COMP1618

Lecture 2 : Loops and Debugging programs

Lecture Outline

We will discuss the following main topics:

- Decisions and Loops
  - The if, if-else Statement
  - Comparing String Objects
  - The switch Statement
  - The while, do-while
  - The break and continue Statements
- Debugging programs in java NetBeans IDE

Motivation

- A program is a sequence of programming statements

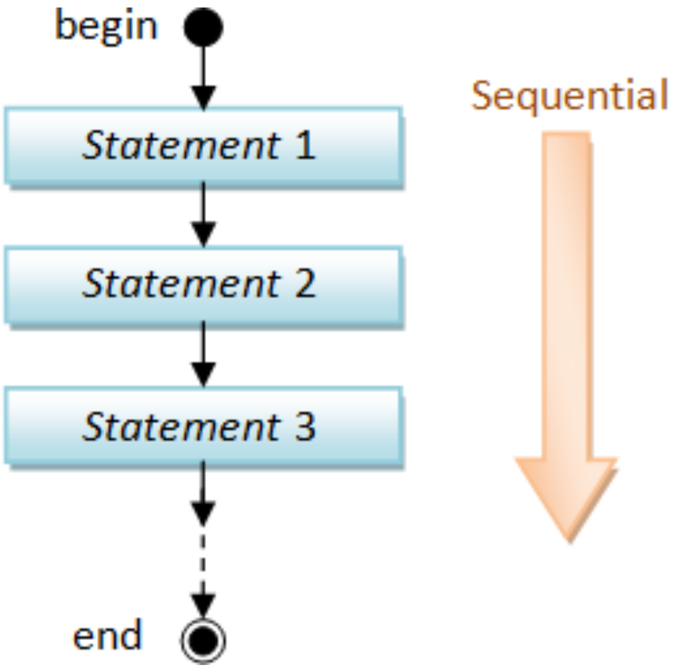
- Make a decision (conditional) when a user hit a button?

The Traffic Light: An Everyday IF Statement

IF light is **red**, THEN stop!

IF light is **yellow**, THEN what???

IF light is **green**, THEN go!

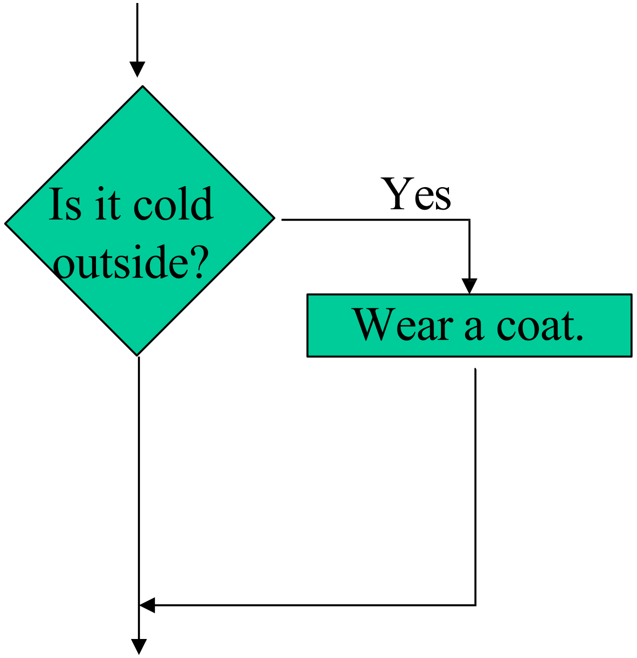


The `if` Statement

- Decides whether a section of code executes or not.
- Uses a `boolean` to decide whether the next statement or block of statements executes.

```
if (condition) {
  statement;
}
```

- Modeled as a flow chart.

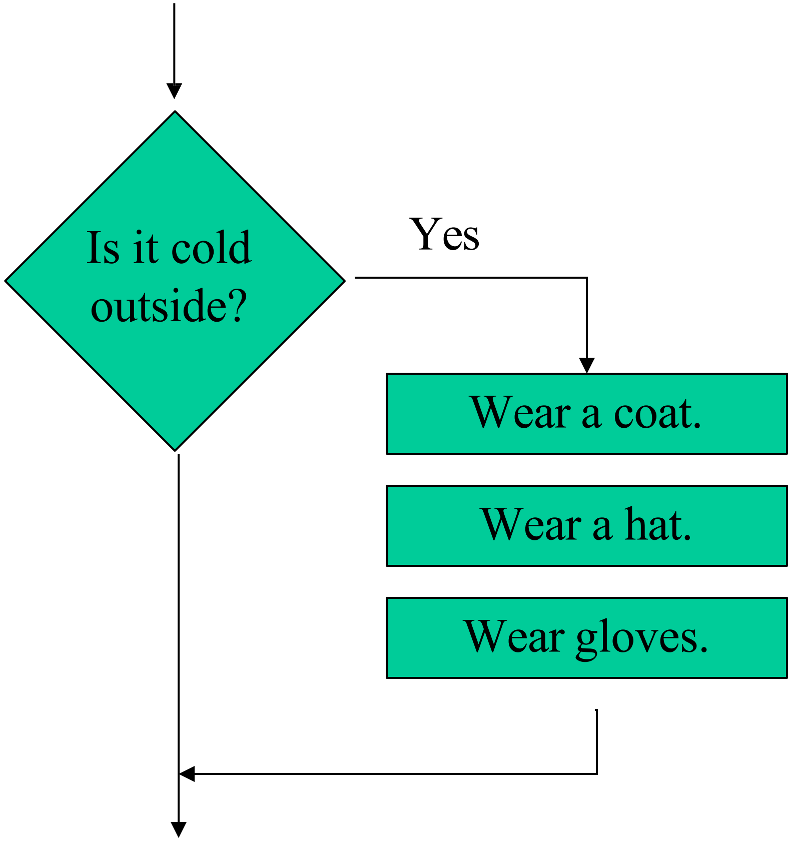


```
if (coldOutside){
   wearCoat();
}
```

The `if` Statement

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

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



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

Boolean expresion

- A *boolean expression* is any variable or calculation that results in a *true* or *false* condition.
- In most cases, the `boolean` expression, used by the `if` statement, uses *relational operators.*

Relational Operator	Boolean 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?

Note:
== vs. =
as in x = y

`if` Statements and Boolean Expressions

```
if  (x > 95)
   System.out.println("X is greater than 95");

if(x == 95){
   System.out.println("X is equal to 95");
}
```

It is recommended to use curly braces

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

if(total > MAX)
   System.out.println(“Found new max");
   MAX = total;
```

Only this statement is conditionally executed.

Example: AverageScore.java

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;
}
```

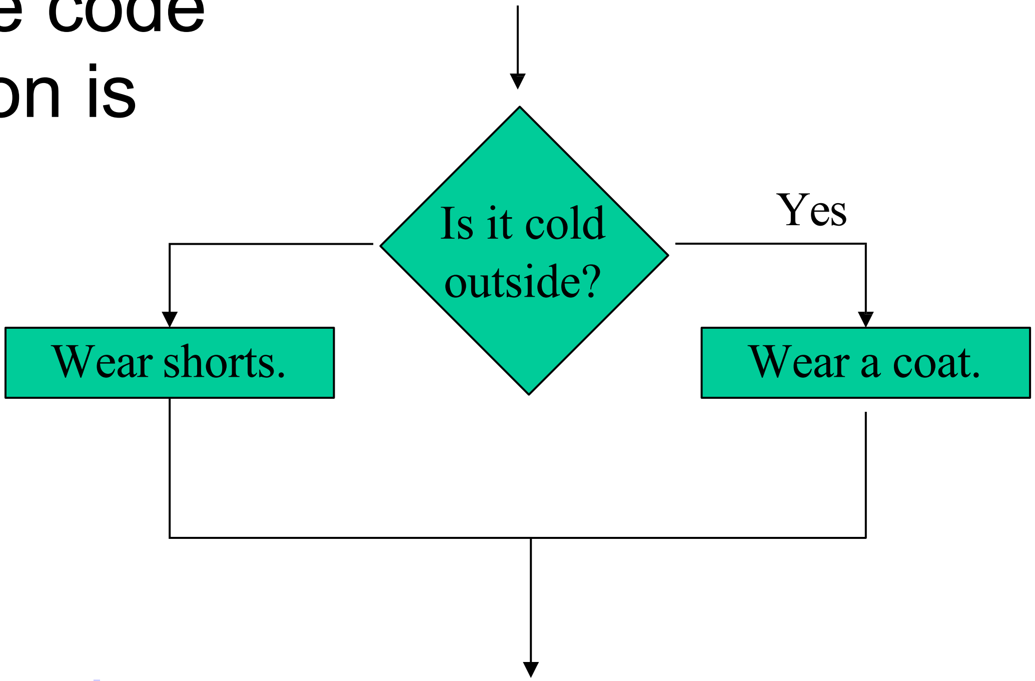
- Later, this condition can be tested:

