Decision Structures

Lecture 3a

Topics (1 of 2)

- The `if` Statement
- The `if-else` Statement
- Nested `if` statements
- The `if-else-if` Statement
- Logical Operators
- Comparing `String` Objects

Topics (2 of 2)

- More about Variable Declaration and Scope
- The Conditional Operator
- The `switch` Statement
- Displaying Formatted Output with `System.out.printf` and `String.format`

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.

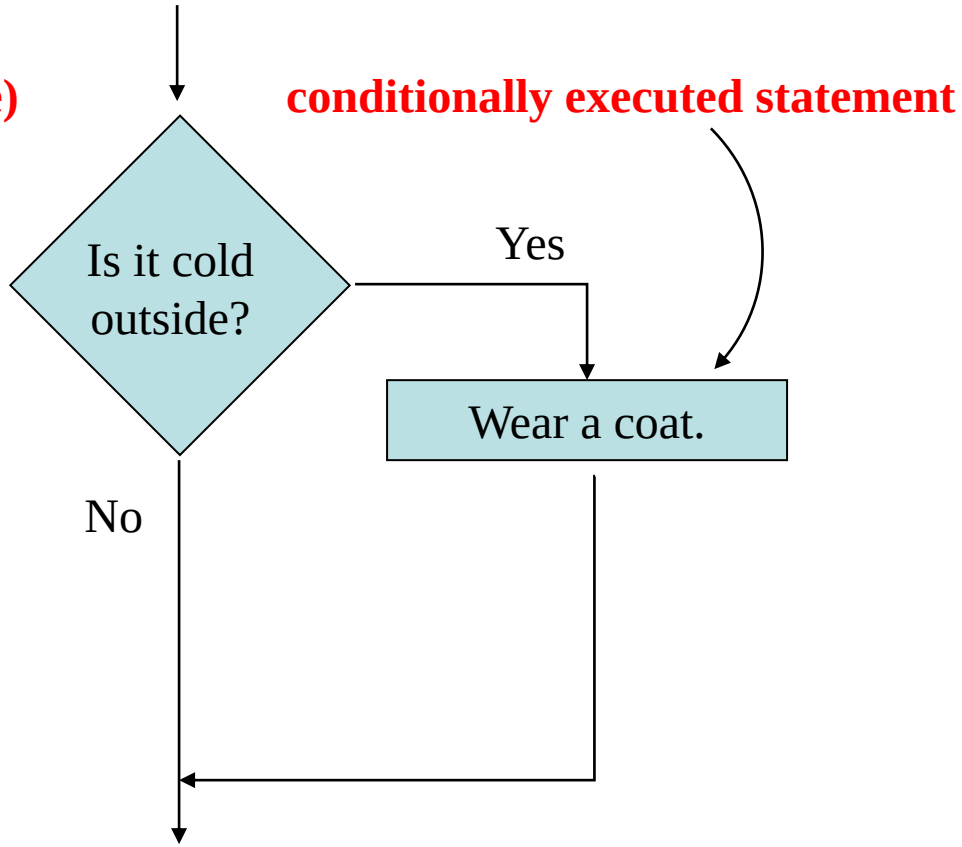
if (boolean expression is true)
execute next statement.

Flowcharts (1 of 2)

- If statements can be modeled as a flow chart.

Boolean Expression (True or False)

```
if (coldOutside)
    wearCoat();
```

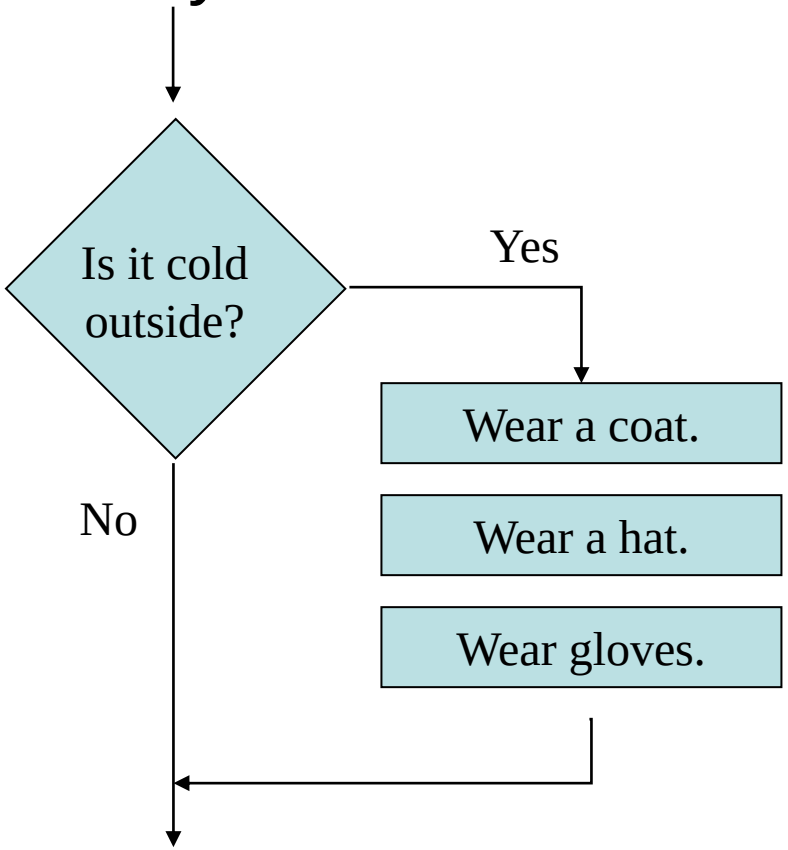


Flowcharts (2 of 2)

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

```
if (coldOutside)
{
   wearCoat();
   wearHat();
   wearGloves();
}
```

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



Relational Operators

- Typically, the `boolean` expression, used by the `if` statement, uses *relational operators (they are all binary).*

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

Relational Operators

- Point to ponder #1

  Why is it called relational operator?

  It determines whether specific relationship(s) exist(s) between 2 values.

- 2 > 3 false
- 5 >= 1 true
- 7 == 5 false
- 8 != 5 true

Relational Operators

- Point to ponder #2

  What is the difference between the expressions below?

  `x = 2 and x == 2?`

  The first is using an assignment operator to assign 2 to the variable x (assignment expression). The second one is using an equality operator to compare 2 with the value of x (boolean expression).

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?

`if` Statements and Boolean Expressions

```
int x = 4, y = 3;
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.");
}
```

e.g.: AverageScore.java (Convert the code to use the Scanner class)

Programming Style and `if` Statements (1 of 3)

- 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 (2 of 3)

- 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.

Programming Style and `if` Statements (3 of 3)

Be careful to not prematurely terminate an if statement!

```
int x = 4, y = 3;
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.");
}
```

Block `if` Statements (1 of 3)

- 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 (2 of 3)

- 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.

Block `if` Statements (3 of 3)

- Point to ponder #3

  Why is the output of this code?

```
int x = 4, y = 3;
if(x == y)
   x = y;
   System.out.println("The value of x is: " + x);
```

The value of x is: 4

```
int x = 4, y = 3;
if(x != y)
   x = y;
   System.out.println("The value of x is: " + x);
```

The value of x is: 3

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.

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

- Later, this condition can be tested:

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

Flags

- For instance, checking whether 5 is prime or not.

```
boolean isPrime = true;
if (5 % 4 == 0)
   isPrime = false;
if (5 % 3 == 0)
   isPrime = false;
if (5 % 2 == 0)
   isPrime = false;

if (isPrime)
   System.out.println("5 is prime");
System.out.println("Finished!");
```

Comparing Characters (1 of 2)

- Characters can be tested with relational operators.
- Characters are stored in memory using the Unicode character format.
- Unicode is stored as a sixteen (16) bit number.
- Characters are *ordinal*, meaning they have an order in the Unicode character set.
- Since characters are ordinal, they can be compared to each other.

