Java

(Recommended Reading: Budd Chapters 1-4)

- Object Oriented Language
- Descends from C++ and Smalltalk (Syntax looks like C++).
- Fixes many C++ features.
- Garbage collected.
- Good for résumés.
- Has safety properties.

Java Programs

1. Source code: `MyClass.java`
2. Compiled to byte-code: `MyClass.class`
3. Virtual Machine
   (a) Interprets byte-codes
   (b) Compiles byte-codes to machine code and runs natively.

Java Environment

javac compiles `.java` to `.class`.

java executes `.class` in runtime.

javadoc generates class documentation from `.java` files.

"Main Program" is class file with *static* function:

```
void main(String[] args);
```

Command line arguments are in array `args`.

A first Program

```
public class HelloWorld extends Object
{
   /**
    * Prints a familiar greeting.
    **/
   public static void main(String[] args)
   {
      System.out.println("Hello World!");
   }
}
```

To run this:

```
fiji% javac HelloWorld.java
fiji% java HelloWorld
Hello World!
```

Class Library

Java comes with an extensive class library.

Many common data structures are already implemented.

The the class reference is online and available in book form.

Date Example

Found class `java.util.Date`. Program to print out date:

```
import java.util.Date;

public class PrintDate
{
   /**
    * print out the date.
    **/
   public static void main(String[] args)
   {
      Date now = new Date();
      System.out.println(now.toString());
   }
}
```


Types

So far we have seen `Date`, `String`, and `OutputStream` objects.

Java also supports *primitive* types:

- boolean – either `true` or `false`.
- char – 16 bit unicode.
- byte – 8 bit signed.
- short – 16 bit signed.
- int – 32 bit signed.
- long – 64 bit signed.
- float – 32 bit signed.
- double – 64 bit signed.

Variables

Java has hybrid object model – not everything is an oject.

A variable value is either

1. A reference to an object (like C++ references) or `null`.

   ```
   Date now = new Date();
   System.out.println(now.toString());
   ```

2. A primitive type

   ```
   int i = 0;
   double pi = 3.14159;
   ```

Command Line Arguments

```
public class ReverseArgs
{
   public static void main(String[] args)
   {
      for (int i=args.length-1; i >= 0; i--)
         System.out.println(args[i]);
   }
}
```


Values

object values are references to data in the heap. Why?

primitive values are just values.

Rooted Class Hierarchy

All classes are descendents of `Object`.

Some inherited methods are:

`Object clone()` Makes shallow copy. No constructor called.

`boolean equals(Object)` Comparison of objects, default compares pointers.

`Class getClass()` Gets the runtime class of this object.

`String toString()` Converts object to a string.

Making Classes

```
public class Point
{
   private int x;
   private int y;

   public Point() { x = 0; y = 0; } // Constructor
   public int getX() { return x; }
   public int getY() { return y; }
}
```

`super` and `this`

Inside a method definition

super is a reference to the parent object.

this is a reference to the object itself.

```
public Point(int x, int y)
{
   this.x = x;
   this.y = y;
}
```

These variables can be used to avoid ambiguity between instance, and parameter variables with the same names.

These can be used in first line of constructors to specify parent constructors (with `super`) or alternative construtor (with `this`).

```
public Point() { this(0,0); }
```


Strings

Strings are a builtin class in Java, `java.lang.String`.

There is special `+` operator that does concatination of Strings:

```
String myString = "this" + "is" + "my" + "string";

Date now = new Date();
System.out.println("Today is: " + now);

System.out.println("I have " + 5 + "dollars.");
```

Method `int String.length()` returns string's length.

Strings are immutable. Why?

More Point Methods

Define the `toString()` method for `Point` so that

```
(new Point(5,6)).toString()
```

is "(5,6)"

Arrays

Arrays in Java are objects. They are 0-indexed.

```
int halfsize = 10;
int[] array = int[2*halfsize];

for (int i=0; i < array.length; i++)
   array[i] = i;


String[] strings1 = { "hi", "there" };

// or

String[] strings2 = new String[2];
strings2[0] = "hi";
strings2[1] = "there";
```

Multidimensional Arrays

Just arrays of arrays:

```
int[][] a = new int[3][];

a[0] = { 1, 2, 3 };
a[1] = null;
a[2] = { 4, 5 };

int[][] b = { { 1, 2, 3 }, null, { 4, 5 } };

System.out.println(a[2][0]);     // prints 3
```


Memory Model

All objects are in heap.

```
Date now = new Date();
Date alsoNow = now;
Date nowAgain = now.clone();

// compare pointers
boolean comp1 = (now == alsoNow);
boolean comp2 = (now == nowAgain);

// compare objects
boolean comp3 = (now.equals(alsoNow));
boolean comp4 = (now.equals(nowAgain));
boolean comp5 = (now.equals(new Date()));
```


Subclasses

Given the following class definitions

```
public class Bird
{
   public String tweet() { return "tweet"; }
}

public class Owl extends Bird
{
   public String tweet() { return "hoot"; }
   public String bob() { return "bob bob; }
}

public class Dog
{
   public String bark() { return "woof"; }
}
```


Type Sysyem

Java is *statically* and *dynamically* typed (C++ is only statically typed).

Consider: `Bird bird = new Owl();`

Static: each variable has a type.

- A variable of type `Bird` can only reference descendents of `Bird`.
- Only methods defined on `Bird`s can be invoked via a `Bird` variable. Legal: `bird.tweet()` Illegal: `bird.bob()`

Type System (cont)

Dynamic: Each object knows what type it really is.