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

if-else Statements

- Adds the ability to conditionally execute code when the if condition is false.

```
if (condition){
    statement; //true
}
else {
    statement; //false
}
```



- See example: Division.java

`if-else-if` Statements

- `if-else-if` statements can become very complex.
- Imagine the following decision set.

  if it is very cold, wear a heavy coat,

  else if it is chilly, wear a light jacket,

  else if it is windy wear a windbreaker,

  else if it is hot, wear no jacket.

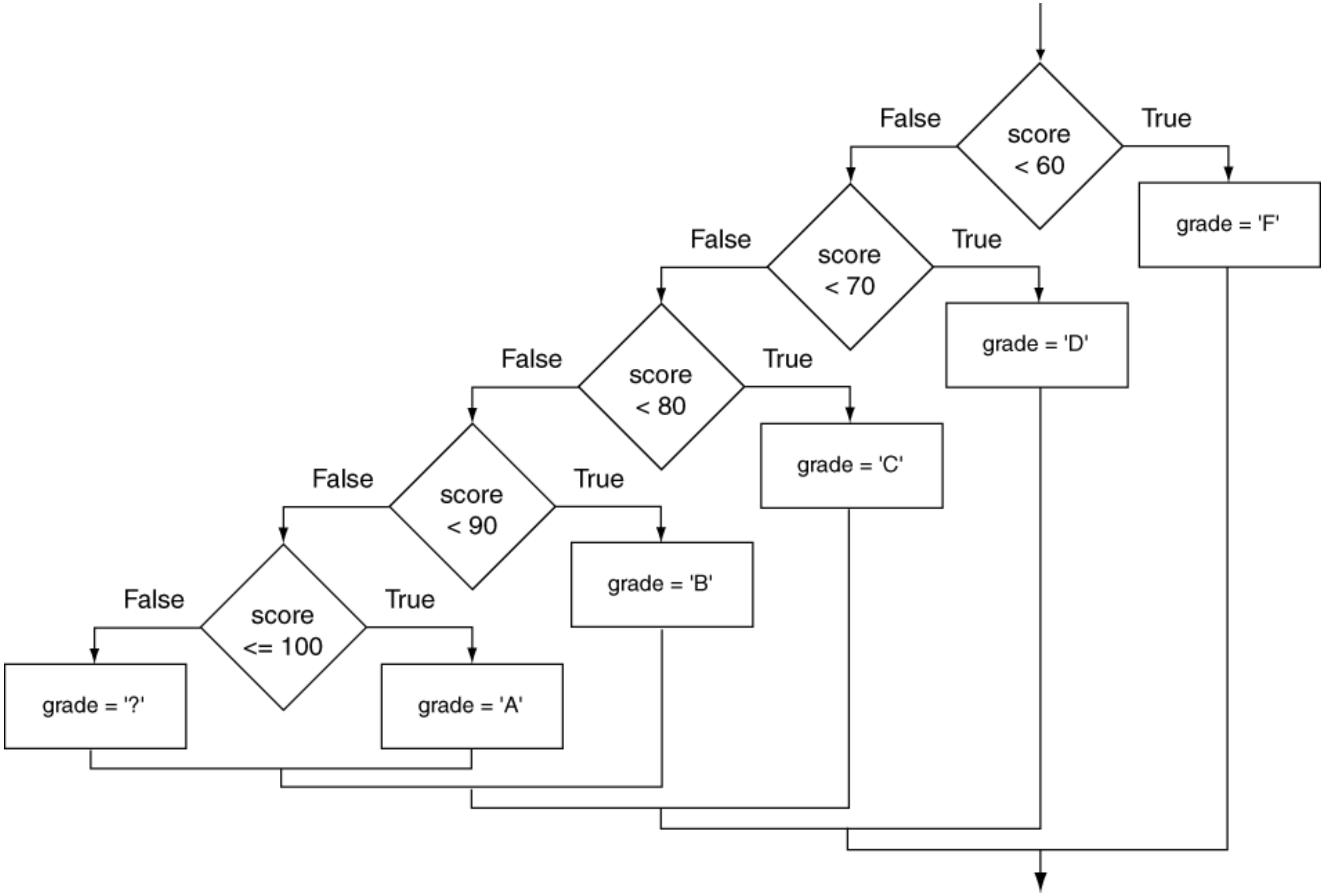
- See example: TestGrade.java, TestResults.java

`if-else-if` Statements

```
if (expression){
   statement or block;
}
else if (expression){;
     statement or block
     // Put as many else ifs as needed here
}
else {
     statement or block;
}
```

- Care must be used since `else` statements match up with the immediately preceding unmatched `if` statement.

