Java Coding 2

Decisions, decisions…!

The `if` Statement

© Media Bakery.

An `if` statement is like a fork in the road. Depending upon a decision, different parts of the program are executed.

The `if` Statement

- The `if` statement allows a program to carry out different actions depending on the nature of the data to be processed.

© DrGrounds/iStockphoto.

This elevator panel “skips” the thirteenth floor. The floor is not actually missing— the computer that controls the elevator adjusts the floor numbers above 13.

The if Statement

- Flowchart with two branches

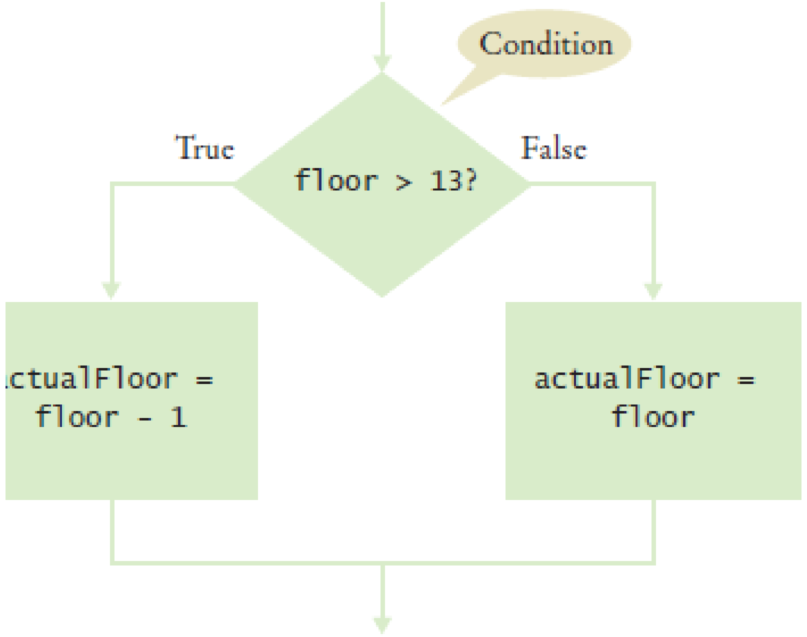


Figure 1
Flowchart for if Statement

- You can include as many statements in each branch as you like.

The if Statement

- Flowchart with one branches

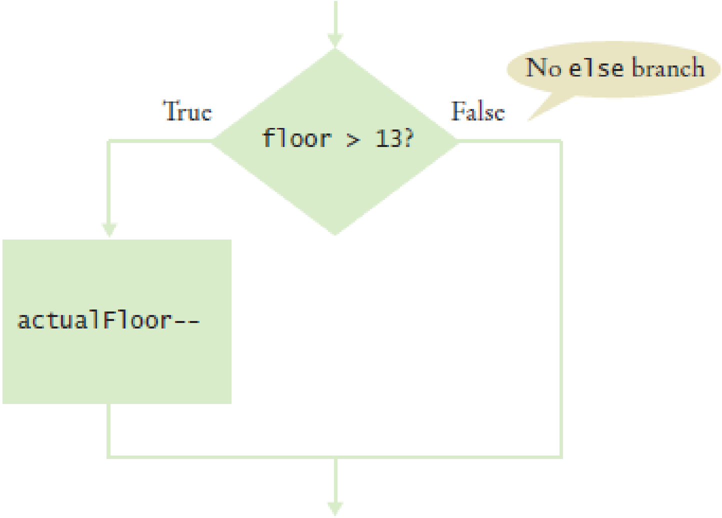
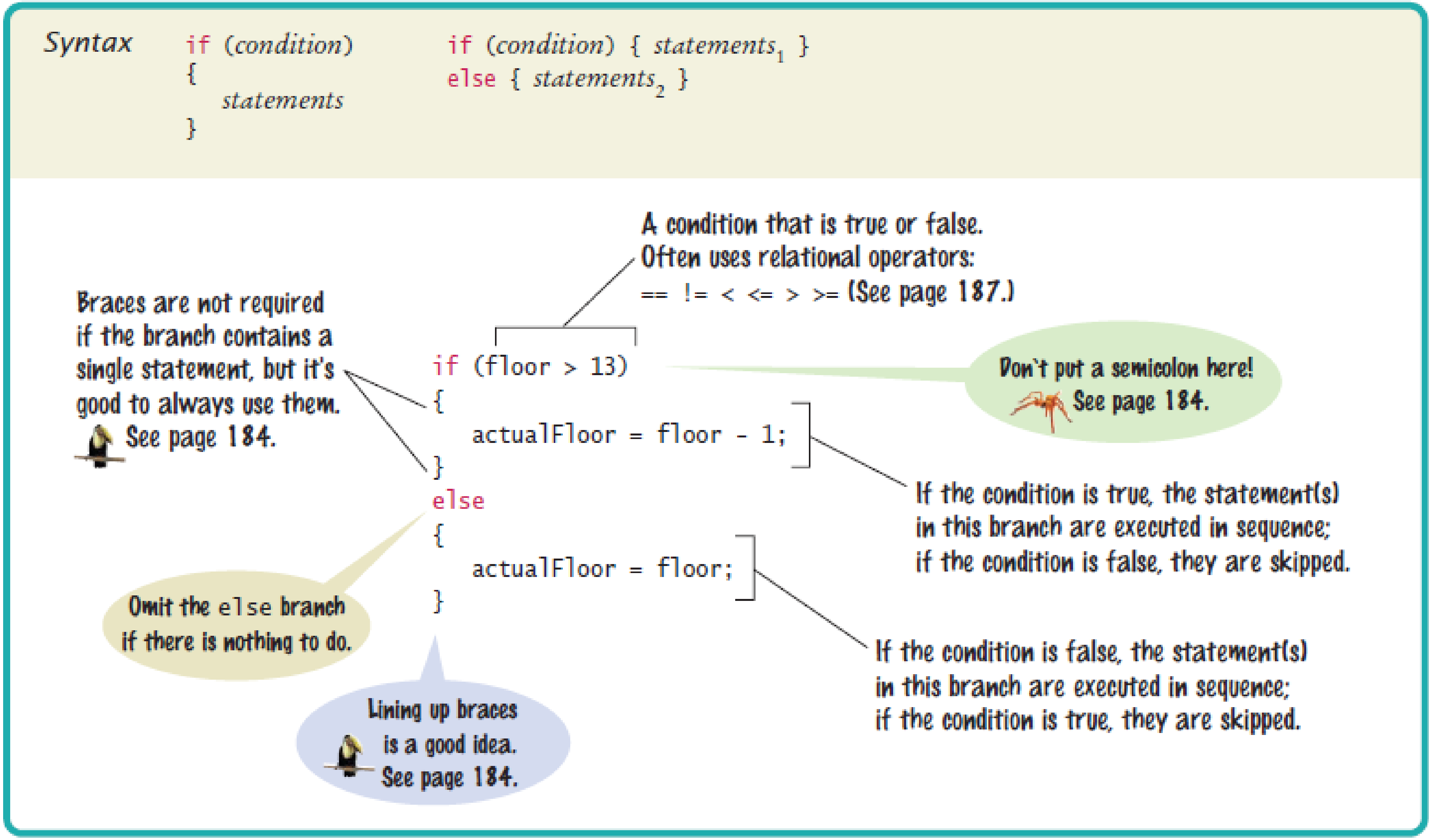


