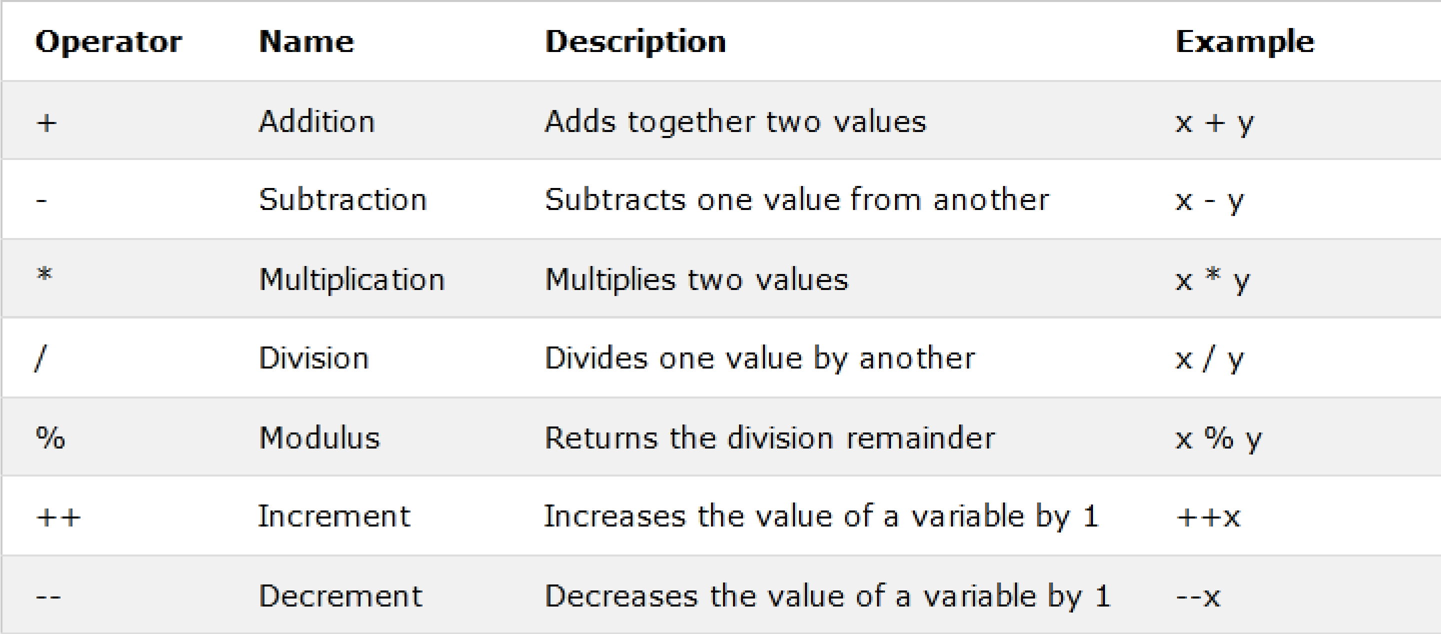
Programming in Java

3. Operators and Control Statements

Operators

- Operators are symbols that perform operations on variables and values.
  - Expressions are syntactically valid combinations of variables and operators
- Java supports a similar set of operators that most programming languages do :
- The main category of operators are
  - Arithmetic: The usual operations of +,-,/,* with a few twists
  - Relational: Comparison operators that return a true or false like <,<=,==,>,>=
  - Logical: Combine two logical values with && (and), || (or), ! (not_
  - Assignment: Already seen this when assigning a value to a variable
  - Unary: Operates on a single variable, the unary minus that changes sign x = -y;
  - Bitwise: Performs bit level operations, rarely used in modern programming, left over from C
  - Ternary: Takes three operands

Arithmetic Operators



Operator	Name	Description	Example
+	Addition	Adds together two values	x + y
-	Subtraction	Subtracts one value from another	x - y
*	Multiplication	Multiplies two values	x * y
/	Division	Divides one value by another	x / y
%	Modulus	Returns the division remainder	x % y
++	Increment	Increases the value of a variable by 1	++x
--	Decrement	Decreases the value of a variable by 1	--x

- These work just like you would expect with a couple of interesting quirks

Mixed Mode Arithmetic

- Obviously, we can only use these operators with numeric values
- We cannot add a Boolean and an int for example
- For each binary operator, Java can only execute it when both operands are the same
  - Java can't add an integer and a float because they have different internal operations
  - If you try to do this, it's called mixed mode arithmetic
- When one operand is an integral and the other is a floating point number
  - Java will convert the int or long to a double, then do do the operations
  - 2 + 4.8 becomes 2.0 + 4.8 and the result is a float 6.8
- It is considered good practice to explicitly cast the operand for readability and to prevent unintended data casts
  - (double)2 + 4.8
  - Or if you want integer addition 2 + (int)4.8 → 6

The Division Operator

- There are actually two division operations
- The first is floating point division
  - (float)/(float) → (float)
  - *9.0 / 2.0 → 3.5*
- The second is integer division
  - (int)/(int) → (int)
  - *9 / 2 → 3*
  - Note: the remainder is dropped but can be evaluated using %
  - *9 % 2 → 1*
- When the operands are mixed, as discussed in the previous slide
  - The integral value is converted to a floating point and floating point division is used

