Applets

Until now, the Java we have written was designed to be used in applications—“standalone” programs that can be invoked from the command line (or from a menu).

As you probably know, one of Java’s strengths is its facility for defining applets that can run inside of Web browsers.

There are several differences between applications and applets:

- Applets are meant to be invoked by <applet> tags in HTML files.
- Java instantiates an applet in response to a request from a Web browser.
- When an applet is instantiated, Java calls the init method and then the start method of the new applet. There is no role for a main method.
- An applet is run in a pane within a Web-browser window. The pane’s size is determined by the <applet> tag. Thus, an applet should not call

What changes do we need to make to transform an application into an applet?

1. Import the applet package.

   ```java
   import java.applet.*;
   ```

2. Make the class extend Applet (or JApplet) rather than JFrame, and change its name appropriately.
3. Replace the main method with an init method. The init method should tell where to place the new applet in its surrounding container.

```java
public void init() {
    setLayout(new BorderLayout());
    add("Center", canvas);
}
```

To keep the same behavior, we need to specify a BorderLayout, because the default layout for applets is FlowLayout.

4. The event-handling code for closing the window can be eliminated, because applets have no close button.

**Invoking an applet**

An applet is invoked from a Web page using an HTML tag

```html
<APPLET [CODEBASE="..."]
    CODE="...
    WIDTH="...
    HEIGHT="...
    [ALT="...
    NAME="...
    ALIGN=left | right | top | ...
    VSPACE="...
    HSPACE="..."]
</APPLET>
```

Text for non-Java supporting browsers.

The APPLET element is the mechanism to invoke an applet.

- A browser that understands this element will ignore all contents of the APPLET element except the PARAM elements.
- Browsers that do not understand this element should ignore it and the PARAM elements, and instead process the content of the element.
Thus the contents are alternate HTML that is displayed if the applet is not invoked.

In the **APPLET** tag, three parameters are required:

- **CODE** is the name of the file that contains the compiled Applet subclass. This name is relative to the base URL of the applet, and cannot be an absolute URL. (Instead of **CODE**, an **OBJECT** tag may be given, pointing to a serialized applet.)

- **WIDTH** and **HEIGHT** give the initial width and height (in pixels) of the applet display area.

In addition, several parameters are optional.

- **CODEBASE** specifies the base URL of the applet. It is not required when the applet is located relative to the HTML document.

- **ALT** specifies parsed character data to be displayed if the browser understands the **APPLET** tag but can’t or won’t run it.

- **NAME** specifies a name for the applet instance, which allows applets on the same page to communicate with each other.

- **ALIGN** specifies the display alignment.

- **VSPACE** and **HSPACE** specify the reserved space around the applet (in pixels).

The **PARAM** element is a mechanism for passing general-purpose parameters to applets.

- **NAME** is the name of the parameter, and

- **VALUE** will be obtained by the applet with the `getParameter()` method.

For more information, see the Java tutorial [http://java.sun.com/docs/books/tutorial/getStarted/applet/](http://java.sun.com/docs/books/tutorial/getStarted/applet/).

**Example invocation:**
The class java.applet.Applet

All applets inherit from java.applet.Applet.

This class provides the necessary structure to coordinate with a Web Browser.

There are a number of key methods in java.applet.Applet. You will likely override some of them to provide interesting functionality for your applet.

- A zero-argument (default) constructor
- init()
- start()
- stop()
- destroy()

The lifecycle of an applet

1. When a document with an applet is opened,
   - an instance of the applet's controlling class is created (the default constructor is called),
   - the init() method is called to initialize the applet. and
• the `start()` method is called to start the applet.

2. The applet is "rendered" via a call to `repaint()`, as we mentioned last lecture.

3. When a document with an applet is no longer displayed (e.g., because the user leaves the page it is on), the browser invokes the `stop()` method.

4. When the user returns to the page, the applet is started again.

5. After the `stop()` method is called, the `destroy()` method may be called to return any resources that are being used.

Reloading a page

Reloading provides a cleanup sequence, followed by the loading sequence.

