# Generic Overview

## Generics Are ...

Writing code without specifying data types

Yet type-safe

- Everything has datatype

A way to make our code generic


Making the case for generics why?

Building a generic class

Using a generic class

Defining generic methods

Leveraging generic constraints

# Use Case Why ?

```
public class OperationResult
{
    public OperationResult()
    {
    }

    public OperationResult(bool success, string message) : this()
    {
        this.Success = success;
        this.Message = message;
    }

    public bool Success { get; set; }
    public string Message { get; set; }
}
```

```
public class OperationResultDecimal
{
    public OperationResultDecimal()
    {
    }

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

    public decimal Result { get; set; }
```

```
public class OperationResultInteger
{
}

public class OperationResultString
{
}
```

DRY PRINCIPLE : Do Not Repeat Yourself

# Generic Class (Building)

```
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; }
}

/// <summary>
/// Provides a decimal amount and message
/// useful as a method return type.
/// </summary>
public class OperationResultDecimal
{
    public OperationResultDecimal()
    {
    }

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

    public decimal Result { get; set; }
```

Any type: int, bool, string

## 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>
```

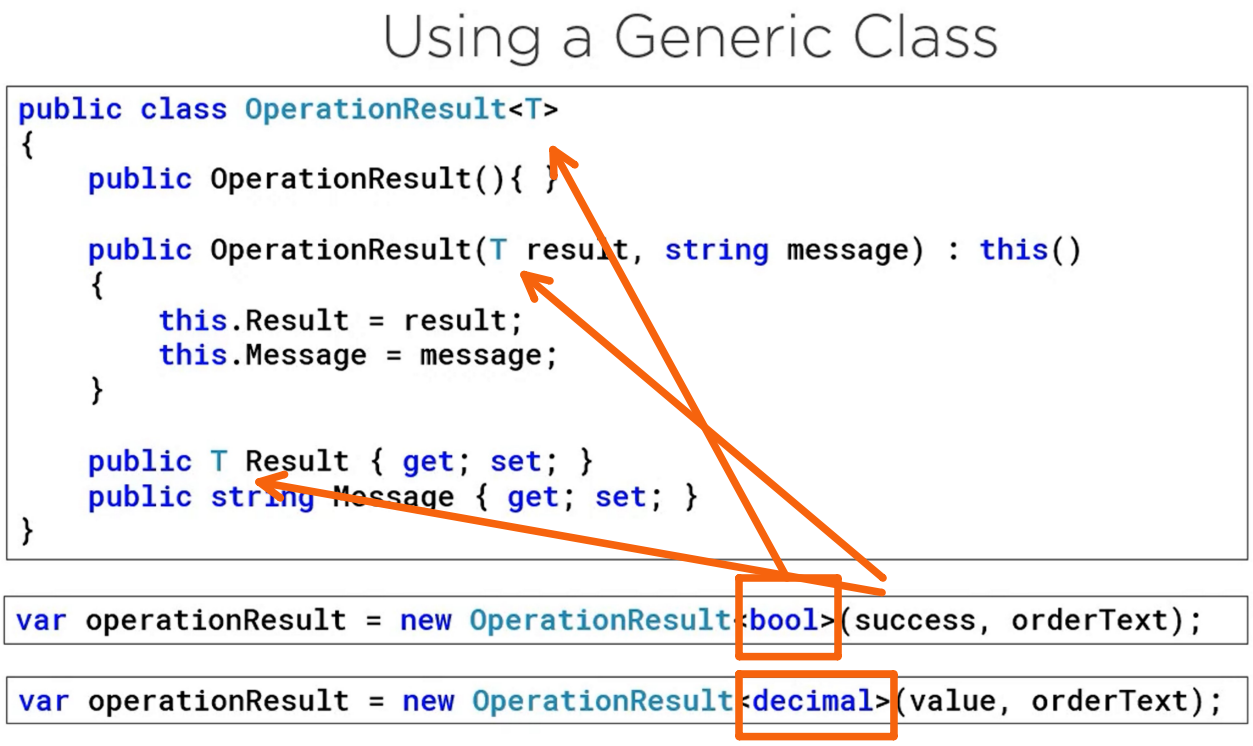
### Avoid:

Using generics when not needed

Using single-letter names when defining multiple type parameters

Use a descriptive name instead

# Generic Class (Using)



# Generic Methods

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

    return value;
}
```

```
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;
}
```

→ Can return ANY value type from DB.

```
.RetrieveValue<int>("Select ...", 42);

.RetrieveValue<string>("Select ...", "test");
```

## 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 Constraints

```
RetrieveValue<Vendor>("Select ...", vendor);
```

Can a SQL instance instance return a Vendor Object?

| GENERIC CONSTRAINT | CONSTRAINS T TO |
|---|---|
| `where T : struct` | Value type |
| `where T : class` | Reference type |
| `where T : new()` | Type with parameterless constructor |
| `where T : Vendor` | Be or derive from Vendor |
| `where T : IVendor` | 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
```