

Conditionals

Conditionals will allow our programs to make choices rather than always doing exactly the same sequence of statements. The most common conditional is `if`, which can be used to choose whether or not to do a sequence of statements in the code, and is sometimes used with `else`, to choose between doing one or the other of two sequences. A conditional will check a boolean value and do one thing if the value is true, something else (possibly nothing) if the value is false.

Booleans

The boolean type is used to store data about whether something is true or false. Under the hood, we can use a single binary bit to store off/on to represent 0/1 or false/true.

In Java, the possible literal values of booleans are `false` and `true`.

In some cases, we would create a boolean variable and set its value directly to establish whether something is true

```
boolean dogCanFly = true; // magic dog, yay!  
boolean movieIsScary = false;
```

However, it is very common for booleans to get their values from expressions instead of being directly set like this.

Boolean Expressions – Relational Operators

A boolean expression is an expression that is evaluated and comes out as a single true or false value. Boolean expressions are usually written using **relational operators** to check for equality, greater than, less than, etc. We can also combine multiple boolean values in more complex expressions.

A relational operator is used in an expression between two values of the same type, and this expression evaluates to either true or false. Our standard relational operators are

operator	Called	true when
<code>a == b</code>	“equals equals”	a and b hold same value
<code>a != b</code>	“not equals”	a and b hold different values
<code>a > b</code>	“greater than”	a holds a larger value than b
<code>a < b</code>	“less than”	a holds a smaller value than b
<code>a >= b</code>	“greater than or equal to”	a holds a value larger or same as b
<code>a <= b</code>	“less than or equal to”	a holds a value smaller or same as b

Note that each expression has an opposite which is true when it is false and false when it is true. `a == b` is true when `a != b` is false and false when it is true. The opposite of `a > b` is `a <= b` and the opposite of `a < b` is `a >= b`.

`==` vs `.equals()`

The `==` operator always compares the values on either side. So if we use it between two variables, it compares their values. For primitive types, this behaves the way we expect. For reference types, we have to remember that their values are the memory addresses they store.

So if we create two separate objects, it won't matter if we set all their instance variables to the same values, the `==` operator will say they are not equal, since they are at different memory addresses.

So what if we say something like

```
String catStr = "Cat";
String dogStr = "Dog";
String cee = "C";
String comboCat = cee + "at";
boolean same1 = (catStr == dogStr);
boolean same2 = (catStr == comboCat);
// parens not required, just for readability
```

What value does `same1` end up with? false. That's what we expect, but it isn't happening because "Cat" and "Dog" are different sequences of letters, it is happening because they are at different memory addresses.

So, what value does `same2` end up with? false!

We think of `s1` and `s2` as both having the value "Cat" but that isn't true. String is a reference type, so they each hold a memory address. If we go to the memory address in `s1`, we will indeed find "Cat" and if we go to the memory address in `s2` we will also find "Cat" but `==` isn't looking at that, it is just looking at the memory addresses¹.

¹ Sometimes, `==` will *seem* to work for Strings. Specifically, if you said

```
String c1 = "Cat";
```

```
String c2 = "Cat";
```

```
boolean same = (c1 == c2);
```

then `same` it would be true. This is not because `==` is checking the letters, but because Java has a special system for literal String values that tries to save space by just making one of each String literal used in the program and reusing the memory address it was created at every time that literal comes up. So when it sees "Cat" the first time, it creates a String object at, say, MM100 and stores that in `c1`, and when it sees "Cat" again, it just stores MM100 in `c2` as well, to save space. Figuring out when it will be smart enough to do this can be tricky... so don't try! Just use `.equals` except when checking for nulls!

So we have this problem checking String equality because String is a class. But because String is a class, it can also provide a solution. The String class has a method called equals() that results in a boolean value. If we want to check if two String variables hold the same letters in the same order, we can instead tell one string to do its equals method on another String.

```
String catStr = "Cat";
String dogStr = "Dog";
String cee = "C";
String comboCat = cee + "at";
boolean same1 = (catStr.equals(dogStr));
boolean same2 = (catStr.equals(comboCat));
```

This time, same1 would come out to false, because "Cat" and "Dog" are different, and same2 would come out true because "Cat" and "Cat" are the same, even though they live at different memory addresses.

