

Class 5: Classes and Objects

**1.00/1.001 - Introduction to
Computation and Problem Solving**

Fall 2005

Objects

- **Objects are ‘things’**
 - Recall the description of libraries, books, paperback books from Session 1
- **We decompose programming problems into a set of objects that are an intuitive representation of the situation. Examples:**
 - **A building**
 - List the objects
 - Each of these may contain other objects. List them
 - **Data packets going through a router**
 - List the objects
 - Each of these may contain other objects. List them

Objects

- **A building**
 - **Objects:** Beam, Slab, Wall, Window, Pipe, AirVent, WaterBoiler,
 - Window contains GlassPane, Sash,
 - Members such as GlassPane have attributes such as coating, height, width
- **Data packets going through a router**
 - **Objects:** Network Link In and Link Out, Router. Each of these may contain other objects:
 - Network link: contains data packets
 - Router: contains packet queues, buffers, processor

Bank Example

- **Customer:**
 - **Data fields:**
 - List them
 - **Methods or behaviors:**
 - List them
- **Account:**
 - **Data fields:**
 - List them
 - **Methods:**
 - List them
- **Transactions**
 - **Fields:**
 - List them
 - **Methods:**
 - List them

Bank Example

- **Customer:**
 - **Data fields:**
 - name, address, occupation, dateOfBirth, ...
 - **Methods or behaviors:**
 - Create new, changeAddress, changeName, delete
- **Account:**
 - **Data fields:**
 - Customer, balance, type, transactions
 - **Methods:**
 - Create, close, checkBalance, reportTransactions
- **Transactions**
 - **Fields:**
 - Deposit, Withdrawl, WireTransfer, payCheck, ATMWithdrawl
 - **Methods:**
 - Create, executeTransaction,

Bird Example

```
public class Head( ) { ..... }

public class Body( ) { ..... }

public class Wing( ) { .....
    public void changeColor( ) { ..... } }

public class Bird( ) { .....
    Head birdHead= new Head( );
    Wing leftWing= new Wing( );
    Wing rightWing= new Wing( );
    public Wing getWing( ) { ..... }
    public void fly( ) { ..... }
    public void changeStatus( ) { ..... } }
public class Scene( ) { .....
    Bird bird= new Bird( );
    Wing wing= bird.getWing( );
    public void changeWingColor( ) { wing.changeColor( ); }
    public void changeFlyStatus( ) { bird.changeStatus( ); }
}
```

Modeling Objects

- **Modeling objects (choosing the right problem representation) is like modeling in general**
 - There is generally no single ‘right’ answer (even in our problem sets!)
 - There are standard patterns/paradigms that have been found to be flexible, correct, efficient, etc.
 - We will introduce you to ‘software patterns’
 - There are many standard objects in Java
 - You can build your own library of objects in Java that you can then use in future programs

Classes

- **A class is a pattern or template from which objects are made**
 - You may have many birds in a simulation
 - One bird class (or more if there’s more than one type of bird)
 - Many bird objects (actual instances of birds)
 - Simulation
 - Another example:
 - JOptionPane is a class in the Swing package. A program may have many dialog boxes, each of which is an object of class JOptionPane

Class Definition

- **Classes contain:**
 - **Data (members, fields)**
 - primitive data types, like int or double (e.g. bird weight)
 - References to other Objects (e.g. bird beak)
 - **Methods (also called functions or procedures)**
 - Actions that an object can execute (e.g. bird flies/moves)

Class Definitions

- **Classes come from:**
 - Java class libraries: JOptionPanel, Vector, Array, etc. There are several thousand classes (see Javadoc on your computer)
 - Class libraries from other sources: VideoPlayers, 3D graphics renderers, etc.
 - Classes that you write yourself
- **Classes are usually the “nouns” in a problem statement (e.g. bird)**
- **Methods are usually the “verbs” (e.g. flies)**

Building Classes

- **Classes “hide” their implementations details from the user (another programmer using prewritten class):**
 - Their data is not generally accessed directly, and the details of how their methods work do not need to be known to ‘outside’ objects or programs.
 - Data is almost always private. This is a Java keyword denoting “can’t be accessed from methods outside the class”.

Building Classes

- **Objects are typically used by calling their methods.**
 - The outside user knows what methods the object has, and what results they return. Period. (Usually.)
 - The details of how their methods are written are not known to ‘outsiders’
 - **Methods are usually public (keyword) unless their only use is by other methods within the class.**
 - By insulating the rest of the program from object details, it is much easier to build large programs, and to reuse objects from previous work.
 - This is called encapsulation or information hiding.

