Multithreaded Programming
Multithreading fundamentals, The Thread Class and Runnable Interface, Creating Thread, Creating Multiple Threads, Determining When a Thread Ends, Thread Priorities, Synchronization, using Synchronization Methods, The Synchronized Statement, Thread Communication using notify(), wait() and notifyAll(), suspending, Resuming and stopping Threads.

Enumerations, Autoboxing and Annotations

Enumerations, Java Enumeration are class types, The Values() and Valueof() Methods, Constructors ,methods ,instance variables and enumerations, Autoboxing, Annotations (metadata)

Multithreading fundamentals

Multitasking: Running 2 or more programmes simultaneously at the same time.

Multithreading: dividing the programs into smaller chunks called threads and running them simultaneously.

Thread:is a lightweight process that executes some task.

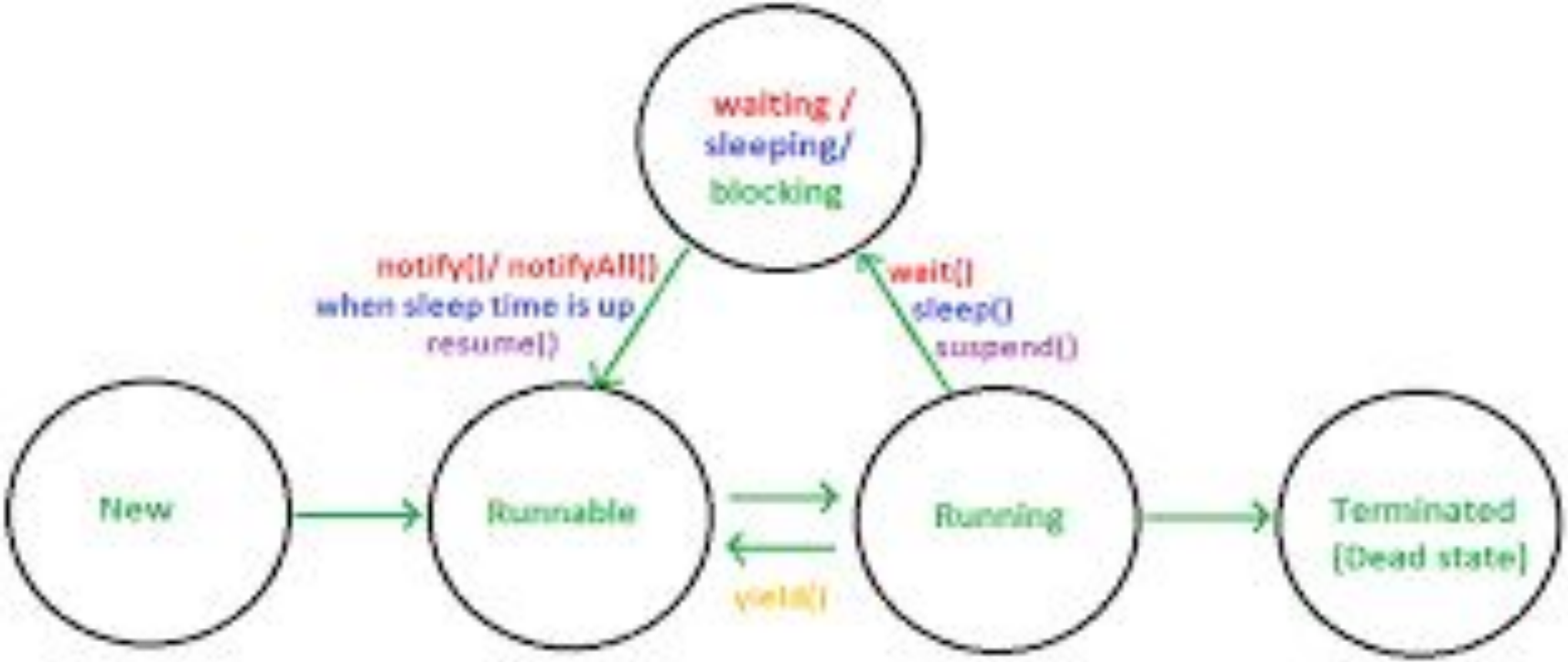

Fig. THREAD STATES

Life Cycle of thread
New: thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a born thread.
Runnable: A thread in this state is considered to be executing its task
Running:`

Waiting: a thread transitions to the waiting state while the thread waits for another thread to perform a task. A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.

Terminated: A runnable thread enters the terminated state when it completes its task or otherwise terminates.

Threads are lightweight than processes, it takes less time, less resource to create a thread.
Threads share their parent process data and code
• Context switching between threads is usually less expensive than between processes.

