41. Motivation for Generic Types

This chapter starts the lecture about *generics*: Generic types and generic methods. With *generics* we are aiming at more general types (classes, structs, interfaces, etc). The measure that we will bring into use is *type parametrization*.

This chapter is intended as motivation. Type parameterized types will be the topic of Chapter 42 and type parameterized methods will be treated in Chapter 43.

41.1. Operations on sets

Lecture 11 - slide 2

In this chapter we decide to develop and use the class `Set`. We use the class `Set` as a motivating example. It is our goal, once and for all, to be able to write a class `Set` that supports all possible types of elements. It is the intention that the class `Set` can be used in any future program, in which there is a need for sets.

It is noteworthy that .NET has not supported a mathematical set class until version 3.5. As of version 3.5, the class `HashSet<T>` supports sets, see also Section 45.1. Thus, at the time of writing this material, there was no set class available in the .NET Framework.

The class `Set` should represent a mathematical set of items. We equip class `Set` with the usual and well-known set operations:

- *aSet*.**`Member`**(*element*)
- *aSet*.**`Insert`**(*element*)
- *aSet*.**`Delete`**(*element*)
- *aSet*.**`Count`**
- *aSet*.**`Subset`**(*anotherSet*)
- *aSet*.**`GetEnumerator`**()
- *aSet*.**`Intersection`**(*anotherSet*)
- *aSet*.**`Union`**(*anotherSet*)
- *aSet*.**`Diff`**(*anotherSet*)

The set operations `Intersection`, `Union`, and `Diff` are handled in Exercise 11.1.

41.2. The classes IntSet and StringSet

Lecture 11 - slide 3

Let us imagine that we first encounter a need for sets of integers. This causes us (maybe somewhat narrow-minded) to write a class called `IntSet`. Our version of class `IntSet` is shown in Program

41.1. The version provided in the paper version of the material is abbreviated to save some space. The version in the web version is complete with all details.

```
using System;
using System.Collections;

public class IntSet {

  private int capacity;
  private static int DefaultCapacity = 10;
  private int[] store;
  private int next;

  public IntSet(int capacity){
    this.capacity = capacity;
    store = new int[capacity];
    next = 0;                  // The next place to insert
  }

  public IntSet(): this(DefaultCapacity){
  }

  public IntSet(int[] elements): this(elements.Length){
    foreach(int  el in elements) this.Insert(el);
  }

  // Copy constructor
  public IntSet(IntSet s): this(s.capacity){
    foreach(int  el in s) this.Insert(el);
  }

  public bool Member(int  element){
    for(int idx = 0; idx < next; idx++)
      if (element.Equals(store[idx]))
        return true;
    return false;
  }

  public void Insert(int  element){
    if (!this.Member(element)){
      if (this.Full){
        Console.WriteLine("[Resize to {0}]", capacity * 2);
        Array.Resize<int>(ref store, capacity * 2);
        capacity = capacity * 2;
      }
      store[next] = element;
      next++;
    }
  }

  public void Delete(int  element){
    bool found = false;
    int foundIdx = 0;
    for(int idx = 0; !found && (idx < next); idx++){
      if (element.Equals(store[idx])){
         found = true;
         foundIdx = idx;
      }
    }
    if (found){   // shift remaining elements left
      for(int idx = foundIdx+1; idx < next; idx++)
```

```
59          store[idx-1] = store[idx];
60        store[next-1] = default(int  );
61        next--;
62      }
63    }
64
65    // Additional operations: Count, Subset, ToString, Full, and GetEnumerator
66
67  }
```

Program 41.1 *The class IntSet.*

The class `IntSet` is an example of an everyday implementation of integer sets. We have not attempted to come up with a clever representation that allows for fast set operations. The `IntSet` class is good enough for small sets. If you are going to work on sets with many elements, you should use a set class of better quality.

We chose to represent the elements in an integer array. We keep track of the position where to insert the next element (by use of the instance variable `next`). If there is not enough room in the array, we use the `Array.Resize` operation to make it larger. We delete elements from the set by shifting elements in the array 'to the left', in order to avoid wasted space. This approach is fairly expensive, but it is good enough for our purposes. The `IntSet` class is equipped with a `GetEnumerator` method, which returns an iterator. (We encountered iterators (enumerators) in the `Interval` class studied in Section 21.3. See also Section 31.6 for details on iterators. The `GetEnumerator` details are not shown in the paper version). The enumerator allows for traversal of all elements of the set with a **`foreach`** control structure.

A set is only, in a minimal sense, dependent on the types of elements (in our case, the type `int`). It does not even matter if the type of elements is a value type or a reference type (see Section 14.1 and Section 13.1 respectively). We do, however, apply equality on the elements, via use of the `Equals` method. Nevertheless, the type `int` occurs many times in the class definition of `IntSet`. We have emphasized occurrences of `int` with color marks in Program 41.1.

```
1  using System;
2  using System.Collections;
3
4  class App{
5
6   public static void Main(){
7     IntSet s1 = new IntSet(),
8             s2 = new IntSet();
9
10    s1.Insert(1); s1.Insert(2);  s1.Insert(3);
11    s1.Insert(4); s1.Insert(5);  s1.Insert(6);
12    s1.Insert(5); s1.Insert(6);  s1.Insert(8);
13    s1.Delete(3); s1.Delete(6);  s1.Insert(9);
14
15    s2.Insert(8); s2.Insert(9);
16
17    Console.WriteLine("s1: {0}", s1);
18    Console.WriteLine("s2: {0}", s2);
19
20 // Exercises:
21 // Console.WriteLine("{0}", s2.Intersection(s1));
```

```
22 // Console.WriteLine("{0}", s2.Union(s1));
23 // Console.WriteLine("{0}", s2.Diff(s1));
24
25    if (s1.Subset(s2))
26      Console.WriteLine("s1 is a subset of s2");
27    else
28      Console.WriteLine("s1 is not a subset of s2");
29
30    if (s2.Subset(s1))
31      Console.WriteLine("s2 is a subset of s1");
32    else
33      Console.WriteLine("s2 is not a subset of s1");
34  }
35 }
```

Program 41.2 *A client of IntSet.*

In Program 41.2 we see a sample application of class `IntSet`. We establish two empty integer sets `s1` and `s2`, we insert some numbers into these, and we try out some of the set operations on them. The comment lines 20-23 make use of set operations which will be implemented in Exercise 11.1. The output of Program 41.2 confirms that `s2` is a subset of `s1`. The program output is shown in Listing 41.3 (only on web).

We will now assume that we, a couple of days after we have programmed class `IntSet`, realize a need of class `StringSet`. Too bad! Class `StringSet` is almost like `IntSet`. But instead of occurrences of `int` we have occurrences of `string`.

We know how bad it is to copy the source text of `IntSet` to a new file called `StringSet`, and to globally replace 'int' with 'string'. When we need to modify the set class, all our modifications will have do be done twice!

For illustrative purposes - and despite the observation just described - we have made the class `StringSet`, see Program 41.4 (only on web). We have also replicated the client program, in Program 41.5 (only on web) and the program output in Listing 41.6 (only on web).

