Lecture - Control Flows

Overview

Expressions, statements and blocks are the most essential elements in the programming languages. It provides you with tools to dynamically respond to the data and inputs your program operates on. Expressions are like formula that evaluates to a single value. Statements are instructions that tells the program to perform certain actions. Blocks are group of statements enclosed in curly braces {}. Flow control in Java includes concepts, such as branching and looping statements. Because most branching and looping statements are controlled by Boolean expressions, we will also discuss the Java type *boolean* in more detail.

In this part of the lecture, you will learn about:

1. Expressions, statements and blocks.
2. Flow Controls
    1. branching mechanisms
    2. loops, and
3. debugging.

Expressions, Statements and Blocks

We have already learnt about variables in Lesson 1 and operators earlier in this part of lecture. We briefly discuss what the expressions and statements are in context of the Java language.

Expression

In Java, an expression is a construct made up of variables (fields), operators, and method invocations that combine to evaluate to a single value. It's like a mathematical equation or formula, but within the context of the Java programming language. Expressions are fundamental building blocks for writing Java code. They are used in various contexts, such as:

- Assigning values to variables
- Performing calculations
- Making decisions in conditional statements
- Passing arguments to methods
- Returning values from methods
- Understanding how expressions work is crucial for writing correct and efficient Java code.

Examples of expressions in Java:

```
x + y * z (arithmetic expression)
name.length() (method invocation expression)
age >= 18 (comparison expression)
isStudent && hasScholarship (logical expression)
```

Statements

A statement in Java is a complete instruction that directs the program to perform a specific action. It's like a sentence that tells the computer what to do. Unlike expressions, which always evaluate to a single value, statements don't necessarily produce a value but instead execute an action or change the program's state.

Here are some key characteristics of statements in Java:

Types: There are various types of statements in Java, each serving a specific purpose:

- Declaration: Defining variables or methods (e.g., `int age = 25;, public void sayHello() {...}`)
- Assignment: Assigning values to variables (e.g., `name = "John";`)
- Control flow: Statements controlling the program flow (`e.g., if (x > 0) { ... }, for (int i = 0; i < 10; i++) { ... }`)

- Method invocation: Calling functions within your code or external libraries (`e.g., System.out.println("Hello");`)
- Return: Statements returning values from methods (`e.g., return x;`)

Execution: Java code is executed line by line, meaning statements are processed sequentially according to their order in the program.

Scope: Each statement exists within a specific scope, determining where it can be accessed and used.

Examples of statements in Java:

```
int x = 10;
if (age > 18) {
         System.out.println("You are eligible to vote.");
}
String greeting = sayHello("World");
```

Statements are essential for achieving various tasks within your Java program. They form the core of your code, building logic, performing actions, and controlling the program's flow. By understanding different types of statements and their proper usage, you can create effective and well-structured Java applications.

Blocks

A block is a group of zero or more statements between balanced braces and can be used anywhere a single statement is allowed. The following example, illustrates the use of blocks:

```
class BlockDemo {
     public static void main(String[] args) {
          boolean condition = true;
          if (condition) { // begin block 1
               System.out.println("Condition is true.");
          } // end block one
          else { // begin block 2
               System.out.println("Condition is false.");
          } // end block 2
     }
}
```

Control Flow

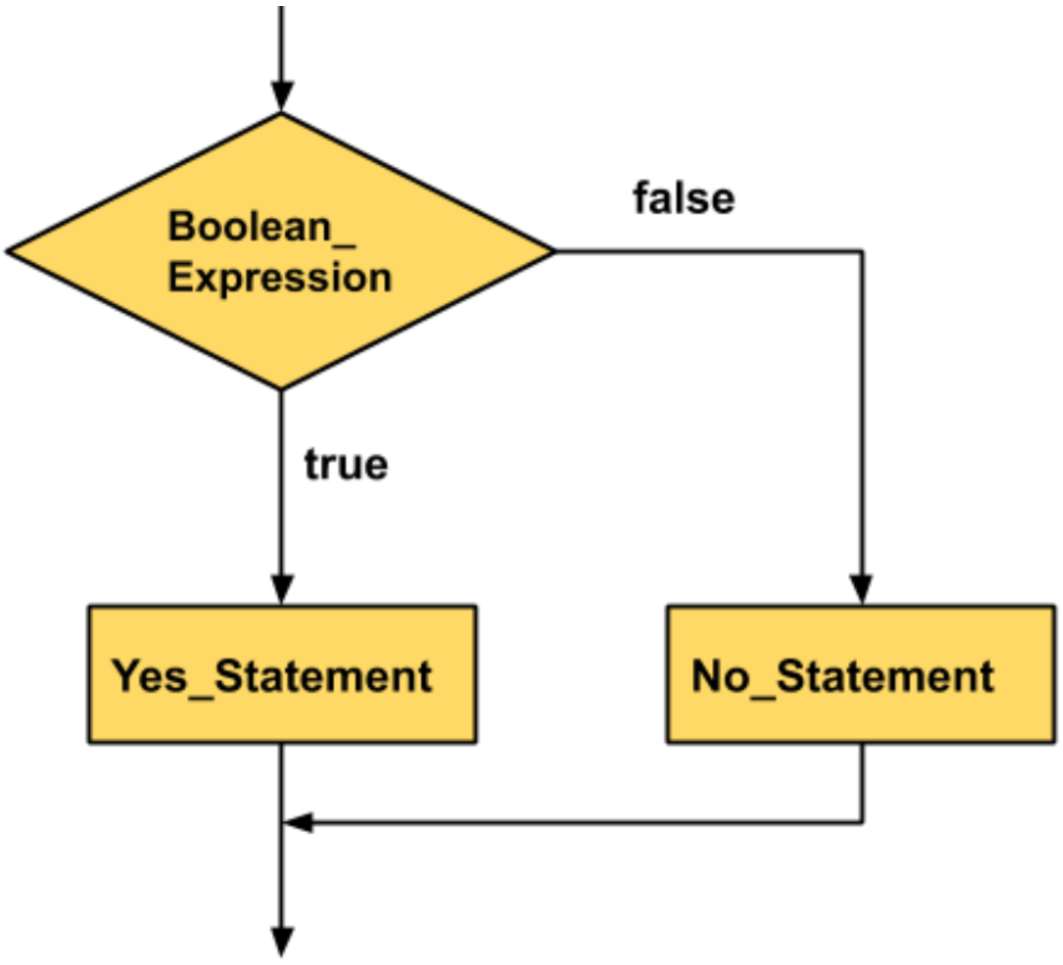
As in most programming languages, flow control in Java refers to its branching and looping mechanisms. Java has several branching mechanisms:

- `if`-`else`,
- `if`,
- and `switch` statements

(We'll learn about `throw` in the second half of the course.) Most branching statements are controlled by Boolean expressions. A Boolean expression evaluates to either *true* or *false*. The primitive type *boolean* only takes the values *true* or *false*.

if-else Statements

An if-else statement chooses between two alternative statements based on the value of a Boolean expression.



