08 Making Decisions

[Close all sections] Only visible sections will be included when printing this document.

References: L&L 5.1–5.3, 6.1, 6.2

▽ Flow of control

The order in which program statements are executed by the computer is called the **flow of control**. The default order of statement execution through a method is sequential:

- each statement one after the other
- in the order they are written

Programming statements (constructs) can modify that order, to:

- decide whether or not to execute a particular statement (this section); or
- perform a statement over and over repeatedly (next section)

▽ Conditional statements

A conditional statement lets us choose which statement (or block of statements) will be executed next. Such statements:

- are also called selection or branching statements; and
- give us the power to make basic decisions

Java's conditional statements:

`if`	Do some action only *if* some condition is true
`if-else`	Do one action if the condition is true, do a different action if the condition is false
`switch`	Choose an action based on matching a value (each alternative is "labelled" by a different value)

▽ Comparing values to make decisions

Decisions are made based on the value of Boolean expressions (i.e., expressions that can be evaluated to `true` or `false`). At the lowest level, almost every Boolean value is the result of comparing one thing with another.

Comparison Operators

Equality operators	Meaning	Example
`==`	equal to	`7 == 5` is `false`
`!=`	not equal to	`7 != 5` is `true`

Relational operators	Meaning	Example
`<`	less than	`7 < 5` is `false`
`>`	greater than	`7 > 5` is `true`
`<=`	less than or equal to	`7 <= 7` is `true`
`>=`	greater than or equal to	`7 >= 5` is `true`

Comparing primitive values

- `int` (and `byte`, `short`, `long`): easy, all the equality and relational operators work exactly as defined.
- `char`: based on a character's position in the Unicode table of characters, so
  - `'a' < 'z'` is true, but
  - `'A' < 'a'` is true, because the capital letters come before lower case ones in the table, and so `'A' < 'z'` is also true
- `double` (and `float`) are not stored precisely, so `==` will not always work as expected; two values that look the same to us do not look *exactly* the same to the computer.
  - Rather than `==`, check if the *difference* is 'sufficiently small'
  - e.g., `Math.abs(d1 – d2) < 0.00001` (where 0.00001 is the degree of difference you are willing to accept given the application area of your program)

Comparing objects: `equals()` and `compareTo()`

The comparison operators really only work on primitive data: integers, characters, floating-point numbers and…

…object references (because behind the scenes these are integer-valued memory addresses). Why should we care?

```
String one = new String("Some text");
String another = new String("Some text");
System.out.println( one == another ); //prints false
```

In the above example `one` and `another` refer to two *different* objects in memory. The `==` operator only checks if the two references refer to the same object, *not* whether they represent the same *value* ("Some text"). In the more complex programs you will write in the future there will be times when this is useful—that is, your program will need to check that two object references, obtained from different places, are actually the same object in memory—but right now you are far more likely to want to know if two objects represent the same *value*.

The `equals()` method

All Java *objects* have the method `public boolean equals(Object o)` that can be used to test if two objects represent the same *value* (i.e., all their internal data has the same value). Given `one` and `another` from above:

```
System.out.println( one.equals(another) ); //prints true
```

But: in some classes, `equals()` is no better than `==` (the default implementation is very simple) so check the documentation for a class before assuming it will work.

The `compareTo()` method

`String`s have the method `public int compareTo(String o)` (if you're feeling adventurous, look up the Java API documentation for `Comparable`). This returns:

- a value *less than zero* if the first String belongs before the second (based on a character-by-character comparison);
- a value *greater than zero* if the first String belongs after the second; or
- exactly zero if they are identical.

Consider a new example:

```
String alpha = "alpha";
String beta = "alpha";
String gamma = "gamma";
System.out.println( alpha.compareTo(beta) ); //prints 0
System.out.println( alpha.compareTo(gamma) ); //prints -6
System.out.println( gamma.compareTo(alpha) ); //prints 6
```

Comparisons are case-sensitive. If you need a more lenient comparison that ignores the case (and so is more like comparing the position of words in a dictionary), then use the method `compareToIgnoreCase(String o)`.

Why do only some examples use `new String()`?

