An Example

Consider a simple type called `Point2D` that represents a point in a small two-dimensional space. Each of the two coordinates can be represented by a `short`, for a total of four bytes for the entire object. Now suppose that you want to store in memory an array of ten million points. How much space would be required for them? The answer to this question depends greatly on whether `Point2D` is a reference type or a value type. If it is a reference type, an array of ten million points would actually store ten million references. On a 32-bit system, these ten million references consume almost 40 MB of memory. The objects themselves would consume at least the same amount. In fact, as we will see shortly, each `Point2D` instance would occupy at least 12 bytes of memory, bringing the total memory usage for an array of ten million points to a whopping 160MB! On the other hand, if `Point2D` is a value type, an array of ten million points would store ten million points – not a single extra byte wasted, for a total of 40MB, four times less than the reference type approach (see Figure 3-1). This difference in *memory density* is a critical reason to prefer value types in certain settings.

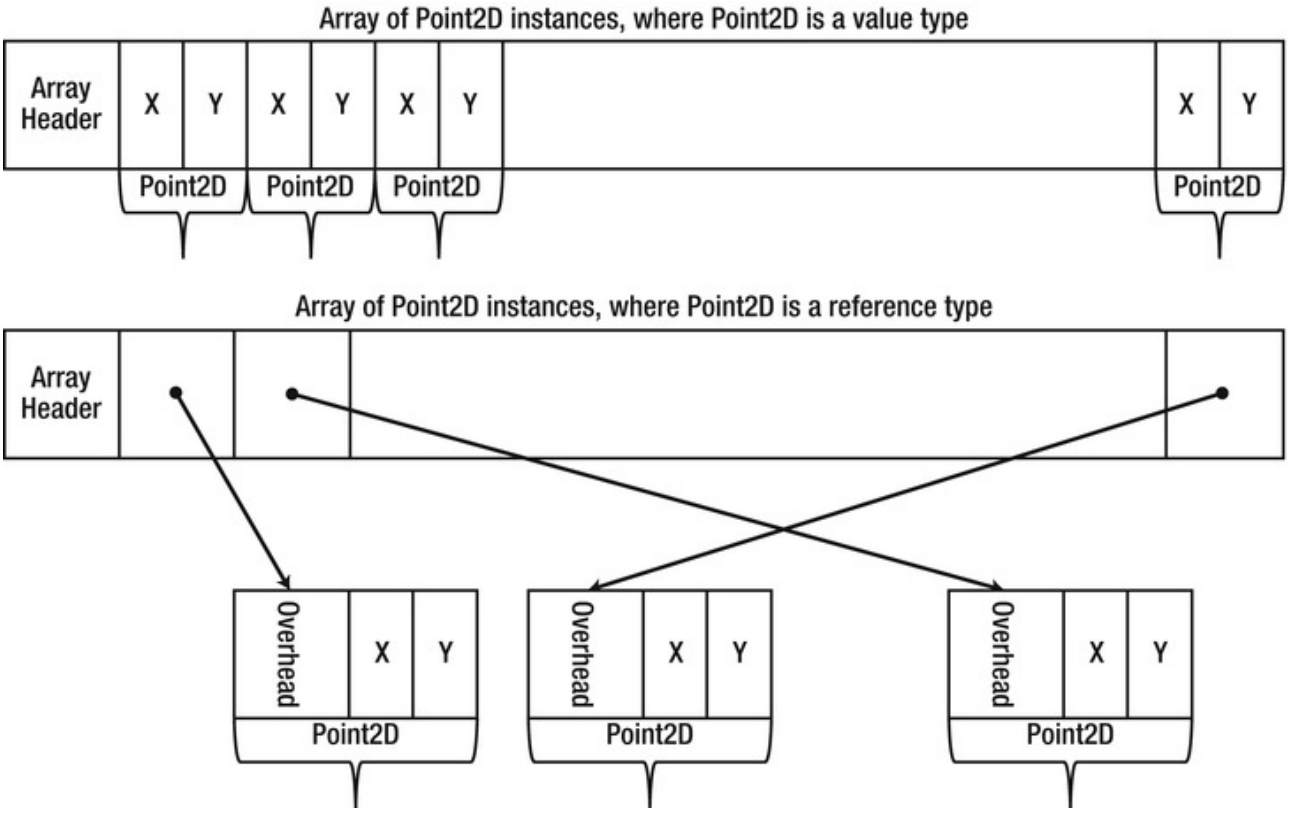

Figure 3-1. An array of Point2D instances in the case Point2D is a reference type versus a value type

Note There is another disadvantage to storing references to points instead of the actual point instances. If you want to traverse this huge array of points sequentially, it is much easier for the compiler and the hardware to access a contiguous array of Point2D instances than it is to access through references the heap objects, which are not guaranteed to be contiguous in memory. As we shall see in Chapter 5, CPU cache considerations can affect the application's execution time up to an order of magnitude.

It is inevitable to conclude that understanding the details of how the CLR lays out objects in memory and how reference types differ from value types is crucial for the performance of our applications. We begin by reviewing the fundamental differences between value types and reference types at the language level and then dive into the internal implementation details.