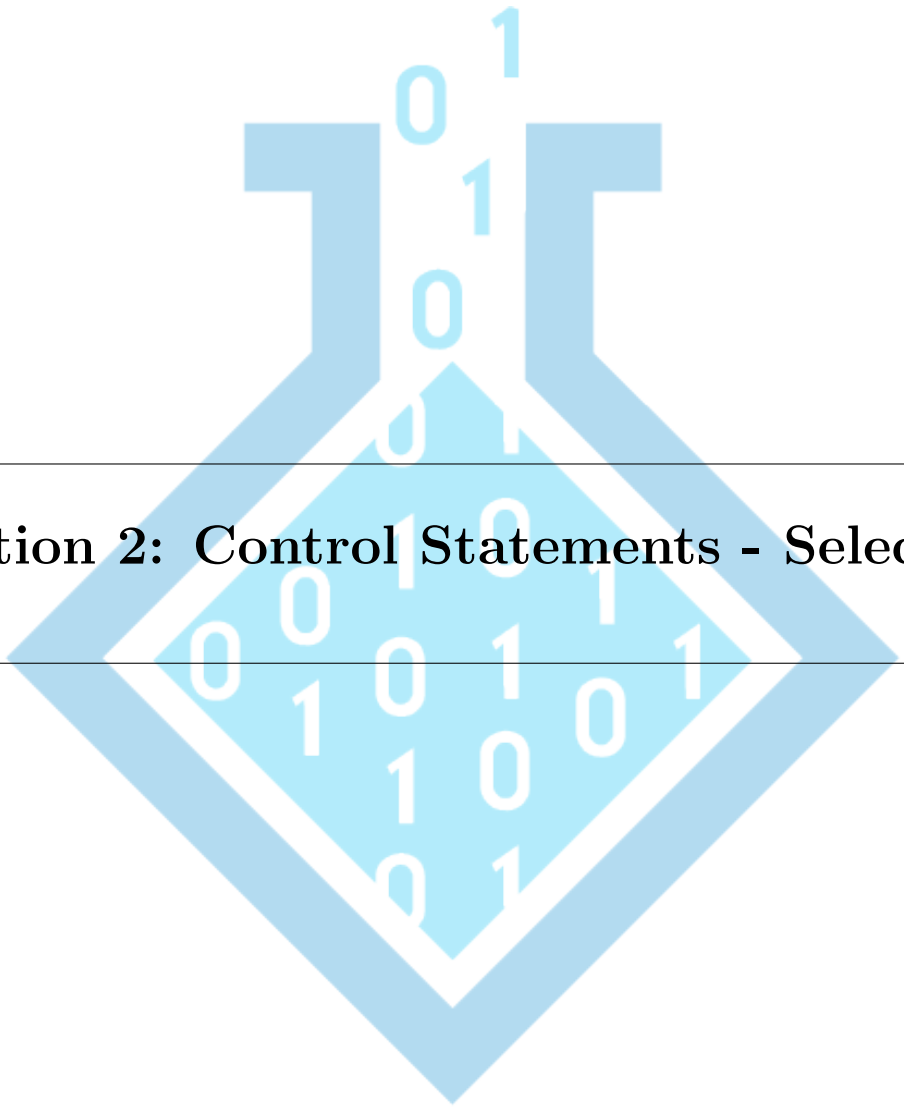

Section 2: Control Statements - Selections

Computer Science 101: Introduction to Java and Algorithms

June 5, 2019

Contents

1 Pre-Chapter

1.1 "final" Keyword (When Used With Variables)

We have talked about keywords before, but most of them we have never used. In this Chapter I will use this keyword, so it might help to know what it does. the "final " keyword does a number of things depending on how you use it. Most of these things we will cover later, so I will cover the use of final with those things when we cover them. In this instance, we are using it with a variable.