- So `bird.getClass()` returns the `Class` object for `Owl`.
- Type casts tell compiler that object is really a subclass:

```
Owl owl = (Owl)bird;
owl.bob();
```

- Above cast only works if `bird` is really a reference to an `Owl` or a descendent of `Owl`. It is a *runtime exception* otherwise. E.g. `(Owl)(new Bird())`.

Static/Dynamic Type

The *static type* of an expression is the type the compiler associates with the expression.

The *dynamic type* of an expression is actual type of the value of the expression when the program is running.

Assignment is allowed if the *static type* of RHS is a descendent of the *static type* of LHS: `Bird bird = new Owl();`

The *static type* of a type-cast expression is the type of the cast.

A cast succeeds if *dynamic type* of expression is a descendent of the type of the cast. `(Owl)bird;`

Type System, E.g. (cont)

What are the results of each line?
Which lines cause *compile-time* or *run-time* errors?

```
Bird b1 = new Owl();
Bird b2 = new Bird();

b1.tweet();
b2.tweet();
b1.bob();

Owl o0 = b1;
Owl o1 = (Owl)b1;
Owl o2 = (Owl)b2;
Bird b3 = (Bird)(new Dog());
```


Operator `instanceof`

Consider

```
public String animate(Bird b)
{
   String out = b.tweet();
   if (b instanceof Owl)
      out = out + ", " + ((Owl)b).bob();
}
```

Functions, Methods, etc.

instance variables or *member fields*: Each instance of an object has its own copies.

member functions or *methods*: Functions invoked on instances of objects. Implicit `this` reference in method body refers to the object that the method was invoked on. `bird.tweet();`

Static variables or *class variables*: One per class. E.g. `System.out` or `Integer.MAX_VALUE` .

Static functions or *class functions*: No implicit `this` reference. Invoked like `<class name>.<function name>`. E.g. `Integer.parseInt("500")`

Static functions and static variables are declared with `static` modifier.

Methods: Scoping

Scoping:

- Parameter variable names shadow instance variable names. Use `this` to access instance variables.
- Superclass instance variables (and methods) are shadowed by subclass instance variables (and methods). Use `super` to access superclass variables.

Methods: Overloading

Overloading:

- Operators cannot be overloaded. I.e. `*` can only multiply numbers.
- Methods and static functions can be overloaded. The *signature* of a function is a combination of its name and its argument types.
- Overloading uses static types not dynamic types of arguments.

Methods: Overriding

Overriding:

- In C++ terms, all Java methods are *virtual*: All method invocations are *dynamically dispached*.
- A method in a subclass *overrides* same-signature method in superclass.
- Dangerous to combine overloading with overriding.

Overriding & Overloading (E.g.)

```
public class Bird
{
   public String tweet() { return "tweet"; }
   public String peck(Bird b)
      { return "A bird pecks a bird"; }
   public String peck(Owl o)
      { return "A bird pecks an owl"; }
}

public class Owl extends Bird
{
   public String tweet() { return "hoot"; }
   public String bob() { return "bob bob"; }
   public String peck(Bird b)
      { return "An owl pecks a bird"; }
   public String peck(Owl o)
      { return "An owl pecks an owl"; }
}
```


Overriding & Overloading (cont)

What are the results of?

```
Bird b1 = new Bird();
Bird b2 = new Bird();
Bird bo1 = new Owl();
Bird bo2 = new Owl();
Owl o1 = new Owl();
Owl o2 = new Owl();


b1.peck(b2);
b1.peck(bo2);
b1.peck(o2);

o1.peck(b2);
o1.peck(bo2);
o1.peck(o2);
```

```
bo1.peck(b2);
bo1.peck(bo2);
bo1.peck(o2);
```


Subclassing

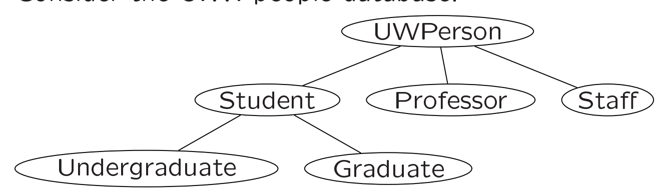
Use `extends` keyword.

Subclasses must define their own constructors. If not, a default one will be used. If any constructor is defined, the default constructor will not be used.

A class can have only one parent class – no *multipule inheritance*.

Multipule Inheritance?

Consider the U.W. people database:



Who teaches classes/sections? Graduates and Professors do:

```
public class Graduate extends Student
{  ...
   public void teach(Course c) {...}
}
public class Professor extends UWPerson
{  ...
   public void teach(Course c) {...}
}
```

Multipule Ineritance (cont)

How can we register teachers to teach courses?

```
public class Course
{
   ...
   public void registerTeacher(UWPerson p)
   {
      p.teach(this);
   }
}
```

Whats wrong?

Multipule Ineritance (cont)

Second try:

```
public void registerTeacher(UWPerson p)
{
   if (p instanceof Professor)
   {
      Professor prof = (Professor)p;
      prof.teach(this);
   }
   else if (p instanceof Graduate)
      ((Graduate)p).teach(this);
}
```

Works, but clunky.

Interfaces

While multipule-inheritance might be a bad idea, but the idea of having variable for objects that have the `teach(Course)` method is useful.

Solution: *interfaces*

A java class can implement any number of interfaces.

```
public interface Teacher
{
   public void teach(Course c);
}
```

Any class that *implements* this interface can be put in a `Teacher` variable.