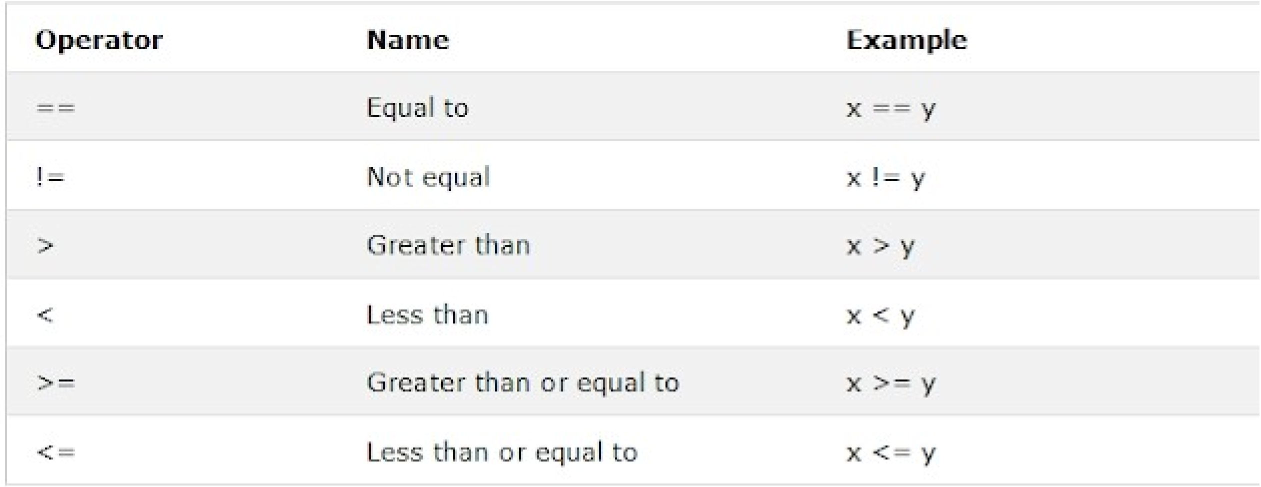
Unary Operators

- Used to increment, decrement, or negate a value.
  - **-** , Negates the value.
  - **+** , Indicates a positive value
    - Converts byte, char, or short to int
    - This is a side effect
  - **++** , Increments by 1.
    - Postfix (i++): Uses value first, then increments.
    - Prefix (++i): Increments first, then uses value.
  - **--** , Decrements by 1.
    - Postfix (i--): Uses value first, then decrements.
    - Prefix (--i): Decrements first, then uses value.
  - **!** , Inverts a boolean value.

Assignment Operators

- Can be combined with arithmetic operators
- Form is x (op)= y
  - Short hand for x = x (op) y
  - += , Add and assign.
    - X += 4 → X = X + 4
  - -= , Subtract and assign.
    - X -= 4 → X = X - 4
  - *= , Multiply and assign.
    - X *= 4 → X = X * 4
  - /= , Divide and assign.
    - X /= 4 → X = X / 4
  - %= , Modulo and assign.
    - X %= 4 → X = X % 4
- Use of this form can lead to confusing code with complex expressions

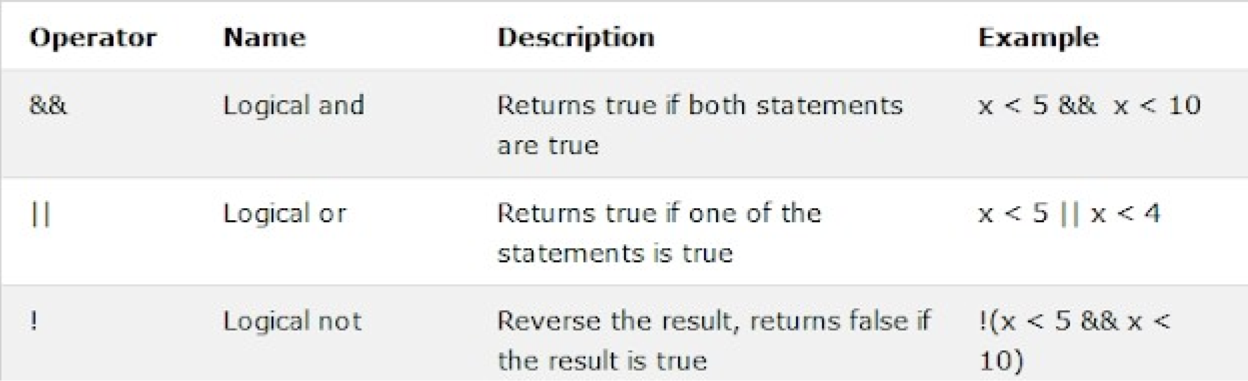
Comparison Operators



Operator	Name	Example
==	Equal to	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

- Always returns a Boolean value
- Only defined for numeric values
  - The expressions *x == y* and *x != y* where x and y are floating point numbers may not be corrrect
  - This is due the inherent problem of accuracy when rounding floating point values

Logical Operators



Operator	Name	Description	Example
&&	Logical and	Returns true if both statements are true	x < 5 && x < 10
||	Logical or	Returns true if one of the statements is true	x < 5 || x < 4
!	Logical not	Reverse the result, returns false if the result is true	!(x < 5 && x < 10)

