Chapter: 8

Threads, Enumerations & Autoboxing

Multithreaded programming basics, Enumerations, Autoboxing, static import & Annotations

8.1 Multithreading Introduction

- ***Thread:*** Java contains **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***. Thus, ***multithreading*** is a specialized form of ***multitasking***.
- ***Process:*** A **process** is, in essence, a ***program that is executing***.
- There are two distinct types of **multitasking**:
  - ***Process-based:*** *Process-based multitasking* allows a computer to run two or more ***programs*** concurrently. In process based multitasking, a ***program*** is the smallest unit of code that can be ***dispatched*** by the ***scheduler***. (Eg: run the **Java compiler** at the same time **browsing the Internet**.)
  - ***Thread-based:*** 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** at once. (Eg: a text editor can be **formatting text** at the same time that it is **printing**, as long as these two actions are being performed by two separate threads.)

[**Multithreading helps to utilize idle time:** Most I/O devices: **network ports**, **disk drives**, or the **keyboard**, are much slower than the ***CPU***. Thus, a program will often spend a majority of its ***execution time*** waiting to send or receive information to or from a device. By ***multithreading***, a program can execute another task during this ***idle time***. Eg: while one part of your program is sending a file over the Internet, another part can be reading keyboard input, and still another can be buffering the next block of data to send.]

- Java's ***multithreading features*** work in both *single-processor-systems* and *multiprocessor/multicore systems*.
  - In a **single-core system**, concurrently executing threads share the CPU, with each thread receiving a slice of CPU time. Therefore, in a single-core system, two or more threads do not actually run at the same time, but idle CPU time is utilized.
  - In **multiprocessor/multicore systems**, it is possible for two or more threads to actually execute simultaneously. In many cases, this can further improve program efficiency and increase the speed of certain operations.
- ***States of a THREAD:*** A thread can be in one of several states.
  - It can be ***running***.
  - It can be ***ready to run*** as soon as it gets CPU time.
  - A thread can be ***blocked*** when ***waiting*** for a resource.
  - A running thread can be ***suspended*** (which is a temporary halt to its execution)
  - It can later be ***resumed***.
  - A thread can be ***terminated*** (in which case its execution ends and cannot be resumed)
- ***SYNCHRONIZATION:*** ***synchronization*** allows the ***execution of threads*** to be *coordinated in certain well-defined ways*. Java has a complete subsystem devoted to **synchronization**, and its key features are also described here.

8.2 Thread Class and Runnable Interface

Java's *multithreading system* is built upon the class ***Thread*** and its companion interface, ***Runnable***. Both are packaged in ***java.lang***.

- ***Thread*** class encapsulates a **thread of execution**. To create a **new thread**, your program will either extend ***Thread*** or implement the ***Runnable*** interface. The ***Thread*** class defines several methods that help manage threads. Commonly used are:

Method	*Meaning*	*Method*	*Meaning*
`final String getName()`	Obtains a thread's **name**.	`void start()`	**Starts** a thread by calling its `run()` method.
`final int getPriority()`	Obtains a thread's **priority**.	`final boolean isAlive()`	**Determines** whether a thread is **still running**.
`void run()`	**Entry point** for the thread.	`static void sleep( long milliseconds)`	**Suspends** a thread for a specified period of **milliseconds**.
`final void join()`	**Waits** for a thread to **terminate**.		

- ***MAIN THREAD:*** All processes have ***at least one thread of execution***, which is usually called the ***main thread***, because it is the one that is **executed when your program begins**. Thus, the ***main thread*** is the thread that all of the preceding example programs in the book have been using. From the ***main thread***, you can create other ***threads***.

8.3 Creating a Thread_1 (by implementing the *Runnable* interface)

You **create a thread** by ***instantiating*** an object of type ***Thread***. The ***Thread*** class *encapsulates an object* that is ***runnable***. As mentioned, Java defines ***two ways*** in which you can create a ***runnable*** **object**:

- **implement** the ***Runnable*** interface.
- **extend** the ***Thread*** class

NOTE: Both approaches still use the ***Thread*** class to **instantiate**, **access**, and **control** the *thread*. The only difference is how a *thread-enabled class is created*.

- The ***Runnable*** interface **abstracts** a unit of executable code. You can construct a ***thread on any object*** that implements the ***Runnable*** interface.
  - ***Runnable*** defines only one method called ***run()***, which is declared like this: `public void run()`
    - Inside ***run()***, you will define the ***code that constitutes*** the ***new thread***.
    - ***run()*** can call other ***methods***, use other ***classes***, and ***declare variables*** just like the ***main thread***.
    - ***run()*** establishes the ***entry point*** for *another, concurrent thread of execution* within your program. This thread will ***end*** when ***run() returns***.
- After creating a **class** that implements ***Runnable***, you will ***instantiate*** an object of type ***Thread*** on an object of that class. ***Thread*** defines several *constructors*. The one that we will use first is: `Thread(Runnable threadOb)`
  - ***threadOb*** is an **instance** of a class that implements the ***Runnable*** interface. It defines where execution of the thread'll begin.

❒ Once created, the ***new thread*** will *not start running until* you call its ***start()*** method, which is declared within ***Thread***.

☞ ***start()*** executes a call to ***run()***. The ***start()*** method is: ***void start()***

Example 1: Here is an example that ***creates a new thread*** and starts it running:

```
/* Objects of MyThread can be run in their own threads
   because MyThread implements Runnable. */

class MyThread implements Runnable {
    String thrdName;
    MyThread(String name) { thrdName = name; }

// Entry point of thread.
    public void run() {       //Threads start executing here.
        System.out.println(thrdName + " starting.");
            try {  for(int j=0; j < 10; j++) {
                    Thread.sleep(400);       //Suspends for 400 milisec
                    System.out.println("In " + thrdName + ", j is " + j); } }
            catch(InterruptedException exc) {
                    System.out.println(thrdName + " interrupted."); }

        System.out.println(thrdName + " terminating.");          }
}
```

```
class UseThreads {
public static void main(String args[]) {
    System.out.println("Main thread starting.");

    // First, construct a MyThread object. A runnable object.
    MyThread mt = new MyThread("Child #1");

    // Next, construct a thread from that object. A thread on that object
    Thread newThrd = new Thread(mt);

    // Finally, start execution of the thread.
    newThrd.start();                    //Start running the thread
      for(int i=0; i<50; i++) {  System.out.print(".");
        try {Thread.sleep(100);}    //Suspends for 100 milisec
        catch(InterruptedException exc) {
            System.out.println("Main thread interrupted.");}
        }
    System.out.println("Main thread ending.");          }}
```

☛ First, ***MyThread*** implements ***Runnable***. This means that an object of **type *MyThread*** is *suitable for use as a thread* and can be passed to the ***Thread*** *constructor.*

☛ Inside ***run()***, a loop is established that counts from ***0*** to ***9***. Notice the call to ***sleep()***.In ***run()***, ***sleep()*** pauses the thread for ***400 milliseconds*** each time through the loop. This lets the thread run slow enough for you to watch it execute. Another reason to delay is that run ***main()*** thread and ***mt*** concurrently (discussed later).

☛ Inside ***main()***, a new ***Thread*** object is created by the following sequence of statements:

MyThread `mt = new MyThread("Child #1");`	**[1]** First an object of ***MyThread*** is **constructed**.
Thread newThrd = new `Thread(mt);`	**[2]** This object is then used to **construct** a Thread object. This is possible because ***MyThread*** implements ***Runnable***.
`newThrd.start();`	**[3]** Finally, execution of the new thread is started by calling ***start()***. This causes the child thread's ***run()*** method to begin.

▶ After calling ***start()***, execution returns to ***main()***, and it enters ***main()***'s ***for*** loop. Notice that this loop iterates ***50*** times, pausing ***100 milliseconds*** each time through the loop.

☛ Both threads continue running, sharing the CPU in **single-CPU systems**, until their loops finish.

❒ **Role of *timing difference* in *multithreading* in single-CPU systems:** To illustrate the fact that the ***main thread*** and ***mt*** execute ***concurrently***, it is necessary to ***keep main() from terminating until mt is finished***.

*[Here, this is done through the timing differences between the two threads. Because the calls to **sleep( )** inside **main( )**'s for loop cause a total delay of **5 seconds** (**50** iterations times **100 milliseconds**), but the total delay within **run( )**'s loop is only **4 seconds** (**10** iterations times **400 milliseconds**), **run( )** will finish approximately **1** second before **main( )**. As a result, both the **main** thread and **mt** will execute concurrently until **mt** ends. Then, about 1 second later **main( )** ends. However, Java provides better ways for one thread to wait until another completes, discussed later.]*

NOTE

[1] As a general rule, a program continues to run until all of its threads have ended. The **main thread** is a convenient place to perform the **orderly shutdown** of a program, Eg: **closing of files**. It also provides a well-defined exit point for programs. So, it often makes sense for **main thread** to finish **last**. Fortunately, it is trivially easy for the **main thread** to wait until the **child threads** have completed.

[2] ***sleep( )* method :** The ***sleep()*** method causes the thread from which it is called to suspend execution for the specified period of ***milliseconds***. Its general form:

static void sleep(long milliseconds) throws InterruptedException

The number of milliseconds to suspend is specified in ***milliseconds***.
This method can throw an ***InterruptedException***. Thus, calls to it must be wrapped in a ***try block***.
The ***sleep()*** method also has a second form, which allows you to specify the period in terms of ***milliseconds*** and ***nanoseconds*** if you need that level of precision.

8.3.1 Fast execution of Thread and Giving name of the Thread

❒ It is possible to have a thread ***begin execution as soon as it is created***. In the case of ***MyThread*** (in Example 1), this is done by instantiating a ***Thread object*** inside ***MyThread***'s constructor.

