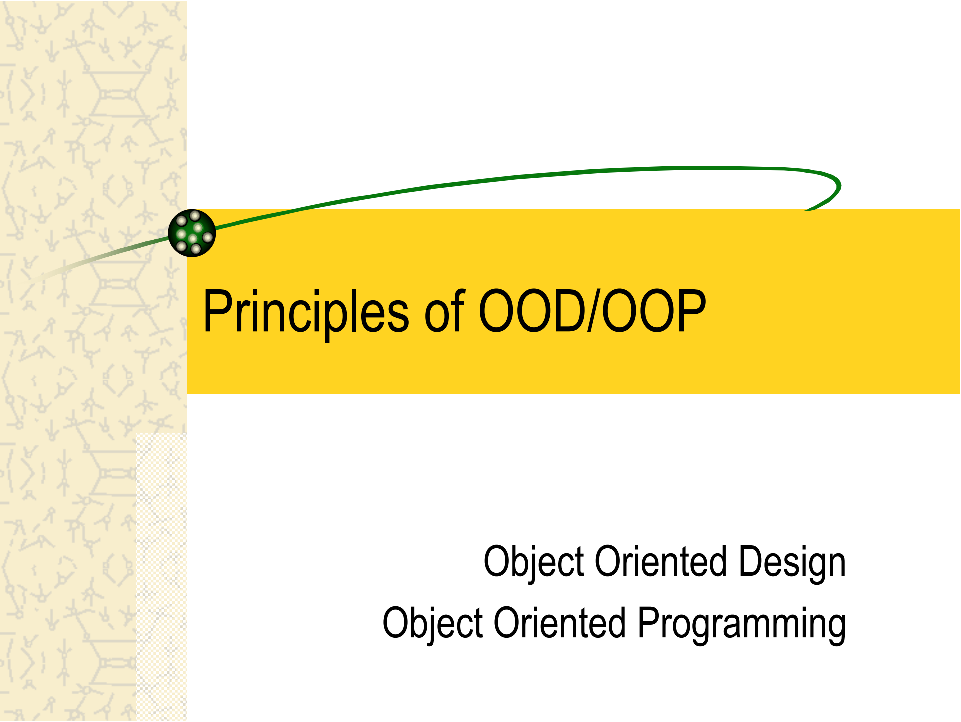
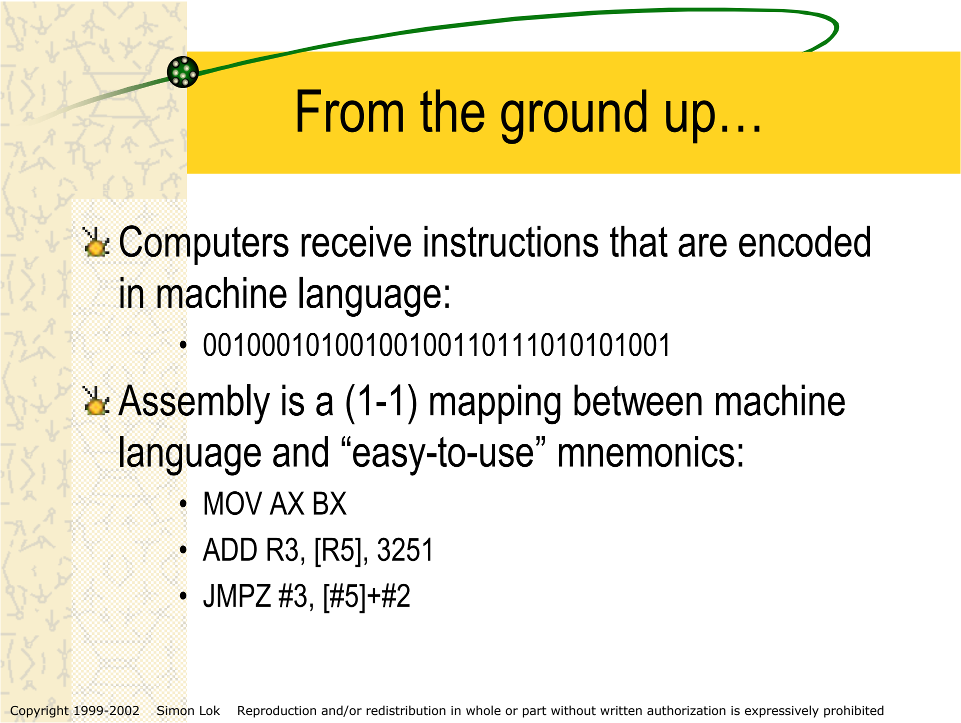


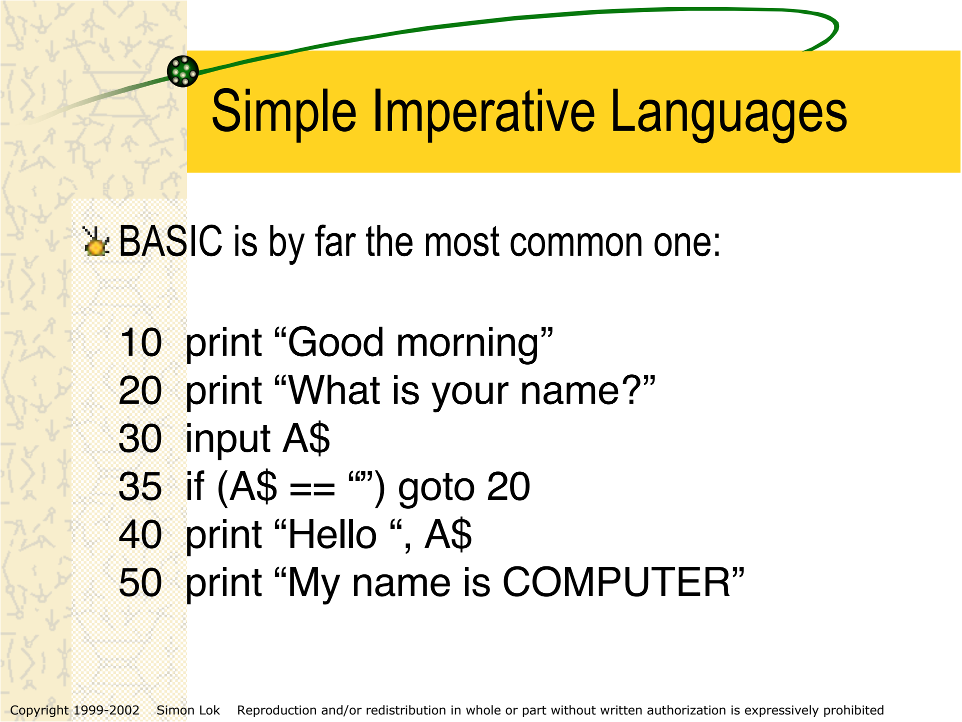
Principles of OOD/OOP

Object Oriented Design
Object Oriented Programming



From the ground up…

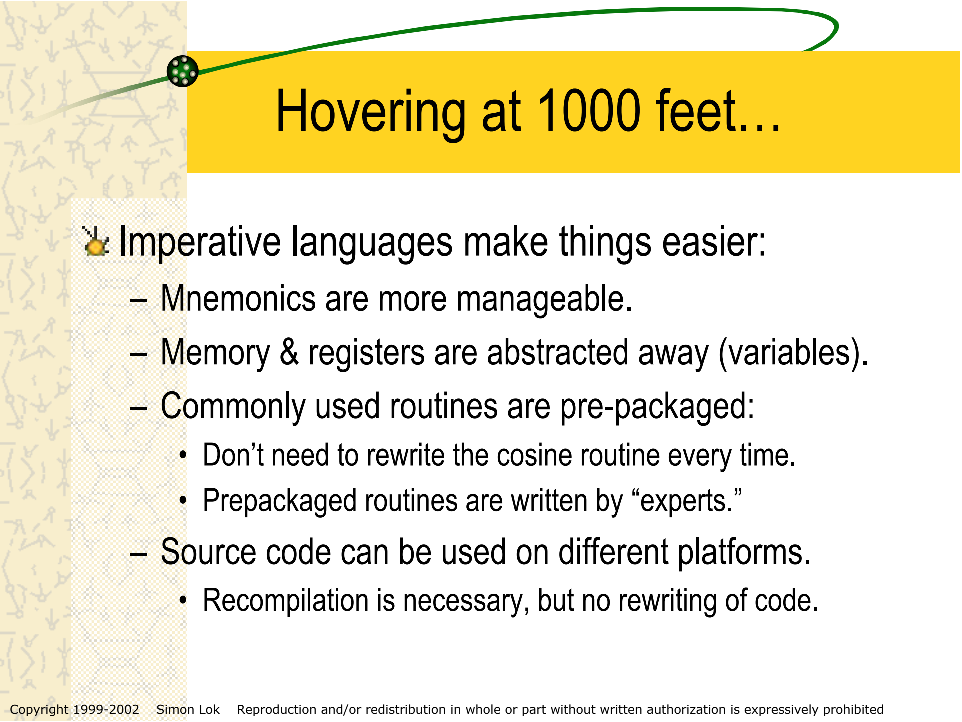
- Computers receive instructions that are encoded in machine language:
  - 00100010100100100110111010101001
- Assembly is a (1-1) mapping between machine language and “easy-to-use” mnemonics:
  - MOV AX BX
  - ADD R3, [R5], 3251
  - JMPZ #3, [#5]+#2



Simple Imperative Languages

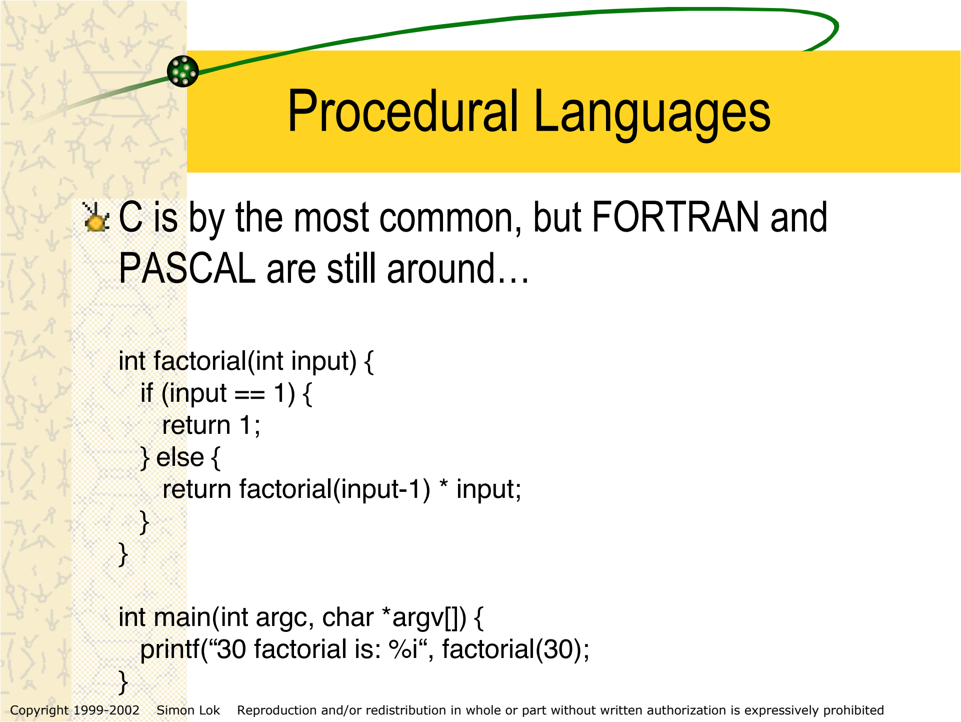
- BASIC is by far the most common one:

```
10  print “Good morning”
20  print “What is your name?”
30  input A$
35  if (A$ == “”) goto 20
40  print “Hello “, A$
50  print “My name is COMPUTER”
```



Hovering at 1000 feet…

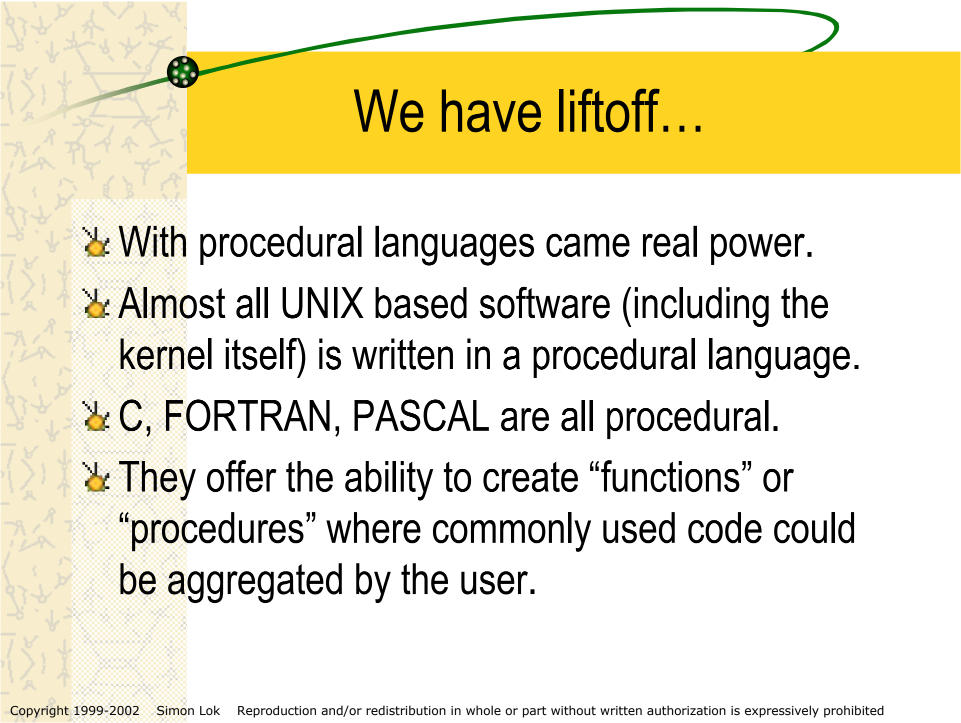
- Imperative languages make things easier:
  - Mnemonics are more manageable.
  - Memory & registers are abstracted away (variables).
  - Commonly used routines are pre-packaged:
    - Don’t need to rewrite the cosine routine every time.
    - Prepackaged routines are written by “experts.”
  - Source code can be used on different platforms.
    - Recompilation is necessary, but no rewriting of code.



Procedural Languages

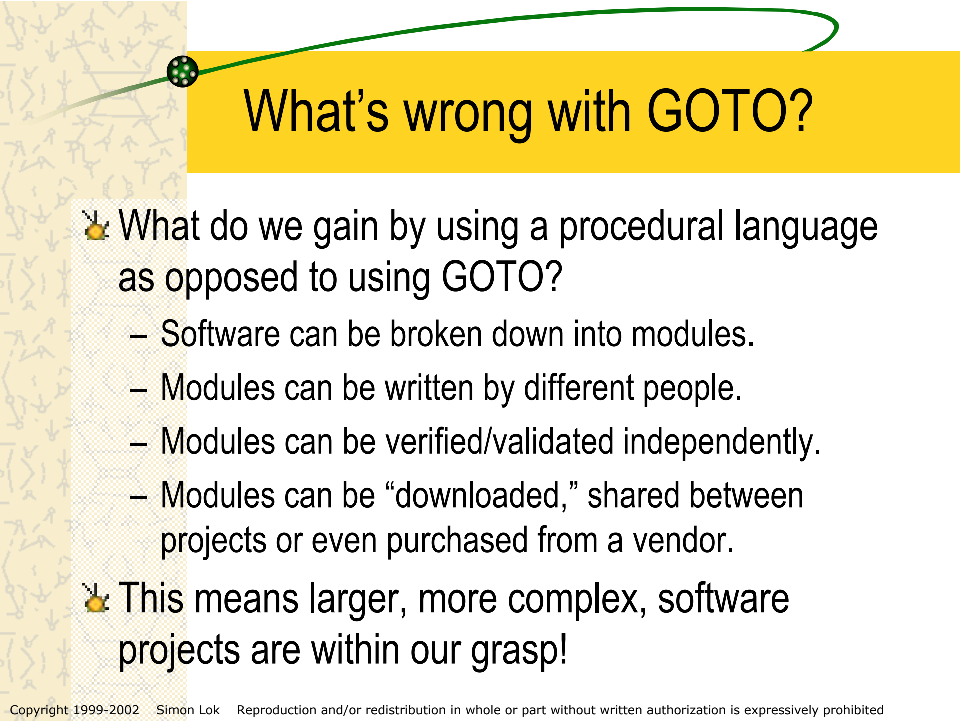
- C is by the most common, but FORTRAN and PASCAL are still around…

```
int factorial(int input) {
   if (input == 1) {
      return 1;
   } else {
      return factorial(input-1) * input;
   }
}

int main(int argc, char *argv[]) {
   printf(“30 factorial is: %i“, factorial(30);
}
```



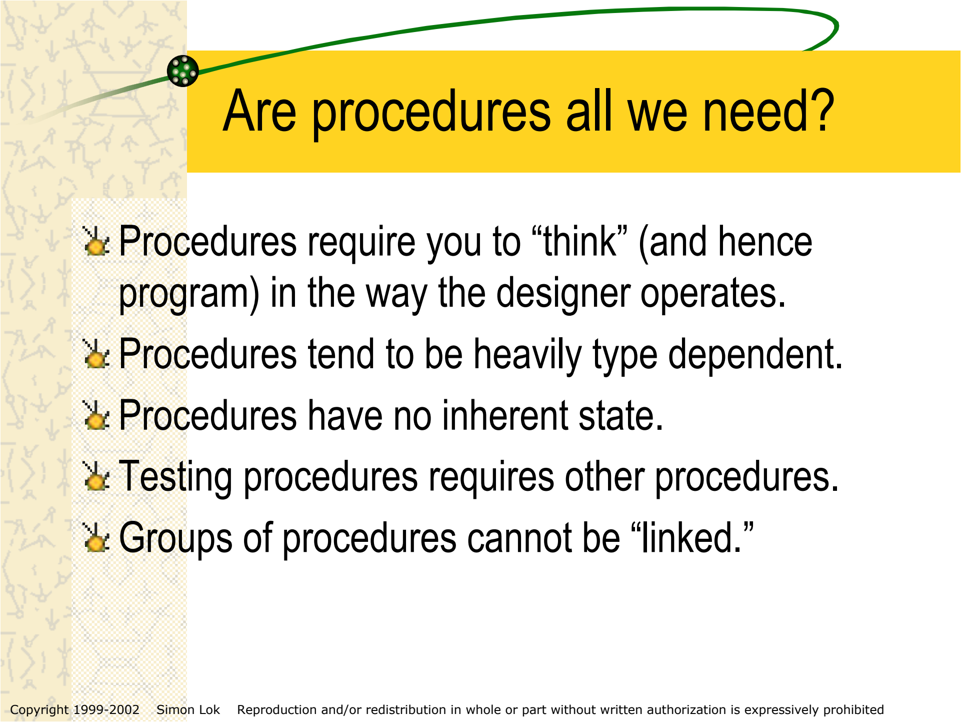
We have liftoff…

- With procedural languages came real power.
- Almost all UNIX based software (including the kernel itself) is written in a procedural language.
- C, FORTRAN, PASCAL are all procedural.
- They offer the ability to create “functions” or “procedures” where commonly used code could be aggregated by the user.



