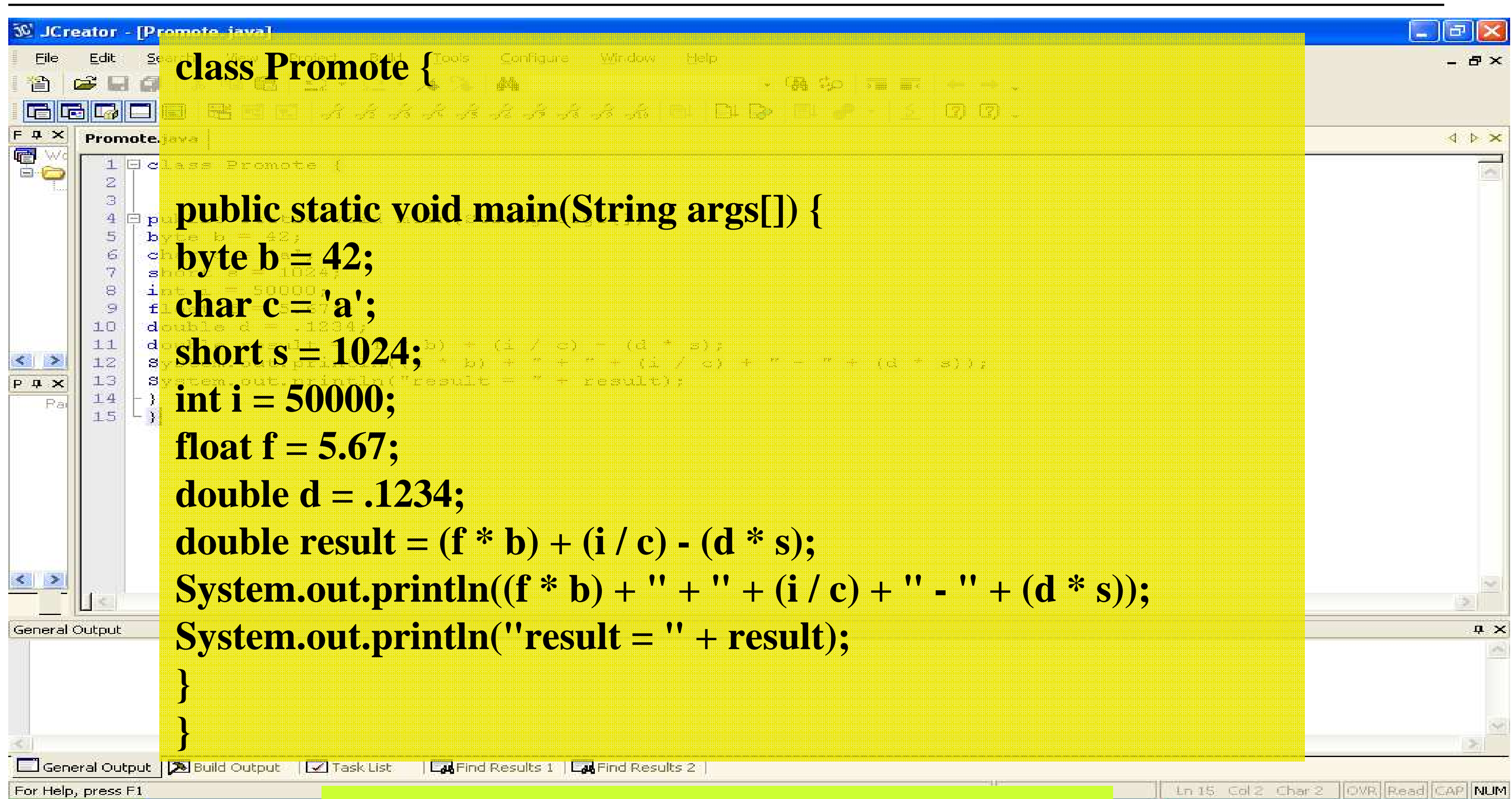
COMP455 - OBJECT ORIENTED PROGRAMMING

Dr. Constandinos X. Mavromoustakis

(4 Nov. 2009)

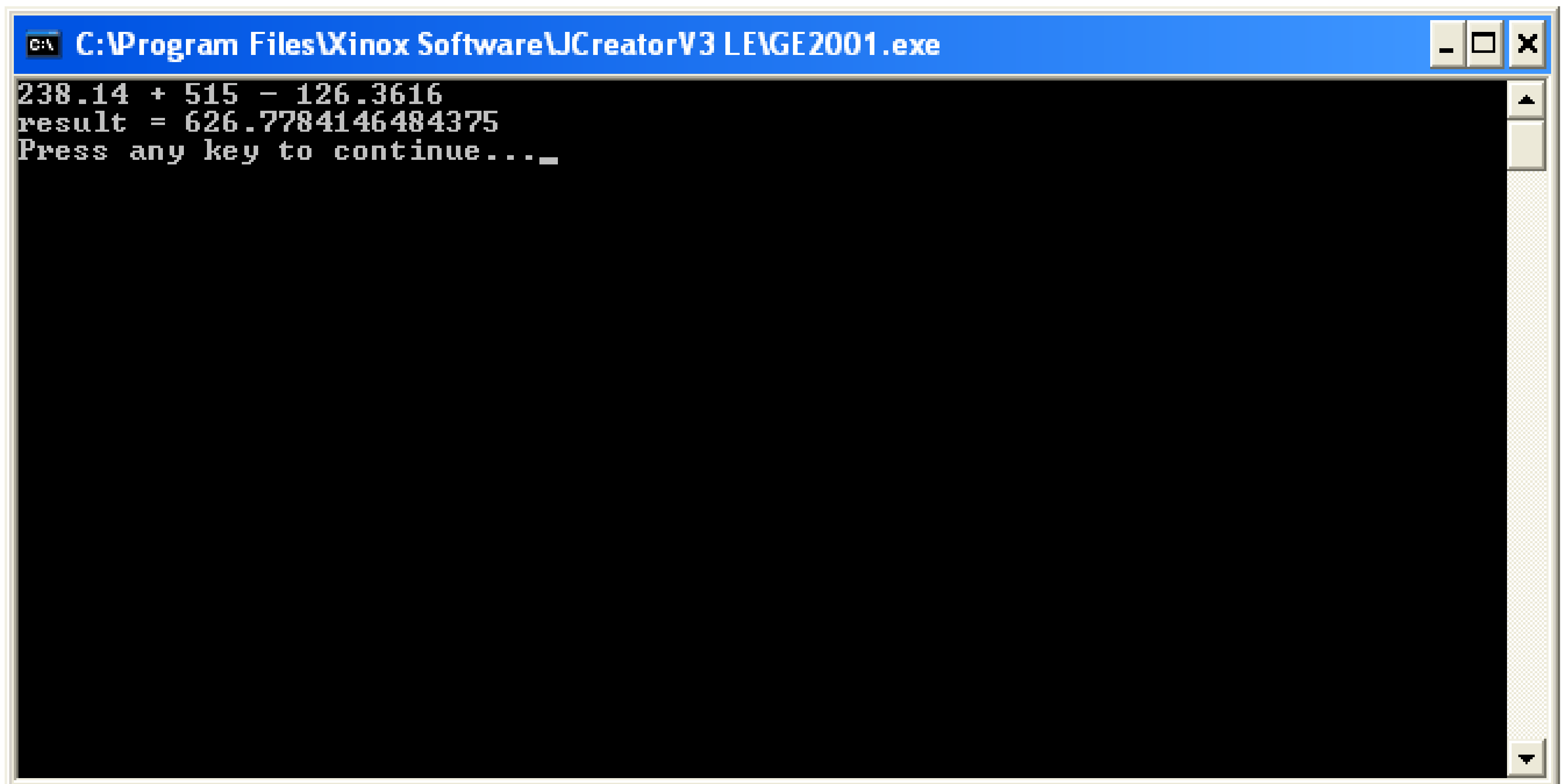
Arithmetical Example




```
class Promote {

public static void main(String args[]) {
byte b = 42;
char c = 'a';
short s = 1024;
int i = 50000;
float f = 5.67;
double d = .1234;
double result = (f * b) + (i / c) - (d * s);
System.out.println((f * b) + " + " + (i / c) + " - " + (d * s));
System.out.println("result = " + result);
}
}
```

