Types, Values and Declarations

Data items have a value and a type and are stored in variables.

Associated with a variable are:

- a name (its identifier)
- where it is stored (its reference or address)
- the value stored

  A value can be a simple value (e.g. an integer), an address (i.e. a reference) or a structured value (e.g. an object).
- its type

  The type determines the range of allowed values and the operations allowed on these values.

Chapter 3

Binding time

An important factor which controls the power, flexibility and efficiency of a language is ***when*** different program features are associated (***bound***) to each other.

Early binding can give ***efficiency***.

Late binding can give ***flexibility***.

Possible binding times are:

- Compile time
- Load time
- Run time
  - block (e.g. procedure, function or method) entry
  - statement execution

Name-declaration binding

The same identifier, e.g. `x` , can be declared several times in a program.

An important question is:

Given a use of `x`, to which declaration of `x` is it associated?

Definition: ***name-declaration*** binding

the association between the name of an identifier and the declaration of that identifier

Name-declaration binding

When does name-declaration binding occur?

Compile time:

Pascal, Java and C++: the name-declaration binding can be determined by examining the program text alone.

Definition: ***scope***

the piece of program text in which an identifier is visible

Definition: ***scope rules***

Another name for the name-declaration binding rules.

Pascal, Java and C++ have ***static scope*** because the scope can be determined statically (ie at compile time)

Name-declaration binding: Pascal

```
program Sample(input, output);
    var x, y : Real;
    procedure op1;
      var y, z : Integer;
    begin
       ...
       x := 27.4;
       ...
       y := 34;
       ...
    end {op1};
begin
     ...
     y := 3.7;
     ...
end {Sample}.
```


Name-declaration binding: Java

```
class Sample {
    private double x, y;
    public void op1() {
      int y, z;
      ...
      x = 27.4;
      ...
      y = 34;
      ...
    } //op1
    public void op2() {
      ...
      y = 3.7;
      ...
    } //op2
 } // Sample
```

Static scope rules

In Pascal, program **`Ex`** and procedure **`op1`** are ***blocks***.

In Java, we can regard class **`Ex`** and methods **`op1`** and **`op2`** as blocks.

In Pascal, block **`op1`** is ***nested*** within block **`Ex`**.

In Java, blocks **`op1`** and **`op2`** are ***nested*** within block **`Ex`**.

The piece of program text in which an identifier is ***visible*** and may therefore be used is known as its ***scope***.

Static scope rules

The static scope rules are:

- The use of an identifier is always bound to its most local declaration.
- An identifier is not visible outside the block in which it is declared.
- Identifiers declared in enclosing blocks are visible in inner blocks, unless they have been re-declared.

Static scope rules

Questions:

What is the binding of the use of `x` in `op1`?

What is the binding of the use of `y` in `op1`?

Can the variable `y` defined in block `Sample` be accessed in block `op1`?

Which `y` is assigned to the value `3.7` ?

Static typing

Definition: ***type checking***

Ensuring that the use of a variable is compatible with its declared type.

If ***name-type binding*** occurs at compile time then type-checking can be done just by examining the text. This is called ***static typing***.

Related: ***Strong typing*** all expressions have a well defined type. *Often* at compile time.

Ada and Algol 68 are strongly typed. In Pascal, ***variant records*** give a loop-hole in the type-checking.

Java is said to be strongly typed.

Static typing

Static typing leads to programs which are:

reliable: because type errors are detected at compile time;

efficient: because type checks do not have to be made at run time;

understandable: because the connection (binding) between the use of an identifier and its declaration can be determined from the program text.

Dynamically typed languages

If ***name-type binding*** occurs at run time then we have ***dynamic typing***.

The type of a variable depends on its current value, i.e. a variable has a ***tag*** giving the type of the variable it is currently holding. E.g. Python, APL, SNOBOL4, Smalltalk, LISP and many scripting languages.

Often, the type of a variable is not given in its declaration.

Benefits: can be very flexible but does slow down running speed. Such languages are usually run under the control of an interpreter which performs the required bindings.

Dynamically typed languages

Errors:

Dynamic typing can lead to run-time errors

e.g. divide a Boolean by 2.