In this case the method sequence would be—

• `stop()`
• `destroy()`
• `Constructor()` (Create a new applet object.)
• `init()`
• `start()`

Here is an example of how these methods of `java.applet.Applet` can be overridden, simply to highlight when they are executed:

```java
import java.applet.Applet;
import java.awt.Label;

public class MethodDemo extends Applet {
    private Label label;
    private void init() {
        System.out.println("Applet.init()");
    }

    public void start() {
        System.out.println("Applet.start()");
        label = new Label("Hello, world!");
        add(label);
    }

    private void stop() {
```
System.out.println("Applet.stop()")
remove(label);
}

private void destroy() {
  System.out.println("Applet.destroy()"神通);  
}

Running this applet would display the following output on the screen:

Applet.init()
Applet.start()

The applet window itself would have _________________ displayed top and center.

Exiting the page with the applet would cause the following to be displayed:

Applet.stop()
Applet.destroy()

Concurrent in Java

Concurrency is becoming ubiquitous in applications because—

- Most applications are decomposed into multiple subsystems or modules.
- The goal of this partitioning is to leave the individual modules highly cohesive while achieving a very high degree of independence among the modules.
- Coupling can be further reduced by letting each module be temporally autonomous.
- Temporally autonomous modules can operate in parallel, if resources are available.

In practice, however, making modules independent and loosely coupled is advantageous even if they are run on the same processor.
In Java, a process may be composed of any number of threads. Roles for concurrent architectures: Java threads have many applications. Here are some examples.

- Different Java threads can manage multiple input streams in a single computation.
- Multiple threads aid in being able to insure certain services remain highly available, e.g., `inetd` spawns off a task to handle each incoming connection request. What’s the advantage of this?

- The Java garbage collector runs as a low-priority thread. What is the advantage of this?

When is concurrency inappropriate?

- When the granularity is too fine. What does this mean?

Sources of overhead:

- There is overhead associated with switching between threads (task switching).
- There are synchronization costs when two threads must communicate.

- Sequential dependencies are easier to enforce via a single thread of control.
- Multithreading raises issues of consistency and liveness.

How do threads differ from processes?

- Any flow of computation has a state.
• A computation also has an environment (the address space, protection domain, allocated resources).

A process has its own state and its own environment.

A thread has its own state, but only those aspects of the environment that are associated with control flow (e.g., stack and CPU registers).

**The thread class**

The class `java.lang.Thread` contains constructors for threads and methods for working with them.

Among the constructors are—

• the zero-parameter constructor `Thread()`, which constructs threads named `Thread1`, `Thread2`, etc.

• the one-parameter constructor `Thread(String)`, which constructs threads named after their parameter.

The main thing a thread can do is `run()`. Every thread-based class must define a method `run()`. (As we shall see, it actually overrides the `run()` method of the interface `Runnable`.)

Here is a diagram of the states a thread can be in:

After a thread is created, it must be `start()`ed.

This places it in the `ready` state.

Eventually, the system will assign a processor to a thread; when this happens, the highest-priority ready thread begins to execute.

A thread stops running because of one of six conditions.

• It goes back to the ready state if—
threads are timesliced and its quantum expires
it is interrupted by another process, or
it yields the processor to a waiting process.

- It waits on a particular object, say \texttt{obj}. (Note that \texttt{obj} is the receiver, not \texttt{this}.)

It then waits until another object wakes it up via an \texttt{obj.notify()} call (only one thread wakes up in this case), or via an \texttt{obj.notifyAll()} method (which wakes up all threads waiting on \texttt{obj}).

- It “goes to sleep” for \texttt{m} milliseconds by executing the \texttt{sleep(long \texttt{m})} method.

- It is suspended. Once it has been suspended, it may not execute until its \texttt{resume()} method is later called (obviously by another thread).

- It performs I/O. Once it has issued an I/O request, it may not execute until the I/O completes.

- It dies, usually by having its \texttt{run()} method complete.

