# Jain College of Engineering and Research

## DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING (AIML)

**OOP with JAVA BCS306A**

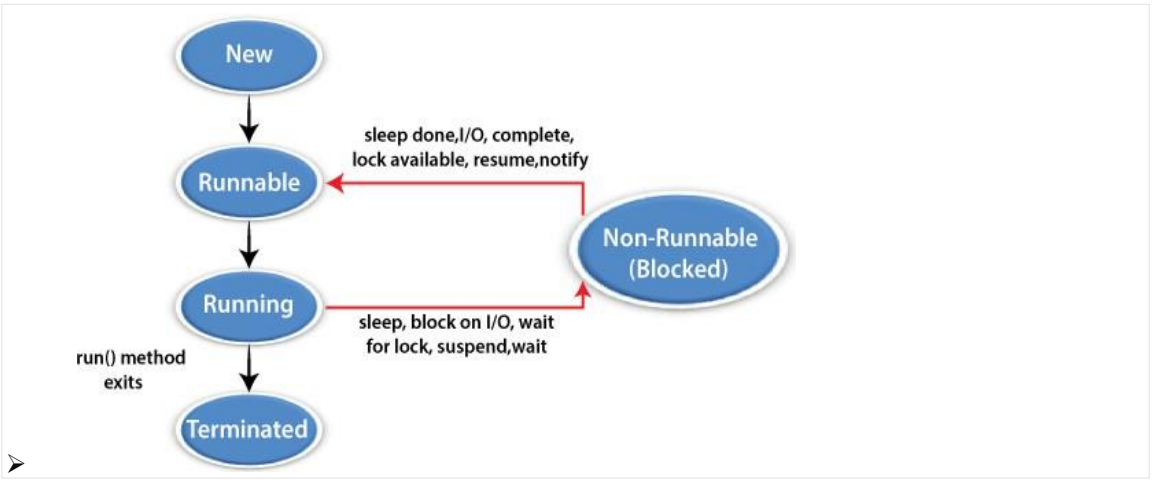
# Module -5

## Multithreaded Programming

- Multithreading in Java allows for concurrent execution of multiple parts of a program, known as threads.
- This contrasts with process-based multitasking, where each program is a separate unit, whereas threads share the same address space and belong to the same process.
- Multithreading is advantageous due to its lower overhead compared to process-based multitasking.
- It helps in maximizing processing power by minimizing idle time, especially in interactive and networked environments where tasks like data transmission and user input are slower compared to CPU processing speed.
- Multithreading enables more efficient use of available resources and smoother program execution by allowing other threads to run while one is waiting.

## The Java Thread Model

- Java's runtime system heavily relies on threads to enable asynchronous behavior, which helps in utilizing CPU cycles more efficiently.
- Unlike single-threaded systems that use event loops with polling, Java's multithreading eliminates the need for such mechanisms. In single-threaded environments, blocking one thread can halt the entire program, leading to inefficiencies and potential domination of one part over others.
- With Java's multithreading, one thread can pause without affecting other parts of the program, allowing idle time to be utilized elsewhere.
- This is particularly beneficial for tasks like animation loops, where pauses between frames don't halt the entire system. Multithreading in Java works seamlessly on both single-core and multicore systems, with threads sharing CPU time on single-core systems and potentially executing simultaneously on multicore systems.
- Threads in Java can exist in various states, including running, ready to run, suspended, blocked, or terminated. Each state represents a different stage of thread execution, with the ability to suspend, resume, or terminate threads as needed.
- Overall, Java's multithreading capabilities contribute to more efficient and responsive software development.




## Thread Priorities

- **Thread Priorities** : Java assigns each thread a priority to determine its relative importance. Higher-priority threads are given preference during context switches, but priority doesn't affect the speed of execution.
- **Thread States**: Threads can be in various states like running, ready to run, suspended, blocked, or terminated. These states govern the behavior of threads in the system.
- **Synchronization**: Java provides mechanisms like monitors to enforce synchronicity between threads, ensuring that shared resources are accessed safely. Synchronization is achieved through the use of synchronized methods and blocks.
- **Messaging**: Java facilitates communication between threads through predefined methods that all objects have. This messaging system allows threads to wait until they are explicitly notified by another thread.
- **Thread Class and Runnable Interface** : Java's multithreading system is built around the **Thread** class and the **Runnable** interface. Threads can be created either by extending the **Thread** class or implementing the **Runnable** interface.
- **Main Thread**: Every Java program starts with a main thread, which is automatically created. The main thread is crucial for spawning other threads and often performs shutdown actions at the end of the program.
- **Thread Methods** : Java's **Thread** class provides various methods for managing threads, including **getName()**, **getPriority()** , **isAlive()**, **join()**, **run()**, **sleep()**, and **start()**.

## Creating a Thread

Implementing Runnable Interface: To create a thread, you implement the Runnable interface in a class. This interface abstracts a unit of executable code and requires implementing a single method called run().

Runnable's run() Method: Inside the run() method, you define the code that constitutes the new thread. This method can call other methods, use other classes, and declare variables just like the main thread can.

Instantiating Thread Object: After implementing Runnable, you instantiate an object of type Thread within that class. The Thread constructor requires an instance of a class that implements Runnable and a name for the thread.

Starting the Thread: The new thread doesn't start running until you call its start() method. This method initiates a call to run(), effectively starting the execution of the new thread.