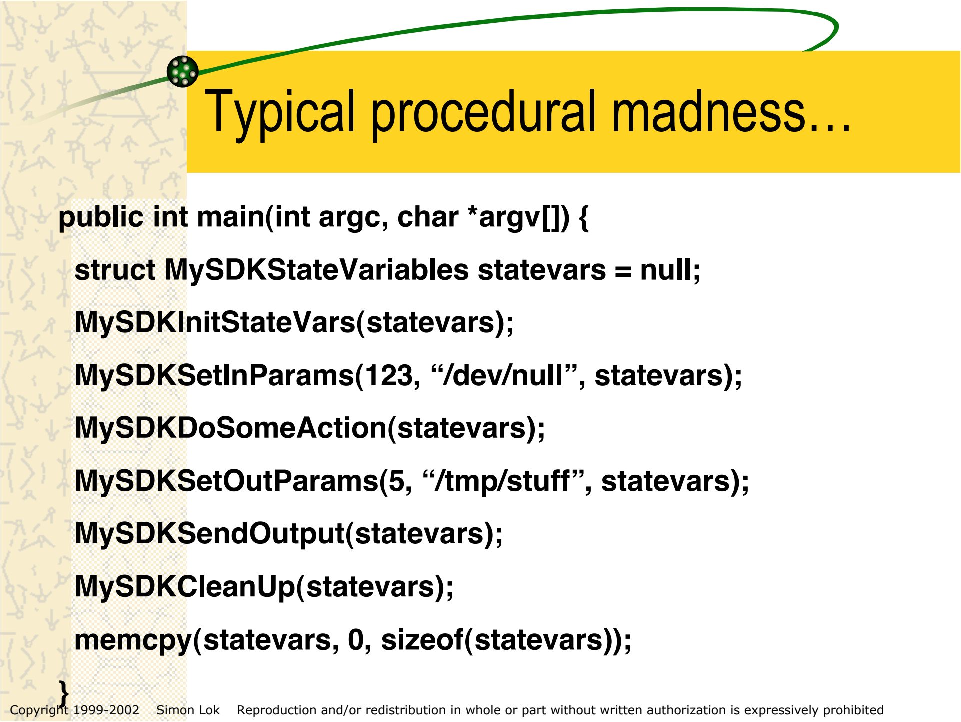
What's wrong with GOTO?

- What do we gain by using a procedural language as opposed to using GOTO?
  - Software can be broken down into modules.
  - Modules can be written by different people.
  - Modules can be verified/validated independently.
  - Modules can be "downloaded," shared between projects or even purchased from a vendor.
- This means larger, more complex, software projects are within our grasp!



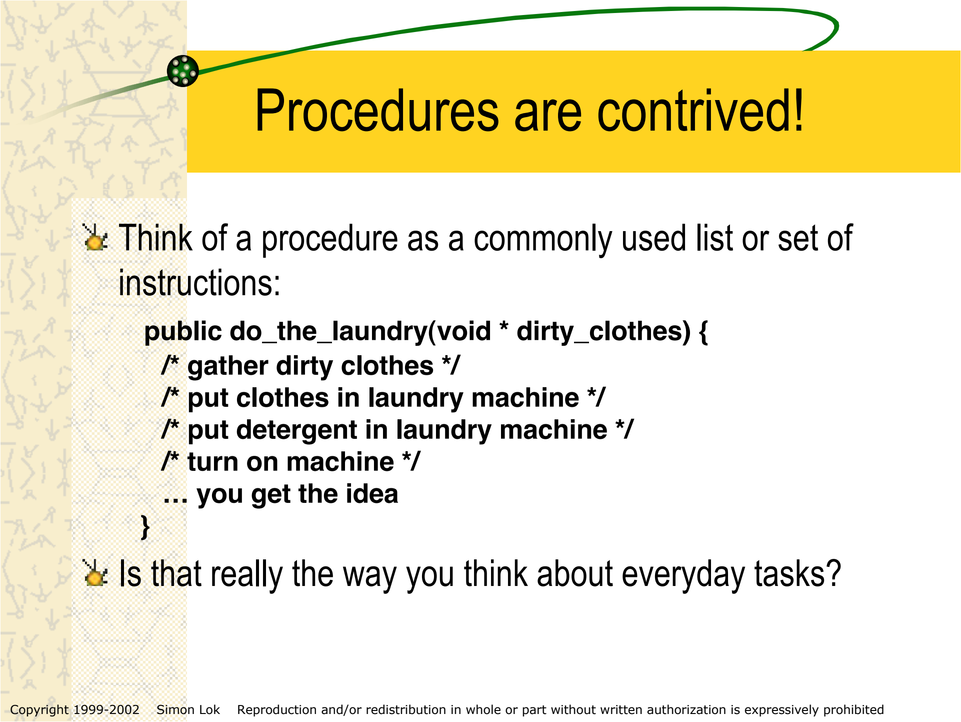
Are procedures all we need?

- Procedures require you to "think" (and hence program) in the way the designer operates.
- Procedures tend to be heavily type dependent.
- Procedures have no inherent state.
- Testing procedures requires other procedures.
- Groups of procedures cannot be "linked."



Typical procedural madness…

```
public int main(int argc, char *argv[]) {
  struct MySDKStateVariables statevars = null;
  MySDKInitStateVars(statevars);
  MySDKSetInParams(123, “/dev/null”, statevars);
  MySDKDoSomeAction(statevars);
  MySDKSetOutParams(5, “/tmp/stuff”, statevars);
  MySDKSendOutput(statevars);
  MySDKCleanUp(statevars);
  memcpy(statevars, 0, sizeof(statevars));
}
```

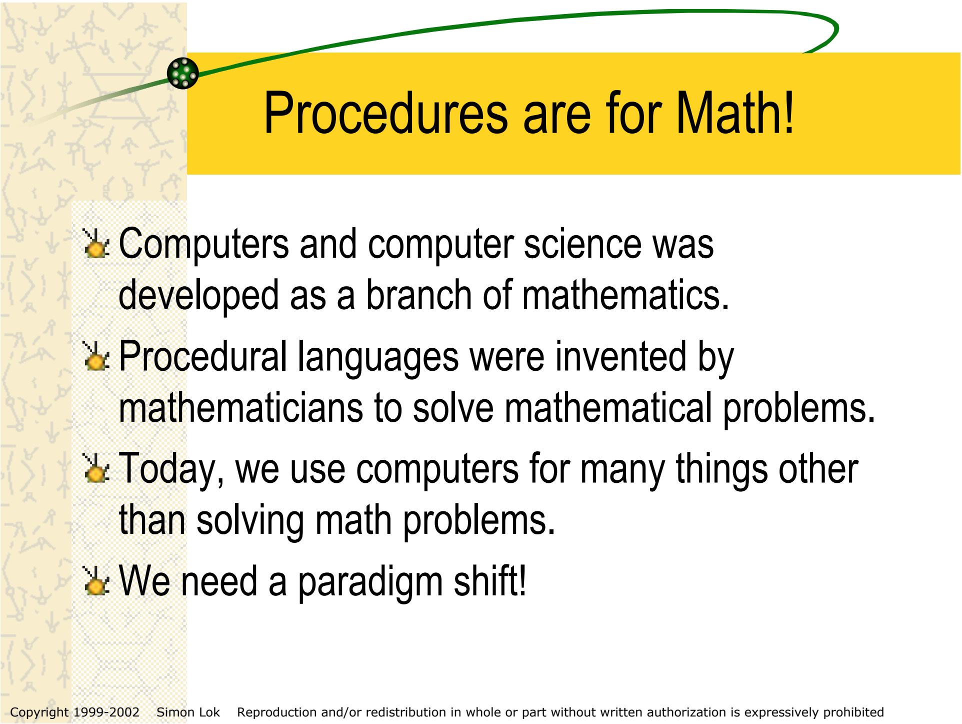


Procedures are contrived!

- Think of a procedure as a commonly used list or set of instructions:

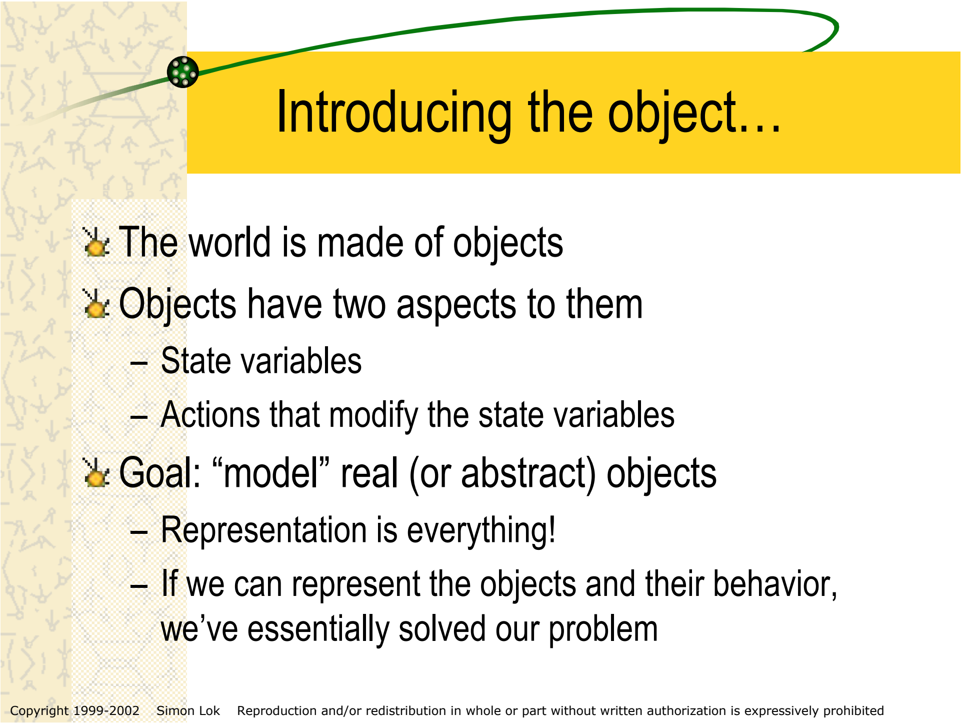
```
public do_the_laundry(void * dirty_clothes) {
  /* gather dirty clothes */
  /* put clothes in laundry machine */
  /* put detergent in laundry machine */
  /* turn on machine */
  … you get the idea
}
```

- Is that really the way you think about everyday tasks?



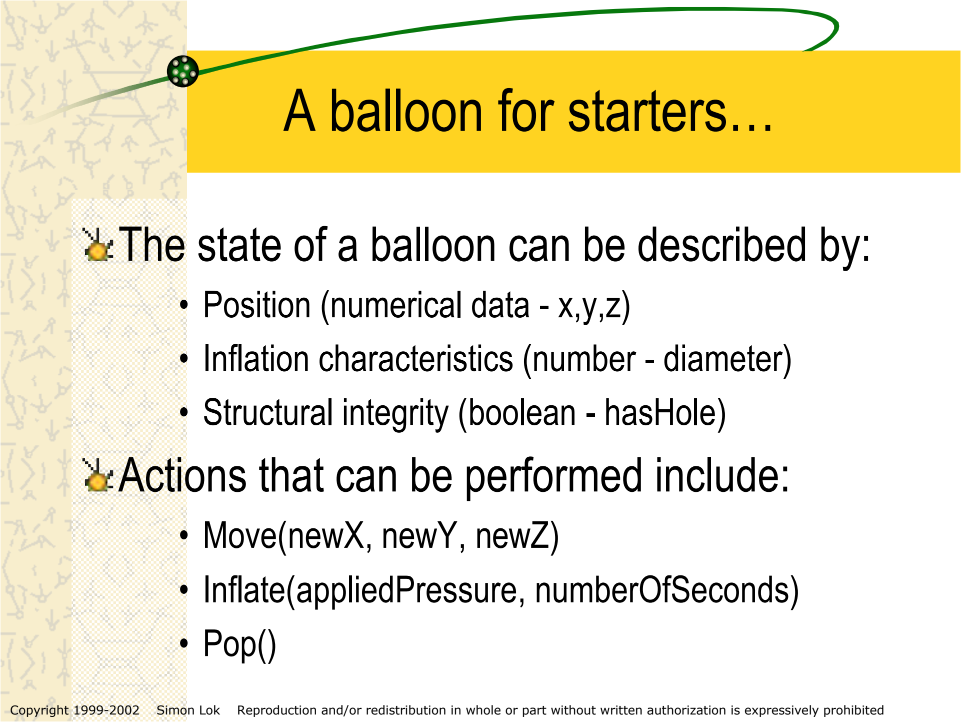
Procedures are for Math!

- Computers and computer science was developed as a branch of mathematics.
- Procedural languages were invented by mathematicians to solve mathematical problems.
- Today, we use computers for many things other than solving math problems.
- We need a paradigm shift!



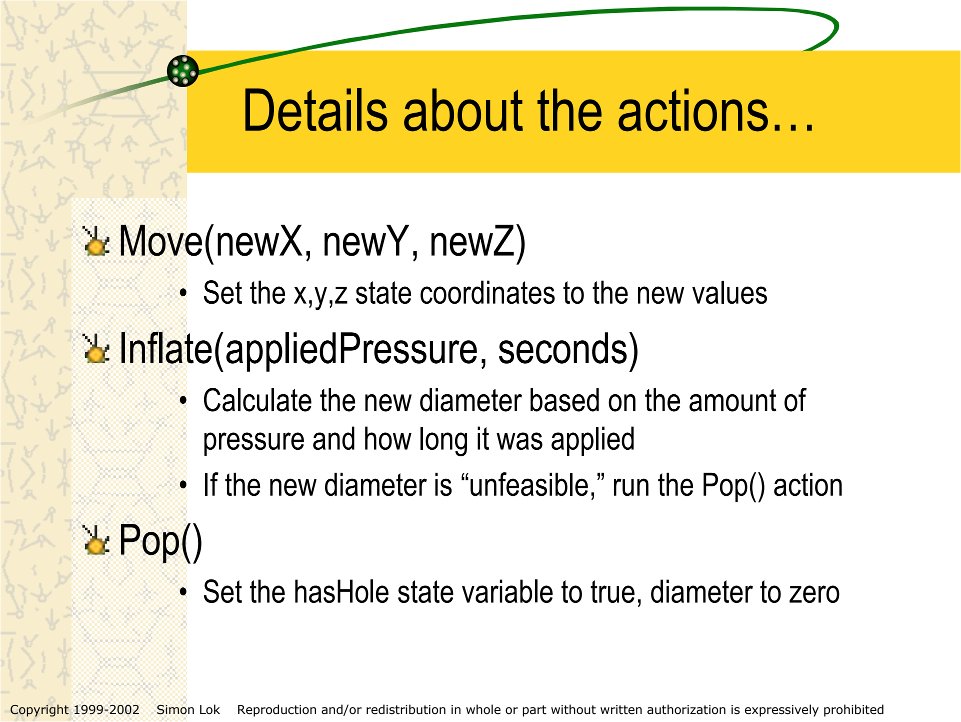
Introducing the object…

- The world is made of objects
- Objects have two aspects to them
  – State variables
  – Actions that modify the state variables
- Goal: “model” real (or abstract) objects
  – Representation is everything!
  – If we can represent the objects and their behavior, we’ve essentially solved our problem



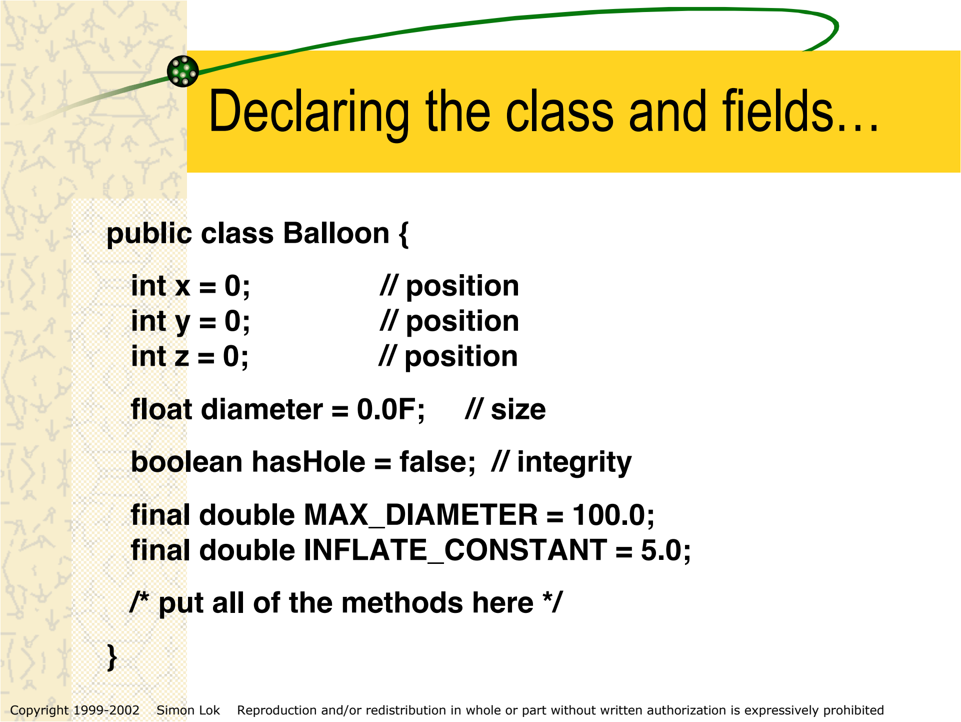
A balloon for starters…

- The state of a balloon can be described by:
  - Position (numerical data - x,y,z)
  - Inflation characteristics (number - diameter)
  - Structural integrity (boolean - hasHole)
- Actions that can be performed include:
  - Move(newX, newY, newZ)
  - Inflate(appliedPressure, numberOfSeconds)
  - Pop()



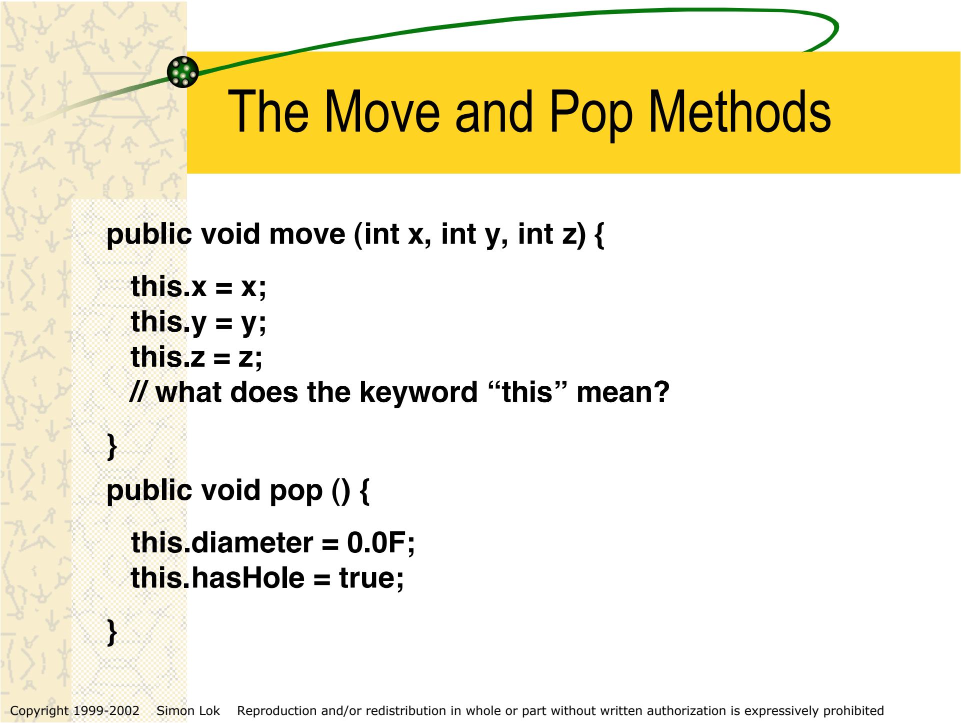
Details about the actions…

- Move(newX, newY, newZ)
  - Set the x,y,z state coordinates to the new values
- Inflate(appliedPressure, seconds)
  - Calculate the new diameter based on the amount of pressure and how long it was applied
  - If the new diameter is “unfeasible,” run the Pop() action
- Pop()
  - Set the hasHole state variable to true, diameter to zero



