CSC 221: Computer Programming I

Fall 2005

Class design & strings

- design principles
- cohesion & coupling
- class examples: Hurricane, TaxReturn
- objects vs. primitives
- String methods
- class example: Bounce

1

Object-oriented design principles so far:

- a **class** should model some entity, encapsulating all of its state and behaviors
  e.g., `Circle, Die, Dot, DotRace, …`
- a **field** should store a value that is part of the state of an object (and which must persist between method calls)
  e.g., `xPosition, yPosition, color, diameter, isVisible, …`
  - can be primitive (e.g., `int, double`) or object (e.g., `String, Die`) type
  - should be declared private to avoid outside tampering with the fields – provide public accessor methods if needed
  - static fields should be used if the data can be shared among all objects
  - final-static fields should be used to define constants with meaningful names
- a **constructor** should initialize the fields when creating an object
  - can have more than one constructor with different parameters to initialize differently
- a **method** should implement one behavior of an object
  e.g., `moveLeft, moveRight, draw, erase, changeColor, …`
  - should be declared public to make accessible – helper methods can be private
  - local variables should be used to store temporary values that are needed
  - if statements should be used to perform conditional execution

2

Cohesion

cohesion describes how well a unit of code maps to an entity or behavior

in a highly cohesive system:

- each class maps to a single, well-defined entity – encapsulating all of its internal state and external behaviors
- each method of the class maps to a single, well-defined behavior

advantages of cohesion:

- highly cohesive code is easier to read
  - don't have to keep track of all the things a method does
  - if the method name is descriptive, it makes it easy to follow code
- highly cohesive code is easier to reuse
  - if the class cleanly models an entity, can reuse it in any application that needs it
  - if a method cleanly implements a behavior, it can be called by other methods and even reused in other classes

3

Coupling

coupling describes the interconnectedness of classes

in a loosely coupled system:

- each class is largely independent and communicates with other classes via small, well-defined interfaces

advantages of loose coupling:

- loosely coupled classes make changes simpler
  - can modify the implementation of one class without affecting other classes
  - only changes to the interface (e.g., adding/removing methods, changing the parameters) affect other classes
- loosely coupled classes make development easier
  - you don't have to know how a class works in order to use it
  - since fields/local variables are encapsulated within a class/method, their names cannot conflict with the development of other classes.methods

4

Class example: Hurricane

suppose we wanted to enter and access information on a hurricane

- create a `Hurricane` object with a name
- enter a wind speed reading
- get the hurricane's name
- get the last wind speed reading
- get the highest wind speed reading
- get the current category level for the hurricane

state? fields?

constructor?

methods?

5

Hurricane class

```
public class Hurricane
{
    private String officialName;
    private int lastWindSpeed;
    private int highestWindSpeed;

    public Hurricane(String name)
    {
        officialName = name;
        lastWindSpeed = 0;
        highestWindSpeed = 0;
    }

    public String getName()
    {
        return officialName;
    }

    public int getLastWindSpeed()
    {
        return lastWindSpeed;
    }

    public int getHighestWindSpeed()
    {
        return highestWindSpeed;
    }
```

```
    public void enterReading(int currentWindSpeed)
    {
        lastWindSpeed = currentWindSpeed;
        if (currentWindSpeed > highestWindSpeed) {
            highestWindSpeed = currentWindSpeed;
        }
    }

    public int category()
    {
        if (lastWindSpeed > 155) {
            return 5;
        }
        else if (lastWindSpeed > 130) {
            return 4;
        }
        else if (lastWindSpeed > 110) {
            return 3;
        }
        else if (lastWindSpeed > 95) {
            return 2;
        }
        else if (lastWindSpeed > 73) {
            return 1;
        }
        else {
            return 0;        // ???
        }
    }
}
```

cohesive?

