Semantic Differences between Reference Types and Value Types

Reference types in .NET include classes, delegates, interfaces, and arrays. `string` (`System.String`), which is one of the most ubiquitous types in .NET, is a reference type as well. Value types in .NET include structs and enums. The primitive types, such as `int`, `float`, `decimal`, are value types – but .NET developers are free to define additional value types using the `struct` keyword.

On the language level, reference types enjoy *reference semantics*, where the identity of the object is considered before its content, while value types enjoy *value semantics*, where objects don't have an identity, are not accessed through references, and are treated according to their content. This affects several areas of the .NET languages, as Table 3-1 demonstrates.

Table 3-1. *Semantic Differences Between Value Types and Reference Types*

Criteria	Reference Types	Value Types
Passing an object to a method	Only the reference is passed; changes propagate to all other references	The object's contents are copied into the parameter (unless using the `ref` or `out` keywords); changes do not affect any code outside of the method
Assigning a variable to another variable	Only the reference is copied; two variables now contain references to the same object	The contents are copied; the two variables contain identical copies of unrelated data
Comparing two objects using `operator==`	The references are compared; two references are equal if they reference the same object	The contents are compared; two objects are equal if their content is identical on a field-by-field level

These semantic differences are fundamental to the way we write code in any .NET language. However, they are only the tip of the iceberg in terms of how reference types and value types and their purposes differ. First, let's consider the memory locations in which objects are stored, and how they are allocated and deallocated.