Chapter 6: Equals and GetHashCode

Section 6.1: Writing a good GetHashCode override

`GetHashCode` has major performance effects on Dictionary<> and HashTable.

Good `GetHashCode` Methods

- should have an even distribution
  - every integer should have a roughly equal chance of returning for a random instance
  - if your method returns the same integer (e.g. the constant '999') for each instance, you'll have bad performance
- should be quick
  - These are NOT cryptographic hashes, where slowness is a feature
  - the slower your hash function, the slower your dictionary
- must return the same HashCode on two instances that `Equals` evaluates to true
  - if they do not (e.g. because `GetHashCode` returns a random number), items may not be found in a `List`, `Dictionary`, or similar.

A good method to implement `GetHashCode` is to use one prime number as a starting value, and add the hashcodes of the fields of the type multiplied by other prime numbers to that:

```
public override int GetHashCode()
{
    unchecked // Overflow is fine, just wrap
    {
        int hash = 3049; // Start value (prime number).

        // Suitable nullity checks etc, of course :)
        hash = hash * 5039 + field1.GetHashCode();
        hash = hash * 883 + field2.GetHashCode();
        hash = hash * 9719 + field3.GetHashCode();
        return hash;
    }
}
```

Only the fields which are used in the `Equals`-method should be used for the hash function.

If you have a need to treat the same type in different ways for Dictionary/HashTables, you can use IEqualityComparer.

Section 6.2: Default Equals behavior

`Equals` is declared in the `Object` class itself.

```
public virtual bool Equals(Object obj);
```

By default, `Equals` has the following behavior:

- If the instance is a reference type, then `Equals` will return true only if the references are the same.
- If the instance is a value type, then `Equals` will return true only if the type and value are the same.
- `string` is a special case. It behaves like a value type.

```
namespace ConsoleApplication
{
    public class Program
    {
        public static void Main(string[] args)
        {
            //areFooClassEqual: False
            Foo fooClass1 = new Foo("42");
            Foo fooClass2 = new Foo("42");
            bool areFooClassEqual = fooClass1.Equals(fooClass2);
            Console.WriteLine("fooClass1 and fooClass2 are equal: {0}", areFooClassEqual);
            //False

            //areFooIntEqual: True
            int fooInt1 = 42;
            int fooInt2 = 42;
            bool areFooIntEqual = fooInt1.Equals(fooInt2);
            Console.WriteLine("fooInt1 and fooInt2 are equal: {0}", areFooIntEqual);

            //areFooStringEqual: True
            string fooString1 = "42";
            string fooString2 = "42";
            bool areFooStringEqual = fooString1.Equals(fooString2);
            Console.WriteLine("fooString1 and fooString2 are equal: {0}", areFooStringEqual);
        }
    }

    public class Foo
    {
        public string Bar { get; }

        public Foo(string bar)
        {
            Bar = bar;
        }
    }
}
```

Section 6.3: Override Equals and GetHashCode on custom types

For a class `Person` like:

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    public string Clothes { get; set; }
}

var person1 = new Person { Name = "Jon", Age = 20, Clothes = "some clothes" };
var person2 = new Person { Name = "Jon", Age = 20, Clothes = "some other clothes" };

bool result = person1.Equals(person2); //false because it's reference Equals
```

But defining `Equals` and `GetHashCode` as follows:

```
public class Person
{
    public string Name { get; set; }
```

```
    public int Age { get; set; }
    public string Clothes { get; set; }

    public override bool Equals(object obj)
    {
        var person = obj as Person;
        if(person == null) return false;
        return Name == person.Name && Age == person.Age; //the clothes are not important when
comparing two persons
    }

    public override int GetHashCode()
    {
        return Name.GetHashCode()*Age;
    }
}

var person1 = new Person { Name = "Jon", Age = 20, Clothes = "some clothes" };
var person2 = new Person { Name = "Jon", Age = 20, Clothes = "some other clothes" };

bool result = person1.Equals(person2); // result is true
```

Also using LINQ to make different queries on persons will check both `Equals` and `GetHashCode`:

```
var persons = new List<Person>
{
     new Person{ Name = "Jon", Age = 20, Clothes = "some clothes"},
     new Person{ Name = "Dave", Age = 20, Clothes = "some other clothes"},
     new Person{ Name = "Jon", Age = 20, Clothes = ""}
};

var distinctPersons = persons.Distinct().ToList();//distinctPersons has Count = 2
```

Section 6.4: Equals and GetHashCode in IEqualityComparator

For given type `Person`:

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    public string Clothes { get; set; }
}

List<Person> persons = new List<Person>
{
    new Person{ Name = "Jon", Age = 20, Clothes = "some clothes"},
    new Person{ Name = "Dave", Age = 20, Clothes = "some other clothes"},
    new Person{ Name = "Jon", Age = 20, Clothes = ""}
};

var distinctPersons = persons.Distinct().ToList();// distinctPersons has Count = 3
```

But defining `Equals` and `GetHashCode` into an `IEqualityComparator` :

```
public class PersonComparator : IEqualityComparer<Person>
{
    public bool Equals(Person x, Person y)
    {
        return x.Name == y.Name && x.Age == y.Age; //the clothes are not important when comparing
```

```
two persons;
    }

    public int GetHashCode(Person obj) { return obj.Name.GetHashCode() * obj.Age; }
}

var distinctPersons = persons.Distinct(new PersonComparator()).ToList();// distinctPersons has
Count = 2
```

Note that for this query, two objects have been considered equal if both the `Equals` returned true and the `GetHashCode` have returned the same hash code for the two persons.