Any class that implements `Teacher` must define its own `teach(Course)` function.

Interfaces (E.g.)

```
public class Graduate extends Student implements Teacher
{
   ...
   public void teach(Course c) {...}
}

public class Professor extends UWPerson implements Teacher
{
   ...
   public void teach(Course c) {...}
}
```

registerTeacher

Now we can define registerTeacher as:

```
public class Course
{
   ...
   public void registerTeacher(Teacher t)
   {
      t.teach(this);
   }
}
```

It can be called as:

```
Course c1 = ...;
Course c2 = ...;
Professor p = ...;
Graduate g = ...;

c1.registerTeacher(p);
c2.registerTeacher(g);
```


Enumeration Interface

Consider java.util.Enumeration:

```
public interface Enumeration
{
   public boolean hasMoreElements();
   public Object nextElement();
}
```


Polymorphism

Interfaces provide means of *polymorphism*.

A *polymorphic* function is one that can by applied to arguments of different types:

```
public static void printEnumeration(Enumeration e)
{
   while (e.hasMoreElements())
      System.out.println(e.nextElement());
}
```

java.util.StringTokenizer implements Enumeration:

```
StringTokenizer st = new StringTokenizer("this is a test");
printEnumeration(st);
```

You can get an Enumeration from a java.util.Vector:

```
Vector v = ...;
printEnumeration(v.elements());
```


IntegerRange Class

```
public class IntegerRange implements Enumeration
{
   int min; int max;
   public IntegerRange(int minIn, int maxIn)
   {
      min = minIn;
      max = maxIn;
   }
   public boolean hasMoreElements()
   {
      return min <= max;
   }
   public Object nextElement()
   {
      return new Integer(min++);
   }
}
```

Subclasses and Interfaces

Subclassing:

- Inheritance of code
- Polymorphism

Inferfaces:

- Polymorphism

Abstract Classes

Sometimes it does not make sense ever instanciate a class, E.g. `UWPerson`.

Sometimes we want to specify part of an implementation but not all of it.

Classes declared `abstract` can not be instantiated.

Abstract classes may declare abstract methods that non-abstract subclasses must implement.

InputStream

```
public abstract class InputStream
{
   // simple default implementations.
   public int available() { return 0; }
   public void close() {}
   public boolean markSupported() { return false; }
   public void reset() {}

   // abstract read() method, overridden by subclass
   // depending on its input source.
   public abstract int read() throws IOExcption;

   // default implementations that use read()
   public int read(byte[] b, int off, int len)
   {
      ...
      int b = read();
      ...
   }
   public int read(byte[] b) throws IOException
      { return read(b, 0, b.length); }
   public long skip(long n)
      { ... }
}
```


StringInputStream

```
public class StringInputStream extends InputStream
{
   private byte[] data;
   private int length;
   private int cursor;

   public StringInputStream(String in)
   {
      data = in.getBytes();
      length = in.length();
      cursor=0;
   }

   public int read()
   {
      if (cursor >= length)
        return -1;

      return (int)data[cursor++];
   }
}
```

Abstract Classes – Summary

Abstract `InputStream` class made `StringInputStream` easy to create.

`StringInputStream` only had to implement `int read()` method.

`StringInputStream` inherited `read(byte[])` and `read(byte[], int, int)`.

Inherited `read(byte[],int,int)` used `StringInputStream`'s `int read()` method.

Generic Collections

All classes are subclasses of `Object`.

To make a collection of any type use `Object` as type.

E.g. `java.util.Vector` and `java.util.Hashtable`.

It is easy to insert elements:

```
Vector v = new Vector();
v.add(new Bird());
```

But, cast is required to remove elements:

```
Bird b = (Bird)v.elementAt(0);
```

Java does not provide generic support like C++'s *templates* (only arrays).

Primitive Wrappers

Any `Object` can be put in a `Vector`.

Java primitive `int`s are not `Objects`.

Can not have a `Vector` of `int`.

Can *wrap* `int` in `Integer` class:

```
Vector v = new Vector();
v.add(new Integer(5));

int i = ((Integer)(v.lastElement())).intValue();
```

Wrapper classes also have useful class functions and class variables: E.g. `Integer.parseInt(String)` and `Integer.MAX_VALUE`.

ReverseFile

```
public class ReverseFile
{
   public static void main(String[] args)
   {
      try
      {
         BufferedReader in
            = new BufferedReader(new FileReader(args[0]));
         Stack s = new Stack();

         String line;
         while ((line = in.readLine()) != null)
            s.push(line);

         while (!s.empty())
            System.out.println((String)s.pop());
      }
      catch (Exception e)
      {
         e.printStackTrace(System.out);
      }
   }
}
```

Exception Handling

Can use *exceptions* instead of having functions return some error status.

Constructors can throw exceptions if they fail (no return value possible).

Can seperate error handling code from general case code.

For `ReverseFile` make things could go wrong:

- `args` could be empty,
- `args[0]` might not contain a file name, or
- an IO error could happen reading the file.

`ConsecutiveIntegers` Class

```
public class ConsecutiveIntegers1
{
   int min, max;
   public ConsecutiveIntegers1(int minIn, int maxIn)
      { min = minIn; max = maxIn; }

   public int size() { return(max - min + 1); }

   public int intAt(int i)
                 throws ArrayIndexOutOfBoundsException
   {
      if (i < 0 || i >= size())
         throw new ArrayIndexOutOfBoundsException();

      return min+i;
   }

   public Object elementAt(int i)
                 throws ArrayIndexOutOfBoundsException
      { return new Integer(intAt(i)); }

   public Enumeration elements()
      { return new IntegerRange(min,max); }
}
```


Inner Classes

Inner classes are private classes nexted inside other classes.