In Java, boolean expressions in `if` and loop statements must be enclosed in parentheses. It is good style to enclose them in parentheses in other contexts too. If the boolean expression of an `if` evaluates to *true,* the Yes_Statement is executed. If it evaluates to *false,* the No_Statement is executed. Each Yes_Statement and No_Statement branch of an if-else can be made up of a **single statement** or a **compound statement**. A compound statement is a list of statements enclosed by a pair of braces ({}). The statements inside the braces can themselves be single statements, variable declarations, compound statements, `if` statements or loops.

Advanced: Essentially any valid Java code can occur inside a compound statement, including new class declarations.

Each statement ends with a semicolon (;), but the close brace has an implicit semicolon. If there is an "empty" statement after the `if` but before the `else` caused by an explicit semicolon after the brace, the compiler will complain.

```
class Main {
    static public void main (String[] args) {
        int a;
        if (true) {
            a = 0; ]
        }
        else {
            a = 1;
        }
    }
}
```

If you have have more than one statement, you need the braket.

By convention, the commands within the Yes_Statement and No_Statement are indented. The amount of indentation should be consistent throughout the code. It can either be a fixed number of spaces (2, 4, 8 are common) or a tab stop. Spaces are more common. ()

In Python and Fortran, the leading whitespace affects the meaning of the code; in most languages, including Java, it does not.

There are multiple conventions for where to put the braces. In this course, we will put the opening brace on the same line as the `if`, and the closing brace aligned with the `if`. This is also recommended by the original developers of Java language.

If the branch of the `if` is just a single statement you can omit the pair of braces. It is, however, good coding practice to always include them.

Exercise: Why? What coding errors can be avoided by using the braces?

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

        boolean iLoveJava = true;
        int hoursSpentCodingJava = 10;
        if (iLoveJava) {
            System.out.println("Everyone loves Java!");
        } else {
            System.out.println("You will get there!");
        }

        if (hoursSpentCodingJava > 10000) {
            System.out.println("Java L33T!");
        } else
            System.out.println("Practice leads to mastery!");
    }
}
```

Everyone loves Java!
Practice leads to mastery!

Exercise: Change the constants at the start of this method to see the other branches being executed.

Omitting the *else* Part

The *else* part of an *if-else* statement may be omitted to obtain what is often only called an **if statement**. If the Boolean_Expression is true, then the Action_Statement is executed. Otherwise, the Action_Statement is not executed and the program moves on to the next statement.

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

        int numberOfSales = 101;
        int salary = 100000;
        int bonus = 10000;

        if (numberOfSales > 100) {
            salary += bonus;
            System.out.println("Congratulations! You received a bonus!");
        }

        System.out.println("Your salary this year: " + salary);

    }
}
```

Congradulations! You received a bonus!
Your salary this year: 110000

Nested Statements

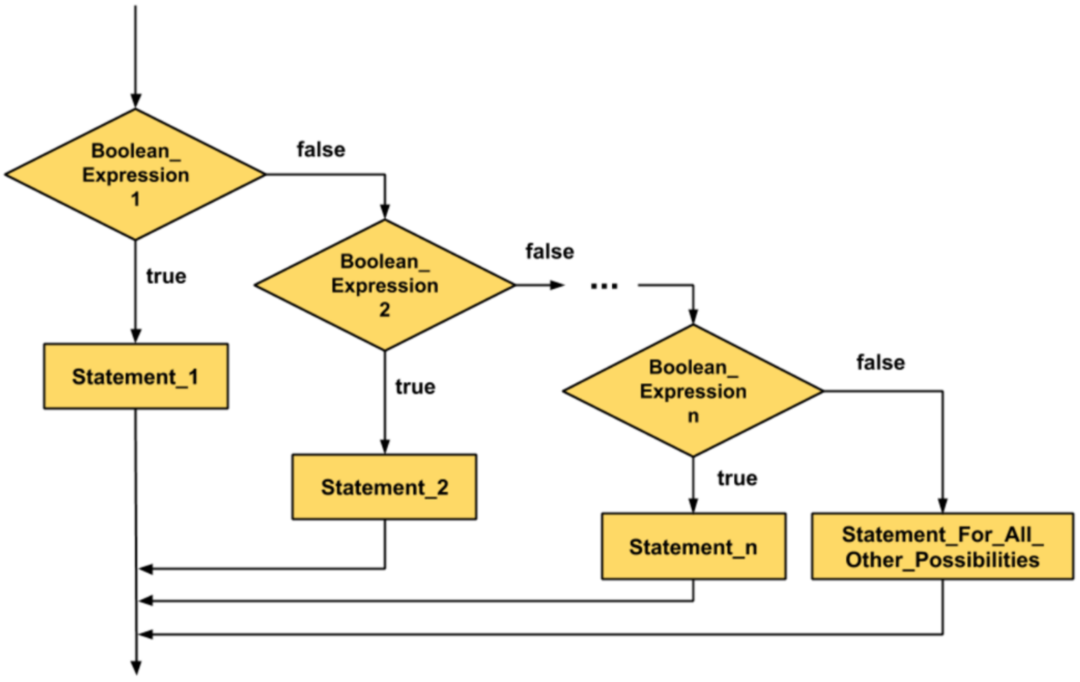
if-else and *if* statements can contain any single or compound statements, including other *if-else* and *if* statements. In this case, we speak of **nested statements**. Each level of a nested *if-else* or *if* should be indented further than the previous level to make your code readable:

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

        int income = 30000;
        int numberOfKids = 2;
        double tax = 0;

        if (income > 18201) {
            tax = income * 0.19;

            if (numberOfKids>0) {
                tax = tax - 500; //tax relief
            }
        }

        System.out.printf("You owe $%.2f in taxes.", tax);

    }
}
```

You owe $5200.00 in taxes

Multiway *if-else* Statements

The multiway if-else statement is simply a normal *if-else* statement that nests another *if-else* statement at every *else* branch. The Boolean_Expressions are evaluated in the order they are written until one evaluates to *true*. The final *else* is optional.



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

        int numberOfSales = 75;
        int salary = 100000;
        int smallBonus = 2000;
        int mediumBonus = 5000;
        int bigFatBonus = 10000;

        if(numberOfSales > 200) {
            salary += bigFatBonus;
        } else if(numberOfSales > 100) {
            salary += mediumBonus;
        } else if(numberOfSales > 50) {
            salary += smallBonus;
        } else {
            System.out.println("No bonus this year. :(");
        }

        System.out.println("Your salary this year: " + salary);

    }
}
```

Your salary this year: 102000

The *switch* Statement

The *switch* statement is the only other kind of Java statement that implements multiway branching. When a *switch* statement is evaluated, one of a number of different branches is executed. The choice of which branch to execute is determined by a controlling expression enclosed in parentheses after the keyword *switch.*

```
switch (Controlling_Expression)
{
  case Case_Label_1:
          Statement_Sequence_1
          break;
  case Case_Label_2:
          Statement_Sequence_2
          break;
  case Case_Label_n:
          Statement_Sequence_n
          break;
  default:
}
```

⚠ Beware that the controlling expression must evaluate to a `char`, `int`, `short`, `byte` or `String` (or an enumerated type, which we will see later). Note that it cannot be a `long`.