Thread intercommunication is relatively easy than process communication
Better resource utilization.
Simpler program design in some. situations.
More responsive programs

Thread class and Runnable Interface

Multithreading is built on the
Thread class and
its companion interface Runnable.
Both are packaged in java.lang.

All processes have atleast on thread of execution, which is usually called the main thread.
Thread can be created in two ways.

- implementing Runnable interface
- extending Thread class

implementing Runnable interface
the Runnable interface signature
public interface Runnable {
void run();
}

Steps to create thread by implementing Runnable
a class need to only implement a single method called run()
thread can be created by any method (like main or constructor) with following steps

1. Create object of class that implements run method
2. Create thread by passing object current class that implements run method.
Thread t = new Thread(Runnable obj)
3. Call start() method which in turn calls run method and starts thread.

```
class MyThread implements Runnable {
public void run() {
for(int i=0;i<40;i++)
System.out.println(i);
}
}
public static void main(String[] args) {
// first, construct a MyThread object
MyThread ta = new MyThread();
//next, construct a thread from that object.
Thread t = new Thread(ta);
```

extending Thread class

```
Class Test extends Thread
{
public void run()
{
for(int i=0; i<40;i++)
System.out.println(i);
System.out.println("end of run");
}
Test t =new Test(); t.start();
```

```
class MyThread implements Runnable {
public void run() {
for(int i=0;i<40;i++)
System.out.println(i);
}
public static void main(String[] args) {//first, construct a MyThread object
MyThread ta = new MyThread();//next, construct a thread from that object.
Thread t = new Thread(ta);
```

Even though we don't create any thread, java treats the main method as the first thread called main thread.
It is the thread from which other "child" threads will be spawned, it must be the last thread to finish execution because it performs various shutdown actions.
•Controlling main thread
o it can be controlled through a Thread object.
To do so, you must obtain a reference to i by calling the method currentThread(), which is a public static member of Thread.

Example: Getting reference to main thread:

```
Thread th=Thread.currentThread();
th.getName(); //gives thread name
```

```
public class Demo {
public static void main(String args[]) {
Thread th=Thread.currentThread();
System.out.println("Current thread: " + th.getName());
th.setName("My Thread"); //change the name of the thread
System.out.println("After name change: " + th);
try {
for(int n = 4; n > 0; n--)
{
System.out.println(n);
Thread.sleep(1000);
} }
catch (InterruptedException e) {
System.out.println("Main thread interrupted");
}}
```

sleep() method
static void sleep (long milliseconds)
throws InterruptedException
causes the thread from which it is called to suspend execution for the specified period of milliseconds.

Choosing an approach

*If you want to override many methods of Thread class then extend the thread class.

If you want to override only run method then better to implement runnable interface.

Creating Multiple Threads

```
class myThread implements Runnable { String name; // name of thread
public myThread(String threadName)
{
name = threadName:
}
public void run()
{
try
for(int i = 5; i >0; i--)
System.out.println( name + ":" + i), Thread sleep(1000);
} }
catch (InterruptedException e)
{
System.out.println(name + "Interrupted");
System.out.println( riame + " exiting.");
}}
```

all child threads execute concurrently
Threads are started in 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, because of differences in timing or environment, output from the program may differ.

Determining When a Thread Ends

As main is a parent for all child process, we want main to finish last over all threads.

How to ensure the main finishes last. Java provides 2 methods for this

i). is Alive() helps to determine whether a thread has finished.
ii). join () wait for a thread to finish.

Syntax for isAlive():
final boolean is Alive()
returns true if the thread upon which it is called is still running otherwise false.

Syntax for join():
final void join() throws InterruptedException
This method waits until the thread on which it is called terminates.

```
class MyThread implements Runnable {
String name;
public MyThread ( String threadName)
{ name = threadName;}
public void run()
{
try { for (int i = 3; i > 0; i--)
System.out.println (name + ":" + i); Thread.sleep(1000);
}catch (InterruptedException e)
{
System.out.println (name + "Interrupted");
System.out.println (name +* exiting ").
}
public class DemoJoin {
public static void main(String args[]) throws Exception (Thread t1 = new Thread(new
MyThread ("myThread"));
t1.start();
System.out.println("myThread status="t1.isAlive());
t1.join(); //main thread stops and waits for 11 thread to get terminated
System.out.println(\n Main Thread Exiting..in");
```

Thread Priorities
All Java threads have a priority in the range 1-10.
JVM selects to run a highest priority thread
3 constants defined in Thread class:
Thread.MAX_PRIORITY -10
Thread.MIN_PRIORITY -1
Thread.NORM_PRIORITY (Default Priority) 5
In case two threads have the same priority a FIFO ordering is followed.
JVM uses a preemptive, priority based scheduling algorithm.
A thread with a higher priority than the thread currently running enters the Runnable State. The lower priority thread is preempted and the higher priority thread is scheduled to run.