Figure 2
Flowchart for if Statement with No else Branch

- When there is nothing to do in the else branch, omit it entirely

```
int actualFloor = floor;
if (floor > 13)
{
   actualFloor--;
} // No else needed
```

Syntax 4.1 The if Statement



Syntax

```
if (condition)
{
   statements
}
```

```
if (condition) { statements1 }
else { statements2 }
```

```
if (floor > 13)
{
   actualFloor = floor - 1;
}
else
{
   actualFloor = floor;
}
```

A condition that is true or false. Often uses relational operators: == != < <= > >= (See page 187.)

Braces are not required if the branch contains a single statement, but it's good to always use them. See page 184.

Don't put a semicolon here! See page 184.

If the condition is true, the statement(s) in this branch are executed in sequence; if the condition is false, they are skipped.

Omit the else branch if there is nothing to do.

If the condition is false, the statement(s) in this branch are executed in sequence; if the condition is true, they are skipped.

Lining up braces is a good idea. See page 184.

section_1/ElevatorSimulation.java

```java
import java.util.Scanner;

/**
   This program simulates an elevator panel that skips the 13th floor.
*/
public class ElevatorSimulation
{
   public static void main(String[] args)
   {
      Scanner in = new Scanner(System.in);
      System.out.print("Floor: ");
      int floor = in.nextInt();

      // Adjust floor if necessary

      int actualFloor;
      if (floor > 13)
      {
         actualFloor = floor - 1;
      }
      else
      {
         actualFloor = floor;
      }

```

Continued

section_1/ElevatorSimulation.java

```
      System.out.println("The elevator will travel to the actual floor "
         + actualFloor);
   }
}
```

Program Run:

```
Floor: 20
The elevator will travel to the actual floor 19
```

Decision

- The Java if statement

```
if (condition)
    statement_T;
```

```
if (condition)
    statement_T;
else
    statement_F;
```

- where
  - statement is any Java statement
  - condition is a boolean expression

Conditions

- Any expression with Boolean result
  - boolean variable
    - canVote   taxable   found
  - Method with boolean result
    - exists(filename)   isSent(myEmail)
  - Operand relationalOperator Operand
    (where relationalOperator is: >, <, >=, <=, ==, !=)
    - age >= 18   speed != 0   year % 4 == 0
  - Boolean expressions combined with logicalOperator
    (where logicalOperator is: &&, ||, !)
    - *height > 2 && weight <= 80   x < 5 || x > 10*
    - *! exists(filename)   aChar >= '0' && aChar <= '9'*

Note the use of ==
as opposed to =

Relational operators also
work for char & boolean

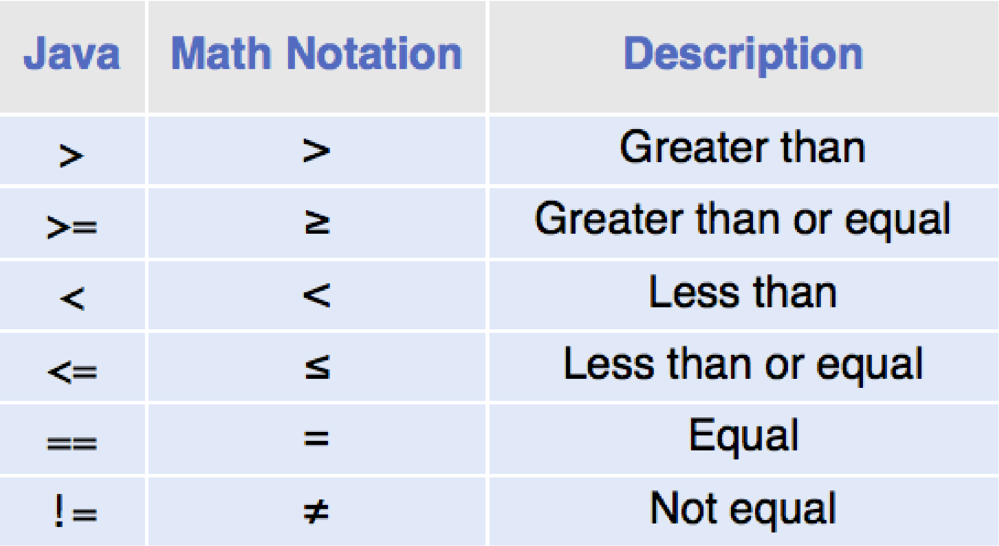
Ordering is defined by
Unicode for char

Conditions

- Note: cannot write “0 <= x < 10” must say “x >= 0 && x < 10”
- For non-primitive types, == & != will compile but may not always give the expected result!
  - For String’s use: string1.equals(string2) or string1.equalsIgnoreCase(string2)
  - For ordering use: string1.compareTo(string2) *{ neg., zero, pos. result}*
- aChar >= ‘0’ && aChar <= ‘9’
  - Tests whether aChar contains a digit or not by comparing the ASCII codes
  - Similar idea is used to test for Letters and to convert between upper & lower case
  - Only works for English
  - Use Character.isDigit(aChar); & Character.toUpperCase(aChar); etc.
- No need for the “== true” in “if (x > 0 == true)” or “if (canVote == true)”
  - “if (x > 0 == false)” or “if (canVote == false)” is equally bad
  - Rewrite as “if (x <= 0)” or “if (!canVote)”
- Comparing real numbers: if(Math.abs(real1 – real2) < epsilon)

Comparing Values: Relational Operators

- Relational operators compare values:



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

- The == denotes equality testing:

```
floor = 13; // Assign 13 to floor
if (floor == 13) // Test whether floor equals 13
```

- Relational operators have lower precedence than arithmetic operators:

```
floor - 1 < 13
```

Comparing Floating-Point Numbers

- Consider this code:

```
double r = Math.sqrt(2);
double d = r * r -2;
if (d == 0)
{
   System.out.println("sqrt(2)squared minus 2 is 0");
}
else
{
   System.out.println("sqrt(2)squared minus 2 is not 0 but " + d);
}
```

- It prints:

```
sqrt(2)squared minus 2 is not 0 but 4.440892098500626E-16
```

- This is due to round-off errors
- When comparing floating-point numbers, don't test for equality.
  - Check whether they are close enough.