Each branch statement in a switch statement starts with the reserved word `case`, followed by a *constant* called a **case label**, followed by a colon, and then a sequence of statements. Each case label must be of the same type as the controlling expression. Case labels need not be listed in order or span a complete interval, but each one may appear only once.

The optional **default label** is usually used last. If no case label matches, then the only statements executed are those following the `default` label (if there is one). Even if the case labels cover all possible outcomes in a given `switch` statement, it is still a good practice to include a `default` section. In this case, it can, for example, be used to output an error message.

Each sequence of statements should be followed by a **break statement**, which tells the program execution to continue after the `switch` block. If you don't add a `break` statement, the next *case* label will be evaluated. The switch statement ends when it executes a `break` statement, or when the end of the switch statement is reached.

```
import java.util.Scanner;

public class DayOfWeek {

    public static void main (String[] args) {

        Scanner keyboard = new Scanner(System.in);

        System.out.println("Enter day of the week (1=Monday, 7=Sunday):");
        int dayOfWeek = keyboard.nextInt();

        switch (dayOfWeek) {
            case 1:
                System.out.println("Today is Monday");
                break;

            case 2:
                System.out.println("Today is Tuesday");
                break;   do until it find the break.

            case 3:
                System.out.println("Today is Wednesday");
                break;

            case 4:
                System.out.println("Today is Thursday");
                break;

            case 5:
                System.out.println("Today is Friday");
                break; //see what happens if you remove this break statement

            default:   Good to have default.
                System.out.println("This is not a work day");
                break;
        }
    }
}
```

If the `break` statement is omitted inadvertently, the compiler will **not** issue an error message.

This is to be like a C switch statement. C was developed when saving a byte of memory was important, and this "fall-through" behaviour sometimes allowed more efficient code.

Exercise: Remove the break statement(s) from one or more of the cases above, and observe how it also runs the code from the following case.

Advanced: The only time it is appropriate not to have a break statement is if there are multiple cases that should run the same code, as in the following example. Note that the "fall-through" only applies when there is no code. Whenever there is code, there is a `break`.

Example

```
class Main {
    static public void main (String[] args) {
        int a = 1;
        switch (a) {
            case 1:
            case 3:
            case 5:
            case 7:
            case 9:
                System.out.println("odd, single digit");
                break;
            default :
                System.out.println("even or multi-digit");
                break;
        }
    }
}
```

In this case run the same code

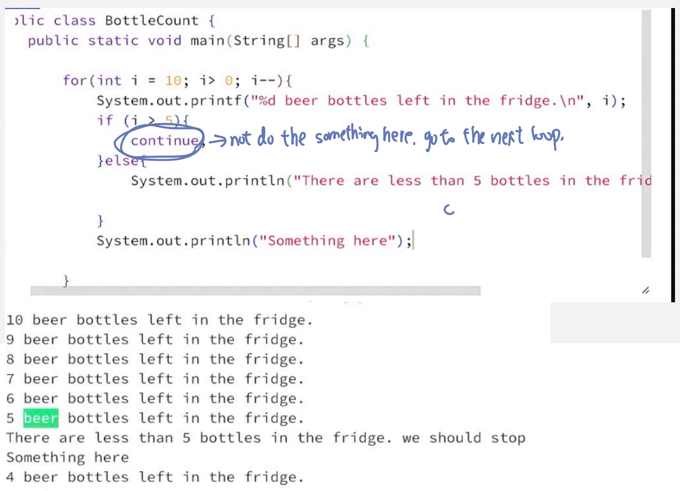
odd, single digit

Continue Keyword

Sometimes it is useful to force an early iteration of a loop. That is, you might want to continue running the loop but stop processing the remainder of the code in its body for this particular iteration. Continue statement performs such a action. Example below demonstrates it's usage.

```
// Demonstration of Continue keyword.
class ContinueDemo{
    public static void main(String[] args){
        for(int i=0; i<10; i++){
            System.out.println(i + " ");
            if(i%2 == 0) continue;
            System.out.println("");
        }
    }
}
```

0 1
2 3
4 5
6 7
8 9



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

      for(int i = 10; i> 0; i--){
          System.out.println("%d beer bottles left in the fridge.\n", i);
          if (i > 5){
              continue;
          }else{
              System.out.println("There are less than 5 bottles in the frid

          }
          System.out.println("Something here");

    }
```

→not do the something here. go to the next loop.

