Repetition Structures

Chapter 8

Chapter Contents

Chapter Objectives

- Expand on intro to repetition, Chapter 4
- Examine `for` loops in more detail
- Compare, contrast `while` and `do` loops
- Introduce recursion
- Look at event-driven programming and state diagrams
- Take a first look at algorithm analysis

8.1 Introductory Example: the Punishment of Gauss

- Problem:
  As a young student, Gauss was disciplined with the task of summing the numbers from 1 through 100. He solved the problem almost immediately. We will learn his strategy later.

  We will construct a method, that given n, will sum 1 + 2 + … + n

Object Centered Design

- Objects:

Objects	Kind	Type	Movement	Name
limit value, n	varying	`int`	received	`n`
1+2+...+n	varying	`int`	returned	

- Specification of method

```
public class Formula
{   public static int summation(int n)
      {  .  .  .     }
}
```

Operations/ Algorithm

1. Initialize `runningTotal` to 0
2. Initialize `count` to 1
3. Loop as long as `count <= n`
   a. Add `count` to `runningTotal`
   b. Add 1 to `count`
4. Return `runningTotal`

`for` Loop Version of `summation()` Method

```
public static int summation(int n)
  {
  int runningTotal = 0;

  for (int count = 1; count <= n; count++)
    runningTotal += count;

  return runningTotal;
  }
```

Note driver program for `summation()`, Figure 8.2

8.2 Repetition: The `for` Loop Revisited

- Counter-controlled loops: loops where a set of statements is executed once for each value in a specified range

Initialization expression

Loop condition

Step expression

```
for (int count = n; count >= 0; count--)
runningTotal += count; // descending form
```

Loop body

Nested Loops: Displaying a Multiplication Table

- The statement that appears within a **for** statement may itself be a **for** statement

```
for (int x = 1; x < lastX; x++)
  for (int y = 1; y <= lastY; y++)
  {   product = x * y;
      theScreen.println("x * y = "+product);
  }
```

Inner loop

Outer loop

Warning

- If the body of a counting loop alters the values of any variables involved in the loop condition, then the number of repetitions may be changed

```
for (int I = 0; I <= limit; I++)
{
  theScreen.println(I);
  limit++;
}
```

What happens in this situation?

Forever Loops

- Occasionally a need exists for a loop that runs for an indefinite number of times

```
for ( ;  ;  )
    statement;
```

- The above statement will do this … it will run forever … unless …
- The body of the loop contains a statement that will terminate the loop when some condition is satisfied

The `break` Statement

- Two forms
```
break;
break identifier;
```
- First form used to terminate execution of an enclosing loop or `switch` statement
- Second form used to transfer control to a statement with `identifier` as the label
```
  identifier : Statement;
```

Use of the `break` Statement

```
for ( ;   ;   )
   { statement;
        . . .
     if (termination_condition)
         break;
         . . .
      statement;
   }
```

Could be statements which offer a menu choice. Termination condition would be `choice == 'Q'`

Then these would be statements which process the menu choice when it is not 'Q'

In either case, when the user enters 'Q' to Quit, the termination condition is met and the loop terminates

The `continue` Statement

- Two forms:
```
continue;
continue label;
```
- First form transfers control to the innermost enclosing loop
  - current iteration terminated, new one begins
- Second form transfers control to enclosing labeled loop
  - current iteration terminated, new one begins

Returning From a Loop

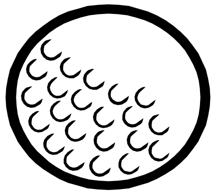
```
for ( ;  ;  )
   {theScreen.print(MENU);
    choice = theKeyboard.readChar()
    if (choice >=0 && choice <= 5)
          return choice;
    theScreen.println( "error .. "); }
```

- Assuming this forever loop is in a value returning method when one of options 0 – 5 is chosen.
  - the loop is terminated and …
  - the menu choice returned by the function
  - invalid choices keep the loop running

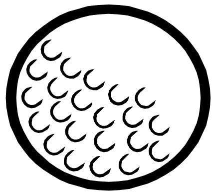
8.3 Repetition: The `while` loop

- This is a looping structure that tests for the termination condition <u>at the top</u> of the loop
  - also called a <u>pretest</u> loop
  - a simpler syntax than placing a **`break`** in an **`if`** statement at the beginning of a forever loop

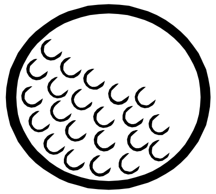
Example: Follow the Bouncing Ball



Consider a ball that when dropped, it bounces to a height one-half to its previous height.



