Software Engineering1
(Java)

CSY1019
(Week 4)

The Class `String`

- We've used constants of type `String` already.

  `"Enter a whole number from 1 to 99."`
- A value of type `String` is a
  - Sequence of characters
  - Treated as a single item.

String Constants and Variables

- Declaring

```
String greeting;
 greeting = "Hello!";
```

or

```
String greeting = "Hello!";
```

or

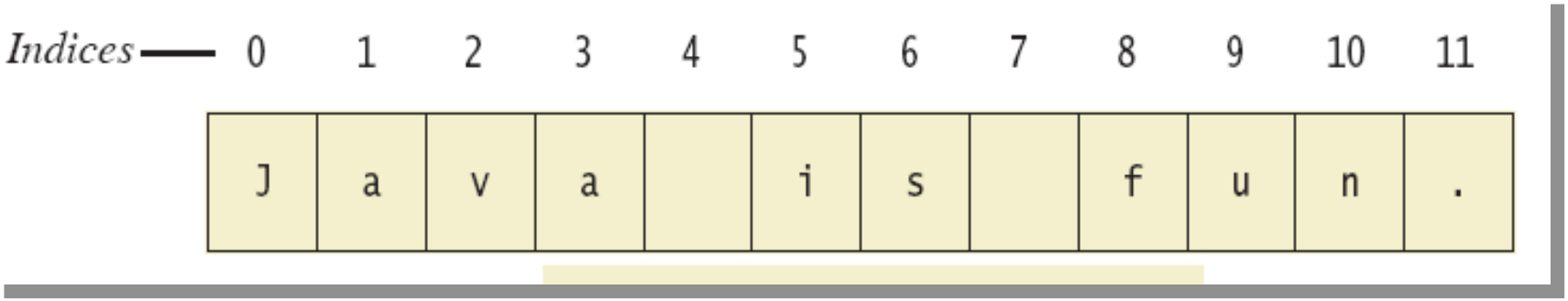
```
String greeting = new String("Hello!");
```

- Printing

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

Example: StringDemo.java

String Indices



- Positions start with 0, not 1.
  - The 'J' in "Java is fun." is in position 0
- A position is referred to an an *index.*
  - The `'f'` in `"Java is fun."` is at index 8.

`String` Methods

`charAt(`*Index*`)`
Returns the character at *Index* in this string. Index numbers begin at 0.

`compareTo(`*A_String*`)`
Compares this string with *A_String* to see which string comes first in the lexicographic ordering. (Lexicographic ordering is the same as alphabetical ordering when both strings are either all uppercase letters or all lowercase letters.) Returns a negative integer if this string is first, returns zero if the two strings are equal, and returns a positive integer if *A_String* is first.

`concat(`*A_String*`)`
Returns a new string having the same characters as this string concatenated with the characters in *A_String*. You can use the ⇓ operator instead of `concat`.

`equals(`*Other_String*`)`
Returns true if this string and *Other_String* are equal. Otherwise, returns false.

`String` Methods

`equalsIgnoreCase(`*Other_String*`)` Behaves like the method `equals`, but considers uppercase and lowercase versions of a letter to be the same.
`indexOf(`*A_String*`)` Returns the index of the first occurrence of the substring *A_String* within this string. Returns -1 if *A_String* is not found. Index numbers begin at 0.
`lastIndexOf(`*A_String*`)` Returns the index of the last occurrence of the substring *A_String* within this string. Returns -1 if *A_String* is not found. Index numbers begin at 0.

`String` Methods

`length()` Returns the length of this string.
`toLowerCase()` Returns a new string having the same characters as this string, but with any uppercase letters converted to lowercase.
`toUpperCase()` Returns a new string having the same characters as this string, but with any lowercase letters converted to uppercase.

`String` Methods

`replace(`*OldChar*, *NewChar*`)` Returns a new string having the same characters as this string, but with each occurrence of *OldChar* replaced by *NewChar*.
`substring(`*Start*`)` Returns a new string having the same characters as the substring that begins at index *Start* of this string through to the end of the string. Index numbers begin at 0.
`substring(`*Start*, *End*`)` Returns a new string having the same characters as the substring that begins at index *Start* of this string through, but not including, index *End* of the string. Index numbers begin at 0.
`trim()` Returns a new string having the same characters as this string, but with leading and trailing whitespace removed.

