



Flow of Control

Chapter 3

Objectives

- Use Java branching statements
- Compare values of primitive types
- Compare objects such as strings
- Use the primitive type **boolean**
- Use simple enumerations in a program
- Use color in a graphics program
- Use **JOptionPane** class to create yes-no dialog box

Outline

- The **if-else** Statement
- The Type **boolean**
- The **switch** statement
- (optional) Graphics Supplement

Flow of Control

- *Flow of control* is the order in which a program performs actions.
 - Up to this point, the order has been sequential.
- A *branching statement* chooses between two or more possible actions.
- A *loop statement* repeats an action until a stopping condition occurs.

The **if-else** Statement: Outline

- Basic **if-else** Statement
- Boolean Expressions
- Comparing Strings
- Nested **if-else** Statements
- Multibranch **if-else** Statements
- The **switch** Statement
- (optional) The Conditional Operator
- The **exit** Method

The *if-else* Statement

- A branching statement that chooses between two possible actions.
- Syntax

```
if (Boolean_Expression)  
    Statement_1  
else  
    Statement_2
```

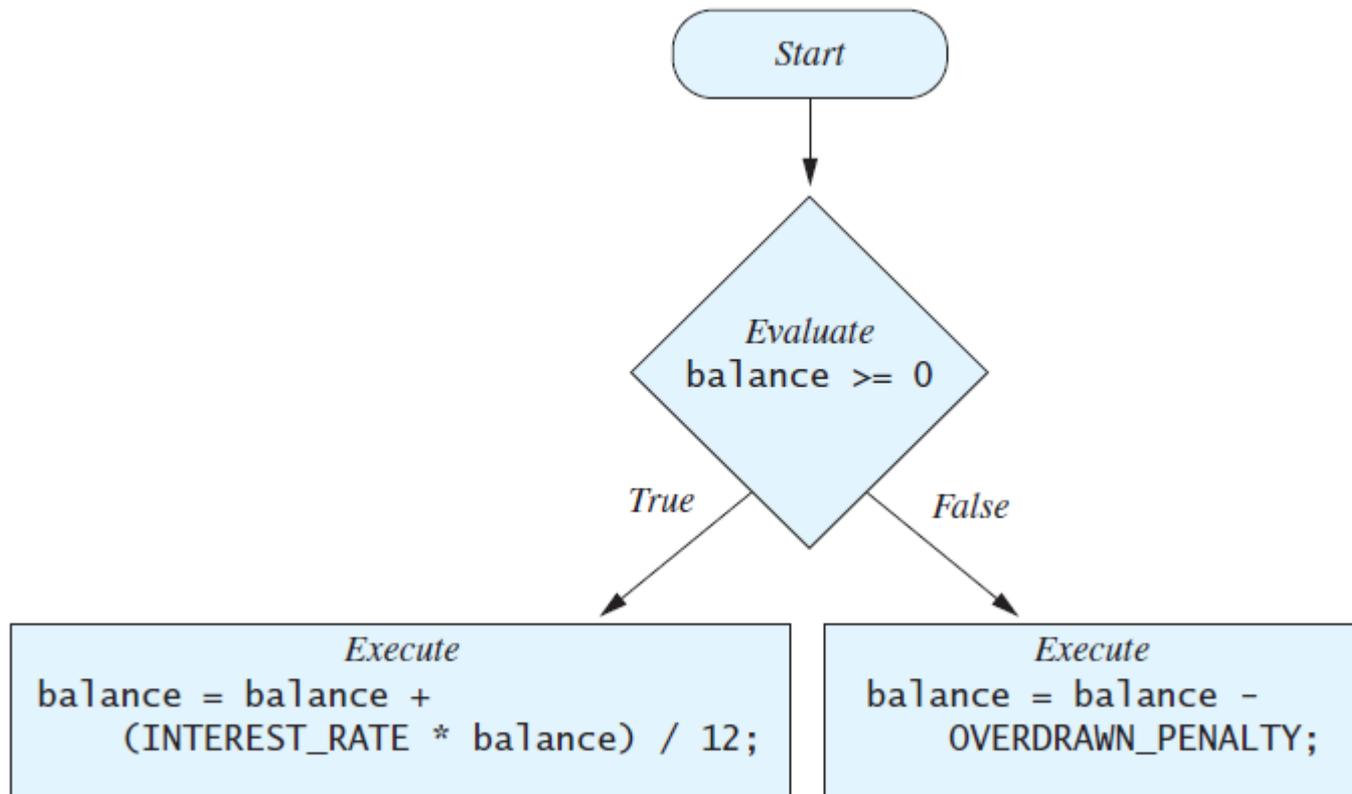
The **if-else** Statement

- Example

```
if (balance >= 0)
    balance = balance + (INTEREST_RATE * balance) / 12;
else
    balance = balance - OVERDRAWN_PENALTY;
```