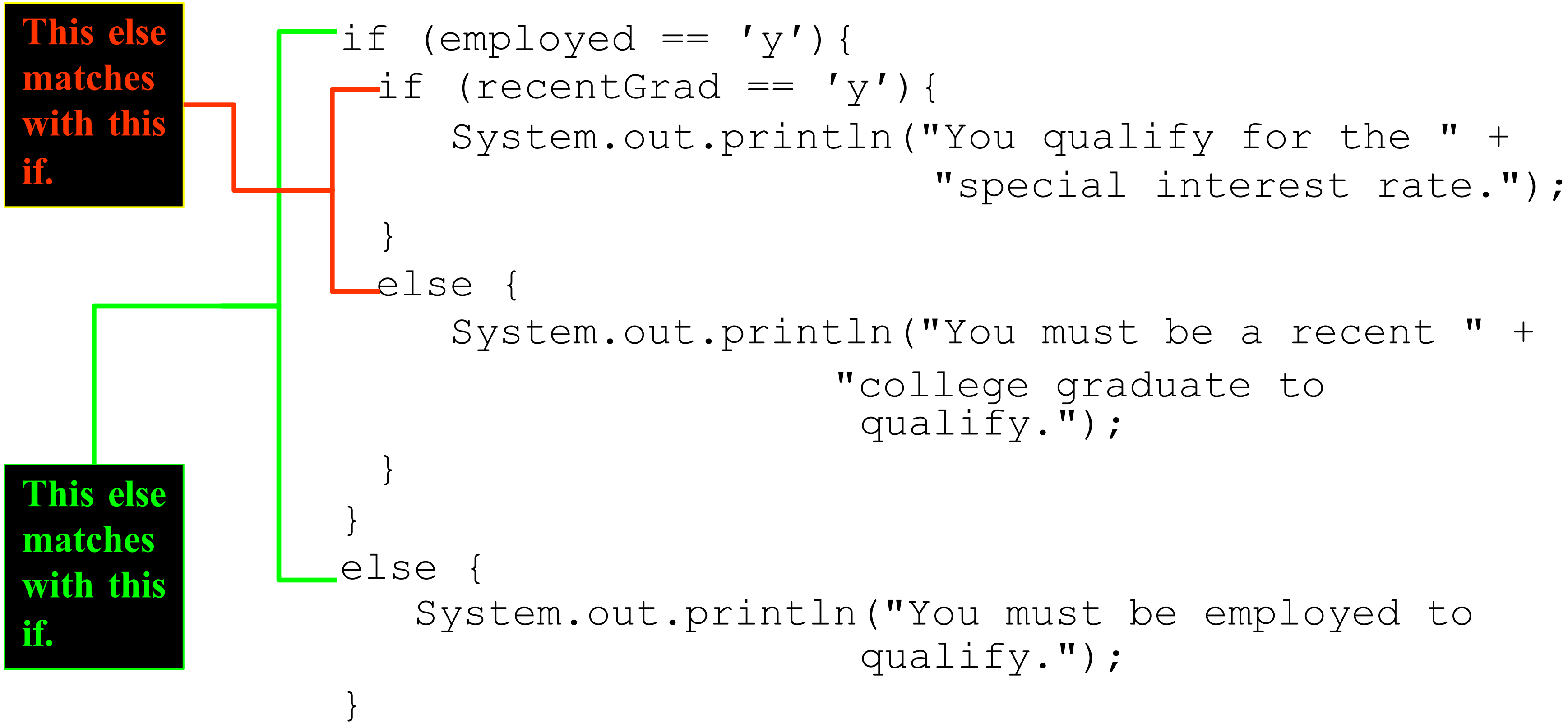
if-else-if Flowchart



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

Example



This else matches with this if.

This else matches with this if.

```
if (employed == 'y'){
  if (recentGrad == 'y'){
      System.out.println("You qualify for the " +
                                  "special interest rate.");
  }
  else {
      System.out.println("You must be a recent " +
                           "college graduate to
                            qualify.");
  }
}
else {
    System.out.println("You must be employed to
                            qualify.");
}
```

Code readability

- 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.

Logical Operators

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

Operator	Meaning	Example
&&	Logical AND	(x >= 1) && (x <= 100)
||	Logical OR	(x < 1) || (x > 100)
!	Logical NOT	!(x == 8)

The && Operator

- 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.

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

- See example: LogicalAnd.java

The || Operator

- 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

- 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

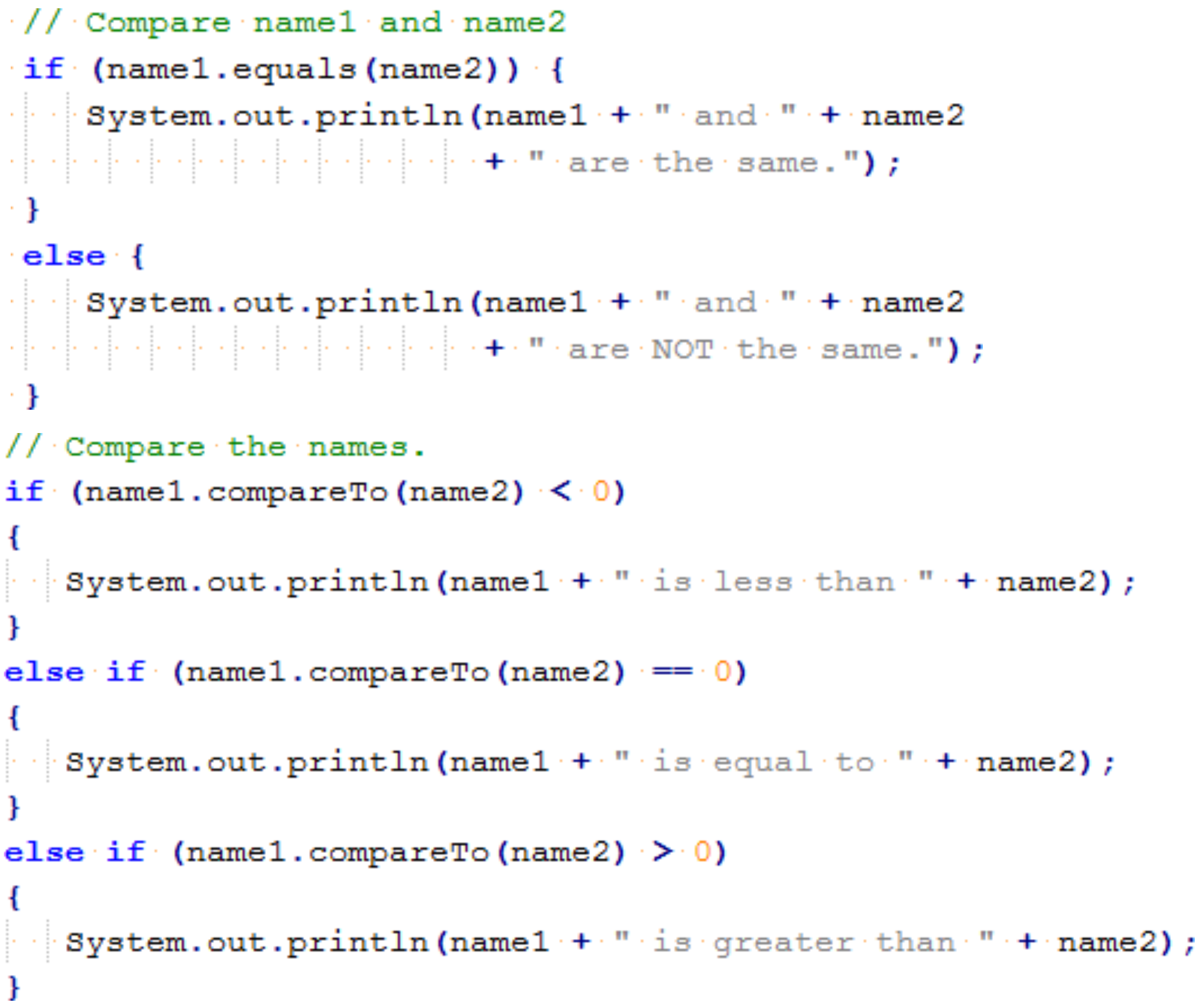
Operator Precedence

Operator 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.

Comparing `String` Objects

- 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.
- See example: GoodStringCompare.java
- See example: StringCompareTo.java

Comparing `String` Objects



```
 // Compare name1 and name2
 if (name1.equals(name2)) {
     System.out.println(name1 + " and " + name2
                          + " are the same.");
 }
 else {
     System.out.println(name1 + " and " + name2
                          + " are NOT the same.");
 }
// Compare the names.
if (name1.compareTo(name2) < 0)
{
    System.out.println(name1 + " is less than " + name2);
}
else if (name1.compareTo(name2) == 0)
{
    System.out.println(name1 + " is equal to " + name2);
}
else if (name1.compareTo(name2) > 0)
{
    System.out.println(name1 + " is greater than " + name2);
}
```

Ignoring Case in String Comparisons

- 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`
- See example: SecretWord.java

The `switch` Statement

- The `if-else` statement allows you to make 2 (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 or *character* type 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

