Unit 2: Variable and Type

After this unit, students should be able to:

- appreciate the concept of variables as an abstraction
- understand the concept of types and subtypes
- contrast between statically typed language vs. dynamically typed language
- contrast between strongly typed language vs. weakly typed language
- be familiar with Java variables and primitive types
- understand widening type conversion in the context of variable assignments and how subtyping dictates whether the type conversion is allowed.

Data Abstraction: Variable

One of the important abstractions that are provided by a programming language is the *variable*. Data are stored in some location in computer memory. But we should not be referring to the memory location all the time. First, referring to something like `0xFA49130E` is not user-friendly; Second, the location may change. A *variable* is an abstraction that allows us to give a user-friendly name to a piece of data in memory. We use the *variable name* whenever we want to access the *value* in that location, and *pointer to the variable* or *reference to the variable* whenever we want to refer to the address of the location.

Type

As programs get more complex, the number of variables that the programmer needs to keep track of increases. These variables might be an abstraction over different types of data: some variables might refer to a number, some to a string, some to a list of numbers, etc. Not all operations are meaningful over all types of data.

To help mitigate the complexity, we can assign a *type* to a variable. The type communicates to the readers what data type the variable is an abstraction over, and to the compiler/interpreter what operations are valid on this variable and how the operation behaves. In lower-level programming languages like C, the type also informs the compiler how the bit representing the variable should be interpreted.

As an example of how types can affect how an operation behaves, let's consider Python. Suppose we have two variables `x` and `y`, storing the values `4` and `5` respectively and we run `print x + y`. If `x` and `y` are both strings, you would get `45`; if `x` and `y` are integers, you would get `9`; if `4` is an integer and `5` is a string, you would get an error.

In the last instance above, you see that assigning a type to each variable helps to keep the program meaningful, as the operation `+` is not defined over an integer and a string in Python.

Java and Javascript, however, would happily convert `4` into a string for you, and return `45`.

Dynamic vs. Static Type

Python and Javascript are examples of *dynamically typed* programming languages. The same variable can hold values of different types, and checking if the right type is used is done during the execution of the program. Note that, the type is associated with the *values*, and the type of the variable changes depending on the value it holds. For example, we can do the following:

Javascript

```javascript
let i = 4;   // i is an integer
i = "5"; // ok, i is now a string
```

Python

```python
i = 4   // i is an integer
i = "5" // ok, i is now a string
```

Java, on the other hand, is a *statically typed* language. We need to *declare* every variable we use in the program and specify its type. A variable can only hold values of the same type as the type of the variable, so we can't assign, for instance, a string to a variable of type `int`. Once a variable is assigned a type, its type cannot be changed.

```java
int i;   // declare a variable
i = 4;   // ok
i = "5"; // error, cannot assign a string to an `int`
```

The type that a variable is assigned with when we declare the variable is also known as the *compile-time type*. During the compilation, this is the only type that the compiler is aware of. The compiler will check if the compile-time type matches when it parses the variables, expressions, values, and function calls, and throw an error if there is a type mismatch. This type-checking step helps to catch errors in the code early.

Strong Typing vs. Weak Typing

A *type system* of a programming language is a set of rules that govern how the types can interact with each other.

A programming language can be strongly typed or weakly typed. There are no formal definitions of "strong" vs. "weak" typing of a programming language, and there is a spectrum of "strength" between the typing discipline of a language.

Generally, a *strongly typed* programming language enforces strict rules in its type system, to ensure *type safety*, i.e., to catch type errors during compile time rather than leaving it to runtime.

On the other hand, a *weakly typed* (or *loosely typed*) programming language is more permissive in terms of typing checking. C is an example of a static, weakly typed language. In C, the following is possible:

```
int i;   // declare a variable
i = 4;   // ok
i = (int)"5"; // you want to treat a string as an int? ok, as you wish!
```

The last line forces the C compiler to treat the string (to be more precise, the address of the string) as an integer, through typecasting.

In contrast, if we try the following in Java:

```
int i;   // declare a variable
i = 4;   // ok
i = (int)"5"; // error
```

we will get the following compile-time error message:

```
|  incompatible types: java.lang.String cannot be converted to int
```

because the compiler enforces a stricter rule and allows typecasting only if it makes sense.

Type Checking with A Compiler

In addition to checking for syntax errors, the compiler can check for type compilability according to the compile-time type, to catch possible errors as early as possible. Such type checking is made possible with static typing. Consider the following Python program:

```
i = 0
while (i < 10):
    # do something that takes time
```

```
4       i = i + 1
5   print("i is " + i)
```

The type mismatch error on Line 5 is only caught when Line 5 is executed. Since the type of the variable `i` can change during run time, Python (and generally, dynamically typed languages) can not tell if Line 5 will lead to an error until it is evaluated at run-time.

In contrast, statically typed language like Java can detect type mismatch during compile time since the compile-time type of a variable is fixed.

Primitive Types in Java

We now switch our focus to Java, particularly to the types supported. There are two categories of types in Java, the *primitive types* and the *reference types*. We will first look at primitive types in this unit.

Primitive types are types that holds numeric values (integers, floating-point numbers) as well as boolean values (`true` and `false`).

For storing integral values, Java provides four types, `byte` , `short` , `int` , and `long` , for storing 8-bit, 16-bit, 32-bit, 64-bit signed integers respectively. The type `char` stores 16-bit unsigned integers representing UTF-16 Unicode characters.

For storing floating-point values, Java provides two types, `float` and `double` , for 32-bit and 64-bit floating-point numbers.

Unlike reference types, which we will see later, primitive type variables never share their value with each other, i.e., if we have:

```
1   int i = 1000;
2   int j = i;
3   i = i + 1;
```

`i` and `j` each store a copy of the value `1000` after Line 2. Changing `i` on Line 3 does not change the content of `j` .

Subtypes

An important concept that we will visit repeatedly in CS2030/S is the concept of subtypes.

Let S and T be two types. We say that T is a *subtype* of S if *a piece of code written for variables of type S can also safely be used on variables of type T*.

We use the notation $T <: S$ or $S :> T$ to denote that T is subtype of S.

The subtype relationship is transitive, i.e., if $S <: T$ and $T <: U$, then $S <: U$. It is also reflexive, for any type S, $S <: S$.

We also use the term *supertype* to denote the reversed relationship: if T is a subtype of S, then S is a supertype of T.

Subtyping Between Java Primitive Types

Consider the range of values that the primitive types can take, Java defines the following subtyping relationship:

`byte` <: `short` <: `int` <: `long` <: `float` <: `double`

`char` <: `int`

Valid subtype relationship is part of what the Java compiler checks for when it compiles. Consider the following example:

```
1  double d = 5.0;
2  int i = 5;
3  d = i;
4  i = d; // error
```

Line 4 above would lead to an error:

```
|  incompatible types: possible lossy conversion from double to int
```

but Line 3 is OK.

This example shows how subtyping applies to type checking. *Java allows a variable of type T to hold a value from a variable of type S only if $S <: T$*. This step is called *widening type conversion*. Such conversion can happen during assignment or parameter passing.

Additional Readings

- Java Tutorial: Primitive Data Types and other Language Basics