CSC220 (CSI)
Computational Problem Solving

Object-Oriented Design

The College of New Jersey

Please turn off your cell phone!

Program Development

- The creation of software involves four basic activities:
  - establishing the requirements
    - what to do, not how to do it
  - creating a design
    - how a program will accomplish its requirements
  - implementing the code
    - the process of translating a design into source code
  - testing the implementation
- These activities are not strictly linear – they overlap and interact

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

- 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

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

- The `static` modifier associates the method or variable with the class rather than with an object of that class
- Static methods are sometimes called class methods
  - Invoked through the class name:
- Static variables are sometimes called class variables
  - 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 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

```
//********************************************
//  SloganCounter.java       Author: Lewis
//  Demonstrates the use of the static mod
//********************************************
public class SloganCounter {
   //-----------------------------------
   //  Creates several Slogan objects and
   //  objects that were created.
   //-----------------------------------
   public static void main(String[] args) {
      Slogan obj;
      obj = new Slogan("Remember the Alamo.");
      System.out.println(obj);
      obj = new Slogan("Don't Worry. Be Happy.");
      System.out.println(obj);
      obj = new Slogan("Live Free or Die.");
      System.out.println(obj);
      obj = new Slogan("Talk is Cheap.");
      System.out.println(obj);
      obj = new Slogan("Write Once, Run Anywhere.");
      System.out.println(obj);
      System.out.println();
      System.out.println("Slogans created: " + Slogan.getCount());
   }
}
```

Output

```
Remember the Alamo.
Don't Worry. Be Happy.
Live Free or Die.
Talk is Cheap.
Write Once, Run Anywhere.

Slogans created: 5
```

```
//********************************************************************
//  Slogan.java       Author: Lewis/Loftus
//  Represents a single slogan string.
//********************************************************************
public class Slogan{
   private String phrase;
   private static int count = 0;

   //  Constructor: Sets up the slogan and counts the number of
   //  instances created.
   public Slogan(String str){
      phrase = str;
      count++;
   }

   //  Returns this slogan as a string.
   public String toString(){
      return phrase;
   }

   //  Returns the number of instances of this class that have been
   //  created.
   public static int getCount(){
      return count;
   }
}
```

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
- Inheritance is discussed in detail in Chapter 9

Dependency

- A dependency exists when one class relies on another in some way, usually by invoking the methods of the other
- 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);
```

```
//********************************************
//  RationalTester.java       Author: Lewis
//  Driver to exercise the use of multiple
//********************************************
public class RationalTester{
   //  Creates some rational number objects
   //  operations on them.
   public static void main(String[] args)
      RationalNumber r1 = new RationalNumbe
      RationalNumber r2 = new RationalNumbe
      RationalNumber r3, r4, r5, r6, r7;
      System.out.println("First rational nu
      System.out.println("Second rational number: " + r2);
      if (r1.isLike(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);
   }
}
```

Output

```
First rational number: 3/4
Second rational number: 1/3
r1 and r2 are NOT equal.
The reciprocal of r1 is: 4/3
r1 + r2: 13/12
r1 - r2: 5/12
r1 * r2: 1/4
r1 / r2: 9/4
```

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

Continue

```
    //  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);
    }

    //  Subtracts the rational number passed as a parameter from this
    //  rational number.
    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);
    }
```

continue

continue

```
   //  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());
   }

   //  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;
      }
   }
```

continue

continue

