COP 3330: Object-Oriented Programming Summer 2007

Introduction to Classes – Part 3

Instructor : Mark Llewellyn
markl@cs.ucf.edu
HEC 236, 823-2790
http://www.cs.ucf.edu/courses/cop3330/sum2007

School of Electrical Engineering and Computer Science
University of Central Florida

One More Look At `this`

- The constructor methods in the class `Triple` are overloaded. There is a default constructor as well as a specific constructor.

- The default constructor is:

```
public Triple(){
        this(0, 0, 0);}
```

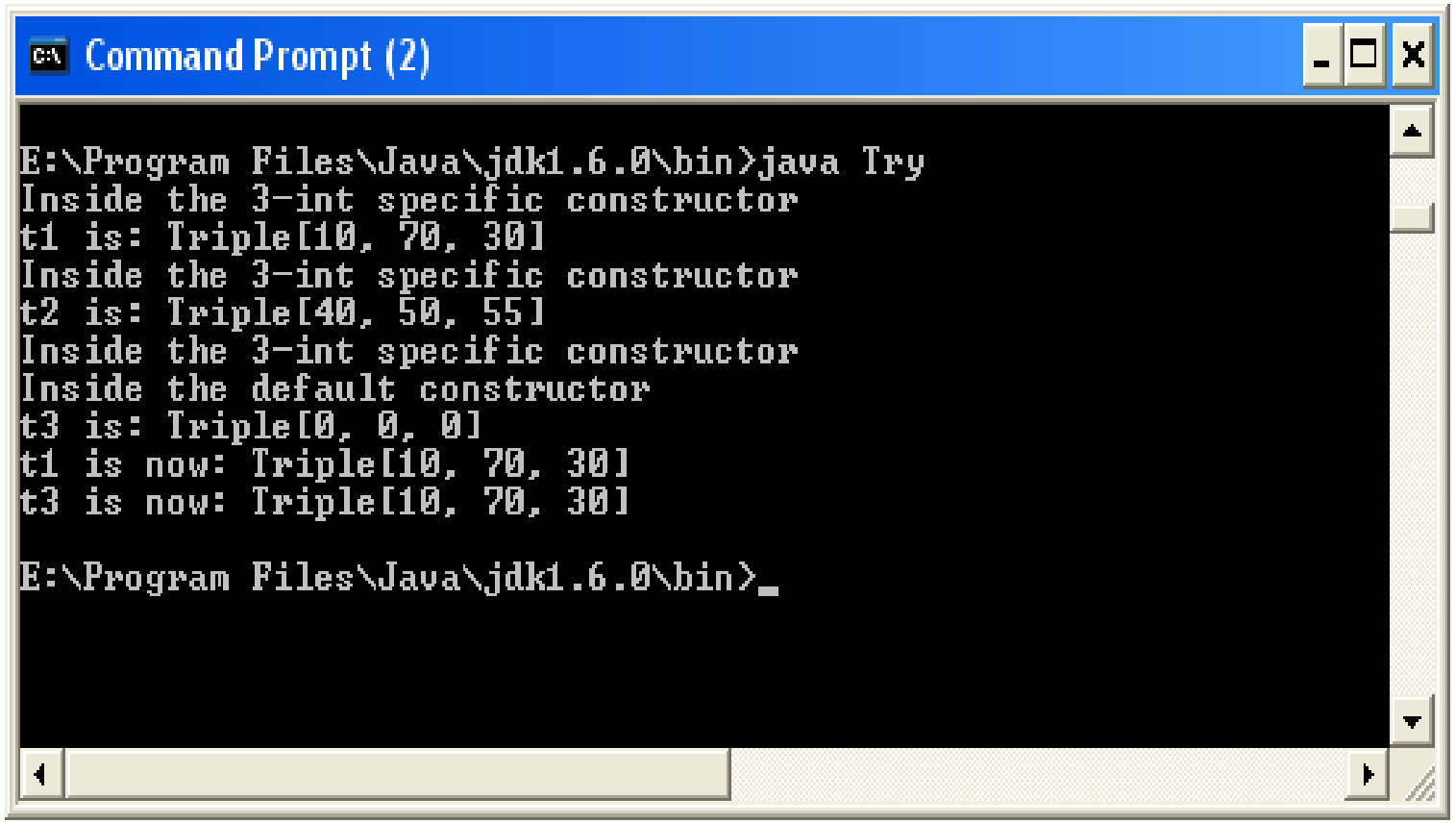
- The specific constructor is:

```
public Triple(int a, int b, int c) {
        setValue(1, a);
        setValue(2, b);
        setValue(3, c);
}
```