```
10 beer bottles left in the fridge.
9 beer bottles left in the fridge.
8 beer bottles left in the fridge.
7 beer bottles left in the fridge.
6 beer bottles left in the fridge.
5 beer bottles left in the fridge.
There are less than 5 bottles in the fridge. we should stop
Something here
4 beer bottles left in the fridge.
```

The Conditional Operator

We have covered it briefly in previous lesson. Here we will discuss it's usage with branching in Java. The **conditional operator** is a variant on certain forms of the *if-else* statement. It is also referred to as the **ternary operator** or **arithmetic if**:

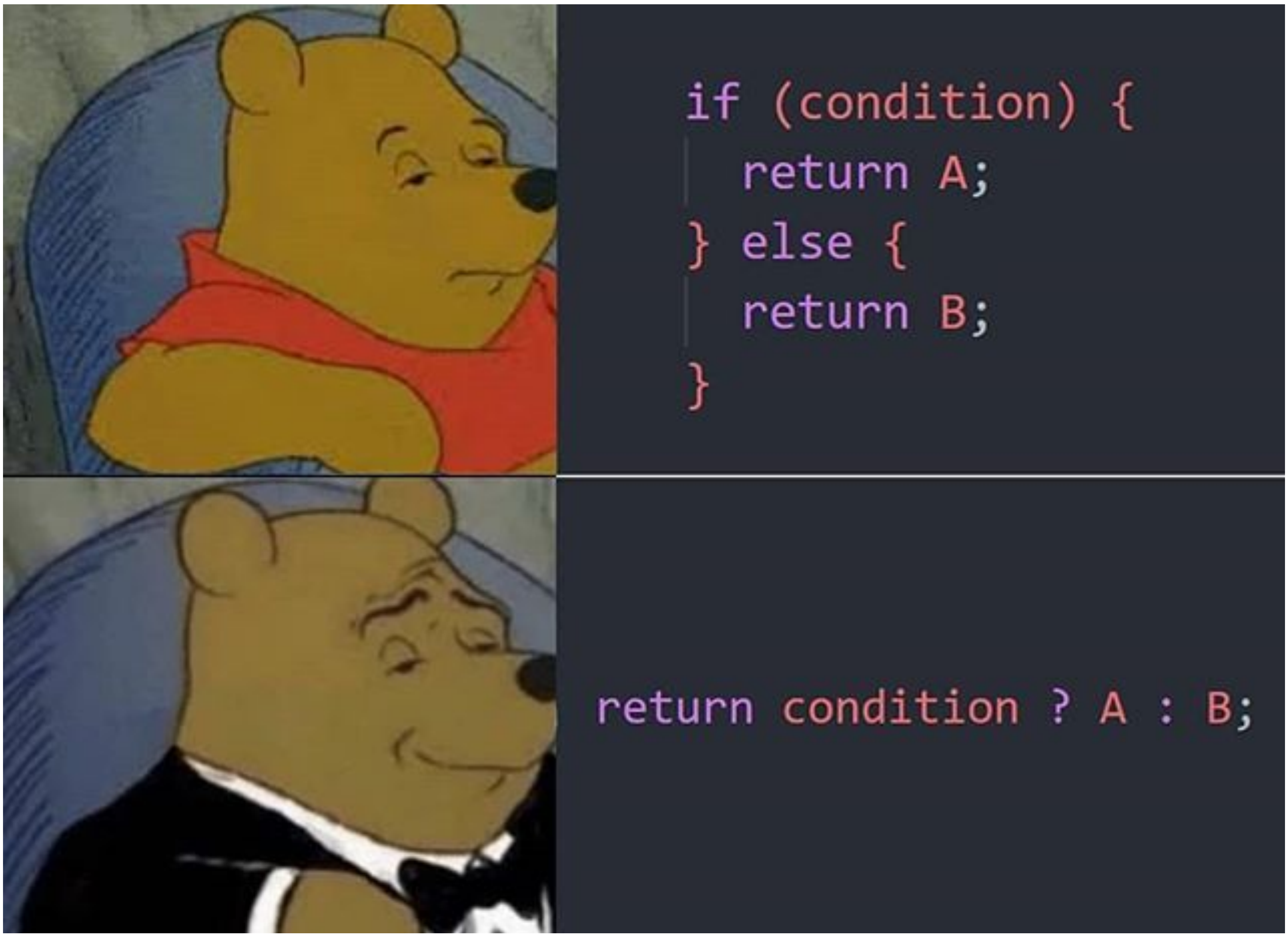
```
if(n1>n2)
     max=n1;
```

```
else
    max = n2;
```

Using the ternary operator is equivalent to the example above:

```
max = (n1 > n2) ? n1 : n2;
```

The expression to the right of the assignment operator is a **conditional operator expression**. If the Boolean expression is *true*, the expression evaluates to the value of the first expression (n1). Otherwise, it evaluates to the value of the second expression (n2).



```
if (condition) {
    return A;
} else {
    return B;
}
```

```
return condition ? A : B;
```

Note that this is not just a difference in notation. The ternary operator can be used where statements are not allowed, such as in a method argument. For example, change the following code to use `if` and `else`.

```
class Main {
    public static void main(String[] args) {
        double a = 3;
        System.out.println(Math.sqrt(Math.sin(a > 0 ? a : -a)));
    }
}
```

0.375659...

Notice that you either had to introduce a new variable or had to copy a lot of code. Duplicate code is harder to maintain, because it is easy to forget to update one of the copies.

Loops

Loops in Java are similar to those in other high-level languages. Java has three types of loop statements:

- `while`
- `do`-while,
- and `for` statements

The code that is repeated in a loop is called the **body** of the loop. Each repetition of the loop body is called an **iteration** of the loop.

The *while* Statement

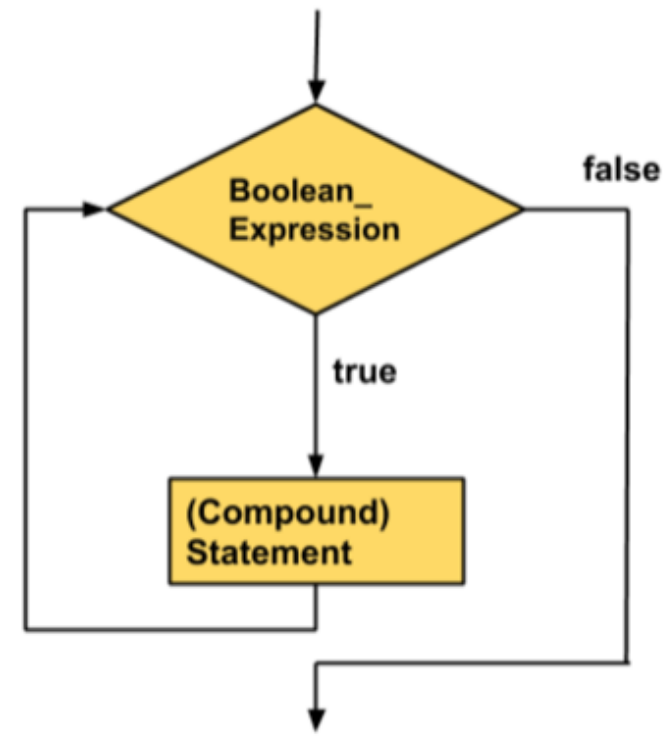
A *while* statement is used to repeat a portion of code (*i.e.*, the loop body) based on the evaluation of a Boolean expression. This Boolean expression is checked *before* the loop body is executed. If the expression evaluates to *false*, the loop body is not executed and control passes to the next statement after the loop. Before the execution of each following iteration of the loop body, the Boolean expression is checked again. If it still evaluates to *true*, the loop is executed again. If it evaluates to *false*, the loop statement ends.

```
while (Boolean_Expression)
    Statement
```

In case the loop body has multiple statements (a compound statement) these need to be enclosed in a pair of braces {}.

