UNIT-IV-
Learning Material(R-17)
Exception Handling and Multithreading

Objective:

To familiarize the concepts of Exception Handling and Multithreading.

Syllabus:

Exception Handling- exception-handling fundamentals, uncaught exceptions, using try and catch, multiple catch clauses, nested try statements, throw, throws, finally, user-defined exceptions.

Multithreading-Introduction to multitasking, thread life cycle, creating threads, synchronizing threads, thread groups.

Learning Outcomes

Upon successful completion of the course, the students will be able to

- Understand the concepts and applications of exception handling.
- Apply exception handle mechanism to handle run time errors in java.
- Write a program to handle multiple exception.
- Create user defined exception.
- Understand threads concepts and its life cycle in java.
- Understand how multiple threads can be created within java program.
- Apply threads concept to an application.

Learning Material

➢ EXCEPTION-HANDLING FUNDAMENTALS:

- 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.

- Java exception handling is managed via five keywords: **try**, **catch**, **throw**, **throws**, and **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 runtime 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 before a method returns is put in a **finally** block.

This is the general form of an exception-handling block:

```
try {
// block of code to monitor for errors
}
```

```
catch (ExceptionType1 exOb) {
// exception handler for ExceptionType1
}
catch (ExceptionType2 exOb) {
// exception handler for ExceptionType2
}
// ...
finally {
// block of code to be executed before try block ends
}
```

Here, *ExceptionType* is the type of exception that has occurred. The remainder of this chapter describes how to apply this framework.

- 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.
- Exceptions are broadly classified into two categories

- Checked Exceptions: Checked Exceptions are those for which the compiler checks to see whether they have been handled in your programs or not. These Exceptions are not sub classes of class RuntimeException.
- Unchecked Exceptions: Run –Time exceptions are not checked by the compiler.These Exceptions are derived from class RuntimeException.

➢ UNCAUGHT EXCEPTIONS

- Before you learn how to handle exceptions in your program, it is useful to see what happens when you don't handle them. This small program includes an expression that intentionally causes a divide-by-zero error.

```
class Exc0 {
public static void main(String args[]) {
int d = 0;
int a = 42 / d;
}
}
```

- When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then *throws* this exception. This causes the execution of **Exc0** to stop, because once an exception has been thrown, it must be *caught* by an exception handler and dealt with immediately.
- Here is the output generated when this example is executed by the standard Java JDK run-time interpreter:

```
java.lang.ArithmeticException: / by zero at Exc0.main(Exc0.java:4)
```

- Notice how the class name, **Exc0**; the method name, **main**; the filename, **Exc0.java**; and the line number, **4**, are all included in the simple stack trace.

- The stack trace will always show the sequence of method invocations that led up to the error. For example, here is another version of the preceding program that introduces the same error but in a method separate from **main( )**:

```
class Exc1 {
static void subroutine() {
int d = 0;
int a = 10 / d;
}
public static void main(String args[]) {
Exc1.subroutine();
}
}
```

- The resulting stack trace from the default exception handler shows how the entire call stack is displayed:

```
java.lang.ArithmeticException: / by zero
at Exc1.subroutine(Exc1.java:4)
at Exc1.main(Exc1.java:7)
```

- As you can see, the bottom of the stack is **main**'s line 7, which is the call to **subroutine( )**,
  which caused the exception at line 4. The call stack is quite useful for debugging, because it pinpoints the precise sequence of steps that led to the 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. Doing so provides two benefits. First, it allows you to fix the error. Second, it prevents the program from automatically terminating.

- To guard against and handle a run-time error, simply enclose the code that you want to monitor inside a **try** block. Immediately following the **try** block, include a **catch** clause that specifies the exception type that you wish to catch.

- To illustrate how easily this can be done, the following program includes a **try** block and a **catch** clause which processes the **ArithmeticException** generated by the division-by-zero error:

```
class Exc2 {
public static void main(String args[]) {
int d, a;
try { // monitor a block of code.
d = 0;
a = 42 / d;
System.out.println("This will not be printed.");
} catch (ArithmeticException e) { // catch divide-by-zero error
System.out.println("Division by zero.");
}
System.out.println("After catch statement.");
}
}
```

- This program generates the following output:

  Division by zero.

  After catch statement.

- Notice that the call to **println( )** inside the **try** block is never executed. Once an exception is thrown, program control transfers out of the **try** block into the **catch** block. Put differently, **catch** is not "called," so execution never "returns" to the **try** block from a **catch**.

- Thus, the line "This will not be printed." is not displayed. Once the **catch** statement has executed, program control continues with the next line in the program following the entire **try**/**catch** mechanism. A **try** and its **catch** statement form a unit. The scope of the **catch** clause is restricted to those statements specified by the immediately preceding **try** statement.

➢ 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 MultiCatch {
public static void main(String args[]) {
try {
int a = args.length;
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.");
}
}
```

- This program will cause a division-by-zero exception if it is started with no command-line parameters, since **a** will equal zero. It will survive the division if you provide a commandline argument, setting **a** to something larger than zero.
- But it will cause an **ArrayIndexOutOfBoundsException**, since the **int** array **c** has a length of 1, yet the program attempts to assign a value to **c[42]**.

Here is the output generated by running it both ways:

```
C:\\>java MultiCatch
a = 0
Divide by 0: java.lang.ArithmeticException: / by zero
```

```
After try/catch blocks.
C:\\>java MultiCatch TestArg
a = 1
Array index oob: java.lang.ArrayIndexOutOfBoundsException:
42
After try/catch blocks.
```

- When you use multiple **catch** statements, it is important to remember that exception subclasses must come before any of their super classes.

- This is because a **catch** statement that uses a super class will catch exceptions of that type plus any of its subclasses.

- Thus, a subclass would never be reached if it came after its super class. Further, in Java, unreachable code is an error. For example, consider the following program:

```
class SuperSubCatch {
public static void main(String args[]) {
try {
int a = 0;
int b = 42 / a;
} catch(Exception e) {
System.out.println("Generic Exception catch.");
}
catch(ArithmeticException e) { // ERROR - unreachable
System.out.println("This is never reached.");
}
}
}
```

- If you try to compile this program, you will receive an error message stating that the second **catch** statement is unreachable.

- Since **ArithmeticException** is a subclass of **Exception**, the first **catch** statement will handle all **Exception**-based errors, including **ArithmeticException**.

- This means that the second **catch** statement will never execute. To fix the problem, reverse the order of the **catch** statements.

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 = args.length;
int b = 42 / a;
```

```
System.out.println("a = " + a);
try { // nested try block

if(a==1) a = a/(a-a); // division by zero

if(a==2) {
int c[] = { 1 };
c[42] = 99; // generate an out-of-bounds exception
}
} catch(ArrayIndexOutOfBoundsException e) {
System.out.println("Array index out-of-bounds: " + e);

}
} catch(ArithmeticException e) {
System.out.println("Divide by 0: " + e);
}
}
}
```

The program works as follows.

- When you execute the program with no command-line arguments, a divide-byzero exception is generated by the outer **try** block. Execution of the program by 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. Here are sample runs that illustrate each case:

```
C:\\>java NestTry
Divide by 0: java.lang.ArithmeticException: / by zero
```

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

- Nesting of **try** statements can occur in less obvious ways when method calls are involved. For example, you can enclose a call to a method within a **try** block. Inside that method is another **try** statement. In this case, the **try** within the method is still nested inside the outer **try** block, which calls the method. Here is the previous program recoded so that the nested **try** block is moved inside the method **nesttry( )**:

```
class MethNestTry {
static void nesttry(int a) {
try { // nested try block

if(a==1) a = a/(a-a); // division by zero

if(a==2) {
int c[] = { 1 };
c[42] = 99; // generate an out-of-bounds exception
}
} catch(ArrayIndexOutOfBoundsException e) {
System.out.println("Array index out-of-bounds: " + e);
}
}
public static void main(String args[]) {
```

```
try {
int a = args.length;

int b = 42 / a;
System.out.println("a = " + a);
nesttry(a);
} catch(ArithmeticException e) {
System.out.println("Divide by 0: " + e);
}
}
}
```

THROW:

- So far, you have only been catching exceptions that are thrown by the Java run-time system. However, it is possible for your program to throw an exception explicitly, using the **throw** statement. The general form of **throw** is shown here:

  throw *ThrowableInstance*;

- Here, *ThrowableInstance* must be an object of type **Throwable** or a subclass of **Throwable**. Simple types, such as **int** or **char**, as well as non-**Throwable** classes, such as **String** and **Object**, cannot be used as exceptions. There are two ways you can obtain a **Throwable** object: using a parameter into a **catch** clause, or creating one with the **new** operator.

- The flow of execution stops immediately after the **throw** statement; any subsequent statements are not executed. The nearest enclosing **try** block is inspected to see if it has a **catch** statement that matches the type of the exception.

- If it does find a match, control is transferred to that statement. If not, then the next enclosing **try** statement is inspected, and so on. If no matching **catch** is found, then the default exception handler halts the program and prints the stack trace.

- Here is a sample program that creates and throws an exception. The handler that catches the exception rethrows it to the outer handler.

```
// Demonstrate throw.
class ThrowDemo {
static void demoproc() {
try {
throw new NullPointerException("demo");
} catch(NullPointerException e) {
System.out.println("Caught inside demoproc.");
throw e; // rethrow the exception
}
}
public static void main(String args[]) {
try {
demoproc();
} catch(NullPointerException e) {
System.out.println("Recaught: " + e);
}
}
}
```

- This program gets two chances to deal with the same error. First, **main( )** sets up an exception context and then calls **demoproc( )**.

- The **demoproc( )** method then sets up another exception-handling context and immediately throws a new instance of **NullPointerException,** which is caught on the next line. The exception is then rethrown.

Here is the resulting output:

Caught inside demoproc.
Recaught: java.lang.NullPointerException: demo
The program also illustrates how to create one of Java's standard exception objects. Pay
close attention to this line:
throw new NullPointerException("demo");

- Here, **new** is used to construct an instance of **NullPointerException**. All of Java's built-in run-time exceptions have two constructors: one with no parameter and one that takes a string parameter.

- When the second form is used, the argument specifies a string that describes the exception. This string is displayed when the object is used as an argument to **print( )** or **println( )**. It can also be obtained by a call to **getMessage( )**, which is defined by **Throwable**.

➢ THROWS:

- If a method is capable of causing an exception that it does not handle, it must specify this behaviour 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. This is the general form of a method declaration that includes a **throws** clause:

  type method-name(parameter-list) throws *exception-list*
  {
  // body of method
  }

- Here, *exception-list* is a comma-separated list of the exceptions that a method can throw. Following is an example of an incorrect program that tries to throw an exception that it does not catch. Because the program does not specify a **throws** clause to declare this fact, the program will not compile.

```
// This program contains an error and will not compile.
class ThrowsDemo {
static void throwOne() {
System.out.println("Inside throwOne.");
throw new IllegalAccessException("demo");
}
public static void main(String args[]) {
throwOne();
}
}
```

- To make this example compile, you need to make two changes. First, you need to declare that **throwOne( )** throws **IllegalAccessException**. Second, **main( )** must define a **try**/**catch** statement that catches this exception.