The Empty String

- A string can have any number of characters, including zero.
- The string with zero characters is called the *empty* string.
- The empty string is useful and can be created in many ways including

```
String s3 = "";
```

The String class

```
String greeting = “Good Morning”;
System.out.println(greeting);
```

Output: Good Morning

Refer Java DocAPI for many methods in class library

```
int length = greeting.length();//length = 12
String lowercase = greeting.toLowerCase();
```

Example: StringLength.java, String Methods.java

String Concatenation

- The ‘+’ operator can be used to concatenate one or more strings

```
String message = “Hello”;
String name = “David James!”;
System.out.println(message + name);
```

Output: Hello David James!

Reading Character in Java

To read a character:

- Read input as a string
- Get the first character of the string

```
String answer = scan.next(); //read a string
char ch = answer.charAt(0); //retrieve first character
System.out.println(ch); // prints character
```

Example: ReadCharacter.java

Parse Methods: to convert Strings to Numbers

```
// Store 1 in bVar.
byte bVar = Byte.parseByte("1");

// Store 2599 in iVar.
int iVar = Integer.parseInt("2599");

// Store 10 in sVar.
short sVar = Short.parseShort("10");

// Store 15908 in lVar.
long lVar = Long.parseLong("15908");

// Store 12.3 in fVar.
float fVar = Float.parseFloat("12.3");

// Store 7945.6 in dVar.
double dVar = Double.parseDouble("7945.6");
```

Dialog Boxes

- A *dialog box* is a small graphical window that displays a message to the user or requests input.
- A variety of dialog boxes can be displayed using the `JOptionPane` class.
- Two of the dialog boxes are:
  - Message Dialog - a dialog box that displays a message.
  - Input Dialog - a dialog box that prompts the user for input.

The `JOptionPane` Class

- The `JOptionPane` class is not automatically available to your Java programs.
- The following statement must be before the program's class header:

```
import javax.swing.JOptionPane;
```

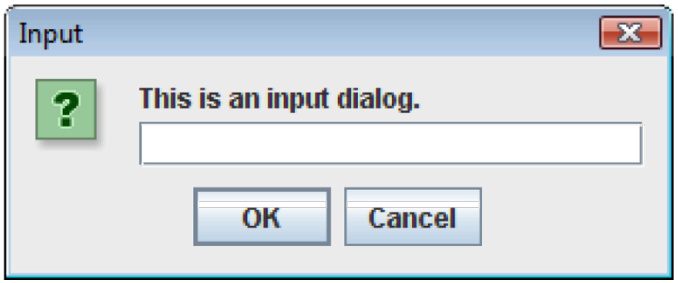
- This statement tells the compiler where to find the `JOptionPane` class.

The `JOptionPane` Class

The `JOptionPane` class provides methods to display each type of dialog box.

Message dialog

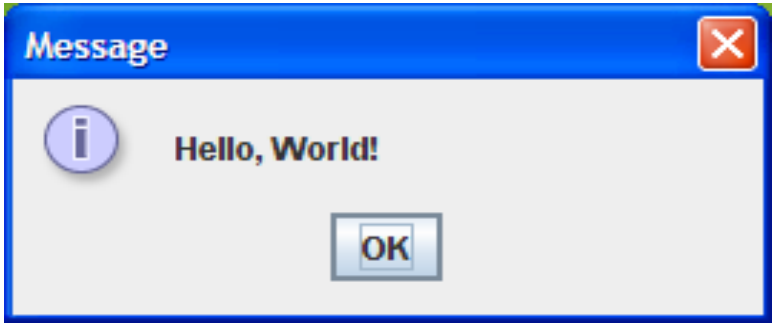
Input dialog



Reading and Printing data using Dialog Boxes

String name = JOptionPane.showInputDialog("What's your name?") ;

JOptionPane.showMessageDialog(null, "Hello, World!") ;



Reading an Integer with an Input Dialog

```
int number;
String str;
str = JOptionPane.showInputDialog("Enter a number.");
number = Integer.parseInt(str);
```

