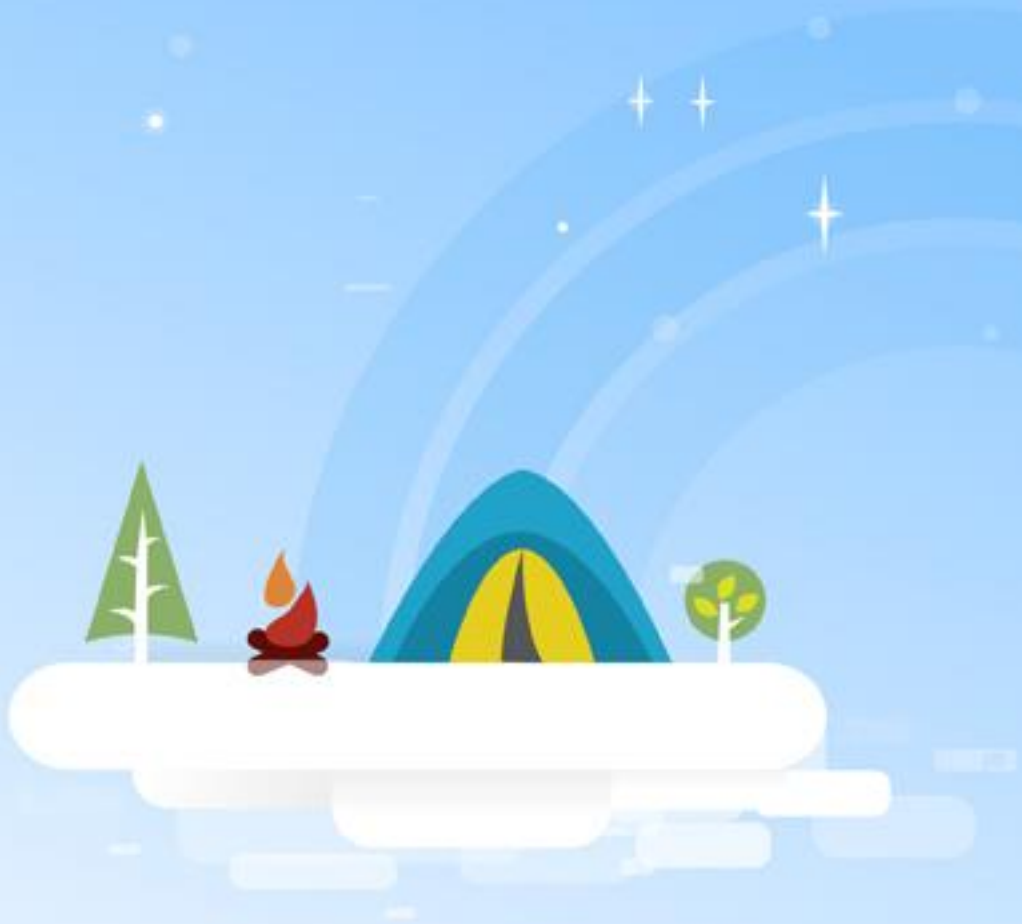
Establishing Relationship



Ridi Ferdiana | ridi@acm.org

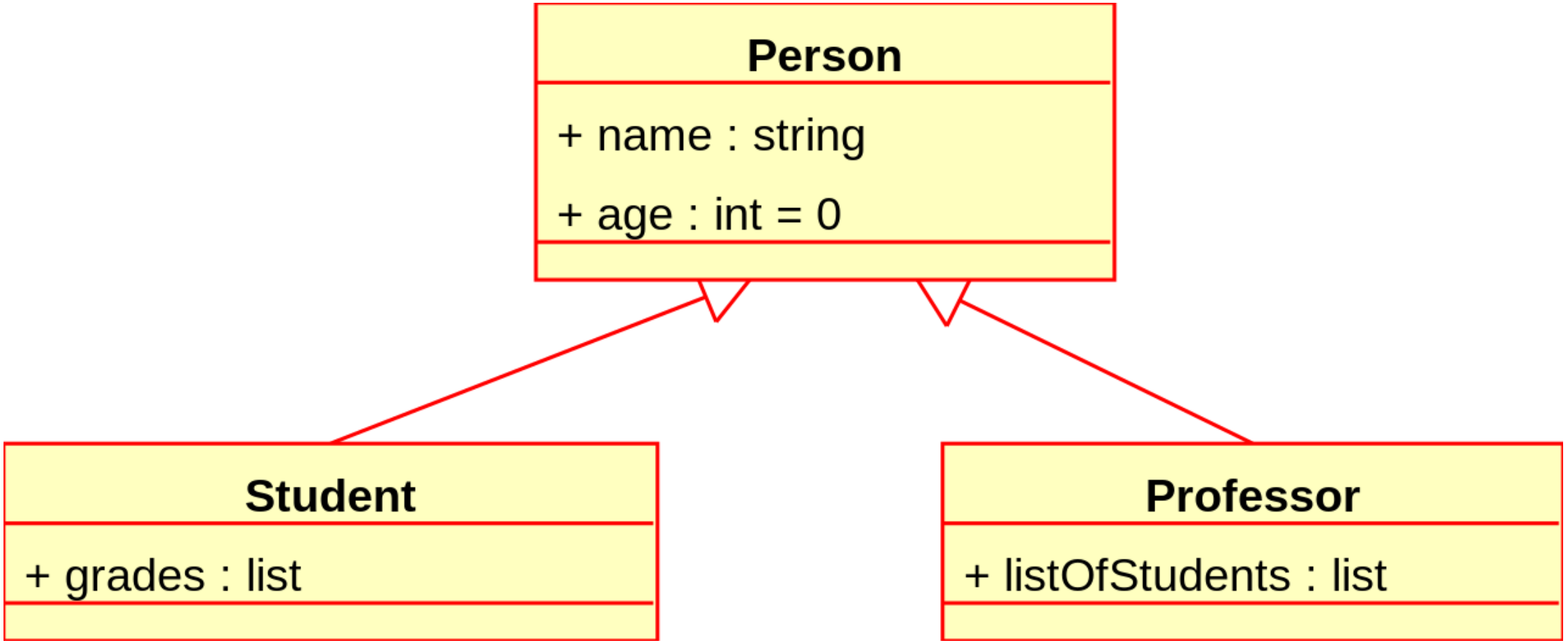
Version 1.1.0

Overview

