Exception Handling

Exception Handling

- An exception is an abnormal condition that arises in a code sequence at run time.
- In other words, **an exception is a run-time error**.
- In computer languages that do not support exception handling, errors must be checked and handled manually—typically through the use of error codes, and so on.
- This approach is as cumbersome as it is troublesome. Java’s exception handling avoids these problems

Exception Handling

- 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.

Exception Handling

- 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.

Exception Handling

- 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.

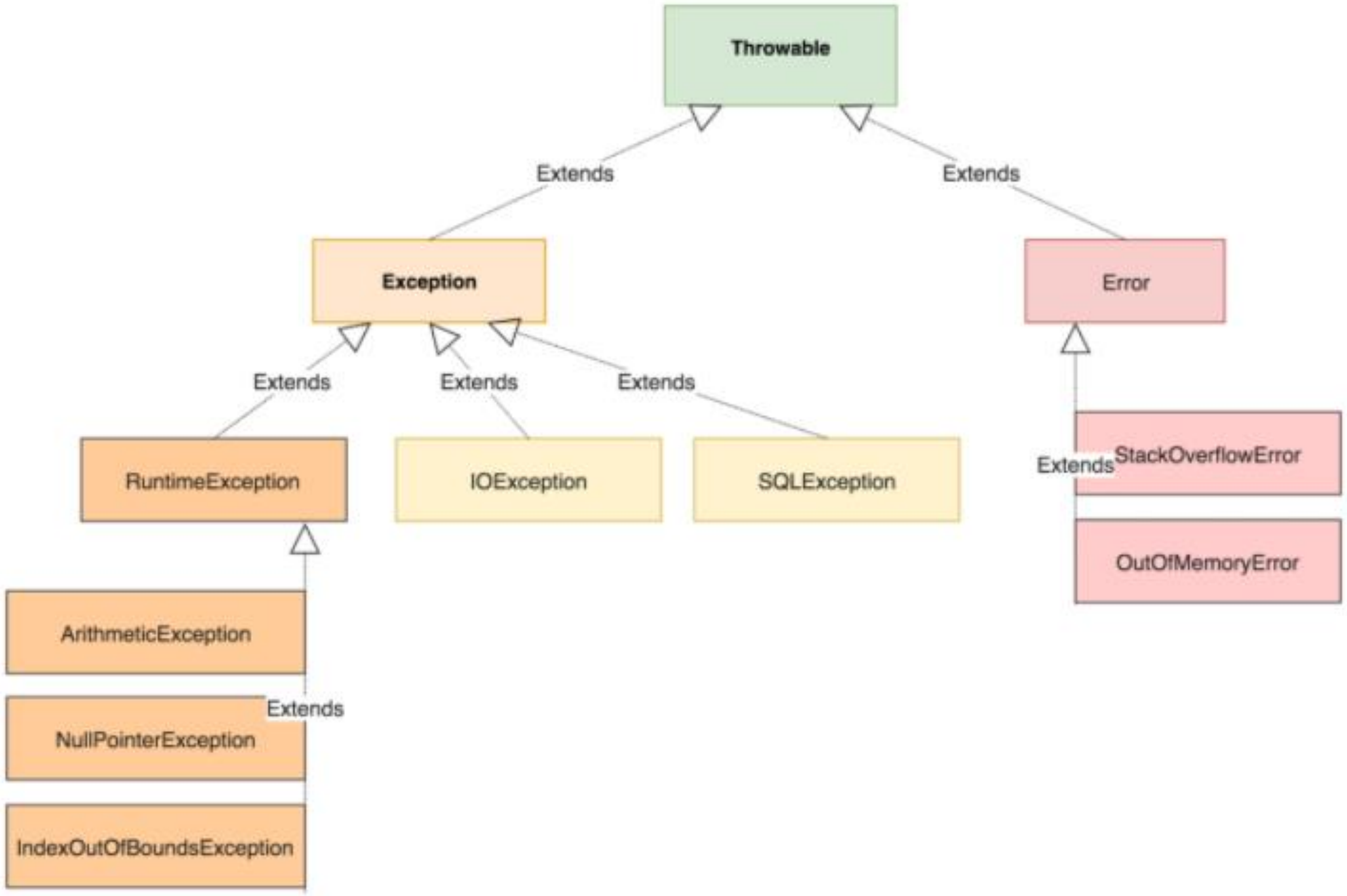
Exception Handling

- 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 Handling

```
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 after try block ends
}
```

Exception Handling



Exception Handling

- **Throwable** - at the top of the exception class hierarchy
- **Exception** - 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.
- **RunTimeException -** Exceptions of this type are automatically defined for the programs that you write and include things such as division by zero
- **Error** - which defines exceptions that are not expected to be caught under normal circumstances by your program. Used by JRE

Exception Handling

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

O/P:Exception in thread "main" java.lang.ArithmeticException: / by zero
at Exc0.main(Exc0.java:4)

- Exception is caught by the ***default handler*** provided by the Java run-time system.
- Any exception that is not caught by your program will ultimately be processed by the default handler.
- The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program.