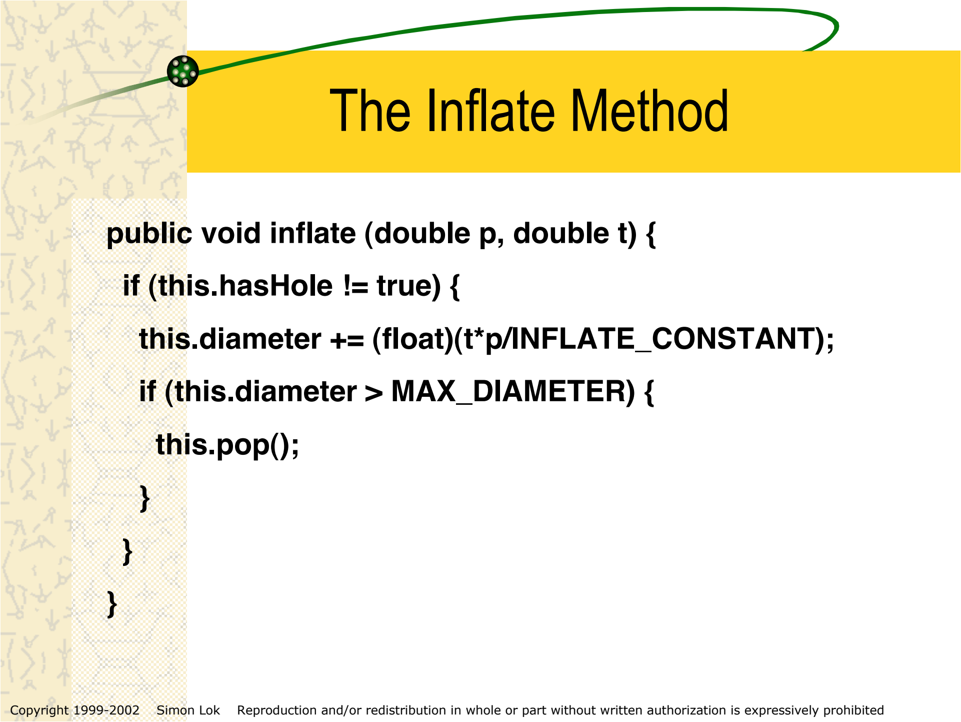
Declaring the class and fields…

```
public class Balloon {

   int x = 0;                // position
   int y = 0;                // position
   int z = 0;                // position

   float diameter = 0.0F;      // size

   boolean hasHole = false;  // integrity

   final double MAX_DIAMETER = 100.0;
   final double INFLATE_CONSTANT = 5.0;

   /* put all of the methods here */

}
```



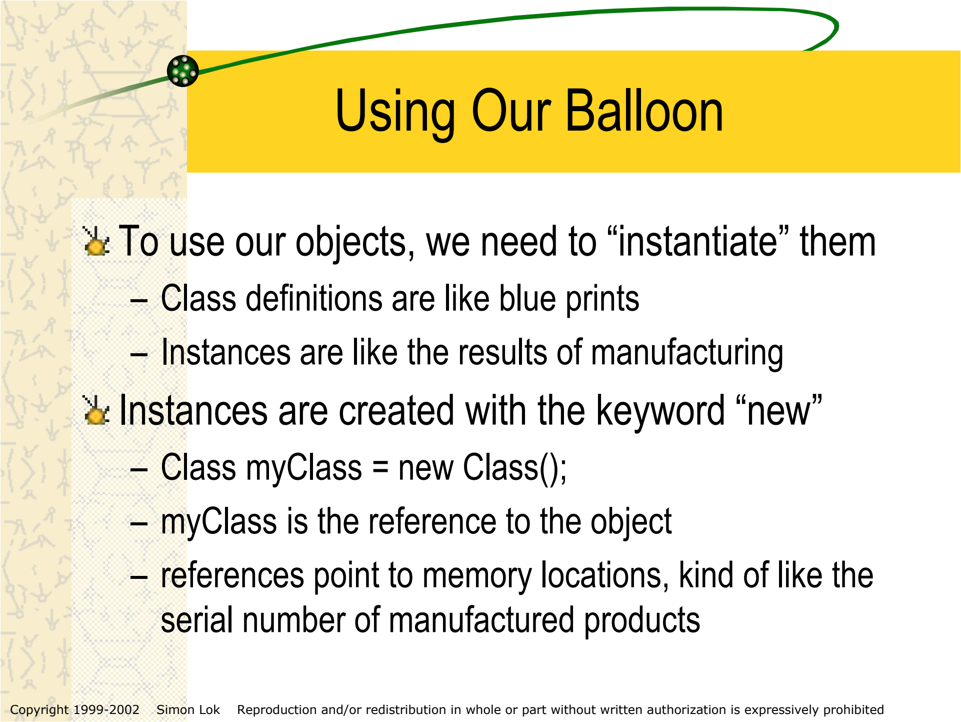
The Move and Pop Methods

```
public void move (int x, int y, int z) {

   this.x = x;
   this.y = y;
   this.z = z;
   // what does the keyword "this" mean?

}
public void pop () {

   this.diameter = 0.0F;
   this.hasHole = true;

}
```



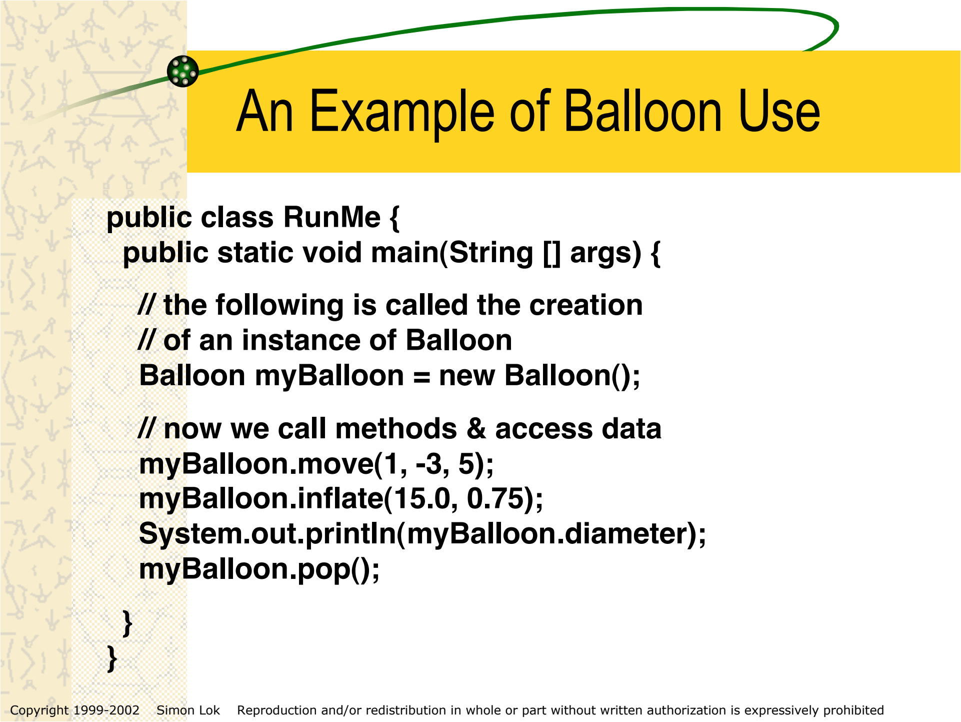
The Inflate Method

```
public void inflate (double p, double t) {
  if (this.hasHole != true) {
    this.diameter += (float)(t*p/INFLATE_CONSTANT);
    if (this.diameter > MAX_DIAMETER) {
      this.pop();
    }
  }
}
```



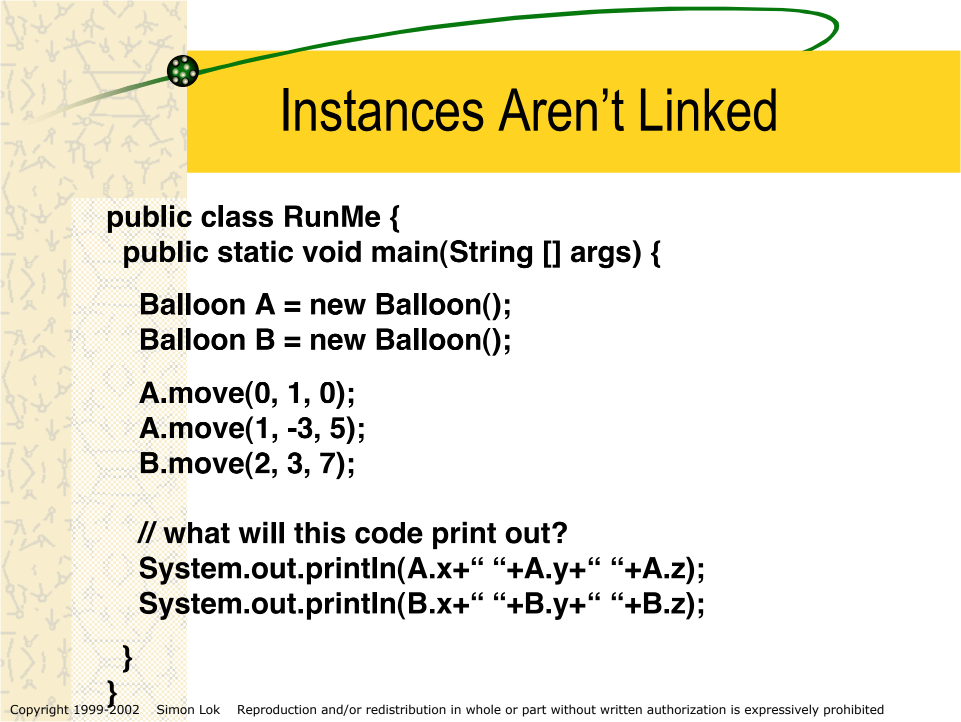
Using Our Balloon

- To use our objects, we need to “instantiate” them
  - Class definitions are like blue prints
  - Instances are like the results of manufacturing
- Instances are created with the keyword “new”
  - Class myClass = new Class();
  - myClass is the reference to the object
  - references point to memory locations, kind of like the serial number of manufactured products



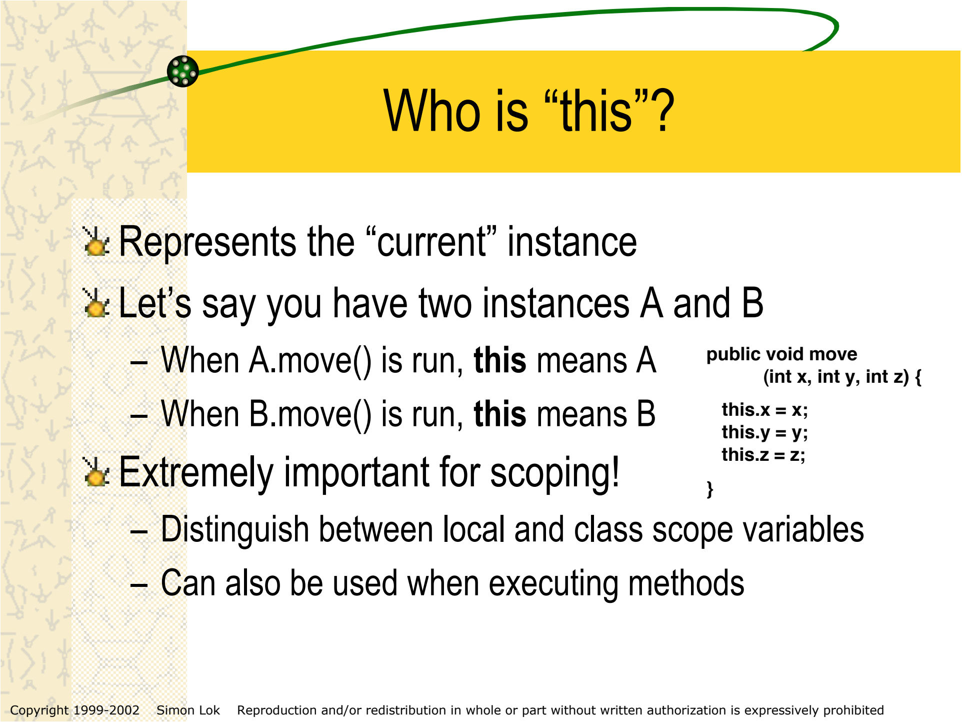
An Example of Balloon Use

```
public class RunMe {
  public static void main(String [] args) {

    // the following is called the creation
    // of an instance of Balloon
    Balloon myBalloon = new Balloon();

    // now we call methods & access data
    myBalloon.move(1, -3, 5);
    myBalloon.inflate(15.0, 0.75);
    System.out.println(myBalloon.diameter);
    myBalloon.pop();

  }
}
```



Instances Aren’t Linked

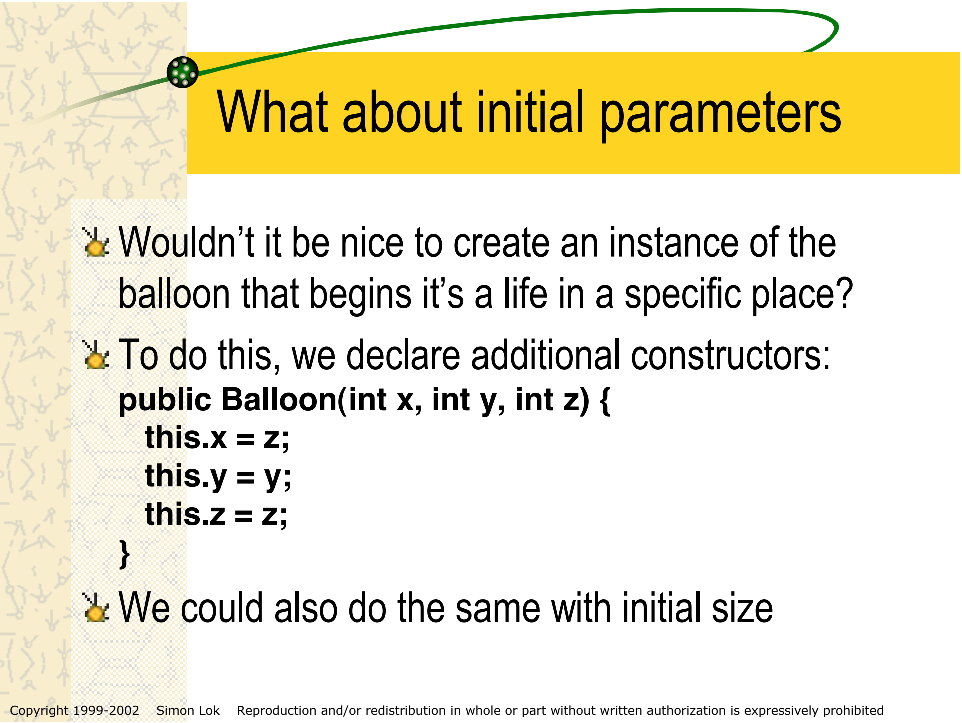
```
public class RunMe {
  public static void main(String [] args) {

    Balloon A = new Balloon();
    Balloon B = new Balloon();

    A.move(0, 1, 0);
    A.move(1, -3, 5);
    B.move(2, 3, 7);

    // what will this code print out?
    System.out.println(A.x+“ “+A.y+“ “+A.z);
    System.out.println(B.x+“ “+B.y+“ “+B.z);

  }
}
```



Who is “this”?

- Represents the “current” instance
- Let’s say you have two instances A and B
  - When A.move() is run, **this** means A
  - When B.move() is run, **this** means B
- Extremely important for scoping!
  - Distinguish between local and class scope variables
  - Can also be used when executing methods

```
public void move
        (int x, int y, int z) {

  this.x = x;
  this.y = y;
  this.z = z;
}
```

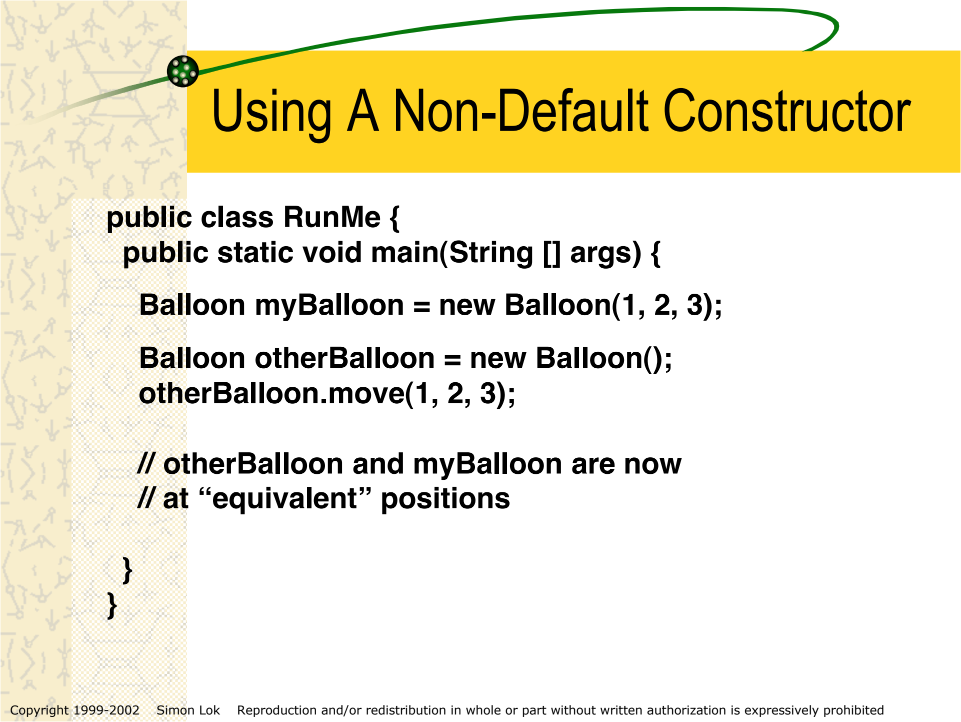


What about initial parameters

- Wouldn’t it be nice to create an instance of the balloon that begins it’s a life in a specific place?
- To do this, we declare additional constructors:

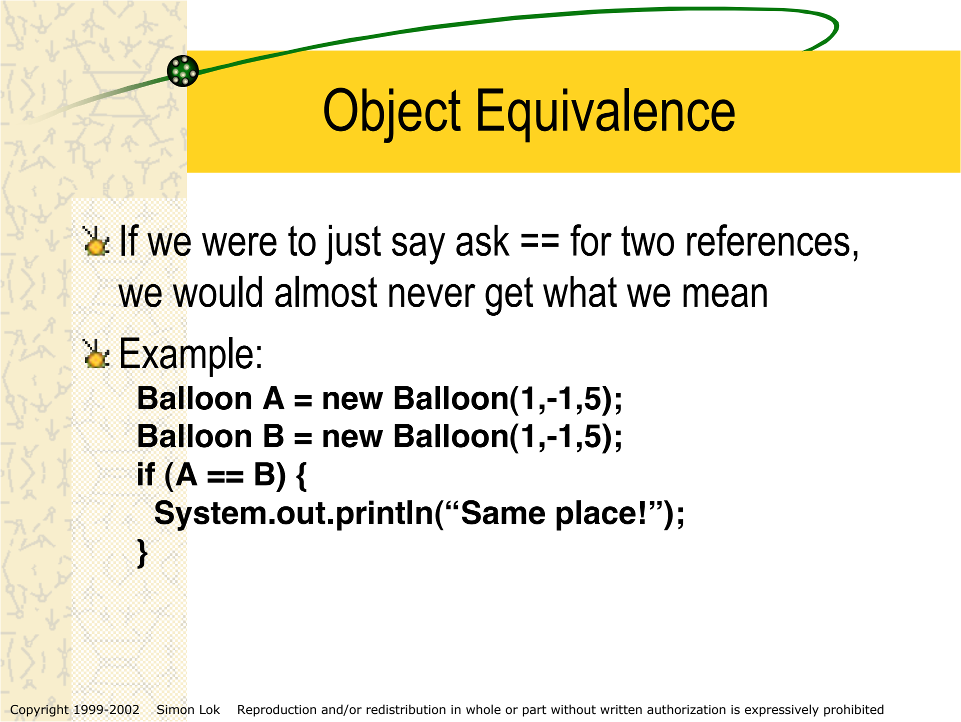
```
public Balloon(int x, int y, int z) {
  this.x = z;
  this.y = y;
  this.z = z;
}
```

