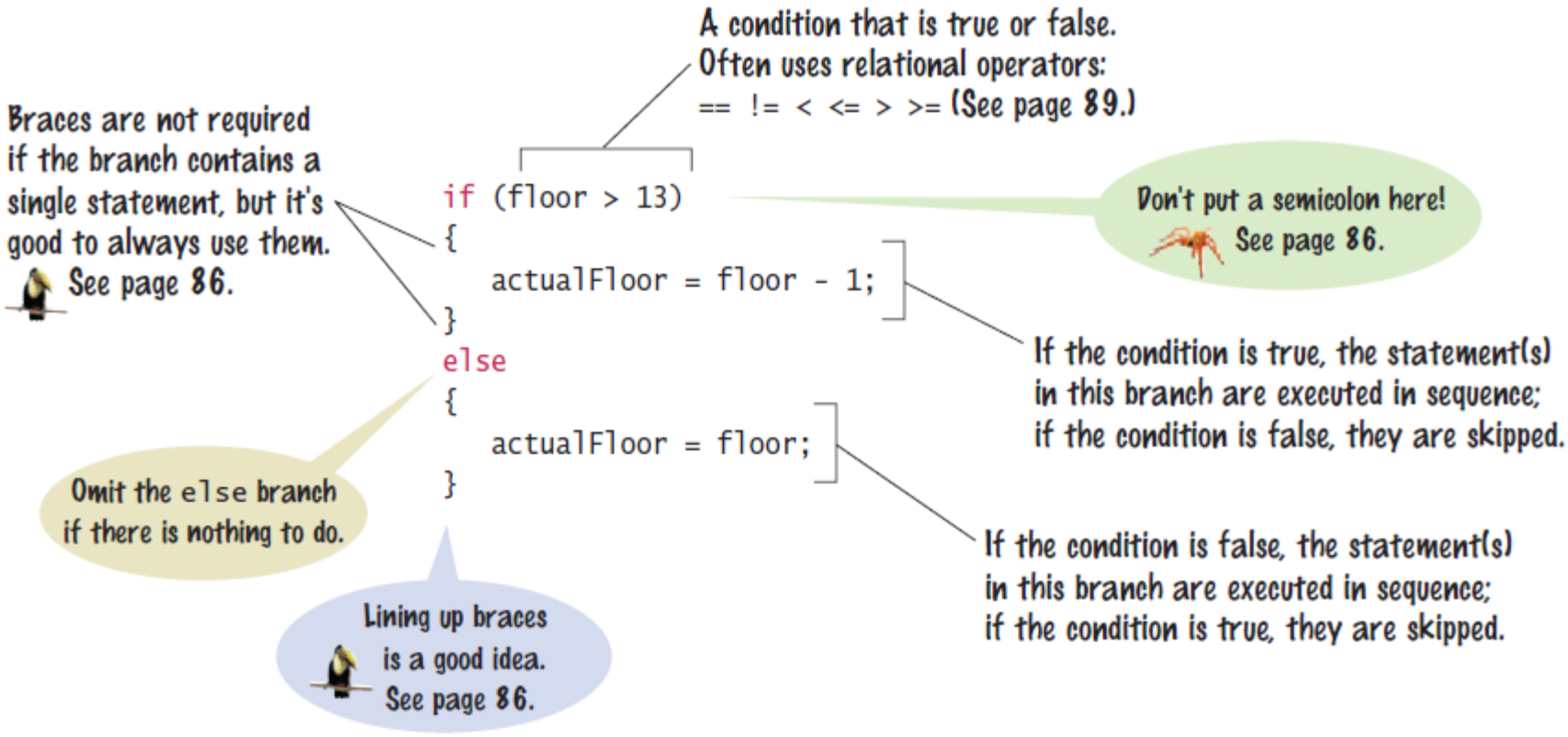
LECTURE 3

CONDITIONALS AND LOOPS

Lecture Goals

- To implement decisions using the `if` statement
- To compare integers, floats, and Strings
- To write statements using the Booleans
- To implement `while`, `for`, and `do` loops
- To hand-trace the execution of a program
- To become familiar with common loop algorithms
- To understand nested loops
- To implement programs that read and process data sets

Syntax 3.1: The `if` statement



Common Error 3.1

A semicolon after an `if` statement

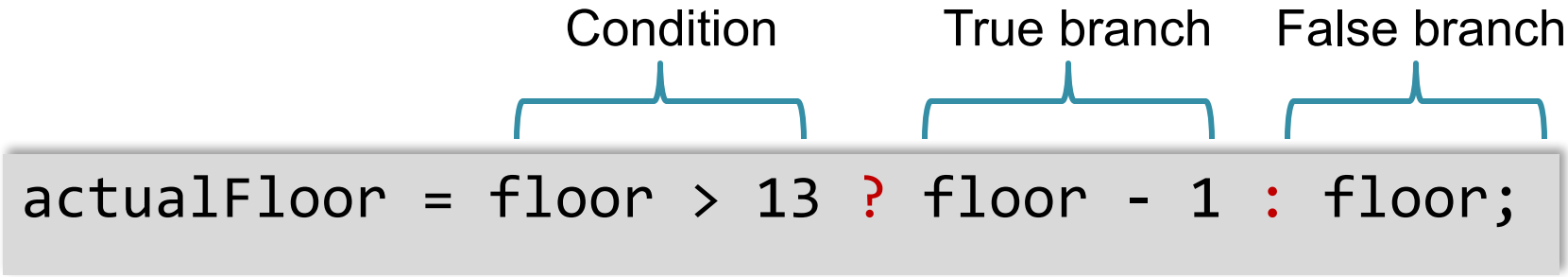
- It is easy to forget and add a semicolon after an `if` statement.
  - The true path is now the space just before the semicolon

```
if (floor > 13) ;
{
   floor--;
}
```

  - The 'body' (between the curly braces) will always be executed in this case

The Conditional Operator

- A ‘shortcut’ you may find in existing code
  - It is not used in this book



  - Includes all parts of an if-else clause, but uses:
    - `?` To begin the true branch
    - `:` To end the true branch and start the false branch

3.2 Comparing Numbers and Strings

- Every `if` statement has a condition
  - Usually compares two values with an operator

