CSE 1201
Object Oriented Programming

Object Oriented Design

Acknowledgement

- For preparing the slides I took materials from the following sources
  - Course Slides of Dr. Tagrul Dayar, Bilkent University
  - Java book "*Java Software Solutions*" by Lewis & Loftus.

Program Development

- The creation of software involves four basic activities:
  - establishing the requirements
  - creating a design
  - implementing the code
  - testing the implementation
- These activities are not strictly linear – they overlap and interact

Requirements

- *Software requirements* specify the tasks that a program must accomplish
  - what to do, not how to do it
- Often an initial set of requirements is provided, but they should be critiqued and expanded
- It is difficult to establish detailed, unambiguous, and complete requirements
- Careful attention to the requirements can save significant time and expense in the overall project

Design

- A *software design* specifies how a program will accomplish its requirements
- That is, a software design determines:
  - how the solution can be broken down into manageable pieces
  - what each piece will do
- An object-oriented design determines which classes and objects are needed, and specifies how they will interact
- Low level design details include how individual methods will accomplish their tasks

Implementation

- *Implementation* is the process of translating a design into source code
- Novice programmers often think that writing code is the heart of software development, but actually it should be the least creative step
- Almost all important decisions are made during requirements and design stages
- Implementation should focus on coding details, including style guidelines and documentation

Testing

- *Testing* attempts to ensure that the program will solve the intended problem under all the constraints specified in the requirements
- A program should be thoroughly tested with the goal of finding errors
- *Debugging* is the process of determining the cause of a problem and fixing it

Outline

Software Development Activities

Identifying Classes and Objects

Static Variables and Methods

Class Relationships

Interfaces

Enumerated Types Revisited

Method Design

Testing

GUI Design and Layout

Identifying Classes and Objects

- The core activity of object-oriented design is determining the classes and objects that will make up the solution
- The classes may be part of a class library, reused from a previous project, or newly written
- One way to identify potential classes is to identify the objects discussed in the requirements
- Objects are generally nouns, and the services that an object provides are generally verbs

Identifying Classes and Objects

- A partial requirements document:

The user must be allowed to specify each product by its primary characteristics, including its name and product number. If the bar code does not match the product, then an error should be generated to the message window and entered into the error log. The summary report of all transactions must be structured as specified in section 7.A.

Of course, not all nouns will correspond to a class or object in the final solution

Guidelines for Discovering Objects

- Limit responsibilities of each analysis class
- Use clear and consistent names for classes and methods
- Keep analysis classes simple

Limit Responsibilities

- Each class should have a clear and simple purpose for existence.
- Having classes with too many responsibilities make them difficult to understand and maintain.
- A good test for this is trying to explain the functionality of a class in a few sentences.

Limiting Responsibilities

- As the design progresses, and more feedback is gotten from potential end-users, the trend of an project is to become more complicated
- Therefore it is probably ok to have tiny objects.
- It is still possible to play out a skinny class in your project and later decide that it can be merged with other classes.

Use Clear and Consistent Names

- Companies sometimes spend millions just to change their name into a catchier one.
- You should give a similar effort to let your classes and methods have suitable names.
- class names should be nouns.
- Not finding a good name could mean the boundaries of your class is too fuzzy
- Having too many simple classes is ok if you have good and descriptive names for them.

Keep Classes Simple

- **In this first step, your imagination should not be crippled with worrying about details like object relationships**

Identifying Classes and Objects

- Remember that a class represents a group (classification) of objects with the same behaviors
- Generally, classes that represent objects should be given names that are singular nouns
- Examples: `Coin`, `Student`, `Message`
- A class represents the concept of one such object
- We are free to instantiate as many of each object as needed

Identifying Classes and Objects

- Sometimes it is challenging to decide whether something should be represented as a class
- For example, should an employee's address be represented as a set of instance variables or as an `Address` object
- The more you examine the problem and its details the more clear these issues become
- When a class becomes too complex, it often should be decomposed into multiple smaller classes to distribute the responsibilities