What will be the output

Output



- control statements..and repetition statements
  - ALL the same with C/C++
  - Basic structure
  - Defer rarely on the usage BUT not on their syntax

if …else

- if …else

```
if ( studentGrade >= 60 )
System.out.println( "Passed" );

or

if ( grade >= 60 )
    System.out.println( "Passed" );
else
   System.out.println( "Failed" );
```

control statements..and repetition statements

while

- while repetition statement

```
int product = 3;

    while ( product <= 100 )
      product = 3 * product;
```

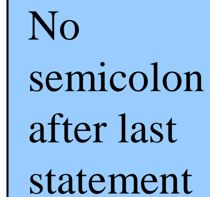
for Repetition Structure

- General format when using **for** loops

```
for ( initialization; LoopContinuationTest;
      increment )
   statement
```

- Example

```
for( int counter = 1; counter <= 10; counter++ )
```



`while` / for Repetition Structure

- `for` loops can usually be rewritten as `while` loops

```
initialization;
while ( loopContinuationTest){
   statement
   increment;
}
```

- Initialization and increment
  - For multiple variables, use comma-separated lists

```
for (int i = 0, j = 0;  j + i <= 10; j++, i++)
```

control statements..and repetition statements

- for Repetition Statement

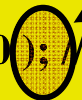
```
1  // ForCounter.java
 2  // Counter-controlled repetition with the for repetition
statement.
 3
 4  public class ForCounter
 5  {
 6     public static void main( String args[] )
 7     {
 8        // for statement header includes initialization,
 9        // loop-continuation condition and increment
10        for ( int counter = 1; counter <= 10; counter++ )
11
12
13        System.out.println(); // output a newline
14     } // end main
15  } // end class ForCounter
```

- Syntax:

```
do
{
        //statements here
}
while(logical expression is true);
```

do...while repetition statement

```
1  // DoWhileTest.java
 2  // do...while repetition statement.
 3
 4  public class DoWhileTest
 5  {
 6     public static void main( String args[] )
 7     {
 8        int counter = 1; // initialize counter
 9
10        do
11        {
12           System.out.printf( "%d  ", counter );
13           ++counter;
14        } while ( counter <= 10 ); // end do...while
15
16        System.out.println(); // outputs a newline
17     } // end main
18  } // end class DoWhileTest
```



Watch this out!

The **while** statement tests a condition at the ***start*** of the loop

The **do-while** statement tests a condition at the ***end*** of the loop

Compare them in your minds!

Exercise…with control statements..and repetition statement

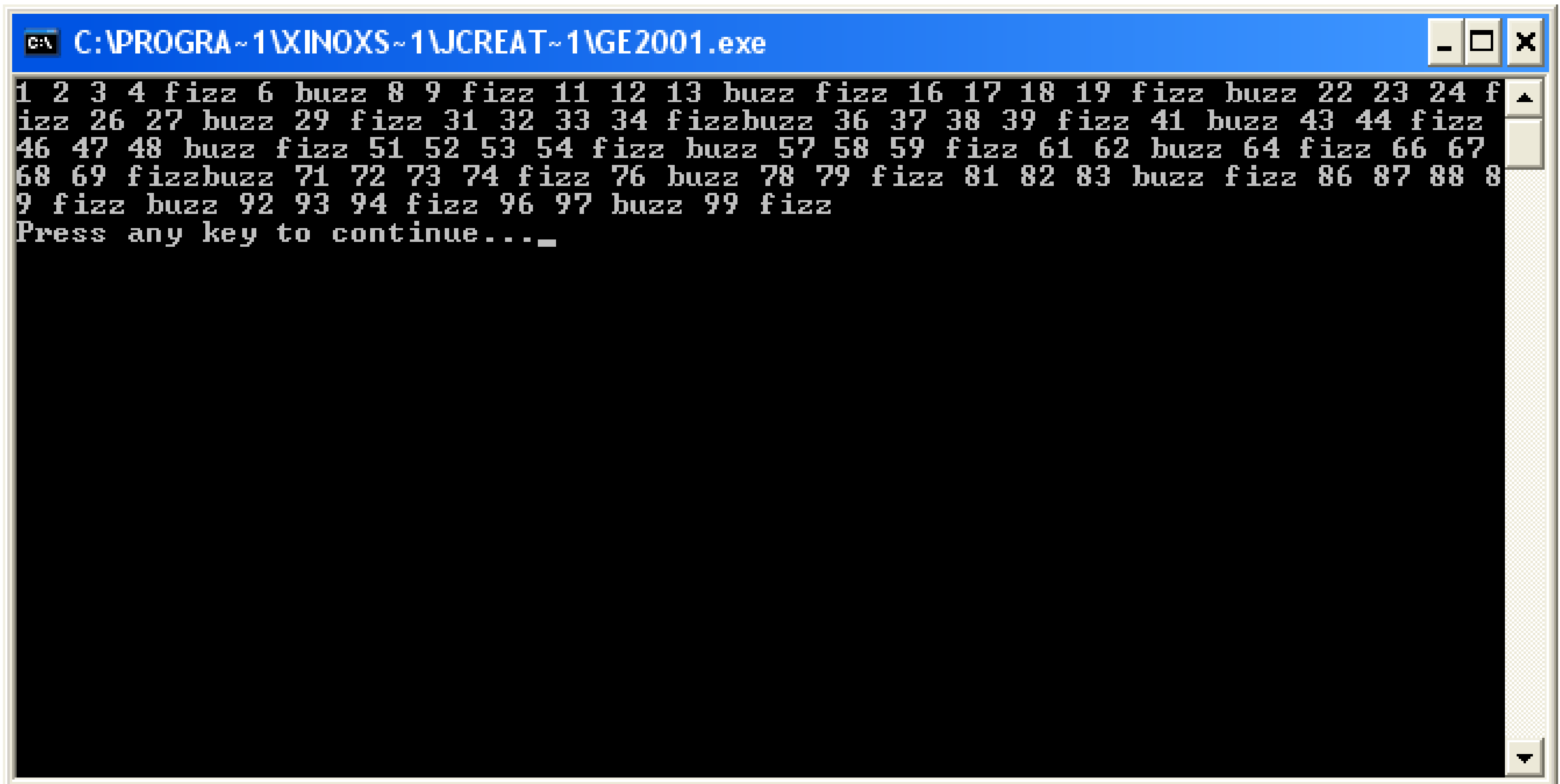
Write a complete Java program plays the game "Fizzbuzz". The program should count upto 100, replacing each multiple of 5 with the word "fizz", each multiple of 7 with the word "buzz", and each multiple of both with the word "fizzbuzz". The program should use the modulo operator (%) to determine if a number is divisible by another.

Write the code!

Exercise…with control statements..and repetition statement

```
public class FizzBuzz {                    // Everything in Java is a class
  public static void main(String[] args) { // Every program must have main()
    for (int i = 1; i <= 100; i++) {          // count from 1 to 100
      if (((i % 5) == 0) && ((i % 7) == 0)) // Is it a multiple of 5 & 7?
        System.out.print("fizzbuzz");
      else if ((i % 5) == 0)              // Is it a multiple of 5?
        System.out.print("fizz");
      else if ((i % 7) == 0)              // Is it a multiple of 7?
        System.out.print("buzz");
      else System.out.print(i);           // Not a multiple of 5 or 7
      System.out.print(" ");
    }
    System.out.println();
  }
}
```

