsException e)
{
System.out.println("Array index oob: " + e);
}
System.out.println("After try/catch blocks.");
}
finally
{
System.out.println(" Finally block");
}
}
```

Output:

```
C:\>java MultipleCatches a = 0
Divide by 0: java.lang.ArithmeticException: / by zero
After try/catch blocks.
Finally block
```

4) USER DEFINED EXCEPTION:

User can create their own exception types to handle situations specific to user applications. This is quite easy to do: just define a subclass of **Exception .**

The **Exception** class does not define any methods of its own. It inherits the methods provided by **Throwable**. Thus, all exceptions, including those that you create, have the methods defined by **Throwable** available to them.

Exception defines four constructors. Two support chained exceptions. The other two are shown here:

Exception( )
Exception(String *msg*)

The first form creates an exception that has no description. The second form lets you specify a description of the exception.

Program:

```
class MyException extends Exception
{
int detail; MyException(int
a)
{
detail = a;
}
```

```
public String toString()
{
return "MyException:" + detail ;
}
}
class ExceptionDemo
{
static void compute(int a) throws MyException
{
System.out.println("Called compute”);
 if(a > 10)
throw new MyException(a);
System.out.println("Normal exit");
}
public static void main(String args[])
{
try
{
compute(1);
compute(20);
}
catch (MyException e)
{
System.out.println("Caught " + e);
}
}
}
```

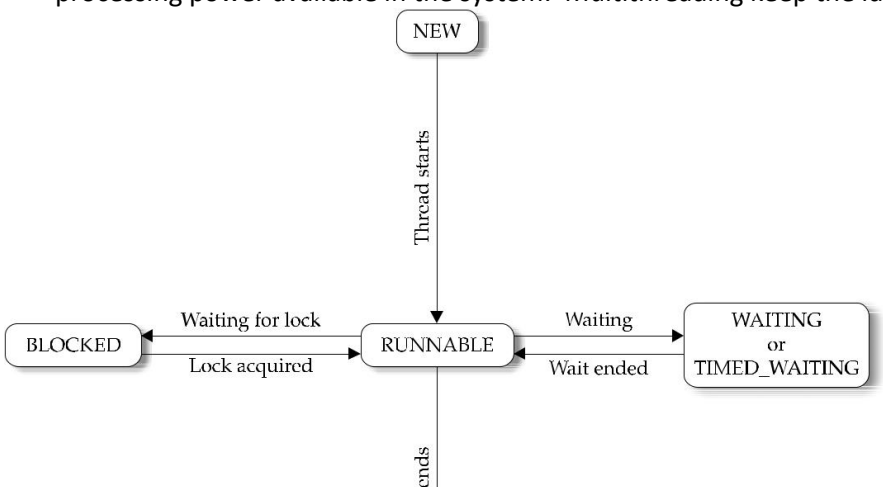
Output:

```
Called compute(1) Normal exit
    Called compute(20)
    Caught
    MyException[20]
```

11. Discuss the life cycle of threads with neat diagram.

Answer:

- Thread is the smallest unit of dispatchable code. This means that a single program can perform two or more tasks simultaneously. For instance, a text editor can format text at the same time that it is printing, as long as these two actions are being performed by two separate threads.
- Threads are lighter weight. They share the same address space and cooperatively share the same heavyweight process. Interthread communication is inexpensive, and context switching from one thread to the next is lower in cost.
- Multithreading enables to write efficient programs that make maximum use of the processing power available in the system. Multithreading keep the idle time to a minimum.



Threads exist in several states.

6) A thread can be ***running***. It can be *ready to run* as soon as it gets CPU time.
7) A running thread can be ***suspended***, which temporarily halts its activity.
8) A suspended thread can then be ***resumed***, allowing it to pick up where it left off.
9) A thread can be ***blocked*** when waiting for a resource.
10) At any time, a thread can be **terminated**, which halts its execution immediately. Once terminated, a thread cannot be resumed.

Obtaining A Thread's State

A thread can exist in a number of different states. You can obtain the current state of a thread by calling the **getState( )** method defined by **Thread**. It is shown here:

```
Thread.State getState( )
```

It returns a value of type **Thread.State** that indicates the state of the thread at the time at which the call was made. **State** is an enumeration defined by **Thread**. Here are the values that can be returned by **getState( )**:

Value	State
BLOCKED	A thread that has suspended execution because it is waiting to acquire a lock.
NEW	A thread that has not begun execution.

RUNNABLE	A thread that either is currently executing or will execute when it gains access to the CPU.
TERMINATED	A thread that has completed execution.
TIMED_WAITING	A thread that has suspended execution for a specified period of time, such as when it has called sleep( ). This state is also entered when a timeout version of wait( ) or join( ) is called.
WAITING	A thread that has suspended execution because it is waiting for some action to occur. For example, it is waiting because of a call to a non-timeout version of wait( ) or join( ).

Given a **Thread** instance, you can use **getState( )** to obtain the state of a thread. For example, the following sequence determines if a thread called **thrd** is in the **RUNNABLE** state at the time **getState( )** is called:

```
Thread.State ts = thrd.getState();

if(ts == Thread.State.RUNNABLE) // ...
```

It is important to understand that a thread's state may change after the call to **getState( )**. Thus, depending on the circumstances, the state obtained by calling **getState( )** may not reflect the actual state of the thread only a moment later. For this (and other) reasons, **getState( )** is not intended to provide a means of synchronizing threads. It's primarily used for debugging or for profiling a thread's run-time characteristics.

12. Develop a program for creating threads by using Thread class and Runnable interface.
    (or) Why do we need both start() and run() method both? can we achieve it with only run() method?

Answer:

We can call run() method if we want but then it would behave just like a normal method and we would not be able to take the advantage of multithreading. When the run method gets called though start() method then a new separate thread is being allocated to the execution of run method, so if more than one thread calls start() method that means their run method is being executed by separate threads (these threads run simultaneously).

Java defines two ways in which this can be accomplished:

1) By extending the **Thread** class.
2) By implement the **Runnable** interface.

2. Creating thread by using Thread class:

Create a new class that extends **Thread**, and then create an instance of that class. The extending class must override the **run( )** method, which is the entry point for the new thread. It must also call **start( )** to begin execution of the new thread.

```
class multithreading extends Thread

{
```

```
    public void run()
    {
       try
       {
          System.out.println ("Thread " +   Thread.currentThread().getId() + " is
                running");
        }
       catch (Exception e)
       {
           System.out.println ("Exception is caught");
       }
    }
}
 public class multithreadingdemo
{
    public static void main(String[] args)
    {
          multithreading m1=new multithreading();
          multithreading m2=new multithreading();
         m1.start();
        m2.start();
}
}
```

Output:

Thread 21 is running

Thread 22 is running

5. <u>Creating thread by using Runnable interface:</u>

The easiest way to create a thread is to create a class that implements the **Runnable** interface. To implement **Runnable**, a class need only implement a single method called **run( )**, which is declared like this:

public void run( )

Inside **run( )**, you will define the code that constitutes the new thread. It is important to understand that **run( )** can call other methods, use other classes, and declare variables, just like the main thread can. The only difference is that **run( )** establishes the entry point for another, concurrent thread of execution within your program. This thread will end when **run( )** returns.

After you create a class that implements **Runnable**, you will instantiate an object of type **Thread** from within that class. **Thread** defines several constructors. The one that we will use is shown here:

Thread(Runnable *threadOb)*

Thread(Runnable *threadOb*, String *threadName*)

Thread(String *threadName*)

In this constructor, *threadOb* is an instance of a class that implements the **Runnable** interface. This defines where execution of the thread will begin. The name of the new thread is specified by *threadName*.

After the new thread is created, it will not start running until you call its **start( )** method, which is declared within **Thread**. In essence, **start( )** executes a call to **run( )**. The **start( )** method is shown here:

void start( )

Here is an example that creates a new thread and starts it running:

```
class multithreading implements Runnable
{
  public void run()
   {
     try
      {
         System.out.println ("Thread " +   Thread.currentThread().getId() + " is
                  running");

      }
      catch (Exception e)
      {
          System.out.println ("Exception is caught");
```

```
    }
  }
}
 public class multithreadingdemo
{
   public static void main(String[] args)
   {
          multithreading m1=new multithreading();
              Thread t1 = new Thread(m1);
          multithreading m2=new multithreading();
         Thread t2 = new Thread(m2);
          t1.start();
          t2.start();
}
}
```

Output:

```
Thread 21 is running
Thread 22 is running
```

6. Develop a java program that provide synchronization for two threads deposit and withdraw in a bank application.
   (or) What is thread synchronization? Discuss with an example.
   Answer:

When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called *synchronization*. Java provides unique, language-level support for it.

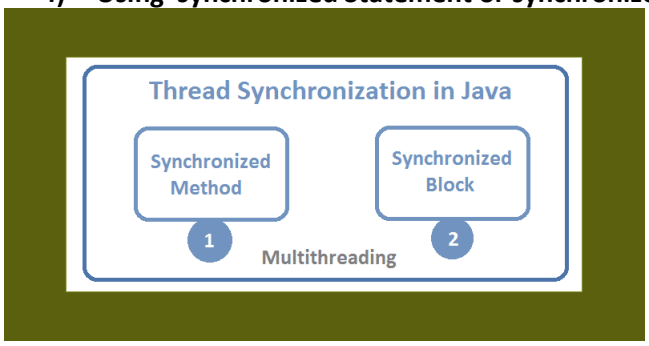
The synchronization is mainly used to

- To prevent thread interference.
- To prevent consistency problem.

Synchronization can be done in two ways:

3) Using Synchronized Methods

4) Using synchronized Statement or synchronized block



1)Using Synchronized Methods:

Synchronization is easy in Java, because all objects have their own implicit monitor associated with them. To enter an object's monitor, just call a method that has been modified with the **synchronized** keyword. While a thread is inside a synchronized method, all other threads that try to call it (or any other synchronized method) on the same instance have to wait. To exit the monitor and

relinquish control of the object to the next waiting thread, the owner of the monitor simply returns from the synchronized method.

Program:

```
class bank{
   int amount=5000;
synchronized void transaction(int n,char c ){
  try
  {
  if (c=='d')
{
System.out.println("Before deposit Balance ="+amount);
    amount =amount+n;
System.out.println("After deposit Balance ="+amount);
}
   else if(c=='w')  {
     System.out.println("Before withdraw Balance ="+amount);
     amount = amount-n;
```