Comparing Floating-Point Numbers

- To avoid roundoff errors, don't use == to compare floating-point numbers.
- To compare floating-point numbers test whether they are *close enough*: $|x - y| \le \varepsilon$

```
final double EPSILON = 1E-14;
if (Math.abs(x - y) <= EPSILON)
{
   // x is approximately equal to y
}
```

- ε is commonly set to 10^{-14}

Comparing Strings

- To test whether two strings are equal to each other, use `equals` method:

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

- Don't use `==` for strings!

```
 if (string1 == string2) // Not useful
```

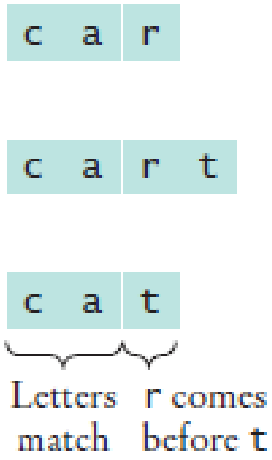
- `==` tests if two strings are stored in the same memory location
- `equals` method tests equal contents

Comparing Strings – `compareTo` Method

- `compareTo` method compares strings in lexicographic order - dictionary order.
- `string1.compareTo(string2) < 0` means:
  - `string1` comes before `string2` in the dictionary
- `string1.compareTo(string2) > 0` means:
  - `string1` comes after `string2` in the dictionary
- `string1.compareTo(string2) == 0` means:
  - `string1` and string2 are `equal`

Lexicographic Ordering

- Lexicographic Ordering



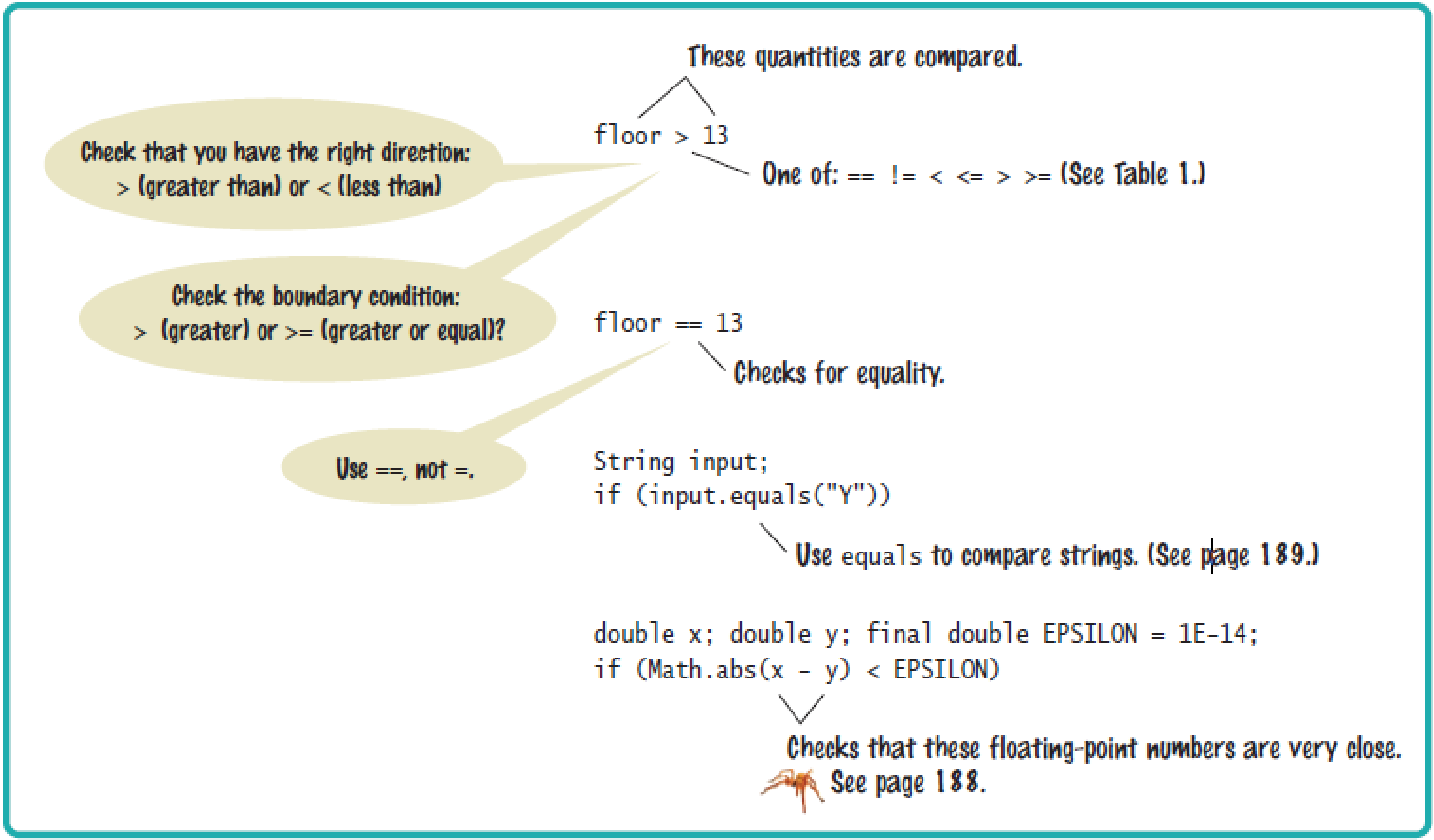
Lexicographic Ordering

Lexicographic Ordering

- Differences in dictionary ordering and ordering in Java
  - All uppercase letters come before the lowercase letters. "Z" comes before "a"
  - The space character comes before all printable characters
  - Numbers come before letters
  - Ordering of punctuation marks varies
- To see which of two terms comes first in the dictionary, consider the first letter in which they differ

Corbis Digital Stock.

Syntax 4.2 Comparisons



These quantities are compared.

```
floor > 13
```

One of: == != < <= > >= (See Table 1.)

Check that you have the right direction: > (greater than) or < (less than)

Check the boundary condition: > (greater) or >= (greater or equal)?

```
floor == 13
```

Checks for equality.

