## Learned from Exercise – Exercism Crypto Square

Actual log

1. Learned about REGEX expressions in order to apply various string normalizations  
   as parts of a cryptographic ciphering technique.
2. Broke through my faulty understanding of Polymorphism being confined and exclusively comprised of interface and base-type useages. The catch-block of a try-catch statement f.e.  
   employs a third form of polymorphism by binding a thrown exception type instance to the argument of the catch-block parentheses causing any type of exception to be polymorphised as whatever exception type is used in the signature of the catch-block parentheses.  
   This then constitutes polymorphism as no cast, interface or base type got crucially involved.
3. Discovered a neat way to hide complex types using type aliases

REGEX expressions for Spaces and Punctuation removal

You can use a REGEX expression to remove both with respectively “\p{P}” and “\s”.  
These can be explained using [this website](https://regex101.com/r/ZFIKpv/1) to:

* \p{P} matches any kind of punctuation character matches the characters literally   
  (case sensitive)
* \s matches any whitespace character (equivalent to [\r\n\t\f\v ])

Conundrum: What’s the type of a Catch-block exception argument?

A computer code with text

Description automatically generated

Conventional employ of the one-of-three Polymorphism pillar of the OOP paradigm asks us to restrict  
the potential use-case of a specific type to a generalized type by a shared baseclass or interface.

This allows a de-coupling between the statically typed nature of C# and the ability of a use-case context to reference children of a certain hierarchy without actually knowing about all the sub-types.  
Like so, expected changes to the structure of a hierarchy do not cause changes to implementations.

However, inside of the catch-block of a try-catch statement we find the result of a “ex.GetType()” to be the output of an identical application of this method chain to the casted-from sub-type.

This adjacent situation to the conventional Polymorphism ones is caused by the retention of casted-from sub-types by reference through a bound intermediary variable inside of the catch parentheses.  
Despite an observation of the “Exception” typing of the “ex” variable, any retrieval of the type information through the implied respective method chain does not apply to the observed type at all.

This means that Polymorphism includes cases wherein a ploymorphised variable does not lose any of the original typing, and merely serves an intermediary generic state including a binding to the original.  
Technically speaking, this also implies that whilst seemingly obvious, no upcast is performed here.

The typing of a catch-block exception argument is literally what’s observed as the typing of the exception argument typing, but the Runtime Type Idendification (RTTI) type introspection mechanism is instructed to skip over the explicit argument typing and focus on the bound-to orginal instance.

In other words, we actually do refer to a polymorphised Exception variable inside of the catch-block.  
Yet, we also observe that method calls on that variable are sent down to the actual original instance.

**The answer to “what’s the output of the console in the code screenshot” would then be “the FQN of the OutOfMemoryException type” and not the seemingly plausible “FQn of the Exception type”.**

**We also learned that Polymorhism is defined by an unaware user and an origin-aware compiler;  
and not by employing either an interface or a baseclass. Polymorphism isn’t restricted to two forms.**

How to hide complicated types, devoid of verbosity

## Custom Type Alias Using using Directive in C#

### Introduction

In C# 10, a feature was introduced that allows you to create type aliases using the using directive. This feature simplifies the usage of complex types by allowing you to define a more readable alias for them. This is particularly useful for improving code readability and maintainability.

### Version Requirements

* C# Version: This feature is available starting from C# 10.
* .NET Version: You need to be using a version of .NET that supports C# 10, which is .NET 6.0+

### Syntax

The syntax for defining a type alias using the using directive is straightforward:

using AliasName = FullTypeName;

### Aspects of the Custom Type Alias

### Purpose and Reasoning

* **Improving Readability**:  
  Complex type signatures can make code difficult to read and understand.   
  Using an alias makes the code more concise and easier to read.
* **Maintaining Maintainability**:  
  There’s an application of dependency inversion wherein the definition of a type is stored in one place, changeable for all by a single edit, rather than explicitly stated for every use.
* **Reducing Redundancy**:  
  Helps in reducing redundancy when a complex type is used multiple times in the codebase.

### Upsides

* **Code Simplification**:  
  Makes complex type signatures simpler
* **Centralized Type Management**:  
  Changing the underlying type becomes easier and less error-prone
* **Enhanced Readability**:  
  Improves code readability by replacing long type names with shorter, more descriptive ones

### Downsides

* **Indirect Type Information**:   
  When reading the code, you need to refer back to the alias definition to understand the underlying type, which can sometimes hinder quick comprehension.
* **Limited Scope**:   
  Aliases defined using the using directive are limited to the scope of the namespace or file in which they are declared. This can sometimes be restrictive.

### Utilities and Roles

* **Simplifying Generics**:   
  Especially useful when dealing with complex generic types
* **Consistent Naming**:   
  Helps in maintaining consistent naming conventions for commonly used complex types
* **Documentation**:   
  Acts as a form of documentation by providing meaningful names for complex types

### Conclusion

Using the using directive to define type aliases in C# 10 is a powerful tool for improving code readability and maintainability, especially when dealing with complex types. While it has some limitations, its benefits in terms of simplifying code and centralizing type management make it a valuable feature for modern C# development.