41.3. The class ObjectSet

Lecture 11 - slide 4

In Section 41.2 we learned the following lesson:

> There is an endless number of *Type*`Set` classes. One for each *Type*. Each of them is similar to the others.

We will now review the solution to the problem which was used in Java before version 1.5, and in C# before version 2. These are the versions of the two languages prior to the introduction of generics.

The idea is simple: We implement a set class of element type `Object`. We call it `ObjectSet`. The type `Object` is the most general type in the type system (see Section 28.2). All other types inherit from the class `Object`.

Below, in Program 41.7 we show the class `ObjectSet`. In the paper version, only an outline with a few constructors and methods is included. The web version shows the full definition of class `ObjectSet`.

```
1  using System;
2  using System.Collections;
3
4  public class ObjectSet {
5
6    private int capacity;
7    private static int DefaultCapacity = 10;
8    private Object[] store;
9    private int next;
10
11   public ObjectSet(int capacity){
12     this.capacity = capacity;
13     store = new Object[capacity];
14     next = 0;
15   }
16
17   // Other constructors
18
19   public bool Member(Object  element){
20     for(int idx = 0; idx < next; idx++)
21       if (element.Equals(store[idx]))
22         return true;
23     return false;
24   }
25
26   public void Insert(Object  element){
27     if (!this.Member(element)){
28       if (this.Full){
29         Console.WriteLine("[Resize to {0}]", capacity * 2);
30         Array.Resize<Object>(ref store, capacity * 2);
31         capacity = capacity * 2;
32       }
33       store[next] = element;
34       next++;
35     }
36   }
37
38   // Other methods
39
40 }
```

Program 41.7 *An outline of the class ObjectSet.*

We can now write programs with a set of `Die`, a set of `BankAccount`, a set of `int`, etc. In Program 41.8 (only on web) we show a program, similar to Program 41.2, which illustrates sets of `Die` objects. (The class `Die` can be found in Section 10.1).

The main problem with class `ObjectSet` is illustrated below in Program 41.10. In line 12-20 we make a set of dice (`s1`), a set of integers (`s2`), a set of strings (`s3`), and set of mixed objects (`s4`). Let

us focus on `s1`. If we take a die out of `s1` with the purpose of using a `Die` operation on it, we need to typecase the element to a `Die`. This is shown in line 23. From the compiler's point of view, all elements in the set `s1` are instances of class `Object`. With the cast `(Die)o` in line 23, we guarantee that each element in the set is a `Die`. (If an integer or a playing card should sneak into the set, an exception will be thrown). - The output of the program is shown in Listing 41.11 (only on web).

```
1  using System;
2  using System.Collections;
3
4  class App{
5
6   public static void Main(){
7     Die d1 = new Die(6),  d2 = new Die(10),
8         d3 = new Die(16), d4 = new Die(8);
9     int sum = 0;
10    string netString = "";
11
12    ObjectSet
13      s1 = new ObjectSet(                  // A set of dice
14             new Die[]{d1, d2, d3, d4}),
15      s2 = new ObjectSet(                  // A set of ints
16             new Object[]{1, 2, 3, 4}),
17      s3 = new ObjectSet(                  // A set of strings
18             new string[]{"a", "b", "c", "d"}),
19      s4 = new ObjectSet(                  // A set of mixed things...
20             new Object[]{new Die(6), "a", 7});
21
22    foreach(Object o in s1){
23       ((Die)o).Toss();
24       Console.WriteLine("{0}", (Die)o);
25    }
26
27    // Some details have been left out
28
29  }
30 }
```

Program 41.10 *A client of ObjectSet - working with set of different types.*

41.4. Problems

Lecture 11 - slide 5

The classes `IntSet`, `StringSet` and `ObjectSet` suffer from both programming and type problems:

- Problems with **`IntSet`** and **`StringSet`**
  - Tedious to write both versions: *Copy and paste* programming.
  - Error prone to maintain both versions
- Problems with **`ObjectSet`**
  - Elements of the set must be downcasted in case we need to use some of their specialized operations
  - We can create an inhomogeneous set
    - A set of "apples" and "bananas"

Generic types, to be introduced in the following chapter, offer a type safe alternative to `ObjectSet`, in which we are able to avoid type casting.

42. Generic Types

Generic types are types that carry type parameters. Type parameterized classes will be of particular importance. The motivation for working with type parameterized classes was gained in Chapter 41.

42.1. The generic class Set<T>

Lecture 11 - slide 7

Let us, right away, present the generic set class **`Set<T>`**. It is shown in Program 42.1. As usual, we show an abbreviated version of the class in the paper edition of the material.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Collections;
4
5  public class Set<T> {
6
7    private int capacity;
8    private static int DefaultCapacity = 10;
9    private T[] store;
10   private int next;
11
12   public Set(int capacity){
13     this.capacity = capacity;
14     store = new T[capacity];
15     next = 0;
16   }
17
18   public Set(): this(DefaultCapacity){
19   }
20
21   public Set(T[] elements): this(elements.Length){
22     foreach(T el in elements) this.Insert(el);
23   }
24
25   // Copy constructor
26   public Set(Set<T> s): this(s.capacity){
27     foreach(T el in s) this.Insert(el);
28   }
29
30   public bool Member(T element){
31     for(int idx = 0; idx < next; idx++)
32       if (element.Equals(store[idx]))
33         return true;
34     return false;
35   }
36
37   public void Insert(T element){
38     if (!this.Member(element)){
39       if (this.Full){
40         Console.WriteLine("[Resize to {0}]", capacity * 2);
41         Array.Resize<T>(ref store, capacity * 2);
42         capacity = capacity * 2;
43       }
```

```
      store[next] = element;
      next++;
    }
  }

  public void Delete(T element){
    bool found = false;
    int  foundIdx = 0;
    for(int idx = 0; !found && (idx < next); idx++){
      if (element.Equals(store[idx])){
         found = true;
         foundIdx = idx;
      }
    }
    if (found){   // shift remaining elements left
      for(int idx = foundIdx+1; idx < next; idx++)
        store[idx-1] = store[idx];
      store[next-1] = default(T);
      next--;
    }
  }

  // Additional operations: Count, Subset, ToString, Full, and GetEnumerator

}
```

Program 42.1 *The class* ***Set<T>***.

The advantage of class **`Set<T>`** over class **`ObjectSet`** becomes clear when we study a client of **`Set<T>`**. Please take a look at Program 42.2 and compare it with Program 41.10. We are able to work with both sets of value types, such as **`Set<int>`**, and sets of reference types, such as **`Set<Die>`**. When we take an element out of the set it is not necessary to cast it, as in Program 41.10. Notice that a **`foreach`** loop does not provide the best illustration of this aspect, because the *type* in `foreach(`*`type`*` var in collection)` is used implicitly for casting a value in collection to *type*. The only way to access elements in a set is to use its iterator. Please take a look at Exercise 11.2 if you wish to go deeper into this issue.

