**Week 5 – Threads and Java Tools**

**Learning Outcomes**

By the end this lesson, students will:

* Define “thread” and “multi-threaded application” and explain why they are important to modern software development.
* Identify the parts of a thread’s life – in other words, its states.
* Implement a thread by extending Java’s Thread class.
* Implement a thread using Java’s runnable interface.
* Explain the relationship between the number of cores a processor has, the number of threads an application uses, and the percentage utilization of the processor.
* Explain why software testing is important for modern applications.
* Define various terms related to the process of software testing.
* Distinguish between the various kinds of testing: unit testing, integration testing, and performance testing.
* Compare testing state with testing behavior.
* Explain the purpose of annotations in Java source code.
* Explain the purpose and syntax of the various assert statements junit supports.
* Install junit onto your machine and make it available in a Java project you write.
* Write and execute a junit test for a simple application you write.
* Write javadoc comments that describe the purpose, inputs, and outputs of classes and their components.
* Build javadoc documentation using Java's javadoc command-line tool.
* Identify reasons to use a Java package.
* Explain what a Java package is in terms of both how it is used and how it is stored.
* Compile a class from the command line so that it belongs to a particular package.
* Import a class you write that belongs to a particular package.
* Identify reasons for using jar files to group together related java classes.
* Create a jar file that stores the contents of a particular package.
* Provide access to the classes you've added to a jar file in an application you write.

We will achieve these learning outcomes by using the various tools that are identified and described in this lesson.

**Threading**

Modern central processing units (CPUs) are made up of *cores*. A core is like a mini-processor that works with its fellow cores to perform the work that applications request of the CPU. In the old days, a CPU had just one core, a single channel through which all requests would pass. Today, though, with multiple cores, a CPU can pay attention to and do many things at once. It’s like you when you listen to the radio and drive at the same time. Whenever you do multiple things at once, you are acting as a multi-core processor.

This architecture, in turn, allows today’s applications to perform multiple tasks at once. This ability is called multitasking. Multitasking enables an application to perform multiple tasks at once, such as updating the screen and listening to user requests simultaneously. An application that can do this is called multi-threaded, because each separate task that seems to be done simultaneously is handled through something called a thread.

A thread is concurrent request for the CPU’s attention. Multiple threads can operate alongside each other. The CPU serves each thread, deciding, on the fly, which of its cores are going to do each thread’s work.

Depending on how many cores the CPU has, there may be a one-to-one correspondence between threads and cores. For example, if you have a 4-core (i.e. quad-core) CPU, and you design your application to use four threads, each thread will be performed on one core. This will make maximum use of the CPU. If you monitor the application’s CPU usage on you computer using the Windows task manager, for example, you will see 100% usage, because all four of its cores would be used. If that same application didn’t use multiple threads but instead did everything in the main thread (the one that each application claims from the operating system by default for its use), the your application would use no more than 25% of the CPU’s power, because it would exercise only one of its four cores.

In Java, a thread is realized as a class. That class could be designed either to extend *Thread*, a built-in class in Java’s java.lang package (remember, java.lang is the class that comes for free without your having to import it), or as implementing the interface *Runnable*, which also comes from java.lang. Both approaches are valid. The “implementing Runnable” approach is nice when you want to create a class to be a Thread that must extend or descend from some other class. The “implementing Runnable” approach becomes a little more complicated when you need to manage data in the implementing class that must be accessible as the Thread operates. We’ll look at both approaches in a coming example.

*The “extends Thread” approach*

One way to create a Thread is to create a class that extends Thread. When you do that, you should declare a constructor that, at the very least, takes in a String that represents the Thread’s name. You also must implement the run() function, in which you place all the logic the Thread is supposed to implement. This overrides the base class’s run() function.

public class MyThread extends Thread {

public MyThread(String name) {

super(name);

other stuff you need to do to initialize private data members

}

public void run() {

the logic the class is supposed to perform

}

}

When you want to create a Thread, then, you do this:

MyThread th = new MyThread(name);

And you tell the th object to start()

th.start();

This, in turn, calls the th object’s run() function, so the work of the thread will be done.