Use ==, not =.

```
String input;
if (input.equals("Y"))
```

Use `equals` to compare strings. (See page 189.)

```
double x; double y; final double EPSILON = 1E-14;
if (Math.abs(x - y) < EPSILON)
```

Checks that these floating-point numbers are very close.
See page 188.

Relational Operator Examples

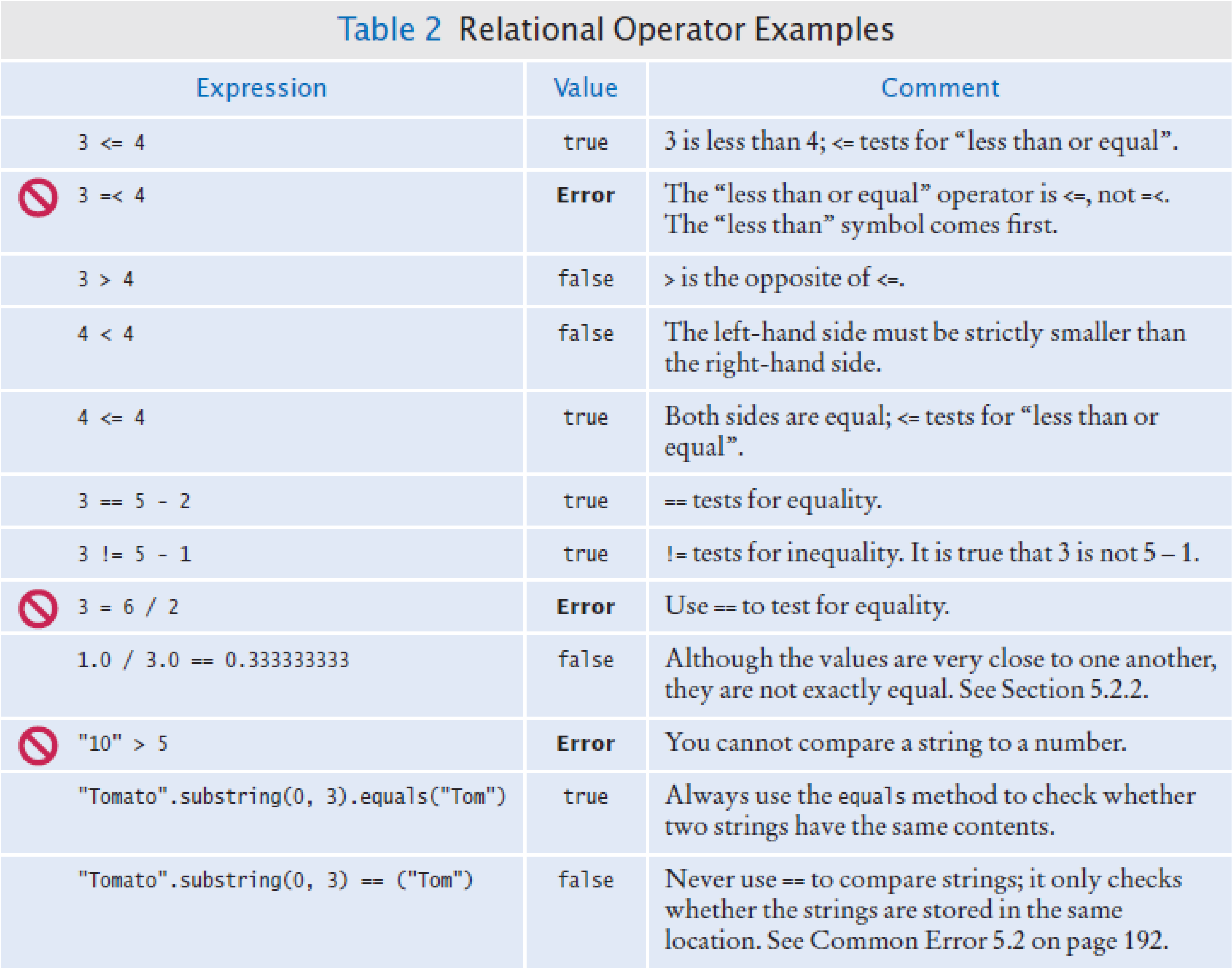


Table 2 Relational Operator Examples

Expression	Value	Comment
`3 <= 4`	`true`	3 is less than 4; <= tests for "less than or equal".
🚫 `3 =< 4`	**Error**	The "less than or equal" operator is <=, not =<. The "less than" symbol comes first.
`3 > 4`	`false`	> is the opposite of <=.
`4 < 4`	`false`	The left-hand side must be strictly smaller than the right-hand side.
`4 <= 4`	`true`	Both sides are equal; <= tests for "less than or equal".
`3 == 5 - 2`	`true`	== tests for equality.
`3 != 5 - 1`	`true`	!= tests for inequality. It is true that 3 is not 5 – 1.
🚫 `3 = 6 / 2`	**Error**	Use == to test for equality.
`1.0 / 3.0 == 0.333333333`	`false`	Although the values are very close to one another, they are not exactly equal. See Section 5.2.2.
🚫 `"10" > 5`	**Error**	You cannot compare a string to a number.
`"Tomato".substring(0, 3).equals("Tom")`	`true`	Always use the `equals` method to check whether two strings have the same contents.
`"Tomato".substring(0, 3) == ("Tom")`	`false`	Never use == to compare strings; it only checks whether the strings are stored in the same location. See Common Error 5.2 on page 192.

Self Check

Supply a condition in this `if` statement to test whether the user entered a Y:

```
System.out.println("Enter Y to quit.");
String input = in.next();
if (. . .)
{
   System.out.println("Goodbye.");
}
```

Answer: `input.equals("Y")`

Self Check

Give two ways of testing that a string `str` is the empty string.

Answer: `str.equals("")` or `str.length() == 0`

Self Check

Which of the following comparisons are syntactically incorrect? Which of them are syntactically correct, but logically questionable?

```
String a = "1”;
String b = "one”;
double x = 1;
double y = 3 * (1.0 / 3);
```

a. `a == "1"`
b. `a == null`
c. `a.equals("")`
d. `a == b`
e. `a == x`
f. `x == y`
g. `x - y == null`
h. `x.equals(y)`

Answer: Syntactically incorrect: e, g, h. Logically questionable: a, d, f.

