Fundamentals of programming

Table of Contents

These notes are about some of the fundamentals of programming. They apply to all languages, but the details vary between different languages. They will make little sense by themselves. They need to be studied alongside writing and testing real programs in a language such as C, Java, Javacript or Python.

There are no tests or exercises here. It should be studied alongside one or more real languages, which should include programming exercises.

Data type

Programs handle data, with different types. The common types are:

• *integer* – whole numbers, like 35 and -7

• *real/float* – numbers with decimal parts, like 4.6 or 3.2. In mathematics these are called *real numbers*. In programming they are sometimes called *floating point types*, or just floats. That comes from how these values are represented in binary.

• *boolean* – these are true or false values

• *character* – examples of characters are alphabetic letters like a-z and A-Z, digits like ‘3’ and ’9’, punctuation like ‘!’ and so on. A space character ‘ ‘ is just another character. Check the difference between a variable name and a character value. For example

x=’f’

makes the *variable named x* to have the *value* character ‘f’. In many programming languages a character is enclosed in quotes, like ‘f’. Check we usually need to include other scripts, to handle characters like Ω and Ϋ and π.

• *string* – a string of characters, like ‘Hello boys!’

• *date/time*

• *pointer/reference* Data values are stored in memory. A pointer or reference is a way to find that value in memory – a pointer ‘points’ to the value. In some languages (such as C), a pointer is simply the address in RAM of the value - these are called raw pointers. Java and Python do not use raw pointers, because they are a common source of bugs.

• *records* - A record is a type which has several parts, called *fields* or *attributes*. For example, in a payroll application, for each employee we would need to manage their name, payroll number, their department, and several other fields. We might do this by defining a record type with these fields. Then we would make variables of this type, and assign to their name, payroll number and so on.

Pascal has a type called record to match this. C has a struct which is equivalent. Java and Python and JavaScript have objects, which are similar, but not identical, since they also have methods - code to do things.

• *arrays* - An array is a simple type of *data structure* – a way of referring to a set of data values in one thing. An array can be thought of as a set of numbered boxes. Then we can refer to the first box, the second box, the ninth box and so on. The box number is called the *index*. The values in an array are the *elements* of the array, and the index values often start at 0, not 1. The index is in [square brackets].

user-defined data types This means a type which is not built-into the language, but which is defined in a program.

In Python and Java and JavaScript, classes are user-defined type.

Basic concepts

A computer program is, mostly, a set of instructions which the computer executes. These instructions are usually called *statements*. There are different types of statements. Programs are combinations of

sequence (one statement after another, in correct order)

selection (conditional statements or 'ifs')

iteration (loops)

Comments

A comment is text in a program which the compiler or interpreter will ignore.

Comments are marked in different ways in different language. In Java and C and C++, a line comment starts // and runs to the end of the line. In Python a comment starts #. In html comments starts <!-- and end -->

Comments are used to

- Say the filename, date written, version number, author name, copyright information and so on
- Explain the code – say what a variable represents, what the purpose is, how to use it, why this solution is used.

Variable declaration

In many languages, a variable must be *declared* before it is used. This often includes saying what the *data type* of the variable is. For example in C:

```
int fileCount;
```

declares the variable named fileCount, and says it is an integer data type.

Why do some languages require this? Because

- It is easier for a human to read and understand code if it starts with a list of variables used
- It enables the interpreter or compiler to check for typos in variables (because mistyped variables will not have been declared, and the compiler can identify these)
- The type allows the interpreter or compiler to check for some errors. For example trying to multiply strings must be a mistake.

Constant declaration

A *constant* is a value which does not change. A constant is usually a literal, such as 3 or 7.2. You cannot change 3.

But a constant might also be given a name. For example in JavaScript we can say const PI = 3.1416;

Assignment

An assignment statement gives (assigns) a value to a variable. For example

```
y=x+2*(z+3)
```

When this is executed, the computer calculates the value of x+2*(z+3), and assigns that value to y

Selection

This is often called an *if statement*.