Usually, you’ll want to have multiple Thread objects running at once. One way to do this is to stick them in an ArrayList:

ArrayList<MyThread> threads = new ArrayList<MyThread>();

threads.add(new MyThread(“firstthread”));

threads.add(new MyThread(“secondthread”));

threads.add(new MyThread(“thirdthread”));

You could then tell all the threads to start:

for (MyThread th : threads) {

th.start();

}

You could also tell all the threads to join up as a team that won’t move on to the rest of the program until all of them in the team have completed their work. That is what the join() function does:

for (MyThread th: threads) {

try {

th.join();

System.out.printlin(th.getName() + “ completed its work”);

} catch (Exception ex) {

System.out.println(“Something bad happened.”);

}

}

th.join() will join th to the set of running threads. What happens is that we won’t move on outside of the try block until all the threads that have been joined to the team have completed their work. However, as each individual thread completes its work, we will move to the System.out.println(th.getName() + “completed its work”); line, so that we will be able to see the order in which the threads finish. But we won’t leave the try..catch until everyone is done.

*The “implements Runnable” approach*

This approach is slightly different. Instead of extending Thread, you create a target *Runnable* object that a vanilla Thread object will coordinate. A *Runnable* object is one that has a run() function, just like a Thread, whose life is managed by a Thread.

The Runnable interface requires you to implement *public void run()*. The run() function is where you specify the meat-and-potatoes of the thread: what it is supposed to do. So, you would declare the class like so:

class MyRunnable implements Runnable {

public void run() {

//put what the thread is supposed to do here

}

}

If you define a Runnable class like this, you can then “thread-ify” it by creating a Thread object to wrap around it. For example, if I have created a Runnable class called MyRunnable such as above, I can create an actual instance of MyRunnable like so:

MyRunnable mine = new MyRunnable();

I can then create a Thread object to wrap around it:

Thread thrd = new Thread(mine, “name of thread”);

The Thread class, again, is defined in java.lang. It is what knows how to negotiate with the operating system for the CPU’s attention. When we want to start the thread, we tell it to start:

thrd.start();

This, in turn, behind the scenes, calls the MyRunnable class’s run() function to do the work of the thread. Agains, Thread.start() calls MyRunnable.run() automatically.

Usually, we will want to have multiple Thread objects running simultaneously. We might have an arraylist of them.

ArrayList<Thread> threads = new ArrayList<Thread>();

for (int i = 0; i < numThreads; i++) {

threads.add(new Thread(new MyRunnable(), “Thread-” + i));

}

We could then have a separate for loop to launch the threads:

for (Thread t : threads) {

t.start();

}

Once they’ve all been started, we can tell each of them to join each other as a group of threads that will block the rest of the program from continuing until all the threads that are running finish their work:

for (Thread t : threads) {

try {

t.join();

} catch (Exception ex) {

}

}

The effect of this for loop and try..catch is that the program won’t continue beyond that try..catch until all the running threads have finished their work. We could actually put code after t.join() within the for loop to report something as each thread t finishes, like so:

for (Thread t : threads) {

try {

t.join();

System.out.println(t.getName() + “ finished its work.”);

} catch (Exception ex) {

System.out.println(“The thread could not join for some reason.”);

}

}

After each thread completes its work, the message “… finished its work.” will be printed. But the program won’t proceed beyond the try..catch and for loop until all the threads that have “joined” are done.

**Example – extending Thread:** In this example, we use multiple threads to determine the prime numbers in a range of numbers. We implement the threads by extending the Thread class.

import java.util.ArrayList;

class PrimeThread extends Thread {

private int startNum;

private int endNum;

private ArrayList<Integer> primes;

public PrimeThread(String name, int stNum, int eNum) {

super(name);

setStartNum(stNum);

endNum = eNum;

primes = new ArrayList<Integer>();

}

public ArrayList<Integer> getPrimes() {

return primes;

}

public void setStartNum(int val) {

if (val % 2 == 0) {

startNum = val + 1;

} else {

startNum = val;

}

}

public boolean isPrime(int num) {

int cap;

if (num == 1) {

return false;

} else if (num %2 == 0) {

return false;

} else {

cap = (int)(Math.sqrt(num));

for (int i = 3; i <= cap; i=i+2) {

if (num % i == 0) {

return false;

