```
i = x;
```

but because of a keying error, it has the assignment statement

```
i = y;
```

In JavaScript (or any other language that uses dynamic type binding), no error is detected in this statement by the interpreter—the type of the variable named `i` is simply changed to an array. But later uses of `i` will expect it to be a scalar, and correct results will be impossible. In a language with static type binding, such as Java, the compiler would detect the error in the assignment `i = y`, and the program would not get to execution.

Note that this disadvantage is also present to some extent in some languages that use static type binding, such as Fortran, C, and C++, which in many cases automatically convert the type of the RHS of an assignment to the type of the LHS.

Perhaps the greatest disadvantage of dynamic type binding is cost. The cost of implementing dynamic attribute binding is considerable, particularly in execution time. Type checking must be done at run time. Furthermore, every variable must have a run-time descriptor associated with it to maintain the current type. The storage used for the value of a variable must be of varying size, because different type values require different amounts of storage.

Finally, languages that have dynamic type binding for variables are usually implemented using pure interpreters rather than compilers. Computers do not have instructions whose operand types are not known at compile time. Therefore, a compiler cannot build machine instructions for the expression A + B if the types of A and B are not known at compile time. Pure interpretation typically takes at least 10 times as long as it does to execute equivalent machine code. Of course, if a language is implemented with a pure interpreter, the time to do dynamic type binding is hidden by the overall time of interpretation, so it seems less costly in that environment. On the other hand, languages with static type bindings are seldom implemented by pure interpretation, because programs in these languages can be easily translated to very efficient machine code versions.

5.4.3 Storage Bindings and Lifetime

The fundamental character of an imperative programming language is in large part determined by the design of the storage bindings for its variables. It is therefore important to have a clear understanding of these bindings.

The memory cell to which a variable is bound somehow must be taken from a pool of available memory. This process is called **allocation**. **Deallocation** is the process of placing a memory cell that has been unbound from a variable back into the pool of available memory.

The **lifetime** of a variable is the time during which the variable is bound to a specific memory location. So, the lifetime of a variable begins when it is bound to a specific cell and ends when it is unbound from that cell. To investigate storage bindings of variables, it is convenient to separate scalar

(unstructured) variables into four categories, according to their lifetimes. These categories are named static, stack-dynamic, explicit heap-dynamic, and implicit heap-dynamic. In the following sections, we discuss the definitions of these four categories, along with their purposes, advantages, and disadvantages.

5.4.3.1 Static Variables

Static variables are those that are bound to memory cells before program execution begins and remain bound to those same memory cells until program execution terminates. Statically bound variables have several valuable applications in programming. Globally accessible variables are often used throughout the execution of a program, thus making it necessary to have them bound to the same storage during that execution. Sometimes it is convenient to have subprograms that are history sensitive. Such a subprogram must have local static variables.

One advantage of static variables is efficiency. All addressing of static variables can be direct;[5] other kinds of variables often require indirect addressing, which is slower. Also, no run-time overhead is incurred for allocation and deallocation of static variables, although this time is often negligible.

One disadvantage of static binding to storage is reduced flexibility; in particular, a language that has only static variables cannot support recursive subprograms. Another disadvantage is that storage cannot be shared among variables. For example, suppose a program has two subprograms, both of which require large arrays. Furthermore, suppose that the two subprograms are never active at the same time. If the arrays are static, they cannot share the same storage for their arrays.

C and C++ allow programmers to include the `static` specifier on a variable definition in a function, making the variables it defines static. Note that when the `static` modifier appears in the declaration of a variable in a class definition in C++, Java, and C#, it also implies that the variable is a class variable, rather than an instance variable. Class variables are created statically some time before the class is first instantiated.

5.4.3.2 Stack-Dynamic Variables

Stack-dynamic variables are those whose storage bindings are created when their declaration statements are elaborated, but whose types are statically bound. **Elaboration** of such a declaration refers to the storage allocation and binding process indicated by the declaration, which takes place when execution reaches the code to which the declaration is attached. Therefore, elaboration occurs during run time. For example, the variable declarations that appear at the beginning of a Java method are elaborated when the method is called and the variables defined by those declarations are deallocated when the method completes its execution.

---

5. In some implementations, static variables are addressed through a base register, making accesses to them as costly as for stack-allocated variables.

As their name indicates, stack-dynamic variables are allocated from the run-time stack.

Some languages—for example, C++ and Java—allow variable declarations to occur anywhere a statement can appear. In some implementations of these languages, all of the stack-dynamic variables declared in a function or method (not including those declared in nested blocks) may be bound to storage at the beginning of execution of the function or method, even though the declarations of some of these variables do not appear at the beginning. In such cases, the variable becomes visible at the declaration, but the storage binding (and initialization, if it is specified in the declaration) occurs when the function or method begins execution. The fact that storage binding of a variable takes place before it becomes visible does not affect the semantics of the language.

The advantages of stack-dynamic variables are as follows: To be useful, at least in most cases, recursive subprograms require some form of dynamic local storage so that each active copy of the recursive subprogram has its own version of the local variables. These needs are conveniently met by stack-dynamic variables. Even in the absence of recursion, having stack-dynamic local storage for subprograms is not without merit, because all subprograms share the same memory space for their locals.