We seek a program which displays the number of the bounce and the height of the bounce, until the height of the bounce is very small



Behavior

```
For ball bounce results:
Enter height of ball drop -> 10

Bounce 1 = 5
Bounce 2 = 2.5
 ...
```

- Prompt for and receive height of drop
- Display bounce number and height of bounce

Objects

Objects	Kind	Type	Name
current height	varying	`real`	`height`
bounce number	varying	`int`	`bounce`
a very small number	constant	`real`	`SMALL_NUM`

Operations/ Algorithm

1. Initialize **`bounce`** to 0
2. Prompt, read value for **`height`**
3. Display original **`height`** value with label
4. Loop:
   a. if **`height < SMALL_NUM`**, terminate loop
   b. replace **`height`** with **`height/2`**
   c. add 1 to **`bounce`**
   d. display **`bounce`** and **`height`**

   End Loop

Coding and Trial

- Note source code, Figure 8.4 sample run
- Note use of while statement

```
while (height >= SMALL_NUMBER)
   { height *= REBOUND_FACTOR;
     bounce++;
     theScreen.println( … );
   }
```

Syntax

```
while (loop_condition)
    statement;
```

- Where
  - `while` is a keyword
  - `loop_condition` is a boolean expression
  - `statement` is a simple or compound statement

`while` Statement Behavior

```
while (loop_condition)
    statement;
```

1. `loop_condition` evaluated
2. If `loop_condition` is true
   - `statement` executed
   - control returns to step 1

Otherwise:

- Control transferred to statement following the `while`

Note – possible that statement is <u>never</u> executed – called "zero-trip behavior"

Loop Conditions vs. Termination Conditions

- Forever loop
  - continues repetition when condition is <u>false</u>
  - terminates when condition is <u>true</u>
- `while` loop is exactly opposite
  - continues repetition while condition is <u>true</u>
  - terminates when it goes <u>false</u>
- Warning for either case:
  - Make sure condition is affected by some statement in the loop to eventually result in loop termination

8.4 Repetition: The `do` Loop

- `while` loop evaluates loop condition <u>before</u> loop body is executed
- We sometimes need looping with a <u>posttest</u> structure
  - the loop body will <u>always</u> execute at least once
- Example: Making a Program Pause
  We seek a method that, given a length of time will make a program pause for that length of time

Preliminary Analysis

- `System` class from `java.lang` provides a method `currentTimeMillis()`
  - returns number of millisec since 1/1/1970
- We will record the results of the function at the start of our pause method
  - repeatedly view results to determine elapsed time
- Called “busy-waiting” technique

Objects

Objects	Kind	Type	Movement	Name
num sec	varying	`double`	received	`seconds`
num millsec	varying	`long`		`milliseconds`
starting time	constant	`long`		`START_TIME`
current time	varying	`long`		`currentTime`

Method specification

```
public class Controller
{ public static void pause(double seconds)
    {  . . . }
}
```

Operations/Algorithm

1. Receive `seconds`
2. Initialize `START_TIME`
3. If `seconds` > 0
   a. compute `milliseconds` from `seconds`
   b. loop
      get `currentTime`
      if `currentTime-START_TIME > milliseconds`
         terminate repetition
      End loop
   else
      display error message

Coding and Testing

- Note source code Figure 8.5, driver Figure 8.6
- Note looping mechanism

```
do
   currentTime = System.currentTimeMillis();
while (currentTime - START_TIME <=
                              milliSeconds);
```

Syntax

```
do
   statement
while (loop_condition);
```

- Where
  - `do` and `while` are keywords
  - `statement` is simple or compound
  - `loop_condition` is a boolean expression
  - note requirement of semicolon at end

Behavior

- When execution reaches a `do` loop:
  1. `statement` is executed
  2. `loop_condition` is evaluated
  3. if `loop_condition` is true
     control returns to step 1.
     otherwise
     control passes to 1st statement following loop structure

Loop Conditions vs. Termination Conditions

- `do` loop and `while` loop both use loop condition that
  - continues repetition as long as it is true
  - terminates repetition when false
- forever loop does the opposite
- If `do` loop or `while` loop used x <= 7 then forever loop would use x > 7 (exact opposite comparison)