So, the rule is *almost* that we always use .equals instead of == for Strings. But there is a case where it is important to use == instead. Remember that null is the value for memory address that means no object has been assigned. So if we want to check if a String is currently null, we do want to check for the address, not the contents – if it is null then there *are* no contents, and trying to look at them will blow up our program with a null pointer exception.

Suppose we have a Bunny class with a name instance variable and we want to check whether the bunny's name has been set or still has the default null value.

```
boolean bunHasName = (bunnyVar.name == null); // correct
boolean badBunHasName = (bunnyVar.name.equals(null));
// error bait
```

The first line checks for whether the variable is null. The second line either comes out false or the program blows up with a null pointer exception.

So, use .equals with Strings *except* when checking for nulls.

What if we need to check for *inequality* of Strings. We still use .equals, we just need to know where to put the !. It goes out front.

```
boolean different = !(catStr.equals(dogStr));
```

If

the if structure uses a boolean to determine whether or not to execute a sequence of statements in code. An if is always looking for its boolean to be true. The if syntax in Java is

```
if (condition) {  
    // code to do only if the condition is true  
}
```

Notice that like a class or method, the if is a structure with curly braces², not a semicolon. The part inside the curly braces is called the *body* of the if, which we often casually call “in the if.” In the body of the if you could put any code that you could have anywhere else in a method – create variables, do operations, print, whatever.

The condition in the parens of the if must evaluate to a boolean value. In the case that boolean comes out true, Java will run all statements inside the body of the if. But when the boolean comes out false, Java will skip all of that and jump to the next statement after the end curly of the if. Booleans have no third options.

The condition in the parentheses of the if could be a single boolean variable, but is most often a boolean expression.

For example, if we already have variables for whether a submission was late, for a fee, and for a current balance, and we are deducting a fee from the balance as a penalty for late submissions, we could say:

```
if (submissionLate) {  
    System.out.println("Late Penalty: $" + fee);  
    funds = funds - fee;  
}
```

Either the submission was late or not, so `submissionLate` is either true or false. If it is true, we will print a message and deduct the fee, otherwise we will skip those statements and go on with whatever is next in the program.

Another example: if we already have variables `ponyPrice` and `funds`, and we want to celebrate buying a pony and pay for it if our funds are high enough to afford the pony’s price, we could write

```
if (ponyPrice <= funds) {  
    System.out.println("Buying a pony yay!");  
    funds = funds - ponyPrice;  
}
```

Either the `ponyPrice` is less than or equal to our current funds, or not, so this expression will evaluate to either true or false. If true, we will print and subtract the price from the fund, if false, the program will skip those two statements.

² Technically, you could write the if without the curly braces, but for this course I require curly braces on all structures for readability and to avoid some common errors.

If by itself is appropriate in the case when we have a sequence of steps that we either want to do or skip. This commonly happens if we have code that we *want* to do, but some requirement needs to be met (we have enough money to buy a pony) or if we have code that we only have to do if there is some problem (make the user pay a fee if their submission is late).

If *always* looks for its condition to be true. We can adjust how we write the condition to fit with this. Suppose instead of imposing a fee for late submissions, we want to celebrate those that aren't late. We can use our logical operator NOT to adjust for this, to check for the opposite of lateness being true:

```
if (! submissionLate) {  
    System.out.println("Thank you for submitting on time!");  
}
```

If Else

In many cases when our programs make choices, we aren't just choosing whether or not to do something, we instead have two options, and we want to do one or the other. In that case we would pair the if with an else. Now we will have one sequence of statements in the if, and another sequence of statement in the else.