Inner classes have “outer” class’s instance variables and methods in their scope.

Inner classes have access to private instance variables.

Reconsider `ConsecutiveIntegers.elements()`

`ConsecutiveIntegers.elements()`

```
public class ConsecutiveIntegers2
{
   int min, max;

   ...

   private class OurEnumeration implements Enumeration
   {
      int current = 0;
      public boolean hasMoreElements()
      {
         return (current < size());
      }

      public Object nextElement()
      {
         return elementAt(current);
      }
   }

   public Enumeration elements()
      { return new OurEnumeration(); }
}
```

Anonymous Classes

Inner classes can be *anonymous* extending either an object or an interface:

```
public class ConsecutiveIntegers3
{
   int min, max;
   ...
   public Enumeration elements()
   {
      return(new Enumeration() {
         int current = 0;
         public boolean hasMoreElements()
         {
            return (current < size());
         }

         public Object nextElement()
         {
            return elementAt(current);
         }
      });
   }
}
```


Reflection

Java *reflection* is the ability to dynamically refer to the Java class system inside a Java program.

A language construct is *first class* if it can be stored in variables, passed to functions, and returned from functions.

Java reflection via `java.lang.Class` and `java.lang.reflect.*`
(We will discuss other OO *metaclass* systems later)

Two dynamic ways to get a `Class` object:

```
Object someObject = "hello";
String className = "java.lang.String";

Class c1 = Class.forName(className);
Class c2 = someObject.getClass();
```

One static way to get a `Class` object:

```
Class c3 = String.class;
```


Using a `Class`

Some useful functions are:

`Static Class forName(String name)` Returns `Class` object for `name`.

`Class[] getInterfaces()` Returns array of interfaces implemented by this class.

`String getName()` Returns complete class name:
E.g. `String.class.getName()` returns `"java.lang.String"`.

`boolean isInstance(Object o)` Returns `true` if `o` is an instance of this class. A dynamic version of `instanceof`.

`boolean isAssignableFrom(Class c)` Returns `true` if `c` is more specific than this class.
E.g. `Bird.class.isAssignableFrom(Owl.class)` is `true`.

`Object newInstance()` Creates a new instance of this class by calling the default constructor.

`java.lang.reflect.*`

java.lang.reflect.Method Provides abstraction for method selection and invocation. Can get `Method` objects.

java.lang.reflect.Field Provides abstraction for field setting and getting.

java.lang.reflect.Constructor Provides abstraction for invoking constructors (especially non-default constructors).

Advantages of Reflection

Applications that can talk about themselves are highly customizable. Consider:

```
public abstract class Filter
{
   public abstract String filterString(String in);
}
```


FilterMain

```
public abstract class FilterMain
{
   public static void main(String[] args)
   {
      try
      {
         Filter f = (Filter)Class.forName(args[0]).newInstance();
         BufferedReader in
            = new BufferedReader(new InputStreamReader(System.in));

         String line;
         while((line = in.readLine()) != null)
         {
            System.out.print(f.filterString(line));
         }
         System.out.print(f.filterString(null));
      }
      catch (Exception e)
      {
         e.printStackTrace(System.err);
      }
   }
}
```


FirstTenFilter

```
public class FirstTenFilter extends Filter
{
   int count = 0;
   public String filterString(String line)
   {
      if (count++ < 10)
         return line + "\n";
      return "";
   }
}
```

Running `java FilterMain FirstTenFilter` is the same as running `head -10` from the UNIX prompt.

WordCountFilter

```
public class WordCountFilter extends Filter
{
   int numWords = 0, numLines = 0, numChars = 0;

   public String filterString(String line)
   {
      if (line != null)
      {
         numLines++;
         numChars += line.length() + 1;
         numWords += (new StringTokenizer(line)).countTokens();
         return "";
      }
      else
         return("\t" + numLines + "\t" + numWords
                     + "\t" + numChars + "\n");
   }
}
```

Threads

A modern operating system runs many processes (running programs) at once.

Processes can communicate through IPC (interprocess communication).

These "heavy weight" processes are protected from each other by the operating system.

Often times it is useful to write programs as if they were multiple processes. But protection of operating system is not needed and overhead is too much.

Threads are *light weight processes*.

A process can have many running threads. Threads share the processes memory. Each thread has its own call stack.

Threads vs State-Machines

Consider animation of unfriendly creatures in Doom-like game.

state-machine: At each clock tick animate one step of creature movement. Must have kept state to determine where creature currently is. Then must generate next movement.

```
Creature[] creatures = ...;

while(alive())
{
   for(int i=0; i < creatures.length; i++)
      creature.animate();
}
```

This works, but gets complicated very quickly as creatures need more diverse sequences of moves.

Threads vs State-Machines (cont)

threads: Each creature is run by its own thread. At each clock tick creature thread gets awakened. Creature thread can do whatever it wants. Creature thread puts itself to sleep when it has finished its turn. Creature's state is stored on the creature thread's call stack.

```
Creature[] creatures = ...;
for(int i=0; i < creatures.length i++)
   creatures.start();  // start creature threads.

while(alive()) {
   nofityAll();  // this notifies all threads waiting
}                // on 'this' object.
```

This assumes that each creature puts itself to sleep by *waiting* on `this` object.

