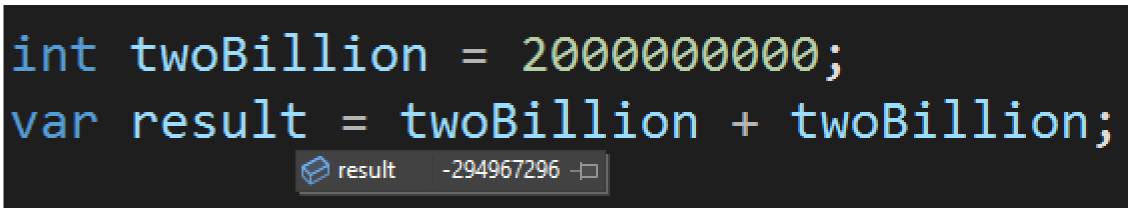
21. What is the purpose of the “checked” keyword?

Brief summary: The “checked” keyword is used to define a scope in which arithmetic operations will be checked for overflow.

The “checked” keyword is used to define a scope in which arithmetic operations will be checked for overflow.

To understand this slightly mysterious sentence we must first understand how arithmetic operations work in C# in general. In everyday programming, we don’t think too much about it, and perhaps we even assume the “programming arithmetics” is exactly the same as arithmetics we learned about in school. For example, we assume that the sum of two positive numbers must be a positive number. This is perfectly valid in real life, but not necessarily in programming.

For example, if I add two billion to two billion in C#, I will not get four billion. Instead, I will get this:



```
int twoBillion = 2000000000;
var result = twoBillion + twoBillion;
```

result -294967296

The result is -294967296. I added two positive numbers, and I got a negative number as a result.

To understand why this happened it is crucial to understand binary numbers, and how are they represented in the computer’s memory. Revisit the “How does the binary number system work?” lecture to find out.

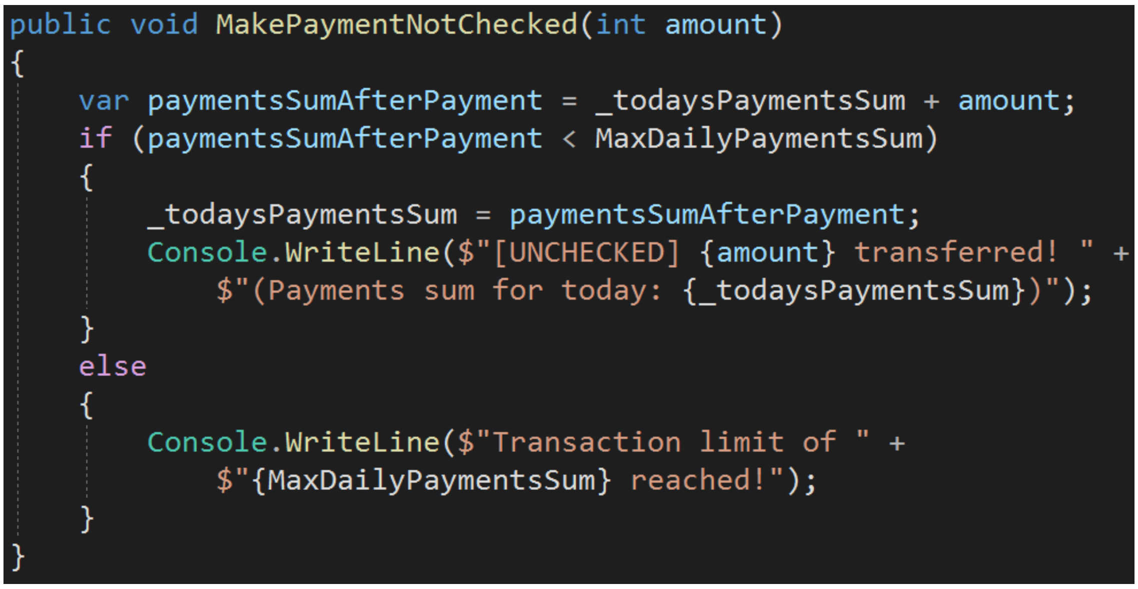
I assume that by now you know that every number we use when programming is simply a sequence of bits. The important thing to realize is that on a limited number of bits we can store a limited number (the same as in a decimal number system - the biggest number represented with 3 digits is 999). For example, with 4 bits the largest number that can be represented is 15 (because if each bit is set to one, then the number is 1 + 2 + 4 + 8 = 15.