The **if-else** Statement

- Figure 3.1 The Action of the **if-else** Statement [sample program](#)
Listing 3.1



The **if-else** Statement

Sample
screen
output

```
Enter your checking account balance: $505.67
```

```
Original balance $505.67
```

```
After adjusting for one month of interest and penalties,  
your new balance is $506.51278
```

```
Enter your checking account balance: $-15.53
```

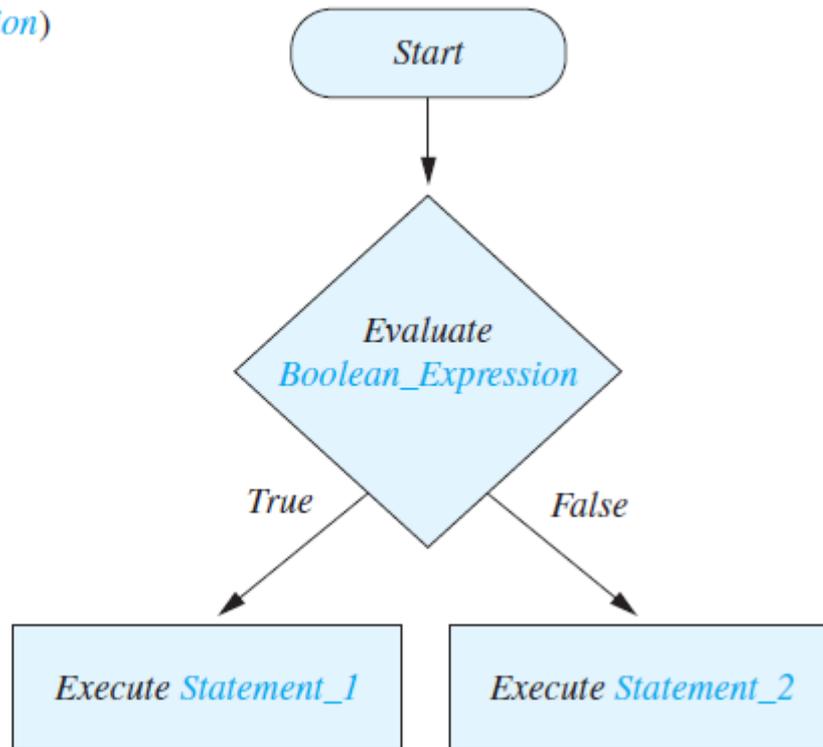
```
Original balance $-15.53
```

```
After adjusting for one month of interest and penalties,  
your new balance is $-23.53
```

Semantics of the *if-else* Statement

- Figure 3.2

```
if (Boolean_Expression)  
    Statement_1  
else  
    Statement_2
```