The computer will take different actions (execute different code) for different data values. For example

```
if y<2*z
  {
  a=1
  b=2
  }
else
  {
  a=4
  b=5
  }
```

If y is less than 2*z, a=1 and b=2 will be executed, and if not, a=4 and b=5. Usually the condition depends on other computations, so we do not know, when we write the program, which block will be executed.

This code is indented. That is, the if block and the else block are set in from the left edge. This is to make the structure clearer. In Python it is essential.

Another kind of selection is the *ternary operator.* For example

```
y = z<5 ? 4*z : 5*z;
```

Then if z is less than 5, 4z is assigned to y. Otherwise, 5z is assigned to y. We can think of it as - is z<5? If so, y is 4z, and if not, its 5z.

Iteration

Iteration means *repeating code*. Iteration is often called *looping*. There are different forms:

Definite iteration

This is when we can see, when we write the program, how many times it will repeat, like

```
for n=1 to 10
  .. do something
next
```

Indefinite – check at start

This is when we repeat code *while* some condition is true, and we check the condition at the loop start.

For example suppose we want to add up 1 + 2 + 3 +4.. until the total is 10 or more:

```
total=0
n=0
while total<10:
 total=total+n
 n=n+1
```

prin

After the loop, n is 5. It repeats the loop 4 times. At the end of the last time through, n becomes 5. But the total then is 10, so the condition total<10 is false, and the looping finishes.

Because we check at the start, we might get zero repeats. For example

```
total=0
n=0
while total>0:
 total=total+n
 n=n+1
```

Here the first time in, total>0 is false. So we do not repeat at all, and n does not change from 0.

Indefinite – check at end

In many languages this is done by do..while or repeat..until.

```
total=0
n=0
do
 total=total+n
 n=n+1
while total>10:
```

We do not reach the condition to the end of the loop. This means we execute the loop body *at least once*.

Nested selection and nested iteration structures

The statements in the body of a loop or conditional can be any type, including loops and conditionals. So we often have loops nested inside loops.

For example in a chess game we might need to do something for every square on the board. So we go through every row, and for each one, go through each column:

```
for row in range(1,9):
 for column in range(1,9):
   print("Row = ",row,", column = ", column)
```

Subroutines, procedure, functions

Program code is usually split into small blocks, called in different languages subroutines, procedures, functions or methods.

A subroutine can be called from other code. This means the path of execution switches to the subroutine, flows through it, then returns to the instruction following the call.

Data can be passed into a subroutine through parameters, and passed back through a return instruction.

One subroutine can call other subroutines from within it.

Execution starts in one block, and this calls other subroutines, which can call others. In this way code with many thousands of lines of code can be divided into subroutines each with twenty lines or less.

For example:

```
def average(x,y): # define a function named average
 result=(x+y)/2 # calculation
 return result # send back the result

x=4 # execution starts here
y=8
a=average(x,y) # call the function and assign to a
```

```
print(a)
```

Meaningful identifier names

Identifiers are the names of variables, functions and other things which can be chosen by the programmer (the rest are *reserved words*).

These should be chosen to say what the thing represents. For example, a function to calculate tax should be called findTax – and not func23. A function which deletes a file should be called something like deleteFile – and not df. A variable representing the current date should be called currentDate. The less imaginative you are, the better.

Why? Because it makes the code more readable (making more sense to a human more easily).

In turn that makes the code easier to write, bugs easier to find, makes it easier for the author or other programmers to understand, fix and extend the code, and reduces the need for comments.

Arithmetic

Programming languages offer the ability to do arithmetic. Addition subtraction and multiplication usually work as expected, with * the common symbol for multiplication.

A *digit* is a single symbol. The digits in base 10 are 0 to 9. 317 is one *number* with three digits.

There are usually two forms of division. Real or floating point division works as expected, so 3.0/4.0 is 0.75. *Integer division* gives the quotient only and ignores the remainder. For example 7 divided by 2 is 3 remainder 1, and we just get the 3 and lose the 1. There is a way to get the remainder, often written 7 % 2.