When it comes to final with variables, we would use it in the form of

final	TYPE	$VARIABLE\_NAME$	=	VALUE

So for example,

```
final int CONTINUE = 1;
```

What does "final" do to a variable? Well, it makes the variable constant, meaning, once you set the variable, it cannot be changed. In the above code snippet, we assigned it to 1, and for the duration of the program, it will stay 1. We cannot change the value of that variable or else the compiler will give us an error. If we need to change the value of CONTINUE, then we shouldn't use the final keyword.

1.2 // comments

You have probably seen code snippets where you see something like "// do this stuff". This is called a comment. Comments are used to provide additional information for the person reading your code. the "//" tells the compiler to just ignore all other text after "//" until the end of that line. Java also has multi-line comments where you can tell the compiler to ignore multiple lines in your source code. The "/* comment here */" is used for milti-line comments. So for example,

```
// a single line comment
// if I want to comment multiple lines with this
// then I have to use // each line
int i = 0; // I can also do it at the end of some piece of code and the code will work
//int i = 0; however this code will be totally ignored because it is commented out.

/*
   This is a mult line comment
   I do not need to put anything else to tell the compiler
   that I want it to ignore all this text
```

```
        int i =0; code inside of here will also be ignored.
    */
```

1.3 Propositional Calculus

Many college degrees in Computer Science require a course in formal logic. The most common taught is propositional calculus, which is also known as sentential logic, statement logic, propositional logic, or zeroth-order logic (there are multiple orders of logic). Logic is the cornerstone of human thought, science, engineering and mathematics. All of these can be reduced to logical statements. It's been around since the ancient Greek philosophers, most notably Aristotle, and further advanced by the Stoic Philosophers. In Aristotle's metaphysical writings, he introduced the foundations of propositional logic. He wrote about the "Law of Excluded Middle" and the "Law of Contradiction". In propositional logic, that is to say that every statement is either true or false and that no statement is both true and false.

We will not be covering propositional logic in this course other than to introduce some of the origins of the logical statements found in programming. My main purpose here is merely to introduce the subject so that you know it exists and to see some of terms. If you would like to learn more about propositional logic, I will put some useful links here for you to explore.
<https://www.geeksforgeeks.org/proposition-logic/>
<https://www.iep.utm.edu/prop-log/#H1>

2 Relational and Logical Operators

In this part of the series, we introduce the relational operators. Interestingly enough, they are all the same operators that exist in basic mathematics. However, this would be our first experience with logic. By that I mean it will be helpful to start thinking of things in terms of true and false. For example, $1 < 2$ is a true statement. Whereas $5 > 1$ is a false statement. That's all there really is to say about it.

2.1 The Relational Operators

Math Symbol	Programming Symbol	Definition	Example	Value
$>$	$>$	is greater than	$5 > 4$	true
$\geq$	$>=$	is greater than or equal to	$5 >= 5$	true
$<$	$<$	is less than	$4 < 10$	false
$\leq$	$<=$	is less than or equal to	$5 <= 10$	true
$=$	$==$	is equal to	$8 == 4$	false
$\neq$	$!=$	is not equal to	$8 != 4$	true

Take note that in programming, we combine symbols such as $<=$ instead of $\leq$, why? Well simply because, do you see $\leq$ on your keyboard? If you do, then I have no idea what keyboard you are using.

A more important one to note is the 'equal to' operator. In mathematics we use $=$ as both assignment and to express an equality relation. In fact, we generally use them as one in the same. However, a computer needs to make a distinction due to additional things the compiler must do. In programming, we use $=$ as assignment. When we say

```
int a = 5;
```

what we are actually saying is, hey compiler, create a variable in memory with enough space to store an integer and store the value 5 there. Now take the case,

```
int a = 5;
int b =0;
a = b;
```

Here we tell the compiler to again create the variable a and store the value 5 and then create the variable b and store the value 0. Then we are assigning the value that is stored in the variable a into the variable b. That is, we are copying the value stored in variable a into b. So when we try to ask the program, "is a equal to b" we cannot use the $=$ symbol. If we did, the computer would think that we are trying to assign something. Instead we use the symbols $==$ to ask if something is equal. This will make more sense when we try to use these operators.