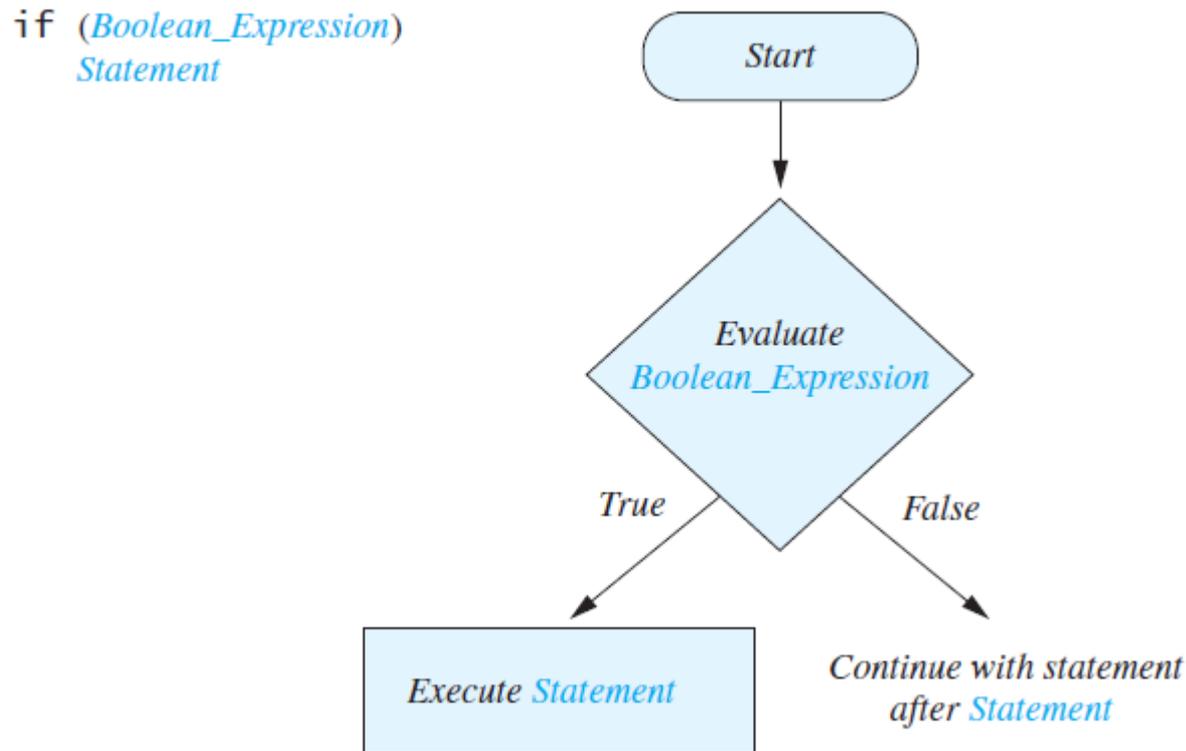
Compound Statements

- To include multiple statements in a branch, enclose the statements in braces.

```
if (count < 3)
{
    total = 0;
    count = 0;
}
```

Omitting the **else** Part

- FIGURE 3.3 The Semantics of an **if** Statement without an **else**



Introduction to Boolean Expressions

- The value of a *boolean expression* is either **true** or **false**.
- Examples

`time < limit`

`balance <= 0`

Java Comparison Operators

- Figure 3.4 Java Comparison Operators

Math Notation	Name	Java Notation	Java Examples
=	Equal to	==	<code>balance == 0</code> <code>answer == 'y'</code>
≠	Not equal to	!=	<code>income != tax</code> <code>answer != 'y'</code>
>	Greater than	>	<code>expenses > income</code>
≥	Greater than or equal to	>=	<code>points >= 60</code>
<	Less than	<	<code>pressure < max</code>
≤	Less than or equal to	<=	<code>expenses <= income</code>

Compound Boolean Expressions

- Boolean expressions can be combined using the "and" (`&&`) operator.

- Example

```
if ((score > 0) && (score <= 100))  
    ...
```

- Not allowed

```
if (0 < score <= 100)  
    ...
```

Compound Boolean Expressions

- Syntax

(Sub_Expression_1) && (Sub_Expression_2)

- Parentheses often are used to enhance readability.
- The larger expression is true only when both of the smaller expressions are true.

Compound Boolean Expressions

- Boolean expressions can be combined using the "or" (`||`) operator.

- Example

```
if ((quantity > 5) || (cost < 10))
```

```
...
```

- Syntax

```
(Sub_Expression_1) || (Sub_Expression_2)
```

Compound Boolean Expressions

- The larger expression is true
 - When either of the smaller expressions is true
 - When both of the smaller expressions are true.
- The Java version of "or" is the *inclusive or* which allows either or both to be true.
- The *exclusive or* allows one or the other, but not both to be true.

Negating a Boolean Expression

- A boolean expression can be negated using the "not" (!) operator.

- Syntax

!(Boolean_Expression)

- Example

(a || b) && !(a && b)

which is the *exclusive or*

Negating a Boolean Expression

- Figure 3.5 Avoiding the Negation Operator

! (A Op B) Is Equivalent to (A Op B)	
<	>=
<=	>
>	<=
>=	<
==	!=
!=	==

Java Logical Operators

- Figure 3.6

Name	Java Notation	Java Examples
Logical <i>and</i>	&&	<code>(sum > min) && (sum < max)</code>
Logical <i>or</i>		<code>(answer == 'y') (answer == 'Y')</code>
Logical <i>not</i>	!	<code>!(number < 0)</code>

Boolean Operators

- FIGURE 3.7 The Effect of the Boolean Operators **&&** (and), **| |** (or), and **!** (not) on Boolean values

Value of <i>A</i>	Value of <i>B</i>	Value of <i>A && B</i>	Value of <i>A B</i>	Value of <i>! (A)</i>
true	true	true	true	false
true	false	false	true	false
false	true	false	true	true
false	false	false	false	true