Exponentiation means raising a number to some power. This is often **. So for example 2**3 = 2 X 2 X 2 = 8.

Languages usually follow *BODMAS*. So 2+3*4 is 14, not 20. We use brackets to change the order. So (2+3)*4 is 20

Rounding means reducing the number of decimal places in a value, maybe to zero. So if we round 23.41 to 1 decimal place, we get 23.4. If the digit lost is 5 or more, we round up. So if we round 23.*48*, we get 23.*5*.

Truncation means just cutting off a number. So if we truncate 23.4999 to one decimal place, we get 23.4

Relational operations

These compare two values, like x > y for x is greater than y, and give a boolean, true or false, result

In most languages we have > greater than, < less than, >= greater than or equal to, <= less than or equal to.

Not equal to is usually !=

Testing for equality needs care. A = is an assignment, so we usually *use == to test if two numbers are equal.*

```
x=4
y=3+1
if (x==y):
 print("Correct")
```

Boolean operations

One boolean operator is AND. So x AND y is true only if x is true AND y is true.

Boolean operators work with boolean value and produce a boolean value – true or false.

OR means either. So x OR y is true if either x or y is true, or both.

NOT is reverse. Not x is true if x is false, and vice versa.

XOR is either but not both. So x XOR y is true if x is true and y false, or x is false and y true. It is the same as not equal to, with boolean values

```
x=4
y=6
print(x==4 and y==6) # true
print(x==2 or y==6) # true
print(not(x==3)) # true
print(x==4 != y==6) # same as xor - no actual xor in
Python
```

See the section about logic in mathematics fundamentals.

Constants and variables

A variable can change in value during execution, while a constant cannot.

In C we can have named constants like this:

```
#define LOOP_LIMIT 1000
..
count=0;
while (count<LOOP_LIMIT)
 {
```

```
 ..
 }
```

Why is this better than simply saying

```
while (count< 1000)
```

?

Because

- It is more *readable*
- We can *change* the loop limit just in one place. But if we have used 1000 in many places through the program, we would need to search and replace that every time

String-handling operations

A string is a sequence of characters. For example “black cat” is a sequence of nine characters, one of them being the space character.

Characters are part of a *character set* – examples are *ASCII* and *Unicode*. Each character in a set has a unique identifier integer, called its *code point* or character code. For example in ASCII the character code of “A” is 65, “B” is 66, “C” is 67, “a” is 96 and so on. The code of “0” is 48. “0” is a character, while 0 is a number.

Languages usually have a collection of facilities related to strings, such as:

length – find the length of a string eg “abcd” has length 4

position – find one string inside another eg “123xx67” contains “xx” starting at index 3 (index 0 is the first position)

substring – get part of a string eg “012cat67” starting at 3, length 3 is “cat”

concatenation – join two strings eg “black “ + “cat” is “black cat”

For example

```
myString="test string"
print (len(myString)) # 11
print( myString.find("rin") ) # 7
print( myString[1:4]) # est
print("one"+"two") # onetwo
```

We would also expect to be able to convert between characters, their codes, string and integer, string and float, string and date:

```
print( ord("A")) # 65 char code of A
print( chr(66)) # B character with code 66

x=213
s=str(x) # convert int to string
```

```
s="456"
x=int(s) # string to int

x=3.4
s=str(x) # float to string

s="45.6"
x=float(s) # string to float

import datetime # this is Python

x = datetime.datetime.now()
print(x, type(x)) # x is type datetime
s=str(x) # date to string
```

Random numbers

Random numbers are often useful. For example, we might want to test some code that sorts numbers into order, and we might want to test it with 1000 numbers. But typing in 1000 values is impractical. Better to generate those numbers by code with a random number generator.

