Generics

Generics introduces the concept of type parameter this allows to design classes and methods that defines the data type(s) that will be encapsulating, until the consumer creates a new instance of an object or calls the method in case.

```
// Declare the generic class.
public class GenericList<T>
{
    public void Add(T input) { }
}
class TestGenericList
{
    private class ExampleClass { }
    static void Main()
    {
        // Declare a list of type int.
        GenericList<int> list1 = new GenericList<int>();
        list1.Add(1);

        // Declare a list of type string.
        GenericList<string> list2 = new GenericList<string>();
        list2.Add("");

        // Declare a list of type ExampleClass.
        GenericList<ExampleClass> list3 = new GenericList<ExampleClass>();
        list3.Add(new ExampleClass());
    }
}
```

Defining the Generic class with the Type Parameter T

Using T as a method parameter

Instantiating the generic class, defining the data type of the type parameter

The type parameter can represent any data type

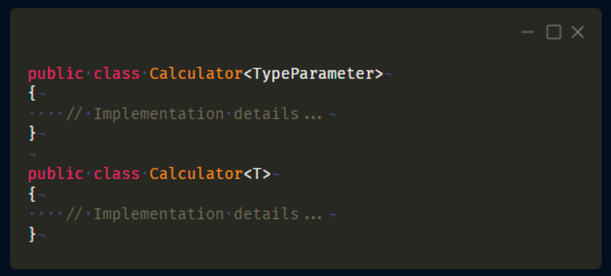
Generics

Generics

Generics allow you to create flexible and reusable code that can work with different types. They provide a way to write algorithms and data structures that are independent of specific data types.

The compiler performs type checking at compile time, preventing type-related errors.

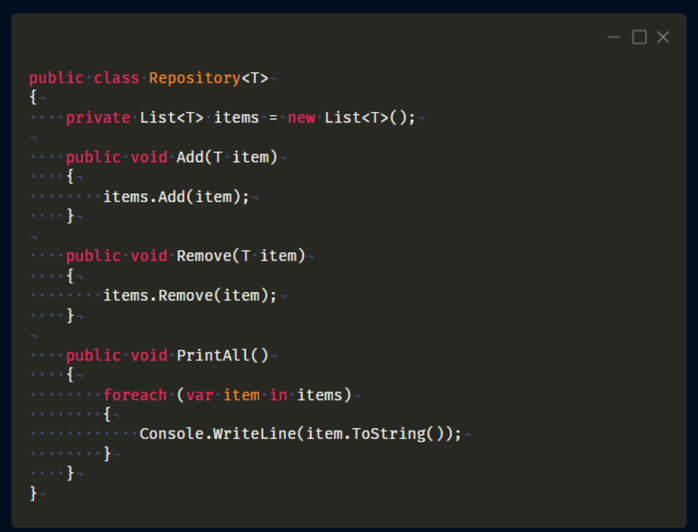
Performance improvements: Generics eliminate the need for boxing and unboxing operations, resulting in better performance.



```
public class Calculator<TypeParameter>
{
    // Implementation details ...
}

public class Calculator<T>
{
    // Implementation details ...
}
```

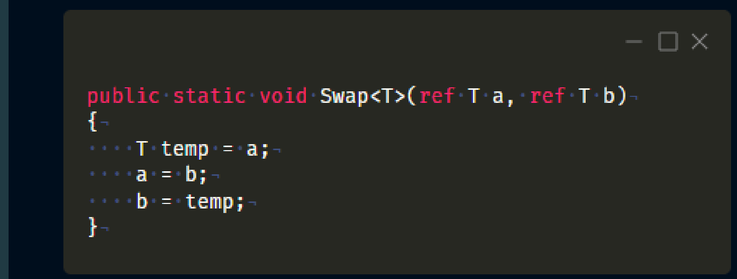
Type parameters

Placeholders for specific types that are provided when using a generic class or method.
Defined using angle brackets (<>) commonly represented by T.



```
public class Repository<T>
{
    private List<T> items = new List<T>();

    public void Add(T item)
    {
        items.Add(item);
    }

    public void Remove(T item)
    {
        items.Remove(item);
    }

    public void PrintAll()
    {
        foreach (var item in items)
        {
            Console.WriteLine(item.ToString());
        }
    }
}
```