Using `==`

- `==` is appropriate for determining if two integers or characters have the same value.

```
if (a == 3)
```

where `a` is an integer type

- `==` is **not** appropriate for determining if two floating points values are equal. Use `<` and some appropriate tolerance instead.

```
if (abs(b - c) < epsilon)
```

where `b`, `c`, and `epsilon` are floating point types

Using `==`

- `==` is not appropriate for determining if two objects have the same value.
 - `if (s1 == s2)`, where `s1` and `s2` refer to strings, determines only if `s1` and `s2` refer to a common memory location.
 - If `s1` and `s2` refer to strings with identical sequences of characters, but stored in different memory locations, `(s1 == s2)` is false.

Using `==`

- To test the equality of objects of class String, use method **`equals`**.

```
s1.equals(s2)
```

or

```
s2.equals(s1)
```

- To test for equality ignoring case, use method **`equalsIgnoreCase`**.

```
("Hello".equalsIgnoreCase("hello"))
```

`equals` and `equalsIgnoreCase`

- Syntax

`String.equals(Other_String)`

`String.equalsIgnoreCase(Other_String)`

Testing Strings for Equality

- View [sample program](#) Listing 3.2
`class StringEqualityDemo`

```
Enter two lines of text:  
Java is not coffee.  
Java is NOT COFFEE.  
The two lines are not equal.  
The two lines are not equal.  
But the lines are equal, ignoring case.
```

Sample
screen
output

Lexicographic Order

- Lexicographic order is similar to alphabetical order, but is it based on the order of the characters in the ASCII (and Unicode) character set.
 - All the digits come before all the letters.
 - All the uppercase letters come before all the lower case letters.

Lexicographic Order

- Strings consisting of alphabetical characters can be compared using method `compareTo` and method `toUpperCase` or method `toLowerCase`.

```
String s1 = "Hello";  
String lowerS1 = s1.toLowerCase();  
String s2 = "hello";  
if (s1.compareTo(s2) == 0  
    System.out.println("Equal!");
```

Method `compareTo`

- Syntax

`String_1.compareTo(String_2)`

- Method `compareTo` returns

- a negative number if `String_1` precedes `String_2`
- zero if the two strings are equal
- a positive number if `String_2` precedes `String_1`.

Nested **if-else** Statements

- An **if-else** statement can contain any sort of statement within it.
- In particular, it can contain another **if-else** statement.
 - An **if-else** may be nested within the "if" part.
 - An **if-else** may be nested within the "else" part.
 - An **if-else** may be nested within both parts.

Nested Statements

- Syntax

```
if (Boolean_Expression_1)
    if (Boolean_Expression_2)
        Statement_1)
    else
        Statement_2)
else
    if (Boolean_Expression_3)
        Statement_3)
    else
        Statement_4);
```

Nested Statements

- Each **else** is paired with the nearest unmatched **if**.
- **If used properly**, indentation communicates which **if** goes with which **else**.
- Braces can be used like parentheses to group statements.

Nested Statements

- Subtly different forms

First Form

```
if (a > b)
{
    if (c > d)
        e = f;
}
else
    g = h;
```

Second Form

```
if (a > b)
    if (c > d)
        e = f;
    else
        g = h;

// oops
```

Compound Statements

- When a list of statements is enclosed in braces (`{ }`), they form a single *compound statement*.
- Syntax

```
{  
    Statement_1;  
    Statement_2;  
    ...  
}
```

Compound Statements

- A compound statement can be used wherever a statement can be used.
- Example

```
if (total > 10)
{
    sum = sum + total;
    total = 0;
}
```

