Classes, objects and all that

- data abstraction and protection mechanism
- originally from Simula 67, via C++ and others

```java
class thing {
  public part:
    methods: functions that define what operations can be done on this kind of object
  private part:
    functions and variables that implement the operation
}
```

- defines a new data type "thing"
  - can declare variables and arrays of this type, pass to functions, return them, etc.
- object: an instance of a class variable
- method: a function defined within the class
  - (and visible outside)
- private variables and functions are not accessible from outside the class
- not possible to determine HOW the operations are implemented, only WHAT they do

Classes & objects  (adapted from Flanagan, Java in a Nutshell)

- in Java, everything is part of some object
  - all classes are derived from class Object

```java
public class Circle {
  double x, y;     // center
  double r;        // radius
  public double circum() { return 2 * 3.14159 * r; }
  public double area() { return 3.14159 * r * r; }
}
```

- member functions are defined within the class
- all objects are created dynamically
- have to call new to construct an object

```java
Circle c; // null reference:
  // doesn't yet refer to an object
  c = new Circle(); // now it does (initialized to 0)
c.r = 3;
System.out.println("area = " + c.area());
```
Constructors: making a new object

```java
public Circle(double x, double y, double r) {
    this.x = x;
    this.y = y;
    this.r = r;
}
```

```java
Circle c;
c = new Circle(1, 2.2, 3.4);
// or Circle c = new Circle(1, 2.2, 3.4);
```

- "this" is the object being constructed or running the code

- can use multiple constructors with different arguments to construct in different ways:
  ```java
  public Circle(double r) { x = y = 0.0; this.r = r; }
  public Circle(Circle c) { x = c.x; y = c.y; r = c.r; }
  ```

- one constructor can invoke another
  ```java
  public Circle(double r) { this(0, 0, r); }
  public Circle(Circle c) { this(c.x, c.y, c.r); }
  public Circle() { this(0, 0, 1); } // unit circle at (0,0)
  ```

Class variables & instance variables

- every object is an instance of some class
  - created dynamically by calling `new`

- class variable: a variable declared `static` in class
  - only one instance of it in the entire program
  - exists even if the class is never instantiated
  - the closest thing to a global variable in Java

```java
public class Circle {
    static int num_circles = 0;
    static final double PI = 3.14159265358979323846;
    double x, y;     // center
    double r;        // radius

    public double circum() { return 2 * PI * r; }
    public double area() { return PI * r * r; }
    public Circle(double x, double y, double r) {
        num_circles++;
        this.x = x; this.y = y; this.r = r;
    }
    public Circle(double r) { this(0, 0, r); }
    public Circle(Circle c) { this(c.x, c.y, c.r); }
    public Circle() { this(0, 0, 1); } // unit circle
}
```
Class methods

- most methods associated with an object instance
- if declared static, amounts to a global function

```java
class Circle {
    public static boolean equals(Circle c1, Circle c2) {
        return c1.r == c2.r;
    }
    public boolean equals(Circle c) {
        return this.r == c.r;
    }
    public static void main(String[] args) {
        Circle c1 = new Circle(1.23);
        Circle c2 = new Circle(12.3);
        if (equals(c1, c2)) ... // compares contents
        if (c1.equals(c2)) ... // compares contents
        if (c1 == c2) ...       // object equality
    }
}
```

- some classes are entirely static members and class functions, e.g., Math, System, Color

Destruction & garbage collection

- interpreter keeps track of what objects are currently in use
- memory can be released when last use is gone
  - release does not usually happen right away
  - has to be garbage-collected
- garbage collection happens automatically
  - separate low-priority thread manages garbage collection
- no control over when this happens
  - can set object reference to null to encourage it

- Java has no destructor (unlike C++)
  - can define a finalize() method for a class to reclaim other resources, close files, etc.
  - no guarantee that a finalizer will ever be called

- garbage collection is a great idea
  - but this is not a great design
"Real" example: regular expressions

- simple class to look like RE in assignment 1
  - instead of opaque type in C
- uses the Java 1.4 regex mechanism
- provides a better interface (or at least less clumsy)

```java
import java.util.regex.*;

public class re {
    Pattern p;
    Matcher m;

    public re(String pat) {
        p = Pattern.compile(pat);
    }

    public void compile(String s) {
        p = Pattern.compile(s);
    }

    public boolean match(String s) {
        m = p.matcher(s);
        return m.find();
    }

    public int start() {
        return m.start();
    }

    public int end() {
        return m.end();
    }
}
```

