1 Writing Applets

An *applet* is an application which is intended to be embedded inside another application (usually a web browser) rather than being executed on its own. The ability to write applets that will be executed within a web page can perhaps be singled out as the most important aspect of the Java platform.

Almost everything that we have learned so far about graphical user interface programming and custom graphics applies just as well to applets. However, there are a few differences between programs that are intended to run stand-alone and programs that are intended to run as applets.

2 The `Applet` Class

The `Applet` class (`java.applet.Applet`) is derived from `java.awt.Panel`. The exact class hierarchy is below:

```
java.lang.Object
  |
  +--java.awt.Component
        |
        +--java.awt.Container
              |
              +--java.awt.Panel
                    |
                    +--java.applet.Applet
```

Any applet must be a subclass of `Applet`. In other words, in order to write an applet that can be executed within a web page, you must write a class which extends `java.applet.Applet`.

The differences between a stand-alone application and an applet are as follows:

- An applet should *not* have a constructor.
- An applet should *not* have a `main()` method.
- An applet should perform initialization in the `init()` method.
- An applet should free any resources in the `destroy()` method.

3 Details of the `Applet` Class

The methods from the `Applet` class taken from the Java documentation, related to applet running:

```
public void init()

       Called by the browser or applet viewer to inform this
       applet that it has been loaded into the system.  It
       is always called before the first time that the start
       method is called.
```

A subclass of Applet should override this method if it has initialization to perform. For example, an applet with threads would use the init method to create the threads and the destroy method to kill them.

The implementation of this method provided by the Applet class does nothing.

public void destroy()

Called by the browser or applet viewer to inform this applet that it is being reclaimed and that it should destroy any resources that it has allocated. The stop method will always be called before destroy.

A subclass of Applet should override this method if it has any operation that it wants to perform before it is destroyed. For example, an applet with threads would use the init method to create the threads and the destroy method to kill them.

The implementation of this method provided by the Applet class does nothing.

public void start()

Called by the browser or applet viewer to inform this applet that it should start its execution. It is called after the init method and each time the applet is revisited in a Web page.

A subclass of Applet should override this method if it has any operation that it wants to perform each time the Web page containing it is visited. For example, an applet with animation might want to use the start method to resume animation, and the stop method to suspend the animation.

The implementation of this method provided by the Applet class does nothing.

public void stop()

Called by the browser or applet viewer to inform this applet that it should stop its execution. It is called when the Web page that contains this applet has been replaced by another page, and also just before the applet is to be destroyed.

```
        A subclass of Applet should override this method if
        it has any operation that it wants to perform each
        time the Web page containing it is no longer
        visible. For example, an applet with animation might
        want to use the start method to resume animation, and
        the stop method to suspend the animation.

        The implementation of this method provided by the
        Applet class does nothing.
```

Note that all these four methods need to return after performing their function - none of them can be treated as the `main()` method of a stand-alone application. To have an execution thread independent of any user events, one must use Java `Threads`.

4 A Very Simple Applet Example

```
import java.awt.*;

public class HelloWorldApplet extends java.applet.Applet {

    private Label label;

    public void init() {
        label = new Label("HELLO, WORLD!");
        add(label);
    }
}
```

5 An Introtuction to Threads

This is the last painful thing we will have to learn about in this course. But, we have to do this, since it is crucial in a lot of applications, and applets.

You all probably know what a *process* or *task* is in the context of computers and operating systems. A computer can run multiple processes seemingly all at the same time. For example, it moves your mouse, processes user input, and browses the internet concurrently. This is possible by *processes*, each program runs in a separate memory area as a separate process. A *thread* is very similar, different only in that the multiple processes all operate on the *same* memory area. In other words, threads are used for doing more than one job concurrently in a single program.

You have already seen examples of threads, without realizing it. For instance, the execution starting at the `main()` method of an application is a thread. But, all event handling methods are called by *other* system threads. So, in effect, any AWT program is a multithreaded program.

5.1 How to Create Threads

The class Java provides for writing multi-threaded programs is `Thread`. There are two ways of writing multi-threaded programs: One is to extend `Thread`, the other is to implement the `Runnable` interface, and have a `Thread` member variable within the class to handle the execution. The second is the more useful, yet harder method, so we will see how to do the job with the first method first.

The `Thread` class has two very important methods. One is the `public void start()` method, which causes the thread to start execution. The second is the `public void run()` method, which is where the execution starts (in a new thread) when `start()` is called. So, in order to get any useful functionality, you must extend the `Thread` class, and override the `run()` method with a method that contains the code you want executed concurrently with other parts of the program.

5.2 A Simple Thread Example

Here is a simple thread example. What it does is print a line on `System.out` every second.

```
public class Tick extends Thread {
    public void run() {
        while (true) {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                // Ignored
            }

            System.out.println("Tick!");
        }
    }
}
```

Here is another class, which prints to standard output every five seconds:

```
public class Tock extends Thread {
    public void run() {
        while (true) {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                // Ignored
            }

            System.out.println("Tock!!!");
        }
    }
}
```

And here is the main class that will execute both.

```
public class Main extends Object {


    public static void main (String args[]) {
        new Tick().start();
        new Tock().start();
    }

}
```

Type in all three classes, and execute `Main`. Watch the output.

6 More about Threads

Now that we have seen how to create a thread using the easy method, which is to extend the `Thread` class, and override the `run()` method. This time, we will see how to do this the harder – but more useful – way.

The second way to create a thread is to make a class implement the `Runnable` interface. This is a very simple interface – there is only one method that you need to implement, and that method is `public void run()`.