Threads: Synchronization

When there are multiple threads in the system running at once, sequentially written code might not work as it originally did.

Threads can trample on each other. Given a code block:

```
x = x + 1;
```

Say $x = 10$ and two threads run this block at the same time.

x	Thread 1	Thread 2
10	`x+1` ⇒ 11	
10		`x+1` ⇒ 11
10		`x = 11`
11		
11	`x = 11`	
11		

The code executed twice and the value of `x` is only `11`. It should be `12`.

Synchronization (cont)

(learn more about synchronization in Operating Systems)

A method can be synchronized on `this`. A block of code can be synchronized on any object.

The object that a block of code or method is synchronized on holds the *lock* on that block of code. No other thread can enter a synchronized block that uses the same object as its lock.

```
public synchronized int incX()
{
   x = x + 1;
   return x;
}
```

or

```
synchronized (<expression>)
{
   <code>
}
```


Threads and Methods

Threads of control are an orthogonal issues to classes. Threads make method invocations on objects.

Any thread can invoke any method on any object.

Where to use Threads

Good places to use Threads include:

- GUI Animation.
- Server Applications.
- IO intensive Applications.

Why? Because threads allow you to easily do multiple things roughly at the same time.

Abstract Window Toolkit

The Java JDK provides the AWT writing GUI applications or applets.

Applications extend `java.awt.Frame`.

Applets extend `java.applet.Applet`.

Subclasses of `java.awt.Component` can be used to build complicated interfaces.
E.g. `Button`, `Label`, `Scrollbar`, `Panel`.

Any `Panel` can be configured with a `LayoutManager`.
E.g. `BorderLayout`, `GridLayout`.

Any `Component` can be added to the `Panel` in the given layout.

The *event handler* is a *thread* that manages user events such as mouse clicks and keystrokes.

Event `Listener`s can be registered with a `Component`.

`ColorFrame`

```
import java.awt.*; import java.awt.event.*;

public class ColorFrame extends Frame
{
   Canvas c;

   public static void main(String[] args)
   {
      Frame f = new ColorFrame();
      f.show();
   }

   public ColorFrame()
   {
      setSize(200,200);

      c = new Canvas();
      c.setBackground(new Color(0,0,0));

      Scrollbar scrollBar = new Scrollbar(Scrollbar.VERTICAL,0,5,0,260);

      setLayout(new BorderLayout());
      add("Center", c);
      add("East", scrollBar);
```


`ColorFrame` (cont)

```
      /* listener for scrollbar events */
      scrollBar.addAdjustmentListener(new AdjustmentListener() {
        public void adjustmentValueChanged(AdjustmentEvent e)
         {
            c.setBackground(new Color(e.getValue(),0,0));
         }
      });

      /* add event listener to quit when window is closed */
      addWindowListener(new WindowAdapter() {
         public void windowClosing(WindowEvent e)
         {
            System.exit(0);
         }
      });
   }
}
```


Highlights

Main program first instantiates the frame then shows it.

A `Frame` *is a* `Panel` which *is a* `Component`.

Inner classes for event handling are very useful.

An event listener either needs to implement the `*Listener` interface or extend the `*Adapter` object.

Creating Components

GUI AWT interfaces are built up by combining `Components`.

Not every component you might want is implemented for you.

When implementing your own component:
Make your component behave as other components behave

An Extended Example

Looking at example component that demonstrates:

- the *Java event model*,
- use of `Components`, and
- good Object Oriented design principles.

`ValueSlider` Example

Say we want a scrollbar that has an editable text area.

This new scrollbar should be have just like `java.awt.Scrollbar`s do:

- It implements the `Adjustable` interface.
- It allows registration of an `AdjustmentListener`.

Additionally, its value can be changed by editing the text area.

Extending `Panel`

To make a composite component, extend `Panel`

```
public class ValueSlider extends Panel
{
   public TextField field;
   public Scrollbar slider;
   public int value;

   public ValueSlider(int orient, int init, int step, int min, int max)
   {
      value = init;

      setLayout(new BorderLayout());

      slider = new Scrollbar(orient, init, step, min, max);
      field = new TextField(Integer.toString(init));

      add("Center",slider);
      add("South",field);
   }
}
```


`ValueSlidersFrame`

```
public class ValueSlidersFrame extends Frame
{
   public ValueSlidersFrame()
   {
      setSize(150,500);

      setLayout(new GridLayout(1,3));

      add(new ValueSlider1(Scrollbar.VERTICAL,0,5,0,260));
      add(new ValueSlider2(Scrollbar.VERTICAL,0,5,0,260));

      ValueSlider3 vs3 = new ValueSlider3(Scrollbar.VERTICAL,0,5,0,260);

      add(vs3);

      vs3.addAdjustmentListener(new AdjustmentListener() {
         public void adjustmentValueChanged(AdjustmentEvent e)
         {
            System.out.println(e.getValue());
         }
      });
   }
}
```

Setting the Value

We want to be able to set the slider value:

```
public void setValue(int val)
{
   slider.setValue(val);
   value = slider.getValue();
   field.setText(Integer.toString(value));
}

public int getValue()
{
   return value;
}
```


Listening for Events

Need to listen for events on `field` and `slider`.

```
public class ValueSlider
                extends Panel
                implements AdjustmentListener, ActionListener
{
   ...
   public ValueSlider(...)
   {
      ...

      slider.addAdjustmentListener(this);
      field.addActionListener(this);
   }
}
```


Implement Listener Methods

Must implement `adjustmentValueChanged(...)` and `actionPerformed(...)`

```
public void actionPerformed(ActionEvent e)
{
   try {
      setValue(Integer.parseInt(field.getText()));
   } catch (NumberFormatException nfe) {
      setValue(value);
   }
}

public void adjustmentValueChanged(AdjustmentEvent e)
{
   setValue(e.getValue());
}
```


Allow `AdjustmentListeners`

Need to allow `AdjustmentListeners` to be added to listen to this slider's events.

