

CSC 221: Computer Programming I

Fall 2009

interaction & repetition

- modular design: dot races
- constants, static fields
- conditional repetition, while loops
- logical operators, cascading if-else, variable scope
- counter-driven repetition, for loops
- simulations: volleyball scoring

1

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

2

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

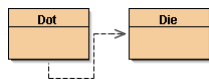
DotRace class: create a `DotRace` (with two dots)
move both dots a single step
reset both dots back to the beginning
display both dots' color & position
run an entire race

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 String getColor() {
        return this.dotColor;
    }

    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.getColor() + ": " +
            this.getPosition());
    }
}
```

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 String getColor() {
        return this.dotColor;
    }

    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.getColor() + ": " +
            this.getPosition());
    }
}
```

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 String getColor() {
        return this.dotColor;
    }

    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.getColor() + ": " +
            this.getPosition());
    }
}
```

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 String getColor() {
        return this.dotColor;
    }

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

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

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

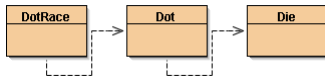
    public void showPosition() {
        System.out.println(this.getColor() + ": " +
            this.getPosition());
    }
}
```

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

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

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

    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();
    }
}
```

CREATE AND PLAY

8

Conditional repetition

running a dot race is a tedious task

- you must call `step` and `showStatus` repeatedly to see each step in the race

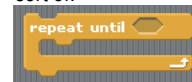
a better solution would be to automate the repetition

in Java, a while loop provides for *conditional repetition*

- similar to an if statement, behavior is controlled by a condition (Boolean test)
- as long as the condition is true, the code in the loop is executed over and over

```
while (BOOLEAN_TEST) {  
    STATEMENTS TO BE EXECUTED  
}
```

sort of:



when a while loop is encountered:

- the loop test is evaluated
- if the loop test is true, then
 - the statements inside the loop body are executed in order
 - the loop test is reevaluated and the process repeats
- otherwise, the loop body is skipped

9

Loop examples

```
int num = 1;  
while (num < 5) {  
    System.out.println(num);  
    num++;  
}
```

```
int x = 10;  
int sum = 0;  
while (x > 0) {  
    sum += x;  
    x -= 2;  
}  
System.out.println(sum);
```

```
int val = 1;  
while (val < 0) {  
    System.out.println(val);  
    val++;  
}
```

10

Recall: paper folding puzzle

if you started with a regular sheet of paper and repeatedly fold it in half, how many folds would it take for the thickness of the paper to reach the sun?

calls for conditional repetition

start with a single sheet of paper
as long as the thickness is less than the distance to the sun, repeatedly
fold & double the thickness

in pseudocode:

```
while (this.thickness < DISTANCE_TO_SUN) {  
    this.fold();  
}
```

11

PaperSheet class

```
public class PaperSheet {  
    private double thickness; // thickness in inches  
    private int numFolds; // the number of folds so far  
  
    public PaperSheet(double initial) {  
        this.thickness = initial;  
        this.numFolds = 0;  
    }  
  
    /**  
     * Folds the sheet, doubling its thickness as a result  
     */  
    public void fold() {  
        this.thickness *= 2;  
        this.numFolds++;  
    }  
  
    /**  
     * Repeatedly folds the sheet until the desired thickness is reached  
     * @param goalDistance the desired thickness (in inches)  
     */  
    public void foldUntil(double goalDistance) {  
        while (this.thickness < goalDistance) {  
            this.fold();  
        }  
    }  
  
    public int getNumFolds() {  
        return this.numFolds;  
    }  
}
```

12

Recall: random sequence generation

for HW3, you added a method that printed multiple sequences

- a simple version would be:

```
/**
 * Displays a set number of random letter sequences of the specified
 * length
 * @param numSequences the number of sequences to generate & display
 * @param seqLength the number of letters in the random sequences
 */
public void displaySequences(int numSequences, int seqLength) {
    int sequencesSoFar = 0;
    while (sequencesSoFar < numSequences) {
        System.out.println(this.randomSequence(seqLength));
        sequencesSoFar = sequencesSoFar + 1;
    }
}
```

however, printing one sequence per line makes it difficult to scan through a large number

- better to put multiple words per line, break line when close to "full"

13

SequenceGenerator class

this can be accomplished using % (the *remainder* operator)

- $(x \% y)$ evaluates to the remainder after dividing x by y

e.g., $7 \% 2 \rightarrow 1$ $100 \% 2 \rightarrow 0$ $13 \% 5 \rightarrow 3$