```
     System.out.println("After withdraw Balance ="+amount);
}
Thread.sleep(400);
   }
    catch(Exception e)
{
System.out.println(e);}
  }
}
  class deposit extends Thread
{
bank t;
deposit(bank x){
t=x;
}
public void run(){
t.transaction(4000,'d');
}
}
class withdraw extends Thread{
bank t;
withdraw(bank x){
t=x;
}
public void run(){
t.transaction(2000,'w');
}
}

public class testsync{
public static void main(String args[]){
bank obj = new bank();
```

```
deposit t1=new deposit(obj);

withdraw t2=new withdraw(obj);

t1.start();

t2.start();

}

}
```

Output:

```
Before deposit Balance =5000
After deposit Balance =9000
Before withdraw Balance =9000
After withdraw Balance =7000
```

2)The synchronized Statement: (synchronized block)

While creating **synchronized** methods within classes that you create is an easy and effective means of achieving synchronization, it will not work in all cases. To understand why, consider the following. Imagine that you want to synchronize access to objects of a class that was not designed for multithreaded access. That is, the class does not use **synchronized** methods.

Further, this class was not created by you, but by a third party, and you do not have access to the source code. Thus, you can't add **synchronized** to the appropriate methods within the class. How can access to an object of this class be synchronized? Fortunately, the solution to this problem is quite easy: You simply put calls to the methods defined by this class inside a **synchronized** block.

This is the general form of the **synchronized** statement:

```
synchronized(object) {
  // statements to be synchronized
}
```

Here, *object* is a reference to the object being synchronized. A synchronized block ensures that a call to a method that is a member of *object* occurs only after the current thread has successfully entered *object*'s monitor.

Program:

```
class bank{
   int amount=5000;
void transaction(int n,char c ){
  try
```

```
  {
  if (c=='d')
{
System.out.println("Before deposit Balance ="+amount);
    amount =amount+n;
System.out.println("After deposit Balance ="+amount);
}
   else if(c=='w')  {
     System.out.println("Before withdraw Balance ="+amount);
     amount = amount-n;
     System.out.println("After withdraw Balance ="+amount);
}
Thread.sleep(400);
  }
    catch(Exception e){System.out.println(e);}
 }
}

class deposit extends Thread{
bank t;
deposit(bank x){
t=x;
}
public void run(){
synchronized(t)
{
t.transaction(4000,'d');
}
}
}

class withdraw extends Thread{
```

```
bank t;
withdraw(bank x){
t=x;
}
public void run(){
synchronized(t){
t.transaction(2000,'w');
}
}
}

public class testsync{
public static void main(String args[]){
bank obj = new bank();
deposit t1=new deposit(obj);
withdraw t2=new withdraw(obj);
t1.start();
t2.start();
}
}
```

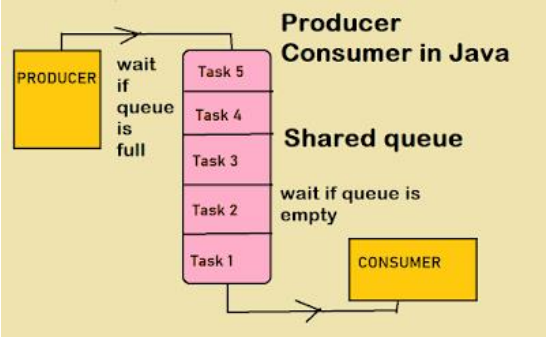
Output:

```
Before deposit Balance =5000
After deposit Balance =9000
Before withdraw Balance =9000
After withdraw Balance =7000
```

7. Write a java program for inventory problem to illustrate the usage of thread synchronized keyword and interthread communication process. They have three classes called consumer, producer and stock.
   (Or) Explain about interthread communication with example program.

- **Java includes an elegant interprocess communication mechanism via the wait( ), notify( ), and notifyAll( ) methods.** **Inter-thread communication is all about allowing synchronized threads to communicate with each other.** These methods are implemented as **final** methods in **Object**, so all classes have them. All three methods can be called only from within a **synchronized** context.

wait( ) tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls **notify( )**.
notify( ) wakes up a thread that called **wait( )** on the same object.
notifyAll( ) wakes up all the threads that called **wait( )** on the same object. One of the threads will be granted access.



Consider the following sample program that implements a simple form of the producer/ consumer problem. It consists of four classes: **Queue**, the queue that you're trying to synchronize; **Producer**, the threaded object that is producing queue entries; **Consumer**, the threaded object that is consuming queue entries; and **PC**, the tiny class that creates the single **Q**, **Producer**, and **Consumer**.

```
class Queue

{

int n;

boolean valueSet = false;


synchronized int get()

{

while(!valueSet)

try {

wait();

}

catch(InterruptedException e)

{

System.out.println("InterruptedException caught");

}

System.out.println("Got: " + n);

valueSet = false;

notify();
```

```
return n;
}



synchronized void put(int x)
 {
while(valueSet)
try {
wait();
}
catch(InterruptedException e)
{
System.out.println("InterruptedException caught");
}
n = x;
valueSet = true;
System.out.println("Put: " + n);
notify();
}
}

class Producer extends Thread {
   Queue q;
Producer(Queue q1) {
q = q1;
}
public void run() {
int i = 0;
while(true) {
q.put(i++);
}
```

```
}
}

class Consumer extends Thread{
   Queue q;
Consumer(Queue q1) {
q = q1;
}

public void run() {
while(true) {
q.get();
}
}
}
```

```
public class PCFixed
{
public static void main(String args[])
{
Queue q = new Queue();
Producer p=new Producer(q);
Consumer c= new Consumer(q);
p.start();
c.start();
System.out.println("Press Control-C to stop.");
}
}
```

Output:

Put: 1

Got: 1

Put: 2

Got: 2

Put: 3

Got: 3

Put: 4

Got: 4

Put: 5

Got: 5

10. Explain about creating multiple threads and thread priorities.

The program can create as many threads as it needs.