If we want to represent a bigger number, we simply need more bits of memory. Every basic numeric type in C# has a certain number of bits that it occupies in memory. For example, for integer, it’s 32 bits (which FYI is 4 bytes - one byte is 8 bits). This means, the largest number an integer can be is 2147483647, which is a little more than two billion. So what happens when we add two billion to two billion? In “real” mathematics it would give 4 billion, but such a huge number is simply impossible to represent with integer type in C#. In this case, so-called “number overflow” happens, resulting in an unexpected result. This number is not random, and it's determined by how the addition of binary numbers works. You can read more about it here:
https://www.sciencedirect.com/topics/computer-science/binary-addition#:~:text=Addition%20is%20said%20to%20overflow,in%20the%20remaining%20four%20bits.

It is crucial to understand that when number overflow happens **no exception is thrown** - the program continues to work normally. You may be a bit surprised by it - usually when we do something invalid, like accessing a nonexistent index in an array or dividing by zero - an exception is thrown, informing us what happened. Exceptions are a good thing, actually. It’s better to be clearly informed that something went wrong.

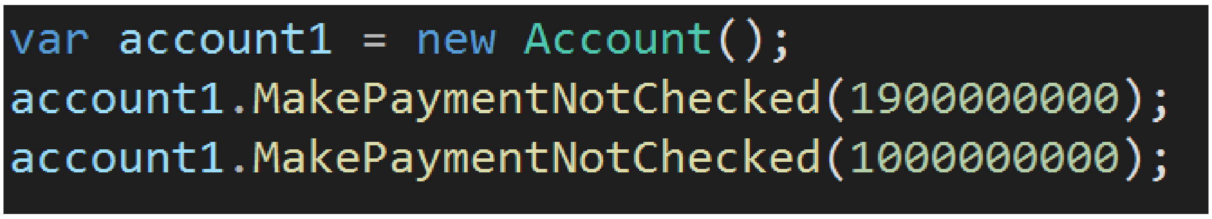
The number overflow is a “**silent failure**” - the program doesn’t work correctly, but it continues to work without exception. This can have disastrous effects. Invalid data may be stored in databases, overwriting old, valid data. Also, the program may continue and allow further invalid operations.

For example, imagine a banking system, which stores a sum of daily transactions and blocks any further payments if some limit has been exceeded. Let’s imagine a very rich customer who is allowed to pay up to two billion of some currency daily. If the sum of daily payments exceeds two billion, the next payments will be blocked. Let’s see a sample code:



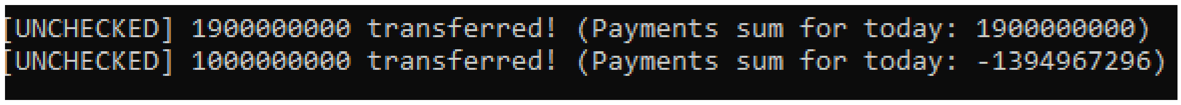
```
public void MakePaymentNotChecked(int amount)
{
    var paymentsSumAfterPayment = _todaysPaymentsSum + amount;
    if (paymentsSumAfterPayment < MaxDailyPaymentsSum)
    {
        _todaysPaymentsSum = paymentsSumAfterPayment;
        Console.WriteLine($"[UNCHECKED] {amount} transferred! " +
            $"(Payments sum for today: {_todaysPaymentsSum})");
    }
    else
    {
        Console.WriteLine($"Transaction limit of " +
            $"{MaxDailyPaymentsSum} reached!");
    }
}
```