Java allows String objects to be defined by literal values (characters inside double quote marks, `"like this"`) as well as with the more general object creation keyword `new`. Most of the time, if you need a String and you already know what text it has to contain you should just use a literal value in double quotes, because it's simpler to write. But if do that with some of the equality testing examples here and from the lectures you'll find they don't work as described. This is because the Java compiler is clever enough to recognise when two String variables are initialised with the same text value, and it makes them the same object in memory!

It's not necessary to remember this quirk/feature of the Java compiler, only that if you're testing if two objects contain the same *value* then use the `equals()` method of one of them, whether you're working with String objects or some other class of object.

▽ Boolean Operators

Individual comparisons are useful, but what if we need to know that two or more things are *all* true, or we're satisfied if *any* of a number of comparisons is true? Boolean operators combine individual boolean expressions.

Operator English name	Java symbol	Meaning	(Highly artificial) examples
And	`&&`	Requires that *both* operands are `true`	`true && true` is `true` `true && false` is `false`
Or	`||`	Requires that *either* operand is `true`	`true || false` is `true` `false || false` is `false`
Not	`!`	Negates (flips) the operand	`!true` is `false` `!false` is `true`

Full operator precedence list

Arithmetic and Boolean operators can be mixed, since you may need to evaluate an arithmetic expression and *then* compare it against some other value (or the result of another arithmetic expression). The combined operator precedence list is, from highest to lowest:

Method calls	Methods that return values may be used in Boolean expressions (e.g., String's `length()` method). Method calls have high precedence since the surrounding Boolean expression cannot be evaluated until their outcome is known
`!` `+` `-`	negate, unary plus, unary minus (as in `+5` positive 5 or `-5` negative 5)
`*` `/` `%`	multiply, divide, modulo
`+` `-`	add, subtract, and also String concatenation (+)
`==` `!=`	is equal to, is not equal to
`<` `<=` `>` `>=`	less than, less than or equal to, greater than, greater than or equal to
`&&`	logical *and*
`||`	logical *or*

Parentheses () can always be used to override the default precedence or to make clear what is intended.

▽ `if`, `if` with `else`, and `switch`

`if`

Selects whether or not to execute a block of statements. Its syntax is the following (where the comments must be replaced by valid expressions and statements):

```
if ( boolean expression ) {
    statement(s)
}
```

A real-life example: You're in a car and would like the window open. Do you wind down the window? Only *if* it is not already open.

`if-else`

Selects between two alternative blocks of statements. Its syntax is the following (where the comments must be replaced by valid expressions and statements):

```
if ( boolean expression ) {
    statement(s)
} else {
    statement(s)
}
```

A real-life example: You need to get to work and you can drive *or* walk, depending on the weather. *If* it's raining you decide to take the car, otherwise (*else*) you walk.

Nesting `if`s (an example)

The *statements* inside an `if` or `else` block can be anything, including more `if`s. If the logic of your program is complicated, it may be simpler to break it down by nesting decisions. For example, given three numbers `n1`, `n2` and `n3`, which is the smallest?

```
//Assume n1, n2 and n3 already declared and assigned
if (n1 < n2) {
    if (n1 < n3) {
        //n1 is smallest
    } else {
        //n3 is smallest
    }
} else {
    if (n2 < n3) {
        //n2 is smallest
    } else {
        //n3 is smallest
    }
}
```

Sequences of `if`s

Sometimes, rather than doing one thing or another thing, there may be multiple alternative paths, based on a series of tests. The structure of an `if` statement for this is:

```
if ( boolean expression 1 ) {
    statement(s)
} else if ( boolean expression 2 ) {
    statement(s)
} else if ( boolean expression 3 ) {
    statement(s)
} else {
    statement(s)
}
```

Example

Each subsequent `if` is only checked if all preceding ones were `false`. An example, assigning a grade label based on a score between 0 and 100.

```
int score;
String grade;

//Assume score is assigned some value before...

if (score >= 80) {
    grade = "HD";
} else if (score >= 70) {
    grade = "DN";
} else if (score >= 60) {
    grade = "CR";
} else if (score >= 50) {
    grade = "PP";
} else {
    grade = "NN";
}
```

Guidelines for nesting or having a sequence of `if`s

- Decide on the 'set up' before the branch
  - what variables are needed to inform the decision?
  - where will there values come from?
- Work out the logic: what decisions need to be made and in what order?