Generic classes

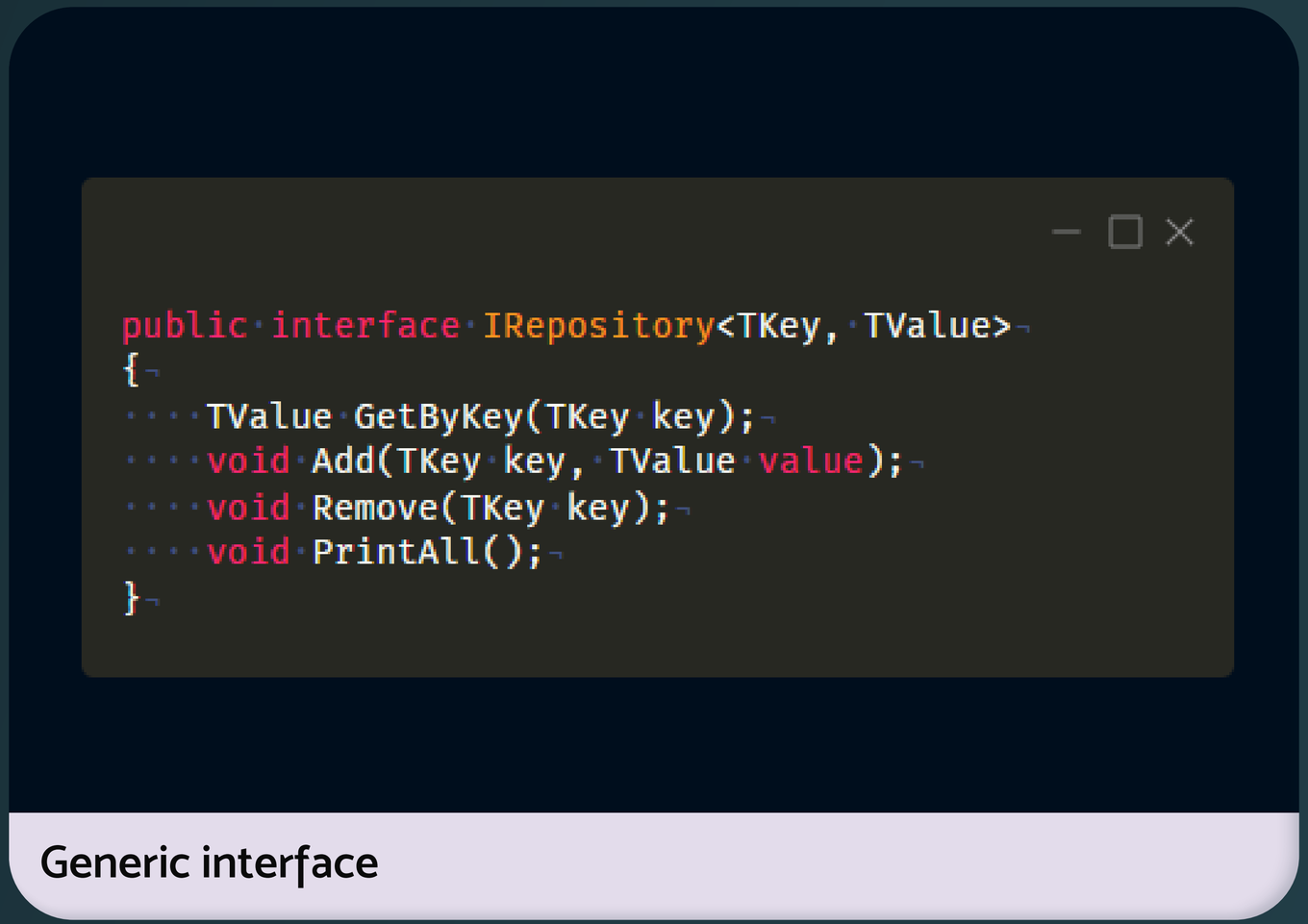


```
public static void Swap<T>(ref T a, ref T b)
{
    T temp = a;
    a = b;
    b = temp;
}
```