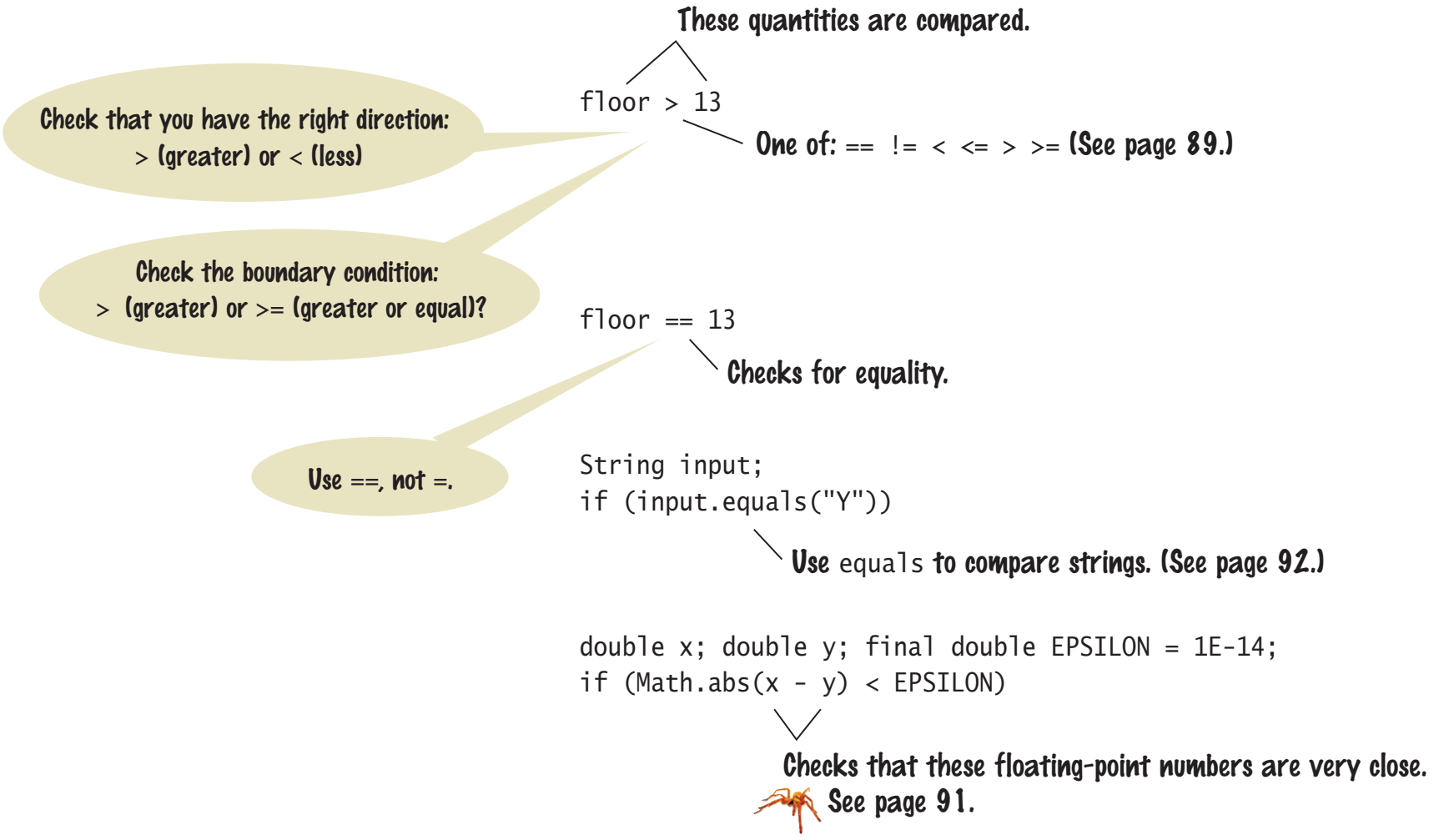
```
if (floor > 13)
   ..
if (floor >= 13)
   ..
if (floor < 13)
   ..
if (floor <= 13)
   ..
if (floor == 13)
   ..
```

Beware!

Table 1 Relational Operators

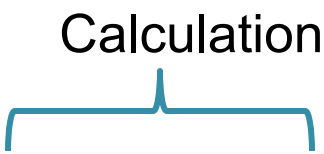
Java	Math Notation	Description
>	>	Greater than
>=	≥	Greater than or equal
<	<	Less than
<=	≤	Less than or equal
==	=	Equal
!=	≠	Not equal

Syntax 3.2: Comparisons



Operator Precedence

- The comparison operators have lower precedence than arithmetic operators
  - Calculations are done before the comparison
  - Normally your calculations are on the ‘right side’ of the comparison or assignment operator




```
actualFloor = floor + 1;
```

```
if (floor > height + 1)
```

Relational Operator Use (1)

Table 2 Relational Operator Examples

Expression
`3 <= 4`
`3 =< 4`
`3 > 4`
`4 < 4`
`4 <= 4`
`3 == 5 - 2`
`3 != 5 - 1`

Relational Operator Use (2)

Table 2 Relational Operator Examples

`3 = 6 / 2`
`1.0 / 3.0 == 0.333333333`
`"10" > 5`
`"Tomato".substring(0, 3).equals("Tom")`
`"Tomato".substring(0, 3) == ("Tom")`

Comparing Strings

- Strings and other Objects are 'special'
- Do not use the `==` operator with Strings
  - The following compares the locations of two strings, and not their contents

```
if (string1 == string2) ...
```

- Instead use the String's `equals` method:

```
if (string1.equals(string2)) ...
```

Common Error 3.2

- Comparison of Floating-Point Numbers
  - Floating-point numbers have limited precision
  - Round-off errors can lead to unexpected results

```
double r = Math.sqrt(2.0);
if (r * r == 2.0)
{
   System.out.println("Math.sqrt(2.0) squared is 2.0");
}
else
{
   System.out.println("Math.sqrt(2.0) squared is not 2.0
    but " + r * r);
}
```

```
Output:
Math.sqrt(2.0) squared is not 2.0 but 2.0000000000000044
```

The use of EPSILON

- Use a very small value to compare the difference if floating-point values are '*close enough*'
  - The magnitude of their difference should be less than some threshold
  - Mathematically, we would write that x and y are close enough if: $|x - y| < \varepsilon$

```
final double EPSILON = 1E-14;
double r = Math.sqrt(2.0);
if (Math.abs(r * r - 2.0) < EPSILON)
{
   System.out.println("Math.sqrt(2.0) squared is approx.
    2.0");
}
```

Common Error 3.3

- Using == to compare Strings
  - == compares the locations of the Strings
- Java creates a new String every time a new word inside double-quotes is used
  - If there is one that matches it exactly, Java re-uses it

```
String nickname = "Rob";
. . .
if (nickname == "Rob") // Test is true
```

```
String name = "Robert";
String nickname = name.substring(0, 3);
. . .
if (nickname == "Rob") // Test is false
```

Lexicographical Order

- To compare Strings in ‘dictionary’ order
  - When compared using `compareTo`, string1 comes:
    - Before `string2` if

```
string1.compareTo(string2) < 0
```

    - After `string2` if

```
string1.compareTo(string2) > 0
```

    - Equal to `string2` if

```
string1.compareTo(string2) == 0
```

  - Notes
    - All UPPERCASE letters come before lowercase
    - ‘space’ comes before all other printable characters
    - Digits (0-9) come before all letters
    - See Appendix A for the Basic Latin Unicode (ASCII) table

3.3 Multiple Alternatives

- What if you have more than two branches?
- Count the branches for the following earthquake effect example:
  - 8 (or greater)
  - 7 to 7.99
  - 6 to 6.99
  - 4.5 to 5.99
  - Less than 4.5

When using multiple `if` statements, test general conditions after more specific conditions.

Table 3 Richter Scale

Value	Effect
8	Most structures fall
7	Many buildings destroyed
6	Many buildings considerably damaged, some collapse
4.5	Damage to poorly constructed buildings

if, else if multiway branching

```
if (richter >= 8.0)   // Handle the 'special case' first
{
  System.out.println("Most structures fall");
}
else if (richter >= 7.0)
{
  System.out.println("Many buildings destroyed");
}
else if (richter >= 6.0)
{
  System.out.println("Many buildings damaged, some collapse");
}
else if (richter >= 4.5)
{
  System.out.println("Damage to poorly constructed buildings");
}
else    // so that the 'general case' can be handled last
{
  System.out.println("No destruction of buildings");
}
```

What is wrong with this code?

```
if (richter >= 8.0)
{
  System.out.println("Most structures fall");
}
if (richter >= 7.0)
{
  System.out.println("Many buildings destroyed");
}
if (richter >= 6.0)
{
  System.out.println("Many buildings damaged, some collapse");
}
if (richter >= 4.5)
{
  System.out.println("Damage to poorly constructed buildings");
}
```

Another way to multiway branch

- The `switch` statement chooses a `case` based on an integer value.
- `break` ends each `case`
- `default` catches all other values

```
int digit = . . .;
switch (digit)
{
   case 1: digitName = "one";   break;
   case 2: digitName = "two";   break;
   case 3: digitName = "three"; break;
   case 4: digitName = "four";  break;
   case 5: digitName = "five";  break;
   case 6: digitName = "six";   break;
   case 7: digitName = "seven"; break;
   case 8: digitName = "eight"; break;
   case 9: digitName = "nine";  break;
   default: digitName = "";     break;
}
```

If the `break` is missing, the case *falls through* to the next case's statements.