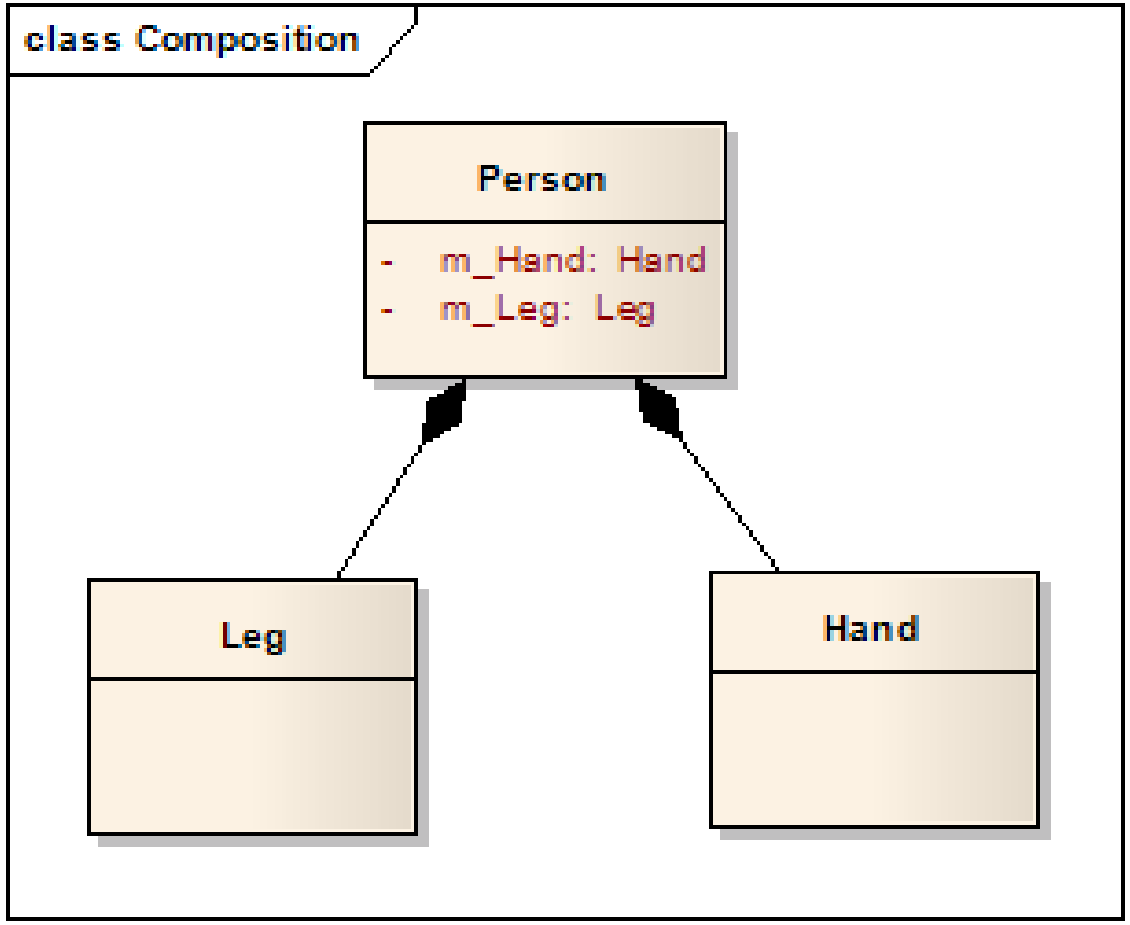
- Understanding Relationship in Object
- Relating the object with Collection
- Combining the object with Operator
- Accessing the object through Properties

