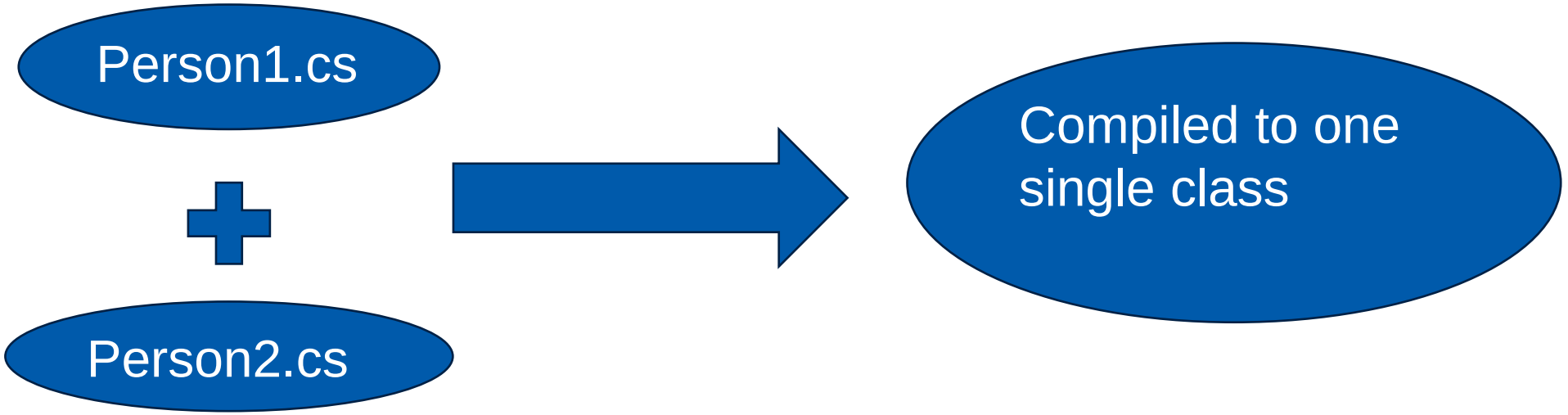
Programming Language - C#

Collections and Generics

Partial Classes

- Special ability to implement the functionality of a single class into multiple files
- All these files are combined into a single class file when the application is compiled.
- Created using the partial keyword



Advantages

- Multiple developers can work simultaneously with a single class in seperate files.
- Separate UI design code and business logic code
- Better maintenance of large classes by compacted them.
- Usage in LINQ to SQL or entity framework

Object class

Method	Description
GetType	Returns type of the object
Equals	Compares two object instances , returns true if they are equal otherwise false
RefereceEquals	Compares two object instances, returns true if both are same instances otherwise false
getToString	Converts an instance to a string type
GetHashCode	Returns hashcode for an object

Assignment : Partial classes

Assume your working for vehicle insurance company , your associates are working in different locations . How will you make them to develop one single class that can compute the insurance for two-wheeler s and four wheelers.

Static Variables & functions

- The keyword 'static' means that only one instance of a given variable exists for a class.

- Static variables can be initialized outside the member function or class definition.

- Static functions can access only static variables.

- Static functions exist even before the object is created.

- To manipulate and use the values of static variables, you can define a function as static function.

Assignment – Static

John a software developer in Zed Axis Technology needs to check how many times a function is called. For the same, he has been asked to create the function called “CountFunction”. Help John to create this function.

Indexers

- An indexer allows an instance of a class or struct to be indexed as an array.
- The class will behave like a virtual array.
- [] array access operator is used.
- Indexers are similar to properties the main difference is that the accessor will have parameter list unlike properties.

