Lecture 33 — ToString, Equality, Namespaces

J. Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

July 27, 2016

Acknowledgments: W.D. Bishop

Part I

ToString

Printing Objects

What happens if we try writing an object to the console with `Console.WriteLine( )`?

The class name is just printed to the console. That's not very helpful.

The system tries to convert the class to a console-printable form and that's the default behaviour.

The `ToString` Method

A `ToString( )` method automatically exists for every C# type.

The `ToString( )` method converts the value of an object to a string format in a (hopefully) meaningful way.

This method applies to both simple types and structured types.

The `override` Keyword

To replace the default behaviour of an existing method, like `ToString`, we need to make use of the `override` keyword.

This is our way of indicating that we want to replace the default behaviour provided for all objects in the system.

Now we are ready to write an implementation for `Coordinates`.

Implementing ToString for Coordinates

```
public override string ToString( )
{
    return "(" + x + ", " + y + ", " + z + ")";
}
```

After this, whenever an object of type `Coordinates` is to be printed to the screen, it will be formatted as `(x, y, z)`. E.g., (1, 7, -5).

Implementing ToString for Coordinates

The code below:

```
Coordinates c4 = new Coordinates( 1, 7, -5 );
Console.WriteLine( c4 );
```

will produce as output: `(1, 7, -5)`

Beyond Console Printing

The examples thus far have used `Console.WriteLine` but there are other scenarios where we might convert an object to a `string`.

`Console.WriteLine` has called `ToString` for us, but in other situations we are responsible for doing so.

The `ToString` method can be explicitly called any time we wish to convert an object to a `string`.

Example:

```
string positionText = "Your position: " + c4.ToString();
```

Part II

Object Equality

Equality

When using equality operators, == and `!=`, we have thus far used them on simple types like `int`; `7 == 0` evaluates to `false`.

When it comes to reference types, however, things don't always behave as we would expect.

What does it mean for two `Coordinates` objects to be equal?

Use of the == Operator on Objects

Our first instinct is to use the == operator.
This results in a test of reference equality.

This means that two objects will be considered equal if they are referring to the same object in memory.

```
Coordinates c1 = new Coordinates( );
Coordinates c2 = c1;
Coordinates c3 = new Coordinates( );
```

A test of `c2 == c1` will evaluate to true.
A test of `c1 == c3` will evaluate to false.

Use of the == Operator on Objects

Both `c1` and `c3` are initialized as (0, 0, 0).

Does this make intuitive sense when we look at this scenario?

```
Coordinates c1 = new Coordinates( 1.5, -2.0, 10 );
Coordinates c2 = c1;
Coordinates c3 = new Coordinates( 1.5, -2.0, 10 );
```

Even though they're both (1.5, -2.0, 10), `c1` and `c3` will not be considered equal by the == operator.

Comparing Objects

Our first approach to solving this might be to have the following statement to check if `c1` and `c3` are equal

```
c1.X == c3.X && c1.Y == c3.Y && c1.Z == c3.Z
```

This is tedious and error-prone every time it's needed in the code (and impractical if the objects have many fields).

Apply a previous solution to this problem: make it a function!

Comparing Objects

You might find it logical to implement a function `Equals` for this.
It turns out, so did the designers of C#.

Like `ToString( )`, every object in C# automatically has an `Equals` method that you can use to compare two objects.

Also like the default `ToString( )`, the default implementation of `Equals` is not very useful. It does the same as ==.

Implementing `Equals`

We can write our own implementation for `Equals`, but must follow these rules:

- `x.Equals(x)` returns true
- `x.Equals(y)` returns the same value as `y.Equals(x)`
- `if (x.Equals(y) && y.Equals(z))` returns true, then `x.Equals(z)` returns true
- Successive invocations of `x.Equals(y)` return the same value as long as the objects referenced by `x` and `y` are not modified
- `x.Equals(null)` returns false

Following these rules will prevent unexpected (surprising) behaviour.

Implementing `Equals`

The method signature for `Equals` is:

```
public override bool Equals( object obj )
```

The signature compares against a general `object` and not the same class where this method is being implemented.

Recall from earlier the concept of explicit type casting: we tell the system explicitly to treat one type as another.

In this example, we need to explicitly cast `object` to `Coordinates`.

Implementing Equals

Add the following to the Coordinates class:

```
public override bool Equals( object obj )
{
    if ( obj == null )
    {
        return false;
    }
    Coordinates other = (Coordinates) obj;

    return ( x == other.x )
        && ( y == other.y )
        && ( z == other.z );
}
```

Problem with the Previous Implementation

There is a problem with the implementation of `Equals` on the previous slide.

What if we called `Equals` with an actual parameter that is not an instance of `Coordinates`?

An error at this statement:

```
Coordinates other = (Coordinates) obj;
```

There is, however, a way to prevent this error: the `is` keyword.

The `is` Keyword

The `is` keyword gives us the ability to check if an object is an instance of a given type.

It is a binary operator that takes an object (on the left hand side) and a type name (on the right hand side), and evaluates to true or false.

Example: `obj is Coordinates`

Apply this keyword to prevent an error in the `Equals` method.

Implementing `Equals` with `is`

Add the following to the `Coordinates` class:

```
public override bool Equals( object obj )
{
    if ( obj == null )
    {
        return false;
    }
    if ( obj is Coordinates )
    {
        Coordinates other = (Coordinates) obj;

        return ( x == other.x )
            && ( y == other.y )
            && ( z == other.z );
    }
    else
    {
        return false;
    }
}
```

After Implementing `Equals`

After implementing the `Equals` method, what happens here?