Using the RE class

- excerpt from a sort of grep

```java
class something {
    public static void main(String[] args) {
        ...;
        re r = new re(args[0]);
        try {
            String s;
            while ((s = in.readLine()) != null) {
                if (r.match(s))
                    System.out.println(s);
            }
        } catch (Exception e) {
            System.err.println("IOException " + e);
        }
    }
}
```
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();
...

```c
Widget *but = new Widget(BUTTON);
b->setbgcolor(0xFF00FF);
b->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
}
```

class Button extends Widget {
    int state;
    // other vars specific to Buttons
}

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`

```
widget

button

scrollbar
```
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

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(bool) {...}
}

    CheckButton b;
    Scrollbar s;

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

Inheritance and subclasses

- example: a Ring class derived from Circle

class Ring extends Circle {
    double r0;       // inner radius

    public double area() {
        return super.area() - Circle.PI * r0*r0;
    }

    public Ring(double x, double y,
                double r0, double r) {
        super(x, y, r); // has to come first
        this.r0 = r0;
    }

    // access superclass methods with super()

    // access class variables, constants and methods
    with class name

    Color.red
    Math.cos(Math.PI)
    System.out.println("...")
Object hierarchy

- all objects are derived from class Object
  - e.g., a Circle is an Object
  - a Ring is a Circle is an Object

Object -> Circle -> Ring
  -> 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 Circle.equals is Object.equals
- tests for same reference, i.e., same object
- to compare for equal radius, overload equals

```java
class Circle {  // one defn of equality
    public boolean equals(Circle c) {
        return c.r == this.r;
    }
}
```

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?
Overriding, dynamic method lookup

- `Ring.area()` overrides `Circle.area()`
  - arguments are identical

- all functions are implicitly virtual:
  - a reference to the superclass calls the subclass method for a subclass object

```java
Ring r = new Ring(0, 0, 0.5, 1);
System.out.println("ring area = " + r.area());

Circle c = r;
System.out.println("ring area = " + c.area());
```
// calls `r.area()`

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.");
    }
}
```

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 containing (has) an object rather than inheriting from it

- "is-a" versus "has-a"
  - inheritance describes "is-a" relationships
  - composition describes "has-a" relationships
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
double atof(String str) {
    return Double.parseDouble(str);
}

System.out.println(Double.MAX_VALUE);

Integer I = new Integer(123); // constructor
int i = I.intValue(); // get value

String s = I.toString();
char ch = 'a'; // 16-bit Unicode
if (Character.isLetterOrDigit(ch)) ...
```

Interfaces

- an interface is like a class
  - declares a type
  - only declares methods (not implementations)
    - and constants (“final”)
  - in effect, it’s like an abstract class
    - can’t exist on its own
- any class can implement the interface
  - i.e., provide implementations of the interface methods
    - and can provide other methods as well
- the only way to simulate function pointers and function objects
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) {
        t.put(name, value);
    }
    String get(String name) {
        return t.get(name);
    }
}
```

Wrappers again

- From Campione & Walrath Java Tutorial, p 489:

```java
public class Freq {
    private static final Integer ONE = new Integer(1);

    public static void main(String args[]) {
        Map m = new Treemap();
        // Initialize frequency table from command line
        for (int i=0; i<args.length; i++) {
            Integer freq = (Integer) m.get(args[i]);
            m.put(args[i], (freq==null ? ONE :
                             new Integer(freq.intValue() + 1)));
        }
        System.out.println(m.size() + " distinct words detected:");
        System.out.println(m);
    }
}
```
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.cmp(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());
```
Visibility

- **private, public, protected**

  ```java
  public class foo {    // people can use this class
    private v;          // can’t see this variable
    public void f();    // can use this public method
  }
  ```

- **public class, method or variable**
  - visible everywhere

- **private method or variable**
  - only by methods of the class

- **protected method or variable**
  - only by methods of the class, subclasses, and other classes in the same package

- **default visibility (“package” visibility)**
  - only visible in class that defines it and other classes in the same package
  (but not subclasses in other packages)

- **package**
  - a group of related and possibly cooperating classes
  - all non-private variables and members are visible to all other classes in the package
  - loosely, like mutual friends in the C++ sense