**Motivation for Generic Methods**

- Overloaded methods are often used to perform similar operations on different types of data.
- It is possible to replace multiple overloaded methods more concisely and elegantly using a just one generic method.
- For example, a single generic method could take as an argument an int array, a double array, and a char array respectively.

**Overloaded-Methods Implementation** (Page 1)

- Example:
  ```csharp
  public static void Main(string[] args)
  {
    ...
    int[] intArray = {1, 2, 3, 4, 5, 6};
    double[] doubleArray = {1.1, 2.2, 3.3};
    char[] charArray = {'b', 'i', 'l', 'i'};
    DisplayArray(intArray);
    DisplayArray(doubleArray);
    DisplayArray(charArray);
    ...
  }
  ```

**Overloaded-Methods Implementation** (Page 2)

- Example (con):
  ```csharp
  private static void DisplayArray(int myArray)
  {
    ...
  }
  ```
  ```csharp
  private static void DisplayArray(double myArray)
  {
    ...
  }
  ```
  ```csharp
  private static void DisplayArray(char myArray)
  {
    ...
  }
  ```

**Generic-METHOD Implementation** (Page 1)

- Format for a variable parameter:
  ```csharp
  accessModifier [static] [type] methodName<T>(T variableName)
  ```
  ```csharp
  { ...
  }
  ```

- Format for an array parameter:
  ```csharp
  accessModifier [static] [type] methodName<T>(T[] variableName)
  ```
  ```csharp
  { ...
  }
  ```

- All generic methods have a type-parameter list in chevrons (e.g. `<T>`)—follows the
method name in the declaration
+ Each type-parameter list contains one or more type parameters (for example T or T[]) that substitute for the specific type

6 [Generic-METHOD IMPLEMENTATION] (Page 2)
× It is recommended that type parameters be specified as individual capital letters
+ Typically a type parameter that represents the type of an element in an array (or
other collection) is named E for “element” or T for “type”
× If compiler cannot find a method declaration (generic or nongeneric) that matches a
method call, or if there are multiple matches, a compilation error occurs

7 [Generic-METHOD IMPLEMENTATION] (Page 3)
× A type parameter is used in place of actual type names and can be used to declare:
+ Return types of a method
+ Parameter types
+ Local variables in the body of a method

8 [Generic-METHOD IMPLEMENTATION] (Page 4)
× Example:
public static void Main(string[] args)
{
...
   int[] intArray = {1, 2, 3, 4, 5, 6};
   double[] doubleArray = {1.1, 2.2, 3.3};
   char[] charArray = {'b', 'i', 'l', 'l'};
   DisplayArray(intArray);
   DisplayArray(doubleArray);
   DisplayArray(charArray);
   ...
}
private static void DisplayArray<T>(T[] myArray)
{
...

12 [TYPE CONSTRAINTS] (Page 1)
× It’s possible to compare two objects of the same type if that type implements the
generic interface IComparable<T> (of namespace System)
+ A benefit of implementing this interface is that its objects can be used with the
  sorting and searching methods of classes in System.Collections.Generic namespace
+ The structures in the Framework Class Library that correspond to the simple types
  (int, double, string etc.) all implement this interface

13 [TYPE CONSTRAINTS] (Page 2)
× Even though IComparable objects can be compared, they cannot be used with generic
code by default, because not all types implement interface IComparable<T>
× However, we can restrict the types that can be used with a generic method or class to
  ensure that they meet certain requirements
× This feature, known as a type constraint, restricts the type of the argument supplied to
  a particular type parameter

14 [TYPE CONSTRAINTS] (Page 3)
This restriction is important, because not all objects can be compared.

All IComparable<T> objects are guaranteed to have a CompareTo() method.

**Type Constraints**

- C# provides several kinds of type constraints.
- A class constraint indicates that the type argument must be an object of a specific base class or one of its subclasses.
- An interface constraint indicates that the type argument’s class must implement a specific interface.

You can specify that the type argument must be a reference type or a value type by using the reference-type constraint (class) or the value-type constraint (struct), respectively.

Finally, you can specify a constructor constraint—new()—to indicate that the generic code can use operator new to create new objects of the type represented by the type parameter.

If a type parameter is specified with a constructor constraint, the type argument’s class must provide a public parameterless or default constructor to ensure that objects of the class can be created without passing constructor arguments; otherwise, a compilation error occurs.

It is possible to apply multiple constraints to a type parameter.

To create type constraints, simply provide a comma-separated list of constraints in the where clause.

If there is a class constraint, reference-type constraint or value-type constraint, it must be listed first (only one of these types of constraints can be used for each type parameter).

Interface constraints (if any) are listed next.

The constructor constraint is listed last (if there is one).

**Type Constraints**

Format:

```
accessModifier [static] type MethodName<T>( [parameterList] ) where T : typeConstraintList
```

Example:

```
private static void
```

**Overloading Generic Methods**

A generic method may be overloaded.

Each overloaded method must have a unique signature (as discussed in Chapter 7).

A class can provide two or more generic methods with the same name but different method parameters.

A generic method can be overloaded by nongeneric methods with the same method name.

When the compiler encounters a method call, it searches for the method declaration that best matches the method name and the argument types specified in the call.