Example of priority of a Thread:

```
class TestMultiPriority extends Thread
{
public void run()
{
System.out.println("running thread name is:"+Thread.currentThread().getName());
System.out.println("running thread priority is:"+Thread.currentThread().getPriority());
}
public static void main(String args[]){
TestMultiPriority m1=new TestMultiPriority();
TestMultiPriority m2=new TestMultiPriority();
m1.setPriority (Thread.MIN_PRIORITY);
m2.setPriority (Thread.MAX_PRIORITY);
m1.start();
m2.start();}}
```

Using yield().
Causes the currently
running thread to yield() to any
other threads with the same
priority, waiting to be scheduled.

It can stop the currently executing thread

It will give a chance to other waiting threads of the same priority.

Synchronization

When more than one thread try to access a shared resource, we need to ensure that resource will be used by only one thread at a time. The process by which this is achieved is called synchronization.

Keyword : synchronized

Can apply on Methods & Blocks

Cannot apply on Variable & class

Thread Synchronization

There are two types of thread synchronization mutual exclusive and inter-thread communication.

Mutual Exclusive

Synchronized method.

Synchronized block.

Cooperation or Inter-thread Communication

wait(), notify(), notifyAll()

Synchronization Methods

- If you declare any method as synchronized, it is known as synchronized method.
- Synchronized method is used to lock an object for any shared resource.
- Synchronization in java is the capability to control the access of multiple threads to any shared resource.

```
synchronized void print() {
.....//code
```

The Synchronized Statement

Synchronized block can be used to perform synchronization on any specific resource of the method.

- Suppose you have 50 lines of code in your method, but you want to synchronize only 5 lines, you can use synchronized block.

*Synchronized block is used to lock an object for any shared resource

- Scope of synchronized block is smaller than the method.
- Synchronized block is used to lock an object for any shared resource.

General form:

```
synchronized (object reference expression) {
//code block
}
```

It is locking mechanism controlled by JVM

Thread Communication using notify(),wait() and notifyAll()

Inter-thread communication or Co-operation is all about allowing synchronized threads to communicate with each other.

Cooperation (Inter-thread communication) is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed.

Allowing synchronized threads to communicate with each other

- Implemented by following methods of Object class.

wait()
notify()
notifyAll()

wait()
It causes current thread to wait until either another thread invokes
It must be called from synchronized context i.e. from block or method. It means before wait() method is called
Syntax:
public final void wait()throws InterruptedException
public final void wait(long timeout)throws InterruptedException

notify()
Wakes up a single thread that is waiting on this object's monitor.
Syntax: public final void notify()

notifyAll()
Wakes up all threads that are waiting on this object's monitor
Syntax: public final void notifyAll()

suspending, Resuming and stopping Threads
a program used suspend() and resume(),
and stop() which are methods defined by
Thread.
The suspend() method of thread class puts the thread from running to waiting state
The suspended thread can be resumed using the resume() method
They have the form shown below:
final void suspend ( )
final void resume ( )
final void stop ()

Program with Synchronization methods and Synchronization blocks

```
class Table{
synchronized void printTable(int n){
for(int i=1;i<=5;i++)
{
try{
System.out.println(n*i);
Thread.sleep(400):
}catch(Exception e)
{
System.out.println(e);
}
class MyThread1 extends Thread{
Table t;
MyThread1(Table t)
{
this.t=t;}
public void run(){
t.printTable(5);
}
```

```
class MyThread2 extends Thread{
Table t;
My Thread2(Table t){
this.t=t;
}
public void run(){
t.printTable(100);}
class TestSynchronization1{
public static void main(String args]){
Table obj = new Table();//only one object
My Thread1 t1=new MyThread1 (obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
},
```

```
// Synchronized Block
class Table{
void printTable(int n){
synchronized(this){
for(int i=1;i<=5;i++){
System.out.println(n*i);
try{
Thread.sleep(400);
}catch(Exception e){
System.out.println(e);}
} } } }
//synchronized block
```

Enumerations

An enumeration is a list of named constants. In Java, enumerations define class types. That is, in Java, enumerations can have constructors methods and variables. An enumeration is created using the keyword enum. Following is an example -

```
enum Person
{
Married, Unmarried, Divorced,
Widowed
}
```

The identifiers like Married, Unmarried etc. are called as enumeration Constants. Each such constant is implicitly considered as a public static final member of Person.