```
    //  Determines if this rational number is equal to the one passed
    //  as a parameter. Assumes they are both reduced.
    public boolean isLike(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;
    }
    //  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;
    }
}
```

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
- An aggregate object contains references to other objects as instance data
- This is a special kind of dependency; the aggregate relies on the objects that compose it

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

public class StudentBody
{
   //-----------------------------------------------------
   //  Creates some Address and Student objects and prin
   //-----------------------------------------------------
   public static void main (String[] args)
   {
      Address school = new Address ("800 Lancaster Ave."
                                    "PA", 19085);
      Address jHome = new Address ("21 Jump Street", "Ly
                                   "VA", 24551);
      Student john = new Student ("John", "Smith", jHome

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

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

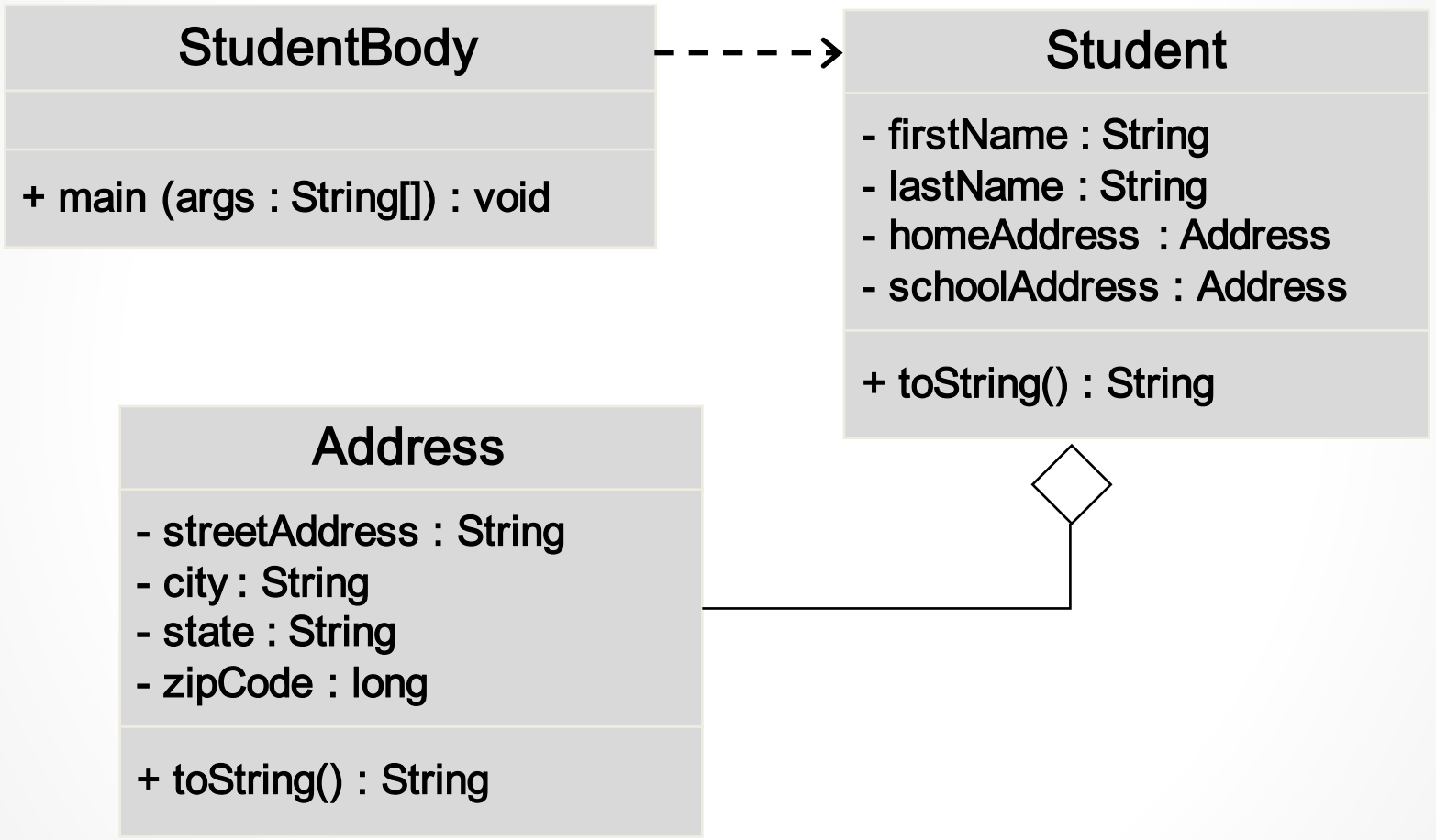
Output

