

CSC 221: Computer Programming I

Fall 2004

variables and expressions (cont.)

- counters and sums
- expression evaluation, mixed expressions
- int vs. real division, casting
- final variables

repetition and simulation

- conditional repetition, while loops
- examples: dot race, paper folding puzzle, sequence generator, songs
- counter-driven repetition, for loops
- simulations

1

Counters and sums

two common uses of variables:

- to count the number of occurrences of some event
 - ✓ initialize once to 0
 - ✓ increment the counter for each occurrence
- to accumulate the sum of some values
 - ✓ initialize once to 0
 - ✓ add each new value to the sum

recall from TEST 1:

- `numGrades` counts the number of grades
- `gradeSum` accumulates the sum of all grades

recall `ESPTester` class

```
public class GradeCalculator
{
    private int numGrades;    // number of grades entered
    private double gradeSum; // sum of all grades entered

    /**
     * Constructs a grade calculator with no grades so far
     */
    public GradeCalculator()
    {
        numGrades = 0;
        gradeSum = 0;
    }

    /**
     * Adds a new grade to the running totals.
     * @param newGrade the new grade to be entered
     */
    public void addGrade(double newGrade)
    {
        gradeSum += newGrade;
        numGrades++;
    }

    /**
     * Calculates the average of all grades entered so far.
     * @return the average of all grades entered
     */
    public double averageGrade()
    {
        if (numGrades == 0) {
            return 0.0;
        }
        else {
            return gradeSum/numGrades;
        }
    }
}
```

2

A minor modification

what if grades were always whole numbers?

- we could change the `gradeSum` field and `addGrade` parameter to be ints

problem:

```
GradeCalculator calc =
    new GradeCalculator();

calc.addGrade(90);
calc.addGrade(91);

double avg =
    calc.averageGrade();

System.out.println(avg);
```

outputs 90, not 90.5

WHY?

```
public class GradeCalculator
{
    private int numGrades;    // number of grades entered
    private int gradeSum;    // sum of all grades entered

    /**
     * Constructs a grade calculator with no grades so far
     */
    public GradeCalculator()
    {
        numGrades = 0;
        gradeSum = 0;
    }

    /**
     * Adds a new grade to the running totals.
     * @param newGrade the new grade to be entered
     */
    public void addGrade(int newGrade)
    {
        gradeSum += newGrade;
        numGrades++;
    }

    /**
     * Calculates the average of all grades entered so far.
     * @return the average of all grades entered
     */
    public double averageGrade()
    {
        if (numGrades == 0) {
            return 0.0;
        }
        else {
            return gradeSum/numGrades;
        }
    }
}
```

3

Expressions and types

in general, Java operations are *type-preserving*

- if you add two ints, you get an int $2 + 3 \rightarrow 5$
- if you add two doubles, you get a double $2.5 + 3.5 \rightarrow 6.0$
- if you add two Strings, you get a String $"foo" + "2u" \rightarrow "foo2u"$

this applies to division as well

- if you divide two ints, you get an int – any fractional part is discarded!
 $8/4 \rightarrow 2$ $9/4 \rightarrow (2.25) \rightarrow 2$ $-9/4 \rightarrow (-2.25) \rightarrow -2$
 $1/2 \rightarrow ???$ $999/1000 \rightarrow ???$ $x/(x+1) \rightarrow ???$

for mixed expressions, the int is automatically converted to double first

$3.2 + 1 \rightarrow (3.2 + 1.0) \rightarrow 4.2$

$1 / 2.0 \rightarrow (1.0 / 2.0) \rightarrow 0.5$

4

int vs. real division

if `gradeSum` is an `int` variable, then `int` division occurs

$$(90+91)/2 \rightarrow 181/2 \rightarrow 90$$

since the return type for `averageGrade` is `double`, `90` is converted to `90.0` before returning

