Five-Minute Review

1. What is a *class hierarchy*?
2. Which graphical *coordinate system* is used by Java (and most other languages)?
3. Why is a *collage* a good methapher for GObjects?
4. What is a *CFG*? What is it useful for?
5. What is the *language* defined by a CFG?

Five-Minute Review

1. What is an *expression*? A *term*?
2. What is a *variable declaration*?
3. What is an *assignment*?
4. What is *precedence*? *Associativity*?
5. How are expressions evaluated?

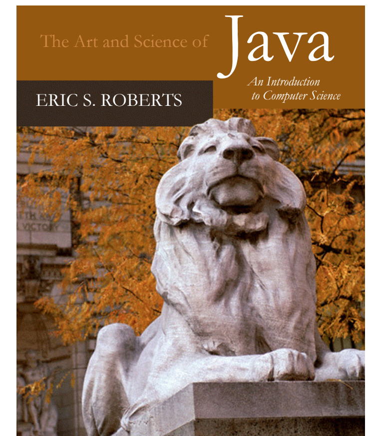
Programming – Lecture 4

Statement Forms (Chapter 4)

- Repeat N-Times Pattern
- Repeat-Until-Sentinel Pattern
- `if/else`
- `? :`
- `switch`
- `while`, `do while`
- Loop-and-a-Half Pattern
- `for`
- Animation

CHAPTER 4

Statement Forms



The statements was interesting but tough.

—Mark Twain, *The Adventures of Huckleberry Finn,* 1884

4.1 Statement types in Java

4.2 Control statements and problem solving

4.3 The **`if`** statement

4.4 The **`switch`** statement

4.5 The **`while`** statement

4.6 The **`for`** statement

Statement Types

Recall: Expressions, have type, are interested in *value*

Recall: Classes, Methods, Statements

Statements: units of run-time execution, are interested in *side effect*

Distinguish

- Expression statements
- Declaration statements
- Control flow statements
- Compound statements (blocks)

Expression Statements

The following *expressions*, terminated with semicolon, form *expression statements*

- *Assignment expressions*
  `aValue = 8933.234;`
- Any use of **`++`** or **`--`**
  `aValue++;`
- Method invocations
  `System.out.println("Hello!");`
- Object creation expressions
  `new Bicycle();`

Note: the book lists simple statements, defined as "expression ;", instead of listing expression/declaration statements. However, declarations are not expressions, and not all expressions (such as "17") lead to valid statements.

Statement Types in Java

- Programs in Java consist of a set of **classes**. Those classes contain **methods**, and each of those methods consists of a sequence of **statements**.
- Statements in Java fall into three basic types:
  - Simple statements (expression statements or declaration statements)
  - Compound statements
  - Control statements
- C**ompound statements** (also called **blocks**) consist of a sequence of statements enclosed in curly braces.
- **Control statements** fall into two categories:
  - **Conditional statements** that specify some kind of test
  - **Iterative statements** that specify repetition

Control Statements and Problem Solving

- Before looking at the individual control statement forms in detail, it helps to look more holistically at a couple of programs that make use of common control patterns.
- The next few slides extend the **`Add2Integers`** program from Chapter 2 to create programs that add longer lists of integers. These slides illustrate three different strategies:
  - Adding new code to process each input value
  - Repeating the input cycle a predetermined number of times
  - Repeating the input cycle until the user enters a special sentinel value

`if` Statement

```
if (condition) {
    statements to be executed if the condition is true
}

if (condition) {
    statements to be executed if the condition is true
} else {
    statements to be executed if the condition is false
}
```

Cascading `if` Statement

`if (`*condition*$_1$**`) {`**
    statements$_1$
`} else if (`*condition*$_2$**`) {`**
    statements$_2$
. . . *more* **`else`**/**`if`** *conditions* . . .
`} else {`
    statements$_{else}$
`}`

Dangling Else

```
if (x > 0)
  if (doprint)
    println("x positive");
else
  if (doprint)
     println("x not positive");
```

Coding Advice – Conditionals

- To avoid "dangling else":
  If an **`if`** has an **`else`** clause,
  always make **`if`** clause a block (add braces)
- In general, it is a good idea to use blocks with conditionals
- Possible exception: single statements, on same line

Bad:
```
if (x > 0)
    x++;
```

Ok:
```
if (x > 0) x++;
```

Good:
```
if (x > 0) {
    x++;
}
```

Sicherheitslücke

Apples furchtbarer Fehler

Ob Onlinebanking oder Facebook-Login, nichts ist sicher: Ein schwerer Fehler gefährdet Nutzer von iPhones, iPads und Mac-Rechnern. Verschlüsselte Web-Verbindungen lassen sich abfangen und manipulieren.