❒ ***Naming a Thread:*** There is no need for ***MyThread*** to store the name of the thread since it is possible to give a name to a thread when it is created. To do so, use this version of ***Thread***'s constructor:

Thread(Runnable threadOb, String name)

▶ Here, ***name*** becomes the *name of the thread.*

☞ ***Obtain the name of a thread:*** You can obtain the name of a thread by calling ***getName()*** defined by ***Thread***. Its general form:

final String getName( )

☞ ***Setting the name of a thread after its creation:*** You can set the name of a thread after it is created by using setName( ),

final void setName(String threadName)

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

Example 2 : The improved version of the preceding ***MyThread*** (in Example 1) program given in next this reworked version named ***UseThreadsImproved***. This version produces the same output as before. Notice that the thread is stored in ***thrd*** inside ***MyThread***.

```
class MyThread implements Runnable { Thread thrd;          //A reference to the thread is stored in thrd.
/* Construct a new thread.*/    MyThread(String nam) {    thrd = new Thread(this, nam);    //The thread is named when it is created.
/* start the thread */                                    thrd.start();          }
/* Begin execution of new thread */  public void run() {System.out.println(thrd.getName() + " starting.");    //Begin executing the thread.
                                        try { for(int j=0; j<10; j++) {  Thread.sleep(400);
                                                                         System.out.println("In " + thrd.getName() +", j is " + j); } }
                                        catch(InterruptedException exc) {System.out.println(thrd.getName() + " interrupted."); }
                                    System.out.println(thrd.getName() + " terminating.");     }}

class UseThreadsImproved { public static void main(String args[]) {    System.out.println("Main thread starting.");
/* the thread starts */         MyThread mt = new MyThread("Child #1");     // giving the nam variable threads name. nam =  Child #1
/* when it is created */        for(int i=0; i < 50; i++) {    System.out.print(".");
                                                               try { Thread.sleep(100); }
                                                               catch(InterruptedException exc) {System.out.println("Main thread interrupted."); } }
                                System.out.println("Main thread ending.");      }}
```

8.4 Creating a Thread_2 (by *Extending Thread*) Recall INHERITANCE

Implementing ***Runnable*** is one way to create a class that can ***instantiate*** thread objects (In previous two examples we used Runnable). ***Extending Thread*** is the other way. We'll create a program functionally identical to the previous ***UseThreadsImproved*** program by extending ***Thread***.

- When a class ***extends Thread***, it must ***override*** the ***run()*** method, which is the ***entry point for the new thread***. It is possible to override other ***Thread*** methods, but doing so is not required.
  - The class must also call ***start()*** to begin ***execution of the new thread***.

Example 3 (Creating a thread by Extending Thread): We make changes the following things to the previous ***UseThreadsImproved***

- Change the declaration of ***MyThread*** so that it **extends** ***Thread*** rather than **implementing** ***Runnable***, as shown here:
  `class MyThread extends Thread { . . . }`
- The ***thrd*** variable in "`Thread thrd;`" is no longer needed, since ***MyThread*** includes an instance of ***Thread*** and can refer to itself.
- Change the ***MyThread*** **constructor** so that it looks like:

```
MyThread(String name) {    super(name);  // name thread
                           start(); }    // start the thread
```

  - Here ***super*** is used to call this version of ***Thread's constructor***. `Thread(String name);`
    - ***name*** is the ***name of the thread***.
- Change ***run()*** so it calls ***getName()*** directly, without qualifying it with the ***thrd*** variable.

```
class MyThread extends Thread {
/* Construct a new thread. */    MyThread(String name) { super(name);      // name thread
/* start the thread */                                   start();}
/* Begin execution of new thread.*/  public void run() { System.out.println(getName() + " starting.");
                                     try {    for(int j=0; j < 10; j++) {    Thread.sleep(400);
                                                                             System.out.println("In " + getName() + ", j is " + j); }    }
                                     catch(InterruptedException exc) { System.out.println(getName() + " interrupted."); }
                                 System.out.println(getName() + " terminating.");    } }

class ExtendThread { public static void main(String args[]) { System.out.println("Main thread starting.");
        MyThread mt = new MyThread("Child #1");
        for(int i=0; i < 50; i++) {    System.out.print(".");
                                       try { Thread.sleep(100); }
                                       catch(InterruptedException exc) { System.out.println("Main thread interrupted."); }
                                   }
        System.out.println("Main thread ending.");    }}
```

8.4.1 Extending Thread or implementing Runnable -> which is better?

- 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***.
- Some Java programmers feel that ***classes should be extended only when they are being enhanced or modified*** in some way. So, if you ***will not be overriding any of Thread's other methods***, it is probably best to simply **implement** ***Runnable***.
- Also, by **implementing** ***Runnable***, you enable your thread to *inherit a class* other than ***Thread***.

8.4.2 Creating Multiple Threads

A program can spawn as many threads as it needs. Following program creates three child threads:

Example 4: `class MyThread implements Runnable { /* Same as previous Example 2 */ }`

```
class MoreThreads {public static void main(String args[]) {
        System.out.println("Main thread starting.");
        MyThread mt1 = new MyThread("Child #1");
        MyThread mt2 = new MyThread("Child #2");
        MyThread mt3 = new MyThread("Child #3");
for(int i=0; i < 50; i++) {
        System.out.print(".");
        try { Thread.sleep(100); }
        catch(InterruptedException exc) {
                System.out.println("Main thread interrupted."); }
        }
System.out.println("Main thread ending.");    }}
```

- All three ***child threads*** will share the ***CPU***.
- The threads will be ***started in the order in which they are created.*** [However, this may not always be the case. Java is ***free to schedule*** the ***execution*** of threads in its own way. Of course, because of ***differences in timing*** or ***environment***, the precise output from the program may differ.]

8.4.3 `isAlive()` to determine when a THREAD ENDS and `join()` to control WAITING TIME

To keep the **main thread** alive until the other **threads** ended. In previous examples, this was accomplished by having the **main thread** *sleep longer than* the **child threads** that it spawned. ***Thread*** provides two methods to determine if a thread has ended.

❒ ***isAlive():*** You can call ***isAlive()*** on the thread. Its general form is: ***final boolean isAlive()***

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

Example 5: To try ***isAlive( )***, substitute this version of ***MoreThreads*** [3-threads] for the one shown in the ***Example 4*** :

```
class MoreThreads {
public static void main(String args[]) {
    System.out.println("Main thread starting.");
    MyThread mt1 = new MyThread("Child #1");
    MyThread mt2 = new MyThread("Child #2");
    MyThread mt3 = new MyThread("Child #3");
do{ System.out.print(".");
    try { Thread.sleep(100); }
    catch(InterruptedException exc){System.out.println("Main thread interrupted."); }
} while (mt1.thrd.isAlive() || mt2.thrd.isAlive() || mt3.thrd.isAlive());
System.out.println("Main thread ending.");    }}
                                        //This waits until all threads terminate.
```

- ***main()*** ends as soon as the other threads finish. Inside ***while*** condition ***isAlive()*** used to wait for the ***child threads*** to end.

❒ ***join():*** Another way to ***wait for a thread to finish*** is to call ***join()*** :

final void join( ) throws InterruptedException

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

Example 6: Here is a program that uses join() to ensure that the main thread is the last to stop:

```
class JoinThreads {
public static void main(String args[]) {
System.out.println("Main thread starting.");
MyThread mt1 = new MyThread("Child #1");
MyThread mt2 = new MyThread("Child #2");
MyThread mt3 = new MyThread("Child #3");
try { mt1.thrd.join(); System.out.println("Child #1 joined.");
      mt2.thrd.join(); System.out.println("Child #2 joined.");
      mt3.thrd.join(); System.out.println("Child #3 joined.");}
catch(InterruptedException exc) {
        System.out.println("Main thread interrupted.");}
System.out.println("Main thread ending.");    }}
```

```
..... In Child #3, count is 9
o     Child #3 terminating.
u     In Child #2, count is 9
t     Child #2 terminating.
p     In Child #1, count is 9
u     Child #1 terminating.
t     Child #1 joined.
      Child #2 joined.
      Child #3 joined.
      Main thread ending.
```

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

8.5 Priorities of Threads

A thread's priority determines, in part, how much CPU time a thread receives relative to the other active threads. In general, over a given period of time, low-priority threads receive little. High-priority threads receive a lot.

❒ When a ***child thread*** is started, its *priority setting is equal* to that of its ***parent thread***. You can change a thread's priority by calling ***setPriority()***, which is a member of ***Thread***. This is its general form:

final void setPriority(int level)

- Here, ***level*** specifies the new priority setting for the calling thread.
- ***MIN_PRIORITY, MAX_PRIORITY and NORM_PRIORITY:*** The value of level must be within the range ***MIN_PRIORITY*** and ***MAX_PRIORITY***. Currently, these values are 1 and 10, respectively. To return a thread to **default priority**, specify ***NORM_PRIORITY***, which is currently 5. These ***priorities*** are defined as ***static final*** variables within ***Thread***.
- You can obtain the **current priority setting** by calling the ***getPriority()*** method of ***Thread***, shown here:

final int getPriority()

Example 7: The following code demonstrates two threads at different priorities. The threads are created as instances of Priority.

```
class Priority implements Runnable {  int count;       Thread thrd;       static boolean stop = false;     static String currentName;
/* Construct a new thread. Notice that this constructor does not actually start the threads running. No star() method called in here */
        Priority(String name) {      thrd = new Thread(this, name);       count = 0;          currentName = name; }
// Begin execution of new thread.
        public void run() { System.out.println(thrd.getName() + " starting.");
                do { count++;
                        if( currentName.compareTo(thrd.getName() ) != 0) {   currentName = thrd.getName();
                                                                             System.out.println("In " + currentName);   }
                    } while(stop == false && count < 10000000);
                stop = true;
                System.out.println("\n" + thrd.getName() + " terminating."); } }

class PriorityDemo { public static void main(String args[]) {    Priority mt1 = new Priority("High Priority");
                                                                  Priority mt2 = new Priority("Low Priority");
mt1.thrd.setPriority(Thread.NORM_PRIORITY+2);     // set the priorities. Higher than mt2
mt2.thrd.setPriority(Thread.NORM_PRIORITY-2);     // set the priorities
        mt2.thrd.start();          // start the thread mt2
        mt1.thrd.start();          // start the thread mt1
try {   mt1.thrd.join();   mt2.thrd.join();   }        //to make main thread end last
catch(InterruptedException exc) { System.out.println("Main thread interrupted."); }
        System.out.println("\nHigh priority thread counted to " + mt1.count);
        System.out.println("Low priority thread counted to " + mt2.count);        }}
```

