CSC 221: Computer Programming I

Fall 2006

repetition & simulations

- conditional repetition, while loops
- examples: dot race, paper folding puzzle, sequence generator, songs
- infinite (black hole) loops
- counter-driven repetition, for loops
- simulations: volleyball scoring

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
}
```

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

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++;
}
```

3

`runRace` method

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
}
```

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

    . . .

     /**
      * Conducts an entire dot race, showing the status
      * after each step.
      */
    public void runRace() {
        this.reset();
        this.showStatus();
        while (this.getRedPosition() < this.goalDistance &&
               this.getBluePosition() < this.goalDistance) {
            this.step();
            this.showStatus();
        }
    }
}
```

4

Paper folding puzzle

recall:

- 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.thickness *= 2;
    this.numFolds++;
}
```

5

`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;
    }
}
```

6

SequenceGenerator class

recall from HW 1:

- SequenceGenerator had a method for generating a random sequence

```
private String seqAlphabet;  // field containing available letters

public String randomSequence(int seqLength) {
    String seq = "";
    int rep = 0;
     while (rep < seqLength) {
         int index = (int)(Math.random()*this.seqAlphabet.length());
         seq = seq + this.seqAlphabet.charAt(index);
        rep++;
     }
    return seq;
}
```

useful String methods:

```
    int length();           // returns # of chars in String

    char charAt(int index); // returns the character at index
                            //   indexing starts at 0
                            //   i.e., 1st char at index 0
```

note: + will add a char to a String

7

Generating many sequences

for HW1, you added a method that generated and printed 5 sequences

- subsequently, cut-and-pasted 20 copies in order to display 100 sequences

```
public void displaySequences(int seqLength) {
    System.out.println(this.randomSequence(seqLength) + " " +
                       this.randomSequence(seqLength) + " " +
                       this.randomSequence(seqLength) + " " +
                       this.randomSequence(seqLength) + " " +
                       this.randomSequence(seqLength));
}
```

better solution: use a loop to generate and print an arbitrary number

- to be general, add a 2nd parameter that specifies the desired number of sequences

```
public void displaySequences(int seqLength, int numSequences) {
    int rep = 0;
    while (rep < numSequences) {
        System.out.println( this.randomSequence(seqLength) );
        rep++;
    }
}
```

8

Controlling output

printing one word per line makes it difficult to scan through a large number

- better to put multiple words per line, e.g., new line after every 5 words

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

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

e.g., `7 % 2 → 1` `100 % 2 → 0` `13 % 5 → 3`

```
public void displaySequences(int seqLength, int numSequences) {
    int rep = 0;
    while (rep < numSequences) {
        System.out.print( this.randomSequence(seqLength) + " " );
        rep++;

        if (rep % 5 == 0) {              // if rep # is divisible by 5,
            System.out.println();        //   then go to the next line
        }
    }
}
```

9

100 bottles of Dew

recall the Singer class, which 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--;
    }
}
```

10

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++;
}
```

11

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

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--;
}
```

12

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);
```

13

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
}
```

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

14

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);
```

15

Variable scope

recall: the *scope* of a variable is the section of code in which it exists

- for a field, the scope is the entire class definition
- for a parameter, the scope is the entire method
- for a local variable, the scope begins with its declaration & ends at the end of the enclosing block (i.e., right curly brace)

```
public class DiceStuff {
  private Die die;

  . . .

  public void showSevens(int numReps) {
    int count = 0;
    for (int numRolls = 0; numRolls < numReps; numRolls++) {
      if (this.die.roll() + this.die.roll() == 7) {
        count++;
      }
    }
    System.out.println(count);
  }

  . . .
}
```

if the loop counter is declared in the header for the loop, its scope is limited to the loop

→ same loop counter could be used in multiple for loops

16

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?

