Equality

Equality

Should these two user objects be equal?

```
var user1 = new User() {FirstName = "Eric", LastName = "Svensson"};
var user2 = new User() {FirstName = "Eric", LastName = "Svensson"};

if(user1 == user2)
    Console.WriteLine("Equal");
else
    Console.WriteLine("Not equal");
```

Equality

In C# we have two types of equality

- **Value equality**
- **Reference equality**

Not fully understanding these two is a common source for errors

Value equality

Value equality means that the object contains the same values.

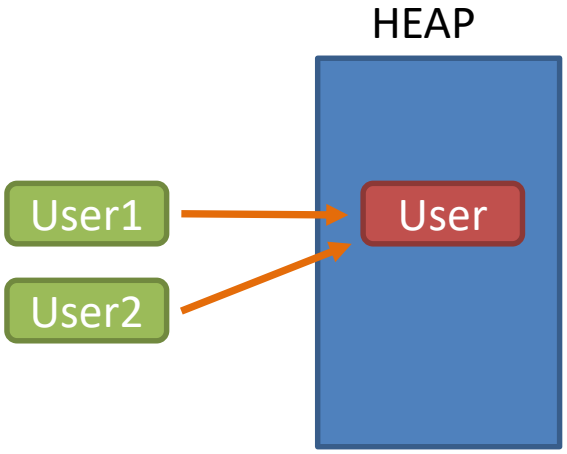
For basic value types like int, char, bool it is easy:

```
char a = 'x';
char b = 'x';
Console.WriteLine(a == b);

int x = 42;
int y = 42;
Console.WriteLine(x == y);
```

Reference equality

If user1 and user2 points to the same object, then they have reference equality

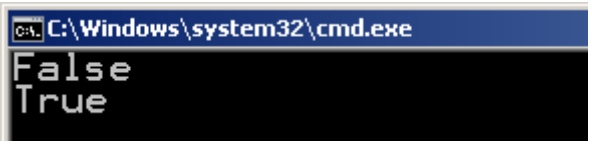
```
var user1 = new User() {FirstName = "Eric",
                        LastName = "Svensson"};
var user2 = user1;

if(user1 == user2)
    Console.WriteLine("Equal");
else
    Console.WriteLine("Not equal");
```



Reference equality

We can use the Object.ReferenceEquals method to test if two objects points to the same object.

```
var user1 = new User() { FirstName = "Eric", LastName = "Svensson" };
var user2 = new User() { FirstName = "Eric", LastName = "Svensson" };

Console.WriteLine(object.ReferenceEquals(user1,user2));

user2 = user1;

Console.WriteLine(object.ReferenceEquals(user1, user2));
```



Sorting

Sorting

Lets say we have defined this User class and list of users:

```
public class User
{
    public string FirstName;
    public string LastName;

    public User(string lastName, string firstName)
    {
        LastName = lastName;
        FirstName = firstName;
    }
}

var users = new User[]
{
    new User("Eric", "Svensson2"),
    new User("Eric", "svensson1"),
    new User("Anders", "Asplund"),
    new User("Urban", "Bläckberg")
};
```

Sorting

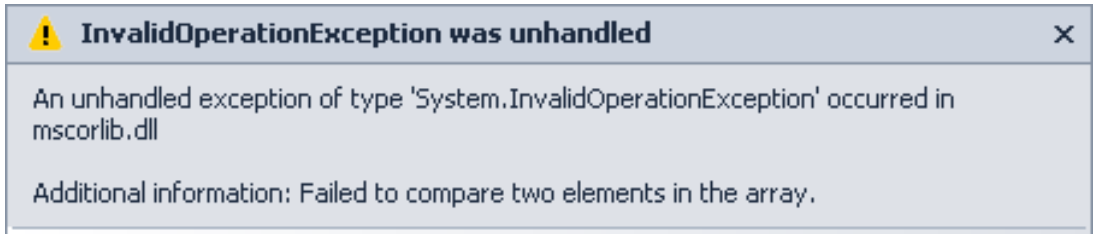
What happens if we try to sort this list?

```
var users = new List<User>
{
    new User("Eric", "Svensson2"),
    new User("Eric", "svensson1"),
    new User("Anders", "Asplund"),
    new User("Urban", "Bläckberg")
};

users.Sort();

foreach (var user in users)
{
    Console.WriteLine("{0} {1}", user.FirstName, user.LastName);
}
```

Sorting

We will get an InvalidOperationException

```
var users = new List<User>
{
    new User("Eric", "Svensson2"),
    new User("Eric", "svensson1"),
    new User("Anders", "Asplund"),
    new User("Urban", "Bläckberg")
};

users.Sort();

foreach (var user in users)
{
    Console.WriteLine("{0} {1}", user.FirstName, user.LastName);
}
```



The problem is that the sort method does not now how to compare two User items.

