**Attributes -** Metadata to an object that can be customized

*NOTE: Attribute name during declaration must have a “Attribute” suffix, which is omitted when using*

* We can also add AttributeTarget to define usage of an attribute
  + What the attribute can be applied to (class/method/all…)
  + Whether the attribute can be inherited by classes that are **child of applying class**
* Declare constructor

**



A screen shot of a computer code

Description automatically generated with low confidence

* You can also declare properties of the Attribute (like a normal class)
* Apply Attribute:

A picture containing text, font, screenshot

Description automatically generated

Retrieving information stored in attribute

1. **A screen shot of a computer program

   Description automatically generated with medium confidenceRetrieving a Single Instance of an Attribute**

Where “t” is typeof(class)

1. **Retrieving Multiple Instances of an Attribute Applied to the Same Scope (only 1 member)**

TestAttribute[] attributes = typeof(TestAttribute[])Attributes.GetCustomAttribute**s**(typeof(member), typeof(TestAttribute))

* Returns an array of all custom attributes applied to this member

1. **Retrieving Multiple Instances of an Attribute Applied to Different Scopes**

If you have a class with the same attribute applied to every member and you want to retrieve the values in all the attributes applied to those members, you must supply every method or member individually to **GetCustomAttributes** and **GetCustomAttribute**.

A screen shot of a computer program

Description automatically generated with low confidence

1. **Find the specific attributeType of a member**

object[] MemberInfo.GetCustomAttributes(Type attributeType, bool inherit)

* Returns an array of all custom attributes applied to this member and identified by Type
* ***attributeType***: the type of attribute to search for. Only attributes that are assignable to this type are returned
* ***Inherit:*** true to search this member’s inheritance chain to find attributes; otherwise false.

**Metadata and Self-Describing Components**

When compiling .NET, a **PE file** is created. Metadata is inserted into one portion of the file, and your code is converted to Microsoft intermediate language (MSIL) and inserted into another portion of the file.

Every type and member that is defined and referenced in a module or assembly is described within metadata. When code is executed, the runtime loads metadata into memory and references it to discover information about your code's classes, members, inheritance, and so on.

Metadata describes every type and member defined in your code in a language-neutral manner. Metadata stores the following information:

* Description of the assembly.
  + Identity (name, version, culture, public key).
  + The types that are exported.
  + Other assemblies that this assembly depends on.
  + Security permissions needed to run.
* Description of types.
  + Name, visibility, base class, and interfaces implemented.
  + Members (methods, fields, properties, events, nested types).
* Attributes.
  + Additional descriptive elements that modify types and members.

PE file:

| **PE section** | **Contents of PE section** |
| --- | --- |
| PE header | The index of the PE file's main sections and the address of the entry point.  The runtime uses this information to identify the file as a PE file and to determine where execution starts when loading the program into memory. |
| MSIL instructions | The Microsoft intermediate language instructions (MSIL) that make up your code. Many MSIL instructions are accompanied by metadata tokens. |
| Metadata | Metadata tables and heaps. The runtime uses this section to record information about every type and member in your code. This section also includes custom attributes and security information. |

More details on metadata: <https://learn.microsoft.com/en-us/dotnet/standard/metadata-and-self-describing-components>