After defining enumeration, we can create a variable of that type. Though enumeration is a class type, we need not use new keyword for variable creation, rather we can declare it just like any primitive data type.
For example,
Person p= Person.Married;
We can use == operator for comparing two enumeration variables. They can be used in . switch-case also. Printing an enumeration variable will print the constant name. That is,
System.out.println(p); // prints as Married

Consider the following program to illustrate working of enumerations:

```
enum Person
{
Married, Unmarried, Divorced, Widowed
}
class EnumDemo
{
public static void main(String args[])
{
Person p1;
p1=Person.Unmarried;
System.out.println("Value of p1 :" + p1);
Person p2= Person.Widowed;
 if(p1==p2){
System.out.println("p1 and p2 are same");
System.out.println("p1 and p2 are different");}
else
switch(p1)
{
case Married: System.out.println("p1 is Married"); break;
case Unmarried: System.out.println("p1 is Unmarried");break;
case Divorced: System.out.println("p1 is Divorced"); break;
case Widowed: System.out.println("p1 is Widowed"); break;
```

The values() and valueOf() Methods

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

public static enum-type|[] values()

public static enum-type valueOf(String str)

The values() method returns an array of enumeration constants. The valueOf() method returns the enumeration constant whose value corresponds to the string passed in str.

```
enum Person
{
Married, Unmarried, Divorced, Widowed
}
class EnumDemo
public static void main(String args[])
Person p:
System.out.println("Following are Person constants:");
Person all-Person.values();
for(Person p1:all)
System.out.println(p1);
p=Person.valueOf("Married");
System.out.println("p contains *+p);
```

Java Enumerations are Class Types

Java enumeration is a class type. That is, we can write constructors, add instance variables and methods, and even implement interfaces. It is important to understand that each enumeration constant is an object of its enumeration type. Thus, when you define a constructor for an enum, the constructor is called when each enumeration constant is created. Also, each enumeration constant has its own copy of any instance variables defined by the enumeration.

```
enum Level {
  LOW,
  MEDIUM,
  HIGH
}
public class Main {
  public static void main(String[] args) {
    Level myVar = Level.MEDIUM;

    switch(myVar) {
      case LOW:
        System.out.println("Low level");
        break;
      case MEDIUM:
         System.out.println("Medium level");
        break;
      case HIGH:
        System.out.println("High level");
        break;
    }
  }
}
for (Level myVar : Level.values()) {
  System.out.println(myVar);
}
```

```
enum Level {
  LOW,
  MEDIUM,
  HIGH
}

Class Level
{
String low=”LOW”;
String medium=”MEDIUM”;
String high=”HIGH”;
}

Level l=new Level();
```

```
enum Apple
{
Jonathan(10),GoldenDel(9), RedDel(12), Winesap(15), Cortland(8);
private int price;
Apple(int p)
{
price = p:
}
int getPrice()
{
return price:
class EnumDemo
{
public static void main(String args[])
{
Apple ap:
System.out.println("Winesap costs Apple Winesap.getPrice());
System.out.println("All apple prices:"}
for(Apple a: Apple values())
System.out.printin(a+ " costs + a getPrice() + "cents.");
}}
```

GoldenDel costs 9 cents. RedDel costs 12 cents. Winesap costs 15 cents. Cortland costs 8 cents.
Here, we have member variable price, a constructor and a member method. When the variable ap is declared in main(), the constructor for Apple is called once for each constant that is specified.
Although the preceding example contains only one constructor an enum can offer two or more overloaded forms, just as can any other class. Two restrictions that apply to enumerations:
an enumeration can't inherit another class.
an enum cannot be a superclass.

Enumerations Inherits Enum

All enumerations automatically inherited from java.lang.Enum. This class defines several methods that are available for use by all enumerations. We can obtain a value that indicates an enumeration constant's position in the list of constants. This is called its ordinal value, and it is retrieved by calling the ordinal() method, shown here:

final int ordinal()
It returns the ordinal value of the invoking constant. Ordinal values begin at zero. We can compare the ordinal value of two constants of the same enumeration by using the compare To() method.
It has this general form:
final int compareTo(enum-type e)