Sorting

To solve this we must implement the IComparable<T> interface

```
public class User: IComparable<User>
{
    public string FirstName;
    public string LastName;

    public User(string firstName, string lastName)
    {
        LastName = lastName;
        FirstName = firstName;
    }

    public int CompareTo(User other)
    {
        int result = FirstName.CompareTo(other.FirstName);
        return result;
    }
}
```

Returns	Meaning
<0	Current instance is less
=0	We are equal
>0	Current instance is more

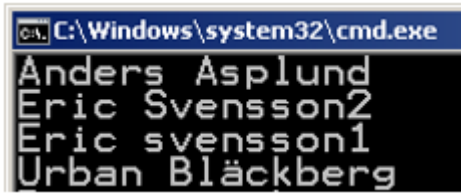
Now the user class can be sorted!

Sorting

But how do we get sorting on both First & Last name?

```
var users = new List<User>
{
    new User("Eric", "Svensson2"),
    new User("Eric", "svensson1"),
    new User("Anders", "Asplund"),
    new User("Urban", "Bläckberg")
};
```

Sort



Wrong order

What we can do is the following modification:

```
public int CompareTo(User other)
{
    int result = FirstName.CompareTo(other.FirstName);

    if(result == 0)
        result = LastName.CompareTo(other.LastName);

    return result;
}
```

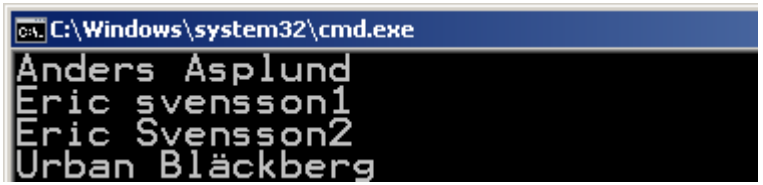
If first name is equal

Sorting

Now we get the desired sorting:

```
var users = new List<User>
{
    new User("Eric", "Svensson2"),
    new User("Eric", "svensson1"),
    new User("Anders", "Asplund"),
    new User("Urban", "Bläckberg")
};
```

Sort




```
public int CompareTo(User other)
{
    int result = FirstName.CompareTo(other.FirstName);

    if(result == 0)
        result = LastName.CompareTo(other.LastName);

    return result;
}
```

Equals

Equals

Lets say we want to remove a user from a list of User:

```
var users = new List<User>
{
    new User("Eric", "Svensson"),
    new User("Johan", "Andersson")
};

var userToRemove = new User("Eric", "Svensson");

bool result = users.Remove(userToRemove);

Console.WriteLine("Success: " + result);
```

What will the output be?

Equals

The problem is that .NET does not know how to compare two User objects.

To fix this we need to implement IEquatable<T>

```
public interface IEquatable<T>
{
    /// <summary>
    /// Indicates whether the current object is equal to another object of the same type.
    /// </summary>
    ///
    /// <returns>
    /// true if the current object is equal to the other parameter; otherwise, false.
    /// </returns>
    /// <param name="other">An object to compare with this object.</param>
    bool Equals(T other);
}
```

Equals

Let's implement IEquatable<T>

```
public class User : IComparable<User>, IEquatable<User>
{
    public string FirstName;
    public string LastName;

    //.. other code

    public bool Equals(User other)
    {
        if (other == null)
            return false;

        if (this.FirstName == other.FirstName &&
            this.LastName == other.LastName)
            return true;
        else
            return false;
    }
```

If we implement this interface, we should also override the **==** and **!=** operators.

Equals

Will this work if we use a non-generic collection?

```
var users = new ArrayList
{
    new User("Eric", "Svensson"),
    new User("Johan", "Andersson")
};

var userToRemove = new User("Eric", "Svensson");

bool userFound = users.Contains(userToRemove);

Console.WriteLine("User found: " + userFound);
```

No, the user is not found!

Equals

To make this work we also need to override the Object.Equals virtual method.

```
public override bool Equals(object obj)
{
    return this.Equals(obj as User);
}
```

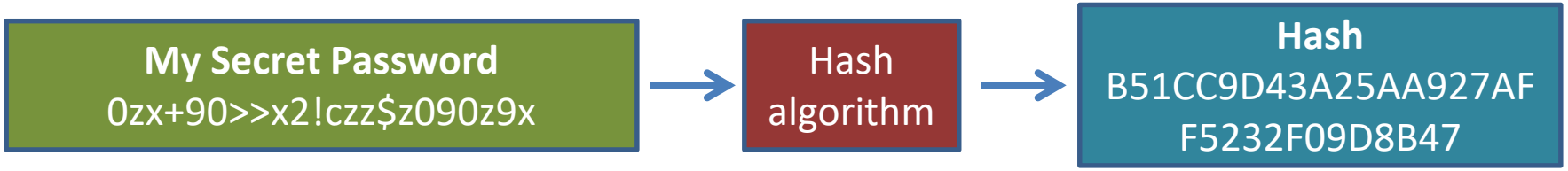
```
public bool Equals(User other)
{
    if (other == null)
        return false;

    if (this.FirstName == other.FirstName &&
        this.LastName == other.LastName)
        return true;
    else
        return false;
}
```

Now, the user is found!