Input do Loops: The Query Approach

- Recall two different types of input loops
  - counting approach
  - sentinel approach
- Counting approach asked for number of inputs to be entered, used `for()` loop
  - requires foreknowledge of how many inputs
- Sentinel approach looked for special valued input to signify termination
  - requires availability of appropriate sentinel value

Query Approach

- Use a do loop
  - loop body always executed at least one time
- Query user at end of loop body
  - user response checked in loop condition

```
do {
   // whatever …
  . . .
 theScreen.print("More inputs? (Y/N) : ");
   response = theKeyboard.readChar();
} while (response=='y' || response =='Y');
```

Query Methods

- Note the excess code required in the loop
  - to make the query
  - to check results
- This could be simplified by writing a method to do the asking
  - method returns boolean result
  - use call of query method as the loop condition

```
do { …
  … } while (Query.moreValues());
```

8.5 Choosing the Right Loop

- Determined by the nature of the problem
- Decisions to be made:

1. Counting or general loop?
    - Ask: does algorithm require counting through fixed range of values?
2. Which general loop?
    - pretest or posttest
    - forever loop with test in mid-loop

Introduction to Recursion

- We have seen one method call another method
  - most often the calling method is `main()`
- It is also possible for a method to call itself
  - this is known as recursion

Example: Factorial Problem Revisited

- Recall from section 5.3
  Given an integer n, calculate n-factorial
  1 * 2 * 3 * … * (n - 1)*n
- One way to define factorials is

$$n! = \begin{cases} 1 & n = 0 \\ n \times (n-1)! & n > 0 \end{cases}$$

This is a recursive definition

Recursive Definitions

$$n! = \begin{cases} 1 & n = 0 \\ n \times (n-1)! & n > 0 \end{cases}$$

- An operation is defined recursively if:
  - it has an anchor or base case
  - it has an inductive or recursive step where the current value produced define in terms of previously define results

Recursive Method for n!

```
public static int factorial(int n)
  {
    if (n == 0)
      return 1;
    else
      return  n * factorial(n - 1); }
```

Anchor case

Inductive step

- Consider: what happens when n < 0?
- Why is this called infinite recursion?

Example 2: Recursive Exponentiation

- Raising a number to an integer power can be also be done with recursion

$$x^n = \begin{cases} 1 & n = 0 \\ x^{n-1} \times x & n > 0 \end{cases}$$

Objects	Type	Kind	Movement	Name
base value	`double`	varying	received	`x`
exponent	`int`	varying	received	`n`
x^n	`double`	varying	returned	

Recursive Exponentiation Method

```
public static double power(double x, int n)
  {   if (n == 0)
          return 1.0;
      else if (n > 0)
          return power(x, n - 1)*x;
      else { theScreen.println("error");
                return 1.0; }
  }
```

Anchor case

Inductive step

What keeps this method from infinite recursion?