To ensure that does not happen, we need to ***check types at run time***. This is not necessary in a statically typed language where all the type checking is done at compile time.

See example p.55 of Comparative Programming Languages.

Dynamic typing <> OO Dynamic Binding

Java and C++ are statically typed.

OO development allows an object of a subclass to be used where an object of a superclass is expected. No problem at run time.

The allowed superclass is determined statically as is the ***set*** of allowable subclasses.

Declaration-reference binding

Definition: ***declaration-reference binding***

the association between the declaration of an identifier and the allocation of storage.

Note that this is concerned with the ***execution*** of a program, i.e. it must be a run time feature. A big question is where storage is allocated: it can be on the stack or on the heap.

Declaration-reference binding can occur at load time, block (i.e. procedure or method) entry or during statement execution.

Definition: ***lifetime***

The duration of an individual declaration-reference binding is called the lifetime of the variable.

Declaration-reference binding

Load time

In Pascal, variables declared in the main program.

In Java, variables declared in the `static void main` method.

Static local variables in Algol 60, C and PL/I.

Space is allocated on the stack

and deallocated on program exit.

Declaration-reference binding

Block entry

In Pascal, Java etc., local variables in procedures or methods are allocated space on block entry, i.e. when the procedure or method is called.

Note that with recursion, a declaration may be bound to several references, i.e. to several different storage locations at the same time.

Space is allocated on the stack

and deallocated on block exit.

Declaration-reference binding

Statement execution

In Java, objects (and therefore their attributes) are allocated space during statement execution using the **`new`** operator as in:

```
theBalloon = new Balloon();
```

Pascal, C++ and Ada make similar use of **`new`** to allocate space during statement execution.

Space is allocated on the ***heap***

and deallocated by the programmer (Pascal, C++, Ada) or by the garbage collector (Java).

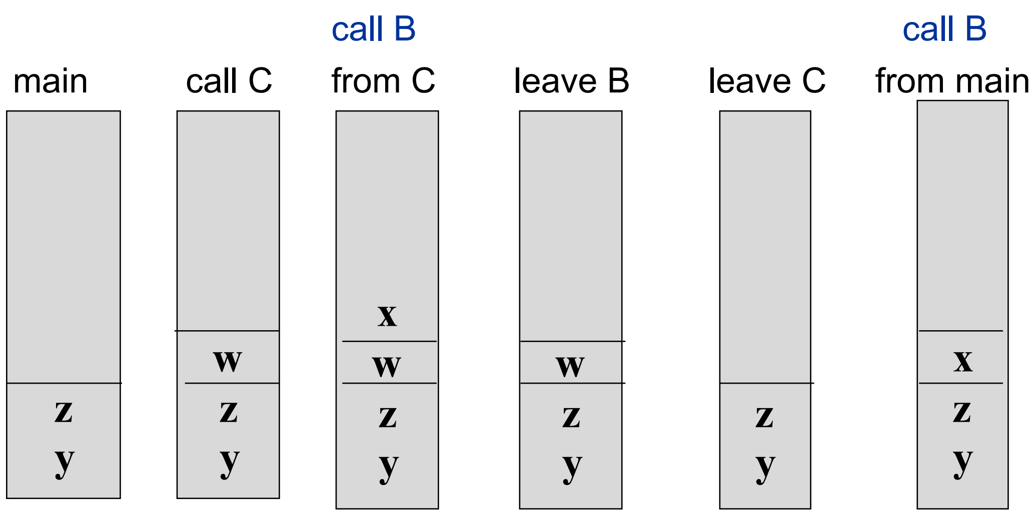
Stack storage allocation

Allocation of space to local variables on block entry is implemented using a stack.

The following example uses Pascal, but Java acts in the same way.

```
program main;
  var y, z : integer;
  procedure B;
    var x : integer;
  begin ... end {B};
  procedure C;
    var w : integer;
  begin ... B; ... end {C};
begin ... C; ... B; ... end.
```


Stack storage allocation



Chapter 6.4

Declaration-reference binding

What if the same variable is used over and over again?

Does it get the same location every time?

