Java Review

Selim Aksoy
Bilkent University
Department of Computer Engineering
saksoy@cs.bilkent.edu.tr

Java

- Java Basics
- Java Program Statements
  - Conditional statements
  - Repetition statements (loops)
- Writing Classes in Java
  - Class definitions
  - Encapsulation and Java modifiers
  - Method declaration, invocation, and parameter passing
  - Method overloading

Programming Rules of Thumb

- *Learn program patterns* of general utility (branching, loops, etc.) and *use relevant patterns* for the problem at hand
- *Seek inspiration* by systematically working test data by hand and ask yourself: "what am I doing?"
- *Declare variables* for each piece of information you maintain when working problem by hand
- *Decompose* problem into manageable tasks
- *Remember* the problem's boundary conditions
- *Validate* your program by tracing it on test data with known output

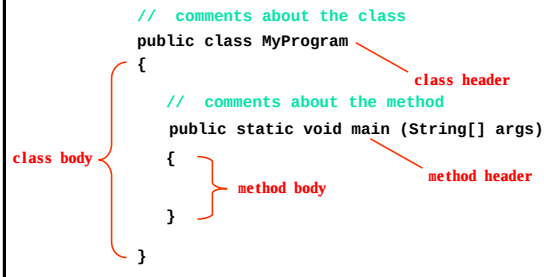
Introduction to Objects

- An *object* represents something with which we can interact in a program
- An object provides a collection of services that we can tell it to perform for us
- The services are defined by methods in a *class* that defines the object
- A class represents a concept, and an object represents the embodiment of a class
- A class can be used to create multiple objects

Java Program Structure

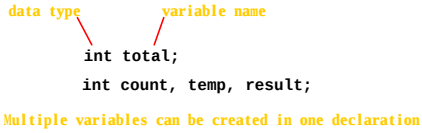
- In the Java programming language:
  - A program is made up of one or more *classes*
  - A class contains one or more *methods*
  - A method contains program *statements*
- Attributes/properties correspond to fields (or variables)
- Behaviors/operations correspond to methods
- A Java application always contains a method called `main`

Java Program Structure



Variables

- A *variable* is a name for a location in memory
- A variable must be *declared* by specifying the variable's name and the type of information that it will hold



Primitive Data

- There are exactly eight primitive data types in Java
- Four of them represent integers:
  - `byte`, `short`, `int`, `long`
- Two of them represent floating point numbers:
  - `float`, `double`
- One of them represents characters:
  - `char`
- And one of them represents boolean values:
  - `boolean`

Numeric Primitive Data

- The difference between the various numeric primitive types is their size, and therefore the values they can store:

Type	Storage	Min Value	Max Value
byte	8 bits	-128	127
short	16 bits	-32,768	32,767
int	32 bits	-2,147,483,648	2,147,483,647
long	64 bits	< -9 x 10^{18}	> 9 x 10^{18}
float	32 bits	+/- 3.4 x 10^{38} with 7 significant digits	
double	64 bits	+/- 1.7 x 10^{308} with 15 significant digits	

Arithmetic Expressions

- An *expression* is a combination of one or more operands and their operators
- *Arithmetic expressions* use the operators:

Addition	+	
Subtraction	-	
Multiplication	*	
Division	/	
Remainder	%	(no ^ operator)

- If either or both operands associated with an arithmetic operator are floating point, the result is a floating point

Division and Remainder

- If both operands to the division operator (/) are integers, the result is an integer (the fractional part is discarded)

```
14 / 3      equals?     4
8 / 12      equals?     0
```

- The remainder operator (%) returns the remainder after dividing the second operand into the first

```
14 % 3      equals?     2
8 % 12      equals?     8
```


String Concatenation

- The *string concatenation operator* (+) is used to append one string to the end of another
- The plus operator (+) is also used for arithmetic addition
- The function that the + operator performs depends on the type of the information on which it operates
  - If at least one operand is a string, it performs string concatenation
  - If both operands are numeric, it adds them
- The + operator is evaluated left to right
- Parentheses can be used to force the operation order