Comparing Characters (1 of 2)

- ```
  char x = 'A';
  if(x < 'C')
    System.out.println("A is less than C");
  ```

  A is less than C

- ```
  x = 'Z';
  if(x < 'a')
    System.out.println("Z is less than a");
  ```

  Z is less than a

`if-else` Statements

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

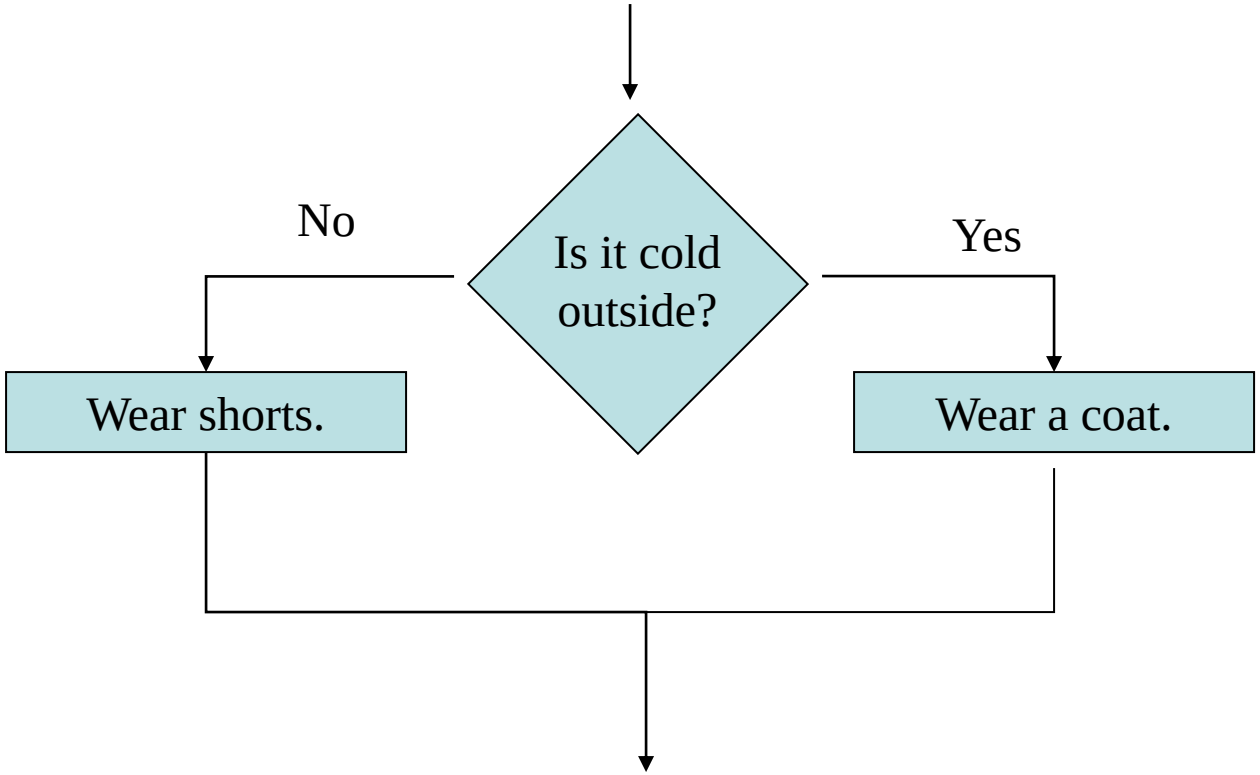
```
if (expression)
   statementOrBlockIfTrue;
else
   statementOrBlockIfFalse;
```

`if-else` Statements

See example: `Division.java`

```
// Get the first number.
 System.out.print("Enter a number: ");
 number1 = keyboard.nextDouble();

 // Get the second number.
 System.out.print("Enter another number: ");
 number2 = keyboard.nextDouble();

 if (number2 == 0)
 {
   System.out.println("Division by zero is not possible.");
   System.out.println("Please run the program again and ");
   System.out.println("enter a number other than zero.");
 }
 else
 {
   quotient = number1 / number2;
   System.out.print("The quotient of " + number1);
   System.out.print(" divided by " + number2);
   System.out.println(" is " + quotient);
 }
```

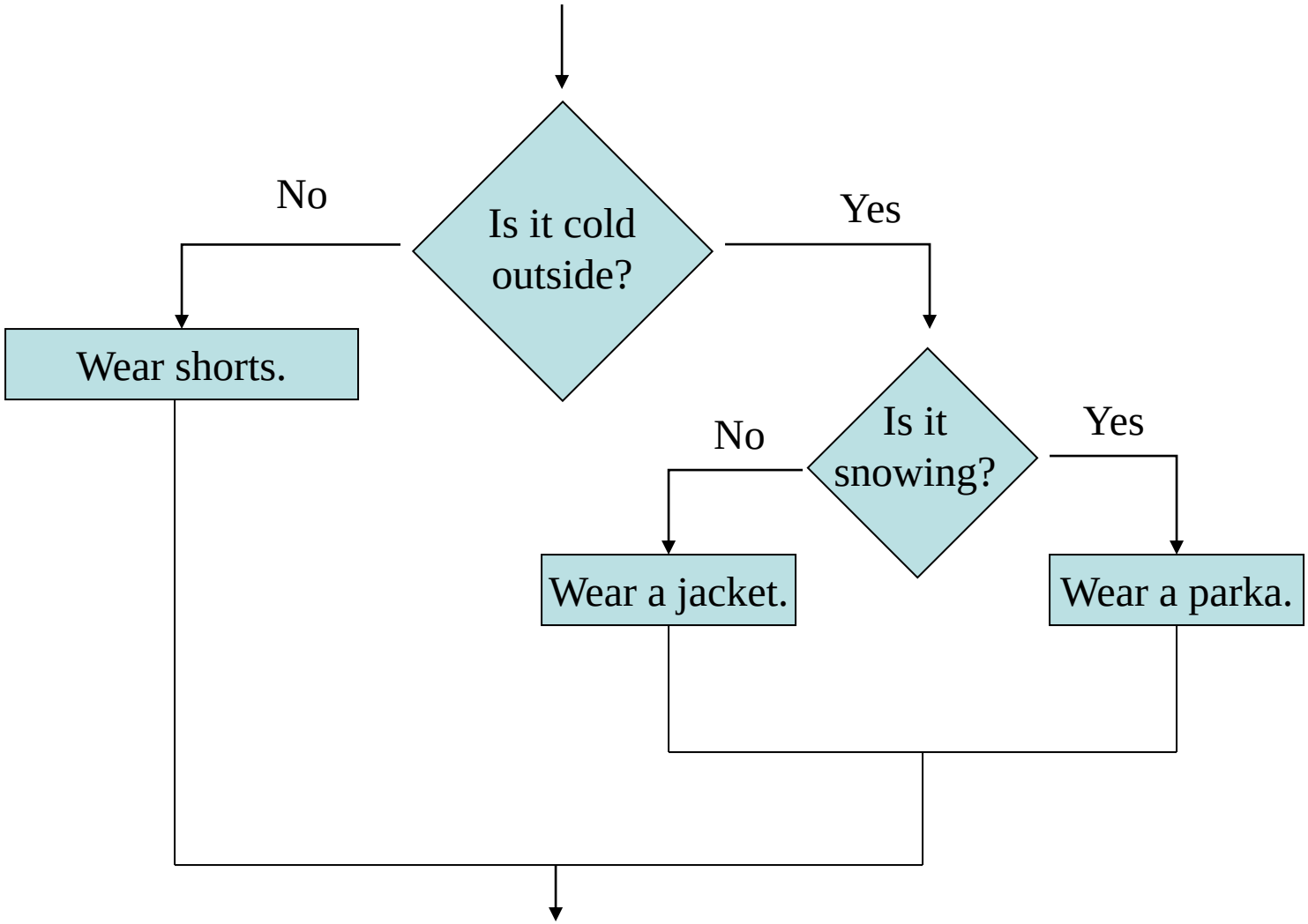
`if-else` Statement Flowcharts



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)
     {
          wearParka();
     }
     else
     {
          wearJacket();
     }
}
else
{
     wearShorts();
}
```

`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.

Alignment and Nested `if` Statements (1 of 2)

This `if` and `else` go together.

This `if` and `else` go together.

```
if (coldOutside)
{
    if (snowing)
    {
        wearParka();
    }
    else
    {
        wearJacket();
    }
}
else
{
    wearShorts();
}
```

Alignment and Nested `if` Statements (2 of 2)

This `if` and `else` go together.

This `if` and `else` go together.