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

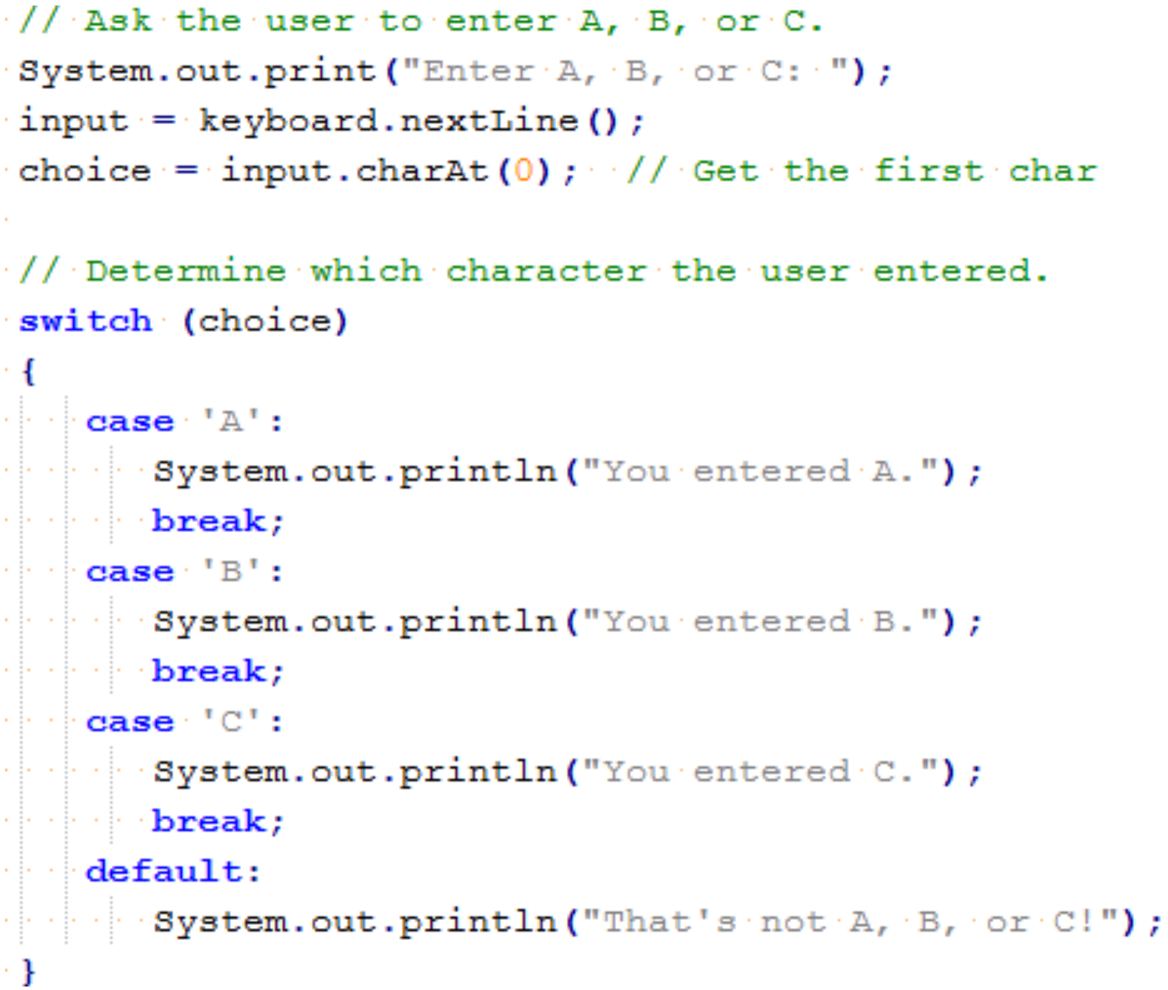
```
case CaseExpression:
     // place one or more statements here
     break;
```

- 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

Example - SwitchDemo.java



```java
// Ask the user to enter A, B, or C.
System.out.print("Enter A, B, or C: ");
input = keyboard.nextLine();
choice = input.charAt(0);  // Get the first char

// Determine which character the user entered.
switch (choice)
{
    case 'A':
        System.out.println("You entered A.");
        break;
    case 'B':
        System.out.println("You entered B.");
        break;
    case 'C':
        System.out.println("You entered C.");
        break;
    default:
        System.out.println("That's not A, B, or C!");
}
```

The `while` Loop

- The `while` loop has the form:

```
while(condition){
   statements;
}
```

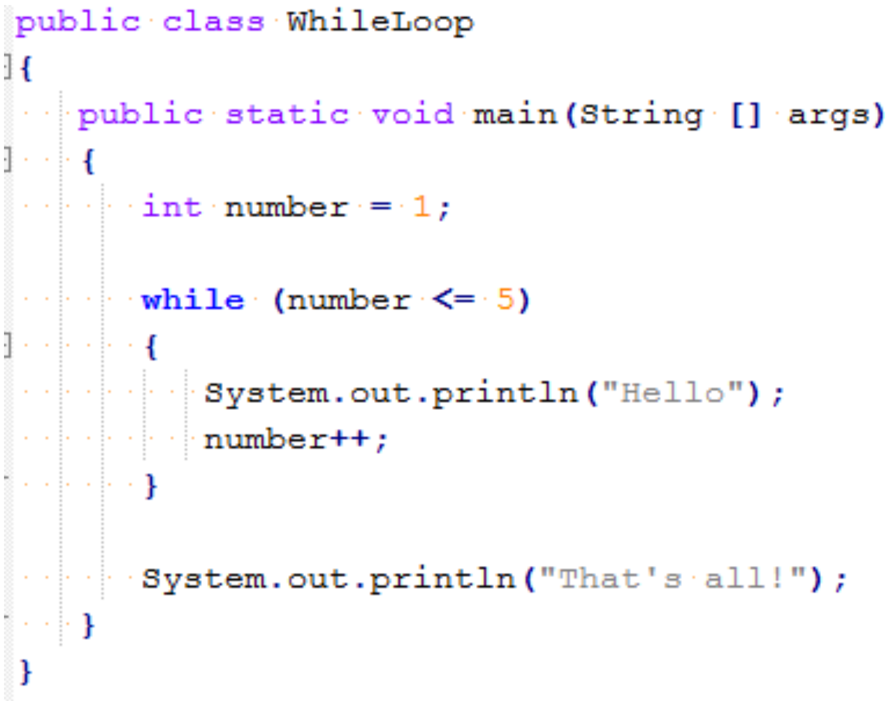
- While the condition is true, the statements will execute repeatedly.
- The `while` loop is a *pretest* loop, which means that it will test the value of the condition prior to executing the loop.

The `while` Loop

```
while(condition){
   statements;
}
```

- Care must be taken to set the condition to false somewhere in the loop so the loop will end.
- Loops that do not end are called *infinite loops.*
- A `while` loop executes 0 or more times. If the condition is false, the loop will not execute.
- Example: WhileLoop.java

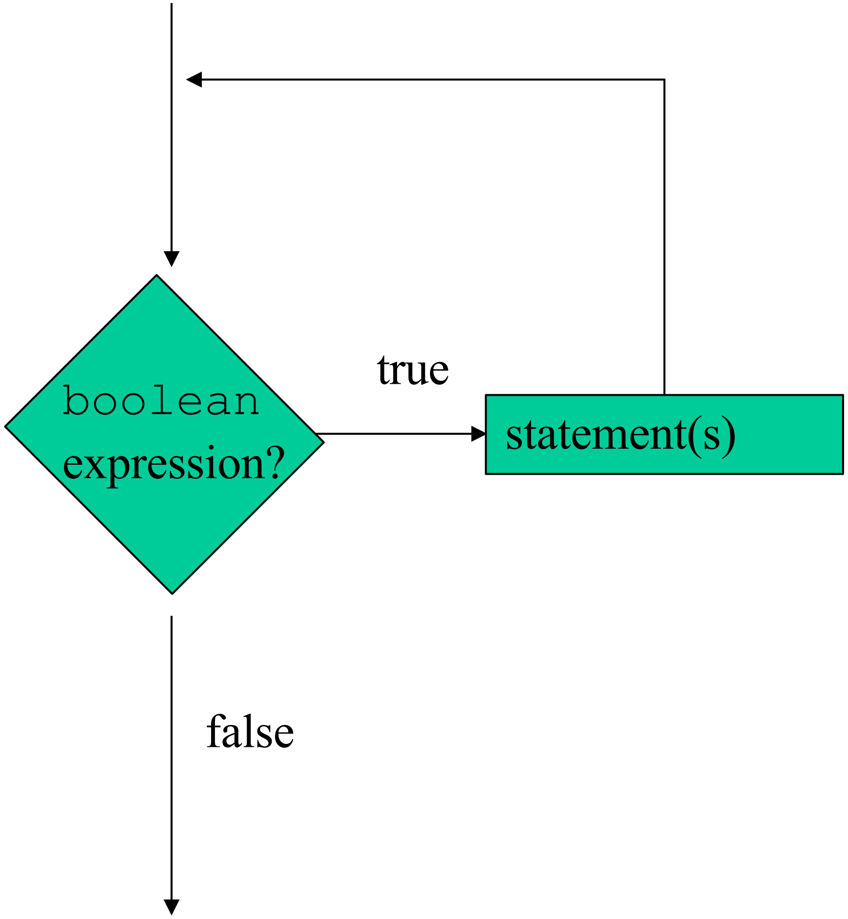
The while loop Flowchart



```
public class WhileLoop
{
    public static void main(String [] args)
    {
        int number = 1;

        while (number <= 5)
        {
            System.out.println("Hello");
            number++;
        }

        System.out.println("That's all!");
    }
}
```



boolean expression?

true

statement(s)

false

- What is the output?