Classes and Objects

- **Classes are 'patterns'**
 - A class can have many data items stored within it
 - A class can have many methods that operate on (and may modify) its data and return results
- **Objects are instances of classes**
 - Objects have their own data, methods and area in memory of computer where they are stored.

Classes and Objects

- **There is no difference between Java library classes and the classes you will build**
 - When you build a class, you are defining a new data type in Java
 - You are essentially extending the Java language by creating a new type of data
- **Classes can also be built from other classes.**
 - A class built on another class extends it.
 - This idea is called *inheritance*, and is covered later in 1.00/1.001.

Using An Existing Class

- **Classes provided in Java libraries are more typically more general than the classes you will write**
- **BigInteger is a Java class that handles arbitrarily large integers**
 - Use it rather than writing your own class to handle arbitrary arithmetic

Using An Existing Class

- **To work with objects:**
 - **First construct them and specify their initial state**
 - **Constructors are special methods of a class that are used to construct and initialize objects of that class.**
 - **They may take arguments (parameters)**
 - **A class may have arbitrary number of different constructors as long as their list of argument types or order are distinct.**
 - **Then apply methods to them**
 - **This is the same as “sending messages” to them to invoke their behaviors**

Constructor for BigInteger Object

- To construct a new BigInteger object, two things are required:

- Give the object a name or identity:

```
BigInteger aLargeInt;  
// Object name is a reference to the object  
// BigInteger is the data type of object  
// referred to by the variable aLargeInt
```

- Create the object (using its constructor)

```
new BigInteger("1000000000000");  
// 'new' allocates memory and calls constructor
```

Constructor for BigInteger Object

- Combine these two things into a single step:

```
BigInteger a= new BigInteger("1000000000000");
```

- We now have a BigInteger object containing the value 1,000,000,000,000. We can now apply methods to it.

Using Methods

- **Methods are invoked using the dot (.) operator**
 - Method always ends with parentheses

```
BigInteger a= new BigInteger("1000000000000");  
BigInteger z= new BigInteger("23");  
BigInteger c= a.add(z);           // c= a + z  
if (z.isProbablePrime(15))      // is z prime?  
    System.out.println("z is probably prime");
```

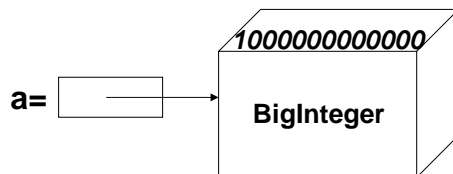
- **Public data fields are also invoked with the dot operator.**

- No parentheses after field name

```
int j = a.somePublicField;      // Example only
```

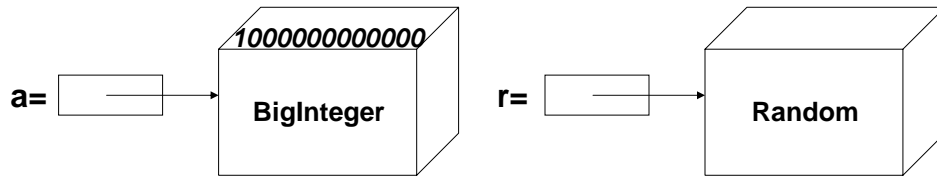
Objects and Names

```
BigInteger a= new BigInteger("1000000000000");
```



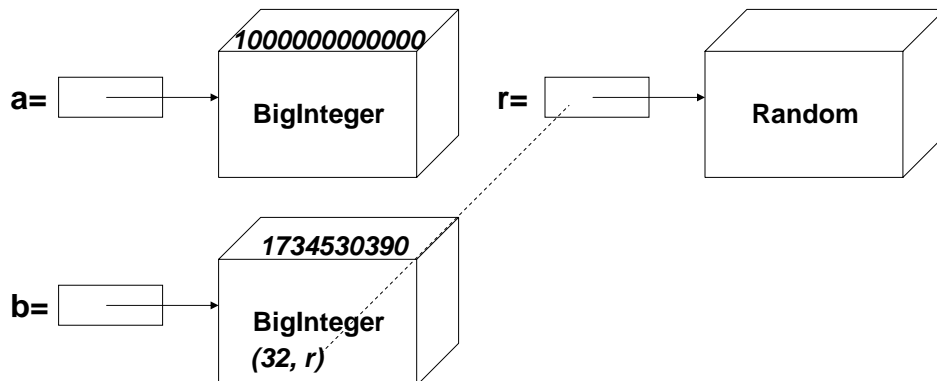
Objects and Names

```
BigInteger a= new BigInteger("1000000000000");  
Random r= new Random();
```