Languages usually have features to generate random integers and random floats:

```
import random

for count in range(0,10): # 10 times..
 n=random.randrange(100) # random int 0 to99
 print(n)

for count in range(0,10): # 10 times..
 n=random.random() # random float between 0 and 1.0
 print(n)
```

Exception handling

Examples of exceptions are:

- An application attempts to read a data file (maybe the application is a game and the file is a saved game). But the file cannot be found – someone deleted it
- The user inputs invalid data – maybe a number with two decimal points like 3.4.56
- The application tries to load data over the web – maybe from the cloud – but the network connection is lost, because someone unplugged a router

An *exception* is

- Something that happens – an event

- Outside the program
- Which is unusual, non-standard, exceptional
- Which affects the program, in a negative way

Some exceptions can be handled. For example if the user inputs invalid data, we can tell them and ask them to try again. Others cannot – if for example computer memory develops a fault.

When an exception happens in a function, the program designers have three possible options -

- handle the exception – fix it
- pass the exception back to the function which called it
- give up and terminate the program

Some languages have a try..catch statement:

```
try
{
.. code which may result in an exception
}
catch
{
.. code to run if the exception happens. The code in the
try clause is rolled back – as if it never happened
}
```

Subroutine parameters and return values

Data values are passed *into* a subroutine through *parameters* – sometimes called *arguments*. Values are passed *out of* a subroutine (back to the code that called the subroutine) as a *return value*.

A subroutine can only return one value. If we need to return several values, they need to be put into a data structure – maybe a record, an object or an array

We use subroutines because:

- They make code more readable
- A single copy of the code can be called many times in the program
- They can be tested in isolation
- They can be re-used in libraries (code collections which can be used in many applications)

Local and global variables

A *local variable* has a value which can only be used within a subroutine.

A *global variable* can be used anywhere in a program.

Usually local variables only begin when execution of the subroutine starts and they disappear when subroutine execution ends. This is their *lifetime*.

Why do we use local variables? Because we do not need need to worry that a variable name used in one subroutine might clash with a variable with the same name in another subroutine. ‘Clash’ means accidentally altering the value in a bug. Real programs contain thousands of variables, but we do not need to remember them all if they are local.

Global variables are also discussed in the section on object-oriented programming.

Stack frames and subroutines

How does parameter passing and returns and local variables actually work?

The runtime maintains a *stack* in memory. Read about stacks under data structures.

For example:

```
1  def average(x,y):
2    answer=(x+y)/2
3    return answer
4
5  val1=int(input("Enter a number"))
6  val2=int(input("enter a number"))
7  av=average(val1, val2)
8  print(av)
```

When execution reaches line 7, there is a *function call*. Here the runtime:

- pushes the return address on the stack (line 7)
- pushes parameters (val1 and val2) on the stack
- switches execution to line 1 (subroutine start)
- at line 1, values for x and y are popped off the stack (val1 and val2)
- at line 2, a local variable is reached. This is set up on the stack
- also at line 2 the assignment is executed, with the result stored in ‘answer’, on the stack
- at line 3 we have a return. This happens:
    1. The local variable is popped off the stack. (the lifetime of local variable 'answer' ends)
    2. The return value (answer) is pushed
    3. The return address (line 7) is popped

4. Control switches to that return address

- At line 7, the return value is popped off the stack and assigned to variable av
- Execution proceeds at line 8 as usual

This is an outline. The details depend on the language used, and whether it is native code or interpreted source code

Why a stack *frame*? Because we need to use a block of data, not just pushing and popping a single value.

Why use a stack? Why not just store values somewhere in memory or in processor registers? Because we need to one subroutine to be able to call another, and that to call another, and so on, and for returns to always go back to where they came from, not just to one place. Using a stack means this will work.

Each call uses memory for the stack. It is possible that so many frames are pushed on the stack that the runtime runs out of memory available for the stack. This is called stack overflow. See the section on recursion.

Recursive techniques

A recursive subroutine is one which sometimes calls itself. We can also have recursive functions, procedures, methods and so on. It is a general programming technique.

For example, suppose we want to calculate the Fibonacci sequence:

1,1,2,3,5,8,13..

The first two numbers are 1, then we add the last two to get the next.

