NETWORK PROGRAMMING

Today

- Today we will start to look at ways to write programs that operate over the network.
- We'll lok at two approaches that are supported by Java:
  - Applets
  - Socket-based communication
- In later lectures we'll look at other netcentric-computing topics such as:
  - Web-programming
  - Security

Applets

- An applet is a Java program that runs in the context of a web page.
- Aplets are part of the Java dream of network distributed code:
  - Thin client
  - All apps are downloaded as needed, per-use
- This never happened on a large scale
  - Optimists would say “hasn’t yet happened”.
- Applets are the small-scale remnent.
- Allow you write code that is placed on a (web) server and automatically distributed to anyone who wants to use it.

Simple Applet

- We'll start with the example `HelloApplet`
  - An applet version of "Hello world".
- This code grabs a 200 by 200 pixel space.
- Paints it yellow.
- Writes "Hello World!" in it.
- Quite a few things to note about this.

- This applet is a subclass of the `Applet` class.
  - We will see another applet super-class later.
- There is no `main()`.
  - We'll see one way around the lack of parameters later
- Instead an applet has the functions:
  - `init()`
  - `start()`
  - `stop()`
  - `destroy()`

- Applets don’t have explicit constructors
  - Because the browser has control over applet creation you can’t rely on what resources are available.
- So `init()` is where you do initialization
- `init()` is called once, after the creation of the applet.
- All environment information should be available by the time `init()` is run.

- `start()` is run whenever the applet becomes visible on the web page.
  - Moving to the page
  - Scrolling to the applet's location

  (Switching tabs is not the same as "becomes visible")
- `start()` may be run multiple times.

- `stop()` is run whenever the applet becomes invisible on the web page.
  - Moving away from the page
  - Scrolling away from the applet's location

  (Switching tabs is not the same as "becomes invisible")
- `stop()` may be run multiple times.
- Idea is that CPU-intensive applet tasks should not typically run when the applet is not visible.

- `destroy()` is called when the applet is shutdown.
  - Like a destructor
- When exactly depends (like so much applet stuff) on the browser.
  - Netscape only runs `destroy()` when it deletes applets from the cache.

- Since an applet runs in a webpage, it has to have a web-page to live in.
- It even has its own HTML tage.
- The next page gives an example.

```
<html>
<head>
<title>
A First Applet
</title>
<body>
Here you go:
<p>
<applet code="HelloApplet.class" width=400 height=400>
Your browser doesn’t support Java, or Java is not
enabled, Sorry!
</applet>
<p>
Isn’t that nice?
</body>
</head>
</html>
```

- The applet part sits in and inbetween the applet tags:

```
<applet>
</applet>
```

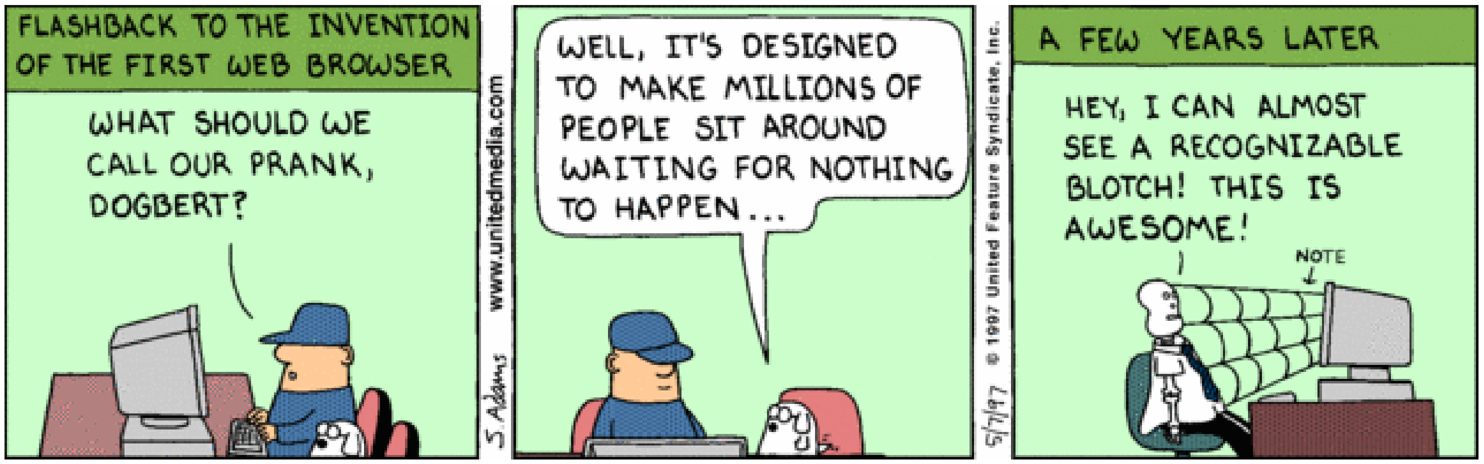
- The tag specifies where the applet code is, and how much screen should be reserved for the applet.
  - The reason there is blank space around the colored chunk in `HelloApplet` is because it specifies that the applet will take up 200 by 200 pixels and the HTML leaves it 400 by 400.
- The text between the tags is displayed if the browser can't handle the code. (there can be arbitrary HTML code here)

