Unit 2b: Typing Model

1 Introduction

A number of typing models exist : static or dynamic; strong or weak; manifest or implicit; and, language-specific idiosyncrasies.

First let's consider the purpose of variables.

1.1 Variables and Typing

Most programming languages have the concept of *variables*: a variable is a storage location associated with a symbolic name, or *identifier*, that contains some piece of information, or *value*; a variable has a data *type* associated with the kind of value it contains.

Types classify values and also determine what operations are valid on a variable of that type. Depending on the language, types can include various numeric types, characters, strings, Booleans, etc.

In an object-oriented language, types can also include references to objects, and in a "pure" object-oriented language all types are represented as objects. Many (but not all) object-oriented languages allow new object types to be defined using classes.

2 Static typing

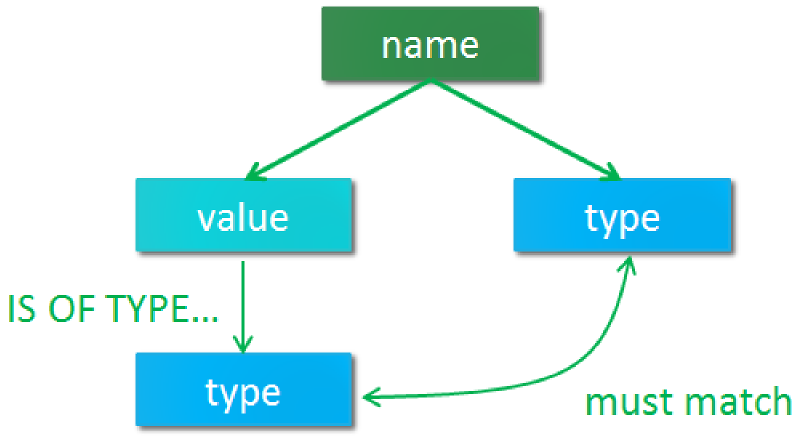
Statically typed languages are those which define and enforce types at compile-time e.g. you could declare the following variables in some statically typed language:

```
int i = 5
string s = "4"
```

Each declaration specifies the type of the variable and assigns a value to it of the appropriate type, and the code compiles successfully. Any of the following would result in a *compiler error*, i.e. program cannot run until error fixed:

```
int j = i + s       invalid operation
i = "10"            assigning invalid value to previously declared variable
string t = 10       assigning invalid value when initialising variable
```

You can think of a variable as being ***bound***, at compile time, to a type through declaration. It can be bound to a value at compile time or at run time via an assignment statement; it has a null value if not bound. In static typing, once a variable name has been bound to a type it can be bound only to values of a matching type; it cannot ever be bound to a value of a different type.



2.1 Manifest & Implicit typing

Statically typed languages may have different ways in which the types of variables are identified at compile time. M*anifest typing* is explicit identification by the programmer of the type of each variable being declared e.g. if variable *i* is going to store integers then its *type* must be declared as integer:

```
int i = 5
```

Some programming languages use *implicit typing*, or type inference, where the type is deduced from context at compile-time, for example the type to which the variable is bound may be determined by the compiler from the value to which it is bound:

```
var i = 5
```

This can make code significantly more compact and readable when type names are long.

3 Dynamic typing

In contrast, dynamic typing leaves type checks until runtime; dynamic implies the possibility of changing as the program runs. Typically in a dynamic language you don't have to declare variables, just assign values to an identifier:

```
i = 5
s = "4"
```

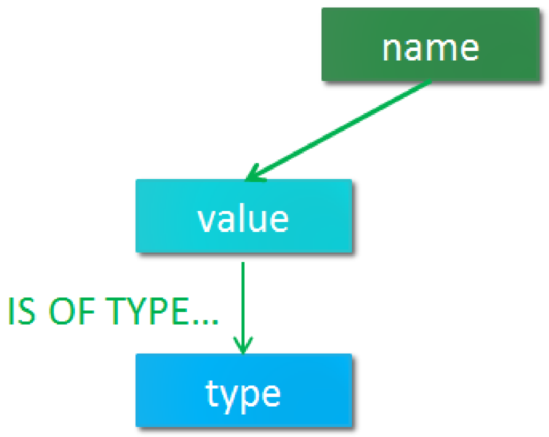
The following might (depending on the specific language) cause a *run-time error*:

```
x = i + s
```

The same identifier can be reassigned to a different type of value, so the following would be valid:

```
s = 3
```

Every variable name is (unless it is null) **bound** only to a value. Names are bound to objects at execution time by means of assignment statements; it is possible to bind a name to objects of different types during the execution of the program.