The disadvantages, relative to static variables, of stack-dynamic variables are the run-time overhead of allocation and deallocation, possibly slower accesses because indirect addressing is required, and the fact that subprograms cannot be history sensitive. The time required to allocate and deallocate stack-dynamic variables is not significant, because all of the stack-dynamic variables that are declared at the beginning of a subprogram are allocated and deallocated together, rather than by separate operations.

Fortran 95+ allows implementors to use stack-dynamic variables for locals, but includes the following statement:

`Save` list

This declaration allows the programmer to specify that some or all of the variables (those in the list) in the subprogram in which `Save` is placed will be static.

In Java, C++, and C#, variables defined in methods are by default stack dynamic. In Ada, all non-heap variables defined in subprograms are stack dynamic.

All attributes other than storage are statically bound to stack-dynamic scalar variables. That is not the case for some structured types, as is discussed in Chapter 6. Implementation of allocation/deallocation processes for stack-dynamic variables is discussed in Chapter 10.

5.4.3.3 Explicit Heap-Dynamic Variables

Explicit heap-dynamic variables are nameless (abstract) memory cells that are allocated and deallocated by explicit run-time instructions written by the programmer. These variables, which are allocated from and deallocated to the heap, can only be referenced through pointer or reference variables. The heap is a collection of storage cells whose organization is highly disorganized because of the

unpredictability of its use. The pointer or reference variable that is used to access an explicit heap-dynamic variable is created as any other scalar variable. An explicit heap-dynamic variable is created by either an operator (for example, in C++) or a call to a system subprogram provided for that purpose (for example, in C).

In C++, the allocation operator, named **new**, uses a type name as its operand. When executed, an explicit heap-dynamic variable of the operand type is created and its address is returned. Because an explicit heap-dynamic variable is bound to a type at compile time, that binding is static. However, such variables are bound to storage at the time they are created, which is during run time.

In addition to a subprogram or operator for creating explicit heap-dynamic variables, some languages include a subprogram or operator for explicitly destroying them.

As an example of explicit heap-dynamic variables, consider the following C++ code segment:

```
int *intnode;       // Create a pointer
intnode = new int; // Create the heap-dynamic variable
...
delete intnode;     // Deallocate the heap-dynamic variable
                                // to which intnode points
```

In this example, an explicit heap-dynamic variable of **int** type is created by the **new** operator. This variable can then be referenced through the pointer, `intnode`. Later, the variable is deallocated by the **delete** operator. C++ requires the explicit deallocation operator **delete**, because it does not use implicit storage reclamation, such as garbage collection.

In Java, all data except the primitive scalars are objects. Java objects are explicitly heap dynamic and are accessed through reference variables. Java has no way of explicitly destroying a heap-dynamic variable; rather, implicit garbage collection is used. Garbage collection is discussed in Chapter 6.

C# has both explicit heap-dynamic and stack-dynamic objects, all of which are implicitly deallocated. In addition, C# supports C++-style pointers. Such pointers are used to reference heap, stack, and even static variables and objects. These pointers have the same dangers as those of C++, and the objects they reference on the heap are not implicitly deallocated. Pointers are included in C# to allow C# components to interoperate with C and C++ components. To discourage their use, and also to make clear to any program reader that the code uses pointers, the header of any method that defines a pointer must include the reserved word **unsafe**.

Explicit heap-dynamic variables are often used to construct dynamic structures, such as linked lists and trees, that need to grow and/or shrink during execution. Such structures can be built conveniently using pointers or references and explicit heap-dynamic variables.

The disadvantages of explicit heap-dynamic variables are the difficulty of using pointer and reference variables correctly, the cost of references to the

variables, and the complexity of the required storage management implementation. This is essentially the problem of heap management, which is costly and complicated. Implementation methods for explicit heap-dynamic variables are discussed at length in Chapter 6.

5.4.3.4 Implicit Heap-Dynamic Variables

Implicit heap-dynamic variables are bound to heap storage only when they are assigned values. In fact, all their attributes are bound every time they are assigned. For example, consider the following JavaScript assignment statement:

```
highs = [74, 84, 86, 90, 71];
```

Regardless of whether the variable named `highs` was previously used in the program or what it was used for, it is now an array of five numeric values.

The advantage of such variables is that they have the highest degree of flexibility, allowing highly generic code to be written. One disadvantage of implicit heap-dynamic variables is the run-time overhead of maintaining all the dynamic attributes, which could include array subscript types and ranges, among others. Another disadvantage is the loss of some error detection by the compiler, as discussed in Section 5.4.2.2. Examples of implicit heap-dynamic variables in JavaScript appear in Section 5.4.2.2.

5.5 Scope

One of the important factors in understanding variables is scope. The **scope** of a variable is the range of statements in which the variable is visible. A variable is **visible** in a statement if it can be referenced in that statement.

The scope rules of a language determine how a particular occurrence of a name is associated with a variable, or in the case of a functional language, how a name is associated with an expression. In particular, scope rules determine how references to variables declared outside the currently executing subprogram or block are associated with their declarations and thus their attributes (blocks are discussed in Section 5.5.2). A clear understanding of these rules for a language is therefore essential to the ability to write or read programs in that language.

A variable is **local** in a program unit or block if it is declared there. The nonlocal variables of a program unit or block are those that are visible within the program unit or block but are not declared there. Global variables are a special category of nonlocal variables. They are discussed in Section 5.5.4.

Scoping issues of classes, packages, and namespaces are discussed in Chapter 11.