```
using System;
using System.Collections;

class App{

 public static void Main(){
   Die d1 = new Die(6),  d2 = new Die(10),
       d3 = new Die(16), d4 = new Die(8);
   int sum = 0;
   string netString = "";


   // Working with sets of dice:
   Set<Die>  s1 = new Set<Die>(       // A set of dice
                      new Die[]{d1, d2, d3, d4});
   foreach(Die d in s1){
      d.Toss();
      Console.WriteLine("{0}", d);
   }

```

```
22    // Working with sets of ints
23    Set<int> s2 = new Set<int>(        // A set of ints
24                        new int[]{1, 2, 3, 4});
25    foreach(int i in s2)
26       sum += i;
27    Console.WriteLine("Sum: {0}", sum);
28
29
30    // Working with sets of strings
31    Set<string> s3 = new Set<string>(  // A set of strings
32                        new string[]{"a", "b", "c", "d"});
33    foreach(string str in s3)
34       netString += str;
35    Console.WriteLine("Appended string: {0}", netString);
36
37 }
38}
```

Program 42.2 *A client of* ***Set<T>*** *- working with sets of different types.*

The output of Program 42.2 is shown in Listing 42.3 (only on web).

Exercise 11.1. *Intersection, union, and difference: Operations on sets*

On the accompanying slide we have shown a generic class **`set<T>`**.

Add the classical set operations intersection, union and set difference to the generic class **`set<T>`**.

Test the new operations from a client program.

Hint: The enumerator, that comes with the class **`set<T>`**, may be useful for the implementation of the requested set operations.

Exercise 11.2. *An element access operation on sets*

The only way to get access to an element from a set is via use of the enumerator (also known as the iterator) of the set. In this exercise we wish to change that.

Invent some operation on the set that allows you to take out an existing element in the set. This corresponds to accessing a given item in an array or a list, for instance via an indexer: `arr[i]` and `lst[j]`. Notice in this context that there is no order between elements in the set. It is not natural to talk about "the first" or "the last" element in the set.

Given the invented operation in **`Set<T>`** use it to illustrate that, for some concrete type **`T`**, no casting is necessary when elements are accessed from **`Set<T>`**

42.2. Generic Types

Lecture 11 - slide 8

Let us now describe the general concepts behind Generic Types in C#. C# supports not only generic classes, but also generic structs (see Section 42.7), generic interfaces (see Section 42.8), and generic delegate types (see Section 43.2). Overall, we distinguish between templates and constructed types:

- Templates
  - **`C<T>`** is not a type
  - **`C<T>`** is a template from which a type can be constructed
  - **`T`** is a *formal type parameter*
- Constructed type
  - The type constructed from a template
  - **`C<int>`**, **`C<string>`**, and **`D<C<int>>`**
  - **`int`**, **`string`**, and **`C<int>`** are *actual type parameters* of **`C`** and **`D`**

When we talk about a generic type we do it in the meaning of a template.

The word "template" is appropriate, and in fact just to the point. But most C# writers do not use it, because the word "template" it used in C++ in a closely related, but slightly different meaning. A template in C++ is a type parameterized class, which is expanded at compile time. Each actual type parameter will create a new class, just like we would create it ourselves in a text editor. In C#, generic classes are able to share the class representation at run-time. For more details on these matters, consult for instance [Golding05].

As a possible coding style, it is often recommended to use capital, single letter names (such as `S`, `T`, and `U`) as formal type parameters. In that way it becomes easier to recognize templates, to spot formal type names in our programs, to keep templates apart from constructed types, and to avoid very name clauses of generic types. In situations where a type takes more than one formal type parameters, an alternative coding style calls for formal type parameter names like `T`*x* and `T`*y*, (such as `TKey` and `TValue`) where *x* and *y* describe the role of each of the formal type parameters.

The ability to have generic types is known as *parametric polymorphism*

42.3. Constraints on Formal Type Parameters

Lecture 11 - slide 9

Let us again consider our implementation of the generic class **`Set<T>`** in Program 42.1. Take a close look at the class, and find out if we make any assumptions about the formal type parameter **`T`** in Program 42.1. Will any type **`T`** really apply? Please consider this, before you proceed!

In **`Set<T>`** it happens to be the case that we do not make any assumption of the type parameter `T`. This is typical for *collection classes* (which are classes that serve as element *containers*).

It is possible to express a number of constraints on a formal type parameter

The more constraints on **`T`**, the more we can do on **`T`**-objects in the body of **`C<T>`**

Sometimes we write a parameterized class, say **`C<T>`**, in which we wish to be able to make some concrete assumptions about the type parameter `T`. You may ask what we want to express. We could, for instance, want to express that

1. **`T`** is a value type, allowing for instance use of the type **`T?`** (nullable types, see Section 14.9) inside **`C<T>`** .
2. **`T`** is a reference type, allowing, for instance, the program fragment **`T v; v = null;`** inside **`C<T>`** .
3. **`T`** has a multiplicative operator **`*`** , allowing for expressions like **`T t1, t2; ... t1 * t2 ...`** in **`C<T>`** .
4. **`T`** has a method named **`M`** , that accepts a parameter which is also of type **`T`** .
5. **`T`** has a C# indexer of two integer parameters, allowing for **`T t; ... t[i, j] ...`** within **`C<T>`** .
6. **`T`** is a subclass of class **`BankAccount`** , allowing for the program fragment **`T ba; ba.AddInterests();`** within **`C<T>`** .
7. **`T`** implements the interface **`IEnumerable`** , allowing **`foreach`** iterations based on T in **`C<T>`** , see Section 31.6 .
8. **`T`** is a type with a parameterless constructor, allowing the expression **`new T()`** in **`C<T>`** .

It turns out that the constraints in 1, 2, 6, 7, and 8 can be expressed directly in C#. The constraints in 4 and 5 can be expressed indirectly in C#, whereas the constraint in 3 cannot be expressed in C#.

Here follows a program fragment that illustrates the legal form of *constraints* on type parameters in generic types in C#. We define generic classes `C`, `E`, `F`, and `G` all of which are subclasses of class `D`. `A` and `B` are classes defined elsewhere. The constraints are colored in Program 42.4.