3.4 Nested Branches

- You can *nest* an if inside either branch of an if statement.
- Simple example: Ordering drinks
  - Ask the customer for their drink order
  - `if` customer orders wine
    - Ask customer for ID
    - `if` customer's age is 21 or over
      - Serve wine
    - Else
      - Politely explain the law to the customer
  - Else
    - Serve customers a non-alcoholic drink

Common Error 3.4

The Dangling `else` Problem

- When an `if` statement is nested inside another `if` statement, the following can occur:

```
double shippingCharge = 5.00; // $5 inside continental U.S.
if (country.equals("USA"))
  if (state.equals("HI"))
    shippingCharge = 10.00;   // Hawaii is more expensive
else // Pitfall!
  shippingCharge = 20.00;     // As are foreign shipment
```

- The indentation level suggests that the `else` is related to the `if` country (“USA”)
  - Else clauses always associate to the closest `if`

Enumerated Types

- Java provides an easy way to name a finite list of values that a variable can hold
  - It is like declaring a new type, with a list of possible values

```
public enum FilingStatus {
 SINGLE, MARRIED,MARRIED_FILING_SEPARATELY }
```

  - You can have any number of values, but you must include them all in the enum declaration
  - You can declare variables of the enumeration type:

```
FilingStatus status = FilingStatus.SINGLE;
```

  - And you can use the comparison operator with them:

```
if (status == FilingStatus.SINGLE) . . .
```

3.7 Boolean Variables

- Boolean Variables
  - A Boolean variable is often called a flag because it can be either up (`true`) or down (`false`)
  - `boolean` is a Java data type
    - `boolean failed = true;`
    - Can be either `true` or `false`
- Boolean Operators: `&&` and `||`
  - They combine multiple conditions
  - `&&` is the *and* operator
  - `||` is the *or* operator

Character Testing Methods

- The `Character` class has a number of handy methods that return a `boolean` value:

```
if (Character.isDigit(ch))
{
   ...
}
```

Character Testing Methods

Method	Examples of Accepted Characters
isDigit	0, 1, 2
isLetter	A, B, C, a, b, c
isUpperCase	A, B, C
isLowerCase	a, b, c
isWhiteSpace	space, newline, tab

Combined Conditions: &&

- Combining two conditions is often used in range checking
  - Is a value between two other values?
- Both sides of the *and* must be true for the result to be true

```
if (temp > 0 && temp < 100)
{
   System.out.println("Liquid");
}
```

A	B	A && B
true	true	true
true	false	false
false	true	false
false	false	false

Combined Conditions: ||

- If only one of two conditions need to be true
  - Use a compound conditional with an or:

```
if (balance > 100 || credit > 100)
{
  System.out.println("Accepted");
}
```

- If either is true
  - The result is true

A	B	A || B
true	true	true
true	false	true
false	true	true
false	false	false

The *not* Operator: !

- If you need to invert a boolean variable or comparison, precede it with !

```
if (!attending || grade < 60)
{
   System.out.println("Drop?");
}
```

```
if (attending && !(grade < 60))
{
   System.out.println("Stay");
}
```

A	!A
true	false
false	true

- If using !, try to use simpler logic:

```
if (attending && (grade >= 60))
```

Boolean Operator Examples

Table 5 Boolean Operator Examples

Expression
`0 < 200 && 200 < 100`
`0 < 200 || 200 < 100`
`0 < 200 || 100 < 200`
`0 < x && x < 100 || x == -1`
`0 < x < 100`

Boolean Operator Examples

Table 5 Boolean Operator Examples

`x && y > 0`
`!(0 < 200)`
`frozen == true`
`frozen == false`

Common Error 3.5

- Combining Multiple Relational Operators

```
if (0 <= temp <= 100)  // Syntax error!
```

  - This format is used in math, but not in Java!
  - It requires two comparisons:

```
if (0 <= temp && temp <= 100)
```

- This is also not allowed in Java:

```
if (input == 1 || 2)  // Syntax error!
```

  - This also requires two comparisons:

```
if (input == 1 || input == 2)
```

Common Error 3.6

Confusing && and || Conditions

- It is a surprisingly common error to confuse && and || conditions.
- A value lies between 0 and 100 if it is at least 0 ***and*** at most 100.
- It lies outside that range if it is less than 0 ***or*** greater than 100.
- There is no golden rule; you just have to think carefully.

Short-Circuit Evaluation: &&

- Combined conditions are evaluated from left to right
  - If the left half of an *and* condition is false, why look further?

```
if (temp > 0 && temp < 100)
{
   System.out.println("Liquid");
}
```

- A useful example:

```
if (quantity > 0 && price / quantity < 10)
```

Short-Circuit Evaluation: ||

- If the left half of the *or* is true, why look further?

```
if (temp <= 0 || temp >= 100)
{
  System.out.println("Not Liquid");
}
```

- Java doesn't!
- Don't do these second:
  - Assignment
  - Output

3.8 Input Validation

- Accepting user input is dangerous
  - Consider the Elevator program:
  - The user may input an invalid character or value
  - Must be an integer
    - Scanner can help!
    - `hasNextInt`
      - True if integer
      - False if not

```
if (in.hasNextInt())
{
   int floor = in.nextInt();
   // Process the input value
}
else
{
   System.out.println("Not integer.");
}
```

  - Then range check value
  - We expect a floor number to be between 1 and 20
    - NOT 0, 13 or > 20

ElevatorSimulation2.java

```
 7  public class ElevatorSimulation2
 8  {
 9     public static void main(String[] args)
10     {
11        Scanner in = new Scanner(System.in);
12        System.out.print("Floor: ");
13        if (!in.hasNextInt())
14        {
15           // Now we know that the user entered an integer
16
17           int floor = in.nextInt();
18
19           if (floor == 13)
20           {
21              System.out.println("Error: There is no thirteenth floor.");
22           }
23           else if (floor <= 0 || floor > 20)
24           {
25              System.out.println("Error: The floor must be between 1 and 20.");
26           }
27           else
28           {
29              // Now we know that the input is valid
```