17

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

18

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 and 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

```
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) {
        roller = new Die(team1Ranking+team2Ranking);
        ranking1 = team1Ranking;
        ranking2 = team2Ranking;
    }

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

        System.out.print("team " + team + " serves: team " +
                         winner + " wins the rally");
        return winner;
    }

    . . .
```


VolleyballSim class

to simulate an entire game

- must specify the number of points required to win
- repeatedly simulate a rally and keep track of the points for each team
- assumes that team 1 gets the first serve
- DOES THIS CODE REQUIRE WINNING BY 2?

```
    . . .

    public int playGame(int winningPoints) {
        int score1 = 0;
        int score2 = 0;
        int servingTeam = 1;

        int winner = 0;
        while (score1 < winningPoints && score2 < winningPoints) {
            winner = this.serve(servingTeam);
            if (winner == 1) {
                score1++;
                servingTeam = 1;
            }
            else {
                score2++;
                servingTeam = 2;
            }

            System.out.println(" (" + score1 + "-" + score2 + ")");
        }
        return winner;
    }
}
```

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 int playGame(int winningPoints)
    {
        int score1 = 0;
        int score2 = 0;
        int servingTeam = 1;

        int winner = 0;
        while ((score1 < winningPoints && score2 < winningPoints)
                                || (Math.abs(score1 - score2) <= 1)) {
            winner = this.serve(servingTeam);
            if (winner == 1) {
                score1++;
                servingTeam = 1;
            }
            else {
                score2++;
                servingTeam = 2;
            }

            System.out.println(" (" + score1 + "-" + score2 + ")");
        }
        return winner;
    }
}
```


VolleyballStats class

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

- can define a class to automate the simulations and display the results
- since the number of games and points to win will change less often, store those in fields with default values
- provide accessor and mutator methods for viewing and changing these fields

```
public class VolleyballStats {
    public static final int INITIAL_REPS = 10000;
    public static final int INITIAL_POINTS = 30;
    private int numGames;
    private int winPoints;

    public VolleyballStats() {
        this.numGames = VolleyballStats.INITIAL_REPS;
        this.winPoints = VolleyballStats.INITIAL_POINTS;
    }

    public int getNumberOfGames() {
        return this.numGames;
    }

    public void setNumberOfGames(int newNum) {
        this.numGames = newNum;
    }

    public int getPointsToWin() {
        return this.winPoints;
    }

    public void setPointsToWin(int newNum) {
        this.winPoints = newNum;
    }

    . . .
```

VolleyballStats class

to view stats on a large number of games,

- call playGames with the desired team rankings
- it creates a VolleyballSim object with those ranking
- it loops to simulate repeated games and maintains stats
- finally, displays the stats nicely

```
    . . .

    /**
     * Simulates getNumberOfGames() volleyball games between teams with
     *   the speciefied 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.getNumberOfGames(); i++) {
            if (matchup.playGame(this.getPointsToWin()) == 1) {
                team1Wins++;
            }
        }

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


BIG PROBLEM!

currently, the serve and playGame methods in VolleyballSim display info about each rally

- this is nice when simulating a single game
- it's not nice when simulating 10,000 games
- for now, can simply comment out the println statements

```
public class VolleyballSim {
    . . .

    public int serve(int team) {
        . . .

        // System.out.print("team " + team + " serves: team " +
        //                  winner + " wins the rally");

        . . .
    }

    private int playGame(int winningPoints) {
        . . .

        // System.out.println(" (" + score1 + "-" + score2 + ")");

        . . .
    }
}
```

a better solution would be to add a field that controls whether output is displayed, e.g.,

```
if (this.showOutput) {
    System.out.println(. . .);
}
```

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!

25

TEST 2

similar format to TEST 1 (including several "extra" points)

- TRUE/FALSE, multiple choice
- short answer, explain code
- trace/analyze/modify/augment code

- expect to be given a class and be asked to
  create/initialize an object of that class, call methods on that object, augment
- expect to trace code segments involving loops & conditionals

study advice:

- see online review sheet for outline of topics covered
- review lecture notes (if not *mentioned* in notes, will not be on test)
- read text to augment conceptual understanding, see more examples & exercises
- review quizzes and homeworks

- feel free to review other sources (lots of Java tutorials online, e.g.,www.javabat.com)

26