OUTPUT:

```
High Priority starting.
In High Priority
Low Priority starting.
In Low Priority
In High Priority
High Priority terminating.
Low Priority terminating.
High priority thread counted to 10000000
Low priority thread counted to 8183
```

- ☛ The ***run()*** method contains a loop that counts the number of iterations. The loop stops when either the ***count*** reaches ***10,000,000*** or the **static variable** ***stop*** is ***true***.
  - Initially, ***stop*** is set to ***false***, but the first thread to finish counting sets ***stop*** to ***true***. This causes the second thread to terminate with its next time slice.
  - Each time through the loop the string in ***currentName*** is checked against the name of the executing thread. If they don't match, it means that a **task-switch** occurred. Each time a **task-switch** happens, the name of the new thread is displayed, and ***currentName*** is given the name of the new thread.
  - Displaying **each thread switch** allows you to watch (in a very imprecise way) when the threads gain access to the CPU.
  - After both threads stop, the number of iterations for each loop is displayed.
- ☛ In this run, the ***high-priority*** thread got a ***vast majority*** of the ***CPU time***. (May vary on System use)

NOTE:

[1] ***String compareTo()*** recall ***Java/C# 2.14***, ***compareTo()*** method is used for **comparing two string lexicographically**. Each character of both the strings is converted into a **Unicode value** for comparison.

- ☝ If both the ***strings are equal*** then this method returns ***0***.
- ☝ It returns ***positive*** if the *first string* is ***lexicographically greater*** than the *second string* else the result would be ***negative***.

[2] Just giving one thread a **high priority** and another **low priority** ***does not necessarily mean that one thread will run faster or more often than the other***. It's just that the ***high-priority thread has greater potential access to the CPU***.

- ✋ We know, the ***CPU time*** a thread receives has profound impact on its ***execution characteristics*** and its ***interaction with other threads*** currently executing in the system. It is important to understand that **factors other than a thread's priority also affect** how much CPU time a thread receives.
  - ☹ For example, if a high-priority thread is waiting on some resource, perhaps for keyboard input, then it will be blocked, and a lower priority thread will run. However, when that high-priority thread gains access to the resource, it can preempt the low-priority thread and resume execution.
  - ☹ The most important factor affecting thread execution is the way the ***operating system implements multitasking and scheduling***. Some **OS** use **preemptive multitasking** in which each thread receives a time slice, **at least occasionally**. Other systems use ***nonpreemptive*** scheduling in which one thread must yield execution ***before another*** thread will execute. In ***nonpreemptive*** systems, it is easy for one thread to ***dominate***, *preventing others from running.*

8.6 Synchronization

To ***coordinate the activities of two or more threads*** we use the ***synchronization*** process. Most common reasons for synchronization:

- ☯ When two or more threads need access to a **shared resource** that can be used by **only one thread at a time**. For example, when one thread is writing to a file, a second thread must be prevented from doing so at the same time.
- ☯ When one thread is **waiting** for an event that is caused by another thread. In this case, there must be some means by which the first thread is held in a suspended state until the event has occurred. Then, the waiting thread must resume execution.
- ❒ ***MONITOR and LOCK:*** Key to synchronization in Java is the *concept of* the ***monitor***, which **controls access to an object**. A ***monitor*** works by implementing the *concept of* a ***lock***. When an object is ***locked by one thread***, no other thread can gain access to the object. When the ***thread*** exits, the object is ***unlocked*** and is available for use by another ***thread***.
  - ☛ All objects in Java have a ***monitor***. Thus, all objects can be ***synchronized***.
- ❒ ***Synchronization:*** **Synchronization** is supported by the keyword ***synchronized*** and a few well-defined methods that all objects have. There are two ways that you can synchronize your code. Both involve the use of the ***synchronized*** keyword.

8.6.1 Synchronized Methods

By applying ***synchronized*** keyword we can **synchronize access** to a method. When that ***method is called***, the **calling *thread*** enters the object's ***monitor***, which then ***locks*** the object. Synchronization is achieved with virtually without any programming

- ☑ While ***locked***, no other ***thread*** can enter the ***method***, or enter any other **synchronized method** defined by the object's class.
- ☑ When the ***thread returns*** from the ***method***, the ***monitor unlocks*** the object, allowing it to be used by the next ***thread***.

❒ ***Example 8:*** Below demonstrates synchronization by controlling access to ***sumArray()***, which sums the elements of an ***int*** array.

```
class SumArray { private int sum;
    synchronized int sumArray(int nums[]) {          // sumArray( ) is synchronized.
    sum = 0;      // reset sum
    for(int i=0; i<nums.length; i++) {          sum += nums[i];
        System.out.println("Running total for " + Thread.currentThread().getName() + " is " + sum);
        try { Thread.sleep(10);   }     // allow task-switch
        catch(InterruptedException exc) { System.out.println("Thread interrupted."); }
        }
  return sum;   }  }

class MyThread implements Runnable {      Thread thrd;
                                          static SumArray sa = new SumArray();
                                          int a[]; int answer;
/* Construct a new thread.*/      MyThread(String name, int nums[]) { thrd = new Thread(this, name);
                                                                      a = nums;
                                  /* start the thread */              thrd.start();   }

/* Begin execution of new thread.*/  public void run() {
                                     int sum;
                                     System.out.println(thrd.getName() + " starting.");
                                     answer = sa.sumArray(a);
                                     System.out.println("Sum for "+ thrd.getName() +" is " + answer);
                                     System.out.println(thrd.getName() + " terminating."); } }
```

OUTPUT:

```
Child #1 starting.
Running total for Child #1 is 1
Child #2 starting.
Running total for Child #1 is 3
Running total for Child #1 is 6
Running total for Child #1 is 10
Running total for Child #1 is 15
Sum for Child #1 is 15
Child #1 terminating.
Running total for Child #2 is 1
Running total for Child #2 is 3
Running total for Child #2 is 6
Running total for Child #2 is 10
Running total for Child #2 is 15
Sum for Child #2 is 15
Child #2 terminating.
```

(The precise output may differ on your computer.)

```
class Sync {  public static void main(String args[]) {   int a[] = {1, 2, 3, 4, 5};
                                                          MyThread mt1 = new MyThread("Child #1", a);
                                                          MyThread mt2 = new MyThread("Child #2", a);
                                                  try { mt1.thrd.join();        mt2.thrd.join();  }        //waits for main thread
                                                  catch(InterruptedException exc) {  System.out.println("Main thread interrupted."); }
                                                  }}
```

- The program creates ***three classes***. The first class is ***SumArray***. It contains the method ***sumArray()***, which sums an integer array.
- The second class is ***MyThread***, which uses a ***static*** object of type ***SumArray*** to obtain the sum of an integer array. This object is called ***sa*** and because it is ***static***, there is only one copy of it that is shared by all instances of ***MyThread***.
- Finally, the class ***Sync*** creates two threads and has each compute the sum of an integer array.

Synchronization effect: Inside ***sumArray()***, ***sleep()*** is called to purposely allow a ***task switch*** to occur, if one can—but it can't. Because ***sumArray()*** is ***synchronized***, it can be used by ***only one thread at a time***. Thus, when the second child ***thread*** begins execution, it does not enter ***sumArray( )*** until after the first child ***thread*** is done with it. This ensures that the correct result is produced.

If synchronized is removed from the declaration of ***sumArray()***, ***sumArray()*** will no longer be synchronized, and ***any number of threads may use it concurrently***.

The problem with this is that the running total is stored in ***sum***, which will be changed by each thread that calls ***sumArray()*** through the ***static*** object ***sa***. Thus, when two threads call ***sa.sumArray()*** at the same time, incorrect results are produced because ***sum*** reflects the summation of both ***threads***, mixed together.

OUTPUT

From the program after ***synchronized*** has been removed from ***sumArray( )***'s declaration.
(The precise output may differ on your computer.)

```
Child #1 starting.
Running total for Child #1 is 1
Child #2 starting.
Running total for Child #2 is 1
Running total for Child #1 is 3
Running total for Child #2 is 5
Running total for Child #2 is 8
Running total for Child #1 is 11
Running total for Child #2 is 15
Running total for Child #1 is 19
Running total for Child #2 is 24
Sum for Child #2 is 24
Child #2 terminating.
Running total for Child #1 is 29
Sum for Child #1 is 29
Child #1 terminating.
```

- As the output shows, both ***child threads*** are calling ***sa.sumArray() concurrently***, and the value of ***sum*** is ***corrupted***.

key points of a synchronized method:

- A ***synchronized method*** is created by preceding its declaration with ***synchronized***.
- For any given object, once a ***synchronized method*** has been called, the object is ***locked*** and ***no synchronized methods*** on the same object can be used by another ***thread of execution***.
- Other ***threads*** trying to call an ***in-use synchronized object*** will enter a ***wait state*** until the object is ***unlocked***.
- When a ***thread*** leaves the ***synchronized method***, the object is ***unlocked***.

8.6.2 Synchronized Statement

Sometimes ***synchronized methods*** don't help of achieving ***synchronization***. *Eg: you might want to* ***synchronize access*** *to some method that is* ***not modified*** *by* ***synchronized***. *[If you want to use a class that was not created by you but by a third party, and you do not have access to the* ***source code****. Thus, it is not possible for you to add* ***synchronized*** *to the appropriate methods within the class. How can access to an object of this class be* ***synchronized****?]*

The solution to this problem is: You simply put ***calls*** to the ***methods defined by this class*** inside a ***synchronized block***.