Von Ole Reißmann

sslKeyExchange.c

opensource.apple.com/source/Security/Security-55471/libsecurity_ssl/lib/sslKeyExchange.c

```
    hashOut.data = hashes + SSL_MD5_DIGEST_LEN;
hashOut.length = SSL_SHA1_DIGEST_LEN;
if ((err = SSLFreeBuffer(&hashCtx)) != 0)
    goto fail;

if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
    goto fail;
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;

    err = sslRawVerify(ctx,
```

Choosing between `if` and `if`/`else`

- As is true with most programming constructs, there is no hard-and-fast rule that will tell you whether you need the basic **`if`** statement or the **`if`**/**`else`** form.
- The best general guideline is to think about the English description of the problem you are trying to solve. If that description contains the words *else* or *otherwise,* there is a good chance that the solution will use the **`else`** keyword.
- As an example, suppose that you wanted to change the **`AverageList`** program so that it didn't include any zero values in the average. Here, you need a simple **`if`** statement that makes sure that the score is not equal to 0.
- If you also wanted to count the nonzero scores, you would need to add an **`else`** clause to increment a counter variable in the case that **`value`** was equal to 0.

Common Forms of the **`if`** Statement

The examples in the book use the **`if`** statement in the following forms:

Single line **`if`** statement

```
if (condition) statement
```

Multiline **`if`** statement with curly braces

```
if (condition) {
    statement
    . . . more statements . . .
}
```

`if`/**`else`** statement with curly braces

```
if (condition) {
    statements_true
} else {
    statements_false
}
```

Cascading **`if`** statement

```
if (condition_1) {
    statements_1
} else if (condition_2) {
    statements_2
. . . more else/if conditions . . .
} else {
    statements_else
}
```

`?:` Operator

$$condition \; \texttt{?} \; expression_{\mathrm{true}} \; \texttt{:} \; expression_{\mathrm{false}}$$

`switch` Statement

```
switch ( expression ) {
  case v1:
    statements to be executed if expression = v1
    break;
  case v2:
    statements to be executed if expression = v2
    break;
  . . . more case clauses if needed . . .
  default:
    statements to be executed if no values match
    break;
}
```

```
public void run() {
   println("This program shows the number of days in a month.");
   int month = readInt("Enter numeric month (Jan=1): ");
   switch (month) {
      case 2:
         println("28 days (29 in leap years)");
         break;
      case 4: case 6: case 9: case 11:
         println("30 days");
         break;
      case 1: case 3: case 5: case 7: case 8: case 10: case 12:
         println("31 days");
         break;
      default:
         println("Illegal month number");
         break;
   }
}
```

Note: the Eclipse default format looks different:

- Cases not indented
- New line for each case

Coding Advice – Switch Statement

- A popular programming mistake: forgetting the **`break`** statement.
- If you really want a "non-trivial" *fall-through case*, indicate so with comment
- Always include **`default`**

```
switch (var) {
// Trivial fall-through from 1 to 2
  case 1: case 2:
    println("var is 1 or 2");
    break;
  case 3:
    println("var is 3");
    // no break here
  case 4:
    println("var is 3 or 4");
    break;
  default:
    println("var is not 1 ... 4");
    break;
}
```

The **Add4Integers** Program

- If you don't have access to control statements, the only way you can increase the number of input values is to add a new statement for each one, as in the following example:

```
public class Add4Integers extends ConsoleProgram {
   public void run() {
      println("This program adds four numbers.");
      int n1 = readInt("Enter n1: ");
      int n2 = readInt("Enter n2: ");
      int n3 = readInt("Enter n3: ");
      int n4 = readInt("Enter n4: ");
      int total = n1 + n2 + n3 + n4;
      println("The total is " + total + ".");
   }
}
```

- This strategy, however, is difficult to generalize and would clearly be cumbersome if you needed to add 100 values.

Add4Integers

```
public class Add4Integers extends ConsoleProgram
{
    public void run() {
        println("This program adds four numbers.");
        int n1 = readInt("Enter n1: ");
        int n2 = readInt("Enter n2: ");
        int n3 = readInt("Enter n3: ");
        int n4 = readInt("Enter n4: ");
        int total = n1 + n2 + n3 + n4;
        println("The total is " + total + ".");
    }
}
```

The Repeat-N-Times Idiom

One strategy for generalizing the addition program is to use the Repeat-N-Times idiom, which executes a set of statements a specified number of times. The general form of the idiom is

```
for (int i = 0; i < repetitions; i++) {
    statements to be repeated
}
```

As is true for all idiomatic patterns in this book, the italicized words indicate the parts of the pattern you need to change for each application. To use this pattern, for example, you need to replace *repetitions* with an expression giving the number of repetitions and include the statements to be repeated inside the curly braces.

The Repeat-N-Times Idiom

One strategy for generalizing the addition program is to use the Repeat-N-Times idiom, which executes a set of statements a specified number of times. The general form of the idiom is

```
for (int i = 0; i < repetitions; i++) {
    statements to be repeated
}
```

The information about the number of repetitions is specified by the first line in the pattern, which is called the **header line**.

The statements to be repeated are called the **body** of the `for` statement and are indented with respect to the header line.

A control statement that repeats a section of code is called a **loop**.

Each execution of the body of a loop is called a **cycle**.

