

CSC 221: Computer Programming I

Fall 2005

graphics & design

- Dot & DotRace revisited
- Circle class implementation
- adding Circle methods
- static fields
- Line class

1

Recall our Dot class

in the final version:

- a constant represented the maximum step size
- the die field was static to avoid unnecessary duplication
- an accessor method was added to access the dot color

```
public class Dot
{
    private static final int MAX_STEP = 5;
    private static Die die;
    private String dotColor;
    private int dotPosition;

    public Dot(String color) {
        die = new Die(MAX_STEP);
        dotColor = color;
        dotPosition = 0;
    }

    public int getPosition() {
        return dotPosition;
    }

    public String getColor() {
        return dotColor;
    }

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

    public void reset() {
        dotPosition = 0;
    }

    public void showPosition() {
        System.out.println(getColor() + ": " + getPosition());
    }
}
```

2

```

public class DotRace
{
    private Dot redDot;
    private Dot blueDot;
    private int goalDistance;

    public DotRace(int goal) {
        redDot = new Dot("red");
        blueDot = new Dot("blue");
        goalDistance = goal;
    }

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

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

    public int getGoalDistance() {
        return goalDistance;
    }

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

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

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

```

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

3

Adding graphics

what if we wanted to display the dot race visually?

- we could utilize the `Circle` class to draw the dots
- unfortunately, `Circle` only has methods for relative movements

<code>moveLeft()</code>	<code>moveRight()</code>
<code>moveUp()</code>	<code>moveDown()</code>
<code>moveHorizontal(dist)</code>	<code>slowMoveHorizontal(dist)</code>
<code>moveVertical(dist)</code>	<code>slowMoveVertical(dist)</code>
- we need a `Circle` method for absolute movement (based on a `Dot`'s position)
- would also be useful to be able to access the circle diameter

let's explore the `Circle` class implementation

4

Circle implementation

the `Circle` class has the following fields:

```
private int diameter;
private int xPosition;
private int yPosition;
private String color;
private boolean isVisible;
```

the constructor initializes those fields to define a default circle

```
/**
 * Create a new circle at default position with default color.
 */
public Circle()
{
    diameter = 30;
    xPosition = 20;
    yPosition = 60;
    color = "blue";
    isVisible = false;
}
```

5

Circle implementation (cont.)

the low-level graphics methods are:

```
/**
 * Draw the circle with current specifications on screen.
 */
private void draw()
{
    if(isVisible) {
        Canvas canvas = Canvas.getCanvas();
        canvas.draw(this, color,
            new Ellipse2D.Double(xPosition, yPosition, diameter, diameter));
        canvas.wait(10);
    }
}

/**
 * Erase the circle on screen.
 */
private void erase()
{
    if(isVisible) {
        Canvas canvas = Canvas.getCanvas();
        canvas.erase(this);
    }
}
```

these methods utilize a `Canvas` and call its methods to draw and erase the `Circle`

how does `Canvas` work?

- LATER
- or maybe not (→ abstraction)

6

Circle implementation (cont.)

the other Circle methods build upon draw and erase

```
/**
 * Make this circle visible. If it was already visible, do nothing.
 */
public void makeVisible()
{
    isVisible = true;
    draw();
}

/**
 * Move the circle horizontally by 'distance' pixels.
 */
public void moveHorizontal(int distance)
{
    erase();
    xPosition += distance;
    draw();
}

/**
 * Move the circle a few pixels to the right.
 */
public void moveRight()
{
    moveHorizontal(20);
}

. . .
```

7

Adding methods

to add new methods, we don't have to understand canvas or even how draw/erase work

- can abstract away those details and write code that builds upon the abstractions

```
/**
 * Move the circle to a specific location on the canvas.
 * @param xpos the new x-coordinate for the circle
 * @param ypos the new y-coordinate for the circle
 */
public void moveTo(int xpos, int ypos)
{
    erase();
    xPosition = xpos;
    yPosition = ypos;
    draw();
}

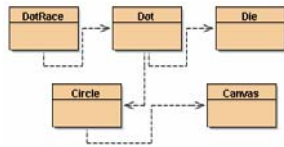
/**
 * Accessor for the circle diameter
 * @return the diameter (in pixels) of the circle
 */
public int getDiameter()
{
    return diameter;
}
```

