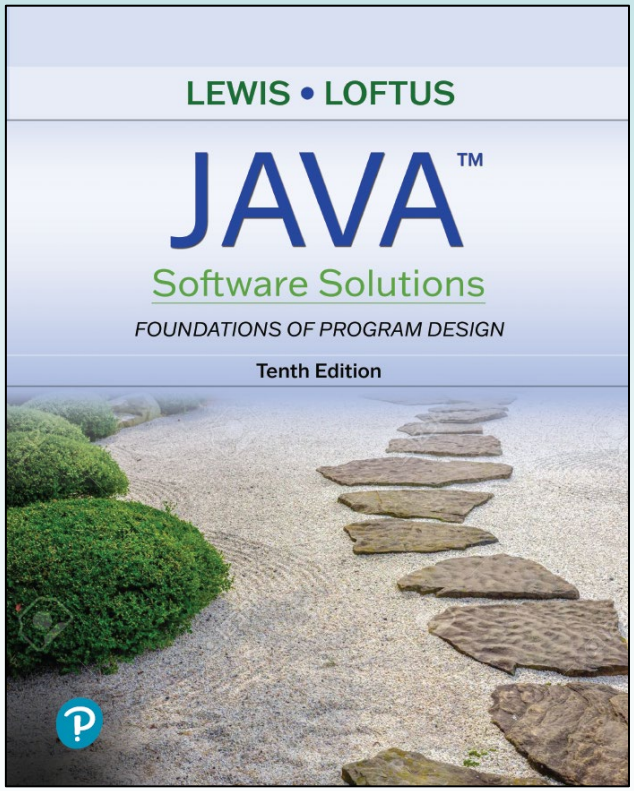
Chapter 5 **B**
Loops



Java Software Solutions
Foundations of Program Design
10th Edition

Edits: 03/17/2023

2/25/2026

John Lewis
William Loftus

§ 5.3 Comparing Data

Some situations:

- Comparing floating point values for equality
- Comparing characters
- Comparing strings (alphabetical order)
- Comparing object vs. comparing object references

Comparing Float Values

- You should rarely use the equality operator (==) when comparing two floating point values (`float` or `double`)
- Two floating point values are equal only if their underlying binary representations match exactly
  - for double, all 64 bits must be the same
- Computations often result in slight differences from what is arithmetically correct
- Often, two floating point numbers are "close enough" even if they aren't exactly equal

```
public class FloatEqual
{
    public static void main( String[] args )
    {
        double sum = 0.0;
        sum = 0.1 + 0.1 + 0.1 ;
        if ( sum == 3/10.0 )
          System.out.println("Equal");
        else
          System.out.println("NOT equal");
    }
}
```

Prints: NOT equal

Comparing Float Values

- Use this technique:

```
final double TOLERANCE = 0.000001;
 . . . .
if ( Math.abs(f1 - f2) < TOLERANCE )
    System.out.println("Essentially equal");
```

- If the difference between the two floating point values is less than the tolerance, they are considered to be equal

- The tolerance could be set to any appropriate level, such as 0.000001

```
public class FloatEqual
{
    public static void main( String[] args )
    {
        double sum = 0.0;
        sum = 0.1 + 0.1 + 0.1 ;
        if ( Math.abs( sum-3/10.0 ) < 0.00001 )
            System.out.println("Equal");
        else
            System.out.println("NOT equal");
    }
}
```

Prints: Equal

Comparing Characters

- Java character data is based on the Unicode character set
- Unicode establishes a particular numeric value for each character and therefore an ordering
- Use relational operators with character data based on this ordering
- The character `'8'` is less than the character `'J'` because it comes before it in the Unicode character set
  - Not obvious, and not a thing to memorize
  - Look it up if you need it

Comparing Characters

- In Unicode, the digit characters (0-9) are contiguous and in order
- Likewise, the uppercase letters (A-Z) and lowercase letters (a-z) are contiguous and in order
- BUT: the uppercase letters are separate from the lowercase letters

Characters	Unicode Values
0 – 9	48 through 57
A – Z	65 through 90
a – z	97 through 122

Comparing Strings

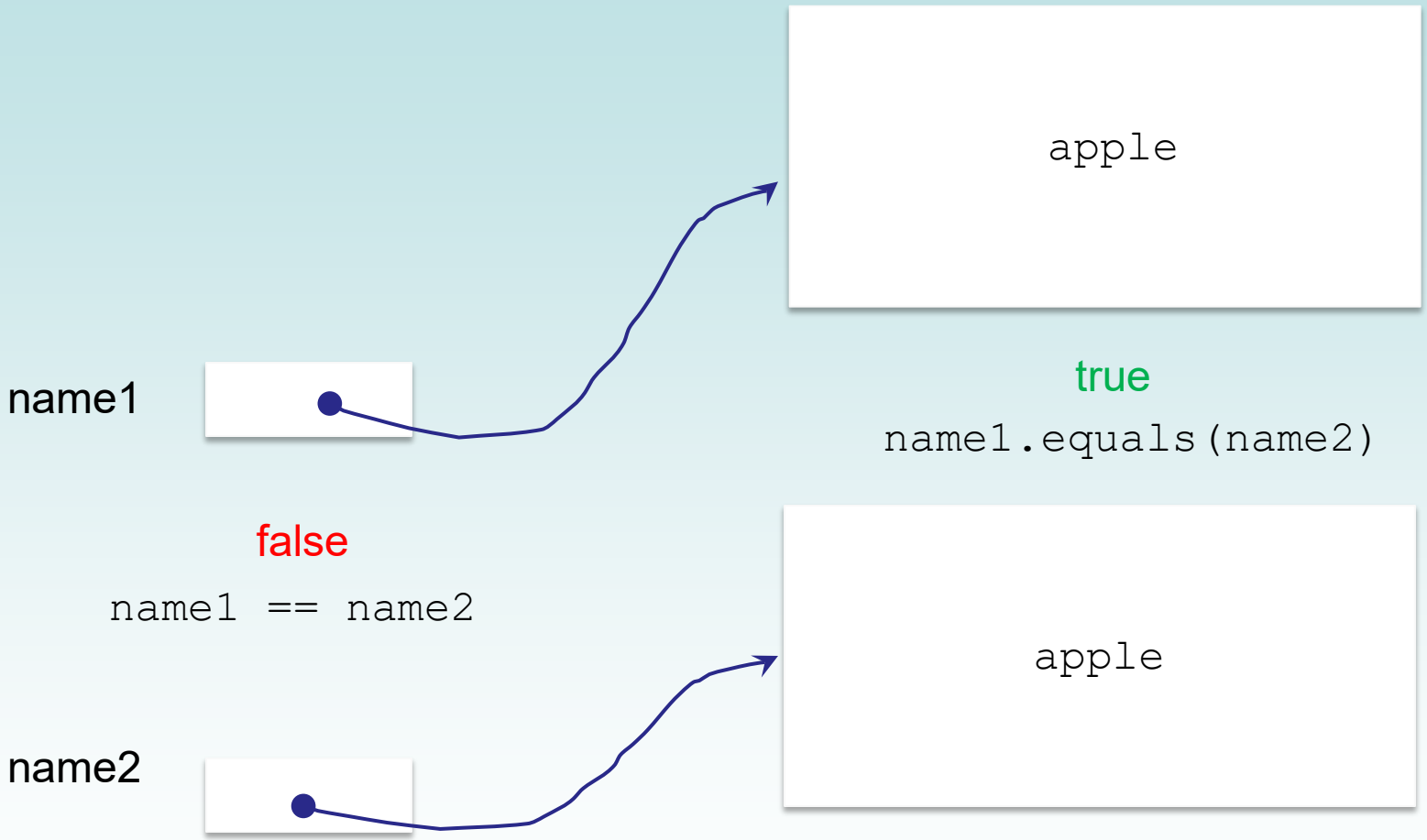
- In Java a character string is an object
- The **`equals`** method determines if two String objects contain exactly the same characters in the same order
- The `equals` method returns a boolean result

```
if ( name1.equals(name2) )
   System.out.println("Same name");
```

```
class EQapple
{
  public static void main( String[] args )
  {
    String name1;  // reference variable
    String name2;  // reference variable
    name1 = new String( "apple" );  // create an object
    name2 = new String( "apple" );  // create a different object

    System.out.println( name1 );  // follow the reference to first object
    System.out.println( name2 );  // follow the reference to second object

    // different String references are not ==
    if (name1 == name2 )
      System.out.println( "This will NOT print." );

    // different Strings with same data are equal
    if (name1.equals(name2) )
      System.out.println( "This will print." );

  }
}
```

Sample Run

```
apple
apple
This will NOT print
This will print
```



Comparing Strings

- We cannot use the relational operators (like <, ==) to compare strings
- The `String` class contains the **`compareTo`** method for determining if one string comes before another
- **`name1.compareTo(name2)`**
  - zero if `name1` and `name2` are equal (contain the same characters)
  - negative value if `name1` is less than `name2`
  - positive value if `name1` is greater than `name2`
- Think of `compareTo` as minus

Comparing Objects

- The == operator can be used with object references.
  - it returns true if the two reference variables point to the same object (are aliases)

  - The == operator does NOT look at objects, only the contents of reference variables.
  - It compares two pointers, not what they point to.