```
John Smith
Home Address:
21 Jump Street
Lynchburg, VA  24551
School Address:
800 Lancaster Ave.
Villanova, PA  19085

Marsha Jones
Home Address:
123 Main Street
Euclid, OH  44132
School Address:
800 Lancaster Ave.
Villanova, PA  19085
```

```
//********************************************************************
//  Student.java       Author: Lewis/Loftus
//  Represents a college student.
//********************************************************************
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;
   }

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

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

      return result;
   }
}
```

```
//********************************************************************
//  Address.java       Author: Lewis/Loftus
//  Represents a street 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(){
      String result;

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

      return result;
   }
}
```

Aggregation in UML



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
- The `this` reference can be used to distinguish the instance variables of a class from corresponding method parameters with the same names

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

Interfaces

- A Java interface is a collection of abstract methods and constants
- An abstract method is a method header without a method body
  - An abstract method can be declared using the modifier `abstract`, but because all methods in an interface are abstract, usually it is left off
  - An interface is used to establish a set of methods that a class will implement

Interfaces

`interface` is a reserved word

None of the methods in an interface are given a definition (body)

```
public interface Doable
{
   public void doThis();
   public int doThat();
   public void doThis2(double value, char ch);
   public boolean doTheOther(int num);
}
```

A semicolon immediately follows each method header

Interfaces

- An interface cannot be instantiated
- Methods in an interface have public visibility by default
- A class formally implements an interface by:
  - stating so in the class header
  - providing implementations for every abstract method in the interface
- If a class declares that it implements an interface, it must define all methods in the interface

Interfaces

```
public class CanDo implements Doable
{
    public void doThis()
    {
        // whatever
    }

    public void doThat()
    {
        // whatever
    }

    // etc.
}
```

`implements` is a reserved word

Each method listed in `Doable` is given a definition

Interfaces

- In addition to (or instead of) abstract methods, an interface can contain constants
- When a class implements an interface, it gains access to all its constants
- A class that implements an interface can implement other methods as well

```
//********************************************************************
//  Complexity.java       Author: Lewis/Loftus
//
//  Represents the interface for an object that can be assigned an
//  explicit complexity.
//********************************************************************

public interface Complexity
{
   public void setComplexity(int complexity);
   public int getComplexity();
}
```

```
//********************************************************************
//  Question.java       Author: Lewis/Loftus
//  Represents a question (and its answer).
//********************************************************************
public class Question implements Complexity {
   private String question, answer;
   private int complexityLevel;

   //  Constructor: Sets up the question with a default complexity.
   public Question(String query, String result) {
      question = query;
      answer = result;
      complexityLevel = 1;
   }

   //  Sets the complexity level for this question.
   public void setComplexity(int level) {
      complexityLevel = level;
   }

   //  Returns the complexity level for this question.
   public int getComplexity() {
      return complexityLevel;
   }

   //  Returns the question.
   public String getQuestion() {
      return question;
   }
```

continue

```
    //  Returns the answer to this question.
    public String getAnswer(){
        return answer;
    }

    //  Returns true if the candidate answer matches the answer.
    public boolean answerCorrect(String candidateAnswer) {
        return answer.equals(candidateAnswer);
    }

    //  Returns this question (and its answer) as a string.
    public String toString(){
        return question + "\n" + answer;
    }
}
```

```
//********************************************************************
//  MiniQuiz.java       Author: Lewis/Loftus
//  Demonstrates the use of a class that implements an interface.
//********************************************************************
import java.util.Scanner;
public class MiniQuiz {
   //  Presents a short quiz.
   public static void main(String[] args){
      Question q1, q2;
      String possible;

      Scanner scan = new Scanner(System.in);
      q1 = new Question("What is the capital of Jamaica?",
                        "Kingston");
      q1.setComplexity(4);
      q2 = new Question("Which is worse, ignorance or apathy?",
                        "I don't know and I don't care");
      q2.setComplexity(10);
```

continue

continue