Infinite Loops

- The following loop will not end:

```
int x = 20;
while(x > 0){
   System.out.println("x is greater than 0");
}
```

- The variable `x` never gets decremented so it will always be greater than 0.
- Adding the **`x--`** above to fix the problem.

```
int x = 20;
while(x > 0){
   System.out.println("x is greater than 0");
   x--;
}
```

The `while` Loop for Input Validation

- *Input validation* is the process of ensuring that user input is valid.

```
System.out.print("Enter a number in the " +
                 "range of 1 through 100: ");
number = keyboard.nextInt();
// Validate the input.
while (number < 1 || number > 100)
{
  System.out.println("That number is invalid.");
  System.out.print("Enter a number in the " +
                   "range of 1 through 100: ");
  number = keyboard.nextInt();
}
```

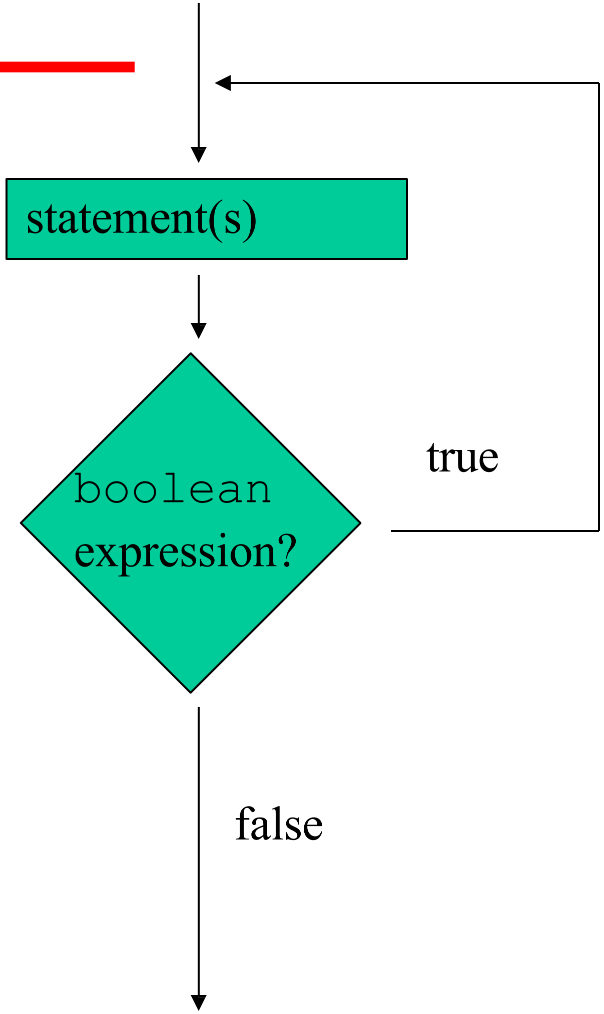
- Example: SoccerTeams.java

The `do-while` Loop

- The `do-while` loop is a *post-test* loop, which means it will execute the loop prior to testing the condition.
- The `do-while` loop (sometimes called called a `do` loop) takes the form:

```
do
{
  statement(s);
} while (condition);
```

- Example: TestAverage1.java

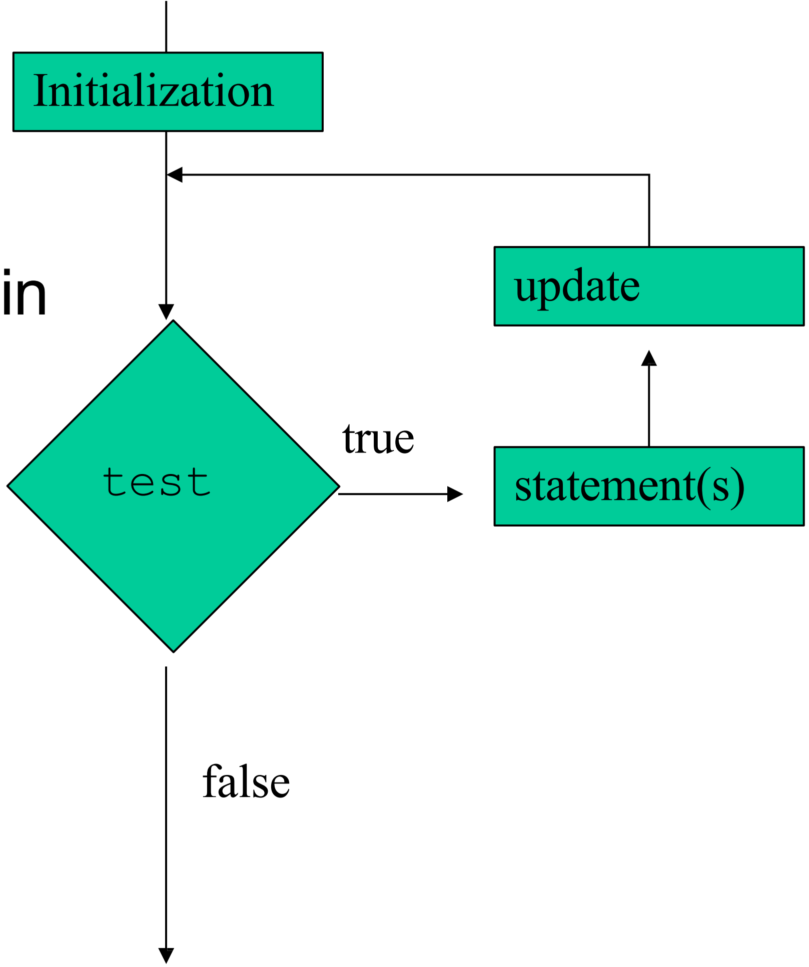


The `for` Loop

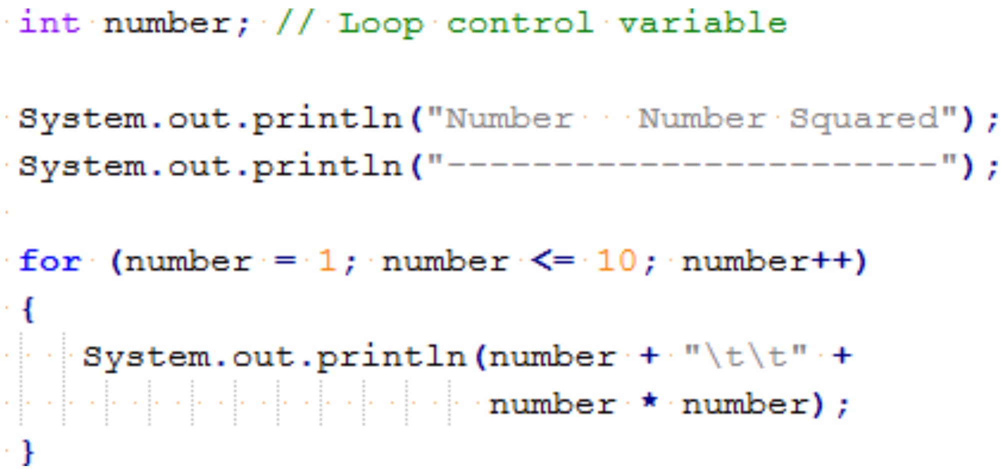
- The `for` loop is a pre-test loop.
- The `for` loop allows to initialize a control variable, test a condition, and modify the control variable all in one line of code.
- The `for` loop takes the form:

```
for(initialization; test; update)
{
    statement(s);
}
```