Exception Handling

- Exception Handling provides following advantages
- First, **it allows you to fix the error**.
- Second, **it prevents the program from automatically terminating**.
- Most users would be confused if your program stopped running and printed a stack trace whenever an error occurred.
- To guard against and handle a run-time error, *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*.

```
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.");
}
}
```

```
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.");
}
}
```

OUTPUT

```
D:\java\bin>java Exc2
Division by zero.
After catch statement.
```

Exception Handling

- Once an exception is thrown, program control transfers out of the **try** block into the **catch** block.
- **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.
- The goal of most well-constructed **catch** clauses should be to resolve the exceptional condition and then continue on as if the error had never happened.

Exception Handling

- 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.
- A catch statement cannot catch an exception thrown by another try statement (except in the case of nested try statements).
- The statements that are protected by try must be surrounded by curly braces. (That is, they must be within a block.)
- You cannot use try on a single statement.

```
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.");
}
}
```

```
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.");
}
}
```

OUTPUT

```
D:\java\bin>java MultiCatch
a = 0
Divide by 0: java.lang.ArithmeticException: / by
zero
After try/catch blocks.
```

```
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.");
}
}
```

OUTPUT

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

Exception Handling

- When you use multiple **catch** statements, it is important to remember that exception subclasses must come before any of their superclasses.
- This is because a **catch** statement that uses a superclass will catch exceptions of that type plus any of its subclasses.
- Thus, a subclass would never be reached if it came after its superclass.
- Further, in Java, unreachable code is an error.

```
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.");
}
/* This catch is never reached because
ArithmeticException is a subclass of Exception. */


catch(ArithmeticException e) { // ERROR - unreachable
System.out.println("This is never reached.");
}
}
}
```

```
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.");
}
/* This catch is never reached
because
ArithmeticException is a subclass of
Exception. */


catch(ArithmeticException e) { //
ERROR - unreachable
System.out.println("This is never
reached.");
}}}
```

OUTPUT

```
D:\java\bin>javac SuperSubCatch.java
SuperSubCatch.java:11: exception
java.lang.ArithmeticException has
already been
caught
catch(ArithmeticException e) { // ERROR -
unreachable
^
1 error
```

```
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.");
}
/* This catch is never reached
because
ArithmeticException is a subclass of
Exception. */


catch(ArithmeticException e) { //
ERROR - unreachable
System.out.println("This is never
reached.");
}}}
```

OUTPUT

```
D:\java\bin>javac SuperSubCatch.java
SuperSubCatch.java:11: exception
java.lang.ArithmeticException has already
been caught
catch(ArithmeticException e) { // ERROR -
unreachable
^
1 error
```

- **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.**

Exception Handling

- 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.

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

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

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

OUTPUT

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

D:\java\bin>java NestTry b n
a = 2
Array index out-of-bounds:
java.lang.ArrayIndexOutOfBoundsException
```

Exception Handling

- 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.
- There are two ways you can obtain a Throwable object:
- *using a parameter in a catch clause*, or *creating one with the new operator*.

Exception Handling

- 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 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.

```
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);
}}}
```

```
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);
}}}
```

OUTPUT

```
D:\java\bin>java ThrowDemo
Caught inside demoproc.
Recaught: java.lang.NullPointerException: demo
```

Exception Handling

- 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.

Exception Handling

- 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 }

- exception-list is a comma-separated list of the exceptions that a method can throw.

```
class ThrowsDemo1 {
static void throwOne() {

System.out.println("Inside throwOne.");
throw new IllegalAccessException("demo");

}

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

OUTPUT

```
D:\java\bin>javac ThrowsDemo1.java
ThrowsDemo1.java:4: unreported exception
java.lang.IllegalAccessException; must
be caught or declared to be thrown
throw new IllegalAccessException("demo");
^
1 error
```

```
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);
}}}
```

OUTPUT

```
D:\java\bin>java ThrowsDemo2
Inside throwOne.
Caught java.lang.IllegalAccessException: demo
```

Exception Handling

- 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.

Exception Handling

- **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.

Exception Handling

- 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.

```
class FinallyDemo {
static void procA() {
try {
System.out.println("inside procA");
throw new RuntimeException("demo");
} finally {System.out.println("procA's finally");
} }

static void procB() {
try {
System.out.println("inside procB");
return;
} finally {System.out.println("procB's finally");}}

static void procC() {
try {
System.out.println("inside procC");
} 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(); }}
```

```
class FinallyDemo {
static void procA() {
try {
System.out.println("inside procA");
throw new RuntimeException("demo");
} finally {System.out.println("procA's
finally");
} }
static void procB() {
try {
System.out.println("inside procB");
return;
} finally {System.out.println("procB's
finally");}}
static void procC() {
try {
System.out.println("inside procC");
} 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(); }}
```