Contd…

```
[access modifier] [return type] this [arguement_}
{
get
{
//get block code
}
set{
//set block code
}
}
```

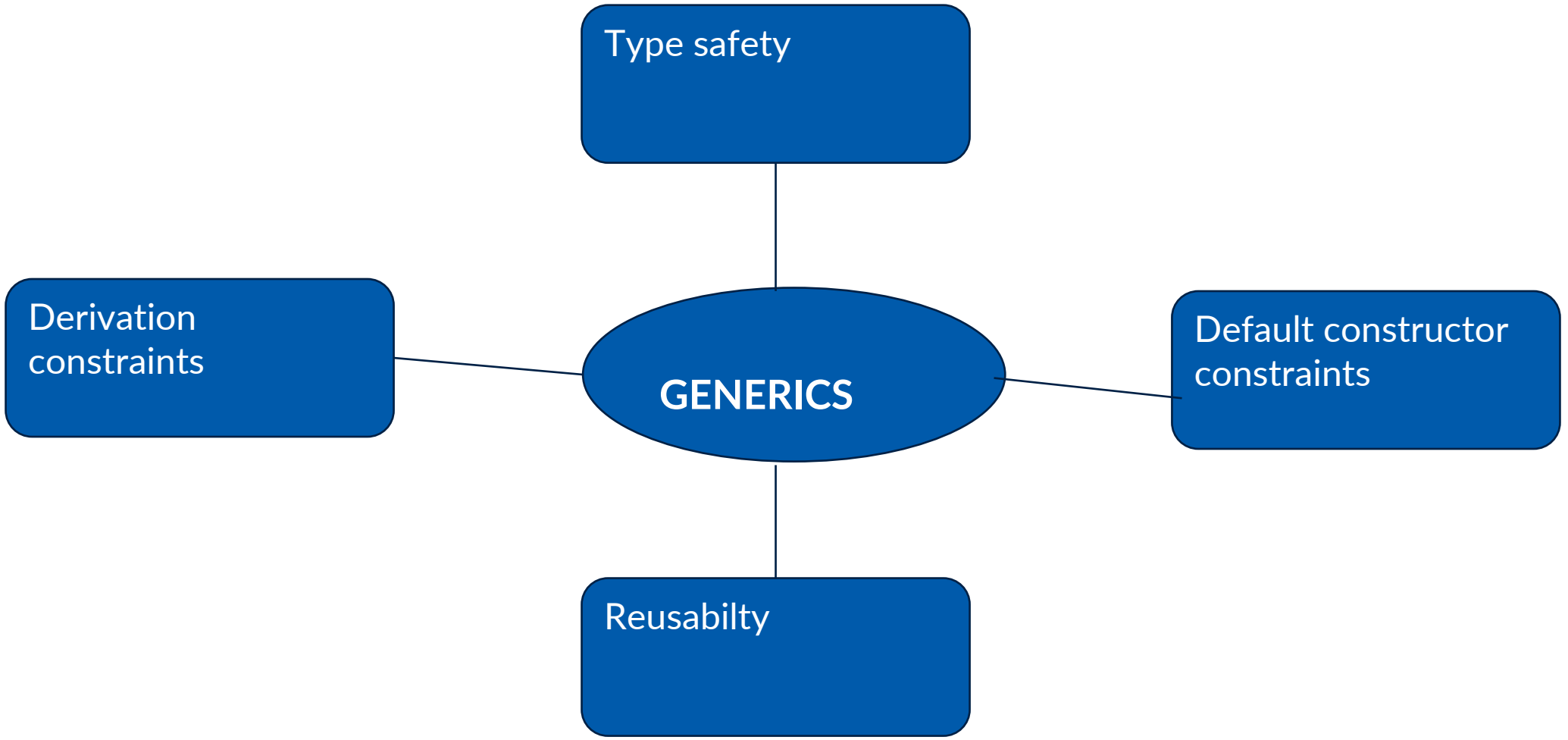

Indexers vs Properties

Indexers	Properties
Indexers are created with this keyword	Properties don't require this keyword
Indexers are identified by signature	Properties are identified by their names
Indexers are accessed using indexes	Properties are accessed but their names
Indexers are instance member so cant be static	Properties can be static as well as instance variables
A get accessor of an indexer has the same formal parameter list as the indexer	A get accessor of a property has no parameters
A set accessor of an indexer has the same formal parameter list as the indexer in addition to the value parameter	A set accessor of a property contains the implicit value parameter

Generics

- Generics introduce to the .NET framework the concept of type parameters
- It IS possible to design classes and methods that defer the specification of one or more types until the class or method is declared and instantiated by client class
- System.Collections.Generics is the namespace that provides the generic classes.
- It allows you to create type safe classes without compromising type safety ,performance, productivity and code bloat