- See example: Squares.java



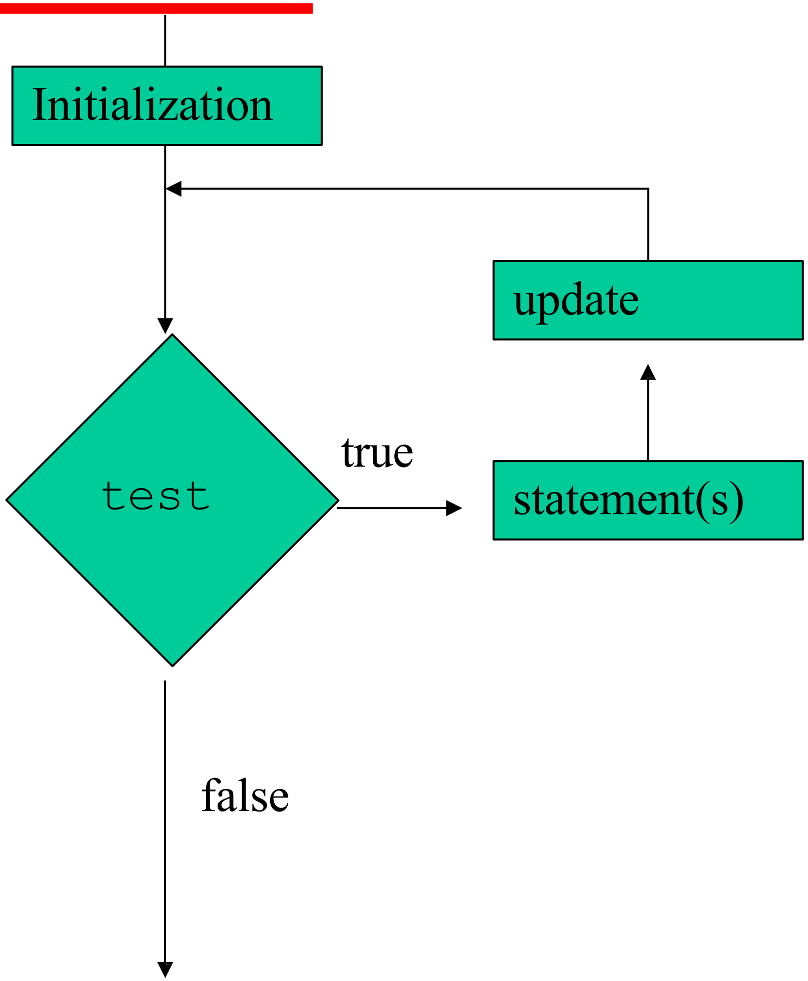
The `for` Loop



```
int number; // Loop control variable

System.out.println("Number    Number Squared");
System.out.println("-----------------------");

for (number = 1; number <= 10; number++)
{
    System.out.println(number + "\t\t" +
                       number * number);
}
```



- What is the output?

Modifying The Control Variable

- You should avoid updating the control variable of a `for` loop within the body of the loop!
- The update section should be used to update the control variable.
- Updating the control variable in the `for` loop body leads to hard to maintain code and difficult debugging.

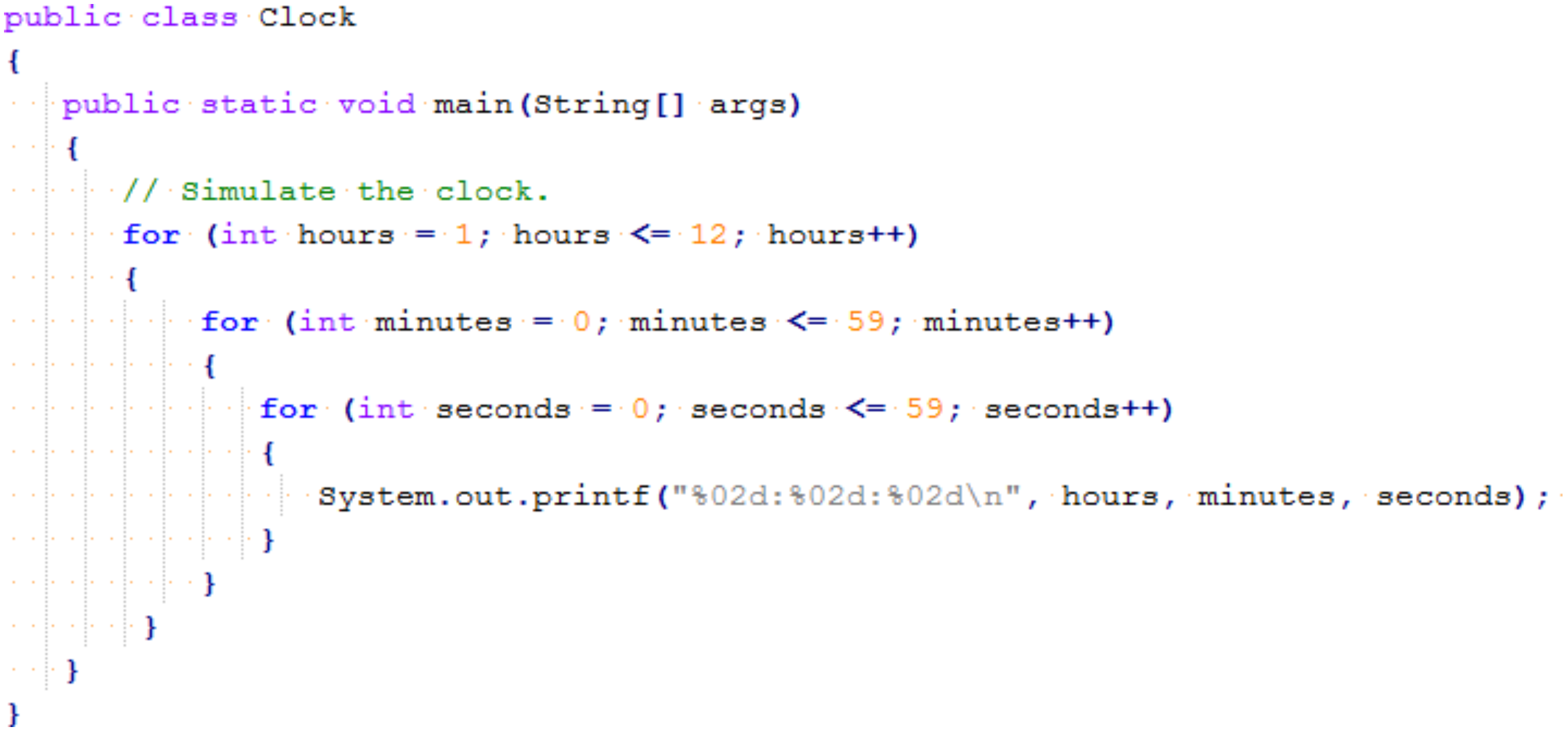
Nested Loops

- Like `if` statements, loops can be nested.
- If a loop is nested, the inner loop will execute all of its iterations for each time the outer loop executes once.

```
for(int i = 0; i < 10; i++)
    for(int j = 0; j < 10; j++)
          loop statements;
```

- The loop statements in this example will execute 100 times.
- Example: Clock.java

Nested Loops - Example



```java
public class Clock
{
   public static void main(String[] args)
   {
      // Simulate the clock.
      for (int hours = 1; hours <= 12; hours++)
      {
         for (int minutes = 0; minutes <= 59; minutes++)
         {
            for (int seconds = 0; seconds <= 59; seconds++)
            {
               System.out.printf("%02d:%02d:%02d\n", hours, minutes, seconds);
            }
         }
      }
   }
}
```