- `code=` indicates the location of the code relative to the web-page.
  - relative path
- Instead we can use `codebase=` to give a URL.

Applet politics

- In the static web-ebvironment of 1994, applets were revolutionary.
  - No dynamic HTML
  - No dynamic GIFs
  - No Flash
- A chance for dynamic content.
- But politics and technical issues killed the dream of wide-spread, large-scale applet use.

- Networks were too slow to download large Applets in reasonable time.


FLASHBACK TO THE INVENTION OF THE FIRST WEB BROWSER
WHAT SHOULD WE CALL OUR PRANK, DOGBERT?
www.unitedmedia.com
S. Adams
WELL, IT'S DESIGNED TO MAKE MILLIONS OF PEOPLE SIT AROUND WAITING FOR NOTHING TO HAPPEN...
5/7/97 © 1997 United Feature Syndicate, Inc.
A FEW YEARS LATER
HEY, I CAN ALMOST SEE A RECOGNIZABLE BLOTCH! THIS IS AWESOME!
NOTE

- `Applet` is an AWT class, and the original AWT was native code that interfaced directly with the local window manager
  - It was thus rather nonportable.
  - “a large graphical C program that Java talked to”
- Applets thus depended how well Sun could get AWT to talk to the (for example) Microsoft Windows window manager.

- Politics also played a role.
- Netscape insisted Java have a “native look and feel” rather than a more limited portable toolkit.
- Microsoft froze the Applet API that its browsers supported at 1.1.
  - Issue resolved by a lawsuit that took years to complete.
- This led to Sun developing the Java Plug-in
  - Version of the JVM that can be added to the browser.
- Also allowed the use of Swing (native Java inferface components) with applets in the form of `JApplet`.

JApplet

- At one level `JApplet` is just a Swing version of `Applet`.
- Consider `HelloAppletWJ`.
- The only difference between this and `HelloApplet` is the use of `JApplet`.

- But you can also exploit the fact that `JApplet` is a combination of `Applet`
  - with `init()`, `start()` and so on

  and `JPanel`
  - with layout managers and easy, recursive, addition of components.
- Note that `Applet` is a subclass of `Panel`, AWT's basic container.

- A more sophisticated use of `JApplet` is in `ShowApplet`

Browser security

- Since An applet can come from any source, it is not sensible to trust them.
- Untrusted applets are ths constrained by a security manager.
  - Part of the browser
- Untrusted applets are typically restricted to:
  - Not read or write files locally.
  - Only open connections to the orignating server.
  - Not start new processes locally
  - Not have native methods.
- Applets can become trusted by using digital signatures.
  (We'll discuss digital signatures when we talk about security).

Other Applet features

- Applets can get information about their environment using the:
  - `getDocumentBase()`
  - `getCodeBase()`

  methods.
- The first returns the URL in which the applet appears, and the second gives the base URL of the applet's class files.
- `java.net` provides ways of handling URL objects.

- The `getImage()` method takes a URL as an argument and retrives an image from that location.
- The `showDocument()` method can retrieve a document from a URL and get the broswer to display it.
  - An optional argument can be used to control where this is shown, including in a new window
- `showDocument()` isn't an applet method, but a method of `AppletContext`, and this can be retrived using `getAppletContext()`.

Multifile applets

- `Dots` is an example of an applet that contains multiple objects.
- As usual this means the source code is in multiple `.java` files, and, when compiled, we get multiple `.class` files.
- As you can see from the example, the associated HTML file just has to refer to the top-level `.class`
  - The `.class` obtained by compiling the `.java` that contains `init()`, `start()` and so on.

