SWEN-601 Software Construction

Boolean Expressions, & Conditional Statements

Activity: Accept the GitHub Classroom Assignment

You will be asked to accept a new assignment at the start of nearly every class.

You should get used to accepting the assignment and starting your new project right after you finish your quiz each day.

Your instructor has provided a GitHub classroom invitation. You should be able to find it under “Homework” on MyCourses.

1. Click the GitHub classroom invitation.
2. Assuming that you have already linked your GitHub account with your name in the class roster, you should be prompted to accept the assignment. Do so.
3. Once the repository is created, copy the URL.
4. Clone the repository to your local file system. As before, the repository will be empty.
5. Create a new IntelliJ Project inside the repository, and push it to GitHub.
6. Create a package named “`activities`” in your `src` folder.
7. You are now ready to begin today’s activities!

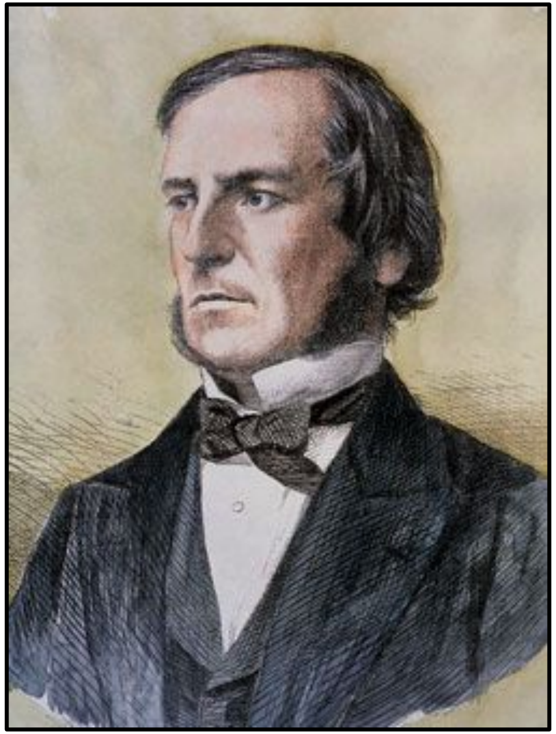
Booleans

- Remember that a `boolean` is a value in Java that can be either `true` or `false`.
  - `true` and `false` are considered reserved words and can't be used in another context (e.g. a variable name).

```
boolean example;

example = true;

example = false;
```



You can thank George Boole for Booleans.

Boolean Expressions: Equality Operators

- A *boolean expression*, also called a *condition*, is an expression that evaluates to either `true`, or `false`.
- Java's **equality operators**, also sometimes called "relational operators," are used to create boolean expressions by comparing values to each other.
- Java supports the following equality operators:

	English Name	Example	Description/Definition
`==`	equal to	`a == b`	`true` if `a` and `b` are *equal*, `false` otherwise
`!=`	not equal to	`a != b`	`true` if `a` and `b` are *not equal*, `false` otherwise
`<`	less than	`a < b`	`true` if `a` is *strictly less than* `b`, `false` otherwise
`<=`	less than or equal to	`a <= b`	`true` if `a` is *less than or equal to* `b`, `false` otherwise
`>`	greater than	`a > b`	`true` if `a` is *strictly greater than* `b`, `false` otherwise
`>=`	greater than or equal to	`a >= b`	`true` if `a` is *greater than or equal to* `b`, `false` otherwise

Boolean Expressions: Equality Operators

```
boolean equalTrue = (3 == 3); // true
boolean equalFalse = (3 == 4); // false

boolean notEqualTrue = (3 != 4); // true
boolean notEqualFalse = (3 != 3); // false

boolean lessThanTrue = (3 < 4); // true
boolean lessThanFalse = (3 < 3); // false

boolean lessThanOrEqualTrue = (3 <= 3); // true
boolean lessThanOrEqualFalse = (4 <= 3); // false

boolean greaterThanTrue = (4 > 3); // true
boolean greaterThanFalse = (3 > 4); // false

boolean greaterThanOrEqualTrue = (3 >= 3); // true
boolean greaterThanOrEqualFalse = (3 >= 4); // false
```

In these examples the values being compared are all numbers.

The == operator can be used to compare non-numeric types as well.

For example (`"abc" == "abc"`) is `true`.

And (`true == false`) is `false`.

Boolean Expressions: Logical Operators

- In addition, Java provides *logical operators* that may also be used in boolean expressions.
- A **logical operator** takes `boolean` operands and produces a `boolean` result.

	English Name	Example	Description/Definition
!	(logical) NOT (also called 'negation')	`!x`	`true` if x is `false`. `false` if x is `true`.
&&	(logical) AND	`a && b`	`true` if a and b are both `true`. `false` if a or b (or both) is `false`.
||	(logical) OR	`a || b`	`true` if a or b (or both) are `true`. `false` if both a and b are `false`.
^	(logical) EXCLUSIVE OR	`a ^ b`	`true` if a OR b (but not both) are `true`. `false` if both a and b are `false` or both a and b are `true`.

Boolean Expressions: Logical Operators

```java
boolean a = true;
boolean b = false;
boolean c = true;
boolean d = false;

boolean notTrue = (!b); // true
boolean notFalse = (!a); // false

boolean andTrue = (a && c); // true
boolean andFalse = (a && b); // false

boolean orTrue = (a || b); // true
boolean orFalse = (b || d); // false

boolean xorTrue = (a ^ b); // true
boolean xorFalse = (a ^ c); // false
```

Logical operators ***only*** work with boolean values.