```
if (coldOutside)
{
if (snowing)
{
wearParka();
}
else
{
wearJacket();
}
}
else
{
    wearShorts();
}
```

Indentation makes the difference!!!

`if-else-if` Statements (1 of 3)

- 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.

`if-else-if` Statements (2 of 3)

```
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 (3 of 3)

```
// Displaying the grade by using nested if-else statements.
if (testScore < 80)
{
  System.out.println("Your grade is C.");
}
else
{
  if (testScore < 90)
  {
    System.out.println("Your grade is B.");
  }
  else
  {
    System.out.println("Your grade is A.");
  }
}

// Displaying the grade by using if-else-if statements.
if (testScore < 80)
  System.out.println("Your grade is C.");
else if (testScore < 90)
  System.out.println("Your grade is B.");
else if (testScore <= 100)
  System.out.println("Your grade is A.");
```

Both produce the same results, but the second decision structure is easier to debug and understand!

- See example: TestResults.java

Logical Operators (1 of 3)

- Java provides two binary *logical operators* (`&&` and `||`) that are used to combine `boolean` expressions into a single one.
- Java also provides one *unary* (`!`) logical operator to reverse the truth of a `boolean` expression.

Logical Operators (2 of 3)

Operator	Meaning	Effect
&&	**(logical) AND**	Connects two `boolean` expressions into one. Both expressions must be true for the overall expression to be true.
||	**(logical) 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.
!	**(logical) 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.

Logical Operators (3 of 3)

Expression	Meaning
`x > y && a < b`	Is x greater than y AND is a less than b?
`x == y || x == z`	Is x equal to y OR is x equal to z?
`!(x > y)`	Is the expression x > y NOT true?

The `&&` Operator (1 of 2)

- The logical AND operator (`&&`) takes two operands that must both be `boolean` expressions.
- The resulting combined expression is true if (and *only* if) both operands are true.
- See example: LogicalAnd.java

Expression 1	Expression 2	Expression1 && Expression2
true	false	false
false	true	false
false	false	false
true	true	true

The && Operator (2 of 2)

```
int temperature = 10;
if (temperature < 20 && temperature > 12)
{
   System.out.println("The temperature is not good.");
}

int temperature = 30;
if (temperature < 20 && temperature > 12)
{
   System.out.println(" The temperature is not good.");
}

int temperature = 15;
if (temperature < 20 && temperature > 12)
{
   System.out.println(" The temperature is not good.");
}
```

The || Operator (1 of 2)

- The logical OR operator (`||`) takes two operands that must both be `boolean` expressions.
- The resulting combined expression is false if (and *only* if) both operands are false.
- Example: LogicalOr.java

Expression 1	Expression 2	Expression1 || Expression2
true	**false**	**true**
false	**true**	**true**
false	**false**	**false**
true	**true**	**true**

The || Operator (2 of 2)

```
int temperature = 10;
if (temperature < 20 || temperature > 12)
{
   System.out.println("The temperature is not good.");
}

int temperature = 30;
if (temperature < 20 || temperature > 12)
{
   System.out.println(" The temperature is not good.");
}

int temperature = 15;
if (temperature < 20 || temperature > 12)
{
   System.out.println(" The temperature is not good.");
}
```

The ! Operator

- The `!` operator performs a logical NOT operation.
- If an *`expression`* is true, `!`*`expression`* will be false.

```
if (!(temperature > 100))
   System.out.println("Below the maximum temperature.");
```

- If `temperature > 100` evaluates to false, then the output statement will be run.

Expression 1	!Expression1
true	**false**
false	**true**

Short Circuiting (1 of 2)

- Logical AND and logical OR operations perform *short-circuit evaluation* of expressions.
- Logical AND will evaluate to false as soon as it sees that one of its operands is a false expression.
- Logical OR will evaluate to true as soon as it sees that one of its operands is a true expression.

Short Circuiting (2 of 2)

- Point to ponder #4

  Short-circuit? Explain this concept for AND and OR operations.

  If the expression on the left side of the && operator is false, the expression on the right side will not be checked (false output). If the expression on the left side of the || operator is true, the expression on the right side will not be checked (true output). Java does not waste CPU time!

Decision Structures

Lecture 3b

Topics (1 of 2)

- The `if` Statement
- The `if-else` Statement
- Nested `if` statements
- The `if-else-if` Statement
- Logical Operators
- Comparing `String` Objects

Topics (2 of 2)

- More about Variable Declaration and Scope
- The Conditional Operator
- The `switch` Statement
- Displaying Formatted Output with `System.out.printf` and `String.format`

Order of Precedence (1 of 5)

- The `!` operator has a higher order of precedence than the `&&` and `||` operators.
- The `&&` and `||` operators have a lower precedence than relational operators like `<` and `>`.
- Parenthesis can be used to force the precedence to be changed.

Order of Precedence (2 of 5)

Order of Precedence	Operators	Description
1	**(unary negation) !**	Unary negation, logical NOT
2	`* / %`	Multiplication, Division, Modulus
3	`+ -`	Addition, Subtraction
4	`< > <= >=`	Less-than, Greater-than, Less-than or equal to, Greater-than or equal to
5	`== !=`	Is equal to, Is not equal to
6	`&&`	Logical AND
7	`||`	Logical NOT
8	`= += -=` `*= /= %=`	Assignment and combined assignment operators.

Order of Precedence (3 of 5)

- Point to ponder #1

  What is the output of those expressions?

```
int a = 5, b = 3, x = 7, y = 4;
if (a > b && x < y) {
   System.out.println("yes");
} else {
   System.out.println("no");
}
```

no

```
int a = 5, b = 3, x = 7, y = 4;
if (a < b || x == y) {
   System.out.println("yes");
} else {
   System.out.println("no");
}
```

no

Order of Precedence (4 of 5)

```
int a = 5, b = 3, x = 7, y = 4;
if (a > b && x < y) {
   System.out.println("yes");
} else {
   System.out.println("no");
}
```

≡

```
int a = 5, b = 3, x = 7, y = 4;
if ((a > b) && (x < y)) {
   System.out.println("yes");
} else {
   System.out.println("no");
}
```

```
int a = 5, b = 3, x = 7, y = 4;
if (a < b || x == y) {
   System.out.println("yes");
} else {
   System.out.println("no");
}
```

```
int a = 5, b = 3, x = 7, y = 4;
if ((a < b) || (x == y)) {
   System.out.println("yes");
} else {
   System.out.println("no");
}
```

Order of Precedence (5 of 5)

- Point to ponder #2

  How can we test if the value of the integer variable `x` is in the range [10,15]?

```
If (x >= 10 && x <= 15) {
    …
}
```

  How about testing if `x` is outside the range [10,15]?

```
If (x < 10 || x > 15) {
    …
}
```