Identifying Classes and Objects

- We want to define classes with the proper amount of detail
- For example, it may be unnecessary to create separate classes for each type of appliance in a house
- It may be sufficient to define a more general `Appliance` class with appropriate instance data
- It all depends on the details of the problem being solved

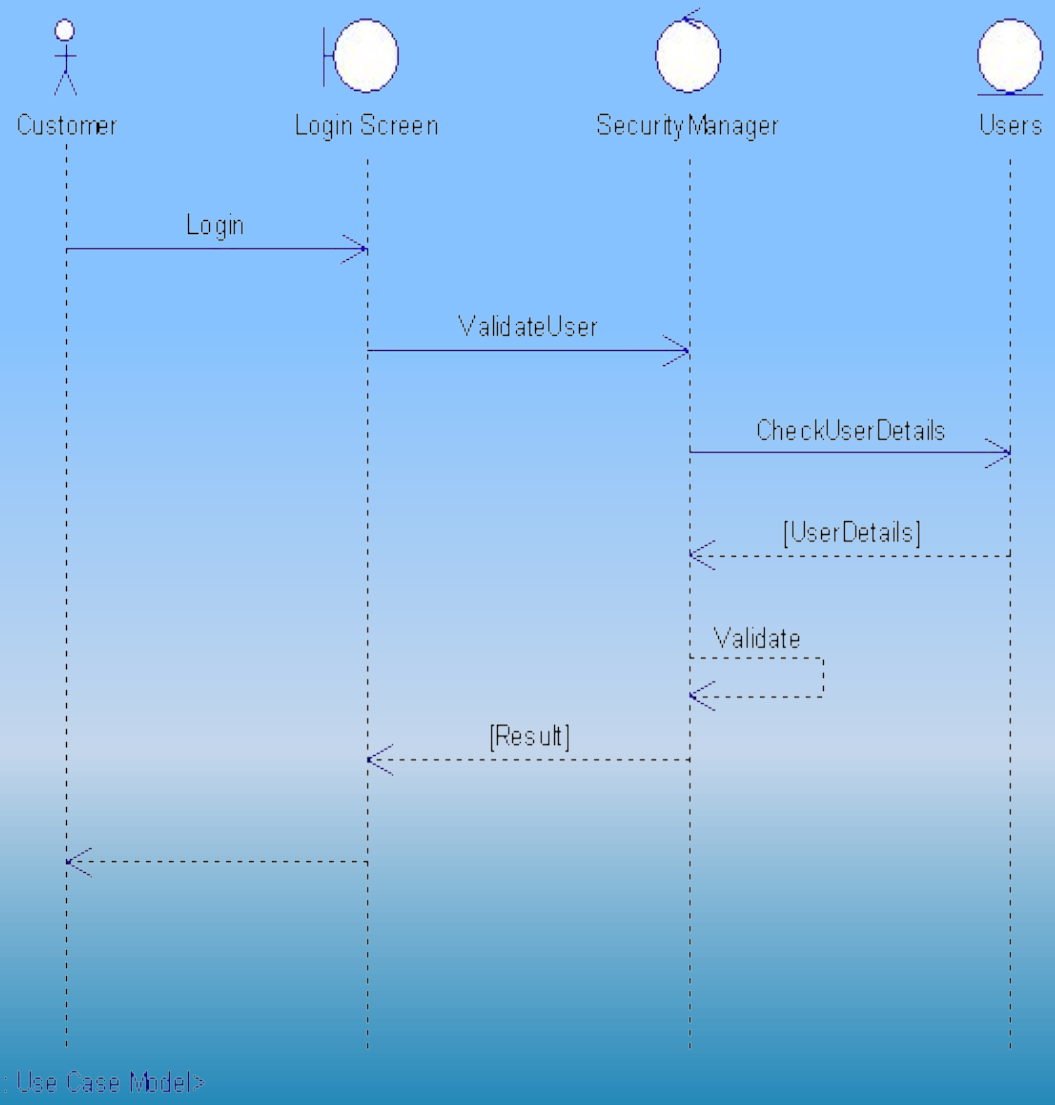
Identifying Classes and Objects

- Part of identifying the classes we need is the process of *assigning responsibilities* to each class
- Every activity that a program must accomplish must be represented by one or more methods in one or more classes
- We generally use verbs for the names of methods
- In early stages it is not necessary to determine every method of every class – begin with primary responsibilities and evolve the design