Comparing Strings

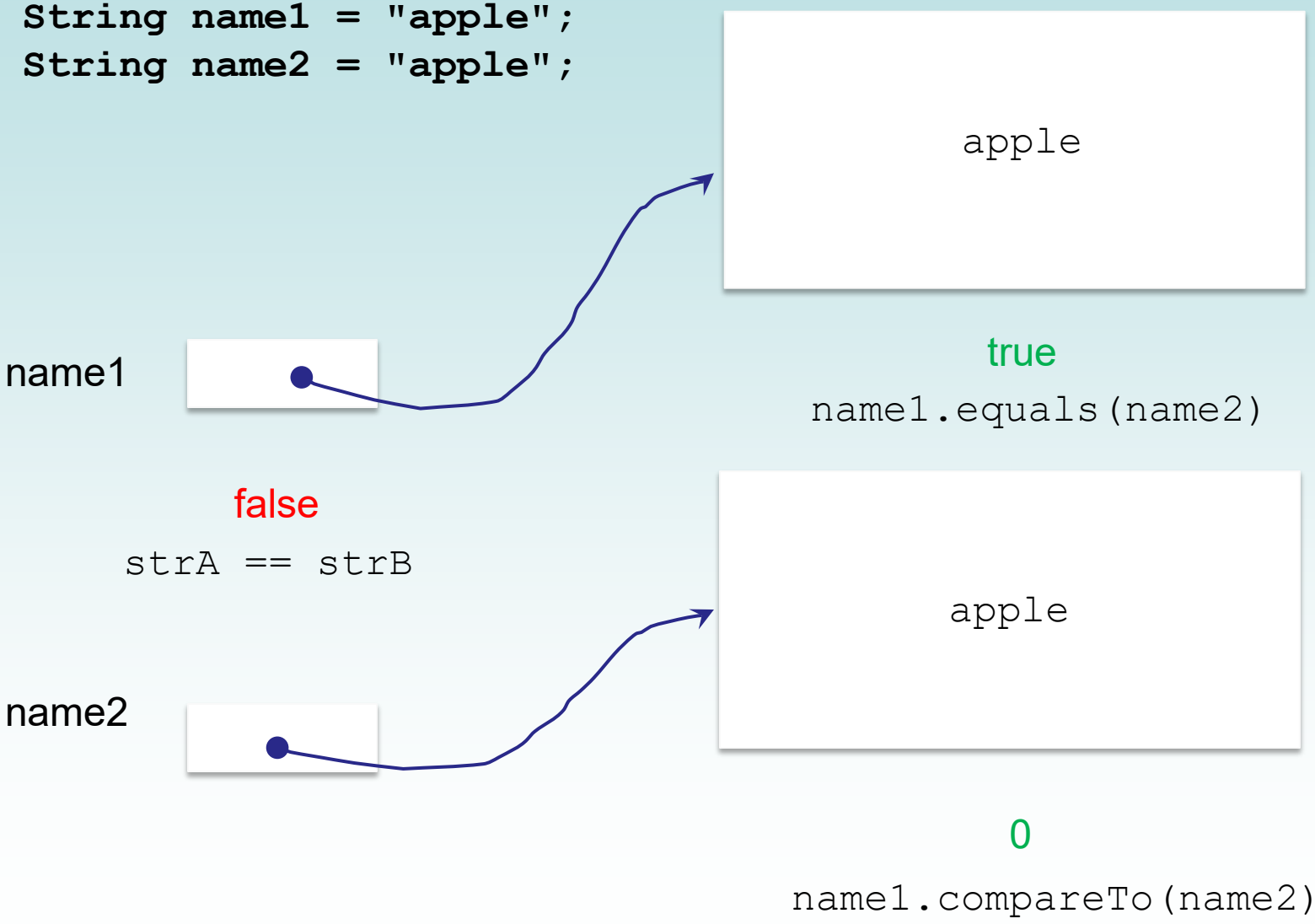
Called *lexicographic ordering*

```
String name1 = "apple";
String name2 = "apple";
                              0
int result = name1.compareTo(name2);
             F
if (result < 0)
   System.out.println(name1 + "comes first");
else                T
   if (result == 0)
      System.out.println("Same name");
   else
      System.out.println(name2 + "comes first");
```

Sample Run

```
Same name
```

```
String name1 = "apple";
String name2 = "apple";
```

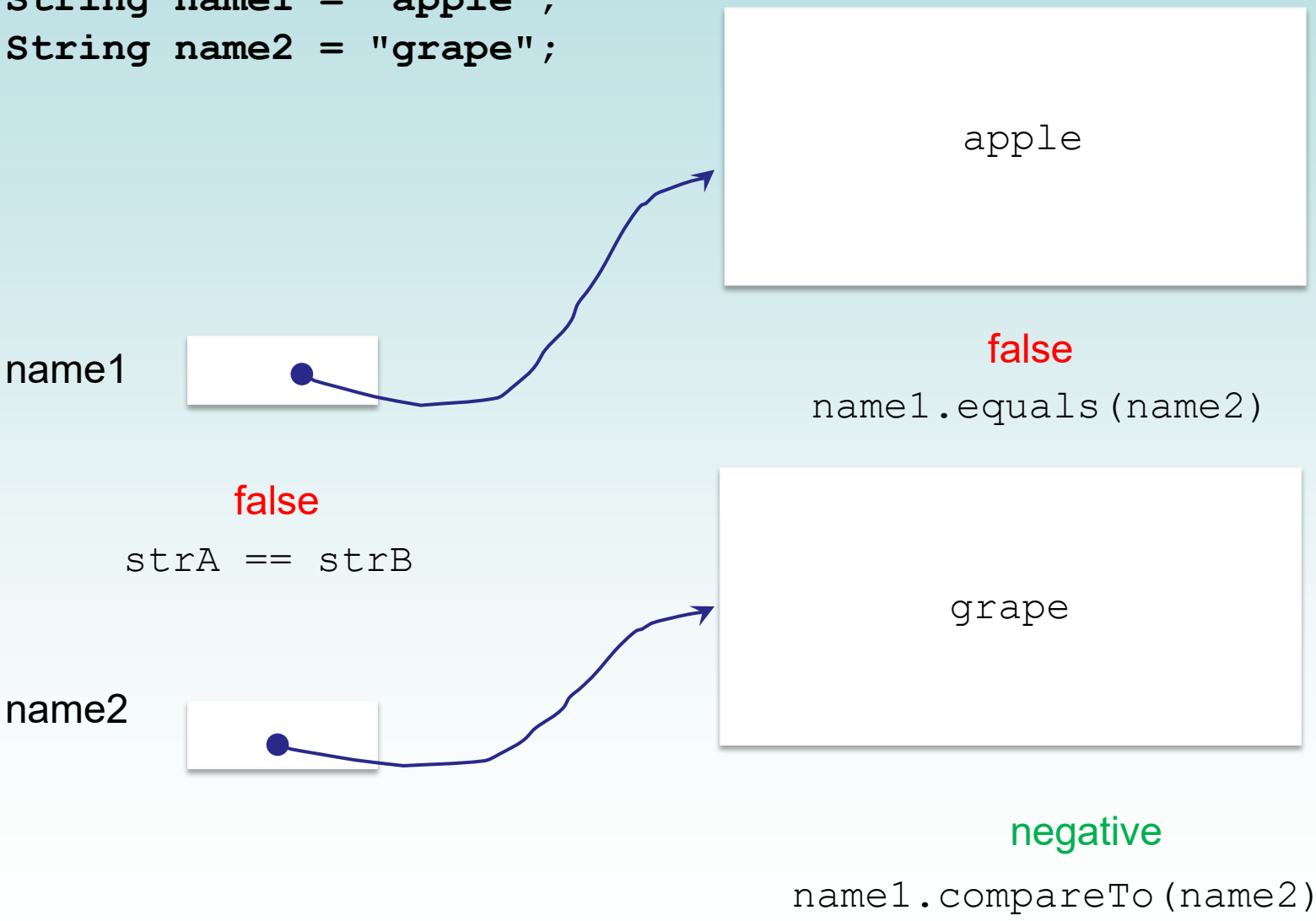


```
String name1 = "apple";
String name2 = "grape";

int result = name1.compareTo(name2);
             T
if (result < 0)
   System.out.println(name1 + "comes first");
else
   if (result == 0)
      System.out.println("Same name");
   else
      System.out.println(name2 + "comes first");
```

Sample Run

```
apple comes first
```

```
String name1 = "apple";
String name2 = "grape";
```



Lexicographic Ordering

- Lexicographic ordering is not strictly alphabetical when uppercase and lowercase characters are mixed
- For example, the string `"Great"` comes before the string `"fantastic"`
  - uppercase letters come before all of the lowercase letters in Unicode
- Also, short strings come before longer strings with the same prefix
- Therefore `"book"` comes before `"bookcase"`

Comparing Objects

- The `==` operator can be used with object references.
  - it returns true if the two references are aliases of the same object
  - The `==` operator does NOT look at objects, only at the contents of reference variables.

