# COSC 1020

## Yves Lespérance

## Lecture Notes

## Week 1 — Computer Hardware

Recommended Readings:
Horstmann: Ch. 1 Sec. 1 to 3, & Ch. 3 Sec. 1, 3, 7, 9
Lewis & Loftus: Ch. 1 Sec. 0 to 2, & Ch. 2 Sec. 0 to 4

## What This Course Is About

Computers are different from devices built for a specific purpose (e.g. a car). They basically process data, but they can be *programmed* to accomplish a wide range of data processing tasks.

A *program* is essentially a list of instructions telling the computer what operations need to be performed. It is analogous to a recipe for cooking a dish. But computers are stupid and the instructions have to be much more detailed.

In this course, we will learn how to *write programs*. We will use a particular programming language, *Java*, but the focus will be on general principles.

Writing and maintaining programs — *software* — is very expensive. Software systems can be very large and complex. Much of what we will learn will be techniques for managing this, what is called *software engineering*. We will also learn computer science techniques for ensuring that our programs are *efficient* and *correct*.



## Computer Hardware

Here are the main hardware components of a typical computer:

See Horstmann p. 8, Fig. 6.



Central Processing Unit (CPU): the brain of the computer, does all arithmetic and logical operations, keeps track of next instruction in program.

Main Memory — RAM & ROM: stores data and programs in binary notation; fast access, limited capacity, non-persistent.

Secondary Storage — hard disk, floppy disk, CD-ROM, tape, etc.: slower access, large capacity, persistent.

Bus: thick set of wires, allows data to be moved between CPU, memory, and other components.

Input/Output Devices — keyboard, mouse, monitor, speakers, printer, etc.: allows interaction with humans.

Network Connection.

## Algorithms

Before you can write a program to solve a problem, you need to develop a method that can be used to solve it. E. g.

**Problem**

Determine how many months it takes to pay back a loan given the loan amount, monthly payment amount, and interest rate.

**Solution Method**

1. Initialize *months_required* to 0.
2. Repeat (a), (b), and (c) while *amount_owed* is greater than 0:
   (a) Add *monthly_ interest* to *amount_owed*.
   (b) Subtract *monthly_payment* from *amount_owed*.
   (c) Increment *months_required* by 1.
3. Report *months_required* as the answer.



In general, a good solution method should be:

- *unambiguous*, i.e., leave no doubt as to which operation needs to be done a t each step,
- *executable*, i.e., doable on the computer, and
- *terminating*, i.e., be guaranteed to eventually come to an end.

A method with these properties is called an *algorithm*. You must have an algorithm to solve your problem before you can implement it in a program.



## Variables

Programs (and algorithms) use *variables*. E.g. `amount` and `numQuarters` in:

```
int amount, numQuarters;
amount = 78;
numQuarters = amount / 25;
amount = amount - numQuarters * 25;
...
```

These are somewhat like variables in mathematics — they can have a value.

But a variable in a program (algorithm) has a *particular* value *at a given time*, and it changes as the program is executed.

This is not the case in math. E.g.

$$x + y = 13$$

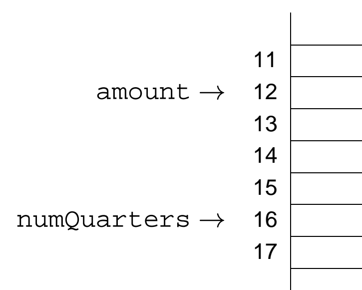
Variables $x$ and $y$ can have different values, but there is no concept of their *current* value.



To understand how variables in a program work, it is helpful to know a bit about computer memory.

Programs that are being executed and their data are stored in the computer's main memory.

Can think of main memory as a bunch of boxes that can hold values, which are represented as sequences of bits (binary digits). Each memory location (byte) has an address.



When a program is loaded into memory, each variable is associated with a particular memory location and its value will be stored there. The variable's name can be viewed as standing for the address of the associated memory location.

How much space is allocated for a variable depends on the type of value it can store. In Java, integers get 4 bytes, characters get 2 bytes, etc.



## Variables in Java

Each variable has three important attributes:

- a *name* that you chose;
- a *value* that you give it and that you can change; and
- a *type*, which limits what kind of value it can hold.

A variable name (identifier) can be any sequence of letters, digits, `$`,or _, but it cannot start with a digit and cannot be a reserved word. It's case sensitive. Follow style guide and use descriptive names!

The type can be a Java *primitive type* such as `int` (integers) or `boolean` (true/false), or an *object/non-primitive type* defined by a class, such as `Point` or `String`, or an array or `interface`.

A variable also corresponds to a particular location in memory, but you don't have to know which location that is.



## Declaring a Variable

When you want to use a variable in a Java program, you must first tell Java what its name and type are. This is called *declaring* the variable. E.g.

```
int age;
```

This declaration does the following:

- it reserves a storage location for the variable;
- it gives that location the name "`age`" (so you can refer to it later);
- it specifies what sort of values can be stored there, in this case, integers only.

Similarly,

```
Point p1;
```

declares a variable `p1` of type `Point`; its value can only be an object that is an instance of the class `Point`.



## Initializing Variables

A variable must be given a value before you try to use it (otherwise your program will produce nonsense). E.g.

```
int age;
IO.println(age);
```

Giving a variable a first value to start from is called *initializing* it.

You can declare a variable and initialize it in one statement. E.g.

```
int age = 28;
```

## Abstraction

In Computer Science/IT, we work a lot with *abstractions*.