Input value validity checking

Input value range checking

ElevatorSimulation2.java

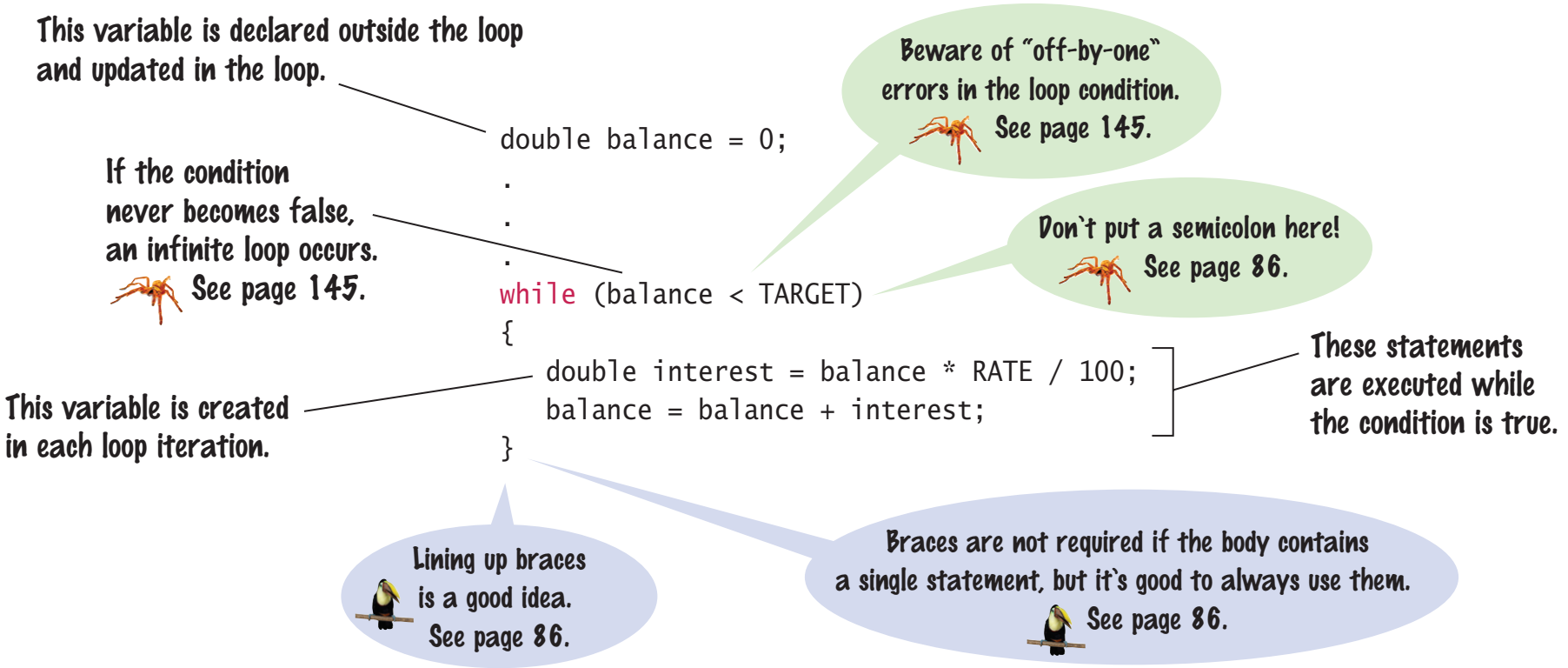
```
30
31             int actualFloor = floor;
32             if (floor > 13)
33             {
34                actualFloor = floor - 1;
35             }
36
37             System.out.println("The elevator will travel to the actual floor "
38                + actualFloor);
39          }
40       }
41       else
42       {
43          System.out.println("Error: Not an integer.");
44       }
45    }
46 }
```

Program Run

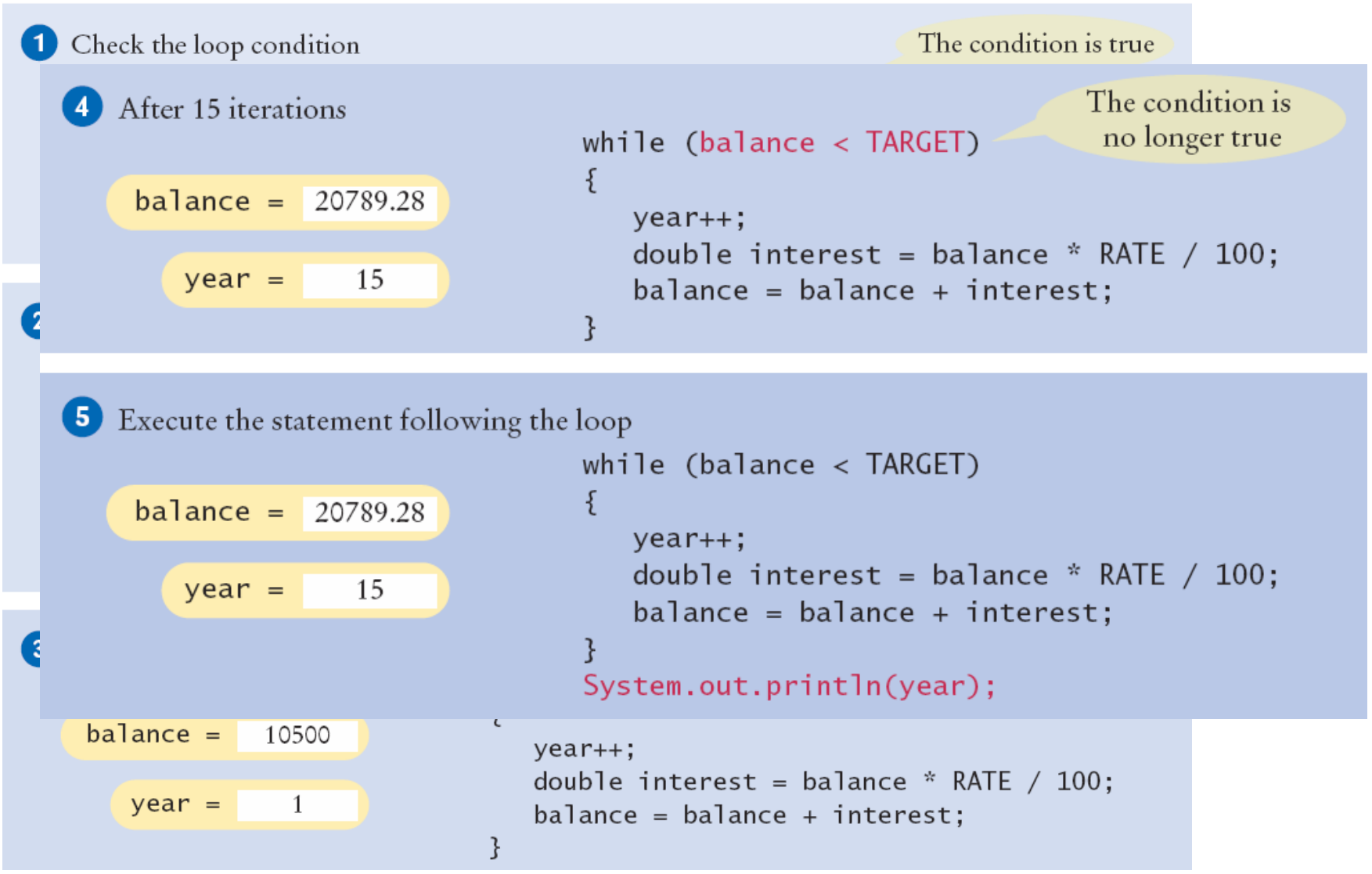
```
Floor: 13
Error: There is no thirteenth floor.
```

LOOPS

Syntax 4.1: `while` Statement



Execution of the Loop



DoubleInvestment.java

```
1  /**
2     This program computes the time required to double an investment.
3  */
4  public class DoubleInvestment
5  {
6     public static void main(String[] args)
7     {
8        final double RATE = 5;
9        final double INITIAL_BALANCE = 10000;
10       final double TARGET = 2 * INITIAL_BALANCE;
11
12       double balance = INITIAL_BALANCE;
13       int year = 0;
14
15       // Count the years required for the investment to double
16
17       while (balance < TARGET)
18       {
19          year++;
20          double interest = balance * RATE / 100;
21          balance = balance + interest;
22       }
23
24       System.out.println("The investment doubled after "
25          + year + " years.");
26    }
27 }
```

Declare and initialize a variable outside of the loop to count `years`

Increment the `years` variable each time through

Program Run

```
The investment doubled after 15 years.
```

`while` Loop Examples (1)

Loop	Output	Explanation

`i = 0; sum = 0;`
`while (sum < 10)`
`{`
`  i++; sum = sum + i;`
`  Print i and sum;`
`}`		

while Loop Examples (2)

Loop	Output	Explanation

`i = 0; sum = 0;`
`while (sum >= 10)`
`{`
`  i++; sum = sum + i;`
`  Print i and sum;`
`}`		

Common Error 4.1

- Don't think "Are we there yet?"
  - The loop body will only execute if the test condition is **True**.
  - "Are we there yet?" should continue if **False**
  - If `bal` should grow until it reaches `TARGET`
    - Which version will execute the loop body?

```
while (bal < TARGET)
{
   year++;
   interest = bal * RATE;
   bal = bal + interest;
}
```

```
while (bal >= TARGET)
{
   year++;
   interest = bal * RATE;
   bal = bal + interest;
}
```

