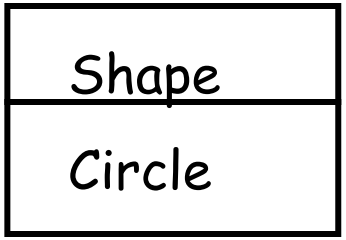
Inheritance in Java

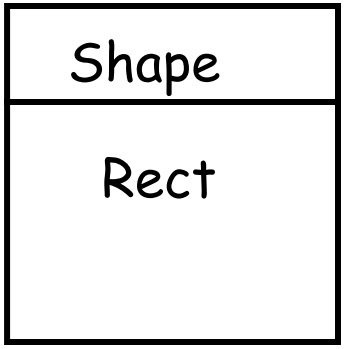
- **a way to create or describe one class in terms of another**
  - "a D is like a B, with these extra properties..."
  - "a D is a B, plus..."
  - B is the **base** class or **super**class
  - D is the **derived** class or **sub**class
  - Java uses **superclass** and **subclass**

- **inheritance is used for classes that model strongly related concepts**
  - objects share some common properties, behaviors, ...
  - and have some properties and behaviors that are different

- **superclass contains aspects common to all**
- **subclasses contain aspects different for different kinds**

Subclasses






```
class Shape{
    int color;
    // other vars common to all Shapes
}
class Circle extends Shape {
    int radius;
    // other vars specific to Circles
}
class Rect extends Shape {
    int ht, wid;
    // other vars specific to Rects
}
```

- **a Circle is a subclass (a kind of) Shape**
  - inherits all members of Shape
  - adds its own members
- **a Rect is also a subclass of Shape**

Object hierarchy

- **all objects are derived from class Object**

```
Object
  -> Math
  -> System
  -> Component -> Container -> JComponent ...
   -> InputStream -> FilterInputStream -> BufferedInputStream
```

- **Object has methods for equals, hashCode, toString, clone, etc.**
  - normally these are extended
- **assignment vs cloning:**

```
r1 = r2;          // refer to the same object
r1 = r2.clone();  // two separate objects
```

- **default X.equals method is Object.equals**
  - tests for same reference, i.e., same object
- **for other definitions of equality, overload equals**

```
class X {
   String str;
   public boolean equals(X r2) {
      return str.equals(r2.str);
}
```

Virtual functions

- **in Java, all functions are implicitly *virtual***
- **if a reference to a superclass type is really a reference to a subclass object, a function call with that reference calls the subclass function**
- **polymorphism: proper function to call is determined at run-time**
  - e.g., drawing Shapes in an array:

```
draw(Shape[] sa) {
  for (int i = 0; i < sa.length; i++)
    sa[i].draw();
}
```

- **virtual function mechanism automatically calls the right draw() function for each object**
  - a subclass may provide its own version of this function, which will be called automatically for instances of that subclass
  - the superclass can provide a default implementation
- **the loop does not change if more subclasses of Shapes are added**

Exceptions are objects

- all derived from class Exception
- multiple catch blocks to catch multiple exceptions (most specific first)
- you can define your own exceptions

```
public class except2 {
   public static void main(String[] args) throws EndOfTheWorld {
      try {
         FileInputStream fin = new FileInputStream(args[0]);
         // etc.
      } catch (FileNotFoundException e) {
         System.err.println("FileNotFoundException " + e);
      } catch (IOException e) {
         System.err.println("IOException " + e);
      } catch (Exception e) {
         e.printStackTrace();
         throw new EndOfTheWorld("repent!");
      }
   }
}
class EndOfTheWorld extends Exception {
   EndOfTheWorld(String s) {
      System.err.println(s + " the end of the world is at hand.");
   }
}
```

Interfaces in Java

- **an interface is like a class**
- **declares a type**
- **only declares methods (not implementations) and constants**
  - 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

```
class foo implements bar {
   // implementation of bar methods
}
```

- **the only way to simulate function pointers and function objects**

Comparison interface for sorting

```
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