Hash functions & GetHashCode

Hashing

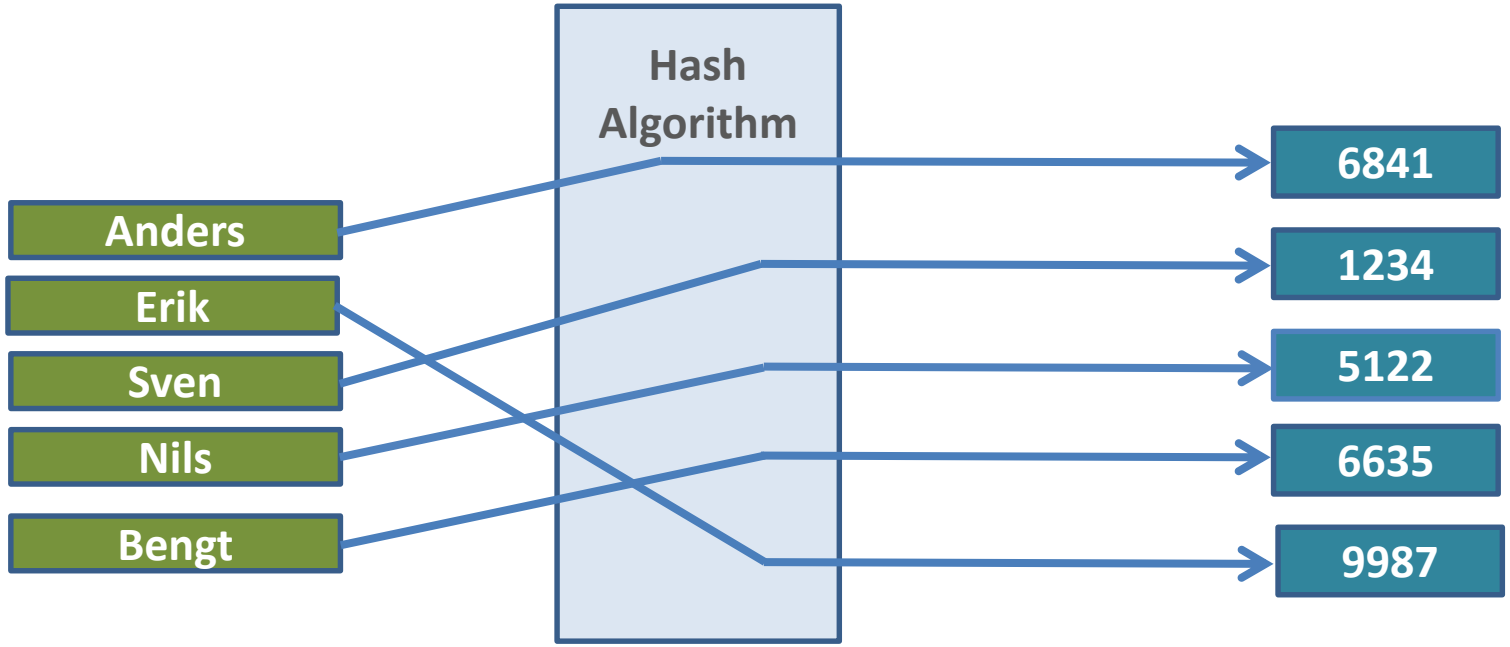
A hash function is any algorithm that maps a dataset of variable length to a smaller data set of a fixed length.