Common Error 4.2

- Infinite Loops
  - The loop body will execute until the test condition becomes **False**.
  - What if you forget to update the test variable?
    - `bal` is the test variable (`TARGET` doesn't change)
    - You will loop forever! (or until you stop the program)

```
while (bal < TARGET)
{
   year++;
   interest = bal * RATE;
   bal = bal + interest;
}
```

Common Error 4.3

- Off-by-One Errors
  - A 'counter' variable is often used in the test condition
  - Your counter can start at 0 or 1, but programmers often start a counter at 0
  - If I want to paint all 5 fingers, when I am done?
    - **Start at 0, use <**

```
int finger = 0;
final int FINGERS = 5;
while (finger < FINGERS)
{
   // paint finger
   finger++;
}
```

0, 1, 2, 3, 4

Start at 1, use <=

```
int finger = 1;
final int FINGERS = 5;
while (finger <= FINGERS)
{
   // paint finger
   finger++;
}
```

1, 2, 3, 4, 5

4.2: Hand-Tracing

n	sum	digit
1729	0	

```
int n = 1729;
int sum = 0;
while (n > 0)
{
   int digit = n % 10;
   sum = sum + digit;
   n = n / 10;
}
System.out.println(sum);
```

- Example: Calculate the sum of digits (1+7+2+9)
  - Make columns for key variables (n, sum, digit)
  - Examine the code and number the steps
  - Set variables to state before loop begins

Tracing Sum of Digits

n	sum	digit
1729	~~0~~	
	9	9

```
int n = 1729;
int sum = 0;
while (n > 0)
{
   int digit = n % 10;
   sum = sum + digit;
   n = n / 10;
}
System.out.println(sum);
```

- Start executing loop body statements changing variable values on a new line
  - Cross out values in previous line

Tracing Sum of Digits

n	sum	digit
~~1729~~	~~0~~	
172	9	9

```
int n = 1729;
int sum = 0;
while (n > 0)
{
   int digit = n % 10;
   sum = sum + digit;
   n = n / 10;
}
System.out.println(sum);
```

- Continue executing loop statements changing variables
  - 1729 / 10 leaves 172 (no remainder)

Tracing Sum of Digits

n	sum	digit
1729	0	
172	9	9
17	11	2

```
int n = 1729;
int sum = 0;
while (n > 0)
{
   int digit = n % 10;
   sum = sum + digit;
   n = n / 10;
}
System.out.println(sum);
```

- Test condition. If true, execute loop again
  - Variable n is 172, Is 172 > 0?, True!
- Make a new line for the second time through and update variables

Tracing Sum of Digits

n	sum	digit
1729	0	
172	9	9
17	11	2
1	18	7

```
int n = 1729;
int sum = 0;
while (n > 0)
{
   int digit = n % 10;
   sum = sum + digit;
   n = n / 10;
}
System.out.println(sum);
```

- Third time through
  - Variable n is 17 which is still greater than 0
- Execute loop statements and update variables

Tracing Sum of Digits

n	sum	digit
1729	0	
172	9	9
17	11	2
1	18	7
0	19	1

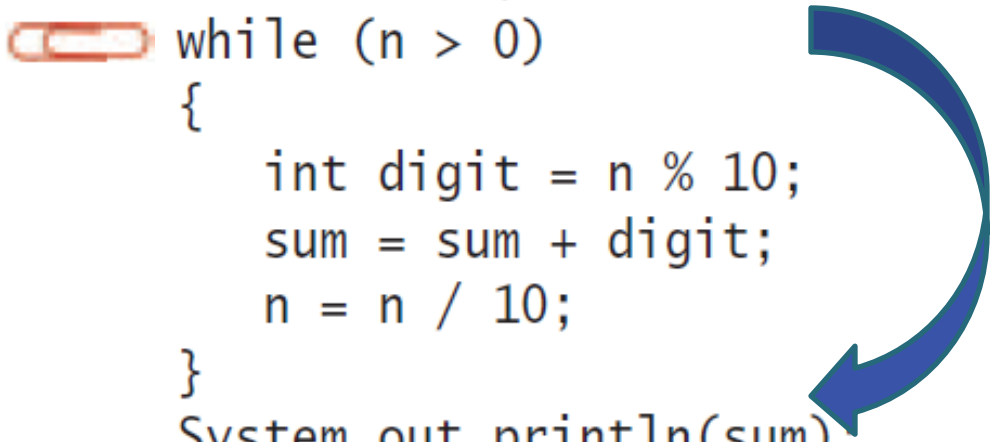
```
int n = 1729;
int sum = 0;
while (n > 0)
{
   int digit = n % 10;
   sum = sum + digit;
   n = n / 10;
}
System.out.println(sum);
```

- Fourth loop iteration:
  - Variable n is 1 at start of loop. 1 > 0? True
  - Executes loop and changes variable n to 0 (1/10 = 0)

Tracing Sum of Digits

n	sum	digit
1729	0	
172	9	9
17	11	2
1	18	7
0	19	1

```
int n = 1729;
int sum = 0;
while (n > 0)
{
   int digit = n % 10;
   sum = sum + digit;
   n = n / 10;
}
System.out.println(sum);
```



- Because n is 0, the expression `(n > 0)` is False
- Loop body is not executed
  - Jumps to next statement after the loop body
- Finally prints the sum!

Summary of the `while` Loop

- `while` loops are very commonly used
  - Initialize variables before you test
  - The condition is tested BEFORE the loop body
    - This is called *pre-test*
    - The condition often uses a counter variable
  - Something inside the loop should change one of the variables used in the test
- Watch out for infinite loops!

4.3 The for Loop

- Use a for loop when you:
  - Can use an integer counter variable
  - Have a constant increment (or decrement)
  - Have a fixed starting and ending value for the counter

```
int i = 5;  // initialize
while (i <= 10)  // test
{
   sum = sum + 1;
   i++; // update
}
```

while version

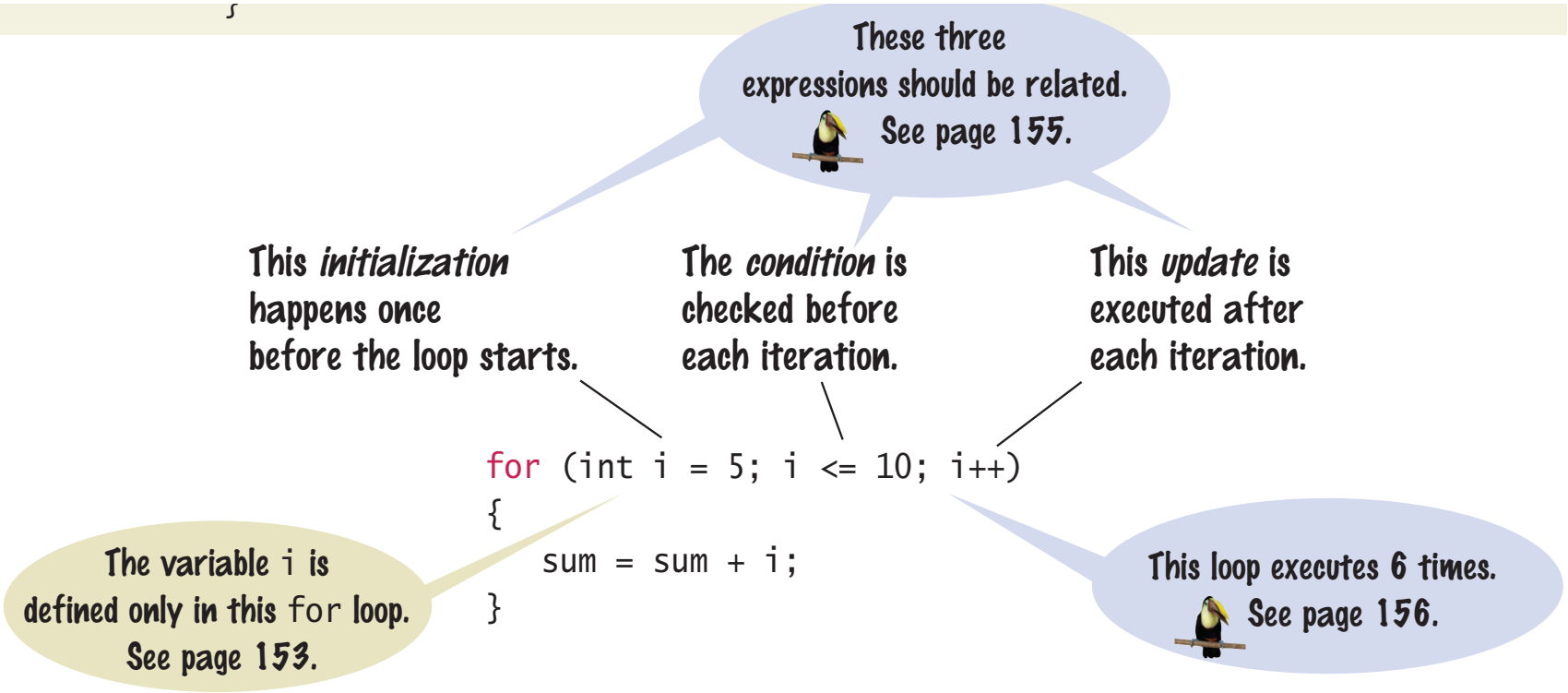
Use a for loop when a value runs from a starting point to an ending point with a constant increment or decrement.

```
for (int i = 5; i <= 10; i++)
{
   sum = sum + i;
}
```