OUTPUT

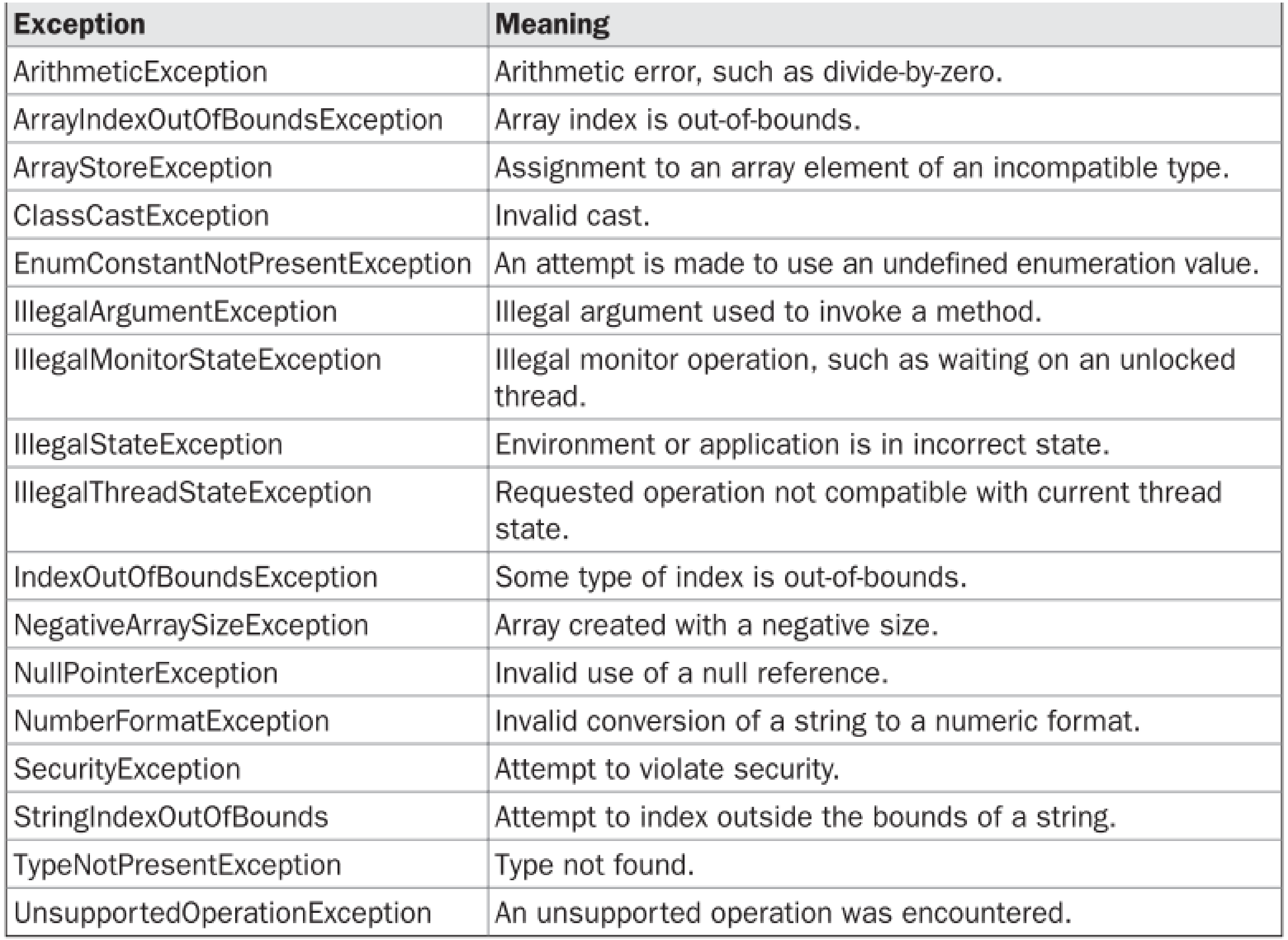
```
D:\java\bin>java FinallyDemo
inside procA
procA's finally
Exception caught
inside procB
procB's finally
inside procC
procC's finally
```

Exception Handling

- 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.

Exception Handling

- Inside the standard package java.lang, Java defines several exception classes.
- The most general of these exceptions are subclasses of the standard type RuntimeException.
- These exceptions need not be included in any method’s throws list.
- These are called ***unchecked exceptions*** because the compiler does not check to see if a method handles or throws these exceptions.