Describe Behavior

- The set of methods also dictate how your objects interact with each other to produce a solution.
- Sequence diagrams can help tracing object methods and interactions

Sequence Diagram Example



Cohesion between Methods

- methods of an object should be in harmony. If a method seems out of place, then your object might be better off by giving that responsibility to somewhere else.
- For example, getPosition(), getVelocity(), getAcceleration(), getColor()

Use clear and Unambiguous Method Names

- **Having good names may prevent others to have a need for documentation.**
- **If you cannot find a good name, it might mean that your object is not clearly defined, or you are trying to do too much inside your method.**

Outline

Software Development Activities

Identifying Classes and Objects

Static Variables and Methods

Class Relationships

Interfaces

Enumerated Types Revisited

Method Design

Testing

GUI Design and Layout

Static Class Members

- Recall that a static method is one that can be invoked through its class name
- For example, the methods of the `Math` class are static:

```
result = Math.sqrt(25)
```

- Variables can be static as well
- Determining if a method or variable should be static is an important design decision

The static Modifier

- We declare static methods and variables using the `static` modifier
- It associates the method or variable with the class rather than with an object of that class
- Static methods are sometimes called *class methods* and static variables are sometimes called *class variables*
- Let's carefully consider the implications of each

Static Variables

- Normally, each object has its own data space, but if a variable is declared as static, only one copy of the variable exists

```
private static float price;
```

- Memory space for a static variable is created when the class is first referenced
- All objects instantiated from the class share its static variables
- Changing the value of a static variable in one object changes it for all others

Static Methods

```
class Helper
{
    public static int cube (int num)
    {
            return num * num * num;
    }
}
```

Because it is declared as static, the method can be invoked as

```
value = Helper.cube(5);
```

Static Class Members

- The order of the modifiers can be interchanged, but by convention visibility modifiers come first
- Recall that the `main` method is static – it is invoked by the Java interpreter without creating an object
- Static methods cannot reference instance variables because instance variables don't exist until an object exists
- However, a static method can reference static variables or local variables

Static Class Members

- Static methods and static variables often work together
- The following example keeps track of how many objects have been created using a static variable, and makes that information available using a static method

```
class MyClass {
   private static int count = 0;

   public MyClass () {
      count++;
   }
   public static int getCount () {
      return count;
   }
}
```

```
      MyClass obj;

      for (int scan=1; scan <= 10; scan++)
         obj = new MyClass();

      System.out.println ("Objects created: " +
       MyClass.getCount());
```

Student Id prolem

- Let's suppose we have a Student class
- How do we assign unique student id's to each student object that we create?
- What if we also want to get the latest Student created? Like:

```
public static String getLatestStudent()
```

The this Reference

- The `this` reference allows an object to refer to itself
- That is, the `this` reference, used inside a method, refers to the object through which the method is being executed
- Suppose the `this` reference is used in a method called `tryMe`, which is invoked as follows:

```
obj1.tryMe();
obj2.tryMe();
```

- In the first invocation, the `this` reference refers to `obj1`; in the second it refers to `obj2`

The this reference

- The `this` reference can be used to distinguish the instance variables of a class from corresponding method parameters with the same names
- The constructor of the `Account` class (from Chapter 4) could have been written as follows:

```
public Account (Sring name, long acctNumber,
                double balance)
{
   this.name = name;
   this.acctNumber = acctNumber;
   this.balance = balance;
}
```