- The **`equals`** method is defined for all objects, but unless we redefine it when we write a class, it has the same meaning as the `==` operator
- The **`equals`** method has been redefined in the **`String`** class to compare the characters in the two strings
- When you write a class, you can redefine the **`equals`** method to return true under whatever conditions are appropriate

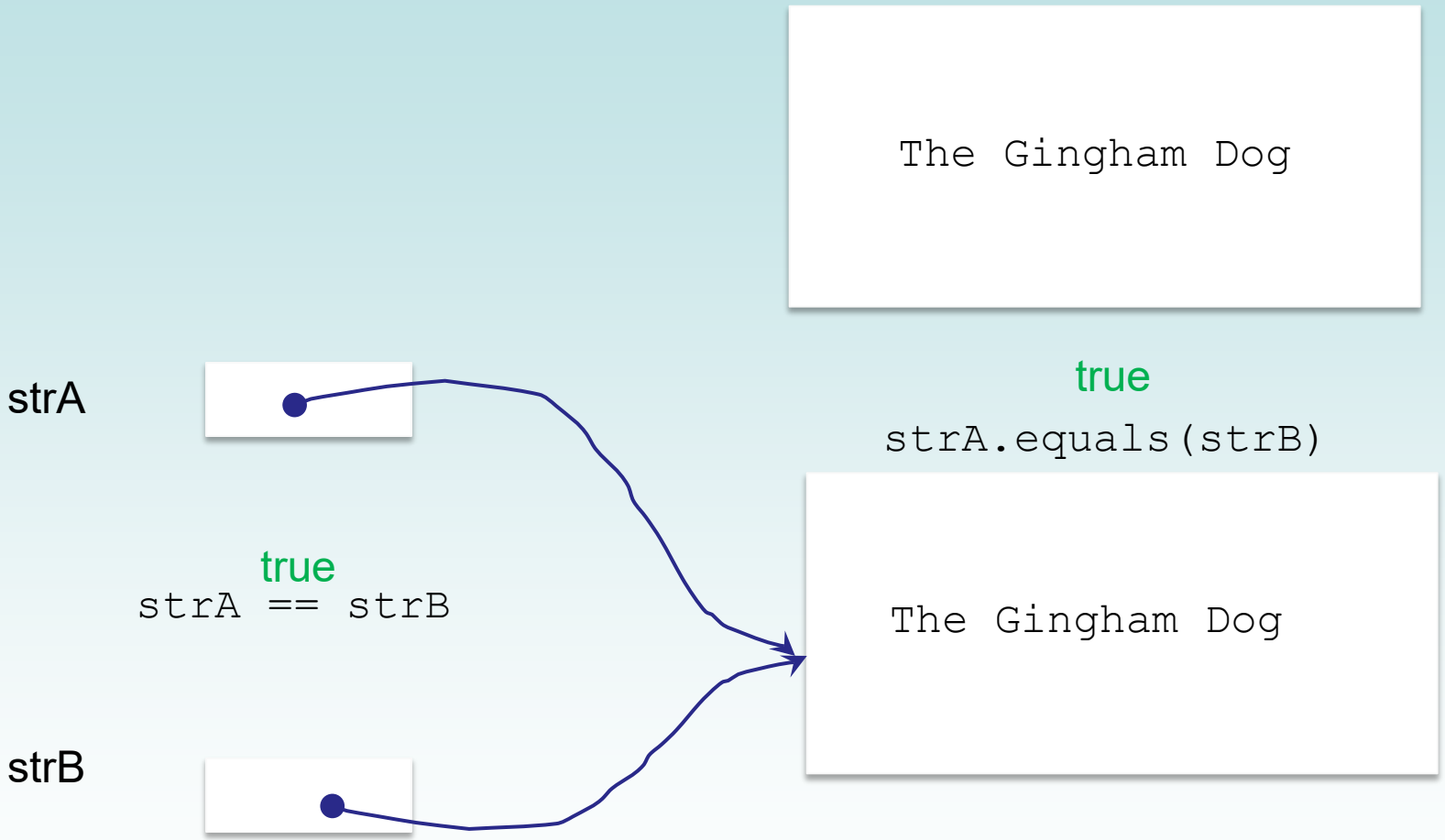
```
class EQdemo
{
  public static void main( String[] args )
  {
    String strA;  // reference variable
    String strB;  // reference variable
    strA = new String( "The Gingham Dog" );  // create an object
    strB = new String( "The Gingham Dog" );  // create a different object

    System.out.println( strA );  // follow the reference to first object
    System.out.println( strB );  // follow the reference to second object

    // different String references are not ==
    if ( strA == strB )
      System.out.println( "This will NOT print." );

    // different Strings with same data are equals
    if (strA.equals(strB) )
      System.out.println( "This will print." );

    // create aliases
    strA = strB;
    if ( strA == strB )
      System.out.println( "NOW this will print." );

  }
}
```

Sample Run

```
The Gingham Dog
The Gingham Dog
This will print
NOW this will print
```

```
strA = new String( "The Gingham Dog" );
strB = new String( "The Gingham Dog" );
```

After `strA = strB;` has executed



Outline

- Boolean Expressions
- The `if` Statement
- Comparing Data
- The `while` Statement
  - Iterators
- The `ArrayList` Class
  - Determining Event Sources
  - Managing Fonts
  - Check Boxes and Radio Buttons

§ 5.4 Repetition Statements

- *Repetition statements* execute a statement multiple times
- Often called *loops*
- They are controlled by boolean expressions
- Three kinds of repetition statements:
  - **`while`**, **`do`**, and **`for`** loops
- The `do` and `for` loops are discussed in Chapter 6

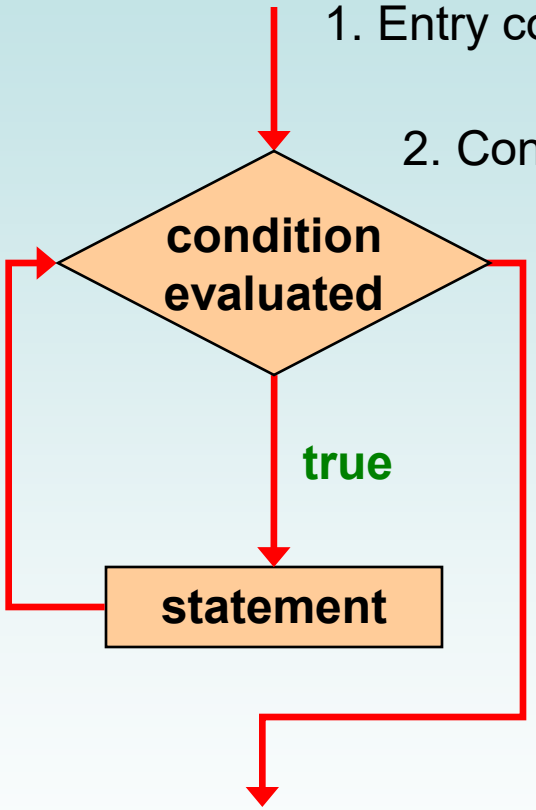
The while Statement

- Syntax of a *while statement* :

```
while ( condition )
    statement ;
```

- If the **condition** is true, the **statement** is executed
- Then the condition is evaluated again, and if it is still true, the statement is executed again
- The statement is executed repeatedly until the condition becomes false

Three Aspects of a Loop



1. Entry condition must be correct

2. Condition testing must be correct

3. Must correctly change what the condition tests

The three aspects must be coordinated with one another.
If not, the loop will execute too few or too many times.

