# Type Conversion

* C# is statically typed at compile time. Meaning after a variable is declared. It cannot be declared again.
* However, it is possible to change the variable type and that is called Type Conversion

## Implicit Conversion

* When you convert the type without any data loss
* No special syntax is needed for this conversion to happen
* Ex: Converting an int type into a double type
  + Double types can hold more values and will not result in data loss when converted
* As for reference types, you can convert a derived class to its base class without any problem.
  + It is because a derived class will have all the members of the base class

## Explicit Conversion

* If there is a risk of losing information, you must perform a **Cast.**
* Ex: going from double type into a int type
  + You might experience data loss.
* An explicit cast is also required when you convert a base class into a derived class.
  + This type of conversion can be spotty in that it will compile correctly but might fail at run time.
  + One case this might happen if you try to convert a derived class to another derive class from the same base class.

## Type Checking

* Is operator
  + Checks if the runtime type of an expression results is compatible with a given type
  + Essentially, if you give an object that is the same type or it can be converted to that type it will return true.
* TypeOf operator
  + Used to check the type at compile-time
  + The operator gets the namespace for a type

# Introduction to Exceptions

* An exception is an event that occurs during the execution of a program that distrust the normal flow of instructions.
* Good when trying to find bugs but bad when deploying your application for everyone to use.
* They are not errors! An error is a serious problem that for the most part cannot be handled by the developer
  + They are fatal to the program at runtime
  + Ex: A stack overflow error and it occurs when your hardware run out of memory to store information (May or may not be solvable by a developer)
* There are 3 types of error
  + Usage Error – Error in your program logic and can be solve by modifying/restructuring your code.
  + Program Error – run-time error that cannot be avoid even with a bug-free code (Ex: Your SDK is corrupt and can’t compile or translate it to machine code).
  + System Failures – run-time error that cannot be handled programmatically in a meaningful way (ex: Your ram hardware is faulty and storing memory becomes corrupt).

## Exception Handling

* Using a try-catch block.
* If you know the block of code you will run may have the risk of throwing an exception, you should put a try block.
* The catch block will then “catch” that error and will run its block of code.
* Optionally, you can put a finally block that will run regardless if your code throws an exception or not
  + Mostly used to clean up resources you used in the try block.

## Throwing Exception

* This allows us to customize the exception that your code will throw.
* Useful for enforcing rules/logic in your system

## Exception Hierarchy

* All exception will inherit from the Exception class
* All run-time exceptions will inherit from the SystemException class

# Introduction to Serialization

* The process of converting an object into a stream of bytes (001110011) or a JSON or XML for storage or transfer.
  + JSON – JavaScript Object Notation and it is an open standard file format and data interchange format
    - Easy for us to read what is exactly stored
  + XML – is just Extensible Markup Language that defines a set of rules
    - Machines can also read it easily
    - We can somewhat read it
* The reverse process is called deserialization

## JSON Serialization

* Popular choice in .Net and it is provided by using System.Text.Json namespace
* All public properties are serialized, and you can specify which ones to exclude
* Fields are also excluded in JSON (not supported)
* It can either convert the object into a string, byte array, or some sort of standardization

# S.O.L.I.D. Principles

* They are five design principles intended to make software design more understandable, flexible, maintainable.
  + Kinda like OOP pillars, but it is just rules to follow for us to write better code

## Single Responsibility Principle

* A class should have one and only one reason to change.
* If one class has more than one responsibility just segregate them into many classes.
* Ex: Software Engineer class, it shouldn’t have the responsibility to manage the financial forms of the company. Instead, you should create another class called Accountant class that should have that responsibility only.

## Open/Close Principle

* A class should be open for extension but closed for modifications.
* It just means you can add new functionality without changing the existing code
* A great way to do this is using interfaces

## Liskov Substitution Principle

* Derived class should be substitutable for their base implementation.
  + When you replace your base class into a derived class, it should not break your code.
* One way to avoid breaking this principle, is inside the derived class method, you should call the parents method.
* I.E. your derived class method/logic should call upon the base class method/logic to prevent any problems.

## Interface Segregation Principle

* You should not be forced to implement methods that you don’t need in an interface
* You should just segregate the interface into multiple interfaces

## Dependency Inversion Principle

* High- and low-level modules should depend on abstractions but not on each other
* If a class uses the design and implementation of another class, it raises the risk that changing one class could break the other class
* To fix this, we must let both classes depend on abstractions (such as using interfaces).