The else is always checking the same condition as the if, but while the if is checking for true, the else is always checking for false. Since any boolean expression will always come out to one or the other, either the if or the else will win. Whichever one wins, we do those statements and skip the others. We would never have else without if.

```
if (condition) {  
    //statements to do if the condition is true  
    //"body of the if"  
} else {  
    // statements to do if the condition is false  
    // "body of the else"  
}
```

Notice that since the else is just a partner to the if, we usually write it right on the same line as the end curly of the if.

Let's decide that if we cannot afford a pony, we will go to work to make more money. We might say something like

```
if (ponyPrice <= funds) {  
    System.out.println("Buying a pony yay!");  
    funds = funds - ponyPrice;  
} else {  
    System.out.println("No pony today!  Back to work.");  
}
```

```
        funds = funds + wages;
    }
```

Now, if the condition comes out true we will do the two lines in the if, but skip the else, and if the condition comes out false, we will skip the two lines inside the if and do the else instead.

If always checks for true and else always checks for false, so we could always change the condition and do them in the other order:

```
    if (ponyPrice > funds) {
        System.out.println("No pony today!  Back to work.");
        funds = funds + wages;
    } else {
        System.out.println("Buying a pony yay!");
        funds = funds - ponyPrice;
    }
}
```

This has exactly the same result as the previous, since we swapped the condition to the opposite, but also swapped contents of the if and else bodies.

We often choose to write if-else with what we think is the more likely, or preferable, case in the if position.

Common Errors

When students are first learning java and using conditionals with a boolean variable they often think they need a relational operator, so they stick in an `== true`. Although this doesn't cause an error, it's pointless, adding an extra step for no reason.

```
// suppose we already have ints x and y and boolean b1 and b2

// NOT ((x == y) == true)
if (x == y) {
    System.out.println("Same ints");
}

// NOT (b1 == true)
if (b1) {
    System.out.println("yes b1");
}

// NOT ((b1 == b2) == true)
if (b1 == b2) {
    System.out.println("Same bools");
}
```

Having an `== false` in a condition is also not an error, but it is generally not how a programmer would write the code, they would use `!` instead.

```
// suppose we already have ints x and y and boolean b1 and b2
```

```
// NOT ((x == y) == false)
if (x != y) {
    System.out.println("Different ints");
}
```

```
// NOT (b1 == false)
if (!b1) {
    System.out.println("no b1");
}
```

```
// NOT ((b1 == b2) == false)
if (b1 != b2) {
    System.out.println("Different bools");
}
```

When we only have one option, then we only need `if` without `else` (it's an error to try to have `else` without `if`). This is right even if the thing we're checking for is something we think of as `false`.

```
// suppose we already have ints x and y and boolean b1 and b2
```

```
// NO, don't use else with empty if!
if (x < y) {
} else {
    System.out.println("x not less");
}
```


```
// YES, one option, just if
// could also write as (x >= y)
if (! (x < y)) {
    System.out.println("x not less");
}
```

Here are two common typos that aren't technically syntax errors, but are definitely not what you want to write

```
// semicolon ruins the if
if (x < y) ; {
    System.out.println("x is less");
}
```

Just as we can add parentheses for readability even when they are not needed, we can also add curly braces for readability. Putting a semicolon right after the condition of an `if` means

that the *if* *only* applies to that semicolon (it controls whether... nothing happens) and any statements in the following curly braces will just always happen, unaffected by the *if*.

```
// suppose b1 and b2 are booleans
// single = is assignment, not checking equality
if (b1  b2) {
    System.out.println("x not less");
}
```

It is technically legal to do a boolean assignment in the condition of an *if*, but it is not *checking* whether *b1* and *b2* are the same, it is *changing* *b1* to be the same as *b2*.

Nesting Conditionals

We can have any code inside the body of a conditional (*if* or *else*) that we would have anywhere else in code, including more conditionals, so we could have an *if-else* inside an *if* or an *if* inside an *else* (remember that we never have an *else* without an *if*). This allows us to say that we will only check for something in the case that we have already checked for something else.