- This is the ***general form*** of a ***synchronized block***: `synchronized(obj_ref) { /* statements to be synchronized */ }`
  - Here, ***obj_ref*** is a ***reference*** to the ***object being synchronized***.
  - Once a ***synchronized block*** has been entered, no other ***thread*** can call a ***synchronized method*** on the object ***referred to*** by ***obj_ref*** until the block has been exited.

Example 9: Another way to synchronize calls to sumArray( ) is to call it from within a synchronized block:

```
class SumArray {                    /* same as Example 8 (previous) */
   /* change */          int sumArray(int nums[]) { // sumArray( ) is not synchronized anymore.
                                    /* same as Example 8 (previous) */  }  }

class MyThread implements Runnable { /* same as Example 8 (previous) */
                                     public void run() {          /* same as Example 8 (previous) */
                          /* change */                synchronized(sa) {  /* Here, calls to sumArray( ) on sa are synchronized. */
                                                          answer = sa.sumArray(a);  }
                                     /* same as Example 8 (previous) */  }

class Sync {  public static void main(String args[]) { /* same as Example 8 (previous) */  }
```

8.6.3 CONCURRENCY utilities and FORK/JOIN Framework

The ***concurrency utilities***, which are packaged in ***java.util.concurrent*** (and its subpackages), support concurrent programming. Among several other items, they offer ***synchronizers***, ***thread pools***, ***execution managers***, and ***locks*** that expand your ***control*** over ***thread execution***. One of the most exciting features of the concurrent ***API*** is the ***Fork/Join Framework***.

The ***Fork/Join Framework*** supports what is often termed parallel programming. This is the name commonly given to the techniques that take advantage of computers that contain two or more processors (including multicore systems) by subdividing a ***task*** into ***subtasks***, with each subtask executing on its own processor.

- ***Fork/Join Framework*** streamlines the development of multithreaded code that automatically scales to utilize the number of processors in a system.
- The ***concurrency utilities*** in general, and the ***Fork/Join Framework*** specifically, are features that you will want to explore after you have become more experienced with ***multithreading***.

8.7 Thread Communication: `notify()`, `wait()`, & `notifyAll()`

Consider the situation: A ***thread*** called ***T*** is executing inside a ***synchronized method*** and needs access to a ***resource*** called ***R*** that is ***temporarily unavailable***. What should ***T*** do?

- ☹ Generally **T *enters*** some form of ***polling loop*** that ***waits for*** **R**, ***T ties up the object***, ***preventing*** other threads' ***access*** to it. This causes a slow program, because it makes some kind of ***QUEUE TYPE JAM*** of threads which is caused by ***thread T while it is waiting for resource R***.
- ☺ A better solution is to have ***T*** temporarily ***relinquish*** *(temporarily give-up/abandon)* **control of the object**, allowing another thread to ***run***. When ***R*** becomes **available**, ***T*** can be ***notified*** and **resume *execution***.
  - 👍 Such an approach relies upon some form of ***inter-thread communication*** in which one ***thread*** can **notify another** that it is ***blocked*** and **be notified** that it can ***resume execution***.

❒ Java supports ***inter-thread communication*** with the ***`wait()`***, ***`notify()`***, and ***`notifyAll()`*** methods. These methods are part of all objects because they are **implemented** by the ***Object class***.

- ☞ These methods should be called only from within a ***`synchronized`*** context. Here is how they are used:
  - ▶ When a thread is *temporarily blocked from **running***, it calls ***`wait()`***. This causes the thread to ***go to sleep*** and the ***monitor*** *for that object to be **released***, allowing ***another thread*** to use the object.
  - ▶ The ***sleeping thread*** is awakened when some ***other thread*** enters the ***same monitor*** & calls ***`notify()`***, or ***`notifyAll()`***.

❒ ***wait( ):*** Following are the various forms of wait( ) defined by Object:

- ❖ The ***first*** form waits until notified.
- ❖ ***second*** form waits until notified or until the specified period of ***milliseconds*** has expired.
- ❖ The ***third*** form allows you to specify the wait period in terms of ***nanoseconds***.

```
final void wait( ) throws InterruptedException
final void wait(long millis) throws InterruptedException
final void wait(long millis, int nanos) throws InterruptedException
```

❒ ***notify( ) and notifyAll( ):*** Here are the general forms for ***`notify()`*** and ***`notifyAll()`***:

```
final void notify()
final void notifyAll()
```

A call to ***notify()*** **resumes *one*** waiting thread.
A call to ***notifyAll()*** notifies ***all*** threads, with the **highest priority** thread gaining access to the object.

❒ ***Spurious/false wakeup:*** Although ***`wait()`*** normally waits until ***`notify()`*** or ***`notifyAll()`*** is called, there is a possibility that in very rare cases the ***waiting thread*** could be ***awakened*** due to a ***spurious wakeup***. Because of the *remote possibility* of a ***spurious/false wakeup***, calls to ***`wait()`*** should take place within a ***loop*** that checks *the condition on which the thread is waiting*.

Example 10: To understand the need for and the application of ***`wait()`*** and ***`notify()`***, we will create a program that simulates the **ticking of a clock** by displaying the words **Tick** and **Tock** on the screen.

- ☛ We will create a **class** called ***`TickTock`*** that contains two methods: ***`tick()`*** and ***`tock()`***.
- ☛ The ***`tick()`*** method displays the word "***`Tick`***", and ***`tock()`*** displays "***`Tock`***".
- ☛ To *run the clock*, two threads are created, one that calls ***`tick()`*** and one that calls ***`tock()`***.
- ☛ The goal is to make the two threads execute in a way that the output from the program displays a consistent "Tick Tock"—that is, a repeated pattern of one tick followed by one tock.

```
class TickTock { String state; // contains the state of the clock
    synchronized void tick(boolean running) {
        if(!running) {      state = "ticked";   notify();
                            return; }           // stop the clock
        System.out.print("Tick ");
        state = "ticked";   // set the current state to ticked
        notify();           // let tock() run: tick() notifies tock()
    // tick() wait for tock() to complete: wait until state is not tocked, while loop continues
        try {   while(!state.equals("tocked")) wait(); }
        catch(InterruptedException exc) {
                System.out.println("Thread interrupted."); }
        }   //tick method ends

    synchronized void tock(boolean running) {
        if(!running) {      state = "tocked";   notify();
                            return; }           // stop the clock
        System.out.println("Tock");
        state = "tocked";   // set the current state to tocked
        notify();           // let tick() run: tock() notifies tick()
    // tock() wait for tick() to complete: wait until state is not ticked, while loop continues
        try {   while(!state.equals("ticked")) wait();}
        catch(InterruptedException exc) {
                System.out.println("Thread interrupted."); }
        }   //tock method ends
}
```

```
class MyThread implements Runnable {    Thread thrd;
                                        TickTock ttOb;
    // Construct a new thread.
    MyThread(String name, TickTock tt) {
        thrd = new Thread(this, name);      ttOb = tt;
        thrd.start();       /* start the thread */  }
// Begin execution of new thread.
public void run() {if(thrd.getName().compareTo("Tick") == 0){
                    for(int i=0; i<5; i++) ttOb.tick(true);
                    ttOb.tick(false);   }
                else {  for(int i=0; i<5; i++) ttOb.tock(true);
                    ttOb.tock(false);   }
                }   }

class ThreadCom { public static void main(String args[]) {
                    TickTock tt = new TickTock();
                    MyThread mt1 = new MyThread("Tick", tt);
                    MyThread mt2 = new MyThread("Tock", tt);
    try {  mt1.thrd.join();     mt2.thrd.join();  }
    catch(InterruptedException exc) {
                System.out.println("Main thread interrupted"); }
}}
```

- ❖ ***tick( )*** and ***tock( )***, which communicate with each other to ensure that a ***Tick*** is always followed by a ***Tock***, which is always followed by a ***Tick***, and so on.
- ❖ Notice the ***state*** field. When the clock is running, state will hold either the string "***ticked***" or "***tocked***", which indicates the current ***state*** of the clock.
- ❖ In ***main( )***, a ***TickTock*** object called ***tt*** is created, and this object is used to start two threads of execution.
- ❖ The threads are based on objects of type ***MyThread***. The ***MyThread*** constructor is passed two arguments. The first becomes the ***name of the thread***. This will be either "***Tick***" or "***Tock***". The second is a ***reference*** to the ***TickTock object***, which is ***tt*** in this case.
- ❖ Inside the ***run( )*** method of ***MyThread***, if the name of the thread is "***Tick***", then calls to ***tick( )*** are made. If the name of the thread is "***Tock***", then the ***tock( )*** method is called.
  - ▶ ***Five calls*** that pass ***true*** as an argument are made to each method. The ***clock*** runs as long as ***true*** is passed. A ***final call*** that passes ***false*** to each method ***stops*** the ***clock***.

