*3.1- The notion of Binding time*

*Binding time* is the time at which a binding is created or, more generally, the time at which any implementation decision is made. A *binding* is an association between two things, such as a name and the thing it names.

There are different times decision might be bound by;

1. *Language design time*
2. *Language implementation time*
3. *Program writing time*
4. *Compile time*
5. *Link time*
6. *Load time*
7. *Run time*

3.2 – Object lifetime and storage Management

Steps to distinguish between name and objects;

1. Creating and destruction object
2. Creation and destruction bindings
3. Deactivation and reactivation of binding that may be temporarily unusable
4. References to variables, subroutines, types, and so on, all of which use bindings

The period of time between the creation and the destruction of a name to object binding is call the binding’s ‘lifetime’.

Object lifetimes generally correspond to one of three principal *storage allocation* mechanisms, used to manage the object’s space:

1. *Static* objects are given an absolute address that is retained throughout the program’s execution.
2. *Stack* objects are allocated and deallocated in last-in, first-out order, usually in

conjunction with subroutine calls and returns.

1. *Heap* objects may be allocated and deallocated at arbitrary times. They require a more general (and expensive) storage management algorithm.

3.3 - Scope Rules

The textual region of the program in which a binding is active is its *scope*. In most modern languages, the scope of a binding is determined statically, that is, at compile time.

Important Scope Parts;

1. Static Scoping

In a language with static (lexical) scoping, the bindings between names and objects can be determined at compile time by examining the text of the program, without consideration of the flow of control at run time.

1. Nested Subroutines

The ability to nest subroutines inside each other, introduced in Algol 60, is a feature of many subsequent languages

1. Declaration Order

In an apparent attempt to simplify the implementation of the compiler, Pascal modified the requirement to say that names must be declared before they are used. There are special mechanisms to accommodate recursive types and sub- routines, but in general, a *forward reference* (an attempt to use a name before its declaration) is a static semantic error.

1. Modules

An important challenge in the construction of any large body of software is to divide the effort among programmers in such a way that work can proceed on multiple fronts simultaneously.

1. Dynamic Scoping

In a language with dynamic scoping, the bindings between names and objects depend on the flow of control at run time, and in particular on the order in which subroutines are called.

3.4 – Implementing Scope

To keep track of the names in a statically scoped program, a compiler relies on a data abstraction called a *symbol table*. In essence, the symbol table is a dictionary: it maps names to the information the compiler knows about them.

3.5 – The Meaning of Names within a Scope

So far in our discussion of naming and scopes we have assumed that there is a one-to-one mapping between names and visible objects at any given point in a program. This need not be the case. Two or more names that refer to the same object at the same point in the program are said to be *aliases*. A name that can refer to more than one object at a given point in the program is said to be *overloaded*. Overloading is in turn related to the more general subject of *polymorphism*, which allows a subroutine or other program fragment to behave in different ways depending on the types of its arguments.

3.6 – The binding of referencing Environments

Static scope rules specify that the refer- encing environment depends on the lexical nesting of program blocks in which names are declared. Dynamic scope rules specify that the referencing environ- ment depends on the order in which declarations are encountered at run time.