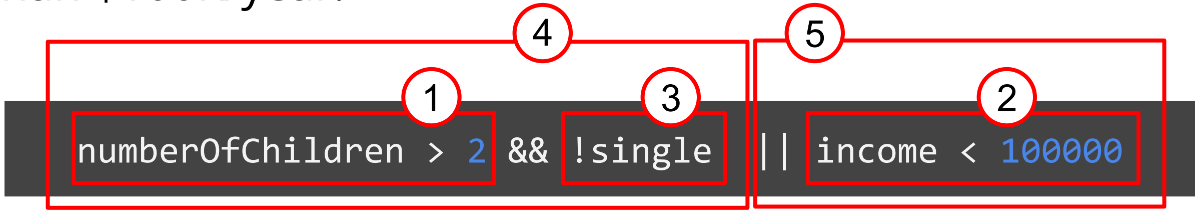
Note that logical NOT is a ***unary*** operator, which means that it takes a single operand.

But logical AND, OR, and XOR are ***binary*** operators: each takes two operands.

Boolean Expressions

- Complex expressions can be created by combining logical operators with equality operators.
- All logical operators have lower *operator precedence* than equality operators.
- Logical NOT has a higher precedence than logical AND and logical OR.
- Like arithmetic operators, logical operators and equality operators of equal precedence are evaluated left to right.
- Exercise: Award a child tax credit to people who have children *and* are either married *or* make less than $100K/year.

Operator Precedence
`==, !=, <, <=, >, >=`
`!`
`&&`
`||`


```
numberOfChildren > 2 && !single || income < 100000
```

Pro tip: Just use parentheses!

```
(numberOfChildren > 2) && (!single || (income < 100000))
```

Short Circuited Operators

- Processing of logical AND and logical OR is "short circuited."
- If the left operand is sufficient to determine the result, and the right operand is not evaluated.
- Given boolean values `a` and `b`:
  - if `a` is `false`, then `a && b` is `false` regardless of the value of `b`
  - if `a` is `true`, then `a || b` is `true` regardless of the value of `b`
- This kind of processing must be used carefully!
- Consider:

```
numberOfChildren > 2 && count++ < 5
```

If the left side is `false`, the logical AND is short circuited, and the right side is not evaluated. The `count` variable is never incremented!

```
single || person.isHomeowner()
```

If `single` is `true`, the logical OR is short circuited, and the right side is not evaluated. The `isHomeowner()` method is never called!

Truth Tables

Truth tables can be used to compute the results of specific boolean expressions.

numberOfChildren > 2	!single	income < 100000	(numberOfChildren > 2) && (!single || (income < 100000))
true	true	true	true
true	true	false	true
true	false	true	true
true	false	false	false
false	true	true	false
false	true	false	false
false	false	true	false
false	false	false	false

Truth Tables

- Operator precedence may produce unexpected results...

`(numberOfChildren > 2) && (!single || (income < 100000))`	`numberOfChildren > 2 && !single || income < 100000`
true	true
true	true
true	true
false	false
false	true
false	false
false	true
false	false

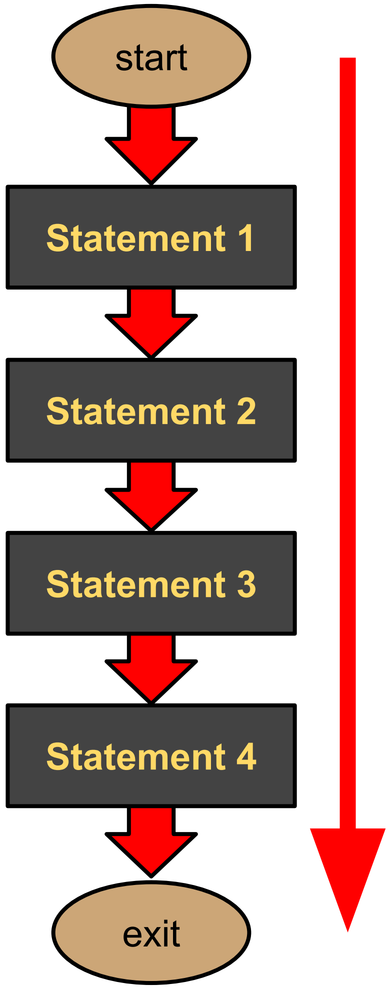
Without parentheses, *anyone* with an income of less than $100,000 gets the child tax credit regardless of the number of children that they have. Is this what was intended?

QUESTIONS?

Sequential Flow of Control

Remember: the sequential execution of statements in a program is referred to as sequential flow of control.

Statements are executed one after the other, in the same order, every time the program is executed.