The usage will be -
e1.compare To(e2);
Here, e1 and e2 should be the enumeration constants belonging to same enum type. If the ordinal value of e1 is less than that of e2, then compareTo() will return a negative value. If two ordinal values are equal, the method will return zero. Otherwise, it will return a positive number.
We can compare for equality an enumeration constant with any other object by using equals(), which overrides the equals() method defined by Object.

```
enum Person
{
Married, Unmarried,
Divorced, Widowed
}
enum MStatus
{
Married, Divorced
}
```

```
class EnumDemo
(
public static void main(String args)
(
Person p1, p2, p3;
MStatus m =MStatus.Married;
System.out.println("Ordinal values are”);
for(Person p Person values())
{System.out.println(p+has a value +p ordinal”);
p1= Person.Married;p2 =Person.Divorced;p3=Person.Marned;
}
if(p1 compareTo(p2)<0)
System.out.println(p1+" comes before "p2);
else if(p1.compareTo(p2)==0)
System.out.println(p1+ is same as "+p2);
System.out.println(p1+" comes after "+p2);
if(p1.equals(p3))
System.out.println(“p1 & p3 are same");
else
if(p1==3)
System out println("p1 & p3 are same");
if(p1 equals(m))
System.out.println(p1 & mare same");
else
System.out.println("p1 & m are not same");
if(p1 m)
System.out.println(p1 & mare same");
```

```
byte-> Byte
class Byte
static Byte valueOf (byte b)
{
byte-> Byte;
}
Byte (byte b)
{
byte-> Byte;
}
}
class Primitive To Object
{
p.s.v.m (String args[])
{
byte x = 100;
Byte y = Byte valueOf (x);
byte x1 = 100;
Byte Y1 = new Byte (x1);
}
}
```

```
// Wrapper class Example: Primitive to Wrapper

import java.lang.*;

public class WrapperExample1
{
    public static void main(String args[])
    {
        //Converting int into Integer
        int a=20;
        Integer i=Integer.valueOf(a);//converting int into Integer
        Integer j=a;//autoboxing, now compiler will write Integer.valueOf(a) internally
        System.out.println(a+" "+i+" "+j);
    }
}
```

Type Wrappers

Java uses primitive types (also called simple types), such as int or double, to hold the basic data types supported by the language. Primitive types, rather than objects, are used for these quantities for the sake of performance. Using objects for these values would add an unacceptable overhead to even the simplest of calculations. Thus, the primitive types are not part of the object hierarchy, and they do not inherit Object. Despite the performance benefit offered by the primitive types, there are times when you will need an object representation. For example, you can't pass a primitive type by reference to a method. Also, many of the standard data structures implemented by Java operate on an object, which means that you can't use these data structures to store primitive types. To handle these (and other) situations, Java provides type wrappers, which are classes that encapsulate a primitive type within an object.

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

Primitive

booleanjava.lang.Boolean

java.lang.Byte

Byte

Char

Wrapper

E

Double java.lang.Double

Float java.lang.Float

java.lang.Integer

Long java.lang.Long

Short java.lang.Short

Void java.lang.Void

Int

java.lang.Character

Character Wrappers: Character is a wrapper around a char. The constructor for Character is Character(char ch)
Here, ch specifies the character that will be wrapped by the Character object being created. To obtain the char value contained in a Character object, call charValue(), shown here:
char charValue()
It returns the encapsulated character.
Boolean Wrappers: Boolean is a wrapper around boolean values. It defines these constructors: Boolean(boolean boolValue)
Boolean(String boolString)`

In the first version, boolValue must be either true or false. In the second version, if boolString contains the string "true" (in uppercase or lowercase), then the new Boolean object will be true. Otherwise, it will be false. To obtain a boolean value. from a Boolean object, use boolean booleanValue()

It return the boolean equivalent of the invoking object.
•The Numeric Type Wrappers: The most commonly used type wrappers are those that represent numeric values. All of the numeric type wrappers inherit the abstract class Number. Number declares methods that return the value of an object in each of the different number formats.
These methods are shown here:
byte byteValue() double doubleValue( ) float floatValue() int intValue()
long longValue() short shortValue()

For example, double Value() returns the value of an object as a double, floatValue() returns the value as a float, and so on. These methods are implemented by each of the numeric type wrappers.
All of the numeric type wrappers define constructors that allow an object to be constructed from a given value, or a string representation of that value. For example, here are the constructors defined for Integer:
Integer(int num) Integer(String str)
If str does not contain a valid numeric value, then a NumberFormatException is thrown. All of the type wrappers override toString(). It returns the human-readable form of the value contained within the wrapper. This allows you to output the value by passing a type wrapper object to println(), for example, without having to convert it into its primitive type.

```
class TypeWrap
{
public static void main(String args[])
Character ch=new Character('#');
System.out.println("Character is + ch.charValue());
Boolean b=new Boolean(true):
System.out.println("Boolean is b.booleanValue());
Boolean b1=new Boolean("false").
System.out.println("Boolean is " + b1.booleanValue());
Integer iOb-new Integer(12): //boxing
int i=iOb.intValue(); //unboxing
System.out.println(i + " is same as + Ob):
Integer a=new Integer("21");
int x=a.intValue():
System.out.println("x is "+x);
String s=Integer.toString(25);
System.out.println("s is "+s):
}}
```

Autoboxing

Java added two important features: autoboxing and auto-unboxing. Autoboxing is the process by which a primitive type is automatically encapsulated (boxed) into its equivalent type wrapper whenever an object of that type is needed. There is no need to explicitly construct an object. Auto-unboxing is the process by' which the value of a boxed object is automatically extracted (unboxed) from a type wrapper when its value is needed. There is no need to call a method such as intValue() or double Value().