```
public void displaySequences(int numSequences, int seqLength) {
    int wordsPerLine = 40 / seqLength;

    int sequencesSoFar = 0;
    while (sequencesSoFar < numSequences) {
        System.out.print(this.randomSequence(seqLength) + " ");
        sequencesSoFar = sequencesSoFar + 1;
        if (sequencesSoFar % wordsPerLine == 0) {
            System.out.println();
        }
    }
}
```

14

Recall: 100 bottles of Dew

the `Singer` class displayed verses of various children's songs

- with a loop, we can sing the entire Bottles song in one method call

```
/**
 * Displays the song "100 bottles of Dew on the wall"
 */
public void bottleSong() {
    int numBottles = 100;
    while (numBottles > 0) {
        this.bottleVerse(numBottles, "Dew");
        numBottles--;
    }
}
```

15

Beware of "black holes"

since while loops repeatedly execute as long as the loop test is true, infinite loops are possible (a.k.a. *black hole* loops)

```
int numBottles = 100;
while (numBottles > 0) {
    this.bottleVerse(numBottles, "Dew");
}
```

PROBLEM?

- a necessary condition for loop termination is that some value relevant to the loop test must change inside the loop
in the above example, `numBottles` doesn't change inside the loop
→ if the test succeeds once, it succeeds forever!
- is it a sufficient condition? that is, does changing a variable from the loop test guarantee termination?

NO – "With great power comes great responsibility."

```
int numBottles = 100;
while (numBottles > 0) {
    this.bottleVerse(numBottles, "Dew");
    numBottles++;
}
```

16

Back to the dot race...

can define a `DotRace` method with a while loop to run the entire race

- in pseudocode:

```
RESET THE DOT POSITIONS
SHOW THE DOTS
while (NO DOT HAS WON) {
    HAVE EACH DOT TAKE STEP
    SHOW THE DOTS
}
```

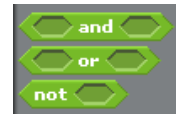
- how do we define the condition for continuing the race?

keep going as long as *neither dot has crossed the finish line*

in other words, *blue dot has not crossed AND red dot has not crossed*

17

Logical operators



Java provides *logical operators* for simplifying such cases

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

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

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

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

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

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

```
public void runRace(int goalDistance) {
    this.reset();
    this.showStatus();
    while (this.redDot.getPosition() < goalDistance &&
           this.blueDot.getPosition() < goalDistance) {
        this.step();
        this.showStatus();
    }
}
```

18

Identifying the winner

three alternatives: tie (both crossed finish line), red wins, blue wins

- can handle more than 2 cases by nesting ifs

```
public void runRace(int goalDistance) {
    this.reset();
    this.showStatus();
    while (this.redDot.getPosition() < goalDistance &&
           this.blueDot.getPosition() < goalDistance) {
        this.step();
        this.showStatus();
    }
    if (this.redDot.getPosition() >= goalDistance &&
        this.blueDot.getPosition() >= goalDistance) {
        System.out.println("IT IS A TIE");
    }
    else {
        if (this.redDot.getPosition() >= goalDistance) {
            System.out.println("RED WINS");
        }
        else {
            System.out.println("BLUE WINS");
        }
    }
}
```

19

Cascading if-else

some curly-braces can be omitted and the cases indented to show structure

- 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;
}
```

```
public void runRace(int goalDistance) {
    this.reset();
    this.showStatus();
    while (this.redDot.getPosition() < goalDistance &&
           this.blueDot.getPosition() < goalDistance) {
        this.step();
        this.showStatus();
    }
    if (this.redDot.getPosition() >= goalDistance &&
        this.blueDot.getPosition() >= goalDistance) {
        System.out.println("IT IS A TIE");
    }
    else if (this.redDot.getPosition() >= goalDistance) {
        System.out.println("RED WINS");
    }
    else {
        System.out.println("BLUE WINS");
    }
}
```

20

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!