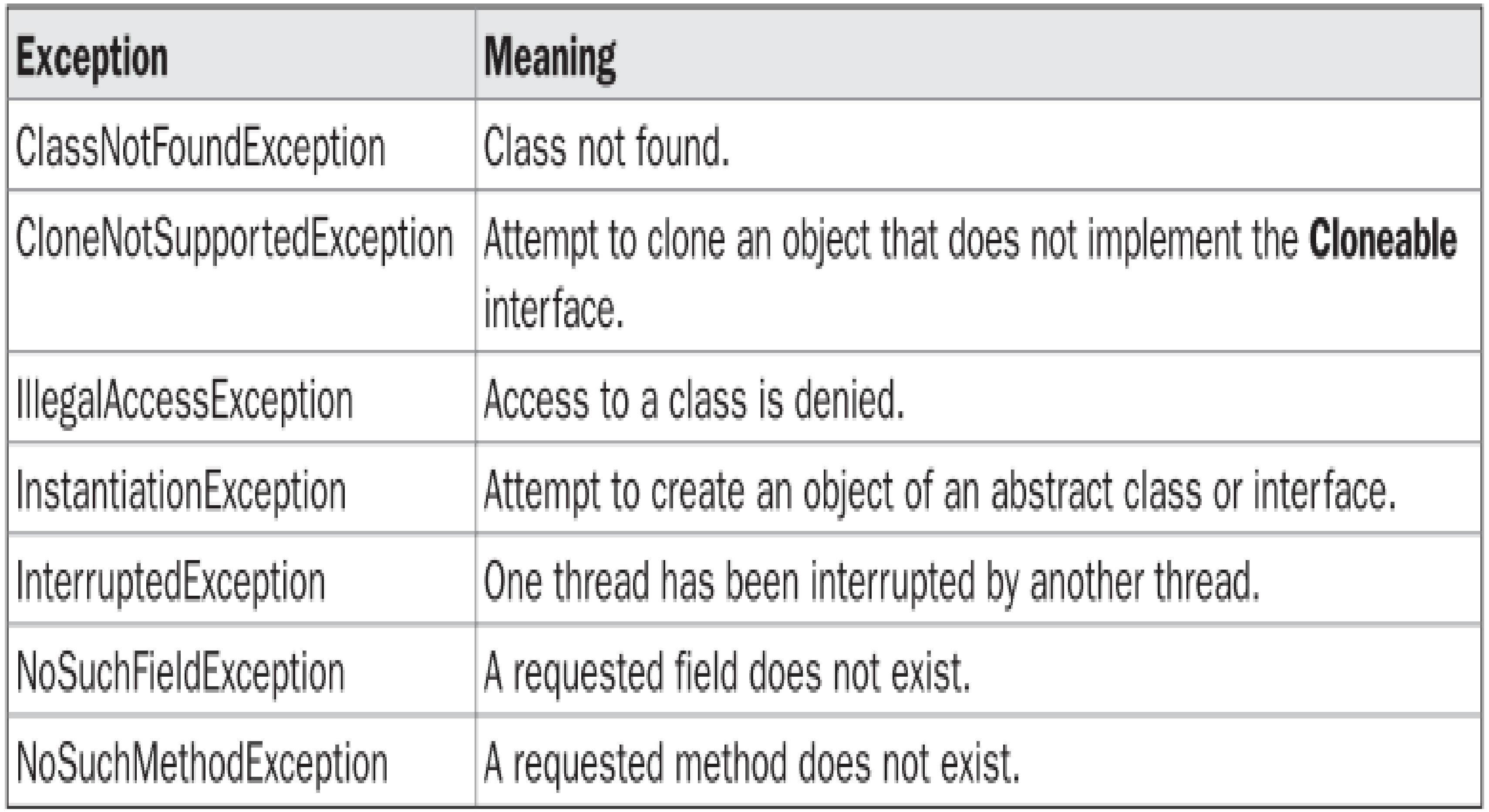


Exception	Meaning
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
EnumConstantNotPresentException	An attempt is made to use an undefined enumeration value.
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.
TypeNotPresentException	Type not found.
UnsupportedOperationException	An unsupported operation was encountered.

TABLE 10-1 Java's Unchecked **RuntimeException** Subclasses Defined in **java.lang**

Exception Handling

- ***Checked exceptions*** - 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.



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.

TABLE 10-2 Java's Checked Exceptions Defined in **java.lang**

User Defined Exceptions

Steps for creating a user-defined exception:

- Create a class with your own class name (this acts the exception name)
- Extend the pre-defined class *Exception*
- Throw an object of the newly create exception

```
class NegativeException extends Exception {
    String msg = "Value cannot be negative";
    NegativeException() {}
    NegativeException(String str) {
           msg = str;
     }
     public String toString() {
           return "NegativeException: " + msg;
      }
}
```

/*overriding the *toString()* method of the *Exception* class to provide meaningful description of my own exception. */

```
class NegativeExceptionDemo {
    public static void main(String[] args){
        try{
            int x = -5;
            if(x < 0){
                throw new NegativeException();
            }
            else {
                System.out.println("x = " + x);
             }
         }
         catch(NegativeException e){
            System.out.println(e);
          }
    }
}
```

- **<u>Output</u>**

```
NegativeException: Value cannot be negative
```

It notifies the user about negative values which are not allowed as input.

Threads

Multithreading

- Java provides built-in support for multithreaded programming.
- 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.
- A thread is a lightweight sub-process, the smallest unit of processing.
- Thus, multithreading is a specialized form of multitasking.

Multithreading

- There are two distinct types of multitasking: ***process based*** and ***thread-based***.
- 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.
- In process based multitasking, a program is the smallest unit of code that can be dispatched by the scheduler.

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.

Multithreading

- Multitasking threads require less overhead than multitasking processes.
- Processes are heavyweight tasks that require their own separate address spaces.
- Interprocess communication is expensive and limited.
- Context switching from one process to another is also costly.
- Threads, on the other hand, are lightweight.
- 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 low cost.
- While Java programs make use of process based multitasking environments, process-based multitasking is not under the control of Java.
- However, multithreaded multitasking is under the control of Java.