4 Static vs Dynamic

Dynamically-typed languages are more flexible: they can save development time and produce more compact source code; they are useful for rapid prototyping and experimentation, and, have advantages in situations where program behaviour is truly dynamic, e.g. processing data sets where structure is not known a priori.

However, lack of compile-time checking can lead to errors that are difficult to identify e.g. assume that the intention here is to assign a value to *number* and then update its value using a calculation, but the developer misspells the variable name in the second line:

```
number = 5
numbr = (number + 15) / 2
```

Static – assuming *number* previously declared, compiler will give error as *numbr* is undeclared

Dynamic – no error, will simply create new variable *numbr*, however, the value of *number* will not be updated as intended, which may lead to incorrect behaviour later

5 Strong typing

In a "type-safe" language, any attempt to misinterpret data is caught at compile time or generates a well-specified error at runtime. Strong typing is a mechanism that helps to enforce safety: a variable may be used only in ways that respect its type —for example, it is impossible to perform string operations on an int value e.g. the following would cause a compiler error or runtime error because there's no defined way to "add" strings and numbers to each other:

```
i = 5
s = "4"
x = i + s
```

Strongly typed languages often have guarantees on runtime behaviour to ensure that no value can be interpreted as something that it is not e.g. if your program tries to access an array element:

```
myArray[i]
```

where i > the length of the array, then a runtime error or exception would be raised.

Strongly typed languages can be *either* static or dynamic. Python is an example of a strongly typed dynamic language, while Scala is a strongly typed static language. The static/dynamic distinction refers to when type-checking is done; strong/weak refers to what can be done with variables depending on their type.

6 Weak typing

Weakly typed languages allow the type system to be subverted in some way – there are loopholes in the type system e.g. some languages do implicit coercion of variable types so that they "just work" i.e. change the type of a variable to suit the context in which it is used:

```
i = 5
s = "4"
x = i + s
```

With implicit coercion the type of one variable may be coerced so that they can be combined, giving a value for x of, for example, "54".

Note that in strongly-typed languages there may be some type conversion between certain related types where it is safe to do so, for example coercion of integer value to floating point value when combined in an expression, and explicit conversion by casting may be possible by casting or use of a type conversion function.

Another "weak" characteristic is the possibility of interpreting a variable as something that it is not, which can be "unsafe" e.g. C/C++ allow very direct access to memory through pointers, and allow pointer type casting without checking, also no array bounds checking – these can lead to unpredictable results when reading from memory and memory corruption when writing to memory, can cause crashes or security vulnerabilities.

Definitions of strong and weak typing are not precise or widely agreed, best to think of these as general characteristics which may be found to a greater or lesser extent in a particular language. In general weak typing means that (to some extent) the programmer doesn't need to worry about exactly what type each variable is, whereas with strong typing the programmer needs to make sure that a variable or value has an appropriate type for any operation that needs to be applied to it.

7 Strong vs Weak

Weakly-typed languages are more flexible: can save development time and produce more compact source code (same comment as for dynamic but remember that weak and dynamic are *not* the same thing); can pass any value as a parameter of a function (widely used in callback functions in JavaScript, for example); and, can be very powerful in expert hands, e.g. optimisations through direct manipulation of memory in C.

However, lack of type-safety can lead to unexpected or dangerous program behaviour.

8 Summary

The devil's in the detail: static/dynamic, strong/weak are general terms that can't be used to fully describe the behaviour of a specific language. Each language has its own type system, and these can differ in very obvious ways or in some much more subtle ways. You need to learn how the language you are using works – don't make assumptions e.g. the key word *var* is used to declare variables in a number of languages:

```
var i = 5
```

but this can have a very different meaning in each, for example:

- *C#* - this is static type inference, compiler will infer that i is an integer
- *JavaScript* – variables are dynamic, can declare with or without var, affects variable scope, not type
- *Scala* – a different meaning which you'll see shortly