Repeat-N-Times Idiom

```
for (int i = 0; i < repetitions; i++) {
      statements to be repeated
}
```

AddNIntegers

```
public class AddNIntegers extends ConsoleProgram
{
   private static final int N = 100;

   public void run() {
      println("This program adds " + N +
            " numbers.");
      int total = 0;
      for (int i = 0; i < N; i++) {
         int value = readInt(" ? ");
         total += value;
      }
      println("The total is " + total + ".");
   }
}
```

Repeat-Until-Sentinel Idiom

```
while (true) {
    prompt user and read in a value
    if (value == sentinel) break;
    rest of loop body
}
```

The Repeat-Until-Sentinel Idiom

A better approach for the addition program that works for any number of values is to use the Repeat-Until-Sentinel idiom, which executes a set of statements until the user enters a specific value called a **sentinel** to signal the end of the list:

```
while (true) {
    prompt user and read in a value
    if (value == sentinel) break;
    rest of loop body
}
```

You should choose a sentinel value that is not likely to occur in the input data. It also makes sense to define the sentinel as a named constant to make the sentinel value easy to change.

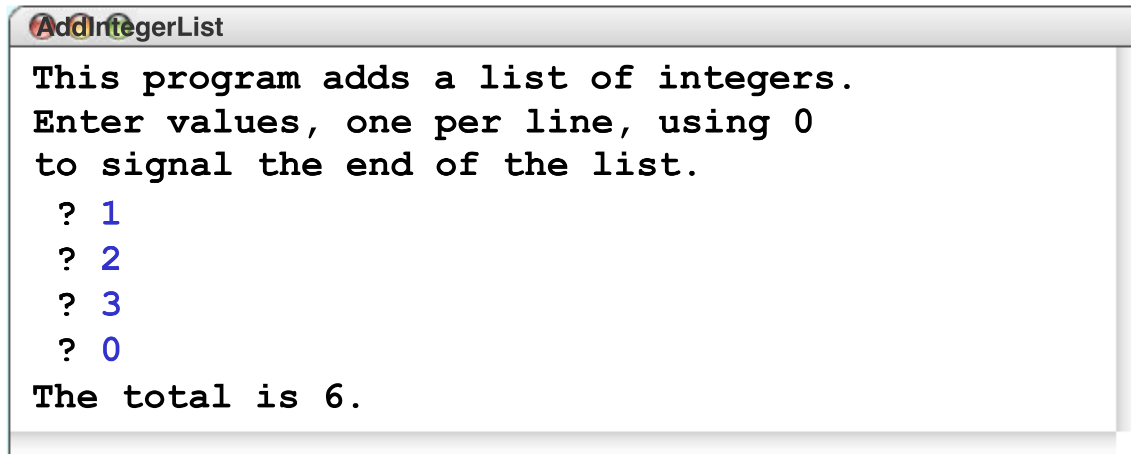
AddIntegerList

```
public class AddIntegerList extends ConsoleProgram {
   private static final int SENTINEL = 0;

   public void run() {
      println("This program adds a list of integers.");
      println("Enter values, one per line, using " +
            SENTINEL);
      println("to signal the end of the list.");
      int total = 0;
      while (true) {
         int value = readInt(" ? ");
         if (value == SENTINEL) break;
         total += value;
      }
      println("The total is " + total + ".");
   }
}
```

AddIntegerList

```
public void run() {
   println("This program adds a list of integers.");
   println("Enter values, one per line, using " + SENTINEL);
   println("to signal the end of the list.");
   int total = 0;
   while (true) {
      int value = readInt(" ? ");
      if (value == SENTINEL) break;
      total += value;
   }
   println("The total is " + total + ".");
}
```

value	total
0	6



skip simulation

Exercise: Control Patterns

Using the **`AddIntegerList`** program as a model, write a new **`AverageList`** program that reads a set of integers from the user and displays their average. Because 0 values might well appear, for example, in an average of exam scores, change the sentinel value so that the input stops when the user enters `-1`. It is important to keep in mind that the average of a set of integers is not necessarily an integer.

The **`AverageList`** program will require the following changes:

- Convert the variable **`total`** to a **`double`** before computing the average
- Change the definition of the **`SENTINEL`** constant
- Keep a count of the number of input values along with the sum
- Update the user messages and program documentation

The `AverageList` Program

```
public class AverageList extends ConsoleProgram {
   public void run() {
      println("This program averages a list of numbers.");
      println("Enter values, one per line, using " + SENTINEL);
      println("to signal the end of the list.");
      int total = 0;
      int count = 0;
      while (true) {
         int value = readInt(" ? ");
         if (value == SENTINEL) break;
         total += value;
         count++;
      }
      double average = (double) total / count;
      println("The average is " + average + ".");
   }

   private static final int SENTINEL = -1;
}
```

The `while` Statement

The **`while`** statement is the simplest of Java's iterative control statements and has the following form:

```
while ( condition ) {
        statements to be repeated
}
```

When Java encounters a **`while`** statement, it begins by evaluating the condition in parentheses, which must have a **`boolean`** value.

If the value of *condition* is **`true`**, Java executes the statements in the body of the loop.