The `AWTEventMulticaster` can be used to make a single `Listener` out of two `Listeners`. It is a `Listener` itself!

When it gets an event, it will send it to both listeners:

```
protected AWTEventMulticaster(EventListener a, EventListener b);
```

```
static public AdjustmentListener add(SomeListener a, SomeListener b);
static public AdjustmentListener remove(SomeListener l, SomeListener oldl);
```

Where `Some` is `Adjustment`, `Action`, etc.

Allowing Listeners (cont)

```
public class ValueSlider
                 extends Panel
                 implements AdjustmentListener, ActionListener, Adjustable
{
   AdjustmentListener listener = null;

   ...

   public void addAdjustmentListener(AdjustmentListener l)
   {
      if (listener == null)
         listener = l;
      else
         listener = AWTEventMulticaster.add(listener,l);
   }

   public void removeAdjustmentListener(AdjustmentListener l)
   {
      if (listener != null)
         listener = AWTEventMulticaster.remove(listener,l);
   }
}
```


Dispatching Events

```
   public void dispatchEvent(int value)
   {
      if (listener != null)
         listener.adjustmentValueChanged(
            new AdjustmentEvent(
                   this,
                   AdjustmentEvent.ADJUSTMENT_VALUE_CHANGED,
                   AdjustmentEvent.TRACK,
                   value));
   }
```

Need to call `dispatchEvent(...)` at end of `actionPerformed(...)` and `adjustmentValueChanged(...)`

Also need to implement rest of methods in `Adjustable` interface. Each should just call corresponding method on `slider`. E.g.

```
public int getMinimum()
   { return slider.getMinimum(); }
```


Result

Now we can use `ValueSlider`s where ever we previously used `Scrollbars`.

As an exercise, modify `ColorFrame` to use the `ValueSlider`.

Access Modifiers

Default permission on classes, variables, and methods is *package protection*.

`private` – Only this class can access.

`protected` – Only this class or subclasses can access.

`public` – Any class can access.

Other Mofifiers

`final` class – subclassing is not allowed.

`final` method – overriding is not allowed.

`final` variable – can not be changed (like C's `const`).

Recall: `abstract`, `static`, and `synchronized`.

There are a few others.

Java Packages

Java classes can be grouped into packages.

Full name of class is: `<package-name>.<class-name>`

```
package my.geometry;

public class Point
{
  ...
}
```

Full name of `Point` is `my.geometry.Point`.

Can, import `my.geometry.*` or `my.geometry.Point` or just use `my.geometry.Point` in code.

Compiled class will be in `my/geometry/Point.class`

Packages provide management of name space – very useful in large software projects.

Object Oriented Metaphor

Objects have state.

Methods are messages sent to objects.

Objects interpret messages as they choose.

Object Oriented Design

- Subclassing
- Code Inheritance
- Polymorphism
- Information Hiding
- Encapsulation
- Frameworks
- Patterns

Subclassing

Many reasons to subclass:

specialization – child is special case of parent.
E.g. `Number`, `Integer`, `Double`.

specification – behavior is implemented by child.
E.g. `InputStream`, `FileInputStream`; Event `Adapter`s.

extension – child provides more functionality.
E.g. `Panel`, `Applet`.

Code Inheritance

Subclasses inherit methods from their parents.

Can make small changes to the behavior of a class by

- subclassing, and then
- overriding a few methods.

Protected interface can provide easier access to superclass.

Example: Hashtable resizing algorithm.

Polymorphism

Can make functions that take many types of objects.

Good for code reuse.

Provides *extensibility*: a subclass with different behavior can be substituted for parent class.

Example: FilterMain and Filters

Information Hiding

Provide *public interface* to object, but hide implementation.

- Clients do not need to know implementation details.
- Implementers do not need to know how object will be used.

Example: Dictionary

Languages can enforce information hiding via access permissions:
`public`, `private`, etc.

Encapsulation

Encapsulate simple objects inside composite object.

Example: `ValueSlider`.

Encapsulation and Information Hiding often go hand in hand: Access to encapsulated objects is not allowed, access is only allowed through composite object.

Frameworks

Frameworks provide implementation of high-level interaction of objects in the system.

Applications implement details.

Example, Smalltalk's Model-View-Controller (MVC).

Can use abstract classes to specify framework. Subclasses would implement the details.

Model-View-Controller

Model knows about data and how to manipulate data.

View knows how to display the data. Queries model to get data.

Controller provides interface for user to manipulate the model and the view.

Advantages:
- Different controllers can provide different interfaces for manipulating the same data.
- Multiple views can be provided for the same model.

Example: Clock

Patterns

A *pattern* is a "well-proven generic scheme for solving a recurring design problem."

- Language Independent
- Come from examples in existing systems.
- Like high-level control structures in languages.

Examples: Flyweight, Adapter, Observer

Flyweight

Reduces memory usage for many objects with same state.