for version

Syntax 4.2: `for` Statement

- Two semicolons separate the three parts
  - Initialization ; Condition ; Update



When to use a `for` Loop?

- Yes, a `while` loop can do everything a `for` loop can do
- `for` loop is more concise
  - Initialization
  - Condition
  - Update
  - All on one line!

In general, the `for` loop:

```
for (initialization; condition; update)
{
   statements
}
```

has exactly the same effect as the `while` loop:

```
initialization;
while (condition)
{
   statements
   update
}
```

Examples of for Loops

Table 2 for Loop Examples

Loop	Values of i	Comment
`for (i = 0; i <= 5; i++)`		

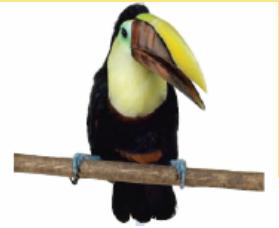
for Loop variable Scope

```
for( int x = 1; x < 10; x = x + 1) {
   // steps to do inside the loop
   // You can use 'x' anywhere in this box
}
if (x > 100)   // Error! x is out of scope!
```

- Scope is the 'lifetime' of a variable.
- When 'x' is declared in the for statement:
  - 'x' exists only inside the 'block' of the for loop { }
- Solution: Declare 'x' outside the for loop

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

Programming Tip 4.1

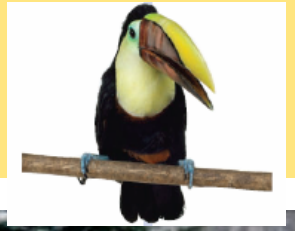


- Use `for` loops for their intended purposes only
  - Increment (or decrement) by a constant value
  - Do not update the counter inside the body
    - Update in the third section of the header

```
for (int counter = 1; counter <= 100; counter++)
{
   if (counter % 10 == 0) // Skip values divisible by 10
   {
      counter++; // Bad style: Do NOT update the counter inside loop
   }
   System.out.println(counter);
}
```

- Most counters start at one ‘end’ (0 or 1)
  - Many programmers use an integer named `i` for ‘index’ or ‘counter’ variable in `for` loops

Programming Tip 4.3



- Count Iterations
  - Many bugs are 'off by one' issues
  - One too many or one too few
- How many posts are there?
- How many pairs of rails are there?

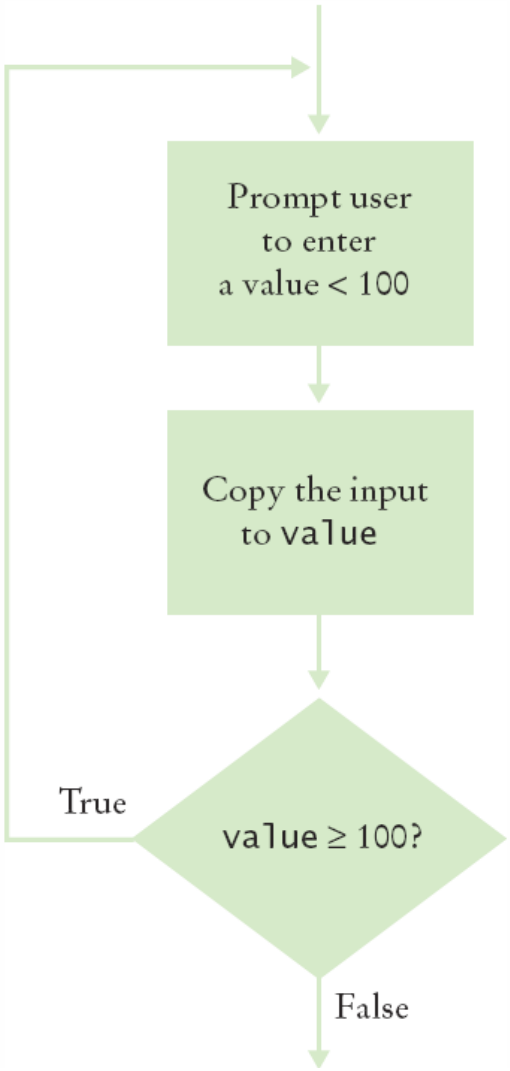
```
final int RAILS = 5;
for (int i = 1; i < RAILS; i++ )
{
   System.out.println("Painting rail " + i);
}
```

```
Painting rail 1
Painting rail 2
Painting rail 3
Painting rail 4
```

Summary of the `for` Loop

- `for` loops are very commonly used
- They have a very concise notation
  - Initialization ; Condition ; Update
  - Initialization happens once at the start
  - Condition is tested every time BEFORE executing the body (*pre-test*)
  - Increment is done at the end of the body
- Great for integer counting, String (array) processing and more

4.4 The do Loop



Use a do loop when you want to:

- Execute the body at least once
- Test the condition AFTER your first loop

```
int i = 1;  // initialize
final int FINGERS = 5;
do
{
   // paint finger
   i++; // update
}
while (i <= FINGERS);  // test
```

Note the semicolon at the end!

do Loop Example

- User Input Validation:
  - Range check a value entered
  - User must enter something to validate first!

```
int value;
do
{
   System.out.println("Enter an integer < 100: ");
   value = in.nextInt();
}
while (value >= 100);  // test
```

4.5 Processing Sentinel Values

Sentinel values are often used:

- When you don't know how many items are in a list, use a 'special' character or value to signal no more items.
- For numeric input of positive numbers, it is common to use the value -1:

A sentinel value denotes the end of a data set, but it is not part of the data.

```
salary = in.nextDouble();
while (salary != -1)
{
   sum = sum + salary;
   count++;
   salary = in.nextDouble();
}
```

Averaging a set of values

- Declare and initialize a ‘sum’ variable to 0
- Declare and initialize a ‘count’ variable to 0
- Declare and initialize an ‘input’ variable to 0
- Prompt user with instructions
- Loop until sentinel value is entered
  - Save entered value to input variable
  - If input is not -1 (sentinel value)
    - Add input to sum variable
    - Add 1 to count variable
- Make sure you have at least one entry before you divide!
  - Divide sum by count and output. Done!