```
while (Boolean_Expression)
{
    Compound_Statement
}
```

thae is no update.



Beware the dreaded **infinite loop** or **dead loop**, *i.e.,* a loop that is executed indefinitely as the Boolean expression never evaluates to *false*. A loop should normally be designed so that the value tested in the Boolean expression is changed in a way that eventually makes it false, and terminates the loop.
Alternatively, the `break` command can be used to exit the loop at any point. This is less clear to read, and is best left for advanced applications.

Another common loop error is the *off-by-one* error, *i.e.,* when a loop repeats the loop body one too many or one too few times.

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

        int count = 10;

        while (count >= 0) {
            System.out.println(count);
            count--;
        }
    }
}
```

10
9
8
7
6
5
4
3
2
1
0

The do-while Statement

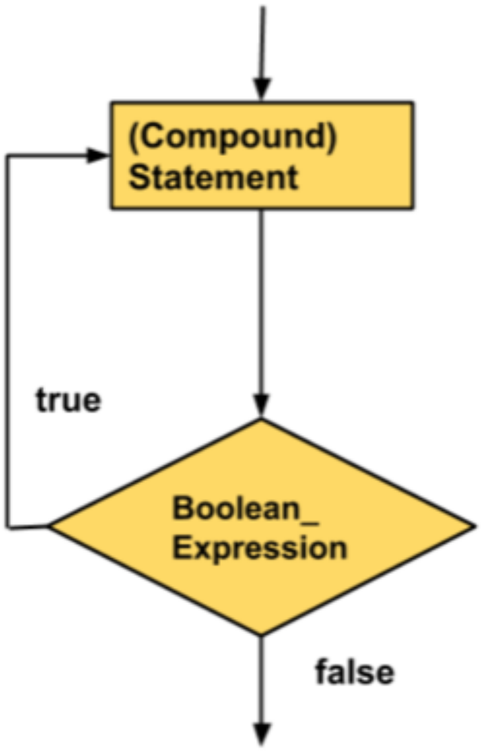
A *do-while* statement is used to execute a portion of code (*i.e.*, the loop body), and then repeat it based on the evaluation of a Boolean expression. Hence, the loop body is executed at least once.

The Boolean expression is checked after each iteration of the loop body. The loop terminates as soon as the expression evaluates to *false.*

```
do
    Statement
while(Boolean_Expression);
```

Like the while statement, the loop body can consist of a single statement or multiple statements enclosed in a pair of braces { }.

```
do
{
    Compound_Statement
} while(Boolean_Expression);
```



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

        int count = -1;

        do {
            System.out.println(count);
            count--;
        } while(count>0);
    }
}
```

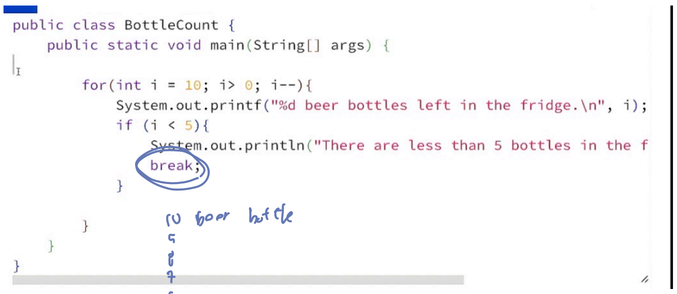
do at least once

do first

and then check if it meet the condition

Don't forget to put a semicolon after the Boolean expression.

The *for* Statement



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

        for(int i = 10; i> 0; i--){
            System.out.printf("%d beer bottles left in the fridge.\n", i);
            if (i < 5){
                System.out.println("There are less than 5 bottles in the f
                break;
            }

        }
    }
}
```

10 beer bottle
9
8
7
6

There are less

The previous types of loop don't keep track of which iteration we are executing. The `for` loop does that for us.

The *for* statement is most commonly used to step through an integer variable in equal increments. It begins with the keyword *for* followed by three expressions in parentheses that describe what to do with one or more controlling variables:

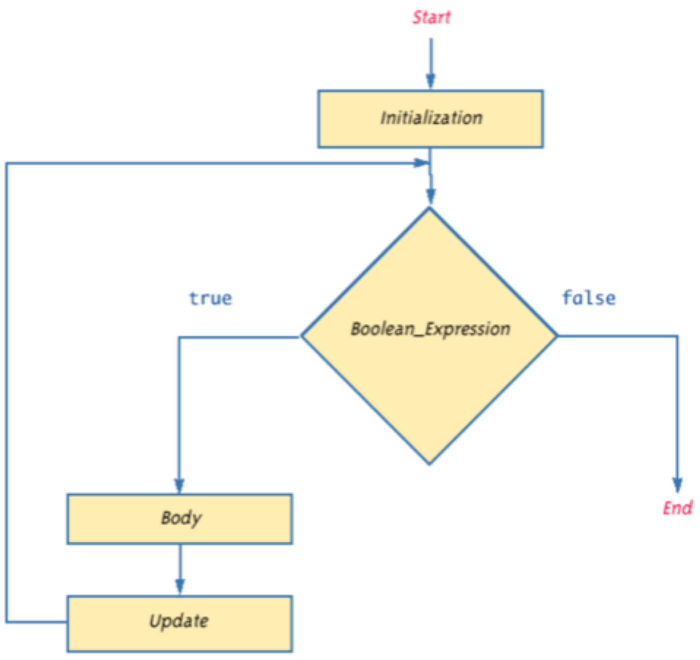
when to stop   what to do next

```
for(Initializing; Boolean_Expression; Update)
    Body
```

The first expression (*Initializing*) tells how the control variable or variables are initialized or declared and initialized before the first iteration The second expression (*Boolean_Expression*) determines when the loop should end, based on the evaluation of a Boolean expression before each iteration. The third expression (*Update*) tells how the control variable or variables are updated after each iteration of the loop body.

The *Body* may consist of a single statement or a list of statements enclosed in a pair of braces { }.

Note that the three control expressions are separated by two, not three, semicolons. There is no semicolon after the closing parenthesis at the beginning of the loop.



```
public class BottleCount {
    public static void main(String[] args) {
        for(int i=10; i>0; i--) {
            System.out.printf("%d beer bottles left in the fridge.\n", i);
        }
    }
}
```

loop body

Any *for* loop can also be written using a *while* loop, for example:

```
int i=10;
while(i>0)
{
    System.out.printf("%d bottles of beer left in the fridge.\n", i);
    i--;
}
```

A *for* loop can contain multiple initialization actions separated by commas:

```
for(int i=0, j=100; i<j; i++) {}
```

Caution must be used when combining a declaration with multiple actions. It is, for example, illegal to combine multiple type declarations with multiple actions, for example. To avoid possible problems, it is best to declare all variables outside the for statement.

Advanced: This is not specific to `for` loops. A comma is just an operator in Java, which evaluates the left hand argument, discards its value and then evaluates the right hand argument. That explains why you can't have multiple declarations in a `for` loop, since a declaration is not part of an expression.

A *for* loop can contain multiple update actions, separated by commas:

```
for(int i=0, j=100; i<j; i++, j--) {}
```

However, a for loop can contain only one Boolean expression to test for ending the loop. This would not make sense since we either perform the next iteration or we don't. However, the boolean expression can contain many `&&` and `||` terms.

```
for(int i=0, j=100; i<40 && j>50; i++, j--) {}
```

Sneak-peak: for-each loops

There is another type of `for` loop that we will come across when we learn about data structures. Instead of iterating by doing arithmetic on the loop variable, it iterates over all values in a data structure.

