1) What is actually happening here, is that we are trying to perform a refference equality.  
We are creating two new objects, where we set each one's value based on another integer.  
When comparing them using the == operator, it is like we are asking two people, both named John if they are the same person.  
Unlike Equals operator, where the assertion would succeed, as we are only comparing their value, and not the actual refference.

2) See CustomClassSort project in solution submitted.

3) See NaturalNumberProblem project

4) An abstract class, which can be inherited by multiple other classes, can define properties and/or functions which, subclasses can implement or overide.   
Abstracts are used where multiple classes share a common behaviour.  
Classes can only inherit from one abstract class.

An interface class only defines (no implementation, no fields) functionality which classes that inherit from it, must implement.

Interfaces are used when various implementations share only a method signature.

Classes can implement multiple interfaces.

5)The HashCode, equals() and == operators are all very closely related.

The GetHashCode method provides a numeric value, used to insert and identify an object in a hash-based collection, for algorithms that need quick checks of object equality.

Broadly, HasCode is considered as unique and finger-print like. Actually, the number of hashcodes available is relatively limited, therefore,2 objects that are the same share the same HashCode, but mathematically speaking, an object's hashcode is never actually unique, (around 4.2 billion unique number as HashCode is an int32).

The == operator refers to the refference equality whereas the Equals() function refers to the value equality. So, for example lets consider a custom Class called Customer, where 2 string properties exists, Name and Surname.

When instantiating 2 classes, setting the same values as below:

Customer c1 = new Customer {Name="John", Surname="Smith"} ;

Customer c2 = new Customer {Name="John", Surname="Smith"} ;

By running the below we will retrieve false in both cases.

Console.WriteLine(c1 == c2);

Console.WriteLine(c1.Equals(c2));

The reasons is because they are not the same refferences (2 different objects of the same object type). They do have the same values (Name and Surname) but their signature (HashCode) is different.

Therefore, in order to compare 2 objects of type customer we need to override the Equals method

(by comparing Name and Surname properties, along with null and type validations). Something like below:

public override bool Equals(object obj){

if (obj==null)

return false;

if (!(obj is Customer))

return false;

return this.Name== ((Customer)obj).Name &&

this.Surname== ((Customer)obj).Surname;

}

In order to override Equals method though, we need to override the GetHasCode() method as well, as it will still used the default, inherited from Object.

In this case the GetHasCode will look something like this:

public override int GetHashCode(){

return this.Name.GetHashCode() ^ this.Surname.GetHashCode();

}

6) An good example of an abstract class I have used in the past is when having multiple connection types to an application.

For example, an application could work with Oracle, MSSql, SQLLite etc.

An abstract class would define the functionality that its members who will inherit it will need to override (in some cases a default action can be defined in the abstract class, using it as a base class) which can be called by the child.

So an abstract class could have a LoadConnectionString function, which could have a default behaviour, but can also have abstract functions, that its members need to override.

Structs are very similar to classes, with the main difference being the way they are stored in memory.  
Structs are value types, allocated either on the stack or inline in containing types, whereas Classes are reference types, allocated on the heap and garbage-collected.

Struct allocation and de-allocation are in general cheaper than allocation and de-allocation of refference types.

In structs, each variable contains its own copy of the data, where as in the case of classes, multiple variables can hold the same memory address of a variable and a change in the class value would affect all instances.

Overall, structs are preffered to classes when the following 3 cases apply:

a) A single value is needed, such as primitive types (int, decimal, etc.).

b) The value is immutable (never altered).

c) The value is not boxed or unboxed frequently (cast to other types).

Static classes are sealed, therefore cannot be inherited and an instance can never be created. Also they only inherit from the Object class.

A static class can be used to compute functionality that does have not any instantiation of its members, and uses static content.

In generatl I consider static classes are a good use for retrieving configuration info which is fixed and do not change based on the user, and retrieving database configuration values that are constant and never change.

7) An event is a notification sent by an object to signal the occurrence of an action.

A delegate is a function pointer, holding a refference of one or more methods at runtime.

A delegate is in plain words an agreement, or a contract, between 2 or more code parts.

Let's take an example, of notifying customers by email and by sms once a procedure is completed.

We declare a delegate and the event in our main program, and once the procedure is completed we raise the event.

The sms sending program and the email sending program both have a function which is invoked once the event is triggered.

So, if we later decide to also perform another operation based on the end of the procedure, we simply add the same function which will be invoked.

The problem with the example is that the delegate does not have an object or EventArgs on its declaration.

Therefore, the information cannot be carried to the event, and in return, we will never be able to know which methods can be invoked.

Since a basic validation of the even is to check if there are any objects registered, the actions will never be called.

8) A monitor is a mechanism used in multithreading applications, for ensuring that only one thread accesses a certain critical piece of code.

A monitor has a lock, where one thread only may acquire it at any time.

Monitor works the same was as in saying lock (static object) and performing an operation. The difference is only the code, where we say Monitor.enter, where we enter the critical part, and Monitor.exit when we finish.

Mutext works similarly to monitor, with the difference that Monitor ensures lock from with a single process, whereas Mutex can ensure will mutual exclusion from multiple processes.

9) See FTP\_Download in solution.