Inheritance and subclasses

- a way to create or describe one class in terms of another
  - "a D is like a B, with these extra properties..."
  - "a D is a B, plus..."
  - B is the base class or superclass
  - D is the derived class or subclass
    - Perl, C++ use base/derived; Java uses super/sub
- inheritance is used for classes that model strongly related concepts
  - objects share some properties, behaviors, etc.
  - and have some properties and behaviors that are different
- real-world example: GUI "widgets" or "controls"
  - aspects common to all widgets:
    - position, size, background color, caption, ...
  - aspects different for different kinds of widgets:
    - how to draw, responses to events, ...
- one class is a natural extension of the other
  - sometimes you care about the difference
    - drawing: a button is not a pull-down menu is not a text area
  - sometimes you don’t
    - set/get caption, set background color, get dimensions

C-style Widget

- a struct to hold the data
- functions to create, modify, etc.
- a type field & conditional code to distinguish different types

```c
struct Widget {
  int type;         // what kind of widget
  int bgcolor;
  Rect position;
  ...
};

setbgcolor(int col);
setcaption(string);
draw();
getwidth();
...

Widget *but = new Widget(BUTTON);
but->setbgcolor(0xFF00FF);
but->draw(); ...
```
Problems with this approach

- each function (such as draw()) has to have code for all possible widgets

```c
void draw(Widget *wp) {
    switch (wp->type) {
        case BUTTON: ...  
        case SCROLLBAR: ...
        case MENU: ... // etc
    }
}
```

- code for handling each type is scattered all over
- each piece has to be changed when a new kind of widget is added

Subclasses

```c
class Widget {
    int bgcolor;  // other vars common to all Widgets
};
```
```c
class Button extends Widget {
    int state;    // other vars specific to Buttons
};
```
```c
class Scrollbar extends Widget {
    int min, max, current;  // other vars specific to Scrollbars
}
```

- a Button is a subclass (a kind of) Widget
  - inherits all members of Widget
  - adds its own members
- a Scrollbar is also a subclass of Widget
More subclasses

- subclasses can add their own data members
- can add their own member functions
- can override superclass functions with functions of same name and argument types

```java
class Scrollbar extends Widget {
    int min, max, current;
    public void draw() {...} // overrides
    public void setslider(int) {...}
}

class CheckButton extends Widget {
    boolean checked;
    public void draw() {...} // overrides
    public void setstate(boolean) {...}
}

    CheckButton b;
    Scrollbar s;

    b.draw(); // call CheckButton.draw
    s.draw(); // call Scrollbar.draw
```

Inheritance and subclasses

- example: a PerlRE class derived from RE
- adds Perl RE capabilities like richer REs and matching substrings

```java
public class PerlRE extends RE {
    // maybe some internal representation of RE
    String[] matches; // maybe matched strings

    public int match(String s) {
        // do Perl matching
        return 1;
    }

    public PerlRE(String s) {
        super(s); // invoke parent constructor
        // maybe compile s into internal rep
    }

    public static void main(String[] args) {
        PerlRE pre = new PerlRE("a|b|c");
        pre.match("abc");
        RE re = pre;
        re.match("bcd");
    }
}
```
Object hierarchy

- all objects are derived from class Object
  - e.g., an RE is an Object
  - a PerlRE is an RE is an Object

Object -> RE -> PerlRE
  -> Math
  -> System
  -> Component -> Container -> Panel -> Applet
  -> Button
  -> Label
  -> etc.
  -> InputStream -> FilterInputStream
  -> BufferedInputStream

- Object has methods for equals, hashCode, toString, clone, etc.
  - normally these are extended
- default RE.equals is Object.equals
  - tests for same reference, i.e., same object
- to compare for equal regexp, overload equals
  
```java
class RE {  // one defn of equality
    public boolean equals(RE r2) {
        return re.equals(r2.re);
    }
}
```

Overriding, dynamic method lookup

- PerlRE.equals() overrides RE.equals()
  - arguments are identical
- all functions are implicitly virtual: a reference to the superclass calls the subclass method for a subclass object

PerlRE pre = new PerlRE("a|b|c");
System.out.println("Perl RE = " + pre.match("a");

RE re = pre;
System.out.println("RE = " + re.match("a");
    // calls pre.match()
Virtual Functions