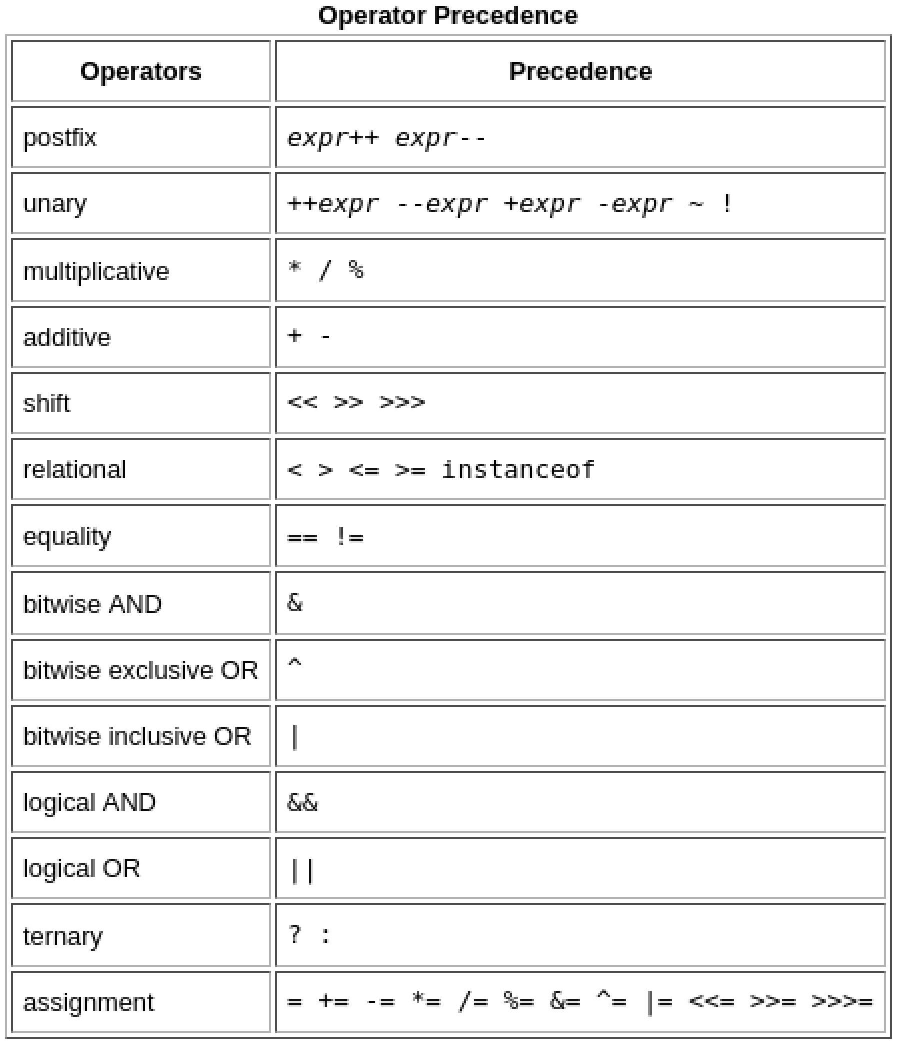
- These operations are short circuited
  - Evaluation only proceeds as long as necessary to predict the final result
  - This means in x && y, if x is false, then y is not evaluated since no matter what it is, the final result is false
  - And in x || y, if x is true, then y is not evaluated since no matter what it is, the final result is true

Bitwise Operators

- These are logical operators that do logical operations bit by bit
  - Operates on integral values, not Booleans
- This was extensively used in C
  - Java “inherited” it via C++
  - It’s only used in very rare cases in Java
  - Will not be covered in this course
- Mentioned because the bit wise operators for and is & and or is |
  - Easy to typo and confuse with the standard logical operators && and ||

Operator Precedence

- Just like in math, operators have precedence
  - 8 + 2 * 4
  - The multiplication is done first because of precedence
  - Precedence can be overwritten with ()
  - (8 + 2) * 4
  - The addition is done first because of the ()
- Best practice
  - Use () to make precedence explicit
  - Helps avoid subtle bugs
  - Also makes the code much more readable



Operator Precedence

Operators	Precedence
postfix	`expr++ expr--`
unary	`++expr --expr +expr -expr ~ !`
multiplicative	`* / %`
additive	`+ -`
shift	`<< >> >>>`
relational	`< > <= >= instanceof`
equality	`== !=`
bitwise AND	`&`
bitwise exclusive OR	`^`
bitwise inclusive OR	`|`
logical AND	`&&`
logical OR	`||`
ternary	`? :`
assignment	`= += -= *= /= %= &= ^= |= <<= >>= >>>=`

Lab 3-1

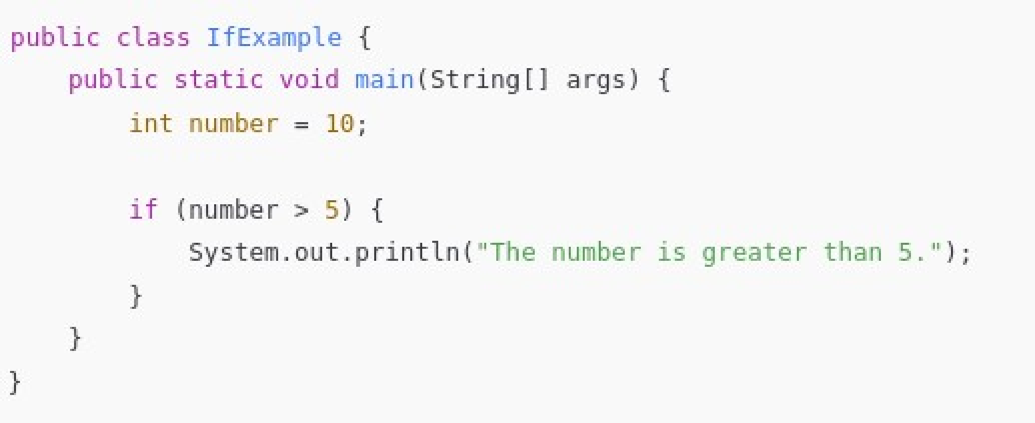
Operators

Control Statement

- Control statements occur in every programming language
- There are three basic types
  - If-then logical decisions
  - Looping constructs
  - Select or Switch statements
- Note: Some of the following is taken from the official Java Tutorial
  - https://docs.oracle.com/javase/tutorial/java/nutsandbolts/flow.html

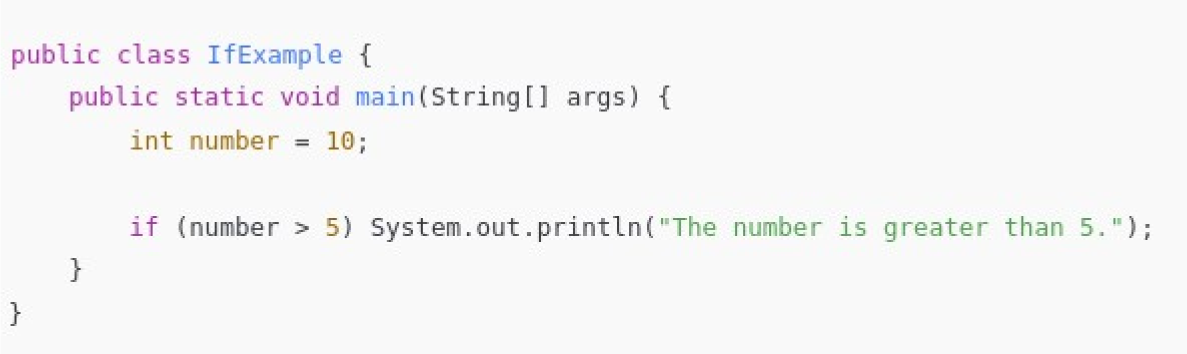
If Statements

- If statements executes a certain section of code only if a particular test evaluates to true.
  - The conditional code is usually a block delimited by {}
  - However, if there is only one line, the block can be replaced by a single statement
  - Always use the block form unless there is a good reason not to
  - This improves the readability of the code
  - The same is true for the else block



```java
public class IfExample {
    public static void main(String[] args) {
        int number = 10;

        if (number > 5) {
            System.out.println("The number is greater than 5.");
        }
    }
}
```