Call and Return

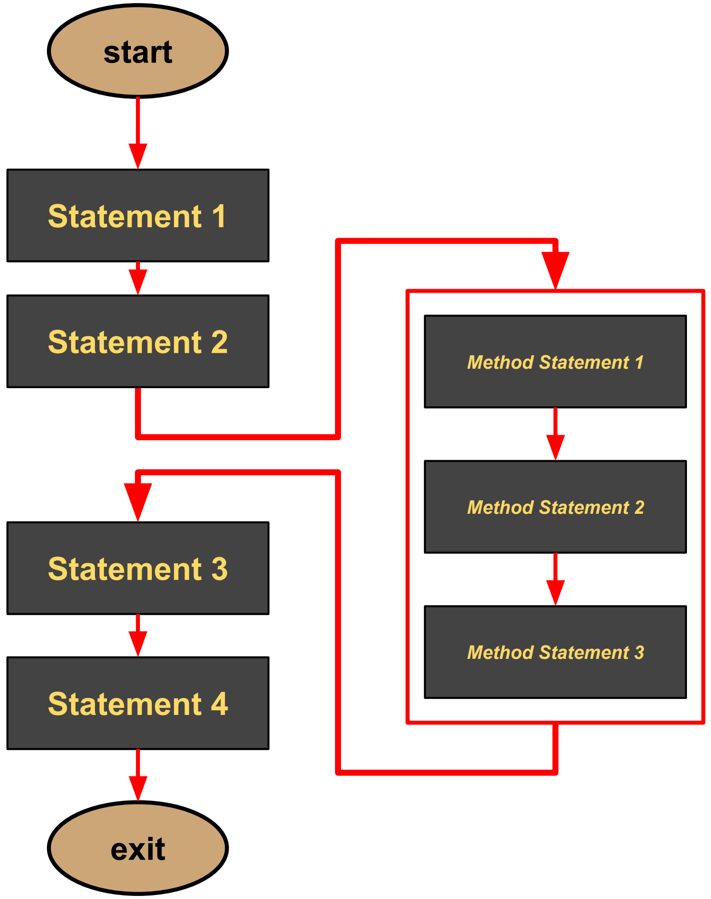
A normal flow of control begins.

When the method is ***called***, control jumps to the first statement in the body of the method.

Control flows through the statements in the method...

...before ***returning*** to the location in the program from which it was called.

From here, the normal flow of control continues.



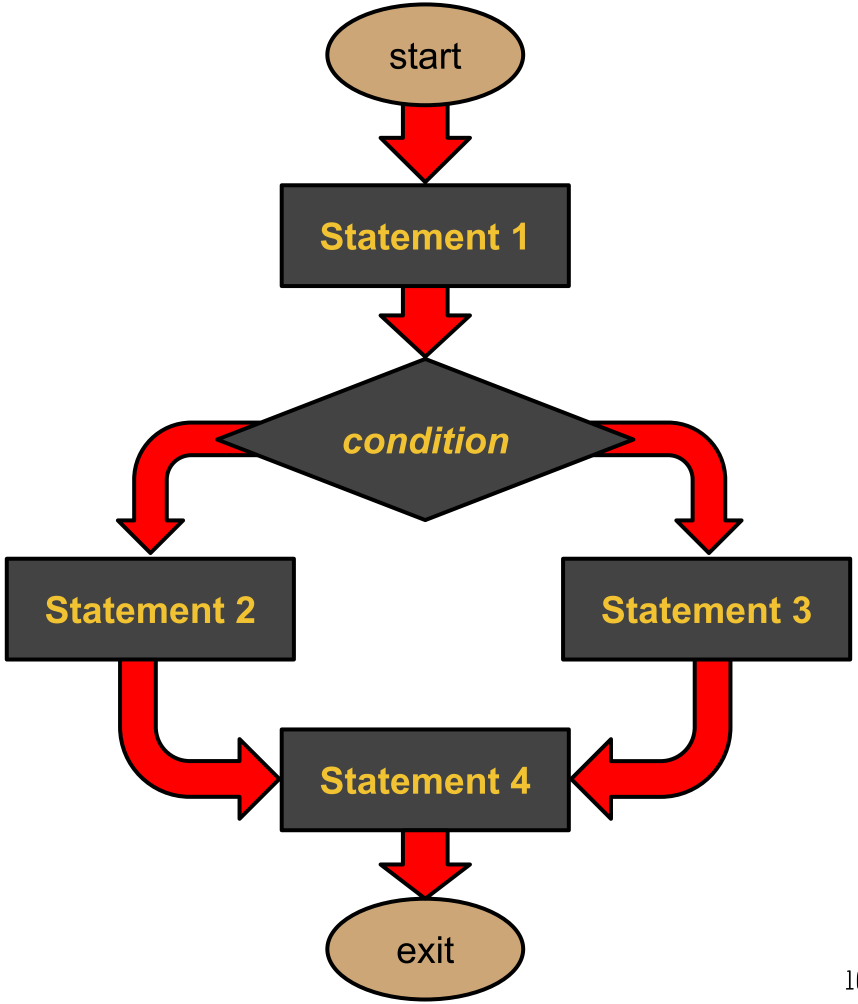
Branching Flow of Control

But sometimes, it is preferable for a program to choose between two different branches in the code based on some ***condition*** that may change.

If the ***condition*** is `true`, then execute statement 2...