Example use of the `Triple` class

```
//Developer:  Mark Llewellyn          Date: June 2007
//Illustrate this and other Java features – uses Triple class
public class Try{
   public static void main(String[] args){
        Triple t1 = new Triple(10, 70, 30); //calls specific
   constructor
        System.out.println("t1 is: " + t1);
        Triple t2 = new Triple(40, 50, 55); //calls specific
   constructor
        System.out.println("t2 is: " + t2);
        Triple t3 = new Triple();  //calls default constructor
        System.out.println("t3 is: " + t3);
        t3 = t1;
        System.out.println("t1 is now: " + t1);
        System.out.println("t3 is now: " + t3);
   }
}
```


Output From class Try



Combinations-Permutations

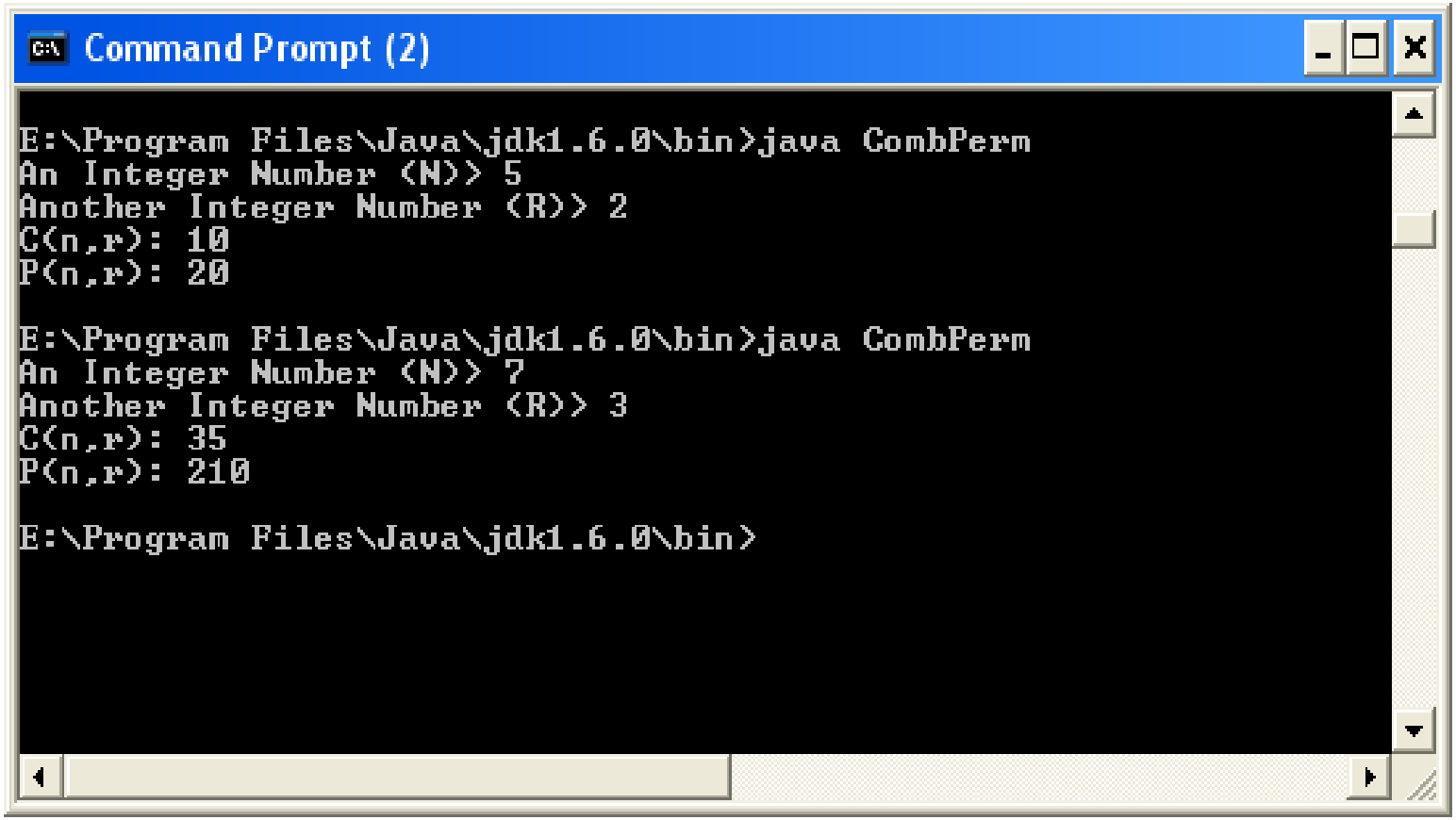
Parameter Passing Example

```
import java.io.*;
public class CombPerm {
   public static void main(String args[])throws IOException {
      int n,r,combVal,permVal;

      BufferedReader stdin =
         new BufferedReader (new InputStreamReader (System.in));

      System.out.print("An Integer Number (N)> ");
      System.out.flush();
      n = Integer.parseInt(stdin.readLine().trim());
      System.out.print("Another Integer Number (R)> ");
      System.out.flush();
      r = new Integer(stdin.readLine().trim()).intValue();

      // Calculate Combination and permutation
      combVal = comb(n,r);
      permVal = perm(n,r);

      System.out.println("C(n,r): " + combVal);
      System.out.println("P(n,r): " + permVal);
   }
```