Nested Loops

Loops can be nested, just like other Java structures. When nested, the inner loop iterates from beginning to end for every single iteration of the outer loop.

```
public class Matrix {
    public static void main(String[] args) {
        for(int row=1; row<4; row++) {
            for(int col=1; col<5; col++) {
                System.out.printf("%d/%d \t", row, col);
            }
            System.out.print("\n");
        }
    }
}
```

The *break* and *continue* Statements

The **break statement** consists of the keyword *break* followed by a semicolon. It ends the nearest enclosing *switch* or *loop* statement.

```
import java.util.Random;     Library Random
public class RandomNumberGenerator {
    public static void main(String[] args) {
        Random rand = new Random();
        int upperbound = 10;
        int count = 0;
        while(true) {
            int randomNumber = rand.nextInt(upperbound);
            System.out.println("random number generated: " + randomNumber);
            count += 1;
            if(randomNumber > 6) {
                break;
            }
        }
        System.out.printf("%d numbers generated", count);
    }
}
```

The **continue statement** consists of the keyword *continue* followed by a semicolon. When executed, the continue statement ends the current loop body iteration of the nearest enclosing loop statement.

```
import java.util.Random;
public class CrackingMyOwnNumber {
    public static void main(String[] args) {
        Random rand = new Random();
        int upperbound = 10;
        int randomNumber = rand.nextInt(upperbound);

        while(upperbound>0) {
            System.out.println("+ I think your number is " + upperbound);
            if(upperbound!=randomNumber) {
                System.out.println("- nope");
                upperbound--;
                continue;
            }
            System.out.println("- you got it!");
            break;
        }
    }
}
```

When loop statements are nested, any *break* or *continue* statement applies to the innermost, containing loop statement.

In a *for* loop, the continue statement transfers control to the update expression.

There is a type of *break* statement that, when used in nested loops, can end any containing loop, not just the innermost loop. To achieve this, we use a **labeled break statement**. If an enclosing loop statement is labelled with an Identifier, the following version of the break statement will exit the labelled loop, even if it is not the innermost enclosing loop:

```
break someIdentifier;
```

To label a loop, simply precede it with an Identifier and a colon:

```
someIdentifier:
```

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

        int i=7;

        outerLoop:  Identifier
        while(i<20) {
            for(int j=1; j<i; j++) {
                System.out.print("*");

                if(i==10)
                    break outerLoop;  identifier

                System.out.print("\n");
                i++;
            }
        }

        System.out.println("\nEnough looping!");
    }
}
```

The *exit* Statement

A *break* statement will end a loop or *switch* statement, but will not end the program. The *exit* method will immediately end the program as soon as it is invoked:

```
System.exit(0);
```

The exit statement takes one integer argument . By tradition, a zero argument is used to indicate a normal ending of the program. However, the cleanest way to end a program is to always let it run to the end of its *main* function, except when you want to report that an error occurred.

Enhanced For Loop

It is a modified form of for loop introduced in Java 5 and used specifically through collections. We will discuss it further when we talk about Arrays and Collections. However, I show you a basic syntax of enhanced for loop.

```
for (dataType variableName : collectionName) {
    // code to be executed for each element
}
```

Debugging

A *bug* is an error or flaw in a program that leads to incorrect or unexpected program execution. The process of finding and fixing bugs is termed **debugging**. Regardless of whether you are hunting down bugs or are trying to understand how exactly your program is executed, the following techniques will help you look under your code's hood.

Defects and Bugs both refer to the issues in the software testing. However, defects is generally preferred word over bug due to several reasons. This will be discussed in SWEN90006 unit which deals specifically with software testing.

Tracing Variables

Tracing variables involves watching one or more variables change value while a program is running. This can make it easier to discover errors in a program and debug them. Many Integrated Development Environments (IDEs) have a built-in utility that allows variables to be traced without making any changes to the program.

Another way to trace variables is to simply insert temporary output statements in a program, such as:

```
//tracing the variable count
System.out.println("count=" + count);
```

Once you have found and corrected the error, you can simply remove or comment out the trace statements.

Using Assertions

An **assertion** is a sentence that says (asserts) something about the state of a program. An assertion must be either *true* or *false*, and should be *true* if a program is working properly. Assertions can be placed in a program as comments:

```
assert Boolean_Expression;
```

If assertion checking is turned on and the Boolean_Expression evaluates to *false*, the program terminates and outputs an *assertion failed* error message. Otherwise, the program finishes execution normally.

```
import java.util.Scanner;
public class DrinkOrder {
    public static void main(String[] args) {
        Scanner keyboard = new Scanner(System.in);
        System.out.println("The menu only lists beer. What would you like?");
        String order = keyboard.nextLine();
        assert order.equals("beer"):" Sorry, mate! Try somewhere else.";
        System.out.println("Coming right up.");
    }
}
```

By default, all programs run with assertion checks turned off. That is why you must explicitly enable assertion checking:

```
$ java -enableassertions DrinkOrder
```

In order to run the above program with assertion checking turned on, copy it to your local machine, compile it and execute it using the command above.

Quiz

The following quiz is designed to help you test your understanding of control flow in Java.

The quiz is not marked.

Question 1

How many loyalty points will a customer have at the end if the following program code is executed via

```
$ java LoyaltyProgram 60 10
```

Program excerpt:

```
public static void main(String[] args) {
        if(args.length < 2) {
            System.out.println("2 arguments required");
            System.exit(1);
        }
        int totalBill = Integer.parseInt(args[0]);
        int points = Integer.parseInt(args[1]);
        if(totalBill > 100) {
            System.out.println("You get 20 points!");
            points += 20;
        } else if (totalBill > 50) {
            System.out.println("You get 10 points!");
            points += 10;
        } else {
            System.out.println("You get 1 point!");
            points += 1;
        }
        System.out.printf("Current points: %d\n", points);
    }
```

- 21
- [✓ 20]
- 30
- 70

Question 2

What is the output of this program if you run it as follows:

```
$ java LuckyNumber 6 8 9999 10 -11
```

Program excerpt:

```java
public class LuckyNumber {
    public static void main(String[] args) {
        boolean isFound = false;
        int luckyNumber = 9999;
        for(int i=0; i<args.length; i++) {
            System.out.println(args[i]);
            if(Integer.parseInt(args[i])==luckyNumber) {
                isFound = true;
                break;
            }
        }
        if(isFound)
            System.out.println("Lucky number found!");
        else
            System.out.println("Lucky number not found!");
    }
}
```

[✓
6

8

9999

Lucky number found!
]

○
6

8

9999

10

-11

Lucky number found!

○ Lucky number not found!

6

8

9999

Lucky number not found!

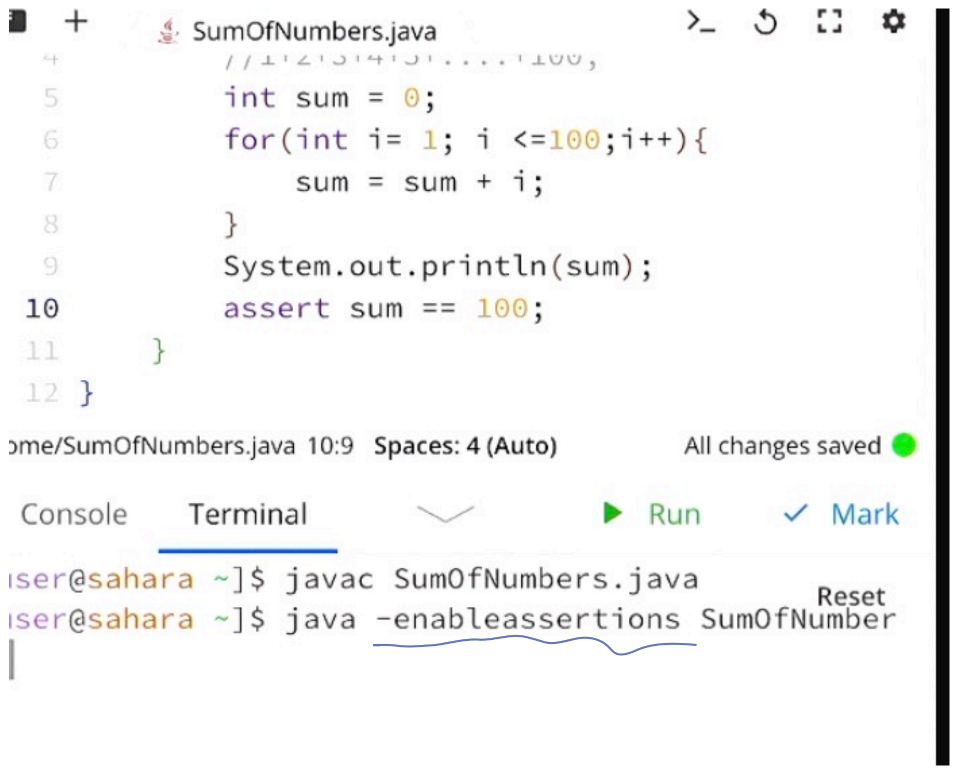
Sum of Numbers

Write a for loop that calculates the sum of the numbers from one to a hundred.

Add an assertion to check the correctness of your result given that

$$sum = \frac{N * (N + 1)}{2}$$

An assertion is a command of the form `assert(boolean_expression)` that prints an error and exits the program if the boolean expression is false.



```java
        //1+2+3+4+5+....+100;
        int sum = 0;
        for(int i= 1; i <=100;i++){
            sum = sum + i;
        }
        System.out.println(sum);
        assert sum == 100;
    }
}
```

```
user@sahara ~]$ javac SumOfNumbers.java
user@sahara ~]$ java -enableassertions SumOfNumber
```

Pyramids

Write a program that prints out a pyramid of stars using command-line arguments and a nested for-loop.

```
$ java Pyramid 5 left
```

The program execution above should lead to the following result:

```
*
**
***
****
*****
```

This pyramid is *left*-aligned as indicated by the second command-line argument.

```
$ java Pyramid 3 right
```

The program execution above should lead to the following, *right*-aligned result:

```
   *
  **
 ***
```

Advanced: Support the "A" option to print a capital A or the chosen size, so that

```
$ java Pyramid 5 A
```

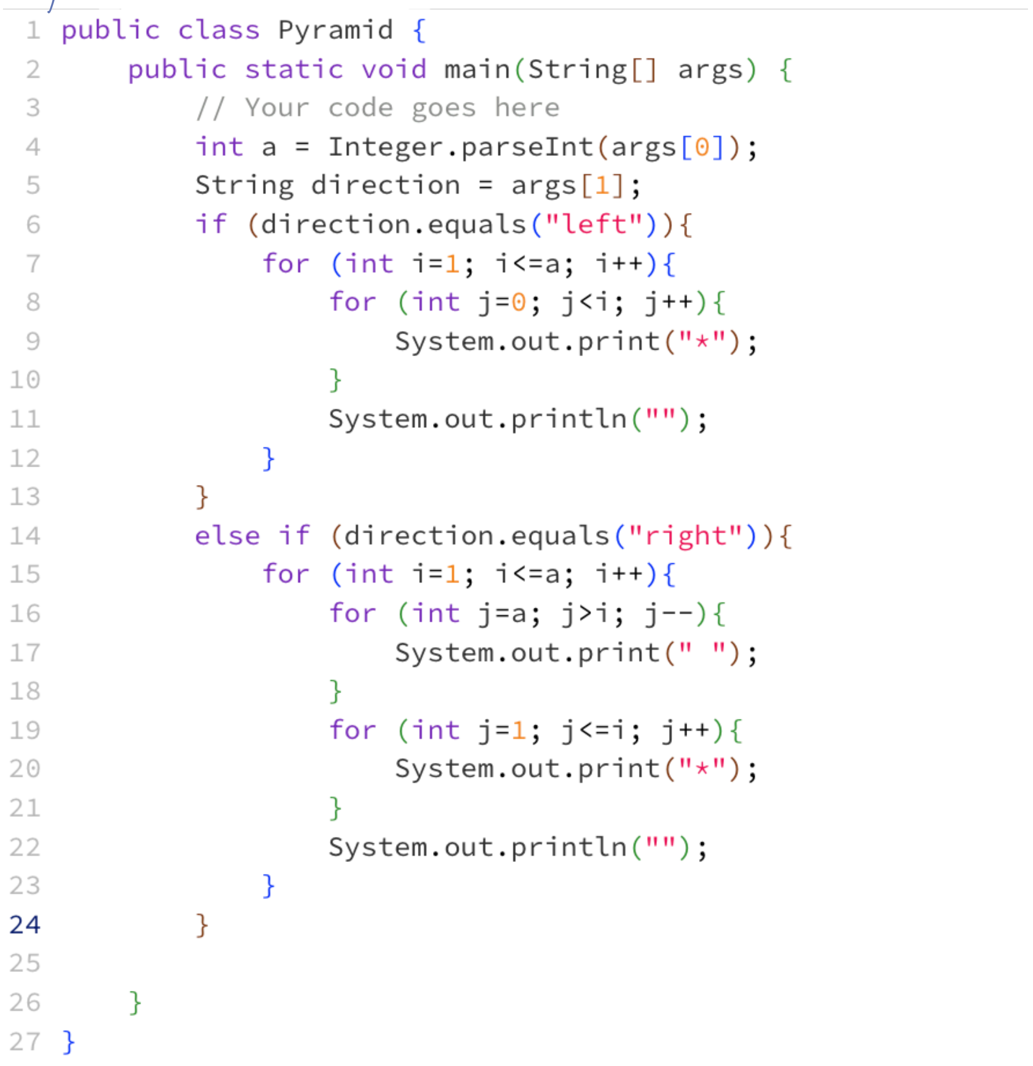
produces

```
     **
    *  *
   ******
  *      *
 *        *
```

or its mirror image.