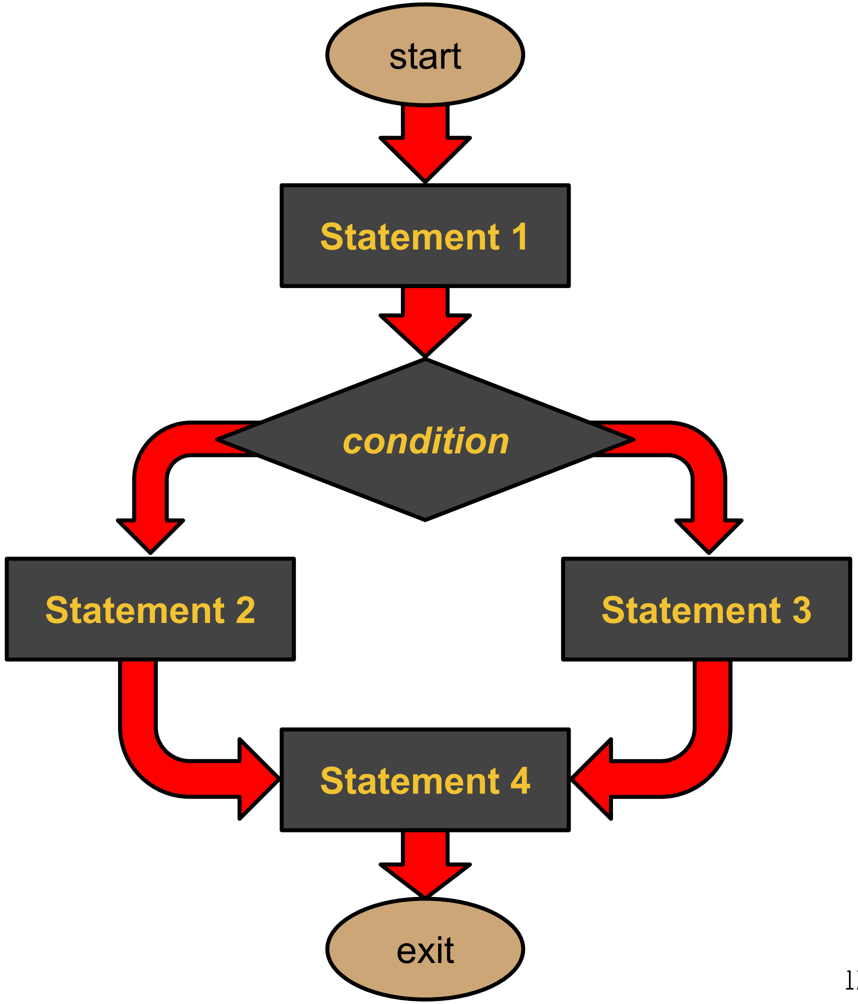
...but if the ***condition*** is `false`, then execute statement 3 instead.

Eventually, execution may return to the same sequence.



Conditional Statements

- A ***conditional statement*** allows the program to choose which statement will be executed next.
  - Conditional statements are sometimes called *selection statements*
- Conditional statements give the program the ability to make basic decisions based on the evaluation of a `boolean` expression, i.e. one that evaluates to either `true` or `false`.
  - If something is `true`, do *this*.
  - If it is `false`, do *that* instead.



Conditional Statements

- In Java the conditional statements are:
  - `if` statement - if the condition specified in the Header is true, the statements in the body of the `if` statement are executed.
  - `if else` statement - if the condition specified in the header is true, the statements in the body of the `if` are executed, otherwise the statements in the body of the `else` are executed instead.
  - `switch` statement - executes one of a set of cases depending on whether or not the input value is equal to the value specified in the case.
  - **the ternary condition** (`?:`) - given a condition and pair of expressions, evaluates the first expression if the condition is true, and the second expression if the condition is false.

The if Statement

- The if statement has the following syntax:

```
if( condition ) {

   statement(s);

}
statement(s);
```

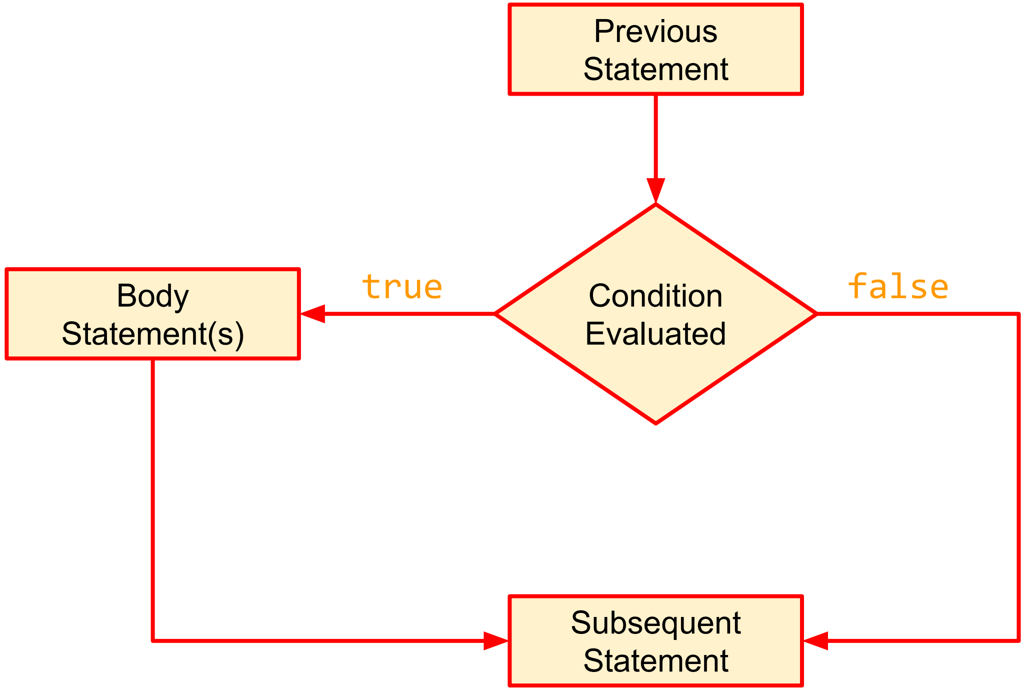
The Head { Body } pattern is used. The *condition* specified in the Head evaluates to either `true` or `false`.

