CSC 221: Computer Programming I

Fall 2006

Class design & strings

- design principles
- cohesion & coupling
- class example: graphical DotRace
- static (non-final) fields
- class example: TaxReturn
- objects vs. primitives
- String methods

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

Recall our Dot class

in the final version:

- a constant represented the maximum step size
- the die field was static to avoid unnecessary duplication

```
public class Dot {
    private static final int MAX_STEP = 5;
    private static Die die = new Die(Dot.MAX_STEP);

    private String dotColor;
    private int dotPosition;

    public Dot(String color) {
        this.dotColor = color;
        this.dotPosition= 0;
    }

    public int getPosition() {
        return this.dotPosition;
    }

    public void step() {
        this.dotPosition += Dot.die.roll();
    }

    public void reset() {
        this.dotPosition = 0;
    }

    public void showPosition() {
        System.out.println(this.dotColor + ": " +
                           this.dotPosition);
    }
}
```

5

Recall our DotRace class

- added a finish line to the race
- stored as a field the goal distance as a field and provided an accessor method
- modified the step method to check if either Dot has finished

```
public class DotRace {
    private Dot redDot;
    private Dot blueDot;
    private int goalDist;

    public DotRace(int distance) {
        this.redDot = new Dot("red");
        this.blueDot = new Dot("blue");
        this.goalDist = distance;
    }

    public int getRedPosition() {
        return this.redDot.getPosition();
    }

    public int getBluePosition() {
        return this.blueDot.getPosition();
    }

    public void step() {
        if (this.getRedPosition() >= this.goalDist && this.getBluePosition() >= this.goalDist) {
           System.out.println("The race is over: it's a tie.");
        }
        else if (this.getRedPosition() >= this.goalDist) {
           System.out.println("The race is over: RED wins!");
        }
        else if (this.getBluePosition() >= this.goalDist) {
           System.out.println("The race is over: BLUE wins!");
        }
        else {
            this.redDot.step();
            this.blueDot.step();
        }
    }

    public void showStatus() {
        this.redDot.showPosition();
        this.blueDot.showPosition();
    }

    public void reset() {
        this.redDot.reset();
        this.blueDot.reset();
    }
}
```

QUESTION: can we make it so that the status is automatically shown after every move?

6

Adding graphics

what if we wanted to display the dot race visually?

- we could utilize the `Circle` class to draw the dots
- recall: `Circle` has methods for relative movements
  `moveLeft(), moveRight(), moveUp(), moveDown()`
  `moveHorizontal(dist), slowMoveHorizontal(dist)`
  `moveVertical(dist), slowMoveVertical(dist)`
- in principle, each step of size `N` can be drawn by calling `slowMoveHorizontal(N)`

PROBLEM: the way that `Dot` is designed, there can be numerous steps in between each call to `showPosition`

- keep track of the total step distance covered since the last call to `showPosition`
- when `showPosition` is called, move the `dotImage` that distance and reset
- e.g.,

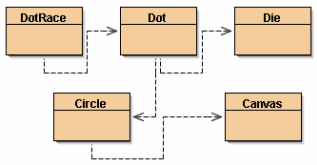
```
redDot.step();              → at position 3,   3 since last show
redDot.showPosition();      → move 3 pixels,   reset
redDot.step();              → at position 7,   4 since last show
redDot.step();              → at position 8,   5 since last show
redDot.showPosition();      → move 5 pixels,   reset
```

7

Adding graphics

due to our modular design, changing the display is easy

- each `Dot` object will maintains and display its own `Circle` image



- add `distanceToDraw` field to store distance covered since last show
- `showPosition` moves the `Circle` image (instead of displaying text)

```
public class Dot {
    private static final int MAX_STEP = 5;
    private static Die die = new Die(Dot.MAX_STEP);

    private String dotColor;
    private int dotPosition;
    private Circle dotImage;
    private int distanceToDraw;

    public Dot(String color) {
        this.dotColor = color;
        this.dotPosition= 0;
        this.dotPosition = 0;

        this.dotImage = new Circle();
        this.dotImage.changeColor(color);
        this.dotImage.makeVisible();
        this.distanceToDraw = 0;
    }

    public int getPosition() {
        return this.dotPosition;
    }

    public void step() {
        int distance = Dot.die.roll();
        this.dotPosition += distance;
        this.distanceToDraw += distance;
    }

    public void reset() {
        this.dotImage.moveHorizontal(-this.dotPosition);
        this.distanceToDraw = 0;
        this.dotPosition = 0;
    }

    public void showPosition() {
        this.dotImage.moveHorizontal(this.distanceToDraw);
        this.distanceToDraw = 0;
    }
}
```

8

Graphical display

note: no modifications are necessary in the `DotRace` class!!!

- this shows the benefit of modularity
- not only is modular code easier to write, it is easier to change/maintain
- can isolate changes/updates to the class/object in question
- to any other interacting classes, the methods look the same

EXERCISE: make these modifications to the `Dot` class

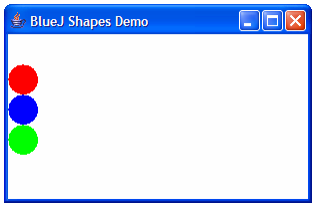
- add `dotImage` and `distanceToDraw` fields
- modify the constructor to initialize `dotImage` and `distanceToDraw` appropriately
- modify `step`, `reset`, and `showPosition` to update `dotImage` and `distanceToDraw` appropriately

9

Better graphics

the graphical display is better than text, but still primitive

- dots are drawn on top of each other
- would be nicer to have the dots aligned vertically