Advantages of Generics



Features of Generics

- **Default values**
  - It is not possible to assign default values to generic type, you can use default keyword to assign
- **Constraints**
  - if u need to invoke some methods from the generic type you have to add constraints
- **Inheritance**
  - a generic class can implement generic interface
  - provide a type argument when deriving from generic base class

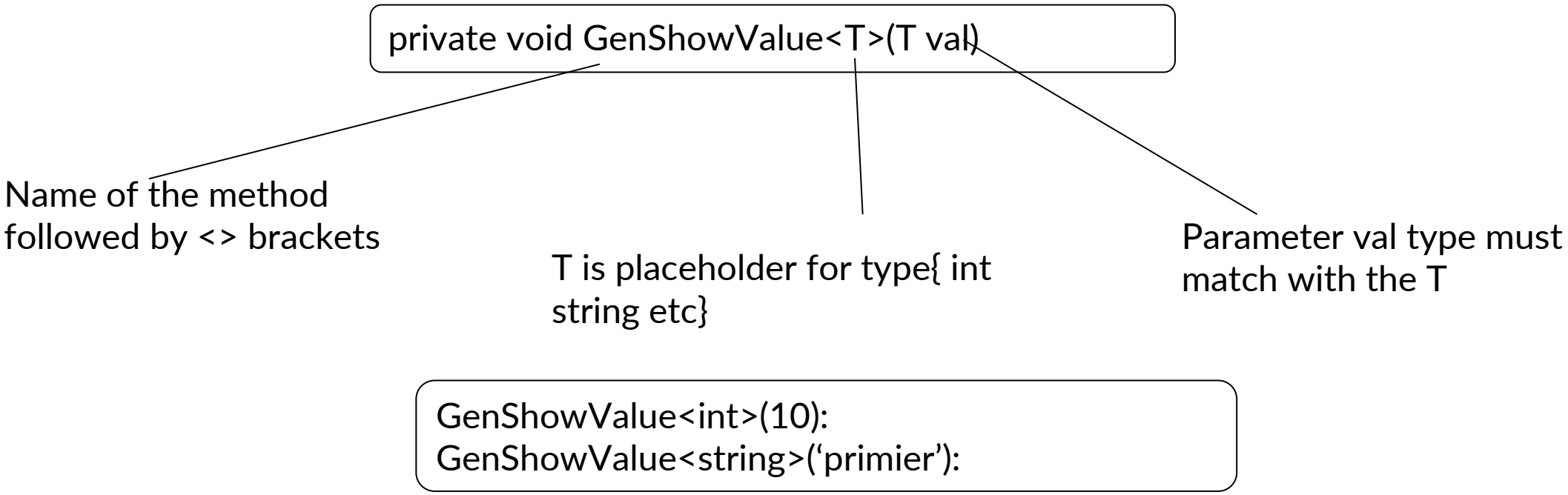
Generic Class

```
public class Printer<T>
{
        private T data;

        public T value
        {
        get{
        return this.data;
        }

        set{
        this.data=value;
        }}}
```

```
class TestPrinter
{
        static void Main(String args[])
        {

        Printer<string> name = new Printer<Printer>();
        name.value="welcome to hexaware";

        Printer<float> version =new Printer<float>():
        version.value=5.0F;

        C.W.L(name.value); C.W.L(version.value):
}
```

- Useful for encapsulating operations that are not specific to a particular datatype.
- Used in collection framework
- By creating a generic class, you can create a collection that is type-safe at compile-time.