If the ***condition*** is `true`, the ***statement***(s) in the Body are executed sequentially. If `false`, the ***statement***(s) are skipped.

Execution continues with statements following the body.

“`if`” is a reserved word in Java, like “`new`,” “`int`,” and “`boolean`.” It can’t be used anywhere other than an `if` statement.

Logic of an `if` Statement



Boolean Expressions as Conditions

- The ***condition*** in an `if` statement must evaluate to `true` or `false`.
- The condition may be any boolean value, which includes any expression that combines logical operators (NOT, AND, OR) with equality operators (==, >, etc.).
  - Any of the previous examples of boolean expressions may be used.

```
if(( numberOfChildren > 2) && (!single || (income < 100000))) {
    taxRate = taxRate - 0.05;
}
```

If the condition is `true`...

...then the statement is executed. If the condition is `false`, the statement is skipped.

Activity: Evens & Odds

%

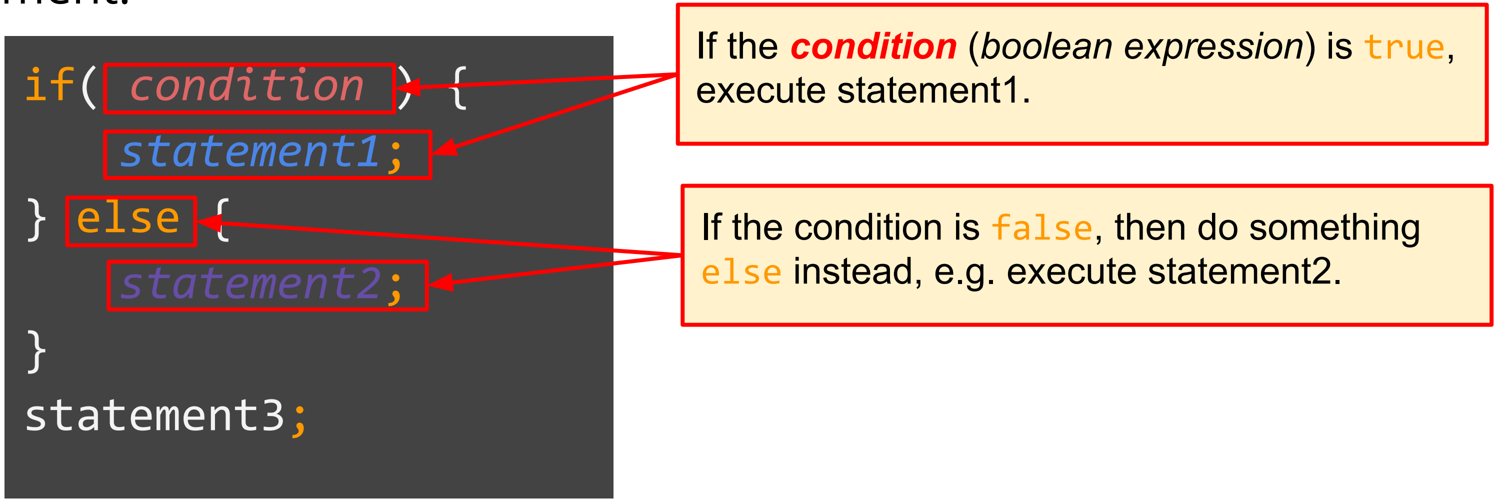
Remember that modulo (%) can be used to compute a remainder.

1. Write a method that, given an integer, prints a message if the integer is even.
2. Write a `main` method that prompts the user to enter an integer and calls your other method to print a message if the number is even.

```
Enter a number: 2078654
The number is even.
```

The if else Statement

- An *else clause* can be added to an if statement to make it an if else statement.



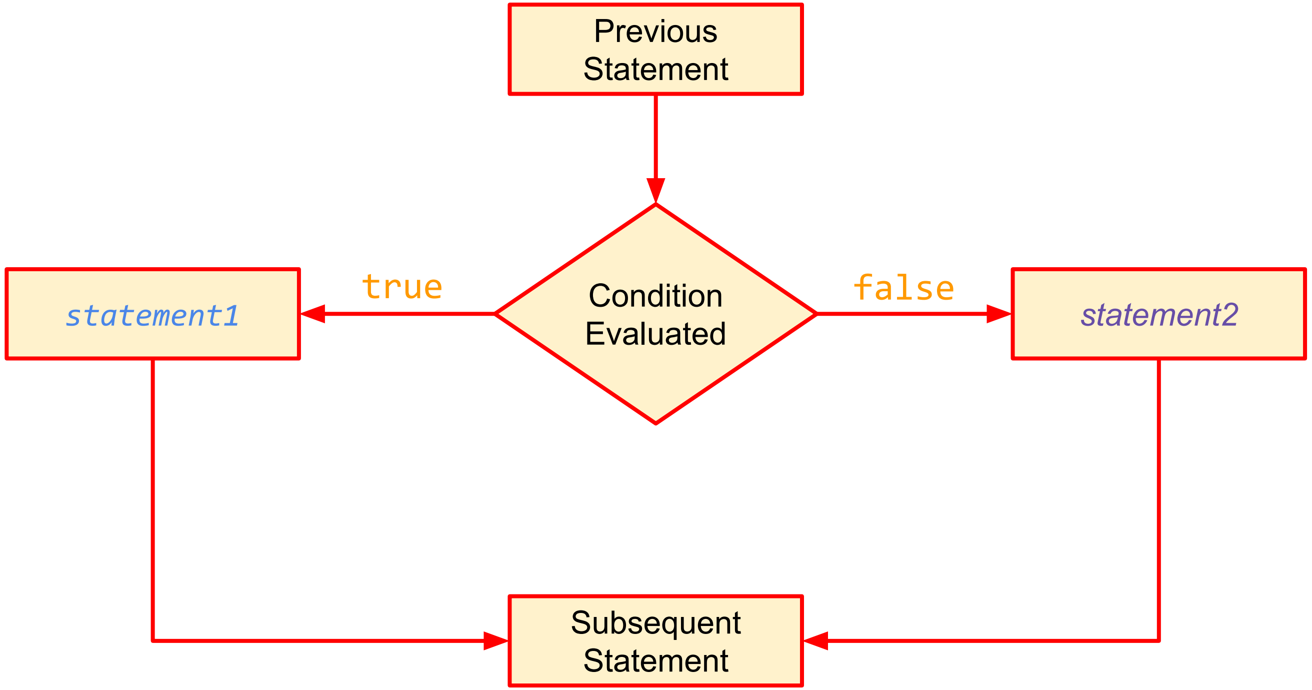
```
if( condition ) {
    statement1;
} else {
    statement2;
}
statement3;
```