- what if we have bunch of different Widgets and want to draw them all in a loop?
- virtual function mechanism lets each object carry information about what functions to apply
- when a reference to a superclass type is really a reference to a subclass object
- and you use that reference to call a function
- this calls the subclass function
- "polymorphism": proper function to call is determined at run-time
  - e.g., drawing Widgets in an array:
    ```java
draw_all(Widget[] wa) { 
  for (int i = 0; i < wa.length; i++)
    wa[i].draw();
}
```
- virtual function mechanism automatically calls the right draw() function for each object
  - "virtual" means that a subclass may provide its own version of this function, which will be called automatically for instances of that subclass
  - superclass can provide a default implementation
- the loop does not change if more kinds of widgets are added

Exceptions are objects

- all derived from class Exception
- multiple catch blocks to catch multiple exceptions
  - caught in order of most specific first
- you can define your own exceptions

```java
public void savefile(String s, String f) throws EndOfTheWorld {
  try {
    FileOutputStream out = new FileOutputStream(f);
    out.write(s.getBytes());
    out.close();
  } catch (FileNotFoundException e) {
    System.err.println(e + " can't open " + f);
  } catch (IOException e) {
    System.err.println(e + " savefile error");
  } catch (Exception e) {
    System.err.println(e + " utterly unexpected error");
    throw new EndOfTheWorld("repent!");
  }
}

class EndOfTheWorld extends Exception {
  EndOfTheWorld(String s) {
    System.err.println(s + " the end of the world is at hand.");
  }
}
```
Inheritance principles

- classes are supposed to match the natural objects in the application
- derive specific types from a general type
  - collect common properties in the superclass
  - add special properties in the subclasses
- distinctions are not always clear
  - is a radiobutton a button or not?
  - should there be separate classes for horizontal and vertical scrollbars?
  - is a checkbutton a radiobutton or vice versa or neither?

Summary of inheritance

- a way to describe a family of types
- by collecting similarities (superclass)
- and separating differences (subclasses)
- polymorphism: proper member functions determined at run time

- not every class needs inheritance
  - may complicate without compensating benefit
- use composition instead of inheritance?
  - an object contains (has) an object rather than inheriting from it
- "is-a" versus "has-a"
  - inheritance describes "is-a" relationships
  - composition describes "has-a" relationships
Interfaces

- an interface is like a class
- declares a type
- only declares methods (not implementations)
  - and constants ("final")
  - methods are implicitly public
  - constants are implicitly public static final

- any class can implement the interface
  - i.e., provide implementations of the interface methods
  - and can provide other methods as well
  - and can implement several interfaces

- the only way to simulate function pointers and function objects

Comparison interface for sorting

```java
interface Cmp {
    int cmpf(Object x, Object y);
}

class Icmp implements Cmp { // Integer comparison
    public int cmpf(Object o1, Object o2) {
        int i1 = ((Integer) o1).intValue();
        int i2 = ((Integer) o2).intValue();
        if (i1 < i2)
            return -1;
        else if (i1 == i2)
            return 0;
        else
            return 1;
    }
}

class Scmp implements Cmp { // String comparison
    public int cmpf(Object o1, Object o2) {
        String s1 = (String) o1;
        String s2 = (String) o2;
        return s1.compareTo(s2);
    }
}
```

- whole lot of casting going on
- can't do an illegal cast, but don't find out till runtime
Sort function using an interface

```java
void sort(Object[] v, int left, int right, Cmp cmpf) {
    int i, last;
    if (left >= right) // nothing to do
        return;
    swap(v, left, rand(left, right));
    last = left;
    for (i = left+1; i <= right; i++)
        if (cmpf.cmpf(v[i], v[left]) < 0)
            swap(v, ++last, i);
    swap(v, left, last);
    sort(v, left, last-1, cmpf);
    sort(v, last+1, right, cmpf);
}
```

```java
Integer[] iarr = new Integer[n];
String[] sarr = new String[n];
Quicksort.sort(iarr, 0, n-1, new Icmp());
Quicksort.sort(sarr, 0, n-1, new Scmp());
```

Wrapper types

- most library routines work on Objects
  - don't work on basic types like int
- have to "wrap" basic types in objects to pass to library functions, store in Vectors, etc.
  - Character, Integer, Float, Double, etc.
- wrappers also include utility functions and values