Now let’s say the client makes a payment of 1 900 000 000 (almost two billion), and then tries to make the next one of 1 000 000 000 (one billion).



```
var account1 = new Account();
account1.MakePaymentNotChecked(1900000000);
account1.MakePaymentNotChecked(1000000000);
```

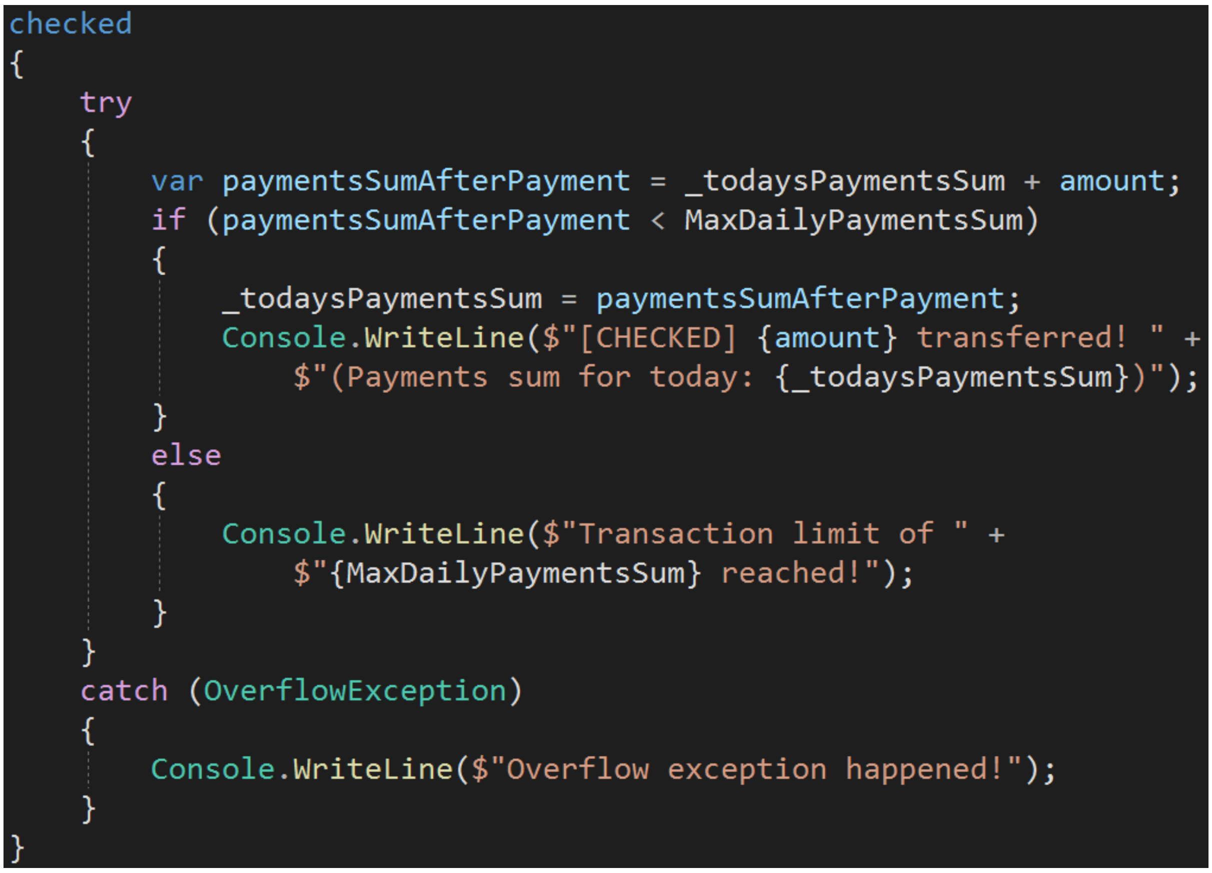
The second transaction should be blocked because the sum is over the limit of two billion, but actually, it will be allowed, because a daily sum becomes a negative number due to arithmetic overflow. And of course, any negative number is less than two billion.



```
[UNCHECKED] 1900000000 transferred! (Payments sum for today: 1900000000)
[UNCHECKED] 1000000000 transferred! (Payments sum for today: -1394967296)
```