≡

```
If (!(x >= 10 && x <= 15)) {
    …
}
```

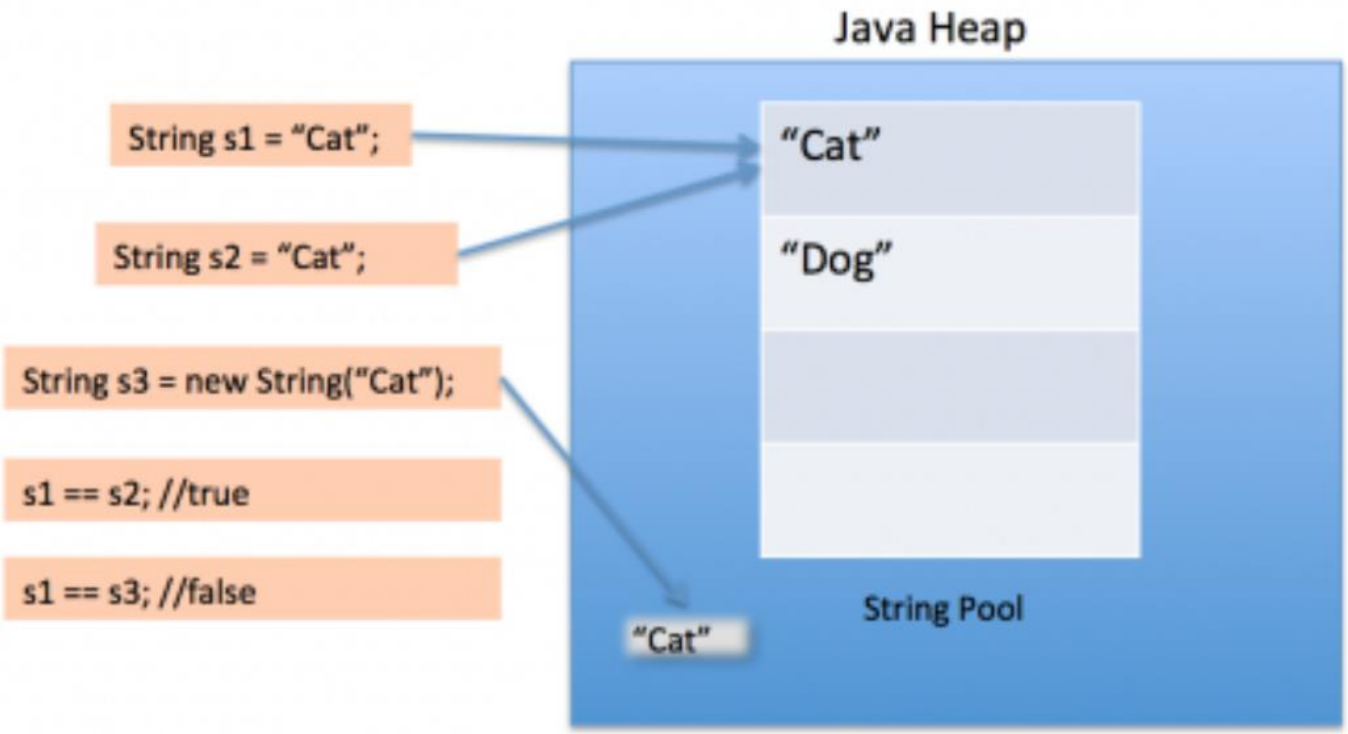
Comparing `String` Objects (1 of 9)

- In most cases, you cannot use the relational operators to compare two `String` objects.
- Reference variables contain the address of the object they represent.
- Unless the references point to the same object, the relational operators will not return true.

Comparing `String` Objects (2 of 9)

```
String name1 = "John", name2 = "John";
String name3 = new String("John");
String name4 = new String("John");

System.out.println(name1==name2);
true
System.out.println(name2==name3);
false
System.out.println(name3==name4);
false

System.out.println("@" + Integer.toHexString(System.identityHashCode(name1)));
@7d6f77cc
System.out.println("@" + Integer.toHexString(System.identityHashCode(name2)));
@7d6f77cc
System.out.println("@" + Integer.toHexString(System.identityHashCode(name3)));
@5aaa6d82
System.out.println("@" + Integer.toHexString(System.identityHashCode(name4)));
@73a28541
```

Comparing `String` Objects (3 of 9)



Comparing `String` Objects (4 of 9)

- To compare the contents of two String objects correctly, you should use the String class's equals method.

```
String name1 = "John", name2 = "John";
String name3 = new String("John");
String name4 = new String("John");

System.out.println(name1.equals(name2));
true
System.out.println(name2.equals(name3));
true
System.out.println(name3.equals(name4));
true
```

- See example: [StringCompare.java]

Comparing `String` Objects (5 of 9)

- Point to ponder #3

  What is the output of the expression below?

```
String name1 = "JOHN";
String name2 = "john".toUpperCase();
System.out.println(name1==name2);
```

false

```
String name1 = "JOHN";
String name2 = "john".toUpperCase();
System.out.println(name1.equals(name2));
```

true

Comparing `String` Objects (6 of 9)

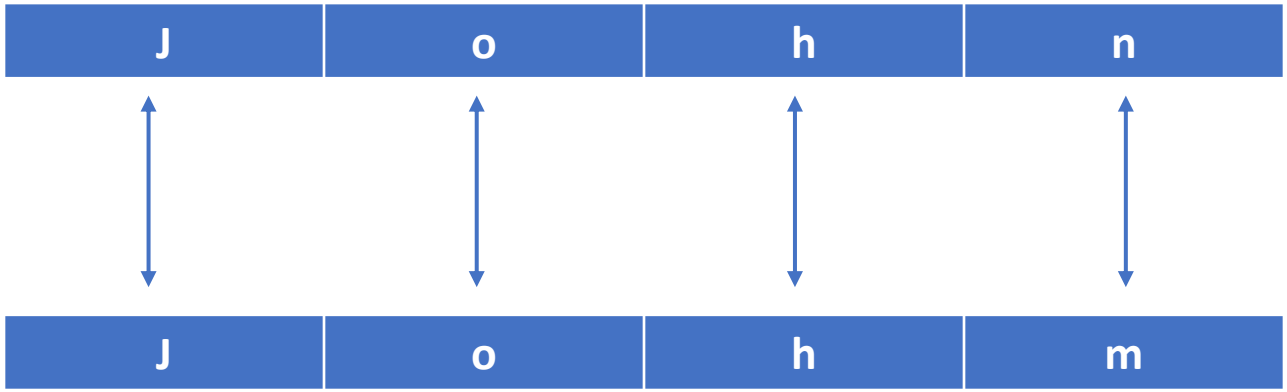
- The String class also provides the `compareTo` method, which is used to determine whether one String is greater than, equal to, or less than another String.

```
String name1 = "John", name2 = "John";
System.out.println(name1.comparedTo(name2));
0

name1 = "John", name2 = "Johm";
System.out.println(name1.comparedTo(name2));
1 [positive means greater than]

name1 = "Johm", name2 = "John";
System.out.println(name1.comparedTo(name2));
-1 [negative means lower than]
```

- See example: [StringCompareTo.java]

Comparing `String` Objects (7 of 9)

- When you use the `compareTo` method to compare two strings, the strings are compared character by character. This is often called a *lexicographical comparison*.



Comparing `String` Objects (8 of 9)

- In the `String` class the `equals` and `compareTo` methods are case sensitive.
- In order to compare two `String` objects that might have different case, use:
  - `equalsIgnoreCase`, or `compareToIgnoreCase`

```
String name1 = "JOHN", name2 = "John";
System.out.println(name1.equalsIgnoreCase(name2));
true
```

- See example: SecretWord.java

Comparing `String` Objects (9 of 9)

- Point to ponder #4

  What is the output of the expression below, 0, +, -?

```
String name1 = "John", name2 = "john";
System.out.println(name1.compareTo(name2));
```

-32

- What about now?

```
String name1 = "John", name2 = "john";
System.out.println(name1.compareToIgnoreCase(name2));
```

0

The Conditional Operator (1 of 4)

- 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 (2 of 4)

- 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 (3 of 4)

- Example:

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

first operand | second operand | third operand

- This line is functionally equivalent to:

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

The Conditional Operator (4 of 4)

- Point to ponder #5

  Convert the if-else statement below into a conditional operator (single line)

```
if (score < 60)
   System.out.println("Your grade is: Fail.");
else
   System.out.println("Your grade is: Pass.");
```

```
System.out.println("Your grade is: " +
                   (score < 60 ? "Fail." : "Pass."));
```

Decision Structures

Lecture 3c

Topics (1 of 2)

- The `if` Statement
- The `if-else` Statement
- Nested `if` statements
- The `if-else-if` Statement
- Logical Operators
- Comparing `String` Objects

