Unit-III

Introduction

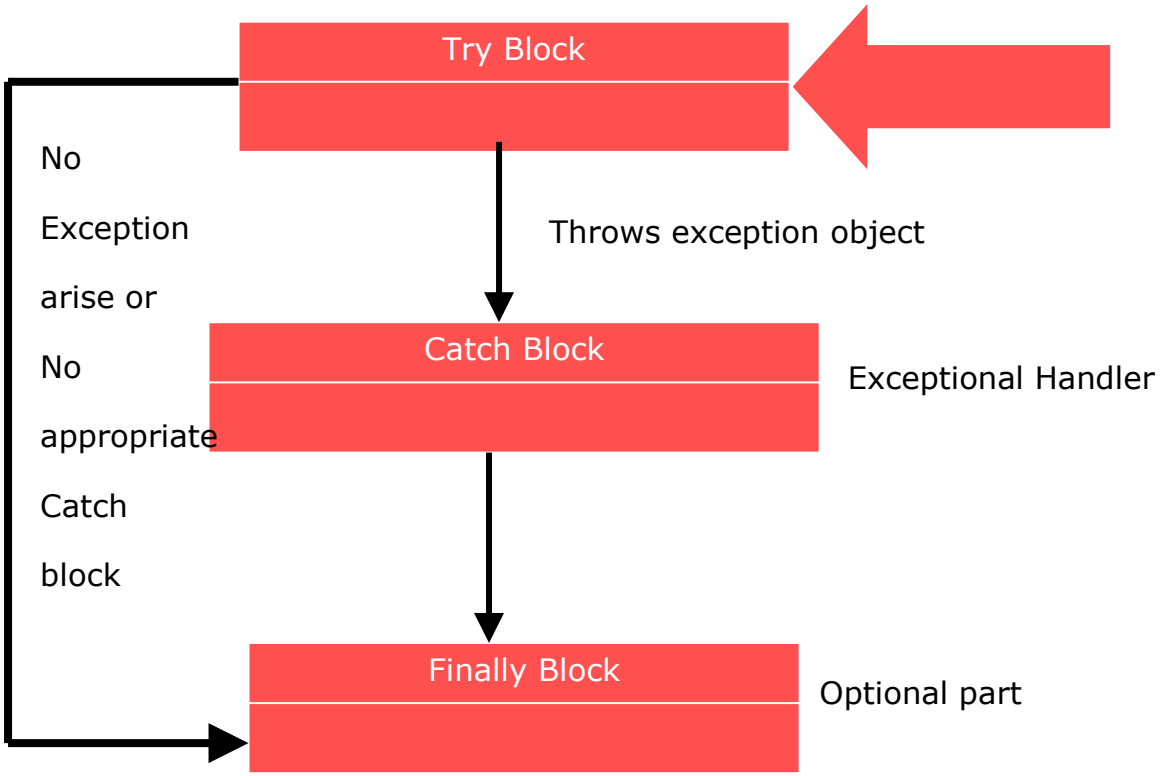
- An exception is an event that occurs during the execution of a program that disrupts the normal flow of instruction.

Or

- An abnormal condition that disrupts Normal program flow.
- There are many cases where abnormal conditions happen during program execution, such as
  - Trying to access an out - of – bounds array elements.
  - The file you try to open may not exist.
  - The file we want to load may be missing or in the wrong format.
  - The other end of your network connection may be non – existence.
- If these cases are not prevented or at least handled properly, either the program will be aborted abruptly, or the incorrect results or status will be produced.
- When an error occurs with in the java method, the method creates an exception object and hands it off to the runtime system.
- The exception object contains information about the exception including its type and the state of the program when the error occurred. The runtime system is then responsible for finding some code to handle the error.
- In java creating an exception object and handling it to the runtime system is called throwing an exception.
- Exception is an object that is describes an exceptional (i.e. error) condition that has occurred in a piece of code at run time.
- When a 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.
- System generated exceptions are automatically thrown by the Java runtime system

General form of 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
}
```



- By using exception to managing errors, Java programs have have the following advantage over traditional error management techniques:
  - Separating Error handling code from regular code.
  - Propagating error up the call stack.
  - Grouping error types and error differentiation.

For Example:

```
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. In this example, we haven't supplied any exception handlers of our own, so the 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. Here is the output generated when this example is executed.

```
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**

Try and Catch Blocks

- If we don't want to prevent the program to terminate by the exception we have to trap the trap the exception using the try block. So we can place the statements that may causes an exception in the try block.