```
class multithreading extends Thread
{
   public void run()
   {
      try
      {
          System.out.println ("Thread " +   Thread.currentThread().getId() + " is
                running");
       }
      catch (Exception e)
      {
           System.out.println ("Exception is caught");
      }
```

```
   }
}
 public class multithreadingdemo
{
   public static void main(String[] args)
   {
         multithreading m1=new multithreading();
         multithreading m2=new multithreading();
m1.start();
m2.start();
}
}
```

Output:

```
Thread 21 is running
Thread 22 is running
```

Using isAlive( ) and join( )

Often you will want the main thread to finish last. This can be accomplished by calling **sleep( )** within **main( )**, with a long enough delay to ensure that all child threads terminate prior to the main thread. However, this is hardly a satisfactory solution, and it also raises a larger question: How can one thread know when another thread has ended? Fortunately, **Thread** provides a means by which you can answer this question.

Two ways exist to determine whether a thread has finished.

3) First, you can call **isAlive( )** on the thread. This method is defined by **Thread**, and its general form is shown here:

```
final boolean isAlive( )
```

The **isAlive( )** method returns **true** if the thread upon which it is called is still running. It returns **false** otherwise. **isAlive( )** is occasionally useful.

4) Most commonly used method to wait for a thread to finish is called **join( )**, shown here:

final void join( ) throws InterruptedException

This method waits until the thread on which it is called terminates. Its name comes from the concept of the calling thread waiting until the specified thread *joins* it. Additional forms of **join( )** allow you to specify a maximum amount of time that you want to wait for the specified thread to terminate.

```
class jointhread extends Thread{
   public void run() {
       try {
         System.out.println(Thread.currentThread().getName());
            Thread.sleep(100);
         }
         catch (InterruptedException e) {
           System.out.println("Thread interrupted");
         }

   }
}
public class JoinExample {
   public static void main(String[] args) {
      jointhread t1 = new jointhread();
      jointhread t2 = new jointhread();

      t1.start();
      t2.start();

      System.out.println("t1 Alive - " + t1.isAlive());
      System.out.println("t2 Alive - " + t2.isAlive());

try {
```

```
            t1.join();
            t2.join();
        }
        catch (InterruptedException e) {
            System.out.println("Thread interrupted");
        }
        System.out.println("t1 Alive - " + t1.isAlive());
        System.out.println("t2 Alive - " + t2.isAlive());
        System.out.println("Processing finished");
    }
}
```

Output:

```
Thread-0
Thread-1
t1 Alive - true
t2 Alive - true
t1 Alive - false
t2 Alive - false
Processing finished
```

THREAD PRIORITIES:

- Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run. In theory, higher-priority threads get more CPU time than lower-priority threads. In practice, the amount of CPU time that a thread gets often depends on several factors besides its priority. (For example, how an operating system implements multitasking can affect the relative availability of CPU time.)
- A higher-priority thread can also preempt a lower-priority one. For instance, when a lower-priority thread is running and a higher-priority thread resumes (from sleeping or waiting on I/O, for example), it will preempt the lower-priority thread.

To set a thread's priority, use the **setPriority( )** method, which is a member of **Thread**.

```
final void setPriority(int level)
```

Here, *level* specifies the new priority setting for the calling thread. The value of *level* must be within the range **MIN_PRIORITY** and **MAX_PRIORITY**. Currently, these values are 1 and 10, respectively. To return a thread to default priority, specify **NORM_PRIORITY**, which is currently 5. These priorities are defined as **static final** variables within **Thread**.

You can obtain the current priority setting by calling the **getPriority( )** method of **Thread**, shown here:

```
final int getPriority( )
```

12. Explain about deadlock .

- Deadlock occurs when two threads have a circular dependency on a pair of synchronized objects.
- For example, suppose one thread enters the monitor on object X and another thread enters the monitor on object Y.
- If the thread in X tries to call any synchronized method on Y, it will block as expected. However, if the thread in Y, in turn, tries to call any synchronized method on X, the thread waits forever, because to access X, it would have to release its own lock on Y so that the first thread could complete. Deadlock is a difficult error to debug for two reasons:

3) In general, it occurs only rarely, when the two threads time-slice in just the right way.
4) It may involve more than two threads and two synchronized objects.

```
// Online Java Compiler

// Use this editor to write, compile and run your Java code online


public class TestThread

{

  static String r1 = "java";

  static String r2 = "program";


  public static void main(String args[]) {

    ThreadDemo1 T1 = new ThreadDemo1();

    ThreadDemo2 T2 = new ThreadDemo2();

    T1.start();

    T2.start();

  }
```

```
static class ThreadDemo1 extends Thread
{
    public void run()
    {
        synchronized (r1)
        {
          System.out.println("Thread 1: Holding lock 1...");

          System.out.println("Thread 1: Waiting for lock 2...");

          synchronized (r2)
          {
            System.out.println("Thread 1: Holding lock 1 & 2...");
          }
        }
     }
  }
  static class ThreadDemo2 extends Thread
  {
     public void run()
     {
        synchronized (r2)
        {
          System.out.println("Thread 2: Holding lock 2...");