```
1  class C<S,T>: D
2      where T: A, ICloneable
3      where S: B {
4   ...
5  }
6
7  class E<T>: D
8      where T: class{
9   ...
10 }
11
12 class F<T>: D
13     where T: struct{
14  ...
15 }
16
17 class G<T>: D
18     where T: new(){
19  ...
20 }
```

Program 42.4 *Illustrations of the various constraints on type parameters.*

The class `C` has formal type parameters `S` and `T`. The first constraint requires that `T` is `A`, or a subclass of `A`, and that it implements the interface `IClonable`. Thus, only class `A` or subclasses of `A` that implement `IClonable` can be used as actual parameter corresponding to `T`. The type parameter `S` must be `B` or a subclass of `B`.

The class `E` has a formal type parameter `T`, which must be a class. In the same way, the class `F` has a formal type parameter `T`, which must be a struct.

The class `G` has a formal type parameter `T`, which must have a parameterless constructor.

As a consequence of the inheritance rules in C#, only a single class can be given in a constraint. Multiple interfaces can be given. A class should come before any interface. Thus, in line 2 of Program 42.4, where `T` is constrained by `A, ICloneable`, `A` can be a class, and everything after `A` in the constraint need to be interfaces.

42.4. Constraints: Strings of comparable elements

Lecture 11 - slide 10

We will now program a generic class with constraints. We will make a class `String<T>` which generalizes `System.String` from C#. An instance of `String<T>` contains a sequence of `T`-values/ `T`-objects. In contrast, an instance of `System.String` contains a sequence of Unicode characters. With use of `String<T>` we can for instance make a string of integers, a string of bank accounts, and a string of dice.

Old-fashioned character strings can be ordered, because we have an ordering of characters. The ordering we have in mind is sometimes called *lexicographic ordering*, because it reflects the ordering of words in dictionaries and encyclopedia. We also wish to support ordering of our new generalized strings from `String<T>`. It can only be achieved if we provide an ordering of the values/objects in `T`. This is done by requiring that `T` implements the interface `IComparable`, which has a single method `CompareTo`. For details on `IComparable` and `CompareTo`, please consult Section 31.5.

Now take a look at the definition of `String<T>` in Program 42.5. In line 3 we state that `String<T>` should implement the interface `IComparable<String<T>>`. It is important to understand that we hereby commit ourselves to implement a `CompareTo` method in `String<T>`.

You may be confused about the interface `IComparable`, as discussed in Program 42.5 in contrast to `IComparable<S>`, which is used as `IComparable<String<T>>` in line 3 of Program 42.5. `IComparable<S>` is a generic interface. It is generic because this allows us to specify the parameter to the method `CompareTo` with better precision. We discuss the generic interface `IComparable<S>` in Section 42.8.

There is an additional important detail in line 3 of Program 42.5, namely the constraint, which is colored. The constraint states that the type `T` must be `IComparable` itself (again using the generic version of the interface). In plain English it means that there must be a `CompareTo` method available on the type, which we provide as the actual type parameter of our new string class. Our plan is, of course, to use the `CompareTo` method of `T` to program the `CompareTo` method of `String<T>`.

```
1  using System;
2
3  public class String<T>: IComparable<String<T>> where T: IComparable<T>{
4
5    private T[] content;
6
7    public String(){
8      content = new T[0];
9    }
10
11   public String(T e){
12     content = new T[]{e};
13   }
14
15   public String(T e1, T e2){
16     content = new T[]{e1, e2};
17   }
18
19   public String(T e1, T e2, T e3){
20     content = new T[]{e1, e2, e3};
21   }
22
23   public int CompareTo(String<T> other){
24     int thisLength = this.content.Length,
25         otherLength = other.content.Length;
26
27     for (int i = 0; i < Math.Min(thisLength,otherLength); i++){
28       if (this.content[i].CompareTo(other.content[i]) < 0)
29          return -1;
30       else if (this.content[i].CompareTo(other.content[i]) > 0)
```

```
31          return 1;
32      }
33      // longest possible prefixes of this and other are pair-wise equal.
34      if (thisLength < otherLength)
35         return -1;
36      else if (thisLength > otherLength)
37         return 1;
38      else return 0;
39    }
40
41    public override string ToString(){
42      string res = "[";
43      for(int i = 0; i < content.Length;i++){
44        res += content[i];
45        if (i < content.Length - 1) res += ", ";
46      }
47      res += "]";
48      return res;
49    }
50
51  }
```

Program 42.5 *The generic class* ***`String<T>`***.

In line 5 we see that a string of `T`-elements is represented as an array of `T` elements. This is a natural and straightforward choice. Next we see four constructors, which allows us to make strings of zero, one, two or three parameters. This is convenient, and good enough for toy usage. For real life use, we need a general constructor that accepts an array of `T` elements. The can most conveniently be made by use of parameter arrays, see Section 20.9.

After the constructors, from line 23-39, we see our implementation of `CompareTo`. From an overall point of view we can observe that it uses `CompareTo` of type `T`, as discussed above. This is the **blue** aspects in line 28 and 30. It may be sufficient to make this observation for some readers. If you want to understand what goes on inside the method, read on.

Recall that `CompareTo` must return a negative result if the current object is less than `other`, 0 if the current object is equal to `other`, and a positive result if the current object is greater than `other`. The for-loop in line 27 traverses the overlapping prefixes of two strings. Inside the loop we return a result, if it is possible to do so. If the for-loop terminates, the longest possible prefixes of the two string are equal to each other. The lengths of the two strings are now used to determine a result.

If `T` is the type `char`, if the current string is "abcxy", and if `other` is "abcxyz", we compare "abcxy" with "abcxy" in the for loop. "abcxy" is shorter than "abcxyz", and therefore the result of the comparison -1.

The method `ToString` starting in line 41 allows us to print instances of `String<T>` in the usual way.

In Program 42.6 we see a client class of `String<T>`. We construct and compare strings of integers, strings of strings, strings of doubles, strings of booleans, and strings of dice. The dimmed method `ReportCompare` activates the `String<T>` operation `CompareTo` on pairs of such strings. `ReportCompare` is a generic method, and it will be "undimmed" and explained in Program 43.1. Take a look at the program output in Listing 42.7 and be sure that you can understand the results.

```
using System;

class StringApp{

  public static void Main(){

    ReportCompare(new String<int>(1, 2),
                  new String<int>(1));
    ReportCompare(new String<string>("1", "2", "3"),
                  new String<string>("1"));
    ReportCompare(new String<double>(0.5, 1.7, 3.0),
                  new String<double>(1.0, 1.7, 3.0));
    ReportCompare(new String<bool>(true, false),
                  new String<bool>(false, true));
    ReportCompare(new String<Die>(new Die(), new Die()),
                  new String<Die>(new Die(), new Die()));
  }

  public static void ReportCompare<T>(String<T> s, String<T> t)
    where T: IComparable<T>{
    Console.WriteLine("Result of comparing {0} and {1}: {2}",
                      s, t, s.CompareTo(t));
  }