It is also possible to \texttt{stop()} a thread, but this is not recommended. See http://java.sun.com/j2se/1.4/docs/guide/misc/threadPrimitiveDeprecation.html.

A dead thread will eventually have its memory reclaimed by the system.

**Thread priorities and scheduling:**

Every Java thread has a priority, somewhere between \texttt{Thread.MIN_PRIORITY} to \texttt{Thread.MAX_PRIORITY} (1 to 10).

When a new thread is created, it is assigned a default priority of \texttt{Thread.NORM_PRIORITY} (5).

Each new thread inherits the priority of the thread that created it.
In a non-preemptive system, a thread will run to completion before any other thread of equal priority gets a chance to execute.

But most new Java systems are preemptible; a thread will only run until its timeslice expires.

The Java scheduler runs the highest-priority thread at each time. If several threads are tied for highest priority, a preemptive scheduler will run them in round-robin fashion.

A thread’s priority can be changed with the `setPriority(int)` method.

A thread can call `yield()` to give other threads a chance to execute. What happens if—

- a thread tries to yield to a higher-priority thread?
  
- a thread tries to yield to a lower-priority thread?

Therefore, when a thread yields, it yields to another thread of equal priority.

Which systems is `yield()` more useful on, timesliced or non-timesliced?

Let us enumerate the ways that a higher-priority thread can become ready and preempt the running thread.

- 
  
- 
Note that the scheduler occasionally runs lower-priority threads when necessary to avoid starvation. Therefore, programs should not rely on thread priority for algorithm correctness.

**The Runnable interface and a sample thread**

As we said earlier, a thread implements the `Runnable` interface.

The `java.lang.Runnable` interface declares only a single method, `run()`.

Thus, any class can implement `Runnable` by just defining a `run()` method.

For example, here’s a `Runnable` class (an example of the service pattern) that just prints a message.

```java
public class SimpleMessagePrinter implements Runnable {
    protected String msg_;  // The msg. to print
    protected JTextArea txt_; // Where to print it

    public SimpleMessagePrinter(String m, JTextArea txt) {
        msg_ = m; txt_ = txt;
    }

    public void run() {
        txt_.append(msg_);
    }
}
```

*Note: The name of the method you define is `run()`, but the name of the method you call in the thread is `start()`.*

Calling `Thread.start()` causes `Runnable.run()` to run in a thread.
Two ways of creating threads

There are two ways to use the Thread class:

- As above, by implementing Runnable. Threads are created with the `Thread(Runnable)` constructor.
- By subclassing `Thread`, and overriding the `run()` method.

Implementing the Runnable interface

Here is an example of the first approach, a `Clock` class taken from the Java Tutorial, [http://java.sun.com/docs/books/tutorial/essential/threads/clock.html](http://java.sun.com/docs/books/tutorial/essential/threads/clock.html)

Clock implements the Runnable interface.

Clock then creates a thread and provides itself as an argument to the `Thread`'s constructor.

When created in this way, the `Thread` gets its run method from the object passed into the constructor. The code for doing this is shown in bold here:

```java
import java.awt.Graphics;
import java.util.*;
import java.text.DateFormat;
import java.applet.Applet;

public class Clock extends Applet implements Runnable {
    private Thread clockThread = null;
    public void start() {
        if (clockThread == null) {
            clockThread = new Thread(this, "Clock");
            clockThread.start();
        }
    }
}
```
public void run() {
    Thread myThread = Thread.currentThread();
    while (clockThread == myThread) {
        repaint();
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            // the VM doesn't want us to sleep
            // anymore, so get back to work
        }
    }
}

public void paint(Graphics g) {
    // get the time and convert it to a date
    Calendar cal = Calendar.getInstance();
    Date date = cal.getTime();
    // format it and display it
    DateFormat dateFormatter =
        DateFormat.getTimeInstance();
    g.drawString(dateFormatter.format(date), 5, 10);
} // overrides Applet's stop method, not Thread's

public void stop() {
    clockThread = null;
}

Which constructor is it that gives the Thread its run method?