```
        System.out.print(q1.getQuestion());
        System.out.println(" (Level: " + q1.getComplexity() + ")");
        possible = scan.nextLine();
        if (q1.answerCorrect(possible))
           System.out.println("Correct");
        else
           System.out.println("No, the answer is " + q1.getAnswer());

        System.out.println();
        System.out.print(q2.getQuestion());
        System.out.println(" (Level: " + q2.getComplexity() + ")");
        possible = scan.nextLine();
        if (q2.answerCorrect(possible))
           System.out.println("Correct");
        else
           System.out.println("No, the answer is " + q2.getAnswer());
    }
}
```

Sample Run

```
What is the capital of Jamaica? (Level: 4)
Kingston
Correct

Which is worse, ignorance or apathy? (Level: 10)
apathy
No, the answer is I don't know and I don't care
```

Interfaces

- A class can implement multiple interfaces
- The interfaces are listed in the `implements` clause
- The class must implement all methods in all interfaces listed in the header

```
class ManyThings implements interface1, interface2
{
   // all methods of both interfaces
}
```

The Comparable Interface

- Any class can implement `Comparable` to provide a mechanism for comparing objects of that type

```
if (obj1.compareTo(obj2) < 0)
   System.out.println ("obj1 is less than obj2");
```

- The value returned from `compareTo` should be negative is `obj1` is less that `obj2`, 0 if they are equal, and positive if `obj1` is greater than `obj2`

- It's up to the programmer to determine what makes one object less than another

The Iterator Interface

- An iterator is an object that allows you to process a collection of items one at a time
- It lets you step through each item in turn and process it as needed
- An iterator is created formally by implementing the `Iterator` interface, which contains three methods
  - The `hasNext` method returns a boolean result – true if there are items left to process
  - The `next` method returns the next object in the iteration
  - The `remove` method removes the object most recently returned by the `next` method

Iterators

- Several classes in the Java standard class library are iterators
- The `Scanner` class is an iterator
  - the `hasNext` method returns true if there is more data to be scanned
  - the `next` method returns the next scanned token as a string
- The `Scanner` class also has variations on the `hasNext` method for specific data types (such as `hasNextInt`)

```
//*************************************************************
//  URLDissector.java       Au
//  Demonstrates the use of Sc
//  using alternative delimite
//*************************
import java.util.Scanner;
import java.io.*;
public class URLDissector {
   //  Reads urls from a file
   public static void main(Str
      String url;
      Scanner fileScan, urlSca
      fileScan = new Scanner(n
      // Read and process each
      while (fileScan.hasNext(
         url = fileScan.nextLi
         System.out.println("U

         urlScan = new Scanner
         urlScan.useDelimiter(

         //  Print each part o
         while (urlScan.hasNex
            System.out.println

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

Sample Run

```
URL: www.google.com
   www.google.com

URL: www.linux.org/info/gnu.html
   www.linux.org
   info
   gnu.html

URL: thelyric.com/calendar/
   thelyric.com
   calendar

URL: www.cs.vt.edu/undergraduate/about
   www.cs.vt.edu
   undergraduate
   about

URL: youtube.com/watch?v=EHCRimwRGLs
   youtube.com
   watch?v=EHCRimwRGLs
```

The Iterable Interface

- Another interface, `Iterable`, establishes that an object provides an iterator
- The `Iterable` interface has one method, `iterator`, that returns an `Iterator` object
- Any `Iterable` object can be processed using the for-each version of the `for` loop
- Note the difference: an `Iterator` has methods that perform an iteration; an `Iterable` object provides an iterator on request

Method Design

- As we've discussed, high-level design issues include:
  - identifying primary classes and objects
  - assigning primary responsibilities
- After establishing high-level design issues, its important to address low-level issues such as the design of key method
  - A method should be relatively small, so that it can be understood as a single entity
  - A potentially large method should be decomposed into several smaller methods as needed for clarity
  - A public service method of an object may call one or more private support methods to help it accomplish its goal
  - Support methods might call other support methods if appropriate