}

}

return true;

}

}

public void run() {

primes = new ArrayList<Integer>();

for (int i = startNum; i <= endNum; i = i + 2) {

if (isPrime(i)) {

primes.add(i);

System.out.println(i);

}

}

}

}

public class PrimeThreadExample {

public static void main(String[] args) {

int threadCount = 2;

ArrayList<PrimeThread> primeThreads = new ArrayList<PrimeThread>();

for (int i = 0; i < threadCount; i++) {

primeThreads.add(new PrimeThread("Thread-"+i,10000000\*i,

10000000\*(i+1)));

}

for (PrimeThread th : primeThreads) {

th.start();

}

for (PrimeThread th : primeThreads) {

try {

th.join();

System.out.println("End of " + th.getName());

} catch (Exception ex) {

}

}

System.out.println();

System.out.println("Here is the full list of primes ...");

ArrayList<Integer> primes;

for (PrimeThread th : primeThreads) {

primes = th.getPrimes();

for (int i : primes) {

System.out.println(i);

}

}

}

}

**Example – implementing Runnable:** In this version, we use multiple threads to determine the prime numbers in a range of numbers, but we implement it using the Runnable interface.

import java.util.ArrayList;

class PrimeThread implements Runnable {

private int startNum;

private int endNum;

private String name;

private ArrayList<Integer> primes;

public PrimeThread(String name, int stNum, int eNum) {

setStartNum(stNum);

endNum = eNum;

this.name = name;

primes = new ArrayList<Integer>();

}

public ArrayList<Integer> getPrimes() {

return primes;

}

public void setStartNum(int val) {

if (val % 2 == 0) {

startNum = val + 1;

} else {

startNum = val;

}

}

public boolean isPrime(int num) {

int cap;

if (num == 1) {

return false;

} else if (num %2 == 0) {

return false;

} else {

cap = (int)(Math.sqrt(num));

for (int i = 3; i <= cap; i=i+2) {

if (num % i == 0) {

return false;

}

}

return true;

}

}

public void run() {

primes = new ArrayList<Integer>();

for (int i = startNum; i <= endNum; i = i + 2) {

if (isPrime(i)) {

primes.add(i);

System.out.println(i);

}

}

}

}

public class PrimeThreadTest {

public static void main(String[] args) {

int threadCount = 2;

ArrayList<PrimeThread> primeThreads = new ArrayList<PrimeThread>();

ArrayList<Thread> threads = new ArrayList<Thread>();

PrimeThread prTh;

for (int i = 0; i < threadCount; i++) {

prTh = new PrimeThread("Thread-"+i,10000000\*i, 10000000\*(i+1));

primeThreads.add(prTh);

threads.add(new Thread(prTh,"Thread-"+i));

}

for (Thread t : threads) {

t.start();

}

for (Thread t : threads) {

try {

t.join();

System.out.println("End of " + t.getName());

} catch (Exception ex) {

}

}

System.out.println();

System.out.println("Here is the full list of primes ...");

ArrayList<Integer> primes;

for (PrimeThread pt : primeThreads) {

primes = pt.getPrimes();

for (int i : primes) {

System.out.println(i);

}

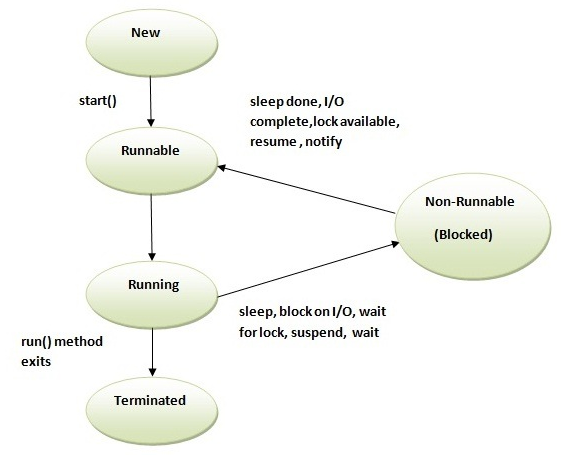
}

}

}

**The Life Cycle of a Thread**

A thread can pass through several different states during its life. This is shown here, which comes from http://www.javatpoint.com/life-cycle-of-a-thread. Notice that start() actually puts a thread in the *runnable* state, but it is up to the thread scheduler, a Java feature that negotiates with the operating system, to actually start running it. We can exit the running state by calling the thread’s sleep() or wait() functions, or by having other threads interrupt it by establishing a lock on a particular object. Establishing locks, which is often necessary when there is a shared memory resource or file that must be coordinated among multiple threads, is beyond what we want to cover here.