At the end of each cycle, Java reevaluates *condition* to see whether its value has changed. If *condition* evaluates to **`false`**, Java exits from the loop and continues with the statement following the closing brace at the end of the **`while`** body.

`while`, `do while` Statements

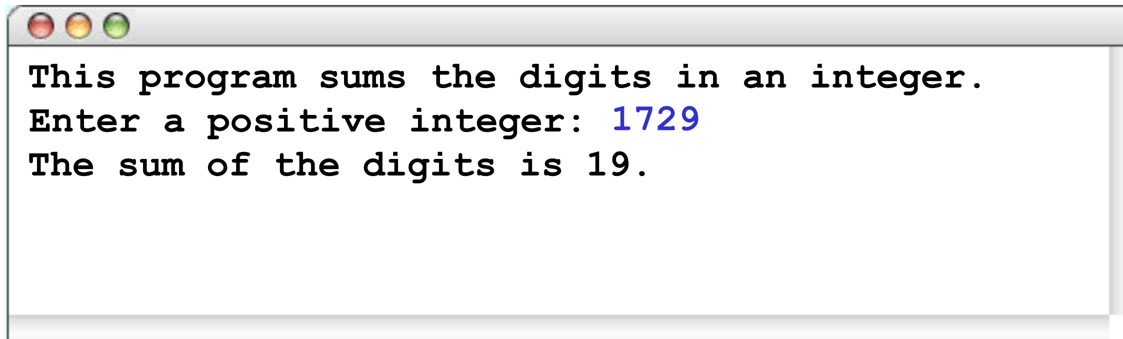
```
while ( condition ) {
        statements to be repeated
}
```

```
do {
        statements to be repeated
} while ( condition )
```

DigitSum

```
public void run() {
   println("This program sums the digits in an integer.");
   int n = readInt("Enter a positive integer: ");
   int dsum = 0;
   while ( n > 0 ) {
      dsum += n % 10;
      n /= 10;
   }
   println("The sum of the digits is " + dsum);
}
```

n	dsum
0	19



skip simulation

The Loop-and-a-Half Pattern

The **`while`** statement in Java always tests the condition at the beginning of each cycle of the loop. Sometimes, however, you need to perform some computation *before* you can make the test. In those situations, the **loop-and-a-half** pattern is very useful:

```
while (true) {
    computation necessary to make the test
    if (test for completion) break;
    computation for the rest of the loop cycle
}
```

Because the condition in the **`while`** statement itself is always **`true`**, this loop would continue forever without some other strategy to indicate completion. The loop-and-a-half pattern uses the **`if`** and **`break`** statements to exit the loop. When the test for completion becomes **`true`**, Java executes the **`break`** statement, which causes the loop to exit, skipping the rest of the cycle.

Loop-and-a-Half Pattern

```
while (true) {
    computation necessary to make the test
    if (test for completion) break;
    computation for the rest of the loop cycle
}
```

Repeat-Until-Sentinel Revisited

The repeat-until-sentinel pattern presented at the beginning of the chapter is an example of the loop-and-a-half pattern.

```
while (true) {
    prompt user and read in a value
    if (value == sentinel) break;
    rest of loop body
}
```

Although it might at first seem as if this pattern is more complex than the more basic form of the **while** statement, it turns out to be considerably easier for most students to use. Several studies have demonstrated that students who use the loop-and-a-half pattern are far more likely to write correct code than those who don't.

Repeat-Until-Sentinel Revisited

```
while (true) {
    prompt user and read in a value
    if (value == sentinel) break;
    rest of loop body
}
```

`for` Statement

```
for ( init ; test ; step ) {
    statements to be repeated
}
```

Java evaluates a **`for`** statement by executing the following steps:

1. Evaluate *init*, which typically declares a **control variable**.
2. Evaluate *test* and exit from the loop if the value is **`false`**.
3. Execute the statements in the body of the loop.
4. Evaluate *step,* which usually updates the control variable.
5. Return to step 2 to begin the next loop cycle.

The **`for`** Statement

The **`for`** statement in Java is a particularly powerful tool for specifying the control structure of a loop independently from the operations the loop body performs. The syntax looks like this:

```
for ( init ; test ; step ) {
    statements to be repeated
}
```

Java evaluates a **`for`** statement by executing the following steps:

1. Evaluate *init*, which typically declares a **control variable**.
2. Evaluate *test* and exit from the loop if the value is **`false`**.
3. Execute the statements in the body of the loop.
4. Evaluate *step,* which usually updates the control variable.
5. Return to step 2 to begin the next loop cycle.

Comparing **`for`** and **`while`**

The **`for`** statement

```
for ( init ; test ; step ) {
    statements to be repeated
}
```

is functionally equivalent to the following code using **`while`**:

```
init;
while ( test ) {
    statements to be repeated
    step;
}
```

The advantage of the **`for`** statement is that everything you need to know to understand how many times the loop will run is explicitly included in the header line.

for

```
for ( init ; test ; step ) {
    statements to be repeated
}
```

Equivalent to:

```
{
  init ;
  while ( test ) {
    statements to be repeated
    step ;
  }
}
```

Exercise: Reading `for` Statements

Describe the effect of each of the following **`for`** statements:

1.
```
for (int i = 1; i <= 10; i++)
```

This statement executes the loop body ten times, with the control variable **`i`** *taking on each successive value between 1 and 10.*

2.
```
for (int i = 0; i < N; i++)
```

This statement executes the loop body **`N`** *times, with* **`i`** *counting from* **`0`** *to* **`N-1`**. *This version is the standard Repeat-N-Times idiom.*

3.
```
for (int n = 99; n >= 1; n -= 2)
```

This statement counts backward from 99 to 1 by twos.