Generic Methods



- Generic method constraints : where T

Generic Interfaces

```
interface IBook<T>
{
void add(T book);

void delete():

T get();

}
```

```
class Book<T>:IBook<T>
{
T book;
public void add(T book)
{
      this.book=book;
}
public void delete()
{
```

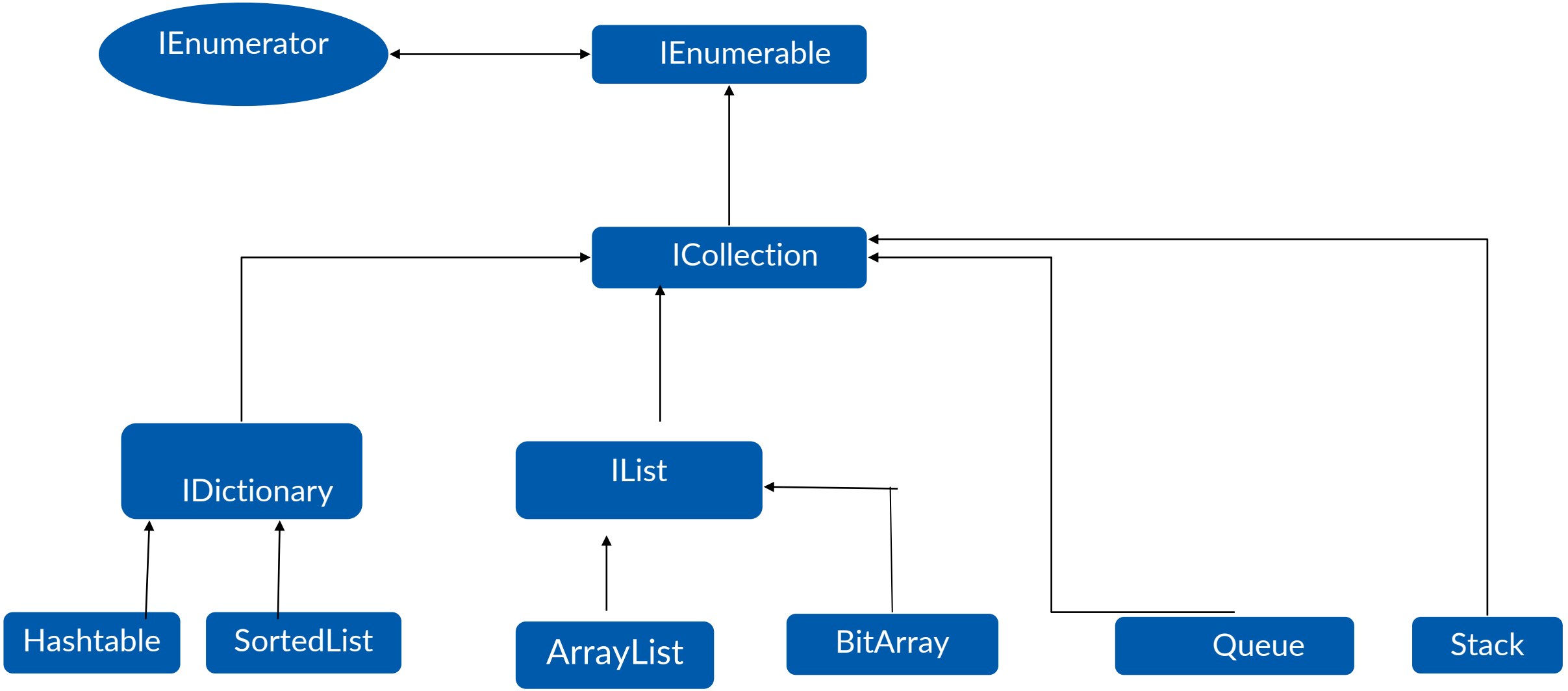
```
this.book=default(T);
}

public T get()
{
      return this.book;
}}
```

- It is often useful for generic collection classes or for the generic classes that represent items in the collection
- We use generic interfaces to avoid boxing and unboxing operations on value types.

Collections

- The .NET framework provides specialized classes for data storage and retrieval
- These classes provide support for stacks , queues lists and hash tables.
- Most collections classes implement the same interfaces, and these interfaces may be inherited to create new collection classes that fit more specialized data storage needs
- Collection classes are defined as part of **System. Collections** or **System.Collections.Generics**
- Using generic collection classes provides increased type safety and, in some cases, can provide better performances especially when storing value types.