**Software Testing Using Unit Tests**

We learned that software suffers from three major problems:

* It is immobile
* It is rigid
* It is fragile

Basically, software is hard to change and hard to adapt to new situations. When you do try to change or adapt it, it breaks. Bugs pop up, and then we have to make more changes, which break more things that used to work.

So, testing is important to keep software running right. The testing has to be continuous and rigorous and comprehensive. That can be a daunting task unless you’re able to automate it to some degree.

The most popular approach for meeting this need is unit testing. We now will learn how to do unit testing in Java. Java makes it easier thanks to a class called JUnit.

**What is a software test?**

A software test is a piece of software that executes another piece of software (the one being tested) and verifies that that code behaves correctly and produces the expected results. The wider the coverage or comprehensiveness of the testing, the more confidence you can have that it is working correctly.

**Software tests generally test two things:**

* Behavior test: does the software execute the expected sequence of instructions?
* State test: does the software produce the expected results?

**Terminology**

We now consider several terms related to testing.

**Application / Code under test**

The application or code we are testing

**Test Fixture**

The precondition or value for which the test is conducted. For example, it could be the input values the feature you are testing will work with to produce results. You’ll compare the results you receive with what you’d expect to receive for that input – i.e. for that test fixture.

**Unit Test**

A piece of code that executes specific functionality in the code under test and asserts a certain behavior or state. These usually target a small unit of code like a method or class. In other words, they are local. When writing one, you try to eliminate as many wrinkles and external dependencies as you can. Unit tests are not good at testing complex user interaction or component interactions. For those, you’d want to use an Integration Test instead, which are not as highly automatable as unit tests are.

We are going to focus on unit tests in this course.

**Integration (or Functional) Test**

Another kind of test. These test whether the functionality of the entire application works as it should, or that the entire application behaves as expected.

**Performance Test**

Still another kind of test. These benchmark the performance of an application, especially under high load.

**Behavior vs. State Testing**

Behavior testing focuses on verifying that the correct functions were called with the correct parameters. In other words, are the various pieces interacting correctly? State testing focuses on the results of those calls. We will focus on state testing.

**What should be tested?**

You usually don’t test trivial things like getter and setter functions. Instead, test, functions that perform critical calculations on which other parts of your application depend.

**Where should test code go?**

Usually, test code should be put in a separate project so as not to pollute the main code you will distribute in production. However, to keep things simple for us in this class, we will put the test code in the same project.

**How are tests implemented in Java?**

How tests are implemented varies from language to language. In Java, however, there is a well-established testing framework called JUnit. That is what we will learn here.

JUnit is available at <http://junit.org/>. You have to download both junit.jar and hamcrest-core.jar. The current versions are junit-4.12.jar are hamcrest-core-1.3.jar.

Then, how you integrate them depends on whether you are using Eclipse or doing things from the command line.

**JUnit from the command line**

For the command line approach, when you run the test, you need to specify the classpath for the java utility. The classpath is where java will seek out classes your code depends on to run. Here are the full instructions for setting up your environment.

<http://www.tutorialspoint.com/junit/junit_environment_setup.htm>

For example, on my mac, I had to do this:

export JAVA\_HOME=/Library/Java/Home

export JUNIT\_HOME=/Library/JUNIT

export CLASSPATH=.:$JUNIT\_HOME/junit4.12.jar:$JUNIT\_HOME/hamcrest-core-1.3.jar

This will give you access to the classes that comprise the JUnit framework.