The corrected example is shown here:

```
// This is now correct.
class ThrowsDemo {
static void throwOne() throws IllegalAccessException {
System.out.println("Inside throwOne.");
throw new IllegalAccessException("demo");
}
public static void main(String args[]) {
try {
throwOne();
} catch (IllegalAccessException e) {
System.out.println("Caught " + e);
}
}
}
```

Here is the output generated by running this example program:

```
inside throwOne
caught java.lang.IllegalAccessException: demo
```

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.

- Any time a method is about to return to the caller from inside a **try/catch** block, via an uncaught exception or an explicit return statement, the **finally** clause is also executed just before the method returns.

- This can be useful for closing file handles and freeing up any other resources that might have been allocated at the beginning of a method with the intent of disposing of them before returning.

- The **finally** clause is optional. However, each **try** statement requires at least one **catch** or a **finally** clause. Here is an example program that shows three methods that exit in various ways, none without executing their **finally** clauses:

class FinallyDemo
{
static void procA()
{
try {
	// Execute a try block normally.
	static void procC()
	{
	try {
	System.out.println("inside procC");

System.out.println("inside procA");
throw new RuntimeException("demo");
} finally
{
 	
System.out.println("procA's finally");
} }
// Return from within a try block.
 	
static void procB() {
try {
System.out.println("inside procB");
return;
} finally
{
System.out.println("procB's finally");
}
}
	} finally {
	System.out.println("procC's finally");
	} }
	public static void main(String args[])
	{
	try {
	procA();
	}
	catch (Exception e)
	{
	System.out.println("Exception caught");
	}
	procB();
	procC();
	}
	}

- In this example, **procA( )** prematurely breaks out of the **try** by throwing an exception. The **finally** clause is executed on the way out. **procB( )**'s **try** statement is exited via a **return** statement.

- The **finally** clause is executed before **procB( )** returns. In **procC( )**, the **try** statement executes normally, without error. However, the **finally** block is still executed.

Note : If a **finally** block is associated with a **try**, the **finally** block will be executed upon conclusion of the **try**. Here is the output generated by the preceding program:

```
inside procA
procA's finally
Exception caught
inside procB
procB's finally
inside procC
procC's finally.
```

➢ USER-DEFINED EXCEPTIONS

- ➢ Inside the standard package **java.lang**, Java defines several exception classes. A few have been used by the preceding examples. The most general of these exceptions are subclasses of the standard type **RuntimeException**.

- ➢ Since **java.lang** is implicitly imported into all Java programs, most exceptions derived from **RuntimeException** are automatically available. Furthermore, they need not be included in any method's **throws** list.
- ➢ In the language of Java, these are called *unchecked exceptions* because the compiler does not check to see if a method handles or throws these exceptions. The unchecked exceptions defined in **java.lang** are listed in the Table 1.

- ➢ Table 2 lists those exceptions defined by **java.lang** that must be included in a method's **throws** list if that method can generate one of these exceptions and does not handle it itself. These are called *checked exceptions.* Java defines several other types of exceptions that relate to its various class libraries.

Table-1

Java's **Unchecked** RuntimeException Subclasses

Exception	Meaning
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBounds	Attempt to index outside the bounds of a string.
UnsupportedOperationException	An unsupported operation was encountered.

Table-2

Java's **Checked** Exceptions Defined in java.lang

Exception	Meaning
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does not implement the **Cloneable** interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.

- Although Java's built-in exceptions handle most common errors, you will probably want to create your own exception types to handle situations specific to your applications. This is quite easy to do: just define a subclass of **Exception** (which is, of course, a subclass of **Throwable**).

- Your subclasses don't need to actually implement anything—it is their existence in the type system that allows you to use them as exceptions. The **Exception** class does not define any methods of its own. It does, of course, inherit those methods provided by **Throwable**.

- Thus, all exceptions, including those that you create, have the methods defined by **Throwable** available to them. They are shown in Table 10-3. You may also wish to override one or more of these methods in exception classes that you create.

The Methods by Throwable

Method	Description
Throwable fillInStackTrace( )	Returns a Throwable object that contains a completed stack trace. This object can be rethrown.
String getLocalizedMessage( )	Returns a localized description of the exception
String getMessage( )	Returns a description of the exception.
void printStackTrace( )	Displays the stack trace.
Void printStackTrace(PrintStream *stream*)	Sends the stack trace to the specified stream.

Void printStackTrace(PrintWriter *stream*)	Sends the stack trace to the specified stream.
String toString( )	Returns a String object containing a description of the exception. This method is called by println( ) when outputting a Throwable object.

- The following example declares a new subclass of **Exception** and then uses that subclass to signal an error condition in a method. It overrides the **toString( )** method, allowing the description of the exception to be displayed using **println( )**.

// This program creates a custom exception type.
 	
class MyException extends Exception
{
private 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(" + a + ")");
	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); } } }

- This example defines a subclass of **Exception** called **MyException**. This subclass is quite simple: it has only a constructor plus an overloaded **toString( )** method that displays the value of the exception.

- The **ExceptionDemo** class defines a method named **compute( )** that throws a **MyException** object. The exception is thrown when **compute( )**'s integer parameter is greater than 10. The **main( )** method sets up an exception handler for **MyException**, then calls **compute( )** with a legal value (less than 10) and an illegal one to show both paths through the code.

  Here is the result:

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

➢ INTRODUCTION TO MULTITASKING

- A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a *thread,* and each thread defines a separate path of execution.

- Thus, multithreading is a specialized form of multitasking. You are almost certainly acquainted with multitasking, because it is supported by virtually all modern operating systems.

- However, there are two distinct types of multitasking: process-based and thread-based. It is important to understand the difference between the two.
- For most readers, process-based multitasking is the more familiar form.

- A *process* is, in essence, 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. In process-based multitasking, a program is the smallest unit of code that can be dispatched by the scheduler.

- 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.

- Multitasking threads require less overhead than multitasking processes. Processes are heavyweight tasks that require their own separate address spaces.