- We could also do the same with initial size



Using A Non-Default Constructor

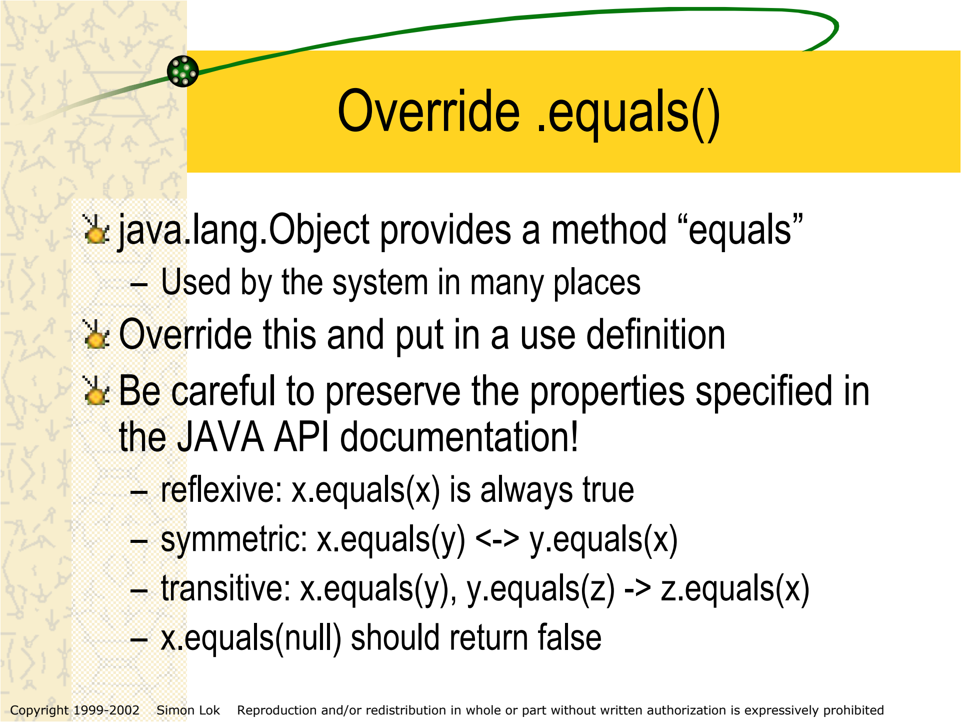
```
public class RunMe {
  public static void main(String [] args) {

    Balloon myBalloon = new Balloon(1, 2, 3);

    Balloon otherBalloon = new Balloon();
    otherBalloon.move(1, 2, 3);

    // otherBalloon and myBalloon are now
    // at “equivalent” positions

  }
}
```



Object Equivalence

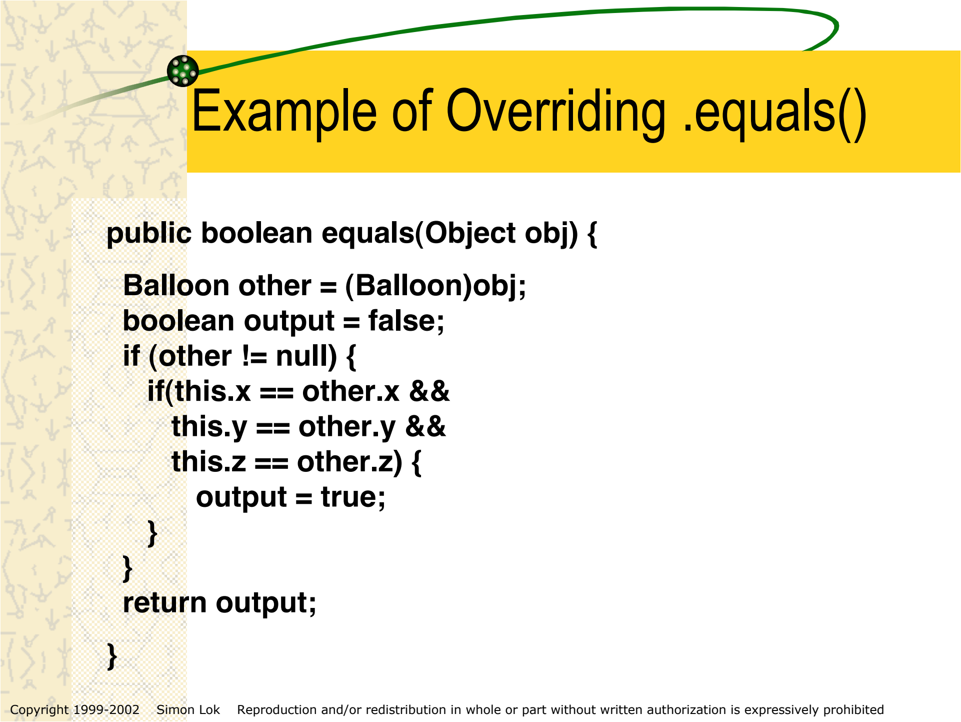
- If we were to just say ask == for two references, we would almost never get what we mean
- Example:

```
Balloon A = new Balloon(1,-1,5);
Balloon B = new Balloon(1,-1,5);
if (A == B) {
  System.out.println(“Same place!”);
}
```



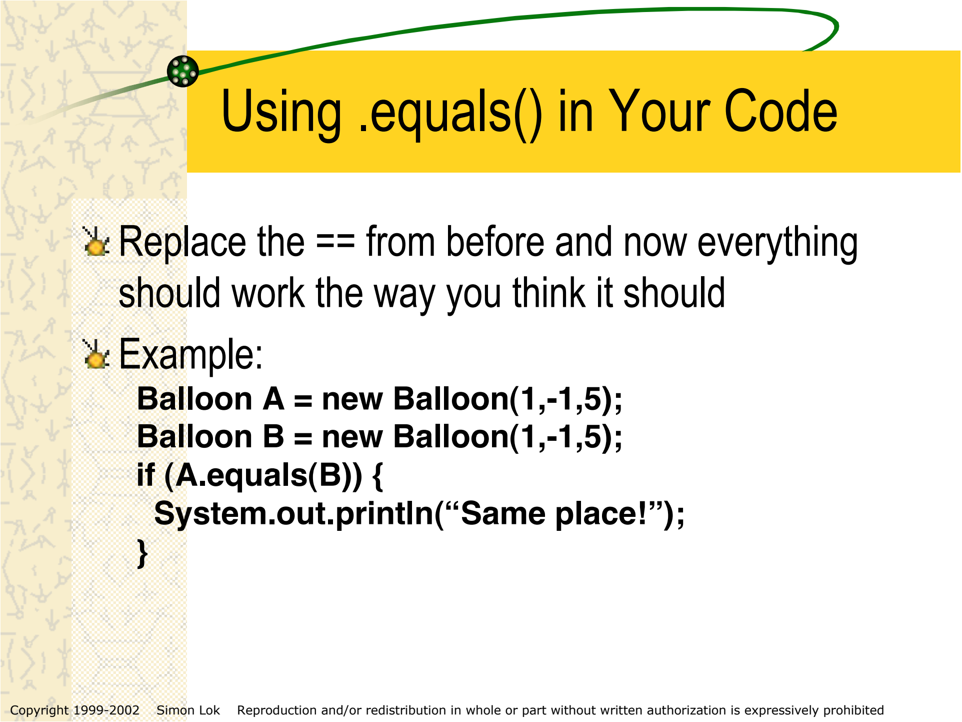
Override .equals()

- java.lang.Object provides a method “equals”
  - Used by the system in many places
- Override this and put in a use definition
- Be careful to preserve the properties specified in the JAVA API documentation!
  - reflexive: x.equals(x) is always true
  - symmetric: x.equals(y) <-> y.equals(x)
  - transitive: x.equals(y), y.equals(z) -> z.equals(x)
  - x.equals(null) should return false



Example of Overriding .equals()

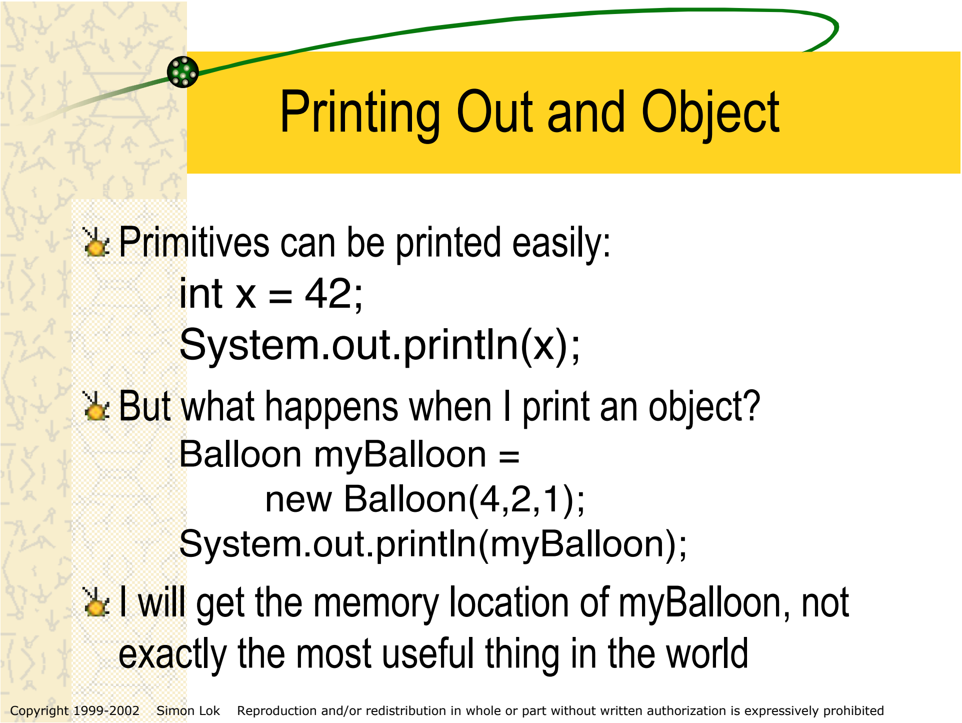
```
public boolean equals(Object obj) {

  Balloon other = (Balloon)obj;
  boolean output = false;
  if (other != null) {
    if(this.x == other.x &&
      this.y == other.y &&
      this.z == other.z) {
        output = true;
    }
  }
  return output;

}
```



Using .equals() in Your Code

- Replace the == from before and now everything should work the way you think it should
- Example:

```
Balloon A = new Balloon(1,-1,5);
Balloon B = new Balloon(1,-1,5);
if (A.equals(B)) {
  System.out.println("Same place!");
}
```



Printing Out and Object

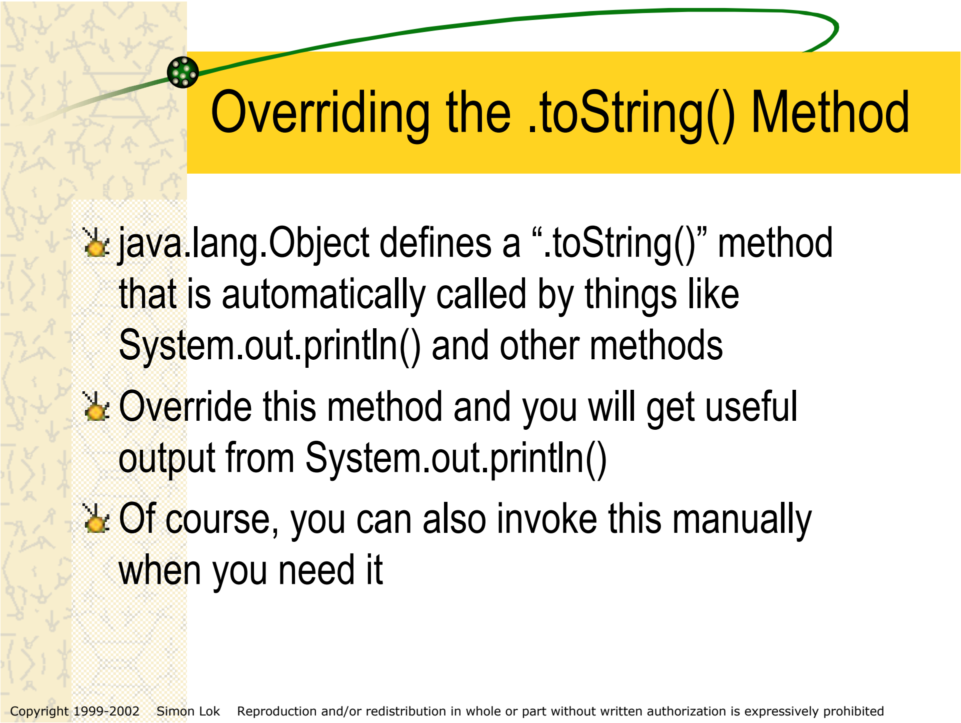
- Primitives can be printed easily:

```
int x = 42;
System.out.println(x);
```

- But what happens when I print an object?

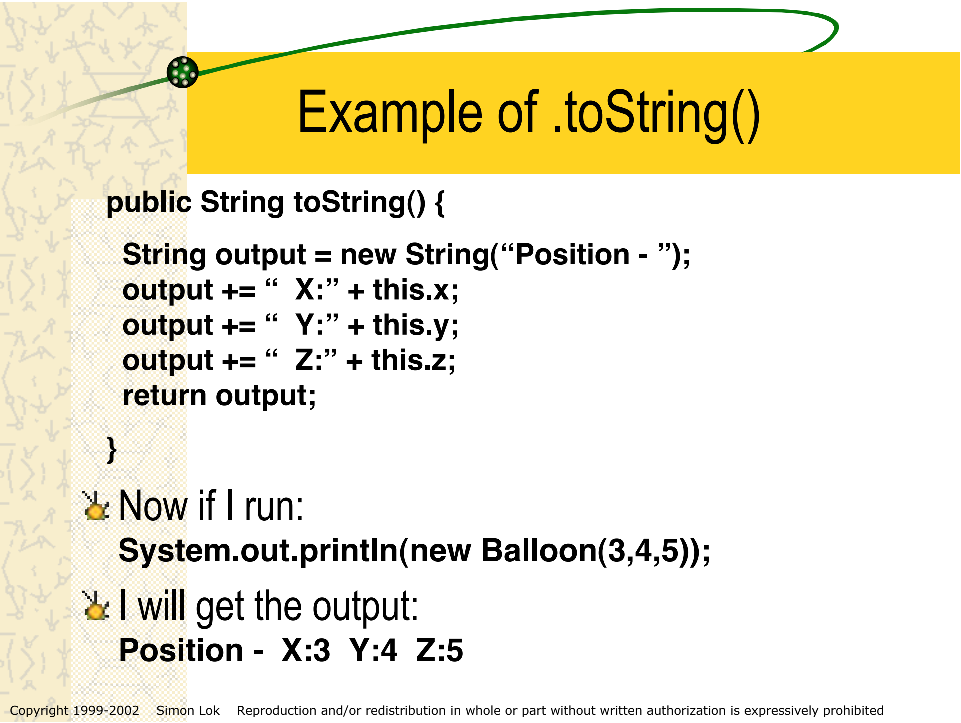
```
Balloon myBalloon =
        new Balloon(4,2,1);
System.out.println(myBalloon);
```

- I will get the memory location of myBalloon, not exactly the most useful thing in the world



Overriding the .toString() Method

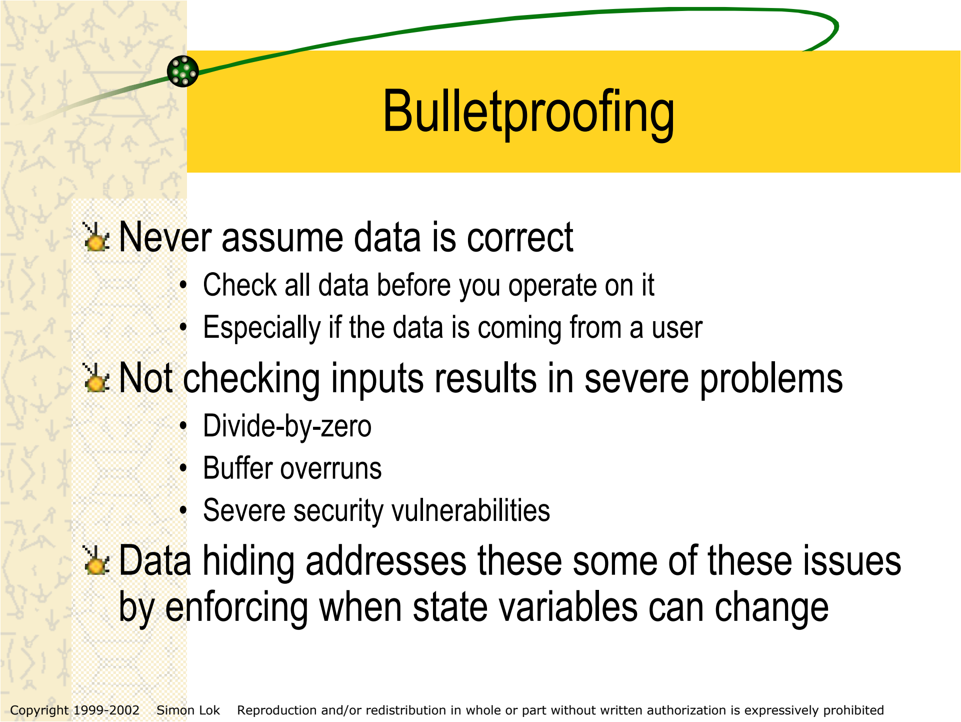
- java.lang.Object defines a “.toString()” method that is automatically called by things like System.out.println() and other methods
- Override this method and you will get useful output from System.out.println()
- Of course, you can also invoke this manually when you need it



Example of .toString()

