CSE 11: Lecture 17

- Packages
- Package naming, and CLASSPATH
- Multithreaded programming
- Synchronized statements and synchronized methods

Final Exam will be...

Friday Dec 12 8:00am-10:30am

Location: here

Closed-book, closed-notes, no calculators. Bring picture ID!

Coverage: Everything...

A practice final is available online (PDF format)

Lecture notes are available online

Final review session will be: Lecture Thursday

Packages

- You know that
  - There are 4 levels of visibility for a method or a variable defined in a class: **`public`**, **`protected`**, **`private`**, and (if none of these is specified) package visibility
  - Package visibility for a method or a variable means: it is accessible from any method in any other class in the same package

- You also know that
  - If you don't do anything unusual, all the classes in the same directory are in the same "default" package

- Let's look at Java packages in a little more detail

Packages, cont'd

- You can think of a class as a nice way to group some data and functions together
- Packages are a higher level of software organization: Packages are a means for organizing related classes together
- In Java, classes in the same package can have special access privileges with respect to one another, and are typically designed to work together, taking advantage of that "package visibility"
- (This is somewhat similar to so-called "friendly visibility" in C++)
- If you do not have a package statement declaring package membership at the top of the .java source code file for a class, then that class is part of the unnamed, "default" package, together with other such classes in the same directory
- Now let's see how to create named packages

Package names

- We have seen some package names already, for example

```
java.lang
java.awt
java.awt.event
```

- These are part of the full names of classes found in these packages, for example

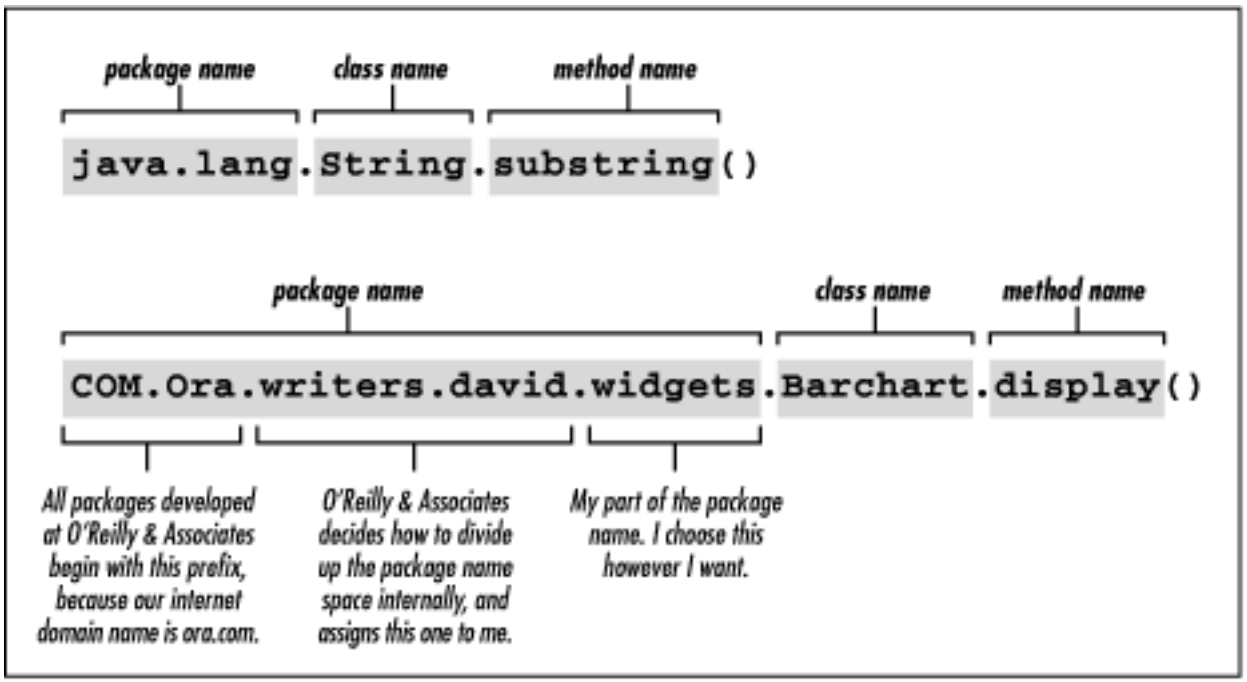
```
java.lang.String
java.awt.Container
java.awt.event.ActionEvent
```