The while Statement

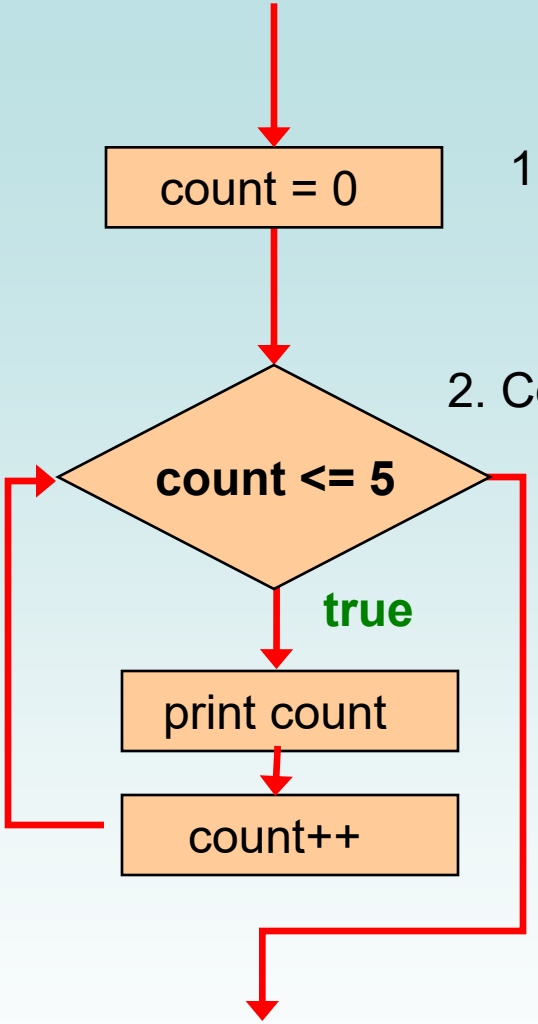
- Example:

```
int count = 0;
while (count <= 5)
{
    System.out.println(count);
    count++ ;
}
```

Sample Run

```
0
1
2
3
4
5
```

- If the condition of a `while` loop is false initially, the statement is never executed
- The body of a `while` loop may execute zero or more times



1. Entry condition

2. Condition test

3. correctly change count

Complete Program

```
public class Loop
{
  public static void main ( String[] args)
  {
    int count = 0;
    while (count <= 5)
    {
      System.out.println(count);
      count++;
    }
  }
}
```

1. Entry condition
2. Condition test
3. correctly change

```
public class Loop
{
  public static void main ( String[] args)
  {
    int count = 0;
    while (count <= 5)
    {
      System.out.println(count);
      count++;
    }
  }
}
```

Perhaps try with BlueJ

Complete Program, with a twist

```
public class Loop
{
  public static void main ( String[] args)
  {
    int count = 0;
    while (count <= 5)
    {
      System.out.println(count);
      count++;
    }
    System.out.println(count);
  }
}
```

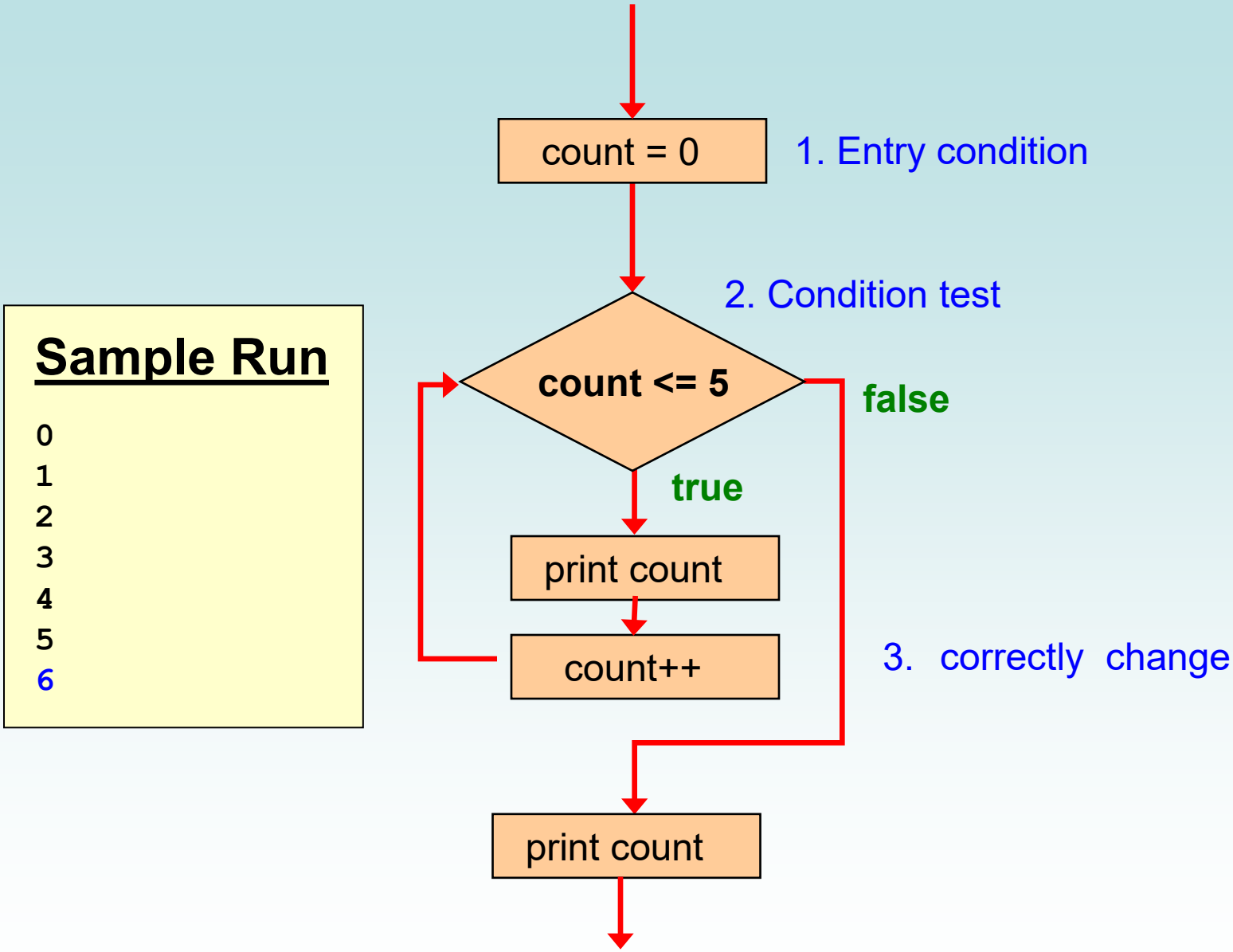
1. Entry condition
2. Condition test
3. correctly change

Sample Run

```
0
1
2
3
4
5
6
```

count is an ordinary int variable.

If some statement changes it (even though inside a loop), it stays changed.



Sample Run

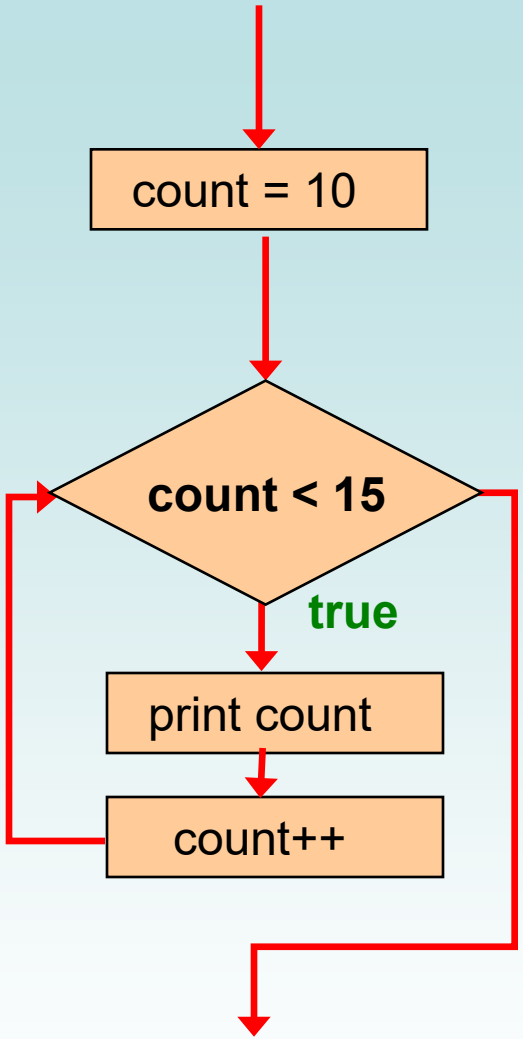
```
0
1
2
3
4
5
6
```

Another while Example

```
int count = 10;
while (count < 15)
{
    System.out.println(count);
    count++;
}
```

1. Entry condition
2. Condition test
3. correctly change

```
10
11
12
13
14
```



1. Entry condition

2. Condition test

10
11
12
13
14

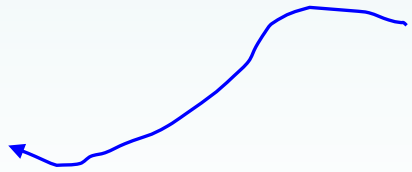
3. correctly change

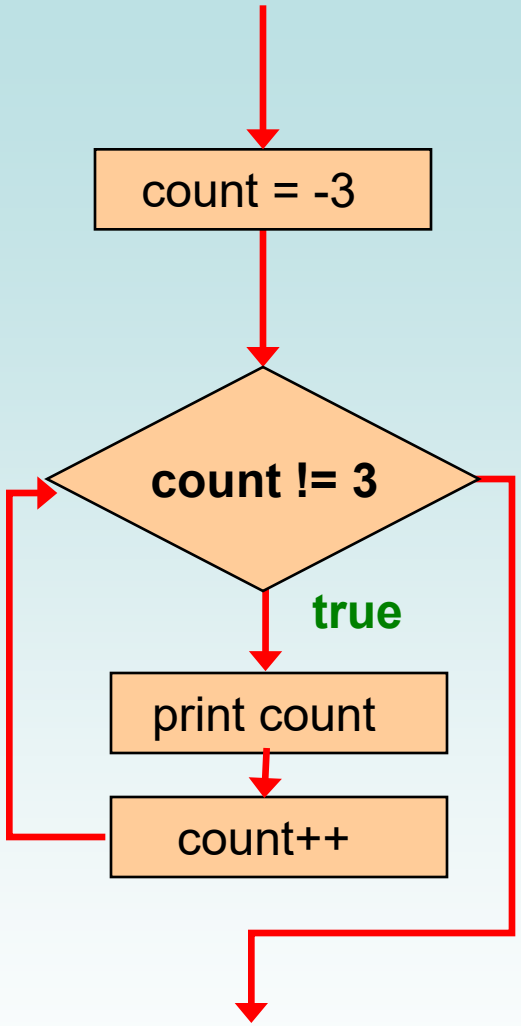
Another while Example

```
int count = -3;
while (count != 3)
{
    System.out.println(count);
    count++;
}
```