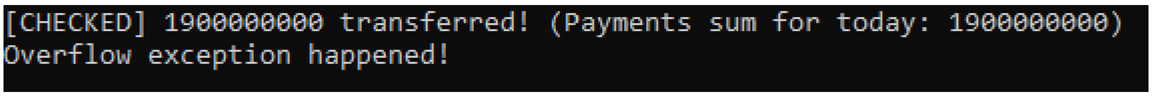
We will allow the client to make more and more payments. And what if those payments are actually done by someone who hacked the client’s account? Now the client may lose all the money instead of some limited sum.

I hope I convinced you that arithmetic overflows are dangerous. So how to deal with them? Well, this is where the “checked” keyword comes in handy. The “checked” keyword defines a scope in which arithmetic operations will be checked for overflow. If it happens, an exception will be thrown.



```
checked
{
    try
    {
        var paymentsSumAfterPayment = _todaysPaymentsSum + amount;
        if (paymentsSumAfterPayment < MaxDailyPaymentsSum)
        {
            _todaysPaymentsSum = paymentsSumAfterPayment;
            Console.WriteLine($"[CHECKED] {amount} transferred! " +
                $"(Payments sum for today: {_todaysPaymentsSum})");
        }
        else
        {
            Console.WriteLine($"Transaction limit of " +
                $"{MaxDailyPaymentsSum} reached!");
        }
    }
    catch (OverflowException)
    {
        Console.WriteLine($"Overflow exception happened!");
    }
}
```

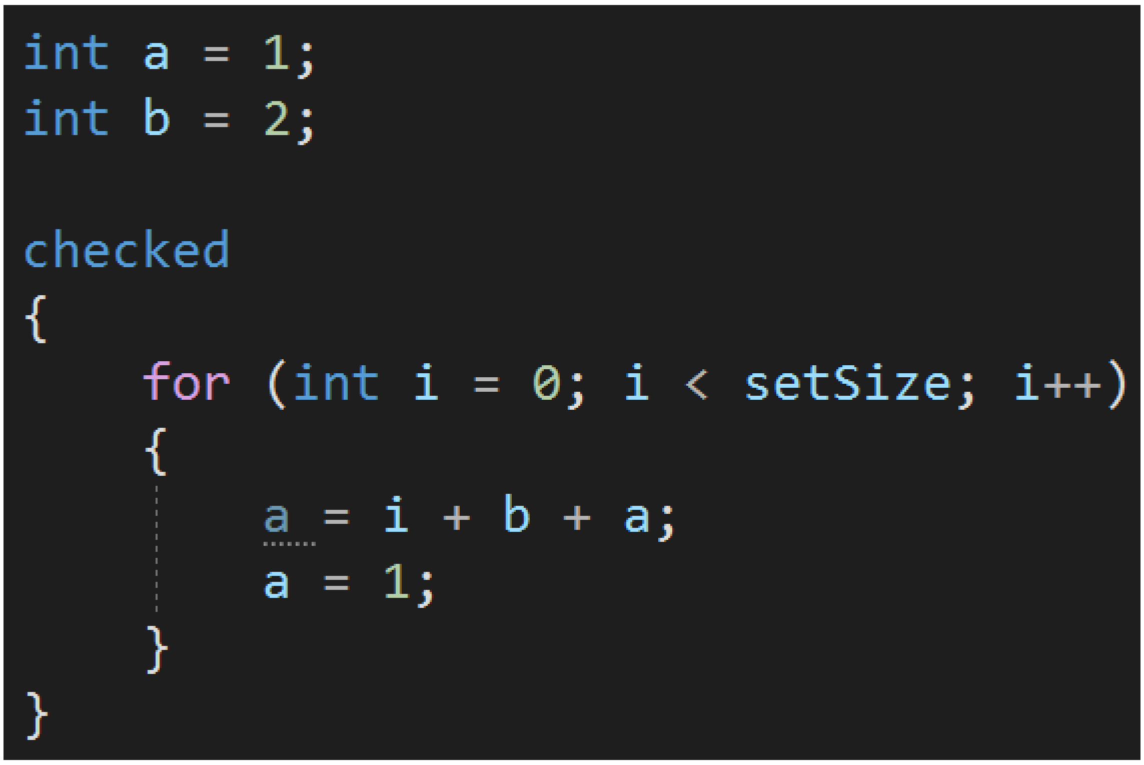
In this scope, any overflow will throw an exception instead of failing silently.



```
[CHECKED] 1900000000 transferred! (Payments sum for today: 1900000000)
Overflow exception happened!
```