8.7 Graphical/Internet Java: A Guessing Game

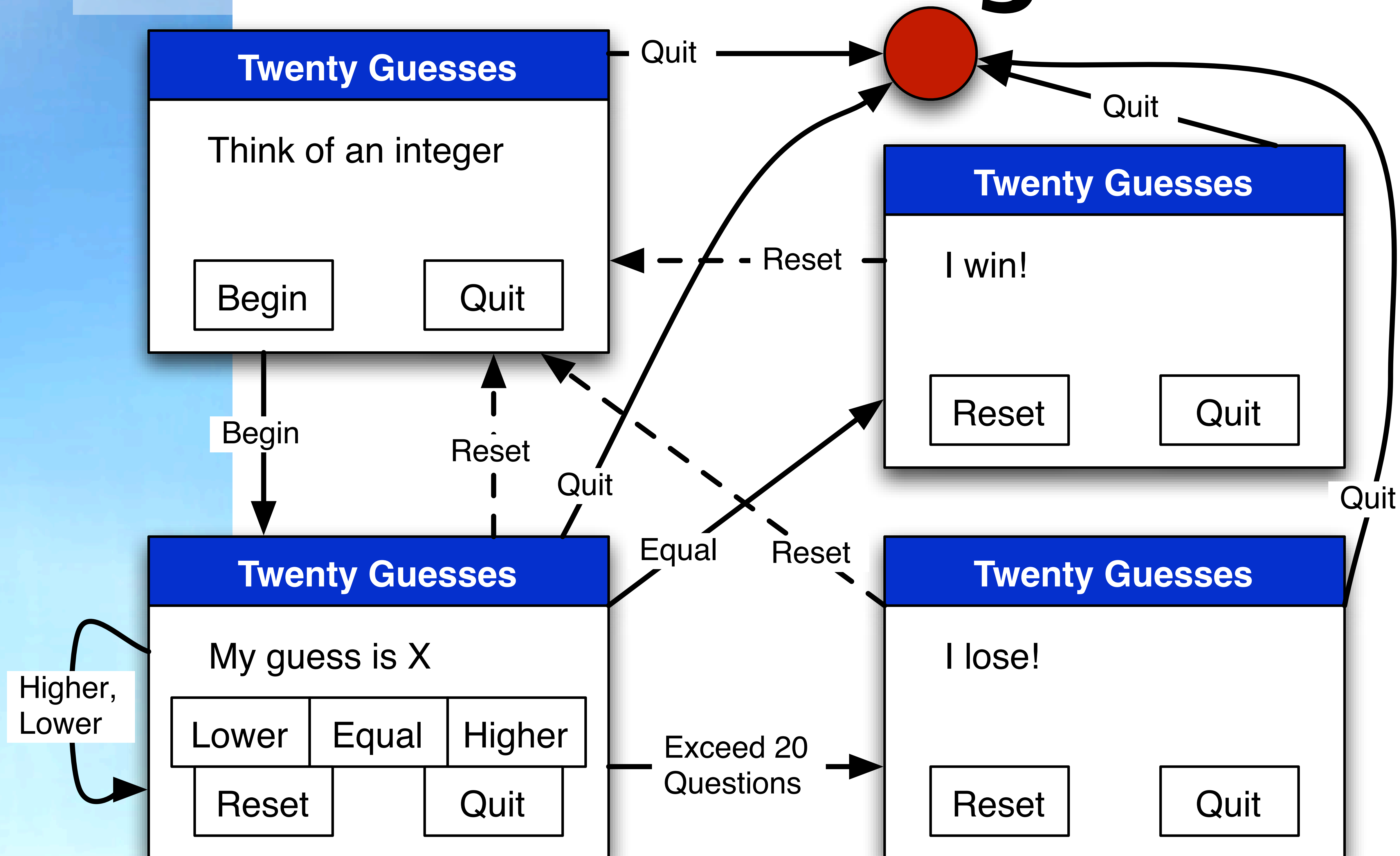
Twenty Guesses:

- One player thinks of an integer
- Second player allowed up to 20 guesses to determine the integer
- Incorrect guess:
  - tell the other player whether guess is high or low
  - other player uses this information to improve guess

Problem: Twenty Guesses

- We seek a program using GUI to play the role of the guesser
- Strategy used is binary-search
  - guesser knows high and low bounds of search
  - guesses half way
  - use high/low feedback to guess halfway of smaller bound

