Module-4:

Advanced Programming Using OOP

GUI Programming

Graphical User Interface

- A **GUI** (graphical user interface) is a system of interactive visual components for computer software.
- Structure
  - Create the icons and widgets that are displayed to a user and organize them inside a screen window.
  - Define functions that will process user and application events.
  - Associate specific user events with specific functions.
  - Start an infinite event-loop that processes user events. When a user event happens, the event-loop calls the function associated with that event.

How does a GUI work?

- A GUI uses windows, icons, and menus to carry out commands, such as opening, deleting, and moving files. Although a GUI operating system is primarily navigated using a mouse, a keyboard can also be used via keyboard shortcuts or the arrow keys.

What are the benefits of GUI?

- Unlike a command-line operating system or CUI, like Unix or MS-DOS, GUI operating systems are much easier to learn and use because commands do not need to be memorized. Additionally, users do not need to know any programming languages. Because of their ease of use and more modern appearance, GUI operating systems have come to dominate today's market.

The Basic GUI Application

- There are two basic types of GUI program in Java: stand-alone applications and applets.
- A stand-alone application is a program that runs on its own, without depending on a Web browser.
- An applet is a program that runs in a rectangular area on a Web page.

JFrame and JPanel

- In a Java GUI program, each GUI component in the interface is represented by an object in the program.
- One of the most fundamental types of component is the window.
- Windows have many behaviors
- They can contain other GUI components such as buttons and menus.

Contd.

- A *JFrame* is an independent window that can, for example, act as the main window of an application. One of the most important things to understand is that a *JFrame* object comes with many of the behaviors of windows already programmed in.

Components and Layout

- Another way of using a *JPanel* is as a container to hold other components
- Java has many classes that define GUI components.
- Componenets must have some containers
  - content.add(displayPanel, BorderLayout.CENTER);
  - content.add(okButton, BorderLayout.SOUTH);
- content refers to an object of type *Jpanel*
- this panel becomes the content pane of the window