If the ***condition*** (*boolean expression*) is true, execute statement1.

If the condition is false, then do something else instead, e.g. execute statement2.

- *Either* ***statement1*** will be executed *or* ***statement2*** will be executed, but **not both**.

Logic of an `if` `else` Statement



Activity: Odds & Evens

%

Remember that modulo (%) can be used to compute a remainder.

1. Update your class so that it also prints ***one*** of the following messages.
   a. The number is even.
   b. The number is odd.

```
Enter a number: 2078653
The number is odd.
```

Nested `if` Statements

- One of the statements executed as part of the body of an `if` or `else` statement could be another `if` statement.
- These are called *nested if statements*.
- For example:

```
if( !single ) {
    taxRate = 0.15;
} else if( income > 100000 ) {
    taxRate = 0.25;
} else {
    taxRate = 0.20;
}
```

Each `else` is always paired with the last unmatched `if`.

Activity: Evens & Odds & Fives

1. Update your class so that it prints ***one*** of the following messages:
   a. The number is divisible by 5.
   b. The number is even.
   c. The number is odd.

%

Remember that modulo (%) can be used to compute a remainder.

```
Enter a number: 2078655
The number is divisible by 5.
```

The `switch` Statement

- The `switch` statement provides another mechanism for making decisions within a program.
- The `switch` statement evaluates an expression and attempts to match the result to one of several possible *cases*.
- Each `case` specifies a value. If the value matches the result of the expression specified in the `switch` statement, the statements within the `case` are executed.
- The flow of control transfers to the statements(s) specified within the *first* matching `case`, but may flow through to the next `case`.
    - This flow from one `case` to another may be intentional, but often is not.
    - A `break` statement is used to interrupt the flow and exit from the **`switch`** statement.
- `switch` and `case` are reserved words (like `if`, `else`, `void`) and may only be used within `switch` statements.

The switch Statement

- The general syntax for a `switch` statement is as follows:

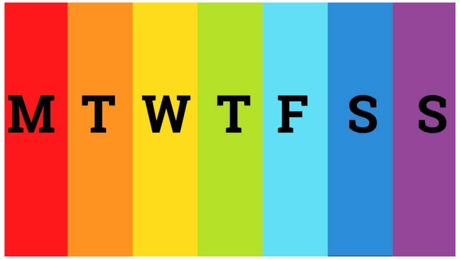
```
switch( expression ) {
    case value1:
        statement1;
        statement2;
    case value2:
        statement3;
        statement4;
    case value3:
        statement5;
        statement6;
}
```

The ***expression*** is evaluated and results in some value (**not** a `boolean` value).

The value is then compared to each `case` until a match is found. In this example, if the expression result matches value2, the flow of control jumps to the first statement inside the corresponding `case`.

The flow of control will continue from one statement to the next until a `break` statement is encountered, or the end of the `switch` statement is reached.

Activity: Day of the Week



Run your code and test it.
What happens?

We will use integers to represent days of the week. Assume that 0 is Monday, 1 is Tuesday, and so on.

1. Create a new class, `Week`.
2. Write a function that, given an integer parameter, returns a `String` that is the name of the day.
   a. e.g. an argument if 2 should return "`Wednesday`"
   b. You **must** use a switch statement.
3. Write a main method that prompts the user to enter a number and prints a message with the corresponding day.

```
Enter the day: 4
It is Friday.
```

The switch Statement

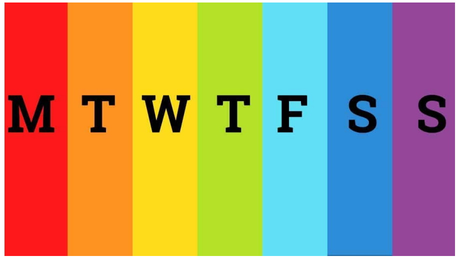
- It is common to use a break statement at the end of each case.
- The break statement causes the flow of control to jump to the end of the switch statement, skipping all remaining cases.

```
switch( expression ) {
   case value1:
      statement1;
      break;
   case value2:
      statement3;
      break;
}
statement4;
```

Flow of control moves to the case that specifies the value that matches the expression, e.g. `value1`.