The most important part of the program is found in the ***tick()*** and ***tock()*** methods of ***TickTock***. Consider the ***tick()*** method

```
synchronized void tick(boolean running) {    if(!running) {    state = "ticked";   notify();   return; }    // stop the clock
                                             System.out.print("Tick ");   state = "ticked";    // set the current state to ticked
                                             notify();    // let tock() run: tick() notifies tock()
/* tick() wait for tock() to complete: */    try {   while(!state.equals("tocked"))  wait(); }
/* wait until state is not tocked, while loop continues */   catch(InterruptedException exc) { System.out.println("Thread interrupted."); } }
```

- ***tick()*** is modified by ***synchronized***. Since, ***wait()*** and ***notify()*** apply only to *synchronized methods*.
- The method begins by checking the value of the ***running***. ***running*** is used to provide a **clean shutdown** of the clock. If it is ***false***, then the clock has been stopped. If this is the case, state is set to "***ticked***" and a call to ***notify()*** is made to enable any waiting thread to run. (Discussed in the last point).
- Assuming that the clock is running when ***tick()*** executes, the word "***Tick***" is displayed, state is set to "***ticked***", and then a call to ***notify()*** takes place. The call to ***notify()*** allows a thread waiting on the same object to run.
- Next, ***wait()*** is called within a ***while*** loop. The call to ***wait()*** causes ***tick()*** to suspend until **another thread** calls ***notify()***. Therefore, the loop will not iterate until **another thread** calls ***notify()*** on the same object. As a result, when ***tick()*** is called, it displays one "***Tick***", lets **another thread** run, and then **suspends**.
- The while loop that calls ***wait()*** checks the value of ***state***, waiting for it to equal "***tocked***", which will be the case only after the ***tock()*** method executes.
  - A ***while*** loop check this condition to prevent a **spurious/false wakeup** from incorrectly restarting the thread. If state does not equal "***tocked***" when ***wait()*** returns, it means that a **spurious wakeup** occurred, and ***wait()*** is simply called again.
- The ***tock()*** method is an exact copy of ***tick()*** except that it displays "***Tock***" and sets state to "***tocked***". Thus, when entered, it displays "***Tock***", calls ***notify()***, and then waits.
- When viewed as a pair, a call to ***tick()*** can only be followed by a call to ***tock()***, which can only be followed by a call to ***tick()***, and so on. Therefore, the two methods are **mutually synchronized**.
- Inside ***if-block*** the reason for the call to ***notify()*** when the clock is stopped is to allow a final call to ***wait()*** to **succeed**. Since, both ***tick()*** and ***tock()*** execute a call to ***wait()*** after displaying their message. ***The problem is that when the clock is stopped, one of the methods will still be waiting***. A final call to ***notify()*** is required for the waiting method to **run**.
  - If ***notify()*** inside ***if-block*** is removed then the program will "***hang***" and you will need to press **CTRL-C** to exit. The reason for this is that when the final call to ***tock()*** calls ***wait()***, there is no corresponding call to ***notify()*** that lets ***tock()*** conclude. Thus, ***tock()*** just **sits there**, ***waiting forever***.

NOTE

[1] If all calls to ***wait()*** and ***notify()*** are removed then ***tick()*** and ***tock()*** won't run/work together.

```
class TickTock { String state;     // contains the state of the clock
        synchronized void tick(boolean running) {
                if(!running) { state = "ticked"; return; } // stop the clock
                System.out.print("Tick ");
                state = "ticked"; // set the current state to ticked
                }
        synchronized void tock(boolean running) {
                if(!running) { state = "tocked"; return; } // stop the clock
                System.out.println("Tock");
                state = "tocked"; /* set the current state to tocked   */      } }
```

OUTPUT of *modified version* ***without*** call to ***wait()*** and ***notify()***:

```
Tick Tick Tick Tick Tick Tock
Tock
Tock
Tock
Tock
```

OUTPUT of ***Example 10*** with proper call to ***wait()*** and ***notify()***:

```
Tick Tock
Tick Tock
Tick Tock
Tick Tock
Tick Tock
```

[2] ***Deadlock*** *and* ***Race condition,*** *and how to avoid them:*

Deadlock: **Deadlock** is a situation in which ***one thread is waiting for another thread*** to do something, but that ***other thread is waiting on the first***. Thus, both threads are suspended, waiting on each other, and **neither executes**.

[Avoiding deadlock is not Easy. For example, deadlock can occur in round-about ways. The cause of the ***deadlock often is not readily understood just by looking at the source code*** to the program because concurrently executing threads can interact in *complex ways at run time.* To avoid deadlock, ***careful programming*** and ***thorough testing*** is required. ***Remember, if a multithreaded program occasionally "hangs," deadlock is the likely cause.***]

Race condition: A **race condition** occurs when two (or more) threads attempt to access a ***shared resource*** at the ***same time***, without ***proper synchronization***.

[For example, one thread may be writing a new value to a variable while another thread is incrementing the variable's current value. Without synchronization, the new value of the variable will depend upon the order in which the threads execute. (Does the second thread increment the original value or the new value written by the first thread?) In situations like this, the two threads are said to be "racing each other," with the final outcome determined by which thread finishes first.]

Like deadlock, a race condition can occur in ***difficult-to-discover*** ways. The solution is prevention: ***careful programming*** that ***properly synchronizes*** access to shared resources.

8.8 Suspending, Resuming, and Stopping Threads

It is sometimes useful to ***suspend*** **execution of a thread**. For example, a separate thread can be used to display the time of day. If the user does not desire a clock, then its thread can be Suspended (and can be enabled if user wants). The mechanisms to ***suspend***, ***stop***, and ***resume*** threads differ between **early versions** of ***Java*** and more **modern versions**, beginning with Java 2.

- Prior to **Java 2**, a program used ***suspend( )***, ***resume( )***, and ***stop( )***, which are methods defined by ***Thread***, to ***pause***, ***restart***, and ***stop*** the execution of a thread. They have the following forms:
  - ***final void resume()***
  - ***final void suspend()***
  - ***final void stop()***

[While these methods seem to be a perfectly reasonable and convenient approach to ***managing the execution of threads***, they must no longer be used. Here's why. The **suspend( )** and **stop()** method of the **Thread class** were deprecated by **Java 2** because these two methods can sometimes cause serious problems that involve **deadlock**. The **resume( )** method is also deprecated because it cannot be used without the **suspend( )** method as its **counterpart**.]

❒ ***To pause, restart, or terminate a thread (modern way):*** A thread must be designed in a way so that the ***run()*** method periodically checks to determine if that thread should ***suspend***, ***resume***, or ***stop*** its own execution.

☞ This is accomplished by establishing ***two*** **flag variables**: one for ***suspend*** and ***resume***, and one for ***stop***. For ***suspend*** and ***resume***, as long as the flag is set to "***running***," the ***run()*** method must continue to *let the thread execute*. If this variable is set to "***suspend***," the thread must ***pause***. For the ***stop flag***, if it is set to "***stop***," the thread must ***terminate***.

Example 11: The following example shows one way to implement your own versions of ***suspend()***, ***resume()***, and ***stop()***:

OUTPUT
```
My Thread starting.
1 2 3 4 5 6 7 8 9 10
11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40
Suspending thread.
Resuming thread.
41 42 43 44 45 46 47 48 49 50
51 52 53 54 55 56 57 58 59 60
61 62 63 64 65 66 67 68 69 70
71 72 73 74 75 76 77 78 79 80
Suspending thread.
Resuming thread.
81 82 83 84 85 86 87 88 89 90
91 92 93 94 95 96 97 98 99 100
101 102 103 104 105 106 107
108 109 110
111 112 113 114 115 116 117 118
119 120
Stopping thread.
My Thread exiting.
Main thread exiting.
```

```
class MyThread implements Runnable {     Thread thrd;
                                         boolean suspended, stopped;
    MyThread(String name) {   thrd = new Thread(this, name);
                              suspended = false;   stopped = false;
                              thrd.start(); }
    public void run() {
        System.out.println(thrd.getName() + " starting.");
        try {     for(int i = 1; i < 1000; i++){ System.out.print(i + " ");
                                  if((i%10)==0) {    System.out.println();
                                                     Thread.sleep(250); }
/* Use synchronized block to check suspended and stopped. */   synchronized(this) { while(suspended){ wait(); }
                                                                 if(stopped) break;       }
                  } }
        catch (InterruptedException exc){ System.out.println(thrd.getName() + " interrupted."); }
        System.out.println(thrd.getName() + " exiting.");
        }
                                          // Stop the thread.
    synchronized void mystop(){ stopped = true;
                                suspended = false; notify();}/* ensures that a suspended thread can be stopped*/
    synchronized void mysuspend() { suspended = true; }      // Suspend the thread.
    synchronized void myresume() { suspended = false;  notify(); }   // Resume the thread.
    }

class Suspend { public static void main(String args[]) {  MyThread ob1 = new MyThread("My Thread");
    try {     Thread.sleep(1000);        // let ob1 thread start executing
              ob1.mysuspend();   System.out.println("Suspending thread."); Thread.sleep(1000);
              ob1.myresume();    System.out.println("Resuming thread.");   Thread.sleep(1000);
              ob1.mysuspend();   System.out.println("Suspending thread."); Thread.sleep(1000);
              ob1.myresume();    System.out.println("Resuming thread.");   Thread.sleep(1000);
              ob1.mysuspend();   System.out.println("Stopping thread.");
              ob1.mystop();    }
    catch (InterruptedException e ) {   System.out.println("Main thread Interrupted");   }
    try { ob1.thrd.join(); }      // wait for thread to finish
    catch (InterruptedException e) { System.out.println("Main thread Interrupted"); }
    System.out.println("Main thread exiting.");   }}
```

☞ The thread class ***MyThread*** defines two ***Boolean*** variables, ***suspended*** and ***stopped***, which govern the **suspension** and **termination** of a thread. Both are initialized to ***false*** by the **constructor**.

☞ The ***run()*** method contains a **synchronized statement block** that checks ***suspended***. If that variable is ***true***, the ***wait()*** method is invoked to suspend the execution of the thread.

☞ To ***suspend*** execution of the thread, call ***mysuspend()***, which sets ***suspended*** to ***true***.

☞ To ***resume*** execution, call ***myresume()***, which sets ***suspended*** to ***false*** and invokes ***notify()*** to **restart the thread**.

☞ To ***stop*** the thread, call ***mystop()***, which sets stopped to ***true***. In addition, ***mystop()*** sets suspended to ***false*** and then calls ***notify()***. These steps are necessary to **stop a suspended thread**.

8.9 Using the Main Thread

All Java programs have at least one thread of execution, called the main thread, which is given to the program automatically when it begins running. The main thread can be handled just like all other threads.