2.2 Logical Operators

The logical operators are directly rooted into propositional logic. What's great is that we use these everyday, often without even realizing it.

Logic	Programming Symbol	Definition	Example	Value
$\&(Conjunction)$	&&	AND	$(5 == 5)$ && $(5! = 10)$	true
$V(Disjunction)$	||	OR	$(3 == 3)$ || $(5 == 3)$	true
$\neg(Negation)$	!	NOT	$!(5! = 5)$	true

In programming (and logic) you can combine these logical operations to evaluate to some boolean value.

2.2.1 AND

In propositional logic, AND is a conjunction of two statements. We as humans have learned to communicate in this way as well, so it is often helpful to say it as a sentence to yourself and see if it makes sense. For example, "Fire is hot and fire burns wood". The two statements, "Fire is hot" and "fire burns wood" are both true about fire. We know this to be true because of our senses. Since the two statements are true, the conjunction of the two statements must also be true.

Let us look at the truth table for the conjunction AND (I will be using the programming symbols for the logical operators),

A	B	A && B
false	false	false
false	true	false
true	false	false
true	true	true

As you can see, the only time the AND operator is true is when both statements are true. If you really think about it, this makes sense when you speak. The next table will just be in plain English.

Statement A	Statement B	A && B	value	Explanation
Fire is cold	Fire is wet	Fire is cold and Fire is wet	false	We know fire isn‘t cold or wet so it also can‘t be both cold and wet
Fire is cold	Fire is hot	Fire is cold and Fire is hot	false	Fire is hot, but we know it isn‘t cold, so the sentence really makes no sense
Fire is hot	Fire is cold	Fire is hot and Fire is cold	false	moving the true part of the sentence to the other side doesn‘t change the fact that fire isn‘t both hot and cold
Fire is hot	Fire burns wood	fire is hot and fire burns wood	true	we know that fire is hot and that fire burns wood, so fire must be hot and also it must burn wood, which makes sense.

The next table, I will use conjunctions of relation statements, similar to what we would find in programming. In this table we will use a variable, i, let $i = 5$

A	B	A && B	Value
$(i == 3)$	$(i >= 10)$	$(i == 3)$ && $(i >= 10)$	false
$(i < 3)$	$(i <= 5)$	$(i < 3)$ && $(i <= 5)$	false
$(i == 5)$	$(i != 5)$	$(i == 5)$ && $(i != 5)$	false
$(i >= 5)$	$(i <= 5)$	$(i >= 5)$ && $(i <= 5)$	true

Note that these two tables both correspond to the AND truth table. Now I would like to do a more complicated example with conjunctions of relational statements. Let $i = 5$ again. Now let us evaluate
$(((i != 5)$ && $(i == 10))$ && $i <= 5)$

First identify the order, we must do the inner parenthesis first.
$((i != 5)$ && $(i == 10))$
Now we need to identify the statements
$(i != 5)$ and $(i == 10)$
Now evaluate these statements
$(i != 5)$ is false
$(i == 10)$ is false
So, $((i != 5)$ && $(i == 10))$ is equivalent to ((false) && (false)).

We can consult the Truth table to see that $((i \; != 5) \; \&\& \; (i == 10))$ is false

Now we have a conjunction of (false && $(i <= 5)$)
we know that $(i <= 5)$ is true
So, we have reduced $(((i \; != 5) \; \&\& \; (i == 10)) \; \&\& \; i <= 5)$ to (false && true)
We can again consult the truth table to see that (false && true) is false.

Therefore, $(((i \; != 5) \; \&\& \; (i == 10)) \; \&\& \; i <= 5)$ evaluates to false.

2.2.2 OR

In propositional logic, OR is the disjunction of two statements. For example, "It is sunny" or "it is cloudy". It is one or the other. The truth table looks like this:

A	B	A || B
false	false	false
false	true	true
true	false	true
true	true	true

As you can see, the only time that the whole disjunction is false is when both statements are false. This makes sense when we talk as well. When using OR we are speaking in terms of one or the other or both. When it comes to OR, we only need one statement to be true to make the whole disjunction true. Here is an English truth table.