The Clock applet’s run method loops until the browser asks it to stop. How often does it loop?

During each iteration of the loop, the clock repaints its display. The paint method figures out what time it is, formats it in a localized way, and displays it.

Subclassing Thread

The other way to customize a thread’s behavior is to subclass Thread (which implements Runnable) and override its empty run method.

Here is a SimpleThread class, which does just that:

    public class SimpleThread extends Thread {
        public SimpleThread(String str) {
            super(str);
        }
    }
public void run() {
    for (int i = 0; i < 10; i++) {
        System.out.println(i + " " + getName());
        try {
            sleep((long) (Math.random() * 1000));
        } catch (InterruptedException e) {}
    }
    System.out.println("DONE! " + getName());
}

The for loop in the run method iterates ten times.

In each iteration the method displays the iteration number and the ___________________, then sleeps for a random interval of up to 1 second.

After the loop has finished, the run method prints “DONE!” along with the name of the thread.

The TwoThreadsDemo class provides a main method that creates two SimpleThreads: one is named "Jamaica" and the other is named "Fiji".

public class TwoThreadsDemo {
    public static void main (String[] args) {
        new SimpleThread("Jamaica").start();
        new SimpleThread("Fiji").start();
    }
}

Which thread will finish first? Can you tell?

While either approach works, the Runnable approach has a few advantages that make it the strategy of choice in most cases:

• Any Java class can implement Runnable, and thus provide behavior for one or more threads.

• Subclassing Thread should be reserved for specializing behavior of a thread abstraction (rather than just providing distinct behavior for an instance of Thread.}
Controlling thread activities

The most commonly used Thread methods are those that cause threads not to do anything at all.

Normally a thread stops when it hits the end of its run() method. However, sometimes you need to stop activities for other reasons or in other senses.

There are several levels of “severity” for stopping activities in threads, including:

- **destroy()** stops and kills a thread without giving it or the Java runtime any chance to intervene, and without unlocking monitors (which we will cover in the next lecture). It is not recommended for routine use.

- **interrupt()** causes any kind of wait (sleep(), wait(), join(), etc.) to abort with an InterruptedException, which can be caught and dealt with in an application-specific way.

- **sleep(long)** causes objects to suspend and automatically resume in a controlled fashion, typically from within the running object itself.

- **wait()** (which is defined in class Object, so is available to all Java objects) causes a thread to suspend until a later notifyAll() or notify() is made (nearly always within some other method of the object invoking the wait).

- **yield()** yields control back to the scheduler, as described above, so it can run other threads before returning to run more of this one.

Let’s look at how our Clock example stopped its activities.

The exit condition for its run method is the exit condition for the while loop because there is no code after the while loop:

```java
while (clockThread == myThread) {
```
This condition indicates that the loop will exit when the currently executing thread is not equal to `clockThread`. When would this ever be the case?

```java
public void stop() { // applets' stop method
    clockThread = null;
}
```

If you revisit the page, the start method is called again and the clock starts up again with a new thread.

Even if you stop and start the applet faster than one iteration of the loop, `clockThread` will be a different thread than `myThread` and the loop will still terminate.

**Thread groups**

Every Java thread is created within a `ThreadGroup`.

By default, each new thread is created to be in the same group as its creator.

However, one can explicitly place a thread into a group:

```java
ThreadGroup myGroup = new ThreadGroup("My group");
Thread myThread = new Thread(myGroup, "My new thread");
```

You can find out what group a thread is in:

```java
ThreadGroup thisGroup = myThread.getThreadGroup();
```

Thread groups nest, so one whole `ThreadGroup` may be part of another.

Threads within a thread group share some access and control policies.

For example, it is not legal to stop a thread that is not in your group. This prevents, for example, a random applet from killing the main Java screen-display update loop thread.
The ThreadGroup class has some methods similar to those in class Thread, but apply them to groups of threads instead of individuals.

Among the few common uses of thread groups is to shut down all of the threads associated with some task by calling `getThreadGroup().stop()`.