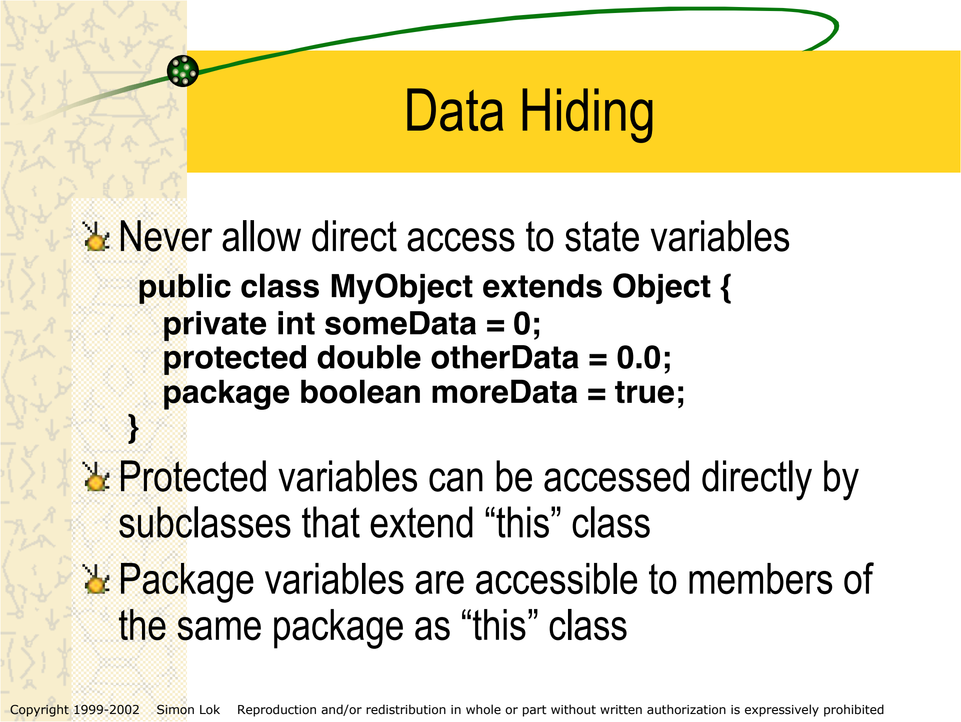
```
public String toString() {

  String output = new String(“Position - ”);
  output += “  X:” + this.x;
  output += “  Y:” + this.y;
  output += “  Z:” + this.z;
  return output;

}
```

- Now if I run:
  System.out.println(new Balloon(3,4,5));
- I will get the output:
  Position - X:3 Y:4 Z:5



Bulletproofing

- Never assume data is correct
  - Check all data before you operate on it
  - Especially if the data is coming from a user
- Not checking inputs results in severe problems
  - Divide-by-zero
  - Buffer overruns
  - Severe security vulnerabilities
- Data hiding addresses these some of these issues by enforcing when state variables can change

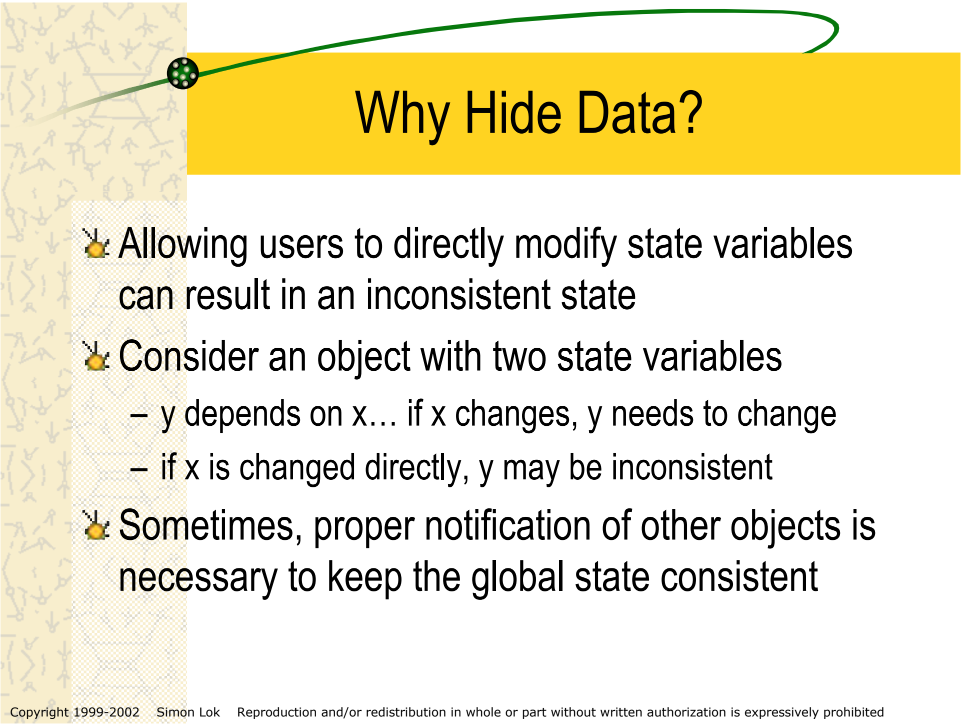


Data Hiding

- Never allow direct access to state variables

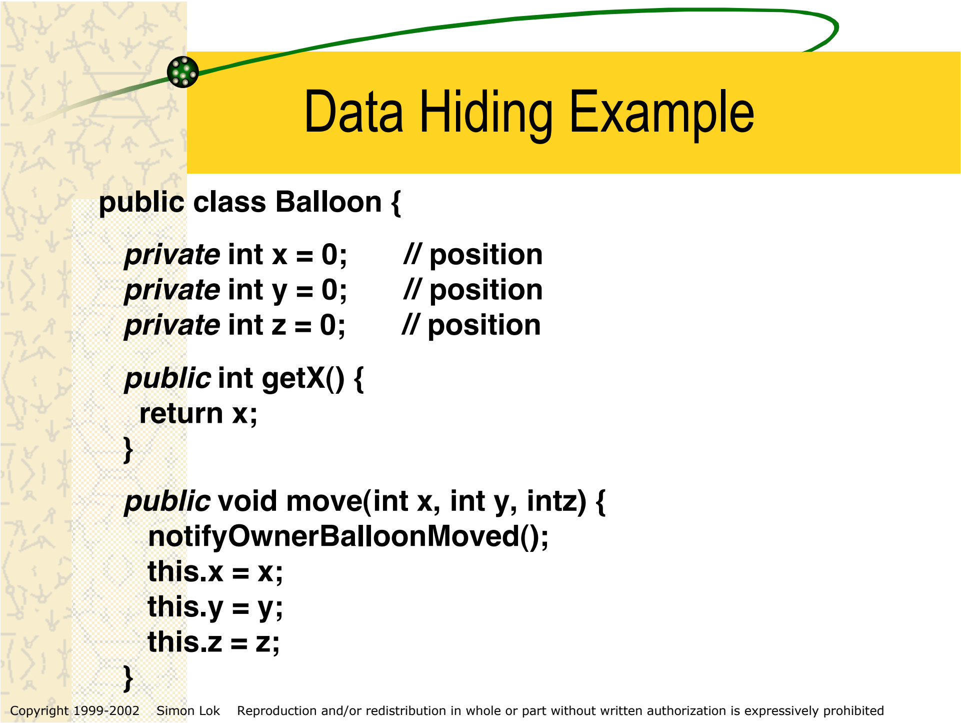
```
public class MyObject extends Object {
  private int someData = 0;
  protected double otherData = 0.0;
  package boolean moreData = true;
}
```

- Protected variables can be accessed directly by subclasses that extend “this” class
- Package variables are accessible to members of the same package as “this” class



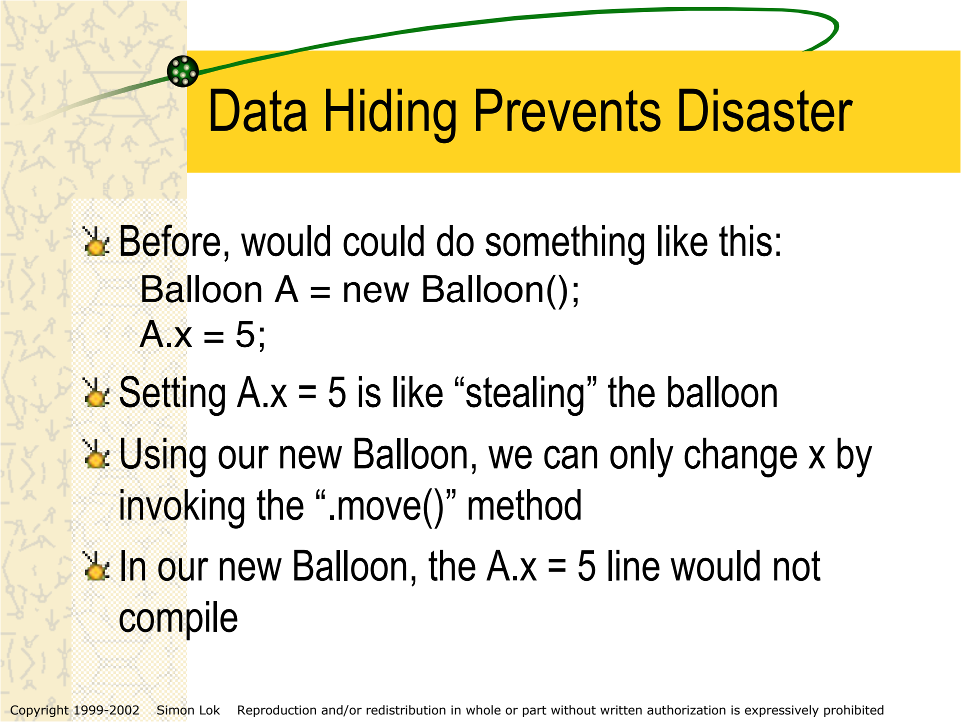
Why Hide Data?

- Allowing users to directly modify state variables can result in an inconsistent state
- Consider an object with two state variables
  - y depends on x… if x changes, y needs to change
  - if x is changed directly, y may be inconsistent
- Sometimes, proper notification of other objects is necessary to keep the global state consistent



Data Hiding Example

```
public class Balloon {

  private int x = 0;      // position
  private int y = 0;      // position
  private int z = 0;      // position

  public int getX() {
    return x;
  }

  public void move(int x, int y, intz) {
    notifyOwnerBalloonMoved();
    this.x = x;
    this.y = y;
    this.z = z;
  }
```

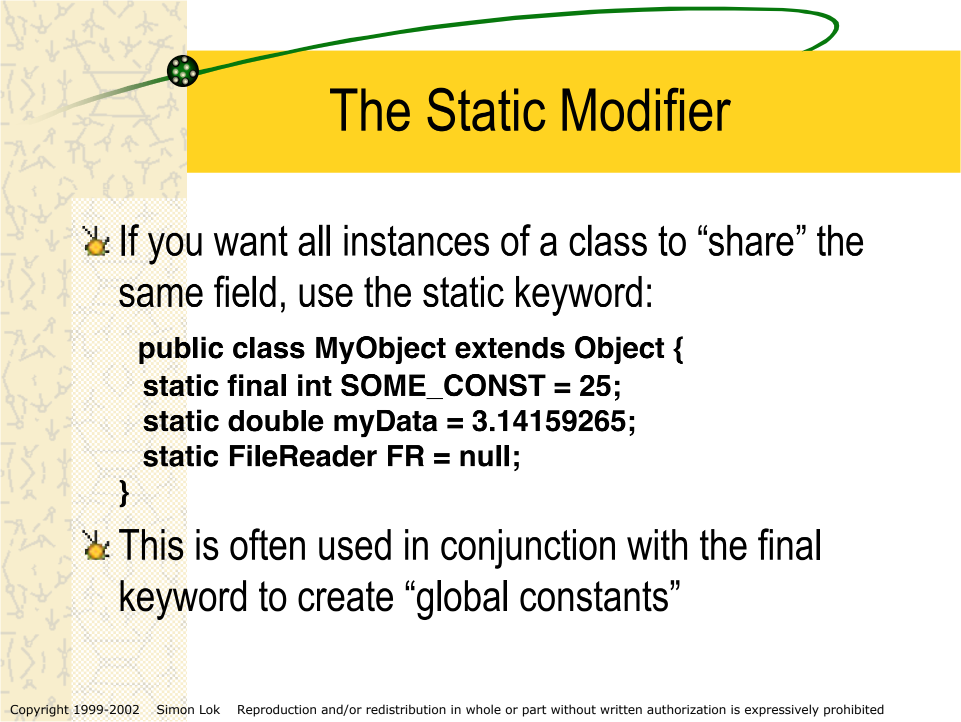


Data Hiding Prevents Disaster

- Before, would could do something like this:

```
Balloon A = new Balloon();
A.x = 5;
```

- Setting A.x = 5 is like “stealing” the balloon
- Using our new Balloon, we can only change x by invoking the “.move()” method
- In our new Balloon, the A.x = 5 line would not compile

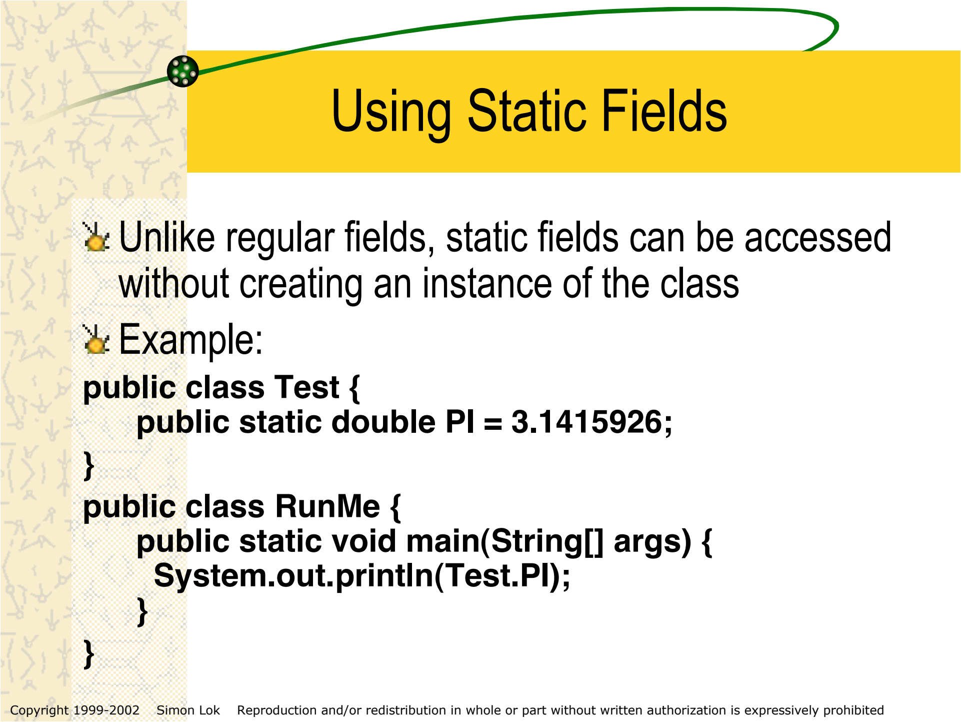


The Static Modifier

- If you want all instances of a class to “share” the same field, use the static keyword:

```
public class MyObject extends Object {
 static final int SOME_CONST = 25;
 static double myData = 3.14159265;
 static FileReader FR = null;
}
```

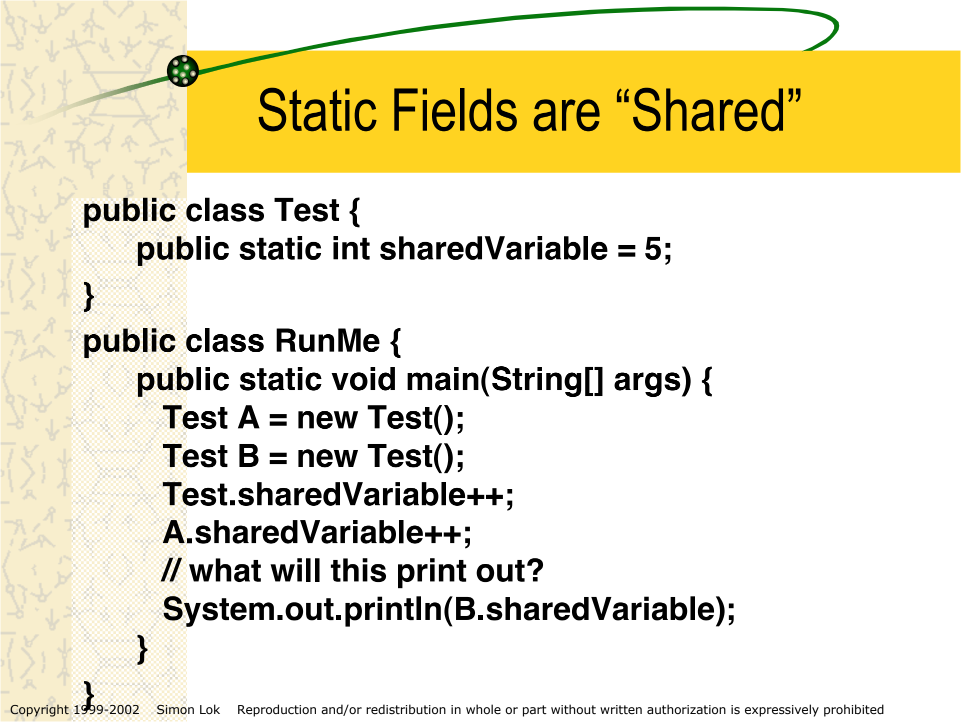
- This is often used in conjunction with the final keyword to create “global constants”



Using Static Fields

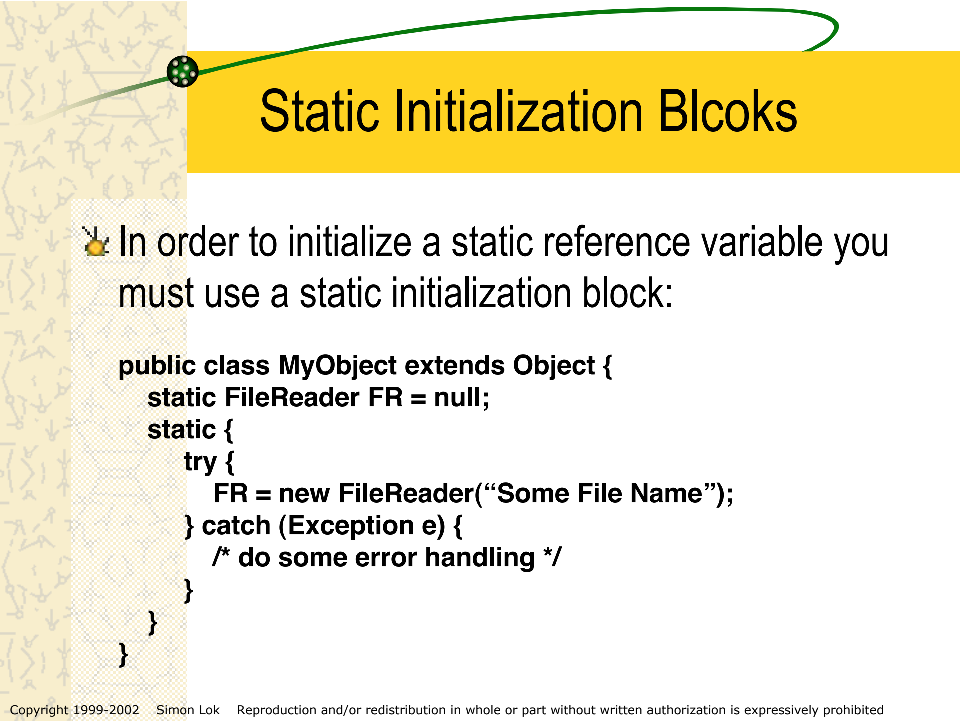
- Unlike regular fields, static fields can be accessed without creating an instance of the class
- Example:

```
public class Test {
      public static double PI = 3.1415926;
}
public class RunMe {
      public static void main(String[] args) {
        System.out.println(Test.PI);
      }
}
```



Static Fields are “Shared”