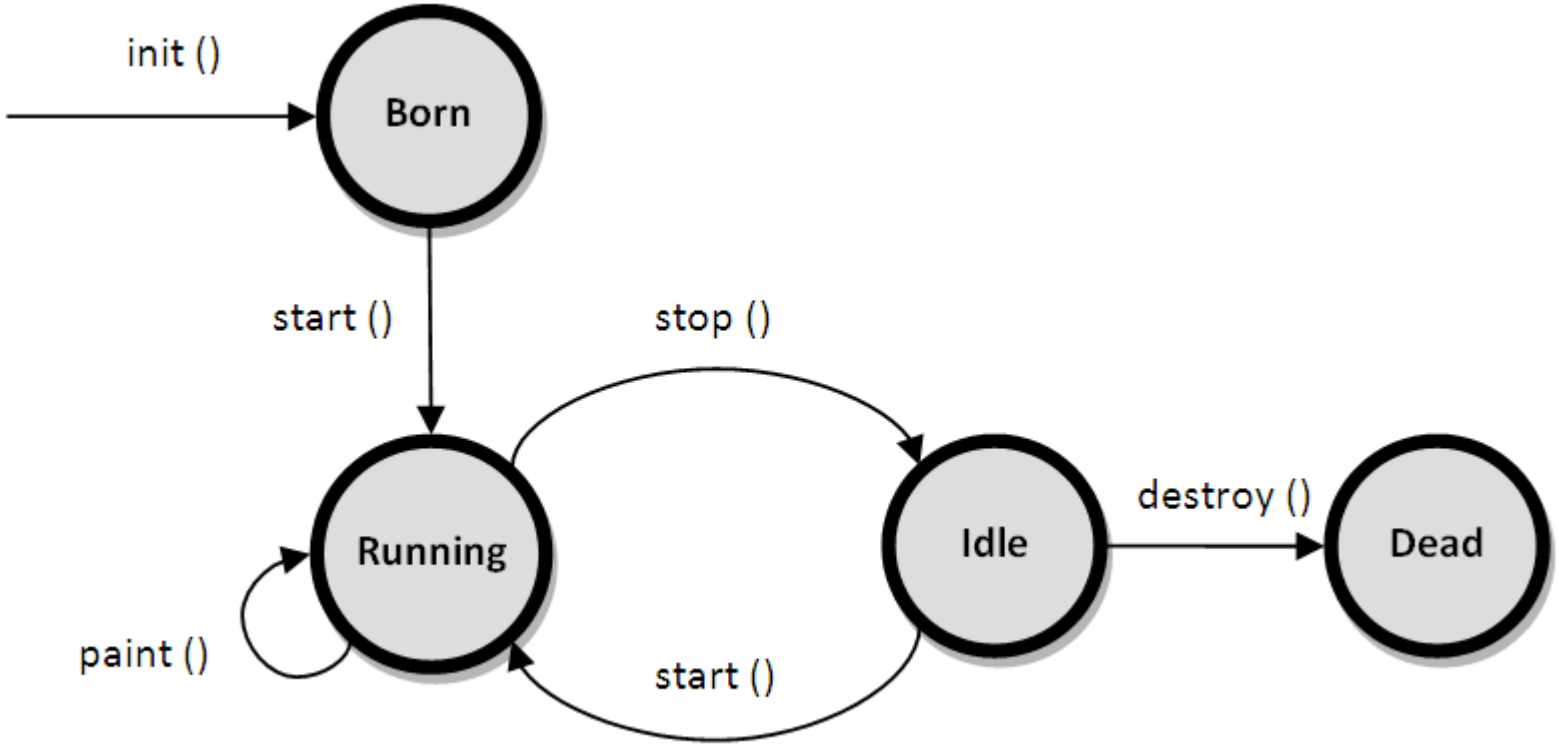
Events and Listeners

- The structure of containers and components sets up the physical appearance of a GUI, but it doesn't say anything about how the GUI behaves.
- GUIs are largely event-driven; When an event occurs, the program responds by executing an event-handling method.
- The most common technique for handling events in Java is to use event listeners. A listener is an object that includes one or more event-handling methods.

Applets and HTML

- Japplet - The *JApplet* class (in package javax.swing) can be used as a basis for writing applets in the same way that *JFrame* is used for writing stand-alone applications.
- To create an applet, you will write a subclass of *JApplet*. The *JApplet* class defines several instance methods that are unique to applets.

Contd.



Applets on Web Pages

- The <applet> tag can be used to add a Java applet to a Web page.
  - E.g. <applet code="HelloWorldApplet.class" height=100 width=250> </applet>
- Applets can use applet parameters to customize their behavior.
- The value what will be returned is always a string.

Graphics and Painting

- The Java API includes a range of classes and methods that are devoted to drawing.
- The physical structure of a GUI is built of components. The term component refers to a visual element in a GUI, including buttons, menus, text-input boxes, scroll bars, check boxes, and so on.
- Java, GUI components are represented by objects belonging to subclasses of the class java.awt.Component
- Most components comes under the Swing GUI.

Contd.

- A JPanel, like any JComponent, draws its content in the method
  - public void paintComponent(Graphics g)
  - the paintComponent() method has a parameter of type *Graphics*.
  - The *Graphics* object will be provided by the system when it calls the written method
  - To do any drawing at all in Java, we need a graphics context.
  - A graphics context is an object belonging to the class java.awt.Graphics.
  - *Graphics* object can draw to only one location.
  - Co-ordinates
  - Color
  - Fonts
  - Shapes
  - Graphics 2D

Mouse Events

- In Java, events are represented as objects.
- When an event occurs, the system collects all the information relevant to the event and constructs an object to contain that information.
- For example, when the user presses one of the buttons on a mouse, an object belonging to a class called *MouseEvent* is constructed.
- When the user presses a key on the keyboard, a *KeyEvent* is created.