Collection Hierarchy



Contd…

- There are some predefined classes in .NET class library which have implemented the concept of collection
    - ArrayList
    - List
    - HashTable
    - Dictionary
    - Queue
    - Stack
    - SortedList
- A custom collection class can be created by implementing Icollection interface

Limitation of Collection class

- **Arraylist** is a highly convenient collection class that can be used without modification to store any reference or value type.

```
System.Collections.ArrayList list1 = new System.Collections.ArrayList(); list1.Add(3);
list1.Add(105);

System.Collections.ArrayList list2 = new System.Collections.ArrayList(); list2.Add("It is
raining in Redmond.");
list2.Add("It is snowing in the mountains.");
```

- But this convenience comes at a cost. Any reference or value type that is added to an **Arraylist** is implicitly up cast to **Object**.

- If the items are value types, they must be boxed when they are added to the list, and unboxed when they are retrieved, which leads to performance decline.

Contd..

- The other limitation is lack of compile-time type checking; because an **Arraylist** casts everything to **Object**, there is no way at compile-time to prevent client code from doing something such as this:

```
System.Collections.ArrayList list = new
System.Collections.ArrayList();
// Add an integer to the list.
list.Add(3);
// Add a string to the list. This will compile, but may cause an error
later. list.Add("It is raining in Redmond.");
int t = 0;
// This causes an InvalidCastException to be returned.
foreach (int x in list)
{
t += x;
 }
```

Need for Generics

- What **Arraylist** and other similar classes really need is a way for client code to specify, on a per-instance basis, the data type that they intend to use.
- That would eliminate the need for the up cast to T:System.Object and would also make it possible for the compiler to do type checking.
- In other words, **Arraylist** needs a type parameter. That is exactly what generics provide.
- In the generic collection, in the N:System.Collections.Generic namespace, the same operation of adding items to the collection resembles this:

Contd…

```
// The .NET Framework 2.0 way to create a list
 List<int> list1 = new List<int>();
// No boxing, no casting:
list1.Add(3);
// Compile-time error:
// list1.Add("It is raining in Redmond.");
```


ArrayList <T>

Bank.cs

```
class Bank{
int bankid;
string bankname
ArrayList<BankAccount> accounts;
}
```

Program.cs

```
ArrayList<BankAccount> bankaccounts =new
ArrayList<BankAccount>();

BankAccount bobj=new
BankAccount(1,'DAVID',5000);
BankAccount bobj1=new
BankAccount(2,'Philip',6000);
BankAccount bobj2=new
BankAccount(3,'Max',7000);

bankaccounts.Add(bobj);
bankaccounts.Add(bobj1);
bankaccounts.Add(bobj2);

foreach(BankAccount b in bankaccounts)
{

//manipulate
}
```

Map – Dictionary<T,Q>

Bank.cs

```
class Bank{
int bankid;
string bankname
Dictionary<int,BankAccount>
accounts;
}
```

Program.cs

```
Dictionary<int,BankAccount> bankaccounts =new
Dictionary<int,BankAccount>();

BankAccount bobj=new BankAccount(1,'DAVID',5000);
BankAccount bobj1=new BankAccount(2,'Philip',6000);
BankAccount bobj2=new BankAccount(3,'Max',7000);

bankaccounts.Add(1,bobj);
bankaccounts.Add(2,bobj1);
bankaccounts.Add(3,bobj2);

foreach(KeyValuePain<int,BankAccount> b in bankaccounts)
{
C.W.L("{0} AND {1}",b.Key,b.Value);
}
```

Assignment - Collections

You are developing an application that includes a class called Order. The application will store a collection of Order objects. The collection must meet the following requirements

- Use strongly typed members
- Process order objects in first in first out order
- Store values for each order object
- Use zero based indices

IEnumerable

- There are two type of IEnumerable interface : generic and non generic
- The IEnumerable interface is in the System.Collections namespace and contains only a single method GetEnumerator()
- The IEnumerable interface is in the System.Collections.Generics namespace.

```
public interface IEnumerable
{
        IEnumerator GetEnumerator():
}
```

```
public interface
Ienumerable<T>:IEnumerable
{
IEnumerator<T> GetEnumerator():
}
```

IEnumerator

- IEnumerator retains its cursors current state unlike IEnumerable
- It defines two methods reset() and movenext()
- It is used to by the foreach statement to iterate through the collection

Thank you

Innovative Services

Passionate Employees

Delighted Customers