4.
```
for (int x = 1; x <= 1024; x *= 2)
```

This statement executes the loop body with the variable **`x`** *taking on successive powers of two from 1 up to 1024.*

Exercise

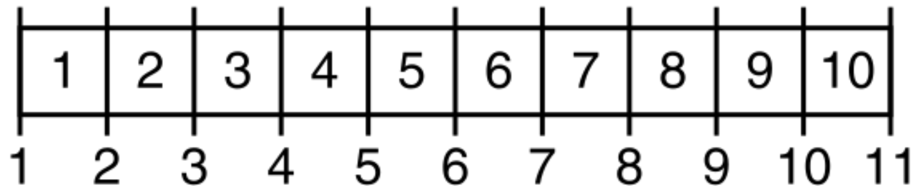
```
for (int i = 1; i <= 10; i++)

for (int i = 0; i < N; i++)

for (int n = 99; n >= 1; n -= 2)

for (int x = 1; x <= 1024; x *= 2)
```

Fence-Post Problem

In the araeostylos [style of temple] it is only necessary to preserve, in a peripteral building, twice the number of intercolumniations on the flanks that there are in front, so that the length may be twice the breadth. Those who use twice the number of columns for the length, appear to ***err****, because they thus make one intercolumniation more than should be used.*

Vitruvius, *On Architecture*

If you build a straight fence 30 meters long with posts spaced 3 meters apart, how many posts do you need?



Coding Advice: Beware of the *Off-By-One Error*

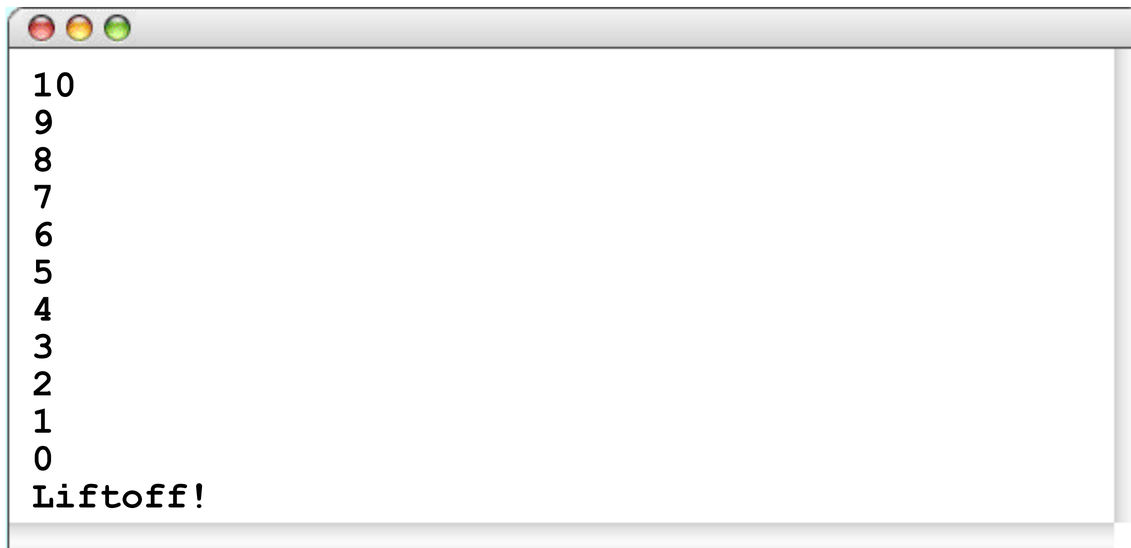
http://www.dsm.fordham.edu/~moniot/Opinions/fencepost-error-history.shtml
https://en.wikipedia.org/wiki/Off-by-one_error