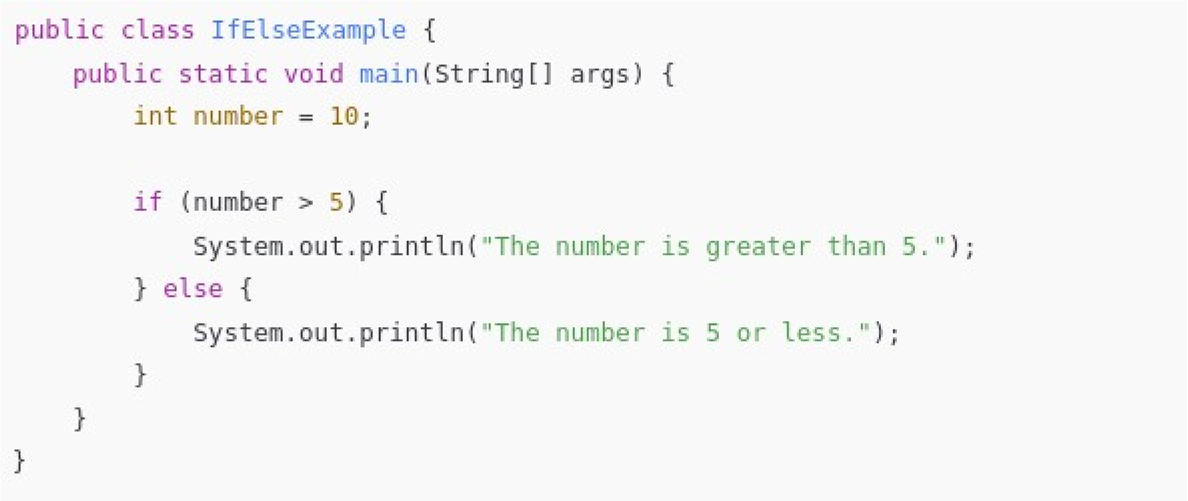


```java
public class IfExample {
    public static void main(String[] args) {
        int number = 10;

        if (number > 5) System.out.println("The number is greater than 5.");
    }
}
```

If Else Statements

- The if-then-else statement provides a secondary path of execution when an "if" clause evaluates to false.

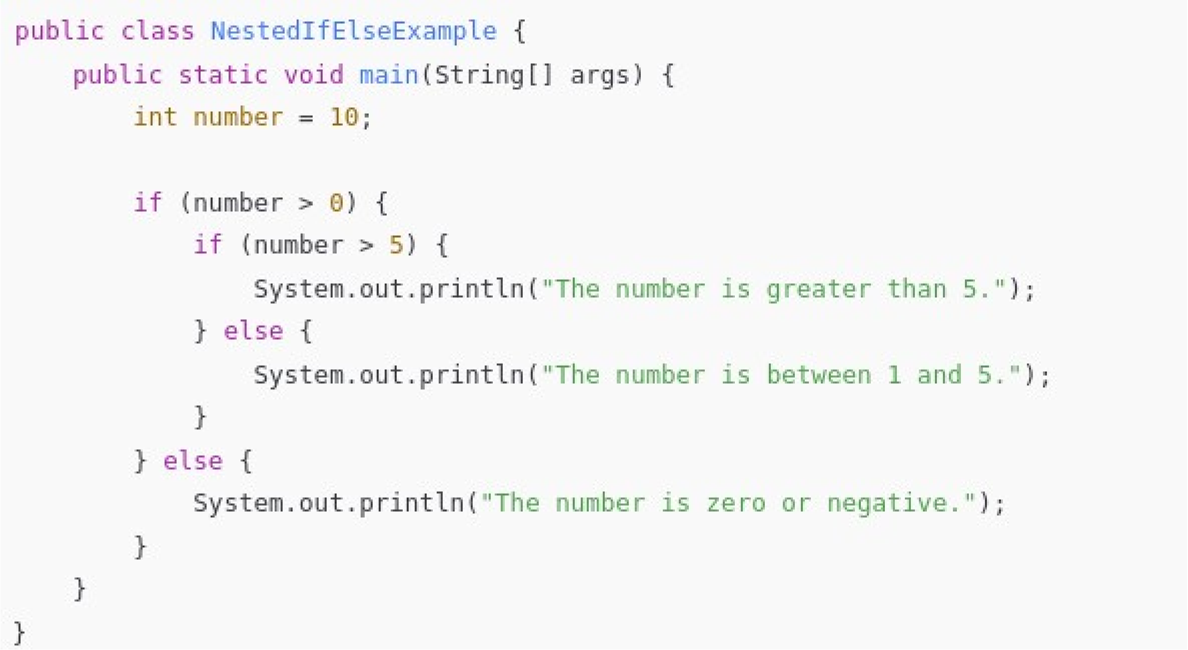


```java
public class IfElseExample {
    public static void main(String[] args) {
        int number = 10;

        if (number > 5) {
            System.out.println("The number is greater than 5.");
        } else {
            System.out.println("The number is 5 or less.");
        }
    }
}
```

Nested If Statements

- If statements can be nested as shown to implement more complex logic
- Caveat
  - If not all the statements have an else clause, it can be confusing as to which else goes with which if
  - Often referred to as the “dangling” else problem