Hashing

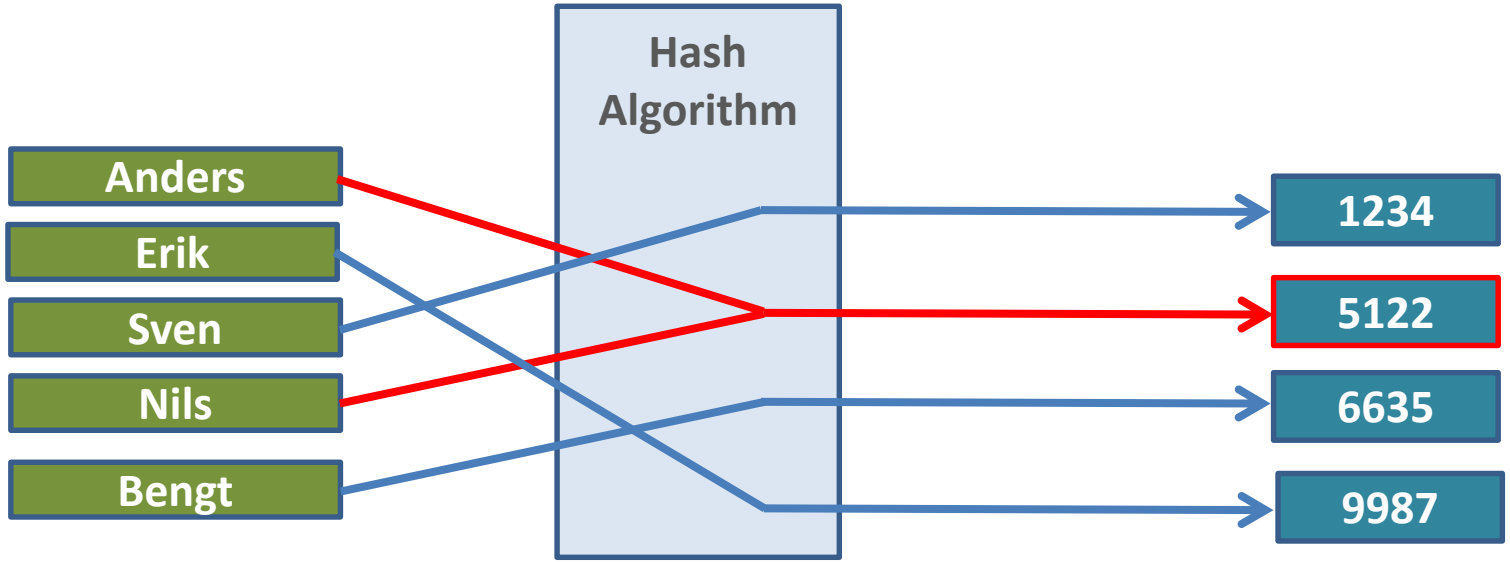
The idea with using hashing is that it is very expensive to:

- **Recreate the original data based from the hash**
- **Modify the original data, but still have the same hash value.**



Hashing

A hash collision occurs when two different inputs creates the same hash value.



Collisions are unavoidable, but with a proper algorithm that risk can often be ignored.

Announcing the first SHA1 collision
https://security.googleblog.com/2017/02/announcing-first-sha1-collision.html

Hashing

Some of the more common hashing algorithms are:

Name	Hash length	Speed	Memory need	Comment
MD5	128 bits	Fast	Low	Not recommended, very weak
SHA-1	160 bits	Fast	Low	Mathematical weakness exists
SHA-2	224,256, 384 or 512 bits	Fast	Low	Not cracked yet.
Bcrypt	448 bits	Slow	Low	
PBKDF2	Custom	Slow	Low	
Scrypt	Custom	Slow	High	Can be made to consume lots of memory

Hashing

Hashing of the string “Hello World!”, generates the following hash:

Algorithm	Hash value
MD5	ED076287532E86365E841E92BFC50D8C
SHA 1	2EF7BDE608CE5404E97D5F042F95F89F1C232871
SHA 256	7F83B1657FF1FC53B92DC18148A1D65DFC2D4B1FA3D677284ADDD200126D9069
SHA 512	861844D6704E8573FEC34D967E20BCFEF3D424CF48BE04E6DC08F2BD58C729743371015EAD891CC3CF1C9D34B49264B510751B1FF9E537937BC46B5D6FF4ECC8

GetHashCode

GetHashCode

What happens if we add two users to a HashSet collection?

```
var userSet = new HashSet<User>();

var userToAdd1 = new User("Eric", "Svensson");
var userToAdd2 = new User("Eric", "Svensson");

userSet.Add(userToAdd1);
userSet.Add(userToAdd2);

Console.WriteLine("The set contains {0} items.", userSet.Count);
```

What will the output be?

GetHashCode

To make this work we also need to override the GetHashCode method

```
public override int GetHashCode()
{
    unchecked
    {
        return ((LastName != null ? LastName.GetHashCode() : 0) * 397) ^
               (FirstName != null ? FirstName.GetHashCode() : 0);
    }
}
```

Now the output will be:

Summary

We should always implement the IEquatable<T> interface when we store object in collections

Methods like Contains, IndexOf, LastIndexOf, and Remove needs this

When implementing IEquatable<T> we should also implement IComparable<T>, Object.Equals & Object.GetHashCode

Exercise #5.1

Let's do exercise #5.1