CSC 221: Computer Programming I

Fall 2006

interacting objects

- modular design: dot races
- constants, static fields
- cascading if-else, logical operators
- variable scope

Dot races

consider the task of simulating a dot race (as on stadium scoreboards)

- different colored dots race to a finish line
- at every step, each dot moves a random distance (say between 1 and 5 units)
- the dot that reaches the finish line first wins!

behaviors?

- create a race (dots start at the beginning)
- step each dot forward a random amount
- access the positions of each dot
- display the status of the race
- reset the race

we could try modeling a race by implementing a class directly

- store positions of the dots in fields
- have each method access/update the dot positions

BUT: lots of details to keep track of; not easy to generalize

A modular design

instead, we can encapsulate all of the behavior of a dot in a class

`Dot` class: create a `Dot` (with a given color)
access the dot's position
take a step
reset the dot back to the beginning
display the dot's color & position

once the `Dot` class is defined, a `DotRace` will be much simpler

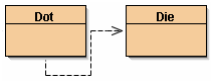
`DotRace` class: create a `DotRace` (with two dots)
access either dot's position
move both dots a single step
reset both dots back to the beginning
display both dots' color & position

3

Dot class

more naturally:

- fields store a Die (for generating random steps), color & position



- constructor creates the Die object and initializes the color and position fields
- methods access and update these fields to maintain the dot's state

CREATE AND PLAY

```
public class Dot {
    private Die die;
    private String dotColor;
    private int dotPosition;

    public Dot(String color) {
        this.die = new Die(5);
        this.dotColor = color;
        this.dotPosition= 0;
    }

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

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

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

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

4

Magic numbers

the Dot class works OK, but what if we wanted to change the range of a dot?

- e.g., instead of a step of 1-5 units, have a step of 1-8
- would have to go and change

  `die = new Die(5);`

  to

  `die = new Die(8);`
- having "magic numbers" like 5 in code is bad practice
  - ✓ unclear what 5 refers to when reading the code
  - ✓ requires searching for the number when a change is desired

```
public class Dot {
    private Die die;
    private String dotColor;
    private int dotPosition;

    public Dot(String color) {
        this.die = new Die(5);
        this.dotColor = color;
        this.dotPosition= 0;
    }

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

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

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

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

5

Constants

better solution: define a *constant*

- a constant is a variable whose value cannot change
- use a constant any time a "magic number" appears in code

a constant declaration looks like any other field except

- the keyword `final` specifies that the variable, once assigned a value, is unchangeable
- the keyword `static` specifies that the variable is shared by all objects of that class
  - since a final value cannot be changed, it is wasteful to have every object store a copy of it
  - instead, can have one static variable that is shared by all

by convention, constants are written in all upper-case with underscores

```
public class Dot {
    private static final int MAX_STEP = 5;

    private Die die;
    private String dotColor;
    private int dotPosition;

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

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

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

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

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

6

Static fields

in fact, it is sometimes useful to have static fields that aren't constants

- if all dots have the same range, there is no reason for every dot to have its own Die
- we could declare the Die field to be static, so that the one Die is shared by all dots

note: methods can be declared static as well

- e.g., `random` is a static method of the predefined `Math` class
- you call a static method by specifying the class name as opposed to an object name: `Math.random()`
- MORE LATER

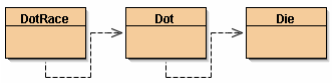
```
public class Dot {
    private static final int MAX_STEP = 5;
    private static Die die = new Die(Dot.MAX_STEP);

    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);
    }
}
```

7

DotRace class

using the `Dot` class, a `DotRace` class is straightforward

- fields store the two Dots



- constructor creates the Dot objects, initializing their colors and max steps
- methods utilize the Dot methods to produce the race behaviors

CREATE AND PLAY

ADD ANOTHER DOT?

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

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

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

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

    public void step() {
        this.redDot.step();
        this.blueDot.step();
    }

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

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

8

Adding a finish line

suppose we wanted to place a finish line on the race

- what changes would we need?

could add a field to store the goal distance

- user specifies goal distance along with max step size in constructor call
- `step` method would not move if either dot has crossed the finish line

```
public class DotRace {
    private Dot redDot;
    private Dot blueDot;
    private int goalDistance;   // distance to the finish line

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

    public int getGoalDistance() {
        return this.goalDistance;
    }

    . . .
```

9

Checking the finish line

to run a step of the race, really need to consider 3 possibilities

- RED has crossed the finish line
- BLUE has crossed the finish line
- neither has crossed, so the race must continue

recall: a simple if statement is a 1-way conditional (*execute this or not*)

```
if (this.redDot.getPosition() >= this.goalDistance) {
    System.out.println("The race is over!");
}
```

an if statement with else case is a 2-way conditional (*execute this or that*)

