codecademy

Conditionals and Control Flow

else Statement

The `else` statement executes a block of code when the condition inside the `if` statement is `false`. The `else` statement is always the last condition.

```
boolean condition1 = false;

if (condition1){
    System.out.println("condition1 is
true");
}
else{
    System.out.println("condition1 is not
true");
}
// Prints: condition1 is not true
```

else if Statements

`else - if` statements can be chained together to check multiple conditions. Once a condition is `true`, a code block will be executed and the conditional statement will be exited.
There can be multiple `else - if` statements in a single conditional statement.

```
int testScore = 76;
char grade;

if (testScore >= 90) {
  grade = 'A';
} else if (testScore >= 80) {
  grade = 'B';
} else if (testScore >= 70) {
  grade = 'C';
} else if (testScore >= 60) {
  grade = 'D';
} else {
  grade = 'F';
}

System.out.println("Grade: " + grade); //
Prints: C
```

if Statement

An `if` statement executes a block of code when a specified boolean expression is evaluated as `true`.

```
if (true) {
    System.out.println("This code executes");
}
// Prints: This code executes

if (false) {
    System.out.println("This code does not execute");
}
// There is no output for the above statement
```

Nested Conditional Statements

A nested conditional statement is a conditional statement nested inside of another conditional statement. The outer conditional statement is evaluated first; if the condition is `true`, then the nested conditional statement will be evaluated.

```
boolean studied = true;
boolean wellRested = true;

if (wellRested) {
  System.out.println("Best of luck today!");
  if (studied) {
    System.out.println("You are prepared for your exam!");
  } else {
    System.out.println("Study before your exam!");
  }
}

// Prints: Best of luck today!
// Prints: You are prepared for your exam!
```

AND Operator

The AND logical operator is represented by `&&`. This operator returns `true` if the `boolean` expressions on both sides of the operator are `true`; otherwise, it returns `false`.

```
System.out.println(true && true); //
Prints: true
System.out.println(true && false); //
Prints: false
System.out.println(false && true); //
Prints: false
System.out.println(false && false); //
Prints: false
```

NOT Operator

The NOT logical operator is represented by `!`. This operator negates the value of a boolean expression.

```
boolean a = true;
System.out.println(!a); // Prints: false

System.out.println(!true) // Prints: true
```

The OR Operator

The logical OR operator is represented by `||`. This operator will return `true` if at least one of the `boolean` expressions being compared has a `true` value; otherwise, it will return `false`.

```
System.out.println(true || true); //
Prints: true
System.out.println(true || false); //
Prints: true
System.out.println(false || true); //
Prints: true
System.out.println(false || false); //
Prints: false
```

Conditional Operators - Order of Evaluation

If an expression contains multiple conditional operators, the order of evaluation is as follows: Expressions in parentheses → NOT → AND → OR.

```
boolean foo = true && (!false || true); //
true
/*
(!false || true) is evaluated first
because it is contained within
parentheses.

Then !false is evaluated as true because
it uses the NOT operator.

Next, (true || true) is evaluation as
true.

Finally, true && true is evaluated as true
meaning foo is true. */
```

DeMorgan's Laws

DeMorgan's Laws can be used to rewrite expressions complex `boolean` expressions.
The first law states that two expressions that are negated together and compared using `&&` is equivalent to two separately negated expressions compared with `||`.
The second law states that two expressions that are compared with `||` and are negated together are equivalent to two separately negated expressions compared with `&&`.

```
int a = 2;
int b = 3;

boolean exp1 = !(a > b && a == b);
// rewrite using first law
exp1 = !(a > b) || !(a == b);

boolean exp2 = !(a < b || a != b);
// rewrite using second law
exp2 = !(a < b) && !(a != b);
```

Equivalent Boolean Expressions

Equivalent `boolean` expressions are separate `boolean` expressions that always result in the same value.
If we were to replace a `boolean` expression in a program with an equivalent `boolean` expression, there would be no impact on the output of the program.

```
int a = 1;
int b = 2;
// the followning expressions are
equivalent
boolean exp1 = !(a == b && b >= a);
boolean exp2 = !(a == b) || !(b >= a);
boolean exp3 = a != b || a < b;

System.out.println(exp1); // Prints: true
System.out.println(exp2); // Prints: true
System.out.println(exp3); // Prints: true
```

Compare Object References

Boolean expressions allow us to compare object references. A Boolean expression is a Java expression that, when evaluated, returns a Boolean value: true or false.

```
a.equals(b)
a.equals(b) && b.equals(c)
```

Comparing Primitive Values

We can use relational operators, such as `==` and `!=`, to compare primitive and reference values.

```
class ComparingPrimitives {
  public static void main(String[] args) {
    System.out.println("Comparing ints:");
    System.out.println(4 == 5); // print
false
    System.out.println(4 != 5); // print
true
    System.out.println(4 == 4); // print
true

    System.out.println("Comparing
chars:");
    System.out.println('a' == 'b'); //
print false
    System.out.println('a' != 'b'); //
print true
    System.out.println('a' == 'a'); //
print true
  }
}
```

Object Reference Aliases

An alias means that more than one reference is tied to the same object.

```
class ComparingAliases {
 public static void main(String[] args) {
   String farmAnimal1 = new String("cat");
   String farmAnimal2 = new String("cow");
   // farmAnimal3 references the same
object as farmAnimal2
   String farmAnimal3 = farmAnimal2;

   // comparing different objects
   System.out.println(farmAnimal1 ==
farmAnimal2); // print false
   // comparing object aliases
   System.out.println(farmAnimal2 ==
farmAnimal3); // print true
 }
}
```

Comparing Object Reference Aliases

We can compare object reference values can be compared, using == and !=, to identify aliases.

```
class ComparingAliases {
 public static void main(String[] args) {
   String farmAnimal1 = new String("cat");
   String farmAnimal2 = new String("cow");
   String farmAnimal3 = farmAnimal2;

   // comparing different objects
   System.out.println(farmAnimal1 ==
farmAnimal2); // print false
   // comparing object aliases
   System.out.println(farmAnimal2 ==
farmAnimal3); // print true
 }
}
```

Comparing Reference Values with Null

We can compare a reference value with null, using == or !=, to determine if the reference actually references an object.

```
class Main
{
   public static void main(String[] args)
   {
     String word = null;
     // checking that `word != null`
avoids NullPointerException error
     if (word != null && word.indexOf("a")
>= 0)
     {
         System.out.println(word + "
contains an a.");
     }
   }
}
```

Custom Class Equals Method

Classes often have their own equals method, which can be used to determine whether two objects of the class are equivalent.

```
class Pet {
  public String name;
  public String breed;
  public Pet (String name, String breed) {
    this.name = name;
    this.breed = breed;
  }
  // custom `equals()` method
  public boolean equals(Pet p) {
    return (p.name == name && p.breed ==
breed);
  }
  public static void main(String[] args) {
    Pet pet1 = new Pet("Air Bud", "Golden
Retriever");
    Pet pet2 = new Pet("Air Bud", "Golden
Retriever");
    // compare with `==`
    System.out.println(pet1 == pet2);
    // compare with `.equals()`
    System.out.println(pet1.equals(pet2));
  }
}
```