In pseudo-code this is

```
fib(n)
  if n=1 or 2 return 1
  else return fib(n-1)+fib(n-2)
```

Recursive code is usually like

```
if (.. something .. )
   here is the answer immediately (base case)
else
   recursive call
```

This means a call to this might mean a recursive call, so another recursive call, but eventually the if triggers, and we get the base case. Then the recursion unwinds.

If we do not, we get an *endless recursion*. These function calls use the stack, like they all do, so eventually we get a *stack overflow*.

Recursion is often the easiest way to write code. For example, suppose we want a function to delete everything in some folder. The problem is the folder might contain folders, and everything in them must be deleted. The solution in pseudocode is

```
function deleteAllIn(someFolder)
 for every thing in someFolder
   if it is a file, delete it
I  if it is a folder
     deleteAllIn(subfolder)
     delete subFolder
```

Programming paradigms

A programming paradigm is an approach to writing programs, using a set of ideas and techniques. Three commonly used paradigms at present are

- procedural programming
- object-oriented and
- functional programming.

Different languages handle these paradigms to different extents, and in slightly different ways. Some languages, such as Javascript and Python, can be used in all three paradigms. C is entirely procedural, and Java is primarily object-oriented. Each paradigm has advantages and disadvantages.

Procedural programming

In procedural programming code is organised into sections known as sub-routines, functions or procedures. Execution starts somewhere (in C and C++ and Java, at a section named main). This initial code usually contains *calls* to other sub-routines, which can call others in turn. When a sub-routine ends, control *returns* to the statement after the call. When main ends, the program ends.

Procedural programming reflects a *top-down approach* to problem-solving. We start with the overall problem, and split it into parts. These sub-problems are smaller and easier. They can be broken down in turn into smaller problems.

For example the task might be a supermarket website for delivered purchases:

Supermarket website								
Register new customer	Edit customer details			Place order				
	Login	Edit profile	Logout	Login in	Book delivery	Place order	Pay	Logout

Each of these tasks can be split further. For example, logging in requires

```
get customer ID and password
check in database
accept or reject
```

This would be in a client-server system. The user interface would be in web pages, so html and CSS and Javascript. Server-side code might be in PHP, accessing a database server.

Each task might be a sub-routine, calling other sub-routines for sub-tasks.

Data flows into sub-routines through parameters, and back out through return values. A stack mechanism is used to make it work.

Object-oriented programming

Object-oriented programming (OOP) has a central idea - an *object*. An object is a bundle of data fields, and code units. An object *encapsulates* relevant data and code into a single thing. The bundle is more or less protected (encapsulated) and can only be accessed in limited ways.

An object has a set of members. These members are *data fields* or *methods* (code blocks, like functions). The data fields are like ordinary variables, except that each object has its own set of values. The data values give an object *state*. If we change the data values, the object changes state. Sometimes an object is *immutable* - it cannot change state after being created.

Why OOP? Two reasons

1. To avoid global data. Usually in procedural programming we have global data, and a set of functions. Any function can access the global data. A bug in any function might corrupt the global data (make it invalid) and in turn this might make other functions to fail. This makes testing very inefficient -any new code in one function requires all other functions to be re-tested. In OOP each object can only access its own data, so we can test this in isolation.
2. It is often easier to think about areas of application in terms of things (objects) doing various actions (methods). For example in a graphics applications the things might be circles, rectangles, lines, text blocks and so on, each with position and colour, which the user might delete or copy of move or stretch and so on. OOP naturally maps these things to objects.

In most languages we can define a *class* to be a *type of object*. Then we have many objects of that type. For example in a graphics program we might have a

Circle class, and then many circle objects with different sizes, positions and colours. A class is a *definition of a type*. Its not a container or data structure like a queue or tree.

We say an object is an *instance* of a class, and making an object means instantiating the class.

Access modifiers

In some languages it is possible to declare class members to be *public or private*. This applies to methods and data fields. A private member can only be accessed from within the class - that is, from code in a method defined in the class. So it can only be used internally. A public member can be accessed from anywhere.