When a break statement is reached, the flow of control jumps to the first statement after the end of the switch statement.

Activity: One Day of the Week



Fix your code so that it returns the correct day of the week.

1. Use break statements!

OK, so now what happens when the user enters a number < 0 or > 6?

```
Enter the day: 4
It is Friday.
```

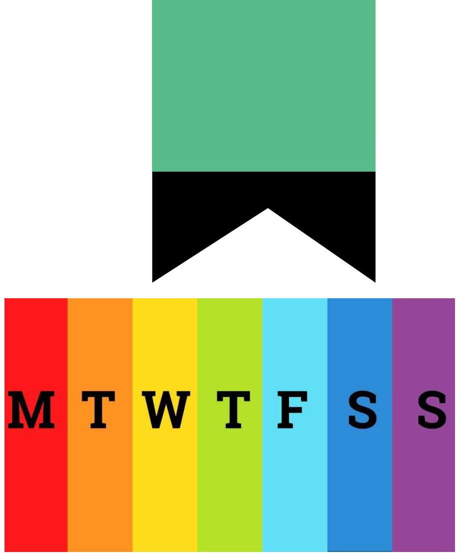
The switch Statement

- The switch statement can have a default case.
- The default case is specified with the default reserved word, has no value, and is executed if none of the other cases match.

```
switch( expression ) {
    case value1:
        statements;
    case value2:
        statements;
    default:
        statements;
}
```

If none of the values match the result of the expression...

...the flow of control jumps to the default case. Think of it as the switch statement's else.



Activity: Wrong Day of the Week

Fix your code so that it returns an appropriate error message if the day is invalid.

1. Use a `default` statement!
2. Return `null` if the day is invalid.

```
Enter the day: -37
That is not a valid day.
```

OK, so now what happens when the user enters a number < 0 or > 6?

The `switch` Statement

- The *expression* of a switch statement must result in an integer (`byte`, `short`, `int`, `long`), a `char`, or a `String`.
  - It *cannot* be `boolean` or floating point.
- It is *similar* to a series of `if`-`else`-`if` statements comparing the same expression to a series of different values for equality.

```
if( expression == value1 ) {
    statements
} else if( expression == value2 ) {
    statements
} else if ( expression == value3 ) {
    statements
} else {
    statements
}
```

One major difference is that `break` statements are not needed to prevent flow of control from moving from one `else-if` to the next.

Another is that `if-else-if` requires a boolean condition while a `switch` statement *cannot* work on a boolean condition. The *implied* condition of a `switch` is *always* equality.

The final `else` is similar to the `default` case; it is only executed if none of the other `if-else-if` statements match.

The `switch` Statement

- How about an example?

```java
String suffix;
switch( dayOfMonth % 10 ) {
  case 1:
     suffix = "st";
     break;
  case 2:
     suffix = "nd";
     break;
  case 3:
     suffix = "rd";
     break;
  default:
     suffix = "th";
     break;
}
System.out.println( "It’s the " + dayOfMonth + suffix +
     " day of the month!" );
```

Q: What would happen without the **`break`** statements?

A: The suffix would always be “th” because flow of control would move from one case to the next, and finally to **`default`**.

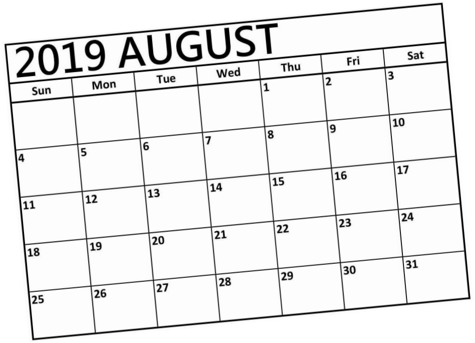
Q: What is the bug in this code?

A: It doesn’t work for 11, 12, or 13 (all of which should end with “th”).

Activity: Day of the Month

Fix the bug in the code on the previous slide.

1. Create a new class, `Calendar`.
2. Write a function that declares an integer parameter and returns a string.
   a. It should return the appropriate suffix for any day between 1 and 31 (inclusive).
   b. It should work for any integer value from 1-31 (including 11, 12, and 13).
   c. It **must** use a `switch` statement.
3. Write a main function that prompts the user to enter a day of the month. Print the number with the appropriate suffix.



Hint: cases without a `break` statement will ***fall through*** to the next case.

```
Enter the day of the month: 11
It is the 11th.
```

The Conditional Operator

- Java has a conditional operator that works sort of like shorthand for an `if-else` statement.
- Its syntax is:

```
var x = condition ? expression1 : expression2;
```

```
var x;
if( condition ) {
    x = expression1;
} else {
    x = expression2;
}
```

The conditional operator works like an `if-else` statement. If the condition is `true`, *expression1* is evaluated. If the condition is `false`, *expression2* is evaluated.

The major difference is that the conditional operator ***returns a value***.