Countdown

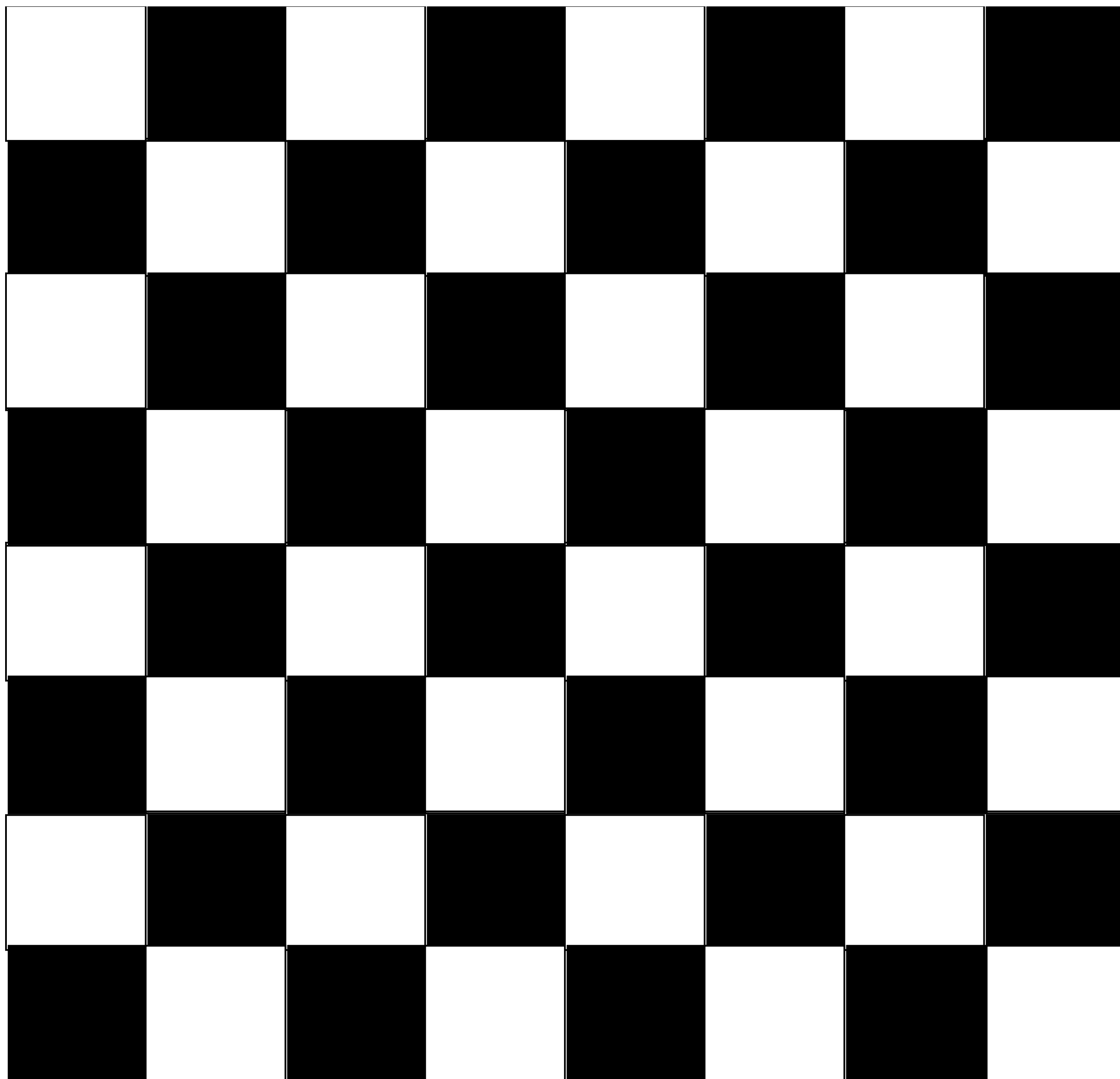
```
public void run() {
   for ( int t = 10 ; t >= 0 ; t-- ) {
      println(t);
   }
   println("Liftoff!");
}
```

t

−1



skip simulation



Nested `for` Statements

- The body of a control statement can contain other statements. Such statements are said to be **nested**.
- Many applications require nested **`for`** statements. The next slide, for example, shows a program to display a standard checkerboard in which the number of rows and number of columns are given by the constants **`N_ROWS`** and **`N_COLUMNS`**.
- The **`for`** loops in the **`Checkerboard`** program look like this:

```
for (int i = 0; i < N_ROWS; i++) {
    for (int j = 0; j < N_COLUMNS; j++) {
        Display the square at row i and column j.
    }
}
```

- Because the entire inner loop runs for each cycle of the outer loop, the program displays **`N_ROWS`** × **`N_COLUMNS`** squares.

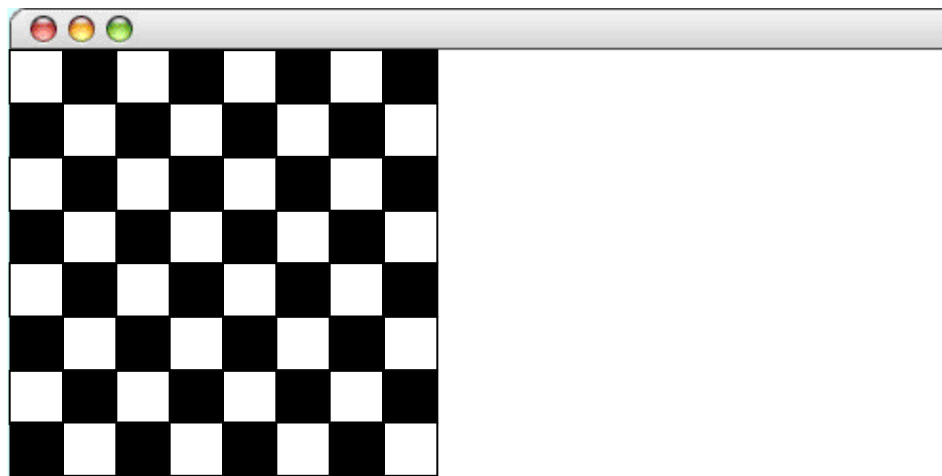
Nested `for`

```
for (int i = 0; i < N_ROWS; i++) {
    for (int j = 0; j < N_COLUMNS; j++) {
        Display the square at row i and column j.
    }
}
```