Data Conversions

- In Java, data conversions can occur in three ways:
  - assignment conversion
  - arithmetic promotion
  - casting
- *Assignment conversion* occurs when a value of one type is assigned to a variable of another
  - Only widening conversions can happen via assignment
- *Arithmetic promotion* happens automatically when operators in expressions convert their operands

Data Conversions

- *Casting* is the most powerful, and dangerous, technique for conversion
  - Both widening and narrowing conversions can be accomplished by explicitly casting a value
  - To cast, the type is put in parentheses in front of the value being converted
- For example, if `total` and `count` are integers, but we want a floating point result when dividing them, we can cast `total`:

```
result = (float) total / count;
```


Creating Objects

- A variable holds either a primitive type or a *reference* to an object
- A class name can be used as a type to declare an *object reference variable*

```
String title;
```

- No object is created with this declaration
- An object reference variable holds the address of an object
- The object itself must be created separately

Creating Objects

- Generally, we use the `new` operator to create an object

```
title = new String ("Java Software Solutions");
```

This calls the String *constructor*, which is a special method that sets up the object

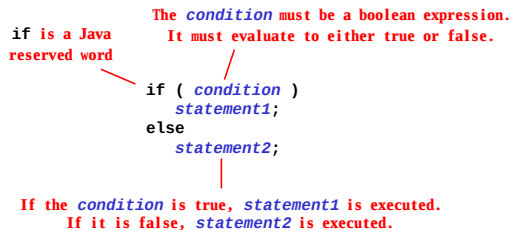
- Creating an object is called *instantiation*
- An object is an *instance* of a particular class

Conditional Statements

- A *conditional statement* lets us choose which statement will be executed next
- Therefore they are sometimes called *selection statements*
- Conditional statements give us the power to make basic decisions
- Java's conditional statements are
  - the *if statement*
  - the *if-else statement*
  - the *switch statement*

The if Statement

- The *if statement* has the following syntax:



Boolean Expressions

- A condition often uses one of Java's *equality operators* or *relational operators*, which all return boolean results:

==	equal to
!=	not equal to
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to

- Note the difference between the equality operator (==) and the assignment operator (=)

Logical Operators

- Boolean expressions can use the following *logical operators*:

!	Logical NOT
&&	Logical AND
||	Logical OR

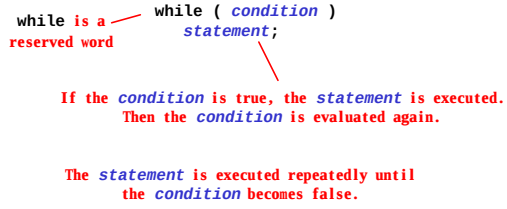
- They all take boolean operands and produce boolean results
- Logical NOT is a unary operator (it operates on one operand)
- Logical AND and logical OR are binary operators (each operates on two operands)

Repetition Statements

- *Repetition statements* allow us to execute a statement multiple times
- Often they are referred to as *loops*
- Like conditional statements, they are controlled by boolean expressions
- Java has three kinds of repetition statements:
  - the *while loop*
  - the *do loop*
  - the *for loop*
- The programmer should choose the right kind of loop for the situation

The while Statement

- The *while statement* has the following syntax:

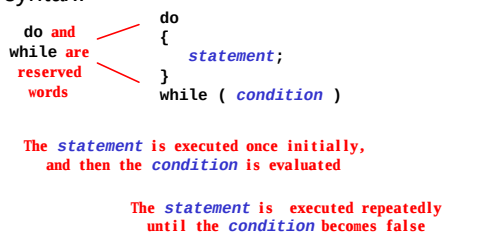


Example

```
//********************************************************************
//  Counter.java       Author: Lewis/Loftus
//
//  Demonstrates the use of a while loop.
//********************************************************************

public class Counter
{
   //-----------------------------------------------------------------
   //  Prints integer values from 1 to a specific limit.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      final int LIMIT = 5;
      int count = 1;

      while (count <= LIMIT)
      {
         System.out.println (count);
         count = count + 1;
      }

      System.out.println ("Done");
   }
}
```