21

Control structures and scope

curly braces associated with if statements & while loops define new scopes

- a variable declared inside an if case or while loop is local to that case/loop
- memory is allocated for the variable only if it is reached
- when that case/loop 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

22

Logic-driven vs. counter-driven loops

sometimes, the number of repetitions is unpredictable

- loop depends on some logical condition, e.g., roll dice until 7 is obtained

often, however, the number of repetitions is known ahead of time

- loop depends on a counter, e.g., show # of random sequences, 100 bottles of beer

```
int sequencesSoFar = 0;
while (sequencesSoFar < numSequences) {
    System.out.println(this.randomSequence(seqLength));
    sequencesSoFar++;
}
```

in general (counting up):

```
int rep = 0;
while (rep < #_OF_REPS) {
    CODE_TO_BE_EXECUTED
    rep++;
}
```

```
int numBottles = 100;
while (numBottles > 0) {
    this.bottleVerse(numBottles, "Dew");
    numBottles--;
}
```

in general (counting down):

```
int rep = #_OF_REPS;
while (rep > 0) {
    CODE_TO_BE_EXECUTED
    rep--;
}
```

23

Loop examples:

```
int numWords = 0;
while (numWords < 20) {
    System.out.print("Howdy" + " ");
    numWords++;
}
```

```
int countdown = 10;
while (countdown > 0) {
    System.out.println(countdown);
    countdown--;
}
System.out.println("BLASTOFF!");
```

```
Die d = new Die();

int numRolls = 0;
int count = 0;
while (numRolls < 100) {
    if (d.roll() + d.roll() == 7) {
        count++;
    }
    numRolls++;
}
System.out.println(count);
```

24

For loops

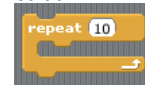
since counter-controlled loops are fairly common, Java provides a special notation for representing them

- a *for loop* combines all of the loop control elements in the head of the loop

```
int rep = 0;
while (rep < NUM_REPS) {
    STATEMENTS_TO_EXECUTE
    rep++;
}

for (int rep = 0; rep < NUM_REPS; rep++) {
    STATEMENTS_TO_EXECUTE
}
```

sort of:



execution proceeds exactly as the corresponding while loop

- the advantage of for loops is that the control is separated from the statements to be repeatedly executed
- also, since all control info is listed in the head, much less likely to forget something
- the scope of the loop variable is the loop body
useful since different loops can reuse the same counter

25

Loop examples:

```
int numWords = 0;
while (numWords < 20) {
    System.out.print("Howdy" + " ");
    numWords++;
}
```

```
for (int numWords = 0; numWords < 20; numWords++) {
    System.out.print("Howdy" + " ");
}
```

```
int countdown = 10;
while (countdown > 0) {
    System.out.println(countdown);
    countdown--;
}
System.out.println("BLASTOFF!");
```

```
for (int countdown = 10; countdown > 0; countdown--) {
    System.out.println(countdown);
}
System.out.println("BLASTOFF!");
```

```
Die d = new Die();

int numRolls = 0;
int count = 0;
while (numRolls < 100) {
    if (d.roll() + d.roll() == 7) {
        count++;
    }
    numRolls++;
}
System.out.println(count);
```

```
Die d = new Die();

int count = 0;
for (int numRolls = 0; numRolls < 100; numRolls++) {
    if (d.roll() + d.roll() == 7) {
        count++;
    }
}
System.out.println(count);
```

26

Simulations

programs are often used to model real-world systems

- often simpler/cheaper to study a model
- easier to experiment, by varying parameters and observing the results

- dot race is a simple simulation
 - utilized Die object to simulate random steps of each dot

in 2001, women's college volleyball shifted from *sideout scoring* (first to 15, but only award points on serve) to *rally scoring* (first to 30, point awarded on every rally). Why?

- shorter games?
- more exciting games?
- fairer games?
- more predictable game lengths?

any of these hypotheses is reasonable – how would we go about testing their validity?