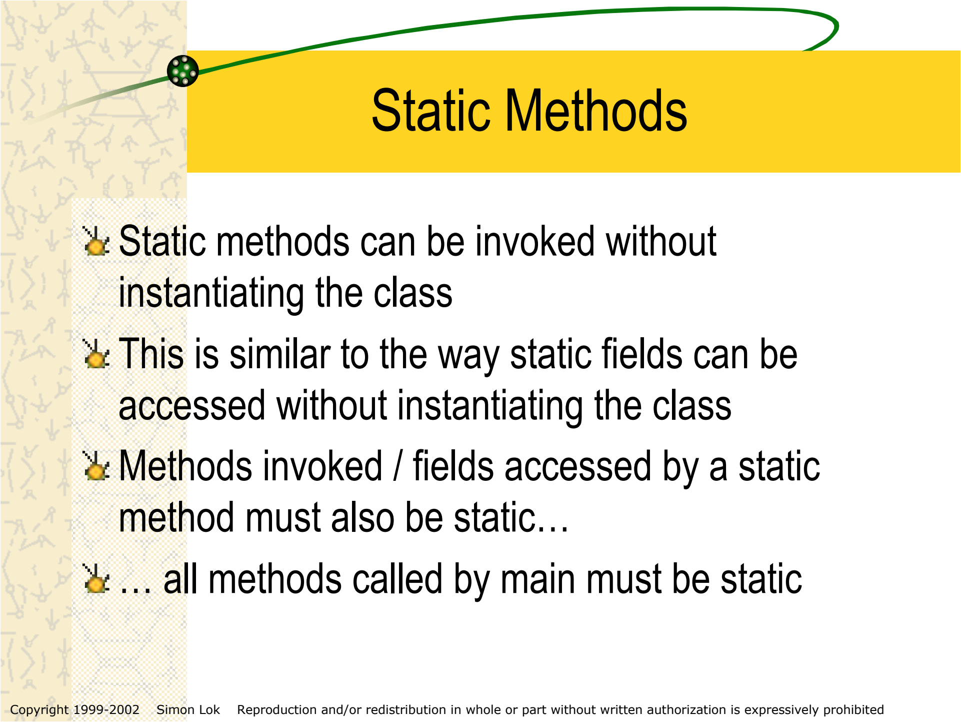
```
public class Test {
      public static int sharedVariable = 5;
}
public class RunMe {
      public static void main(String[] args) {
         Test A = new Test();
         Test B = new Test();
         Test.sharedVariable++;
         A.sharedVariable++;
         // what will this print out?
         System.out.println(B.sharedVariable);
      }
}
```



Static Initialization Blcoks

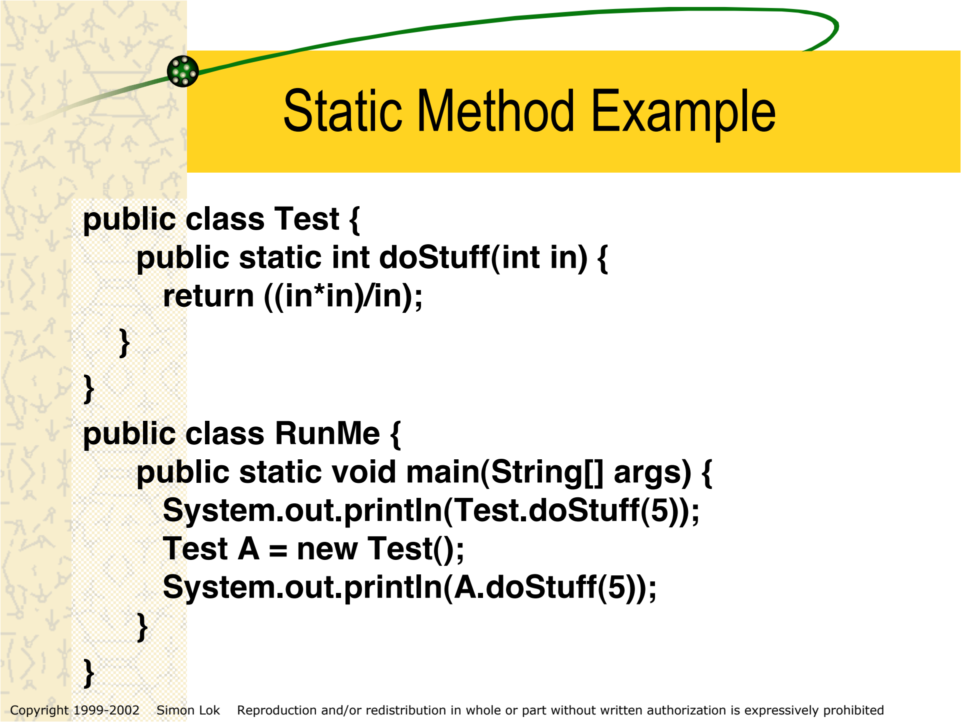
- In order to initialize a static reference variable you must use a static initialization block:

```
public class MyObject extends Object {
   static FileReader FR = null;
   static {
      try {
         FR = new FileReader("Some File Name");
      } catch (Exception e) {
         /* do some error handling */
      }
   }
}
```



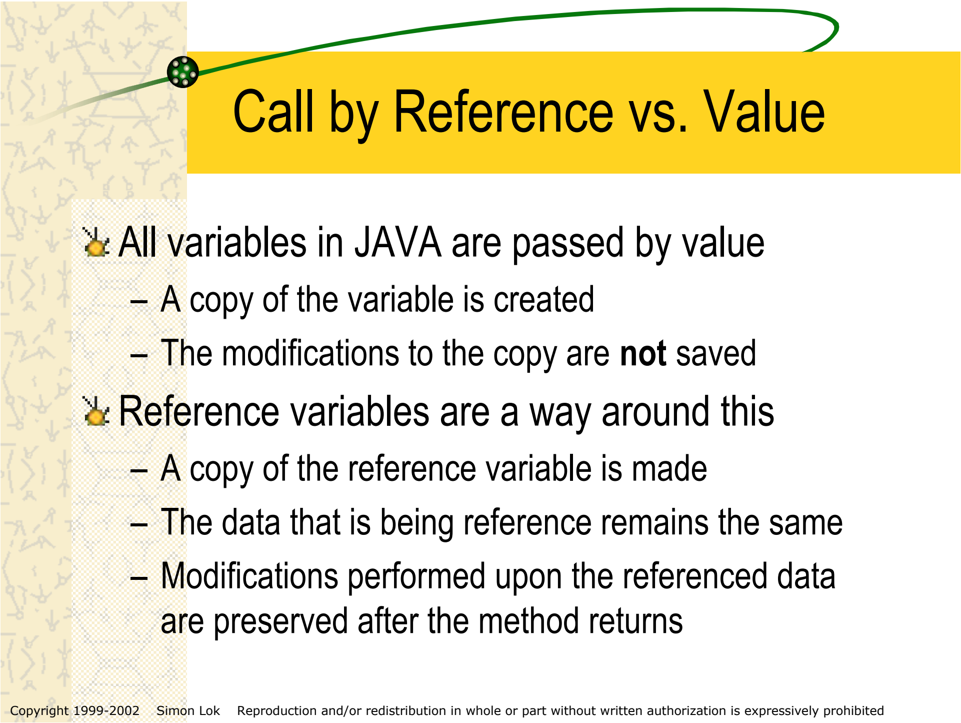
Static Methods

- Static methods can be invoked without instantiating the class
- This is similar to the way static fields can be accessed without instantiating the class
- Methods invoked / fields accessed by a static method must also be static…
- … all methods called by main must be static



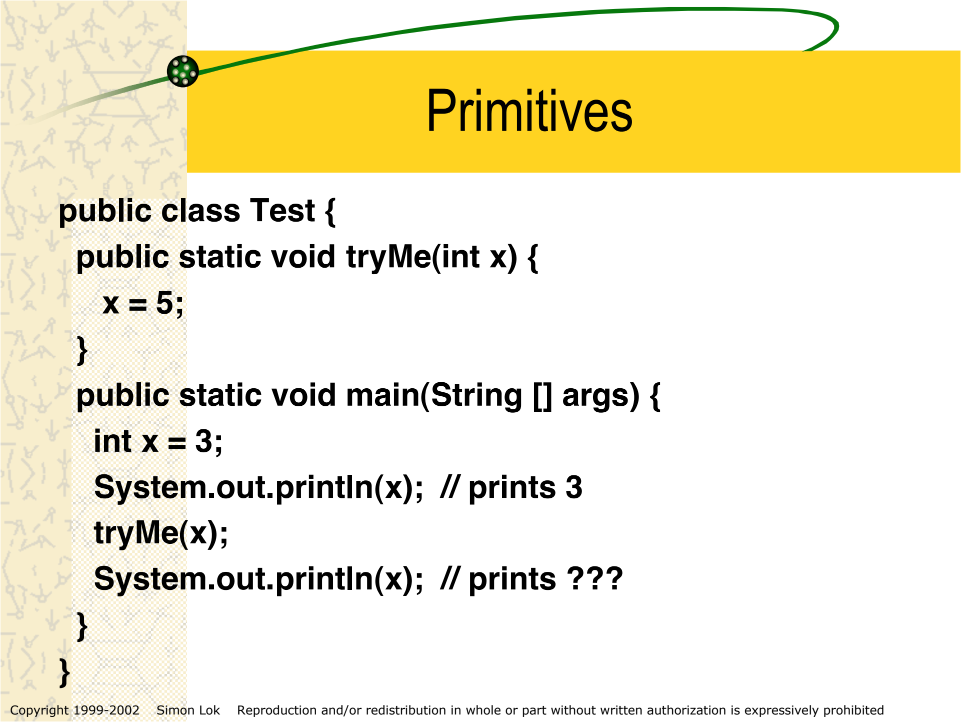
Static Method Example

```
public class Test {
    public static int doStuff(int in) {
      return ((in*in)/in);
   }
}
public class RunMe {
    public static void main(String[] args) {
      System.out.println(Test.doStuff(5));
      Test A = new Test();
      System.out.println(A.doStuff(5));
     }
}
```



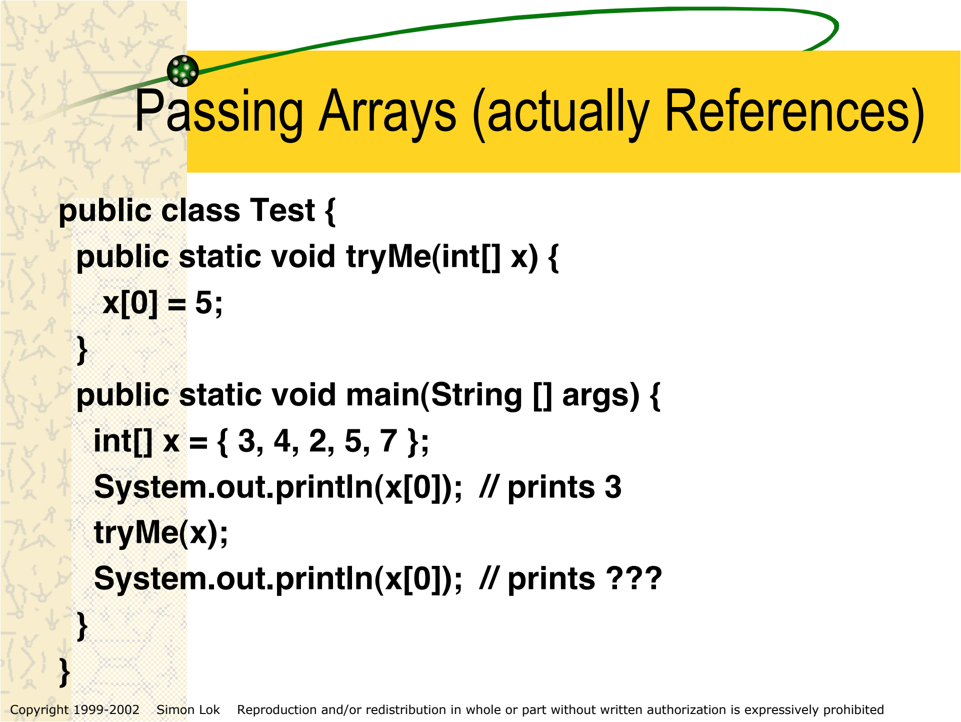
Call by Reference vs. Value

- All variables in JAVA are passed by value
  - A copy of the variable is created
  - The modifications to the copy are **not** saved
- Reference variables are a way around this
  - A copy of the reference variable is made
  - The data that is being reference remains the same
  - Modifications performed upon the referenced data are preserved after the method returns



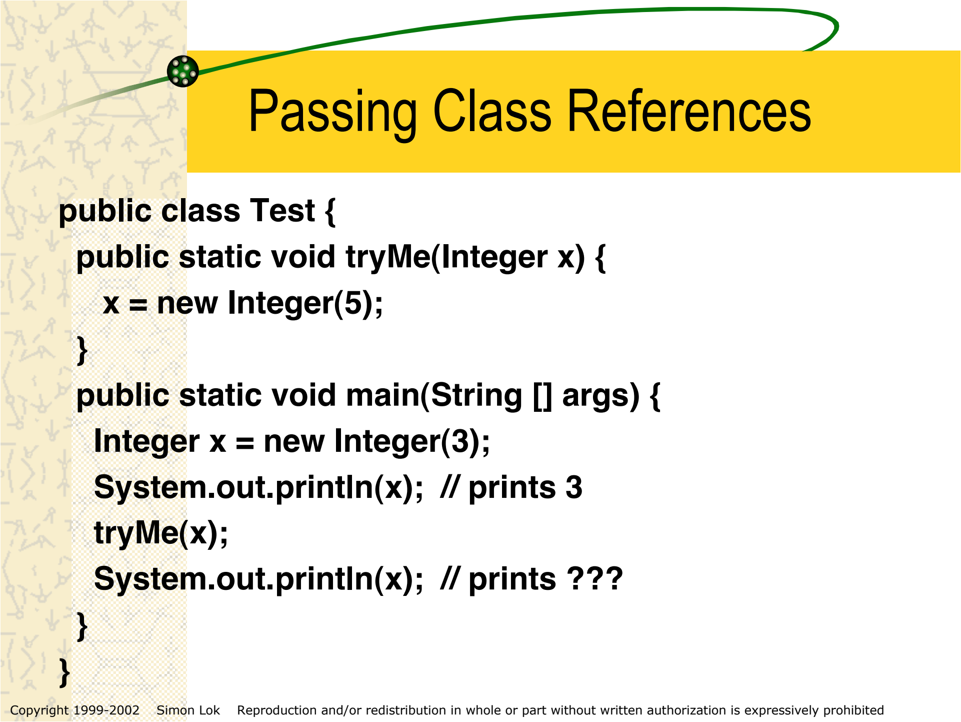
Primitives

```
public class Test {
  public static void tryMe(int x) {
     x = 5;
  }
  public static void main(String [] args) {
    int x = 3;
    System.out.println(x);  // prints 3
    tryMe(x);
    System.out.println(x);  // prints ???
  }
}
```



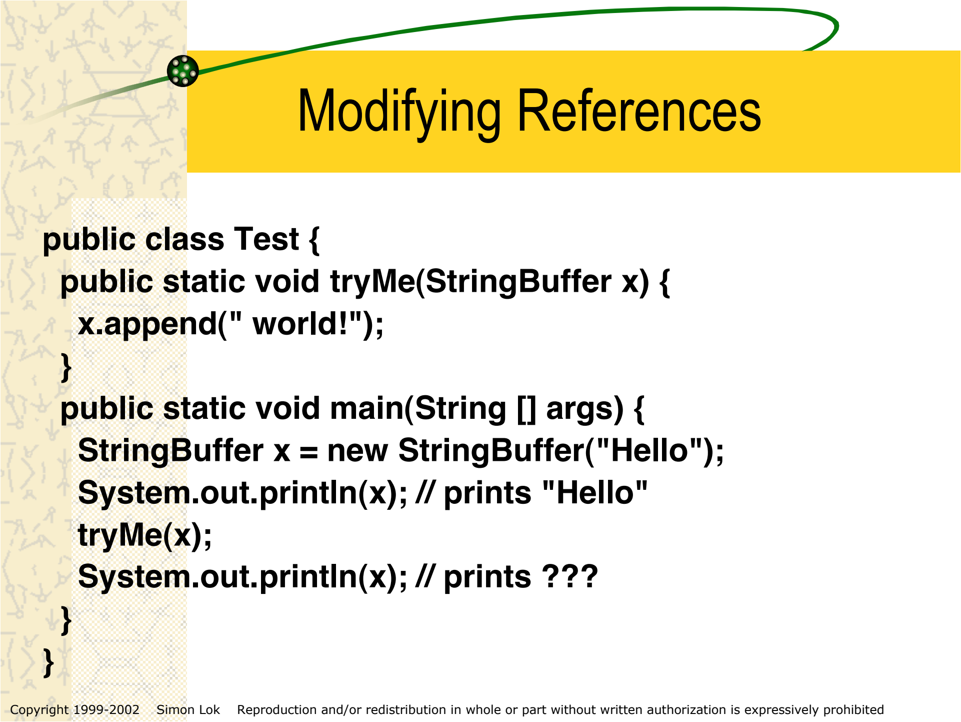
Passing Arrays (actually References)

```
public class Test {
  public static void tryMe(int[] x) {
     x[0] = 5;
  }
  public static void main(String [] args) {
    int[] x = { 3, 4, 2, 5, 7 };
    System.out.println(x[0]);  // prints 3
    tryMe(x);
    System.out.println(x[0]);  // prints ???
  }
}
```



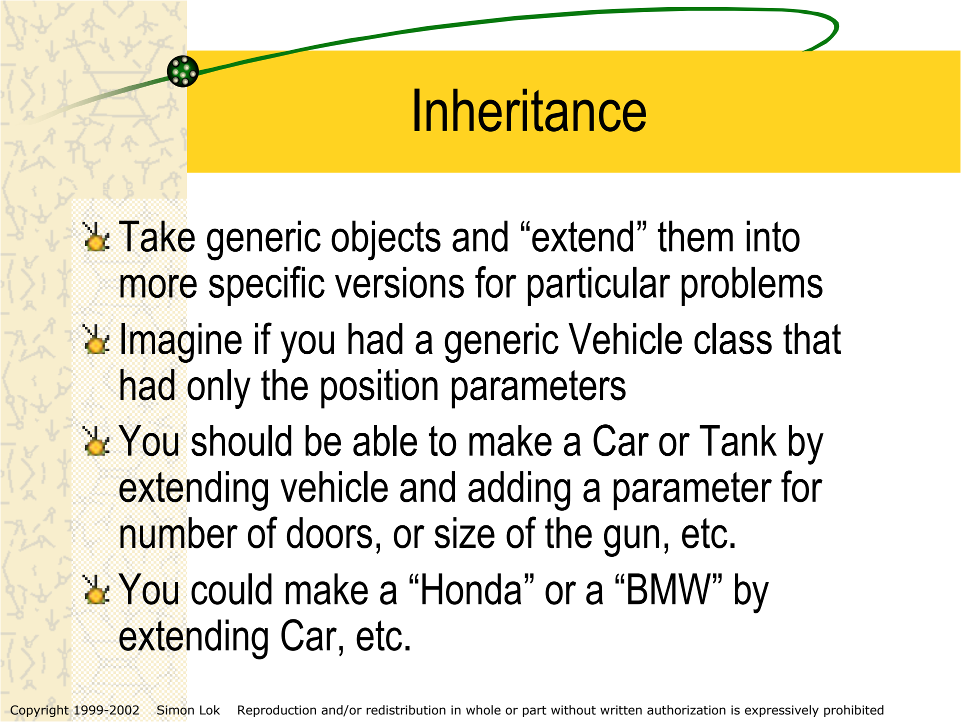
Passing Class References

```
public class Test {
  public static void tryMe(Integer x) {
     x = new Integer(5);
  }
  public static void main(String [] args) {
    Integer x = new Integer(3);
    System.out.println(x);  // prints 3
    tryMe(x);
    System.out.println(x);  // prints ???
  }
}
```