Outline

Software Development Activities

Identifying Classes and Objects

Static Variables and Methods

Class Relationships

Interfaces

Enumerated Types Revisited

Method Design

Testing

GUI Design and Layout

Class Relationships

- Classes in a software system can have various types of relationships to each other
- Three of the most common relationships:
  - Dependency: A *uses* B
  - Aggregation: A *has-a* B
  - Inheritance: A *is-a* B
- Let's discuss dependency and aggregation further

Dependency

- A *dependency* exists when one class relies on another in some way, usually by invoking the methods of the other
- We've seen dependencies in many previous examples
- We don't want numerous or complex dependencies among classes
- Nor do we want complex classes that don't depend on others
- A good design strikes the right balance

Dependency

- Some dependencies occur between objects of the same class
- A method of the class may accept an object of the same class as a parameter
- For example, the `concat` method of the `String` class takes as a parameter another `String` object

```
str3 = str1.concat(str2);
```

- This drives home the idea that the service is being requested from a particular object

Dependency

- The following example defines a class called `Rational` to represent a rational number
- A rational number is a value that can be represented as the ratio of two integers
- Some methods of the `Rational` class accept another `Rational` object as a parameter

Using Rational Class

```
RationalNumber r1 = new RationalNumber (6, 8);
RationalNumber r2 = new RationalNumber (1, 3);
RationalNumber 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);
```

```
public class RationalNumber {
  private int numerator, denominator;

  //-----------------------------------------------------------------
  //  Constructor: Sets up the rational number by ensuring a nonzero
  //  denominator and making only the numerator signed.
  //-----------------------------------------------------------------
  public RationalNumber (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;
  }

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

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

    return new RationalNumber (sum, commonDenominator);
  }

public RationalNumber subtract (RationalNumber op2)   {
    int commonDenominator = denominator * op2.getDenominator();
    int numerator1 = numerator * op2.getDenominator();
    int numerator2 = op2.getNumerator() * denominator;
    int difference = numerator1 - numerator2;

    return new RationalNumber (difference, commonDenominator);
  }
```

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

    return new RationalNumber (numer, denom);
  }

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

```
public boolean equals (RationalNumber op2)
  {
    return ( numerator == op2.getNumerator() &&
             denominator == op2.getDenominator() );
  }

  //-----------------------------------------------------------------
  //  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;
  }
```

Aggregation

- An *aggregate* is an object that is made up of other objects
- Therefore aggregation is a *has-a* relationship
  - A car *has a* chassis
- In software, an aggregate object contains references to other objects as instance data
- The aggregate object is defined in part by the objects that make it up
- This is a special kind of dependency – the aggregate usually relies on the objects that compose it

Aggregation

- In the following example, a `Student` object is composed, in part, of `Address` objects
- A student has an address (in fact each student has two addresses)
- See StudentBody.java (page 304)
- See Student.java (page 306)
- See Address.java (page 307)
- An aggregation association is shown in a UML class diagram using an open diamond at the aggregate end

StudentBody.java

```
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);
```