Understand the object relationship

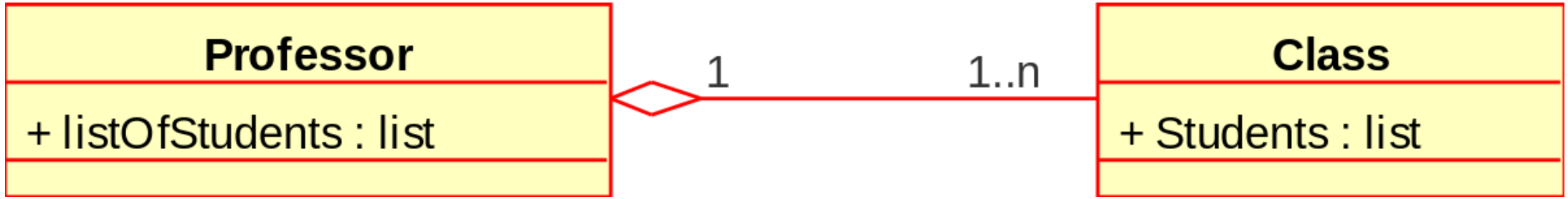
Generalization



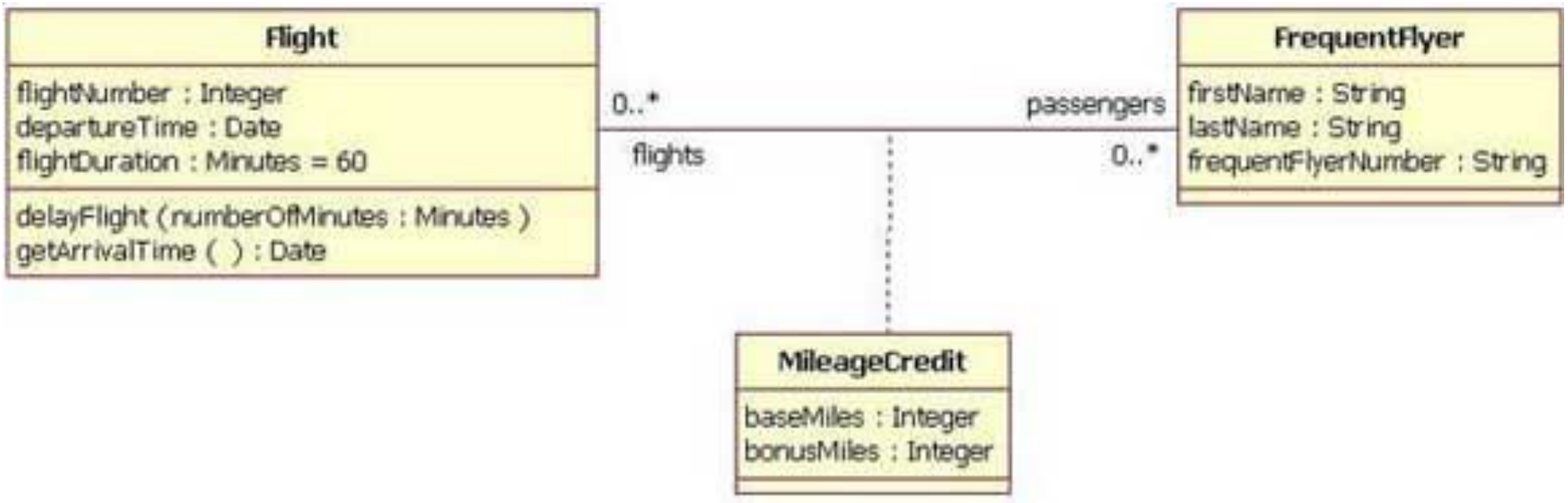
Composition



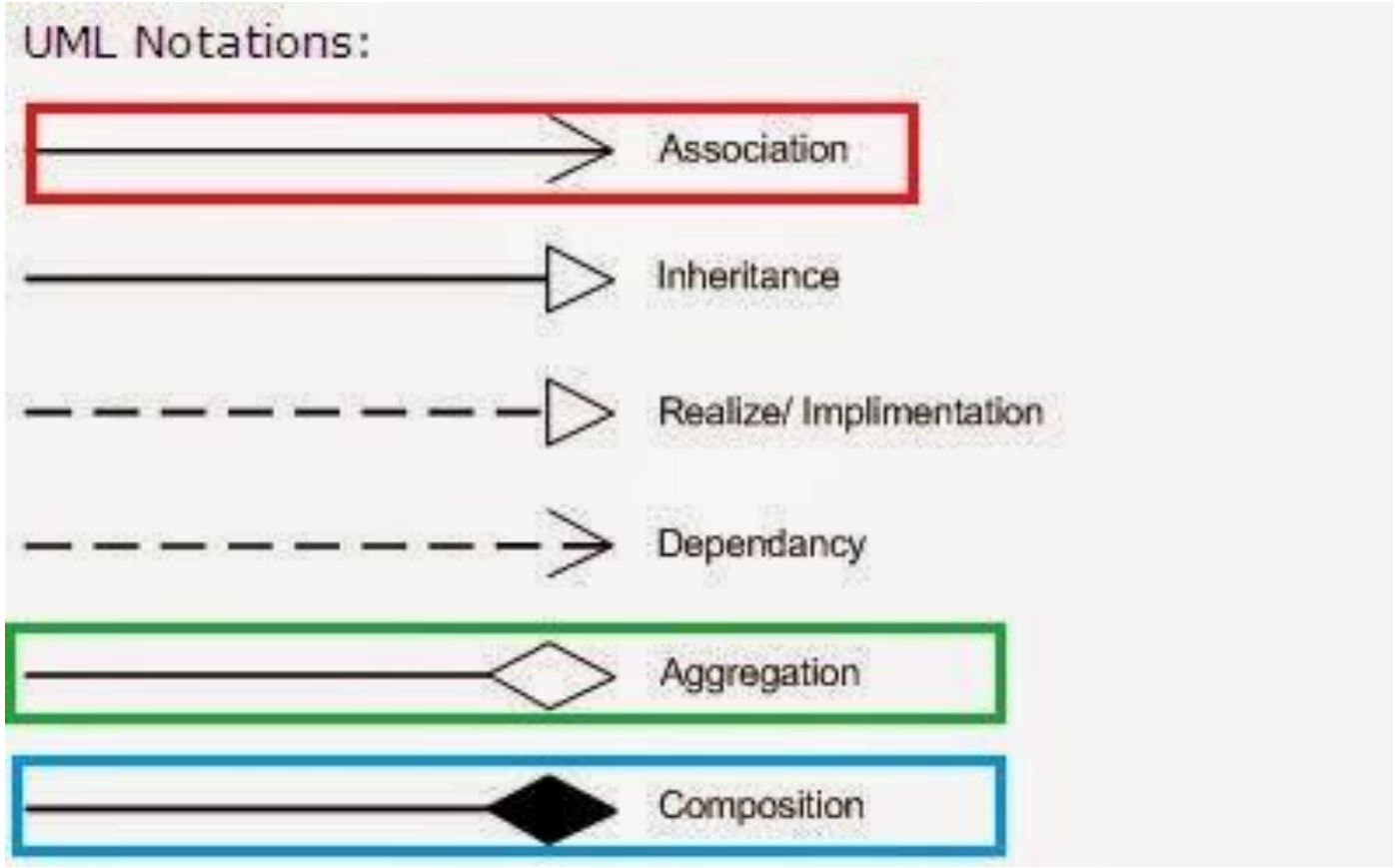
Aggregation



Association



Key Differentiation



Collection

What is Collections

A set or group of related items or records that can be referred to as a unit.