Generic methods

Generics interfaces

Generic interfaces are declared similar to non-generic interfaces, with type parameters specified. They allow you to define interfaces that can work with different types.



Generic interface

Built in generics interfaces

There are several common generic interfaces available in C# that provide useful functionality and are commonly used in various scenarios. Here are explanations of some of these interfaces:

```
IEnumerable<int> numbers = new List<int> { 1, 2, 3, 4, 5 };
foreach (int number in numbers)
{
    Console.WriteLine(number);
}
```

IEnumerable

Is the most basic interface that only allows you to iterate over a collection of objects.

```
ICollection<string> names = new List<string>();
names.Add("John");
names.Add("Jane");
names.Remove("John");
Console.WriteLine(names.Count);
```

ICollection

Extends IEnumerable and adds support for adding and removing items from a collection.

```
IList<double> grades = new List<double> { 90.5, 85.2, 92.7 };
grades.Insert(1, 88.9);
double secondGrade = grades[1];
Console.WriteLine(secondGrade);
```

IList

Extends ICollection and adds support for indexing into a collection. This means you can access items by their position in the collection.

Constraints

Constraints allow you to specify requirements for the types that can be used as type arguments.

They are specified using the “where” keyword followed by the type parameter and the constraint(s).

```
public interface IRepository<TKey, TValue> where TKey : IComparable<TKey>
{
    TValue GetByKey(TKey key);
    void Add(TKey key, TValue value);
    void Remove(TKey key);
    void PrintAll();
}
```