Statement A	Statement B	A || B	value	Explanation
Fire is cold	Fire is wet	Fire is cold or Fire is wet	false	We know fire isn‘t cold or wet so it is neither.
Fire is cold	Fire is hot	Fire is cold or Fire is hot	true	Fire is hot, but we know it isn‘t cold, but since fire is hot, we know it is at least one of those two options
Fire is hot	Fire is cold	Fire is hot or Fire is cold	true	moving the true part of the sentence to the other side doesn‘t change the fact that fire is hot even if it isn‘t cold
Fire is hot	Fire burns wood	fire is hot or fire burns wood	true	we know that fire is hot and that fire burns wood, so fire must be hot and also it must burn wood, so take your pick.

Now again, but using relational statements that you would see in programming. Let $i = 5$

A	B	A || B	Value
$(i == 3)$	$(i >= 10)$	$(i == 3) \,\|\|\, (i >= 10)$	false
$(i < 3)$	$(i <= 5)$	$(i < 3) \,\|\|\, (i <= 5)$	true
$(i == 5)$	$(i \,!= 5)$	$(i == 5) \,\|\|\, (i \,!= 5)$	true
$(i >= 5)$	$(i <= 5)$	$(i >= 5) \,\|\|\, (i <= 5)$	true

Lets use the same statement from the above subsection, but swap out the AND‘s with OR‘s and see what happens to the value. Let us evaluate:

$(((i \,!= 5) \,||\, (i == 10)) \,||\, i <= 5)$

First identify the order, we must do the inner parenthesis first.

$((i \,!= 5) \,||\, (i == 10))$

Now we need to identify the statements

$(i \,!= 5)$ or $(i == 10)$

Now evaluate these statements

$(i \,!= 5)$ is false

$(i == 10)$ is false

So, $((i \,!= 5) \,||\, (i == 10))$ is equivalent to ((false) || (false)).

We can consult the Truth table to see that $((i \,!= 5) \,||\, (i == 10))$ is false

Now we have a disjunction of (false || ($i <= 5$))

we know that ($i <= 5$) is true

So, we have reduced ((($i != 5$) || ($i == 10$)) || $i <= 5$) to (false || true)

We can again consult the truth table to see that (false || true) is true.

Therefore, ((($i != 5$) || ($i == 10$)) || $i <= 5$) evaluates to true.

2.2.3 NOT

In propositional logic, NOT is the equivalent to the negation. Another way to think of this is "the opposite of". Obviously, not true is false and not false is true, so the truth table looks like:

A	!A
false	true
true	false

It's sort of like saying a statement and then pausing at the end and saying "just kidding¡'. Like this, "Fire is cold...JUST KIDDING", we just took a false statement and made it true.

Statement A	!A	value	Explanation
Fire is cold	Fire is cold...NOT	true	We know fire is not cold, but we want the opposite value
Fire is hot	Fire is hot...NOT	false	We know fire is hot, but we want the opposite value

Let's see if we can make our example using the OR's false by using NOT

((($i != 5$) || ($i == 10$)) || $i <= 5$)

First identify the order, we must do the inner parenthesis first.

(($i != 5$) || ($i == 10$))

Now we need to identify the statements

($i != 5$) or ($i == 10$)

Now evaluate these statements

($i != 5$) is false

($i == 10$) is false

So, (($i != 5$) || ($i == 10$)) is equivalent to ((false) || (false)).

We can consult the Truth table to see that (($i != 5$) || ($i == 10$)) is false

Now we have a disjunction of (false || ($i <= 5$))
we know that ($i <= 5$) is true
So, we have reduced ((($i != 5$) || ($i == 10$)) || $i <= 5$) to (false || true)
We can again consult the truth table to see that (false || true) is true.

Therefore, ((($i != 5$) || ($i == 10$)) || $i <= 5$) evaluates to true.