}
```

Program 42.6 *Illustrating Strings of different types.*

```
Result of comparing [1, 2] and [1]: 1
Result of comparing [1, 2, 3] and [1]: 1
Result of comparing [0,5, 1,7, 3] and [1, 1,7, 3]: -1
Result of comparing [True, False] and [False, True]: 1
Result of comparing [[3], [6]] and [[3], [5]]: 1
```

Listing 42.7 *Output from the String of different types program.*

Exercise 11.3. *Comparable Pairs*

This exercise is inspired by an example in the book by Hansen and Sestoft: *C# Precisely*.

Program a class **`ComparablePair<T,U>`** which implements the interface **`IComparable<ComparablePair<T,U>>`**. If you prefer, you can build the class **`ComparablePair<T,U>`** on top of class **`Pair<S,T>`** from an earlier exercise in this lecture.

It is required that **`T`** and **`U`** are types that implement **`Icomparable<T>`** and **`Icomparable<U>`** respectively. How is that expressed in the class **`ComparablePair<T,U>`**?

The generic class **`ComparablePair<T,U>`** should represent a pair *(t,u)* of values/objects where *t* is of type `T` and *u* is of type `U`. The generic class should have an appropriate constructor that initializes both parts of the pair. In addition, there should be properties that return each of the parts. Finally, the class should - of course - implement the operation `CompareTo` because it is prescribed by the interface **`System.IComparable<ComparablePair<T,U>>`**.

Given two pairs p = (a,b) and q= (c,d). p is considered less than q if a is less than c. If a is equal to

c then b and d controls the ordering. This is similar to lexicographic ordering on strings.

If needed, you may get useful inspiration from the `Icomparable` class **`String<T>`** on the accompanying slide.

Be sure to test-drive your solution!

42.5. Another example of constraints

Lecture 11 - slide 11

We will now illustrate the need for the class and struct constraints. We have already touched on these constraints in our discussion of Program 42.4.

In Program 42.8 we have two generic classes `C` and `D`. Each of them have a single type parameter, `T` and `U` respectively. As shown with **red** color in line 7 and 15, the compiler complains. In line 7 we assign the value `null` to the variable `f` of type `T`. In line 15 we make a nullable type `U?` from `U`. (If you wish to be reminded about nullable types, consult Section 14.9). Before you go on, attempt to explain the error messages, which are shown as comments in Program 42.8.

```
1  /* Example from Hansen and Sestoft: C# Precisely */
2
3  class C<T>{
4    // Compiler Error message:
5    // Cannot convert null to type parameter 'T' because it could
6    // be a value type. Consider using 'default(T)' instead.
7    T f = null;
8  }
9
10 class D<U>{
11   // Compiler Error message:
12   // The type 'U' must be a non-nullable value type in order to use
13   // it as parameter 'T' in the generic type or method
14   // 'System.Nullable<T>'
15   U? f;
16 }
```

Program 42.8 *Two generic classes C and D - with compiler errors.*

In Program 42.9 we show new versions of `C<T>` and `D<U>`. Shown in **purple** we emphasize the constraints that are necessary for solving the problems.

The instance variable `f` of type `T` in `C<T>` is assigned to `null`. This only makes sense if `T` is a reference type. Therefore the **class** constraint on `T` is necessary.

The use of `U?` in `D<U>` only makes sense if `U` is a value type. (To understand this, you are referred to the discussion in Section 14.9). Value types in C# are provided by structs (see Section 6.6). The **struct** constraint on `U` is therefore the one to use.

```
1  /* Example from Hansen and Sestoft: C# Precisely */
2
3  class C<T> where T: class{
4    T f = null;
5  }
6
7  class D<U> where U: struct{
8    U? f;
9  }
10
11 class Appl{
12
13   // Does NOT compile:
14   C<double> c = new C<double>();
15   D<A>      d = new D<A>();
16
17   // OK:
18   C<A>      c1 = new C<A>();
19   D<double> d1 = new D<double>();
20
21 }
22
23 class A{}
```

Program 42.9 *Two generic classes C and D - with the necessary constraints.*

In line 11-21 we show clients of `C<T>` and `D<U>`. The compiler errors in line 14 and 15 are easy to explain. The type `double` is not a reference type, and `A`, which is programmed in line 23, is not a value type. Therefore `double` and `A` violate the constraints of `C<T>` and `D<U>`. In line 18 and 19 we switch the roles of `double` and `A`. Now everything is fine.

42.6. Variance

Lecture 11 - slide 12

Consider the question asked in the following box.

A **`CheckAccount`** ***is a*** **`BankAccount`** But is a **`Set<CheckAccount>`** a **`Set<BankAccount>`** ?

You are encouraged to review our discussion of the ***is a*** relation in Section 25.2.

The question is how `Set<T>` is varies when `T` varies. Variation in this context is specialization, cf. Chapter 25. Is `Set<T>` specialized when `T` is specialized?

Take a look at Program 42.10. In line 7-14 we construct a number of bank accounts and check accounts, and we make a set of bank accounts (`s1`, in line 17) and a set of check accounts (`s2`, in line 18). In line 21 and 22 we populate the two sets. So far so good. Next, in line 25 (shown in **purple**) we play the polymorphism game as we have done many times earlier, for example in line

13 of Program 28.17. If `Set<CheckAccount>` ***is a*** `Set<BankAccount>` line 25 of Program 42.10 should be OK (just as line 13 of Program 28.17 is OK).

The compiler does not like line 25, however. The reason is that `Set<CheckAccount>` ***is NOT a*** `Set<BankAccount>`.

If we for a moment assume that `Set<CheckAccount>` ***is a*** `Set<BankAccount>` the rest of the program reveals the troubles. We insert a new `BankAccount` object in `s1`, and via the alias established in line 25, the new `BankAccount` object is also inserted into `s2`. When we in line 34-35 iterate through all the `CheckAccount` objects of the set `s2`, we encounter an instance of `BankAccount`. We cannot carry out a `SomeCheckAccountOperation` on an instance of `BankAccount`.

```
1  using System;
2
3  class SetOfAccounts{
4
5    public static void Main(){
6
7      // Construct accounts:
8      BankAccount ba1 = new BankAccount("John", 0.02),
9                  ba2 = new BankAccount("Anne", 0.02),
10                 ba3 = new BankAccount("Frank", 0.02);
11
12     CheckAccount ca1 = new CheckAccount("Mike", 0.03),
13                  ca2 = new CheckAccount("Lene", 0.03),
14                  ca3 = new CheckAccount("Joan", 0.03);
15
16     // Constructs empty sets of accounts:
17     Set<BankAccount> s1 = new Set<BankAccount>();
18     Set<CheckAccount> s2 = new Set<CheckAccount>();
19
20     // Insert elements in the sets:
21     s1.Insert(ba1);  s1.Insert(ba2);
22     s2.Insert(ca1);  s2.Insert(ca2);
23
24     // Establish s1 as an alias to s2
25     s1 = s2;   // Compile-time error:
26                // Cannot implicitly convert type 'Set<CheckAccount>'
27                // to 'Set<BankAccount>'
28
29     // Insert a BankAccount object into s1,
30     // and via the alias also in s2
31     s1.Insert(new BankAccount("Bodil", 0.02));
32
```