          System.out.println("Thread 2: Waiting for lock 1...");

          synchronized (r1)
          {
            System.out.println("Thread 2: Holding lock 1 & 2...");
          }
```

```
    }
  }
 }}
```

Output

Thread 1: Holding lock 1...

Thread 2: Holding lock 2...

Thread 1: Waiting for lock 2...

Thread 2: Waiting for lock 1...

As this example illustrates, if your multithreaded program locks up occasionally, deadlock is one of the first conditions that you should check for.

13. Explain about suspending, resuming, and stopping threads.

- ✓ **stop**() method in **Thread** – This method terminates the **thread** execution.
- ✓ **suspend**() method in **Thread** – If you want to **stop** the **thread** execution and start it again when a certain event occurs. ...
- ✓ **resume**() method in **Thread** – **resume**() method works with **suspend**() method.

- The mechanisms to suspend, stop, and resume threads differ between early versions of Java, such as Java 1.0, and more modern versions, beginning with Java 2. Prior to Java 2, a program used **suspend( )**, **resume( )**, and **stop( )**, which are methods defined by **Thread**, to pause, restart, and stop the execution of a thread. Although these methods seem to be a perfectly reasonable and convenient approach to managing the execution of threads, they must not be used for new Java programs.

- Here's why. The **suspend( )** method of the **Thread** class was deprecated by Java 2 several years ago. This was done because **suspend( )** can sometimes cause serious system failures. Assume that a thread as obtained locks on critical data structures. If that thread is suspended at that point, those locks are not relinquished. Other threads that may be waiting for those resources can be deadlocked.
- The **resume( )** method is also deprecated. It does not cause problems, but cannot be used without the **suspend( )** method as its counterpart. The **stop( )** method of the **Thread** class, too, was deprecated by Java 2. This was done because this method can sometimes cause serious system failures.
- Assume that a thread is writing to a critically important data structure and has completed only part of its changes. If that thread is stopped at that point, that data structure might be left in a corrupted state. The trouble is that **stop( )** causes any lock the calling thread holds to be released. Thus, the corrupted data might be used by another thread that is waiting on the same lock.
- Because you can't now use the **suspend( )**, **resume( )**, or **stop( )** methods to control a thread, you might be thinking that no way exists to pause, restart, or terminate a thread.
- But, fortunately, this is not true. Instead, a thread must be designed so that the **run( )** method periodically checks to determine whether that thread should suspend, resume, or stop its own execution. This is accomplished by establishing a flag variable that indicates the execution state of the thread. As long as this flag is set to "running," the **run( )** method must continue to let the thread execute. If this variable is set to "suspend," the thread must pause. If it is set to "stop," the thread must terminate. Of course, a variety of ways exist in which to write such code, but the central theme will be the same for all programs.

- The following example illustrates how the **wait( )** and **notify( )** methods that are inherited from **Object** can be used to control the execution of a thread. Let us consider its operation. The **NewThread** class contains a **boolean** instance variable named **suspendFlag**, which is used to control the execution of the thread. It is initialized to **false** by the constructor. The **run( )** method contains a **synchronized** statement block that checks **suspendFlag**. If that variable is **true**, the **wait( )** method is invoked to suspend the execution of the thread. The **mysuspend( )** method sets **suspendFlag** to **true**. The **myresume( )** method sets **suspendFlag** to **false** and invokes **notify( )** to wake up the thread. Finally, the **main( )** method has been modified to invoke the **mysuspend( )** and **myresume( )** methods.

```
class NewThread implements Runnable

{

String name;

Thread t;

boolean suspendFlag;

NewThread(String threadname)

 {

name = threadname;

t = new Thread(this, name);

System.out.println("New thread: " + t);

suspendFlag = false;

t.start();

}


public void run()

{

try

 {

for(int i = 3; i > 0; i--)

{

System.out.println(name + ": " + i);

Thread.sleep(200);

synchronized(this)

 {

while(suspendFlag)

{
```

```
wait();
}
}
}
}
 catch (InterruptedException e)
{
System.out.println(name + " interrupted.");
}
System.out.println(name + " exiting.");
}
void mysuspend()
{
suspendFlag = true;
}
synchronized void myresume()
{
suspendFlag = false;
notify();
}
}

class SuspendResume
 {
public static void main(String args[])
{
NewThread ob1 = new NewThread("One");
NewThread ob2 = new NewThread("Two");
try
 {
Thread.sleep(1000);
ob1.mysuspend();
```