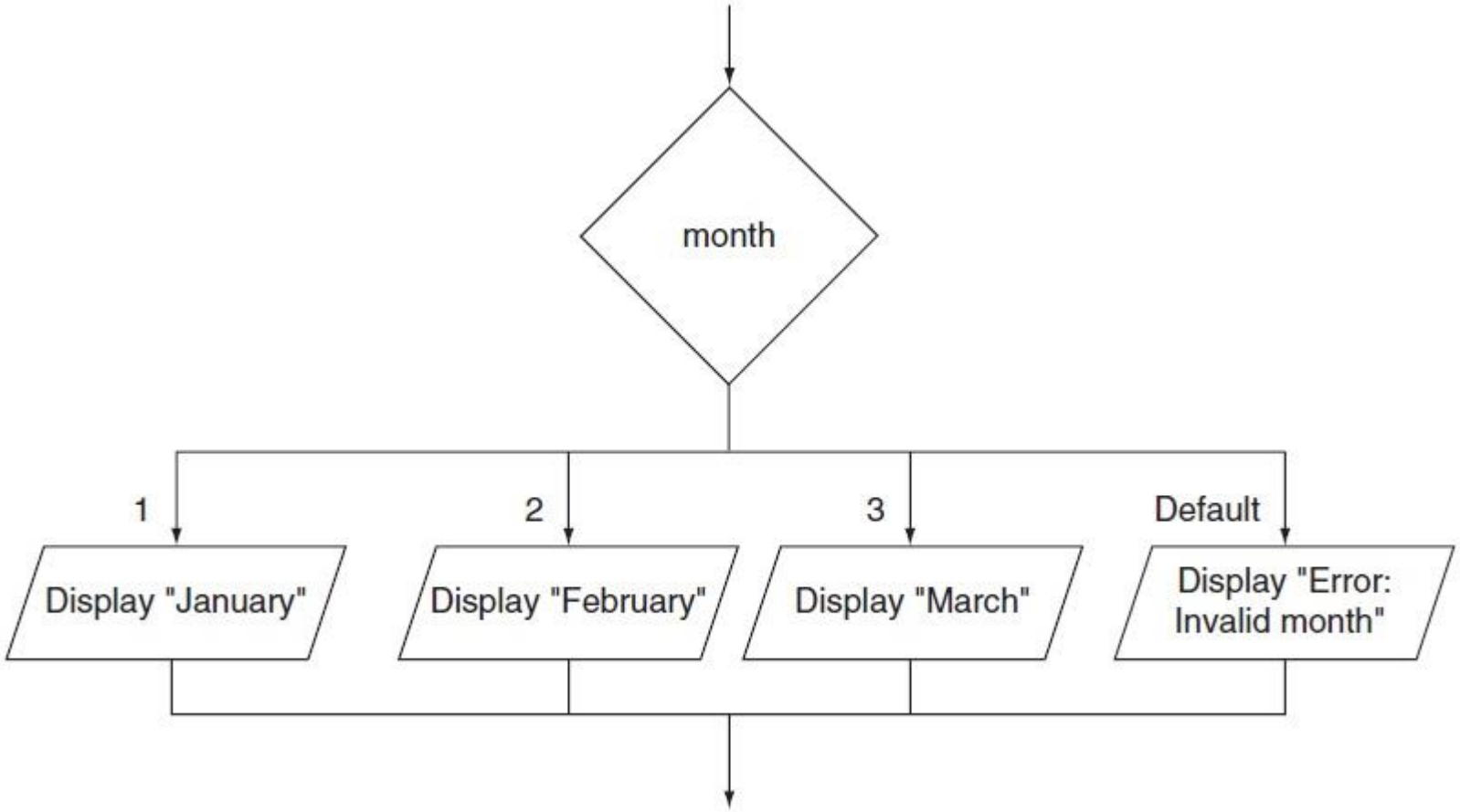
Topics (2 of 2)

- More about Variable Declaration and Scope
- The Conditional Operator
- The `switch` Statement
- Displaying Formatted Output with `System.out.printf` and `String.format`

The `switch` Statement (1 of 5)

- The `if-else` statement allows you to make true / false branches.
- The `switch` statement allows you to use the value of a variable or expression to determine how a program will branch.

The `switch` Statement (2 of 5)



The `switch` Statement (3 of 5)

- 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 CaseExpression:
    // place one or more statements here
    break;
    .
    .
    .
default:
    // place one or more statements here
}
```

The `switch` Statement (4 of 5)

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

- The `switch` statement will evaluate the *SwitchExpression*, which can be a variable or an expression that gives a `char`, `byte`, `short`, `int`, `long`, or `String` value.
- If there is an associated `case` statement that matches that value, program execution will be transferred to that `case` statement.

The `switch` Statement (5 of 5)

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

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

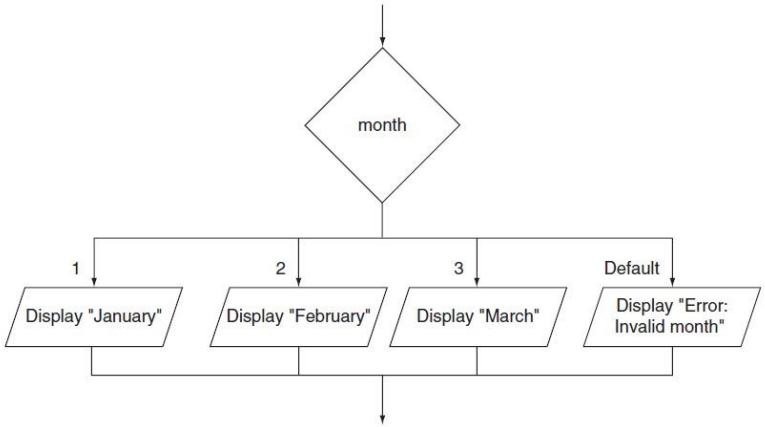
- If the *SwitchExpression* matches the *CaseExpression*, the Java statements between the colon and the `break` statement will be executed.

The `case` Statement (1 of 10)

- 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`.
- The `default` section is optional and will be executed if no *CaseExpression* matches the *SwitchExpression*.

The `case` Statement (2 of 10)

```
switch (month)
{
  case 1:
     System.out.println("January");
     break;
  case 2:
     System.out.println("February");
     break;
  case 3:
     System.out.println("March");
     break;
  default:
     System.out.println("Error: Invalid month");
     break;
}
```



The `case` Statement (3 of 10)

```
switch (month)
{
  case 1:
     System.out.println("January");
     break;
  case 2:
     System.out.println("February");
     break;
  case 3:
     System.out.println("March");
     break;
  default:
     System.out.println("Error: Invalid month");
     break;
}
```

≡?

```
if (month == 1)
{
  System.out.println("January");
}
else if (month == 2)
{
  System.out.println("February");
}
else if (month == 3)
{
  System.out.println("March");
}
else
{
  System.out.println("Error: Invalid month");
}
```

The `case` Statement (4 of 10)

```
switch (month)
{
  case 1:
     System.out.println("January");
     break;
  case 2:
     System.out.println("February");
     break;
  case 3:
     System.out.println("March");
     break;
  default:
     System.out.println("Error: Invalid month");
     break;
}
```

Point to ponder #1:

Input = 2

Output = ?

February

The `case` Statement (5 of 10)

```
switch (month)
{
  case 1:
     System.out.println("January");
     break;
  case 2:
     System.out.println("February");
     break;
  case 3:
     System.out.println("March");
     break;
  default:
     System.out.println("Error: Invalid month");
     break;
}
```

Point to ponder #2:

Input = 4

Output = ?

Error: Invalid month

The `case` Statement (6 of 10)

```
switch (month)
{
  case 1:
      System.out.println("January");
case 2:
      System.out.println("February");
case 3:
      System.out.println("March");
default:
      System.out.println("Error: Invalid month");
}
```

Missing break

Point to ponder #3:

Input = 3

Output = ?

March

Error: Invalid month

The `case` Statement (7 of 10)

- Without the break statement, the program "falls through" all of the statements below the one with the matching case expression.
  - See example: NoBreaks.java
  - See example: PetFood.java
- Sometimes this is what you want.
- See example: SwitchDemo.java

The `case` Statement (8 of 10)

```
// Display pricing for the selected grade.
switch(foodGrade)
{
  case 'a':
  case 'A':
     System.out.println("30 cents per lb.");
     break;
  case 'b':
  case 'B':
    System.out.println("20 cents per lb.");
    break;
  case 'c':
  case 'C':
    System.out.println("15 cents per lb.");
    break;
  default:
    System.out.println("Invalid choice.");
}
```

Point to ponder #4:

Input = b

Output = ?

20 cents per lb.

The `case` Statement (9 of 10)

```
// Display pricing for the selected grade.
switch(foodGrade)
{
  case 'a':
  case 'A':
     System.out.println("30 cents per lb.");
     break;
  case 'b':
  case 'B':
    System.out.println("20 cents per lb.");
    break;
  case 'c':
  case 'C':
    System.out.println("15 cents per lb.");
    break;
  default:
    System.out.println("Invalid choice.");
}
```

Point to ponder #5:

Input = B

Output = ?

20 cents per lb.

The `case` Statement (10 of 10)