You may wonder "why isn't this done by default?" Well, the reason is simple - **it's performance**. Computers are very good at doing arithmetic operations and they do them extremely fast. On the other hand, checking for overflow is actually a relatively complex operation, and it takes some time. If we have a lot of arithmetic operations in the application, the performance impact may be noticeable.

I've created a quick little program that measures the performance difference for checked and unchecked operations. You can find it in the repository attached to the course. In short, this loop is executed and measured in both checked and unchecked context:



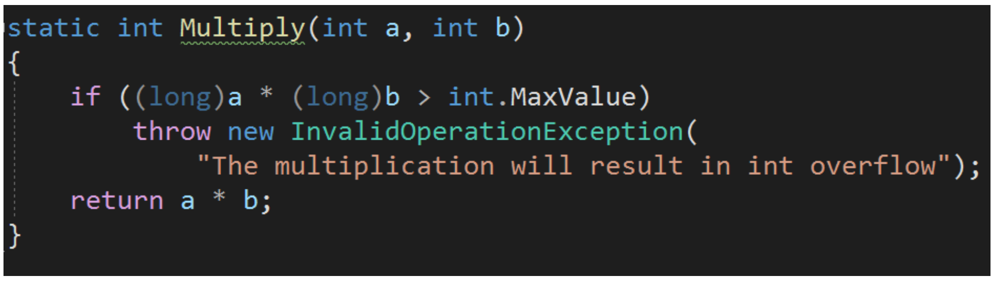
```
int a = 1;
int b = 2;

checked
{
    for (int i = 0; i < setSize; i++)
    {
        a = i + b + a;
        a = 1;
    }
}
```

On my computer, for **setSize** set to **one billion**, it takes on average 3257 milliseconds for **checked** context and 2431 for **unchecked**. That means the **checked operations took 33% more time**. As you can see the difference is not huge, but it is noticeable.

All right. We now have the basics of theory about the checked keyword. Let's think about how to apply it in practice. Here is a couple of tips:

- be aware of the limitations of the types you are using.
- choose proper numeric types for given usages. Do you need a counter of elements the user selected from the list that by design shows no more than 100 elements? Feel free to use **byte** - it's tiny, but the limitation to 255 is enough. On the other hand, what if you need a number representing the total number of financial transactions ever made in your banking application? **Int** sounds good, but what if your application becomes a roaring success and soon the number slightly over two billion is not enough? In this case, **long** may be a better choice - its max value is over 4 billion times larger than the max value of int.
- in case your number must be unlimited (for example you are an astronomer and you want to measure the galaxy size in millimeters) don't forget about **BigInteger** type. BigIntiger is only limited by the size of the memory of your computer, so you can represent gigantic numbers with it.
- remember that an overflow is not always a problem. For example, they are perfectly fine to happen when calculating a hash code of some object.