Avoid Duplication in Branches

- If you have duplicate code in each branch, move it out of the `if` statement.
- Don't do this

```
if (floor > 13)
{
   actualFloor = floor - 1;
   System.out.println("Actual floor: " + actualFloor);
}
else
{
   actualFloor = floor;
   System.out.println("Actual floor: " + actualFloor);
}
```

Avoid Duplication in Branches

- Do this instead

```
if (floor > 13)
{
   actualFloor = floor - 1;
}
else
{
   actualFloor = floor;
}
System.out.println("Actual floor: " + actualFloor);
```

- It will make the code much easier to maintain.
- Changes will only need to be made in one place.

Examples (1)

- Print message when x is positive

  -
    ```
    if ( x > 0 )
        System.out.println( “The value of x is positive”);
    ```
- Print warning if oil pressure is below a specified limit

  -
    ```
    if ( oilPressure < MINIMIUM_OIL_PRESSURE )
        System.out.println( “Warning – low !”);
    ```
- Report whether x is negative or not

  -
    ```
    if ( x < 0 )
        System.out.println( “Negative”);
    else
        System.out.println( “Positive or zero”);
    ```
- Rewrite with alternative condition
- Check user’s password

  -
    ```
    if ( !actualPassword.equals( enteredPassword) )
        System.out.println( “Sorry, incorrect Password”);
    else
        // do secure things!
    ```

Examples (2)

- Compute z as absolute value of x-y

  -
    ```
    if ( x - y < 0 )
        z = y - x;
    else
        z = x - y;
    ```

  -
    ```
    if ( x > y )
        z = x - y;
    else
        z = y - x;
    ```

  -
    ```
    z = x - y;
    if ( z < 0 )
        z = -z;
    ```

Can also use

z = Math.abs(x-y);

Multiple Alternatives: Sequences of Comparisons

- Multiple `if` statements can be combined to evaluate complex decisions.
- You use multiple `if` statements to implement multiple alternatives.

Multiple Alternatives: Sequences of Comparisons

- Example: damage done by earthquake of a given magnitude on the Richter scale:

```
if (richter >= 8.0)
{
   description = "Most structures fall”;
}
else if (richter >= 7.0)
{
   description = "Many buildings destroyed”;
}
else if (richter >= 6.0)
{
   description = "Many buildings considerably damaged, some collapse”;
}
else if (richter >= 4.5)
{
   description = "Damage to poorly constructed buildings”;
}
else
{
   description = "No destruction of buildings”;
}
```

Multiple Alternatives: Sequences of Comparisons

- As soon as one of the four tests succeeds:
  - The effect is displayed
  - No further tests are attempted.
- If none of the four cases applies
  - The final `else` clause applies
  - A default message is printed.

Multiple Alternatives - Flowchart

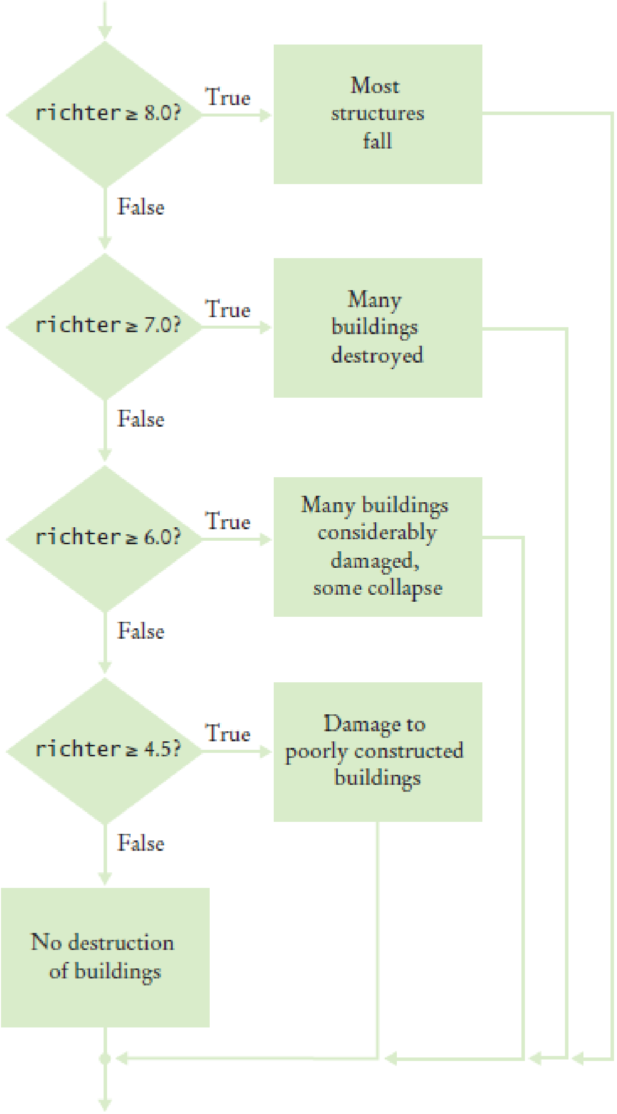


Figure 4
Multiple Alternatives

Multiple Alternatives

- The order of the `if` and `else if` matters
- Error

```
if (richter >= 4.5) // Tests in wrong order
{
   description = "Damage to poorly constructed buildings”;
}
else if (richter >= 6.0)
{
   description = "Many buildings considerably damaged, some collapse”;
}
else if (richter >= 7.0)
{
   description = "Many buildings destroyed”;
}
else if (richter >= 8.0)
{
   description = "Most structures fall”;
}
```

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

Examples (3)

- Given three values stored in variables first, second, third, store the minimum in a variable called min

    -
      ```
      if ( first < second )
          min = first;
      else
          min = second;

      if ( third < min )
          min = third;
      ```
- Generalise…?

Examples (3)

- Begin with simplest case, that of two variables, then work up!
- Could also compare first & second, then first & third, & second & third.