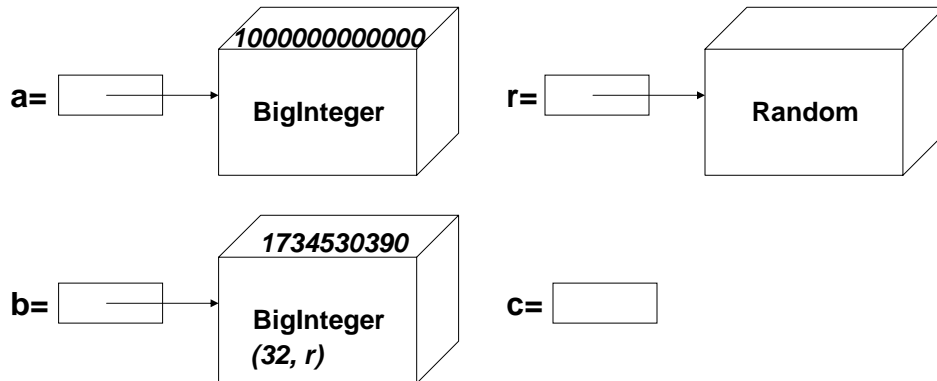
Objects and Names

```
BigInteger a= new BigInteger("1000000000000");  
Random r= new Random();  
BigInteger b= new BigInteger(32, r);
```



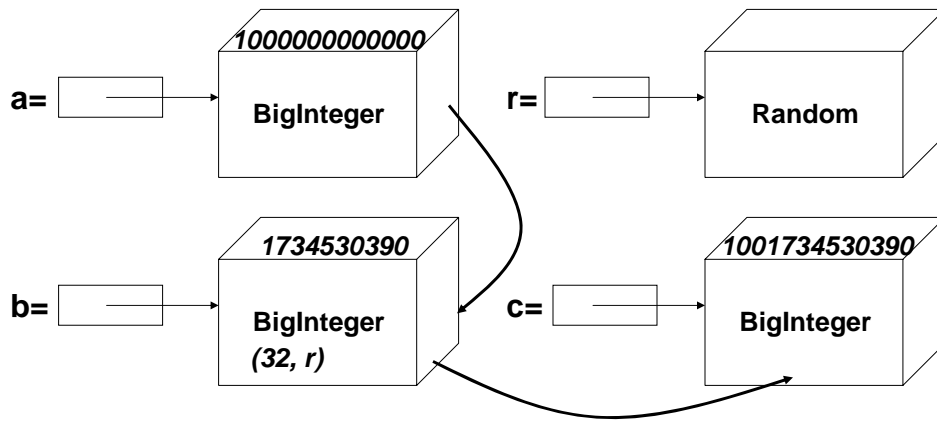
Objects and Names

```
BigInteger a= new BigInteger("1000000000000");  
Random r= new Random();  
BigInteger b= new BigInteger(32, r);  
BigInteger c;
```



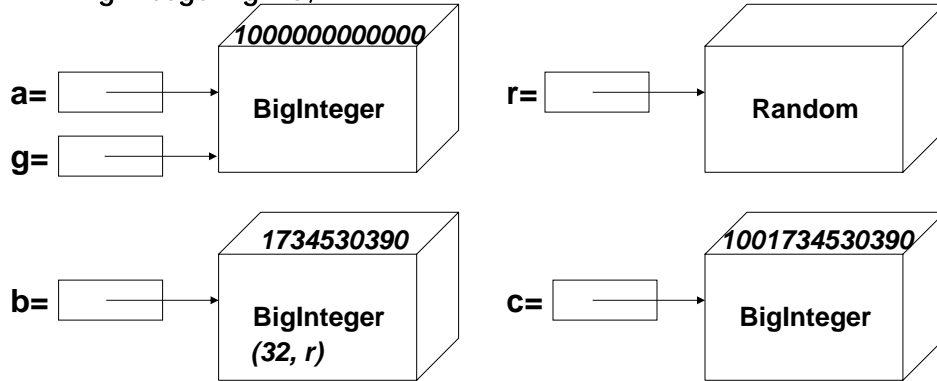
Objects and Names

```
BigInteger a= new BigInteger("1000000000000");  
Random r= new Random();  
BigInteger b= new BigInteger(32, r);  
BigInteger c;  
c= b.add(a);
```



Objects and Names

```
BigInteger a= new BigInteger("1000000000000");
Random r= new Random();
BigInteger b= new BigInteger(32, r);
BigInteger c;
c= b.add(a);
BigInteger g= a;
```