```
System.out.println("Suspending thread One");
Thread.sleep(1000);
ob1.myresume();
System.out.println("Resuming thread One");
ob2.mysuspend();
System.out.println("Suspending thread Two");
Thread.sleep(1000);
ob2.myresume();
System.out.println("Resuming thread Two");
}
catch (InterruptedException e)
 {
System.out.println("Main thread Interrupted");
}
// wait for threads to finish
try
{
System.out.println("Waiting for threads to finish.");
ob1.t.join();
ob2.t.join();
}
 catch (InterruptedException e)
 {
System.out.println("Main thread Interrupted");
}
System.out.println("Main thread exiting.");
}
}
```

Output:

New thread: Thread[One,5,main]

New thread: Thread[Two,5,main]

One: 3

Two: 3

One: 2

Two: 2

One: 1Two: 1

One exiting.

Two exiting.

Suspending thread One

Resuming thread One

Suspending thread Two

Resuming thread Two

Waiting for threads to finish.

Main thread exiting.

14. Explain about type wrappers in java.

Java uses primitive types (also called simple types), such as **int** or **double**, to hold the basic data types supported by the language. Primitive types, rather than objects, are used for these quantities for the sake of performance. Using objects for these values would add an unacceptable overhead to even the simplest of calculations. Thus, the primitive types are not part of the object hierarchy, and they do not inherit **Object**.

Despite the performance benefit offered by the primitive types, there are times when you will need an object representation. For example, you can't pass a primitive type by reference to a method. Also, many of the standard data structures implemented by Java operate on objects, which means that you can't use these data structures to store primitive types. To handle these situations, Java provides *type wrappers,* which are classes that encapsulate a primitive type within an object.

The type wrappers are **Double**, **Float**, **Long**, **Integer**, **Short**, **Byte**, **Character**, and **Boolean**. These classes offer a wide array of methods that allow you to fully integrate the primitive types into Java's object hierarchy.

Character:

Character is a wrapper around a **char**. The constructor for **Character** is Character(char *ch*)

Here, *ch* specifies the character that will be wrapped by the **Character** object being created.

To obtain the **char** value contained in a **Character** object, call **charValue( )**, shown here:

char charValue( )

It returns the encapsulated character.

Boolean

Boolean is a wrapper around **boolean** values. It defines these constructors:

Boolean(boolean *boolValue*)

Boolean(String *boolString*)

In the first version, *boolValue* must be either **true** or **false**. In the second version, if *boolString* contains the string "true" (in uppercase or lowercase), then the new **Boolean** object will be true. Otherwise, it will be false.

To obtain a **boolean** value from a **Boolean** object, use **booleanValue( )**, shown here:

boolean booleanValue( )

It returns the **boolean** equivalent of the invoking object.

The Numeric Type Wrappers

By far, the most commonly used type wrappers are those that represent numeric values. These are **Byte**, **Short**, **Integer**, **Long**, **Float**, and **Double**. All of the numeric type wrappers inherit the abstract class **Number**. **Number** declares methods that return the value of an object in each of the different number formats. These methods are shown here:

byte byteValue( )

double doubleValue( )

float floatValue( )

int intValue( )

long longValue( )

short shortValue( )

For example, **doubleValue( )** returns the value of an object as a **double**, **floatValue( )** returns the value as a **float**, and so on. These methods are implemented by each of the numeric type wrappers.

All of the numeric type wrappers define constructors that allow an object to be constructed from a given value, or a string representation of that value. For example, here are the constructors defined for **Integer**:

Integer(int *num*)

Integer(String *str*)

If *str* does not contain a valid numeric value, then a **NumberFormatException** is thrown. All of the type wrappers override **toString( )**. It returns the human-readable form of the value contained within the wrapper. This allows you to output the value by passing a type wrapper object to **println( )**, for example, without having to convert it into its primitive type.The following program demonstrates how to use a numeric type wrapper to encapsulate a value and then extract that value.

```
// Demonstrate a type wrapper.

class Wrap {

public static void main(String args[]) {

Integer iOb = new Integer(100);

int i = iOb.intValue();

System.out.println(i + " " + iOb); // displays 100 100

}

}
```

This program wraps the integer value 100 inside an **Integer** object called **iOb**. The program then obtains this value by calling **intValue( )** and stores the result in **i**.

The process of encapsulating a value within an object is called *boxing.* Thus, in the program,

this line boxes the value 100 into an **Integer**:

```
Integer iOb = new Integer(100);
```

The process of extracting a value from a type wrapper is called *unboxing*. For example, the

program unboxes the value in **iOb** with this statement:

```
int i = iOb.intValue();
```

The same general procedure used by the preceding program to box and unbox values has been employed since the original version of Java. However, with the release of JDK 5, Java

fundamentally improved on this through the addition of autoboxing, described next.

15. Explain about autoboxing and unboxing.

Beginning with JDK 5, Java added two important features:

autoboxing and *auto-unboxing.*

Autoboxing is the process by which a primitive type is automatically encapsulated (boxed) into its equivalent type wrapper whenever an object of that type is needed. There is no need to explicitly construct an object.

Auto-unboxing is the process by which the value of a boxed object is automatically extracted (unboxed) from a type wrapper when its value is needed. There is no need to call a method such as **intValue( )** or **doubleValue( )**.

The addition of autoboxing and auto-unboxing greatly streamlines the coding of several algorithms, removing the tedium of manually boxing and unboxing values. It also helps prevent errors. Moreover, it is very important to generics, which operates only on objects. Finally, autoboxing makes working with the Collections Framework much easier.

With autoboxing it is no longer necessary to manually construct an object in order to wrap a primitive type. You need only assign that value to a type-wrapper reference. Java automatically constructs the object for you. For example, here is the modern way to construct an **Integer** object that has the value 100:

```
Integer iOb = 100;                                    // autobox an int
```

Notice that no object is explicitly created through the use of **new**. Java handles this for you, automatically.

To unbox an object, simply assign that object reference to a primitive-type variable. For example, to unbox **iOb**, you can use this line:

```
int i = iOb;                                          // auto-unbox
```

Java handles the details for you.

Here is the preceding program rewritten to use autoboxing/unboxing:

```
// Demonstrate autoboxing/unboxing.