The do Statement

- The *do statement* has the following syntax:



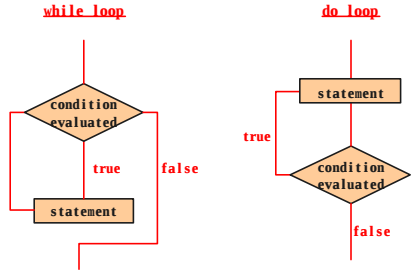
The do Statement

- A `do` loop is similar to a `while` loop, except that the condition is evaluated after the body of the loop is executed
- Therefore the body of a `do` loop will execute at least once

Example

```
//********************************************************************
//  Counter2.java       Author: Lewis/Loftus
//
//  Demonstrates the use of a do loop.
//********************************************************************

public class Counter2
{
   //-----------------------------------------------------------------
   //  Prints integer values from 1 to a specific limit.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      final int LIMIT = 5;
      int count = 0;

      do
      {
         count = count + 1;
         System.out.println (count);
      }
      while (count < LIMIT);

      System.out.println ("Done");
   }
}
```


Comparing while and do



The for Statement

- The *for statement* has the following syntax:

Reserved word

The *initialization* is executed once before the loop begins

The *statement* is executed until the *condition* becomes false

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

The *increment* portion is executed at the end of each iteration

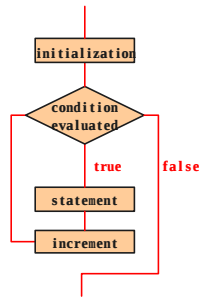
The *condition-statement-increment* cycle is executed repeatedly

The for Statement

- A `for` loop is functionally equivalent to the following `while` loop structure:

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


Logic of a for loop



The for Statement

- Like a `while` loop, the condition of a `for` statement is tested prior to executing the loop body
- Therefore, the body of a `for` loop will execute zero or more times
- It is well suited for executing a loop a specific number of times that can be determined in advance

Example

```
//********************************************************************
//  Counter3.java       Author: Lewis/Loftus
//
//  Demonstrates the use of a for loop.
//********************************************************************

public class Counter3
{
   //-----------------------------------------------------------------
   //  Prints integer values from 1 to a specific limit.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      final int LIMIT = 5;

      for (int count=1; count <= LIMIT; count++)
         System.out.println (count);

      System.out.println ("Done");
   }
}
```


Choosing a Loop Structure

- When you can't determine how many times you want to execute the loop body, use a `while` statement or a `do` statement
  - If it might be zero or more times, use a `while` statement
  - If it will be at least once, use a `do` statement
- If you can determine how many times you want to execute the loop body, use a `for` statement

The switch Statement

- The general syntax of a `switch` statement is:

```
switch ( expression )
{
   case value1 :
      statement-list1;
      break;
   case value2 :
      statement-list2;
      break;
   case value3 :
      statement-list3;
      break;
   case  ...
}
```

switch and case are reserved words

If *expression* matches *value2*, control jumps to here

The switch Statement

- The expression of a `switch` statement must result in an *integral type*, meaning an `int` or a `char`
- It cannot be a `boolean` value, a floating point value (`float` or `double`), a `byte`, a `short`, or a `long`
- The implicit boolean condition in a `switch` statement is equality - it tries to match the expression with a value
- You cannot perform relational checks with a `switch` statement

Comparing Strings

- Remember that a character string in Java is an object
- The `equals` method can be called with strings to determine if two strings contain exactly the same characters in the same order
- The `String` class also contains a method called `compareTo` to determine if one string comes before another in lexicographic order (based on the Unicode character set)
- This is not strictly alphabetical when uppercase and lowercase characters are mixed