```
33      // Activates some CheckAccount operation on a BankAccount object
34      foreach(CheckAccount ca in s2)
35        ca.SomeCheckAccountOperation();
36
37      Console.WriteLine("Set of BankAccount: {0}", s1);
38      Console.WriteLine("Set of CheckAccount: {0}", s2);
39
40
41    }
42
43  }
```

Program 42.10 *Sets of check accounts and bank accounts.*

The experimental insight obtained above is - perhaps - against our intuition. It can be argued that an instance of `Set<CheckAccount>` should be a valid stand in for an instance of `Set<BankAccount>`, as attempted in line 25. On the other hand, it can be asked if the extension of `Set<CheckAccount>` is a subset of `Set<BankAccount>`. (See Section 25.2 for a definition of extension). Or asked in this way: Is the set of set of check accounts a subset of a set of set of bank accounts? As designed in Section 25.3 the set of *CheckAccounts* is a subset of the set of *BankAccount*. But this does not imply that the set of set of `CheckAccount` is a subset of the set of set of `BankAccount`. A set of `CheckAccount` (understood as a single objects) is incompatible with a set of `BankAccount` (understood as a single object).

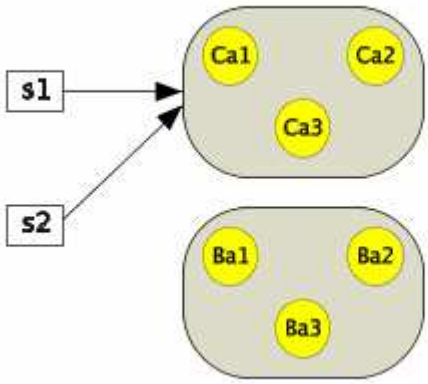

Figure 42.1 *A set of bank accounts and a set of check accounts*

In Program 42.10 we establish the scene illustrated in Figure 42.1. More precisely, the illustration shows the situation as of line 28 of Program 42.10. The problem is that we in line 31 add a new instance of `BankAccount` to `s1`, which refers to an instance of `Set<CheckAccount>`. Later in the program (line 35) this would cause "a minor explosion" if the program was allowed to reach this point . Thus, the real problem occurs if we mutate the set of check accounts that are referred from a variable of static type `Set<BankAccount>`. (See Section 28.10 for the definition of *static type*).

In general, we distinguish between the following kinds of variances in between `Set<T>` and `T`:

- **Covariance**
  - The set types vary in the same way as the element types
- **Contravariance**
  - The set types vary in the opposite way as the element types
- **Invariance**
  - The set types are not affected by the variations of the element types

If Program 42.10 could be compiled and executed without problems (if line 25 is considered OK), then we would have covariance between `Set<T>` and `T`

In C# `Set<T>` is invariant in relation to `T`.

We notice that the problem discussed above is similar to the parameter variance problem, which we discussed in Section 29.2.

C# and Java do both agree on invariance in between `Set<T>` and `T`. But in contrast to C#, Java has a solution to the problem in terms of *wildcard types*. We realized above that `Set<T>` is not a generalization of all sets. In Java 1.5, a wildcard type written as `Set<?>` (a set of unknown) is a generalization of all sets. It is, however, not possible to mutate an object of static type `Set<?>`. If you are interested to known more about generics in Java, you should consult Gilad Bracha's tutorial "Generics in the Java Programming Language", [Bracha2004].

42.7. Generic structs

Lecture 11 - slide 13

It is possible to make type parameterized structs, similar to the type parameterized classes that we have seen in the previous sections.

As an example we will see how we can define the generic struct `Nullable<T>` which defines the type behind the notation `T?` for an arbitrary value type `T`. Nullable types were discussed earlier in Section 14.9. Recall that nullable types enjoy particular compiler support, beyond the translation of `T?` to `Nullable<T>`. This includes support of lifted operators (operators that are extended to work on `T?` in addition to `T`) and support of the `null` value as such.

```
1  using System;
2
3  public struct Nullable<T>
4    where T : struct{
5
6    private T value;
7    private bool hasValue;
8
9    public Nullable(T value){
10     this.value = value;
11     this.hasValue = true;
12   }
13
```

```
14   public bool HasValue{
15     get{
16       return hasValue;
17     }
18   }
19
20   public T Value{
21     get{
22       if(hasValue)
23         return value;
24       else throw new InvalidOperationException();
25     }
26   }
27
28 }
```

Program 42.11 *A partial reproduction of struct* `Nullable<T>`.

The generic struct `Nullable<T>` aggregates a value of type T and a boolean value. The boolean value is stored in the boolean instance variable `hasValue`. If `nv` is of type `Nullable<T>` for some value type `T`, and if the variable `hasValue` of `nv` is `false`, then `nv` is considered to have the value `null`. The compiler arranges that the assignment `nv = null` is translated to `nv.hasValue = false`. This is somehow done behind the scene because `hasValue` is private.

42.8. Generic interfaces: IComparable<T>

Lecture 11 - slide 14

In this section we will take a look at the generic interface `IComparable<T>`. We have earlier in the material (Section 31.5) studied the non-generic interface `Icomparable`, see Program 31.6.

If you review your solution to Exercise 8.6 you should be able to spot the weakness of a class `ComparableDie`, which implements `IComparable`. The weakness is that the parameter of the method `CompareTo` must have an `Object` as parameter. A method with the signature `CompareTo(Die)` does not implement the interface `IComparable`. (Due to static overloading of methods in C#, the methods `CompareTo(Object)` and `CompareTo(Die)` are two different methods, which just as well could have the signatures `ObjectCompareTo(Object)` and `DieCompareTo(Die)`). Thus, as given by the signature of `CompareTo`, we compare a `Die` and any possible object.

In Program 42.12 we reproduce `IComparable<T>`. Program 42.12 corresponds to Program 31.6. (Do not use any of these - both interfaces are parts of the `System` namespace). As it appears, in the generic interface the parameter of `CompareTo` is of type `T`. This alleviates the problem of the non-generic interface `IComparable`.

```
using System;

public interface IComparable <T>{
  int CompareTo(T other);
}
```

Program 42.12 *A reproduction of the generic interface* ***IComparable<T>***.

Below we show a version of class `Die` which implements the interface `IComparable<Die>`. You should notice that this allows us to use `Die` as formal parameter of the method `CompareTo`.

```
using System;

public class Die: IComparable<Die> {
  private int numberOfEyes;
  private Random randomNumberSupplier;
  private const int maxNumberOfEyes = 6;

  public Die(){
    randomNumberSupplier = Random.Instance();
    numberOfEyes = NewTossHowManyEyes();
  }

  public int CompareTo(Die other){
    return this.numberOfEyes.CompareTo(other.numberOfEyes);
  }

  // Other Die methods

}
```

Program 42.13 *A class Die that implements* ***IComparable<T>***.

The implementation of the generic interface is more type safe and less clumsy than the implementation of the non-generic solution

42.9. Generic equality interfaces

Lecture 11 - slide 15

Before reading this section you may want to remind yourself about the fundamental equality operations in C#, see Section 13.5.

There exist a couple of generic interfaces which prescribes `Equals` operations. The most fundamental is `IEquatable<T>`, which prescribes a single `Equals` instance method. It may be attractive to implement `IEquatable` in certain structs, because it could avoid the need of boxing the struct value in order to make use of the inherited `Equals` method from class `Object`.

`IEqualityComparer<T>` is similar, but it also supports a `GetHasCode` method. (Notice also that the signatures of the `Equals` methods are different in the two interfaces. `IEquatable<T>` prescribes `x.Equals(y)` whereas `IEqualityComparer<T>` prescribes `Equals(x,y)`).

Below, in Program 42.14 and Program 42.15 we show reproductions of the two interfaces. Notice again that the two interfaces are present in the namespaces `System` and `System.Collections.Generic` respectively. Use them from there if you need them.