To make the whole disjunction false, we need to make the second statement false. So, what if we change the expression to
((($i != 5$) || ($i == 10$)) || !($i <= 5$))

First identify the order, we must do the inner parenthesis first.
(($i != 5$) || ($i == 10$))
Now we need to identify the statements
($i != 5$) or ($i == 10$)
Now evaluate these statements
($i != 5$) is false
($i == 10$) is false
So, (($i != 5$) || ($i == 10$)) is equivalent to ((false) || (false)).
We can consult the Truth table to see that (($i != 5$) || ($i == 10$)) is false

Now we have a disjunction of (false || !($i <= 5$))
we know that ($i <= 5$) is true, so !($i <= 5$) must be false
So, we have reduced ((($i != 5$) || ($i == 10$)) || !($i <= 5$)) to (false || false)

We can consult the OR truth table to see that false || false is false.

3 True and False

In this section I would like to just do more examples.

Let $i = 5$ and $k = -2$
Let's evaluate ((($i == 5$) || ($i <= k$)) && !(($k != i$) && ($k > 9$)))

Let's start on the left hand side (($i == 5$) || ($i <= k$))
Evaluate the left and right hand side of this disjunction

$(i == 5)$ is true
$(i <= k)$ is false
So $((i == 5) \,||\, (i <= k))$ becomes ((true) || (false)) which is true

so now we have ((true) && $!((k\,!= i)$ && $(k > 9)))$

Now let's move to the right side
$!((k\,!= i)$ && $(k > 9))$
$(k\,!= i)$ is true
$(k > 9)$ is false
so $((k\,!= i)$ && $(k > 9))$ becomes ((true) && (false)) which is false
now $!((k\,!= i)$ && $(k > 9))$ becomes !(false) which is true
so $!((k\,!= i)$ && $(k > 9))$ is true

Now altogether we have ((true) && (true)) which is true
Therefore, $(((i == 5) \,||\, (i <= k))$ && $!((k\,!= i)$ && $(k > 9)))$ evaluates to true

4 if Statement

We have talked about the relational and logical operators and their evaluations all for this moment (and others, but this moment first). These are your tools for evaluating conditions. We combine these conditions with statements. The first being the if statement.

4.1 if

```
if(CONDITION_IS_TRUE) {
    // DO THIS STUFF
}
```

You would use the if-statement whenever you only wanted to execute some code when a condition is met. The if statement simply says, "if this condition is true, then execute the next block of code." You could also imply that if the condition is true, then execute the next block of code, but if the condition is false, don't execute the next block of code.

If-statements are pretty crucial to programming. For example, we all have little bombs inside our cars. The air bag in your car inflates with gas when you slow down real quick, in fact, so intensely that you would go from 55 mph to 0mph in a split second. Well that explosive in your steering wheel is regulated by an accelerometer that is constantly getting your deceleration and probably using an if statement to check if your current deceleration is greater than or equal to the deceleration that you would see in a car crash. If it is, then detonate the explosive so that the gas from the explosion can expand and fill the air bag. If statements literally save your life in cases of terrible car accidents.

Or lets say you are creating a banking system and you only want to allow a user to take out money if they have enough funds to do so. You would need to get the current amount of money in the bank and use an if statement to check if the bank amount is greater than the amount of money the user wishes to withdraw. If it is, then execute the code to withdraw the requested amount.

There are often numerous variables and conditions that you may want in place before some piece of code is executed. This is the use case of the if statement.

A note of caution however. We have been doing some pretty long and complicated conditions for practice. This is generally a very bad idea when programming. You want your conditions to be as simple and easy to read as possible. If they get too long or complicated, then you need to rethink your design. That being said, sometimes you have no choice but to use long, ugly conditions.

