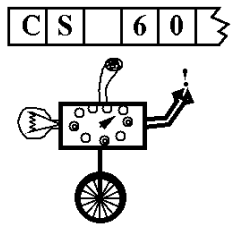
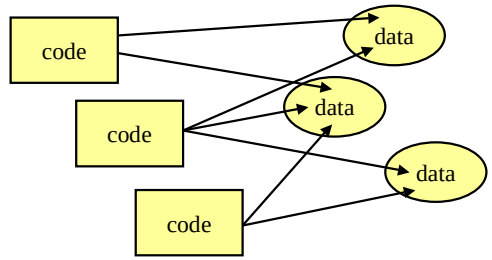


Classes and Objects

October 10, 2001

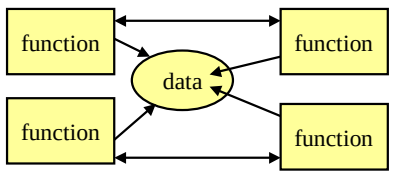
Program Organization (or lack thereof)

- Early programming languages had little support for organizing code and data structures.



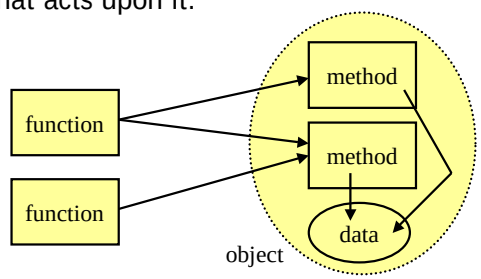
Modularity

- Such programs are unreadable and unmaintainable
  - Change a data structure and *everything* breaks
- Idea: use abstraction and information hiding
  - Access data structure through specific procedures



Object-Oriented Programming

- Objects are a programming language mechanism for combining data with the code that acts upon it.



Object = Code + Data

- Object is made up of
  - data, represented as instance variables (fields)
  - code, the methods of the object
- Normally, the instance variables in an object can be modified (by the methods)
  - But not always; consider `Polylist`.

Class

- Conventional object-oriented languages (including C++ and Java) are arranged around the concept of the class.
  - Specifies the contents of the objects of the class
    - What fields are there
    - What methods are there.
  - Provides a way of creating objects of that class
    - And specifies how fields are initialized (constructors)
    - All objects created by the same class have the same methods
  - Often provides a location for related constants, variables, or code (`static`).
- Note, by the way, that it's possible to have objects as simple collections of code and data, without classes.
  - e.g., JavaScript

Interfaces

- Principal abstraction mechanism in Java is the formal concept of interface.
  - Like a class, except that it only specifies certain required methods
  - The interface cannot specify any implementations.
  - Thus, cannot use "new" on an interface.
- A given class implements the interface by giving definitions for (at least) the methods in the interface.
  - The class must also assert that it implements the interface.
  - The compiler checks that all the methods claimed are actually there.

A Queue Interface

```
public interface QueueI {
  boolean isEmpty();
  void    enqueue(Object o);
  Object  dequeue();
}
```

Improved `Cell` Implementation

```
class Cell {
private Object data;
private Cell next;

public Cell(Object data, Cell next) {
  this.data = data;
  this.next = next;
}

public Object getData()  { return data; }
public Cell   getNext()  { return next; }

public void setData(Object data) { this.data = data; }
public void setNext(Cell next)   { this.next = next; }
}
```

Queue Implementation

```
class Queue implements QueueI {
Cell head;
Cell tail;

public Queue() {head = null; tail = null; }

public void enqueue(Object data) {
  Cell newCell = new Cell(data, null);
  if (head==null) { head = newCell; }
  else {tail.setNext(newCell);}
  tail = newCell;
  }

public boolean isEmpty() { return head == null; }
```

Queue Implementation, cont.

```
// call only if !isEmpty()
public Object dequeue() {
  Object result = head.getData();
  head = head.getNext();
  if( head == null ) { tail = null; }
  return result;
  }
}
```

Queue Test Code

```
public static void testQueue(QueueI q)
  {
  int numItems = 10;

  for( int i = 0; i < numItems; i++ )
    {
    q.enqueue(new Integer(i));
    }

  for( int i = 0; i < numItems; i++ )
    {
    System.out.print(q.dequeue() + " ");
    }
  }
}
```

Doesn't care how Q is implemented!

Interface Observations

- Pitfall:
  - Just because a class implements an interface doesn't guarantee it behaves in the expected way!

```
interface Drawable {
   void Draw();
}
```

- Interfaces provide *dynamic dispatch*
  - When we call `q.dequeue()` we don't know exactly what piece of code will run
    - Depends on the class of the object being passed
    - Different calls may provide objects of different classes
    - Don't know this until run-time
  - Compare with higher-order functions in rex

Creating New Classes From Old

- At least two ways to do this in Java
  - Aggregation/Composition (has-a)
    - A `TreeDraw` object has a `Image` buffer
      - It is not a sort of `Image`.
    - A `Queue` uses a `Polylist` of `Cells`
      - A Queue is *not* a type of `Polylist` or `Cell`
  - Inheritance (is-a)
    - A `TreeDraw` object *is* a specific sort of `Applet`
    - A `Queue` is a specific sort of `Object`.