Multithreading

- In a single-threaded environment, your program has to wait for each of these tasks to finish before it can proceed to the next one—even though the CPU is sitting idle most of the time.
- Multithreading lets you gain access to this idle time and put it to good use.

Multithreading

Multitasking	Multithreading
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
Context switching from one process to another is also costly	Multitasking threads require less overhead than multitasking processes
Interprocess communication is expensive and limited	Interthread communication is inexpensive, and context switching from one thread to the next is low cost
While Java programs make use of process based multitasking environments, process-based multitasking is not under the control of Java	multithreaded multitasking is under the control of Java

Multithreading

- Multithreading enables you to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum.
- This is especially important for the interactive, networked environment in which Java operates, because idle time is common.
- For example, the transmission rate of data over a network is much slower than the rate at which the computer can process it.
- Even local file system resources are read and written at a much slower pace than they can be processed by the CPU

Thread Priority

- 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.
- A higher-priority thread can also preempt a lower-priority one.

Thread Priority

- 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.
- In theory, threads of equal priority should get equal access to the CPU.
- To set a thread’s priority, use the **setPriority( )** method
- This is its general form:
    - final void setPriority(int level)

Thread Priority

- ***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

Context Switching

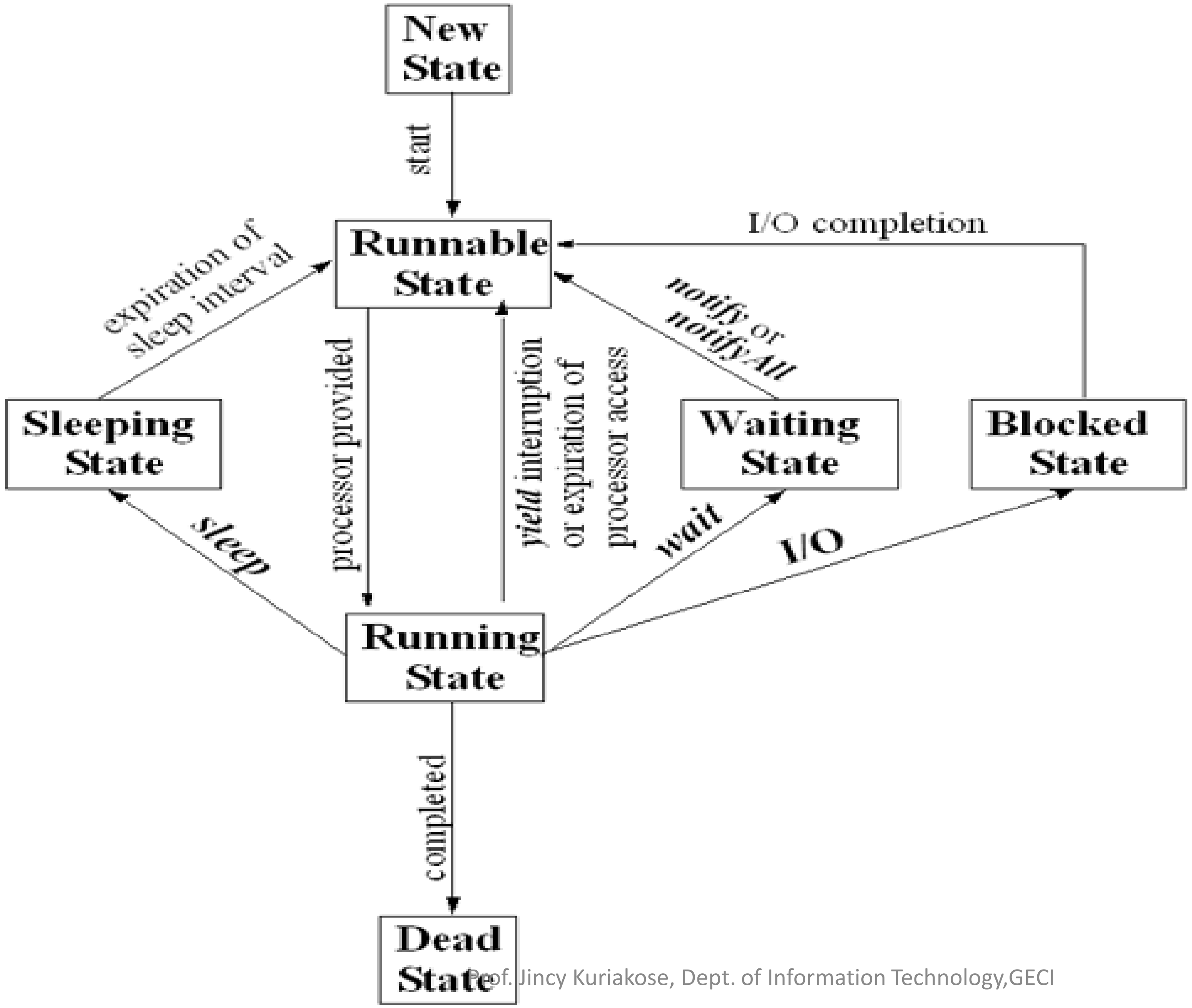
- A threads priority is used to decide when to switch from one running thread to the next. This is called a ***context switch***.
- The rules that determine when a context switch takes place are

a. ***A thread can voluntarily relinquish control***. This is done by explicitly yielding, sleeping, or blocking on pending I/O. In this scenario, all other threads are examined, and the highest-priority thread that is ready to run is given the CPU.

b. ***A thread can be preempted by a higher-priority thread***. In this case, a lower-priority thread that does not yield the processor is simply preempted by a higher-priority thread. This is called ***preemptive multitasking***.