6

Hurricane class

again, highly cohesive classes tend to be easier to modify

- if we wanted to add a method for predicting damage, we don't have to think in terms of wind speed anymore
- instead, can build upon the `category` method

```
public String expectedDamage()
{
    if (category() == 5) {
        return "Complete roof failure on many residences; " +
               "some complete building failures.";
    }
    else if (category() == 4) {
        return "Some complete roof structure failure on small residences.";
    }
    else if (category() == 3) {
        return "Some structural damage to small residences; " +
               "mobile homes are destroyed.";
    }
    else if (category() == 2) {
        return "Some roofing material, door, and window damage; " +
               "considerable damage to mobile homes.";
    }
    else if (category() == 1) {
        return "No real damage to building structures; " +
               "some damage to unanchored mobile homes.";
    }
    else {
        return "There is no hurricane condition.";
    }
}
```

7

Class example: TaxReturn

in the text, a class is developed for calculating a person's 1992 income tax

If your filing status is Single

If the taxable income is over	But not over	The tax is	Of the amount over
$0	$21,450	15%	$0
$21,450	$51,900	$3,217.50 + 28%	$21,450
$51,900		$11,743.50 + 31%	$51,900

If your filing status is Married

If the taxable income is over	But not over	The tax is	Of the amount over
$0	$35,800	15%	$0
$35,800	$86,500	$5,370.00 + 28%	$35,800
$86,500		$19,566.00 + 31%	$86,500

8

TaxReturn class

the cutoffs and tax rates are magic numbers

- represent as constants

fields are needed to store income and marital status

- here, marital status is represented using a constant
- the user can enter a number OR the constant name

```
class TaxReturn
{
    private static final double RATE1 = 0.15;
    private static final double RATE2 = 0.28;
    private static final double RATE3 = 0.31;

    private static final double SINGLE_CUTOFF1 = 21450;
    private static final double SINGLE_CUTOFF2 = 51900;
    private static final double SINGLE_BASE2 = 3217.50;
    private static final double SINGLE_BASE3 = 11743.50;

    private static final double MARRIED_CUTOFF1 = 35800;
    private static final double MARRIED_CUTOFF2 = 86500;
    private static final double MARRIED_BASE2 = 5370;
    private static final double MARRIED_BASE3 = 19566;

    public static final int SINGLE = 1;
    public static final int MARRIED = 2;

    private int status;
    private double income;

    /**
     * Constructs a TaxReturn object for a given income and marital status.
     *   @param anIncome the taxpayer income
     *   @param aStatus either TaxReturn.SINGLE or TaxReturn.MARRIED
     */
    public TaxReturn(double anIncome, int aStatus)
    {
       income = anIncome;
       status = aStatus;
    }

    . . .
```

9

TaxReturn class

the `getTax` method first tests to determine if SINGLE or MARRIED

then tests to determine the tax bracket and calculate the tax

- note: could have just returned the tax in each case instead of assigning to a variable

cohesive?

```
    . . .

    /**
     * Calculates the tax owed by the filer.
     *   @return the amount (in dollars) owed
     */
    public double getTax()
    {
        double tax = 0;

        if (status == SINGLE) {
           if (income <= SINGLE_CUTOFF1) {
               tax = RATE1 * income;
           }
           else if (income <= SINGLE_CUTOFF2) {
               tax = SINGLE_BASE2 + RATE2 * (income - SINGLE_CUTOFF1);
           }
           else {
               tax = SINGLE_BASE3 + RATE3 * (income - SINGLE_CUTOFF2);
           }
        }
        else if (income <= MARRIED_CUTOFF1) {
            tax = RATE1 * income;
        }
        else if (income <= MARRIED_CUTOFF2) {
            tax = MARRIED_BASE2 + RATE2 * (income - MARRIED_CUTOFF1);
        }
        else {
            tax = MARRIED_BASE3 + RATE3 * (income - MARRIED_CUTOFF2);
        }

        return tax;
    }
}
```

10

Unknown status

QUESTION: what if the user entered 3 for marital status?

error?

result?

```
   . . .

   public static final int SINGLE = 1;
   public static final int MARRIED = 2;

   public TaxReturn(double anIncome, int aStatus)
   {
      income = anIncome;
      status = aStatus;
   }

   public double getTax()
   {
       double tax = 0;

       if (status == SINGLE) {
          if (income <= SINGLE_CUTOFF1) {
              tax = RATE1 * income;
          }
          else if (income <= SINGLE_CUTOFF2) {
              tax = SINGLE_BASE2 + RATE2 * (income - SINGLE_CUTOFF1);
          }
          else {
              tax = SINGLE_BASE3 + RATE3 * (income - SINGLE_CUTOFF2);
          }
       }
       else if (income <= MARRIED_CUTOFF1) {
           tax = RATE1 * income;
       }
       else if (income <= MARRIED_CUTOFF2) {
           tax = MARRIED_BASE2 + RATE2 * (income - MARRIED_CUTOFF1);
       }
       else {
           tax = MARRIED_BASE3 + RATE3 * (income - MARRIED_CUTOFF2);
       }

       return tax;
   }
}
```