Event Handling

- For an event to have any effect, a program must detect the event and react to it. In order to detect an event, the program must "listen" for it. Listening for events is something that is done by an object called an event listener. An event listener object must contain instance methods for handling the events for which it listens. For example, if an object is to serve as a listener for events of type *MouseEvent.*
- The body of the method defines how the object responds when it is notified that a mouse button has been pressed. The parameter, evt, contains information about the event.
- The methods that are required in a mouse event listener are specified in an interface named *MouseListener.*
- Mouse Coordinates.
  - *MouseEvent*
- MouseMotionListeners and Dragging
  - public void mouseDragged(MouseEvent evt);
    public void mouseMoved(MouseEvent evt);

Timers, KeyEvents, and State Machines

- A *Timer* generates events at regular intervals.
  - class javax.swing.Timer.
- Timers and Animation
  - the *ActionListener* interface.
  - timer = new Timer(millisDelay, listener);
- Keyboard Events
  - KeyEvent
- Focus Events
  - In Java, objects are notified about changes of input focus by events of type *FocusEvent*. An object that wants to be notified of changes in focus can implement the *FocusListener* interface.
- State Machines
  - In computer science, there is the idea of a state machine, which is just something that has a state and can change state in response to events or inputs. The response of a state machine to an event or input depends on what state it's in. An object is a kind of state machine.

Basic Components

- API are defined by subclasses of the class *JComponent*, which is itself a subclass of *Component*.
- Jbutton
  - Constructors
  - Events
  - Listeners
  - Registration of Listeners
  - Event methods
  - Component methods
- Jlabel
- JCheckBox
- JTextField and JTextArea
- JComboBox
- Jslider

Basic Layout

- Basic Layout Managers
  - *FlowLayout*, *BorderLayout*, and *GridLayout*.
- A *FlowLayout* simply lines up components in a row across the container.
- The default layout for a *JPanel* is a *FlowLayout*;

Menus and Dialogs

- Menus and Menu bars
- Dialogs
- Fine Points of Frames
- Creating Jar Files

Concurrent Programming

- Concurrency is the ability to run several programs or several parts of a program in parallel.
- Concurrency enable a program to achieve high performance and throughput by utilizing the untapped capabilities of underlying operating system and machine hardware.

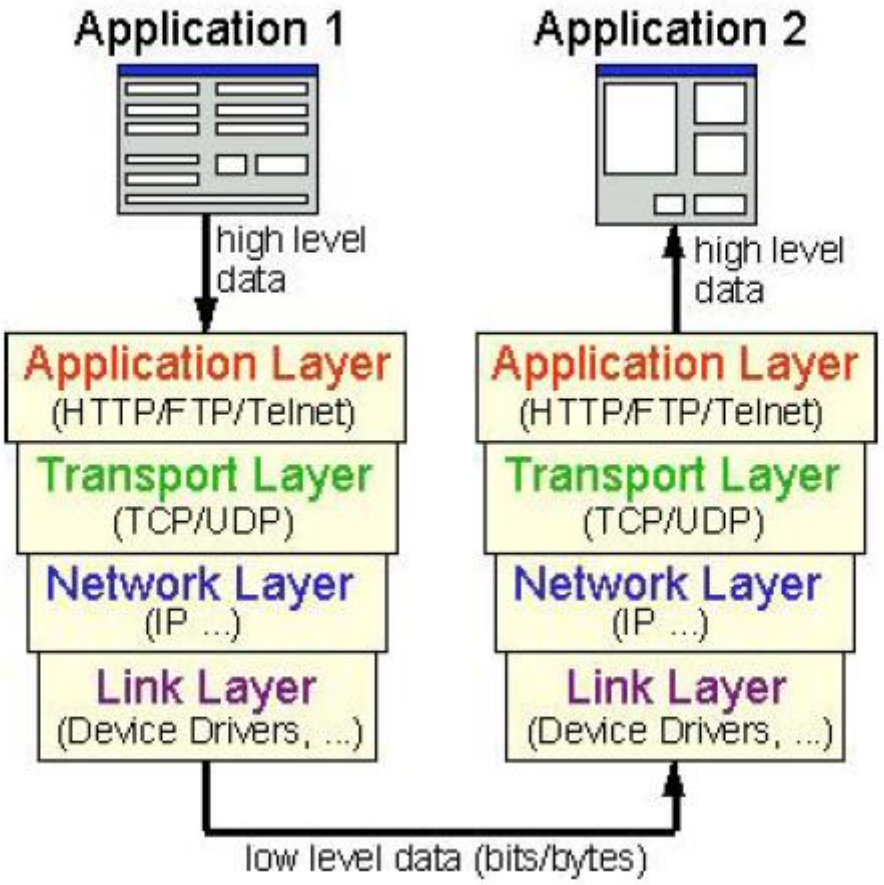
Difference between Concurrent & Parallel programming