Student.java

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

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

public String toString()
   { ….}
}
```

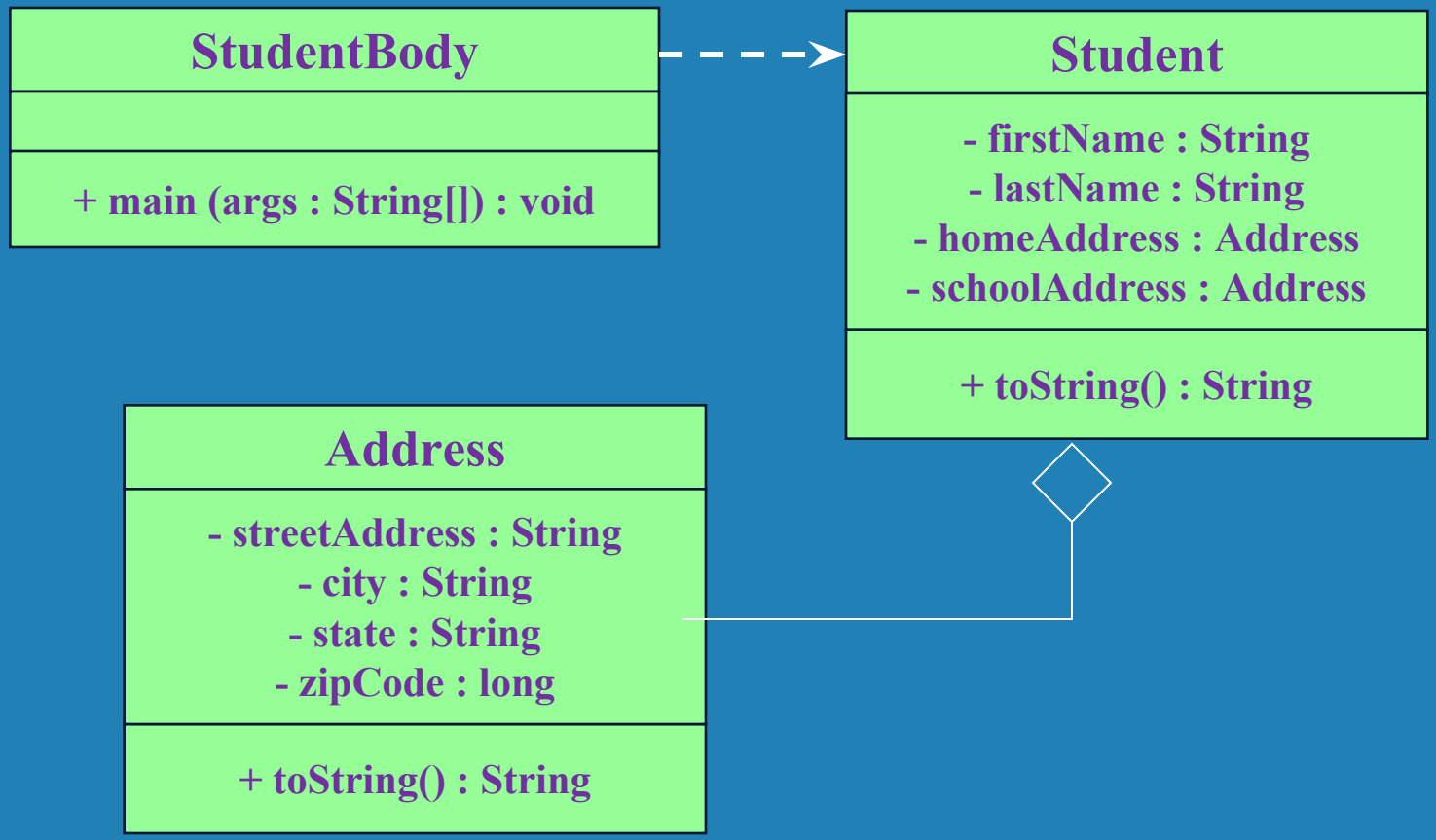
Address

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

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

   //-----------------------------------------------------------------
   //  Returns a description of this Address object.
   //-----------------------------------------------------------------
   public String toString() {}
}
```

Aggregation in UML



Objects as Parameters

- Another important issue related to method design involves parameter passing
- Parameters in a Java method are *passed by value*
- A copy of the actual parameter (the value passed in) is stored into the formal parameter (in the method header)
- Therefore passing parameters is similar to an assignment statement
- When an object is passed to a method, the actual parameter and the formal parameter become aliases of each other

Passing Objects to Methods

- What a method does with a parameter may or may not have a permanent effect (outside the method)
- See ParameterTester.java
- See ParameterModifier.java
- See Num.java
- Note the difference between changing the internal state of an object versus changing which object a reference points to