Then you have to have this piece of code to coordinate the tests.

import org.junit.runner.JUnitCore;

import org.junit.runner.Result;

import org.junit.runner.notification.Failure;

public class TestRunner {

public static void main(String[] args) {

Result result = JUnitCore.runClasses(UnitTests.class);

for (Failure failure : result.getFailures()) {

System.out.println(failure.toString());

}

System.out.println("Passed? " + result.wasSuccessful());

}

}

This file, TestRunner.java, performs the JUnit test on the functions contained in UnitTests.class. It will report the failures to the screen. At the end, it will print whether the software under test passed all the unit tests prescribed in the file UnitTests.

**How to create the unit tests**

Include these import statements:

import org.junit.Test;

import static org.junit.Assert;

Name the class that does the testing the same as the class whose code you are testing but with the word “Test” at the end of its name.

Name the functions that comprise the testing class after the function being tested and with “should\_\_\_\_\_” at the end of its name to indicate what should be true about the results of running that function.

Preface functions with annotations:

@Test

identifies the method as a test method (remember, method and function are synonyms)

@Test(expected = Exception.class)

fails if the method does not throw the named exception

@Test(timeout=100)

fails if the method takes longer than 100 milliseconds

@Before

public void method()

This method is executed before each test. It is used to prepare the test environment.

@After

public void method()

This method is executed after each test. It is used to clean up the test environment, including cleaning up expensive memory structures.

@BeforeClass

public static void method()

This method is executed once, before any test is done. It is used to do time-intensive tasks before any test is done.

@AfterClass

public static void method()

This method is excuted once after all tests have finished.

Include in your functions various assert statements. The JUnit Assert class has several static methods that throw exceptions (specifically, AssertionException objects) when the assert you are testing fails.

assertTrue(boolean condition)

assertFalse(boolean condition)

assertEquals(expected, actual)

assertEquals(expected, actual, tolerance), where tolerance is the number of decimals that must be the same

assertArrayEquals(expected, actual)

assertNull(object) checks that the object is null

assertNotNull(object)

assertSame(expected, actual) – checks to see if they correspond to the same object

assertNotSame(expected, actual)

assertThat(object, matcher), where you can write your own matcher class to test some more complicated condition you want to assert. (see http://tutorials.jenkov.com/java-unit-testing/matchers.html, for example)

**Running JUnit Test from the command line:**

With the classpath set as shown previously, compile all the functions:

javac SoftwareUnderTest.java UnitTests.java TestRunner.java

Then run TestRunner:

java TestRunner

You’ll then see the results, which will include failures and the final determination on whether the code passed the tests.

**Running JUnit Tests in Eclipse**

If you’re using Eclipse, you must put them in the build path for your project by right-clicking on the project in the project explorer, choosing properties, selecting Build Path, selecting Libraries, and clicking the “Add External JARs” button. See <https://www.youtube.com/watch?v=Dk0Lz1riYe4> for a demo.

The Eclipse IDE provides support for running your tests interactively. To run a test, select the class from the Project Explorer, right-click, and select Run As -> JUnit test. This will execute all the test methods in the class.

**Example**

In this example, we will demonstrate how to create and perform a unit test for a very simple application. Our application under test is called ShapeCalc, and it is a menu-driven program that calculates areas and perimeters for rectangles and circles.

Here is the code to be tested – the software under test. We are calling it ShapeCalc.java.

**import** java.util.Scanner;

**public** **class** ShapeCalc {

**public** **double** calcCircleArea(**double** rad) {

**return** Math.***PI*** \* rad \* rad;

}

**public** **double** calcCircleCirc(**double** rad) {

**return** 2 \* Math.***PI*** \* rad;

}

**public** **double** calcRectArea(**double** wid, **double** len) {

**return** wid \* len;

}

**public** **double** calcRectPerim(**double** wid, **double** len) {

**return** 2 \* (wid + len);

}

**public** **void** showMenu() {

System.***out***.println("Menu");

System.***out***.println("1. Circles");

System.***out***.println("2. Rectangles");

System.***out***.print("Enter number of your choice: ");

}

**public** **void** execute() {

Scanner sc = **new** Scanner(System.***in***);

**int** choice;

**double** area;

**double** perim;

**double** radius;

**double** width;

**double** height;

**do** {

showMenu();

System.***out***.print("Enter you choice: ");

choice = sc.nextInt();

**switch** (choice) {

**case** 1:

System.***out***.print("Enter radius: ");

radius = sc.nextDouble();

area = calcCircleArea(radius);

perim = calcCircleCirc(radius);