Life Cycle of a Thread

- The life cycle of the thread in java is controlled by JVM. The java thread states are as follows:
- New
- Runnable
- Running
- Waiting
- Blocked
- Terminated


New State
start
Runnable State
I/O completion
expiration of sleep interval
processor provided
yield interruption or expiration of processor access
notify or notifyAll
Sleeping State
Waiting State
Blocked State
sleep
wait
I/O
Running State
completed
Dead State

New

- The thread is in new state if you create an instance of Thread class but before the invocation of start() method.

Runnable

- The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.

Running

- The thread is in running state if the thread scheduler has selected it.

Waiting:

- A thread that is suspended because it is waiting for some action to occur. For example, it is waiting because of a call to non-timeout version of *wait()* method or *join()* method or *sleep()* method.

Blocked:

- A thread that has suspended execution because it is waiting to acquire a lock or waiting for some I/O event to occur.

Dead

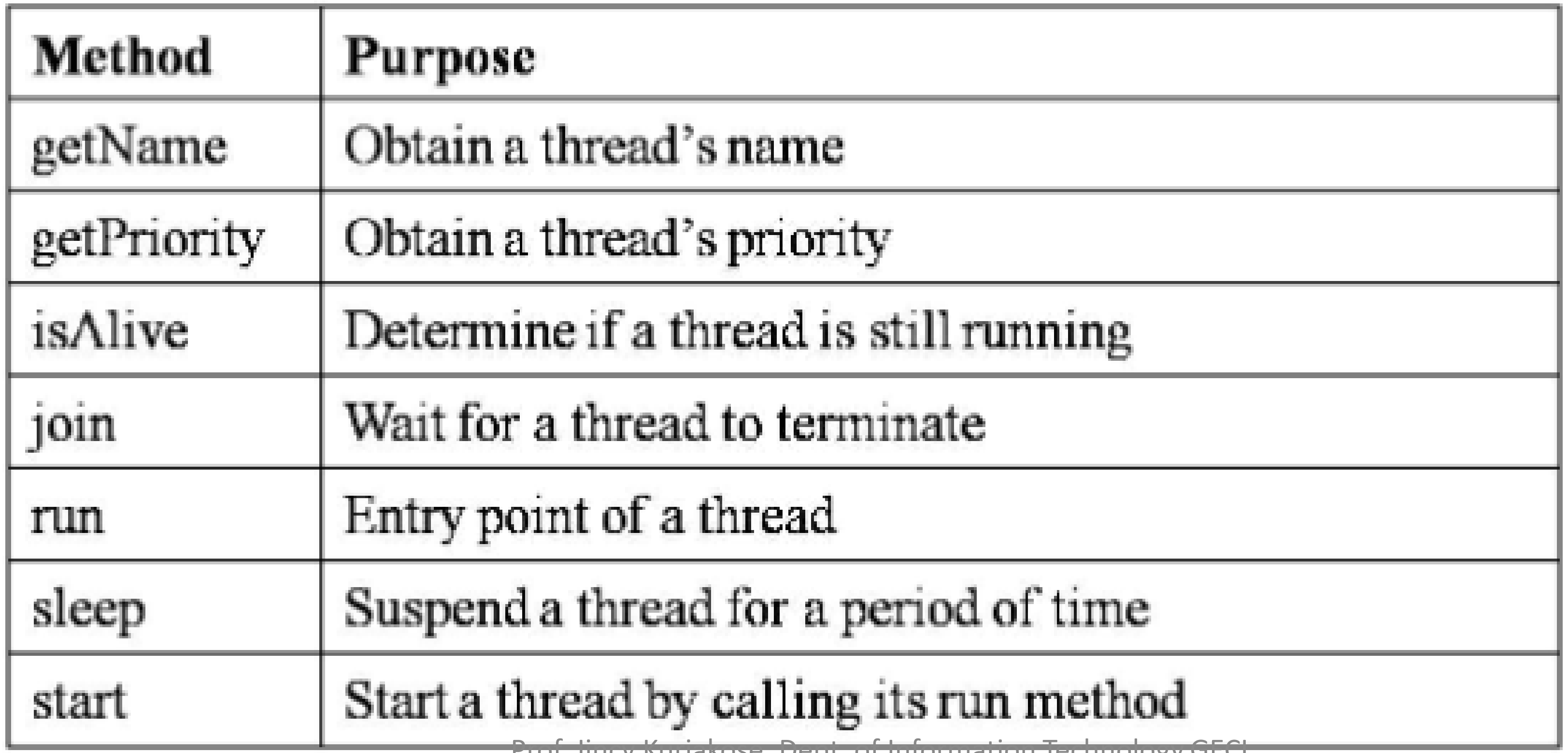
- A thread is in terminated or dead state when its run() method exits.