**Example: An example code snippet demonstrates creating and starting a new thread:**

```
class NewThread implements Runnable {
   Thread t;

   NewThread() {
      // Create a new, second thread
      t = new Thread(this, "Demo Thread");
      System.out.println("Child thread: " + t);
   }

   // 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 ThreadDemo {
   public static void main(String[] args) {
      NewThread nt = new NewThread(); // create a new thread
      nt.t.start(); // Start the thread
      // Main thread continues its execution
      // ...
   }
}
```

**Extending Thread**

Extending Thread Class: To create a thread, you create a new class that extends the Thread class. The extending class must override the run() method, which serves as the entry point for the new thread.

Constructor Invocation: Inside the constructor of the extending class, you can invoke the constructor of the Thread class using super() to specify the name of the thread.

Starting the Thread: After creating an instance of the extending class, you call the start() method to begin execution of the new thread.

Example:

```
class NewThread extends Thread {
NewThread() {
// Invoke Thread constructor to set thread name
super("Demo Thread");
System.out.println("Child thread: " + this);
}
```

```
// 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) {
NewThread nt = new NewThread(); // create a new thread
nt.start(); // start the thread
// Main thread continues its execution
// ...
}
}
```

**Creating Multiple 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);
   }

   // This is the entry point for 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 MultiThreadDemo {
   public static void main(String[] args) { NewThread
      nt1 = new NewThread("One"); NewThread nt2 =
      new NewThread("Two"); NewThread nt3 = new
      NewThread("Three");

      // Start the threads. nt1.t.start();
      nt2.t.start();
      nt3.t.start();

      try {
         // wait for other threads to end
         Thread.sleep(10000);
      } catch (InterruptedException e) { System.out.println("Main
         thread Interrupted");
      }

      System.out.println("Main thread exiting.");
   }
}
```

Sample output from this program is shown here. (Your output may vary based upon the specific execution environment.)

```
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.

## Using isAlive( ) and join( )

Using the **isAlive()** and **join()** methods in Java threads:

- **isAlive() Method**:
  - Defined by the **Thread** class.
  - Returns **true** if the thread upon which it is called is still running.
  - Returns **false** otherwise.
  - Occasionally useful for checking the status of a thread.
- **join() Method**:
  - Also defined by the **Thread** class.
  - Waits until the thread on which it is called terminates.
  - The calling thread waits until the specified thread joins it.
  - Additional forms of **join()** allow specifying a maximum amount of time to wait for the specified thread to terminate.
- **Usage**:
  - **join()** is commonly used to ensure that one thread waits for another thread to finish its execution.
  - This is particularly useful when you want the main thread to finish last or when you need to synchronize the execution of multiple threads.
- **Example**:
  - An improved version of the example code can use **join()** to ensure that the main thread is the last to stop.
  - Additionally, **isAlive()** can be used to check if a thread is still running.

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

   // This is the entry point for 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
          nt1 = new NewThread("One"); NewThread nt2 =
          new NewThread("Two"); NewThread nt3 = new
          NewThread("Three");

          // Start the threads. nt1.t.start();
          nt2.t.start();
          nt3.t.start();

          System.out.println("Thread One is alive: "
                                + nt1.t.isAlive()); System.out.println("Thread
          Two is alive: "
                                + nt2.t.isAlive()); System.out.println("Thread
          Three is alive: "
                                + nt3.t.isAlive());
          // wait for threads to finish try {
            System.out.println("Waiting for threads to finish."); nt1.t.join();
            nt2.t.join();
            nt3.t.join();
          } catch (InterruptedException e) { System.out.println("Main
            thread Interrupted");
          }

          System.out.println("Thread One is alive: "
                                + nt1.t.isAlive()); System.out.println("Thread
          Two is alive: "
                                + nt2.t.isAlive()); System.out.println("Thread
          Three is alive: "
                                + nt3.t.isAlive());

          System.out.println("Main thread exiting.");
        }

      }
```

## Thread Priorities

- **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 over a given period of time.
  - Higher-priority threads can preempt lower-priority ones, meaning they can interrupt the execution of lower-priority threads.

- **Equal Priority Threads**:
  - In theory, threads of equal priority should get equal access to the CPU.
  - However, Java is designed to work in various environments, and the actual behavior may differ depending on the operating system and multitasking implementation.
  - To ensure fairness, threads that share the same priority should yield control occasionally, especially in nonpreemptive environments.
- **Setting Thread Priority**:
  - Use the **setPriority()** method to set a thread's priority.
  - Syntax: **void setPriority(int level)**
  - The **level** parameter specifies the new priority setting for the thread, and it must be within the range of **MIN_PRIORITY** and **MAX_PRIORITY**, currently 1 and 10, respectively.
  - To return a thread to default priority, use **NORM_PRIORITY**, which is currently 5.
- **Getting Thread Priority**:
  - Use the **getPriority()** method to obtain the current priority setting of a thread.
  - Syntax: **int getPriority()**
- **Implementation Considerations**:
  - Implementations of Java may have different behaviors when it comes to scheduling and thread priorities.
  - To ensure predictable and cross-platform behavior, it's advisable to use threads that voluntarily give up CPU time.

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

- **Implicit Monitors**:
  - All objects in Java have their own implicit monitor associated with them.
  - To enter an object's monitor, you can call a method that has been modified with the **synchronized** keyword.
  - While a thread is inside a **synchronized** method of an object, all other threads that try to call synchronized methods on the same instance have to wait.
- **Need for Synchronization** :
  - Synchronization is necessary to ensure thread safety, especially when multiple threads access shared resources concurrently.
  - Without synchronization, concurrent access to shared resources can lead to data corruption, race conditions, and other inconsistencies.

- **Example**:
  - The example program consists of three classes: **Callme**, **Caller**, and **Synch**.
  - The **Callme** class has a method **call()** which prints a message inside square brackets and then pauses the thread for one second using **Thread.sleep(1000)**.
  - The **Caller** class takes a reference to an instance of **Callme** and a message string. It creates a new thread that calls the **run()** method which in turn calls the **call()** method on the **Callme** instance.
  - The **Synch** class creates a single instance of **Callme** and three instances of **Caller**, each with a unique message string. All **Caller** instances share the same **Callme** instance.