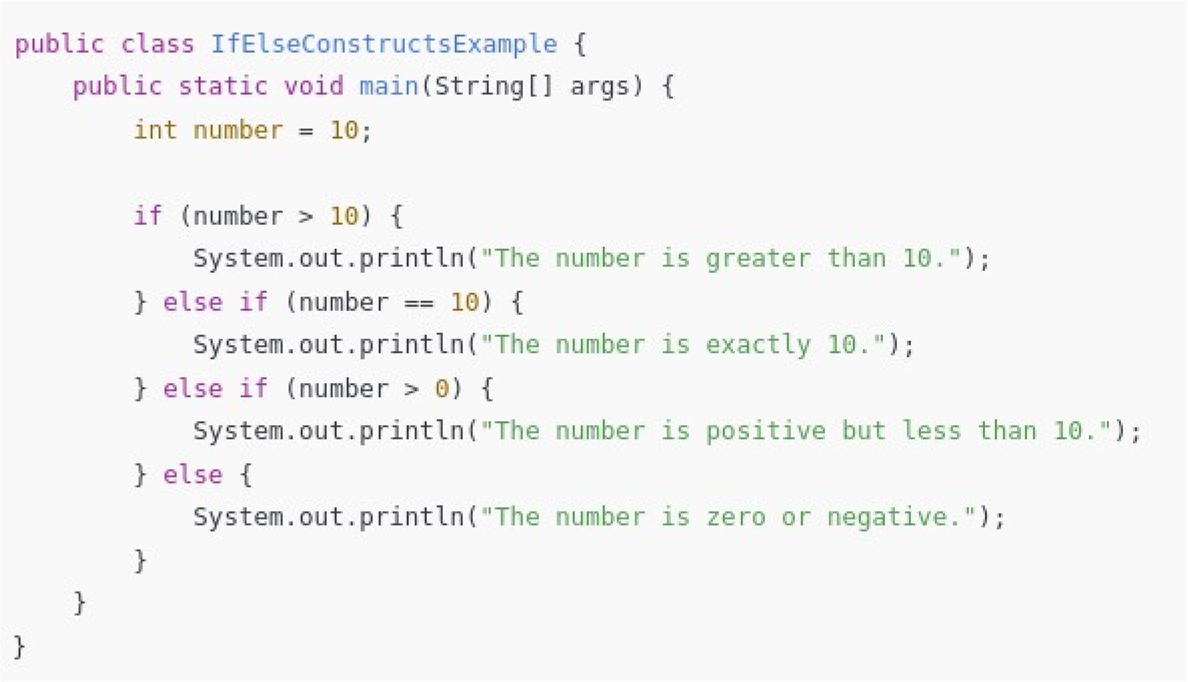


```java
public class NestedIfElseExample {
    public static void main(String[] args) {
        int number = 10;

        if (number > 0) {
            if (number > 5) {
                System.out.println("The number is greater than 5.");
            } else {
                System.out.println("The number is between 1 and 5.");
            }
        } else {
            System.out.println("The number is zero or negative.");
        }
    }
}
```

Else if Statements

- When testing multiple conditions, a cleaner form is the if-else if- else
  - The conditions are checked in turn to find a true result
  - If no true result occurs, then the final else is executed

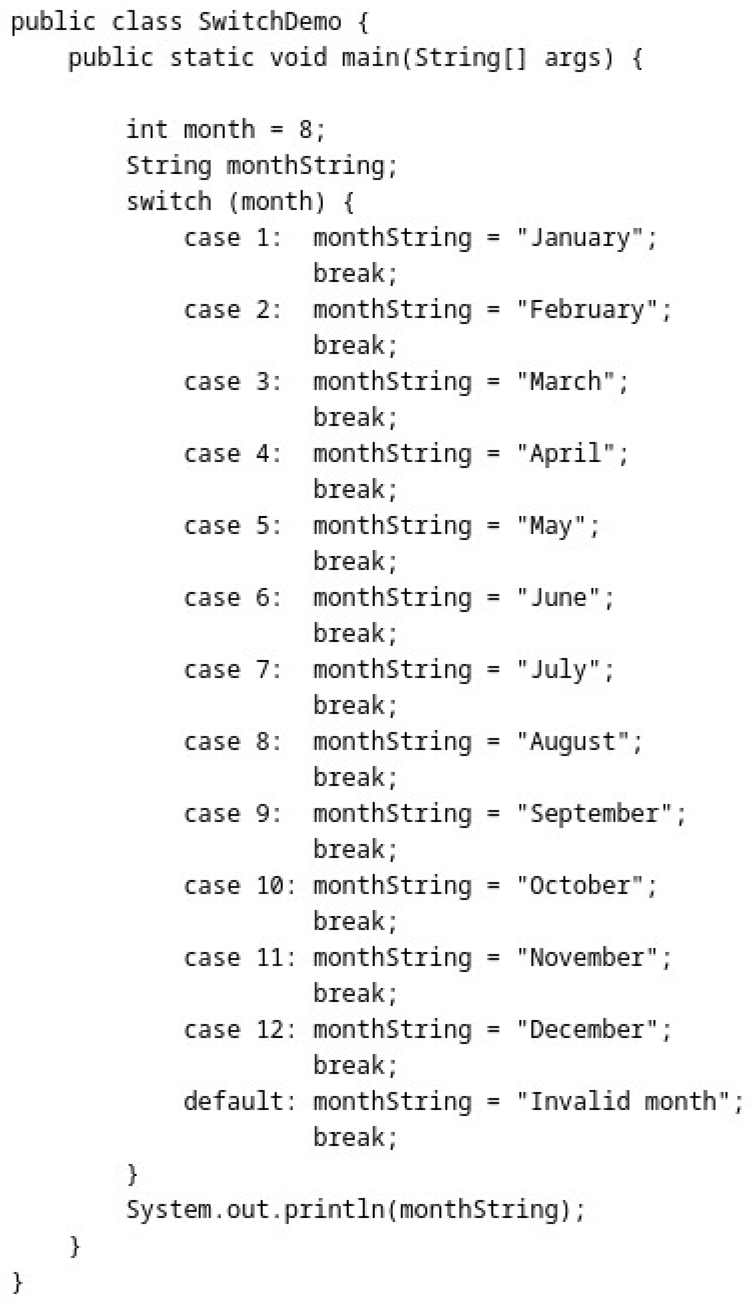


```
public class IfElseConstructsExample {
    public static void main(String[] args) {
        int number = 10;

        if (number > 10) {
            System.out.println("The number is greater than 10.");
        } else if (number == 10) {
            System.out.println("The number is exactly 10.");
        } else if (number > 0) {
            System.out.println("The number is positive but less than 10.");
        } else {
            System.out.println("The number is zero or negative.");
        }
    }
}
```