The addition of autoboxing and auto-unboxing greatly streamlines the coding of several algorithms, removing the tedium of manually boxing and unboxing values. It also helps prevent errors. Moreover, it is very important to generics, which operates only on objects.
Finally, autoboxing makes working with the Collections Framework much easier.
With autoboxing it is no longer necessary to manually construct an object in order to wrap a primitive type. You need only assign that value to a type-wrapper reference. Java automatically constructs the object for you. For example, here is the modern way to construct an Integer object that has the value 100:

Integer iOb = 100; // autobox an int
Notice that no object is explicitly created through the use of new. Java handles this for you, automatically.
To unbox an object, simply assign that object reference to a primitive-type variable:
For example, to unbox iOb, you can use this line:
int i = iOb; // auto-unbox

Java handles the details for you.
Here is the preceding program rewritten to use autoboxing/unboxing:
// Demonstrate autoboxing/unboxing. class AutoBox { public static void main(String args[]) { Integer iOb = 100; // autobox an int int i = iOb; // auto-unbox System.out.println(i
+ " " + iOb); // displays 100 100

Autoboxing and Methods

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

```
// Autoboxing/unboxing
takes place with
// method parameters and return values.
class AutoBox2 {
// Take an Integer parameter and return
// an int value;
static int m(Integer v) {
return v; // auto-unbox to int
}
```

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

This program displays the following result:
In the program, notice that m( ) specifies an Integer parameter and returns an int result.
Inside main(), m() is passed the value 100. Because m() is expecting an Integer, this value is automatically boxed. Then, m( ) returns the int equivalent of its argument. This causes v to be auto-unboxed. Next, this int value is assigned to iOb in main(), which causes the int return value to be autoboxed.

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 prograrh

Autoboxing/unboxing occurs inside expressions. class
AutoBox3 {
public static void main(String args[]) {
Integer iOb, iOb2; int i;
iOb = 100;
System.out.println("Original value of iOb: " + iOb);
// The following automatically unboxes (Ob, performs the increment, and
then reboxesthe result back into 10b
++iOb;
System.out.println("After ++iOb: " + iOb);
// Here, iOb is unboxed, the expression is evaluated, and the result is
reboxed and assigned to 10b2.
iOb2 = iOb + (iOb / 3);
System.out.println("iOb2 after expression: " + iOb2);
// The same expression is evaluated, but the result is not reboxed.
i=iOb + (iOb / 3);
System.out.println("i after expression: " + i);

The output is shown here:
Original value of iOb: 100
After ++iOb: 101
iOb2 after expression: 134 i after expression: 134
In the program, pay special attention to this line:
++iOb;
This causes the value in iOb to be incremented. It works like this: iOb is unboxed, the value is incremented, and the result is reboxed
Auto-unboxing also allows you to mix different types of numeric objects in an expression. Once the values are unboxed, the standard type promotions and conversions are applied. For example, the following program is perfectly valid:

Autoboxing/Unboxing Helps Prevent Errors

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

```
/ An error produced by manual unboxing, class
UnboxingError {
public static void main(String args[]) {
Integer iOb = 1000; // autobox the value 1000
int i = iQb.byteValue(); // manually unbox as byte !!!
System.out.println(i); //does not display 1000 !
```

In general, because autoboxing always creates the proper object, and auto-unboxing always produces the proper value, there is no way for the process to produce the wrong type of object or value. In the rare instances where you want a type different than that produced by the automated process, you can still manually box and unbox values. Of course, the benefits of autoboxing/unboxing are lost. In general, new code should employ autoboxing/unboxing. It is the way that modern Java code will be written.

A Word of Warning
Now that Java includes autoboxing and auto-unboxing, some might be tempted to use objects such as Integer or Double exclusively, abandoning primitives altogether. For example, with autoboxing/unboxing it is possible to write code like this:

```
// A bad use of autoboxing/unboxing! Double a, b, c;'
a = 10.0;
b = 4.0;
c = Math.sqrt(a*a + b*b);
System.out.println("Hypotenuse is " + c);
```

In this example, objects of type Double hold values that are used to calculate the hypotenuse of a right triangle. Although this code is technically correct and does, in fact, work properly, it is a very bad use of autoboxing/unboxing. It is far less efficient than the equivalent code written using the primitive type double. The reason is that 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. Autoboxing/unboxing was not added to Java as a "back door" way of eliminating the primitive types.