- Threads, on the other hand, are lightweight. They share the same address space and cooperatively share the same heavyweight process.

- Multithreading enables you to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum.

➢ THREAD LIFECYCLE:

A Thread in its lifetime goes through various states.

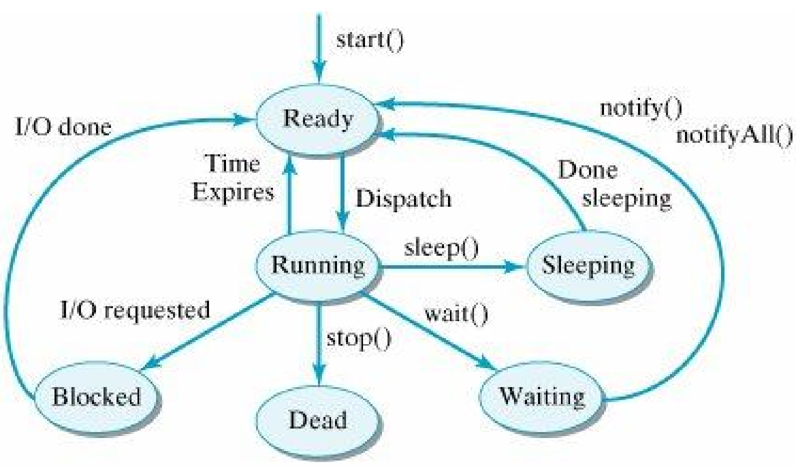
New : When we create a thread naturally, it is in "new" state. The thread is not yet ready to run. The only method can be called from this state is **start().** This method moves to the ready state from which it is automatically moved to **runnable** state by thread scheduler.

Ready: The thread is ready to run (runnable) and waiting to be assigned to a processor by the scheduler. When the thread enters this state first time, it must be through start() method call from New state.

Running: A thread executing in the JVM is in running state. The state can be entered from ready state only when scheduled by the scheduler.

Blocked:A thread is blocked waiting for a monitor lock is in this state .A thread can enter waiting state from running state on any of the following events, like suspend, sleeping, waiting, joining and blocked.

Dead: The thread is destroyed when its run() method completes either normally or abnormally or destroy() or stop() method is called from any state.



Thread Life Cycle

The Thread Class and the Runnable Interface

- Java's multithreading system is built upon the **Thread** class, its methods, and its companion interface, **Runnable**. **Thread** encapsulates a thread of execution.

- To create a new thread, your program will either extend **Thread** or implement the **Runnable** interface.

- The **Thread** class defines several methods that help manage threads. The ones that will be used in this chapter are shown here:

Method	Meaning
getName	Obtain a thread's name.
getPriority	Obtain a thread's priority.
isAlive	Determine if a thread is still running.
join	Wait for a thread to terminate.
run	Entry point for the thread.
sleep	Suspend a thread for a period of time.
start	Start a thread by calling its run method.

➢ CREATING A THREAD:

In the most general sense, you create a thread by instantiating an object of type **Thread**.

Java defines two ways in which this can be accomplished:

• You can implement the **Runnable** interface.

• You can extend the **Thread** class, itself.

The following two sections look at each method, in turn.

Implementing Runnable

- The easiest way to create a thread is to create a class that implements the **Runnable** interface. **Runnable** abstracts a unit of executable code. You can construct a thread on any object that implements **Runnable**. 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*, 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 NewThread implements Runnable
{
Thread t;
NewThread()
{
t = new Thread(this, "Demo Thread");
System.out.println("Child thread: " + t);
t.start();
}
 	
public void run() {
try {
for(int i = 5; i > 0; i—) {
System.out.println("Child Thread: " + i);
Thread.sleep(500);
}
}
catch (InterruptedException e)
{
System.out.println("Child interrupted.");
}
System.out.println("Exiting child thread.");
}
}
	class ThreadDemo
	{
	public static void main(String args[])
	{
	new NewThread();
	try {
	for(int i = 5; i > 0; i—)
	{
	System.out.println("Main Thread: " + i);
	Thread.sleep(1000);
	}
	} catch (InterruptedException e)
	{
	System.out.println("Main thread interrupted.");
	}
	System.out.println("Main thread exiting.");
	}
	}

Inside **NewThread**'s constructor, a new **Thread** object is created by the following statement:

t = new Thread(this, "Demo Thread");

- Passing **this** as the first argument indicates that you want the new thread to call the **run()** method on **this** object. Next, **start( )** is called, which starts the thread of execution beginning at the **run( )** method. This causes the child thread's **for** loop to begin.

- After calling **start( )**, **NewThread**'s constructor returns to **main( )**. When the main thread resumes, it enters its **for** loop. Both threads continue running, sharing the CPU, until their loops finish.

The output produced by this program is as follows:

```
Child thread: Thread[Demo Thread,5,main]
Main Thread: 5
Child Thread: 5
Child Thread: 4
Main Thread: 4
Child Thread: 3
Child Thread: 2
Main Thread: 3
Child Thread: 1
Exiting child thread.
Main Thread: 2
Main Thread: 1
Main thread exiting.
```

- In a multithreaded program, the main thread must be the last thread to finish running. If the main thread finishes before a child thread has completed, then the Java run-time system may "hang."

Extending Thread

- The second way to create a thread is to create a new class that extends **Thread**, and then to 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.