Comparing Float Values

- We also have to be careful when comparing two floating point values (`float` or `double`) for equality
- You should rarely use the equality operator (==) when comparing two floats
- In many situations, you might consider two floating point numbers to be "close enough" even if they aren't exactly equal
- Therefore, to determine the equality of two floats, you may want to use the following technique:

```
if (Math.abs(f1 - f2) < 0.00001)
   System.out.println ("Essentially equal.");
```


Increment and Decrement

- The increment and decrement operators are arithmetic and operate on one operand
- The *increment operator* (++) adds one to its operand
- The *decrement operator* (--) subtracts one from its operand
- The statement

```
count++;
```

  is functionally equivalent to

```
count = count + 1;
```


Assignment Operators

- There are many assignment operators, including the following:

Operator	Example	Equivalent To
+=	x += y	x = x + y
-=	x -= y	x = x - y
*=	x *= y	x = x * y
/=	x /= y	x = x / y
%=	x %= y	x = x % y

Objects and Classes

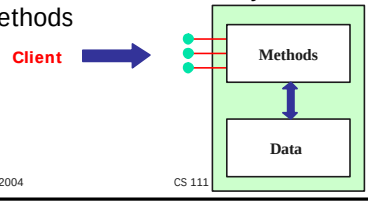
- An object has:
  - *state* - descriptive characteristics
  - *behaviors* - what it can do (or what can be done to it)
- A class is the model or pattern from which objects are created
- For example, consider a coin that can be flipped so that it's face shows either "heads" or "tails"
- The state of the coin is its current face (heads or tails)
- The behavior of the coin is that it can be flipped

Encapsulation

- We can take one of two views of an object:
  - internal - the variables the object holds and the methods that make the object useful
  - external - the services that an object provides and how the object interacts
- Any changes to the object's state (its variables) should be made only by that object's methods
- We should make it difficult, if not impossible, to access an object's variables other than via its methods
- The user, or *client*, of an object can request its services, but it should not have to be aware of how those services are accomplished

Encapsulation

- An encapsulated object can be thought of as a *black box*
- Its inner workings are hidden to the client, which invokes only the interface methods



Visibility Modifiers

- In Java, we accomplish encapsulation through the appropriate use of *visibility modifiers*
- A *modifier* is a Java reserved word that specifies particular characteristics of a method or data value
- Members of a class that are declared with *public visibility* can be accessed from anywhere (public variables violate encapsulation)
- Members of a class that are declared with *private visibility* can only be accessed from inside the class

Visibility Modifiers

	public	private
Variables	Violate encapsulation	Enforce encapsulation
Methods	Provide services to clients	Support other methods in the class

Data Scope

- The *scope* of data is the area in a program in which that data can be used (referenced)
- Data declared at the class level can be used by all methods in that class
- Data declared within a block (enclosed within { and }, if statements, loops) can be used only in that block

Example

```
import java.text.NumberFormat;

public class Account
{
   private NumberFormat fmt = NumberFormat.getCurrencyInstance();
   private final double RATE = 0.035;  // interest rate of 3.5%
   private long acctNumber;
   private double balance;
   private String name;

   //-----------------------------------------------------------------
   //  Sets up the account by defining its owner, account number,
   //  and initial balance.
   //-----------------------------------------------------------------
   public Account (String owner, long account, double initial)
   {
      name = owner;
      acctNumber = account;
      balance = initial;
   }

   //-----------------------------------------------------------------
   //  Validates the transaction, then deposits the specified amount
   //  into the account. Returns the new balance.
   //-----------------------------------------------------------------
   public double deposit (double amount)
   {
      if (amount < 0)  // deposit value is negative
      {
         System.out.println ();
         System.out.println ("Error: Deposit amount is invalid.");
         System.out.println (acctNumber + " " + fmt.format(amount));
      }
      else
         balance = balance + amount;

      return balance;
   }
…
```