PROBLEM: each dot maintains its own state & displays itself

- thus, each dot will need to know how far to move vertically
- but vertical distance depends on what order the dots are created
  (e.g., 1st dot moves 0, 2nd dot moves diameter, 3rd dot moves 2*diameter, ...)
- *how can an individual dot know this?*

10

Option 1: parameters

we could alter the `Dot` class constructor

- takes an additional `int` that specifies the dot number
- the `dotNumber` can be used to determine the vertical offset
- in `DotRace`, must pass in the number when creating each of the dots

```
public class Dot {
    private static final int DIAMETER = 50;
    private static final int MAX_STEP = 5;
    private static Die die = new Die(Dot.MAX_STEP);

    private String dotColor;
    private int dotPosition;
    private Circle dotImage;
    private int distanceToDraw;

    public Dot(String color, int dotNumber) {
        this.dotColor = color;
        this.dotPosition= 0;
        this.dotPosition = 0;

        this.dotImage = new Circle();
        this.dotImage.changeColor(color);
        this.dotImage.changeSize(Dot.DIAMETER);
        this.dotImage.moveVertical(Dot.DIAMETER*(dotNumber-1));
        this.dotImage.makeVisible();
    }
    . . .
```

```
public class DotRace {
    private Dot redDot;
    private Dot blueDot;
    private Dot greenDot;
    private int goalDist;

    public DotRace(int distance)
    {
        redDot = new Dot("red", 1);
        blueDot = new Dot("blue", 2);
        greenDot = new Dot("green", 3);
    }
    . . .
```

this works, but is inelegant

- why should `DotRace` have to worry about dot numbers?
- the `Dot` class should be responsible

11

Option 2: a static (non-final) field

better solution: have each dot keep track of its own number

- this can be accomplished with a *static field*
- when the first object of that class is created, the field is initialized via the assignment
- subsequent objects simply access/update the existing field

```
public class Dot {
    private static final int DIAMETER = 50;
    private static final int MAX_STEP = 5;
    private static Die die = new Die(Dot.MAX_STEP);
    private static int nextNumber = 1;

    private String dotColor;
    private int dotPosition;
    private Circle dotImage;
    private int distanceToDraw;

    public Dot(String color) {
        this.dotColor = color;
        this.dotPosition= 0;
        this.dotPosition = 0;

        this.dotImage = new Circle();
        this.dotImage.changeColor(color);
        this.dotImage.changeSize(Dot.DIAMETER);
        this.dotImage.moveVertical(Dot.DIAMETER*(Dot.nextNumber-1));
        this.dotImage.makeVisible();

        Dot.nextNumber++;
    }
    . . .
```

MAKE MODIFICATIONS & PLAY

12

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

13

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, e.g., `TaxReturn.SINGLE`

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

    . . .
```

14

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

        return tax;
    }
}
```

15

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) {
       this.income = anIncome;
       this.status = aStatus;
    }

    public double getTax() {
        double tax = 0;

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

        return tax;
    }
}
```

16

Checking the status

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

- what is returned if this.status == 3?

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

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

   public double getTax() {
       double tax = 0;

       if (this.status == TaxReturn.SINGLE) {
          if (this.income <= TaxReturn.SINGLE_CUTOFF1) {
              tax = TaxReturn.RATE1 * this.income;
          }
          else if (this.income <= TaxReturn.SINGLE_CUTOFF2) {
              tax = TaxReturn.SINGLE_BASE2 + TaxReturn.RATE2 *
                      (this.income – TaxReturn.SINGLE_CUTOFF1);
          }
          else {
              tax = TaxReturn.SINGLE_BASE3 + TaxReturn.RATE3 *
                      (this.income – TaxReturn.SINGLE_CUTOFF2);
          }
       }
       else if (this.status == TaxReturn.MARRIED) {
           if (this.income <= TaxReturn.MARRIED_CUTOFF1) {
               tax = TaxReturn.RATE1 * this.income;
           }
           else if (income <= MARRIED_CUTOFF2) {
               tax = TaxReturn.MARRIED_BASE2 + TaxReturn.RATE2 *
                       (this.income – TaxReturn.MARRIED_CUTOFF1);
           }
           else {
               tax = TaxReturn.MARRIED_BASE3 + TaxReturn.RATE3 *
                       (this.income – TaxReturn.MARRIED_CUTOFF2);
           }
       }
       return tax;
   }
}
```

17

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?

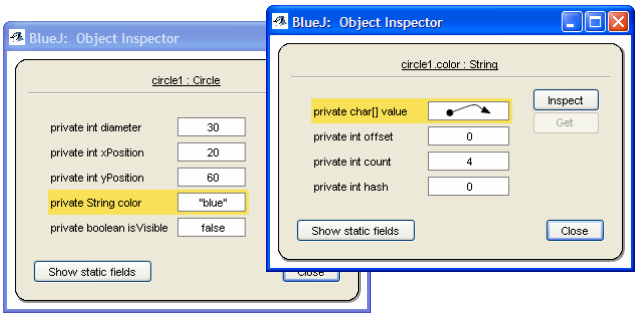
```
   ...

   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) {
      this.income = anIncome;
      this.status = aStatus;
   }

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

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

       return tax;
   }
}
```

18

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



19

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
```

20

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) {
       this.income = anIncome;
       this.status = aStatus;
    }

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

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

        return tax;
    }
}
```

21

String methods

many methods are provided for manipulating Strings

`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

22

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) {
       this.income = anIncome;
       this.tatus = aStatus.toLowerCase();
    }

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

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

        return tax;
    }
}
```

23