An *abstraction* is a set of operations that is provided to some users. How the operations are implemented is hidden from the users. This is called *information hiding* or *encapsulation*. All that the user knows is how to invoke the operations (their names, parameters, etc.) and what results and effects the operations have. This is called the abstraction's *API (application programming interface)*. A user can use the abstraction without knowing the details of its implementation.

When developing software systems that may involve millions of lines of code, abstraction is very important.



## Abstract Data Types

An *abstract data type (ADT)* is:

- a *set of values* that belong to the data type, e.g. integers, strings, trees, etc.;
- a set of *operations* on these values, e.g. addition for integers, concatenation for strings, etc.

Users of the ADT are told:

- how the operations can be invoked (name, parameters, etc.);
- what the operation's *preconditions* are — what is required for the operation to be possible, e.g. for division, the divisor must not be 0;
- what the operation's *postconditions* are — what effects it has and what results it returns.

This is the API of the ADT.

How the ADT is implemented is kept hidden from the users.



## Primitive Numerical Data Types

The main primitive numerical data types in Java are:

- `int`, for positive and negative whole numbers, e.g. `33, 0, -100001`.

  An `int` is stored in a block of 4 bytes of memory (32 bits). Since there is a fixed amount of space, arbitrarily large (small) numbers cannot be represented. The range of values that can be stored in an `int` goes from $-2^{31}$ (or $-2,147,483,648$) to $+2^{31}-1$ (or $+2,147,483,647$). The numbers are stored in binary notation. These limits are denoted by the constants `Integer.MIN_VALUE` and `Integer.MAX_VALUE`.



- `double`, for positive and negative numbers with a decimal fraction, possibly written in exponential notation; actually called *double-precision floating-point* numbers, e.g. `1002.5, -5.3507, 8.55e248` (i.e. $8.55 \times 10^{248}$), `3.76e-7`.

  A `double` occupies 8 bytes of storage (64 bits). The number is first normalized (i.e. written so that the first non-zero digit comes immediately after the decimal point, e.g. $246.55$ becomes $0.24655 \times 10^3$), and the exponent is stored separately from the significant digits (in reality, this is done in binary notation). This allows a range of values from approximately $-1.7^{308}$ to $+1.7^{308}$ with 15 significant decimal digits of accuracy.

  So the range of values allowed is large, but precision is limited!

There are also several other primitive whole number data types in Java; the whole set is:

| type name | storage | range |
|---|---|---|
| `byte` | 1 byte | $-2^7$ to $+2^7 - 1$ |
| `short` | 2 byte | $-2^{15}$ to $+2^{15} - 1$ |
| `int` | 4 byte | $-2^{31}$ to $+2^{31} - 1$ |
| `long` | 8 byte | $-2^{63}$ to $+2^{63} - 1$ |

`byte` and `short` are used when memory use is a concern. `long` is used to to get a larger range than `int`. To write a `long` literal, append `L`, e.g. `3000000000L`.

There is also another floating-point type: `float`, *single-precision floating-point*, which uses less space but provides less precision than `double`. It occupies 4 bytes of storage and provides approximately 7 significant decimal digits of accuracy. To write a `float` literal, append `F`, e.g. `2.3F`.



## Defining Constants

Sometimes, a program has to use a special value, e.g.

5280 (feet per mile)
40 (number of students in a class)

Just using the value in a program is confusing to the reader. It looks like a "magic number". We would like to give the value a meaningful name.

In Java, constants are declared like variables, but the `final` keyword is added, e.g.

```
final int FEET_PER_MILE = 5280;
final int N_STUDENTS = 40;
final double CAN_US_EXCH_RATE 1.5152;
```

`final` means that once the variable has been assigned a value, it cannot be changed.



So what we do is define the constant once at the beginning, and then whenever we want to refer to the value, we use the constant's meaningful name instead.

Not only do such definitions make your program more readable, but they can save a lot of trouble. If the value needs to be changed, only one line needs to be edited.



## Characters

The basic building blocks of strings and text data are *characters*. In Java, these are the values of the primitive data type `char`. They include symbols such as letters, digits, and special characters such as "space", "newline", "backspace", etc.

There are several commonly used character sets. Java uses the *Unicode* character set (see Horstmann, app. 3), which employs 2 bytes per character and can represent the alphabets of most languages in common use.

Other commonly used character sets are ASCII and ISO-Latin-1 which use 1 byte per character. These are essentially a subset of Unicode.

Input/output methods will generally convert automatically between the different different sets as necessary. But reading/printing of non-ASCII/ISO-Latin-1 characters may not be supported.

Letters, digits, and other printable characters can be referred to by writing the character between single quotes, e.g. `'a', 'B', '9', '>'`.

Some special characters can be represented by an *escape sequence*, e.g. newline by `'\n'`, tab by `'\t'`, etc.

Other characters are represented using their hexadecimal code, e.g. `'\u00E9'` for é.

The `Character` class supplies many useful methods including `toLowerCase(c)`, `toUpperCase(c)`, `isLetter(c)`, `isDigit(c)`, `isLowerCase(c)`, `isUpperCase(c)`, `isWhitespace(c)`, etc.

Note that the `char` data type can also be used to store unsigned integers in the range 0 to $2^{16} - 1$.



## Character Strings

A *character string* is a sequence of 0 or more characters. A string can contain a word, a sentence, or any amount of text.

In Java, character strings are objects that are instances of the predefined class `String`. So `String` is not a primitive data type like `int`.

`String` literals are written between double-quotes, e.g. `"Hi there!"`, `"R2d2"`, `"Yves\nLesperance"`, `" "`, `""`.

E.g.

```
double temperature = 21.5;
IO.print("The temperature today is ");
IO.println(temperature);
```