Reading a Double with an Input Dialog

```
double number;
String str;
str = JOptionPane.showInputDialog("Enter a number.");
number = Double.parseDouble(str);
```

Similarly, for float, long, short, byte

Example Program 1 (using JOptionPane)

```
//*******************************************************************
//  GasMileageDialog.java       Java Foundations
//
//  Demonstrates the use of the JOptionPane to read and print numeric data.
//*******************************************************************


public class GasMileageDialog
{
   //-----------------------------------------------------------------
  /  Calculates fuel efficiency based on values entered by the
  //  user.
  //-----------------------------------------------------------------
  public static void main (String[] args)
   {
    double miles, gallons, mpg;
    String input1 = JOptionPane.showInputDialog("Enter the number of miles: ");
    miles = Double.parseDouble(input1);


(more…)
```

Example Program 1 (continued)

```
 String input2 = JOptionPane.showInputDialog("Enter the gallons of fuel used: ");
 gallons = Double.parseDouble(input2);
 mpg = miles / gallons;
 JOptionPane.showMessageDialog(null, “Miles per Gallon: ” + mpg) ;
}
}
```

The `DecimalFormat` Class

- When printing out `double` and `float` values, the full fractional value will be printed.
- The `DecimalFormat` class can be used to format these values.
- In order to use the `DecimalFormat` class, the following `import` statement must be used at the top of the program:

```
import java.text.DecimalFormat;
```

- See examples:

  Format1.java, Format2.java, Format3.java, Format4.java

The `if` Statement

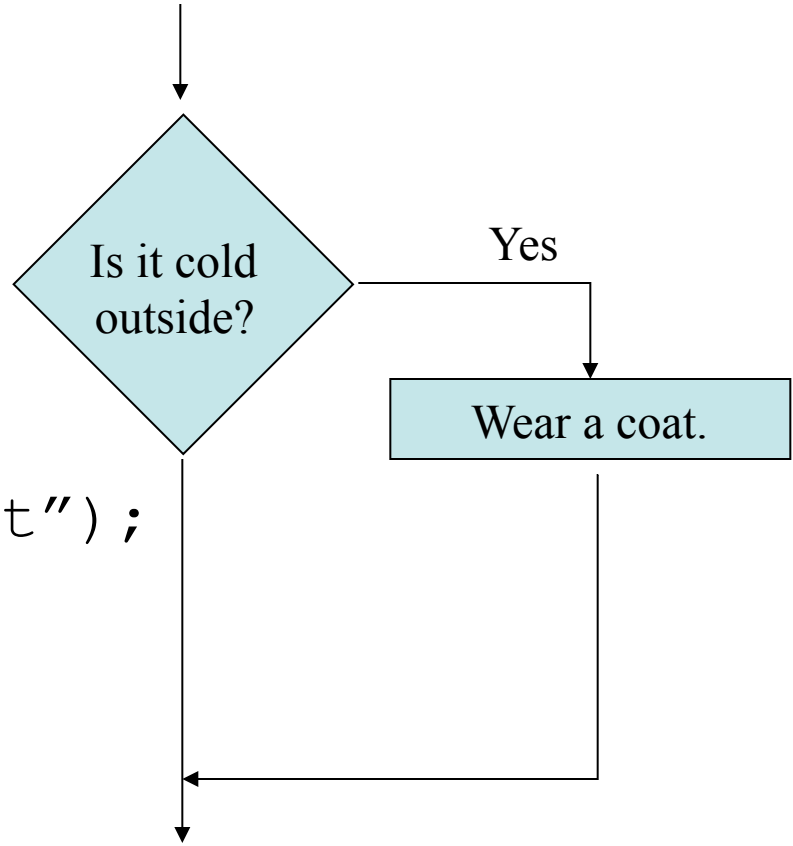
- The `if` statement decides whether a section of code executes or not.
- The `if` statement uses a `boolean` to decide whether the next statement or block of statements executes.

if (boolean expression is true)

execute next statement.

Flowcharts

- If statements can be modeled as a flow chart.

