Conditional Statements, Evaluating Boolean Expressions

Ryan Nixon

Decisions, Decisions

- Most if not all programs are going to need to change what they do according to conditions in the program. This is known as branching, or flow control.

- There are a few constructs in Java to help with these decisions, namely `if-else` statements, `switch` statements, `while` loops and `for` loops

- We're going to cover the statements today and the loops at a later date

Branching with If-Else Statements

- The if-else statement is comprised of two keywords, `if` & `else`.

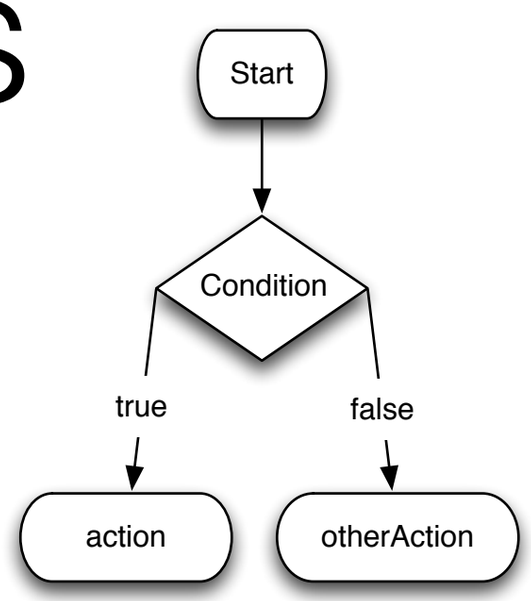
- `if` declares that some action be performed if a Boolean expression resolves to true

- `else` is optional and declares that an action be performed if the `if` statement preceding it resolves to false

- In this way, you are able to use `if` statements alone, but `else` statements require that an `if` statement be written first

Branching with If-Else Statements

- ```
  if (condition is true) {
       action;
  }
  else {
       otherAction;
  }
  ```

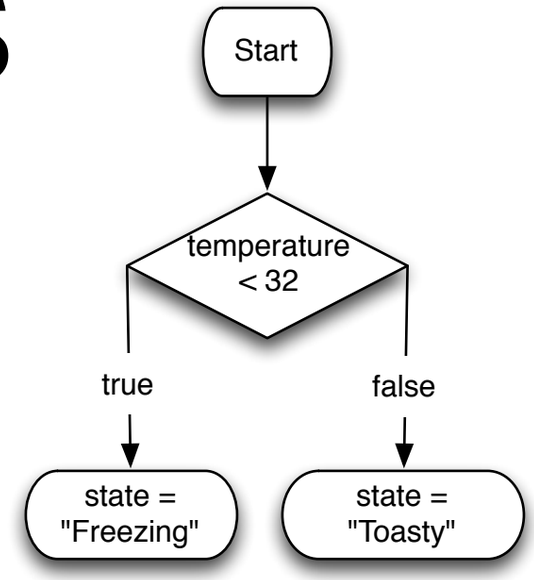


- As with classes and methods, `if-else` statements use braces to define the actions relating to them (their scope). These braces are only required if more than one action occurs, although it is recommended to always use them

- In the example above, `action` will only be performed if `condition` is true, otherwise `otherAction` will be performed
  ```

Branching with If-Else Statements

```
if (temperature < 32) {
     state = "Freezing";
}
else {
     state = "Toasty";
}
```



- It's very easily to think of situations where one thing or another may be done according to conditions. Above, weather conditions may play a part

- The above example reads, "If the temperature is below 32, then the state is freezing. Otherwise the state is toasty."

Branching with If-Else Statements

- In the weather example the < means "less-than" like you would see in a mathematical equation

- There is also > "greater-than", >= "greater-than or equal" (just like ≥ in math), and <= "less-than or equal"

- Lastly you use == and != for equals and not equals, respectively. You should *not* use a single =, that is the assignment operator and would assign a new value to whatever is on the left side of the expression

Boolean Comparisons

- Boolean expressions, just like the data type of the same name, are always true or false
- Using Boolean operators (>, <, >=, etc.) we can define complex conditions that boil down to this simple 1/0 yes/no true/false relationship
- You can also chain multiple Boolean expressions together with `&&` "and" as well as `||` "or". The `!` "not" operator can also manipulate the result, turning a true to a false
- These expressions follow a similar order of operation to math, allowing us to use parentheses to define which expressions are resolved first

Boolean Comparisons

- `&&` specifies that the expression to the left and the expression to the right *must* be true for it to return true. If either are false then `&&` will resolve to false

- ```
  if(bet == 42 && winner == 42)
      betAgain();
  ```
  ```

Boolean Comparisons

- || (double pipes) specifies that either the expression to the left or the expression to the right may be true for it to return true. This means that if both are true it will also resolve to true, but if both are false then it will resolve to false

- ```
  if(RED == BLACK || YELLOW == GRAY)
      seeTheOptometrist();
  ```
  ```

Boolean Comparisons

- `!` takes whatever expression is to the right of it and reverses its value. True becomes false and false becomes true