Behavior of Program: Transition Diagram



Objects

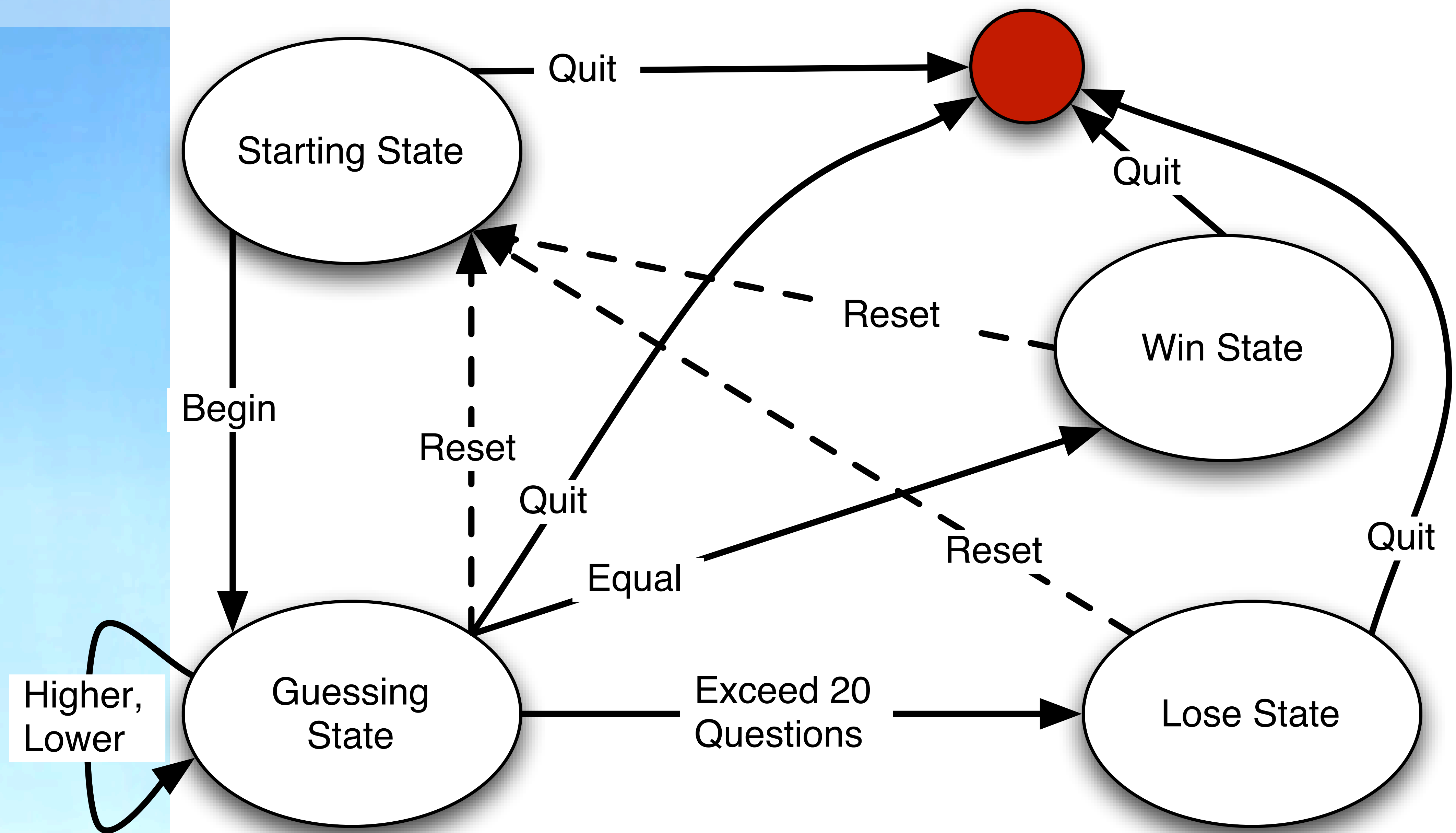
Object	Type	Kind	Name
program		varying	
prompt	`Jlabel`	varying	`myPromptLabel`
Prompt message	`String`	varying	
"Begin" button	`JButton`	varying	`myBeginButton`
"Quit" button	`JButton`	varying	`myQuitButton`
"Lower" button	`JButton`	varying	`myLowerButton`
"Equal" button	`JButton`	varying	`myEqualButton`
"Higher" button	`JButton`	varying	`myHigherButton`
"Reset" button	`JButton`	varying	`myResetButton`
a guess	`int`	varying	`myGuess`
Count of guesses	`int`	varying	`myGuessCount`
low bound	`int`	varying	`myLoBound`
high bound	`int`	varying	`myHiBound`

Operations

1. We need a constructor to build the GUI
2. An `actionPerformed()` method
    a. implement the `ActionListener` interface
    b. register itself as listener for each button
    c. send `addActionListener()` message to each button
3. A `main()` method
    a. create an instance of the class
    b. make it visible

State Diagram

- A transition diagram with most details removed



Coding

- Write a method for each state
  - define its appearance in that state
- `JButtons` have `setText()` method for setting label attribute
  - button with “Begin” in starting state has “Reset” in the other states
- Note full source code Figure 8.14

Applet Version of GUI Guessing Program

- Make the class extend `JApplet` instead of `CloseableFrame`
- Replace `main()` with non-static `init()` method
- Adjust dimensions of applet frame in HTML file to resemble frame for application

Part of the Picture: Intro to Algorithm Analysis

- How did Gauss figure the sum so quickly?
- Consider

  sum = 1 + 2 + … + 99 + 100

  sum = 100 + 99 + + 2 + 1
- Thus

  $$2 * \text{sum} = \underbrace{101 + 101 + \ldots + 101 + 101}_{\text{100 terms}}$$
- So

$$\text{sum} = \frac{100 \times 101}{2} = 5050$$

Analyzing the Algorithms

- In general, the formula is:

$$\text{sum} = \frac{n \times (n-1)}{2}$$

- This is more efficient than the looping algorithm
  - many less operations (additions, assignments, increments, comparisons)
  - This algorithm actually has same number of operations regardless of the value of n
- Important to analyze algorithms for efficiency
  - evaluate number of operations required