1. Entry condition
2. Condition test
3. correctly change

```
-3
-2
-1
0
1
2
```





1. Entry condition

2. Condition test

-3
-2
-1
0
1
2

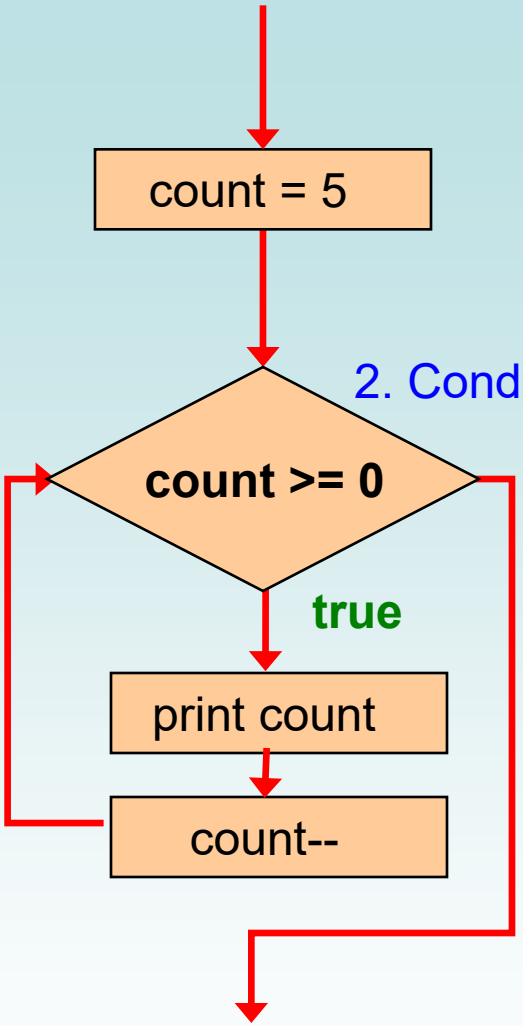
3. correctly change

One more while Example

```
int count = 5;
while (count >= 0)
{
    System.out.println(count);
    count--;
}
```

1. Entry condition
2. Condition test
3. correctly change

```
5
4
3
2
1
0
```



5
4
3
2
1
0

Best Strategy

- Usually the condition tested in the while should remain true for the body of the loop
- Prepare for the next iteration at the bottom of the body
- Other arrangements, although “logically equivalent” can be confusing

What does this do?

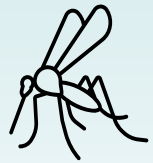
```
int count = 0;
while (count < 5)
{
   count++;
   System.out.println(count);
}
```

1. Entry condition
2. Condition test
3. change

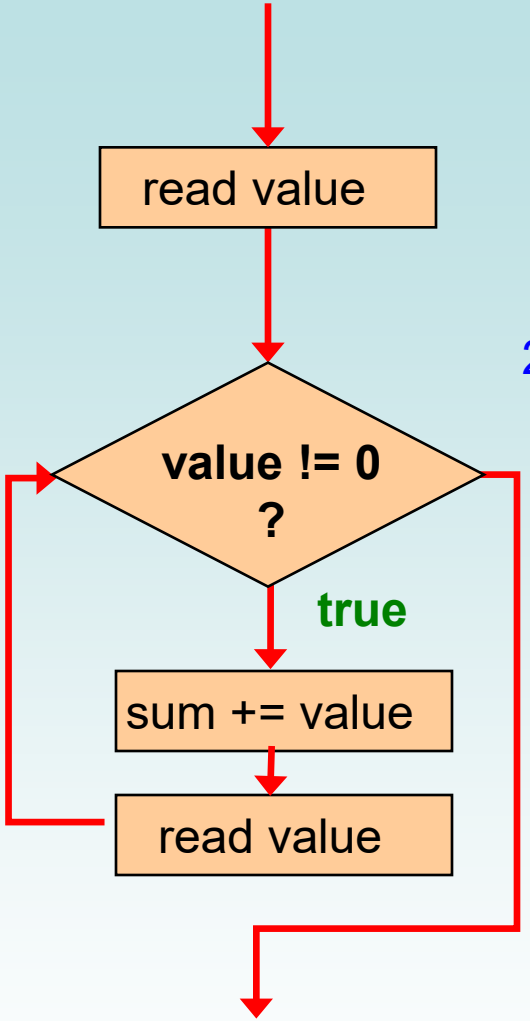
```
1
2
3
4
5
```

Sentinel Values

- A loop can be used to maintain a *running sum*
- A *sentinel value* is an input value you have picked to represent the end of input
  - What value you pick depends on the data
  - Don't pick a value that could be in the data
  - Don't pick a value that might occur in defective data



- See `Average.java`



1. Entry condition

2. Condition test:
   0 is the sentinel value

3. Change condition by inputting another value

```
//********************************************************************
//  Average.java       Author: Lewis/Loftus
//
//  Demonstrates the use of a while loop, a sentinel value of 0, and a
//  running sum.
//********************************************************************

import java.text.DecimalFormat;
import java.util.Scanner;

public class Average
{
   //-----------------------------------------------------------------
   //  Computes the average of a set of values entered by the user.
   //  The running sum is printed as the numbers are entered.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      int sum = 0, value, count = 0;
      double average;

      Scanner scan = new Scanner(System.in);
```

```
System.out.print("Enter an integer (0 to quit): ");
value = scan.nextInt();

while (value != 0)  // sentinel value of 0 to terminate loop
{

   count++;
   sum += value;
   System.out.println("The sum so far is " + sum);

   System.out.print("Enter an integer (0 to quit): ");
   value = scan.nextInt();
}
```

1. Entry condition

2. Condition test

3. change

```
      System.out.println();

      if (count == 0)
         System.out.println("No values were entered.");
      else
      {
         average = (double)sum / count;  // notice the type cast

         DecimalFormat fmt = new DecimalFormat("0.###");
         System.out.println("The average is " + fmt.format(average));
      }
   }
}
```

Sample Run

```
Enter an integer (0 to quit): 25
The sum so far is 25
Enter an integer (0 to quit): 164
The sum so far is 189
Enter an integer (0 to quit): -14
The sum so far is 175
Enter an integer (0 to quit): 84
The sum so far is 259
Enter an integer (0 to quit): 12
The sum so far is 271
Enter an integer (0 to quit): -35
The sum so far is 236
Enter an integer (0 to quit): 0

The average is 39.333
```

Sample Run

```
Enter an integer (0 to quit): 0
No values were entered.
```

Another BlueJ Opportunity!

```
import java.text.DecimalFormat;
import java.util.Scanner;

public class Average
{
   public static void main(String[] args)
   {
      int sum = 0, value, count = 0;
      double average;
      Scanner scan = new Scanner(System.in);

      System.out.print("Enter an integer (0 to quit): ");
      value = scan.nextInt();

      while (value != 0)  // sentinel value of 0 to terminate loop
      {
         count++;

         sum += value;
         System.out.println("The sum so far is " + sum);

         System.out.print("Enter an integer (0 to quit): ");
         value = scan.nextInt();
      }

      System.out.println();
      if (count == 0)
         System.out.println("No values were entered.");
      else
      {
         average = (double)sum / count;  // notice the type cast

         DecimalFormat fmt = new DecimalFormat("0.###");
         System.out.println("The average is " + fmt.format(average));
      }
   }
}
```

Input Validation

- A loop can also be used for *input validation*
- It's a good idea to verify that input is valid
- See `WinPercentage.java`

```
//********************************************************************
//  WinPercentage.java       Author: Lewis/Loftus
//
//  Demonstrates the use of a while loop for input validation.
//********************************************************************

import java.text.NumberFormat;
import java.util.Scanner;

public class WinPercentage
{
   //-----------------------------------------------------------------
   //  Computes the percentage of games won by a team.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      final int NUM_GAMES = 12;
      int won;
      double ratio;

      Scanner scan = new Scanner(System.in);

      System.out.print("Enter the number of games won (0 to "
                       + NUM_GAMES + "): ");
```

