Some objects are more equal than others

The many meanings of equality, value and identity

Roger Orr and Steve Love

ACCU 2011

Possible meanings of “equality”

1. Refer to the same memory location
2. Have the same value
3. Behave the same way

It is - believe it or not - harder than it looks, even ignoring #3.

Language * Equals

Curly bracket languages:

Java	`a == b` always does *something*
Java & C#	`object.[eE]quals` always does *something*
C++ & C#	You can overload the meaning of ==
C# & Java	You can override `[eE]quals` to customize behaviour

...and that's just 3* languages!

(* - yes, C# is sufficiently different to Java in this respect...)

a == b

Out of the box

C++ predefined for all built-ins and library types (e.g. `std::string`), fails to compile for any custom type

Java predefined for *primitive* built-ins, otherwise performs identity comparison for objects

C# predefined or overridden for all built-ins and lib types (references or values), fails to compile for custom value types, performs identity comparison for reference types

a == b

With some work

C++ you can define == for any type. Even built-ins (but this is prohibited)

Java you cannot change its meaning

C# you can define it for any custom type, but you must provide !=

a.[Ee]quals(b)

Out of the box

C++ doesn't have it

a.[Ee]quals(b)

Out of the box

Java overrides equals() for some types

```
public class IntegerEquals
{
  public static void main(String[] args)
  {
     test(10);
     test(1000);
  }

  public static void test(int value)
  {
     System.out.println("Testing " + value);
     Object obj = value;
     Object obj2 = value;
     if (obj.equals(obj2)) System.out.println("Equals");
     if (obj == obj2) System.out.println("==");
  }
}
```

a.[Ee]quals(b)

Out of the box

C# For reference types, the same as Java. For value types, it's more complicated...

```
struct Easy
{
    int X;
    int Y[ 100 ];
}
struct Hard
{
    int X;
    MyType Y;
}
```

null?

```
public class NullEquals
{
  public static void Main()
  {
      object a = null;
      object b = new object();

      if( a.Equals( b ) )
          Console.WriteLine( "Now there’s a thing" );

      if( object.Equals( a, b ) )
          Console.WriteLine( "This should be safe enough" );

  }
}
```

Floating point?

(We'll leave this for Dr Harris, who has made a cursory investigation of this recently...

For now, we note that floating point numbers *might not* obey normal rules for equality.)

So, are there more?

Java - no.

C++ there is `std::equal_to`, which by default performs ==. You can specialise it for your own type.

```
#include <functional>
#include <iostream>

int main()
{
    std::cout << "std::equal_to<int>()(10,10):  " << std::equal_to
        <int>()(10,10) << std::endl;
}
```

C# gets its own page...

C#’s list of equality measures

`object.Equals` (we’ve already seen)
`object.ReferenceEquals`
`IEquatable<T>`
`IEqualityComparer`
`IEqualityComparer<T>`
`EqualityComparer<T>`
`IStructuralEquatable`
`StringComparer`

...and others we’ve probably missed...

C# and value types

`object.ReferenceEquals` has an interesting property:

```
public class RefEqual
{
  public static void Main()
  {
    int ten = 10;
    System.Console.WriteLine(object.ReferenceEquals(ten, ten));
  }
}
```

Java has a similar problem with intern'ed strings

```
public class Intern
{
    private static final String s1 = "Something";
    private static final String s2 = "Some";
    private static final String s3 = "thing";

    public static void main( String[] args )
    {
        if( s1 == s2 + s3 ) System.out.println( "match!" );
    }
}
```

Over*load*ing

```
public class OverloadingEquals
{
  private int value;

  public OverloadingEquals(int initValue)
  {
    value = initValue;
  }

  public boolean equals(OverloadingEquals oe)
  {
    return oe != null && oe.value == value;
  }

  public static void main(String[] args)
  {
    OverloadingEquals oe1 = new OverloadingEquals(10);
    OverloadingEquals oe2 = new OverloadingEquals(10);

    Object obj1 = oe1;
    Object obj2 = oe2;

    System.out.println("oe1.equals(oe2): " + oe1.equals(oe2));
    System.out.println("oe1.equals(obj2): " + oe1.equals(obj2));
    System.out.println("obj1.equals(oe2): " + obj1.equals(oe2));
    System.out.println("obj1.equals(obj2): " + obj1.equals(obj2));
  }
}
```

Other ways of looking at it

The difference between equality and equivalence

```
a.Compare( b )
```

returns 0 when a and b are equal

```
!( a<b ) && !( b<a )
```

is a similar concept

Equality is...

- Reflexive
  - a==a is always true
- Commutative
  - if a==b then b==a
- Transitive
  - if a==b and b==c then a==c
- Reliable
  - Never throws.
  - This means checking for null!

C# rules

(In the list, x, y, and z represent object references that are not null.)

- x.Equals(x) returns true, except in cases that involve floating-point types. See IEC 60559:1989, Binary Floating-point Arithmetic for Microprocessor Systems.
- x.Equals(y) returns the same value as y.Equals(x).
- x.Equals(y) returns true if both x and y are NaN.
- If (x.Equals(y) && y.Equals(z)) returns true, then x.Equals(z) returns true.
- Successive calls to 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.

Java rules

- It is reflexive: for any non-null reference value x, x.equals(x) should return true.
- It is symmetric: for any non-null reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true.
- It is transitive: for any non-null reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.
- It is consistent: for any non-null reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the objects is modified.
- For any non-null reference value x, x.equals(null) should return false.

C++ rules

5.10/4 Each of the operators shall yield true if the specified relationship is true and false if it is false.