```
Try
{
}
```

- If an exception occurs with in the try block, the appropriate exception handler that is associated with the try block handles the exception immediately following the try block, include a catch clause specifies the exception type we wish to catch. A try block must have at least one catch block or finally that allows it immediately.

Catch block

- The catch block is used to process the exception raised. A try block can be one or more catch blocks can handle a try block.
- Catch handles are placed immediately after the try block.

```
Catch(exceptiontype e)
{
    //Error handle routine is placed here for handling exception
}
```

Program 1

```
Class trycatch
{
Public static void main(String args[])
{
Int[] no={1,2,3};
Try
{
System.out.println(no[3]);
}
Catch(ArrayIndexOutOfBoundsException e)
{
System.out.println("Out of bonds");
}
System.out.println("Quit");
}
}
```

Output

```
Out of the Range
Quit
```

Program 2

```
Class ArithExce
{
Public static void main(String args[])
{
Int a=10;
Int b=0;
Try
{
  a=a/b;
System.out.println(“Won’t Print”);
}
Catch(ArithmeticException  e)
{
System.out.println(“Division by Zero error”);
System.out.println(“Change the b value”);

}
System.out.println(“Quit”);
}
}
```

Output

```
Division By zero error
Please change the B value
Quit
```

Note:

- A try ad its catch statement form a unit.
- We cannot use try block alone.
- The compiler does not allow any statement between try block and its associated catch block

Displaying description of an Exception

- Throwable overrides the toString() method (defined by Object) so that it returns a string containing a description of the exception.
- We can display this description in a println statement by simply passing the exception as an argument.

```
catch (ArithmeticException e) {
System.out.println("Exception: " + e);
a = 0; // set a to zero and continue
}
```

- When this version is substituted in the program, and the program is run, each divide-by-zero error displays the following message:
  - Exception: java.lang.ArithmeticException: / by zero

Multiple Catch Blocks

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 commandline parameters, since **a** will equal zero. It will survive the division if you provide a command-line 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
After try/catch blocks.
```

Throw Keyword

- So far, we have only been catching exceptions that are thrown by the Java Run – Time systems. How ever, it is possible for our program to throw an exception explicitly, using the throw statement.
    - 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
  - 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

```
// 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 at least 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 Keyword

- 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. 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

Program

```
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

Finally block

- 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.

```
// Demonstrate finally.
class FinallyDemo {
// Through an exception out of the method.
static void procA() {
try {
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");
}
}
// Execute a try block normally.
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();
}
}
```

- 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. *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
```

Java Built – In 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 Table 10-1. Table 10-2 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

List of Unchecked 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.
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.

Exception	Meaning
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.

List of Checked exceptions

Exception	Meaning
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does r 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.

User defined exceptions

- We can create our own exception by extending exception class.
- The throw and throws keywords are used while implementing user defined exceptions

```
Class ownExcepion extends Exception
{
ownException(String msg)
{
Super(msg);
}
}
```