```
1 using System;
2
3 public interface IEquatable <T>{
4   bool Equals (T other);
5 }
```

Program 42.14 *A reproduction of the generic interface* ***`IEquatable<T>`***.

```
1 using System;
2
3 public interface IEqualityComparer <T>{
4   bool Equals (T x, T y);
5   int GetHashCode (T x);
6 }
```

Program 42.15 *A reproduction of the generic interface* ***`IEqualityComparer<T>`***.

Several operations in generic collections, such as in `List<T>` in Section 45.9, need equality operations. The `IndexOf` method in `List<T>` is a concrete example, see Section 45.11. Using `lst.IndexOf(el)` we search for the element `el` in the list `lst`. Comparison of `el` with the elements of the list is done by the *default equality comparer* of the type `T`. The abstract generic class `EqualityComparer<T>` offers a static `Default` property. The `Default` property delivers the default equality comparer for type `T`. The abstract, generic class `EqualityComparer<T>` implements the interface `IEqualityComparer<T>`.

Unfortunately the relations between the generic interfaces `IEquatable<T>` and `IEqualityComparer<T>`, the class `EqualityComparer<T>` and its subclasses are quite complicated. It seems to be the cases that these interfaces and classes have been patched several times, during the evolution of versions of the .Net libraries. The final landscape of types is therefore more complicated than it could have been desired.

42.10. Generic Classes and Inheritance

Lecture 11 - slide 16

In this section we will clarify inheritance relative to generic classes. We will answer the following questions:

Can a generic/non-generic class inherit from a non-generic/generic class?

The legal and illegal subclassings are summarized below:

- Legal subclassing
  - A generic subclass of a non-generic superclass
  - A generic subclass of a constructed superclass
  - A generic subclass of generic superclass
- Illegal subclassing
  - A non-generic subclass of generic superclass

You can refresh the terminology (generic class/constructed class) in Section 42.2.

The rules are exemplified below.

```
1  using System;
2
3  // A generic subclass of a non-generic superclass.
4  class SomeGenericSet1<T>: IntSet{
5    // ...
6  }
7
8  // A generic subclass of a constructed superclass
9  class SomeGenericSet2<T>: Set<int>{
10   // ...
11 }
12
13 // A generic subclass of a generic superclass
14 // The most realistic case
15 class SpecializedSet<T>: Set<T>{
16   // ...
17 }
18
19 // A non-generic subclass of a generic superclass
20 // Illegal. Compile-time error:
21 // The type or namespace name 'T' could not be found
22 class Set: Set<T>{
23   // ...
24 }
```

Program 42.16 *Possible and impossible subclasses of* ***`Set`*** *classes.*

From line 4 to 6 we are about to program a generic class `SomeGenericSet1<T>` based on a non-generic class `IntSet`. This particular task seems to be a difficult endeavor, but it is legal - in general - to use a non-generic class as a subclass of generic class.

Next, from line 9 to 11, we are about to program a generic class `SomeGenericSet2<T>` based on a constructed class `Set<int>`. This is also OK.

From line 15-17 we show the most realistic case. Here we program a generic class based on another generic class. In the specific example, we are about to specialize `Set<T>` to `SpecializedSet<T>`. The type parameter `T` of `SpecializedSet<T>` also becomes the type parameter of `Set<T>`. In general, it would also be allowed for `SpecializedSet<T>` to introduce additional type parameters, such as in `SpecializedSet<T,S> : Set<T>`.

The case shown from line 22 to 24 is illegal, simply because `T` is not the name of any known type. In line 22, `T` is name of an *actual* type parameter, but `T` is not around! It is most likely that the programmer is confused about the roles of *formal* and *actual* type parameters, see Section 42.2.

42.11. References

[Bracha2004] Gilad Bracha, "Generics in the Java Programming Language", July 2004.

[Golding05] Tod Golding, *Professional .NET 2.0 Generics*. Wiley Publishing, Inc., 2005.

43. Generic Methods

We are used to working with procedures, functions, and methods with parameters. Procedures, functions and methods are all known as abstractions. A parameter is like a variable that generalizes the abstraction. Each parameter of a procedure, a function, or a method is of a particular type. In this chapter we shall see how such types themselves can be passed as parameters to methods. When methods are parameterized with types, we talk about *generic methods*.

43.1. Generic Methods

Lecture 11 - slide 18

In Section 42.2 we realized that a generic type (such as a generic class) is a template from which it is possible to construct a real class. In the same way, a generic method is template from which we can construct a real method.

In C# and similar languages, all methods belong to classes. Some of these classes are generic, some are just simple, ordinary classes. We can have generic methods in both generic types, and in non-generic types.

Our first example in Program 43.1 is the generic method `ReportCompare` in the non-generic class `StringApp`. `ReportCompare` is a method in the client class of `String<T>` which we encountered in Section 42.4. When we first met it, we where not interested in the details of it, so therefore it was dimmed in Program 42.6.

Notice first that the method `ReportCompare` takes two ordinary parameters `s` and `t`. They are both of type `String<T>` for some given type `T`. The method is supposed to report the ordering of `s` relative to `t` via output written to the console. `T` is a (formal) type parameter of the method. Type parameters of methods are given in "triangular brackets" **<...>** in between the method name and the ordinary parameter list. It is highlighted with **purple** in Program 43.1.

The formal type parameter of `ReportCompare` is passed on as an actual type parameter to our generic class `String<T>` from Section 42.4. If we look at our definition of the generic class `String<T>` in Program 42.5 we notice that `T` must implement `Icomparable<T>`. This is a constraint of `T`, identical to one of the constraints of type parameters of types, see Section 42.3. The only way to ensure this in Program 43.1 is to add the constraint to the generic method. This is the **blue** part, see line 15.

Notice in line 7-11 of Program 43.1 that the actual type parameter of `ReportCompare` is not given explicitly. The actual type parameters of the five calls are conveniently inferred from the context. It is, however, possible to pass the actual type parameter explicitly. If we chose to do so, line 7 of Program 43.1 would be

```
ReportCompare<int>(new String<int>(), new String<int>(1));
```

The remaining aspects of `ReportMethod` are simple and straightforward.

```
1  using System;
2
3  class StringApp{
4
5    public static void Main(){
6
7      ReportCompare(new String<int>(), new String<int>(1));
8      ReportCompare(new String<int>(1), new String<int>(1));
9      ReportCompare(new String<int>(1,2,3), new String<int>(1));
10     ReportCompare(new String<int>(1), new String<int>(1,2,3));
11     ReportCompare(new String<int>(1,2,3), new String<int>(1,2,3));
12   }
13
14   public static void ReportCompare<T>(String<T> s, String<T> t)
15     where T: IComparable<T>{
16     Console.WriteLine("Result of comparing {0} and {1}: {2}",
17                       s, t, s.CompareTo(t));
18   }
19
20 }
```

Program 43.1 *The generic method ReportCompare in the generic String programs.*

Let us now study an additional program example with generic methods. Program 43.2 contains a bubblesort method in line 5-11. `Bubblesort` sorts an array of element type `T`, where `T` is a type parameter of the method. The type parameter makes our bubblesort method more general, because it allow us to sort an array of arbitrary type `T`. The only requirement is, quite naturally, that objects/values of type type `T` should be comparable, such that we can ask if one value is less than or equal to another value. This is expressed by the `Icomparable<T>` constraint on T at the end of line 5.

The implementation of bubblesort in Program 43.2 has no surprises. In a double for loop we compare and swap elements. Comparison is made possible because `a[i]` values are of type `T` that implements `Icomparable<T>`. Swapping of elements are done by the `Swap` method via use of C# **ref** parameters, see Section 20.6. Notice that `Swap` is also a generic method, because it can swap values/objects of arbitrary types. Be sure to notice the formal type parameter `T` of `Swap` in line 13.

Finally we have the generic method `ReportArray`, (see line 18-21), which simply prints the values of the array to standard output.