Solution: let objects share state.

Example: Scheme's Symbols, Playing Cards

Adapter

Client needs one interface, existing object provides another interface.

Solution: Wrap existing object in an adapter that has desired interface.

Client communicates with adapter, adapter communicates with object.

Observer

Several objects need to stay consistent with data.

Solution: Have list of objects *observing* data, when data is changed, all observing objects can be notified.

Similar to `Listeners` for `Events` in Java's AWT.

Subtypes in OO

(not Java specific)

Recall, variables can hold "more specific" instances:
`Bird b = new Owl()`

Everything that can be done to/with a `Bird` can be done to/with an `Owl`.

This defines a *subtype* relationship (similar to *subclass*).

type (in OO language) – A collection of operation signatures.

operation signatures – the operation name, the types of the arguments, and the type of the result.

Several sub-type rules: *contravariant rule* and *covariant rule*

Consider `Bird[]` and `Owl[]`. What is subtype relationship?

Subtype Rules

(not Java specific – Java notation abused)

Arrays have two operations, *assignment* and *lookup* (`array[i] = 10` and `x = array[i]`.

Lets view these as methods on array objects:

```
Value Array.elementAt(int index);
void Array.setElementAt(Value v, int index);
```

Recall, that you should be able to do anything with a subtype that you can do with the original type.

Again consider `Bird[]` and `Owl[]`. We have:

```
Bird Bird[].elementAt(int index);
void Bird[].setElementAt(Bird b, int index);
```

and

```
Owl Owl[].elementAt(int index);
void Owl[].setElementAt(Owl o, int index);
```


`Owl[]` and `Bird[]`

Suppose `Owl[]` is a subtype of `Bird[]`

```
Bird[] owls = new Owl[10];
Bird[] birds = new Birds[10];

birds.setElementAt(new Bird(),0);     // good
owls.setElementAt(new Bird(),0);      // not good
```

Suppose `Bird[]` is a subtype of `Owl[]`

```
Owl[] owls = new Owl[10];
Owl[] birds = new Birds[10];

owls.elementAt(0).bob();              // good
birds.elementAt(0).bob();             // not good
```


Co[ntra]variance

Possible type relationships between S and T:

- S is a subtype of T.
- T is a subtype of S.
- S is not related to T.

Given operation signatures: R_s S`.foo`(P_s) and R_t T`.foo`(P_t)

Contravariant rule: S is subtype of T if R_s is subtype of R_t and P_t is subtype of P_s.

Covariant rule: S is subtype of T if R_s is subtype of R_t and P_s is subtype of P_t.

What is type relationship of `Bird[]` and `Owl[]`?

Recall Bird/Owl

```
public class Bird
{
   public String tweet() { return "tweet"; }
   public String peck(Bird b)
      { return "A bird pecks a bird"; }
   public String peck(Owl o)
      { return "A bird pecks an owl"; }
}

public class Owl extends Bird
{
   public String tweet() { return "hoot"; }
   public String bob() { return "bob bob"; }
   public String peck(Bird b)
      { return "An owl pecks a bird"; }
   public String peck(Owl o)
      { return "An owl pecks an owl"; }
}
```

Recall Bird/Owl (cont)

Recall dynamic dispatch:

```
Bird owl1 = new Owl();
Bird owl2 = new Owl();

owl1.tweet();

owl1.peck(owl2);
```

This produces non-symmetric results.

`Object.equals(Object)`

```
public class Point
{
   int x; int y;
   public boolean equals(Object o)
   {
      if (o instanceof Point)
         return false;
      Point p = (Point)o;
      return (p.x == x && p.y == y);
   }
}

public class ColorPoint extends Point
{
   Color c;
   public boolean equals(Object o)
   {
      if (o instanceof ColorPoint)
         return false;
      ColorPoint p = (ColorPoint)o;
      return (super.equals(o) && c.equals(p.c));
   }
}
```


`Object.equals(Object)` (cont)

What are the results of:

```
Point      p1 = new Point(5,10);
Point      p2 = new Point(5,10);
Point     cp1 = new ColorPoint(5,10,Color.red);
Point     cp2 = new ColorPoint(5,10,Color.red);
ColorPoint c1 = new ColorPoint(5,10,Color.red);
ColorPoint c2 = new ColorPoint(5,10,Color.red);

p1.equals(p2);
p1.equals(cp2);
p1.equals(c2);

cp1.equals(p2);
cp1.equals(cp2);
cp1.equals(c2);
```

But `equals()` should be symmetric!!

Multimethods

What we need is *multimethods*!

Change notation of `Object.equals(Object)` to `equals(Object, Object)`.

`Dynamically dispatch` on the runtime types of both `Object`s.

(*non-java code*)

```
boolean equals(Point p, Point q)
{
   return (p.x == q.x && p.y == q.y);
}

boolean equals(ColorPoint p, ColorPoint q)
{
   return (resend && equals(p.color,q.color));
}
```

Multimethods (cont)

Method is chosen at runtime that best matches runtime type of arguments:

```
Point p1 = new Point(5,10);
Point p2 = new Point(5,10);
Point c1 = new ColorPoint(5,10,Color.black);
Point c2 = new ColorPoint(5,10,Color.black);

equals(p1,p2);
equals(p1,c2);
equals(c1,p2);
equals(c1,c2);
```


Multimethod Summary

Good for: arithmetic operations, comparisons, etc.

Examples: `display(Shape, OutputDevice)`

Multimethods dispatch on the runtime types of all arguments.