4.2 Full Code Examples

```
public class HelloWorld{

    public static void main(String []args){
       int menuOption = 2;

       if(menuOption == 1) {
           System.out.println("Option 1: Enter Game");
       }

       if(menuOption == 2) {
           System.out.println("Option 2: You are Playing");
       }

       if(menuOption == 3) {
           System.out.println("Option 1: Enter");
```

```
        }
    }
}
```

5 if-else and if- else if -else Statements

5.1 if else

Remember in the above section when I described the if-statement as follows: "if the condition is true, then execute the next block of code, but if the condition is false, don't execute the next block of code." Well what if we want to do something if the condition fails. What if we could reword it as follows: "if the condition is true, then execute the next block of code, but if the condition is false, don't execute the next block of code, but instead execute this other block of code". What if we have a bunch of if statements and none of them pass, should we just do nothing? Or can we do something in the case that all conditions are evaluated to false?

In comes the if-else statement. It is literally translated as, if the condition is true, execute this code block, otherwise, execute this other code block.

```
if(SOME_CONDITION) {
    // DO THIS STUFF
} else {
    //IF NONE OF THE CONDITIONS ARE TRUE
    //THEN DO THIS STUFF
}
```

If we go back to the banking system idea where we only want to allow a user to take out money that they actually have in the bank, we can use an if statement to check if the funds are available. If they are, then execute and give them their money, otherwise, notify them that they do not have the funds.

```
if(accountBalance > withdrawRequestAmount) {
    // Give them their money
} else {
   // HEY, YOU HAVE NO MONEY TO TAKE OUT
}
```

5.2 if- else if- else

Let‘s continue with our really dumb bank example. Check out this code:

```
public class Bank {

    public static void main(String []args){
       int accountAmount = 1000;
       int withdrawRequestAmount = 1000;

       if(accountAmount > withdrawRequestAmount) {
           System.out.println("Heres your money");
       } else {
           System.out.println("GET A JOB, YOU NEED MONEY");
       }

       if(accountAmount == withdrawRequestAmount) {
           System.out.println("Heres your money");
       } else {
           System.out.println("GET A JOB, YOU NEED MONEY");
       }
    }
}
```

The first two lines of our main method, we just initialize accountAmount and withdrawRequestAmount to $1000. That‘s how much I have at the bank and just so happens to be how much I want to take out of the bank. The first if statement will print “GET A JOB, YOU NEED MONEY“ because accountAmount is not greater than withdrawRequestAmount. Since the if statement evaluates to false, my else block gets executed. The second if statement checks if they are equal and in this case they are, so it prints “Here‘s your money“.

Now wait a second. First the bank to tells me to get a job because I don‘t have enough money, and then it gives me my money? What is going on? Since these are all separate if statements, each one gets checked every single time the main method runs. Also note that I have put the same else block for all of my if statements.

But what if I just want to check if-statements and stop checking them once one is true. Or what if I want to use the same else statement for all the if statements, but I don‘t want to write it over and over. And how do I stop my program from printing those two outputs when I only want one.

In comes the if- else if -else statement. With these statements, the program will go through the if statements until one of them is true, and then it will skip the rest. If none of them are true, then it will execute the else block, which you only need to write once.

We can refactor the code to take advantage of if-else if-else statements.

```
public class Bank {

    public static void main(String []args){
       int accountAmount = 1000;
       int withdrawRequestAmount = 1000;

       if(accountAmount > withdrawRequestAmount) {
           System.out.println("Heres your money");
       } else if(accountAmount == withdrawRequestAmount) {
           System.out.println("Heres your money, you have none left now");
       } else {
           System.out.println("GET A JOB, YOU NEED MONEY");
       }
    }
}
```

We have the same setup as before, accountAmount and withdrawRequestAmount are initialized to $1000. The if statement evaluates to false, so it goes to the else if statement, which evaluates to true, so it prints out “Heres your money, you have none left now“ and then skips the else. Note that we can have as many else if statements as we want. Once the program finds one that is true, it will skip the rest.