8

Adding graphics

due to our modular design, changing the display is easy

- each Dot object will maintain and display its own Circle image



- add Circle field
- constructor creates the circle and sets its color
- showPosition MOVES the Circle (instead of displaying text)

```
public class Dot
{
    private static final int MAX_STEP = 5;
    private static Die die;
    private String dotColor;
    private int dotPosition;
    private Circle dotImage;

    public Dot(String color)
    {
        die = new Die(MAX_STEP);
        dotColor = color;
        dotPosition = 0;

        dotImage = new Circle();
        dotImage.setColor(color);
    }

    public int getPosition()
    {
        return dotPosition;
    }

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

    public void reset()
    {
        dotPosition = 0;
    }

    public void showPosition()
    {
        dotImage.moveTo(dotPosition, dotImage.getDiameter());
        dotImage.makeVisible();
    }
}
```

9

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

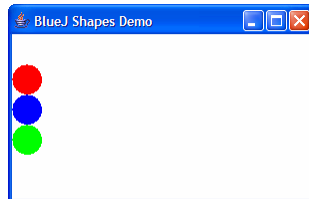
- add Circle field
- create and change color in constructor
- modify showPosition to move and display the circle

10

Better graphics

the graphical display is better than text, but still primitive

- dots are drawn on top of each other (same yPositions)
- would be nicer to have the dots aligned vertically (different yPositions)



PROBLEM: each dot maintains its own state & displays itself

- thus, each dot will need to know what yPosition it should have
- but yPosition depends on what order the dots are created in DotRace (e.g., 1st dot has yPosition = diameter, 2nd dot has yPosition = 2*diameter, ...)
- *how do we create dots with different yPositions?*

11

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 a unique yPosition
- in DotRace, must pass in the number when creating each of the dots

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

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

```
public class Dot
{
    private static final int MAX_STEP = 5;
    private static Die die;
    private String dotColor;
    private int dotPosition;
    private Circle dotImage;
    private int dotNumber;

    public Dot(String color, int num)
    {
        die = new Die(MAX_STEP);
        dotColor = color;
        dotPosition = 0;
        dotImage = new Circle();
        dotImage.setColor(color);
        dotNumber = num;
    }
    ...

    public void showPosition()
    {
        dotImage.moveTo(dotPosition,
            dotImage.getDiameter()*dotNumber);
        dotImage.setVisible();
    }
}
```

this works, but is inelegant

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

12

Option 2: a static field

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

- this requires a new dot to know how many dots have already been created
- 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 MAX_STEP = 5;
    private static Die die;
    private String dotColor;
    private int dotPosition;
    private Circle dotImage;

    private static int nextAvailable = 1;
    private int dotNumber;

    public Dot(String color)
    {
        die = new Die(MAX_STEP);
        dotColor = color;
        dotPosition = 0;
        dotImage = new Circle();
        dotImage.setColor(color);

        dotNumber = nextAvailable;
        nextAvailable++;
    }
    . . .

    public void showPosition()
    {
        dotImage.moveTo(dotPosition,
                        dotImage.getDiameter()*dotNumber);
        dotImage.makeVisible();
    }
}
```

13

Drawing the finish line

if we wanted to draw the finish line

- would need to define a Line class (similar to Circle & Square)
- the finish line belongs to the race, not individual dots
- have a line field, draw it in the constructor

```
public class DotRace
{
    private Dot redDot;
    private Dot blueDot;
    private int goalDistance;
    private Line finishLine;

    public DotRace(int goal) {
        redDot = new Dot("red");
        blueDot = new Dot("blue");
        goalDistance = goal;

        finishLine = new Line(goalDistance, 0, goalDistance, 300);
        finishLine.makeVisible();
    }
    .
    .
    .
}
```

14

Defining a `Line` class

- start with a copy of `Circle`
- what new fields are needed?
- what fields are no longer needed?
- do all the existing methods make sense?
- how do the existing (and meaningful) methods need to be updated?
- how is the low-level drawing & erasing done for lines?

```
new Line2D.Double(x1, y1, x2, y2)
```

*note: a small change had to be made to `Canvas` to make it work for lines
download the latest version*