- In *parallel programming*, parallel processing is achieved through hardware parallelism e.g. executing two processes on two separate CPU cores simultaneously.
- Two Models for Concurrent Programming
  - Shared memory
  - Message passing

Network Programming

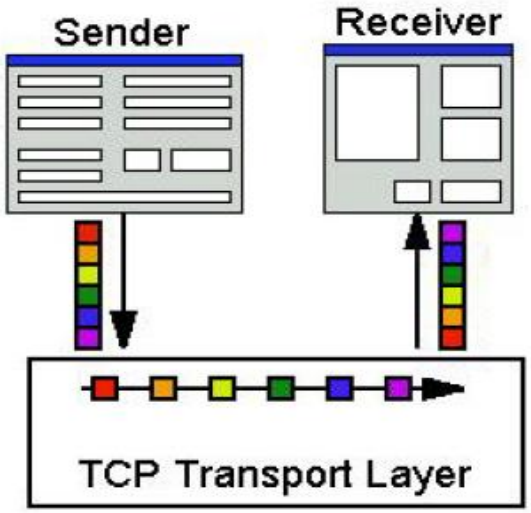
- It involves writing programs that communicate with other programs across a computer network
- A server is an application that provides a "service" to various clients who request the service.
- Everybody can either be a client or a server at any time. This is known as peer-to-peer computing.
- A Protocol is a standard pattern of exchanging information.
- Computers running on the internet typically use one of the following high-level Application
- Layer protocols to allow applications to communicate:
  - o Hyper Text Transfer Protocol (HTTP)
  - o File Transfer Protocol (FTP)
  - o Telnet

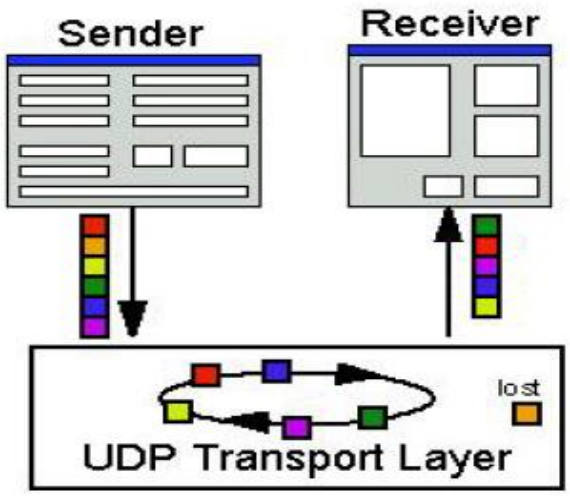
Contd.


Application 1
Application 2
high level data
high level data
Application Layer
(HTTP/FTP/Telnet)
Transport Layer
(TCP/UDP)
Network Layer
(IP ...)
Link Layer
(Device Drivers, ...)
Application Layer
(HTTP/FTP/Telnet)
Transport Layer
(TCP/UDP)
Network Layer
(IP ...)
Link Layer
(Device Drivers, ...)
low level data (bits/bytes)

Contd.

- The JAVA applications that communicate over a network, you are programming in the Application Layer.
- JAVA allows two types of communication via two main types of Transport Layer protocols:
  - TCP
  - UDP
- A port is used as a gateway or "entry point" into an application.