Here is an example that covers checking all combinations of two conditions.

```
if (first_condition) {
    //statements here only depend on first being true
    if (second_condition) {
        // do if both are true
    } else {
        // do if first is true but second is false
    }
    // statements here only depend on first being true
} else {
    // statements here only depend on first being false
    if (second_condition) {
        // do if first is false but second is true
    } else {
        //do if both are false
    }
    //statements here only depend on first being false
}
```

Notice that there is only one structure checking *first_condition*, which we would call the “big *if-else*” or the “outer *if-else*” but there are two separate structures checking *second_condition*, which we might call the “small *if-elses*” or “inner *if-elses*”. Also notice that there is space here for code that is inside the body of the big *if-else* but not inside the small *if-else*, where we can put code that only depends on the first condition.

else if

It is very common to have a series of possible conditions or values we need to check, with a different behavior for each. We would check for the first one, then if that fails check the next, and so on. In that case, we could write a deep nesting of if-else structures:

```
if (first_condition) {  
    //behavior for first condition  
} else {  
    if (second_condition) {  
        //behavior for second condition  
    } else {  
        if (third_condition) {  
            //behavior for third condition  
        } else {  
            //behavior if none of those conditions holds  
        }  
    }  
}
```

This works, but writing it in this way implies that the first condition is the most important, and each subsequent condition is less important or less likely. If they are simply a set of equally important and likely options, we would usually simplify the code by writing an else with an if immediately after it, on the same line:

```
if (first_condition) {  
    //behavior for first condition  
} else if (second_condition) {  
    //behavior for second condition  
} else if (third_condition) {  
    // behavior for third condition  
} else {  
    // behavior if none of those conditions holds  
}
```

This visually implies that while we chose to check first-condition first, these options are roughly at the same level or importance or likelihood. It is also shorter and most people find it easier to read.

Notice that this example ends with an else, not a last if. So if none of the conditions we checked for holds, we will do the else.

If instead we ended with an else + if, then there would be the possibility that we would do nothing at all for this code, if nothing including that last if's condition turned out true.

Sometimes that is what we want

```
if (x < 10) {  
    System.out.println("negative or one digit number");  
}
```

```

    } else if (x < 100) {
        System.out.println( "two digit number");
    } else if (x < 1000) {
        System.out.println( "three digit number");
    }
    // when x is >= 1000 we just don't do anything

```

If the conditions are totally separate issues we are checking for, then ending in an else-if makes sense if we know that sometimes none of those situations holds so we have no actions to take. If the conditions are related and together cover all possibilities, we almost certainly want to end with an else which covers the last possibility.

This is particularly important in a toString.

Conditionals in toString

The job of the toString is to provide us with a nice way to print the content of our class' instance vars. In some cases, for instance when we have a boolean instance variable, or a numeric instance variable that won't be meaningful to the user, it is helpful to use a conditional in the toString.

The toString must always return some String, that is, each time the method runs, there must be exactly one return statement that runs. So we can handle conditionals in toString in one of two main ways: Multiple returns in different parts of the conditional *or* creating temporary variables and having a single return at the end.

Overall, most people prefer to just return at the end, but multiple returns can make sense if we are returning things with totally different formats in different cases. If we are returning the same format, just with different values, then temporary variables are definitely the better option.

Here is a class that uses two returns in its toString to give meaningful output when an instance var is a boolean

```

public class Monster {
    public boolean scary;
    public String toString() {
        if (scary) {
            return "woooooo scaaary monstah!!";
        } else {
            return "a safe friend-shaped monster";
        }
    }
}

```

When a method uses the return keyword, the statement it is in automatically ends the method. So this code does exactly the same thing.

```
public class Monster {
    public boolean scary;
    public String toString() {
        if (scary) {
            return "woooooo scaaary monstah!!";
        }
        return "a safe friend-shaped monster";
    }
}
```

In the second version, if the if runs, the return ends the method³, so we never get to the second return, and otherwise we skip that return and just do the second one. This is another example of expressing our thoughts in how we write functionally identical code: the first version presents scary being true and false as equal options, while the second implies that safe, friend-shaped monsters are the default, and being scary is something we have to check for.