if `gradeSum` is a `double` variable, then `real` division occurs

- the denominator converts into a `double`

$$(90.0+91.0)/2 \rightarrow 181.0/2 \rightarrow 181.0/2.0 \rightarrow 90.5$$

FYI: the `%` operator gives the remainder after performing `int` division

$$12 \% 2 \rightarrow 0 \quad 13 \% 2 \rightarrow 1 \quad 18 \% 5 \rightarrow 3$$

```
public class GradeCalculator
{
    private int numGrades;
    private int gradeSum;

    . . .

    public double averageGrade()
    {
        if (numGrades == 0) {
            return 0.0;
        }
        else {
            return gradeSum/numGrades;
        }
    }
}
```

```
public class GradeCalculator
{
    private int numGrades;
    private double gradeSum;

    . . .

    public double averageGrade()
    {
        if (numGrades == 0) {
            return 0.0;
        }
        else {
            return gradeSum/numGrades;
        }
    }
}
```

5

type casting

OPTION 1: we could get around `int` division by making all variables `doubles`

- BAD** – if the values really are integers, the code should reflect that

OPTION 2: we could create a `double` copy whenever we wanted `real` division

```
double gradeSumDouble = gradeSum;
return gradeSumDouble/numGrades;
```

- TEDIOUS** – have to create a new variable every time

OPTION 3: type casting

- in Java, can convert a value to a different (but compatible) type via

`(NEWTYPE) VALUE`

- same approach as **OPTION 2**, but don't need to explicitly create a new variable

```
public class GradeCalculator
{
    private int numGrades;
    private int gradeSum;

    . . .

    public double averageGrade()
    {
        if (numGrades == 0) {
            return 0.0;
        }
        else {
            return (double)gradeSum/numGrades;
        }
    }
}
```

6

Evaluation order

complex expressions are evaluated in a well-defined order

- in Java, * and / have "higher precedence" than + and -

`1 + 2 * 3 → 1 + (2 * 3) → 1 + 6 → 7`

- between operators of the same precedence, evaluation goes left-to-right

`8 / 4 * 2 → (8 / 4) * 2 → 2 * 2 → 4`

- programmer can force order of evaluation using parentheses (DO IT TO CLARIFY)

`(1 + 2) * 3 → 3 * 3 → 9`

`8 / (4 * 2) → 8 / 8 → 1`

since mixed expressions cast ints automatically, order makes a difference

`100.0 * 5 / 2 →`

`5 / 2 * 100.0 →`

`5 / (2 * 100.0) →`

7

Mixing strings & numbers

as we have seen, can mix Strings and numbers using +

```
int x = 12;
System.out.println("the value is " + x);
```

- if add an int/double and a String, the int/double is converted to a String

```
"the value is " + x → "the value is " + 12
                    → "the value is " + "12"
                    → "the value is 12"
```

be careful:

```
int x = 1, y = 2;
System.out.println(x + y + " is the sum");
System.out.println("the sum is " + x + y);
System.out.println("the sum is " + (x + y));
```

8

Back to the dot race...

showPosition shifts each new dot down 30 pixels in the display

- why 30?

having a "magic number" appear in the code without apparent reason is bad

- unclear to the reader/adaptor
- difficult to maintain

ideally, could use an accessor method on the dotImage to get its diameter

```
dotImage.moveTo(dotPosition, dotImage.getDiameter()*dotNumber);
```

but the Circle class does not provide accessor methods! ☹️

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

    private static int nextAvailable = 1;
    private int dotNumber;

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

        dotNumber = nextAvailable;
        nextAvailable++;
    }

    . . .