Output

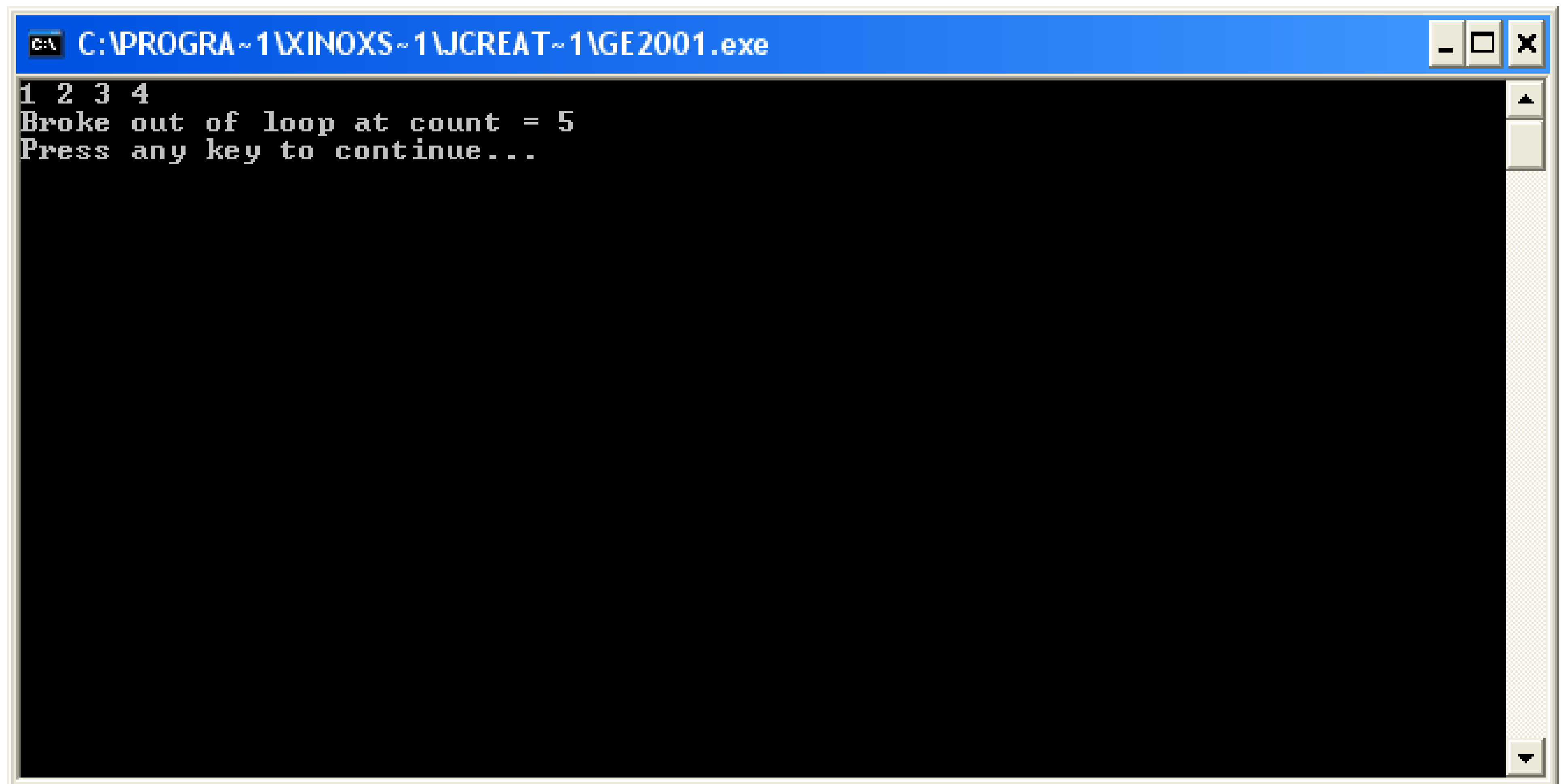


break statement exiting a for statement

(This item is displayed on page 201 in the print version)

```
1  // BreakTest.java
2  // break statement exiting a for statement.
3  public class BreakTest
4  {
5     public static void main( String args[] )
6     {
7        int count; // control variable also used after loop terminates
8
9        for ( count = 1; count <= 10; count++ ) // loop 10 times
10       {
11          if ( count == 5 ) // if count is 5,
12             break;         // terminate loop
13
14          System.out.printf( "%d ", count );
15       } // end for
16
17       System.out.printf( "\nBroke out of loop at count = %d\n", count );
18    } // end main
19 } // end class BreakTest
```

Output



- The switch statement is Java's multiway branch statement. It provides an easy way to dispatch execution to different parts of your code based on the value of an expression.
- It often provides a better alternative than a large series of if-else-if statements. Here is the general form of a switch statement:

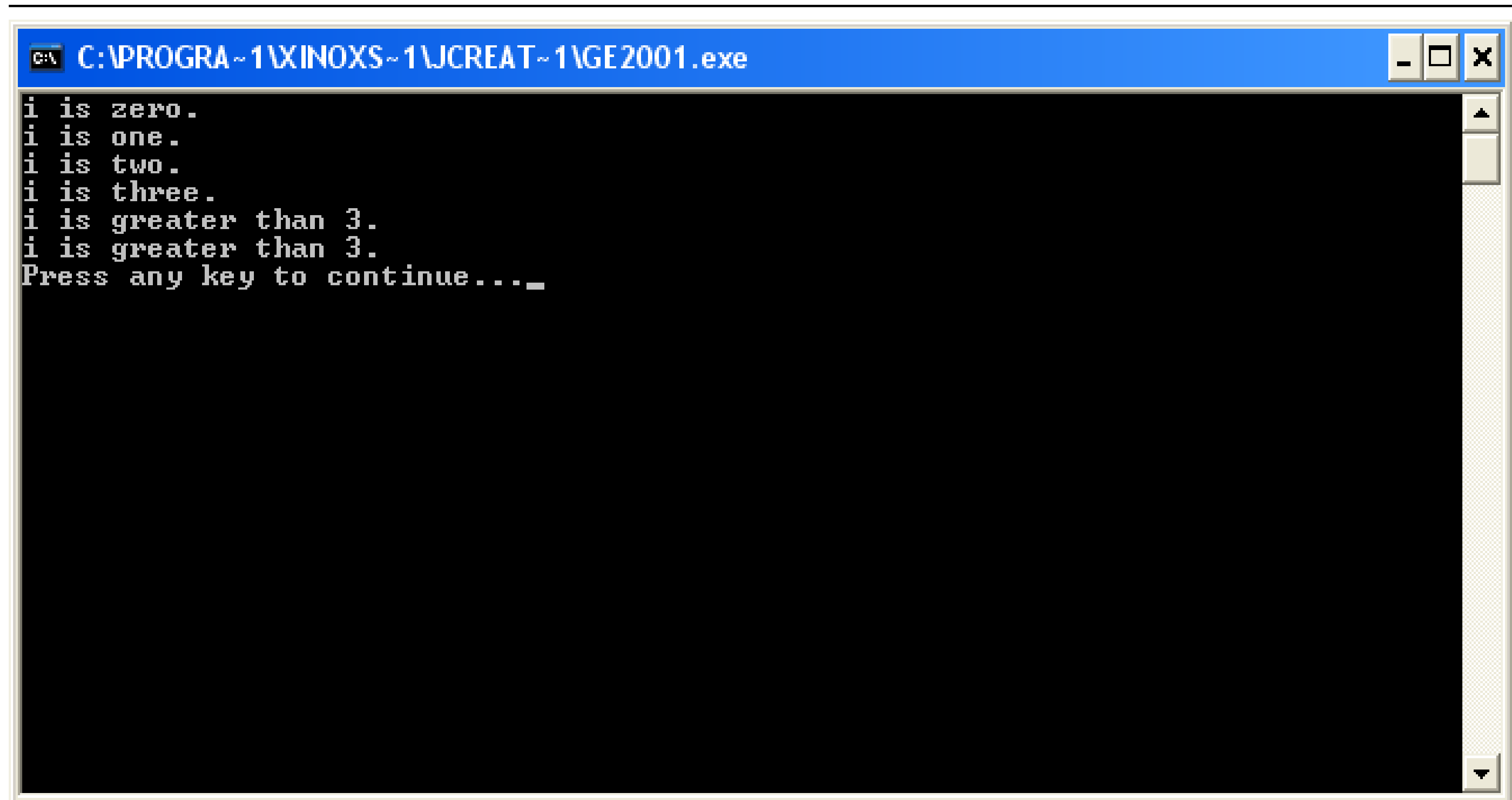
switch

switch (*expression*) {
case *value1*:
// statement sequence
break;
case *value2*:
// statement sequence
break;
.
.
.
case *valueN*:
// statement sequence
break;
default:
// default statement sequence
}

Example… class SampleSwitch

```
class SampleSwitch {
public static void main(String args[]) {
        for(int i=0; i<6; i++)
                switch(i) {
                case 0:
                    System.out.println("i is zero.");
                    break;
                case 1:
                    System.out.println("i is one.");
                break;
                case 2:
                     System.out.println("i is two.");
                break;
                case 3:
                     System.out.println("i is three.");
                break;
        default:
        System.out.println("i is greater than 3.");
        }
    }
}
```

Output



```
/**
 * This class is much like the FizzBuzz class, but uses a switch statement
 * instead of repeated if/else statements
 **/
```

```
public class FizzBuzz2 {
   public static void main(String[] args) {
      for(int i = 1; i <= 100; i++) { // count from 1 to 100
         switch(i % 35) {     // What's the remainder when divided by 35?
         case 0:              // For multiples of 35...
            System.out.print("fizzbuzz ");  // print "fizzbuzz".
            break;                // Don't forget this statement!
         case 5: case 10: case 15:    // If the remainder is any of these
         case 20: case 25: case 30:   // then the number is a multiple of 5
            System.out.print("fizz ");      // so print "fizz".
            break;
         case 7: case 14: case 21: case 28:  // For any multiple of 7...
            System.out.print("buzz ");      // print "buzz".
            break;
         default:                 // For any other number...
            System.out.print(i + " ");      // print the number.
            break;
         }
      }
      System.out.println();
   }
}
```

Tip

- You should RUN (type in and compile & RUN) every single program..
- It is the ***ONLY*** way to learn Java..
  - Even with other people's code..