Combination-Permutation Example (cont.)

```
   //    C(n,r) = n! / (r! * (n-r)!)
    static int comb(int n, int r) {
       return fact(n)/(fact(r)*fact(n-r));
    }
    //    P(n,r) = n! / (n-r)!
   static int perm(int n, int r) {
       return fact(n)/fact(n-r);
    }
    // Factorial
    static int fact(int n) {
       int i,val;
       val = 1; i=1;
       while (i<=n) {
          val = val*i;
          i = i + 1;
       }
       return val;
    }
}
```


Combination-Permutation Example Output



Keep Things Private

- When designing your own classes, keep in mind that instance variables are declared normally to be **`private.`** This convention requires that any access or modification to the instance variables by other classes be through the accessor and mutator methods.
- This information hiding principle helps ensure the integrity of the variables and also normally makes it possible to update or correct the class without requiring changes to other classes that use your class.

Method Overloading

- **Method overloading** is the process of using the same method name for multiple purposes.
  - Useful when you need to write methods that perform similar tasks but need to operate on different types or on different numbers of parameters.

- The signature of each overloaded method must be unique.
  - The signature of a method is based on the names, type, number, or order of the parameters (not return type).

Method Overloading (cont.)

- As an example of method overloading, the package java.lang.Math overloads the method name min() four times.
  - There are methods operating on int, long, float, and double parameters.
  - Their signatures are respectively:
    - min(int, int)
    - min(long, long)
    - min(float, float)
    - min(double, double)

Method Overloading (cont.)

- How does the compiler know which method to invoke?
  - There is no ambiguity as to which overloaded method is invoked. The compiler examines the parameter list of the method call, and invokes the method whose signature best matches the actual parameter list.
  - The process of determining which method to invoke is known as ***overloading resolution*** (also known as ***call resolution***).

Method Overloading (cont.)