Polymorphic equality

```
class Coordinate
{
    public double X { get; set; }
    public double Y { get; set; }

    public override int GetHashCode() // {
           ... }
    public override bool Equals( object
          other )
    {
        var right = other as Coordinate;
        if( right !=null )
            return X == right.X && Y ==
                  right.Y;
        return false;
    }
}
```

```
class Coordinate3d : Coordinate
{
    public double Z { get; set; }

    public override int GetHashCode() // {
           ... }
    public override bool Equals( object
          other )
    {
        var right = other as Coordinate3d;
        if( right != null )
            return base.Equals( other ) &&
                  Z == right.Z;
        return false;
    }
}
```

Polymorphic equality

```
var p1 = new Coordinate { X = 2.3, Y = 5.6 };
var p2 = new Coordinate3d { X = 2.3, Y = 5.6, Z = 10.11 };
```

Ooops...

`p1.Equals( p2 )` is **True**

`p2.Equals( p1 )` is **False**

Polymorphic equality

```
var p1 = new Coordinate { X = 2.3, Y = 5.6 };
var p2 = new Coordinate3d { X = 2.3, Y = 5.6, Z = 10.11 };
```

Ooops...

`p1.Equals( p2 )` is **True**
`p2.Equals( p1 )` is **False**

Implementing `IEquatable<T>` fixes this for C# - except it doesn't!

```
var p3 = new Coordinate3d { X = 2.3, Y = 5.6, Z = 1.11 };
```

More ooops...

`p1.Equals( p2 )` and `p1.Equals( p3 )` but `p2.Equals( p3 )` could **still** be false

LSP The Liskov Substitutability Principle (a.k.a the Least Surprise Principle!)

Incidental and intentional equality

Avoid defining equality just so it can be used in conjunction with something that requires it, e.g. hashed containers.

- C++ `unordered_set` can be given its own equality comparer..
- Java `HashSet` can only use `object.equals()`, so you're stuck with it!
- C# `HashSet` can use a pluggable equality comparer (`IEqualityComparer<T>`)

Equality used for a key compare might be different than for other uses!

== and [Ee]quals are different!

The Ace of Spaces or *An* Ace of Spades?

In Java and C#, override [Ee]quals for a value-check.

In C++, explicitly compare addresses or contents. Copying and slicing can interfere with naive use of addresses.

Identity is important

```
class Thing : IEquatable< Thing >
{
    public override bool Equals( object other )
    {
        return Equals( other as Thing );
    }
    public bool Equals( Thing other )
    {
        if( other != null )
            return Value == other.Value;
        return false;
    }
    public static bool operator==( Thing left, Thing right )
    {
        return left.Equals( right );
    }
    public static bool operator!=( Thing left, Thing right )
    {
        return !( left == right );
    }
    public string Value { get; set; }
}
```

Identity is important

Reference equality matters

Over-ride C#'s default `operator==` at your own risk!

C# `ReferenceEquals`: for mad fools who override == but you have to use it explicitly

```
public bool Equals( Thing other )
{
    if( ! ReferenceEquals( other, null ) )
        return Value == other.Value;
    return false;
}
```

The supporting cast

== and != go together
C# and Java have no `[Nn]otEquals()`
What about comparison? I.e. <, >, `IComparable`, et.al.?

and...

Hashcodes

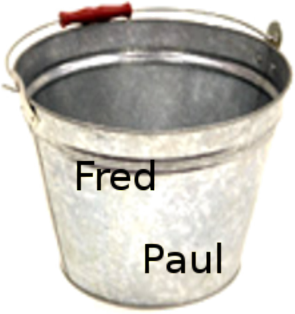
Equality and hashing

> "*classes [..] must [...] guarantee that two objects considered equal have the same hash code*"

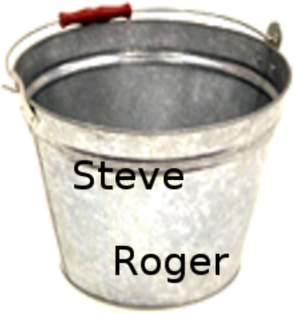
```
public static class Bogus
{
    public String Value;
    @Override public int hashCode()
    {
        return Value.hashCode();
    }
    @Override public boolean equals( Object other )
    {
        return ( ( Bogus )other ).Value.equals( Value );
    }
}
```

Consider what happens if Value changes after inserting into a hashed container...

Buckets of possibility



Bucket 1



Bucket 2

Boolean hilarity

```
using System;

static class Program
{
    struct HashTest
    {
        public bool Enabled;
        public string Value;
    }

    public static void Main()
    {
        var h1 = new HashTest{ Enabled = true, Value = "Great!" };
        var h2 = new HashTest{ Enabled = false, Value = "Great!" };

        Console.WriteLine( h1.GetHashCode() == h2.GetHashCode() );

        h1.Value = "Rubbish!";

        Console.WriteLine( h1.GetHashCode() == h2.GetHashCode() );
    }
}
```

Collections

When are two collections of things equal?

- Having the same items?
- ...in the same order?
- Does it matter?

(as a side note, we can add to the C# list of equality checks, with `SequenceEqual`, which insists on the same items, in the same order - polymorphic equality making some sense).

Making it all simple

Values and references

Know the difference between (polymorphic) reference types and value types in *all languages*. Never mix the two!

Immutability

Making value types immutable has many benefits, far beyond equality.

Polymorphism

If equality is only used for value types, which are immutable and do not participate in inheritance, almost all of the difficulties vanish.

Not making it (deliberately) difficult

Floating point numbers

Don’t play nicely with == or `[Ee]quals()`. There are alternatives.

Intentional vs. incidental

Make equals mean equals, *not* just equals for some cases.

Summary

Equality is hard to define *simply*
even for a single language.
It is *easy* to implement, with common sense rules.

see http://www.javapractices.com, and follow links through Overriding Object methods to implementing equals.

C# in a Nutshell has a deep exploration of equality in C#.