```
Class test
{
Public static void main(String args[])
Int mark=101;
Try
{
if(mark>100)
{
Throw new ownException(“Marks>100”);
}
}
Catch(ownException e)
{
System.out.println (“Exception caughtr”);
System.out.println.(“e.getMessage());
}
Finally
{
System.out.println(“End of prg”);
}
}
}
```

Output:

Exception caught
Marks is > 100
End of program

Multi Threaded Programming

Introduction:

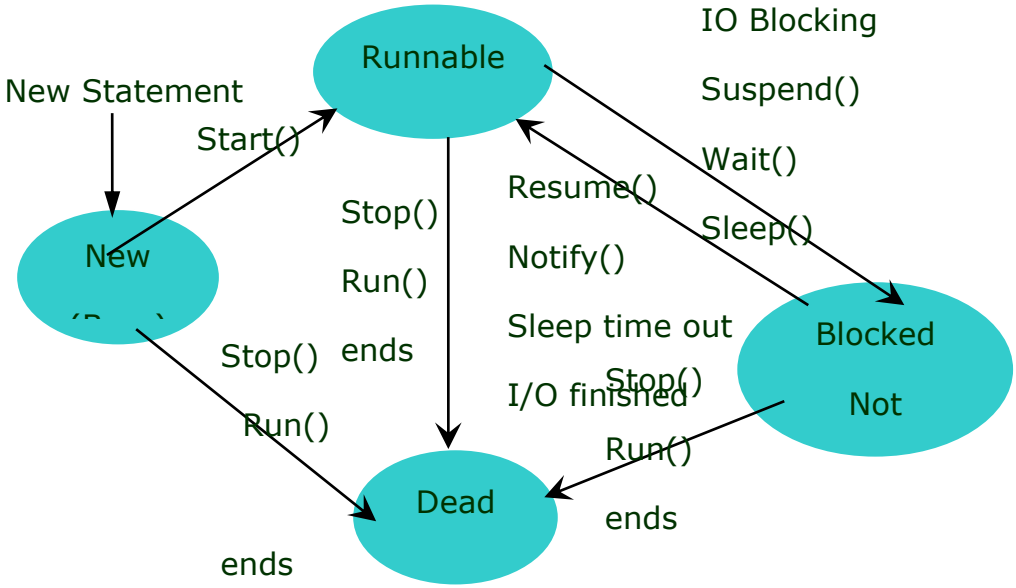
- Java provides a built – in support for multithreaded programming.
- A multithreaded program contains two o more parts that can run concurrently.
- Each part of such a program called thread.
- Each thread defines a separate path of execution.
- Thus multi thread is a specialized form of multi tasking.
- Multi tasking is supported by OS
- There are two distinct types of multi tasking
- Process based multi tasking
- Process is a program that is executing.
- In process based multi tasking, a program is the smallest unit of code that can be dispatched by the scheduler
- Process based multi tasking is a feature that allows computer to run two or more programs concurrently
- For example :
- This tasking enables us to run the Java compiler and text editor at the same time

- Thread based multi tasking
- Thread is a smallest unit of dispatchable code
- The single program can perform two or more tasks simultaneously.
- For example:
- A text editor can format text at the same time that is printing as long as these two actions are performed by two separate threads.
- Multitasking threads require less overhead than multitasking processes.

Thread Model

- One thread can pause without stopping other parts of your program.
  - For example, the idle time created when a thread reads data from a network or waits for user input can be utilized elsewhere.
  - Multithreading allows animation loops to sleep for a second between each frame without causing the whole system to pause.
- When a thread blocks in a Java program, only the single thread that is blocked pauses. All other threads continue to run.

- Thread States
  - Threads exist in several states.
    - A thread can be *running.* It can be *ready to run* as soon as it gets CPU time.
    - A running thread can be *suspended,* which temporarily suspends its activity.
    - A suspended thread can then be *resumed,* allowing it to pick up where it left off.
    - A thread can be *blocked* when waiting for a resource.
    - At any time, a thread can be terminated, which halts its execution immediately.
    - Once terminated, a thread cannot be resumed
- Every thread after creation and before destruction can have any one of four states:
  - Newly created
  - Runnable state
  - Blocked
  - Dead



THREAD LIFE CYCLE

New State

- A thread enters the newly created by using a new operator.
- It is new state or born state immediately after creation. i.e. when a constructor is called the Thread is created but is not yet to run() method will not begin until it start() method is called.
- After the start() method is called, the thread will go to the next state, Runnable state.
- Note : in the above cycle stop(), resume() and suspand are deprecated methods. Java 2 strongly discourage their usage in the program

- **Runnable state**
  - Once we invoke the start() method, the thread is runnable.
  - It is divided into two states:
    - The running state
      - When the thread is running state, it assigned by CPU cycles and is actually running.
    - The Queued state.
      - When the thread is in Queued state, it is waiting in the Queue and competing for its turn to spend CPU cycles
      - It is controlled by Virtual Machine Controller.
      - When we use yield() method it makes sure other threads of the same priority have chance to run.
      - This method cause voluntary move itself to the queued state from the running state.

Blocked state

- The blocked state is entered when one of the following events occurs:
  - The thread itself or another thread calls the suspend() method (it is deprecated)
  - The thread calls an object's wait() method

- The thread itself calls the sleep() method.
- The thread is waiting for some I/O operations to complete.
- The thread will join() another thread.

Dead state

- A thread is dead for any one of the following reasons:
  - It dies a natural death because the un method exists normally.
  - It dies abruptly because an uncaught exception terminates the run method.
  - In particular stop() is used to kill the thread. This is depricated.
  - To find whether thread is alive i.e. currently running or blocked
    - Use isAlive() method
      - If it returns true the thread is alive

Thread priorities

- Java assigns to each thread a priority that determines how that thread should be treated with respect to the others. Thread priorities are integers that specify the relative priority of one thread to another. As an absolute value, a priority is meaningless; a higher-priority thread doesn't run any faster than a lower-priority thread if it is the only thread running. Instead, a thread's 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 simple:
  - *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.
  - *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—no matter what it is doing—by a higher-priority thread. Basically, as soon as a higher-priority thread wants to run, it does. This is called *preemptive multitasking.*

- In cases where two threads with the same priority are competing for CPU cycles, the situation is a bit complicated. For operating systems such as Windows 98, threads of equal priority are time-sliced automatically in round-robin fashion. For other types of operating systems, threads of equal priority must voluntarily yield control to their peers. If they don't, the other threads will not run.

Synchronization

- Because multithreading introduces an asynchronous behavior to your programs, there must be a way for you to enforce synchronicity when you need it. For example, if you want two threads to communicate and share a complicated data structure, such as a linked list, you need some way to ensure that they don't conflict with each other. That is, you must prevent one thread from writing data while another thread is in the middle of reading it. For this purpose, Java implements an elegant twist on an age-old model of interprocess synchronization: the *monitor*. The monitor is a control mechanism first defined by C.A.R. Hoare.

You can think of a monitor as a very small box that can hold only one thread. Once a thread enters a monitor, all other threads must wait until that thread exits the monitor. In this way, a monitor can be used to protect a shared asset from being manipulated by more than one thread at a time.

.

Messaging

- After you divide your program into separate threads, you need to define how they will communicate with each other. When programming with most other languages, you must depend on the operating system to establish communication between threads. This, of course, adds overhead. By contrast, Java provides a clean, low-cost way for two or more threads to talk to each other, via calls to predefined methods that all objects have. Java's messaging system allows a thread to enter a synchronized method on an object, and then wait there until some other thread explicitly notifies it to come out.

Thread class and Runnable interface

The Thread Class and the Runnable InterfaceJava's multithreading system is built upon the **Thread** class, its methods, and its companion interface, **Runnable**. **Thread** encapsulates a thread of execution. Since you can't directly refer to the ethereal state of a running thread, you will deal with it through its proxy, the **Thread** instance that spawned it. 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.

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.

Main method

- 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 .
  - Often it must be the last thread to finish execution because it performs various shutdown actions.

- 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
  - 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.

