Introduction to C#

Collections

Exceptions

Collections

Collections are objects used to group data.

They consist of a single object reference, and a series of values stored just like in variables.

You access all the values through the same object reference, using an index – an ***identifier*** that is unique to each value in the collection.

There are three main types of collection:

- Arrays
- Lists
- Dictionaries

Collections - Arrays

Arrays are the most basic type of collection

They are defined for a ***specific type*** of variable, and always have a ***fixed size***

```
int[] intArray = new int[10];
```

Arrays store values using a ***0-based index*** as an identifier. An array with 5 elements will have the indexes 0-4 for its elements.

Arrays offer good performance, but don’t have many of the conveniences that other collections do. For example, they have to be ***manually resized*** when the amount of elements needs to change.

```
intArray[0] = 57;
intArray[1] = 68;
intArray[2] = 24;
intArray[3] = 12;
```

Collections - Multidimensional Arrays

Arrays can have more than one dimension - adding additional sets of indexes to store data in. Each combination of indexes will be unique, so each new dimension will increase the amount of potential values stored dramatically.

The amount of dimensions has to be defined when the array is created, and every time any value is referenced, an index must be included for each dimension

```
int[] intArray1 = new int[5];
int[,] intArray2 = new int[5,5];
int[,,] intArray3 = new int[5,5,5];
int[,,,] intArray4 = new int[5,5,5,5];

intArray1[3] = 12;
intArray2[0,3] = 57;
intArray3[1,2,5] = 68;
intArray4[2,4,1,2] = 24;
```

Collections - Lists

The List collection works much like an array does, but it also comes with its own conveniences added in:

It is ***dynamically sized***, so if you add more elements into it, it will resize itself as needed.

It has many ***built-in methods*** to deal with things like finding items within the list, sorting and changing items.

As with arrays, it is defined for a specific object type, and can use indexes to refer to individual values. Unlike arrays, they use ***Generics*** to define their type.

```
List<int> listInt = new List<int>();
```

Collections - Lists - Useful members

Some of the useful members in the List collection include:

Add	Inserts a given item into the List
Remove	Removes a given item from the List
RemoveAt	Removes the item at a given index from the List
AddRange	Adds all the items in a given List to the List
FindIndex	Finds the index of a given item
Contains	Checks if a given item is in the list
Count	Returns the number of items that are in the list
Clear	Removes all items in the list

Generics

Generics are a kind of parameter for a type, used to define what other type(s) it should refer to when it is used.

A good example of this is the List and Dictionary classes, that use generics to define which types of values they can store. They are defined like this in their class definitions:

```
List<T>
Dictionary<TKey, TValue>
```

The <T>, <TKey> and <TValue> parts of the class definitions means that they can be replaced with any kind of data type or class, when the actual instance of the class is created, like so:

```
List<int> listInt = new List<int>();
Dictionary<int,string> dict = new Dictionary<int, string>();
```

Collections - Dictionaries

A Dictionary has much in common with a List – they are both single-dimensional collections of objects, they both use generics for their type.

However, a Dictionary uses two types – a Key type and a Value type.

The Value is the same as the type of a List – it is the actual value stored.

```
Dictionary<string,DateTime>
```

The Key, meanwhile, is used as another identifier for that specific value, allowing you to use more complex values than a numeric index to reference a value:

```
dict["today"] = DateTime.Now;
```

Collections - List Initialization

Collections can be created and then assigned value, but much like variables they can also be initialized. This is done through object initalzation:

```
List<int> listInt = new List<int> { 5,7,8,1,2,7,1 };
```

The ***new*** keyword creates the List object, and the code segment after it defines the content of the list. Dictionaries work the same way, but each element is its own list, with the Key and Value of each:

```
new Dictionary<string, string> { {"Eric","eric.lindroth@lexicon.com"},
                                 {"Fredrik","fredrik.odin@lexicon.com"},
                                 {"Kent","kent.gudmundsen@lexicon.com"},
                               };
```

Collections - *foreach* Loops

When dealing with collections, you often need to loop through all of the items contained inside them. While you can use a *while*- or *for*-loop to do this, you can also use a *foreach*-loop:

This kind of loop will take each item stored in the collection, in turn, assign it to a temporary variable, and run through the loop's code with it. After this, it will load the next value into the variable, and repeat, until all items have been looped through.

```
foreach (int item in listInt) {
    Console.WriteLine(item);
}
```

Exceptions

An exception is a message generated by the program, when it encounters an error in the code. They exist in order to make sure that such errors don’t cause crashes or more severe damage to the system.