This is usually combined with public *getter and setter* methods. A getter methods provides read access to a private field - as for example

```
public int getX()
{
  return this.x;
}
```

in Java, letting code read the value of the x field from outside the class. A setter method provides write access, like

```
public void setX(int value)
{
if (value>0)
  this.x=value;
}
```

A setter means code can change a private field, but the change should be *validated*, so only valid values can be set. In the example, x should be greater than 0, and only such a change is allowed.

So this is how encapsulation is being enforced

1. data members are normally private, and
2. access is only allowed through setters that validate changes

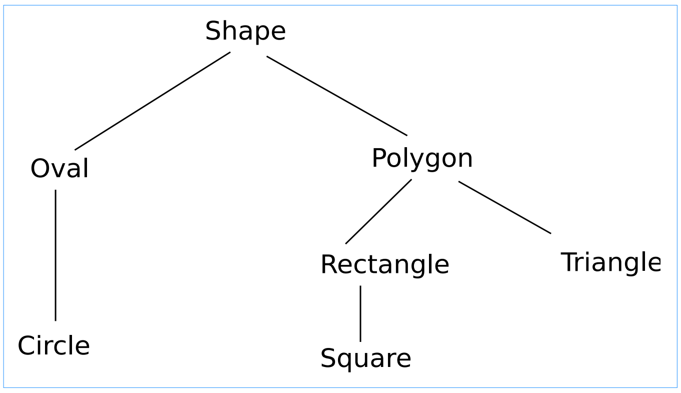
Java also has *protected*, which allows access for sub-class in the same or different packages.

The Python language has no way to enforce access modifiers. But there is a convention that fields that start with an underscore character, like _color, should be treated as internal and should not be accessed directly - because it might break the class, and might change in future versions.

Inheritance

The idea here is to *re-use code*. Having defined a *base class*, we can define a *sub-class* which inherits from it. Then the sub-class has all the members of the base class. Each can be changed (*over-ridden*) and new members added.

In our graphics program we might define an Oval class. But a Circle is a kind of Oval, where the width and height are the same. So when writing the Circle class, we do not start from nothing. we start from (sub-class) the Oval class. A Square is a kind of Rectangle. So we might have a *class hierarchy*.



A Shape might have a colour, a center position, a width and a height. An Oval and a Polygon would inherit those. A Polygon would have an extra field - numberOfSides. This would be 3 for a Triangle, 4 for a Rectangle.

Some of these classes do not make sense to instantiate. We can draw an Oval or a Circle, but we cannot draw a Shape - because we do not know what shape it is. We can draw a Rectangle, but not a Polygon, if we do not know how many sides it has. So Shape and Polygon should be *abstract classes*, which cannot be instantiated.

We can change the methods inherited from a base class simply by re-defining them in a sub-class, with the same name. So when code says

someObject.someMethod()

the runtime will execute the correct version of someMethod - the base class method if someObject is an instance of the base class, or the over-ridden sub-class version if someObject is a sub-class instance. This is called *polymorphism*.

Methods like this, which can be over-ridden in sub-classes, are called *virtual methods*. Often which verion to run is not known until runtime, and this is called dynamic despatch. In Java and Python, all methods are virtual (unless they are final in Java, or static in Python).

Composition and aggregation

We explain with an example. Suppose we have a College class. Colleges have departments, so College objects would have fields for Departments. Department would also be a class, and instances would be Department objects. So a College object contains Department objects.

This is called *composition* - when an object has data fields which are themselves objects. This is a one type of connection between classes. We say a College *has-a* Department.

A different type of class relation is called *aggregation.* If our college closes, its departments will cease to exist. But not the students. They will continue to exist. They exist independently of the college - not true of the departments. Aggregation means placing independent objects as fields in some other objects.

Object-oriented design

Object-oriented design (OOD) means designing a set of related classes suitable for some application usually to support an *information system*.