Applet Parameters

- Because an applet has no `main()`, we can't pass it command line parameters.
  - Not that we have done much of that for regular Java applications.
- Can pass values from the HTML.
- For example:

```
<param name="myPort" value = "1234">
```

- Is read by the applet using:

```
myPort =
Integer.parseInt(getParameter("myPort"));
```

Deploying Applets

- We write applets in the usual way — creating one or more `.java` files in some editor.
- Compiling them is also as usual — we run the compiler to create a `.class` file.
- One way to look at the applet to test it is through an applet viewer.
  - Eclispe includes one
  - One is provided in the standard suite of JAva tools.
- You can also view the webpage that includes the applet in your web browser.

- To deply an applet you need to upload the associated HTML file and the `.class` file to somewhere from which they can be accessed by a web-server.

  (The browser does not need access to the `.java` file)
- You'll see one way to do this when you complete this week's homework.

Sockets

- Java networking support (the `Java.net` package) falls into two parts:
  - Sockets
  - Web-oriented (URL handling)
- Sockets provide access to the standard network protocols for communicating across the Internet.
- Sockets underlie the other mechanisms.
  - Including all the Java web-oriented stuff
- Can build applications using sockets, but you are responsible for the way data is handled *except* how it is transmitted.
  - And you aren't completely isolated from that.

- Three kinds of protocol supported by Java sockets:
  - `Sockets`
  - `DatagramSockets`
  - `MulticastSockets`
- `Sockets` provide a *reliable connection-oriented* service.
- `DatagramSockets` provide a less reliable *connectionless* service.
- `MulticastSockets` send datagram packets to multiple recipients.

Addressing

- Client needs a *hostname* and *port* to connect to a server.
- An analogy:
  - Hostname is the address of the hotel.
  - Port is the room.

Clients and Servers

- Increasingly vague terminology.
- BUT, in a client-server interaction, the side that starts the connection is the *client*.
- Side that accepts the request is the *server*.
- `java.net.Socket` represents one side of a connection.
- `java.net.ServerSocket` is used by the server to listen for connections.
  - Creates a `ServerSocket` and waits in `accept()` until a connection arrives.
  - When a connection arrives, `accept()` creates a `Socket` to communicate.
- Creating one `Socket` for each requested connection allows a server to communicate with multiple clients.

- An example of a socket-based connection is `SimpleServer`
- On the server side, we start with:

```
try {
    ServerSocket serverSocket = new ServerSocket(port);
        :
        :
    }
    catch ( IOException iox ) {
      System.err.println( "Could not listen on port: " + port );
      System.exit(1);
}
```

- The rest of the server code is in the `try{}` block, and we `catch` a common exception, the failure to open the relevant port.
- (The server can also be run in a mode that opens any available port)

- Then (inside the previous `try{}`, we have:

```
try {
      Socket client = serverSocket.accept();
       :

       <code that uses the socket>
    }
    catch ( IOException iox ) {
      System.err.println(iox);
      System.exit(1);
    }
```

- This waits for a client to make a connection, and then creates a socket connection between them.

- Sockets can support two streams, one for receiving information and one for tramsitting information:

```
InputStream inStream = client.getInputStream();
OutputStream outStream = client.getOutputStream();
```

- For ease of handling, we wrap these streams in objects that make it possible to send a line that ends with a carriage return or newline:

```
BufferedReader in
   = new BufferedReader(new InputStreamReader(inStream));
PrintWriter out
   = new PrintWriter(new OutputStreamWriter(outStream));
```

- This allows us to send arbitrary-length lines (appropriate for this example where client and server exchange typed sentences).

- The code on the client side is a bit simpler, but mirrors the Server code:

```
try{
   Socket server = new Socket(host,port);

   InputStream inStream = server.getInputStream();
   OutputStream outStream = server.getOutputStream();

   BufferedReader in
      = new BufferedReader(new InputStreamReader(inStream));
   PrintWriter out
      = new PrintWriter(new OutputStreamWriter(outStream));

    <code that uses the socket>

   }
```

- Note that the client needs to know the host that the server is running on.
- `port` is just an `int`.
- `host` is either a string inclduing a hostname like `www.google.com`, or an IP address like `173.194.73.106`. (That is the IP address that responds when you ping `www.google.com`.)

- The `try{}` block above is followed by two `catch{}`s:

```
    catch (UnknownHostException uhx) {
       System.err.println(uhx);
       System.exit(1);
    }
    catch (IOException iox) {
       System.err.println(iox);
       System.exit(1);
    }
}
```

- Note that both client and server have `main()`
- They will run separate processes, potentially on different machines.

