Value Types and Reference Types

The *type* of a data item defines how much memory is required to store it and the data members that comprise it. The type also determines where an object is stored in memory—the stack or the heap.

Types are divided into two categories: value types and reference types. Objects of these types are stored differently in memory.

- Value types require only a single segment of memory, which stores the actual data.
- Reference types require two segments of memory:
  - The first contains the actual data—and is always located in the heap.
  - The second is a reference that points to where the data is stored in the heap.

Figure 3-9 shows how a single data item of each type is stored. For value types, data is stored on the stack. For reference types, the actual data is stored in the heap, and the reference is stored on the stack.

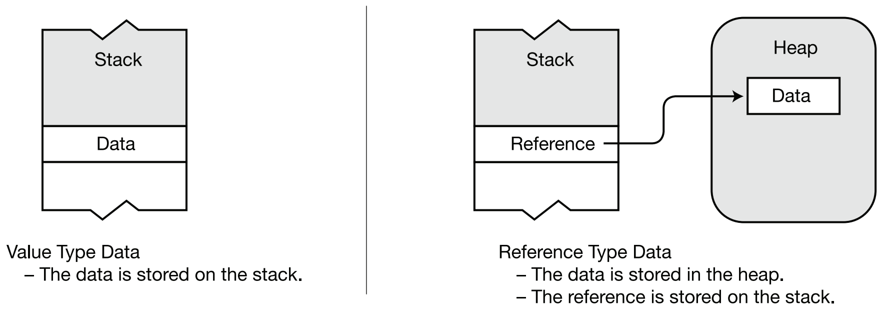

Stack
Data
Value Type Data
– The data is stored on the stack.
Stack
Reference
Heap
Data
Reference Type Data
– The data is stored in the heap.
– The reference is stored on the stack.

Figure 3-9. *Storing data that is not part of another type*

Storing Members of a Reference Type Object

Although Figure 3-9 shows how data is stored when it isn't a member of another object, when it's a member of another object, data might be stored a little differently.

- The data portion of a reference type object is *always* stored in the heap, as shown in Figure 3-9.
- A value type object, or the reference part of a reference type, can be stored in either the stack or the heap, depending on the circumstances.

Suppose, for example, that you have an instance of a reference type called `MyType` that has two members—a value type member and a reference type member. How is it stored? Is the value type member stored on the stack and the reference type split between the stack and the heap, as shown in Figure 3-9? The answer is no.

Remember that for a reference type, the data of an instance is *always* stored in the heap. Since both members are part of the object's data, they're both stored in the heap, regardless of whether they are value or reference types. Figure 3-10 illustrates the case of type `MyType`.

- Even though member `A` is a value type, it's part of the data of the instance of `MyType` and is therefore stored with the object's data in the heap.
- Member `B` is a reference type, and therefore its data portion will always be stored in the heap, as shown by the small box marked "Data." What's different is that its reference is also stored in the heap, inside the data portion of the enclosing `MyType` object.

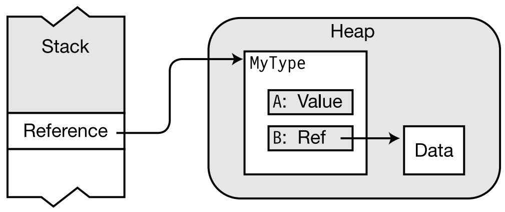


Figure 3-10. *Storage of data as part of a reference type*

■ **Note** For any object of a reference type, all its data members are stored in the heap, regardless of whether they are of value type or reference type.

Categorizing the C# Types

Table 3-3 shows all the types available in C# and what kinds of types they are—value types or reference types. Each reference type is covered later in the text.

Table 3-3. *Value Types and Reference Types in C#*

	Value Types			**Reference Types**

Predefined types	`sbyte byte`
	`short`
	`int`
	`long`
	`bool`
		
		`ushort`
		`uint`
		`ulong`
			`float`
			`double`
			`char`
			`decimal`
				`object`
				`string`
				`dynamic`

User-defined types	`struct`
	`enum`
				`class`
				`interface`
				`delegate`
				`array`

Variables

A general-purpose programming language must allow a program to store and retrieve data.

- A *variable* is a name that represents data stored in memory during program execution.
- C# provides four kinds of variables, each of which will be discussed in detail. These are listed in Table 3-4.

Table 3-4. *The Four Kinds of Variables*

Name	Description
Local variable	Holds temporary data within the scope of a method. Not a member of a type.
Field	Holds data associated with a type or an instance of a type. Member of a type.
Parameter	A temporary variable used to pass data from one method to another method. Not a member of a type.
Array element	One member of a sequenced collection of (usually) homogeneous data items. Can be either local or a member of a type.

Variable Declarations

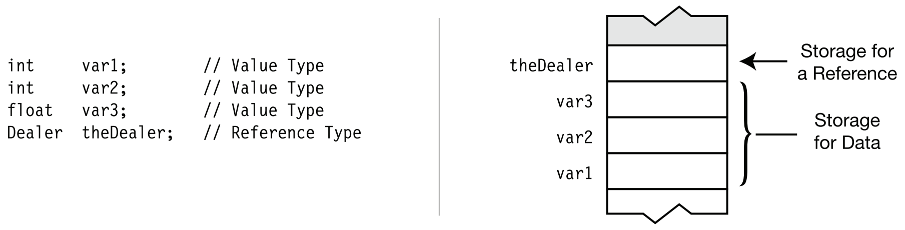
A variable must be declared before it can be used. The variable declaration defines the variable and accomplishes two things:

- It gives the variable a name and associates a type with it.
- It allows the compiler to allocate memory for it.