Switch Statements

- The switch statements test a value and then chose the corresponding code to execute, called a case
  - A more understandable form of the if-else if construct
- When a match is found
  - The code in the case is executed until a break statement is encountered
  - A common error is forgetting the break and executing the code in the following case by mistake
  - If there are no matches, the default case is executed

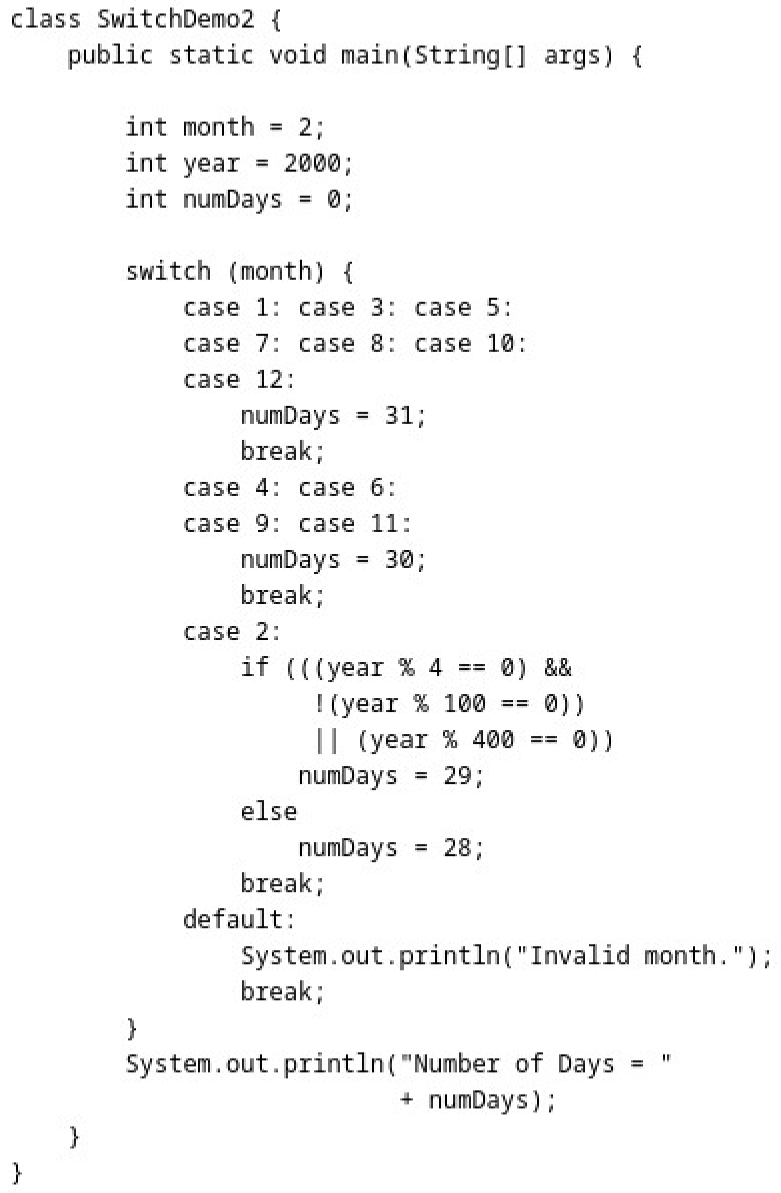


```
public class SwitchDemo {
    public static void main(String[] args) {

        int month = 8;
        String monthString;
        switch (month) {
            case 1:  monthString = "January";
                     break;
            case 2:  monthString = "February";
                     break;
            case 3:  monthString = "March";
                     break;
            case 4:  monthString = "April";
                     break;
            case 5:  monthString = "May";
                     break;
            case 6:  monthString = "June";
                     break;
            case 7:  monthString = "July";
                     break;
            case 8:  monthString = "August";
                     break;
            case 9:  monthString = "September";
                     break;
            case 10: monthString = "October";
                     break;
            case 11: monthString = "November";
                     break;
            case 12: monthString = "December";
                     break;
            default: monthString = "Invalid month";
                     break;
        }
        System.out.println(monthString);
    }
}
```

Switch Statements Fall Through

- Often used when we want to use the same case for different values



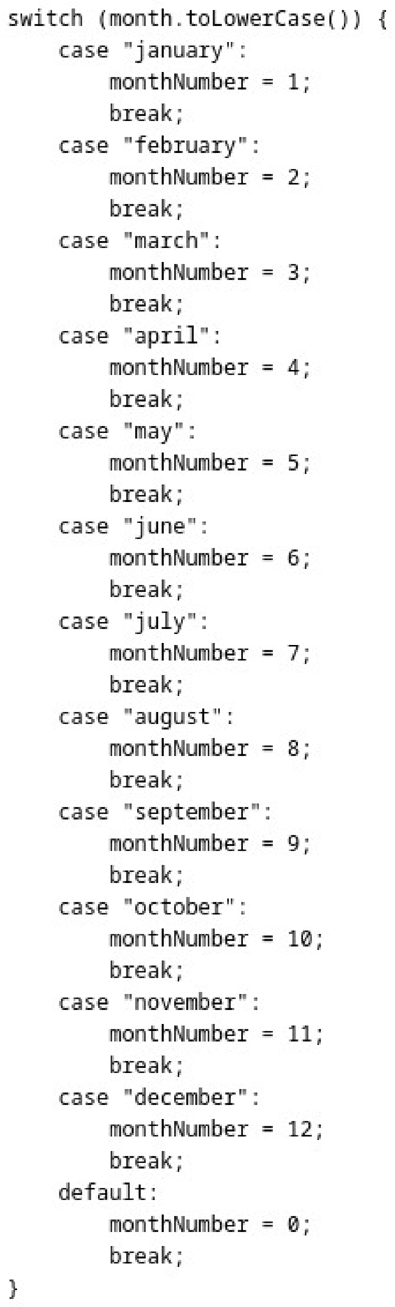
```
class SwitchDemo2 {
    public static void main(String[] args) {

        int month = 2;
        int year = 2000;
        int numDays = 0;

        switch (month) {
            case 1: case 3: case 5:
            case 7: case 8: case 10:
            case 12:
                numDays = 31;
                break;
            case 4: case 6:
            case 9: case 11:
                numDays = 30;
                break;
            case 2:
                if (((year % 4 == 0) &&
                     !(year % 100 == 0))
                     || (year % 400 == 0))
                    numDays = 29;
                else
                    numDays = 28;
                break;
            default:
                System.out.println("Invalid month.");
                break;
        }
        System.out.println("Number of Days = "
                           + numDays);
    }
}
```