The `break` Statement

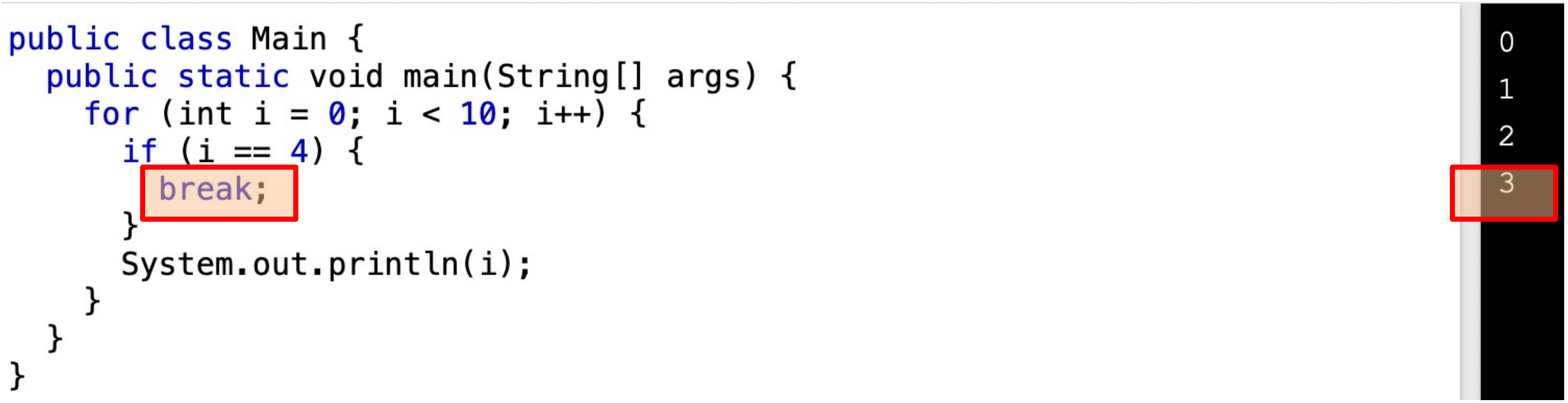
- The `break` statement can be used to terminate a **innermost** loop.

- It is considered bad form to use the `break` statement in this manner.

The continue Statement

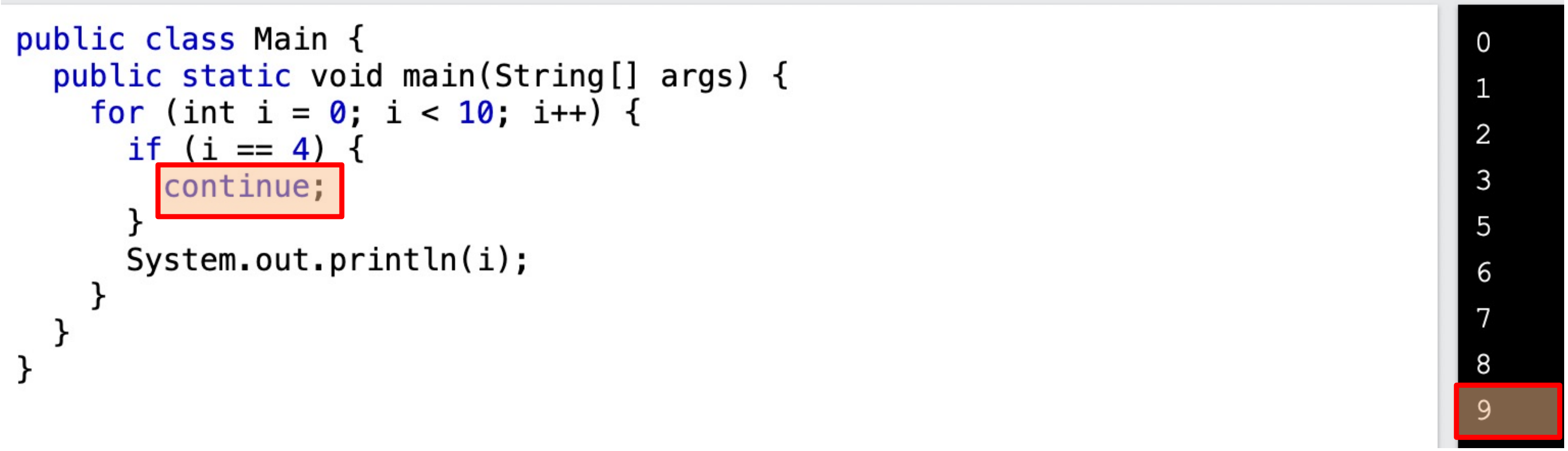
- The continue statement will skip the execution of current iteration of a loop.
- Like the break statement, the continue statement should be avoided because it makes the code hard to read and debug.

Example



```
public class Main {
  public static void main(String[] args) {
    for (int i = 0; i < 10; i++) {
      if (i == 4) {
        break;
      }
      System.out.println(i);
    }
  }
}
```

```
0
1
2
3
```



```
public class Main {
  public static void main(String[] args) {
    for (int i = 0; i < 10; i++) {
      if (i == 4) {
        continue;
      }
      System.out.println(i);
    }
  }
}
```

```
0
1
2
3
5
6
7
8
9
```

Deciding Which Loops to Use

- The `while` loop:

  Use it where you do not want the statements to execute if the condition is false in the beginning.

- The `do-while` loop:
  - Use it where you want the statements to execute at least one time.
- The `for` loop:
  - Use it where there is some type of counting variable that can be evaluated.

Debugging in Java Netbeans

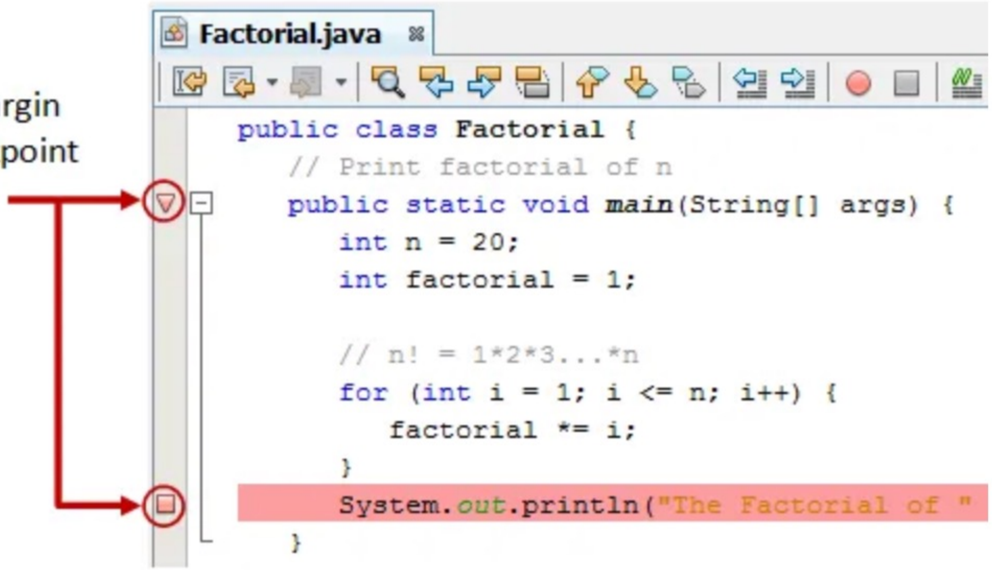
- **Debugging** is the process of finding and resolving defects or problems within a computer program that prevent correct operation of computer software or a system.