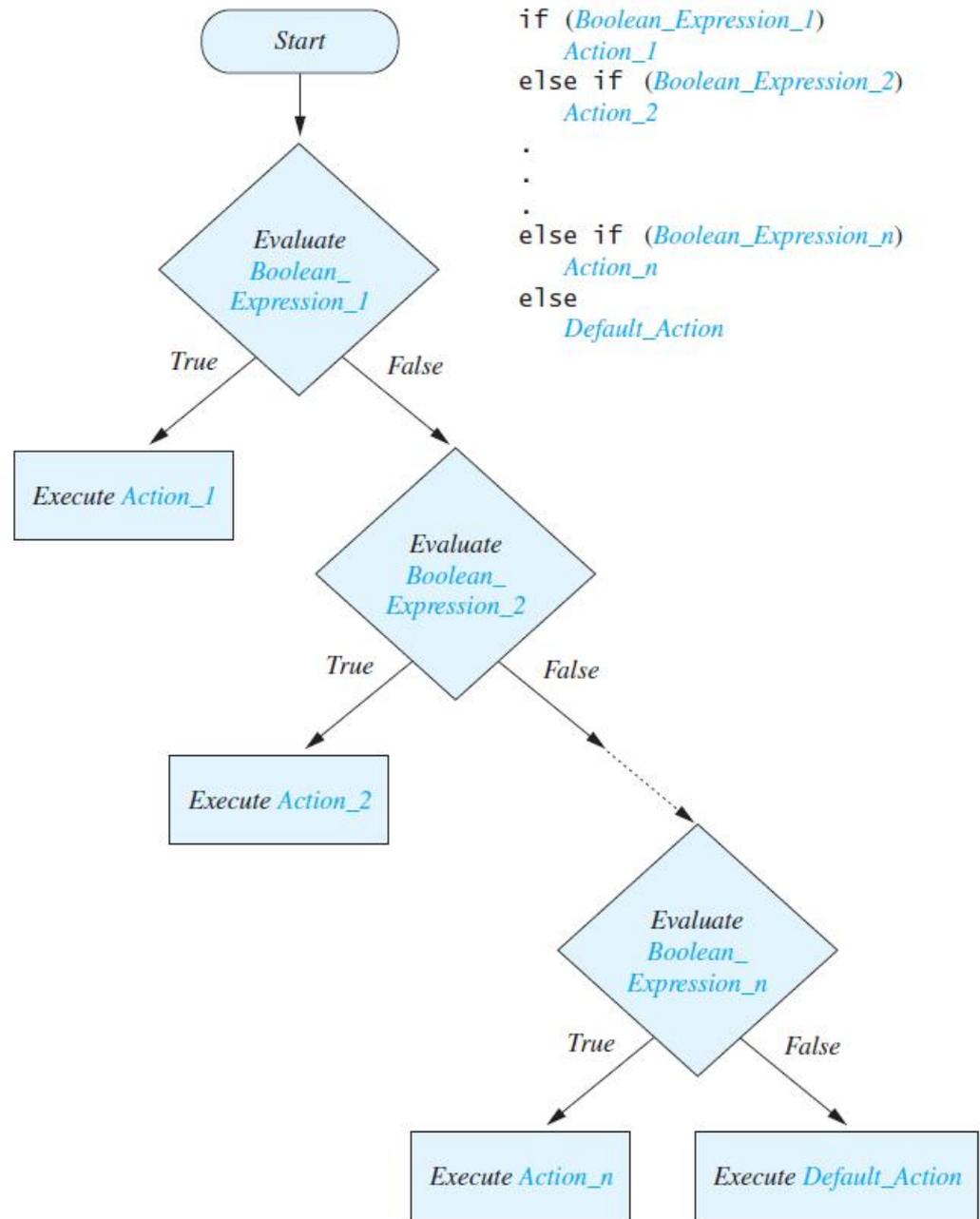
Multibranch **if-else** Statements

- Syntax

```
if (Boolean_Expression_1)  
    Statement_1  
else if (Boolean_Expression_2)  
    Statement_2  
else if (Boolean_Expression_3)  
    Statement_3  
else if ...  
else  
    Default_Statement
```

Multibranch *if-else* Statements

- Figure 3.8 Semantics



Multibranch **if-else** Statements

- View [sample program](#) Listing 3.3
class Grader

```
Enter your score:
```

```
85
```

```
Score = 85
```

```
Grade = B
```

Sample
screen
output

Multibranch **if-else** Statements

- Equivalent code

```
if (score >= 90)
    grade = 'A';
else if ((score >= 80) && (score < 90))
    grade = 'B';
else if ((score >= 70) && (score < 80))
    grade = 'C';
else if ((score >= 60) && (score < 70))
    grade = 'D';
else
    grade = 'F';
```

Case Study – Body Mass Index

- Body Mass Index (BMI) is used to estimate the risk of weight-related problems
- $BMI = \text{mass} / \text{height}^2$
 - Mass in kilograms, height in meters
- Health assessment if:
 - $BMI < 18.5$ Underweight
 - $18.5 \leq BMI < 25$ Normal weight
 - $25 \leq BMI < 30$ Overweight
 - $30 \leq BMI$ Obese

Case Study – Body Mass Index

- **Algorithm**

- Input height in feet & inches, weight in pounds
- Convert to meters and kilograms
 - 1 lb = 2.2 kg
 - 1 inch = 0.254 meters
- Compute BMI
- Output health risk using if statements

View [sample program](#) Listing 3.4

class BMI

The Conditional Operator

```
if (n1 > n2)
```

```
    max = n1;
```

```
else
```

```
    max = n2;
```

can be written as

```
max = (n1 > n2) ? n1 : n2;
```

- The **?** and **:** together are call the *conditional operator* or *ternary operator*.