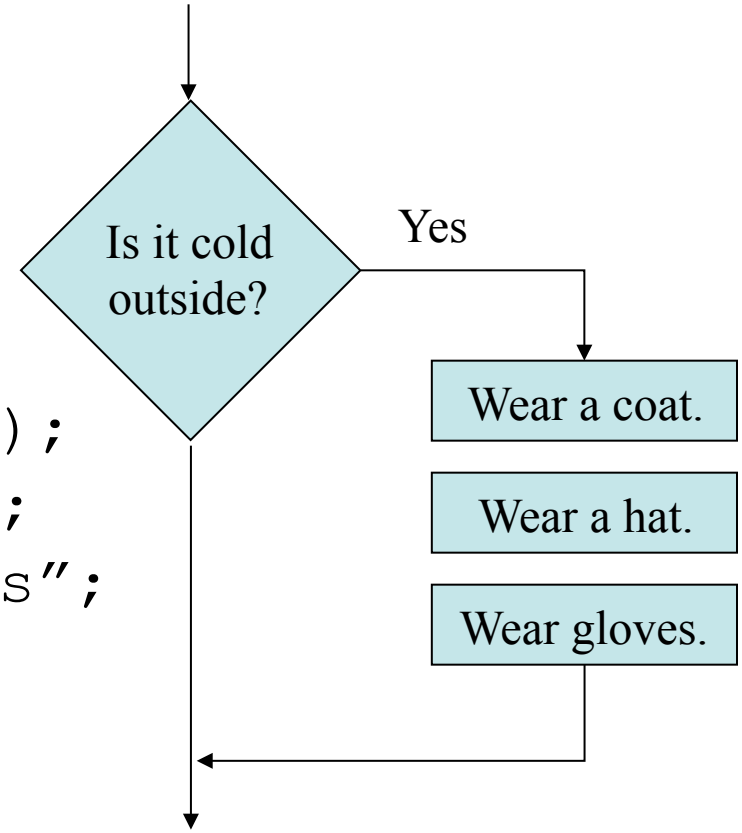



```
if (coldOutside)
System.out.println("Wear Coat");
```

Flowcharts

- A block `if` statement may be modeled as:

```
if (coldOutside)
{
System.out.println("Wear Coat");
System.out.println("Wear Hat");
System.out.println("Wear Gloves";
}
```



Note the use of curly braces to block several statements together.

Relational Operators

Relational Operator	Meaning
>	is greater than
<	is less than
>=	is greater than or equal to
<=	is less than or equal to
==	is equal to
!=	is not equal to

Boolean Expressions

- A *boolean expression* is any variable or calculation that results in a *true* or *false* condition.

Expression	Meaning
`x > y`	Is x greater than y?
`x < y`	Is x less than y?
`x >= y`	Is x greater than or equal to y?
`x <= y`	Is x less than or equal to y.
`x == y`	Is x equal to y?
`x != y`	Is x not equal to y?

Comparing Numbers and Strings

- Every `if` statement has a condition

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

Beware!

ompares two values with an operator

Table 1 Relational Operators

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

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

Calculations

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

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

Comparing Strings

- Strings are a bit ‘special’ in Java
- 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)) ...
```

`if` Statements and Boolean Expressions

```
if (x > y)
   System.out.println("X is greater than Y");

if (x == y)
   System.out.println("X is equal to Y");

if (x != y)
{
   System.out.println("X is not equal to Y");
   x = y;
   System.out.println("However, now it is.");
}
```

See Example: AverageScore.java

Programming Style and `if` Statements

- An `if` statement can span more than one line; however, it is still one statement.

```
if (average > 95)
   grade = 'A';
```

is functionally equivalent to

```
If (average > 95) grade = 'A';
```

Programming Style and `if` Statements

- Rules of thumb:
  - The conditionally executed statement should be on the line after the `if` condition.
  - The conditionally executed statement should be indented one level from the `if` condition.
  - If an `if` statement does not have the block curly braces, it is ended by the first semicolon encountered after the `if` condition.

```
if (expression)
  statement;
```

No semicolon here.

Semicolon ends statement here.

Block `if` Statements

- Conditionally executed statements can be grouped into a block by using curly braces **{ }** to enclose them.
- If curly braces are used to group conditionally executed statements, the `if` statement is ended by the closing curly brace.