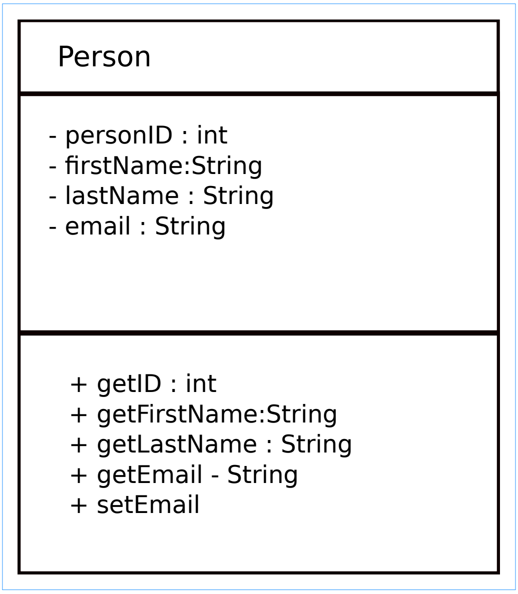
In OOP we code a model within the computer of a real-world system, such as a library, a student records system, a doctor appointments system and so on. Objects model real-world things. Classes model types of things. Methods models actions - what things do.

Creating this model is called OOD. It is the first stage in writing OOP code.

OOP code would use classes which are built-in either to the language itself, or in standard libraries. But OOD produces classes appropriate to the particular application area.

UML

Universal Modelling Language (UML) is a standard for sets of diagrams setting out the structure of informations systems. UML is very extensive, meaning it covers many kinds of diagrams for diffeernt aspects of information systems. We focus on class diagrams, showing what OOP classes are used and how they are related. 'Universal' means it applies to any programming language, not limited to any specific one.

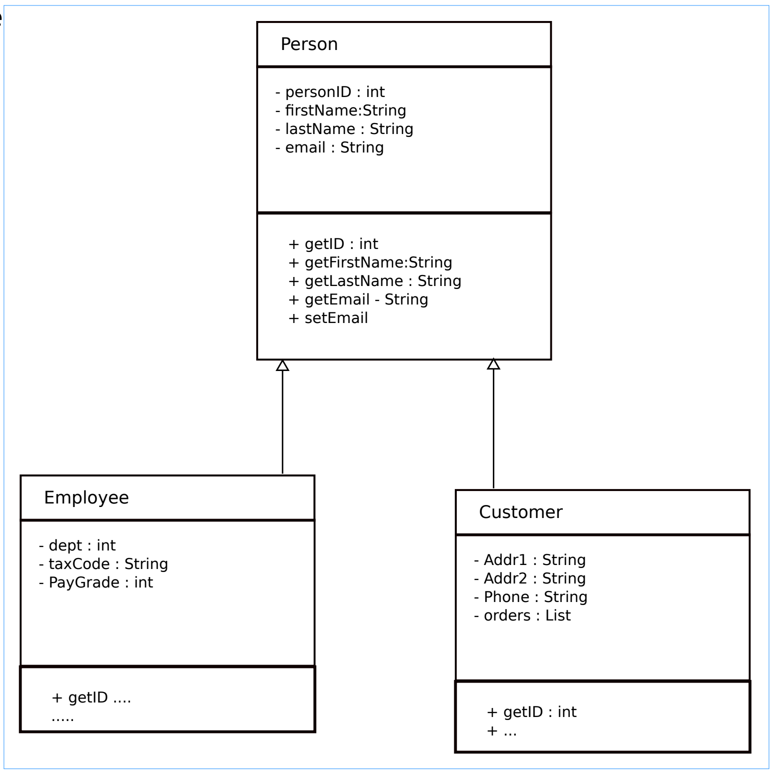


A class is shown as a box divided into three parts. The top part is the class name, in the middle goes fields, and at the bottom methods. For example this shows a class named Person

It has 4 fields, personID, first Name,lastName and email. The *- means these are private.*