Can be acessed and manipulated easily

- Iteration
- Index

Store the item as an object

Generic and Non Generic Collections

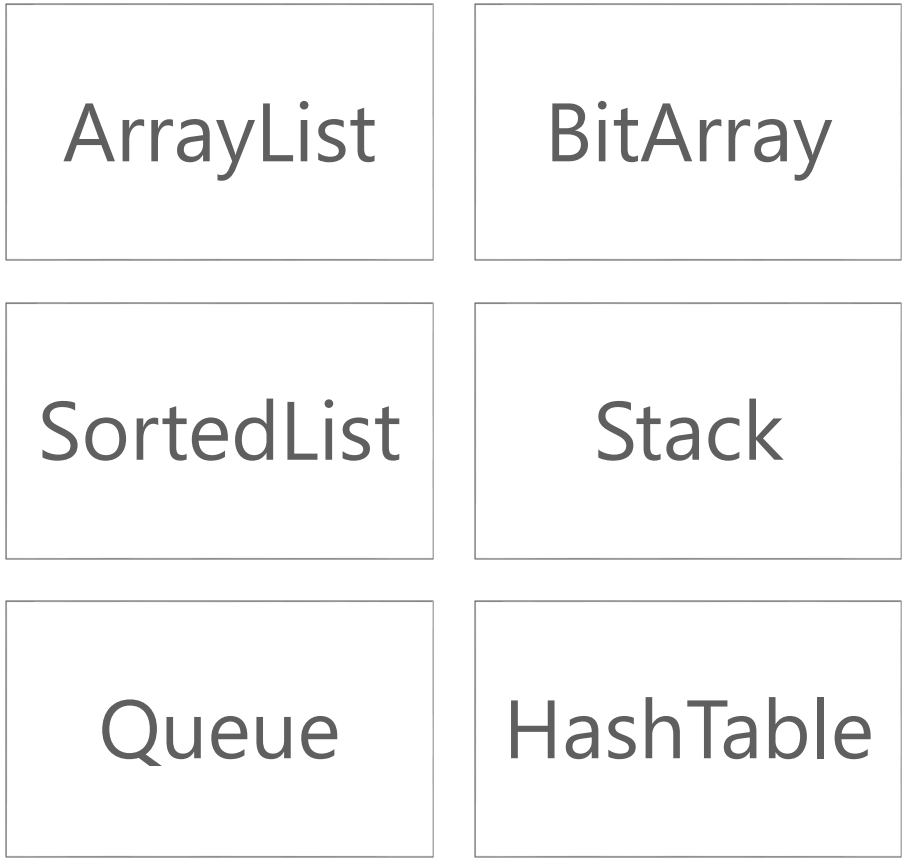
Non Generic

- As an object
- Easy to use hardly to convert
- Available since .NET Framework 1.0

Generic

- As a type (string, int, etc)
- Easy to convert and strict to use
- Available since .NET Framework 2.0