```java
Integer I = new Integer(123); // constructor
int i = I.intValue(); // get value
I = Integer.parseInt("123"); // atoi
I = Integer.valueOf("123");
String s = I.toString();
```

```java
Double D = new Double(123.45);
double d = D.doubleValue();
d = Double.parseDouble("123.45"); // atof
D = Double.valueOf("123.45");
String s = D.toString();
```

```java
double atof(String str) {
    return Double.parseDouble(str);
}
```

```java
System.out.println(Double.MAX_VALUE);
```
Collections and collections framework

- "collection" == container in C++, etc.
  - Set, List (includes array), Map

- Interfaces for standard data types
  - abstract data types for collections
  - can do most operations independently of real type
  - include standard interface for
    - add, remove, size, member test, ...

- Implementations (concrete representations)
  - HashSet, TreeSet
  - ArrayList, LinkedList
  - HashMap, TreeMap

- Algorithms
  - standard algorithms like search and sort
  - work on any Collection of any type
  - that provides standard operations like comparison
  - "polymorphic"

- Iterators
  - uniform mechanism for accessing each element

Collection interface

- Set and List provide (more or less) this interface
  public interface Collection {
    int size();
    boolean isEmpty();
    boolean contains(Object elem);
    boolean add(Object elem);
    boolean remove(Object elem);
    Iterator iterator();
    ...
  }

- Map provides
  public interface Map {
    int size();
    boolean isEmpty();
    Object put(Object key, Object value);
    Object get(Object key);
    Object remove(Object key);
    boolean containsKey(Object key); // + Value
    ...
  }
Collections sort

- ArrayList is an implementation of List
  - like Vector but better
  - adds some of its own methods, like get()
- Collections.sort is a polymorphic algorithm
  - specific type has to implement Comparable

```java
class qsort1 {
    public static void main(String[] argv)
        throws IOException {
        FileReader f1 = new FileReader(argv[0]);
        BufferedReader f2 = new BufferedReader(f1);

        String s;
        List al = new ArrayList(100000);
        while ((s = f2.readLine()) != null)
            al.add(s);
        Collections.sort(al);
        for (int j = 0; j < al.size(); j++)
            System.out.println(al.get(j));
    }
}
```

Interface example: map

- interface defines methods for something
- says nothing about the implementation

```java
interface Map {
    void put(String name, String value);
    String get(String name);
    boolean member(String name);
    // ...
}
```

- classes implement it by defining functions
- have to implement all of the interface

```java
class Hashmap implements Map {
    Hashtable h;
    Hashmap() { h = new Hashtable(); }
    void put(String name, String value) {
        h.put(name, value);
    }
    String get(String name) {
        return h.get(name);
    }
    boolean member(String name) {
        return h.contains(name);
    }
}
class Treemap implements Map {
    RBTree t;
    Treemap() { t = new RBTree(); }
    void put(String name, String value) {
    }
    String get(String name) {
    }
```
Maps, wrappers, iterators

Map hs = new HashMap();

String buf;
while ((buf = f2.readLine()) != null) {
    String nv[] = buf.split("[
    ]+");
    Integer oldv = (Integer) hs.get(nv[1]);
    if (oldv == null)
        hs.put(nv[0], new Integer(nv[1]));
    else {
        int v = oldv.intValue() +
               Integer.parseInt(nv[1]);
        hs.put(nv[0], new Integer(v));
    }
}
for (Iterator i = hs.keySet().iterator();
     i.hasNext(); ) {
    String n = (String) i.next();
    Integer v = (Integer) hs.get(n);
    System.out.println(n + " " + v);
}

Word frequency counter

Map hs = new TreeMap(); // or HashMap
String buf;
while ((buf = f2.readLine()) != null) {
    String nv[] = buf.split("[
    ]+");
    for (int i = 0; i < nv.length; i++) {
        Integer oldv = (Integer) hs.get(nv[i]);
        if (oldv == null)
            hs.put(nv[i], new Integer(1));
        else
            hs.put(nv[i],
                    new Integer(oldv.intValue()+1));
    }
}
for (Iterator it = hs.keySet().iterator();
     it.hasNext(); ) {
    String n = (String) it.next();
    Integer v = (Integer) hs.get(n);
    System.out.println(v + " " + n);
}