- When method overloading, Java requires that the parameter lists for the methods be different. Therefore, the following class BadExample will not compile. The difference in the return types of the overloaded method is **not** a distinguishing factor in allowing the overloading.

```
public class BadExample {
        public double f(int a, int b){
                return 2.5 * (a + b);
        }
        public int f(int a, int b) { //illegal overload
                return (a + b);
        }
        public static void main(String[] args) {
                double x = f(2,4);
                int i = f(2,4);
        }
```


Method Overloading (cont.)

- The compiler must be able to determine which version of the method is invoked by analyzing the parameters of a method call.

- `println` is an overloaded method

```
println(String s)
  System.out.println("abcd");

println(int i)
  System.out.println(5);
```


Method Overloading (cont.)

- The constructors of the classes are often overloaded to provide multiple ways to set up a new object. Recall how this was done in the class `Triple` from Day 9.

```
class T {
    private int x,y;
    public T() { x=0; y=0; }
    public T(int v1, int v2) { x=v1; y=v2; }
}
```

In some other class or method:

```
T obj1 = new T();
T obj2 = new T(5,6);
```


Overloaded Methods

```
static void m(int x, int y) { System.out.println("m-i-i"); }
static void m() { System.out.println("m-noarg"); }
static void m(double x, double y) { System.out.println("m-d-d"); }
static void m(int x, double y) { System.out.println("m-i-d"); }
static void m(double x, int y) { System.out.println("m-d-i"); }
static void m(int x) { System.out.println("m-i"); }
```

To invoke this method

```
m(1,2);
m();
m(1.1,2.2);
m(1,2.2);
m(1.1,2);
m(1);
```


Overloaded Methods (cont.)

```
public class OverloadExample {
     static void m() { System.out.println("m-noarg"); }
     static void m(int x, int y) { System.out.println("m-i-i"); }
     static void m(double x, double y) { System.out.println("m-d-d"); }
     static void m(int x, double y) { System.out.println("m-i-d"); }
     static void m(double x, int y) { System.out.println("m-d-i"); }
     static void m(int x) { System.out.println("m-i"); }
     static void m(short x) { System.out.println("m-s"); }
     static void m(byte x) { System.out.println("m-b"); }
     public static void main(String args[]){
        System.out.print("m(1,2) -- "); m(1,2);
        System.out.print("m() -- "); m();
        System.out.print("m(1.1,2.2) -- "); m(1.1,2.2);
        System.out.print("m(1,2.2) -- "); m(1,2.2);
        System.out.print("m(1.1,2) -- "); m(1.1,2);
        System.out.print("m(1) -- "); m(1);
        System.out.print("m((byte)1) -- "); m((byte)1);
        System.out.print("m((short)1) -- "); m((short)1);
        System.out.print("m((int)1) -- "); m((int)1);
    }    }
```


RationalNum – Overloaded Constructors

```
class RationalNum {
  // Fields: a rational number is numerator/denominator
  private int numerator, denominator;
  // Constructors – Assume that parameters are positive
  public RationalNum(int n, int d) {
    int gcd = gcdivisor(n,d);
    numerator = n/gcd;
    denominator = d/gcd;
  }
  public RationalNum(int n) {
    numerator = n;
    denominator = 1;
  }
  public RationalNum() {
    numerator = 0;
    denominator = 1;
  }
```


RationalNum -- gcdivisor

```
 // gcdivisor -- finds the greatest common divisor of the given two integers
//
 private static int gcdivisor(int n1, int n2) {
   if (n1==0 && n2==0) return 1;
   else if (n1==0) return n2;
   else if (n2==0) return n1;
   else {   // they are not zero, Apply Euclid's algorithm for these positive numbers
     while (n1 != n2) {
       if (n1>n2) n1=n1-n2;
       else n2=n2-n1;
     }
     return n1;
   }
 }
```


RationalNum – add and subtract

// add method -- add the current rational number with another rational number or an integer.

```
public RationalNum add(RationalNum r2) {
  return(new
 RationalNum(numerator*r2.denominator+r2.numerator*denominator,
                          denominator*r2.denominator));
}
public RationalNum add(int n2) {
    return(new
 RationalNum(numerator+n2*denominator,denominator));
}
```