Non Generic Collections

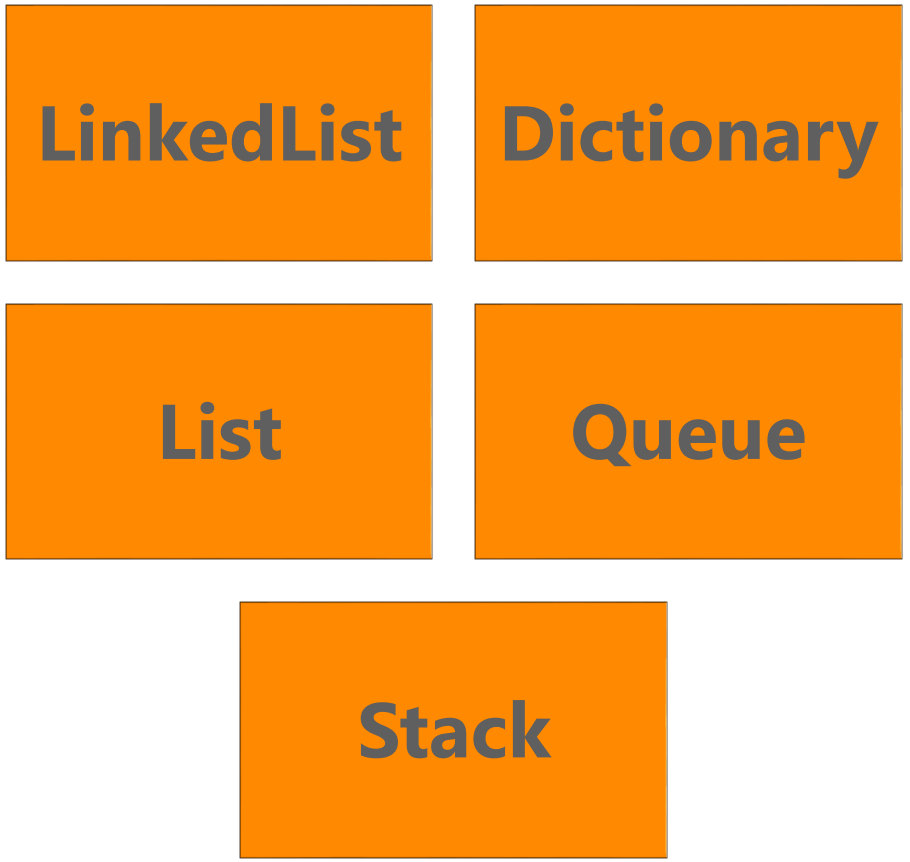


Non Generic Collections

Demo

Introducing List

Generic Collections



Generic Collections

Demo

Introducing List<t>

Using Collections

Requirements	Collection
Do you need a sequential list where the element is typically discarded after its value is retrieved	Queue, Stacks
Do you need to access each element by index	ArrayList, StringCollection, List, Hastable, etc
Do you need to sort the List	Hastable, SortedList, ArryList
Do you need fast searches and retrieval of information?	ListDictionary (small), HashTable
Do you need collections that accept only strings	StringCollection

Operator

Operators Background

- Using methods
  - Reduces clarity
  - Increases risk of errors, syntactic and semantic

```
myIntVar1 = Int.Add(myIntVar2,
            Int.Add(Int.Add(myIntVar3,
                            myIntVar4), 33));
```

- Using operators
  - Makes expressions clear

```
myIntVar1 = myIntVar2 + myIntVar3 + myIntVar4 + 33;
```

Predefined C# Operators

Operator Categories	
Arithmetic	Member access
Logical (Boolean and bitwise)	Indexing
String concatenation	Cast
Increment and decrement	Conditional
Shift	Delegate concatenation and removal
Relational	Object creation
Assignment	Type information
Overflow exception control	Indirection and address

Operator Overloading

- Operator overloading
  - Define your own operators only when appropriate
- Operator syntax
  - Operator **op**, where **op** is the operator being overloaded
- Example

```
public static Time operator+(Time t1, Time t2)
{
      int newHours = t1.hours + t2.hours;
      int newMinutes = t1.minutes + t2.minutes;
      return new Time(newHours, newMinutes);
}
```

Overloading Relational Operators

Relational operators must be paired

- < and >
- <= and >=
- == and !=

Override the Equals method if overloading == and !=

Override the GetHashCode method if overriding equals method

Creating operator

Demo

Overriding Operator in C#

Properties

Why Use Properties?

Properties provide:

- A useful way to encapsulate information inside a class
- Concise syntax
- Flexibility

Is just like get and set Method

Using Accessors

Properties provide field-like access

- Use **get** accessor statements to provide read access
- Use **set** accessor statements to provide write access

```
class Button
{
    public string Caption // Property
    {
        get { return caption; }
        set { caption = value; }
    }
    private string caption; // Field
}
```

Property Types

- Read/write properties
  - Have both **get** and **set** accessors
- Read-only properties
  - Have **get** accessor only
  - Are not constants
- Write-only properties – very limited use
  - Have **set** accessor only
- Static properties
  - Apply to the class and can access only static data

Creating Properties

Demo

Object and properties

Review

- UML notation helps you to define and to design the class relationship.
- Collection helps relate similar or different object with similar way to access
- Operator helps to combine between object
- Properties helps to encapsulate and to access the variables