```
// Controlling the main Thread.

class CurrentThreadDemo {
public static void main(String args[]) {
Thread t = Thread.currentThread();
System.out.println("Current thread: " + t);
// change the name of the thread
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");
}
}
}
Current thread: Thread[main,5,main]
After name change: Thread[My Thread,5,main]
5
4
3
2
1
```

How to create 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.

Implementing thread class

- 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( )

Extending thread class

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**

```
// Create a second thread by extending Thread
class NewThread extends Thread {
NewThread() {
// Create a new, second thread
super("Demo Thread");
System.out.println("Child thread: " + this);
start(); // Start the thread
}
// This is the entry point for the second thread.
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 ExtendThread {
public static void main(String args[]) {
new NewThread(); // create a new thread
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.");
}
```

}

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.* As you will see, Java provides unique, language-level support for it. Key to synchronization is the concept of the monitor (also called a *semaphore*). 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.

You can synchronize your code in either of two ways. Both involve the use of the **synchronized** keyword, and both are examined here.

Using Synchronized Methods

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

While creating **synchronized** methods within classes that you create is an easy and effective means of achieving synchronization, it will not work in all cases. To understand why, consider the following. Imagine that you want to synchronize access to objects of a class that was not designed for multithreaded access. That is, the class does not use **synchronized** methods. Further, this class was not created by you, but by a third party, and you do not have access to the source code. Thus, you can't add **synchronized** to the appropriate methods within the class. How can access to an object of this class be synchronized? Fortunately, the solution to this problem is quite easy: You simply put calls to the methods defined by this class inside a **synchronized** block.

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

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

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

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 {
String msg;
Callme target;
Thread t;
public Caller(Callme targ, String s) {
target = targ;
msg = s;
t = new Thread(this);
t.start();
}
// 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");
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.

Daemon Threads

A "daemon" thread is one that is supposed to provide a general service in the background as long as the program is running, but is not part of the essence of the program. Thus when all of the non-daemon threads complete, the program is terminated. you an find out if a thread is a daemon by calling isDaemon(), and you can turn the "daemonhood" of a thread on and off

with setDaemon().if a thread is a daemon, then any threads it creates will automatically be daemons.