```
If ( first < second && first < third)
        min is first
else if (third < first && third < second)
        min is third
else if ( second < first && second < third)
        min is second
```

Nested Branches

- **Nested** set of statements:
  - An `if` statement inside another
- Example: Federal Income Tax
  - Tax depends on marital status and income

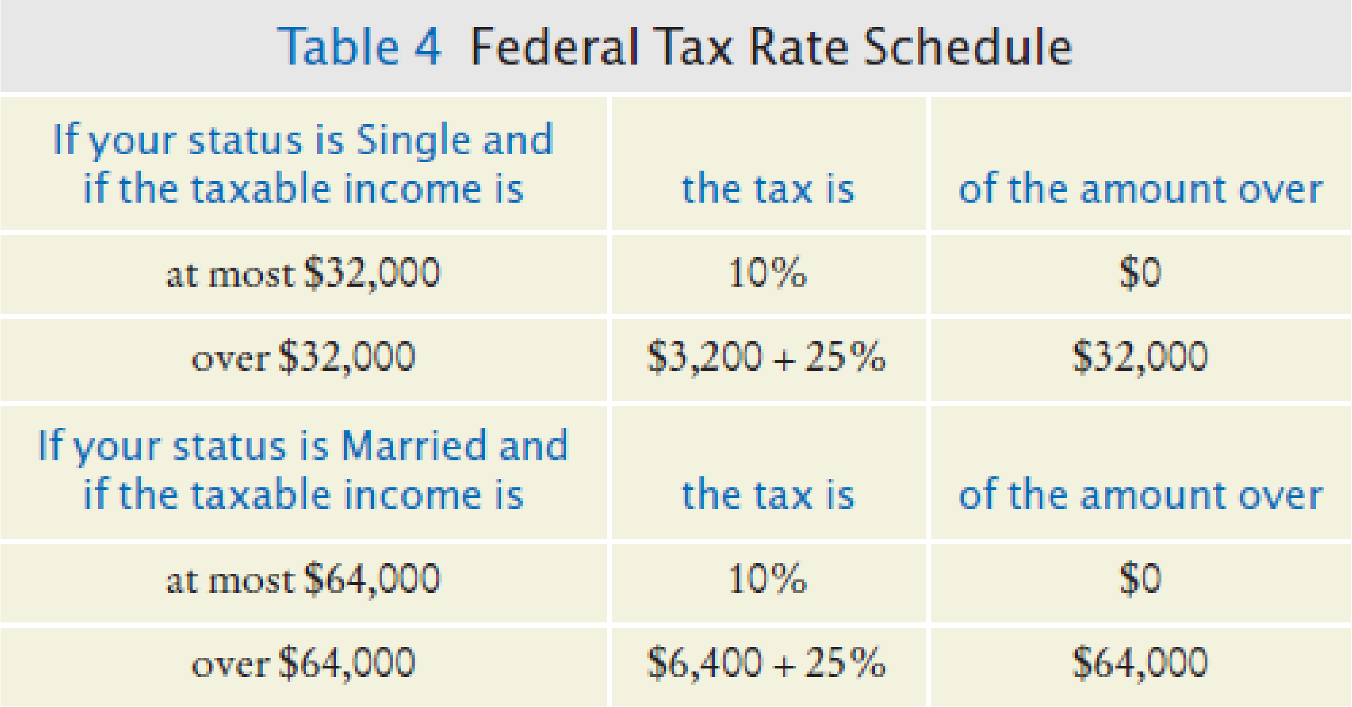


Table 4 Federal Tax Rate Schedule

If your status is Single and if the taxable income is	the tax is	of the amount over
at most $32,000	10%	$0
over $32,000	$3,200 + 25%	$32,000
If your status is Married and if the taxable income is	the tax is	of the amount over
at most $64,000	10%	$0
over $64,000	$6,400 + 25%	$64,000

© ericsphotography/iStockphoto.

Nested Branches

- We say that the income test is *nested* inside the test for filing status
- Two-level decision process is reflected in two levels of if statements in the program
- Computing income taxes requires multiple levels of decisions.

Nested Branches - Flowchart

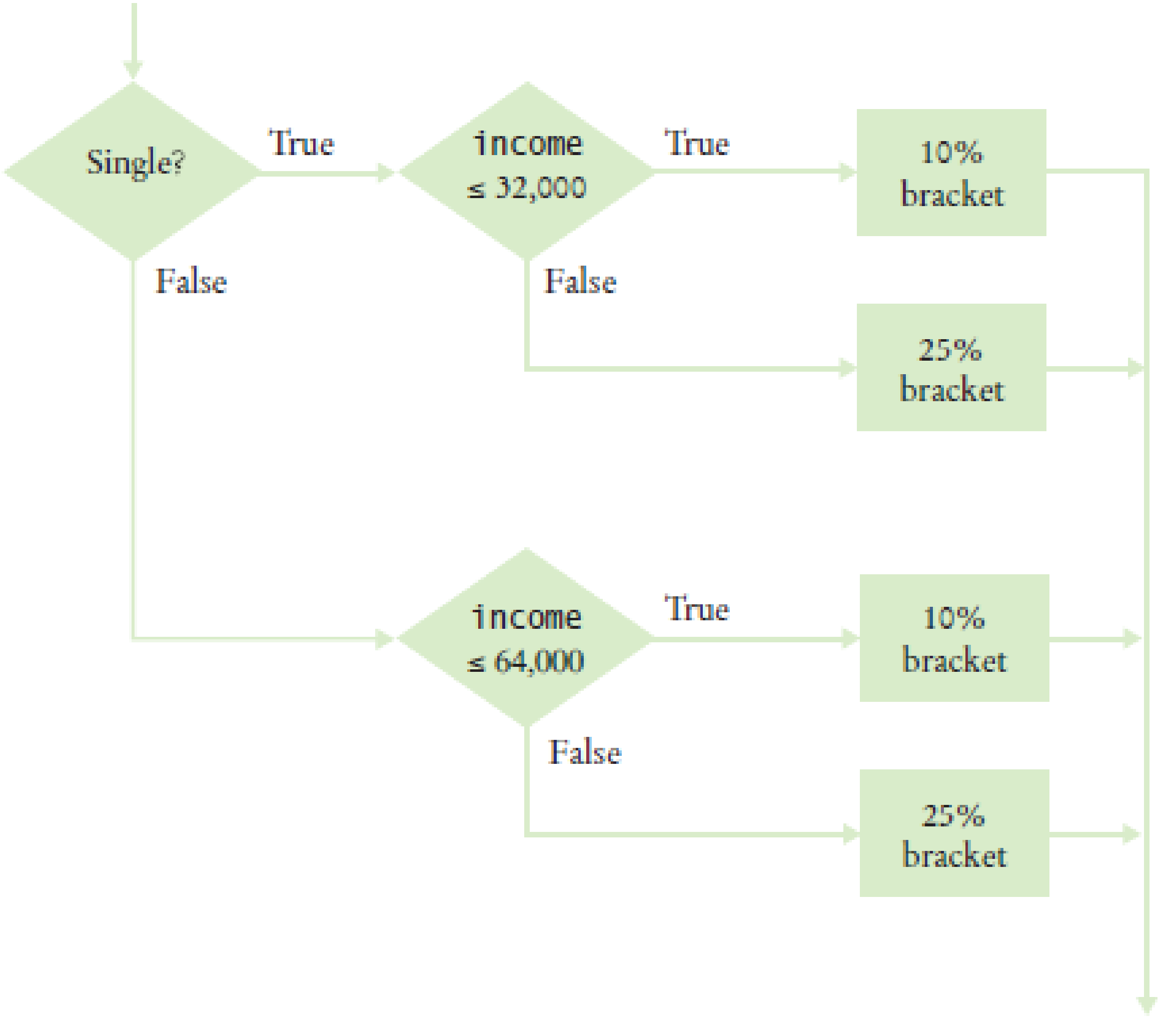


Figure 5 Income Tax Computation

section_4/TaxReturn.java