- ```
  if(!(proximity > 30) && !(workingBrakes))
      considerPanicking();
  ```

- Note that the above condition could also be `(proximity <= 30)` which would equate to the same value. `!` is more commonly used in variables than expressions since it is easier and more readable to write the statement without the `!`.
  ```

Booleans in If Statements

- When starting out with `if` statements, it's best to look at them from the inside out. Start with the parentheses, then the boolean operators, then the chains (`||`, `&&`, etc.) and work your way through to the outer parentheses, if any.

- ```
  if((17 < 34 || 12 == 8) && 42 >= 41.5)
  if((true    || false  ) && 42 >= 41.5)
  if((true              ) && 42 >= 41.5)
  if(true                 && true)
  if(true) //Perform action
  ```

- ```
  if('H' > 'I' || !(81 < 100))
  if('H' > 'I' || !(true    ))
  if('H' > 'I' ||  (false   ))
  if(false     ||  (false   ))
  if(false) //Skip action
  ```

Nested Branching

- Because `if-else` statements allow multiple actions to take place within their braces (scope) there is nothing stopping a programmer from placing an `if-else` statement within another. This is known as nested branching, or nested `if` statements

- ```
  if(halfACondition) {
      if(otherHalf) {
          action;
      }
      else {
          halfAFailure;
      }
  }
  else {
      utterFailure;
  }
  ```
  ```

Nested Branching

- Nested branching can get out of hand if overused. It is best to keep your nesting to a minimum

- You can use the Boolean chaining operators and "else if" to do this. Think of "else if" as just an `else` with an `if` statement in it; it allows you to check for a secondary condition if the first one fails

- ```
  if(halfACondition && otherHalf) {
        action;
  }
  else if(halfACondition) {
        halfAFailure;
  }
  else {
        utterFailure;
  }
  ```

- You may use as many else if's as you like, provided that you do not use an `else` (which resolves to all remaining possible conditions)
  ```

Boolean Values

- Remember the Boolean data type? It can be used in all of these prior examples. If you set the value of the variable to the result of a Boolean expression it can be used to represent that expression in `if-else` statements:

- ```
  bool gasLight = (tankState < 1.0);
  bool oilChange = (now < (lastChange + 90));
  if(gasLight) {
      illuminateGasWarning();
  else if (!oilChange) {
      breatheEasy();
  ```

- Note the use of the `!` not operator
  ```

Switch Statements

- If you are only checking for equality against one variable and you end up with a lot of else ifs, a `switch` statement may help organize your code better

- Switches do not add any new functionality, they just provide a different way of branching your code

Switch Statements

```
• switch(variable) {
      case 'A':
         actionA;
         break;
      case 'B':
         actionB;
         break;
      default:
         actionElse;
   }
```

Switch Statements

- Note the keywords `switch`, `case`, `break` & `default` and the use of colons.

- With a `switch` you run "cases" against a single variable, checking for equality. In Java these cases must be literals such as `"String"` or `5`.

- If a case matches then the code under it is run until the `break;` command is reached. Note that it will continue regardless of it passing other `case` statements. This allows one match to "fall through" to another (just like an `||` would)

- `Default` acts like the `else` statement, executing if none of the cases match

Comparing Objects

- Hopefully conditionals and branching should seem relatively intuitive so far. Unfortunately Java throws a a wrench in the works with objects

- As mentioned previously, objects such as `String` are often treated differently than primitive data types

- Branching statements are one of the biggest offenders of this. You cannot perform `if` statements on objects to compare their values

Comparing Objects

- `if(object1 == object2)`

- This statement compares the *memory locations* of the two objects. It will only return true if they are in fact the same object (albeit with a different name)

- This same behavior applies to the >, <, >= and <= operators. It's best to just avoid using these on objects

.equals(Object)

- `if(object1.equals(object2))`

- Instead, all objects in Java have a `.equals()` method. Programmers who create these objects can then define what "equals" means to them and return a more helpful Boolean value representing the state of the object

- Note that `.equals()` must be defined by the specific object, otherwise Java will fall back to memory location comparisons

.compareTo(Object)

- `int cmp = object1.compareTo(object2);`

- Some but not all objects are Comparable, which means that they have a `.compareTo()` method. This method is like `.equals()` but instead of returning true or false it returns an integer. This integer can then be inspected for the relative value of the passed in object against the source (the owning object for the method)

- If the int is 0 then the two objects are "equal" (like `.equals()`)
  If the int is > 0 then `object1` is greater than `object2`
  If the int is < 0 then `object1` is less than `object2`

- `compareTo()` like `.equals()` is completely up to the programmer to define, so be wary when using it

The Ternary Operator

- An unappreciated addition to the other flow control statements is the ternary operator, or conditional operator. This operator has a less intuitive syntax but is very useful in some situations

- Ternary operators allow very simple if-else statements to be reduced to a single line

The Ternary Operator

- `string variable = (condition) ? “true” : “false”;`

- In the above example, the variable is assigned the string “true” if the boolean condition equates to true, otherwise it is assigned false. It’s best to think of the `?` as the `if` and the `:` as the `else`

- Note that in ternaries both the `if` and the `else` are required, so the `:` and the following value must be provided for the statement to compile

Questions?