- It is possible to send data through the raw input and output streams.
- For example:

  ```
  outStream.write(42)
  ```

  would write a byte to the `outputStream`, and:

  ```
  byte back = (byte)inStream.read()
  ```

  would read a byte from the `inputStream`.
- This kind of interaction is appropriate when all the client and server need to send are individual `byte`s,. `int`s and so on.

- Wrapping the input stream in a `BufferedReader` means that it is possible to read a sequence of data upto a newline or carriage return:

```
String input = (String)in.readLine();
```

- The `BufferedReader`, as its name suggests, holds the incoming data until the line is complete.
- The size of the buffer can be adjusted.
- `PrintWriter` allows whole lines to be written:

```
out.println(msg);
out.flush();
```

- You can also read and write objects as we will see later.

- The last thing to notice about this example is its use of command-line arguments.
- To run the server, I would type (for example):

  ```
  java Server 1234
  ```

  here 1234 is the portnumber that the server sets up to listen on.
- If not port number is entered, `new ServerSocket(port)` with `port` having the value 0.
- This creates a connection on an unused port.
- Similarly I might run the client using:

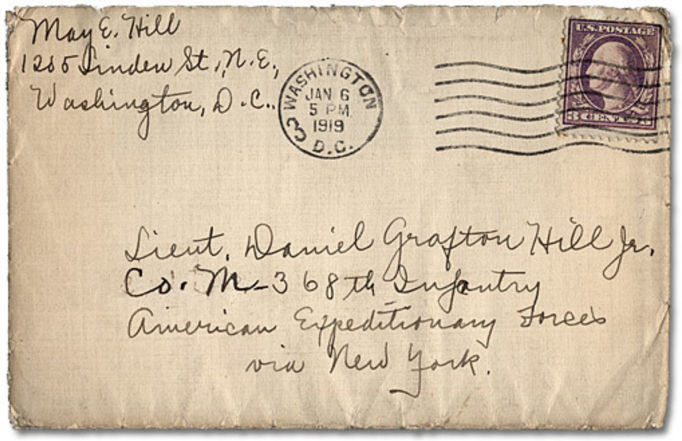
  ```
  java Client localhost 1234
  ```

  which identifies the port and the host to connect to (in this case "localhost" is the same machine as the client, appropriate when both processes are running on the same machine.

Datagram sockets

- As we said before, `Socket` provides a connection-oriented service analgous to a telephone call.
- When we don't want the overhead of opening up an end-to-end connection, we can use `DatagramSocket` which provides a connectionless service.

  Analagous to a letter.
- `HeartBeat` provides an example of this kind of socket.



- The server listens for a connection much as before:

```
DatagramSocket dSock =
   new DatagramSocket(Integer.parseInt(args[0]));
```

- This time we have to specify how large a packet is (in practice the maximum is 8K, though 64K is supposed to be possible).

```
DatagramPacket packet =
  new DatagramPacket(new byte[1024], 1024);
```

- Then we can receive the message and extract the content:

```
dSock.receive(packet);
String message =
   new String(packet.getData(), 0, packet.getLength(), "UTF-8");
```

- On the client side we just assemble the packet and post it off to the right port on the server:

```
DatagramPacket packet =
   new DatagramPacket(data, data.length, address, myPort);

DatagramSocket dSock = new DatagramSocket();
dSock.send(packet);
dSock.close();
```

Sending objects

- `ObjectServer` is our last example.
- This looks rather like the other examples, with two exceptions.
- Thr first is that when the socket is set up, the streams are opened with:

```
ObjectInputStream in =
   new ObjectInputStream(client.getInputStream());
ObjectOutputStream out =
   new ObjectOutputStream(client.getOutputStream());
```

- This allows objects to be passed through the stream:

```
while(true){out.writeObject(processRequest(in.readObject()));
out.flush();
```

- The other difference is that when the `ServerSocket` runs `accept()`, it does so in a separate thread for each request to the port it is listening to.
- This allows it to handle multiple connections.

- Finally, note that to send objects in this way, the need to be serializable.
- The classes `Request`, `DateRequest`, `WorkRequest` and `MyCalculation` show how this may be done.
- Note that in the example, the server performs a computation by calling a method on the object it is passed.

Summary

- This lecture dealt with some aspects of network programming in Java
- It covered Applets and programming using Sockets

  (And even gave an example which combined the two).
- It looked at both `Socket` and `DatagramSocket`, thus covering both connection-oriented and connectionless sockets.