System.***out***.printf("Area = %.2f\tCirc = %.2f\n",area,perim);

**break**;

**case** 2:

System.***out***.print("Enter width and height: ");

width = sc.nextDouble();

height = sc.nextDouble();

area = calcRectArea(width,height);

perim = calcRectPerim(width,height);

System.***out***.printf("Area = %.2f\tPerim = %.2f\n", area,perim);

**break**;

}

} **while** (choice != 3);

}

**public** **static** **void** main(String[] args) {

ShapeCalc me = **new** ShapeCalc();

me.execute();

}

}

Here are the unit tests. We are calling this file ShapeCalcTest.java.

**import** org.junit.Test;

**import** **static** org.junit.Assert.\*;

**public** **class** ShapeCalcTest {

@Test

**public** **void** calcCircleAreaShouldReturnCorrectValue() {

ShapeCalc scalc = **new** ShapeCalc();

*assertEquals*(78.54,scalc.calcCircleArea(5.0),0.01);

}

@Test

**public** **void** calcCircleCircShouldReturnCorrectValue() {

ShapeCalc scalc = **new** ShapeCalc();

*assertEquals*(31.42,scalc.calcCircleCirc(5.0),0.01);

}

@Test

**public** **void** calcRectAreaShouldReturnCorrectValue() {

ShapeCalc scalc = **new** ShapeCalc();

*assertEquals*(30,scalc.calcRectArea(5.0, 6.0),0.01);

}

@Test

**public** **void** calcRectPerimShouldReturnCorrectValue() {

ShapeCalc scalc = **new** ShapeCalc();

*assertEquals*(22,scalc.calcRectPerim(5.0, 6.0),0.01);

}

}

Here is the new TestRunner.java:

import org.junit.runner.JUnitCore;

import org.junit.runner.Result;

import org.junit.runner.notification.Failure;

public class TestRunner {

public static void main(String[] args) {

Result result = JUnitCore.runClasses(ShapeCalcTest.class);

for (Failure failure : result.getFailures()) {

System.out.println(failure.toString());

}

System.out.println("Passed? " + result.wasSuccessful());

}

}

You then run TestRunner:

java TestRunner

This is what will be produced if all the tests succeeded:

Passed? True

If we change one of the tests in ShapeCalcTest to apply the test incorrectly, or if we change the ShapeCalc code to introduce an error, we’ll get output like this:

calcCircleAreaShouldReturnCorrectValue(ShapeCalcTest): expected:<718.54> but was:<78.53981633974483>

Passed? false

**Packages and Organizing Your Code**

A package is a group of related classes. You have been dealing with packages throughout this course. The most fundamental aspects of the language are held in java.lang, a package you get for free without having to import. You’ve also used packages like javax.swing and java.awt.

You can create your own packages to group related classes together. In this unit, you’ll learn how.

First, advantages of packages include …

* You and other programmers can easily determine that these types are related.
* You and other programmers know where to find types that can provide certain kinds functions.
* The names of your types won't conflict with the type names in other packages because the package creates a new namespace.
* You can allow types within the package to have unrestricted access to one another yet still restrict access for types outside the package.

**How do you declare that class belongs to a package?** You include a statement at the top of your program that says package followed by the name of the package. The name of the package can be anything you want, but typically it is something that guarantees uniqueness, like your domain name in reverse:

package edu.lewisu.cs.whateveryouwanttocallthepackage;

When you do that, the .class file for what you are declaring is supposed to be put in the folder that corresponds to that package. In this case, it will be placed in edu/lewisu/cs/whateveryouwanttocallthepackage/

To make sure that that happens, you want to compile it as follows

javac –d . nameofclass.java

The “-d .” part indicates that the current directory is the root, and the package folders will be created underneath that root. If a class is supposed to belong to a package, **you must** include the –d compiler directive.

So, in this example, the .class file will be placed in

<current\_directory>/edu/lewisu/cs/whateveryouwanttocallthepackage/

One of the nice things about doing it this way is that you can distribute your .class files easily without revealing your source files.