class AutoBox
```

```
 {
public static void main(String args[])
{
Integer iOb = 100; // autobox an int
int i = iOb; // auto-unbox
System.out.println(i + " " + iOb);                    // displays 100 100
}
}
```

Autoboxing/Unboxing Boolean and Character Values

Java also supplies wrappers for **boolean** and **char**. These are **Boolean** and **Character**. Autoboxing/unboxing applies to these wrappers, too. For example, consider the following program:

```
// Autoboxing/unboxing a Boolean and Character.

class AutoBox5
{
public static void main(String args[])
{
Boolean b = true;
if(b)
 System.out.println("b is true");
Character ch = 'x';                                          // box a char
char ch2 = ch;                                               // unbox a char
System.out.println("ch2 is " + ch2);
}
}
```

The output is shown here:

```
b is true
ch2 is x
```

CS3391/OBJECT ORIENTED PROGRAMMING

Unit-III

Part-A

2. What is an exception?

Exception is an abnormal condition which occurs during the execution of a program and disrupts normal flow of the program. This exception must be handled properly. If it is not handled, program will be terminated abruptly.

3. How the exceptions are handled in java? OR Explain exception handling mechanism in java?

Exceptions in java are handled using try, catch and finally blocks.

try block : The code or set of statements which are to be monitored for exception are kept in this block.

catch block : This block catches the exceptions occurred in the try block.

finally block : This block is always executed whether exception is occurred in the try block or not and occurred exception is caught in the catch block or not.

4. What is the difference between error and exception in java?

Errors are mainly caused by the environment in which an application is running. For example, OutOfMemoryError happens when JVM runs out of memory. Where as exceptions are mainly caused by the application itself. For example, NullPointerException occurs when an application tries to access null object.

5. Can we write only try block without catch and finally blocks?

No, It shows compilation error. The try block must be followed by either catch or finally block.

You can remove either catch block or finally block but not both.

5. What is unreachable catch block error?

When you are keeping multiple catch blocks, the order of catch blocks must be from most specific to most general ones. i.e sub classes of Exception must come first and super classes later. If you keep super classes first and sub classes later, compiler will show unreachable catch block error.

7. Describe the hierarchy of exceptions in java?

All exception types are subclasses of the built-in class Throwable. Thus, Throwable is at the top of the exception class hierarchy.

Immediately below Throwable are two subclasses that partition exceptions into two distinct branches.

One branch is headed by Exception. This class is used for exceptional conditions that user programs should catch. This is also the class that you will subclass to create your own custom exception types. There is an important subclass of Exception, called RuntimeException. Exceptions of this type are automatically defined for the programs that you write and include things such as division by zero and invalid array indexing.

The other branch is topped by Error, which defines exceptions that are not expected to be caught under normal circumstances by your program. Exceptions of type Error are used by the Java run-time system to indicate errors having to do with the run-time environment, itself. Stack overflow is an example of such an error.

8. What are run time exceptions in java. Give example?

The exceptions which occur at run time are called as run time exceptions. These exceptions are unknown to compiler. All sub classes of java.lang.RunTimeException and java.lang.Error are run time exceptions. These exceptions are unchecked type of exceptions. For example, NumberFormatException, NullPointerException, ClassCastException,

9. What are checked and unchecked exceptions in java?

Checked exceptions are the exceptions which are known to compiler. These exceptions are checked at compile time only. Hence the name checked exceptions. These exceptions are also called compile time exceptions. Because, these exceptions will be known during compile time.

Unchecked exceptions are those exceptions which are not at all known to compiler. These exceptions occur only at run time. These exceptions are also called as run time exceptions. All sub classes of java.lang.RunTimeException and java.lang.Error are unchecked exceptions.

10. What is Re-throwing an exception in java?

Exceptions raised in the try block are handled in the catch block. If it is unable to handle that exception, it can re-throw that exception using throw keyword. It is called re-throwing an exception.

11. What is the use of throws keyword in java?

If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception. You do this by including a throws clause in the method's declaration. A throws clause lists the types of exceptions that a method might throw. This is necessary for all exceptions, except those of type Error or RuntimeException, or any of their subclasses. All other exceptions that a method can throw must be declared in the throws clause. If they are not, a compile-time error will result.

12. What is the difference between final, finally and finalize in java?

final keyword :

final is a keyword which is used to make a variable or a method or a class as "unchangeable".

finally Block :

finally is a block which is used for exception handling along with try and catch blocks. finally block is always executed whether exception is raised or not and raised exception is handled or not. Most of time, this block is used to close the resources like database connection, I/O resources etc.

finalize() Method :

finalize() method is a protected method of java.lang.Object class. It is inherited to every class you create in java. This method is called by garbage collector thread before an object is removed from the memory. finalize() method is used to perform some clean up operations on an object before it is removed from the memory.

13. How do you create customized exceptions in java?

In java, we can define our own exception classes as per our requirements. These exceptions are called user defined exceptions in java OR Customized exceptions. User defined exceptions must extend any one of the classes in the hierarchy of exceptions.

14. What is the difference between throw, throws and throwable in java?

throw In Java :

throw is a keyword in java which is used to throw an exception manually. Using throw keyword, you can throw an exception from any method or block. But, that exception must be of type java.lang.Throwable class or it's sub classes. Below example shows how to throw an exception using throw keyword.

throws In Java :