Methods called is most specific one that matches arguments.

Cecil supports multimethods.

Simulating Double-Dispatch

(*this is Java code*)

```
public class Point
{
   public boolean equals(Point p)
      { return p.equalsPoint(this); }
   public boolean equalsPoint(Point p)
      { return(p.x == x && p.y == y); }
   public boolean equalsColorPoint(ColorPoint p)
      { return equalsPoint(p); }
}

public class ColorPoint
{
   public boolean equals(Point p)
      { return p.equalsColorPoint(this); }
   public boolean equalsColorPoint(ColorPoint p)
      { return (equalPoint(p) && p.c.equals(c)); }
}
```

Note: typically we would define `boolean equals(Object)`.

Double-Dispatch (cont)

What are the results of?

```
Point      p1 = new Point(5,10);
Point      p2 = new Point(5,10);
Point     cp1 = new ColorPoint(5,10,Color.red);
Point     cp2 = new ColorPoint(5,10,Color.red);
ColorPoint c1 = new ColorPoint(5,10,Color.red);
ColorPoint c2 = new ColorPoint(5,10,Color.red);

p1.equals(p2);
p1.equals(cp2);
p1.equals(c2);

cp1.equals(p2);
cp1.equals(cp2);
cp1.equals(c2);
```

Smalltalk

Object-oriented.

All objects in heap.

Rooted class hierarchy.

Only dynamically typed (unlike Java).

Everything is an object: `5`, `nil`, `True`.

First-class closures – like `lambda` in Scheme.

Everything is a message send (method invocation) even control blocks.

Programmer can change any class in hierarchy!

Looks a bit like English.

Message Sends

Unary Messages:

```
Date today
Time now hours
Array new
```

Keyword Messages:

```
array new: 10
someArray at: 5 put: 10
```


Understanding Messages

Consider: `Time now hours + 1`

- Send `now` message to `Time` to get current time.
- Send *current time object* message `hours` to get the current hour (an integer).
- Send the message `+` with argument `1` to this integer.

Blocks

Blocks of code in Smalltalk are *first class*!!
(like `lambda` in Scheme)

```
| c1 c2  x |                "declare variables"
x  := 0.
c1 := [ x := x+1 ].         "c1 is a block"
c2 := [ :i | x := x+i ].    "c2 is a block w/ one param"

c1 value.                   "evaluate c1"
c2 value: 20.               "evaluate c2, with the argument 20"

x.                          "now x equals 21... "
```

Compare to Scheme

```
(let ((c1 0) (c2 0) (x 0))
     (set! x 0)
     (set! c1 (lambda ()  (set! x (+ x 1))))
     (set! c2 (lambda (i) (set! x (+ x i))))
     (c1)
     (c2 10)
     x
)
```


Booleans

Class `Boolean` has two subclasses `True` and `False`

```
True

   printOn: stream
      stream nextPutAll: 'true'

   & b   "evaluating and"
      ^ b

   | b   "evaluating or"
      ^ true

   not
      ^ false

   and: block   "short-circuit and"
      ^ block value

   or: block   "short-circuit or"
      ^ true
```


False

```
False

   printOn: stream
      stream nextPutAll: false'

   & b   "evaluating and"
      ^ false

   | b   "evaluating or"
      ^ b

   not
      ^ true

   and: block   "short-circuit and"
      ^ false

   or: block   "short-circuit or"
      ^ block value
```


Using Booleans

Consider the following:

```
(3=4) & (2>1)

(3=4) | (2>1)

(3=2) not

true & false not

(3=4) and: [(1/0) = 8]
```

Conditionals

```
True

   ifTrue: block
      ^ block value

   ifFalse: block
      ^ nil

   ifTrue: tBlock ifFalse: fBlock
      ^ tBlock value


False

   ifTrue: block
      ^ nil

   ifFalse: block
      ^ block value

   ifTrue: tBlock ifFalse: fBlock
      ^ fBlock value
```


Using Conditionals

Consider the following:

```
3=4 ifTrue: [x := 10].

x=y ifTrue: [x := 8] ifFalse: [x := 9].

x := x=y ifTrue: [8] ifFalse: [9].
```


Iteration

Blocks and boolean classes used to implement iteration also:

```
a := 1.
[a < 10] whileTrue: [Transcript show: a printString.  a := a+1].

a := 1.
[a > 10] whileFalse: [Transcript show: a printString.  a := a+1].

1 to: 10 do: [:x | Transcript show: x printString].
```


Defining classes

Defining a class is just sending a message to its parent class object:

```
Object subclass: #Stack
   instanceVariableNames: 'anArray top'
   classVariableNames: ''
   poolDictionaries: ''
```

Stack Methods

```
Stack

   push: item
      top := top+1.
      anArray at: top put: item


   pop | item |
      item := anArray at: top.
      top := top-1.
      ^ item

   setsize: n
      anArray := Array new: n.
      top := 0.

   isEmpty
      ^ top=0
```


Using the Stack

Consider:

```
S := Stack new.
S setsize: 10.
S inspect.
S push: 'hi there'.
S push: 3.14159.
S pop.
```


Summary of Smalltalk

All objects in heap.

Dynamically typed.

Everything is an object: `5`, `nil`, `True`.

Blocks are first-class closures – like `lambda` in Scheme.

Everything is a message send:
- Control blocks: conditionals, loops.
- Class instantiation.
- Subclassing.

Programmer can change any class in hierarchy!