Can values be remembered from one invocation to the next?

Reference-value binding

Definition: ***reference-value binding***

the association between a store location and a value.

Consider:

```
y = x + 1
```

name-declaration	**name-declaration**
declaration-reference	**declaration-reference**
	reference-value

Finding a value, given a reference, is known as **dereferencing**.

Reference-value binding

If we are to change the value of a variable then we need to know its reference, i.e. where it is stored.

References not needed with constants.

```
final double pi = 3.14159265;
x  =  pi;
```

name-declaration

declaration-value

Constants

Possible in Ada, C++ and Java for the constant to be assigned the value of an expression.

Name-value binding is then at block entry, i.e the constant is given a constant value when it comes into existence.

On block exit, the constant ceases to exist. The next time it is created, it can have a different (constant) value.

Constants

What about constants that you want to use throughout a program?

We can declare a class of useful constants:

```
public class OurConsts {
  public final static int LENGTH = 12;
  public final static boolean OPEN = true;
}
```

Variables that are **`static`** are not associated with an object, but with the class. When they are **`final`**, they are constants.

We can use such a constant as follows:

```
  area = OurConsts.LENGTH*OurConsts.LENGTH;
```

Note class name used, not object name.

Types

The defining characteristics of a type are:

- a set of allowed ***values***;
- a set of ***operations*** used to manipulate the values.

There are many kinds of types found in programming languages:

structured types

- examples: arrays, records, files, objects;

Chapter 7

Types

scalar types

- fractional numbers (**`float, double`**)
- discrete or ordinal types, e.g. integers, booleans, characters.

With an ordinal type, each value (except the smallest and largest) has a unique predecessor and successor.

pointer (i.e. reference) types

- the value is an address.

Types

Early languages only had ***built-in*** or ***primitive*** types.

Example: Algol 60

```
integer a, b;
integer array c[1:10];
```

The user could not define new types.

Types

Pascal and later languages allowed new types to be defined, e.g.:

```
type lots = array [1..10] of integer;
var c : lots;
```

Type definitions define the structure of the type and give it a name.

This structural information does not then have to be repeated each time a variable is declared.

This is especially useful in parameter passing to ensure that actual and formal parameters are of the same type.

Classes as types

Adding more value: associating types with the set of allowable values ***and*** with the set of possible operations.

This gives us the notion of ADTs.

Object-oriented languages go further.

A module ***contains*** a type definition while a class ***is*** a type definition.

Numeric data types

Language rules specify how integer and fractional literal constants are written.

Operators are usually part of the language too. Overloading means fewer names (symbols):

```
2 + 3
2.5 + 3.6
```

More examples:

Algol 60 and Pascal use both / and div

Ada, C++, Fortran and Java overload / and the value of 7/2 is 3 while 7.0/2.0 is 3.5.

Early languages had an exponentiation operator: ^(Algol 60) or ** (Fortran)

Why might this be problematic?

Integers and floats

C and C++ have four integer types: **`char`**, **`short`**, **`int`** and **`long`**, but the number of bytes used to represent each of them is machine dependent.

In Java, the four integer types: **`byte`**, **`short`**, **`int`** and **`long`**, are held in 1, 2, 4 and 8 bytes respectively.

In C, C++ and Java, fractional numbers can be declared as **`float`** or **`double`**.

In Java, a **`float`** is held in 4 bytes and a **`double`** in 8.

Integers and floats

In C, C++ and Java, the constant **`3.7`** is a **`double`**, we write a **`float`** as **`3.7f`**.

Java supports the idea of ***widening***.

We can write:

```
double x = 3;
double y = 3.7f;
```

But

```
float x = 3.7; // error!!
```

It is allowed in C and C++.

Characters

Most modern languages have a **`character`** type.

The character set used is often not defined (e.g. Pascal, C++) and is therefore implementation dependent.

In Pascal, C and Java, we have:

```
'0' < '1' < ... < '9'
'a' < 'b' < ... < 'z'
'A' < 'B' < ... < 'Z'
```

Ada uses the ASCII character set. Ada 95 uses extended ASCII. Java uses Unicode.