throws is also a keyword in java which is used in the method signature to indicate that this method may throw mentioned exceptions. The caller to such methods must handle the mentioned exceptions either using try-catch blocks or using throws keyword. Below is the syntax for using throws keyword.

Throwable In Java :

Throwable is a super class for all types of errors and exceptions in java. This class is a member of java.langpackage. Only instances of this class or it's sub classes are thrown by the java virtual machine or by the throw statement. The only argument of catch block must be of this type or it's sub classes. If you want to create your own customized exceptions, then your class must extend this class. Click here to see the hierarchy of exception classes in java.

26. Which class is the super class for all types of errors and exceptions in java?

java.lang.Throwable is the super class for all types of errors and exceptions in java.

27. Define Process-based multitasking: (Multiprocessing)

A process is a program that is executing. Thus, process-based multitasking is the feature that allows your computer to run two or more programs concurrently. For example, process-based multitasking enables you to run the Java compiler at the same time that you are using a text editor or visiting a web site. In process-based multitasking, a program is the smallest unit of code that can be dispatched by the scheduler.

28. Define Thread-based multitasking: (Multithreading)

In a thread-based multitasking environment, the thread is the smallest unit of dispatchable code. This means that a single program can perform two or more tasks simultaneously. For instance, a text editor can format text at the same time that it is printing, as long as these two actions are being performed by two separate threads. Thus, process-based multitasking deals with the "big picture," and thread-based multitasking handles the details.

29. Differentiate between multitasking and multithreading.

Sl.no.	Multitasking	Multithreading
1	It is a process of executing many processes running simultaneously.	It is a process of executing multiple threads(sub-process).
2	Process is program in execution. They are Heavy weight.	**Thread** is basically a lightweight sub-process. It is a smallest unit of processing.

3	In multi tasking separate memory is allocated to each process.	Threads(sub-process) are sharing common memory.
4	Interprocess communication is expensive	Inter thread communication is inexpensive .
5	Context switching from one process to another is costly.	Context switching from one thread to another is inexpensive.

30. Define Multithreading.

Multithreading enables to write efficient programs that make maximum use of the processing power available in the system. One important way multithreading achieves this is by keeping idle time to a minimum.

31. **List are the States of thread.**

- A thread can be **running**. It can be ready to run as soon as it gets CPU time.
- A running thread can be **suspended**, which temporarily halts its activity.
- A suspended thread can then be **resumed**, allowing it to pick up where it left off.
- A thread can be **blocked** when waiting for a resource.
- At any time, a thread can be **terminated**, which halts its execution immediately. Once terminated, a thread cannot be resumed.

32. How can you create thread in Java?

Java defines two ways for creating thread.

1) By extending the **Thread** class.

2) By implement the **Runnable** interface.

33. What do you mean by synchronization?

When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called synchronization. Java provides unique, language-level support for it.

Synchronization can be done in two ways:

3) Using Synchronized Methods
4) Using synchronized Statement or synchronized block

34. What is the need for synchronization?

The synchronization is mainly used to

- To prevent thread interference.
- To prevent consistency problem.

35. What is meant by monitor?

A **monitor** is an object that is used as a mutually exclusive lock. Only one thread can own a monitor at a given time. When a thread acquires a lock, it is said to have entered the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor. These other threads are said to be waiting for the monitor. A thread that owns a monitor can reenter the same monitor if it so desires.

36. How will you set the priority of the thread?

Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run. In theory, higher-priority threads get more CPU time than lower-priority threads.

To set a thread's priority, use the **setPriority( )** method.

final void setPriority(int *level*)

The value of *level* must be within the range **MIN_PRIORITY** and **MAX_PRIORITY**. Currently, these values are 1 and 10, respectively. To return a thread to default priority, specify **NORM_PRIORITY**, which is currently 5.

Part-B

23. Develop a java program for handling divide by zero exception with multiple catches.
24. Develop a java program for handling ArrayIndexOutOfBoundsException exception with finally block.

25. Develop a java program for exception handling with throw and throws keyword.
26. Explain nested try/catch with example program.
27. Develop a java program with multiple catch clauses.
28. Develop a java program with multicatch that can handle both ArithmeticException and IndexOutOfBoundsException.

29. Discuss about exception handling in nested try/catch.
30. Explain about types of exception with an example.
31. Develop a Java program to implement user defined exception handling.
32. Discuss the life cycle of threads with neat diagram.
33. Develop a program for creating threads by using Thread class and Runnable interface.
34. Why do we need both start() and run() method both? can we achieve it with only run() method?
35. Write a java program that synchronize three threads of the same program and display the content the text supplied through these threads.
36. Develop a java program that provide synchronization for two threads deposit and withdraw in a bank application.
37. Define thread. Explain the state of threads. State reason for synchronization in thread. Write simple concurrent programming to create, sleep and delete thread.
38. What is thread synchronization? Discuss with an example.
39. Write a java program for inventory problem to illustrate the usage of thread synchronized keyword and interthread communication process. They have three classes called consumer, producer and stock.
40. Write a java program that implements a multi-threaded application that has three threads. First thread generates a random integer every 1 second and if the value is even, second thread computes the square of the number and prints. If the value is odd, the third thread will print the value of cube of the number.
41. Develop a java application program for generating four threads to perform the following operation.
    v) Getting N numbers as input
    vi) Printing even numbers
    vii) Printing odd numbers.
    viii) Computing the average
42. Explain about wrapper classes.
43. Elaborate in detail about autoboxing.
44. Discuss about Suspending, Resuming, and Stopping Threads.