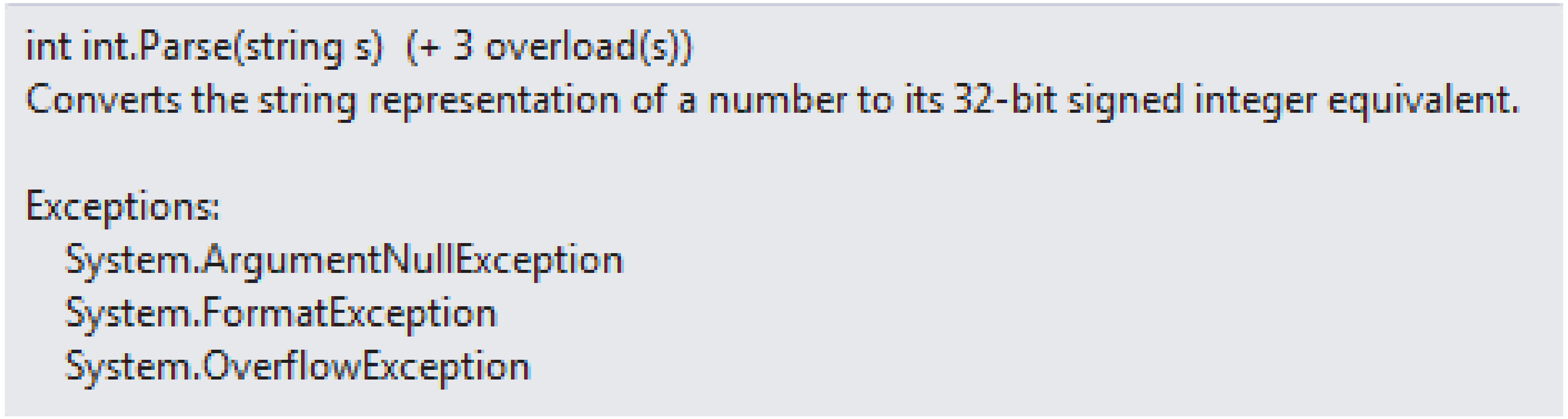
They are created through the **throw** keyword, and use a class called Exception as their base. Once an exception is thrown, the method that is active will end, and the exception will be ”thrown” back to the code that called the method, until it finds a **catch**-statement that matches its type.

```
class CustomException : Exception {
    public CustomException(string message) {
    }
}
```

```
if (value == null) {
    throw new CustomException("value is null");
}
```

Exceptions - Identifying Possible Exceptions

Information about which exceptions a method can cause can be found in the tooltip, under the label ”Exceptions”.



```
int int.Parse(string s)  (+ 3 overload(s))
Converts the string representation of a number to its 32-bit signed integer equivalent.

Exceptions:
   System.ArgumentNullException
   System.FormatException
   System.OverflowException
```

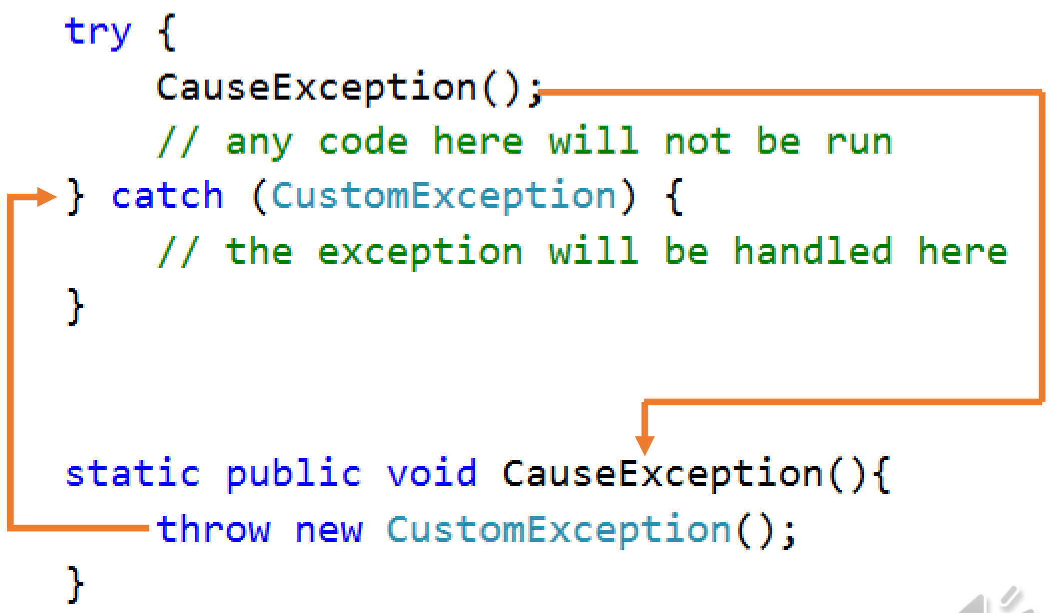
These are all the exceptions that are ***possible***. Trying to handle other specific exceptions is a waste of time, since they can never occur.

Exceptions - *try-catch* Statements

When an exception is thrown, it will need to be caught or it will cause the program to fail. In order to do that, you need a **try** statement to run the code in, and a **catch** statement that matches the exception.

You can catch many exception types with the same try-catch statement, but the same rule applies here as with if-statements – only the first one will be caught.

If a catch statement is not given a type, it will catch **all** exceptions instead.



```
try {
    CauseException();
    // any code here will not be run
} catch (CustomException) {
    // the exception will be handled here
}


static public void CauseException(){
    throw new CustomException();
}
```

Exceptions - *finally*

A **finally** statement may be added to the end of a try-catch, if one wants to have code that is always run, no matter what happens in the try-catch. Even if the method ends in the try-catch, through a **return** or a **throw**, the code in the **finally**-statement will be run.

```
try {
    CauseException();
    // any code here will not be run
} catch (CustomException) {
    // the exception will be handled here
} finally {
    // this code will always happen, even
    // if the method ends in try or catch
}
```