Annotations (Metadata)
Beginning with JDK 5, a new facility was added to Java that enables you to embed supplemental information into a source file. This information, called an annotation, does not change the actions of a program. Thus, an annotation leaves the semantics of a program unchanged. 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. The term metadata is also used to refer to this feature, but the term annotation is the most descriptive and more commonly used.

Annotation Basics

An annotation is created through a mechanism based on the interface. Let's begin with an example. Here is the declaration for an annotation called MyAnno:

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

First, 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, as you will see.

An annotation cannot include an extends clause. However, 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. It overrides hashCode(), equals(), and toString(), which are defined by Object. It also specifies annotation Type( ), which returns a Class object that represents the invoking annotation.

Once you have declared an annotation, you can use it to annotate a declaration. 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 all cases, the annotation precedes the rest of the declaration.
When you apply an annotation, you give values to its members. For example, here is an example of MyAnno being applied to a method:

// Annotate a method.
@MyAnno(str = "Annotation Example", val = 100)
public static void myMeth() { // ...
This annotation is linked with the method myMeth( ). Look closely at the annotation syntax. 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

Specifying a Retention Policy

Before exploring annotations further, it is necessary to discuss annotation retention policies. A retention policy determines at what point an annotation is discarded. Java defines three such policies, which are encapsulated within the java.lang.annotation.RetentionPolicy enumeration. They are SOURCE, CLASS, and RUNTIME

An annotation with a retention policy of SOURCE is retained only in the source file and is discarded during compilation.

An annotation with a retention policy of CLASS is stored in the class file during compilation. However, it is not available through the JVM during run time.

An annotation with a retention policy of RUNTIME is stored in the .class file during compilation and is available through the JVM during run time. Thus, RUNTIME retention offers the greatest annotation persistence.
A retention policy is specified for an annotation by using one of Java's built-in annotations: @Retention. Its general form is shown here:
@Retention (retention-policy)
Here, retention-policy must be one of the previously discussed enumeration constants. If no retention policy is specified for an annotation, then the default policy of CLASS is used.

The following version of MyAnno uses @Retention to specify the RUNTIME retention policy. Thus, MyAnno will be available to the JVM during program execution.
@Retention(Retention Policy.RUNTIME) @interface.
MyAnno {
String str(); int val();
}
09 2017

Obtaining Annotations at Run Time by Use of Reflection

Although annotations are designed mostly for use by other development or deployment tools, if they specify a retention policy of RUNTIME, then they can be queried at run time by any Java program through the use of reflection. Reflection is the feature that enables information about a class to be obtained at run time.. The reflection API is contained in the java.lang.reflect package. There are a number of ways to use reflection, and we won't examine them all here. We will, however, walk through a few examples that apply to annotations.

The first step to using reflection is to obtain a Class object that represents the class whose annotations you want to obtain. Class is one of Java's built-in classes and is defined in java.lang. It is described in detail in Part II. There are various ways to obtain a Class object. One of the easiest is to call getClass(), which is a method defined by Object. Its general form is shown here:
final Class getClass()

It returns the Class object that represents the invoking object. (getClass() and several other reflection-related methods make use of the generics feature. However, because generics are not discussed until Chapter 14, these methods are shown and used here in their raw form. As a result, you will see a warning • message when you compile the following programs. In Chapter 14, you will learn about generics in detail.)

After you have obtained a Class object, you can use its methods to obtain information about the various items declared by the class, including its annotations. If you want to obtain the annotations associated with a specific item declared within a class, you must first obtain an object that represents that item. For example, Class supplies (among others) the getMethod( ), getField( ), and getConstructor( ) methods, which obtain information about a method, field, and constructor, respectively. These methods return objects of type Method, Field, and Constructor.

To understand the process, let's work through an example that obtains the annotations associated with a method. To do this, you first obtain a Class object that represents the class, and then call getMethod() on that Class object, specifying the name of the method. getMethod() has this general form: Method getMethod(String methName, Class ... param Types)

The name of the method is passed in methName. If the method has arguments, then Class objects representing those types must also be specified by param Types. Notice that param Types is a varargs parameter. This means that you can specify as many parameter types as needed, including zero.
getMethod() returns a Method object that represents the method. If the method can't be found, NoSuchMethodException is thrown.
From a Class, Method, Field, or Constructor object. you can obtain a specific annotation associated with that object by calling getAnnotation(). Its general form is shown here:
Annotation getAnnotation (Class anno Type)

Here, anno Type is a Class object that represents the annotation in which you are interested. The method returns a reference to the annotation. Using this reference, you can obtain the values associated with the annotation's members. The method returns null if the annotation is not found, which will be the case if the annotation does not have RUNTIME retention. Here is a program that assembles all of the pieces shown earlier and uses reflection to display the annotation associated with a method.