A simple variable declaration requires at least a type and a name. The following declaration defines a variable named `var2`, of type `int`:

```
Type
 ↓
int var2;
     ↑
    Name
```

For example, Figure 3-11 represents the declaration of four variables and their places on the stack.



```
int     var1;       // Value Type
int     var2;       // Value Type
float   var3;       // Value Type
Dealer  theDealer;  // Reference Type
```

Figure 3-11. *Value type and reference type variable declarations*

Variable Initializers

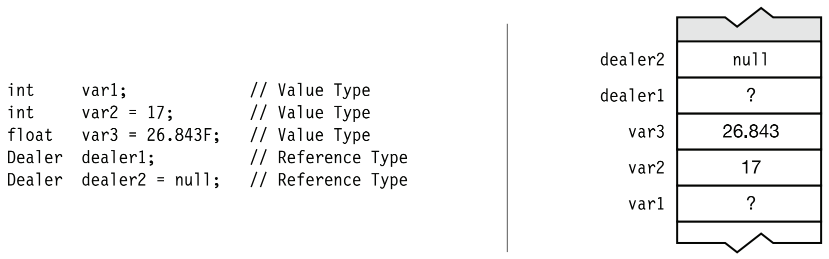
Besides declaring a variable's name and type, you can optionally use the declaration to initialize its memory to a specific value.

A *variable initializer* consists of an equal sign followed by the initializing value, as shown here:

```
          Initializer
              ↓
int var2 = 17;
```

Local variables without initializers have an undefined value and cannot be used until they have been assigned a value. Attempting to use an undefined local variable causes the compiler to produce an error message.

Figure 3-12 shows a number of local variable declarations on the left and the resulting stack configuration on the right. Some of the variables have initializers, and others do not.



```
int     var1;              // Value Type
int     var2 = 17;         // Value Type
float   var3 = 26.843F;    // Value Type
Dealer  dealer1;           // Reference Type
Dealer  dealer2 = null;    // Reference Type
```

Figure 3-12. *Variable initializers*

Automatic Initialization

Some kinds of variables are automatically set to default values if they are declared without an initializer, and others are not. Variables that are not automatically initialized to default values contain undefined values until the program assigns them a value. Table 3-5 shows which kinds of variables are automatically initialized and which are not. I'll cover each of the five kinds of variables later in the text.

Table 3-5. *Summary of Variable Storage*

Variable	Stored In	Auto-initialized	Use
Local variables	Stack or stack and heap	No	Used for local computation inside a function member
Class fields	Heap	Yes	Members of a `class`
Struct fields	Stack or heap	Yes	Members of a `struct`
Parameters Stack		No	Used for passing values into and out of a method
Array elements	Heap	Yes	Members of an array

Multiple-Variable Declarations

You can declare multiple variables in a single declaration statement.

- The variables in a multiple-variable declaration must all be of the same type.
- The variable names must be separated with commas. Initializers can be included with the variable names.

For example, the following code shows two valid declaration statements with multiple variables. Notice that the initialized variables can be mixed with uninitialized variables as long as they're separated by commas. The last declaration statement shown is invalid because it attempts to declare variables of different types in a single statement.

```
// Variable declarations--some with initializers, some without
int    var3 = 7, var4, var5 = 3;
double var6, var7 = 6.52;

Type      Different type
 ↓            ↓
int var8, float var9;       // Error! Can't mix types (int and float)
```

Using the Value of a Variable

A variable name represents the value stored by the variable. You can use the value by using the variable name.

For example, in the following statement, the variable name `var2` represents the *value* stored by the variable. That value is retrieved from memory when the statement is executed.

```
Console.WriteLine("{0}", var2);
```

Static Typing and the dynamic Keyword

One thing you'll have noticed is that every variable declaration includes the *type* of the variable. This allows the compiler to determine the amount of memory it will require at run time and which parts should be stored on the stack and which in the heap. The type of the variable is determined at compile time and cannot be changed at run time. This is called *static typing*.

Not all languages, though, are statically typed. Many, including scripting languages such as IronPython and IronRuby, are *dynamically typed*. That is, the type of a variable might not be resolved until run time. Since these are also .NET languages, C# programs need to be able to use assemblies written in these languages. The problem, then, is that C# needs to be able to resolve at compile time a type from an assembly that doesn't resolve its types until run time.

To solve this problem, C# provides the `dynamic` keyword to represent a specific C# type that knows how to resolve itself at run time.

At compile time, the compiler doesn't do type checking on variables of type `dynamic`. Instead, it packages up any information about the variable's operations and includes that information with the variable. At run time, that information is checked to make sure it's consistent with the actual type to which the variable was resolved. If it doesn't, the run time throws an exception.

Nullable Types

There are situations, particularly when working with databases, where you want to indicate that a variable does not currently hold a valid value. For reference types, you can do this easily, by setting the variable to `null`. When you define a variable of a value type, however, its memory is allocated whether or not its contents have any valid meaning.

What you would like in this situation is to have a Boolean indicator associated with the variable so that, when the value is valid, the indicator is `true`, and when the value is not valid, the indicator is `false`.

Nullable types allow you to create a value type variable that can be marked as valid or invalid so that you can make sure a variable is valid before using it. Regular value types are called *non-nullable types*. I'll explain the details of nullable types in Chapter 25, when you have a better understanding of C#.