Implementing the `Runnable` interface means that the object has a `run()` method, and therefore can be executed by a `Thread`. (Note that, if you call the `run()` method of this object by yourself, it will execute, but it will execute in the calling thread, not in a new thread.)

If we modify the `Tick` object from the last handout to implement `Runnable` rather than extending `Thread`, here is what we get:

```
public class Tick extends Object implements Runnable {
    public void run() {
        while (true) {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                // Ignored
            }

            System.out.println("Tick!");
        }
    }
}
```

Now, if you try to recompile the `Main` class from last time, it will not compile, saying that `Tick` is missing the `start()` method. It is exactly right, since we do not inherit the `start()` method from `Thread` any more. So, how can we get this `Runnable` object to run? The answer is, by constructing a `Thread` that will execute it. The class `Thread` has a constructor which does exactly that:

```
public Thread(Runnable obj)
```

This constructs a thread, which, when the `start()` method is called, will execute the `run()` method of `obj` in a newly created thread. If we let the code do the talking, here is the modified `Main` class that will execute it:

```
public class Main extends Object {


    public static void main (String args[]) {
        new Thread(new Tick()).start();
        new Tock().start();
    }

}
```

We can rewrite that to be more understandable as follows:

```
public class Main extends Object {


    public static void main (String args[]) {
Tick tick = new Tick();
Thread thread = new Thread(tick);
thread.start();

Tock tock = new Tock();
tock.start();
    }

}
```

This, although perfectly correct, is not what is often done when one wants to create a `Runnable` object. What *is* done, is to make the `Runnable` object carry its own `Thread`, as a private member variable, and also implement a suitable `start()` method, so that it functions just as if it had extended `Thread`. Once again, an example is the best explanation, so here is what `Tick` looks like using this approach:

```
public class Tick extends Object implements Runnable {

    private Thread runner;

    public void start() {
        if (runner == null) {
            runner = new Thread(this);
        }

        runner.start();
    }

    public void run() {
        while (true) {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                // Ignored
            }

            System.out.println("Tick!");
        }
    }
}
```

The new thing is the `start()` method. The first `if` constructs a new thread if one is not already created (we do not want to create a new thread if one already exists). It is given `this` as an argument (which is `Runnable`) so that it will execute the `run()` method of this object when `start()`ed. Then, we do exactly that, `start()` the thread.

7 The War of the Threads

Here, we will write a simple application that will simulate bank accounts. We will have several clerks operating on the accounts, moving money from one account to another. Then, we will see if "conservation of money" holds or not.

```
public class Account extends Object {

    private int balance;


    public int getBalance() {
        return balance;
    }

    public void setBalance(int balance) {
        this.balance = balance;
    }


    public Account() {
        balance = 1000;
    }

    public Account(int balance) {
        this.balance = balance;
    }

    public void credit(int amount) {
        balance += amount;
    }

    public boolean debit(int amount) {

        if (balance >= amount) {
            balance -= amount;
            return true;
        } else {
            return false;
        }

    }

}

public class Bank extends Object implements Runnable{

    private Account[] accounts;
    private Clerk[] clerks;
    private Thread runner;


    /** Creates new Bank */
    public Bank(int numAccounts, int numClerks) {
        accounts = new Account[numAccounts];
        clerks = new Clerk[numClerks];

        for (int i=0; i<numAccounts; i++) {
            accounts[i] = new Account(1000);
        }

        for (int i=0; i<numClerks; i++) {
            clerks[i] = new Clerk(this);
        }
    }

    public int getNumAccounts() {
        return accounts.length;
    }

    public Account getAccount(int i) {
        return accounts[i];
    }

    public int getTotalMoney() {
        int total = 0;

        for (int i=0; i<accounts.length; i++) {
            total += accounts[i].getBalance();
        }

        return total;
    }

    public void start() {
        if (runner == null) {
            runner = new Thread(this);
        }

        runner.start();
    }

    public void run() {
        for (int i=0; i<clerks.length; i++) {
            clerks[i].start();
        }


        System.out.println("Total money in the bank: "
                           + getTotalMoney());

        try {
            Thread.sleep(10000);
        } catch (InterruptedException e) {
            // Ignored
        }

        for (int i=0; i<clerks.length; i++) {
            clerks[i].stop();
        }

        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            // Ignored
        }

        System.out.println("Total money in the bank: "
                           + getTotalMoney());

    }

    public static void main(String[] args) {
        Bank theBank = new Bank(20, 400);
        theBank.start();

    }
}

import java.util.Random;

public class Clerk extends Object implements Runnable {

    private Bank bank;
    private Thread runner;
    private boolean go = true;


    public Clerk(Bank bank) {
        this.bank = bank;
    }

    public void start() {
        if (runner == null) {
            runner = new Thread(this);
        }
        runner.start();
    }

    public void stop() {
        go = false;
    }

    public void run() {

        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            // Ignroed.
        }

        Random rand = new Random();

        while (go) {
            int amount = rand.nextInt(50) + 1;
            int fromAccountNumber = rand.nextInt(bank.getNumAccounts());
            int toAccountNumber = rand.nextInt(bank.getNumAccounts());

            Account fromAccount = bank.getAccount(fromAccountNumber);
            Account toAccount = bank.getAccount(toAccountNumber);


            if (fromAccount.debit(amount)) {
                toAccount.credit(amount);
            }


        }
    }
}
```

Here is the sample output from this program:

```
Total money in the bank: 20000
Total money in the bank: 24684
```

And another sample:

```
Total money in the bank: 20000
Total money in the bank: 12315
```