```
// 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");

      // Start the threads. ob1.t.start();
      ob2.t.start();
      ob3.t.start();

      // wait for threads to end try {
```

```
            ob1.t.join();
            ob2.t.join();
            ob3.t.join();
        } catch(InterruptedException e) {
            System.out.println("Interrupted");
        }
    }
}
```

Here is the output produced by this program:

```
[Hello[Synchronized[World]
]
]
```

## Interthread Communication

Interprocess communication using **wait()**, **notify()**, and **notifyAll()** methods in Java:

- **Purpose**:
  - These methods provide a means for threads to communicate and coordinate their activities without using polling, which can waste CPU cycles.
- **Method Definitions**:
  - **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()** or **notifyAll()**.
  - **notify()**: Wakes up a single thread that previously called **wait()** on the same object.
  - **notifyAll()**: Wakes up all threads that previously called **wait()** on the same object. One of the threads will be granted access.
  - All three methods are declared within the **Object** class and can only be called from within a synchronized context.

**Additional Forms of wait()**:

- Additional forms of the **wait()** method exist that allow you to specify a period of time to wait.

- **Spurious Wakeups**:
  - In rare cases, a waiting thread could be awakened due to a spurious wakeup, where **wait()** resumes without **notify()** or **notifyAll()** being called. To handle this, calls to **wait()** are often placed within a loop that checks the condition on which the thread is waiting.
- **Best Practices**:
  - The Java API documentation recommends using a loop to check conditions when waiting, especially due to the possibility of spurious wakeups.

```
// An incorrect implementation of a producer and consumer.

 class Q {
    int n;

    synchronized int get() {
        System.out.println("Got: " + n); return
        n;
    }

    synchronized void put(int n) { this.n
        = n; System.out.println("Put: " +
        n);
    }
}

class Producer implements Runnable {
    Q q;
    Thread t;

    Producer(Q q)
        { this.q = q;
        t = new Thread(this, "Producer");
    }

    public void run() {
        int i = 0;

        while(true) {
            q.put(i++);



                }
            }
          }

          class Consumer implements Runnable {
              Q q;
              Thread t;

              Consumer(Q q) {
                  this.q = q;
                  t = new Thread(this, "Consumer");
              }

              public void run() {
```

```
        while(true) {
            q.get();
        }
    }
}

class PC {
    public static void main(String[] args) { Q q =
        new Q();
        Producer p = new Producer(q);
        Consumer c = new
        Consumer(q);

        // Start the threads. p.t.start();
        c.t.start();

        System.out.println("Press Control-C to stop.");
    }
}
```

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):

```
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
```

```
// A correct implementation of a producer and consumer. class Q
{
    int n;
    boolean valueSet = false;

    synchronized int get() { while(!valueSet)
            try {
                wait();

            } catch(InterruptedException e) { System.out.println("InterruptedException
                caught");
            }

            System.out.println("Got: " + n);
            valueSet = false;
            notif
            y();
            retur
            n n;
    }

    synchronized void put(int n) {
        while(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;
    Thread t;

    Producer(Q q)
        { this.q = q;
        t = new Thread(this, "Producer");
    }

    public void run() {
        int i = 0;
```

```
        while(true) {
            q.put(i++);
        }
    }
}

class Consumer implements Runnable {
    Q q;
    Thread t;

    Consumer(Q q) {
        this.q = q;
        t = new Thread(this, "Consumer");
    }

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

class PCFixed {
    public static void main(String[] args) { Q q =
        new Q();
        Producer p = new Producer(q);
        Consumer c = new
        Consumer(q);

        // Start the threads. p.t.start();
        c.t.start();

        System.out.println("Press Control-C to stop.");
    }
}
```

Inside **get( )**, **wait( )** is called. This causes its execution to suspend until **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 **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 **Consumer** that it should now remove it.

Here is some output from this program, which shows the clean synchronous behavior:

```
Put:  1
Got:  1
Put:  2
Got:  2
```

```
Put:  3
Got:  3
Put:  4
Got:  4
Put:  5
Got:  5
```

## Suspending, Resuming, and Stopping Threads

- **Deprecated Methods**:
  - In early versions of Java (prior to Java 2), thread suspension, resumption, and termination were managed using **suspend()**, **resume()**, and **stop()** methods defined by the **Thread** class.
  - However, these methods were deprecated due to potential issues and risks they posed, such as causing system failures and leaving critical data structures in corrupted states.
- **Reasons for Deprecation**:
  - **suspend()**: Can cause serious system failures, as it doesn't release locks on critical data structures, potentially leading to deadlock.
  - **resume()**: Deprecated as it requires **suspend()** to work properly.
  - **stop()**: Can cause system failures by leaving critical data structures in corrupted states.
- **Alternative Approach**:
  - Instead of using deprecated methods, threads should be designed to periodically check a flag variable to determine whether to suspend, resume, or stop their own execution.
  - Typically, a boolean flag variable is used to indicate the execution state of the thread.
  - If the flag is set to "running," the thread continues to execute. If it's set to "suspend," the thread pauses. If it's set to "stop," the thread terminates.
- **Example Using wait() and notify()**:
  - The **wait()** and **notify()** methods inherited from **Object** can be used to control the execution of a thread.
  - An example provided demonstrates how to use these methods to control thread execution.
  - It involves a boolean flag (**suspendFlag**) to control the execution of the thread.
  - The **run()** method periodically checks **suspendFlag**, and if it's **true**, the thread waits. Methods **mysuspend()** and **myresume()** are used to set and unset the flag and notify the thread to wake up.

```
// Suspending and resuming a thread the modern way. class
NewThread implements Runnable {
   String name; // name of thread
   Thread t;

            boolean suspendFlag;

            NewThread(String threadname) {
               name = threadname;
               t = new Thread(this, name);
               System.out.println("New thread: " + t);
               suspendFlag = false;
}
```

```
    // This is the entry point for thread.
    public void run() {
        try {
            for(int i = 15; i > 0; i--) {
                System.out.println(name + ": " + i);
                Thread.sleep(200); synchronized(this) {
                    while(suspendFlag) {
                        wait();
                    }
                }
            }
        } catch (InterruptedException e) { System.out.println(name +
            " interrupted.");
        }
        System.out.println(name + " exiting.");
    }

    synchronized void mysuspend() {
        suspendFlag = true;
    }

    synchronized void myresume() {
        suspendFlag = false; notify();
    }
}

class SuspendResume {
    public static void main(String[] args) { NewThread
        ob1 = new NewThread("One"); NewThread ob2
        = new NewThread("Two");

        ob1.t.start(); // Start the thread ob2.t.start(); // Start
        the thread

        try {
            Thread.sleep(1000);
            ob1.mysuspend();
            System.out.println("Suspending thread One"); Thread.sleep(1000);
            ob1.myresume();
            System.out.println("Resuming thread One");
            ob2.mysuspend();
            System.out.println("Suspending thread Two"); Thread.sleep(1000);
```

```
            ob2.myresume();
            System.out.println("Resuming thread Two");
        } catch (InterruptedException e) {
            System.out.println("Main thread Interrupted");
        }

        // wait for threads to finish try {
            System.out.println("Waiting for threads to finish."); ob1.t.join();
            ob2.t.join();
        } catch (InterruptedException e) {
            System.out.println("Main thread Interrupted");
        }

        System.out.println("Main thread exiting.");
    }
}
```

When you run the program, you will see the threads suspend and resume. Later in this book, you will see more examples that use the modern mechanism of thread control. Although this mechanism may not appear as simple to use as the old way, nevertheless, it is the way required to ensure that run-time errors don't occur. It is the approach that *must* be used for all new code.

## Obtaining a Thread's State

We can obtain the current state of a thread by calling the **getState( )** method defined by **Thread**. It is shown here:

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

It returns a value of type **Thread.State** that indicates the state of the thread at the time at which the call was made. **State** is an enumeration defined by **Thread**. (An enumeration is a list of named constants. It is discussed in detail in Chapter 12.) Here are the values that can be returned by **getState( )**:

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

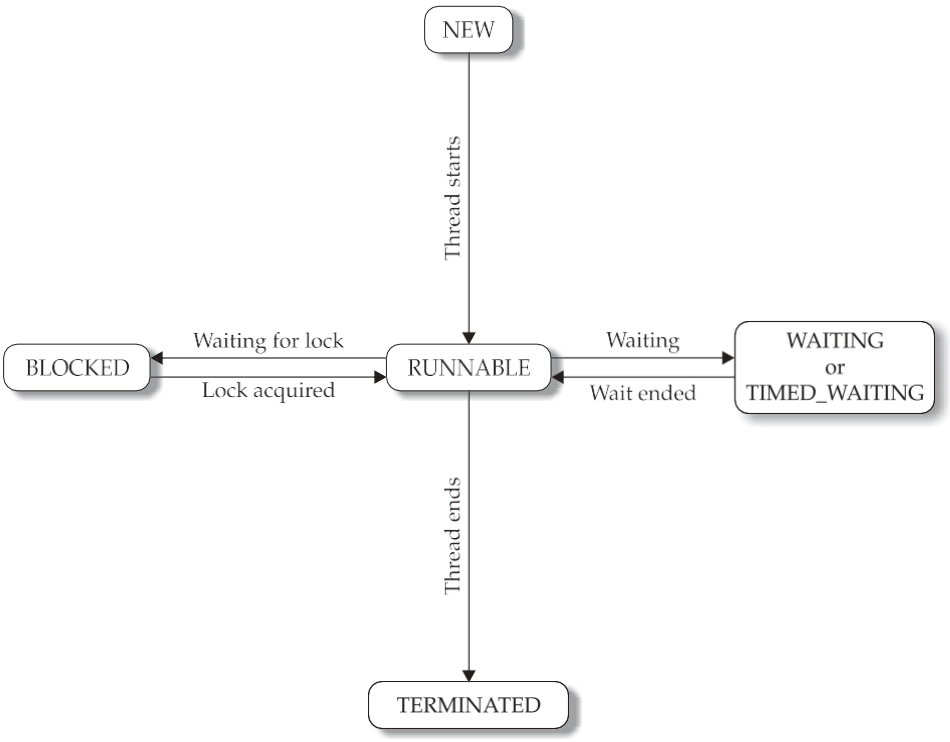



**Figure 11-1** Thread states

Figure 11-1 diagrams how the various thread states relate.

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

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

Thread.State.RUNNABLE) // ...
```