SentinelDemo.java (1)

```
 8    public static void main(String[] args)
 9    {
10       double sum = 0;
11       int count = 0;
12       double salary = 0;
13       System.out.print("Enter salaries, -1 to finish: ");
14       Scanner in = new Scanner(System.in);
15
16       // Process data until the sentinel is entered
17
18       while (salary != -1)
19       {
20          salary = in.nextDouble();
21          if (salary != -1)
22          {
23             sum = sum + salary;
24             count++;
25          }
26       }
27
```

Outside the while loop, declare and initialize variables to use

Since `salary` is initialized to 0, the while loop statements will be executed

Input new `salary` and compare to sentinel

Update running `sum` and `count` to average later

SentinelDemo.java (2)

```
28          // Compute and print the average
29
30          if (count > 0)
31          {
32             double average = sum / count;
33             System.out.println("Average salary: " + average);
34          }
35          else
36          {
37             System.out.println("No data");
38          }
39       }
40    }
```

Prevent divide by 0

Calculate and output the average salary using `sum` and `count` variables

Program Run

```
Enter salaries, -1 to finish: 10 10 40 -1
Average salary: 20
```

Boolean variables and sentinels

- A `boolean` variable can be used to control a loop
  - Sometimes called a ‘flag’ variable

```
System.out.print("Enter salaries, -1 to finish: ");
boolean done = false;
while (!done)
{
   value = in.nextDouble();
   if (value == -1)
   {
      done = true;
   }
   else
   {
      // Process value
   }
}
```

Initialize done so that loop will execute

Set done ‘flag’ to true if sentinel value is found

To input any numeric value…

- When valid values can be positive or negative
  - You cannot use -1 (or any other number) as a sentinel
- One solution is to use a non-numeric sentinel
  - But Scanner's `in.nextDouble` will fail!
  - Use Scanner's `in.hasNextDouble` first
    - Returns a `boolean`: `true` (all's well) or `false` (not a number)
    - Then use `in.nextDouble` if `true`

```
System.out.print("Enter values, Q to quit: ");
while (in.hasNextDouble())
{
   value = in.nextDouble();
   // Process value
}
```

4.7 Common Loop Algorithms

1: Sum and Average Value
2: Counting Matches
3: Finding the First Match
4: Prompting until a match is found
5: Maximum and Minimum
6: Comparing Adjacent Values

Sum and Average Examples

```
double total = 0;
while (in.hasNextDouble())
{
   double input = in.nextDouble();
   total = total + input;
}
```

- Sum of Values
  - Initialize total to 0
  - Use `while` loop with sentinel

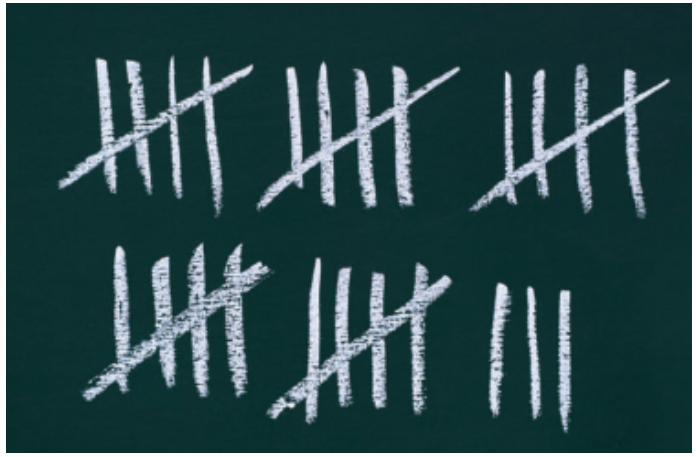
- Average of Values
  - Use Sum of Values
  - Initialize count to 0
    - Increment per input
  - Check for count 0
    - Before divide!

```
double total = 0;
int count = 0;
while (in.hasNextDouble())
{
   double input = in.nextDouble();
   total = total + input;
   count++;
}
double average = 0;
if (count > 0)
  { average = total / count; }
```

Counting Matches

- Counting Matches
  - Initialize count to 0
  - Use a `for` loop
  - Add to count per match

```
int upperCaseLetters = 0;
for (int i = 0; i < str.length(); i++)
{
   char ch = str.charAt(i);
   if (Character.isUpperCase(ch))
   {
      upperCaseLetters++;
   }
}
```



Finding the First Match

```
boolean found = false;
char ch;
int position = 0;
while (!found &&
       position < str.length())
{
   ch = str.charAt(position);
   if (Character.isLowerCase(ch))
   {
      found = true;
   }
   else { position++; }
}
```

A pre-test loop (while or for) will handle the case where the string is empty!

- Initialize `boolean` sentinel to `false`
- Initialize position counter to 0
  - First char in String
- Use a compound conditional in loop

Prompt Until a Match is Found

```
boolean valid = false;
double input;
while (!valid)
{
   System.out.print("Please enter a positive value < 100: ");
   input = in.nextDouble();
   if (0 < input && input < 100) { valid = true; }
   else { System.out.println("Invalid input."); }
}
```

- Initialize `boolean` flag to `false`
- Test sentinel in `while` loop
  - Get input, and compare to range
    - If input is in range, change flag to true
    - Loop will stop executing

Maximum and Minimum

```
double largest = in.nextDouble();
while (in.hasNextDouble())
{
   double input = in.next
   if (input > largest)
   {
      largest = input;
   }
}
```

```
double smallest = in.nextDouble();
while (in.hasNextDouble())
{
   double input = in.nextDouble();
   if (input > smallest)
   {
      smallest = input;
   }
}
```

- Get first input value
  - This is the largest (or smallest) that you have seen so far!
- Loop `while` you have a valid number (non-sentinel)
  - Get another input value
  - Compare new input to largest (or smallest)
  - Update largest (or smallest) if necessary

Comparing Adjacent Values

```
double input = in.nextDouble();
while (in.hasNextDouble())
{
   double previous = input;
   input = nextDouble();
   if (input == previous)
   {
      System.out.println("Duplicate input");
   }
}
```

- Get first input value
- Use `while` to determine if there are more to check
  - Copy input to previous variable
  - Get next value into input variable
  - Compare input to previous, and output if same

Steps to Writing a Loop

Planning:

1. Decide what work to do inside the loop
2. Specify the loop condition
3. Determine loop type
4. Setup variables before the first loop
5. Process results when the loop is finished
6. Trace the loop with typical examples

Coding:

7. Implement the loop in Java

4.8 Nested Loops

- How would you print a table with rows and columns?
  - Print top line (header)
    - Use a `for` loop
  - Print table body…
    - How many rows?
    - How many columns?
  - Loop per row
    - Loop per column

x^1	x^2	x^3	x^4
1	1	1	1
2	4	8	16
3	9	27	81
…	…	…	…
10	100	1000	10000

PowerTable.java

```
1  /**
2     This program prints a table of powers of x.
3  */
4  public class PowerTable
5  {
6     public static void main(String[] args)
7     {
8        final int NMAX = 4;
9        final double XMAX = 10;
10
24       // Print table body
25
26       for (double x = 1; x <= XMAX; x++)
27       {
28          // Print table row
29
30          for (int n = 1; n <= NMAX; n++)
31          {
32             System.out.printf("%10.0f", Math.pow(x, n));
33          }
34          System.out.println();
35       }
36    }
37 }
```

```
         1         2         3         4
         x         x         x         x

         1         1         1         1
         2         4         8        16
         3         9        27        81
         4        16        64       256
         5        25       125       625
         6        36       216      1296
         7        49       343      2401
         8        64       512      4096
         9        81       729      6561
        10       100      1000     10000
```

Body of outer loop

Body of inner loop

Nested Loop Examples (1)

```
for (i = 1; i <= 3; i++)
{
   for (j = 1; j <= 4; j++)  { print "*" }
   print new line
}
```

```
for (i = 1; i <= 4; i++)
{
   for (j = 1; j <= 3; j++) { print "*" }
   print new line
}
```

```
for (i = 1; i <= 4; i++)
{
   for (j = 1; j <= i; j++) { print "*" }
   print new line
}
```