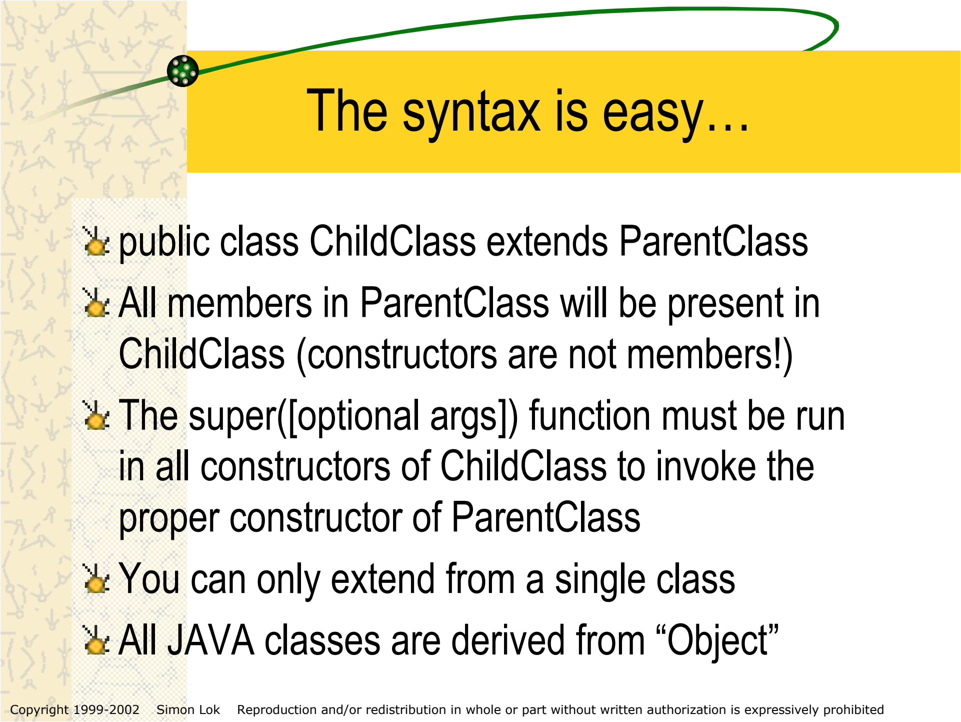
Modifying References

```
public class Test {
  public static void tryMe(StringBuffer x) {
    x.append(" world!");
  }
  public static void main(String [] args) {
    StringBuffer x = new StringBuffer("Hello");
    System.out.println(x); // prints "Hello"
    tryMe(x);
    System.out.println(x); // prints ???
  }
}
```



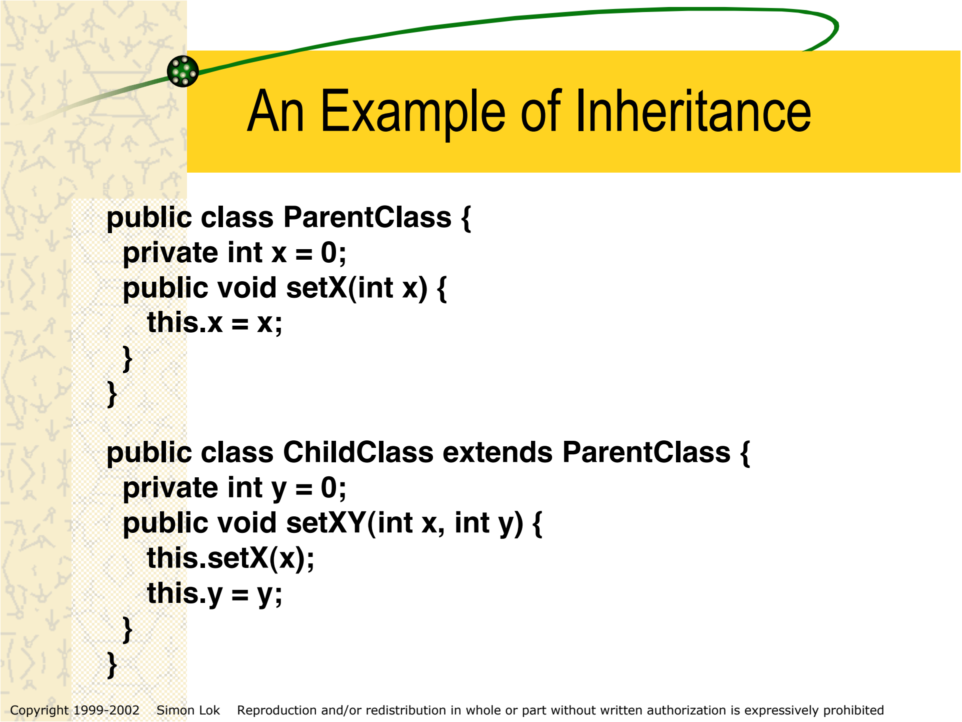
Inheritance

- Take generic objects and “extend” them into more specific versions for particular problems
- Imagine if you had a generic Vehicle class that had only the position parameters
- You should be able to make a Car or Tank by extending vehicle and adding a parameter for number of doors, or size of the gun, etc.
- You could make a “Honda” or a “BMW” by extending Car, etc.



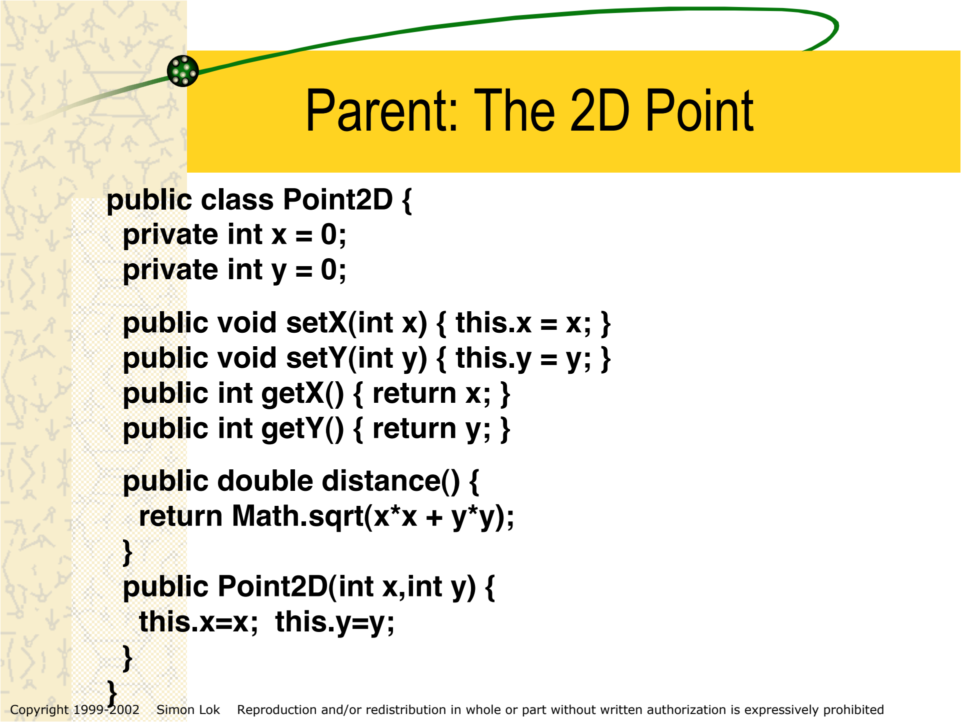
The syntax is easy…

- public class ChildClass extends ParentClass
- All members in ParentClass will be present in ChildClass (constructors are not members!)
- The super([optional args]) function must be run in all constructors of ChildClass to invoke the proper constructor of ParentClass
- You can only extend from a single class
- All JAVA classes are derived from “Object”



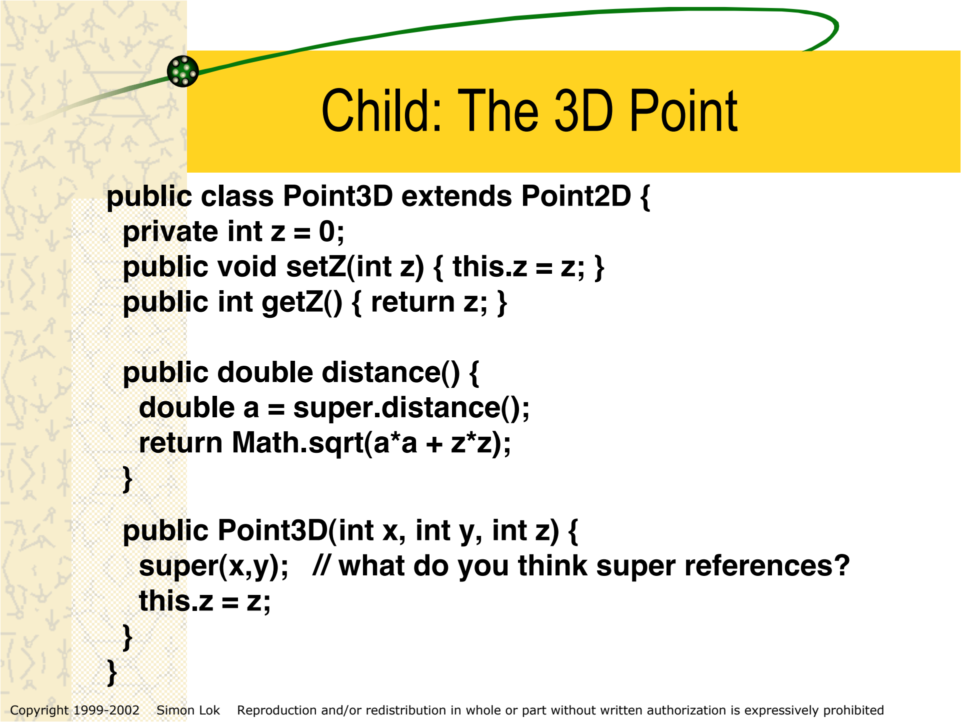
An Example of Inheritance

```
public class ParentClass {
  private int x = 0;
  public void setX(int x) {
     this.x = x;
  }
}

public class ChildClass extends ParentClass {
  private int y = 0;
  public void setXY(int x, int y) {
     this.setX(x);
     this.y = y;
  }
}
```



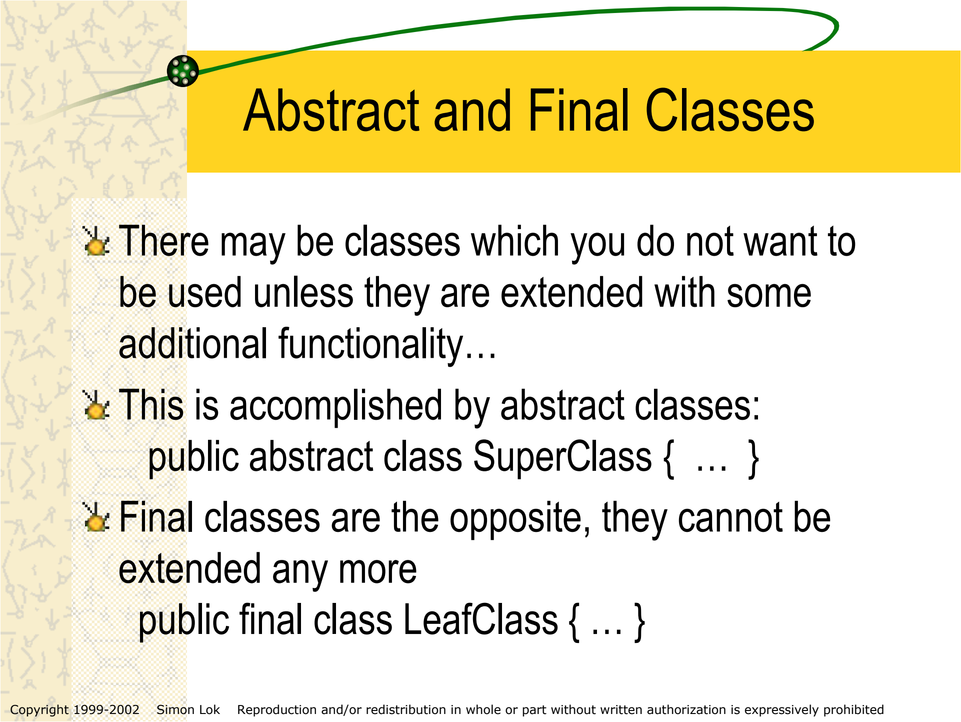
Parent: The 2D Point

```
public class Point2D {
  private int x = 0;
  private int y = 0;

  public void setX(int x) { this.x = x; }
  public void setY(int y) { this.y = y; }
  public int getX() { return x; }
  public int getY() { return y; }

  public double distance() {
    return Math.sqrt(x*x + y*y);
  }
  public Point2D(int x,int y) {
    this.x=x;  this.y=y;
  }
}
```



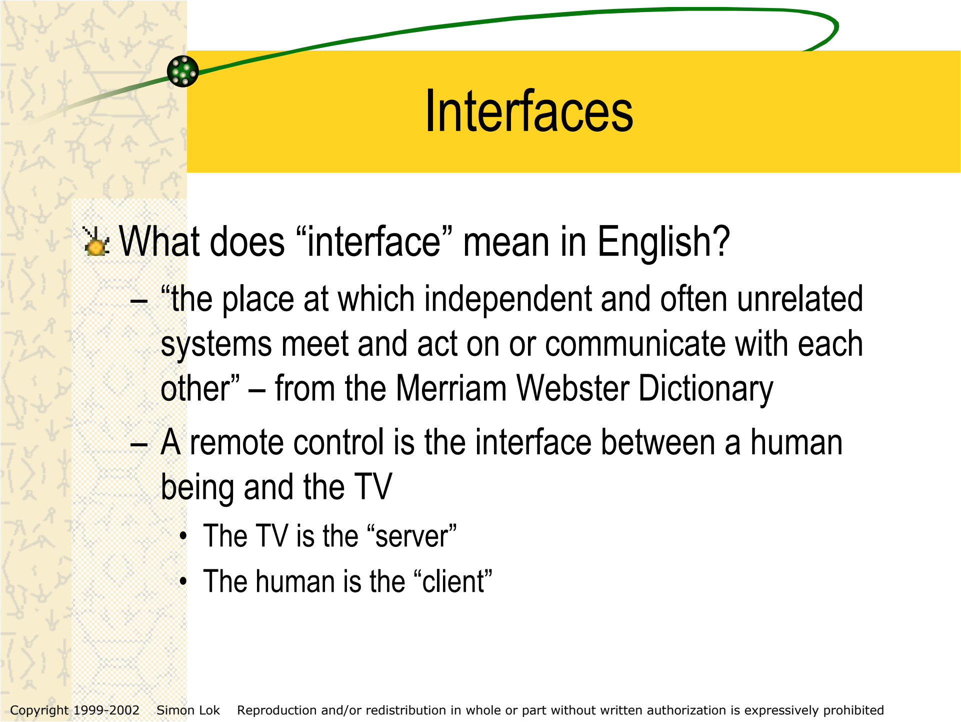
Child: The 3D Point

```
public class Point3D extends Point2D {
  private int z = 0;
  public void setZ(int z) { this.z = z; }
  public int getZ() { return z; }

  public double distance() {
    double a = super.distance();
    return Math.sqrt(a*a + z*z);
  }

  public Point3D(int x, int y, int z) {
    super(x,y);   // what do you think super references?
    this.z = z;
  }
}
```



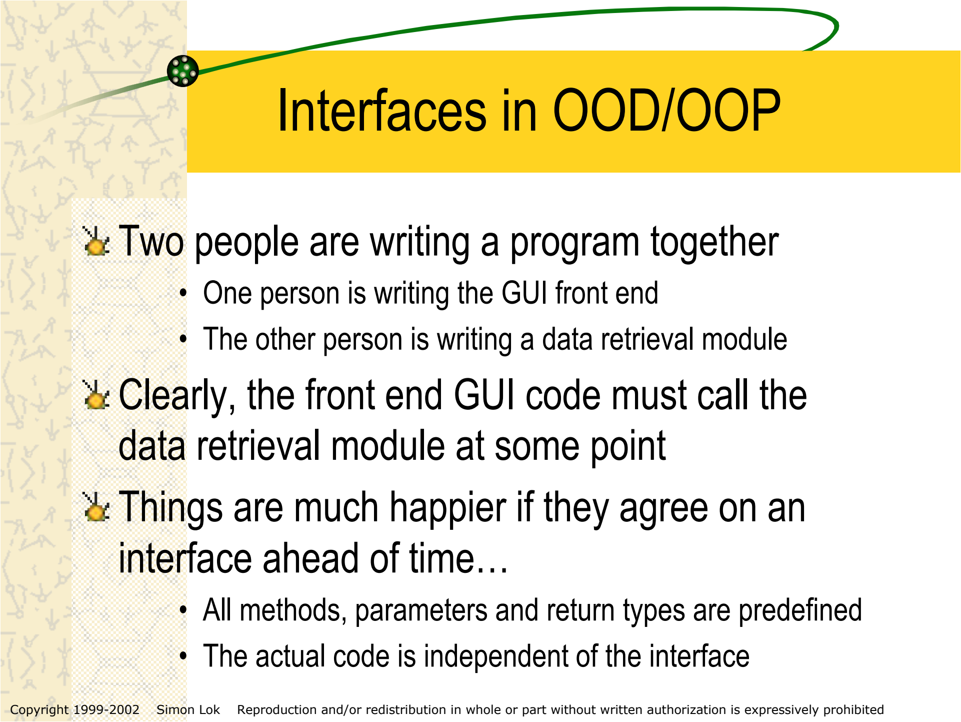
Abstract and Final Classes

- There may be classes which you do not want to be used unless they are extended with some additional functionality…
- This is accomplished by abstract classes:
  public abstract class SuperClass { … }
- Final classes are the opposite, they cannot be extended any more
  public final class LeafClass { … }



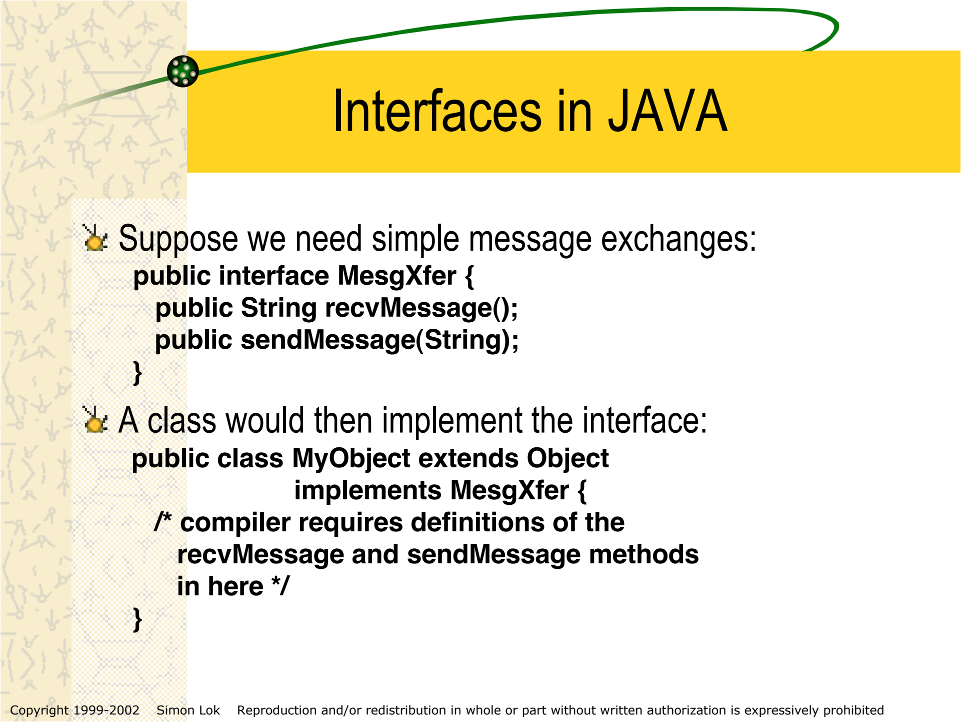
Interfaces

- What does “interface” mean in English?
  - “the place at which independent and often unrelated systems meet and act on or communicate with each other” – from the Merriam Webster Dictionary
  - A remote control is the interface between a human being and the TV
    - The TV is the “server”
    - The human is the “client”