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

# Module -5

## Chapter -2: Enumerations, Autoboxing, and Annotations

## Enumerations

- Enumerations in Java provide a structured way to define a new data type with named constants representing legal values.
- They offer a more robust alternative to using final variables for defining constant values. Enumerations are commonly used to represent sets of related items, such as error codes or states of a device.
- In Java, enumerations are implemented as classes, allowing for constructors, methods, and instance variables, which greatly enhances their capabilities and flexibility.
- They are extensively used throughout the Java API library due to their power and versatility.

### Enumeration Fundamentals

1. **Definition**:
   - Enumerations are created using the `enum` keyword.
   - Constants within the enumeration are called enumeration constants.
2. **Constants Declaration**:
   - Enumeration constants are implicitly declared as public, static, final members of the enumeration type.
   - Each constant is of the type of the enumeration in which it is declared.
3. **Instantiation**:

- Enumerations define a class type, but they are not instantiated using the `new` keyword.
- Enumeration variables are declared and used similarly to primitive types.

4. **Assignment and Comparison**:
    - Enumeration variables can only hold values defined by the enumeration.
    - Constants can be assigned to enumeration variables using the dot notation (`EnumType.Constant`).
    - Constants can be compared for equality using the `==` relational operator.
5. **Switch Statements**:
    - Enumeration values can be used to control switch statements.
    - All case statements within the switch must use constants from the same enum as the switch expression.
    - Constants in case statements are referenced without qualification by their enumeration type name.
6. **Displaying Enumeration Constants**:
    - Enumeration constants are displayed by outputting their names.
    - Enumeration constants are referenced using the dot notation (`EnumType.Constant`)

The following program puts together all of the pieces and demonstrates the **Apple** enumeration:

```
// An enumeration of apple varieties. enum Apple {
    Jonathan, GoldenDel, RedDel, Winesap, Cortland
}

class EnumDemo {
    public static void main(String[] args)
    {
        Apple ap;

        ap = Apple.RedDel;

        // Output an enum value. System.out.println("Value of ap: " + ap);
        System.out.println();

        ap = Apple.GoldenDel;

        // Compare two enum values. if(ap ==
        Apple.GoldenDel)
            System.out.println("ap contains GoldenDel.\n");

        // Use an enum to control a switch statement. switch(ap) {
            case Jonathan: System.out.println("Jonathan is red."); break;
            case GoldenDel:
                System.out.println("Golden Delicious is yellow."); break;
            case RedDel:
                System.out.println("Red Delicious is red."); break;
            case Winesap: System.out.println("Winesap is red."); break;
            case Cortland: System.out.println("Cortland is red."); break;
        }
    }
}
```

The output from the program is shown here:

```
Value of ap: RedDel

ap contains GoldenDel. Golden

Delicious is yellow.
```

## The values( ) and valueOf( ) Methods

- All enumerations automatically contain two predefined methods: **values( )** and **valueOf( )**.
- Their general forms are shown here:

  public static *enum-type* [ ] values( )
  public static *enum-type* valueOf(String *str* )

- The **values( )** method returns an array that contains a list of the enumeration constants.
- The **valueOf( )** method returns the enumeration constant whose value corresponds to the string passed in *str*. In both cases, *enum-type* is the type of the enumeration.
- For example, in the case of the **Apple** enumeration shown earlier, the return type of **Apple.valueOf("Winesap")** is **Winesap**.

The following program demonstrates the **values( )** and **valueOf( )** methods:

```
// Use the built-in enumeration methods.

// An enumeration of apple varieties. enum
Apple {
   Jonathan, GoldenDel, RedDel, Winesap, Cortland
}

class EnumDemo2 {
   public static void main(String[] args)
   {
      Apple ap;

      System.out.println("Here are all Apple constants:");

      // use values()
      Apple[] allapples = Apple.values();
      for(Apple a : allapples)
         System.out.println(a);

      System.out.println();

      // use valueOf()
      ap = Apple.valueOf("Winesap"); System.out.println("ap
      contains " + ap);
```

```
    }
}
```

The output from the program is shown here:

```
Here are all Apple constants:
Jonat
han
Golde
nDel
RedD
el
Wine
sap
Cortl
and

ap contains Winesap
```

Notice that this program uses a for-each style **for** loop to cycle through the array of constants obtained by calling **values( )**. For the sake of illustration, the variable **allapples** was created and assigned a reference to the enumeration array. However, this step is not necessary because the **for** could have been written as shown here, eliminating the need for the **allapples** variable:

```
for(Apple a : Apple.values())
  System.out.println(a);
```

Now, notice how the value corresponding to the name **Winesap** was obtained by calling valueOf( ).

```
ap = Apple.valueOf("Winesap");
```

## Java Enumerations Are Class Types

1. **Enum as Class Type**:
   - Java enumerations are treated as class types.
   - They have similar capabilities as other classes, allowing constructors, instance variables, methods, and interface implementations.
2. **Enumeration Constants**:
   - Each enumeration constant is an object of its enumeration type.