Reading Files From the Internet (URLs)

- A Uniform Resource Locator (i.e., URL) is a reference (or address) to a resource over a network (e.g., on the Internet).
- Example-
  - http://www.cnn.com/
  - http://www.apple.com/ipad/index.html
  - http://en.wikipedia.org/wiki/Computer_science
- A URL resource name may generally contain:
  - a Host Name - The name of the machine on which the resource lives.
    - http://www.apple.com:80/ipad/index.html
  - a Port # (optional) - The port number to which to connect.
    - http://www.apple.com:80/ipad/index.html
  - a Filename - The pathname to the file on the machine.
    - http://www.apple.com:80/ipad/index.html
- In JAVA, there is a URL class defined in the java.net package

Contd.

- Create own URL objects as,
  - URL webPage = new URL("http://www.apple.com/ipad/index.html");
- JAVA will "dissect" the given String in order to obtain information about protocol, hostName, file etc.
- Due to this, JAVA may throw a MalformedURLException ... so we will need to do this:

```
try {
URL webPage = new URL("http://www.apple.com/ipad/index.html");
} catch(MalformedURLException e) {
...
}
```

Example

```
import java.net.*;
public class URLTestProgram {
public static void main(String[] args) {
URL webpage = null;
try {
webpage = new URL("http", "www.apple.com", 80,
"/ipad/index.html");
} catch(MalformedURLException e) {
e.printStackTrace();
}
System.out.println(webpage);
System.out.println("protocol = " +
webpage.getProtocol());
System.out.println("host = " + webpage.getHost());
System.out.println("filename = " +
webpage.getFile());
System.out.println("port = " + webpage.getPort());
System.out.println("ref = " + webpage.getRef());
}
}
```

Output:

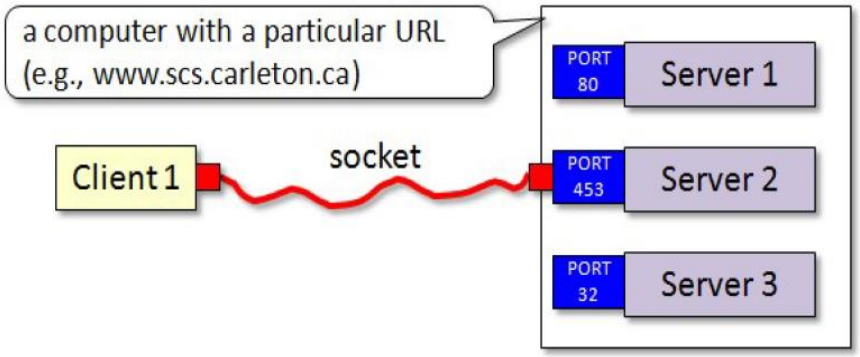
```
http://www.apple.com:80/ipad/index.html
protocol = http
host = www.apple.com
filename = /ipad/index.html
port = 80
ref = null
```

Client/Server Communications

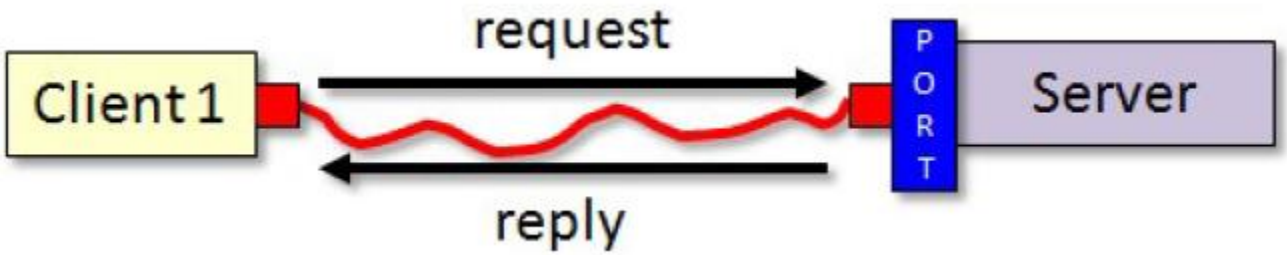
- A server is any application that provides a service and allows clients to communicate with it.
  - a recent stock quote
  - transactions for bank accounts
  - an ability to order products
  - an ability to make reservations
  - a way to allow multiple clients to interact (Auction)
- A client is any application that requests a service from a server
  - stock quotes must be accurate and timely
  - banking transactions must be accurate and stable
  - reservations/orders must be acknowledged

Contd.