Here's another class trying to use multiple returns in a toString

```
public class PieEater {
    public int piesEaten;

    //toString error! Java thinks it doesn't always return
    public String toString() {
        if (piesEaten < 100) {
            return "ate some pies";
        } else if (piesEaten == 100) {
            return "the pie champion!";
        } else if (piesEaten > 100) {
            return "ate possibly too many pies";
        }
    }
}
```

But this one results in a syntax error! Java will complain that toString doesn't always return!

We, as smart humans, can evaluate this logic and note that since any number is in fact less than, equal to, or greater than 100, this *does* always return one of the three possibilities. But the java compiler just checks the format, not the actual logic, and by formatting this as

³ This means there cannot be any statements that could happen after a return happens... but depending on ifs and elses, we can absolutely have statements *lower down* from a return, that could run if that return is in an if that doesn't run.

ending with an else + if, not just an else, our code is saying that there's a last case to check *and that that condition might be false!*

Again, if the conditions we check cover all the possibilities, we should end with an else.

```
public class PieEater {
    public int piesEaten;

    // correct toString, structure reflects logic
    public String toString() {
        if (piesEaten < 100) {
            return "ate some pies";
        } else if (piesEaten == 100) {
            return "the pie champion!";
        } else {
            return "ate possibly too many pies";
        }
    }
}
```

This version doesn't cause an error, because by ending with else, we are showing that if the number isn't less than and isn't equal to 100, it will be greater, and so we always return something.

Now let's do a version with temps and just one return.

```
public class Ghost {
    public int howScary;
    public String name;
    public boolean upstairs.

    public String toString() {
        // making howScary readable
        String scareTemp = "not a";
        if (howScary > 100) {
            scareTemp = " a very";
        } else if (howScary > 50){
            scareTemp = " a kinda";
        }
        // making upstairs readable
        String stairTemp = "downstairs";
        if (upstairs) {
            stairTemp = "upstairs";
        }
        return scareTemp + " scary ghost named " + name
            + " haunts the " + stairTemp + " of the house";
    }
}
```

Users would probably not know what we mean by a ghost being 73 or 44 scary, and it might be hard to work a “true” or “false” into a sentence, so instead of including the number or boolean directly in what we print, we use conditionals to convert them to words that are meaningful to the user. We created temporary String variables. Then we checked the values of `howScary` and `upstairs` to decide what values to set it to, then included these temps as parts of the String returned at the end, just as we included the name instance variable.

One thing to notice is that I ended up including the “a” within the temp instead of in the overall return, because sometimes I needed it first (“a very”) and sometimes after (“not a”) the word that was changing. It sometimes takes a little thought to figure out how the `toString` sentence will come together.

The temp string `scareTemp` started with the value “not a” and the conditional structure ends with `else + if`. This gives `howScary` a default value, and if it is 50 or below, it is never changed from “not a”. This makes the code a tiny bit more efficient, while implying that “not a” is the default. Notice that I could have made either of the other values the default by changing how I did the conditions.

Similarly, the temp for `upstairs` defaults to “downstairs” and is just changed if the `upstairs` instance var is true.

Depending on the complexity of our instance vars, we could end up having multiple temps *and* multiple returns, if our `toString` sometimes has multiple possible values in one format and sometimes multiple possible values in a different format.

Boolean Expressions – Logical Operators

We use **logical operators** to combine multiple booleans, for instance requiring that two booleans are true at the same time.

Our standard logical operators are

Java operator	called	true when
<code>!p</code>	“not”	p is false
<code>a && b</code>	“and”	both a and b are true
<code>a b</code>	“or”	a is true or b is true or both are true

We use `!` to swap between true and false. `!` in front of a false value comes out true, and `!` in front of a true value comes out false.

We use `&&` when we want two things to be true at the same time. We use `||` when we need at least one of two things to be true.

These operators give us the ability to combine multiple requirements that we want to check, all in a single boolean expression.