Example

```
…
   //-----------------------------------------------------------------
   //  Validates the transaction, then withdraws the specified amount
   //  from the account. Returns the new balance.
   //-----------------------------------------------------------------
   public double withdraw (double amount, double fee)
   {
      amount += fee;

      if (amount < 0)  // withdraw value is negative
      {
         System.out.println ();
         System.out.println ("Error: Withdraw amount is invalid.");
         System.out.println ("Account: " + acctNumber);
         System.out.println ("Requested: " + fmt.format(amount));
      }
      else
         if (amount > balance)  // withdraw value exceeds balance
         {
            System.out.println ();
            System.out.println ("Error: Insufficient funds.");
            System.out.println ("Account: " + acctNumber);
            System.out.println ("Requested: " + fmt.format(amount));
            System.out.println ("Available: " + fmt.format(balance));
         }
         else
            balance = balance - amount;

      return balance;
   }
…
```

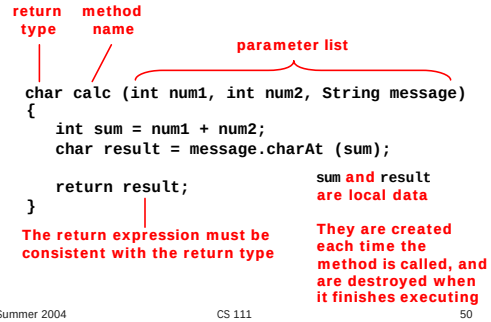

Example

```
…
   //-----------------------------------------------------------------
   //  Adds interest to the account and returns the new balance.
   //-----------------------------------------------------------------
   public double addInterest ()
   {
      balance += (balance * RATE);
      return balance;
   }

   //-----------------------------------------------------------------
   //  Returns the current balance of the account.
   //-----------------------------------------------------------------
   public double getBalance ()
   {
      return balance;
   }

   //-----------------------------------------------------------------
   //  Returns the account number.
   //-----------------------------------------------------------------
   public long getAccountNumber ()
   {
      return acctNumber;
   }

   //-----------------------------------------------------------------
   //  Returns a one-line description of the account as a string.
   //-----------------------------------------------------------------
   public String toString ()
   {
      return (acctNumber + "\t" + name + "\t" + fmt.format(balance));
   }
}
```

Example

```
public class Banking
{
   //-----------------------------------------------------------------
   //  Creates some bank accounts and requests various services.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      Account acct1 = new Account ("Ted Murphy", 72354, 102.56);
      Account acct2 = new Account ("Jane Smith", 69713, 40.00);
      Account acct3 = new Account ("Edward Demsey", 93757, 759.32);

      acct1.deposit (25.85);
      double smithBalance = acct2.deposit (500.00);
      System.out.println ("Smith balance after deposit: " +
                          smithBalance);
      System.out.println ("Smith balance after withdrawal: " +
                          acct2.withdraw (430.75, 1.50));
      acct3.withdraw (800.00, 0.0);  // exceeds balance
      acct1.addInterest();
      acct2.addInterest();
      acct3.addInterest();

      System.out.println ();
      System.out.println (acct1);
      System.out.println (acct2);
      System.out.println (acct3);
   }
}
```


Method Header and Body



The return Statement

- The *return type* of a method indicates the type of value that the method sends back to the calling location
- A method that does not return a value has a `void` return type
- A *return statement* specifies the value that will be returned

```
return expression;
```

- Its expression must conform to the return type

Constructors Revisited

- Recall that a constructor is a special method that is used to initialize a newly created object
- When writing a constructor, remember that:
  - it has the same name as the class
  - it does not return a value
  - it has no return type, not even `void`
  - it typically sets the initial values of instance variables
- The programmer does not have to define a constructor for a class

Overloading Methods

- *Method overloading* is the process of using the same method name for multiple methods
- The *signature* of each overloaded method must be unique
- The signature includes the number, type, and order of the parameters
- The compiler determines which version of the method is being invoked by analyzing the parameters
- The return type of the method is not part of the signature