```
if (expression)
{
  statement1;
  statement2;
}
```

← **Curly brace ends the statement.**

Block `if` Statements

- Remember that when the curly braces are not used, then only the next statement after the `if` condition will be executed conditionally.

```
if (expression)
  statement1;
  statement2;
  statement3;
```

Only this statement is conditionally executed.

Common Error

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

Flags

- A flag is a `boolean` variable that monitors some condition in a program.
- When a condition is true, the flag is set to `true`.
- The flag can be tested to see if the condition has changed.

```
boolean highScore = false;
 if (average > 95)
   highScore = true;
```

- Later, this condition can be tested:

```
 if (highScore)
   System.out.println("That's a high score!");
```

See example: **BooleanFlag.java**

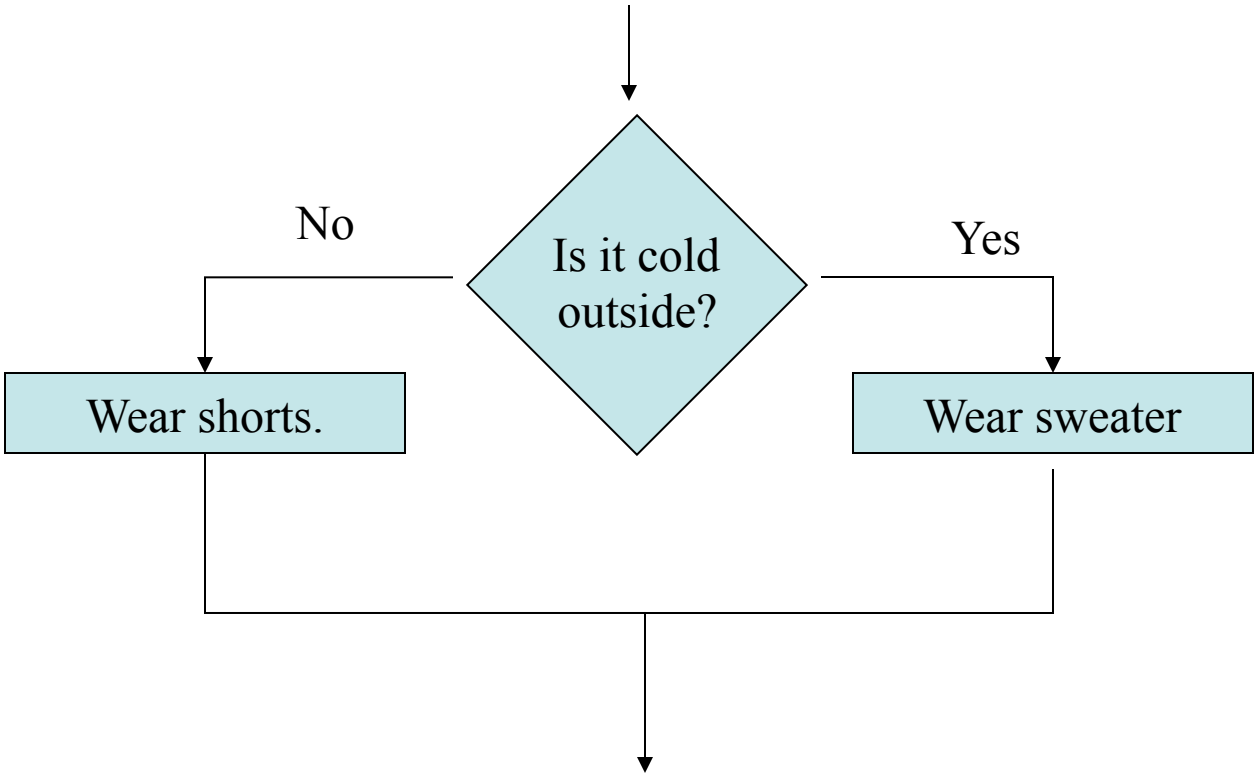
`if-else` Statements

- The `if-else` statement adds the ability to conditionally execute code when the `if` condition is false.

