Static Objects

I've already mentioned static/global objects as a source of root references, but let's now look at that topic in a bit more detail, and with some more background.

Marking a class member as static makes it a class-specific, rather than instance-specific, object. With using non-static members, you would need to declare an instance of the necessary class before you could access its members. On the other hand, static members can be accessed directly by just using the class name.

```
class Person
{
   public int Age=0;
   public static MaxAge=120;
}
```

Listing 1.13: Example of a static member variable.

Listing 1.13 shows both an instance variable (`Age`) and a static variable (`MaxAge`) on a `Person` class. The static variable is being used as a piece of general data across the range of `Person` instances (people aren't usual older than 120), whereas the instance variable is specific to an instance of the class, i.e. an individual person.

To access each member, you would need to write the code in Listing 1.14.

```
Person thisPerson=new Person();
thisPerson.Age=121;

if (thisPerson.Age>Person.MaxAge)
{
   // Validation Failure
}
```

Listing 1.14: Accessing statics.

In Listing 1.14, an instance of a `Person` is created, and it's *only* via the instance variable that the `Age` member is accessible, whereas `MaxAge` is available as a kind of global member on the `Person` type itself.

In C#, statics are often used to define global variables.

Static methods and fields

When you mark a method, property, variable, or event as static, the runtime creates a global instance of each one soon after the code referencing them is loaded and used.

Static members don't need to be created using the `new` keyword, but are accessed using the name of the class they were defined within. They are accessible by all threads in an app domain (unless they are marked with the `[ThreadStatic]` attribute, which I'll come back to in a moment), and are never garbage collected because they essentially *are* root references in themselves.

Statics are a common and enduring source of root references, and can be responsible for keeping objects loaded in memory for far longer than would otherwise be expected.

Listing 1.15 shows the declaration of a static object and its initialization within a static constructor. Once loaded, the static constructor will execute, creating a static instance of the `Customer` class, and a reference will be held to an instance of the `Customer` class for the duration of the application domain (or the thread, if the reference is marked `[ThreadStatic]`).

```
public class MyData
{
  public static Customer Client;
  public static event EventType OnOrdersAdded;
  static MyData()
  {
       // Initialize
       Client=new Customer();
  }
}
```

Listing 1.15: Static reference example.

It's also worth remembering that any classes that subscribe to static events will remain in memory until the event subscription is removed, or the containing app domain finishes.

Static collections can also be a problem, as the collection itself will act as a root reference, holding all added objects in memory for the lifetime of the app domain.

Thread Statics

Sometimes you may want to prevent multiple threads accessing a common set of statics. To do this, you can add the `[ThreadStatic]` attribute to the member, and create multiple static instances of that member – one for each isolated thread (one instance per thread), as in Listing 1.16.

```
[ThreadStatic]
public static int NumberofThreadHits=0;
```

Listing 1.16: Marking a member `[ThreadStatic]` .