    public void showPosition()
    {
        dotImage.moveTo(dotPosition, 30*dotNumber);
        dotImage.setVisible();
    }
}
```

9

Static field for dot size

if can't get the diameter from Circle, will need to store in Dot

- could pass in as parameter to constructor, but not worth it
- store the diameter in a field, can then use to set the circle size (in the constructor) and space dots (in showPosition)
- since all dots will have the same size, make the field static

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

    private static int nextAvailable = 1;
    private int dotNumber;

    private static int dotSize = 40;

    public Dot(String color, int maxStep)
    {
        die = new Die(maxStep);
        dotColor = color;
        dotPosition = 0;
        dotImage = new Circle();
        dotImage.setColor(color);
        dotImage.setSize(dotSize);

        dotNumber = nextAvailable;
        nextAvailable++;
    }

    . . .

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

10

final variables

no dot should be able to change the size of other dots

- the value of a variable can be "locked down" by declaring the variable as *final*
- once given its initial value, a final variable cannot be changed
- any attempt to do so would be caught by the compiler

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

    private static int nextAvailable = 1;
    private int dotNumber;

    private final static int dotSize = 40;

    public Dot(String color, int maxStep)
    {
        die = new Die(maxStep);
        dotColor = color;
        dotPosition = 0;
        dotImage = new Circle();
        dotImage.setColor(color);
        dotImage.setSize(dotSize);

        dotNumber = nextAvailable;
        nextAvailable++;
    }

    . . .

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

11

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

12

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++;
}
```

13

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 A STEP
    SHOW THE DOTS
}
```

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

    . . .

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

14

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
double and display the thickness

in pseudocode:

```
current = INITIAL_THICKNESS;
while (current < DISTANCE_TO_SUN) {
    current *= 2;
    System.out.println(current);
}
```

15

FoldPuzzle class

```
public class FoldPuzzle
{
    private double thickness;

    /**
     * Constructs the FoldPuzzle object
     * @param initial the initial thickness (in inches) of the paper
     */
    public FoldPuzzle(double initial)
    {
        thickness = initial;
    }

    /**
     * Computes how many folds it would take for paper thickness to reach the goal distance
     * @param goalDistance the distance (in inches) the paper thickness must reach
     * @return number of folds required
     */
    public int FoldUntil(double goalDistance)
    {
        double current = thickness;
        int numFolds = 0;

        while (current < goalDistance) {
            current *= 2;
            numFolds++;
            System.out.println(numFolds + ": " + current);
        }
        return numFolds;
    }
}
```

16

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 desiredLength)
{
    String seq = ""; // start with empty seq

    while (seq.length() < desiredLength) { // repeatedly:
        int index = (int)(Math.random()*seqAlphabet.length()); // pick random index
        seq = seq + seqAlphabet.charAt(index); // add letter to seq
    }

    return seq; // return the 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

17

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(randomSequence(seqLength) + " " + randomSequence(seqLength) + " " +
        randomSequence(seqLength) + " " + randomSequence(seqLength) + " " +
        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( randomSequence(seqLength) );
        rep++;
    }
}
```

18

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 % (remainder operator)

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

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

19

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) {
        bottleVerse(numBottles, "Dew");
        numBottles--;
    }
}
```

20

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) {
    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) {
    bottleVerse(numBottles, "Dew");
    numBottles--;
}
```

21

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( 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) {
    bottleVerse(numBottles, "Dew");
    numBottles--;
}
```

in general (counting down):

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

22

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);
```

23

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

24

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);
```

25

Variable scope

scope: the section of code in which a variable 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 d;

    . . .

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

    . . .
}
```

26

Scope questions

can the same parameter/local variable name appear in multiple methods?

- YES: each method defines a new scope, so each occurrence is distinct

```
public class DiceStuff
{
    private Die d;

    public void showSevens(int numReps)
    {
        . . .
    }

    public void showDoubles(int numReps)
    {
        . . .
    }
}
```

this is good, since it means you don't have to worry about what names have been used elsewhere

can a local variable have the same name as a field?

- YES: the local variable takes priority – so masks the field!

```
public void showDoubles(int numReps)
{
    int d = 0;

    . . .
}
```

this is BAD, since you cannot access the Die field anymore. You must be aware of field names when defining local variables

27

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 25, 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?