```
if (this.redDot.getPosition() >= this.goalDistance) {
    System.out.println("The race is over!");
}
else {
    // HANDLE THE CASE WHERE RED HASN'T CROSSED
}
```

here, the case where RED hasn't crossed must be broken into cases (BLUE has crossed, or not)

10

Multi-way conditionals

can get a multi-way conditional by nesting if statements

```
public void step() {
    if (this.redDot.getPosition() >= this.goalDistance) {
        System.out.println("The race is over!");
    }
    else {
        if (this.blueDot.getPosition() >= this.goalDistance) {
            System.out.println("The race is over!");
        }
        else {
            this.redDot.step();
            this.blueDot.step();
        }
    }
}
```

in fact, you can handle an arbitrary number of cases by nesting ifs

- deep nesting can lead to ugly code, however

11

Cascading if-else

to make nested if statements look nicer, some curly-braces can be omitted and the cases indented to show structure

```
public void step() {
    if (this.redDot.getPosition() >= this.goalDistance) {
        System.out.println("The race is over!");
    }
    else if (this.blueDot.getPosition() >= this.goalDistance) {
        System.out.println("The race is over!");
    }
    else {
        this.redDot.step();
        this.blueDot.step();
    }
}
```

such a structure is known as a *cascading if-else*

- control cascades down the cases like water cascading down a waterfall
- if a test fails, control moves down to the next one

```
if (TEST_1) {
    STATEMENTS_1;
}
else if (TEST_2) {
    STATEMENTS_2;
}
else if (TEST_3) {
    STATEMENTS_3;
}
. . .
else {
    STATEMENTS_ELSE;
}
```

12

Combining cases

this code works, but contains redundancy

- there are two cases that result in the same code!

```
public void step() {
    if (this.blueDot.getPosition() >= this.goalDistance) {
        System.out.println("The race is over!");
    }
    else if (this.redDot.getPosition() >= this.goalDistance) {
        System.out.println("The race is over!");
    }
    else {
        this.redDot.step();
        this.blueDot.step();
    }
}
```

fortunately, Java provides *logical operators* for simplifying such cases

`(TEST1 || TEST2)` evaluates to true if either `TEST1` OR `TEST2` is true

`(TEST1 && TEST2)` evaluates to true if either `TEST1` AND `TEST2` is true

`(!TEST)` evaluates to true if `TEST` is NOT true

Logical operators

here, could use `||` to avoid duplication

- print message if *either* blue *or* red has crossed the finish line

```
public void step() {
    if (this.redDot.getPosition() >= this.goalDistance ||
            this.blueDot.getPosition() >= this.goalDistance) {
        System.out.println("The race is over!");
    }
    else {
        this.redDot.step();
        this.blueDot.step();
    }
}
```

warning: the tests that appear on both sides of `||` and `&&` must be complete Boolean expressions

```
(x == 2 || x == 12)   OK
(x == 2 || 12)        BAD!
```

note: we could have easily written `step` using `&&`

- move dots if *both* blue *and* red dots have failed to cross finish line

```
public void step() {
    if (this.redDot.getPosition() < this.goalDistance &&
            this.blueDot.getPosition() < this.goalDistance) {
        this.redDot.step();
        this.blueDot.step();
    }
    else {
        System.out.println("The race is over!");
    }
}
```

EXERCISE

how would we change `step` if we wanted to display the winner?

- RED wins!
- BLUE wins!
- It's a tie!

```
public void step() {
    if ( ??? ) {

        .
        .
        .

    }
    else {
        this.redDot.step();
        this.blueDot.step();
    }
}
```

15

Modularity and scope

key idea: independent modules can be developed independently

- in the real world, a software project might get divided into several parts
- each part is designed & coded by a different team, then integrated together
- internal naming conflicts should not be a problem
  e.g., when declaring a local variable in a method, the programmer should not have to worry about whether that name is used elsewhere

in Java, all variables have a *scope* (i.e., a section of the program where they exist and can be accessed)

- the scope of a field is the entire class (i.e., all methods can access it)
- the scope of a parameter is its entire method
- the scope of a local variable is from its declaration to the end of its method

- so, you can use the same name as a local variable/parameter in multiple methods
- you can also use the same field name in different classes
- in fact, different classes may have methods with the same name!

16

If statements and scope

the curly braces associated with if statements also define new scopes

- a variable declared inside an if case is local to that case
- memory is allocated for the variable only if it is reached
- when that case is completed, the associated memory is reclaimed

```
public double giveChange() {
   if (this.payment >= this.purchase) {
       double change = this.payment – this.purchase;

       this.purchase = 0;
       this.payment = 0;

       return change;
   }
   else {
       System.out.println("Enter more money first.");
       return 0.0;
   }
}
```

if the test fails, then the declaration is never reached and the effort of creating & reclaiming memory is saved

also, cleaner since the variable is declared close to where it is needed

17