Here is the preceding program rewritten to extend **Thread**:

class NewThread extends Thread
{
NewThread()
{
super("Demo Thread");
System.out.println("Child thread: " + this);
start();
}
public void run() {
try {
for(int i = 5; i > 0; i—) {
System.out.println("Child Thread: " + i);
Thread.sleep(500);
}
} catch (InterruptedException e) {
	class ExtendThread
	{
	public static void main(String args[])
	{
	new NewThread();
	try {
	for(int i = 5; i > 0; i—)
	{
	System.out.println("Main Thread: " + i);
	Thread.sleep(1000);
	}
	} catch (InterruptedException e) {
	System.out.println("Main thread interrupted.");
	}
	System.out.println("Main thread

System.out.println("Child interrupted."); } System.out.println("Exiting child thread."); } }	exiting."); } }

- This program generates the same output as the preceding version. As you can see, the child thread is created by instantiating an object of **NewThread**, which is derived from **Thread**.

- Notice the call to **super( )** inside **NewThread**. This invokes the following form of the **Thread** constructor:

  public Thread(String *threadName*)

  Here, *threadName* specifies the name of the thread.

The Main Thread

- When a Java program starts up, one thread begins running immediately. This is usually called the *main thread* of your program, because it is the one that is executed when your program begins. The main thread is important for two reasons:

  • It is the thread from which other "child" threads will be spawned.

  • It must be the last thread to finish execution. When the main thread stops, your program terminates.

- Although the main thread is created automatically when your program is started, it can be controlled through a **Thread** object. To do so, you must obtain a reference to it by calling the method **currentThread( )**, which is a **public static** member of **Thread**.

  Its general form is shown here: static Thread currentThread( )
- This method returns a reference to the thread in which it is called. Once you have a reference to the main thread, you can control it just like any other thread.

- By default, the name of the main thread is **main**. Its priority is 5, which is the default value, and **main** is also the name of the group of threads to which this thread belongs.

- A *thread group* is a data structure that controls the state of a collection of threads as a whole. This process is managed by the particular run-time environment

- The **sleep( )** method causes the thread from which it is called to suspend execution for the specified period of milliseconds.

static void sleep(long *milliseconds*) throws InterruptedException

- The number of milliseconds to suspend is specified in *milliseconds.* This method may throw an **InterruptedException**.

static void sleep(long *milliseconds,* int *nanoseconds*) throws InterruptedException

- The **sleep( )** method has a second form, which allows you to specify the period in terms of milliseconds and nanoseconds.

You can set the name of a thread by using **setName( )**.

You can obtain the name of a thread by calling **getName( )**

These methods are members of the **Thread** class and are declared like this:

final void setName(String *threadName*)

final String getName( )

Creating Multiple Threads

- So far, you have been using only two threads: the main thread and one child thread. However, your program can spawn as many threads as it needs.

For example, the following program creates three child threads:

```
class NewThread implements Runnable
{
String name; // name of thread
Thread t;
NewThread(String threadname)
{
name = threadname;
t = new Thread(this, name);
System.out.println("New thread: " + t);
t.start(); // Start the thread
}
public void run()
 {
try {
for(int i = 5; i > 0; i—) {

System.out.println(name + " exiting.");
}
}
class MultiThreadDemo
{
public static void main(String args[])
{
new NewThread("One"); // start threads
new NewThread("Two");
new NewThread("Three");
 try {
          Thread.sleep(10000);
} catch (InterruptedException e)  {
System.out.println("Main thread
```

System.out.println(name + ": " + i); Thread.sleep(1000); } } catch (InterruptedException e) { System.out.println(name + "Interrupted"); }	Interrupted"); } System.out.println("Main thread exiting."); } }

The output from this program is shown here: New thread: Thread[One,5,main] New thread: Thread[Two,5,main] New thread: Thread[Three,5,main] One: 5 Two: 5 Three: 5 One: 4 Two: 4 Three: 4 One: 3 Three: 3 Two: 3	One: 2 Three: 2 Two: 2 One: 1 Three: 1 Two: 1 One exiting. Two exiting. Three exiting. Main thread exiting.

- As you can see, once started, all three child threads share the CPU. Notice the call to **sleep(10000)** in **main( )**. This causes the main thread to sleep for ten seconds and ensures that it will finish last.

Using isAlive() and join()

- Two ways exist to determine whether a thread has finished. 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. While **isAlive( )** is occasionally useful, the method that you will more commonly use 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.

```
class DemoJoin
{
public static void main(String args[])
 {
NewThread ob1 = new NewThread("One");
NewThread ob2 = new NewThread("Two");
NewThread ob3 = new NewThread("Three");
System.out.println("Thread One is alive: "+ ob1.t.isAlive());
System.out.println("Thread Two is alive: "+ ob2.t.isAlive());
System.out.println("Thread Three is alive: "+ ob3.t.isAlive());
try {
System.out.println("Waiting for threads to finish.");
ob1.t.join();       ob2.t.join();            ob3.t.join();
}
catch (InterruptedException e)
{    System.out.println("Main thread Interrupted");
}
System.out.println("Thread One is alive: "+ ob1.t.isAlive());
System.out.println("Thread Two is alive: "+ ob2.t.isAlive());
```

```
System.out.println("Thread Three is alive: "+ ob3.t.isAlive());
System.out.println("Main thread exiting.");
}
}
```

Sample output from this program is shown here:

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
Thread One is alive: true
Thread Two is alive: true
Thread Three is alive: true
Waiting for threads to finish.
One: 5
Two: 5
Three: 5
One: 4
Two: 4
Three: 4
One: 3
Two: 3
Three: 3
One: 2
Two: 2
Three: 2
One: 1
Two: 1
Three: 1
Two exiting.
Three exiting.
One exiting.
Thread One is alive: false
```

Thread Two is alive: false

Thread Three is alive: false

Main thread exiting.

- As you can see, after the calls to **join( )** return, the threads have stopped executing.

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. (Ex: OS, CPU time.etc)

- To set a thread's priority, use the **setPriority( )** method, which is a member of **Thread**. This is its general form:

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

  final int getPriority( )

➢ 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.*

- A *monitor* is an object that is used as a mutually exclusive lock, or *mutex.* 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.

Using Synchronized Methods

- 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.
- The following program has three simple classes. The first one, **Callme**, has a single method named **call( )**.
- The **call( )** method takes a **String** parameter called **msg**. This method tries to print the **msg** string inside of square brackets. The interesting thing to notice is that after **call( )** prints the opening bracket and the **msg** String, it calls **Thread.sleep(1000)**, which pauses the current thread for one second.
- The constructor of the next class, **Caller**, takes a reference to an instance of the **Callme** class and a **String**, which are stored in **target** and **msg**, respectively. The constructor also creates a new thread that will call this object's **run( )** method.

- The thread is started immediately. The **run( )** method of **Caller** calls the **call( )** method on the **target** instance of **Callme**, passing in the **msg** string.

- Finally, the **Synch** class starts by creating a single instance of **Callme**, and three instances of **Caller**, each with a unique message string. The same instance of **Callme** is passed to each **Caller**.

// This program is not synchronized.
class Callme {
void call(String msg) {
System.out.print("[" + msg);
try {
Thread.sleep(1000);
} catch(InterruptedException e) {
System.out.println("Interrupted");
}
System.out.println("]");
}
}
class Caller implements Runnable {
String msg;
Callme target;
Thread t;
public Caller(Callme targ, String s) {
target = targ;
msg = s;
t = new Thread(this);
	public void run()
	{
	target.call(msg);
	}
	}
	class Synch {
	public static void main(String args[]) {
	Callme target = new Callme();
	Caller ob1 = new Caller(target, "Hello");
	Caller ob2 = new Caller(target, "Synchronized");
	Caller ob3 = new Caller(target, "World");
	// wait for threads to end
	try {
	ob1.t.join();
	ob2.t.join();
	ob3.t.join();
	} catch(InterruptedException e) {
	System.out.println("Interrupted");

t.start(); }	} } }

Here is the output produced by this program: Hello[Synchronized[World]

- As you can see, by calling **sleep( )**, the **call( )** method allows execution to switch to another thread.

- This results in the mixed-up output of the three message strings. In this program, nothing exists to stop all three threads from calling the same method, on the same object, at the same time. This is known as a *race condition,* because the three threads are racing each other to complete the method.

- This example used **sleep( )** to make the effects repeatable and obvious. In most situations, a race condition is more subtle and less predictable, because you can't be sure when the context switch will
- occur. This can cause a program to run right one time and wrong the next.

- To fix the preceding program, you must *serialize* access to **call( )**. That is, you must restrict its access to only one thread at a time. To do this, you simply need to precede **call( )**'s definition with the keyword **synchronized**, as shown here:

```
class Callme {
synchronized void call(String msg) {
                                        ...}
```

- This prevents other threads from entering **call( )** while another thread is using it. After **synchronized** has been added to **call( )**, the output of the program is as follows:

  [Hello]
  [Synchronized]
  [World]

- Any time that you have a method, or group of methods, that manipulates the internal state of an object in a multithreaded situation, you should use the **synchronized** keyword to guard the state from race conditions. Remember, once a thread enters any synchronized method on an instance, no other thread can enter any other synchronized method on the same instance. However, non synchronized methods on that instance will continue to be callable.

The synchronized Statement

- 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. If you want to synchronize only a single statement, then the curly braces are not needed. 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.

- Here is an alternative version of the preceding example, using a synchronized block within the **run( )** method:

// This program uses a synchronized block.
 	
class Callme
{
void call(String msg)
{
System.out.print("[" + msg);
try {
Thread.sleep(1000);
} catch (InterruptedException e)
{
System.out.println("Interrupted");
}
System.out.println("]");
}
}
class Caller implements Runnable {
	// synchronize calls to call()
	public void run()
	{
	synchronized(target)
	{ // synchronized block
	target.call(msg);
	}
	}
	}
	class Synch1
	{
	public static void main(String args[])
	{
	Callme target = new Callme();
	Caller ob1 = new Caller(target, "Hello");
	Caller ob2 = new Caller(target, "Synchronized");

String msg;
Callme target;
Thread t;
public Caller(Callme targ, String s)
{
target = targ;
msg = s;
t = new Thread(this);
t.start();
}
	Caller ob3 = new Caller(target, "World");
	// wait for threads to end
	try {
	ob1.t.join();
	ob2.t.join();
	ob3.t.join();
	} catch(InterruptedException e)
	{
	System.out.println("Interrupted");
	}
	}
	}

- Here, the **call( )** method is not modified by **synchronized**. Instead, the **synchronized** statement is used inside **Caller**'s **run( )** method. This causes the same correct output as the preceding example, because each thread waits for the prior one to finish before proceeding.

Interthread Communication

- Polling is usually implemented by a loop that is used to check some condition repeatedly. Once the condition is true, appropriate action is taken. This wastes CPU time

- To avoid polling, Java includes an elegant interprocess communication mechanism via the **wait( )**, **notify( )**, and **notifyAll( )** methods. 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** method.

• **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 the first thread that called **wait( )** on the same object.

• **notifyAll( )** wakes up all the threads that called **wait( )** on the same object. The highest priority thread will run first.

These methods are declared within **Object**, as shown here:

```
final void wait( ) throws InterruptedException
final void notify( )
final void notifyAll( )
```

- Additional forms of **wait( )** exist that allow you to specify a period of time to wait. The following sample program incorrectly implements a simple form of the producer/consumer problem. It consists of four classes: **Q**, 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**.

// An incorrect implementation of a producer and consumer.
class Q
{
int n;
synchronized int get()
{
System.out.println("Got: " + n);
return n;
	public void run()
	{
	while(true)
	{
	q.get();
	}
	}
	}

}
synchronized void put(int n)
{
this.n = n;
System.out.println("Put: " + n);
} }
class Producer implements Runnable
{
Q q;
Producer(Q q)
{
this.q = q;
new Thread(this, "Producer").start();
}
public void run()
{
int i = 0;
while(true)
{
q.put(i++);
} } }
class Consumer implements Runnable
{
Q q;
Consumer(Q q)
{
this.q = q;
new Thread(this,
	class PC
	{
	public static void main(String args[])
	{
	Q q = new Q();
	new Producer(q);
	new Consumer(q);
	System.out.println("Press Control-C to stop.");
	}
	}
	OUTPUT:
	Put: 1
	Got: 1
	Got: 1
	Got: 1
	Got: 1
	Got: 1
	Put: 2
	Put: 3
	Put: 4
	Put: 5
	Put: 6
	Put: 7
	Got: 7

"Consumer").start(); }	

- Although the **put( )** and **get( )** methods on **Q** are synchronized, nothing stops the producer from overrunning the consumer, nor will anything stop the consumer from consuming the same queue value twice. Thus, you get the erroneous output shown here (the exact output will vary with processor speed and task load):

- As you can see, after the producer put 1, the consumer started and got the same 1 five times in a row. Then, the producer resumed and produced 2 through 7 without letting the consumer have a chance to consume them.
- The proper way to write this program in Java is to use **wait( )** and **notify( )** to signal in both directions, as shown here:

// A correct implementation of a producer and consumer.
class Q {
int n;
boolean valueSet = false;
synchronized int get() {
if(!valueSet)
try {
wait();
} catch(InterruptedException e) {
System.out.println("InterruptedException caught");
}
System.out.println("Got: " + n);
valueSet = false;
	class Consumer implements Runnable {
	Q q;
	Consumer(Q q) {
	this.q = q;
	new Thread(this, "Consumer").start();
	}
	
	public void run() {
	while(true) {
	q.get();
	}
	}
	}

notify();
return n;
}
synchronized void put(int n) {
if(valueSet)
try {
wait();
} catch(InterruptedException e) {
System.out.println("InterruptedException caught");
}
this.n = n;
valueSet = true;
System.out.println("Put: " + n);
notify();
}
}
class Producer implements Runnable {
Q q;
Producer(Q q) {
this.q = q;
new Thread(this, "Producer").start();
}
public void run() {
int i = 0;
while(true) {
q.put(i++);
}
}
}
	class PCFixed {
	public static void main(String args[]) {
	Q q = new Q();
	new Producer(q);
	new Consumer(q);
	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

- Inside **get( )**, **wait( )** is called. This causes its execution to suspend until the **Producer** notifies you that some data is ready. When this happens, execution inside **get( )** resumes. After the data has been obtained, **get( )** calls **notify( )**. This tells **Producer** that it is okay to put more data in the queue. Inside **put( )**, **wait( )** suspends execution until the **Consumer** has removed the item from the queue. When execution resumes, the next item of data is put in the queue, and **notify( )** is called. This tells the **Consumer** that it should now remove it. Here is some output from this program, which shows the clean synchronous behavior:

THREADGROUP

- **ThreadGroup** creates a group of threads. It defines these two constructors:

  ThreadGroup(String *groupName*)

  ThreadGroup(ThreadGroup *parentOb*, String *groupName*)

- For both forms, *groupName* specifies the name of the thread group. The first version creates a new group that has the current thread as its parent. In the second form, the parent is specified by *parentOb.*
- **ThreadGroup** also included the methods **stop( )**, **suspend( )**, and **resume( )**.
- These have been deprecated by Java 2 because they were inherently unstable. Thread groups offer a convenient way to manage groups of threads as a unit. This is particularly valuable in situations in which you want to suspend and resume a number of related threads.

- For example, imagine a program in which one set of threads is used for printing a document, another set is used to display the document on the screen, and another set saves the document to a disk file. If printing is aborted, you will want an easy way to stop all threads related to printing. Thread groups offer this convenience.

UNIT-IV
Assignment-Cum-Tutorial Questions
SECTION-A

Objective Questions

1. Identify the parent class of all the exception in java is [ ]
   a)Throwable b)Throw c) Exception d)Throws

2. What are the two types of exception available in java ? [ ]
   a)Checked and compiled b) Un Checked and compiled
   c)Checked and Un Checked d) Compiled and non- compiled

3. The two subclasses of Throwable are [ ]
   a)Error and AssertionError
   b)Error and Exception
   c)Checked and UnChecked Exception
   d)Error and Runtime Exception

4. Choose the correct option regarding notifyAll() method. [ ]
   a) Wakes up one threads that are waiting on this object's monitor
   b) Wakes up all threads that are not waiting on this object's monitor
   c)Wakes up all threads that are waiting on this object's monitor
   c) None of the above

5. Identify the keyword when applied on a method indicates that only one thread should execute the method at a time. [ ]
   a)volatile b) synchronized c) native d) static

6. The built-in base class in Java, which is used to handle all exceptions is [ ]
   a)Raise
   b)Exception
   c)Error
   d)Throwable

7. Which of the following exceptions is thrown when one thread has been interrupted by another thread? [ ]
   a)ClassNotFoundException
   b)IllegalAccessException
   c)InstantiationException
   d)InterruptedException
   e)NoSuchFieldException

8. Which of the following Exception classes in Java is used to deal with an exception, where an assignment to an array element is of incompatible type? [ ]
   a)ArithmeticException

b)ArrayIndexOutOfBoundsException
c)IllegalArgumentException
d)ArrayStoreException
e)IllegalStateException

9. A programmer has created his own exception for balance in account <1000. The exception is created properly, and the other parts of the programs are correctly defined. Though the program is running but error message has not been displayed. Why did this happen? [ ]
a)Because of the Throw portion of exception.
b)Because of the Catch portion of exception.
c)Because of the main() portion.
d)Because of the class portion.
e)None of the above

10. Choose the correct option for the following program [ ]