Here’s an example. We will have a package called shapes, and we will declare a class called Circle that is supposed to belong to it:

package edu.lewisu.cs.shapes;

public class Circle {

private double radius;

public double getRadius() {

return radius;

}

public void setRadius(double rad) {

radius = Math.abs(rad);

}

public Circle() {

radius = 0;

}

public Circle(double rad) {

setRadius(rad);

}

public double area() {

return Math.PI \* radius \* radius;

}

}

Here is the CircleTest.java class that imports it:

import shapes.Circle;

public class CircleTest {

public static void main(String[] args) {

Circle c = new Circle(5);

System.out.printf("Area=%.2f\n",c.area());

}

}

To compile it from the command line, first compile Circle:

javac –d . Circle.java

The Circle.class file will be placed in the edu/lewisu/cs/shapes subfolder of the current directory.

Then compile CircleTest.java normally:

javac CircleTest.java

During compiling, because of the package statement, the program will look in the shapes subfolder for Circle.class.

**Grouping files using Jar Files**

Each package folder has a set of class files. You can distribute the class files as just one file using jar’s. jar stands for “java archive”. It is a way to group together multiple files so that they are easier to deploy.

To create a jar:

jar cf nameofjar.jar list of files most of which are class files

To extract a jar:

jar xf nameofjar.jar

The jar file can contain an entire directory, and even a hierarchy of subdirectories.

So, for example, you can build a jar file for our edu.lewisu.cs.shapes package:

jar cf shapes.jar edu

You can then add shapes.jar as an external library to the build path for a Java project in eclipse, like we did with JUnit.

For example, suppose shapes.jar is supposed to be the source for Circle.class. This is how I could compile and run CircleTest to grab Circle.class from shapes.jar:

javac –cp .:./shapes.jar ShapeTest.java

javac –cp .:./shapes.jar ShapeTest

The “-cp” stands for classpath, and it allows me to specify where classes should be grabbed from. The “.” means “current directory”. The “:” separates different parts of the classpath. The statement above means that the classpath includes the current directory and the shapes.jar file that is in the current directory. This will work on a mac or linux machine.

Note that, on Windows, the class path would be specified as .;.\shapes.jar. Again “.” means “current directory”, but, on Windows, the “;” is used to separate different pieces of the path instead of a colon, and a backslash is used to separate folder names instead of a forward slash.

**Javadoc comments**

Javadoc provides a way to document your code.

It, too, uses annotations

@author

@return

@param

These appear inside of quotes that begin with /\*\* and end with \*/.

To produce Javadoc, either use the Project >> Javadoc option in Eclipse, or generate it from the command line, like follows:

javadoc –d ./docs Shape.java

**/\*\***

**\***

**\* @author klumpra**

**\* Represents the base level of any kind of shape.**

**\*/**

**public abstract class Shape {**

**/\*\***

**\* The x position of this shape**

**\*/**

**private int x;**

**/\*\***

**\* The y position of this shape.**

**\*/**

**private int y;**

**/\*\***

**\* Creates a new shape with x = y = 0**

**\*/**

**public Shape() {**

**x = 0;**

**y = 0;**

**}**

**/\*\***

**\* Creates a new shape with x and y set to what was passed in.**

**\* @param x This shape's x**

**\* @param y This shape's y**

**\*/**

**public Shape(int x, int y) {**

**this.x = x;**

**this.y = y;**

**}**

**/\*\***

**\* Gets the x of this shape**

**\* @return this shape's x**

**\*/**

**public int getX() {**

**return x;**

**}**

**/\*\***

**\* Gets the y of this shape**

**\* @return this shape's y**

**\*/**

**public int getY() {**

**return y;**

**}**

**/\*\***

**\* Sets the x of this shape**

**\* @param x This shape's x**

**\*/**

**public void setX(int x) {**

**this.x = x;**

**}**

**/\*\***

**\* Sets the y of this shape**

**\* @param y This shape's y**

**\*/**

**public void setY(int y) {**

**this.y = y;**

**}**

**/\*\***

**\* Returns the string representation of the shape**

**\* including x and y values**

**\* @return a string of the format x=val, y=val**

**\*/**

**public String toString() {**

**return String.format("x=%d, y = %d",x,y);**

**}**

**public abstract double area();**

**public abstract double perim();**

**}**