- The port number is used as the server's location on the machine that the server application is running.
- The client and server

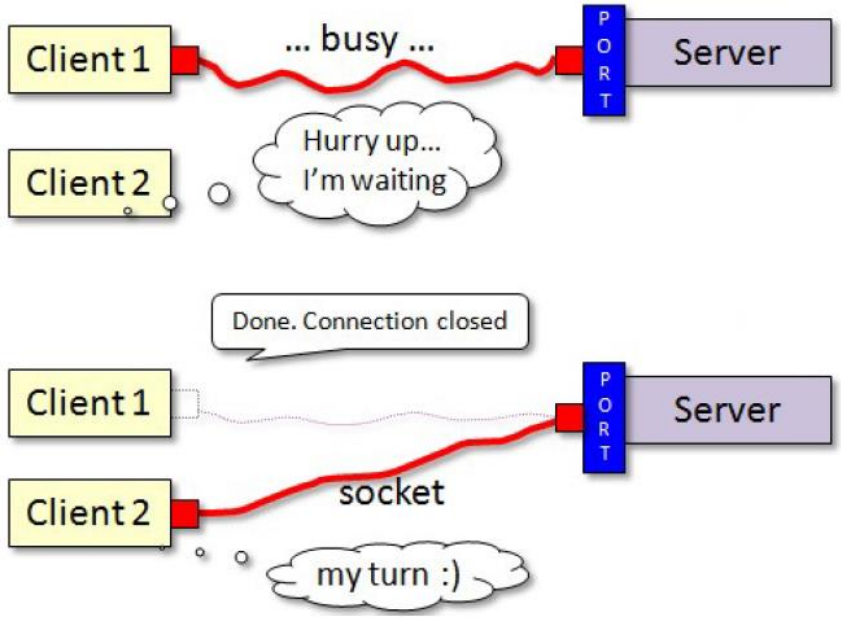


then each read and write to its end of the connection.



Contd.

- Accept();
- Close();
- InetAddress class
- getLocatHost();
- UnknownHostException.
- socket.getInputStream();
- socket.getOutputStream();
- flush();



Datagram Sockets

- UDP
- DatagramSockets
- DatagramPackets
- Buffer (i.e., an array of bytes)
- Each message is sent as a packet.
- Each packet contains:
  - the data of the message (i.e., the message itself)
  - the length of the message (i.e., the number of bytes)
  - the address of the sender (as an InetAddress)
  - the port of the sender

Contd.

```
byte[]  sendBuffer;
DatagramSocket  socket;
DatagramPacket packetToSend ;
socket = new DatagramSocket();
sendBuffer = "This is the data ... need not be a String".getBytes();
packetToSend = new DatagramPacket(sendBuffer, sendBuffer.length,
anInetAddress, aPort);
socket.send(packetToSend);
```

The server code for receiving an incoming packet involves allocating space (i.e., a byte array) for the DatagramPacket and then receiving it. The code looks as follows:

```
byte[] recieveBuffer;
DatagramPacket receivePacket;
recieveBuffer = new byte[INPUT_BUFFER_LIMIT];
receivePacket = new DatagramPacket(recieveBuffer,
recieveBuffer.length);
socket.receive(receivePacket);
```

Contd.