Advantages of multithreading

- More responsive programs.
- You can perform many operations together so it saves time.
- Threads are independent so it doesn't affect other threads if exception occur in a single thread.
- Better resource utilization.
- Simpler program design in some situations.
- Enhanced performance on multi-processor machines

Thread class and Runnable Interface

- To create a new thread – either extend Thread class or implement Runnable interface
- Thread class contains the following methods



Method	Purpose
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 of a thread
sleep	Suspend a thread for a period of time
start	Start a thread by calling its run method

Commonly used Constructors of Thread class

- Thread()
- Thread(String name)
- Thread(Runnable r)
- Thread(Runnable r,String name)

The Main Thread

- All Java programs begin with the one thread : the ***main thread***.
- The main thread is important for two reasons:
    - *It is the thread from which other “child” threads will be spawned.*
    - *Often, it must be the last thread to finish execution because it performs various shutdown actions.*

Multithreading

- 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.

```
class CurrentThreadDemo {
public static void main(String args[]) {
        Thread t = Thread.currentThread();
        System.out.println("Current thread: " + t);
        // change the name of the thread

        System.out.println("Name before change " + t.getName());
        t.setName("My Thread");
        System.out.println("After name change: " + t);

        try {
                for(int n = 5; n > 0; n--) {
                   System.out.println(n);
                   Thread.sleep(1000);
                }
        } catch (InterruptedException e) {
                System.out.println("Main thread interrupted");
        }
   }
}
```

OUTPUT

```
D:\java\bin>java CurrentThreadDemo
Current thread: Thread[main,5,main]
Name before change main
After name change: Thread[My
Thread,5,main]
5
4
3
2
1
```

- In the above output, *[main,5,main],* first main refers to the name of the thread, 5 refers to the default priority of the main thread and last refers to the thread group name which is also main.
- The *sleep()* method is used to pause the current thread (main thread) for certain amount of time which is specified in milliseconds.

Multithreading

- 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.

Creation of a thread by Implementing Runnable Interface

- Following are the steps that must be performed for creating a thread using *Runnable* interface.

1. Create a class and implement *Runnable* interface.
2. Implement *run()* method. This method contains the thread code and is the entry point for every thread.
3. Instantiate a thread object using the constructor *Thread(Runnable obj,String stringName)*
4. Once a thread object is created, you can start it by calling start() method which executes a call to *run()* method.

```
class MyThread implements Runnable {
        MyThread(String name) {
                Thread t = new Thread(this,name);
                System.out.println("Child thread: " + t);
                t.start();
        }
       public void run(){
                try{
                     for(int i = 1; i <= 3; i++) {
                                System.out.println("Child thread: " + i);
                                Thread.sleep(500);
                      }
                }
                catch(InterruptedException e){
                       System.out.println("Child thread is
interrupted!");
                }
                System.out.println("Child thread terminated");
    }
}
```

```
public class ThreadDemo {
   public static void main(String[] args){
       new MyThread("Demo Thread");
       try{
           for(int i = 1; i <= 3; i++) {
               System.out.println("Main thread: " + i);
               Thread.sleep(1000);
           }
         }
         catch(InterruptedException e) {
               System.out.println("Main thread is
interrupted!");
}
               System.out.println("Main thread
terminated");
}
}
```

-
  ```
  Child thread: Thread[Thread-0,5,main]
  Main thread: 1
  Child thread: 1
  Child thread: 2
  Main thread: 2
  Child thread: 3
  Child thread terminated
  Main thread: 3
  Main thread terminated
  ```

Multithreading

- 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.

```
class MyThread extends Thread {
        MyThread(String name) {
                super(name);
                System.out.println("Child thread: " + this);
                start();
        }
       public void run(){
                try{
                     for(int i = 1; i <= 3; i++) {
                                System.out.println("Child thread: " + i);
                                Thread.sleep(500);
                      }
                }
                catch(InterruptedException e){
                       System.out.println("Child thread is
interrupted!");
                }
                System.out.println("Child thread terminated");
    }
}
```

```
public class ThreadDemo {
   public static void main(String[] args){
       new MyThread("Demo Thread");
       try{
            for(int i = 1; i <= 3; i++) {
                System.out.println("Main thread: " + i);
                Thread.sleep(1000);
            }
         }
         catch(InterruptedException e) {
               System.out.println("Main thread is
interrupted!");
}
                System.out.println("Main thread
terminated");
}
}
```

Multithreading

- The **Thread** class defines several methods that can be overridden by a derived class.
- Of these methods, the only one that must be overridden is **run( )**. This is, of course, the same method required when you implement Runnable.
- Many Java programmers feel that **classes should be extended only when they are being enhanced or modified in some way**.

Multithreading

- So, if you will not be overriding any of Thread’s other methods, it is probably best simply to implement **Runnable**.
- This is up to you, of course.

Multithreading

- 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 not a satisfactory solution
- Two ways exist to determine whether a thread has finished. First, you can call **isAlive( )** on the thread.
- The **isAlive()** *method returns true if the thread upon which it is called is still running*. It returns false otherwise.

