

Building Generic Code with Generics



Deborah Kurata

CONSULTANT | SPEAKER | AUTHOR

@deborahkurata | blogs.msmvps.com/deborahk/



Generics Are ...

Writing
code
without
specifying
data types

Yet
type-safe

A way to
make our
code
generic



Overview



Making the case for generics

Building a generic class

Using a generic class

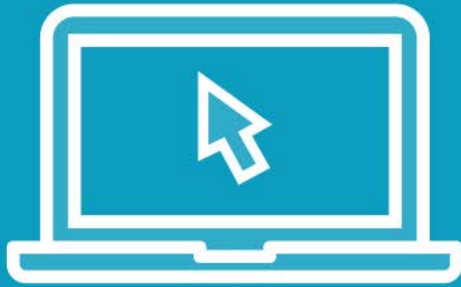
Defining generic methods

Leveraging generic constraints

FAQ



Demo



The case for generics



One Class for Each Data Type

```
public class OperationResult  
{  
}
```

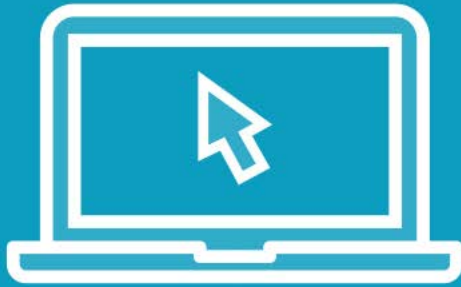
```
public class OperationResultDecimal  
{  
}
```

```
public class OperationResultInteger  
{  
}
```

```
public class OperationResultString  
{  
}
```



Demo



Building a generic class

```
public class OperationResult<T>  
{  
}
```





Generics ...

```
public class OperationResult<T>
{
    public OperationResult()
    {
    }

    public OperationResult(T result, string message) : this()
    {
        this.Result = result;
        this.Message = message;
    }

    public T Result { get; set; }
    public string Message { get; set; }
}
```



Multiple generic parameters

```
public class OperationResult<T, V>
{
    public OperationResult()
    {
    }

    public OperationResult(T result, V message) : this()
    {
        this.Result = result;
        this.Message = message;
    }

    public T Result { get; set; }
    public V Message { get; set; }
}
```



Generic Class Best Practices

Do:

Use generics to build reusable, type-neutral classes

Use T as the type parameter for classes with one type parameter

Prefix descriptive type parameter names with T

```
public class OpResult<TResult, TMessage>
```

Avoid:

Using generics when not needed

Using single-letter names when defining multiple type parameters

Use a descriptive name instead



Using a Generic Class

```
public class OperationResult<T>
{
    public OperationResult(){ }

    public OperationResult(T result, string message) : this()
    {
        this.Result = result;
        this.Message = message;
    }

    public T Result { get; set; }
    public string Message { get; set; }
}
```

```
var operationResult = new OperationResult<bool>(success, orderText);
```

```
var operationResult = new OperationResult<decimal>(value, orderText);
```



Defining Generic Methods

```
public int RetrieveValue(string sql, int defaultValue)
```

```
public T RetrieveValue(string sql, T defaultValue)
```

```
public class VendorRepository<T>
```

```
public T RetrieveValue<T>(string sql, T defaultValue)
```



Generic Method Best Practices

Do:

Use generics to build reusable, type-neutral methods

Use T as the type parameter for methods with one type parameter

Prefix descriptive type parameter names with T

```
public TReturn RetValue<TReturn, TParameter>  
    (string sql, TParameter sqlParameter)
```

Define the type parameter(s) on the method signature

Avoid:

Using generics when not needed

Using single-letter names when defining multiple type parameters

Use a descriptive name instead



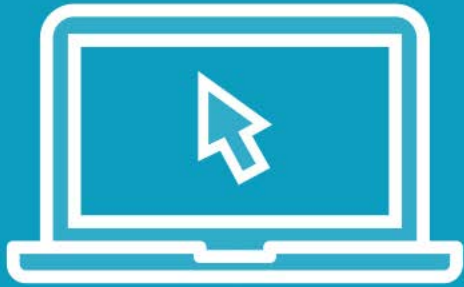
Generic Method

```
public T RetrieveValue<T>(string sql, T defaultValue)
{
    // Call the database to retrieve the value
    // If no value is returned, return the default value
    T value = defaultValue;

    return value;
}
```



Demo



The Case for Generic Constraints



GENERIC CONSTRAINT

where **T** : **struct**

where **T** : **class**

where **T** : **new()**

where **T** : **Vendor**

where **T** : **IVendor**

CONSTRAINS **T** TO

- ◀ Value type
- ◀ Reference type
- ◀ Type with parameterless constructor
- ◀ Be or derive from Vendor
- ◀ Be or implement the IVendor interface



Generic Constraint Syntax

```
public class OperationResult<T> where T : struct
```

```
public T Populate<T>(string sql) where T : class, new()  
{  
    T instance = new T();  
    // Code here to populate an object  
    return instance;  
}
```

```
public T RetrieveValue<T, V>(string sql, V parameter)  
                                where T: struct  
                                where V: struct
```



Limits to Generic Constraints

```
public T RetrieveValue<T>(string sql, T defaultValue)
{
    // Call the database to retrieve the value
    // If no value is returned, return the default value
    T value = defaultValue;

    return value;
}
```

```
public string RetrieveValue(string sql, string defaultValue)
{
    // Call the database to retrieve the value
    // If no value is returned, return the default value
    string value = defaultValue;

    return value;
}
```



Frequently Asked Questions

- What are **generics**?
 - A technique for defining a data type using a variable.
- What are the **benefits** of generics?
 - With generics we can write generalized reusable code that is type-safe, yet works with any data type.



Frequently Asked Questions (cont)

- What is a **generic type parameter**?
 - A placeholder for the specific type
 - For example: `public class OperationResult<T>`
- Where is a **generic type parameter** defined?
 - As part of a class signature
 - Or as part of a method signature

```
public class OperationResult<T>
```

```
public T RetrieveValue<T>(string sql, T defaultValue)
```



Frequently Asked Questions (cont)

- In this example, how do you define the actual type for T?

- `public class OperationResult<T>`

- `var operationResult = new OperationResult<decimal>();`

- `var operationResult = new OperationResult<bool>();`

- What is the purpose of a **generic constraint**?
 - To limit the types accepted for a generic type parameter.



Summary



Making the case for generics

Building a generic class

Using a generic class

Defining generic methods

Leveraging generic constraints