The Annotated Element Interface

The methods getAnnotation() and getAnnotations() used by the preceding examples are defined by the Annotated Element interface, which is defined in java.lang.reflect. This interface supports reflection for annotations and is implemented by the classes Method, Field, Constructor, Class, and Package.

In addition to getAnnotation() and getAnnotations(), Annotated Element defines two other methods. The first is getDeclaredAnnotations(), which has this general form:

```
Annotation[] getDeclaredAnnotations()
```

It returns all non-inherited annotations present in the invoking object. The second is isAnnotation Present(), which has this general form:
boolean isAnnotation Present(Class annoType)
It returns true if the annotation specified by anno Type is associated with the invoking object. It returns false otherwise.

Using Default Values

You can give annotation members default values that will be used if no value is specified when the annotation is applied. A default value is specified by adding a default clause to a member's declaration. It has this general form:

type member() default value;

Here, value must be of a type compatible with type.

Here is @MyAnno rewritten to include default values:

```
// An annotation type declaration that includes defaults.
@Retention (Retention Policy.RUNTIME)
@interface MyAnno {
String str() default "Testing"; int val() default 9000;
```

Marker Annotations

A marker annotation is a special kind of annotation that contains no members. Its sole purpose is to mark a declaration. Thus, its presence as an annotation is sufficient. The best way to determine if a marker annotation is present is to use the method isAnnotation Present(), which is a defined by the Annotated Element interface.

Single-Member Annotations

A single-member annotation contains only one member. It works like a normal annotation except that it allows a shorthand form of specifying the value of the member. When only one member is present, you can simply specify the value for that member when the annotation is applied-you don't need to specify the name of the member. However, in order to use this shorthand, the name of the member must be value.

The Built-In Annotations

Java defines many built-in annotations. Most are specialized, but seven are general purpose. Of these, four are imported from java.lang.annotation: @Retention, @Documented, @Target, and @Inherited. Three-@Override, @Deprecated, and @SuppressWarnings are included in java.lang. Each is described here.

@Retention

@Retention is designed to be used only as an annotation to another annotation.

@Documented
The @Documented annotation is a marker interface 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
The @Target annotation specifies the types of declarations 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 from the ElementType enumeration. This argument specifies the types of declarations to which the annotation can be applied. The constants are shown here along with the type of declaration to which they correspond.

You can specify one or more of these values in a @Target annotation. To specify multiple values, you must specify them within a braces-delimited list. For example, to specify that an annotation applies only to
fields and local variables, you can use this @Target annotation:

```
@Target( { ElementType.FIELD,
ElementType.LOCAL_VARIABLE })
```

@Inherited

@Inherited is a marker annotation that can be used only on another annotation declaration. Furthermore, it affects only annotations that will be used on class declarations. @Inherited causes the annotation for a superclass to be inherited by a subclass. Therefore, when a request for a specific annotation is made to the subclass, if that annotation is not present in the subclass, then its superclass is checked. If that annotation is present in the superclass, and if it is annotated with @Inherited, then that annotation will be returned.

@Override

@Override is a marker annotation that can be used only on methods. 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.

@Deprecated

@Deprecated is a marker annotation. It indicates that a declaration is obsolete and has been replaced by a newer form.

@SuppressWarnings
@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. This annotation can
be applied to any type of declaration.

Some Restrictions

There are a number of restrictions that apply to annotation declarations. First, no annotation can inherit another. Second, all methods declared by an annotation must be without parameters. Furthermore, they must return one of the following:

A primitive type, such as int or double
An object of type String or Class
An enum type
Another annotation type
An array of one of the preceding types

Annotations cannot be generic. In other words, they cannot take type parameters. Finally, annotation methods cannot specify a throws clause.

Context switch:
Context switch is the OS switch from one running thread to the next
The rules to determine a context switch:
1.A thread can voluntarily relinquish control.
This is done by explicitly yielding, sleeping, or blocking on pending I/O. In this scenario, all other threads are examined, and the highest-priority thread that is ready to run is given the CPU.
2.A thread can be preempted by a higher-priority thread
As soon as a higher-priority thread wants to run, it pre-empts the lower priority thread. This is called preemptive multitasking.