Truth tables are one way of seeing the behavior of an expression with logical operators. A truth table lays out a row for each possible combination of the individual booleans involved and then shows how the whole expression comes out. Here is the truth table for !

a	! a
true	false
false	true

Since ! only acts on one boolean, it is a very small table, just showing that the outcome of using a ! is the opposite of the value started with.

Here is the truth table for &&:

a	b	a && b
false	false	false
false	true	false
true	false	false
true	true	true

The table shows that an && expression is false in every case except where both elements are true at the same time.

Here is the truth table for ||:

a	b	a b
false	false	false
false	true	true
true	false	true
true	true	true

The table shows that an || expression is true in every case except where both elements are false.

```
if (age >=0 && age < 18) {  
    System.out.println("child ticket");  
} else if (x < 65) {  
    System.out.println("adult ticket");  
} else if (x < 150) {  
    System.out.println("senior discount");  
} else { // means x is in none of the above categories  
    System.out.println("invalid age, check ID")  
}
```

We earlier had code to check for all combinations of two conditions. If we want to check for two conditions at the same time, we could also use our logical operators. Here is code to check for all possible combinations of two booleans, using if-else-if structure:

```
if (first_condition && second_condition) {
    System.out.println("both are true");
} else if (first_condition) {
    System.out.println("first is true, second is false");
} else if (second_condition) {
    System.out.println("first is false, second is true");
} else {
    System.out.println("both false");
}
```

Note that we didn't have to check for `first_condition && !second_condition` for the case where first is true and second is false: if both were true, we would have stopped in the first if, so we know at least one is false when we get to the first else-if. So if we check `first_condition` and it is true, then it must have been `second_condition` that was false. Same holds for the second else-if. The last else doesn't need to check for them both being false; again, if we know we are covering all possible cases, we can just end with an else to cover the only case we haven't checked yet. So this would be silly and redundant:

```
if (condition1) {
    if (condition1 && condition2) {
        System.out.println("both are true");
    } else if (condition1 && !condition2){
        System.out.println("first is true, second is false")
    }
} else if (!condition1){
    if (!condition1 && condition2) {
        System.out.println("first is false, second is true");
    } else if (!condition1 && !condition2){
        System.out.println("both false");
    }
}
```

So that means this would be equally silly and redundant.

```
if (age >=0 && age < 18) {
    System.out.println("child ticket");
} else if (age >= 18 && x < 65) {
    System.out.println("adult ticket");
} else if (x >= 65 && x < 150) {
    System.out.println("senior discount");
} else if (x < 0 || x >= 150) {
    System.out.println("invalid age, check ID")
}
```

deMorgan's Laws

When writing longer boolean expressions it is important to know how these operators interact. The most important rules are *deMorgan's laws* about the interaction between `!` and `&&` and `||`. The overall guideline is that a `!` in front of another operator can be distributed to each operand, and then swaps the operators between `&&` and `||`.

So,

`! (a && b)`
is the same as

`(! a || ! b) .`

This should make sense if you think about `!` meaning the opposite: the opposite of “both of these are true” is “at least one of these are false.”

Similarly,

`! (a || b)`
is the same as

`(! a && ! b) .`

This should also make sense: the opposite of “at least one of these is true” is “both of these are false”

deMorgan's laws are often useful when you can easily describe the situation you *don't* want, but you are using a tool like an if that checks its condition for true, so we need to write accordingly.

If I have actions to do, but I can't do them if it is above 90 degrees and there are any raccoons present, I might think of it as `(temp > 90 && raccoon_count > 0)` but actually, I want to check not for this bad case, but for the opposite. So the good case is

`! (temp > 90 && raccoon_count > 0)`

We can re-write this using deMorgan's laws as

`! (temp > 90) || ! (raccoon_count > 0)`

If we think about how relational operators work, the opposite of `>` is `<=`, so this can also become

`temp <= 90 || raccoon_count <= 0`

Note that all of these ways of writing the condition are legal and valid. We should choose the one that most clearly expresses the way we are thinking about the situation.