The Conditional Operator

- The conditional operator is useful with print and println statements.

```
System.out.print("You worked " +  
    ((hours > 1) ? "hours" ; "hour"));
```

The `exit` Method

- Sometimes a situation arises that makes continuing the program pointless.
- A program can be terminated normally by `System.exit(0)`.

The `exit` Method

- Example

```
if (numberOfWinners == 0)
{
    System.out.println ("Error: Dividing by zero.");
    System.exit (0);
}
else
{
    oneShare = payoff / numberOfWinners;
    System.out.println ("Each winner will receive $"
+ oneShare);
}
```

The Type **boolean**

- The type **boolean** is a primitive type with only two values: **true** and **false**.
- Boolean variables can make programs more readable.

```
if (systemsAreOK)
```

instead of

```
if ((temperature <= 100) && (thrust >= 12000) &&  
    (cabinPressure > 30) && ...)
```

Boolean Expressions and Variables

- Variables, constants, and expressions of type **boolean** all evaluate to either **true** or **false**.
- A boolean variable can be given the value of a boolean expression by using an assignment operator.

```
boolean isPositive = (number > 0);
```

```
...
```

```
if (isPositive) ...
```

Naming Boolean Variables

- Choose names such as **isPositive** or **systemsAreOk**.
- Avoid names such as **numberSign** or **systemStatus**.

Precedence Rules

- Parentheses should be used to indicate the order of operations.
- When parentheses are omitted, the order of operation is determined by *precedence rules*.

Precedence Rules

- Operations with *higher precedence* are performed before operations with *lower precedence*.
- Operations with *equal precedence* are done left-to-right (except for unary operations which are done right-to-left).

Precedence Rules

- Figure 3.9

Highest Precedence

First: the unary operators +, −, ++, −−, and !

Second: the binary arithmetic operators *, /, %

Third: the binary arithmetic operators +, −

Fourth: the boolean operators <, >, <=, >=

Fifth: the boolean operators ==, !=

Sixth: the boolean operator &

Seventh: the boolean operator |

Eighth: the boolean operator &&

Ninth: the boolean operator ||

Lowest Precedence

Precedence Rules

- In what order are the operations performed?

```
score < min/2 - 10 || score > 90
```

```
score < (min/2) - 10 || score > 90
```

```
score < ((min/2) - 10) || score > 90
```

```
(score < ((min/2) - 10)) || score > 90
```

```
(score < ((min/2) - 10)) || (score > 90)
```

Short-circuit Evaluation

- Sometimes only part of a boolean expression needs to be evaluated to determine the value of the entire expression.
 - If the first operand associated with an `||` is **true**, the expression is **true**.
 - If the first operand associated with an `&&` is **false**, the expression is **false**.
- This is called *short-circuit* or *lazy* evaluation.

Short-circuit Evaluation

- Short-circuit evaluation is not only efficient, sometimes it is essential!
- A run-time error can result, for example, from an attempt to divide by zero.

```
if ( (number != 0) && (sum/number > 5) )
```

- *Complete evaluation* can be achieved by substituting `&` for `&&` or `|` for `||`.

Input and Output of Boolean Values

- Example

```
boolean booleanVar = false;  
System.out.println(booleanVar);  
System.out.println("Enter a boolean value:");  
Scanner keyboard = new Scanner(System.in);  
booleanVar = keyboard.nextBoolean();  
System.out.println("You entered " + booleanVar);
```

Input and Output of Boolean Values

- Dialog

`false`

`Enter a boolean value: true`

`true`

`You entered true`

Input Validation

- You should check your input to ensure that it is within a valid or reasonable range. For example, consider a program that converts feet to inches. You might write the following:

```
int feet = keyboard.nextInt();  
int inches = feet * 12;
```

- What if:
 - The user types a negative number for feet?
 - The user enters an unreasonable value like 100? Or a number larger than can be stored in an int? (2,147,483,647)

Input Validation

- Address these problems by ensuring that the entered values are reasonable:

```
int feet = keyboard.nextInt();  
if ((feet >= 0) && (feet < 10))  
{  
    int inches = feet * 12;  
    ...  
}
```

The **switch** Statement

- The **switch** statement is a multiway branch that makes a decision based on an *integral* (integer or character) expression.
 - Java 7 allows String expressions
- The **switch** statement begins with the keyword **switch** followed by an integral expression in parentheses and called the *controlling expression*.