```
public class Pyramid {
    public static void main(String[] args) {
        int rows = Integer.parseInt(args[0]);
        String alignment = args[1];
        for (var i=0; i<rows; i++) {
            String stars = "";
            if (!alignment.equals("A")) {
                for (var j=0; j<=i; j++) {
                    stars += "*";
                }
                if(alignment.equals("right"))
                    System.out.printf("%" + rows + "s\n", stars);
                else if (alignment.equals("left"))
                    System.out.print(stars + "\n");

            // Advanced option
            } else {    // alignment == "A"
                int indent = rows - i;
                if (i == rows/2) {
                    for (int j = 0; j < 2*(i+1); j++)
                      stars += "*";
                } else {
                    stars = "*";
                    for (int j = 0; j < 2*i; j++)
                        stars += " ";
                    stars += "*";
                }
                System.out.printf("%" + indent + "s%s\n", " ", stars);
            }
        }

    }
}
```

I tryi



```java
public class Pyramid {
    public static void main(String[] args) {
        // Your code goes here
        int a = Integer.parseInt(args[0]);
        String direction = args[1];
        if (direction.equals("left")){
            for (int i=1; i<=a; i++){
                for (int j=0; j<i; j++){
                    System.out.print("*");
                }
                System.out.println("");
            }
        }
        else if (direction.equals("right")){
            for (int i=1; i<=a; i++){
                for (int j=a; j>i; j--){
                    System.out.print(" ");
                }
                for (int j=1; j<=i; j++){
                    System.out.print("*");
                }
                System.out.println("");
            }
        }

    }
}
```

Guess my Number

Build a program that picks a random number and has the user guess it. Use the class *Random* for picking a number between 0 and an upper limit, which is specified as a command-line argument.

Familiarize yourself with the class *Random* and make sure to import it as follows:

```
import java.util.Random;
```

Your program should take a command-line argument, which specifies the range for the random number as in:

```
$ java GuessMyNumber 10
```

The program execution above would instruct the program to create a random number between 0 and 10. It then asks the user to guess the number:

```
What number am I thinking of?
```

The user puts in a number to which your program should respond whether that number is lower or higher:

```
Nope! My number is smaller. Try again!
```

or

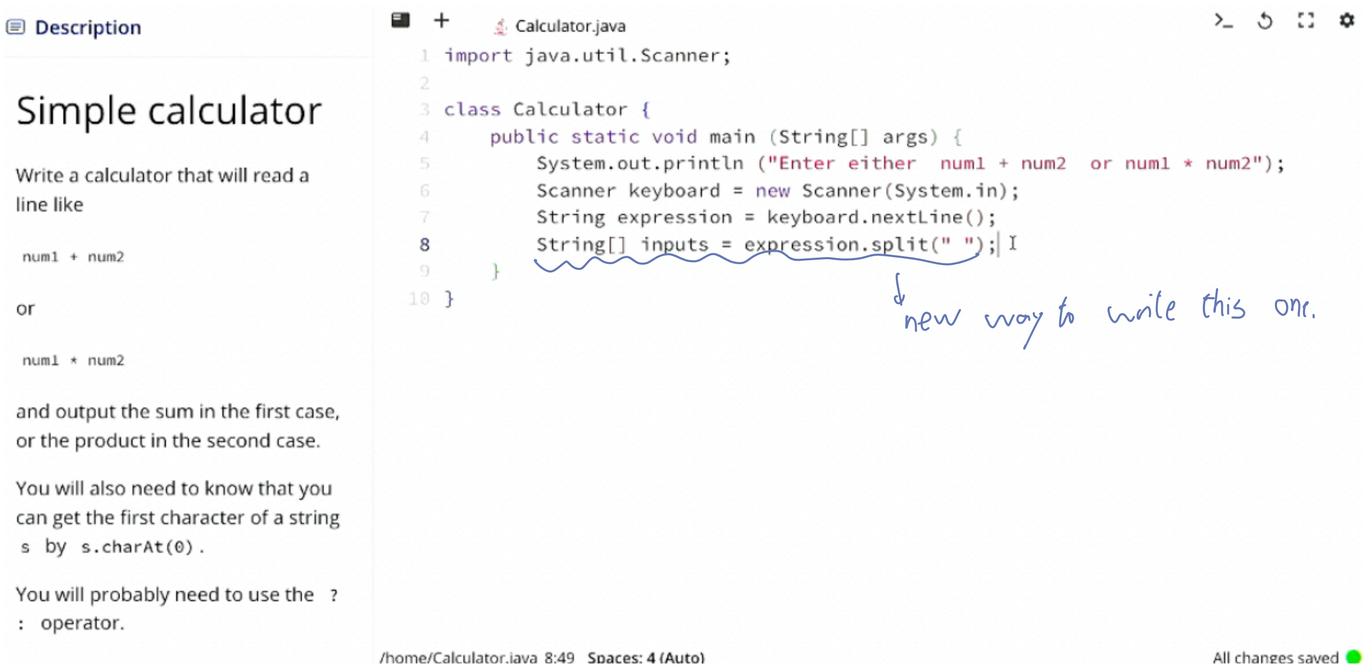
```
Nope! My number is bigger. Try again!
```

If the user guesses the correct number, your program should congratulate the user and show the number of guesses made:

```
You've got it! It only took you 144 trials.
```

Only integer numbers should be considered.

Advanced: Do the reverse, where the program has to guess a number chosen by the user (but not revealed by the user). Make the user enter only "higher", "lower" and "yes". You can use any algorithm you like. Binary search is the most efficient, but it may be boring if the computer always guesses the same number first. You can start with a random guess and then do binary search from there. The random guess doesn't have to be uniform; it could be always closer to the middle to be more like binary search. Make it as sophisticated or simple as you like. Remember, this is only optional.



Description

Simple calculator

Write a calculator that will read a line like

`num1 + num2`

or

`num1 * num2`

and output the sum in the first case, or the product in the second case.

You will also need to know that you can get the first character of a string `s` by `s.charAt(0)`.

You will probably need to use the `? :` operator.

Calculator.java

```java
import java.util.Scanner;

class Calculator {
    public static void main (String[] args) {
        System.out.println ("Enter either  num1 + num2  or num1 * num2");
        Scanner keyboard = new Scanner(System.in);
        String expression = keyboard.nextLine();
        String[] inputs = expression.split(" ");
    }
}
```

new way to write this one.

/home/Calculator.java 8:49 Spaces: 4 (Auto) All changes saved