Checkerboard

```
public void run() {
   double sqSize = (double) getHeight() / N_ROWS;
   for (int i = 0; i < N_ROWS; i++) {
      for (int j = 0; j < N_COLUMNS; j++) {
         double x = j * sqSize;
         double y = i * sqSize;
         GRect sq = new GRect(x, y, sqSize, sqSize);
         sq.setFilled((i + j) % 2 != 0);
         add(sq);
      }
   }
}
```

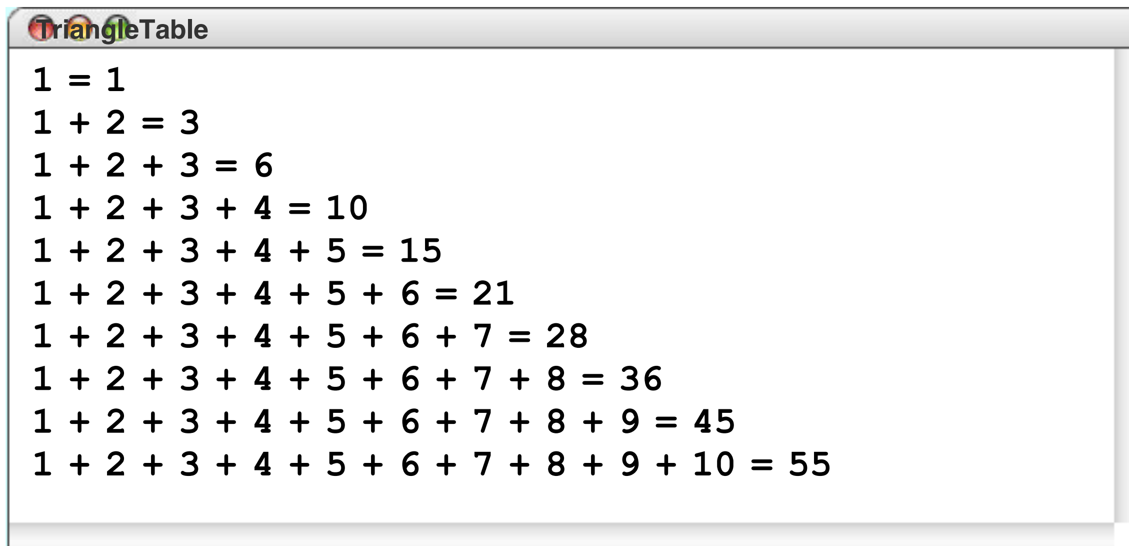
sqSize	i	j	x	y	sq
30.0	8	8	210.0	210.0	



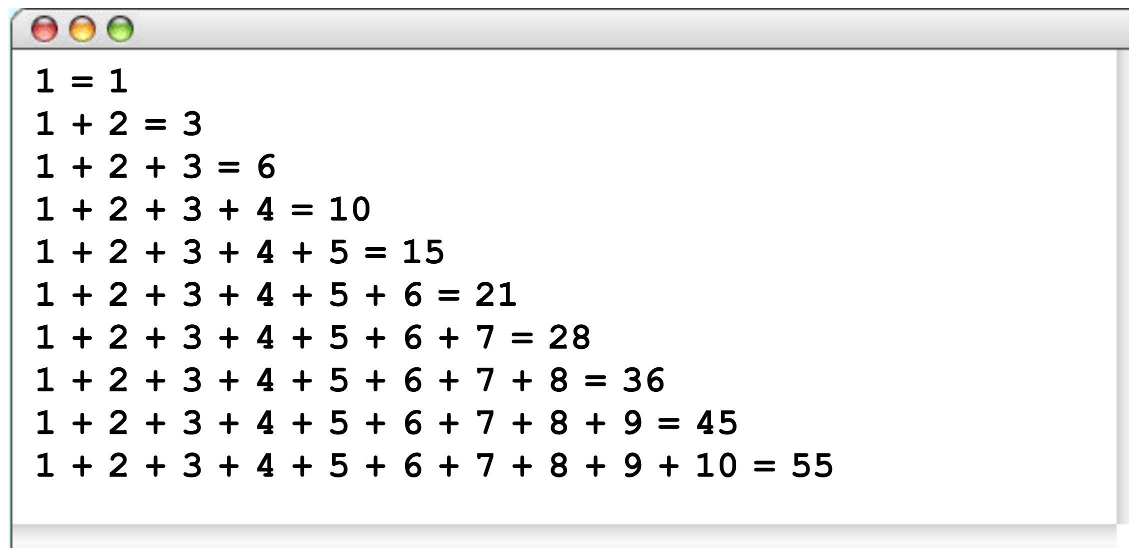
skip simulation

Exercise: Triangle Number Table

Write a program that duplicates the sample run shown at the bottom on this slide, which displays the sum of the first N integers for each value of N from 1 to 10. As the output suggests, these numbers can be arranged to form a triangle and are therefore called **triangle numbers**.