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)
- When an object is passed to a method, the actual parameter and the formal parameter become aliases of each other
- What a method does with a parameter may or may not have a permanent effect (outside the method)
- Note the difference between changing the internal state of an object versus changing which object a reference points to

```
//********************************************************************
//  ParameterTester.java       Author: Lewis/Loftus
//  Demonstrates the effects of passing various types of parameters.
//********************************************************************
public class ParameterTester{

   //  Sets up three variables (one primitive and two objects) to
   //  serve as actual parameters to the changeValues method. Prints
   //  their values before and after calling the method.
   public static void main(String[] args){
      ParameterModifier modifier = new ParameterModifier();

      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");

      modifier.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");
   }
}
```

```
//********************************************
//  ParameterModifier.java       Author: Le
//
//  Demonstrates the effects of changing pa
//********************************************

public class ParameterModifier
{
   //-----------------------------------
   //  Modifies the parameters, printing th
   //  after making the changes.
   //-----------------------------------
   public void changeValues(int f1, Num f2,
   {
      System.out.println("Before changing t
      System.out.println("f1\tf2\tf3");
      System.out.println(f1 + "\t" + f2 + "

      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");
   }
}
```

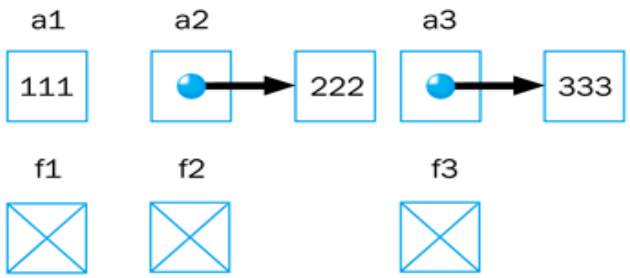
Output

```
Before calling changeValues:
a1  a2  a3
111 222 333

Before changing the values:
f1  f2  f3
111 222 333

After changing the values:
f1  f2  f3
999 888 777

After calling changeValues:
a1  a2  a3
111 888 333
```

```
//********************************************************************
//  Num.java        Author: Lewis/Loftus
//  Represents a single integer as an object.
//********************************************************************
public class Num {
   private int value;

   //  Sets up the new Num object, storing an initial value.
   public Num(int update) {
      value = update;
   }

   //  Sets the stored value to the newly specified value.
   public void setValue(int update) {
      value = update;
   }

   //  Returns the stored integer value as a string.
   public String toString(){
      return value + "";
   }
}
```

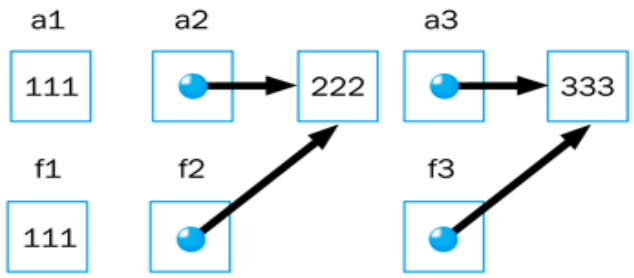
STEP 1

Before invoking `changeValues`



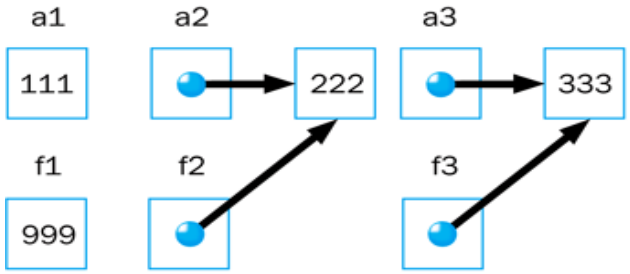
STEP 2

```
tester.changeValues (a1, a2, a3);
```