28

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

VolleyBallSimulator 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 winner is probabilistically correct
- can repeatedly simulate points and keep score til one team wins
- can repeatedly simulate games to assess scoring strategies and their impact

29

VolleyBallSimulator class

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

    /**
     * Constructs a volleyball game simulator.
     * @param team1Ranking the power ranking (0-100) of team 1, the team that serves first
     * @param team2Ranking the power ranking (0-100) of team 2, the receiving team
     */
    public VolleyBallSimulator(int team1Ranking, int team2Ranking)
    {
        roller = new Die(team1Ranking+team2Ranking);
        ranking1 = team1Ranking;
        ranking2 = team2Ranking;
    }

    /**
     * Simulates a single rally between the two teams.
     * @return the winner of the rally (either "team 1" or "team 2")
     */
    public String playRally()
    {
        if (roller.roll() <= ranking1) {
            return "team 1";
        }
        else {
            return "team 2";
        }
    }
}
```

30

VolleyBallSimulator class (cont.)

```
...
/**
 * Simulates an entire game using the rally scoring system.
 * @param winningPoints the number of points needed to win the game (winningPoints > 0)
 * @return the winner of the game (either "team 1" or "team 2")
 */
public String playGame(int winningPoints)
{
    int score1 = 0;        // number of points won by team 1
    int score2 = 0;        // number of points won by team 2
    String winner = "";

    while (score1 < winningPoints && score2 < winningPoints) {
        winner = playRally();
        if (winner.equals("team 1")) {
            score1++;
        }
        else {
            score2++;
        }

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

when comparing Strings, should use the equals method instead of == (more later)

note: the winning team must win by at least 2 points

- is that reflected in this code?
- what changes/additions do we need to make?

31

VolleyBallSimulator class (cont.)

```
...
/**
 * Simulates an entire game using the rally scoring system.
 * @param winningPoints the number of points needed to win the game (winningPoints > 0)
 * @return the winner of the game (either "team 1" or "team 2")
 */
public String playGame(int winningPoints)
{
    int score1 = 0;
    int score2 = 0;
    String winner = "";

    while ((score1 < winningPoints && score2 < winningPoints) ||
           (Math.abs(score1 - score2) <= 1)) {

        winner = playRally();
        if (winner.equals("team 1")) {
            score1++;
        }
        else {
            score2++;
        }

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

can add another condition to the loop test

- keep going if difference in scores is <= 1
- makes use of Math.abs function for absolute value

32

VolleyBallSimulator class (cont.)

```
...
/**
 * Simulates repeated volleyball games and displays statistics
 * @param numGames the number of games to be simulated (numGames > 0)
 * @param winningPoints the number of points needed to win the game (winningPoints > 0)
 */
public void winningPercentage(int numGames, int winningPoints)
{
    int team1Wins = 0;

    for (int game = 0; game < numGames; game++) {
        if (playGame(winningPoints).equals("team 1")) {
            team1Wins++;
        }
    }

    System.out.println("Out of " + numGames + " games to " + winningPoints +
        ", team 1 (" + ranking1 + "-" + ranking2 + ") won: " +
        100.0*team1Wins/numGames + "%");
}
}
```

finally, can define a method to simulate repeated games and maintain stats

annoying feature?

33

Interesting stats

out of 1000 games, 25 points to win:

- team 1 = 80, team 2 = 80 → team 1 wins 49.9% of the time
- team 1 = 80, team 2 = 70 → team 1 wins 69.2% of the time
- team 1 = 80, team 2 = 60 → team 1 wins 87.0% of the time
- team 1 = 80, team 2 = 50 → team 1 wins 95.7% of the time
- team 1 = 80, team 2 = 40 → team 1 wins 99.4% of the time

CONCLUSION: over 25 points, the better team wins!

34

HW4

you will augment the volleyball simulator & conduct numerous experiments

- add `SHOW_POINTS` field to the class, to control whether individual points are displayed during the simulation
 - ✓ initially, `SHOW_POINTS` is true, so `playGame` shows each point
 - ✓ however, `winningPercentage` will set to false before repeated simulations
- generalize `playGame` so that it can simulate rally and sideout scoring
 - ✓ compare the winning percentages using the two scoring systems

HOW MANY GAMES ARE REQUIRED TO OBTAIN CONSISTENT STATS?

WHICH SCORING SYSTEM FAVORS THE UNDERDOG MORE?

CAN GAME LENGTH BE ADJUSTED TO BALANCE THE ADVANTAGE?