```
void sort(Object[] v, int left, int right, Cmp cf) {
   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 (cf.cmpf(v[i], v[left]) < 0)
         swap(v, ++last, i);
   swap(v, left, last);
   sort(v, left, last-1, cf);
   sort(v, last+1, right, cf);
}

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 only 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**

```
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();

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();

double atof(String str) { return Double.parseDouble(str); }
System.out.println(Double.MAX_VALUE);
```

Boxing and unboxing

- **Java 1.5 autobox and unbox somewhat clean up this mess**

```
Integer I = 123; // no need for new Integer()
int i = I;       // no need for I.intValue()
String s = I.toString();

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

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

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

```
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();
    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

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

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

```
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); }

class Treemap implements Map {
    RBTree t;
    Treemap() { t = new RBTree(); }
    void put(String name, String value) { … }
    String get(String name) { … }
```

Word frequency counter

- **count number of occurrences of each word**

```
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);
}
```

Boxing, unboxing

- **boxing cleans up bulky wrapper code**

```
Map<String, Integer> hs = new HashMap<String, Integer>();

String buf;
while ((buf = f2.readLine()) != null) {
   String nv[] = buf.split("[   ]+");
   for (int i = 0; i < nv.length; i++) {
      Integer oldv = hs.get(nv[i]);
      if (oldv == null)
         hs.put(nv[i], 1);
      else
         hs.put(nv[i], oldv+1);
   }
}
for (String n : hs.keySet()) {
   Integer v = hs.get(n);
   System.out.println(v + " " + n);
}
```

Generics, for-each

- **generics tell compiler what type a Collection holds**
  - compiler can do more type checking at compile time
- **for-each loop cleans up iterator code**

```
String s;
List<String> al = new ArrayList<String>();
while ((s = f2.readLine()) != null)
   al.add(s);
Collections.sort(al);
for (String j : al)
   System.out.println(j);
```

- **<?> as a type in a generic matches any type**

- **<? extends T> matches any type that extends T**
  - "bounded wildcard"

Sorting: Java v. C++

```
String s;
List<string> al = new ArrayList<string>();
while ((s = f2.readLine()) != null)
    al.add(s);
Collections.sort(al);
for (String j : al)
    System.out.println(j);
```

```
string tmp;
vector<string> v;
while (getline(cin, tmp))
    v.push_back(tmp);
sort(v.begin(), v.end());
copy(v.begin(), v.end(),
        ostream_iterator<string>(cout,"\n"));
```

Add up a bunch of numbers: Java v. C++

```
while ((buf = f2.readLine()) != null) {
  String nv[] = buf.split("[     ]+");
  for (int i = 0; i < nv.length; i++) {
    try {
      dsum += Double.parseDouble(nv[i]);
    } catch (NumberFormatException e) {
      ;
    }
  }
}
```

```
while (getline(cin, tmp)) {
    istringstream iss(tmp);
    string s;
    while (iss >> s) {
        dsum += atof(s.c_str());
    }
}
```

Word frequency count: Java

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

    Map<String, Integer> hs = new HashMap<String,Integer>(10000);
    String buf;
    while ((buf = f2.readLine()) != null) {
      String nv[] = buf.split("[   ]+");
      for (int i = 0; i < nv.length; i++) {
        Integer oldv = hs.get(nv[i]);
        if (oldv == null)
          hs.put(nv[i], 1);
        else
          hs.put(nv[i], oldv+1);
      }
    }
    for (String n : hs.keySet()) {
      Integer v = hs.get(n);
      System.out.println(v + " " + n);
    }
  }
}
```

Word frequency count: C++ STL

```
#include <iostream>
#include <map>
#include <string>

int main() {
    string temp;
    map<string, int> v;
    map<string, int>::const_iterator i;

    while (cin >> temp)
        v[temp]++;
    for (i = v.begin(); i != v.end(); ++i)
        cout << i->first << " " << i->second << "\n";
}
```