Sample Run

```
Enter the number of games won (0 to 12): -5
Invalid input. Please reenter: 13
Invalid input. Please reenter: 7

Winning percentage: 0.58
```

```
      won = scan.nextInt();

      while (won < 0 || won > NUM_GAMES)
      {
         System.out.print ("Invalid input. Please reenter: ");

         won = scan.nextInt();
      }

      ratio = (double)won / NUM_GAMES;

      System.out.println();
      System.out.println("Winning percentage: " +  ratio  );
   }
}
```

1. Initialize
2. Test
3. Change

Infinite Loops 

- The body of a `while` loop eventually must make the condition false
- If not, it is called an *infinite loop*, which will execute until the user interrupts the program
- This is a common logical error

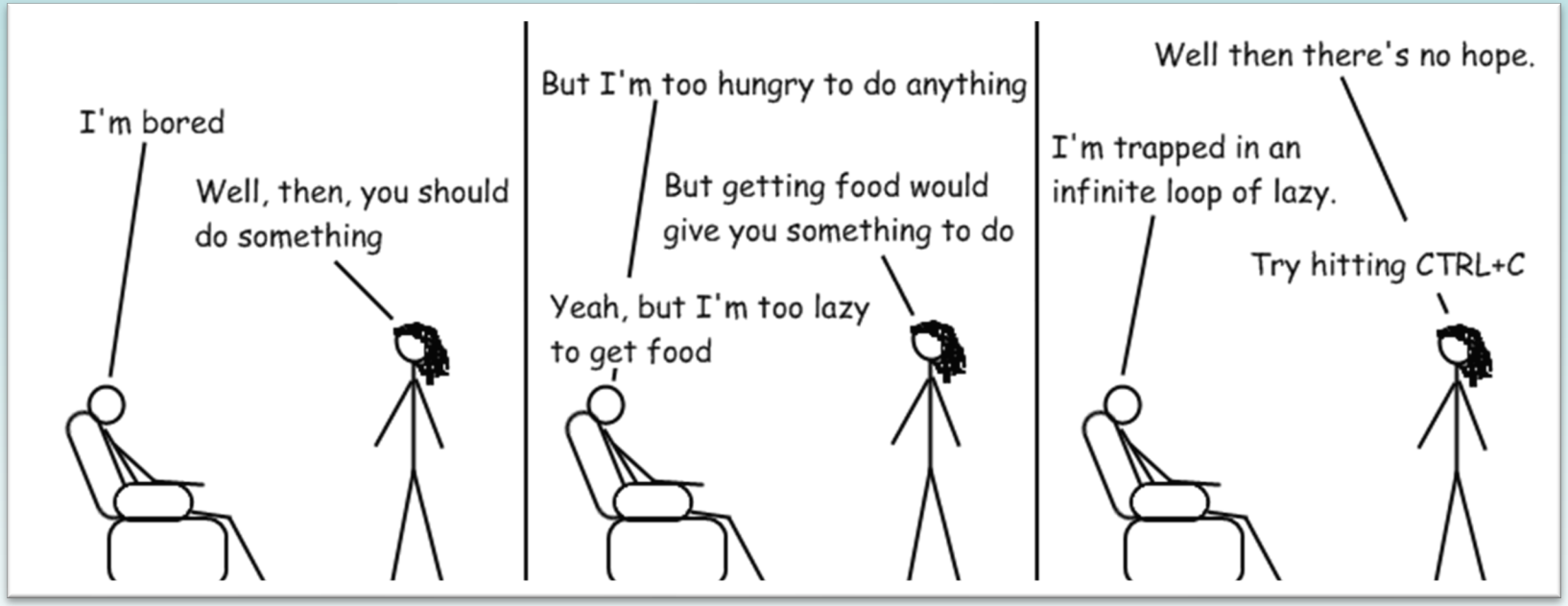
Infinite Loops

- An example of an infinite loop:

```
int count = 1;
while (count <= 25)
{
    System.out.println(count);
    count = count - 1;
}
```

1. Initialize
2. Test
3. Incorrect change

- This loop will continue executing until interrupted (Control-C) or integer overflow produces a bogus positive



Nested Loops

- Similar to nested `if` statements, loops can be nested as well:
  - The body of a loop can contain another loop
- For each iteration of the outer loop, the inner loop runs from start to finish

Nested Loop

How many times will the string "Here" be printed?

```
count1 = 1;
while (count1 <= 7)
{
    count2 = 1;
    while (count2 < 5)
    {
        System.out.println("Here");
        count2++;
    }
    count1++;
}
```

1. Initialize
2. Test
1. Initialize
2. Test
3. change
3. change

Here
Here
Here
Here
} count1 == 1

Here
Here
Here
Here
} count1 == 2

Here
Here
Here
Here
} count1 == 3

Here
Here
Here
Here
} count1 == 4

Here
Here
Here
Here
} count1 == 5

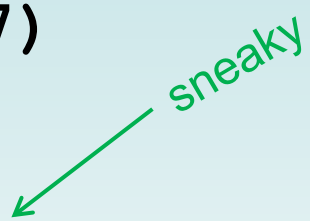
Here
Here
Here
Here
} count1 == 6

Here
Here
Here
Here
} count1 == 7

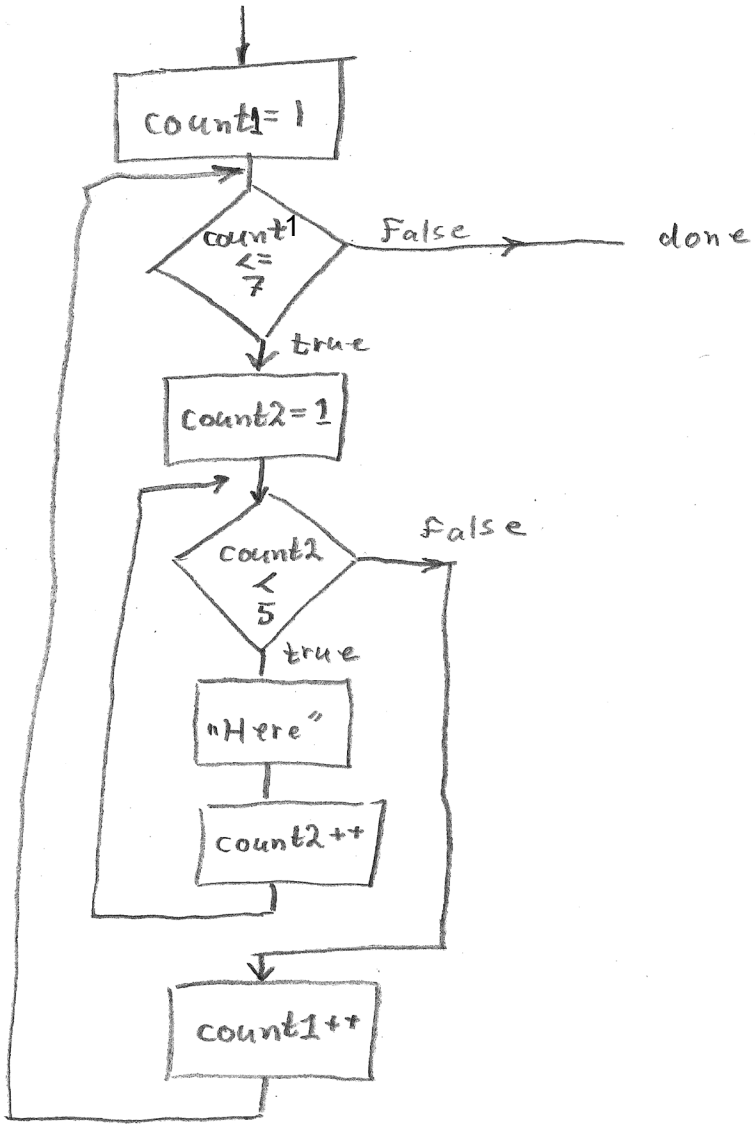
Nested Loop

How many times will the string "Here" be printed?

```
count1 = 1;
while (count1 <= 7)
{
    count2 = 1;
    while (count2 < 5)
    {
        System.out.println("Here");
        count2++;
    }
    count1++;
}
```