We need to extract the data from the packet. We can get the address and port of the sender as well as the data itself from the packet as follows:

```
InetAddress sendersAddress = receivePacket.getAddress();
int sendersPort = receivePacket.getPort();
String sendersData = new String(receivePacket.getData(), 0,
receivePacket.getLength());
```

In this case the data sent was a String

Unit Testing

- Testing Smaller Units of an Application
- Unit tests are implemented as classes with test methods.
- Example:

```
import org.junit.Test;
import static org.junit.Assert.*;

public class MyUnitTest {
    @Test
    public void testConcatenate() {
        MyUnit myUnit = new MyUnit();

        String result = myUnit.concatenate("one", "two");

        assertEquals("onetwo", result);

    }
}
```

Assert Methods

- The class org.junit.Assert
- The list of assert methods,
  - assertArrayEquals()
  - assertEquals()
  - assertTrue() + assertFalse()
  - assertNull() + assertNotNull()
  - assertSame() + assertNotSame()
  - assertThat()

```
import org.junit.Test;
import static org.junit.Assert.*;

public class MyUnitTest {

    @Test
    public void testGetTheStringArray() {
        MyUnit myUnit = new MyUnit();

        String[] expectedArray = {"one", "two", "three"};

        String[] resultArray =  myUnit.getTheStringArray();

        assertArrayEquals(expectedArray, resultArray);
    }
}
```

Matchers

- Matchers is an external addition to the JUnit framework. Matchers were added by the framework called Hamcrest. JUnit 4.8.2 ships with Hamcrest internally, so you don't have to download it, and add it yourself
- Types of Matchers:
  - Chaining Matchers
  - Core Matchers
  - Custom Matchers

```
import static org.junit.Assert.*;
import static org.hamcrest.CoreMatchers.*;
import org.junit.Test;

public class MyMatcherTest {

   @Test
   public void testWithMatchers() {
      assertThat("this string", is("this string"));
      assertThat(123, is(123));
   }
}
```

Contd.

Core		
	any()	Matches anything
	is()	A matcher that checks if the given objects are equal.
	describedAs()	Adds a descrption to a Matcher
Logical		
	allOf()	Takes an array of matchers, and all matchers must match the target object.
	anyOf()	Takes an array of matchers, and at least one of the matchers must report that it matches the target object.
	not()	Negates the output of the previous matcher.
Object		
	equalTo()	A matcher that checks if the given objects are equal.
	instanceOf()	Checks if the given object is of type X or is compatible with type X
	notNullValue() + nullValue()	Tests whether the given object is null or not null.
	sameInstance()	Tests if the given object is the exact same instance as another.

Contd.

- Custom Matchers

```
public static Matcher matches(final Object expected){

   return new BaseMatcher() {

      protected Object theExpected = expected;

      public boolean matches(Object o) {
         return theExpected.equals(o);
      }

      public void describeTo(Description description) {
         description.appendText(theExpected.toString());
      }
   };
}
```

- Testing for Exceptions
  - It checks whether your application throws right exceptions or not.
- **Stub, Mock and Proxy Testing**
  - A **Stub** is an object that implements an interface of a component
    - unit test --> stub
    - unit test --> unit --> stub
    - unit test asserts on results and state of unit
  - A **Mock** is very similar with stub, only it also has methods that determine what methods where called on the Mock.
  - It checks whether the unit handle various return values correctly or not.
  - Along with that it checks whether the unit uses the collaborator correctly or not.
  - unit test --> mock
  - unit test --> unit --> mock
  - unit test asserts on result and state of unit
  - unit test asserts on the methods called on mock
- Proxies in mock testing are mock objects that delegate the method calls to real collaborator objects, but still records internally what methods were called on the proxy.
  - unit test --> collaborator
  - unit test --> proxy
  - unit test --> unit --> proxy --> collaborator
  - unit test asserts on result and state of unit
  - unit test asserts on methods called on proxy

- Subclass Mock Objects
  - Subclass mock objects is a mock object that is created by subclassing the class you want to test, and overriding some of its methods.
  - Example,

```
public class MyUnit {
protected MyDependency dependency = null;
public MyUnit(MyDependency dep) {
 this.dependency = dep;
  }
  public void doTheThing(String param) {
 if("one".equals(param)) {
dependency.callOne();
 } else {
   dependency.callTwo();
  }
 }
}
```

  - Encapsulate calls to MyDependency inside MyUnit
  - Create a subclass of MyUnit
  - Override the MyDependency call encapsualation methods.