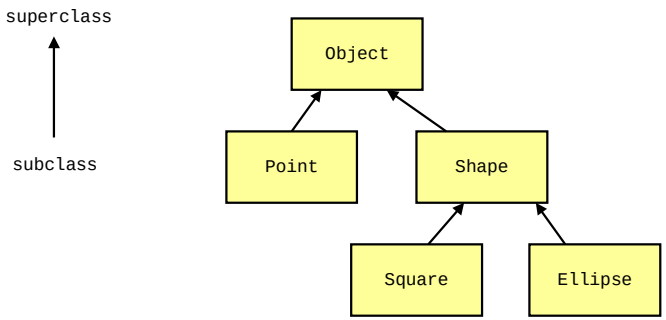
Aggregation (has-a)

```
class Point {
   private int x, y;
   Point(int x, int y)     { this.x = x; this.y = y }
   void move(int x, int y) { this.x = x; this.y = y }
   int getx() { return x; }
   int gety() { return y; }
}

class Shape {
   Point center; // every Shape references a Point
   Shape(int x, int y) { center = new Point(x,y); }
   void move(int x, int y) { center.move(x,y); }
   void draw() {};
   void moveAndDraw(int x, int y) {move(x,y); draw(); }
}
```

Inheritance (is-a)

```
class Square extends Shape {
   Point center;
   int size;
   Square(int x, int y, int size)
        { super(x,y); this.size = size }
   void draw() { ... }
}

class Ellipse extends Shape {
   Point center;
   int width, height;
   Rect(int x, int y, int w, int h) {
        super(x,y); width = w; height = h;
   }
   void draw() { ... }
}
```

Class Inheritance Hierarchy



What Code Runs At Each Call?

```
Shape shape;
Square  square = new Square(10,10,5);
Ellipse ellipse = new Ellipse(20,20,4,7);

ellipse.move(14,12);
ellipse.moveAndDraw(12,14);

shape = ellipse;        // Why no cast needed?
shape.draw();
shape.moveAndDraw(2,2);

shape = square;
shape.moveAndDraw(42,42);
```

Dynamic dispatch strikes again!

How Can Subclasses Differ?

- A subclass can take its superclass and do
  - *Extension*
    - Add new instance variables (`width`)
    - Add new methods, as long as they different names (or types) from inherited methods
    - Add constructors
  - *Overriding*
    - Replace inherited methods with new code with the same name and type (`draw`)
    - Overridden methods can still be invoked, by saying `super.`*`method`* instead of *`method`* or `this.`*`method`*

(from Joshua Bloch, *Effective Java)*

Can You Figure Out The Answer?

```
class InstrumentedHashSet extends HashSet {
   private int addCount = 0;  // count of elements added
   public InstrumentedHashSet() {}
   public InstrumentedHashSet(Collection c) { super(c); }
   public boolean add(Object o) {
        addcount++;
        return super.add(o);
   }
   public boolean addAll(Collection c) {
        addCount += c.size();
        return super.addAll(c);
   }
   public int getAddCount() { return addCount; }
}
...
InstrumentedHashSet ihs = new InstrumentedHashSet();
ihs.addAll(Arrays.asList(new String[]{"A","B","C"}))
```

Pitfalls of Overriding

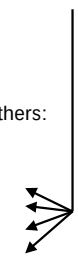
- Overriding breaks encapsulation
  - To do overriding right, need to know details about the *implementation* of the superclass
    - Not just its interface!
    - Full knowledge of which methods call which other methods, and when.
    - What if the implementation of the superclass changes?
  - Some people suggest using only extension unless absolutely necessary.
- Still, there are a few cases where it's safe and useful.
  - Where superclass is designed for overriding

(See example in text, section 7.12)

Safe Overriding

- The Applet class includes the following methods, among others:

```
public void    init()
public void    start()
public boolean mouseDown(Event e, int x, int y)
public boolean mouseDrag(Event e, int x, int y)
public boolean mouseUp  (Event e, int x, int y)
public boolean mouseMove(Event e, int x, int y)
public void    update(Graphics g)
public void    paint(Graphics g)
```



- The Applet design
  - Promises exactly when these methods will be called
  - Explains the default implementation, so you know whether you want to override it.
  - Applets that want to know about mouse clicks can override `mouseDown`; applets that just draw trees can leave the default implementation unchanged (do nothing)

Abstract Classes

- An *abstract* class is a class in which certain methods are left unimplemented.
  - Cannot create objects of this class; they'd be incomplete.
    - Can still have variables of this class, though!
  - To be useful, need to create subclasses that implement the missing methods.

```
abstract class Shape {
   Point center; // every Shape references a Point
   Shape(int x, int y) { center = new Point(x,y); }
   void move(int x, int y) { center.move(x,y); }
   abstract void draw();
   void moveAndDraw(int x, int y) {move(x,y); draw(); }
}
```

Abstract Classes vs. Interfaces

- Somewhat similar concepts
  - C++ has only the former (which get used as interfaces).
  - Both specify a collection of methods
- Differences?