// subtract method -- subtract another rational number or an integer fromthe current rational number.

```
public RationalNum subtract(RationalNum r2) {
  return(new RationalNum(numerator*r2.denominator-
 r2.numerator*denominator, denominator*r2.denominator));
  }
public RationalNum subtract(int n2) {
    return(new RationalNum(numerator-
 n2*denominator,denominator));
}
```


RationalNum – compareTo and toString

```
// compareTo -- compare the current rational number with another rational number or an integer.
// returns 0 if they are equal; returns –1 if the current rational number is less than the given parameter;
// returns 1 otherwise
public int compareTo(RationalNum r2) {
  if (numerator*r2.denominator < r2.numerator*denominator)
   return -1;
  else if (numerator*r2.denominator ==
   r2.numerator*denominator)  return 0;
  else return 1;
}
public int compareTo(int n2) {
  if (numerator < n2*denominator)  return -1;
  else if (numerator == n2*denominator)  return 0;
  else return 1;
}
// toString method -- the string representation of a rational number is numerator/denominator.
public String toString() {
  return(numerator+"/"+denominator);
}
```


RationalNumTest - Example

```
// Demo class for RationalNum class
public class RationalNumTest {
  public static void main( String args[] ) throws IOException {
    RationalNum r1,r2;
    int n1,n2,d1,d2;
    BufferedReader stdin = new BufferedReader(new
   InputStreamReader(System.in));

    System.out.println("The value of new RationalNum(): " + (new
   RationalNum()));
    System.out.println("The value of new RationalNum(3): " + (new
   RationalNum(3)));
    System.out.println("The value of new RationalNum(4,6): " + (new
   RationalNum(4,6)));

    System.out.print("Enter the first numerator: ");   System.out.flush();
    n1 = Integer.parseInt(stdin.readLine().trim());
    System.out.print("Enter the first denominator: "); System.out.flush();
    d1 = Integer.parseInt(stdin.readLine().trim());

    System.out.print("Enter the second numerator: ");   System.out.flush();
    n2 = Integer.parseInt(stdin.readLine().trim());
    System.out.print("Enter the second denominator: "); System.out.flush();
    d2 = Integer.parseInt(stdin.readLine().trim());
```

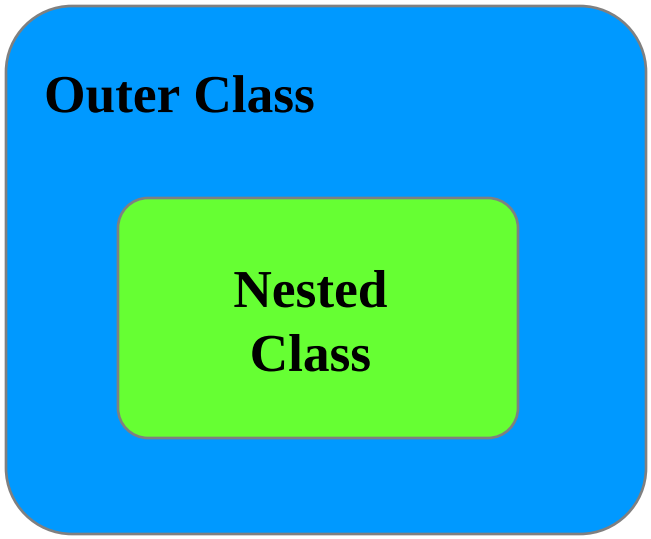

RationalNumTest (cont.)

```
    r1 = new RationalNum(n1,d1);        r2 = new
   RationalNum(n2,d2);

    System.out.println("r1: " + r1);   System.out.println("r2:
   " + r2);

    System.out.println("r1.add(r2): " + r1.add(r2));
    System.out.println("r1.add(3): " + r1.add(3));

    System.out.println("r1.subtract(r2): " + r1.subtract(r2));
    System.out.println("r1.subtract(3): " + r1.subtract(3));

    System.out.println("r1.compareTo(r2): " +
   r1.compareTo(r2));
    System.out.println("r1.compareTo(3): " + r1.compareTo(3));
  }
}
```


Nested Classes

- In addition to a class containing data and methods, it can also contain other classes

- A class declared within another class is called a *nested class*



Nested Classes (cont.)