Recall this from earlier:

```
Coordinates c1 = new Coordinates( 1.5, -2.0, 10 );
Coordinates c2 = c1;
Coordinates c3 = new Coordinates( 1.5, -2.0, 10 );
```

Still `c1 == c3` will return false, because the == operator is still reference equality.

But `c1.Equals( c3 )` will return true.

Another Problem with the `Equals`

You may have noticed we are doing an == comparison on `doubles`.
We've previously said we should have a tolerance; not equality.

For `Coordinates`, there's another motivation to use a tolerance.

Example: if two points are close enough when drawn on the screen we may choose to consider them to be the same.

Part III

Namespaces

Namespaces

A namespace is a collection of associated types.

Namespaces often correspond with packages, libraries, or application programming interfaces (APIs).

Namespaces permit code reuse by ensuring that types defined in one library do not conflict with those defined in another library.

Thus we could define two distinct classes called `Coordinates` and differentiate between them by putting them in different namespaces:

`Real.Coordinates` may represent (x, y)-coordinates;
`Polar.Coordinates` may represent polar (r, θ)-coordinates.

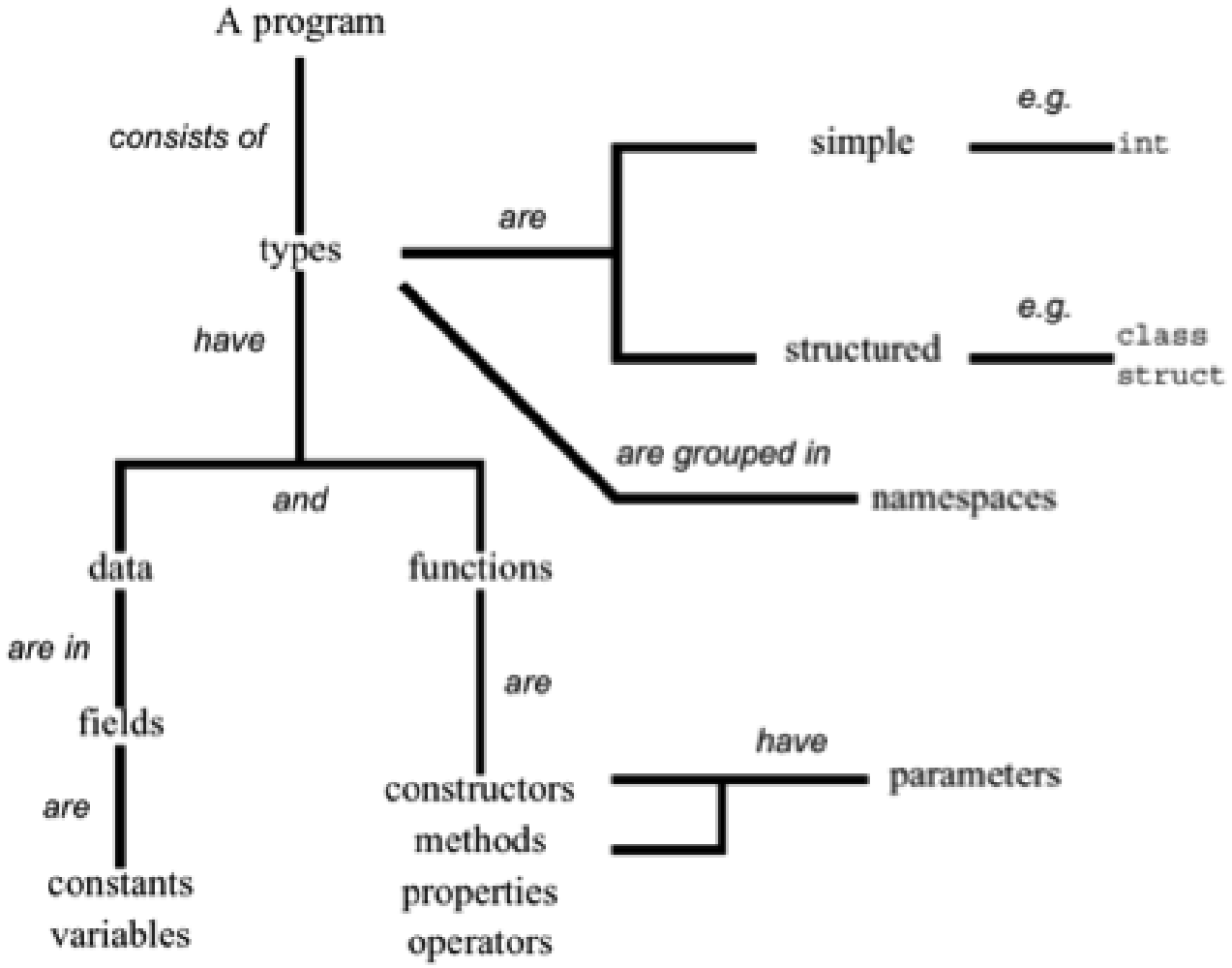
The `using` Keyword

The `using` keyword can be used to eliminate the need for specifying the full name of a type or a function.

For example, the statement `using System;` allows a C# program to use `System.Console.WriteLine( )` as `Console.WriteLine( )`

The scope of a `using` directive is limited to the file in which it appears; each file needs to specify its own `using` namespaces.

C# Type Overview



Source: J. Bishop and N. Horspool, C# Concisely, Pearson Education Canada, 2004.