Switch Statements Tests

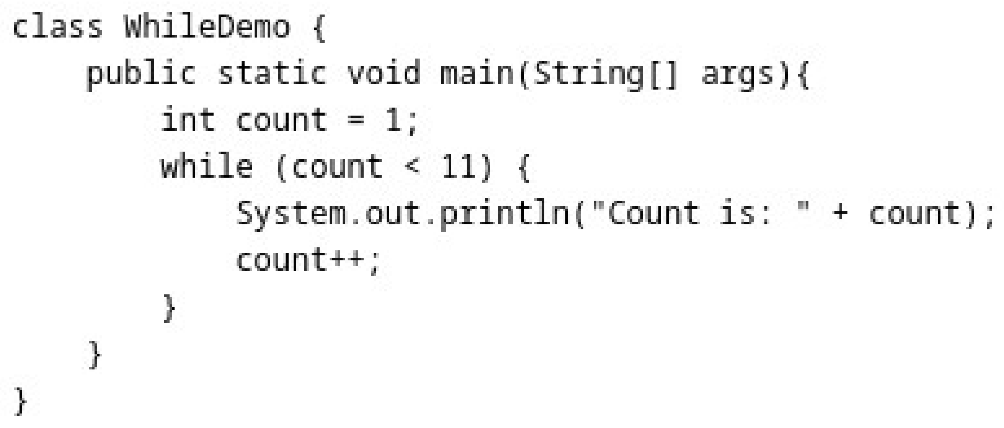
- Normally the switch statement test value is an integer type
- Floats cannot be used
- Strings can also be used.
- In the example ”month” is a string



```
switch (month.toLowerCase()) {
    case "january":
        monthNumber = 1;
        break;
    case "february":
        monthNumber = 2;
        break;
    case "march":
        monthNumber = 3;
        break;
    case "april":
        monthNumber = 4;
        break;
    case "may":
        monthNumber = 5;
        break;
    case "june":
        monthNumber = 6;
        break;
    case "july":
        monthNumber = 7;
        break;
    case "august":
        monthNumber = 8;
        break;
    case "september":
        monthNumber = 9;
        break;
    case "october":
        monthNumber = 10;
        break;
    case "november":
        monthNumber = 11;
        break;
    case "december":
        monthNumber = 12;
        break;
    default:
        monthNumber = 0;
        break;
}
```

While loops

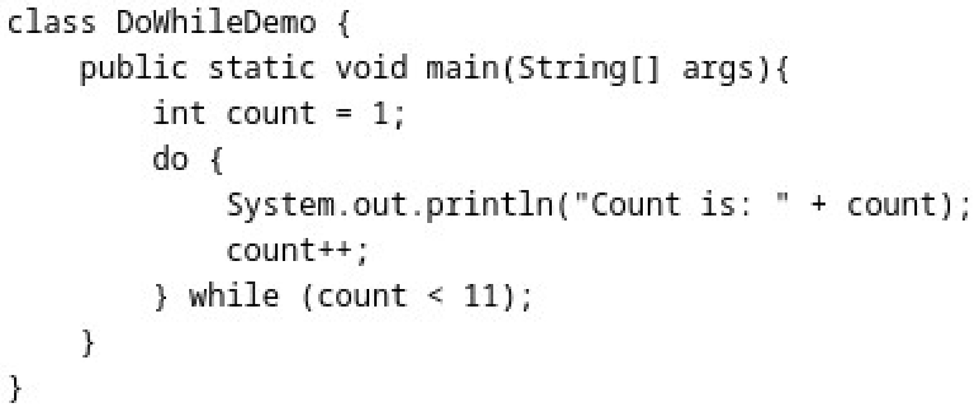
- Continually executes a block of statements while a particular condition is true.
- Continues testing the expression on each iteration and executing its block until the expression evaluates to false.
- A common error is to create an infinite loop
  - For example, forgetting count++ in the example creates an infinite loop



```
class WhileDemo {
    public static void main(String[] args){
        int count = 1;
        while (count < 11) {
            System.out.println("Count is: " + count);
            count++;
        }
    }
}
```

Do-while loops

- A do-while evaluates its expression at the bottom of the loop instead of the top like the while loop
- The statements within the do block are always executed at least once



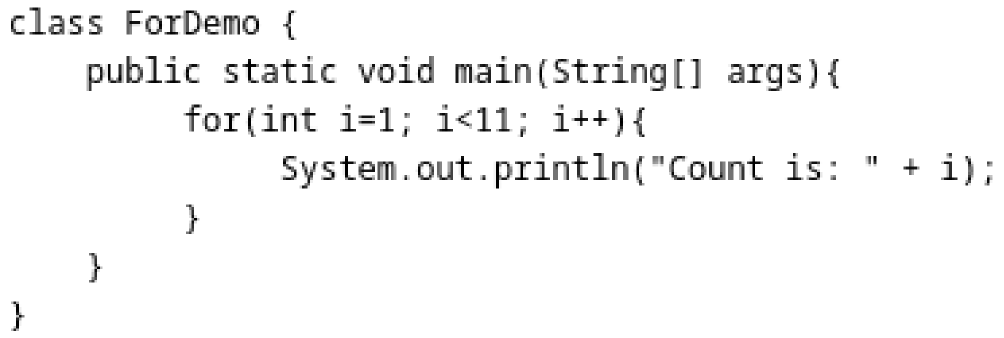
```
class DoWhileDemo {
    public static void main(String[] args){
        int count = 1;
        do {
            System.out.println("Count is: " + count);
            count++;
        } while (count < 11);
    }
}
```

for loops

- Compact way to iterate over a range of values
- The general form of the for statement can be expressed as follows:

```
for (initialization; termination;  increment) {
    statement(s)
}
```