6 switch Statement

The switch statement does the exact same thing that the if- else if- else statement does, just with different syntax. Sometimes if the the conditions are real simple, the if- else if- else gets annoying to write over and over, so you can use a switch statement which tends to be more compact. However, the case statements must be compile time constant. This means that they must be constant values such as numbers, strings, or other things. We cannot do relation comparison in a case statement except for “==“. If we need to do relational comparison, we are stuck with if- else if- else statements.

6.1 switch

The lectures use the form

```
switch(expression) {
  case x:
    // if x true do this
    break;
  case y:
    // if y true do this
    break;
  default:
    // if none are true do this
}
```

However, I prefer to use extra curly brackets for reasons that we will go into when we talk about scope. So I prefer to write it this way

```
switch(expression) {
  case x:
  {
    // if x true do this
  } break;
  case y:
  {
    // if y true do this
  } break;
  default:
  {
    // if none are true do this
  }
}
```

So we can modify our bank program to use constants so that we have an excuse to use the switch statement.

```
public class Bank {

    public static void main(String []args){
       int accountAmount = 1000;
```

```
        int withdrawRequestAmount = 1000;

        switch(withdrawRequestAmount) {
            case 1000:
            {
                System.out.println("Heres your money, you have none left now");
            } break;
            case 1001:
            {
                System.out.println("You dont have enough");
            } break;
            default:
            {
                System.out.println("GET A JOB, YOU NEED MONEY");
            }
        }

    }
}
```

This program looks at withdrawRequestAmount and goes down the cases until it finds "Heres your money, you have none left now".

6.2 convert if-else if- else To switch

Let's suppose we have a game menu. There are a few standard options like New Game, Continue, Settings and Quit. A user generally can only choose one at a time. So let's see what this could look like with an if-else if- else statement.

```
public class Menu {

    public static void main(String []args){
        final int NEW_GAME = 0;
        final int CONTINUE = 1;
        final int SETTINGS = 2;
        final int QUIT = 3;

        int optionSelected = CONTINUE; //get user input somehow
```

```
        if(optionSelected == NEW_GAME) {
            System.out.println("Create an new game");
        } else if(optionSelected == CONTINUE) {
            System.out.println("Continue an existing game");
        } else if(optionSelected == SETTINGS) {
            System.out.println("Change game settings");
        } else if(optionSelected == QUIT) {
            System.out.println("Quit the game");
        } else {
            System.out.println("No option selected");
        }
    }
}
```

In this code snippet you will see something we probably haven‘t used before, at least in this pdf, the “final“ keyword. If you are unsure of what this is, please checkout the Pre-Chapter subsection 1.1. Note that we hardcoded the selected option for the sake of simplicity. In reality you would want to get set the “optionSelected“ variable by getting a value from the keyboard or game controller or something like that. In this case “optionSelected“ is set to “CONTINUE“ which is set to 1.

It is often best practice to use named variables such as “NEW_GAME“ to represent the actual values. It is much easier to understand what “NEW_GAME“ means as opposed to some value when reading the code. Also note that the only relation operator used is “==“ and we are just comparing integer values. When executed this code goes down the if-else if-else statements until it finds a matching value or it reaches the end. So this code would print out “Continue an existing game“. The structure of this code (meaning we only use “==“ and comparing to one value at a time) is perfect for a switch statement.

Let‘s refactor this code to use a switch statement and let us change the option selected to Quit the game.

```
public class Menu {

    public static void main(String []args){
        final int NEW_GAME = 0;
        final int CONTINUE = 1;
        final int SETTINGS = 2;
        final int QUIT = 3;
```

```
        int optionSelected = QUIT;

    switch(optionSelected) {
        case NEW_GAME: {
           System.out.println("Create an new game");
        } break;

        case CONTINUE: {
           System.out.println("Continue an existing game");
        } break;

        case SETTINGS: {
           System.out.println("Change game settings");
        } break;

        case QUIT: {
           System.out.println("Quit the game");
        } break;

        default: {
           System.out.println("No option selected");
       }
     }
   }
}
```

This code would output "Quit the game". The two code snippets both achieve the same thing. Ultimately it will come down to preference as to which you would like to use.