Back to…

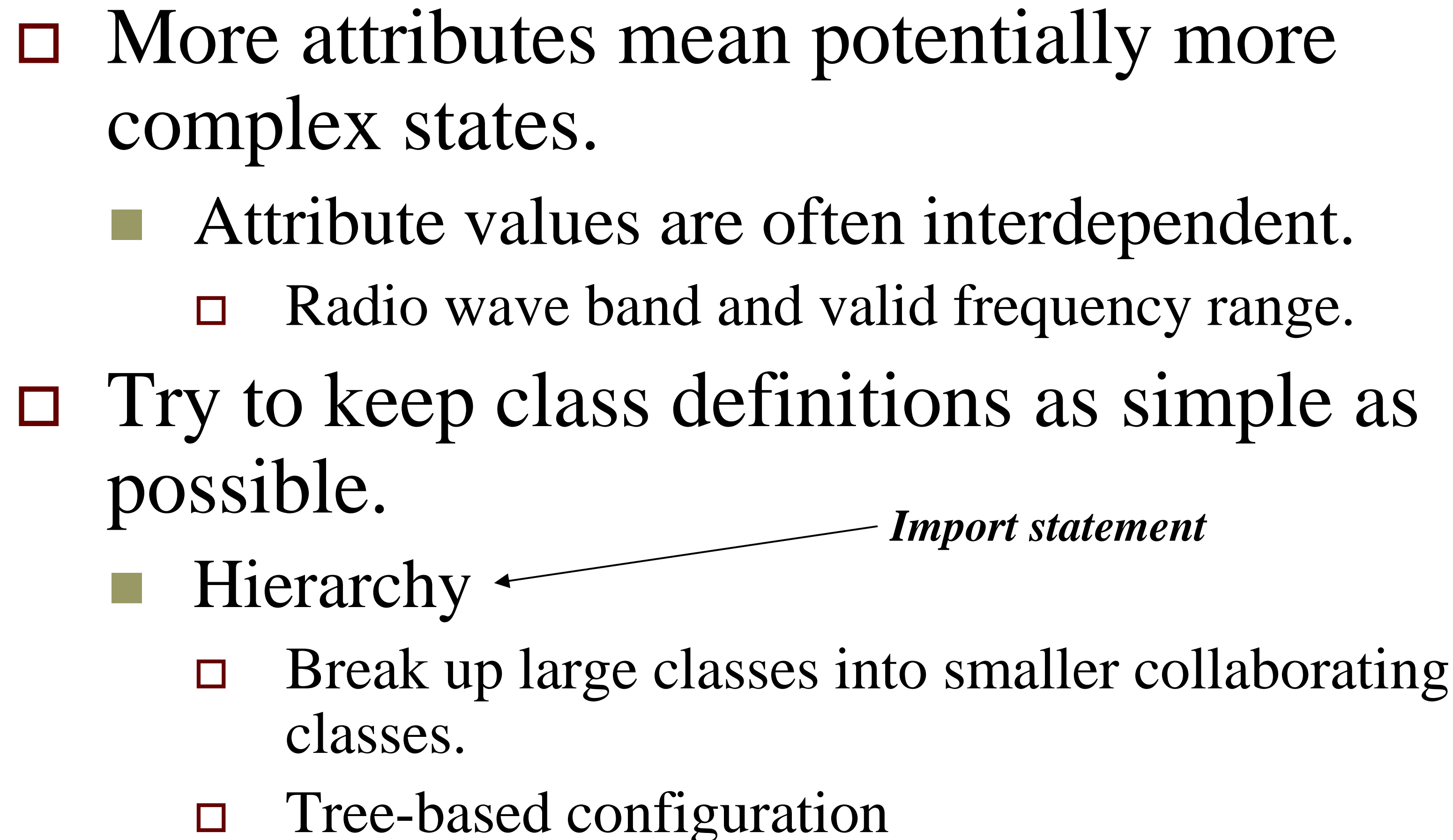
Object State and Complexity

Back to…
Object State and Complexity

- Objects maintain a set of attributes.
  - Port example (Position, course, speed)
  - ATM (Account name, balance, transactions)
- Current set of values is the object's state.
  - Simple state: on/off.
  - Complex state: process controller (under certain values).
- Attributes defined by *fields* in a class.

```
    // Define a class called ClassName.
class ClassName {
    // Method definitions go here.
    ...
    // Field definitions go here.
    ...
}
```

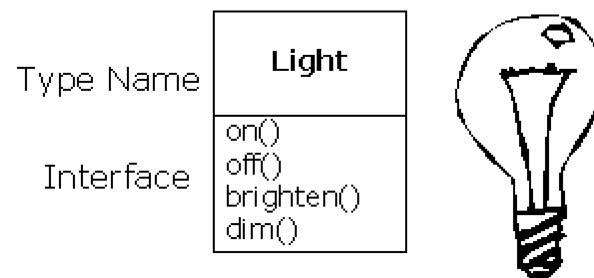
State and Complexity (cont.)



- More attributes mean potentially more complex states.
  - Attribute values are often interdependent.
    - Radio wave band and valid frequency range.
- Try to keep class definitions as simple as possible.
  - Hierarchy ← *Import statement*
    - Break up large classes into smaller collaborating classes.
    - Tree-based configuration

Let’s model in an object way

- But how do you get an object to do useful work for you? There must be a way to make a request of the object so that it will do something, such as complete a transaction, draw something on the screen, or turn on a switch.



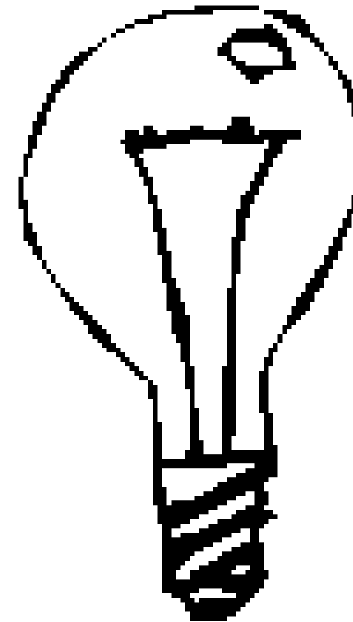
simple example might be a representation of a light bulb:

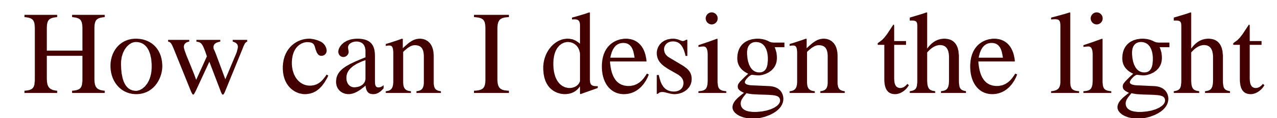
Type Name

Light

Interface

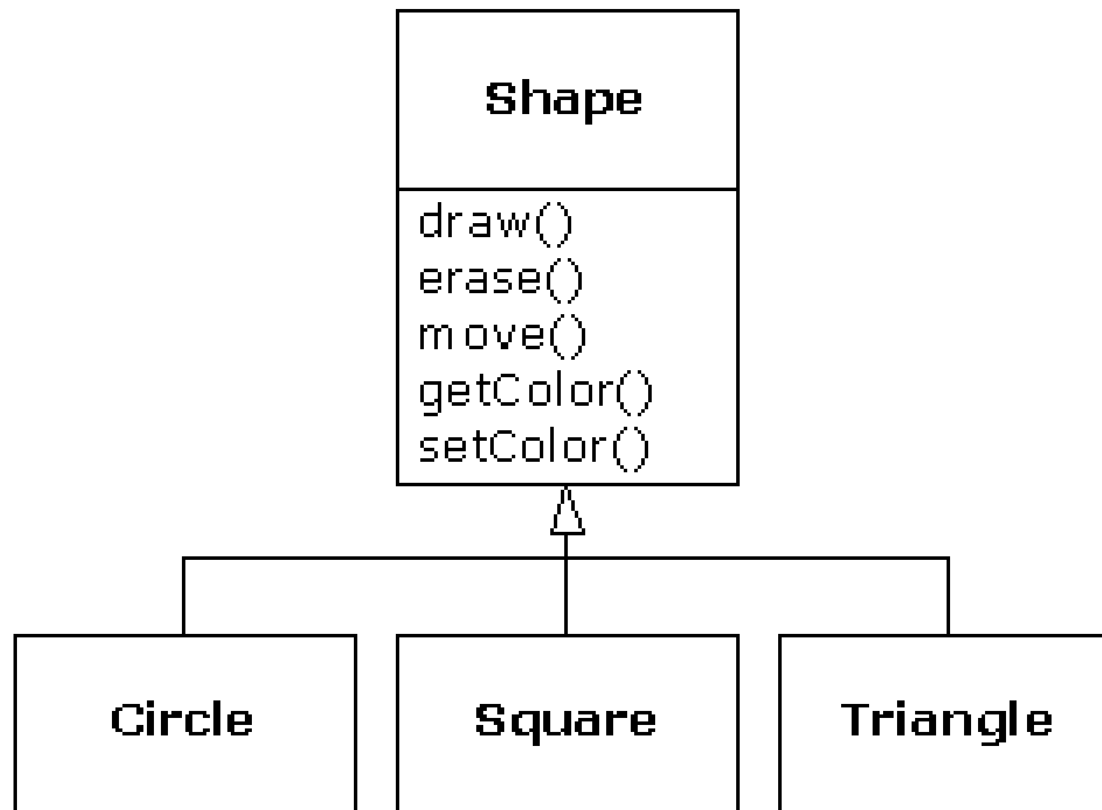
on()
off()
brighten()
dim()



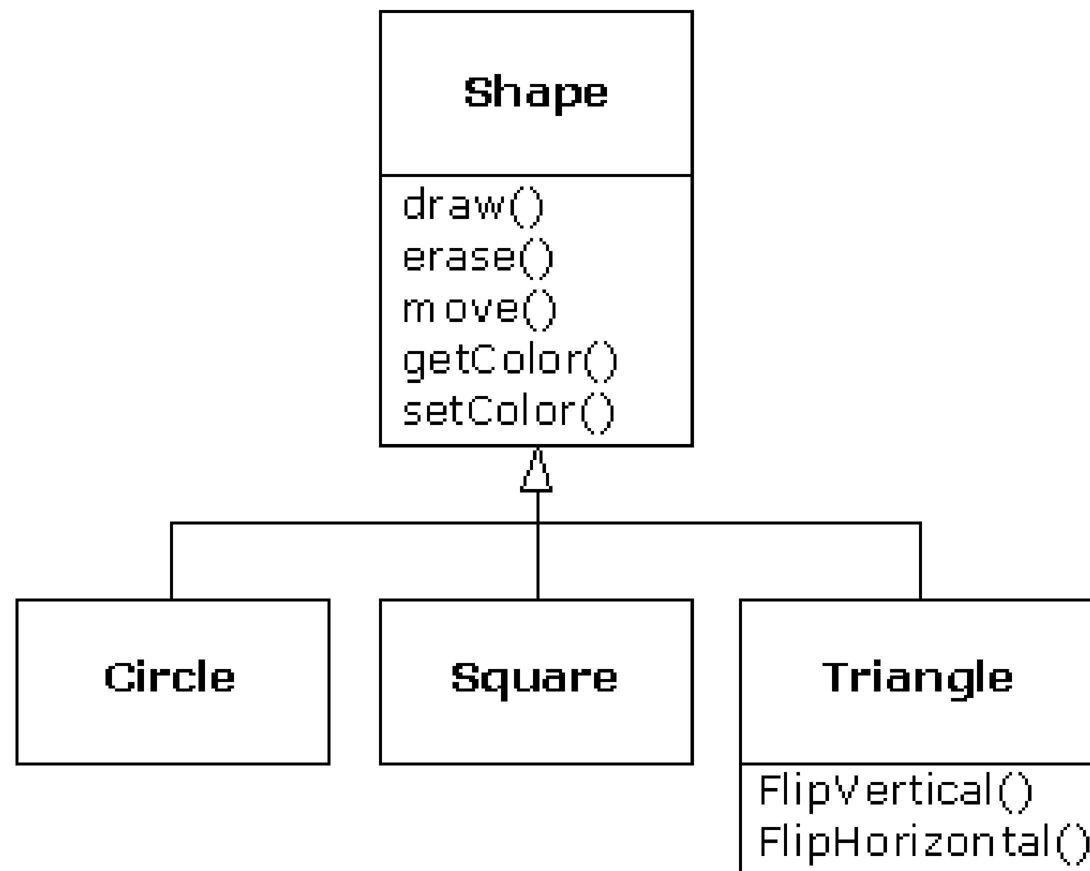



```
Light lt = new Light();
lt.on();
```

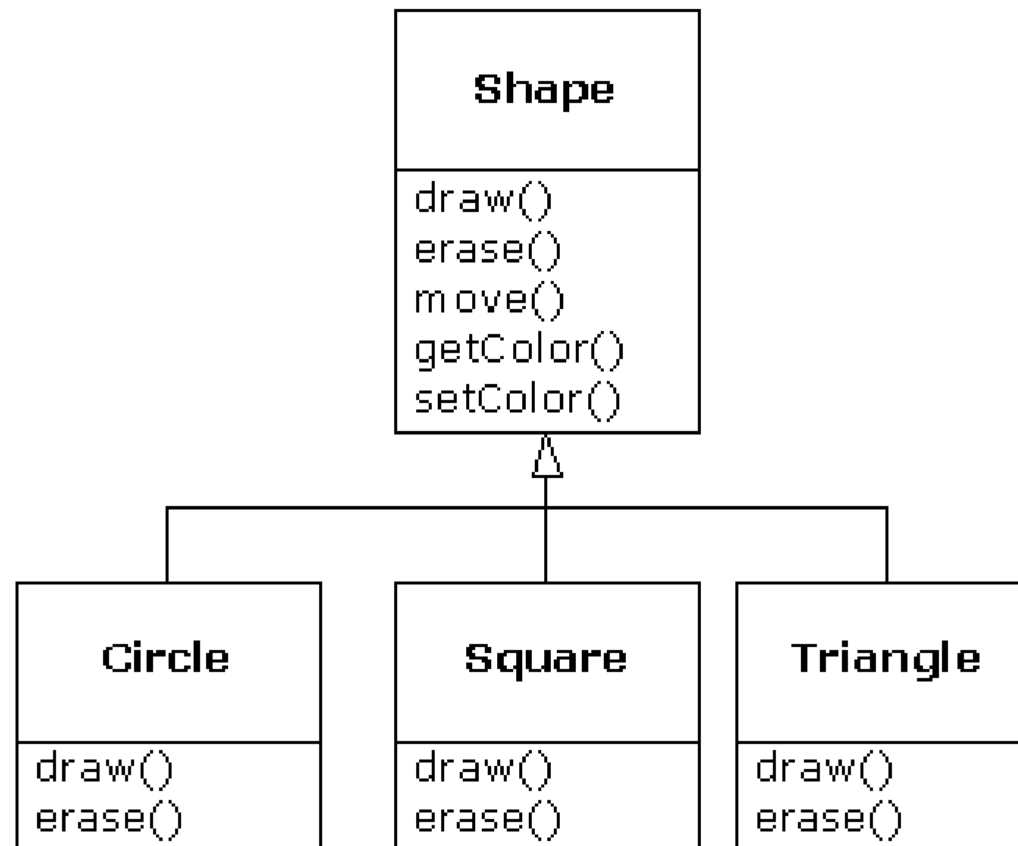
Let's model the class Shape



Class Shape



Class Shape



- Methods and fields may be arranged in any order.
  - We tend to list methods before fields.
- Methods describe the behavior of instances.
- Fields are variables.
  - Instance variables (most common).
  - Class variables.