Nested Loop Examples (2)

```
for (i = 1; i <= 3; i++)
{
   for (j = 1; j <= 5; j++)
   {
      if (j % 2 == 0) { Print "*" }
      else { Print "-" }
   }
   System.out.println();
}
```

```
for (i = 1; i <= 3; i++)
{
   for (j = 1; j <= 5; j++)
   {
      if (i % 2 == j % 2) { Print "*" }
      else { Print " " }
   }
   System.out.println();
}
```

RandomDemo.java

```
1  /**
2     This program prints ten random numbers between 0 and 1.
3  */
4  public class RandomDemo
5  {
6     public static void main(String[] args)
7     {
8        for (int i = 1; i <= 10; i++)
9        {
10          double r = Math.random();
11          System.out.println(r);
12       }
13    }
14 }
```

Program Run

```
0.2992436267816825
0.4386017604531353 7
0.7365753471168408
0.6880250194282326
0.1608272403783395
0.5362876579988844
0.3098705906424375
0.6602909916554179
0.1927951611482942
0.8632330736331089
```

4.9 Random Numbers/Simulations

- Games often use random numbers to make things interesting
  - Rolling Dice
  - Spinning a wheel
  - Pick a card
- A simulation usually involves looping through a sequence of events
  - Days
  - Events

Simulating Die Tosses

- Goal:
  - Get a random integer between 1 and 6

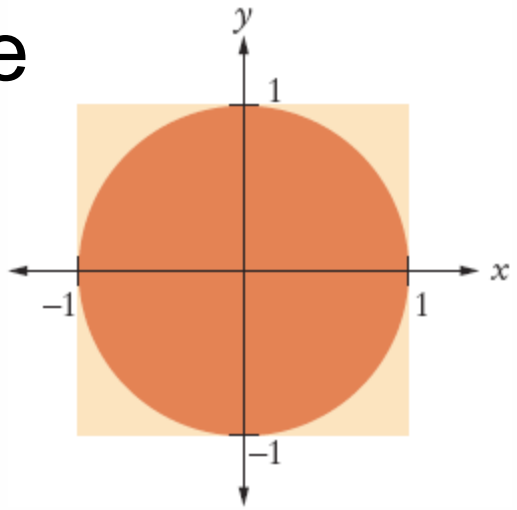
```
1  /**
2     This program simulates tosses of a pair of dice.
3  */
4  public class Dice
5  {
6     public static void main(String[] args)
7     {
8        for (int i = 1; i <= 10; i++)
9        {
10          // Generate two random numbers between 1 and 6
11
12          int d1 = (int) (Math.random() * 6) + 1;
13          int d2 = (int) (Math.random() * 6) + 1;
14          System.out.println(d1 + " " + d2);
15       }
16       System.out.println();
17    }
18 }
```

Program Run

```
5 1
2 1
1 2
5 1
1 2
6 4
4 4
6 1
6 3
5 2
```

The Monte Carlo Method

- Used to find approximate solutions to problems that cannot be precisely solved
- Example: Approximate PI using the relative areas of a circle inside a square
  - Uses simple arithmetic
  - Hits are inside circle
  - Tries are total number of tries
  - Ratio is 4 x Hits / Tries



MonteCarlo.java

```
1  /**
2     This program computes an estimate of pi by simulating dart throws onto a square.
3  */
4  public class MonteCarlo
5  {
6     public static void main(String[] args)
7     {
8        final int TRIES = 10000;
9
10       int hits = 0;
11       for (int i = 1; i <= TRIES; i++)
12       {
13          // Generate two random numbers between -1 and 1
14
15          double r =
16          double x =
17          r = Math.
18          double y =
19
20          // Check
21
22          if (x * x
23       }
24
25          /*
26             The ratio hits / tries is approximately the same as the ratio
27             circle area / square area = pi / 4
28          */
29
30          double piEstimate = 4.0 * hits / TRIES;
31          System.out.println("Estimate for pi: " + piEstimate);
32       }
33 }
```

Program Run

```
Estimate for pi: 3.1504
```

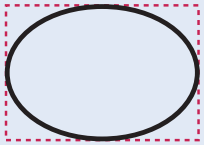
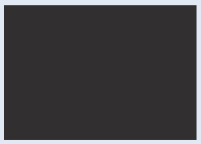
The 'empty' body

- You probably have developed the habit of typing a semicolon at the end of each line
- Don't do this with loop statements!
  - The loop body becomes very short!
    - Between the closing) and ;
    - What type of loop do I have now?

```
while (bal < TARGET);
{
   year++;
   interest = bal * RATE;
   bal = bal + interest;
}
```

Drawing Graphical Shapes

Table 4 Graphics Methods

Method	Result	Notes
`g.drawRect(x, y, width, height)`		`(x, y)` is the top left corner.
`g.drawOval(x, y, width, height)`		`(x, y)` is the top left corner of the box that bounds the ellipse. To draw a circle, use the same value for width and height.
`g.fillRect(x, y, width, height)`		The rectangle is filled in.
`g.fillOval(x, y, width, height)`		The oval is filled in.

Drawing Graphical Shapes

`g.drawLine(x1, y1, x2, y2)`		`(x1, y1)` and `(x2, y2)` are the endpoints.
`g.drawString("Message", x, y)`		`(x, y)` is the basepoint.
`g.setColor(color)`	From now on, draw or fill methods will use this color.	Use `Color.RED`, `Color.GREEN`, `Color.BLUE`, and so on. (See Table 10.1 for a complete list of predefined colors.)

TwoRowsOfSquares.java

```
11    public static void draw(Graphics g)
12    {
13       final int width = 20;
14       g.setColor(Color.BLUE);
15
16       // Top row. Note that the top left corner of the drawing has coordinates (0, 0)
17       int x = 0;
18       int y = 0;
19       for (int i = 0; i < 10; i++)
20       {
21          g.fillRect(x, y, width, width);
22          x = x + 2 * width;
23       }
24       // Second row, offset from the first one
25       x = width;
26       y = width;
27       for (int i = 0; i < 10; i++)
28       {
29          g.fillRect(x, y, width, width);
30          x = x + 2 * width;
31       }
32    }
```

Summary: `if` Statement

- The if statement allows a program to carry out different actions depending on the nature of the data to be processed.
- Relational operators (`< <= > >= == !=`) are used to compare numbers and `Strings`.
- Do not use the `==` operator to compare `Strings`.
  - Use the `equals` method instead.
  - The `compareTo` method compares `Strings` in lexicographic order.
- Multiple `if` statements can be combined to evaluate complex decisions.
- When using multiple `if` statements, test general conditions after more specific conditions.

Summary: Boolean

- The Boolean type `boolean` has two values, true and false.
  - Java has two Boolean operators that combine conditions: `&&` (*and*) and `||` (*or*).
  - To invert a condition, use the `!` (*not*) operator.
  - The `&&` and `||` operators are computed lazily: As soon as the truth value is determined, no further conditions are evaluated.
  - De Morgan's law tells you how to negate `&&` and `||` conditions.
- You can use `Scanner` `hasNext` methods to ensure that the data is what you expect.

Summary: Types of Loops

- There are three types of loops:
  - `while` Loops
  - `for` Loops
  - `do` Loops
- Each loop requires the following steps:
  - Initialization (setup variables to start looping)
  - Condition (test if we should execute loop body)
  - Update (change something each time through)

Summary

- A loop executes instructions repeatedly while a condition is true.
- An off-by-one error is a common error when programming loops.
  - Think through simple test cases to avoid this type of error.
- The `for` loop is used when a value runs from a starting point to an ending point with a constant increment or decrement.
- The `do` loop is appropriate when the loop body must be executed at least once.