Then, when finally implementing (coding) the solution:

- Use incremental development
  - start with the outermost `if`
  - always use blocks around the statements controlled by the `if`
  - test at each step (does the algorithm correctly distinguish between each alternative?)
- Use comments to keep track of what is happening or known at various points in the code

`switch`

Selects between many alternative sequences of statements by comparing a given value with the value for each *case*.

```
switch ( expression ) {
    case value 1 : statement(s)
                   break;
    case value 2 : statement(s)
                   break;
    .
    .
    .
    case value n : statement(s)
                   break;
    default: statement(s)
}
```

The `break` jumps to the end of the `switch` block. Without it, execution keeps going through each set of statements after the `case` where it entered.

The *expression* can be represent an integer value, character value, enum value or String. The `switch` statement is very versatile.

▽ ☑ Test Yourself: Boolean conditions and program constructs

Activity: Check your understanding of some Boolean expressions

Consider the following program fragment:

```
Scanner sc = new Scanner(System.in);
char c = 'Y';
String s = new String("whatever");
int i = 34;
boolean b = false;
```

Assuming the lines above were correctly placed in the `main()` method of a valid program, could the following expressions be used as the condition for an `if` statement? If *yes*, what would the result be, if *no*, why not? Create and fill in a table like the one below. Check your answers when you are done.

Expression	Valid?	Result/Reason
`c =! '2'`	Hide Solution No	Hide Solution Should be !=
`c < 'y'`	Hide Solution Yes	Hide Solution true (Unicode ordering)
`s == "whatever"`	Hide Solution Yes	Hide Solution false (s is a different object)
`s == s`	Hide Solution Yes	Hide Solution true
`s.equals("whatever")`	Hide Solution Yes	Hide Solution true
`s.compareTo("whatever")`	Hide Solution No	Hide Solution int not boolean
`sc.next()`	Hide Solution No	Hide Solution String not boolean
`sc.nextBoolean()`	Hide Solution Yes	Hide Solution true/false (depends upon value entered)
`b`	Hide Solution Yes	Hide Solution false
`b == false`	Hide Solution Yes	Hide Solution true
`b != true`	Hide Solution Yes	Hide Solution true
`b = false`	Hide Solution Yes	Hide Solution false (assignment returns value of right hand side)
`s.charAt(0) > c`	Hide Solution Yes	Hide Solution true (Unicode ordering has upper case first)
`s.length() == i`	Hide Solution Yes	Hide Solution false (8 ≠ 34)
`"cat" + "dog"`	Hide Solution No	Hide Solution String not boolean

Activity: Identifying appropriate conditional statements

Assuming the user has been asked for their name and has entered this into a `String` variable `name`, for each of the following scenarios, identify which conditional statement should be used:

a. Display their name prefaced with the word "Hello"
   ○ `if` ○ `if-else` ○ `switch` ○ nested `if` ● no conditional required

b. Display the phrase "Thank you sir" if the name variable begins "Mr"
   ● `if` ○ `if-else` ○ `switch` ○ nested `if` ○ no conditional required

c. Display the phrase "an odd name…" if the length of the name variable is odd and "an even name…" if the length is not odd.
   ○ `if` ● `if-else` ○ `switch` ○ nested `if` ○ no conditional required

d. Display the word "two" if there are two characters in the name variable, "three" if the name variable is three characters long, "four" if of length four, and so on, up to "nine". Only one message should be displayed.
   ○ `if` ○ `if-else` ● `switch` ○ nested `if` ○ no conditional required

Hide Solutions

Activity: Attempt the code tracing problems involving conditionals in the appendix of code tracing problems.