```
if (expression)
  statementOrBlockIfTrue;
else
  statementOrBlockIfFalse;
```

- See example: Division.java

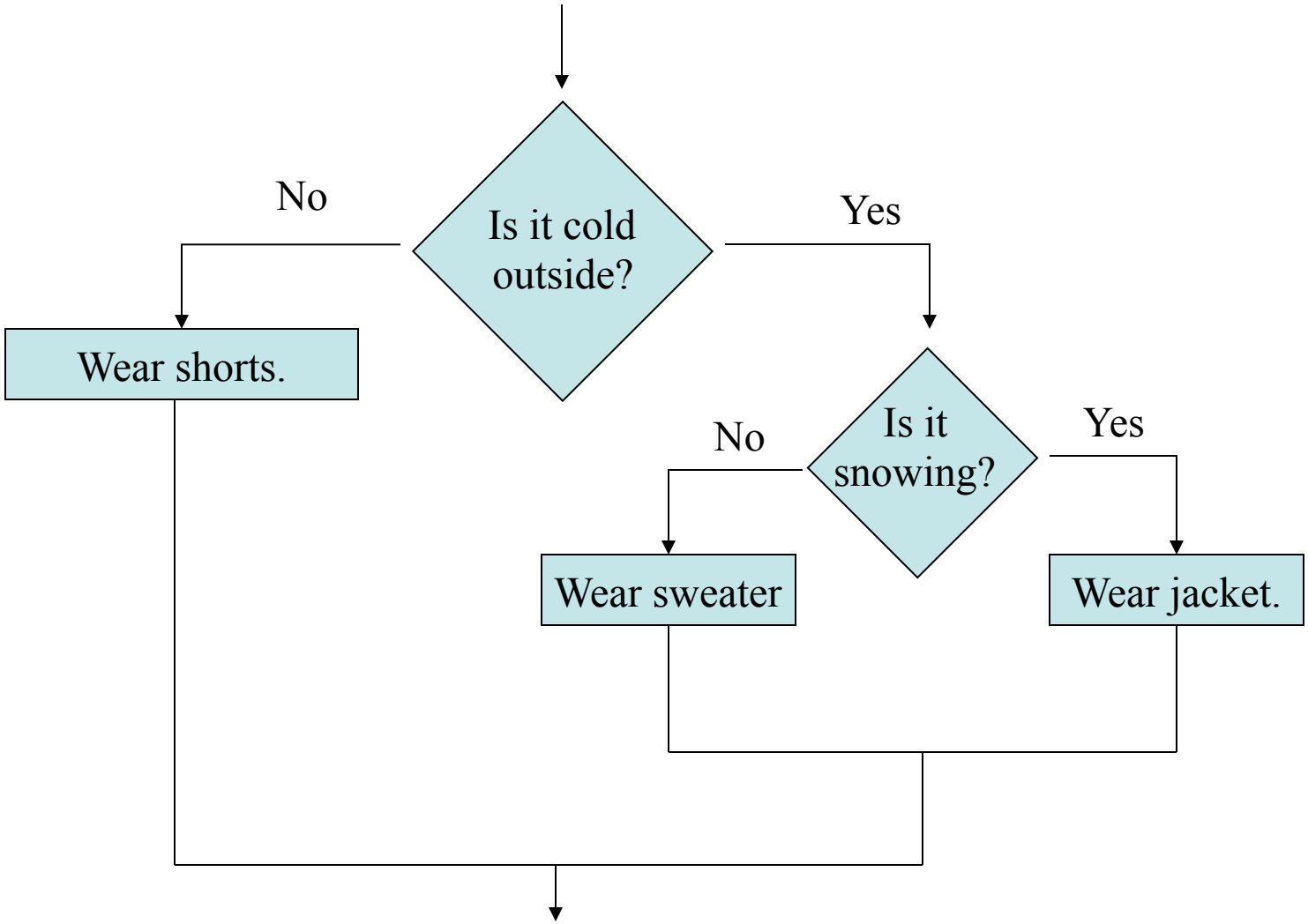
`if-else` Statement Flowcharts



Nested `if` Statements

- If an `if` statement appears inside another `if` statement (single or block) it is called a *nested `if`* statement.
- The nested `if` is executed only if the outer `if` statement results in a true condition.
- See example: LoanQualifier.java

Nested `if` Statement Flowcharts



Nested `if` Statements