- Constructors can be defined for enums, and they are called when each enumeration constant is created.
- Instance variables defined within the enum are associated with each enumeration constant separately.

3. **Example with Apple Enum**:
   - An example is provided with an `Apple` enum representing different varieties of apples.
   - Each constant has an associated price stored in an instance variable.
   - A constructor `Apple(int p)` initializes the price for each apple variety.
   - A method `getPrice()` returns the price of the apple variety.
4. **Usage Example**:
   - In the `main()` method, the prices of different apple varieties are displayed.
   - The constructor is called for each enumeration constant to initialize the prices.
   - Instance methods can be called on enumeration constants to retrieve associated data.
5. **Overloaded Constructors**:
   - Enumerations can have two or more overloaded constructors, just like other classes.
   - An example is provided with a default constructor initializing the price to -1 when no price data is available.

```
// Use an enum constructor. enum Apple {
    Jonathan(10), GoldenDel(9), RedDel, Winesap(15), Cortland(8); private int price; // price of each

    apple

    // Constructor
    Apple(int p) { price = p; }

    // Overloaded constructor Apple() { price
    = -1; }

    int getPrice() { return price; }
}
```

## Enumerations Inherit Enum

1. **Inheritance**:
    - All Java enumerations automatically inherit from the `java.lang.Enum` class.
    - While you can't inherit a superclass when declaring an enum, `java.lang.Enum` is implicitly inherited by all enums.
2. `java.lang.Enum` **Methods**:
    - `ordinal()` **Method**:
        - Returns the ordinal value of the invoking enumeration constant.
        - Ordinal values start at zero and increase sequentially.
        - Example: `Apple.Jonathan.ordinal()` would return 0.
    - `compareTo()` **Method**:
        - Compares the ordinal value of the invoking constant with another constant of the same enumeration.
        - Returns a negative value if the invoking constant's ordinal value is less than the other constant's, zero if they're equal, and a positive value if it's greater.
        - Example: `Apple.Jonathan.compareTo(Apple.RedDel)` would return a negative value.
    - `equals()` **Method**:
        - Overrides the `equals()` method defined by `Object`.
        - Compares an enumeration constant with any other object.
        - Returns true only if both objects refer to the same constant within the same enumeration.
        - Example: `Apple.Jonathan.equals(Apple.Jonathan)` would return true.

3. **Comparing Enumeration Constants**:
   - Enumeration constants can be compared for equality using the `==` operator.
   - The `equals()` method can also be used to compare constants for equality, ensuring they belong to the same enumeration.
4. **Example Program**:
   - Demonstrates the usage of `ordinal()`, `compareTo()`, and `equals()` methods with enumeration constants.

```
// Demonstrate ordinal(), compareTo(), and equals().

// An enumeration of apple varieties. enum Apple {
    Jonathan, GoldenDel, RedDel, Winesap, Cortland
}

class EnumDemo4 {
    public static void main(String[] args)
    {
        Apple ap, ap2, ap3;

        // Obtain all ordinal values using ordinal(). System.out.println("Here are all apple constants" +
                                            " and their ordinal values: "); for(Apple a : Apple.values())
            System.out.println(a + " " + a.ordinal());

        ap = Apple.RedDel; ap2 =
        Apple.GoldenDel; ap3 =
        Apple.RedDel;

        System.out.println();

        // Demonstrate compareTo() and equals() if(ap.compareTo(ap2) < 0)
            System.out.println(ap + " comes before " + ap2);

        if(ap.compareTo(ap2) > 0)
            System.out.println(ap2 + " comes before " + ap);

        if(ap.compareTo(ap3) == 0) System.out.println(ap + " equals " +
            ap3);

        System.out.println(); if(ap.equals(ap2))
            System.out.println("Error!");

        if(ap.equals(ap3))
            System.out.println(ap + " equals " + ap3);
```

```
        if(ap == ap3)
            System.out.println(ap + " == " + ap3);

    }
}
```

The output from the program is shown here:

```
Here are all apple constants and their ordinal values: Jonathan 0
GoldenDel 1
RedDel 2
Winesap 3
Cortland 4

GoldenDel comes before RedDel RedDel
equals RedDel

RedDel equals RedDel
RedDel == RedDel
```

## Another Enumeration Example

- An automated "decision maker" program was created. In that version, variables called **NO**, **YES**, **MAYBE**, **LATER**, **SOON**, and **NEVER** were declared within an interface and used to represent the possible answers. While there is nothing technically wrong with that approach,
- the enumeration is a better choice. Here is an improved version of that program that uses an **enum** called **Answers** to define the answers.

```
// An improved version of the "Decision Maker"
// program from Chapter 9. This version uses an
// enum, rather than interface variables, to
// represent the answers. import

java.util.Random;

// An enumeration of the possible answers. enum Answers {
    NO, YES, MAYBE, LATER, SOON, NEVER
}

class Question {
    Random rand = new Random(); Answers ask() {
    int prob = (int) (100 * rand.nextDouble());

        if (prob < 15)
            return Answers.MAYBE; // 15% else if
        (prob < 30)
            return Answers.NO;          //  15%
        else if (prob < 60)
            return Answers.YES;         // 30%
```

```
        else if (prob < 75)
            return Answers.LATER; // 15% else if
        (prob < 98)
            return Answers.SOON; // 13% else
            return Answers.NEVER; // 2%
    }
}

class AskMe {
    static void answer(Answers result) { switch(result) {
            case NO: System.out.println("No"); break;
            case YES: System.out.println("Yes"); break;
            case MAYBE: System.out.println("Maybe");
                break;
            case LATER: System.out.println("Later"); break;
            case SOON: System.out.println("Soon"); break;
            case NEVER: System.out.println("Never"); break;
        }
    }

    public static void main(String[] args) { Question q = new
        Question(); answer(q.ask());
        answer(q.ask());
        answer(q.ask());
        answer(q.ask());
    }
}
```