Overloading Methods

Version 1

```
float tryMe (int x)
{
   return x + .375;
}
```

Version 2

```
float tryMe (int x, float y)
{
   return x*y;
}
```

Invocation

```
result = tryMe (25, 4.32)
```

Object Relationships

- Some use associations occur between objects of the same class
- For example, we might add two `Rational` number objects together as follows:

```
r3 = r1.add(r2);
```

- One object (`r1`) is executing the method and another (`r2`) is passed as a parameter

Example

```
//********************************************************************
//  Rational.java       Author: Lewis/Loftus
//
//  Represents one rational number with a numerator and denominator.
//********************************************************************

public class Rational
{
   private int numerator, denominator;

   //-----------------------------------------------------------------
   //  Sets up the rational number by ensuring a nonzero denominator
   //  and making only the numerator signed.
   //-----------------------------------------------------------------
   public Rational (int numer, int denom)
   {
      if (denom == 0)
         denom = 1;

      // Make the numerator "store" the sign
      if (denom < 0)
      {
         numer = numer * -1;
         denom = denom * -1;
      }

      numerator = numer;
      denominator = denom;

      reduce();
   }

   //-----------------------------------------------------------------
   //  Returns the numerator of this rational number.
   //-----------------------------------------------------------------
   public int getNumerator ()
   {
      return numerator;
   }

   //-----------------------------------------------------------------
   //  Returns the denominator of this rational number.
   //-----------------------------------------------------------------
   public int getDenominator ()
   {
      return denominator;
   }
...
```


Example

```
...
   //-----------------------------------------------------------------
   //  Returns the reciprocal of this rational number.
   //-----------------------------------------------------------------
   public Rational reciprocal ()
   {
      return new Rational (denominator, numerator);
   }

   //-----------------------------------------------------------------
   //  Adds this rational number to the one passed as a parameter.
   //  A common denominator is found by multiplying the individual
   //  denominators.
   //-----------------------------------------------------------------
   public Rational add (Rational op2)
   {
      int commonDenominator = denominator * op2.getDenominator();
      int numerator1 = numerator * op2.getDenominator();
      int numerator2 = op2.getNumerator() * denominator;
      int sum = numerator1 + numerator2;

      return new Rational (sum, commonDenominator);
   }

   //-----------------------------------------------------------------
   //  Subtracts the rational number passed as a parameter from this
   //  rational number.
   //-----------------------------------------------------------------
   public Rational subtract (Rational op2)
   {
      int commonDenominator = denominator * op2.getDenominator();
      int numerator1 = numerator * op2.getDenominator();
      int numerator2 = op2.getNumerator() * denominator;
      int difference = numerator1 - numerator2;

      return new Rational (difference, commonDenominator);
   }
...
```


Example

```
...
   //-----------------------------------------------------------------
   //  Multiplies this rational number by the one passed as a
   //  parameter.
   //-----------------------------------------------------------------
   public Rational multiply (Rational op2)
   {
      int numer = numerator * op2.getNumerator();
      int denom = denominator * op2.getDenominator();

      return new Rational (numer, denom);
   }

   //-----------------------------------------------------------------
   //  Divides this rational number by the one passed as a parameter
   //  by multiplying by the reciprocal of the second rational.
   //-----------------------------------------------------------------
   public Rational divide (Rational op2)
   {
      return multiply (op2.reciprocal());
   }

   //-----------------------------------------------------------------
   //  Determines if this rational number is equal to the one passed
   //  as a parameter.  Assumes they are both reduced.
   //-----------------------------------------------------------------
   public boolean equals (Rational op2)
   {
      return ( numerator == op2.getNumerator() &&
               denominator == op2.getDenominator() );
   }
...
```