Exercise: Triangle Number Table



Design Issues: Triangle Number Table

As you think about the design of the **`TriangleTable`** program, it will help to keep the following thoughts in mind:

- The program involves two nested loops. The outer loop runs through each of the values of N from 1 to the maximum; the inner loop prints a series of values on each output line.
- The individual elements of each output line are easier to display if you call **`print`** instead of **`println`**. The **`print`** method is similar to **`println`** but doesn't return the cursor position to the beginning of the next line in the way that **`println`** does. Using **`print`** therefore makes it possible to string several output values together on the same line.
- The nth output line contains n values before the equal sign but only $n-1$ plus signs. Your program therefore cannot print a plus sign on each cycle of the inner loop but must instead skip one cycle.

TriangleTable

```
public class TriangleTable extends ConsoleProgram {

   private static final int MAX_VALUE = 10;

   public void run() {
      for (int n = 1; n <= MAX_VALUE; n++) {
         int total = 0;
         for (int i = 1; i <= n; i++) {
            if (i > 1) print(" + ");
            print(i);
            total += i;
         }
         println(" = " + total);
      }
   }
}
```

Simple Graphical Animation

The **`while`** and **`for`** statements make it possible to implement simple graphical animation. The basic strategy is to create a set of graphical objects and then execute the following loop:

```
for (int i = 0; i < N_STEPS; i++) {
   update the graphical objects by a small amount
   pause(PAUSE_TIME);
}
```

On each cycle of the loop, this pattern updates each animated object by moving it slightly or changing some other property of the object, such as its color. Each cycle is called a **time step**.

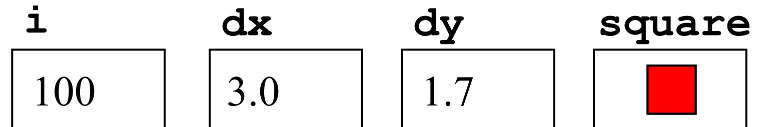
After each time step, the animation pattern calls **`pause`**, which delays the program for some number of milliseconds (expressed here as the constant **`PAUSE_TIME`**). Without the call to **`pause`**, the program would finish faster than the human eye can follow.

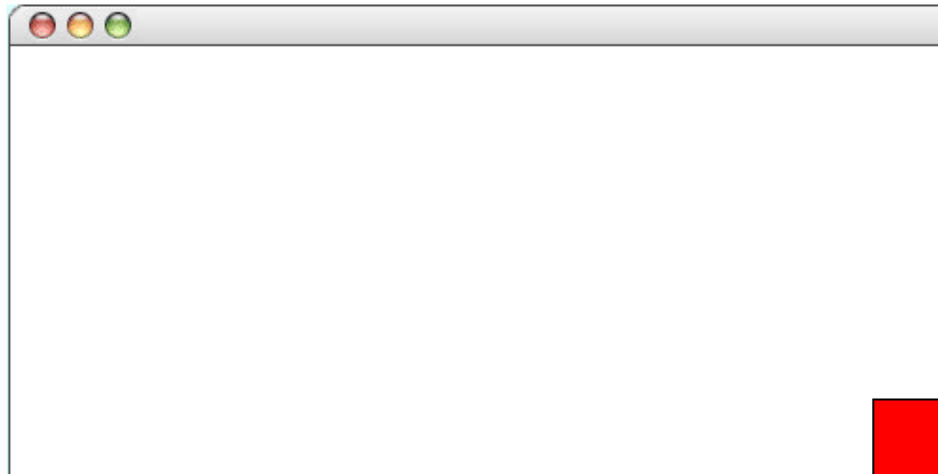
Simple Graphical Animation

```
for (int i = 0; i < N_STEPS; i++) {
    update the graphical objects by a small amount
    pause(PAUSE_TIME);
}
```

AnimatedSquare

```
public void run() {
   GRect square = new GRect(0, 0, SQUARE_SIZE, SQUARE_SIZE);
   square.setFilled(true);
   square.setFillColor(Color.RED);
   add(square);
   double dx = (getWidth() - SQUARE_SIZE) / N_STEPS;
   double dy = (getHeight() - SQUARE_SIZE) / N_STEPS;
   for (int i = 0; i < N_STEPS; i++) {
      square.move(dx, dy);
      pause(PAUSE_TIME);
   }
}
```





skip simulation

Summary

- *Compound statements* (*blocks*) provide structure and scoping
- Java *control statements* include **`if/else`**, **`switch`**, **`while`**, **`for`**
- Common *idioms* are Repeat N-Times, Repeat-Until-Sentinel, and Loop-and-a-Half
- Beware of missing **`break`**s
- Beware of the *off-by-one* error