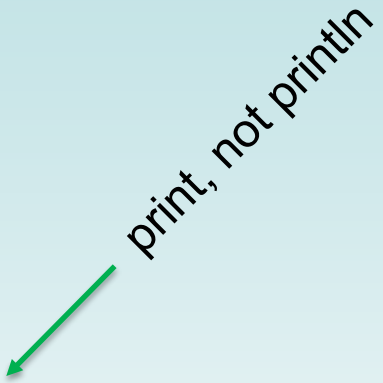


7 * 4 = 28

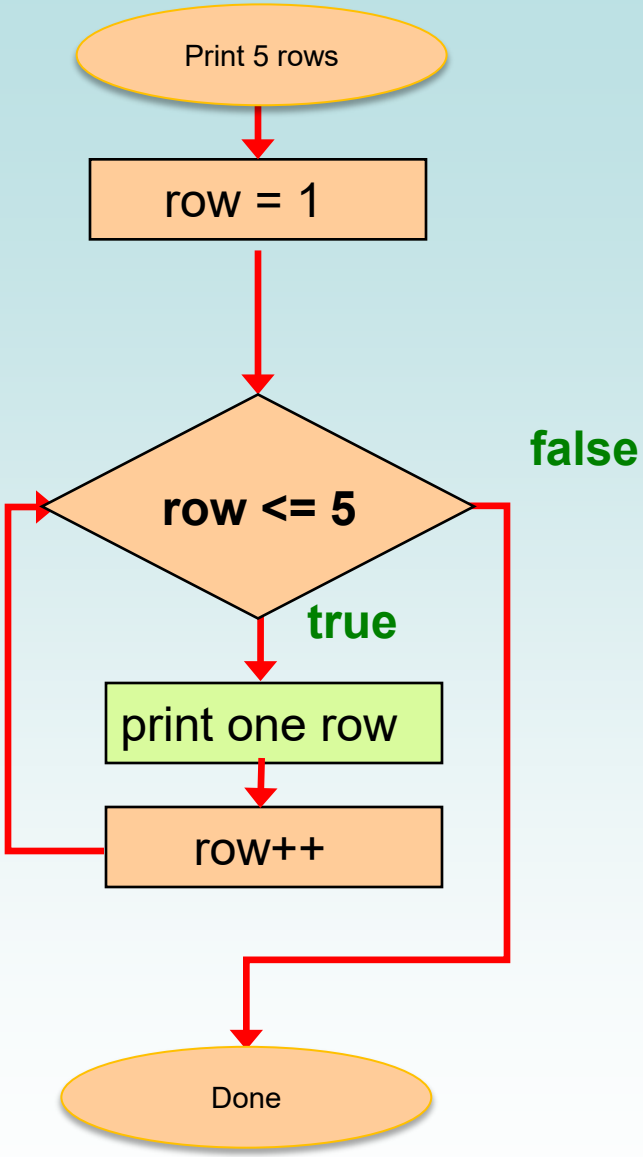


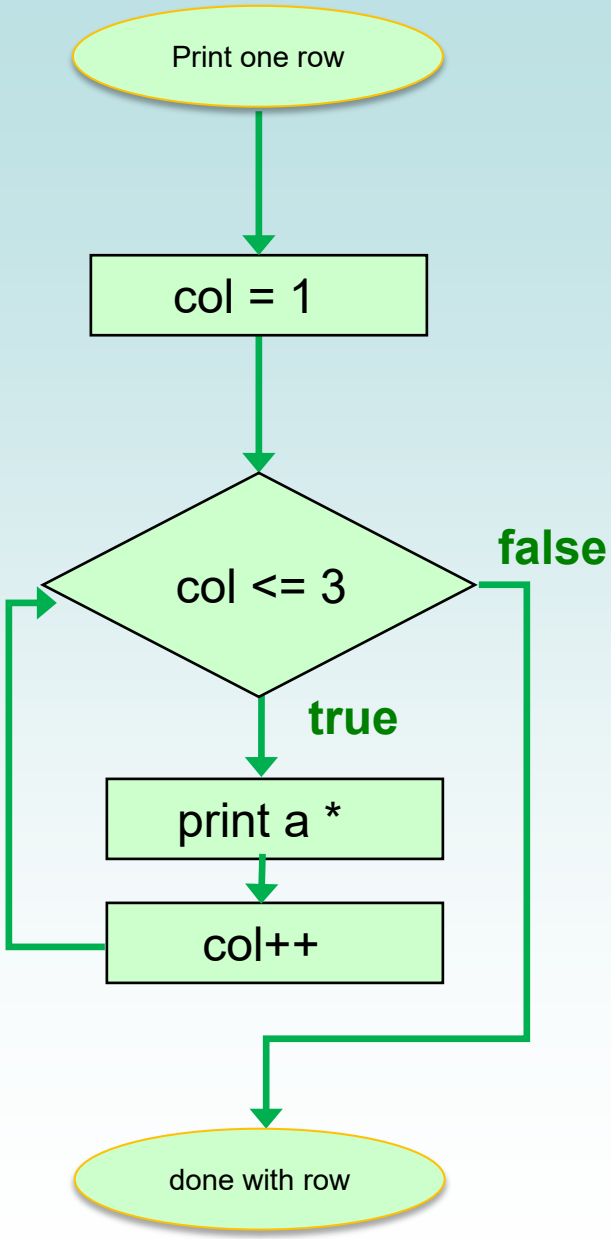
Rectangle of Stars

```
int row = 1;
while (row <= 5)
{
    int col = 1;
    while ( col <= 3)
    {
        System.out.print("*");
        col++;
    }
    System.out.println();
    row++;
}
```




```
***
***
***
***
***
```



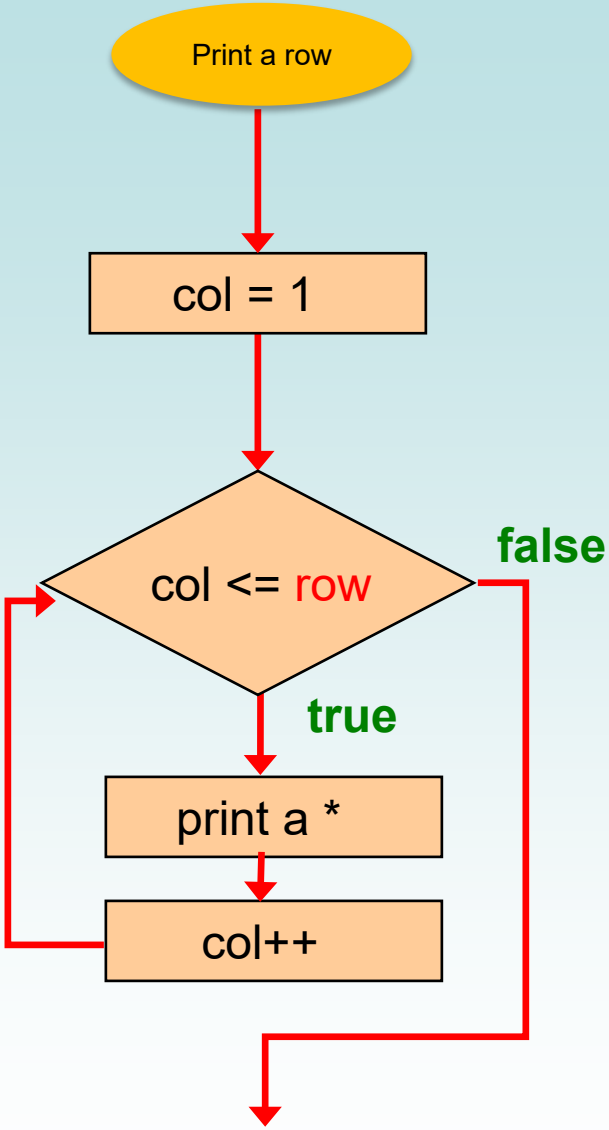


***

Triangle of Stars

```
int row = 1;
while (row <= 5)
{
    int col = 1;
    while ( col <= row)
    {
        System.out.print("*");
        col++;
    }
    System.out.println();
    row++;
}
```

```
*
**
***
****
*****
```



User Interaction

- Often you want the program to work for several cases, one after another, until the user quits.
- Ask the user if they want to keep going after each case.
- User enters “Y” or “N”

```
import java.util.Scanner;

public class KeepGoing
{

   public static void main(String[] args)
   {
      String str, another = "y";

      Scanner scan = new Scanner(System.in);

      while (another.equalsIgnoreCase("y"))  // allows y or Y
      {
         System.out.println("Enter data:");
         str = scan.nextLine();
         System.out.println("You entered: " + str);
         System.out.print("More data(y/n)? ");
         another = scan.nextLine();
      }
   }

}
```

Palindromes

- A palindrome is a string that reads the same backward as forward
- For us, spaces and case will matter:
- radar
- step on no pets
- able was I ere I saw elba
- For some palindromes, spaces, punctuation, and case don’t matter:
- Never odd or even
- Tons of UFO snot
- Eva, can I stab bats in a cave?