❒ To access the ***main thread***, you must obtain a ***Thread*** object that refers to it. You do this by calling the ***currentThread()*** method, which is a ***static*** member of ***Thread***. Its general form is:

```
static Thread currentThread( )
```

☛ This method returns a **reference** to the thread in which it is called. Therefore, if you call ***currentThread()*** while *execution is inside the* ***main thread***, you will obtain a ***reference*** to the ***main thread***. By this ***reference***, you can control the ***main thread***.

❒ ***Example 12:*** Following obtains a **reference** to the **main thread**, and then gets and sets the **main thread**'s name and priority.

```
class UseMain { public static void main(String args[]) {
                    Thread thrd;
// Get the main thread */   thrd = Thread.currentThread();

// Display main thread's name.
    System.out.println("Main thread is called: " + thrd.getName());
// Display main thread's priority.
    System.out.println("Priority: " + thrd.getPriority());
    System.out.println();
```

```
                // Set the name and priority.
System.out.println("Setting name and priority.\n");
thrd.setName("Thread #1");
thrd.setPriority(Thread.NORM_PRIORITY+3);
System.out.println("Main thread is now called: " + thrd.getName());
System.out.println("Priority is now: " + thrd.getPriority());
}}
```

❒ ***Be careful*** about what operations you perform on the **main thread**. For example, if you add the following code to the end of ***main()***, the program will ***never terminate*** because it will be waiting for the main thread to end!

```
try { thrd.join(); } //waiting for main thread to terminate
catch(InterruptedException exc) {System.out.println("Interrupted");}
```

OUTPUT:

```
Main thread is called: main
Priority: 5
Setting name and priority.
Main thread is now called: Thread #1
Priority is now: 8
```

NOTE: ***Effective usage of multithreading:***

[1] The key to effectively utilizing multithreading is to **think concurrently rather than serially**. For example, when you have two subsystems within a program that are fully independent of each other, consider making them into individual threads.

[2] If you create ***too many threads***, you can actually ***degrade the performance*** of your program rather than enhance it.

[3] Remember, overhead is associated with context switching. If you create ***too many threads***, ***more CPU time*** will be spent changing contexts than in executing your program!

8.10 Enumerations : Introduction

An enumeration is a list of **named constants** that define a ***new data type***. An ***object*** of an ***enumeration type*** can hold only the values that are defined by the list. It is a way to precisely define a ***new type of data that has a fixed number of valid values***.

☼ Enumerations are common in everyday life. For example:

- An enumeration of the ***coins*** used in the U.S. is ***penny***, ***nickel***, ***dime***, ***quarter***, ***half-dollar***, and ***dollar***.
- An enumeration of the ***months*** in the year consists of the names ***January*** through ***December***.
- An enumeration of the ***days*** of the ***week*** is ***Sunday***, ***Monday***, . . . , ***Saturday***.

❒ Enumerations are useful to define a **set of values** that represent a **collection of items**. For example, to represent a set of status codes, such as ***success***, ***waiting***, ***failed***, and ***retrying***, which indicate the progress of some action.

☹ In the past, such values were defined as ***final variables***, but ***enumerations*** offer a more ***structured approach***.

❒ ***Creating Enumerations:*** An enumeration is created using the ***enum*** keyword. Eg: An enumeration that lists various transportation:

```
enum Transport { CAR, TRUCK, AIRPLANE, TRAIN, BOAT }
```

☞ The identifiers ***CAR***, ***TRUCK***, and so on, are called ***enumeration constants***. Each is implicitly declared as a ***public***, ***static*** member of ***Transport***.

☞ In Java, *enumeration constants* are called *self-typed*, where "*self*" refers to the *enclosing enumeration*. i.e. **enumeration constants' *type*** is the ***type*** of the ***enumeration*** in which the constants are declared, which is ***Transport*** in this case.

❒ ***Declare and Use an enumeration variable:*** However, even though ***enumerations*** define a ***class type***, you ***do not instantiate*** an ***enum*** using ***new***. Instead, ***you declare and use an enumeration variable in much the same way that you do one of the primitive types***. For example, to declare ***tp*** as a variable of *enumeration type* ***Transport***: `Transport tp;`

☛ Because ***tp*** is of *type* ***Transport***, the only values that it can be assigned are those defined by the ***enumeration***. For example, this assigns ***tp*** the value ***AIRPLANE***: `tp = Transport.AIRPLANE;` Notice, ***AIRPLANE*** is qualified by ***Transport***.

❒ ***Comparison between enumeration constants:*** Two ***enumeration constants*** can be compared for ***equality*** by using the **= =** *relational operator*. Eg: `if(tp == Transport.TRAIN){}` compares the value in ***tp*** with the ***TRAIN*** constant.

☝ An ***enumeration value*** can be used to control a ***switch*** statement. ***All of the case statements must use constants from the same enum***. Eg: this switch is perfectly valid:

```
switch(tp) { case CAR: //…
             case TRUCK: //… }
```

▶ NOTE : The **type** of the **enumeration** in the ***switch*** expression has already **implicitly specified** the ***enum*** type of the *case constants*. i.e. there is no need to use ***Transport.TRUCK*** in *case statements*. In fact, attempting to qualify the *constants in the case statements* with their ***enum*** type name will cause a ***compilation error***.

❒ When an *enumeration constant* is displayed, such as in a ***println()*** statement, **its name is output**. For example:

```
System.out.println(Transport.BOAT);
```

the name ***BOAT*** will be displayed.

Example 13: The following program puts together all of the pieces and demonstrates the Transport enumeration:

```
/* Declare an enumeration. */
enum Transport { CAR, TRUCK, AIRPLANE, TRAIN, BOAT }

class EnumDemo { public static void main(String args[]) {
    Transport tp;  //declaring Declare a Transport reference.
    tp = Transport.AIRPLANE; // Assign tp the constant AIRPLANE.
    System.out.println("Value of tp: " + tp); // Output an enum value.
    System.out.println();
    tp = Transport.TRAIN;
    if(tp == Transport.TRAIN)  // Compare two enum values.
    System.out.println("tp contains TRAIN.\n");

// Use an enum to control a switch statement.
switch(tp) {
case CAR: System.out.println("A car carries people."); break;
case TRUCK: System.out.println("A truck carries freight."); break;
case AIRPLANE: System.out.println("An airplane flies."); break;
case TRAIN: System.out.println("A train runs on rails."); break;
case BOAT: System.out.println("A boat sails on water."); break;
    } }}
```

OUTPUT:

```
Value of tp: AIRPLANE
tp contains TRAIN.
A train runs on rails.
```

❒ There is no rule that requires **enumeration constants** to be in **uppercase** (CAR, not car). Enumerations often replace final variables, which have traditionally used uppercase, some programmers believe that uppercasing enumeration constants is also appropriate.

❒ ***Java Enumerations Are Class Types:*** Unlike the way enumerations are implemented in some other languages, Java implements enumerations as ***class types***. Although you don't instantiate an ***enum*** using ***new***, it otherwise acts much like other classes. For example, you can give it ***constructors***, add ***instance variables*** and ***methods***, and even implement ***interfaces***.

❒ ***values() and valueOf()***: ***values()*** and ***valueOf()*** are predefined in all enumerations. Their general forms:

```
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***.

Example 14: The following program demonstrates the ***values()*** and ***valueOf()*** methods:

```
Transport tp;
System.out.println("All Transport constants");
Transport allTransports[] = Transport.values();   // use values()
for(Transport t : allTransports) System.out.println(t);
                                                  // use valueOf()
tp = Transport.valueOf("AIRPLANE");
System.out.println("tp contains " + tp);
```

```
All Transport constants
CAR
TRUCK
AIRPLANE
TRAIN
BOAT
tp contains AIRPLANE
```

- The return type of ***Transport.valueOf("TRAIN")*** is ***Transport***. The value returned is ***TRAIN***.
- Notice a ***for-each style for loop*** is used to cycle through the array of constants obtained by calling ***values()***. This form is also valid: `for(Transport t : Transport.values()) System.out.println(t);`

8.11 Enumerations: Constructors, Methods, Instance Variables

Each ***enumeration constant*** is an ***object*** of its ***enumeration type***. Thus, an enumeration can define *constructors*, add *methods*, and have *instance variables*.

- The *defined constructor* is called when each *enumeration constant* is created.
- Each *enumeration constant* can call any *method* defined by the enumeration.
- Each *enumeration constant* has its own copy of *instance variables* defined by the **enumeration**.

Example 15: Illustrates the use of constructor, instance variable, and method. It gives each type of transportation a typical speed.