- ***Each instance*** (object) has its own copy of the instance variables.
  - `speed` attribute of Ship.
  - Alteration in one instance does not affect other's.
- ***All instances*** share a class variable.
  - Altered via one instance, changed in all.

NoteMain1

Example:
Using the SimpleNote Class

```
class NoteMain1 {
    public static void main(String[] args){
        // Create two new SimpleNote objects
        // ready for messages.
        SimpleNote milk = new SimpleNote(),
                   swimming = new SimpleNote();

        // ‘Write’ a message on one note.
        milk.setMessage("We need more milk..");
        // Check the message.
        System.out.println(milk.getMessage());
    }
}
```

Defining Attributes (blackboard paradigm)

```
    // A class to represent a sticky note, that
    // could be stuck to a wall, door,
    // refrigerator, etc. This simplified version
    // just stores the message and provides access
    // to it.

class SimpleNote {
    // Omit the methods for simplicity.
    ...
    // The variable used to store the text.
    // The note is blank to start with.
    private String message = "";
}
```

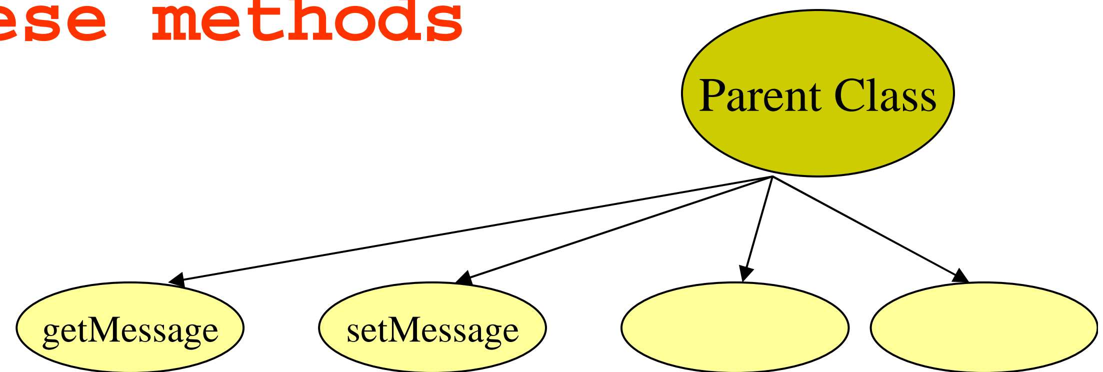
Methods of the *SimpleNote* Class

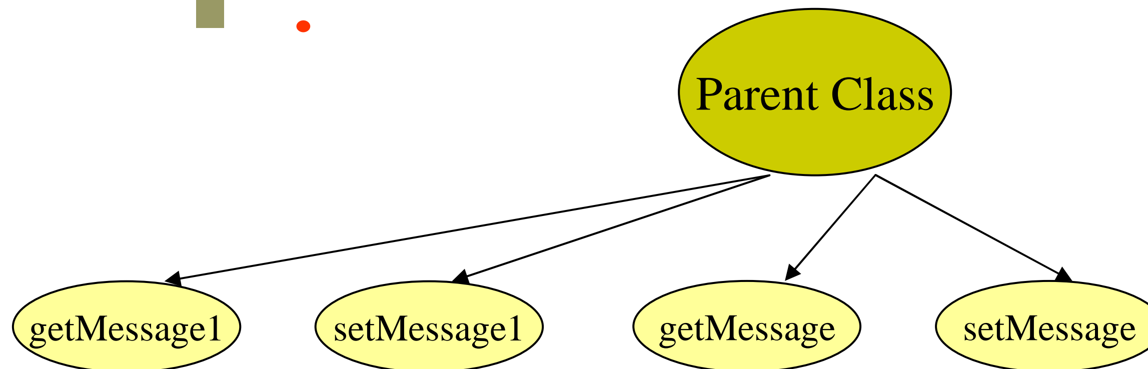
```
class SimpleNote {
        // Return what the message is.
    public String getMessage() {
        return message;
    }

        // Change the message.
    public void setMessage(String m) {
        message = m;
    }

    // The variable used to store the text.
    // The note is blank to start with.
    private String message = "";
}
```

- `getMessage` is an *accessor* method.
  - It returns the value of its associated attribute.
  - It has a ***return type (returns a value)***, matching the type of the attribute.
- `setMessage` is a *mutator* method.
  - It changes the value of its associated attribute.
  - It has a *formal argument*, matching the type of the attribute.

Common Types of Methods

- We create these methods
  - getMessage
  - setMessage
  - .
  - .
  - .





  - We also can change these method’s operations by adding to parent’s Class the modification

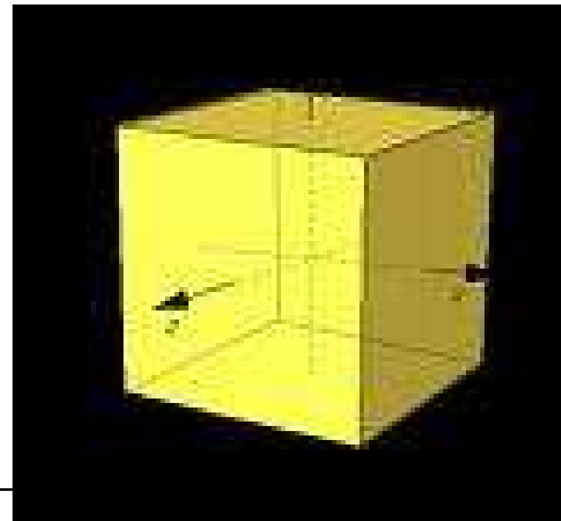
- Accessor methods have a return type:
  - `public` **`String`** `getMessage()`
- Methods with a return type have a return statement in their bodies:
  - **`return`** `message;`
- The value following `return` is passed back to the method caller.

Primitive Types

- Major primitive types:
  - `boolean, char, double, int`
- Minor primitive types:
  - `byte, float, long, short`
- Primitive-types have no methods
- `String` is not a primitive type

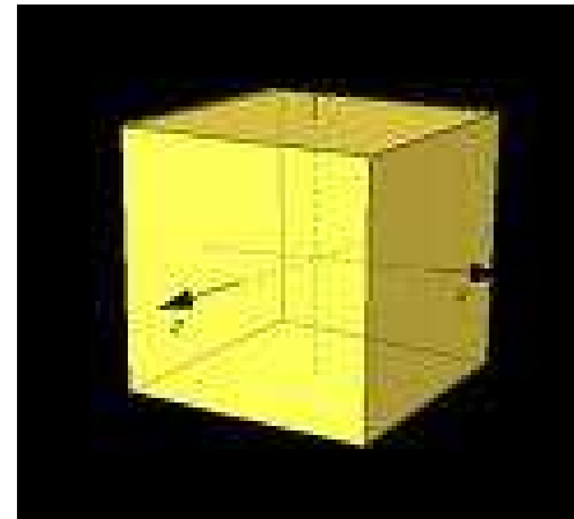
Create a class…

Implementing a “Box”



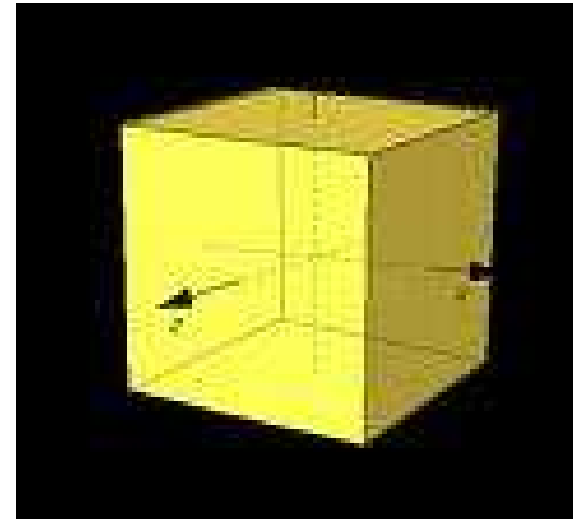
Create a class…
Implementing a “Box”

```
class Box {
        double width;
        double height;
        double depth;
// display volume of a box



void volume() {
        System.out.print("Volume is ");
        System.out.println(width * height * depth);

        }
}
```

