C# : Generics

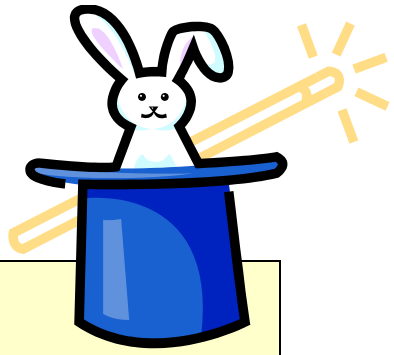
<T> is for Type

Overview

- What are generics?
- Generic classes
- Generic constraints
- Generic methods

Why Generics?

```
ArrayList _rabbits = new ArrayList()
{
    new Rabbit { Name = "Fluffy" },
    new Rabbit { Name = "Duffy" },
};
```



```
Rabbit firstRabbit = (Rabbit)_rabbits[0];
Rabbit secondRabbit = _rabbits[1] as Rabbit;

if (_rabbits[2] is Rabbit)
{
    Rabbit thirdRabbit = _rabbits[2] as Rabbit;
}

_rabbits.Add("This will be an unpleasant suprise!");
```

Solution?

```
public class RabbitList : ArrayList
{
    public int Add(Rabbit newRabbit)
    {
        return base.Add(newRabbit);
    }

    public new Rabbit this[int index]
    {
        get { return base[index] as Rabbit; }
        set { base[index]
    }

    // ...
}
```

```
RabbitList _rabbits = new RabbitList()
{
    new Rabbit { Name = "Fluffy" },
    new Rabbit { Name = "Duffy" },
    new Rabbit { Name = "Muffy" }
};

Rabbit firstRabbit = _rabbits[0];
Rabbit secondRabbit = _rabbits[1];
```

Generics

- **Generics types allow code reuse with type safety**
  - Class defers specification of a type until instantiated by client
  - Internal algorithms remain the same, only the type changes

```
List<Rabbit> _rabbits = new List<Rabbit>
{
    new Rabbit { Name = "Fluffy" },
    new Rabbit { Name = "Duffy" },
    new Rabbit { Name = "Muffy" }
};

Rabbit firstRabbit = _rabbits[0];
Rabbit secondRabbit = _rabbits[1];

_rabbits.Add("This is an error!");
```

Generic Collections

- **System.Collections.Generic**
  - HashSet<T>
  - List<T>
  - Queue<T>
  - Stack<T>
  - Dictionary<TKey, TValue>
- **Benefits over System.Collections**
  - Type safety
  - Performance (no boxing for value types)

```
Dictionary<string, Rabbit> rabbits =
                new  Dictionary<string, Rabbit>();

rabbits.Add(rabbit.Name, rabbit);
```

Generic Type Parameters

- **Use the type parameter as a placeholder**
  - Client must specify the type parameter
- **Type parameter name commonly starts with T**

```
public class MagicHat<T>
{
    public void Add(T thing)
    {
        _things.Add(thing);
    }

    List<T> _things;
}
```

```
Rabbit rabbit = new Rabbit { Name = "Fluffy" };


MagicHat<Rabbit> _rabbitHat = new MagicHat<Rabbit>();
_rabbitHat.Add(rabbit);
```

Generic Constraints

- **One or more restrictions on the type parameter**
  - Force type to be a struct or class
  - For type to have a public default constructor
  - Force type to implement interface or derive from base class

```
public class MagicHat<TAnimal>
    where TAnimal: IAnimal
{
    public void FeedAll()
    {
        foreach (TAnimal animal in _animals)
        {
            animal.Feed();
        }
    }
    …
    List<TAnimal> _animals;
}
```

```
public interface IAnimal
{
    void Feed();
}
```

Generic Type Classifications

- **Unbound generic types**
  - Unbound type is the blueprint to create other types (List<>)
- **Open and closed generic types**
  - Open generic has type parameters (List<T>)
  - Closed generic has no type parameters (List<int>)
- **At runtime, code executes in a closed, constructed type**

```
public class AnimalCollection<TAnimal> : ICollection<TAnimal>
{
    // ...
}
```

```
public class RabbitCollection : ICollection<Rabbit>
{
    // ...
}
```

Generic Methods

- **A method requiring a type parameter**
  - Static or instance method
  - Can specify constraints
  - Type parameter part of the method signature

```
public IEnumerable<T> FindByType<T>() where T:class
{
    foreach(TAnimal animal in _animals)
    {
        if (animal is T)
        {
            yield return animal as T;
        }
    }
}
```

The default Keyword

- **Used to assign a default value**
  - Avoids conflict when you don't know if type is a reference or value type

```
public TAnimal FindByName(string name)
{
    foreach (TAnimal animal in _animals)
    {
        if (animal.Name == name)
        {
            return animal;
        }
    }

    return default(TAnimal);
}
```

Generic Interfaces

- **Many examples in the .NET framework**
  - IEnumerable<T>, IList<T>, IComparable<T>

```
public interface IAnimal : IComparable<IAnimal>
{
    string Name { get; }
    void Feed();
}
```

```
public class Rabbit : IAnimal
{
    ...

    public int CompareTo(IAnimal other)
    {
        return Name.CompareTo(other.Name);
    }
}
```

Generic Delegates

- Useful for defining events
- .NET includes Func<>, Action<>, and Predicate<>

```
public void Add(TAnimal newAnimal)
{
    _animals.Add(newAnimal);
    if (AnimalAdded != null)
    {
        AnimalAdded(this,
            new AnimalAddedEventArgs { Name = newAnimal.Name });
    }
}

public event EventHandler<AnimalAddedEventArgs> AnimalAdded;
```

Generics and Variance

- Generic collections are invariant in C# 3.0
- C# 4.0 introduces variance with generics

```
List<Rabbit> rabbits = new List<Rabbit>();

ProcessAnimals(rabbits);
```

```
void ProcessAnimals(IEnumerable<IAnimal> rabbits)
{
      ...
}
```

Summary

- Generics create type safe abstractions
  - Classes, structs, interfaces, methods, delegates
- Apply constraints as required
  - Allows for more specific algorithms