11

Checking the status

could add an extra test to make sure status is 1 (SINGLE) or 2 (MARRIED)

- what is returned if status == 3?

```
   . . .

   /**
    * Calculates the tax owed by the filer.
    *   @return the amount (in dollars) owed
    */
   public double getTax()
   {
       double tax = 0;

       if (status == SINGLE)
       {
          if (income <= SINGLE_CUTOFF1) {
              tax = RATE1 * income;
          }
          else if (income <= SINGLE_CUTOFF2) {
              tax = SINGLE_BASE2 + RATE2 * (income - SINGLE_CUTOFF1);
          }
          else {
              tax = SINGLE_BASE3 + RATE3 * (income - SINGLE_CUTOFF2);
          }
       }
       else if (status == MARRIED) {
           if (income <= MARRIED_CUTOFF1) {
               tax = RATE1 * income;
           }
           else if (income <= MARRIED_CUTOFF2) {
               tax = MARRIED_BASE2 + RATE2 * (income - MARRIED_CUTOFF1);
           }
           else {
               tax = MARRIED_BASE3 + RATE3 * (income - MARRIED_CUTOFF2);
           }
       }

       return tax;
   }
}
```

12

A nicer version?

suppose we wanted to allow the user to enter a word for marital status

will this work?

not quite – you can't use == on Strings

WHY?

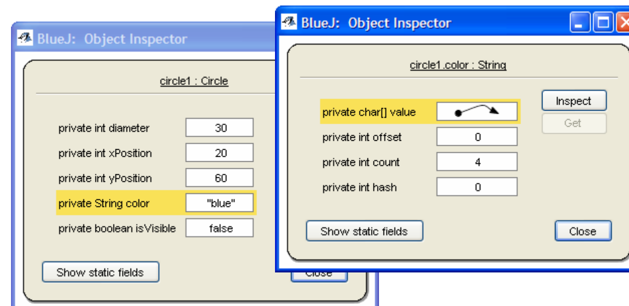
```
   ...

   private String status;
   private double income;

   /**
    * Constructs a TaxReturn object for a given income and status.
    *   @param anIncome the taxpayer income
    *   @param aStatus either "single" or "married"
    */
   public TaxReturn(double anIncome, String aStatus)
   {
      income = anIncome;
      status = aStatus;
   }

   /**
    * Calculates the tax owed by the filer.
    *   @return the amount (in dollars) owed
    */
   public double getTax()
   {
       double tax = 0;

       if (status == "single") {
          . . .
       }
       else if (status == "married") {
          . . .
       }

       return tax;
   }
}
```

13

Strings vs. primitives

although they behave similarly to primitive types (int, double, char, boolean), Strings are different in nature

- String is a class that is defined in a separate library: `java.lang.String`
  → a String value is really an object
- you can call methods on a String
- also, you can *Inspect* the String fields of an object



14

Comparing strings

comparison operators (`<` `<=` `>` `>=`) are defined for primitives but not objects

```
String str1 = "foo";      // EQUIVALENT TO String str1 = new String("foo");
String str2 = "bar";      // EQUIVALENT TO String str2 = new String("bar");
if (str1 < str2) …        // ILLEGAL
```

`==` and `!=` are defined for objects, but don't do what you think

```
if (str1 == str2) …                  // TESTS WHETHER THEY ARE THE
                                     // SAME OBJECT, NOT WHETHER THEY
                                     // HAVE THE SAME VALUE!
```

Strings are comparable using the `equals` and `compareTo` methods

```
if (str1.equals(str2)) …             // true IF THEY REPRESENT THE
                                     // SAME STRING VALUE

if (str1.compareTo(str2) < 0) …      // RETURNS -1 if str1 < str2
                                     // RETURNS  0 if str1 == str2
                                     // RETURNS  1 if str1 > str2
```