Example

```
...
   //-----------------------------------------------------------------
   //  Returns this rational number as a string.
   //-----------------------------------------------------------------
   public String toString ()
   {
      String result;

      if (numerator == 0)
         result = "0";
      else
         if (denominator == 1)
            result = numerator + "";
         else
            result = numerator + "/" + denominator;

      return result;
   }

   //-----------------------------------------------------------------
   //  Reduces this rational number by dividing both the numerator
   //  and the denominator by their greatest common divisor.
   //-----------------------------------------------------------------
   private void reduce ()
   {
      if (numerator != 0)
      {
         int common = gcd (Math.abs(numerator), denominator);

         numerator = numerator / common;
         denominator = denominator / common;
      }
   }

   //-----------------------------------------------------------------
   //  Computes and returns the greatest common divisor of the two
   //  positive parameters. Uses Euclid's algorithm.
   //-----------------------------------------------------------------
   private int gcd (int num1, int num2)
   {
      while (num1 != num2)
         if (num1 > num2)
            num1 = num1 - num2;
         else
            num2 = num2 - num1;

      return num1;
   }
}
```


Example

```
public class RationalNumbers
{
   //-----------------------------------------------------------------
   //  Creates some rational number objects and performs various
   //  operations on them.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      Rational r1 = new Rational (6, 8);
      Rational r2 = new Rational (1, 3);
      Rational r3, r4, r5, r6, r7;

      System.out.println ("First rational number: " + r1);
      System.out.println ("Second rational number: " + r2);

      if (r1.equals(r2))
         System.out.println ("r1 and r2 are equal.");
      else
         System.out.println ("r1 and r2 are NOT equal.");

      r3 = r1.reciprocal();
      System.out.println ("The reciprocal of r1 is: " + r3);

      r4 = r1.add(r2);
      r5 = r1.subtract(r2);
      r6 = r1.multiply(r2);
      r7 = r1.divide(r2);

      System.out.println ("r1 + r2: " + r4);
      System.out.println ("r1 - r2: " + r5);
      System.out.println ("r1 * r2: " + r6);
      System.out.println ("r1 / r2: " + r7);
   }
}
```

Example

```
public class Address
{
   private String streetAddress, city, state;
   private long zipCode;

   //-----------------------------------------------------------------
   //  Sets up this Address object with the specified data.
   //-----------------------------------------------------------------
   public Address (String street, String town, String st, long zip)
   {
      streetAddress = street;
      city = town;
      state = st;
      zipCode = zip;
   }

   //-----------------------------------------------------------------
   //  Returns this Address object as a string.
   //-----------------------------------------------------------------
   public String toString()
   {
      String result;

      result = streetAddress + "\n";
      result += city + ", " + state + "  " + zipCode;

      return result;
   }
}
```


Example

```
public class Student
{
   private String firstName, lastName;
   private Address homeAddress, schoolAddress;

   //-----------------------------------------------------------------
   //  Sets up this Student object with the specified initial values.
   //-----------------------------------------------------------------
   public Student (String first, String last, Address home,
                   Address school)
   {
      firstName = first;
      lastName = last;
      homeAddress = home;
      schoolAddress = school;
   }

   //-----------------------------------------------------------------
   //  Returns this Student object as a string.
   //-----------------------------------------------------------------
   public String toString()
   {
      String result;

      result = firstName + " " + lastName + "\n";
      result += "Home Address:\n" + homeAddress + "\n";
      result += "School Address:\n" + schoolAddress;

      return result;
   }
}
```


Example

```
//********************************************************************
//  StudentBody.java       Author: Lewis/Loftus
//
//  Demonstrates the use of an aggregate class.
//********************************************************************

public class StudentBody
{
   //-----------------------------------------------------------------
   //  Creates some Address and Student objects and prints them.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      Address school = new Address ("800 Lancaster Ave.", "Villanova",
                                    "PA", 19085);

      Address jHome = new Address ("21 Jump Street", "Lynchburg",
                                   "VA", 24551);
      Student john = new Student ("John", "Smith", jHome, school);

      Address mHome = new Address ("123 Main Street", "Euclid", "OH",
                                   44132);
      Student marsha = new Student ("Marsha", "Jones", mHome, school);

      System.out.println (john);
      System.out.println ();
      System.out.println (marsha);
   }
}
```