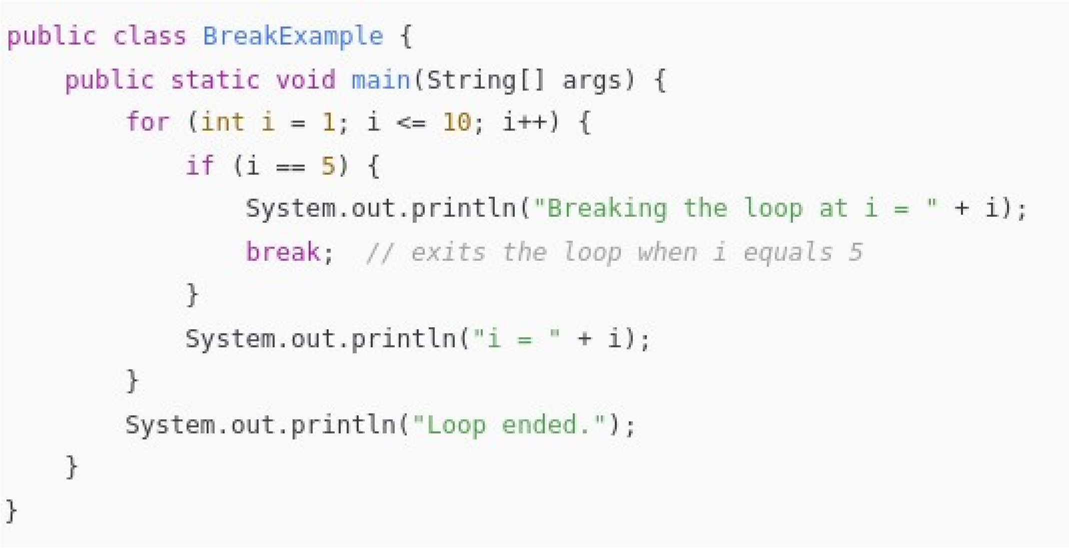
  - The initialization expression initializes the loop; it's executed once, as the loop begins.
  - When the termination expression evaluates to false, the loop terminates.
  - The increment expression is invoked after each iteration through the loop to increment or decrement a value.
  - All of the terms are optional



```
class ForDemo {
    public static void main(String[] args){
         for(int i=1; i<11; i++){
              System.out.println("Count is: " + i);
         }
    }
}
```

Break

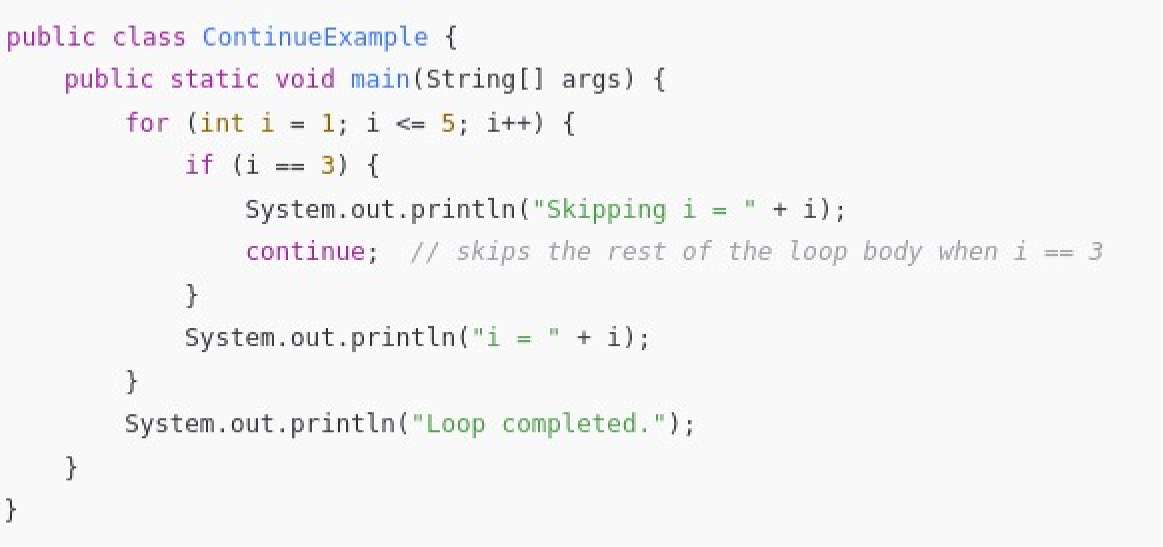
- A break statement immediately terminates a loop and resumes at the first statement following the loop



```java
public class BreakExample {
    public static void main(String[] args) {
        for (int i = 1; i <= 10; i++) {
            if (i == 5) {
                System.out.println("Breaking the loop at i = " + i);
                break;  // exits the loop when i equals 5
            }
            System.out.println("i = " + i);
        }
        System.out.println("Loop ended.");
    }
}
```

Continue

- A continue statement immediately terminates the current iteration of the loop and starts the next iteration



```
public class ContinueExample {
    public static void main(String[] args) {
        for (int i = 1; i <= 5; i++) {
            if (i == 3) {
                System.out.println("Skipping i = " + i);
                continue;  // skips the rest of the loop body when i == 3
            }
            System.out.println("i = " + i);
        }
        System.out.println("Loop completed.");
    }
}
```

Lab 3-2

Control Flow

Java™