```
if (coldOutside)
{
     if (snowing)
     {
          System.out.println("Wear Jacket");
     }
     else
     {
          System.out.println("Wear Sweater");
     }
}
else
{
   System.out.println("Wear Shorts");
}
```

`if-else` Matching

- Curly brace use is not required if there is only one statement to be conditionally executed.
- However, sometimes curly braces can help make the program more readable.
- Additionally, proper indentation makes it much easier to match up else statements with their corresponding `if` statement.

`if-else-if` Statements

```
if (expression_1)
{
   statement;
   statement;
   etc.
}
else if (expression_2)
{
   statement;
   statement;
   etc.
}
```

If `expression_1` is true these statements are executed, and the rest of the structure is ignored.

Otherwise, if `expression_2` is true these statements are executed, and the rest of the structure is ignored.

Insert as many else if clauses as necessary

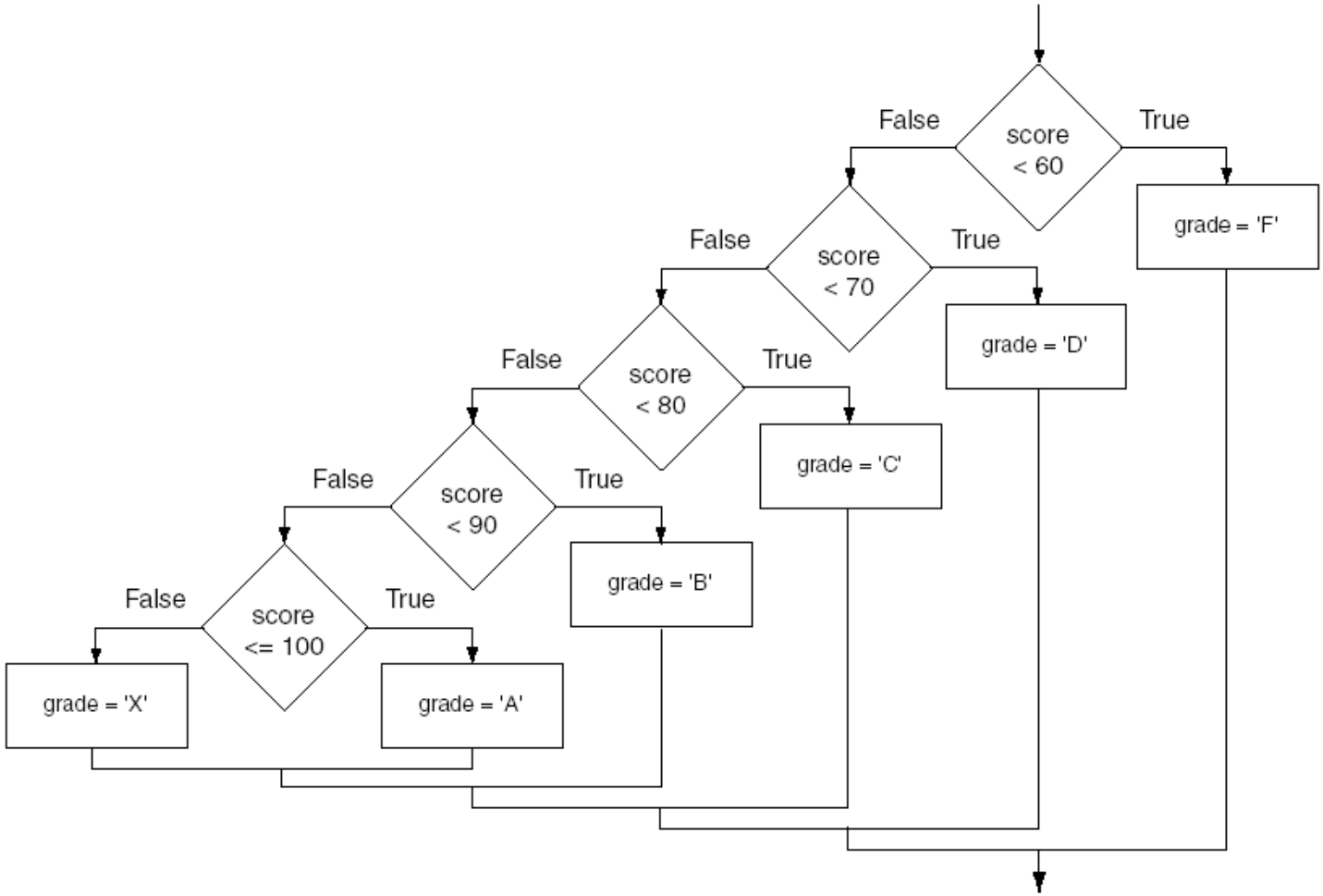
```
else
{
   statement;
   statement;
   etc.
}
```

These statements are executed if none of the expressions above are true.

`if-else-if` Statements

- Nested `if` statements can become very complex.
- The `if-else-if` statement makes certain types of nested decision logic simpler to write.
- Care must be used since `else` statements match up with the immediately preceding unmatched `if` statement.
- See example: TestResults.java

`if-else-if` Flowchart



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");
}
```