15

A nicer version

to test whether two Strings are the same, use the equals method

- what is returned if status == "Single" ?

```
    . . .

    private String status;
    private double income;

    /**
     * Constructs a TaxReturn object for a given income and status.
     *   @param anIncome the taxpayer income
     *   @param aStatus either "single" or "married"
     */
    public TaxReturn(double anIncome, String aStatus)
    {
       income = anIncome;
       status = aStatus;
    }

    /**
     * Calculates the tax owed by the filer.
     *   @return the amount (in dollars) owed
     */
    public double getTax()
    {
        double tax = 0;

        if (status.equals("single")) {
           . . .
        }
        else if (status.equals("married")) {
           . . .
        }

        return tax;
    }
}
```

16

String methods

many methods are provided for manipulating Strings

Method	Description
`boolean equals(String other)`	returns true if other String has same value
`int compareTo(String other)`	returns -1 if less than other String, 0 if equal to other String, 1 if greater than other String
`int length()`	returns number of chars in String
`char charAt(int index)`	returns the character at the specified index (indices range from 0 to str.length()-1)
`int indexOf(char ch)` `int indexOf(String str)`	returns index where the specified char/substring first occurs in the String (-1 if not found)
`String substring(int start, int end)`	returns the substring from indices start to (end-1)
`String toUpperCase()`	returns copy of String with all letters uppercase
`String toLowerCase()`	returns copy of String with all letters lowercase

17

An even nicer version

we would like to allow a range of valid status entries

- "s" or "m"
- "S" or "M"
- "single" or "married"
- "Single" or "Married"
- "SINGLE" or "MARRIED"
- . . .

to be case-insensitive

- make the status lowercase when constructing

to handle first letter only

- use `charAt` to extract the char at index 0

```
   . . .

   private String status;
   private double income;

   /**
    * Constructs a TaxReturn object for a given income and status.
    *   @param anIncome the taxpayer income
    *   @param aStatus either "single" or "married"
    */
   public TaxReturn(double anIncome, String aStatus)
   {
      income = anIncome;
      status = aStatus.toLowerCase();
   }

   /**
    * Calculates the tax owed by the filer.
    *   @return the amount (in dollars) owed
    */
   public double getTax()
   {
       double tax = 0;

       if (status.charAt(0) == 's') {
           . . .
       }
       else if (status.charAt(0) == 'm') {
           . . .
       }

       return tax;
   }
}
```

18

Class example: Bounce

suppose we want to animate a circle, so that it bounces around the `Canvas`

- since `Canvas` and `Circle` are highly cohesive, it greatly simplifies the task

✓ need to create a class with a `Circle` as field

✓ at each step, move the `Circle` a set amount along a trajectory
  - need to recognize when the `Circle` reaches a wall
  - if so, change the trajectory so that it bounces off the wall

✓ to avoid having the user specify each step, we want to automate the repetition

all movements will utilize well-defined, public methods of the `Circle` class

- the implementation of `Bounce` is separate from the implementation of `Circle`
- as long as the required methods are still implemented, changes can be made to the `Circle` class without affecting `Bounce`
- thus, they are *loosely coupled*

19

Bounce class

the changes in X & Y positions are defined by `xDelta` & `yDelta`

- each is initialized to be a random int between ±`DELTA_RANGE`
- at each step, the X & Y positions of the ball are updated by `xDelta` and `yDelta`, resp.
- note: need to stop at a window edge and reverse the delta

note: the `go` method calls itself to produce (uncontrolled) repetition!

```
public class Bounce
{
    private static final int WINDOW_SIZE = 300;
    private static final int DELTA_RANGE = 5;

    private int xDelta;
    private int yDelta;
    private Circle ball;

    public Bounce()
    {
        ball = new Circle();
        ball.makeVisible();

        xDelta = (int)((2*DELTA_RANGE+1)*Math.random())-DELTA_RANGE;
        yDelta = (int)((2*DELTA_RANGE+1)*Math.random())-DELTA_RANGE;
    }

    public void go()
    {
        int newX = Math.max(0, Math.min(ball.getXPosition()+xDelta,
                                        WINDOW_SIZE-ball.getSize()));
        if (newX == 0 || newX == WINDOW_SIZE-ball.getSize()) {
            xDelta = -xDelta;
        }

        int newY = Math.max(0, Math.min(ball.getYPosition()+yDelta,
                                        WINDOW_SIZE-ball.getSize()));
        if (newY == 0 || newY == WINDOW_SIZE-ball.getSize()) {
            yDelta = -yDelta;
        }

        ball.moveTo(newX, newY);

        go();       // SCARY! BUT OK FOR NOW
    }
}
```

20