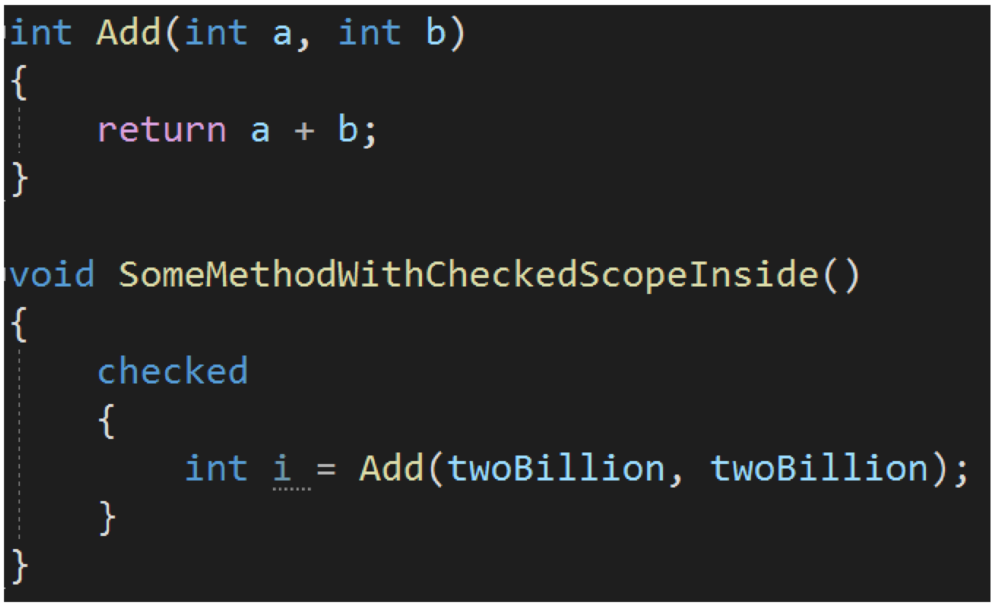
- if you have even the slightest concern that an undesired overflow may happen, you have two choices:
    - put this code in the checked context so an exception is thrown in case of an overflow
    - check for overflow before an actual operation, for example like this:



```
static int Multiply(int a, int b)
{
    if ((long)a * (long)b > int.MaxValue)
        throw new InvalidOperationException(
            "The multiplication will result in int overflow");
    return a * b;
}
```

      In my test application, it turned out that checking the overflow like this is actually better from the performance point of view than using the checked keyword (the test took 3257 milliseconds for checked scope and 3017 for manual checking for overflow). Please note that which one is performance-wise better depends a lot on a particular situation. If you are in doubt, it's best to run some benchmarks on your own.
- if you really need to, you can set the project setting to check arithmetic operations by default. In this case, if you want some code to be unchecked, you can use the "unchecked" keyword to define a scope in which the arithmetic operations are not checked.

Before we move on, there is an important caveat you must know about: the overflow check only applies to the immediate code block, not to any function calls inside the block. To understand this, let's consider the following code:



```
int Add(int a, int b)
{
    return a + b;
}

void SomeMethodWithCheckedScopeInside()
{
    checked
    {
        int i = Add(twoBillion, twoBillion);
    }
}
```

What do you think will happen? At the first glance, you might think that the OverflowException will be thrown. After all, we call the Add method in the checked scope, so adding two billion to two billion shall cause an overflow.
Well, actually it’s not true. The “checked” keyword doesn’t affect any methods that are called within it. If we want this code to actually be checked, we must add the “checked” keyword **inside** the Add method.

Let’s summarize. The “checked” keyword is used to define a scope in which arithmetic operations will be checked for overflow. If this keyword will not be used (or overflow checking will not be enabled on the project level) the arithmetic overflow will not cause an exception, but will simply result in an invalid value.

Bonus questions:

- "**What is the purpose of the "unchecked" keyword?**"
  This keyword defines a scope in which check of arithmetic overflow is disabled. It makes sense to use it in projects in which the checking for overflow is enabled for an entire project (can be set on the project level settings).

- "**What is a silent failure?**"
  It’s a kind of failure that happens without any notification to the users or developers - they are not informed that something went wrong, and the application moves on, possibly in an invalid state.

- "**What is the BigInteger type?**"
  It’s a numeric type that can represent an integer of any size - it is limited only by the application’s memory. It should be used to represent gigantic numbers (remember that max long is over 4 billion times larger than max int, which is a bit more than two billion, so BigInteger should be used instead of long only to represent unthinkably large numbers).