```
enum Transport { CAR(65), AIRPLANE(600), TRAIN(70), BOAT(22);
                 private int speed;  // instance variable, transport speed
                 Transport(int s) { speed = s; }   // Constructor
/* Add a method */  int getSpeed() { return speed; } }
```

```
OUTPUT:  Airplane speed 600 m/h.
         All Transport speeds:
         CAR speed is 65 m/h.
         AIRPLANE speed is 600 m/h.
         TRAIN speed is 70 m/h.
         BOAT speed is 22 m/h.
```

```
class EnumDemo3 { public static void main(String args[]) {
    Transport tp;
                        // Display speed of an airplane.
    System.out.println("airplane speed: " +
                       Transport.AIRPLANE.getSpeed() + " m/h.\n");
                        // Display all Transports and speeds.
    System.out.println("All Transport speeds: ");
    for(Transport t : Transport.values())
    System.out.println(t + " speed : " + t.getSpeed() + " m/h.");
}}
```

- Here the *instance variable* is ***speed***, ***Transport(int s)*** is the *constructor*, ***getSpeed()*** is a *method*,
- When the variable ***tp*** is declared in ***main()***, the **constructor** for ***Transport*** is called once for each constant that is specified.
- Notice how **the *arguments* to the *constructor* are specified**, by *putting them inside parentheses*, after each constant, as shown here: `CAR(65), TRUCK(55), AIRPLANE(600), TRAIN(70), BOAT(22);`
  - These values are passed to the ***s*** parameter of ***Transport(int s)***, which then assigns this value to ***speed***.
- In ***main()*** the speed of an airplane is obtained using ***getSpeed()*** by the call : ***Transport.AIRPLANE.getSpeed()***
- The speed of each transport is obtained by cycling through the *enum* using a ***for loop***. Because there is a copy of speed for each *enum* constant, the value associated with one constant is separate and distinct from the value associated with another constant.

- Notice, the ***list of enumeration constants is terminated*** *by a* ***semicolon***. That is, the last constant, ***BOAT***, is followed by a *semicolon*. When an enumeration contains *other members*, the enumeration list must end in a *semicolon*.
- An *enum* can offer two or more *overloaded constructors*, just as can any other class.

8.12 Restrictions of inheritance, *JAVA.LANG.ENUM's ordinal() and compareTo()*

- An ***enum*** **can't *inherit*** another class.
- An ***enum*** **cannot be** a ***superclass***. This means that an ***enum*** can't be ***extended***.
- Remember that each of the *enumeration constants* is an *object* of the class in which it is defined.
- ***FINAL variable and ENUM:*** ***Enum***s are appropriate when you are working with lists of items that must be represented by identifiers. A final variable is appropriate when you have a constant value, Eg: an array size, that will be used in many places.

Although you can't inherit a **superclass** when declaring an ***enum***, all enumerations automatically inherit one: ***java.lang.Enum***. This class defines several methods among those methods, two methods that you may occasionally employ: ***ordinal()*** and ***compareTo()***.

- ***ordinal()***: The ***ordinal()*** method obtains a value that indicates an ***enumeration constant's position*** in the ***list of*** **constants**. This is called its ***ordinal value***. General form of ***ordinal()*** method: `final int ordinal()`
  - It *returns* the *ordinal value* of the invoking ***constant***. Ordinal values begin at ***zero***. [Thus, in the ***Transport*** enumeration, ***CAR*** has an ordinal value of ***zero***, ***TRUCK*** has an ordinal value of ***1***, ***AIRPLANE*** has an ordinal value of ***2***, and so on.]
- ***compareTo()***: To **compare** the **ordinal value** of two constants of the **same enumeration** use the ***compareTo()*** method. It has this general form: `final int compareTo(enum-type e)`
  - Here, ***enum-type*** is the **type** of the **enumeration** and ***e*** is the *constant being compared to the invoking constant*. Remember, ***both*** the ***invoking constant*** and ***e*** must be of the ***same enumeration***.
    - ☑ If the **invoking constant** has an ordinal value ***less*** than **e**'s, then ***compareTo()*** returns a ***negative*** value.
    - ☑ If the two ordinal values are the ***same***, then ***compareTo()*** returns ***zero***.
    - ☑ If the **invoking constant** has an ordinal value ***greater*** than **e**'s, then ***compareTo()*** returns a ***positive*** value.

```
enum Transport { CAR, TRUCK, AIRPLANE, TRAIN, BOAT }
class EnumDemo4 {
public static void main(String args[]) {
        Transport tp, tp2, tp3;
        tp = Transport.AIRPLANE;
        tp2 = Transport.TRAIN;
        tp3 = Transport.AIRPLANE;
```

```
System.out.println("All Transport constants" + " their ordinal values: ");
for(Transport t : Transport.values()) System.out.println(t + " " + t.ordinal());

if(tp.compareTo(tp2) < 0) System.out.println(tp + " comes before " + tp2);
if(tp.compareTo(tp2) > 0) System.out.println(tp2 + " comes before " + tp);
if(tp.compareTo(tp3) == 0) System.out.println(tp + " equals " + tp3);      }}
```

8.13 BOXING - UNBOXING and TYPE WRAPPERS (Recall Java/C# 6.18)

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

❒ ***NUMBER class:*** All of the **numeric type wrappers** (**Byte**, **Short**, **Integer**, **Long**, **Float**, and **Double**) inherit the ***abstract class Number***. ***Number*** declares methods that return the value of an object in each of the **different numeric types**. These methods are :

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

☛ **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** and **Double**:

```
Integer(int num)                        Integer(String str) throws NumberFormatException
Double(double num)                      Double(String str) throws NumberFormatException
```

▶ If **str** does not contain a valid numeric value, then a **NumberFormatException** is thrown.

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

❒ ***Boxing:*** The process of encapsulating a value within an object (type-wrapping) is called ***boxing***. Before JDK 5, all ***boxing*** took place **manually**, by ***explicitly constructing*** an ***instance*** of a ***wrapper*** with the desired value. Eg: to *manually box* **100** into an **Integer**:

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

☛ In this example, a new **Integer object** with the value **100** is explicitly created and a reference to this object is assigned to **iOb**.

❒ ***Unboxing:*** The process of ***extracting*** a value from a ***type wrapper*** is called ***unboxing***. Before JDK 5, all ***unboxing*** also took place **manually**, by ***explicitly calling*** a ***method*** on the ***wrapper*** to obtain its value. Eg: to *manually unboxes* the value in **iOb** into an **int**.

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

☛ Here, **intValue()** returns the value ***encapsulated*** within **iOb** as an **int**.

8.14 Autoboxing/Unboxing

Autoboxing/unboxing greatly simplifies and streamlines code that must ***convert primitive types into objects, and vice versa***. It also contributes greatly to the usability of **generics**. **Autoboxing/unboxing** is directly related to Java's **type wrappers** (Boxing/Unboxing), and to the way that values are moved into and out of an **instance of a wrapper**.

❒ ***Autoboxing:*** ***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:*** ***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()**.

☯ ***Procedure of Autoboxing/Unboxing:*** 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 created*** through the use of **new** and the constructor **Integer()**.

➲ To **unbox an object**, simply assign that ***object reference*** to a **primitive-type variable**. For example, to ***unbox iOb***:

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

❒ ***Autoboxing with Methods:*** ***Autoboxing*** *automatically occurs* whenever a ***primitive type*** must be converted into an ***object***, and ***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***.

Example 16: In the following: ***Autoboxing/unboxing*** takes place with method ***parameters*** and ***return*** values.

```
class AutoBox2 {  static void m(Integer v) { System.out.println("m() received " + v); }      // Integer wrapper parameter.
                  static int m2() { return 10; }          // This method returns an primitive int.
                  static Integer m3() { return 99; }      // returns an wrapped Integer: autoboxing 99 into an Integer.

                  public static void main(String args[]) {
                        m(199);                           // Pass an int to m(). Which will be autoboxed to Integer, since m() has an Integer parameter.
                        Integer iOb = m2();               //autoboxing primitive int value returned by m2() to an Integer type object iOb
                        System.out.println("Return value from m2() is " + iOb);

                        int i = m3();       // auto-unboxed into an int: assigning m3() returned wrapped Integer value to a primitive int type variable i
                        System.out.println("Return value from m3() is " + i);

                                            // In following case, iOb is auto-unboxed and its value promoted to double, which is the type needed by sqrt().
                        System.out.println("Square root of iOb is " + Math.sqrt(iOb));    }}
```

- Here ***m( )*** specifies an ***Integer object*** parameter. Inside ***main( )***, ***m( )*** is passed the ***int*** value ***199***. Since ***m( )*** is expecting an ***Integer object***, this value is ***autoboxed***
- Next, ***m2( )*** is called. It returns the ***int*** value ***10***. This ***int*** value is assigned to ***iOb*** in ***main( )***. Because ***iOb*** is an ***Integer***, the value returned by ***m2( )*** is ***autoboxed***.
- Next, ***m3( )*** is called. It returns an ***Integer object*** that is ***auto-unboxed*** into an ***int***.
- Finally, ***Math.sqrt( )*** is called with ***iOb*** as an argument. In this case, ***iOb*** is ***auto-unboxed*** and its value ***promoted*** to ***double***, since *primitive* ***double*** *is* expected by ***Math.sqrt( )***.

❒ ***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:

Example 17: *Autoboxing/ unboxing* in inctement expression and elevation in an expression

```
class AutoBox3 { public static void main(String args[]) {  Integer iOb, iOb2;      // Integer objects
                                                            int i;                  // primitive int variable
                 iOb = 99;   System.out.println("Original value of iOb: " + iOb);

  Autoboxing/unboxing occurs in expressions.
                 ++iOb;      // automatically unboxes iOb, performs the increment, and then reboxes the result back into iOb.
                 System.out.println("After ++iOb: " + iOb);
                 iOb += 10;  // iOb is unboxed, its value is increased by 10, and the result is boxed and stored back in iOb.
                 System.out.println("After iOb += 10: " + iOb);
                 iOb2 = iOb + (iOb / 3);   // iOb is unboxed, the expression is evaluated, and the result is reboxed and assigned to iOb2.
                 System.out.println("iOb2 after expression: " + iOb2);
                 i = iOb + (iOb / 3);      // The same expression is evaluated, but the result is not reboxed.
                 System.out.println("i after expression: " + i); }}
```

OUTPUT:

```
Original value of iOb: 99
After ++iOb: 100
After iOb += 10: 110
iOb2 after expression: 146
i after expression: 146
```

☛ Notice, **`++iOb;`** causes the value in ***`iOb`*** to be ***`incremented`***. It works like this: ***`iOb`*** is ***unboxed***, the value is **incremented**, and the result is ***reboxed***.

❒ ***Integer numeric objects in SWITCH:*** Because of **auto-unboxing**, you can use ***integer numeric objects***, such as an ***Integer***, 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.

NOTE (RESTRICTION):

☠ ***Autoboxing/unboxing was not added to Java as a "ALTERNATIVE" to primitive types:*** Each ***autobox*** and ***auto-unbox*** adds overhead that is not present if the ***primitive type*** is used. In general, you should ***restrict*** your ***use of the `type wrappers`*** to only those cases in which an **object representation** of a **primitive type** is ***required***. For example, it is possible to write code like this:

`Double a, b, c; // A bad use of autoboxing/unboxing!` `a = 10.2;  b = 11.4;  c = 9.8;` `Double avg = (a + b + c) / 3;`	Although this code is technically correct and work properly, but it is a **very bad use of *autoboxing/unboxing***. It is far ***less efficient*** than the equivalent code written using the **primitive type *`double`***.

8.15 STATIC import (an extended use of **import** keyword. Recall Packages: Java/C# 5.5)

By following ***`import`*** with the keyword ***`static`***, an **import statement** can be used to import the **static members** of a ***class*** or ***interface***. This is called ***`static import`***. When using ***static import***, it is possible to ***refer to static members directly by their names, without having to qualify them with the name of their class***. This simplifies and shortens the syntax required to use a **static member**. Example:

Solution of the quadratic equation: $ax^2 + bx + c = 0$	
*General way: **specify class name***	*With **`static import`***
// Find first solution. x = (-b + **Math.sqrt**(**Math.pow**(b, 2) - 4 * a * c)) / (2 * a); *// Find second solution.* x = (-b - **Math.sqrt**(**Math.pow**(b, 2) - 4 * a * c)) / (2 * a);	`import static java.lang.Math.sqrt;` *//Static import* `import static java.lang.Math.pow;` *//Static import* `x = (-b + sqrt(pow(b, 2) - 4*a*c)) / (2*a);` `x = (-b - sqrt(pow(b, 2) - 4*a*c)) / (2*a);`
Because ***`pow()`*** and ***`sqrt()`*** are **static methods**, they must be called through the use of their class' name, ***`Math`***.	Eliminate the ***class name*** through the use of ***static import***. Class name ***`Math`*** is no longer necessary to qualify ***`sqrt()`*** or ***`pow()`***.

❒ ***General forms of the `import static` statement:*** There are two general forms of the import static statement.

[1] The ***first form***, which is used by the preceding example, ***brings into view a single name***. Its general form is:

`import static pkg.type-name.static-member-name;`

- Here, ***`type-name`*** is the name of a class or interface that contains the desired static member (Eg: ***`Math`***). Its full package name is specified by ***`pkg`***, (Eg: ***`java.lang`***).
- The name of the member is specified by ***`static-member-name`***, (Eg: ***`sqrt`***).

[2] The ***second form*** of static import ***imports all static members***. Its general form is :

`import static pkg.type-name.*;`

- If you will be using many static methods or fields defined by a class, then this form lets you bring them into view without having to specify each individually. For example: To bring both ***`pow()`*** and ***`sqrt()`*** *(and all other **static members** of **Math**)* into view: `import static java.lang.Math.*;`

☛ Of course, ***`static import`*** is not limited just to the ***`Math`*** class or just to **methods**. For example, this brings the ***static field `System.out`*** into view: ***`import static java.lang.System.out;`***

☑ After this statement, you can **output to the console** without qualifying ***`System`***, as: `out.println("string");`

❒ ***Import the static members of user defined classes:*** you can use ***`static import`*** to import the static members of ***classes*** and ***interfaces*** you create. Doing so is especially convenient when you define several ***static members*** that are used frequently throughout a large program.

NOTE: [1] Remember, one reason that Java organizes its ***libraries*** into ***packages*** is to avoid ***namespace collisions***. When you import ***static members***, you are bringing those members into the ***global namespace***. Thus, you are ***increasing the potential for namespace conflicts*** and the ***inadvertent hiding*** of other names.

[2] ***Static import*** is designed for those situations in which you are using a ***static member*** repeatedly, such as when performing a ***series of mathematical computations***. In essence, you should use, but not abuse, this feature.

8.16 Annotations (Metadata)

An ***annotation*** enables you to **embed supplemental information** into a ***source file***. An **annotation**, does not change the actions of a program. However, this information can be used by various tools, during both ***development*** and ***deployment***. For example, an annotation might be processed by a ***source-code generator***, by the ***compiler***, or by a ***deployment tool***.

The term ***metadata*** is also used to refer to this feature, but the term ***annotation*** is the most descriptive, and more commonly used. *Annotation is a large and sophisticated topic, so we'll take a short overview.*

- ***Creating Annotations:*** An annotation is created through a mechanism based on the **interface**. Eg: To declare an ***annotation*** called ***MyAnno:***

```
@interface MyAnno { String str();   int val(); }  // A simple annotation type.
```

  - Notice the ***@*** that precedes the keyword ***interface***. This tells the compiler that an annotation type is being declared.
  - Next, notice the two members str( ) and val( ). All annotations consist solely of method declarations. However, you don't provide bodies for these methods. Instead, Java implements these methods. Moreover, the methods act much like fields.

- All ***annotation types*** automatically ***extend*** the ***Annotation interface***. Thus, ***Annotation*** is a ***super-interface*** of all ***annotations***. It is declared within the ***java.lang.annotation*** package.

- Originally, ***annotations*** were used to **annotate only declarations**. In this usage, ***any type of declaration can have an annotation associated with it***. For example, *classes*, *methods*, *fields*, *parameters*, and *enum constants* can be annotated. Even an *annotation* can be annotated. In such cases, the annotation precedes the rest of the declaration. Beginning with JDK 8, you can also annotate a ***type use***, such as a ***cast*** or a method ***return type***.

- ***Applying annotation:*** When you *apply an annotation*, you ***give values to its members***. Here is an example of ***MyAnno*** being applied to a method called **myMethd()**.

```
@MyAnno(str = "Annotation Example", val = 100)
public static void myMethd() { // ...
```

  - This annotation is **linked** with the method ***myMethd()***.
  - The name of the annotation, preceded by an ***@***, is followed by a **parenthesized** list of ***member initializations***. To give a *member* a **value**, that member's name is ***assigned*** a value.
    - Therefore, in the example, the string ***"Annotation Example"*** is assigned to the ***str*** member of ***MyAnno***.
    - Notice that **no parentheses** follow ***str*** in this assignment. *When an* ***annotation member*** *is given a* ***value****, only its name is used*. Thus, ***annotation members*** look like ***fields*** in this context.

- ***Marker annotations:*** Annotations that ***don't have parameters*** are called **marker annotations**. These are specified ***without*** passing any ***arguments*** and ***without*** using ***parentheses***. Their sole purpose is to *mark an item with some attribute.*

- ***Built-in annotations of Java:*** Java defines many built-in annotations. Most are specialized, but ***nine*** **are general purpose**. Four are imported from ***java.lang.annotation***: ***@Retention, @Documented, @Target,*** and ***@Inherited***. Five are included in ***java.lang***, ***@Override, @Deprecated, @SafeVarargs, @FunctionalInterface***, and ***@SuppressWarnings***,

The General Purpose Built-in Annotations	
Annotation	***Description***
@Retention	Specifies the ***retention policy*** that will be associated with the ***annotation***. The retention policy determines how long an annotation is present during the ***compilation*** and ***deployment process***.
@Documented	A marker annotation that tells a tool that an annotation is to be documented. It is designed to be used only as ***an annotation*** to ***an annotation declaration***.
@Target	Specifies the ***types of items*** to which an ***annotation can be applied***. It is designed to be used only as *an annotation* to another *annotation*. ***@Target*** takes one argument, which must be a ***constant*** or ***array of constants*** from the ***ElementType enumeration***, which defines various constants, such as ***CONSTRUCTOR***, ***FIELD***, and ***METHOD***. The argument determines the types of items to which the annotation can be applied. If ***@Target*** is not specified, the annotation can be used on any declaration.
@Inherited	A ***marker annotation*** that causes the ***annotation*** for a *superclass* to be ***inherited*** by a *subclass*.
@Override	A ***method*** annotated with ***@Override*** must ***override*** a method from a ***superclass***. If it doesn't, a ***compile-time error*** will result. It is used to ensure that a ***superclass method*** is actually ***overridden***, and ***not simply overloaded***. This is a ***marker annotation***.
@Deprecated	A ***marker annotation*** that indicates that a declaration is *obsolete* and has been ***replaced by a newer form***.
@SafeVarargs	A ***marker annotation*** that indicates that ***no unsafe actions*** related to a ***varargs*** parameter in a ***method*** or ***constructor*** occur. Can be applied only to ***static*** or ***final methods or constructors***.
@SuppressWarnings	Specifies that one or more ***warnings*** that might be issued by the ***compiler*** are to be suppressed. The ***warnings*** to suppress are specified by ***name***, in ***string form***.
@FunctionalInterface	A ***marker annotation*** that is used to annotate an ***interface declaration***. It indicates that the ***annotated interface is a functional interface***, which is an ***interface*** that contains ***one and only one abstract method***. ***Functional interfaces*** are used by ***lambda expressions***. (See Chapter 10 for details on functional interfaces.) It is important to understand that ***@FunctionalInterface*** is ***purely informational***. *(Any interface with exactly one abstract method is, by definition, a functional interface.)*

Example 18: Here is an example that uses ***@Deprecated*** to mark the ***MyClass*** class and the ***getMsg()*** method. When you try to compile this program, warnings will report the use of these deprecated elements.

```
@Deprecated
class MyClass {                    // Deprecate a class.
        private String msg;
        MyClass(String m) { msg = m; }

                                   // Deprecate a method within a class.
        @Deprecated
        String getMsg() { return msg; }
/* ... */ }
```

```
class AnnoDemo {
public static void main(String args[]) {
        MyClass myObj = new MyClass("test");
        System.out.println(myObj.getMsg());
        }}
```

NOTE

To ***java.lang.annotation***, ***JDK*** 8 adds the annotations ***@Repeatable*** and ***@Native***.

- ***@Repeatable*** supports ***repeatable annotations***, which are annotations that can be applied more than once to a single item.
- ***@Native*** is used to ***annotate a constant field*** accessed by ***executable*** (i.e., native) ***code***. Both are special-use annotations.