Multiple type parameters

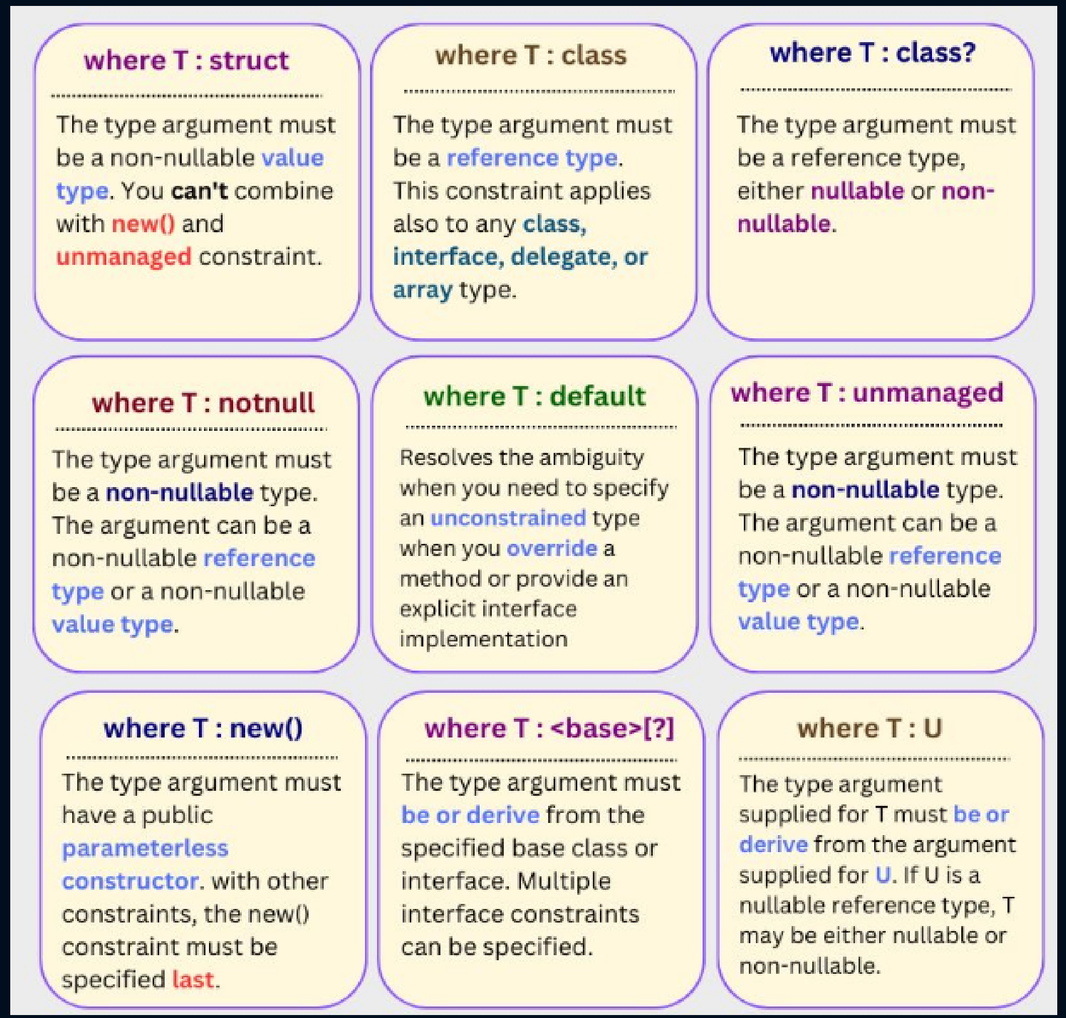
```
public interface IRepository<T> : IDisposable
{
    void Add(T newEntity);
}

public class SqlRepository<T> : IRepository<T> where T : class, IEntity
{
    DbContext _ctx;
    DbSet<T> _set;

    public SqlRepository(DbContext ctx)
    {
        _ctx = ctx;
        _set = _ctx.Set<T>();
    }

    public void Add(T newEntity)
    {
        if(newEntity.IsValid())
        {
            _set.Add(newEntity);
        }
    }
}
```

Generic with constraint implementation

Constraints and its limitations

When using constraints with generic types in C#, there are some limitations to be aware of:

- Limited constraints options (class, struct, etc)
- Cannot use sealed classes as constraints
- Constraints are not inherited. Each derived class must explicitly specify its own
- Limited flexibility with type parameters. For example using the new() constraint restricts the creation of a new instance of T inside the class.
- Constraints can lead to more complex code. Applying overly struct constraints can make the code less flexible



Constraints

Demo

We will check the following:

Built in generics like IEnumerable, IList

Create a generic class and interface

Add Constraints

You can check the following Repository for some examples:

C# fundamentals

Task

Exercise

Generic Data Storage System

Objective: Create a generic data storage system that can store and retrieve different types of data using C# generics, the only constraint is that these types of data need to have a default constructor

Features:

1. Generic Data Storage Class:
   - Define a generic class called DataStorage<T> that can store items of any type T, consider the constraint indicated before. This class should internally use a collection (e.g., a List<T>).
2. Methods:
   - Implement methods to Add, Remove, and Retrieve items from the data storage.
   - Implement a method to display all items in the storage.
3. Console Application:
   - Create a console application that uses the DataStorage<T> class to store and retrieve different types of data.

```
DataStorage<Person> personStorage = new DataStorage<Person>();
personStorage.AddItem(new Person { Name = "John", Age = 30 });
personStorage.AddItem(new Person { Name = "Jane", Age = 25 });

Console.WriteLine("People in storage:");
personStorage.DisplayItems();
```