27

Volleyball simulations

conducting repeated games under different scoring systems may not be feasible

- may be difficult to play enough games to be statistically valid
- may be difficult to control factors (e.g., team strengths)
- might want to try lots of different scenarios

simulations allow for repetition under a variety of controlled conditions

VolleyballSim class:

- must specify the relative strengths of the two teams, e.g., power rankings (0-100)
 - if team1 = 80 and team2 = 40, then team1 is twice as likely to win any given point

- given the power ranking for the two teams, can simulate a point using a Die
 - must make sure that the winner is probabilistically correct

- can repeatedly simulate points and keep score until one team wins
- can repeatedly simulate games to assess scoring strategies and their impact

28

VolleyballSim class

to simulate a single rally
with correct probabilities

- create a Die with # sides equal to the sums of the team rankings

e.g., team1=60 & team2=40,
then 100-sided Die

- to determine the winner of a rally, roll the Die and compare with team1's ranking

e.g., if roll <= 60, then team1
wins the rally

- DOES THIS CODE REQUIRE WINNING BY 2?

```
public class VolleyballSim {
    private Die roller; // Die for simulating points
    private int ranking1; // power ranking of team 1
    private int ranking2; // power ranking of team 2

    public VolleyballSim(int team1Ranking, int team2Ranking) {
        this.roller = new Die(team1Ranking+team2Ranking);
        this.ranking1 = team1Ranking;
        this.ranking2 = team2Ranking;
    }

    public int playPoint() {
        if (this.roller.roll() <= this.ranking1) {
            return 1;
        }
        else {
            return 2;
        }
    }

    public int playGame(int winningPoints) {
        int score1 = 0;
        int score2 = 0;

        int winner = 0;
        while (score1 < winningPoints && score2 < winningPoints) {
            winner = this.playPoint();
            if (winner == 1) {
                score1++;
            }
            else {
                score2++;
            }
        }

        return winner;
    }
}
```

29

VolleyballSim class

to force winning by 2, must
add another condition to
the while loop – keep
playing if:

- neither team has reached
the required score

OR

- their scores are within 1
of each other

```
public class VolleyballSim {
    private Die roller; // Die for simulating points
    private int ranking1; // power ranking of team 1
    private int ranking2; // power ranking of team 2

    public VolleyballSim(int team1Ranking, int team2Ranking) {
        this.roller = new Die(team1Ranking+team2Ranking);
        this.ranking1 = team1Ranking;
        this.ranking2 = team2Ranking;
    }

    public int serve() {
        if (this.roller.roll() <= this.ranking1) {
            return 1;
        }
        else {
            return 2;
        }
    }

    public int playGame(int winningPoints) {
        int score1 = 0;
        int score2 = 0;

        int winner = 0;
        while ((score1 < winningPoints && score2 < winningPoints)
            || (Math.abs(score1 - score2) <= 1)) {
            winner = this.serve();
            if (winner == 1) {
                score1++;
            }
            else {
                score2++;
            }
        }

        return winner;
    }
}
```

30

VolleyballStats class

simulating a large number of games is tedious if done one at a time

- can define a class to automate the simulations
- `playGames` creates a `VolleyballSim` object & loops to simulate games
- also maintains stats and displays at end

```
public class VolleyballStats {
    public int numGames;
    public int numPoints;

    public VolleyballStats(int games, int points) {
        this.numGames = games;
        this.numPoints = points;
    }

    /**
     * Simulates repeated volleyball games between teams with
     * the specified power rankings, and displays statistics.
     * @param rank1 the power ranking (0..100) of team 1
     * @param rank2 the power ranking (0..100) of team 2
     */
    public void playGames(int rank1, int rank2) {
        VolleyballSim matchup = new VolleyballSim(rank1, rank2);

        int team1Wins = 0;
        for (int i = 0; i < this.numGames; i++) {
            if (matchup.playGame(this.numPoints) == 1) {
                team1Wins++;
            }
        }

        System.out.println("Assuming (" + rank1 + "-" + rank2 +
            ") rankings over " + this.numGames +
            " games to " + this.numPoints + ":");
        System.out.println(" team 1 winning percentage: " +
            100.0*team1Wins/this.numGames + "%");
    }
}
```

31

Interesting stats

out of 10,000 games, 30 points to win:

- team 1 = 80, team 2 = 80 → team 1 wins 50.1% of the time
- team 1 = 80, team 2 = 70 → team 1 wins 70.6% of the time
- team 1 = 80, team 2 = 60 → team 1 wins 87.1% of the time
- team 1 = 80, team 2 = 50 → team 1 wins 96.5% of the time
- team 1 = 80, team 2 = 40 → team 1 wins 99.7% of the time

CONCLUSION: over 30 points, the better team wins!

32