Step 1: Encapsulate calls to MyDepedency

```
public class MyUnit {
  protected MyDependency dependency =
null;
  public MyUnit(MyDependency dep) {
    this.dependency = dep;
  }
  public void doTheThing(String param) {
    if("one".equals(param)) {
      callOne();
    } else {
      callTwo();
    }
  }
  protected void callOne() {
      dependency.callOne();
  }
  protected void callTwo() {
      dependency.callTwo();
  }
}
```

Step 2: Create a Subclass Mock of MyUnit

```
public MyUnitMock extends MyUnit {

  protected boolean callOneCalled = false;
  protected boolean callTwoCalled = false;
  @Override
  protected void callOne() {
    this.callOneCalled = true;
    super.callOne();
  }
  @Override
  protected void callTwo() {
    this.callTwoCalled = true;
    super.callTwo();
  }

}
```

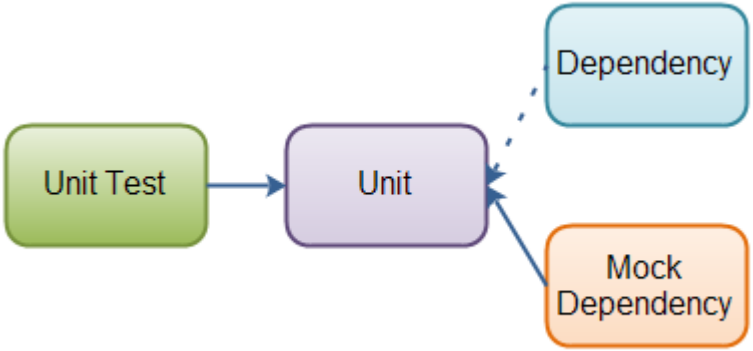
Step 3: Using the MyUnitMock Class in Your Unit Tests

```
@Test
public void test() {
  MyUnitMock myUnitMock = new MyUnitMock();

   myUnitMock.doTheThing("one");

   assertTrue (myUnitMock.callOneCalled);
   assertFalse(myUnitMock.callTwoCalled);

   //reset mock before next call
   myUnitMock.callOneCalled = false;

   myUnitMock.doTheThing("two");

   assertFalse(myUnitMock.callOneCalled);
   assertTrue (myUnitMock.callTwoCalled);
}
```

Summary

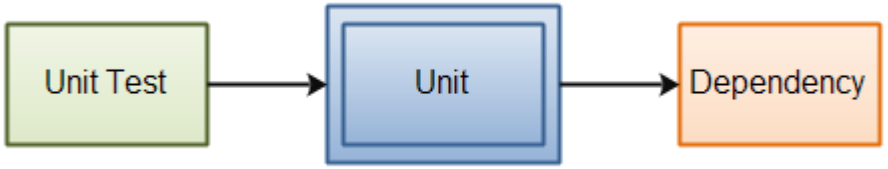
- it is possible to test almost all of a class by using subclass mocks, as described
- situations where it works better to use a completely separate mock dependency object with the original class instead
- Use it when it make sense, and don't use it when it doesn't make sense. Use your judgement!

- **IO Testing**
  - ByteArrayInputStream
  - ByteArrayOutputStream
  - CharArrayReader
  - CharArrayWriter
- **Servlet Unit Testing**
- **Unit Testing with Dependency Injection Containers**
  - If you are using a dependency injection container in your application, you can use the container to inject mock objects into your units during unit testing



- **Database Unit Testing**
  - Rollback Transactions
  - CRUD Testing
  - Test Data Creation

- **Code Coverage**
  - Code coverage means measuring how much of your code is executed during your unit tests.
  - To measure code coverage you need a coverage tool. Here is a small, but probably not exhaustive, list of code coverage tools for Java:
  - IntelliJ IDEA Coverage
  - Emma - http://emma.sourceforge.net/
  - EclEmma - http://www.eclemma.org/
- **Design for Testability**
  - **Why Encapsulation Makes Testing Harder**



  - **Replaceable Dependencies**
  - **Turning Private into Protected**
  - **Encapsulating Calls to Dependencies in Protected Methods**
  - **Move Code out of Boundary Classes**
- **Test First Development**