Interfaces in OOD/OOP

- Two people are writing a program together
    - One person is writing the GUI front end
    - The other person is writing a data retrieval module
- Clearly, the front end GUI code must call the data retrieval module at some point
- Things are much happier if they agree on an interface ahead of time…
    - All methods, parameters and return types are predefined
    - The actual code is independent of the interface



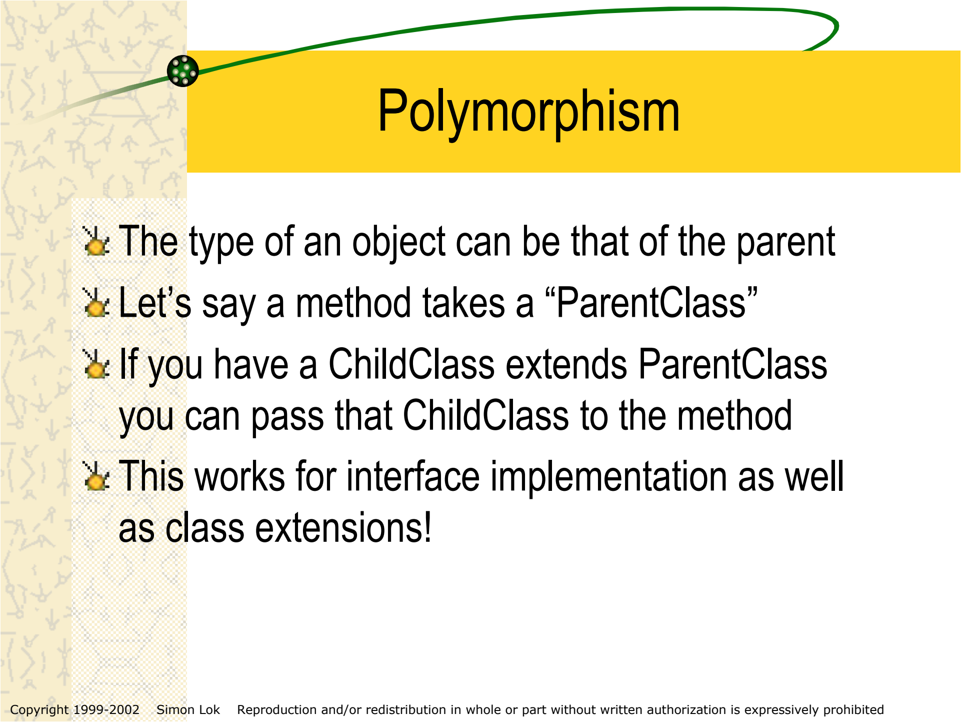
Interfaces in JAVA

- Suppose we need simple message exchanges:

```
public interface MesgXfer {
   public String recvMessage();
   public sendMessage(String);
}
```

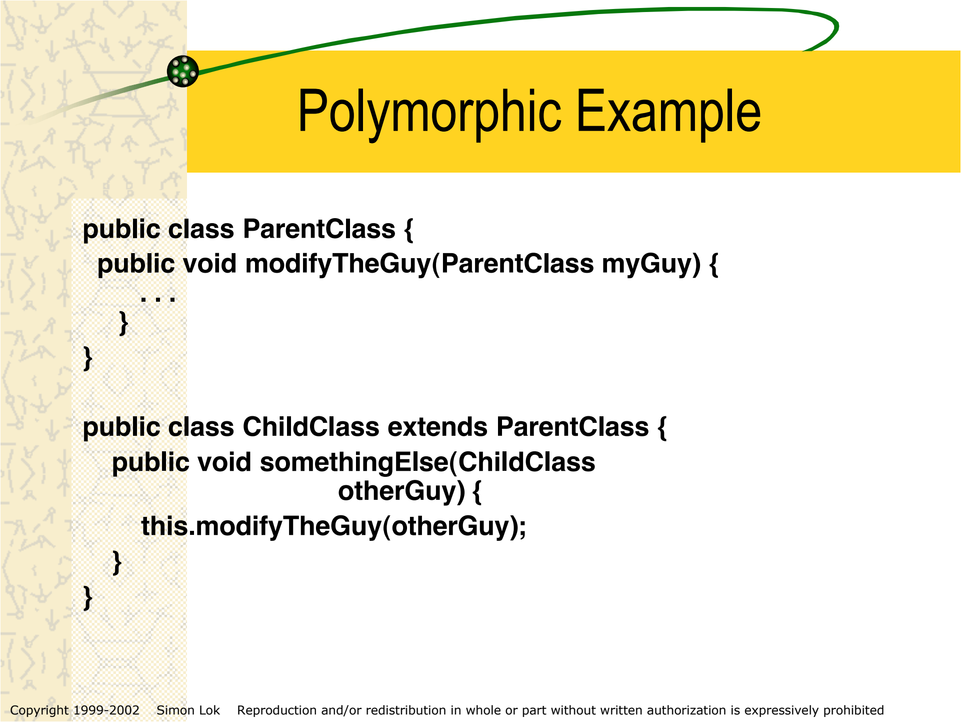
- A class would then implement the interface:

```
public class MyObject extends Object
                      implements MesgXfer {
   /* compiler requires definitions of the
      recvMessage and sendMessage methods
      in here */
}
```



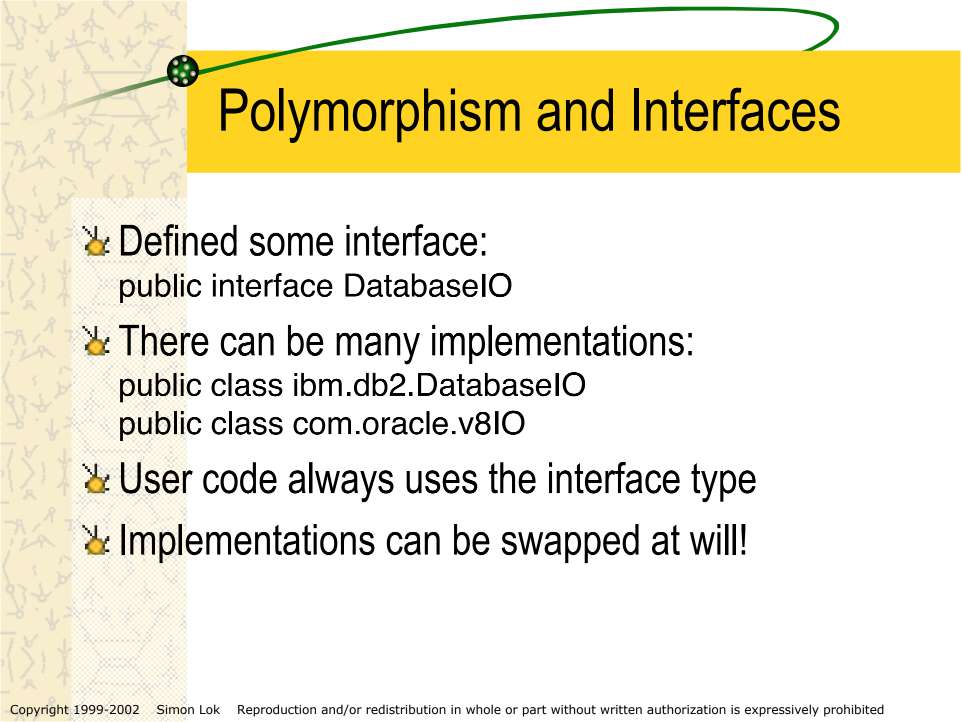
Polymorphism

- The type of an object can be that of the parent
- Let’s say a method takes a “ParentClass”
- If you have a ChildClass extends ParentClass you can pass that ChildClass to the method
- This works for interface implementation as well as class extensions!



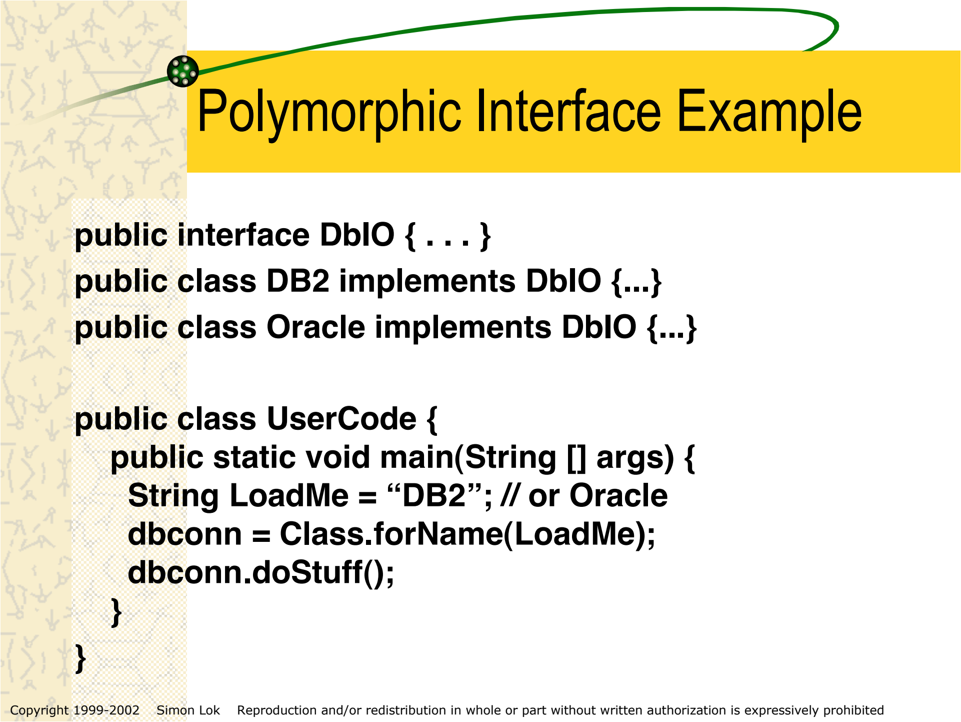
Polymorphic Example

```
public class ParentClass {
 public void modifyTheGuy(ParentClass myGuy) {
      . . .
   }
}

public class ChildClass extends ParentClass {
   public void somethingElse(ChildClass
                               otherGuy) {
      this.modifyTheGuy(otherGuy);
   }
}
```



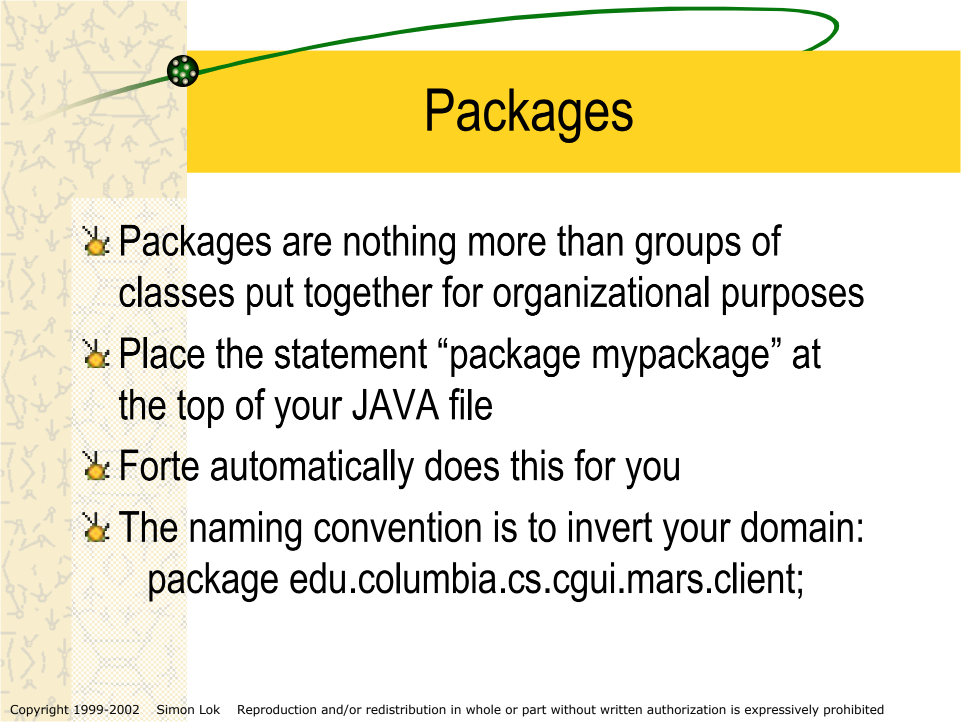
Polymorphism and Interfaces

- Defined some interface:
  public interface DatabaseIO
- There can be many implementations:
  public class ibm.db2.DatabaseIO
  public class com.oracle.v8IO
- User code always uses the interface type
- Implementations can be swapped at will!



Polymorphic Interface Example

```
public interface DbIO { . . . }
public class DB2 implements DbIO {...}
public class Oracle implements DbIO {...}

public class UserCode {
   public static void main(String [] args) {
     String LoadMe = “DB2”; // or Oracle
     dbconn = Class.forName(LoadMe);
     dbconn.doStuff();
   }
}
```



Packages

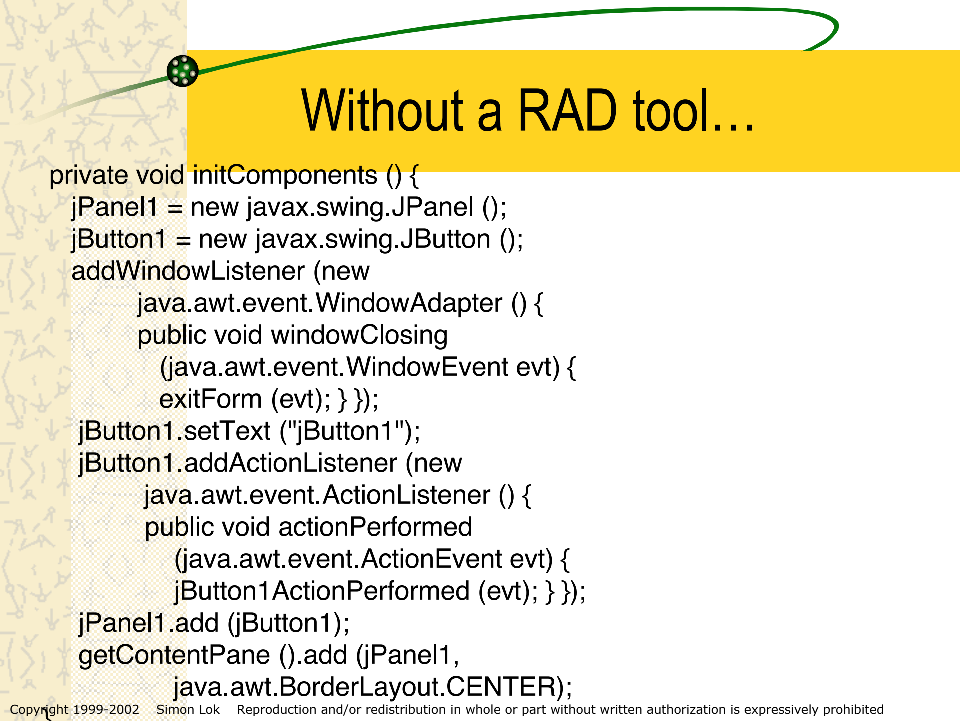
- Packages are nothing more than groups of classes put together for organizational purposes
- Place the statement “package mypackage” at the top of your JAVA file
- Forte automatically does this for you
- The naming convention is to invert your domain:
  package edu.columbia.cs.cgui.mars.client;

Event Driven Programming

- We have classes, but we're still instantiating the classes in a linear (imperative/procedural) way
  - This is okay for CLIs, but not for GUIs
  - For GUIs, we need multiple POEs
- GUI actions (like clicking a button) generate events that each trigger their own POE
- Multiple buttons results in multiple possible (and unpredictable) paths execution

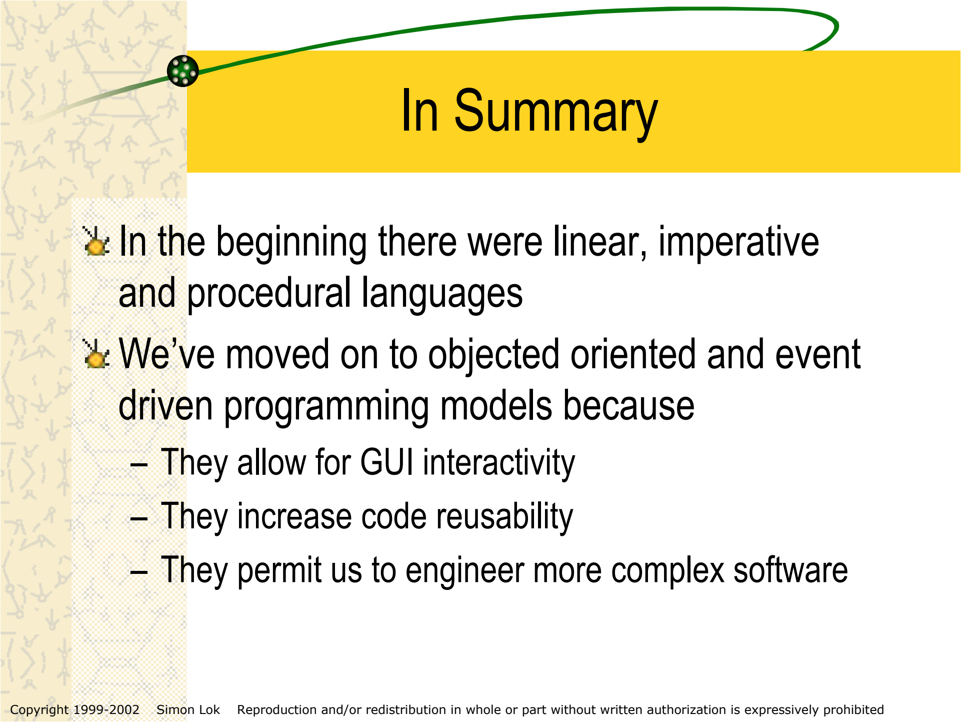
RAD tools are key!

- Using Forte, you can startup a project using the "Swing Forms" :: "JFrame" template and it will popup a "toplevel" window
- You can then drag and drop components (like buttons, scroll bars, etc…) onto the window
- Double clicking on a button brings you to the POE where execution will begin when the button is clicked



Without a RAD tool…

```
private void initComponents () {
   jPanel1 = new javax.swing.JPanel ();
   jButton1 = new javax.swing.JButton ();
   addWindowListener (new
            java.awt.event.WindowAdapter () {
            public void windowClosing
               (java.awt.event.WindowEvent evt) {
               exitForm (evt); } });
    jButton1.setText ("jButton1");
    jButton1.addActionListener (new
             java.awt.event.ActionListener () {
             public void actionPerformed
                 (java.awt.event.ActionEvent evt) {
                 jButton1ActionPerformed (evt); } });
    jPanel1.add (jButton1);
    getContentPane ().add (jPanel1,
                 java.awt.BorderLayout.CENTER);
}
```



In Summary

- In the beginning there were linear, imperative and procedural languages
- We've moved on to objected oriented and event driven programming models because
  - They allow for GUI interactivity
  - They increase code reusability
  - They permit us to engineer more complex software