The **switch** Statement

- A list of cases follows, enclosed in braces.
- Each case consists of the keyword **case** followed by
 - A constant called the *case label*
 - A colon
 - A list of statements.
- The list is searched for a case label matching the controlling expression.

The **switch** Statement

- The action associated with a matching case label is executed.
- If no match is found, the case labeled **default** is executed.
 - The **default** case is optional, but recommended, even if it simply prints a message.
- Repeated case labels are not allowed.

The **switch** Statement

- Syntax

```
switch (Controlling_Expression)  
{  
    case Case_Label:  
        Statement(s);  
        break;  
    case Case_Label:  
    ...  
    default:  
    ...  
}
```

The **switch** Statement

- View [sample program](#) Listing 3.5
class MultipleBirths

```
Enter number of babies: 1  
Congratulations.
```

```
Enter number of babies: 3  
Wow. Triplets.
```

```
Enter number of babies: 4  
Unbelievable; 4 babies.
```

```
Enter number of babies: 6  
I don't believe you.
```

Sample
screen
output

The **switch** Statement

- The action for each case typically ends with the word **break**.
- The optional **break** statement prevents the consideration of other cases.
- The controlling expression can be anything that evaluates to an integral type.

Enumerations

- Consider a need to restrict contents of a variable to certain values
- An enumeration lists the values a variable can have
- Example

```
enum MovieRating {E, A, B}  
MovieRating rating;  
rating = MovieRating.A;
```

Enumerations

- Now possible to use in a **switch** statement

```
switch (rating)
{
    case E: //Excellent
        System.out.println("You must see this movie!");
        break;
    case A: //Average
        System.out.println("This movie is OK, but not great.");
        break;
    case B: // Bad
        System.out.println("Skip it!");
        break;
    default:
        System.out.println("Something is wrong.");
}
```

Enumerations

- An even better choice of descriptive identifiers for the constants

```
enum MovieRating
    {EXCELLENT, AVERAGE, BAD}
rating = MovieRating.AVERAGE;

case EXCELLENT:    ...
```

(Optional) Graphics Supplement: Outline

- Specifying a Drawing Color
- A **JOptionPane** Yes/No Window

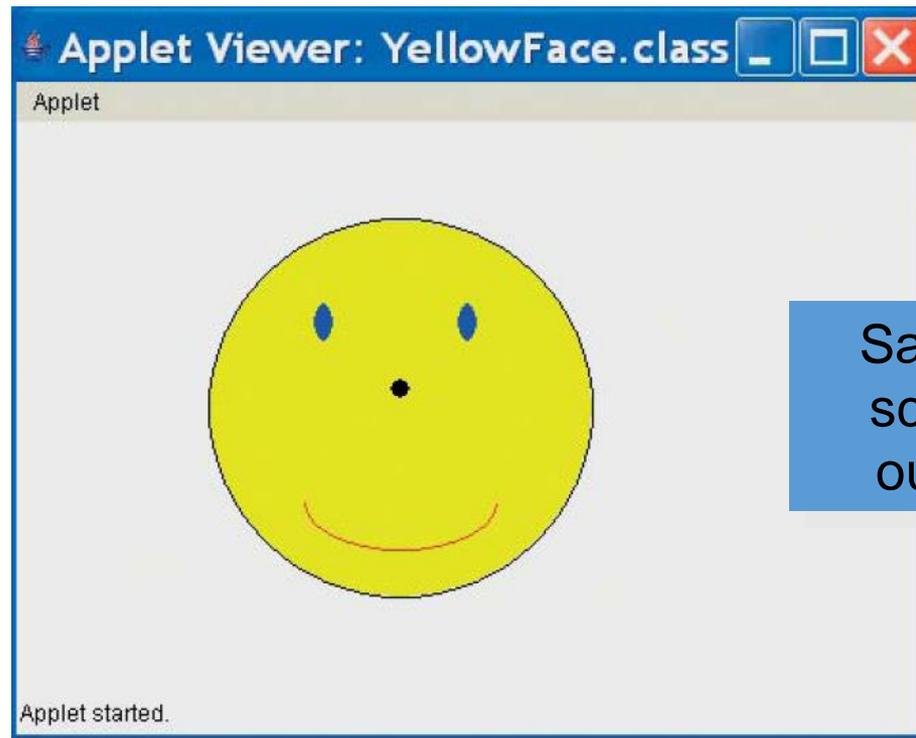
Specifying a Drawing Color

- When drawing a shape inside an applet's **paint** method, think of the drawing being done with a pen that can change colors.
- The method **setColor** changes the color of the "pen."
`canvas.setColor(Color.YELLOW) ;`
- Drawings done later appear on top of drawings done earlier.