▽ `enum`erated types

Sometimes we need a variable to represent a very limited set of possible values. For example:

- Game difficulty: beginner, intermediate, expert
- Cardinal compass direction: N, S, E, W
- Student status: Pre-enrolment, Enrolled, Studying, Graduated
- Day of the Week: Sunday, Monday, …, Saturday

While we *could* use existing data types and define each possible value as a constant, this leads to problems. Let's consider some alternatives, illustrated using some of the examples above:

- Using Strings, we could define the game difficulty constants as

  ```
  final String BEGINNER = "beginner";
  final String INTERMEDIATE = "intermediate";
  final String EXPERT = "expert";
  ```

  and then to store a game difficulty

  ```
  String difficulty = BEGINNER; //OK
  ```

  but what stops us from doing this?

  ```
  difficulty = "Some random piece of text"; //Not OK
  ```

- Or using characters we could define the cardinal directions as

  ```
  final char NORTH = 'N', SOUTH = 'S', EAST = 'E', WEST = 'W';
  ```

  but there's nothing to stop us from doing

  ```
  char direction = '?'; //Not a valid cardinal direction!
  ```

- Or using integers we could define constants for the days of the week:

  ```
  final int SUNDAY = 0;
  final int MONDAY = 1;
  ...
  final int SATURDAY = 6;
  ```

In none of these examples do we really care about the *value* of the constant, only their *identity* (*which* game difficulty or *which* day of the week). So instead we can define a new *enumerated* data type. Inside the class block for a program but *outside* any method we would put:

- For game difficulty:

  ```
  public enum Difficulty { BEGINNER, INTERMEDIATE, EXPERT };
  ```

- For cardinal directions:

  ```
  public enum Direction { NORTH, SOUTH, EAST, WEST };
  ```

- For student status:

  ```
  public enum Status { PRE_ENROLMENT, ENROLLED, STUDYING, GRADUATED };
  ```

- And for days of the week:

  ```
  public enum Day {
      SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
  }
  ```

To refer to a value of an enumerated type use the pattern *TypeName.VALUE_NAME*, as in `Difficulty.BEGINNER` or `Direction.NORTH` or `Day.SUNDAY`. To declare a variable of an enum type we do the same as for other class types:

```
Difficulty challenge;
Direction facing;
Day dayOfWeek;
```

but unlike other class types, the instances of an enum are created automatically (and stored as static final attributes (i.e., constants) in the enum type), so we can assign a value to `dayOfWeek` with

```
dayOfWeek = Day.TUESDAY;
```

And we can use `==` to compare an enum variable against one of the possible values, as in:

```
if (dayOfWeek == Day.MONDAY) {
    System.out.println("I hate Mondays -- Garfield");
}
```

When using a `switch` with an enum type we can (must, in fact) omit the enum's name in each `case`, as in:

```
//Assign the correct name to dayNameGiven Day enum defined earlier
Day today = Day.FRIDAY; //would be set elsewhere
String dayName;
switch (dayName) {
    case SUNDAY: dayName = "Sunday";
                 break;
    case MONDAY: dayName = "Monday";
                 break;
    case TUESDAY: dayName = "Tuesday";
                  break;
    case WEDNESDAY: dayName = "Wednesday";
                    break;
    case THURSDAY: dayName = "Thursday";
                   break;
    case FRIDAY: dayName = "Friday";
                 break;
    default: dayName = "Saturday";
}
```

▽ Appendix: Full rules for Boolean operators

A "truth table" shows the result for each Boolean operator applied to different operands. That is, given the value of a boolean value `p` and another boolean value `q`, the table indicates the result of `p && q` (`p` *and* `q`), `p || q` (`p` *or* `q`) and `!p` (*not* `p`).

p	q	p && q	p || q	!p
T	T	T	T	F
T	F	F	T	F
F	T	F	T	T
F	F	F	F	T

Just like code tracing, creating a truth table for a Boolean expression you have defined in a program can be a useful way to confirm that it works correctly for all situations or to identify mistakes.

The following are additional rules that will be explored more in later units in your degree program, but they can be useful immediately.

Rule name	Meaning/effect
Idempotence	`a && false == false` `a || true == true`
Identity	`a && true == a` `a || false == a`
De Morgan's Laws	`!(a && b) == !a || !b` `!(a || b) == !a && !b`
Absorption	`a && (a || b) == a` `a || (a && b) == a`