- **Debugging Terminology**
    - Breakpoint: a line of code where you want to "pause" the execution of a program
    - Continue: will continue the execution of the program until the next breakpoint or until the program terminates

Set an initial Breakpoint

- Before starting the debugger, you need to set at least one breakpoint to suspend the execution inside the program
- Set a breakpoint by by clicking on the left-margin of the line or selecting “breakpoint - Toggle line Breakpoint”

Click the margin to set Breakpoint

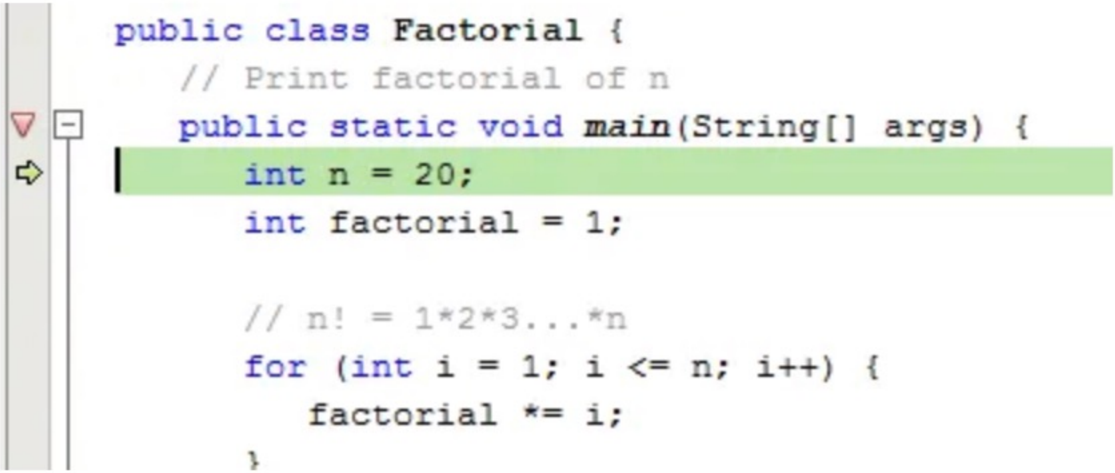


```
public class Factorial {
   // Print factorial of n
   public static void main(String[] args) {
      int n = 20;
      int factorial = 1;

      // n! = 1*2*3...*n
      for (int i = 1; i <= n; i++) {
         factorial *= i;
      }
      System.out.println("The Factorial of "
   }
```

Start Debugging

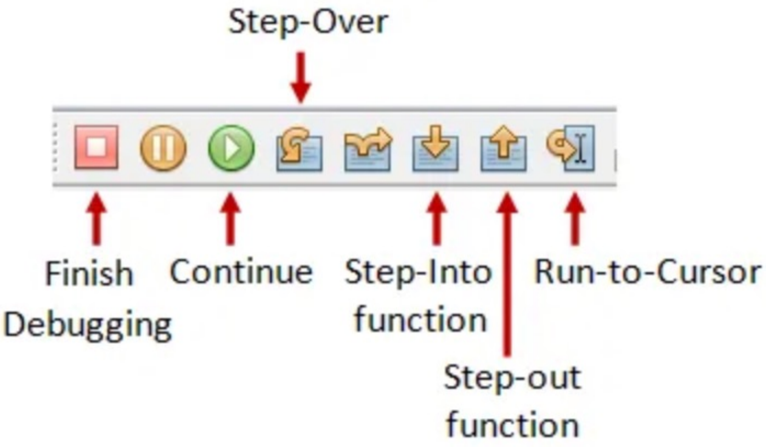
- Right click anywhere on the source code ⇒ "Debug File". The program begins execution but suspends its operation at the breakpoint.
- The highlighted line (also pointed to by a green arrow) indicates the statement to be executed in the next step.



```
public class Factorial {
   // Print factorial of n
   public static void main(String[] args) {
      int n = 20;
      int factorial = 1;

      // n! = 1*2*3...*n
      for (int i = 1; i <= n; i++) {
         factorial *= i;
      }
```

Some buttons for debugging

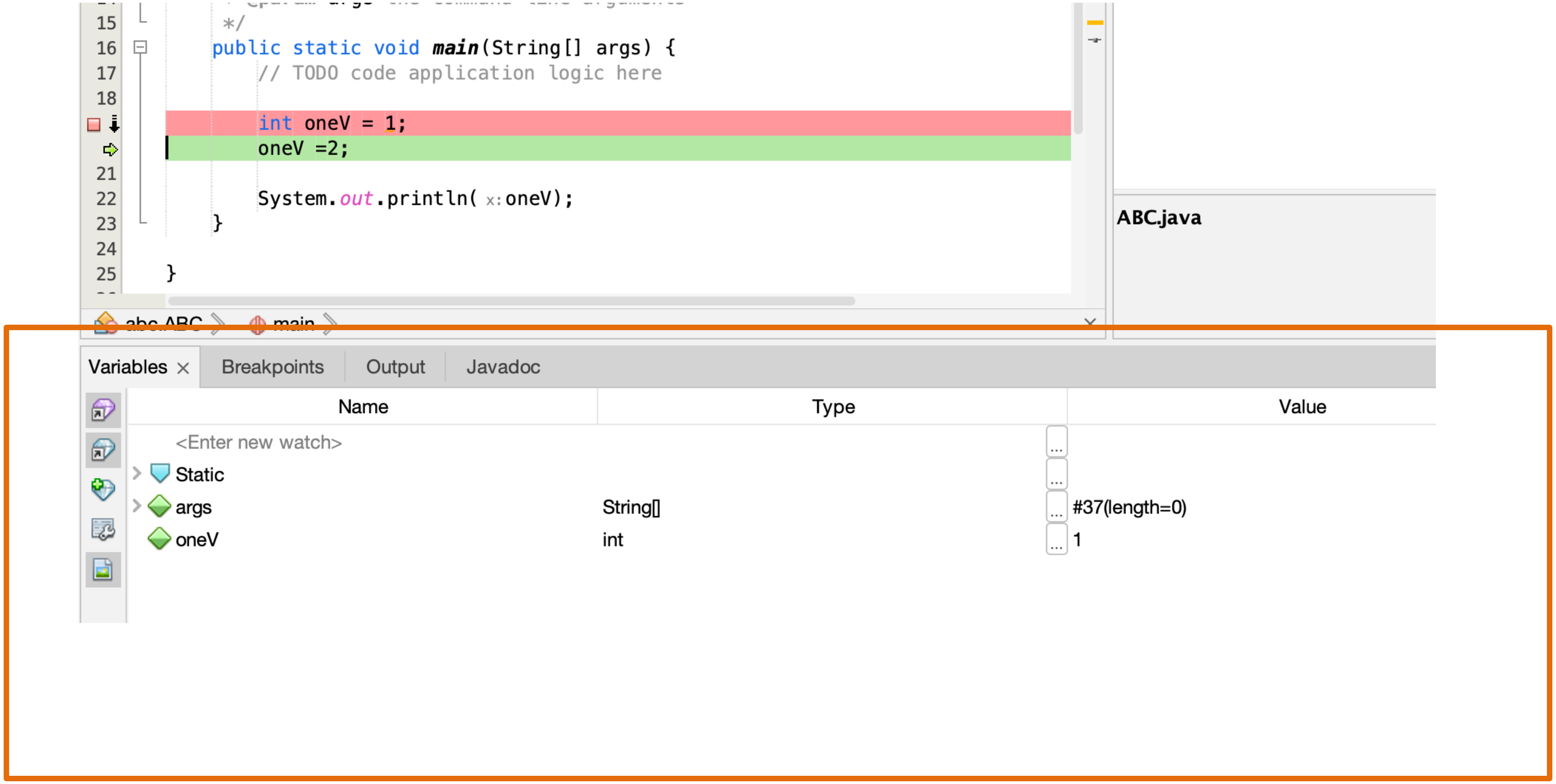


panel for debug

- Step Over button means executing the program by one step.
- "Continue" executes the program execution, up to the next breakpoint, or the end of the program.
- Step-into-function means stepping into a function and Step-out-function means going out from that function.
- *Run to Cursor* allows you to select a line of code where you want execution of the program to pause

Watching variables

We can observe variables with its varying values during the debugging process.



```
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here

        int oneV = 1;
        oneV =2;

        System.out.println( x:oneV);
    }

}
```

Name	Type	Value
<Enter new watch>		
Static		
args	String[]	#37(length=0)
oneV	int	1

Summary

- We have covered:
  - The `if`, `if-else` Statement
  - Logical Operators
  - Comparing `String` Objects
  - The `switch` Statement
  - The `while, do-while, for` Loop
  - Debugging programs in java NetBeans IDE