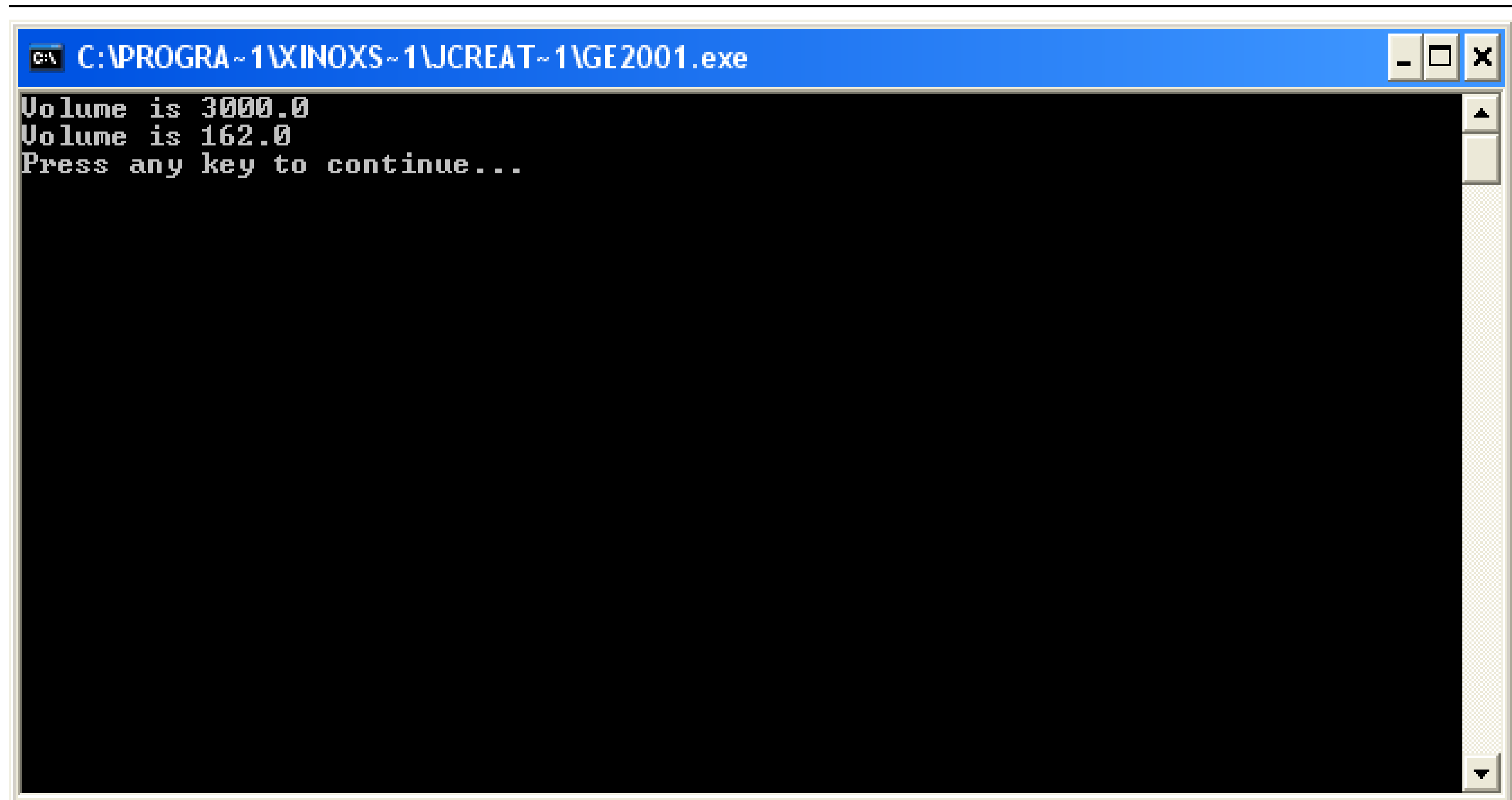


Implementing a "Box"
Re-creating the class Box/reinstantiate

```
class BoxDemo3 {
        public static void main(String args[]) {
                Box mybox1 = new Box();
                Box mybox2 = new Box();
                // assign values to mybox1's instance variables
                mybox1.width = 10;
                mybox1.height = 20;
                mybox1.depth = 15;
                /* assign different values to mybox2's
                instance variables */
                mybox2.width = 3;
                mybox2.height = 6;
                mybox2.depth = 9;
                // display volume of first box
                mybox1.volume();
                // display volume of second box
                mybox2.volume();
        }
```

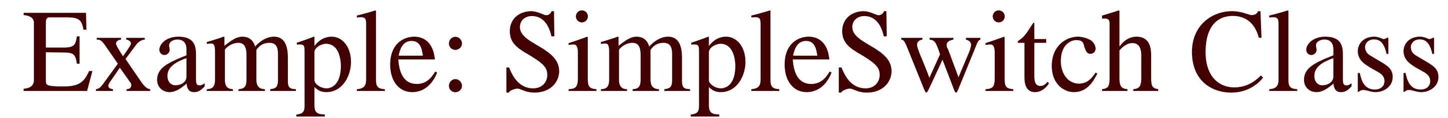


Output after compiling…



Another Example

SimpleSwitch Class



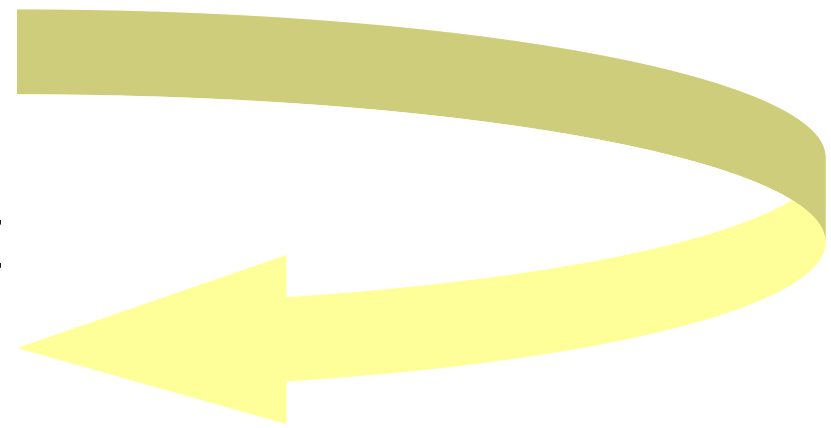

```
    // A class with a boolean attribute.
    // A simple on/off switch.
class SimpleSwitch {
    // Method definitions go here.
    ...

    // Whether the switch is on or off.
    // The switch is off by default.
    private boolean on = false;
}
```

The Public Interface of a Class

```
class SimpleSwitch {
        // Turn the switch on.
    public void switchOn() {
        ...
    }

    public void switchOff() {
        ...
    }

    public boolean isTheSwitchOn() {
    }


   ...code of the SimpleSwitch
}
```

Bridging Methods

```
class SimpleSwitch {
        // Turn the switch on.
    public void switchOn(){
        setOn(true);
    }
    ...
    private boolean getOn(){
        return on;
    }

    private void setOn(boolean o){
        on = o;
    }

    private boolean on = false;
}
```



Our checkpoints up to now!!

- The public interface of a class should reflect real-world usage.
  - e.g., `switchOn, switchOff`.
- Extra methods required to bridge between public interface and private implementation.
  - Accessors and mutators may be sufficient.
- An object may invoke its own methods.

Accessor Methods

- Potentially provide outside access to a private attribute
  - public accessor available from all other classes
  - public accessor harmless for primitive type attributes
  - private accessor limits availability to other objects of the same class
    - Keeps implementation details hidden

Mutator Methods

- Necessary for external modification of a private attribute
- More care required with visibility
  - Public mutator potentially allows objects of *all* other classes to alter the state
  - Method body could make checks to ensure validity of mutation

```
class C {
    ...
    public type getField(){
        return field;
    }

    private void setField(type f){
        field = f;
    }

    private type field;
}
```

Constructors - Initializing State

- An object should start its existence in a valid and consistent state (initial state)
- Constructors are used to achieve this aim
- A constructor is never used to receive messages from other objects
  - Called automatically as part of an object's creation
- We can have ***any number of constructors***!

