**Variables & Data Types**

**Identifier rule**:

* Local Variables: Camel Case (numberCount) 🡪 private data
* Constants: Pascal Case (MaxZoom) 🡪 everything else

Table

Description automatically generated

Real Numbers:

float number = 1.2f;

decimal number = 1.2m;

**Is, As Keyword**

The **is keyword** returns true if the conversion is compatible, else returns false.

Boolean result = object\_type\_to **is** object\_type\_From;

The **as keyword**

1. **object** obj = **new** **object**();
2. Console.WriteLine(obj);
4. // This will raise a runtime InvalidCastException as "Unable to cast object of type 'System.Object' to type 'Tutpoint'"
5. Tutpoint tutpoint = (Tutpoint)obj;

Therefore, we can use the “as” keyword to provide a check for compatible typecasting. In case of compatible, it will return the value of the new object type otherwise, null will be returned.

1. // Creating object of object type
2. **object** obj = **new** **object**();
3. Console.WriteLine(obj);
5. // 'as' keyword used to convert 'obj' of type 'object' to 'Tutpoint' type
6. // It will return null as the conversion of object type to Tutpoint is incompatible
7. Tutpoint tutpoint = obj **as** Tutpoint;
8. **if** (tutpoint != **null**) …
9. **else** …

Note that obj is not modified (still have type **object**)

**Object, Var, Dynamic**

* + The **object** –keyword represents the System.Object type, which is the root type in the C# class hierarchy. This keyword is often used when there’s no way to identify the object type at compile time. You need to use explicit casts to convert a variable declared as object to a specific type

(Passing into functions)

object objExample = 10;

Console.WriteLine(objExample.GetType()); // Prints System.Int32

objExample = (int)objExample + 10; // Requires explicit cast

objExample = "test"; // Can assign different type at runtime (since static type is System.Object

* + The **var** keyword, is used for implicitly typed local variables and for anonymous types. When a variable is declared by using the var keyword, the variable’s type is inferred from the initialization string at compile time. The type of the variable can’t be changed at run time. If the compiler can’t infer the type, it produces a compilation error

(Like auto in C++, automatically realize variable type)

var varExample = 10 // varExample’s static typed is System.Int32

Console.WriteLine(objExample.GetType()); // Prints System.Int32

varExample += 10

~~varExample = “test”~~

* + The **dynamic** keyword makes certain scenarios that traditionally relied on the object keyword easier to write and maintain. In fact, the dynamic type uses the System.Object type under the hood, but *unlike object it doesn’t require explicit cast operations at compile time, because it identifies the type at run time only*

dynamic dynamicExample = 10;

Console.WriteLine(dynamicExample.GetType()); // Prints System.Int32

dynamicExample = dynamicExample + 10; // Didn’t require explicit cast

dynamicExample = "test"; // Can assign different type at runtime (since static type is System.Object

**Parameter Modifiers** (passing into functions)

* + 1. The **out** keyword pass by reference the arguments (variable must be assigned before return)
  + Classic workaround for returning multiple values from method (tuples)
  + A screen shot of a computer program

    Description automatically generated with low confidence
    1. The **ref** keyword pass by reference the arguments, require the variable to be initialized
    2. The **in** keyword pass by reference the arguments, does not allow the method to modify the argument

Properties are not variables so cannot be passed as modified parameters