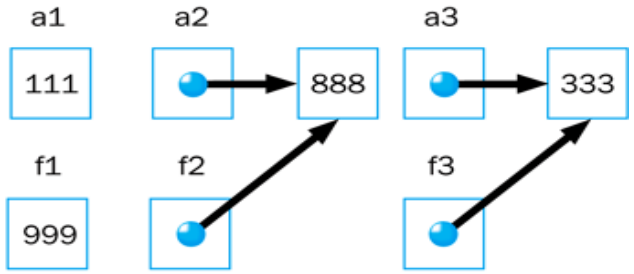
STEP 3

```
f1 = 999;
```



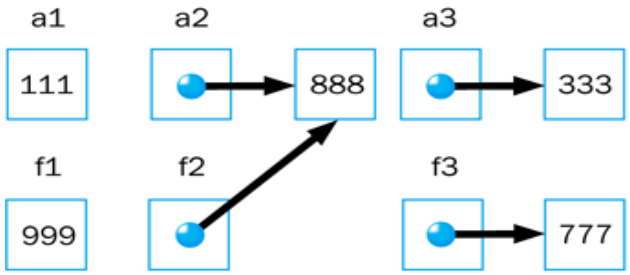
STEP 4

```
f2.setValue (888);
```



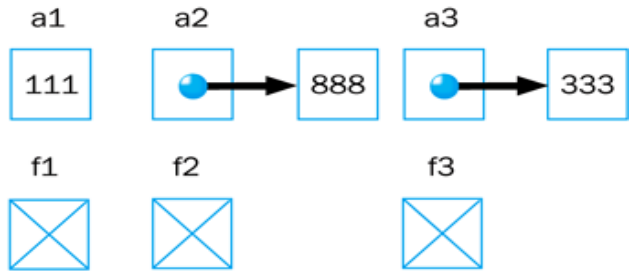
STEP 5

```
f3 = new Num (777);
```



STEP 6

After returning from `changeValues`



Method Overloading

- Method overloading is the process of giving a single method name multiple definitions in a class
- 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

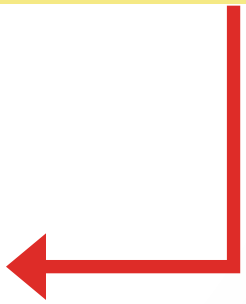
Method Overloading

- 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

Testing

- Testing can mean many different things
  - It includes running a completed program with various inputs
  - It includes any evaluation performed by human or computer to assess quality (Some evaluations should occur before coding even begins)
- The goal of testing is to find errors
- The earlier we find an problem, the easier and cheaper it is to fix

Testing

- As we find and fix errors, we raise our confidence that a program will perform as intended
- We can never really be sure that all errors have been eliminated
- So when do we stop testing?
  - Conceptual answer: Never
  - Cynical answer: When we run out of time
  - Better answer: When we are willing to risk that an undiscovered error still exists

Reviews

- A review is a meeting in which several people examine a design document or section of code
- It is a common and effective form of human-based testing
- Presenting a design or code to others:
  - makes us think more carefully about it
  - provides an outside perspective
- Reviews are sometimes called inspections or walkthroughs

Test Cases

- A test case is a set of input and user actions, coupled with the expected results
- Often test cases are organized formally into test suites which are stored and reused as needed
- For medium and large systems, testing must be a carefully managed process
- Many organizations have a separate Quality Assurance (QA) department to lead testing efforts

Defect and Regression Testing

- Defect testing is the execution of test cases to uncover errors
- The act of fixing an error may introduce new errors
- After fixing a set of errors we should perform regression testing – running previous test suites to ensure new errors haven't been introduced
- It is not possible to create test cases for all possible input and user actions
- Therefore we should design tests to maximize their ability to find problems

Black-Box Testing

- In black-box testing, test cases are developed without considering the internal logic
- They are based on the input and expected output
- Input can be organized into equivalence categories
- Two input values in the same equivalence category would produce similar results
- Therefore a good test suite will cover all equivalence categories and focus on the boundaries between categories

White-Box Testing

- White-box testing focuses on the internal structure of the code
- The goal is to ensure that every path through the code is tested
- Paths through the code are governed by any conditional or looping statements in a program
- A good testing effort will include both black-box and white-box tests

Summary

- software development activities
- determining the classes and objects that are needed for a program
- the relationships that can exist among classes
- the static modifier
- writing interfaces
- the design of enumerated type classes
- method design and method overloading
- testing