1. String concatenation instead of StringBuilder

String concatenation works in such a way that every time when you add something to a string, a new address in the memory is being allocated. The previous string is copied to a new location with the newly added part. This is inefficient. On the other hand we have StringBuilder which keeps the same position in the memory without performing the copy operation. Thanks to the strings' appending by means of StringBuilder the process is much more efficient, especially in case of multiple append operations.

```
//INCORRECT
List values = new List(){"This ","is ","Sparta ","!"};
string outputValue = string.Empty;
foreach (var value in values)
{
    outputValue += value;
}
```

```
//CORRECT
StringBuilder outputValueBuilder = new StringBuilder();
foreach (var value in values)
{
    outputValueBuilder.Append(value);
}
```

2. LINQ – 'Where' with 'First' instead of FirstOrDefault

A lot of programmers find a certain set of elements by means of 'Where' and then return the first occurrence. This is inappropriate, because the 'First' method can be also applied with the 'Where' condition. What's more, it shouldn't be taken for granted that the value will always be found. If "First" is used when no value is found, an exception will be thrown. Thus, it's better to use FirstOrDefault instead. When using FirstOrDefault, if no value has been found, the default value for this type will be returned and no exception will be thrown.

```
//INCORRECT
List numbers = new List(){1,4,5,9,11,15,20,21,25,34,55};
return numbers.Where(x => Fibonacci.IsInFibonacciSequence(x)).First();
```

```
//PARTLY CORRECT
return numbers.First(x => Fibonacci.IsInFibonacciSequence(x));
```

```
//CORRECT
return numbers.FirstOrDefault(x => Fibonacci.IsInFibonacciSequence(x));
```

3. Casting by means of ‘(T)’ instead of ‘as (T)’ when possibly not castable

It’s common that software developers use simple ‘(T)’ casting, instead of ‘as (T)’. And usually it doesn’t have any negative influence because casted objects are always castable. Yet, if there is even a very slight probability that an object can be under some circumstances not castable, „as (T)” casting should be used. See Difference Between C# Cast Syntax and the AS Operators for more details.

```
1//INCORRECT
2var woman = (Woman)person;
```

```
1//CORRECT
2var woman = person as Woman;
```

4. Not using mapping for rewriting properties

There are a lot of ready and very powerful C# mappers (e.g. AutoMapper). If a few lines of code are simply connected with rewriting properties, it’s definitely a place for a mapper. Even if some properties aren’t directly copied but some additional logic is performed, using a mapper is still a good choice (mappers enable defining the rules of rewriting properties to a big extend).

5. Incorrect exceptions re-throwing

C# programmers usually forget that when they throw an exception using „throw ex” they loose the stack trace. It is then considerably harder to debug an application and to achieve appropriate log messages. When simply using „throw” no data is lost and the whole exception together with the stack trace can be easily retrieved.

```
01
02//INCORRECT
03try
04{
05    //some code that can throw exception [...]
06}
07catch (Exception ex)
08{
09    //some exception logic [...]
10    throw ex;
11}
12
```

```
01//CORRECT
02try
03{
04    //some code that can throw exception [...]
05}
```

```
catch (Exception ex)
{
    //some exception logic [...]
    throw;
}


```

6. Not using ‘using’ for objects disposal

Many C# software developers don’t even know that ‘using’ keyword is not only used as a directive for adding namespaces, but also for disposing objects. If you know that a certain object should be disposed after performing some operations, always use the ‘using’ statement to make sure that the object will actually be disposed.

```

//the below code:
using(SomeDisposableClass someDisposableObject = new SomeDisposableClass())
{
    someDisposableObject.DoTheJob();
}
//does the same as:
SomeDisposableClass someDisposableObject = new SomeDisposableClass();
try
{
    someDisposableObject.DoTheJob();
}
finally
{
    someDisposableObject.Dispose();
}

```

7. Using ‘foreach’ instead of ‘for’ for anything else than collections

Remember that if you want to iterate through anything that is not a collection (so through e.g. an array), using the ‘for’ loop is much more efficient than using the ‘foreach’ loop. See Foreach vs For Performance for more details.

8. Retrieving or saving data to DB in more than 1 call

This is a very common mistake, especially among junior developers and especially when using ORMs like Entity Framework or NHibernate. Every DB call consumes some amount of time and therefore it’s crucial to decrease the amount of DB calls as much as possible. There are many ways to do so:

- Using fetching (Eager Loading)
- Enclosing DB operations in transactions
- In case of a really complex logic, just moving it to the DB layer by building a stored procedure