It has 4 methods - getID and so on. The *+ means these are public.*

# would mean protected.



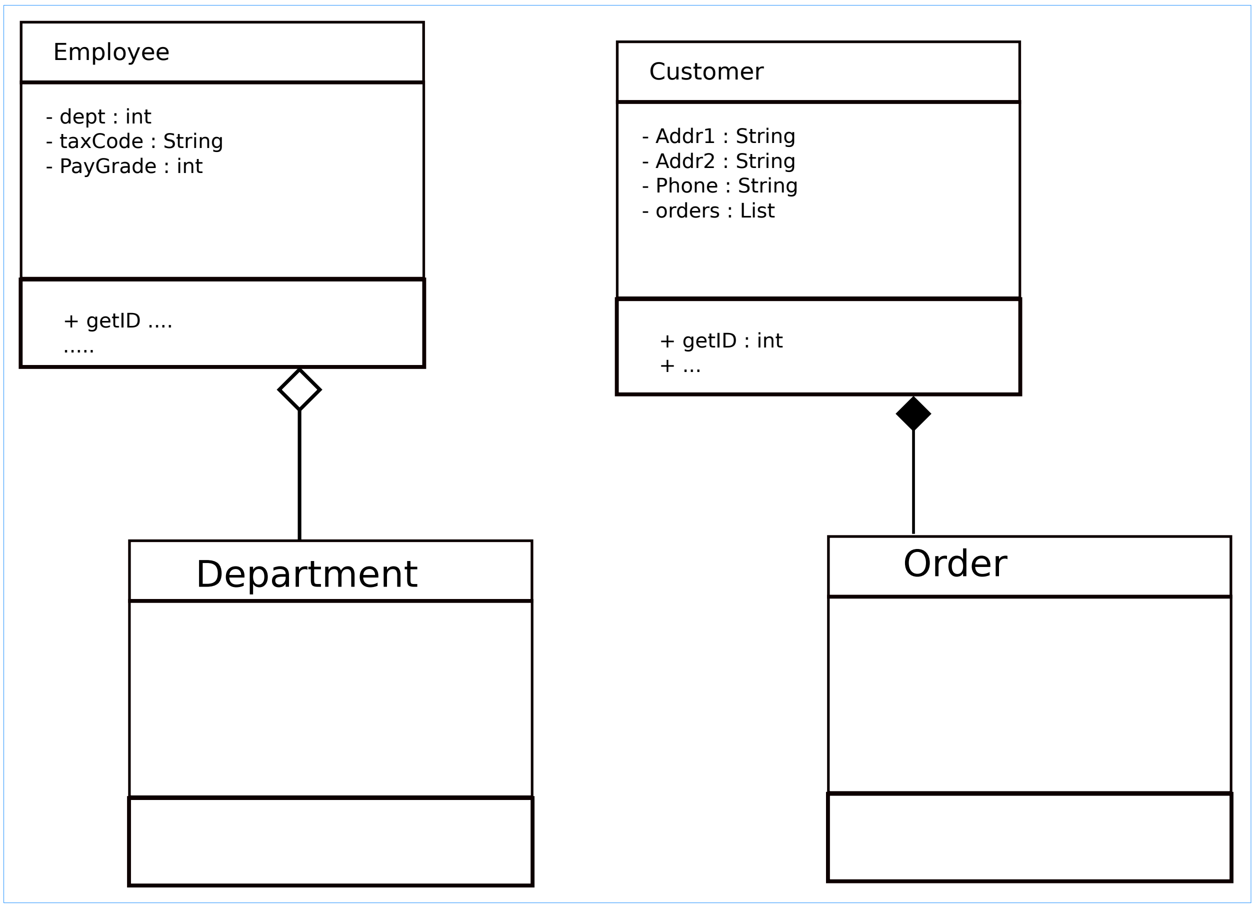
We can also show inheritance between classes. For example:

Here the Employee class and Customer class are sub-classes of Person, so both inherit personID, firstName and so on.

They also have additional fields and methods as appropriate.

Note the *lines with arrows for inheritance.*

We can also show composition and aggregation:



Department to Employee is *aggregation - white diamond*. If an employee is fired, their department still exits.

Order to Customer is *composition - black diamond.* We cannot have an order without a customer placing the order.