The Conditional Operator

- The conditional operator *returns a value*.
    - The condition determines which of two expressions will be evaluated.
    - Both expressions must return a value of ***the same type***.
- For example:

```
int max = ( value1 > value2 ? value1 : value2 );
```

If `value1` is greater than `value2`, return `value1`. If not, return `value2`.

```
int max;
if( value1 > value2 ) {
    max = value1;
} else {
    max = value2;
}
```

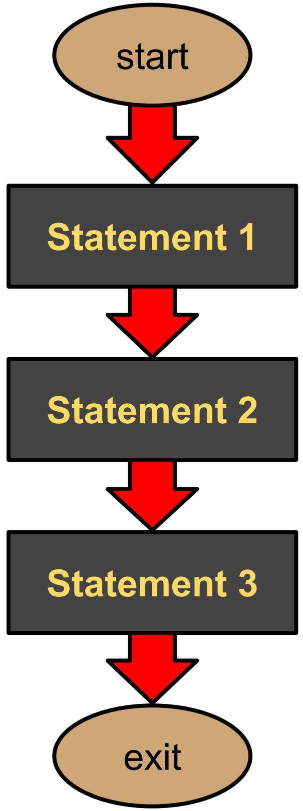
Which is shorthand for doing something like this.

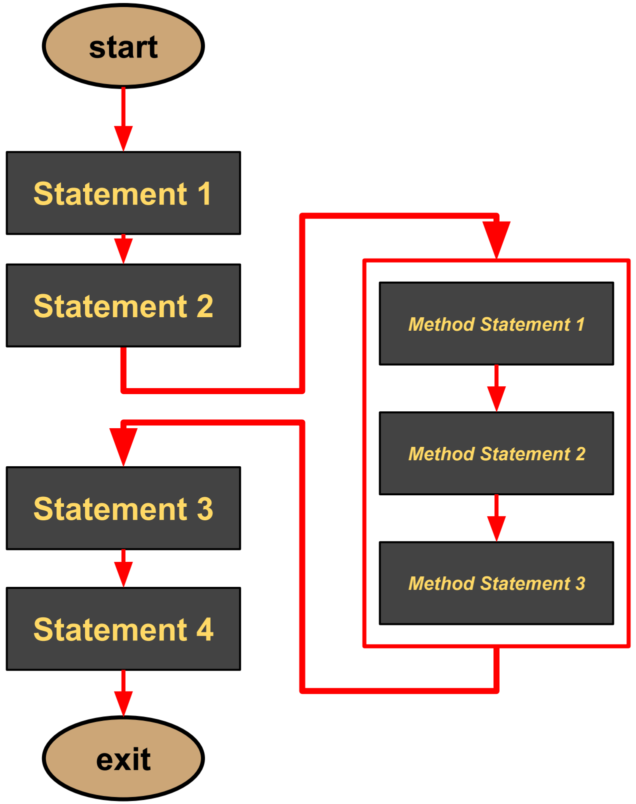
Activity: Making Change

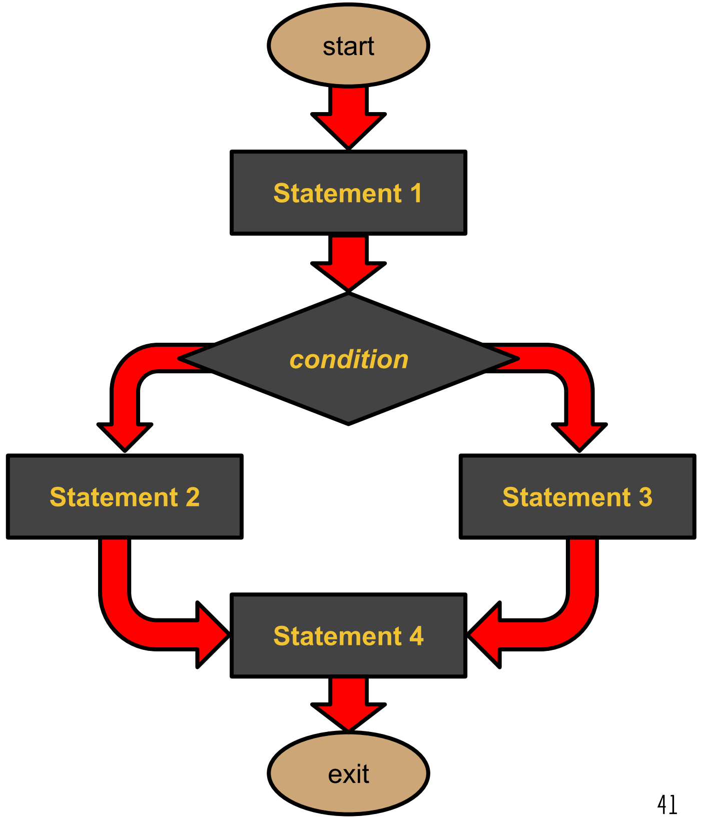
1. Create a new class, `Register`.
2. Write a function that declares two `float` parameters: one for a `charge`, and one for the `payment`.
   a. It should return a string in the format: "`Your change is D dollars and C cents.`"
   b. Dollars and/or cents should ***not*** be plural if the change includes only one dollar and/or one cent.
   c. It **must** use a unary operator.
3. Write a main to prompt the user to enter the charge and the payment. Print the change.

```
Enter the charge: 8.99
Enter the payment: 10.00
Your change is 1 dollar and 1 cent.
```

Flows of Control


start
Statement 1
Statement 2
Statement 3
exit


start
Statement 1
Statement 2
Method Statement 1
Method Statement 2
Method Statement 3
Statement 3
Statement 4
exit


start
Statement 1
condition
Statement 2
Statement 3
Statement 4
exit