ParameterPassing

```
ParameterTester tester = new ParameterTester();

    int a1 = 111;
    Num a2 = new Num (222);
    Num a3 = new Num (333);

    System.out.println ("Before calling changeValues:");
    System.out.println ("a1\ta2\ta3");
    System.out.println (a1 + "\t" + a2 + "\t" + a3 + "\n");

    tester.changeValues (a1, a2, a3);

    System.out.println ("After calling changeValues:");
    System.out.println ("a1\ta2\ta3");
    System.out.println (a1 + "\t" + a2 + "\t" + a3 + "\n");
```

ParameterTester

```
class ParameterTester
{
  //-----------------------------------------------------------------
  //  Modifies the parameters, printing their values before and
  //  after making the changes.
  //-----------------------------------------------------------------
  public void changeValues (int f1, Num f2, Num f3)
  {
    System.out.println ("Before changing the values:");
    System.out.println ("f1\tf2\tf3");
    System.out.println (f1 + "\t" + f2 + "\t" + f3 + "\n");

    f1 = 999;
    f2.setValue(888);
    f3 = new Num (777);

    System.out.println ("After changing the values:");
    System.out.println ("f1\tf2\tf3");
    System.out.println (f1 + "\t" + f2 + "\t" + f3 + "\n");
  }
}
```

Num

```
class Num {
   private int value;

public Num (int update) {
      value = update;
   }

public void setValue (int update)
   {
      value = update;
   }

public String toString () {
      return value + "";
   }
}
```

Method Overloading

- *Method overloading* is the process of giving a single method name multiple definitions
- If a method is overloaded, the method name is not sufficient to determine which method is being called
- The *signature* of each overloaded method must be unique
- The signature includes the number, type, and order of the parameters

Method Overloading

- The compiler determines which method is being invoked by analyzing the parameters

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

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

Invocation

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

Method Overloading

- The `println` method is overloaded:

```
println (String s)
println (int i)
println (double d)
```

and so on...

- The following lines invoke different versions of the `println` method:

```
System.out.println ("The total is:");
System.out.println (total);
```

Overloading Methods

- The return type of the method is not part of the signature
- That is, overloaded methods cannot differ only by their return type
- Constructors can be overloaded
- Overloaded constructors provide multiple ways to initialize a new object

Overloading Methods

- Constructors can be overloaded
- An overloaded constructor provides multiple ways to set up a new object

- See SnakeEyes.java
- See Die.java

Snake Eyes

```
final int ROLLS = 500;
int snakeEyes = 0, num1, num2;

Die die1 = new Die();    // creates a six-sided die
Die die2 = new Die(20);  // creates a twenty-sided die

for (int roll = 1; roll <= ROLLS; roll++)
{
   num1 = die1.roll();
   num2 = die2.roll();

   if (num1 == 1 && num2 == 1)  // check for snake eyes
      snakeEyes++;
}

System.out.println ("Number of rolls: " + ROLLS);
System.out.println ("Number of snake eyes: " + snakeEyes);
System.out.println ("Ratio: " + (float)snakeEyes/ROLLS);
```

Die Class

```
public class Die {
   private final int MIN_FACES = 4;
   private int numFaces;   // number of sides on the die
   private int faceValue;  // current value showing on the die

   //  Defaults to a six-sided die. Initial face value is 1.
   public Die () {
      numFaces = 6;
      faceValue = 1;
   }
   //  Explicitly sets the size of the die. Defaults to a size of
   //  six if the parameter is invalid.  Initial face value is 1.
   public Die (int faces)  {
      if (faces < MIN_FACES)
         numFaces = 6;
      else
         numFaces = faces;
      faceValue = 1;
   }
```

Die Cont.

```
//-----------------------------------------------------------------
  //  Rolls the die and returns the result.
  //-----------------------------------------------------------------
  public int roll ()
  {
    faceValue = (int) (Math.random() * numFaces) + 1;
    return faceValue;
  }

  //-----------------------------------------------------------------
  //  Returns the current die value.
  //-----------------------------------------------------------------
  public int getFaceValue ()
  {
    return faceValue;
  }
}
```