Specifying a Drawing Color

- View [sample program](#), Listing 3.6

class YellowFace



Sample
screen
output

Specifying a Drawing Color

- Figure 3.10 Predefined Colors for the `setColor` Method

<code>Color.BLACK</code>	<code>Color.MAGENTA</code>
<code>Color.BLUE</code>	<code>Color.ORANGE</code>
<code>Color.CYAN</code>	<code>Color.PINK</code>
<code>Color.DARK_GRAY</code>	<code>Color.RED</code>
<code>Color.GRAY</code>	<code>Color.WHITE</code>
<code>Color.GREEN</code>	<code>Color.YELLOW</code>
<code>Color.LIGHT_GRAY</code>	

A Dialog Box for a Yes-or-No Question

- Used to present the user with a yes/no question
- The window contains
 - The question text
 - Two buttons labeled **yes** and **no**.

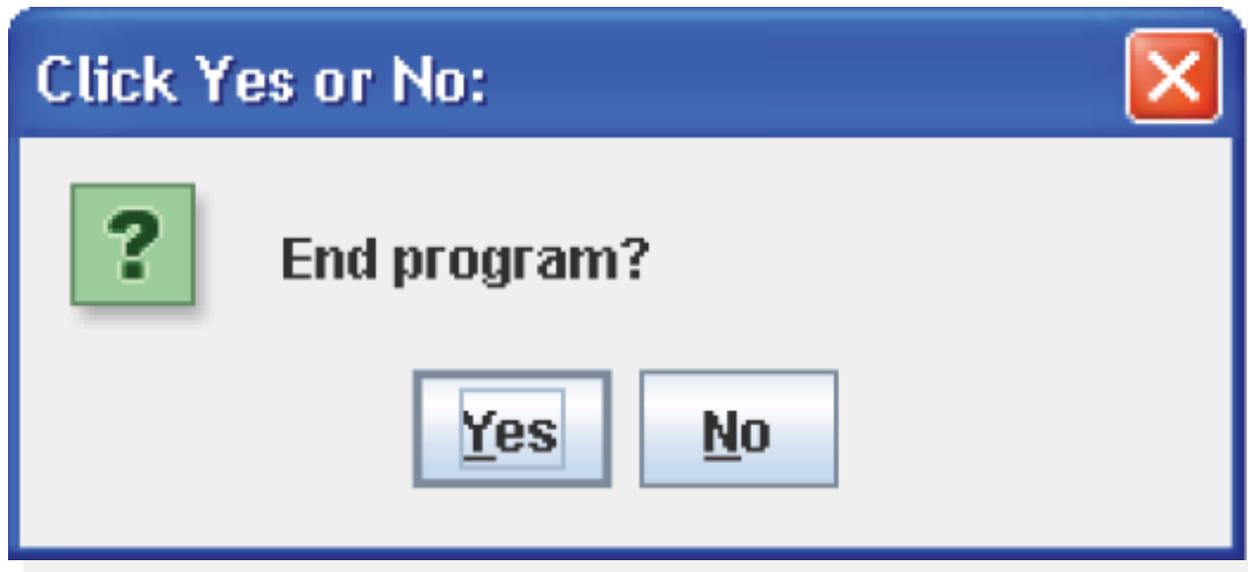
A Dialog Box for a Yes-or-No Question

- Example

```
int answer =
    JOptionPane.showConfirmDialog(null,
        "End program?",
        "Click Yes or No:",
        JOptionPane.YES_NO_OPTION);
if (answer == JOptionPane.YES_OPTION)
    System.exit(0);
else if (answer == JOptionPane.NO_OPTION)
    System.out.println("One more time");
```

A Dialog Box for a Yes-or-No Question

- Figure 3.11 A Yes-or No-Dialog Box



Summary

- You have learned about Java branching statements.
- You have learned about the type **boolean**.
- (optional) You have learned to use color and the **JOptionPane** yes/no window.