- Rules for using class names in Java:
  - you do not need to use the full, package-qualified name of classes in the java.lang package
  - in a class definition, you do not need to use the package-qualifed name of classes in the same package
  - and for other packages, you do not need to use the full, package-qualified name if you import the class using an import statement

Package names, cont'd

- Package names are hierarchical: they correspond to the hierarchical directory structure on a machine, and to the Internet domain and host names for the machine
- On a machine, the package name is relative to paths on the CLASSPATH,
- CLASSPATH is a list of directories containing .class files; this list by default includes the current working directory, and the parent directory of the standard libraries
- So, for example:
  - the classes for the `java.lang` package would be in a directory `java/lang` under some directory on the CLASSPATH for your machine
  - the classes for a package `com.ora.writers.david.widgets` would be in a directory `writers/david/widgets` under some directory on the CLASSPATH on the machine `ora.com`
- Package naming is part of the beginning of a standard for Uniform Resource Identifiers (URIs) that will provide a persistent identifier for every downloadable object in the world: potentially trillions of them, forming the global infrastructure for all computing resources

Package names: a picture



Creating a package, and putting classes in it

- To create a package, you first create a directory with a certain pathname relative to a directory on the CLASSPATH (e.g. the current working directory):

```
% mkdir foo/bar
```

- Then put a package statement in each source code file in that directory that declares the contents of the file to be in the corresponding package:

```
package foo.bar;

public class MyClass {   // etc...
```

- The package statement must appear as the first statement (i.e., comments can come before it, but nothing else) in the file

- Now another program that wants to use classes in that package without using their fully qualified name can import them (all of them, with *; or individually):

```
import foo.bar.*;
```

Setting CLASSPATH

- CLASSPATH is the name of an “environment variable” maintained by the operating system
- You can check the value of CLASSPATH under Unix (most shells) with either of these commands:

```
printenv CLASSPATH
echo $CLASSPATH
```

- If CLASSPATH is undefined, Java will use the current working directory and the Java standard library directories as the CLASSPATH
- If CLASSPATH is defined, Java will use the Java standard library directories and the directories on the CLASSPATH only
- You can set the value of CLASSPATH under Unix with the **`setenv`** command, if you are using the csh shell
- The CLASSPATH variable should be a list of directory pathnames, separated by **`:`**’s ... Example: If you want to use the P6 classes without having to copy them to your own directory, you could do this:

```
setenv CLASSPATH .:~/../public/P6
```

Threads and multithreaded programming

- In CSE 11, the code you have written has been *single-threaded:* only a single sequence of statements is executed
- Even when using complicated branching and iteration, your code executes in a single sequence of instructions, i.e., a single thread
- (Other threads may have been running within the Java Virtual Machine: the garbage collector, the Swing event dispatcher, etc., but your code has been single-threaded)

- For some purposes, you might want instead to write a *multi-threaded* program:
  - When working over a network, you might be downloading data from several different machines at once; multithreading would permit doing this in parallel, instead of having to wait until the first download finishes before starting on the next
  - When doing an animation, it might be convenient to have each character interact with the others as its own thread of execution
  - etc.

- Java provides good support for multithreaded programming, which we'll now look at briefly

Multithreaded programming in Java

- The key to doing multithreaded programming in Java is the **`Thread`** class
- Here are the Thread methods we will consider:

```
public class Thread extends Object implements Runnable {

   public Thread() {...

   public static native void yield() {...

   public void run() {...

   public synchronized native void start() {...
```

- Now let's look at how to use these methods...

How to create a thread

- If you want to do multithreaded programming in Java, the basic way to do it is this:
  - Define classes that extend Thread, and that override Thread's run() method
  - Create instances of these classes, and call their start() methods
  - Calling the start() method of a Thread object does this:
    - It starts a new thread within the Java runtime environment.
    - This new thread executes the statements within the body of the Thread object's run() method.
    - The start() method returns as soon as it starts the thread; the thread keeps running until the run() method returns (or the Thread's stop() method is called, or the thread is interrupted, etc.)
  - Depending on whether the Java implementation uses preemptive or nonpreemptive thread scheduling, a Thread may have to call yield() in order to let other Threads execute

Multithreaded programming: an example

```
public class Ones extends Thread {
  public void run () {
    while(true) // just print a sequence of 1's
      System.out.print(1);
  }
}
-----------------------------------------------------------
public class Zeros extends Thread {
  public void run () {
    while(true)  // just print a sequence of 0's
      System.out.print(0);
  }
}
-----------------------------------------------------------
public class Test {
  public static void main( String args[] ) {
    Zeros z = new Zeros();
    Ones o = new Ones();
    o.start();
    z.start();
```

An example, cont'd

- Q: What happens when that multithreaded program runs?