Multithreading

- The method that you will more commonly use to wait for a thread to finish is called **join( )**
- *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 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 + ": " + i);
        Thread.sleep(1000);
      }
     } catch (InterruptedException e) {
        System.out.println(name + "
   interrupted.");
} System.out.println(name + " exiting.");
} }
```

```
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());
  // wait for threads to finish
  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."); }
    }
```

Multithreading

- 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***.
- Key to synchronization is the concept of the ***monitor*** (also called a ***semaphore***).
- A monitor is an object that is used for obtaining a mutually exclusive lock, or ***mutex***.
- Once a thread acquires a lock, it is said to have entered the monitor.
- When one thread is inside the monitor, no other thread is allowed to acquire a lock until that thread exits the monitor.

Multithreading

- 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.
- Java implements synchronization through language elements, most of the complexity associated with synchronization has been eliminated.
- You can synchronize your code in either of two ways. Both involve the use of the ***synchronized*** keyword

- Every object in Java has an implicit monitor associated with it.
- To enter an object’s monitor, a method which is modified using *synchronized* keyword should be called.
- Using *synchronized* keyword we can create:
    - Synchronized methods
    - Synchronized blocks

```
class MyName{
   void printName(){
      try{
           System.out.print("[Hello ");
          Thread.sleep(200);
          System.out.print(“synchronized ");
          Thread.sleep(500);
          System.out.println(“word");
    }
    catch(Exception e){
         System.out.println("Thread Interrupted");}
    }
}
class MyThread implements Runnable {
    Thread t;
    MyName myname;
    MyThread(MyName myname){
        this.myname = myname;
        t = new Thread(this);
        t.start();
  }
  public void run() {
    myname.printName(); }
}
```

```
public class Driver{
 public static void main(String[] args){
    MyName myname = new MyName();
    MyThread thread1 = new MyThread(myname);
    MyThread thread2 = new MyThread(myname);
    MyThread thread3 = new MyThread(myname);
    try {
            thread1.t.join();
            thread2.t.join();
            thread3.t.join();
    }
    catch(InterruptedException e){
        System.out.println("Main thread is
interrupted!");
   }
   System.out.println("Main thread terminated");
  }
}
```

Synchronized Methods

- A method that is modified with the *synchronized* keyword is called a synchronized method. Syntax of a synchronized method is as follows:

```
synchronized return-type method-name(parameters)
{ … }
```

```
class MyName{
   synchronized void printName(){
      try{
           System.out.print("[Hello ");
           Thread.sleep(200);
           System.out.print("synchronized ");
           Thread.sleep(500);
           System.out.println("word");
    }
    catch(Exception e){
          System.out.println("Thread Interrupted");}
    }
}
class MyThread implements Runnable {
    Thread t;
    MyName myname;
    MyThread(MyName myname){
         this.myname = myname;
         t = new Thread(this);
         t.start();
   }
   public void run() {
     myname.printName(); }
}
```

```
public class Driver{
  public static void main(String[] args){
    MyName myname = new MyName();
    MyThread thread1 = new MyThread(myname);
    MyThread thread2 = new MyThread(myname);
    MyThread thread3 = new MyThread(myname);
    try {
            thread1.t.join();
            thread2.t.join();
            thread3.t.join();
    }
    catch(InterruptedException e){
        System.out.println("Main thread is
interrupted!");
   }
   System.out.println("Main thread terminated");
  }
}
```

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.

Synchronized Blocks

- 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.
- If 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.
- If the 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.

Synchronized Blocks

- The solution to this problem is 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
  - }
- 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.

```
class MyName{
   void printName(){
      try{
           System.out.print("[Hello ");
           Thread.sleep(200);
           System.out.print(“synchronized ");
           Thread.sleep(500);
           System.out.println(“word");
    }
    catch(Exception e){
          System.out.println("Thread Interrupted");}
    }
}
class MyThread implements Runnable {
    Thread t;
    MyName myname;
    MyThread(MyName myname){
         this.myname = myname;
         t = new Thread(this);
         t.start();
   }
   public void run() {
    synchronized(myname){
     myname.printName();
    }
  }
}
```

```
public class Driver{
  public static void main(String[] args){
     MyName myname = new MyName();
     MyThread thread1 = new MyThread(myname);
     MyThread thread2 = new MyThread(myname);
     MyThread thread3 = new MyThread(myname);
     try {
              thread1.t.join();
              thread2.t.join();
              thread3.t.join();
     }
     catch(InterruptedException e){
          System.out.println("Main thread is 
interrupted!");
    }
    System.out.println("Main thread terminated");
   }
}
```

Interthread Communication

- All these methods belong to Object class as final so that all classes have them.
- They must be used within a synchronized block only.
- **wait()-**It tells the calling thread to give up the lock and go to sleep until some other thread enters the same monitor and calls notify().
- **notify()-**It wakes up one single thread that called wait() on the same object. It should be noted that calling notify() does not actually give up a lock on a resource.
- **notifyAll()-**It wakes up all the threads that called wait() on the same object.

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