```java
/**
   A tax return of a taxpayer in 2008.
*/
public class TaxReturn
{
   public static final int SINGLE = 1;
   public static final int MARRIED = 2;

   private static final double RATE1 = 0.10;
   private static final double RATE2 = 0.25;
   private static final double RATE1_SINGLE_LIMIT = 32000;
   private static final double RATE1_MARRIED_LIMIT = 64000;

   private double income;
   private int status;

   /**
      Constructs a TaxReturn object for a given income and
      marital status.
      @param anIncome the taxpayer income
      @param aStatus either SINGLE or MARRIED
   */
   public TaxReturn(double anIncome, int aStatus)
   {
      income = anIncome;
      status = aStatus;
   }

```

Continued

section_4/TaxReturn.java

```
   public double getTax()
   {
      double tax1 = 0;
      double tax2 = 0;

      if (status == SINGLE)
      {
         if (income <= RATE1_SINGLE_LIMIT)
         {
            tax1 = RATE1 * income;
         }
         else
         {
            tax1 = RATE1 * RATE1_SINGLE_LIMIT;
            tax2 = RATE2 * (income - RATE1_SINGLE_LIMIT);
         }
      }
      else
      {
         if (income <= RATE1_MARRIED_LIMIT)
         {
            tax1 = RATE1 * income;
         }
         else
         {
            tax1 = RATE1 * RATE1_MARRIED_LIMIT;
            tax2 = RATE2 * (income - RATE1_MARRIED_LIMIT);
         }
      }

      return tax1 + tax2;
   }
}
```

section_4/TaxCalculator.java

```java
import java.util.Scanner;

/**
   This program calculates a simple tax return.
*/
public class TaxCalculator
{
   public static void main(String[] args)
   {
      Scanner in = new Scanner(System.in);

      System.out.print("Please enter your income: ");
      double income = in.nextDouble();

      System.out.print("Are you married? (Y/N) ");
      String input = in.next();
      int status;
      if (input.equals("Y"))
      {
         status = TaxReturn.MARRIED;
      }
      else
      {
         status = TaxReturn.SINGLE;
      }

      TaxReturn aTaxReturn = new TaxReturn(income, status);

      System.out.println("Tax: "
            + aTaxReturn.getTax());
   }
}
```

Continued

section_4/TaxCalculator.java

Program Run

```
Please enter your income: 80000
Are you married? (Y/N) Y
Tax: 10400.0
```

Self Check

How would you modify the `TaxCalculator.java` program in order to check that the user entered a correct value for the marital status (i.e., `Y` or `N`)?

Answer: Change else in line 22 to

```
else if (maritalStatus.equals("N"))
```

and add another branch after line 25:

```
else
{
   System.out.println( "Error: Please answer Y or N.");
}
```

Examples (4)

- Avoid divide-by-zero errors

  -
    ```
    if ( y = 0 )
          System.out.println( “Error: can’t divide by zero”);
    else
          z = x / y;
          System.out.println( “The result is “ + z );
    ```
- Use braces (curly brackets) to form compound statement

```
{

  statement;
  statement;
      ⋮
  statement;
}
```

≡

```
statement;
```

Examples (5)

- Choosing between three alternatives:

  -
    ```
    if ( x < 0 )
         System.out.println( “Negative”);
    else {
         if ( x == 0 )
             System.out.println( “Zero”);
         else
             System.out.println( “Positive”);
    }
    ```

  -
    ```
    if ( x >= 0 ) {
         if ( x == 0 )
            System.out.println( “Zero”);
         else
            System.out.println( “Positive”);
    }
    else
         System.out.println( “Negative”);
    ```

Examples (6)

- A neater way of writing mutually exclusive alternatives (nested if):

```
if ( x < 0 )
     System.out.println( “Negative”);
else if ( x == 0 )     // & x >= 0
     System.out.println( “Zero”);
else if ( x < 5)       // & x >= 0 & x != 0
     System.out.println( “1 - 4 inclusive”);
else                   System.out.println( “>= 5”);

                       // & x >= 0 & x != 0 & x >= 5
```

Distinguish…

```
if (cond)
     print “A”
else if (cond)
     print “B”
else if (cond)
     print “C”
else
     print “D”
```

```
if (cond)
     print “A”
if (cond)
     print “B”
if (cond)
     print “C”
if (cond)
     print “D”
```

Boolean Variables and Operators

- To store the evaluation of a logical condition that can be true or false, you use a Boolean variable.
- The boolean data type has exactly two values, denoted false and true.

```
boolean failed = true;
```

- Later in your program, use the value to make a decision

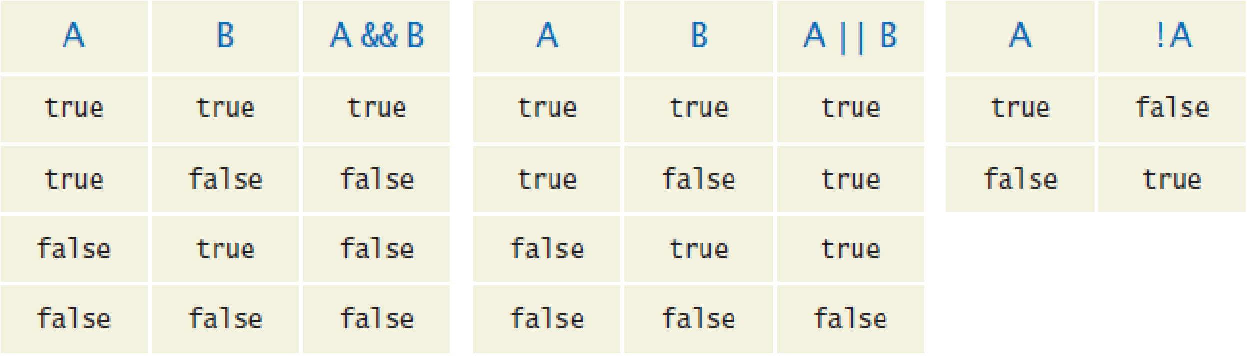
```
if (failed) // Only executed if failed has been set to true
{ . . . }
```

- A Boolean variable is also called a flag because it can be either up (true) or down (false).