## Type Wrappers

- Java provides type wrapper classes that encapsulate primitive types within objects.
- Type wrapper classes include **`Double`**, **`Float`**, **`Long`**, **`Integer`**, **`Short`**, **`Byte`**, **`Character`**, and **`Boolean`**.
- These classes offer methods that allow integration of primitive types into Java's object hierarchy.
- Java's autoboxing feature automatically converts primitive types to their corresponding wrapper classes when necessary, and vice versa.
- This simplifies the process of working with both primitive types and objects, as conversions are handled implicitly by the compiler.

Overall, type wrappers in Java allow primitive types to be used in situations where objects are required, providing a bridge between the world of primitive types and the object-oriented nature of Java. Autoboxing further simplifies the interaction between primitive types and objects by handling conversions automatically.

# Character

- **Character** is a wrapper around a **char**. The constructor for **Character** is Character(char *ch*)
- Here, *ch* specifies the character that will be wrapped by the **Character** object being created.
- However, beginning with JDK 9, the **Character** constructor was deprecated, and beginning with JDK 16, it has been deprecated for removal. Today, it is strongly recommended that you use the static method **valueOf( )** to obtain a **Character** object.
- It is shown here:
    - static Character valueOf(char *ch*)
- It returns a **Character** object that wraps *ch*.
- To obtain the **char** value contained in a **Character** object, call **charValue( )**, shown here: char charValue( )
- It returns the encapsulated character.

# Boolean

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

Boolean(boolean *boolValue*)
Boolean(String *boolString*)

- In the first version, *boolValue* must be either **true** or **false**. In the second version, if *boolString* contains the string "true" (in uppercase or lowercase), then the new **Boolean** object will be true. Otherwise, it will be false.
- However, beginning with JDK 9, the **Boolean** constructors were deprecated, and beginning with JDK 16, they have been deprecated for removal. Today, it is strongly recommended that you use the static method **valueOf( )** to obtain a **Boolean** object. It has the two versions shown here:

static Boolean valueOf(boolean *boolValue*)
static Boolean valueOf(String *boolString*)

- Each returns a **Boolean** object that wraps the indicated value.
- To obtain a **boolean** value from a **Boolean** object, use **booleanValue( )**, shown here: boolean booleanValue( )
- It returns the **boolean** equivalent of the invoking object.

## The Numeric Type Wrappers

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

  These methods are shown here:

```
byte    byteValue(    )
double doubleValue( )
float floatValue( )
int     intValue(    )
long longValue( )
short shortValue( )
```

- For example, **doubleValue( )** returns the value of an object as a **double**, **floatValue( )** returns the value as a **float**, and so on.
- These methods are implemented by each of the numeric type wrappers.
- All of the numeric type wrappers define constructors that allow an object to be constructed from a given value, or a string representation of that value.
- For example, here are the constructors defined for **Integer**:

```
Integer(int              num)
Integer(String str)
```

- If *str* does not contain a valid numeric value, then a **NumberFormatException** is thrown.
- Here are two of the forms supported by **Integer**:

```
static Integer valueOf(int val)
static Integer valueOf(String valStr) throws NumberFormatException
```

Here, *val* specifies an integer value and *valStr* specifies a string that represents a properly formatted numeric value in string form. Each returns an **Integer** object that wraps the specified value. Here is an example:

```
Integer iOb = Integer.valueOf(100);
```

After this statement executes, the value 100 is represented by an **Integer** instance. Thus, **iOb** wraps the value 100 within an object. In addition to the forms **valueOf( )** just shown, the integer wrappers, **Byte**, **Short**, **Integer**, and **Long**, also supply a form that lets you specify a radix.

All of the type wrappers override **toString( )**. It returns the human-readable form of the value contained within the wrapper. This allows you to output the value by passing a type wrapper object to **println( )**, for example, without having to convert it into its primitive type.

The following program demonstrates how to use a numeric type wrapper to encapsulate a value and then extract that value.

```
// Demonstrate a type wrapper.

class Wrap {
   public static void main(String[] args) { Integer iOb =

      Integer.valueOf(100); int i = iOb.intValue();

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

This program wraps the integer value 100 inside an **Integer** object called **iOb**. The program then obtains this value by calling **intValue( )** and stores the result in **i**.
The process of encapsulating a value within an object is called *boxing*. Thus, in the program, this line boxes the value 100 into an **Integer**:

```
Integer iOb = Integer.valueOf(100);
```

The process of extracting a value from a type wrapper is called *unboxing*. For example, the program unboxes the value in **iOb** with this statement:

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

The same general procedure used by the preceding program to box and unbox values has been available for use since the original version of Java. However, today, Java provides a more streamlined approach, which is described next.

# Autoboxing

- Modern versions of Java have included two important features: *autoboxing* and *auto-unboxing*.
- Autoboxing is the process by which a primitive type is automatically encapsulated (boxed) into its equivalent type wrapper whenever an object of that type is needed.
- There is no need to explicitly construct an object. Auto-unboxing is the process by which the value of a boxed object is automatically extracted (unboxed) from a type wrapper when its value is needed.
- There is no need to call a method such as **intValue( )** or **doubleValue( )**.
- Autoboxing and auto-unboxing greatly streamline the coding of several algorithms, removing the tedium of manually boxing and unboxing values.
- They also help prevent errors. Moreover, they are very important to generics, which operate only on objects. Finally, autoboxing makes working with the Collections Framework
- With autoboxing, it is not necessary to manually construct an object in order to wrap a primitive type. You need only assign that value to a type-wrapper reference. Java automatically constructs the object for you. For example, here is the modern way to construct an **Integer** object that has the value 100:

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

Notice that the object is not explicitly boxed. Java handles this for you, automatically.

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

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

Java handles the details for you.

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

```
// Demonstrateautoboxing/unboxing. class AutoBox {
    public static void main(String[] args) { Integer iOb = 100; //

        autobox an int int i = iOb; // auto-unbox

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

## Autoboxing and Methods

In addition to the simple case of assignments, autoboxing automatically occurs whenever a primitive type must be converted into an object; auto-unboxing takes place whenever an object must be converted into a primitive type. Thus, autoboxing/unboxing might occur when an argument is passed to a method, or when a value is returned by a method. For example, consider this:

```
// Autoboxing/unboxing takes place with
// method parameters and return values.

class AutoBox2 {
    // Take an Integer parameter and return
    // an int value;
    static int m(Integer v) {
        return v ; // auto-unbox to int
    }

    public static void main(String[] args) {
        // Pass an int to m() and assign the return value
        // to an Integer.  Here, the argument 100 is autoboxed
        // into an Integer.  The return value is also autoboxed
        // into an Integer.
        Integer iOb = m(100);

        System.out.println(iOb);
    }
}
```

This program displays the following result:

```
100
```

# Autoboxing/Unboxing Occurs in Expressions

In general, autoboxing and unboxing take place whenever a conversion into an object or from an object is required. This applies to expressions. Within an expression, a numeric object is automatically unboxed. The outcome of the expression is reboxed, if necessary. For example, consider the following program:

```
// Autoboxing/unboxing occurs inside expressions.

class AutoBox3 {
    public static void main(String[] args) {

        Integer iOb, iOb2; int i;

        iOb = 100;
        System.out.println("Original value of iOb: " + iOb);

        // The following automatically unboxes iOb,
        // performs the increment, and then reboxes
        // the result back into iOb.
        ++iOb;
        System.out.println("After ++iOb: " + iOb);

        // Here, iOb is unboxed, the expression is
        // evaluated, and the result is reboxed and
        // assigned to iOb2. iOb2 = iOb +
        (iOb / 3);
        System.out.println("iOb2 after expression: " + iOb2);

        // The same expression is evaluated, but the
        // result is not reboxed. i = iOb + (iOb
        / 3);
        System.out.println("i after expression: " + i);

    }
}
```

The output is shown here:

```
Original value of iOb: 100 After
++iOb: 101
iOb2 after expression: 134 i after
expression: 134
```

In the program, pay special attention to this line:

```
++iOb;
```

This causes the value in **iOb** to be incremented. It works like this: **iOb** is unboxed, the value is incremented, and the result is reboxed.

Auto-unboxing also allows you to mix different types of numeric objects in an expression. Once the values are unboxed, the standard type promotions and conversions are applied. For example, the following program is perfectly valid:

```
class AutoBox4 {
   public static void main(String[] args) {

      Integer iOb = 100; Double dOb
      = 98.6;

      dOb = dOb + iOb;
      System.out.println("dOb after expression: " + dOb);
   }
}
```

The output is shown here:

```
dOb after expression: 198.6
```

As you can see, both the **Double** object **dOb** and the **Integer** object **iOb** participated in the addition, and the result was reboxed and stored in **dOb**.

Because of auto-unboxing, you can use **Integer** numeric objects to control a **switch** statement. For example, consider this fragment:

```
Integer iOb = 2; switch(iOb) {
   case 1: System.out.println("one"); break;
   case 2: System.out.println("two"); break;
   default: System.out.println("error");
}
```

When the **switch** expression is evaluated, **iOb** is unboxed and its **int** value is obtained.

As the examples in the programs show, because of autoboxing/unboxing, using numeric objects in an expression is both intuitive and easy. In the early days of Java, such code would have involved casts and calls to methods such as **intValue( )**.

## Autoboxing/Unboxing Boolean and Character Values

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

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

class AutoBox5 {
    public static void main(String[] args) {

        // Autobox/unbox a boolean. Boolean b =
        true;

        // Below, b is auto-unboxed when used in
        // a conditional expression, such as an if. if(b)
        System.out.println("b is true");

        // Autobox/unbox a char. Character ch = 'x'; //
        box a char char ch2 = ch; // unbox a char

        System.out.println("ch2 is " + ch2);
    }
}
```

The output is shown here:

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

## Autoboxing/Unboxing Helps Prevent Errors

In addition to the convenience that it offers, autoboxing/unboxing can also help prevent errors. For example, consider the following program:

```
// An error produced by manual unboxing. class
UnboxingError {
    public static void main(String[] args) {

        Integer iOb = 1000; // autobox the value 1000

        int i = iOb.byteValue(); // manually unbox as byte !!!

        System.out.println(i);  // does not display 1000 !
    }
}
```