Palindromes

To mechanically determine if a string is a palindrome, match left and right characters until one or none are left.

```
radar
radar
radar
```

```
step on no pets
step on no pets
step on no pets
step on no pets
step_on no_pets
step on no pets
step on no pets
step on_no pets
```

```
//********************************************************************
//  PalindromeTester.java       Author: Lewis/Loftus
//
//  Demonstrates the use of nested while loops.
//********************************************************************

import java.util.Scanner;

public class PalindromeTester
{
   //-----------------------------------------------------------------
   //  Tests strings to see if they are palindromes.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      String str, another = "y";
      int left, right;

      Scanner scan = new Scanner(System.in);

      while (another.equalsIgnoreCase("y"))  // allows y or Y
      {
         System.out.println("Enter a potential palindrome:");
         str = scan.nextLine();
```

00000000011111
01234567890234
step on no pets

```
            left = 0;
            right = str.length() - 1;

            while (str.charAt(left) == str.charAt(right) && left < right)
            {
               left++;
               right--;
            }

            System.out.println();

            if (left < right)
               System.out.println("That string is NOT a palindrome.");
            else
               System.out.println("That string IS a palindrome.");

            System.out.println();
            System.out.print("Test another palindrome (y/n)? ");
            another = scan.nextLine();
         }
      }
   }
```

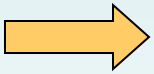
```
            while
            {
               lef
               rig
            }

            System

            if (le
               Sys
            else
               Sys

            System
            System
            anothe
        }
    }
}
```

```
& left < right)

alindrome.");

drome.");

n)? ");
```

Sample Run

```
Enter a potential palindrome:
radar

That string IS a palindrome.

Test another palindrome (y/n)? y
Enter a potential palindrome:
able was i ere i saw elba

That string IS a palindrome.

Test another palindrome (y/n)? y
Enter a potential palindrome:
tons of ufo snot

That string is NOT a palindrome.

Test another palindrome (y/n)? n
```

Outline

- Boolean Expressions
- The `if` Statement
- Comparing Data
- The `while` Statement
-  Iterators
- The `ArrayList` Class
  - ☺ Determining Event Sources
  - ☺ Managing Fonts
  - ☺ Check Boxes and Radio Buttons

§5.5 Iterators

- An *iterator* is a type of object which can access a collection of items one at a time
- It steps through each item in turn
- An iterator has a **`hasNext`** method that returns true if there is at least one more item to process
- The **`next`** method returns the next item

Iterators

- Important points to note:
  - You can iterate only in one direction
  - Iteration can be done only once
  - If you reach the end of series you're done
  - If you need to iterate again get a new Iterator

Iterators

- Several classes in the Java standard class library are iterators
- The **`Scanner`** class is an iterator
  - the **`hasNext`** method returns true if there are more items to be scanned
  - the **`next`** method returns the next item as a string
- The `Scanner` class also has variations on the `hasNext` method for specific data types
  - such as `hasNextInt`

☺ Iterators: see separate slides for file reading

- A **`Scanner`** can read from a file
- Let's read and process a list of URLs stored in a file
- Set up one scanner to read each line of the input until the end of the file is encountered
- Set up another scanner for each URL to process each part of the path
- See `URLDissector.java`

The file `urls.inp` contains:

```
www.google.com
www.linux.org/info/gnu.html
thelyric.com/calendar/
www.cs.vt.edu/undergraduate/about
youtube.com/watch?v=EHCRimwRGLs
```

```
Scanner fileScan ;
fileScan = new Scanner(new File("urls.inp"));
```

This opens the file `urls.inp` and creates a Scanner to read through it:

```
url = fileScan.nextLine();
urlScan = new Scanner(url);
```

This reads a complete line from the file and then creates a Scanner to scan through it.

```
import java.util.Scanner;
import java.io.*;

public class URLDissector
{
   public static void main(String[] args) throws IOException
   {
      String url;
      Scanner fileScan, urlScan;
      fileScan = new Scanner(new File("urls.inp"));

      // Read and process each line of the file
      while ( fileScan.hasNext() )
      {
         url = fileScan.nextLine();
         System.out.println("URL: " + url);

         // set up a Scanner for this line
         urlScan = new Scanner(url);
         urlScan.useDelimiter("/");     // tokens are separated by /

         //  Print each part of the url
         while ( urlScan.hasNext() )
            System.out.println ( "    " + urlScan.next() );

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

```
        // Rea
        while
        {
            url
            Sys

            url
            url

            //
            whi


            Sys
        }
    }
}
```

Sample Run

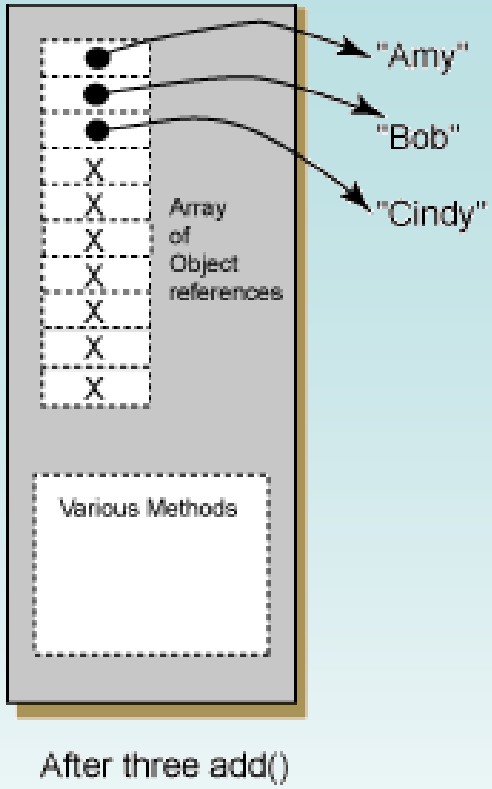
```
URL: www.google.com
    www.google.com

URL: www.linux.org/info/gnu.html
    www.linux.org
    info
    gnu.html

URL: thelyric.com/calendar/
    thelyric.com
    calendar

URL: www.cs.vt.edu/undergraduate/about
    www.cs.vt.edu
    undergraduate
    about

URL: youtube.com/watch?v=EHCRimwRGLs
    youtube.com
    watch?v=EHCRimwRGLs
```

Outline

- Boolean Expressions
- The `if` Statement
- Comparing Data
- The `while` Statement
  - ☺ Iterators
-  The `ArrayList` Class
  - ☺ Determining Event Sources
  - ☺ Managing Fonts
  - ☺ Check Boxes and Radio Buttons

§ 5.6 The ArrayList Class

- An `ArrayList` object stores a list of object references and is often processed using a loop



After three add()

- Each object reference in the list is called an *element*
- The `ArrayList` class is part of the `java.util` package

The ArrayList Class

- Each object reference in the list has a numeric index
- An `ArrayList` object grows and shrinks as needed, adjusting its capacity as necessary
- See https://programmedlessons.org/Java9/chap85/ch85_01.html

The ArrayList Class

- Index values of an `ArrayList` begin at 0 (not 1):

0	"Amy"
1	"Bob"
2	"Cindy"

- Elements can be inserted and removed
- The indexes of the elements are adjusted accordingly

The ArrayList Class

- The type of object reference stored in the list is established when the `ArrayList` object is created:

```
ArrayList<String> names = new ArrayList<String>();

ArrayList<Book> list = new ArrayList<Book>();
```

- This makes use of Java *generics*, which provide additional type checking at compile time
- An `ArrayList` object cannot store primitive types
  - The elements are always pointers (references)
  - Use a wrapper class if you want a list of a primitive type

ArrayList Methods

- Some `ArrayList` methods:

- **`boolean add(E obj)` – add an object to the end of the list**
- **`void add(int index, E obj)` – insert an object at index, move objects up to make room**
- **`Object remove(int index)` – delete the object at index. Return a pointer to it. Move objects down.**
- **`Object get(int index)` – return a reference to the object at index**
- **`boolean isEmpty()` – true / false**
- **`int size()` – number of objects in the list**
- **`int indexOf( Object obj )` – return the index of the first object obj in the list, or -1 if not found.**

```
//********************************************************************
//  Beatles.java       Author: Lewis/Loftus
//
//  Demonstrates the use of a ArrayList object.
//********************************************************************

import java.util.ArrayList;

public class Beatles
{
   //-----------------------------------------------------------------
   //  Stores and modifies a list of band members.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      ArrayList<String> band = new ArrayList<String>();

      band.add("Paul");
      band.add("Pete");
      band.add("John");
      band.add("George");
```

```
        System.out.println (band);
        int location = band.indexOf ("Pete");
        band.remove (location);

        System.out.println (band);
        System.out.println ("At index 1: " + band.get(1));
        band.add (2, "Ringo");

        System.out.println ("Size of the band: " + band.size());
        int index = 0;
        while (index < band.size())
        {
           System.out.println(band.get(index));
           index++;
        }
    }
}
```

Output

```
[Paul, Pete, John, George]
[Paul, John, George]
At index 1: John
Size of the band: 4
Paul
John
Ringo
George
```

```
import java.util.ArrayList;
public class Beatles
{
   public static void main(String[] args)
   {
      ArrayList<String> band = new ArrayList<String>();

      band.add("Paul");
      band.add("Pete");
      band.add("John");
      band.add("George");
      System.out.println (band);
      int location = band.indexOf ("Pete");
      band.remove (location);

      System.out.println (band);
      System.out.println ("At index 1: " + band.get(1));
      band.add (2, "Ringo");

      System.out.println ("Size of the band: " + band.size());
      int index = 0;
      while (index < band.size())
      {
         System.out.println(band.get(index));
         index++;
      }
   }
}
```

Output

```
[Paul, Pete, John, George]
[Paul, John, George]
At index 1: John
Size of the band: 4
Paul
John
Ringo
George
```

Summary

- Chapter 5 focused on:
  - boolean expressions
  - the if and if-else statements
  - comparing data
  - while loops
  - ☺ iterators
  - the `ArrayList` class
  - ☺ more GUI controls