Constructor initializes..

- A constructor initializes an object immediately upon creation. It has the same name as the class in which it resides and is syntactically similar to a method.
- Once defined, the constructor is automatically called immediately after the object is created, before the ***new*** operator completes.
- Constructors look a little strange because they have no return type, not even void. This is because the implicit return type of a class' constructor is the class type itself. It is the constructor's job to initialize the internal state of an object so that the code creating an instance will have a fully initialized, usable object immediately.

A No-Arg Constructor

```
class SimpleController {
    public SimpleController() {
        setTemperature(293.0);
    }
    ...
    public void setTemperature(double t){
        temperature = t;
    }

    private double temperature;
}
```

Constructors with Arguments

```
class Controller {
    public Controller(){
        double defaultT = 293.0;
        setTemperature(defaultT);
    }

    public Controller(double initialT){
        setTemperature(initialT);
    }
    ...
}
```

Creating Controllers

```
class ControllerMain3 {
    public static void main(String[] args){
        // Create controllers using the two
        // different constructors.
        Controller control1 = new Controller(),
                   control2 = new Controller(275.0);
        ...
    }
}
```

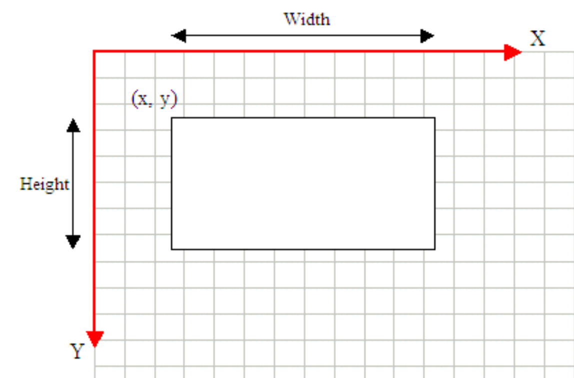
Identifier Usage

- How to identify in Java code the usage of each method..
  - Identifiers used, so far, for:
    - Names of classes
    - Formal argument names
    - Method names
    - Constructor names
    - Method variable names
    - Field names

The meaning of an identifier depends upon the context in which it is used

We want to create a class
Defining a Rect(angle) with the associated calculations/mechanisms that a Rect can perform.

What we should first do????
Define the 4 points…

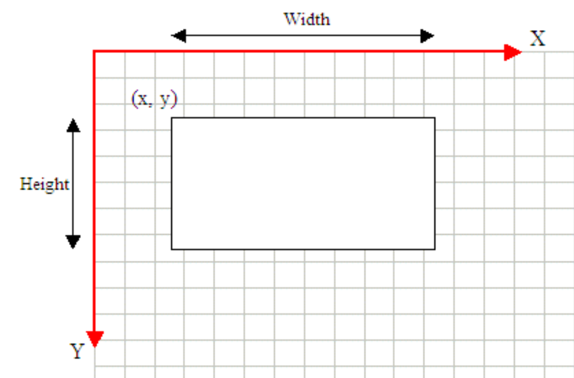


- We want to create a class
  - Defining a Rect(angle) with the associated calculations/mechanisms that a Rect can perform.
- What we should first do????
  - Define the 4 points…

class Rect

```
public class Rect {
   // These are the data fields of the class
   public int x1, y1, x2, y2;

   /**
    * The is the main constructor for the class.  It simply uses its arguments
    * to initialize each of the fields of the new object.  Note that it has
    * the same name as the class, and that it has no return value declared in
    * its signature.
    **/
   public Rect(int x1, int y1, int x2, int y2) {
      this.x1 = x1;
      this.y1 = y1;
      this.x2 = x2;
      this.y2 = y2;
   }
```

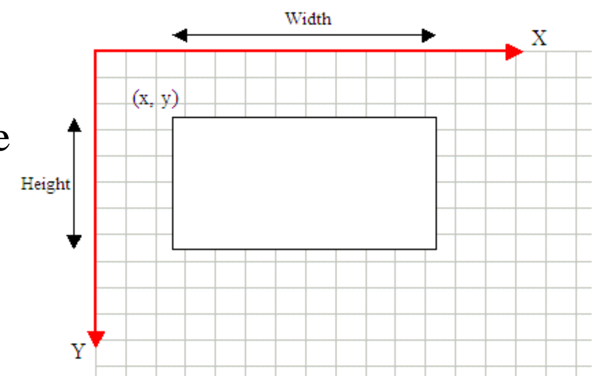
this refers to what?????



class Rect

```
public class Rect {
   // These are the data fields of the class
   public int x1, y1, x2, y2;

   /**
    * The is the main constructor for the class.  It simply uses its arguments
    * to initialize each of the fields of the new object.  Note that it has
    * the same name as the class, and that it has no return value declared in
    * its signature.
    **/
   public Rect(int x1, int y1, int x2, int y2) {
      this.x1 = x1;
      this.y1 = y1;
      this.x2 = x2;
      this.y2 = y2;
   }
  /*** This is another constructor.  It defines itself in terms of the above
      **/
     public Rect(int width, int height)
                 {  this(0, 0, width, height);
  }
```

Let's add another constructor



Let’s add some more constructors for our Rect calculations

```
/** This is yet another constructor. */
  public Rect() { this(0, 0, 0, 0); }

  /** Move the rectangle by the specified amounts */
  public void move(int deltax, int deltay) {
     x1 += deltax; x2 += deltax;
     y1 += deltay; y2 += deltay;
  }
  /** Test whether the specified point is inside the rectangle */
  public boolean isInside(int x, int y) {
     return ((x >= x1)&& (x <= x2)&& (y >= y1)&& (y <= y2));
  }
/**
   * This is a method of our superclass, Object.  We override it so that
   * Rect objects can be meaningfully converted to strings, can be
   * concatenated to strings with the + operator, and can be passed to
   * methods like System.out.println()
   **/
  public String toString() {
     return "[" + x1 + "," + y1 + "; " + x2 + "," + y2 + "]";
  }
}
```

class Averager

(class to compute the running average of numbers passed to it)

```
public class Averager {
   // Private fields to hold the current state.
   private int n = 0;
   private double sum = 0.0, sumOfSquares = 0.0;

   /**
    * This method adds a new datum into the average.
    **/
   public void addDatum(double x) {
      n++;
      sum += x;
      sumOfSquares += x * x;
   }

   /** This method returns the average of all numbers passed to addDatum() */
   public double getAverage() { return sum / n; }

   /** This method returns the standard deviation of the data */
   public double getStandardDeviation() {
        return Math.sqrt(((sumOfSquares - sum*sum/n)/n));
   }

   /** This method returns the number of numbers passed to addDatum() */
   public double getNum() { return n; }

   /** This method returns the sum of all numbers passed to addDatum() */
   public double getSum() { return sum; }
```

class Averager

```
/** This method returns the sum of the squares of all numbers. */
   public double getSumOfSquares() { return sumOfSquares; }

   /** This method resets the Averager object to begin from scratch */
   public void reset() { n = 0; sum = 0.0; sumOfSquares = 0.0; }

   /**
    * This nested class is a simple test program we can use to check that
    * our code works okay.
    **/
   public static class Test {
      public static void main(String args[]) {
         Averager a = new Averager();
         for(int i = 1; i <= 100; i++) a.addDatum(i);
         System.out.println("Average: " + a.getAverage());
         System.out.println("Standard Deviation: " +
                            a.getStandardDeviation());
         System.out.println("N: " + a.getNum());
         System.out.println("Sum: " + a.getSum());
         System.out.println("Sum of squares: " + a.getSumOfSquares());
      }
   }
}
```

- Class definitions capture the behavior and attributes of typical objects
- Behavior modeled by method definitions
- Attributes modeled by field definitions
  - Privacy helps to encapsulate object state
- Accessor and mutator methods provide access to object state

Our checkpoints up to now!! (cont.)

- Accessor and mutator methods can be written to follow a common pattern
- Object initialization is provided through constructors
  - Constructors may take arguments (or not)
- The meaning of an identifier depends upon the context in which it is used

Self-review exercises

Chapter 3 Intro to classes/ Deitel

Chapters 4,5 Control Statements (to refresh these concepts)/ Deitel

And their review questions