Cusp/SuperStock.

Boolean Variables and Operators

- You often need to combine Boolean values when making complex decisions
- An operator that combines Boolean conditions is called a Boolean operator.
- The && operator is called **and**
    - Yields `true` only when both conditions are `true`.
- The || operator is called **or**
    - Yields the result `true` if at least one of the conditions is `true`.



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

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

A	!A
true	false
false	true

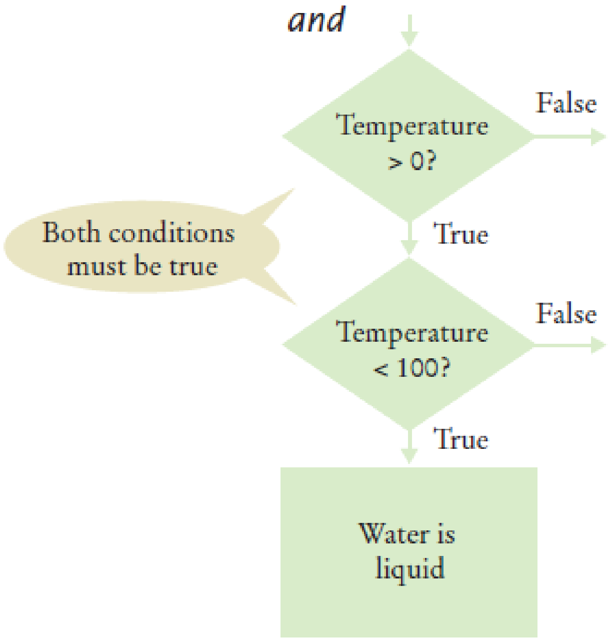
Figure 9 Boolean Truth Tables

Boolean Variables and Operators

- To test if water is liquid at a given temperature

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

- Flowchart

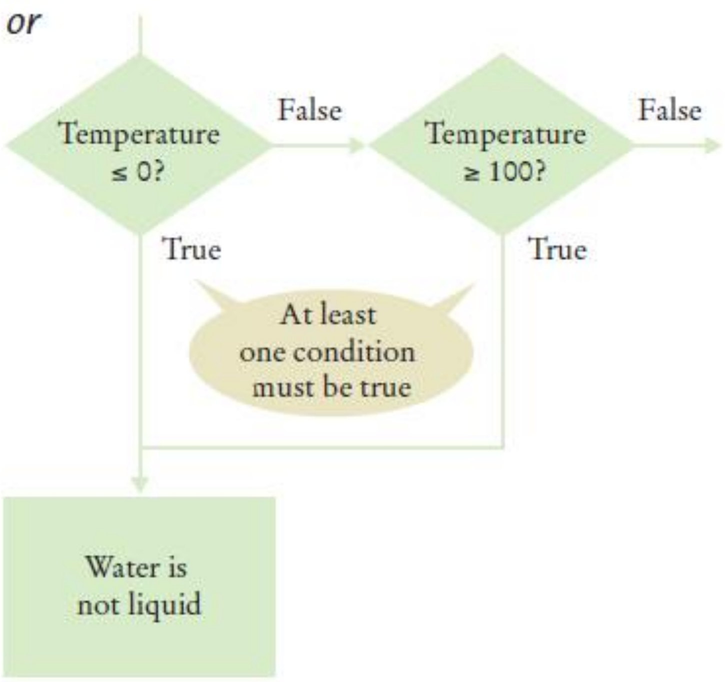


Boolean Variables and Operators

- To test if water is **not** liquid at a given temperature

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

- Flowchart



Boolean Variables and Operators

- To *invert* a condition use the *not* Boolean operator
- The `!` operator takes a single condition
  - Evaluates to `true` if that condition is `false` and
  - Evaluates to `false` if the condition is `true`
- To test if the Boolean variable `frozen` is `false`:

```
if (!frozen) { System.out.println("Not frozen"); }
```

Self Check 4.33

Suppose x and y are two integers. How do you test whether both of them are zero?

Answer: `x == 0 && y == 0`

Self Check 4.34

How do you test whether at least one of them is zero?

Answer: `x == 0 || y == 0`

Self Check 4.35

How do you test whether exactly one of them is zero?

Answer:

```
(x == 0 && y != 0) || (y == 0 && x != 0)
```

Self Check 4.36

What is the value of `!!frozen`?

Answer: The same as the value of `frozen`.

Application: Input Validation

- You need to make sure that the user-supplied values are valid before you use them.
- Elevator example: elevator panel has buttons labeled 1 through 20 (but not 13)
- The number 13 is invalid

```
if (floor == 13)
{
   System.out.println("Error: There is no thirteenth floor.");
}
```

- Numbers out of the range 1 through 20 are invalid

```
if (floor <= 0 || floor > 20)
{
   System.out.println("Error: The floor must be between 1 and 20.");
}
```

Application: Input Validation

- To avoid input that is not an integer

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

Section_8/ElevatorSimulation2.java

```java
import java.util.Scanner;

/**
   This program simulates an elevator panel that skips the 13th floor, checking for
   input errors.
*/
public class ElevatorSimulation2
{
   public static void main(String[] args)
   {
      Scanner in = new Scanner(System.in);
      System.out.print("Floor: ");
      if (in.hasNextInt())
      {
         // Now we know that the user entered an integer

         int floor = in.nextInt();

```

Continued

Section_8/ElevatorSimulation2.java

```java
         if (floor == 13)
         {
            System.out.println("Error: There is no thirteenth floor.");
         }
         else if (floor <= 0 || floor > 20)
         {
            System.out.println("Error: The floor must be between 1 and 20.");
         }
         else
         {
            // Now we know that the input is valid

            int actualFloor = floor;
            if (floor > 13)
            {
               actualFloor = floor - 1;
            }

            System.out.println("The elevator will travel to the actual floor "
               + actualFloor);
         }
      }
      else
      {
         System.out.println("Error: Not an integer.");
      }
   }
}
```

Continued

Section_8/ElevatorSimulation2.java

Program Run

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

Self Check 4.39

Your task is to rewrite lines 19–26 of the ElevatorSimulation2 program so that there is a single if statement with a complex condition. What is the condition?

```
if (. . .)
{
   System.out.println("Error: Invalid floor number");
 }
```

Answer:

```
floor == 13 || floor <= 0 || floor > 20
```