```
class demo
{
        void show() throws CalssNotFoundException{}
}
class demo2 extends demo
{
void show() throws IllegalAccessException, classNotFoundException, ArithmeticException
{
        System.out.println("In Demo1 show");
}
        public static void main(String arg[])
{
try{
        demo2   d=new demo2();
        d.show();
  }
catch(Exception e)  {}
        }
 }
```

a.Does not compile
b.Compiles successfully
c.Compiles successfully and prints "In Demo1 show"
d.Compiles but does not execute.

11. If the assert statement returns false, what is thrown? [ ]
a)Exception b) Assert c) assertion d) assertion Error

12. Choose the best possible answer for the following program [ ]

```
class demo
{
        void show() throws ArithmeticException
         { }
}
class demo2 extends demo
```

```
{
        void show()
        {
        System.out.println("In Demo1 show");
}
public static void main(String arg[])
{
        demo2 d=new demo2();
        d.show();
} }
```

a.Does not compile
b.Compiles successfully
c.Compiles successfully and prints "In Demo1 show"
b.Compiles but does not execute.

13. How can Thread go from waiting to runnable state? [ ]
a)notify / notifAll
b)bWhen sleep time is up
c)Using resume() method when thread was suspended
d)All

14. Predict the output of the following program [ ]

```
class A implements Runnable{
     public void run(){
          try{
               for(int i=0;i<4;i++){
                    Thread.sleep(100);
                    System.out.println(Thread.currentThread().getName());
               }
          }catch(InterruptedException e){
          }
     }
}

public class Test{
     public static void main(String argv[]) throws Exception{
          A a = new A();
          Thread t = new Thread(a, "A");
          Thread t1 = new Thread(a, "B");
          t.start();
          t.join();
          t1.start();
     }
}
```

a) A A A A B B B B b) A B A B A B A B
c) Output order is not guaranteed d) Compilation succeed but Runtime Exception

15. What will be output of the following program code? [ ]

```
public class Test implements Runnable{
  public void run(){
 System.out.print("go");
 }
public static void main(String arg[]) {
     Thread t = new Thread(new Test())
    t.run();
        t.run();
        t.start();
    }
}
```

a) Compilation fails.

b) An exception is thrown at runtime

c) go" is printed

d) “gogo" is printed

16. Choose the correct option for Deadlock situation [ ]

a) Two or more threads have circular dependency on an object

b) Two or more threads are trying to access a same object

c)Two or more threads are waiting for a resource

d) None of these

17. Predict the output of following Java program [ ]

```
class Main {
  public static void main(String args[]) {
    try {
      throw 10;
    }
    catch(int e) {
      System.out.println("Got the  Exception " + e);
    }
 }
```

```
}
```

a) Got the Exception 10

b)Got the Exception 0

c)Compiler Error

d)None of the above

18. What is the output of the following program [ ]

```
class Test extends Exception { }
  class Main {
   public static void main(String args[]) {
      try {
         throw new Test();
      }
      catch(Test t) {
         System.out.println("Got the Test Exception");
      }
      finally {
         System.out.println("Inside finally block ");
      }
  }
}
```

a)Got the Test Exception Inside finally block
b)Got the Test Exception
c)Inside finally block
d)Compile error.

19. What is the output of the following program [ ]

```
 class Test
{
    public static void main(String[] args)
    {
        try
        {
            int a[]= {1, 2, 3, 4};
            for (int i = 1; i <= 4; i++)
            {
                System.out.println ("a[" + i + "]=" + a[i] + "n");
            }
        }
```

```
            catch (Exception e)
            {
                System.out.println ("error = " + e);
            }

            catch (ArrayIndexOutOfBoundsException e)
            {
                System.out.println ("ArrayIndexOutOfBoundsException");
            }
        }
    }
```

a) Compiler error
b)Run time error
c)ArrayIndexOutOfBoundsException
d)Error Code is printed
e)Array is printed

20. Predict the output of the following program. [ ]

```
class Test
{   int count = 0;

    void A() throws Exception
    {
        try
        {
            count++;

            try
            {
                count++;

                try
                {
                    count++;
                    throw new Exception();

                }

                catch(Exception ex)
                {
                    count++;
                    throw new Exception();
                }
            }

            catch(Exception ex)
            {
```

```
                count++;
            }
        }

        catch(Exception ex)
        {
            count++;
        }

    }

    void display()
    {
        System.out.println(count);
    }

    public static void main(String[] args) throws Exception
    {
        Test obj = new Test();
        obj.A();
        obj.display();
    }
}
```

a)4 b)5 c)6 d)Compile Error

SECTION-B

Descriptive Questions

1. Define Exception? What are the three categories of exceptions? Also discuss the advantages of exception handling
2. Explain the keywords used in exception handling.
3. Implement a multiple exception handling for the following problem Read n+1 strings to string array and prints their lengths to get ArrayIndexOutOfBoundsException and NullPointerException
4. Write a java program to calculate the student total marks and percentage for class test with six subjects. The marks should be 0 to 10 only, if marks entered not in the range then raise an exception MarksNotInRangeException**.(**Create user defined exception and throw it).
5. Can a try block be written without a catch block? Justify.

6. Can we nest a try statement inside another try statement. Write the necessary explanation and example for this.
7. Differentiate multi tasking and multithreading.
8. Draw a neat sketch of thread life cycle.
9. What is synchronization and how do we use it in java.
10. Write a Java program to create two threads from main such that one thread calculates the factorial of a given number and another thread checks whether the given number is prime or not.
11. Write a Java program to print the messages in the following sequence

    For every 3 seconds " Welcome" message

    For every 2 seconds "Hello" message

    For every 5 seconds " Bye" message