Using the BigInteger Class

```
import java.math.*;           // For BigInteger
import java.util.*;          // For Random
public class BigIntTest {
    public static void main(String[] args) {
        BigInteger a= new BigInteger("1000000000000");
        Random r= new Random();           // Random nbr generator
        BigInteger b= new BigInteger(32, r); // Random
        BigInteger c;
        c= b.add(a);                       // c= b+a
        BigInteger g= a;
        BigInteger d= new BigInteger(32, 10, r); // Prime
        BigInteger e;
        e= c.divide(d);                     // e= c/d
        if (d.isProbablePrime(10))
            System.out.println("d is probably prime");
        else
            System.out.println("d is probably not prime");
        BigInteger f= d.multiply(e);        // f= d*e
    }
}
```

Exercise 1: Existing Class

- Use the `BigDecimal` class (floating point numbers) to:
 - Construct `BigDecimal a = 13 x 10500`
 - Construct `BigDecimal b` randomly
 - Hint: Construct a random `BigInteger`, then use the appropriate `BigDecimal` constructor. See Javadoc
 - Compute `BigDecimal c = a + b`
 - Compute `BigDecimal d = c / a`
 - Look up rounding type in Javadoc
 - Print out `a, b, c, d` after computing each one

Exercise 1: Existing Class

- Write the program in stages:
 - Construct `a`, print it. Compile and debug
 - Don't count the zeros!
 - Add constructing `b`, print it. Compile and debug
 - Do the addition and division. Compile and debug

Exercise 2: Writing A Class

- In homeworks, you will be writing your own classes
 - You've already seen classes in all our examples, but they're not typical
 - They just have a single method, main()
 - Most classes don't have a main() method
- To build a program, you'll write several classes, one of which has a main() method

Point Class

```
public class SimplePoint {
    private double x, y;           // Data members
    public SimplePoint( ) {       // Constructor
        x= 0.0;
        y= 0.0; }
    // Methods
    public double getX( ) { return x; }
    public double getY( ) { return y; }
    public void setX(double xval) { x= xval; }
    public void setY(double yval) { y= yval; }
    public void move(double del taX, double del taY) {
        x += del taX;
        y += del taY; }
} // End of class SimplePoint

// This isn't a program because it doesn't have main()
// but it can be used by classes with a main()
```

Point Class, main()

```
public class SimplePoint1 {
    public static void main(String[] args) {
        SimplePoint a= new SimplePoint( );
        SimplePoint b= new SimplePoint( );
        double xa= a.getX( );
        double ya= a.getY( );
        System.out.println("a= (" + xa + " , " + ya + ")");
        a.move(-9.0, 7.5);
        System.out.println("a= (" + a.getX( ) +
            " , " + a.getY( ) + ")");
    }
}
```

Exercise 2

- **Write a different SimplePoint class that uses polar coordinates instead of Cartesian coordinates**
 - Implement the same public methods as the previous SimplePoint class
 - Use *r* and *theta* as the private data fields
 - Recall that:
 - $x = r \cos(\theta)$
 - $y = r \sin(\theta)$
 - $r = \sqrt{x^2 + y^2}$
 - $\theta = \tan^{-1}(y/x)$
 - Use the Java Math class (capital M)
 - Use `Math.atan2()` for the arctan function
- **Use the same main() as before**

Why Make Data Members Private?

- **By building a class with public methods but private data, you only commit to an interface, not an implementation**
 - If you need to change implementation, you can do so without breaking any code that depends on it, as long as the set of methods stays the same
 - Changing coordinate systems, computational methods, etc., is quite common, as in this example. This allows flexibility as software grows and changes

Point Class, Polar Coordinates

```
class SimplePoint {
    private double r, theta;    // Data members
    public SimplePoint() {    // Constructor
        r= 0.0;
        theta= 0.0; }
    // Methods (trig functions use radians)
    public double getX( ) { return r* Math.cos(theta); }
    public double getY( ) { return r* Math.sin(theta); }
    public void setX(double xval) {
        double yval = r*Math.sin(theta);
        r= Math.sqrt(xval*xval + yval*yval);
        theta= Math.atan2(yval, xval); }
}
```

Point Class, Polar, p.2

```
public void setY(double yval) {
    double xval = r*Math.cos(theta);
    r= Math.sqrt(xval*xval + yval*yval);
    theta= Math.atan2(yval, xval); }
public void move(double del taX, double del taY) {
    double xval = r*Math.cos(theta);
    double yval = r*Math.sin(theta);
    xval += del taX;
    yval += del taY;
    r= Math.sqrt(xval*xval + yval*yval);
    theta= Math.atan2(yval, xval);
}
}

// Can be invoked from same main( ) as before and
// produces the same results (other than rounding errors)
```