- A nested class has access to the variables and methods of the outer class, even if they are declared private.
- In certain situations this makes the implementation of the classes easier because they can easily share information.
- Furthermore, the nested class can be protected by the outer class from external use.
- This is a special relationship and should be used with care.

Nested Classes (cont.)

- A nested class produces a separate bytecode file
- If a nested class called Inside is declared in an outer class called Outside, two bytecode files will be produced:

  Outside.class

  Outside$Inside.class

- Nested classes can be declared as static, in which case they cannot refer to instance variables or methods
- A nonstatic nested class is called an *inner class*

Two Classes-not nested

```
public class LL {


}

class Node {


}
```


Two Classes - nested

```
public class LL {

    static protected class Node {

    }

}
```


Time -- Example

```
class Time {
  private int hour, minute;
  public Time (int h, int m) { hour = h; minute = m; }
  public void printTime () {
    if ((hour == 0) && (minute == 0))
      System.out.print("midnight");
    else if ((hour == 12) && (minute == 0))
      System.out.print("noon");
    else {
      if      (hour == 0) System.out.print(12);
      else if (hour > 12) System.out.print(hour-12);
      else                System.out.print(hour);

      if (minute < 10) System.out.print(":0" + minute);
      else             System.out.print(":" + minute);

      if (hour < 12) System.out.print("AM");
      else           System.out.print("PM");
    }
  }
```


Time – Example (cont.)

```
public Time addMinutes (int m) {
    int totalMinutes = (60*hour + minute + m) % (24*60);
    if (totalMinutes < 0)
      totalMinutes = totalMinutes + 24*60;
    return new Time(totalMinutes/60, totalMinutes%60);
  }

public Time subtractMinutes (int m) { return addMinutes(-m); }

public boolean priorTo (Time t) {
    return ((hour < t.hour) || ((hour == t.hour) && (minute <
   t.minute)));
  }

public boolean after (Time t2) { return t2.priorTo(this); }

}
```


Time – Example (cont.)

```
public class Time2 {
  public static void main (String[] args) {
    Time t1 = new Time(0,0),
         t2 = new Time(12,0),
         t3 = new Time(8,45),
         t4 = new Time(14,14);

     System.out.print("midnight - "); t1.printTime();
    System.out.println();
     System.out.print("noon - ");     t2.printTime();
    System.out.println();
     System.out.print("8:45AM - ");   t3.printTime();
    System.out.println();
     System.out.print("2:14PM - ");   t4.printTime();
    System.out.println();

     t1 = t1.addMinutes(4*60);
     System.out.print("4:00AM - ");   t1.printTime();
    System.out.println();

     t1 = t1.addMinutes(-2*60);
     System.out.print("2:00AM - ");   t1.printTime();
    System.out.println();
```


Time – Example (cont.)

```
        t1 = t1.addMinutes(-6);
        System.out.print("1:54AM - ");  t1.printTime();
        System.out.println();
        t1 = t1.addMinutes(-2*60);
        System.out.print("11:54PM - "); t1.printTime();
        System.out.println();
        t1 = t1.subtractMinutes(8);
        System.out.print("11:46PM - "); t1.printTime();
        System.out.println();
        t1 = t1.subtractMinutes(24*60);
        System.out.print("11:46PM - "); t1.printTime();
        System.out.println();

         System.out.println("true - " +  t1.priorTo(new Time(23, 47)));
         System.out.println("false - " + t1.priorTo(new Time(3, 47)));

         System.out.println("true - " +  (new Time(23, 47)).after(t1));
         System.out.println("false - " + (new Time(3, 47)).after(t1));
      }
   }
```


Time Example – Output