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");
}
```

Common Error

The Dangling `else` Problem

- When an `if` statement is nested inside another `if`

```
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`

The Conditional Operator

- The *conditional operator* is a ternary (three operand) operator.
- You can use the conditional operator to write a simple statement that works like an `if-else` statement.

The Conditional Operator

- The format of the operators is:

```
BooleanExpression ? Value1 : Value2
```

- This forms a conditional expression.
- If *`BooleanExpression`* is true, the value of the conditional expression is *`Value1`*.
- If *`BooleanExpression`* is false, the value of the conditional expression is *`Value2`*.

The Conditional Operator

- Example:

```
z = x > y ? 10 : 5;
```

- This line is functionally equivalent to:

```
if(x > y)
  z = 10;
else
  z = 5;
```

- See example: ConsultantCharges.java

Logical Operators

Operator	Meaning	Effect
&&	**AND**	Connects two `boolean` expressions into one. Both expressions must be true for the overall expression to be true.
||	**OR**	Connects two `boolean` expressions into one. One or both expressions must be true for the overall expression to be true. It is only necessary for one to be true, and it does not matter which one.
!	**NOT**	The ! operator reverses the truth of a `boolean` expression. If it is applied to an expression that is true, the operator returns false. If it is applied to an expression that is false, the operator returns true.

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");
}
```

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

- If either is true
  - The result is true

The *not* Operator: !

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

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

A	!A
true	false
false	true

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

- If using !, try to use simpler logic:

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

The `switch` Statement

- The `if-else` statement allows you to make true / false branches.
- The `switch` statement allows you to use an ordinal value to determine how a program will branch.
- The `switch` statement can evaluate an *integer* type ,*character* type, or String (Java 7) variable and make decisions based on the value.

The `switch` Statement

- The `switch` statement takes the form:

```
switch (SwitchExpression)
{
  case CaseExpression:
    // place one or more statements here
    break;
  case CaseExpression:
    // place one or more statements here
    break;

    // case statements may be repeated
    //as many times as necessary
  default:
    // place one or more statements here
}
```

The `switch` Statement

- The `switch` statement takes an ordinal value (`byte`, `short`, `int`, `long`, `char or String`) as the *SwitchExpression*.

```
switch (SwitchExpression)
{
  …
}
```

- The `switch` statement will evaluate the expression.
- If there is an associated `case` statement that matches that value, program execution will be transferred to that `case` statement.

The `switch` Statement

- Each `case` statement will have a corresponding *CaseExpression* that must be unique.

```
case CaseExpression:
    // place one or more statements here
    break;
```

- If the *SwitchExpression* matches the *CaseExpression*, the Java statements between the colon and the `break` statement will be executed.

The `case` Statement

- The `break` statement ends the `case` statement.
- The `break` statement is optional.
- If a `case` does not contain a `break`, then program execution continues into the next `case`.
  - See example: NoBreaks.java
  - See example: PetFood.java
- The `default` section is optional and will be executed if no *CaseExpression* matches the *SwitchExpression*.
- See example: SwitchDemo.java