  A: It depends on whether the Java implementation uses preemptive or nonpreemptive thread scheduling

- Preemptive scheduling: By choosing a maximum length of time a thread can continuously, a preemptive scheduling mechanism guarantees that no single thread uses more than its fair share of the processor. If a thread runs for that amount of time without yielding control to another thread, the scheduler preempts the thread and causes it to stop running so that another thread can run.

- Nonpreemptive scheduling (also sometimes called "cooperative multitasking"): A nonpreemptive scheduling mechanism cannot preempt threads. A nonpreemptive scheduler relies on the individual threads to yield control of the processor frequently, so that it can provide reasonable performance. A thread explicitly yields control by calling the Thread object's yield() method.

An example, cont'd

- So, with preemptive scheduling, the test program might output something like

```
11110001110000011111000111100000011111110000000011111000001110011
11111000001111111000000110010111100000011111110000001111100000001
11111110000000000011111111000011111110111111000000011111001000011111
11110011111011111100000001111000000011111111111  ...
```

- With nonpreemptive scheduling, the test program will output

```
11111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111  ...
```

- Nonpreemptive multithreading doesn't work well for this example!

- Some platforms may use nonpreemptive scheduling for Java threads. To get reasonable threadswitching on any platform, we make the threads call yield() periodically...

Multithreaded programming: a revised example

```
public class Ones extends Thread {
  public void run () {
    long count = 0;
    while(true) { // just print a sequence of 1's
       System.out.print(1);
       if(++count % 10 == 0) yield();  // yield every 10 steps
    }
  }
}
-----------------------------------------------------------
public class Zeros extends Thread {
  public void run () {
    long count = 0;
    while(true)  { // just print a sequence of 0's
        System.out.print(0);
       if(++count % 10 == 0) yield();  // yield every 10 steps
    }
  }
}
```

A revised example, cont'd

- Now, even with nonpreemptive scheduling, the test program will output

```
11111111110000000001111111111000000000011111111100000000011111111
11100000000011111111110000000000111111111000000000111111111000000
00000011111111110000000000111111111000000000111111111000000000011
11111111100000000011111111110000000001111111110000000011111111111
11000000000111111111100000000001111111110000000000 ...
```

Synchronization and locks

- Each Java virtual machine can support many threads of execution at once
- These threads independently execute code that operates on objects residing in a shared main memory
  - (This is the main difference between multithreading and multiprocessing: Processes do not share memory, but multiple threads within the same process do.)
- How to make sure that these independent actions will produce a nice result? Answer: *synchronization*
- To synchronize threads, every Java object has a *lock* associated with it
- Only one thread can acquire the lock for an object at any one time; no other thread can acquire the lock until the lock is released by the thread that has the lock
- *Synchronized statements and methods* are the way to acquire and release an object's lock

Synchronized statements

- A synchronized statement has the form

```
synchronized (<expr>) {
   <body>
}
```

- The synchronized statement has this semantics:
  - First, the the expression <expr> is evaluated. It must evaluate to a non-null object reference
  - Next, the lock on that object is acquired. If the lock is already owned by another thread, the current thread stops executing and waits for it to be released
  - Then, the statements in the body of the synchronized statement are executed
  - Finally, the lock is released.

- Note: nothing prevents another thread from modifying the “locked” object, if that other thread is not synchronized on it!

- Synchronization is essential for ensuring deterministic behavior in multithreaded programming, but the programmer has to understand when and how to use it...

Synchronized methods

- An instance method can be declared synchronized
- This is exactly the same as declaring that the body of the method is synchronized on the "this" object
- For example:

```
public synchronized void setTime(int hour, int min, int sec) {
   h = hour; m = min; s = sec;
}
```

is the same as

```
public void setTime(int hour, int min, int sec) {
   synchronized(this) { h = hour; m = min; s = sec; }}
```

- Without synchronization here, two threads simultaneously executing the setTime method of an object might put the object in an inconsistent state

- In general, methods that might not work correctly if executed by two threads simultaneously should be declared synchronized, or they should be documented as not being thread-safe

Next time

- ✔ Compile-time and run-time type checking
- ✔ Type checking and dynamic method binding
- ✔ Final review of CSE 11 topics