```
// Display pricing for the selected grade.
switch(foodGrade)
{
  case 'a':
  case 'A':
     System.out.println("30 cents per lb.");
     break;
  case 'b':
  case 'B':
    System.out.println("20 cents per lb.");
    break;
  case 'c':
  case 'C':
    System.out.println("15 cents per lb.");
    break;
  default:
    System.out.println("Invalid choice.");
}
```

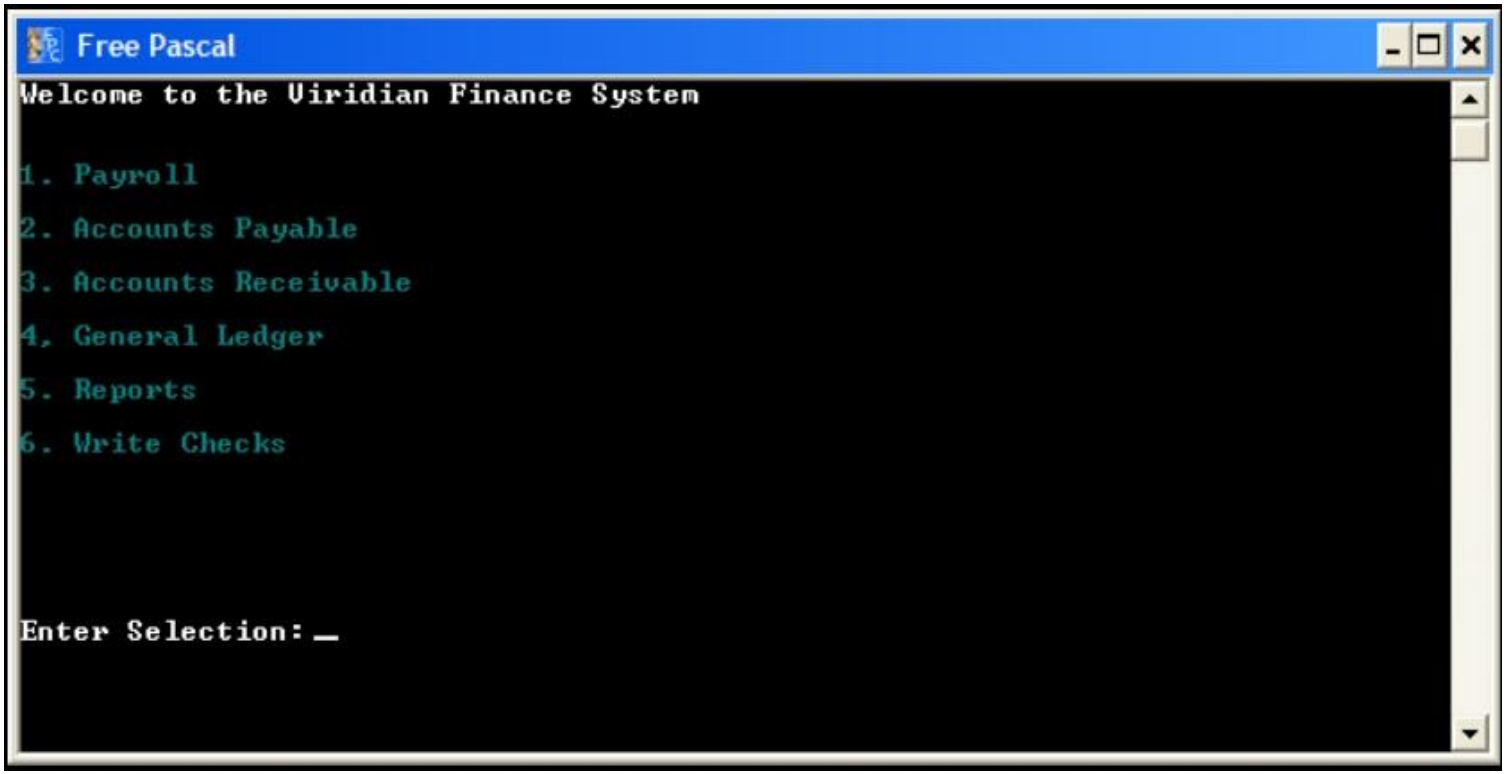
≡?

```
if (foodGrade == 'a' || foodGrade == 'A')
{
  System.out.println("30 cents per lb.");
}
else if (foodGrade == 'b' || foodGrade == 'B')
{
  System.out.println("20 cents per lb.");
}
else if (foodGrade == 'c' || foodGrade == 'C')
{
  System.out.println("15 cents per lb.");
}
else
{
  System.out.println("Invalid choice.");
}
```

The `switch` Statement

Point to ponder #6:

The switch statement is very popular to write an important part of programming applications. Can you guess which one?



The `switch` Statement

```
System.out.print("Enter your menu choice: ");
menuChoice = keyboard.nextLine();
switch(menuChoice)
{
  case 'a':
  case 'A':
     add(5,2);
     break;
  case 'm':
  case 'M':
     multiply(5,2);
     break;
  case 'd':
  case 'd':
     divide(5,2);
     break;
default:
     System.out.println("Invalid choice.");
}
```

The `System.out.printf` Method (1 of 16)

- You can use the `System.out.printf` method to perform formatted console output.
- The general format of the method is:

```
System.out.printf(FormatString, ArgList);
```

The `System.out.printf` Method (2 of 16)

```
System.out.printf(FormatString, ArgList);
```

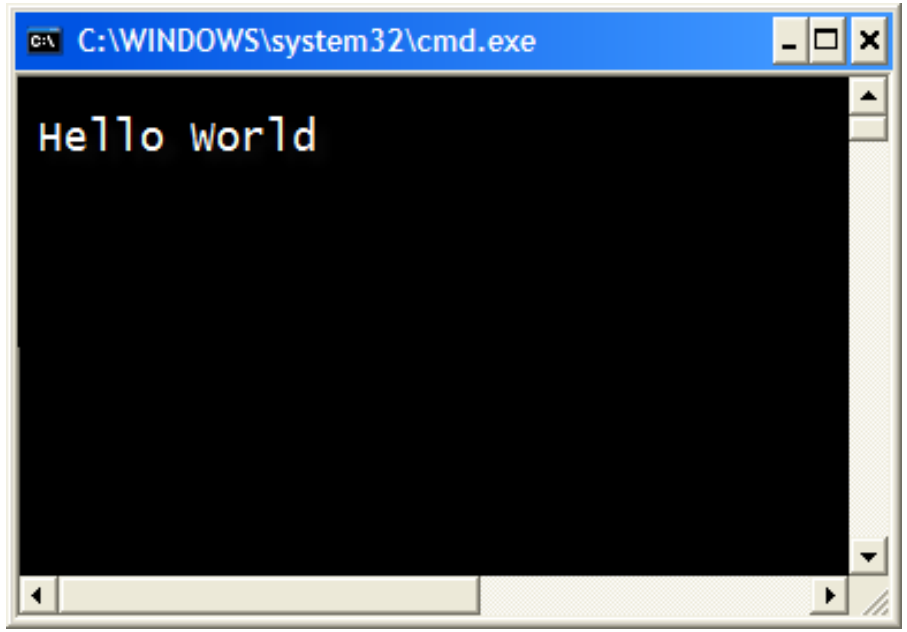
`FormatString` **is a string that contains text and/or special formatting specifiers.**

`ArgList` **is optional. It is a list of additional arguments that will be formatted according to the format specifiers listed in the format string.**

The `System.out.printf` Method (3 of 16)

- A simple example:

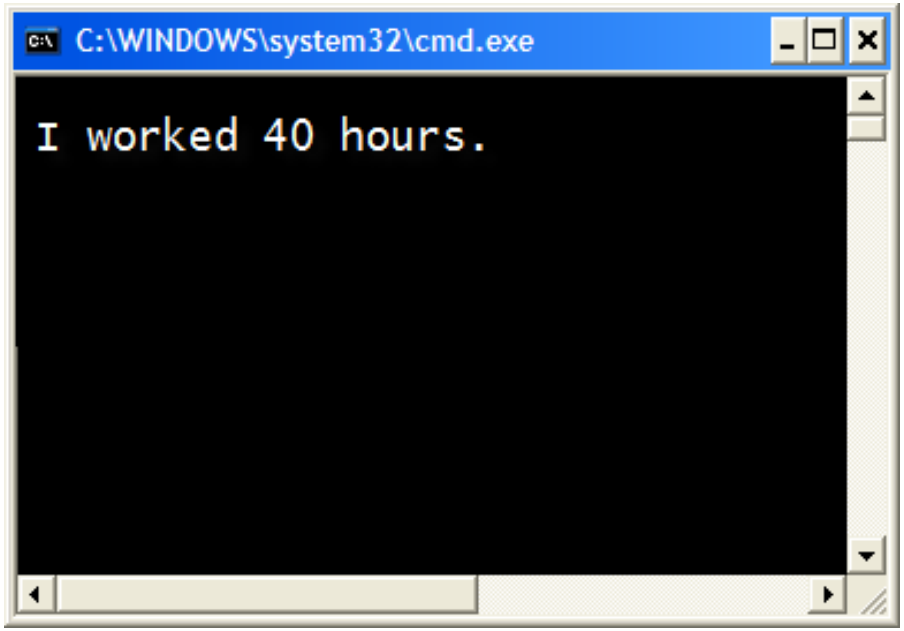
```
System.out.printf("Hello World\n");
```



The `System.out.printf` Method (4 of 16)

- Another example:

```
int hours = 40;
System.out.printf("I worked %d hours.\n", hours);
```



The `System.out.printf` Method (5 of 16)

```
int hours = 40;
System.out.printf("I worked %d hours.\n", hours);
```

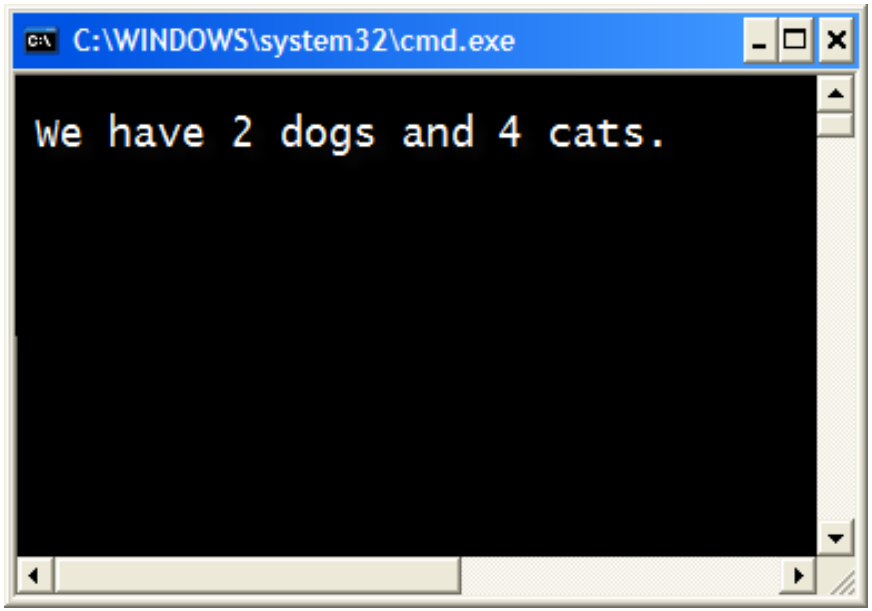
The `%d` format specifier indicates that a decimal integer will be printed.

The contents of the `hours` variable will be printed in the location of the `%d` format specifier.

The `System.out.printf` Method (6 of 16)

- Another example:

```
int dogs = 2, cats = 4;
System.out.printf("We have %d dogs and %d
                    cats.\n", dogs, cats);
```



The `System.out.printf` Method (7 of 16)

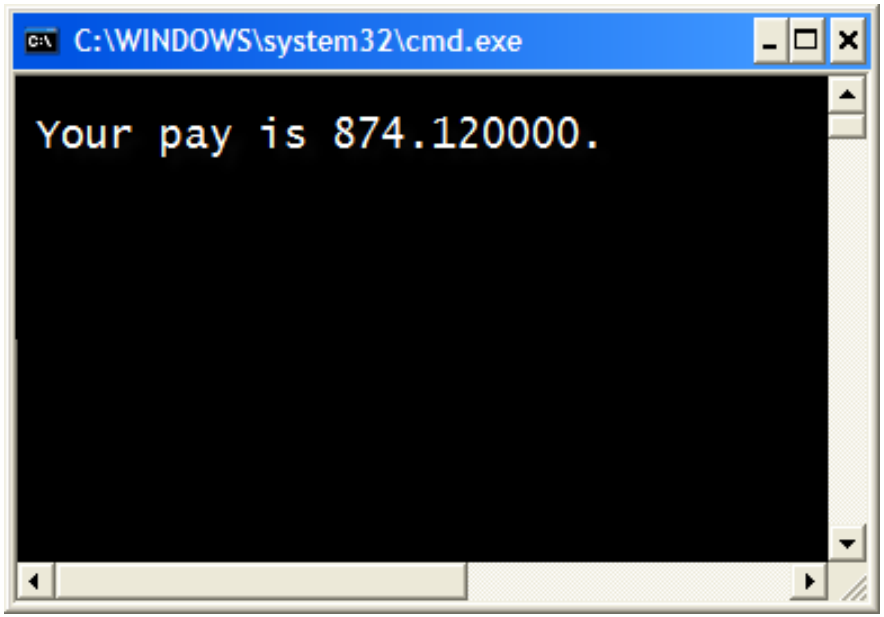
```
int dogs = 2, cats = 4;
System.out.printf("We have %d dogs and %d
                    cats.\n", dogs, cats);
```

The contents of the `dogs` and `cats` variables will be printed in the location of the first and second `%d` format specifiers respectively.

The `System.out.printf` Method (8 of 16)

- Another example:

```
double grossPay = 874.12;
System.out.printf("Your pay is %f.\n", grossPay);
```



The `System.out.printf` Method (9 of 16)

- Another example:

```
double grossPay = 874.12;
System.out.printf("Your pay is %f.\n", grossPay);
```

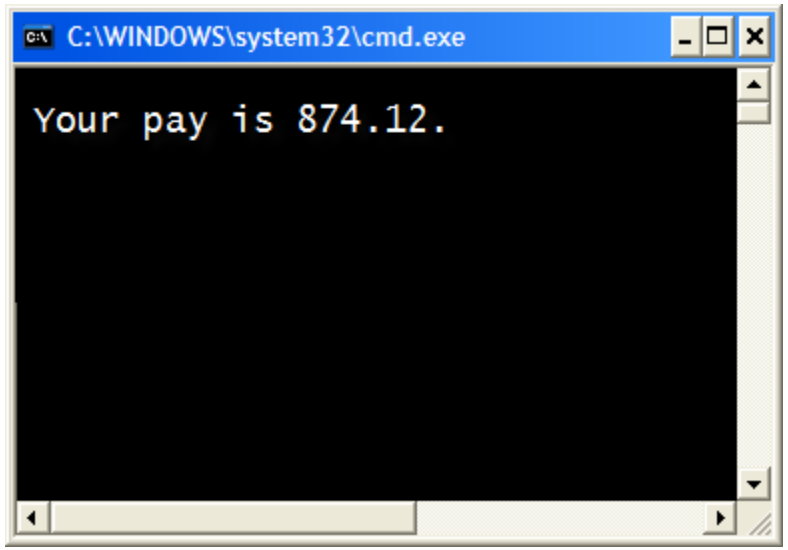
The `%f` format specifier indicates that a floating-point value will be printed.

The contents of the `grossPay` variable will be printed in the location of the `%f` format specifier.

The `System.out.printf` Method (10 of 16)

- Another example:

```
double grossPay = 874.12111;
System.out.printf("Your pay is %.2f.\n",
                  grossPay);
```



The `System.out.printf` Method (11 of 16)

- Another example:

```
double grossPay = 874.12;
System.out.printf("Your pay is %.2f.\n", grossPay);
```

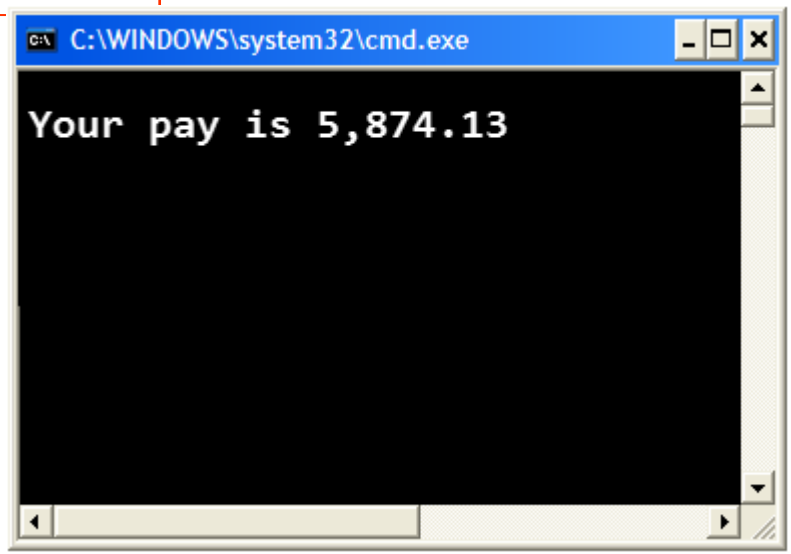
The `%.2f` format specifier indicates that a floating-point value will be printed, rounded to two decimal places.

The `System.out.printf` Method (12 of 16)

- Another example:

```
double grossPay = 5874.127;
System.out.printf("Your pay is %,.2f.\n", grossPay);
```

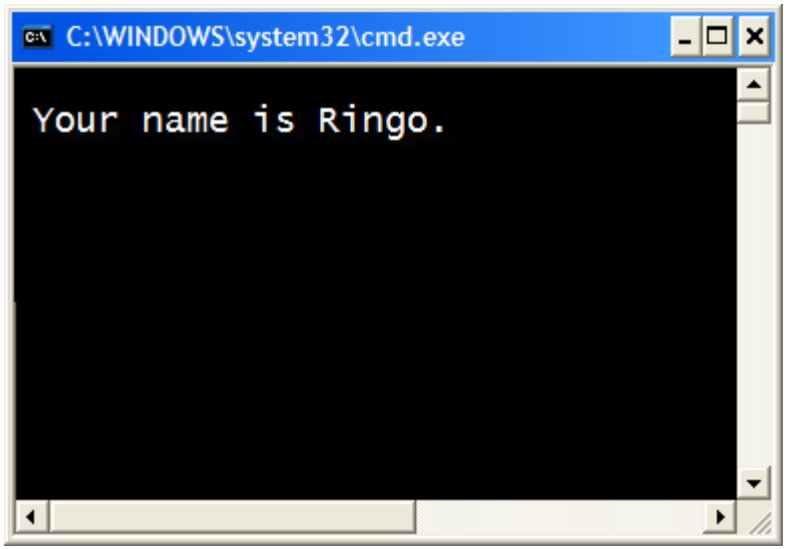
The `%,.2f` format specifier indicates that a floating-point value will be printed with comma separators, rounded to two decimal places.



The `System.out.printf` Method (13 of 16)

- Another example:

```
String name = "Ringo";
System.out.printf("Your name is %s.\n", name);
```

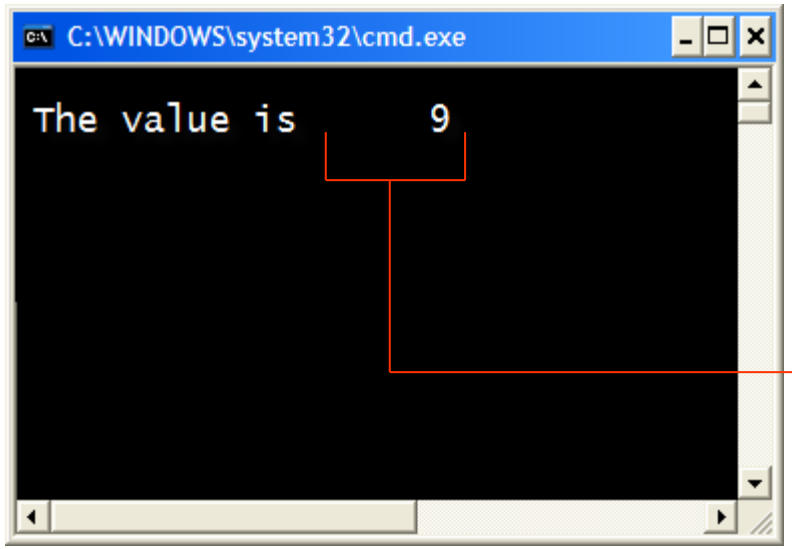


The `%s` format specifier indicates that a string will be printed.

The `System.out.printf` Method (14 of 16)

- Specifying a field width:

```
int number = 9;
System.out.printf("The value is %6d\n", number);
```

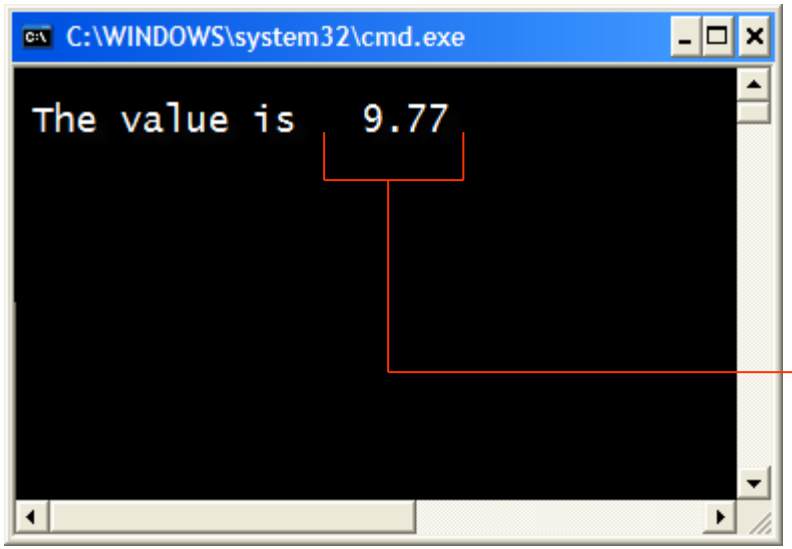


The `%6d` format specifier indicates the integer will appear in a field that is 6 spaces wide.

The `System.out.printf` Method (15 of 16)

- Another example:

```
double number = 9.76891;
System.out.printf("The value is %6.2f\n", number);
```



The `%6.2f` format specifier indicates the number will appear in a field that is 6 spaces wide and be rounded to 2 decimal places.

The `System.out.printf` Method (16 of 16)

- When displaying numbers, the general syntax for writing a format specifier is:

  `%[flags][width][.precision]conversion`

  `Flags` = padding with zeros, left-justifying, comma separators

  `Width` = minimum field width for the value.

  `.`*`precision`* = number of decimal places that the number should be rounded to

  `Conversion` = conversion character, such as `f` for floating-point, `d` for decimal integer, or `s` for String.

The `String.format` Method (1 of 3)

- The `String.format` method works exactly like the `System.out.printf` method, except that it does not display the formatted string on the screen.
- Instead, it returns a reference to the formatted string.
- You can assign the reference to a variable, and then use it later.

The `String.format` Method (2 of 3)

- The general format of the method is:

```
String.format(FormatString,ArgumentList);
```

FormatString **is a string that contains text and/or special formatting specifiers.**

ArgumentList **is optional. It is a list of additional arguments that will be formatted according to the format specifiers listed in the format string.**

The `String.format` Method (3 of 3)

CurrencyFormat2.java

```
/**
This program demonstrates how to use the String.format method to format a
number as currency.
*/
public class CurrencyFormat2
{
  public static void main(String[] args)
  {
     double monthlyPay = 5000.0;
     double annualPay = monthlyPay * 12;
     String output = String.format("Your annual pay is $%,.2f", annualPay);

     System.out.print(output);
  }
}
```

```
Your annual pay is $60,000.00
```

The `String.format` Method (3 of 3)

`CurrencyFormat3.java`

```
/**
This program demonstrates how to use the String.format method to format a
number as currency.
*/
public class CurrencyFormat3
{
  public static void main(String[] args)
  {
     double monthlyPay = 5000.0;
     double annualPay = monthlyPay * 12;

     System.out.print(String.format("Your annual pay is $%,.2f", annualPay));
  }
}
```

```
Your annual pay is $60,000.00
```