```
1  using System;
2
3  class SortDemo{
4
5    static void BubbleSort<T>(T[] a) where T: IComparable<T>{
6     int n = a.Length;
7     for (int i = 0; i < n - 1; ++i)
8       for (int j = n - 1; j > i; --j)
9         if (a[j-1].CompareTo(a[j]) > 0)
10           Swap(ref a[j-1], ref a[j]);
11   }
12
```

```
13    public static void Swap<T>(ref T a, ref T b){
14      T temp;
15      temp = a; a = b; b = temp;
16    }
17
18    public static void ReportArray<T>(T[] a){
19      foreach(T t in a) Console.Write("{0,4}", t);
20      Console.WriteLine();
21    }
22
23    public static void Main(){
24      double[] da = new double[]{5.7, 3.0, 6.9, -5,3, 0.3};
25
26      Die[] dia = new Die[]{new Die(), new Die(),  new Die(),
27                            new Die(),  new Die(),  new Die()};
28
29      ReportArray(da); BubbleSort(da); ReportArray(da);
30      Console.WriteLine();
31      ReportArray(dia); BubbleSort(dia); ReportArray(dia);
32      Console.WriteLine();
33
34      // Equivalent:
35      ReportArray(da); BubbleSort<double>(da); ReportArray(da);
36      Console.WriteLine();
37      ReportArray(dia); BubbleSort<Die>(dia); ReportArray(dia);
38    }
39
40  }
```

Program 43.2 *A generic bubble sort program.*

In the `Main` method we make an array of doubles and an array of dice. Values of type `double` are comparable. We compile the program with a version of class `Die` that implements `IComparable<T>`, such as the `Die` class of Program 42.13. The calls of `BubbleSort` in line 29 and 31 do not supply an actual type parameter to `BubbleSort<T>`. The compiler is smart enough to infer the actual type parameter from the declared types of the variables `da` and `dia` respectively. In line 35 and 37 we show equivalent calls of `BubbleSort` to which we explicitly supply the actual type parameters `double` and `Die`.

The output of Program 43.2 is shown in Listing 43.3 (only on web).

43.2. Generic Delegates

Lecture 11 - slide 19

Delegates were introduced in Section 22.1. Recall from there that a delegate is a type of methods. In the previous section we learned about generic methods. It therefore not surprising that we also need to discuss generic delegates.

In Program 22.3 we introduced a delegate `NumericFunction`, which covers all function from `double` to `double`. In the same program we also introduced `Compose`, which composes two numeric functions to a single numeric function. In mathematical notation, the composition of f and g is

denoted $f \circ g$, and it maps x to $f(g(x))$. We are now going to generalize the function `Compose`, such that it can be used on other functions of more general signatures.

Let us assume that we work with two functions f and g of the following signatures:

- $g : T \rightarrow U$
- $f : U \rightarrow S$

Thus, g maps a value of type T to a value of type U. f maps a value of type U to a value of type S. The composite function $f \circ g$ therefore maps a value of type T to a value of type S via a value of type U:

- $f \circ g : T \rightarrow S$

In line 6 of Program 43.4 we show a delegate called `Function`, which is a function type that maps a value of type S to values of type T. (It corresponds to `NumericFunction` in Program 22.3). In line 10-13 of Program 43.4 we show the function `Compose`, which we motivated above. `Function` is a *generic delegate* because it is type parameterized. `Compose` is a generic method, as discussed in Section 43.1. The generic method `PrintTableOfFunction`, shown in line 16-23, takes a `Function` `f` and an array `inputValues` of type `S[]`, and it applies and prints `f(s)` on each element `s` of `inputValues`.

```
1  using System;
2
3  public class CompositionDemo {
4
5    // A function from S to T
6    public delegate T Function <S,T>(S d);
7
8    // The generic function for function composition
9    // from T to S via U
10   public static Function<T,S> Compose<T,U,S>
11                     (Function<U,S> f, Function<T,U> g){
12     return delegate(T d){return f(g(d));};
13   }
14
15   // A generic PrintTable function
16   public static void PrintTableOfFunction<S,T>
17                     (Function<S,T> f, string fname,
18                      S[] inputValues){
19     foreach(S s in inputValues)
20       Console.WriteLine("{0,35}({1,-4:F3}) = {2}", fname, s, f(s));
21
22     Console.WriteLine();
23   }
24
25   // DieFromInt: int -> Die
26   public static Die DieFromInt(int i){
27     return new Die(i);
28   }
29
30   // Round: double -> int
31   public static int Round(double d){
32     return (int)(Math.Round(d));
```

```
33   }
34
35   public static void Main(){
36     double[] input = new double[25];
37     for(int i = 0; i < 25; i++)
38       input[i] = (double) (i*2);
39
40     // Compose(DieFromInt, Round): double -> Die
41     // (via int)
42
43     PrintTableOfFunction(Compose<double,int,Die>(DieFromInt, Round),
44                          "Die of double",
45                          input);
46   }
47
48 }
```

Program 43.4 *An example that involves other types than double.*

In line 43 of `Main` we compose the two functions `DieFromInt` and `Round`. They are both programmed explicitly, in line 26 and 31 respectively. The function `Round` maps a `double` to an `int`. The function `DieFromInt` maps an `int` to a `Die`. Thus, `Compose(DieFromInt, Round)` maps a `double` to a `Die`. Notice how we pass the three involved types `double`, `int`, and `Die` as actual type parameters to `Compose` in line 43.

The version of class `Die` used in Program 43.4 can, for instance, be the class shown in Program 12.6. The parameter of the constructor determines the maximum number of eyes of the die.

The output of Program 43.4 is shown in Listing 43.5 (only on web).

43.3. Generic types and methods - Pros and Cons

Lecture 11 - slide 21

In this final section about generic types and methods we will briefly summarize the advantages and disadvantages of generics.

- Advantages
  - Readability and Documentation
    - More precise indication of types.
    - Less downcasting from class Object
  - Type Checking
    - Better and more precise typechecking
  - Efficiency
    - There is a potential for more efficient programs
    - Less casting - fewer boxings
- Disadvantages
  - Complexity
    - Yet another abstraction and parametrization-level on top of the existing

This ends the general discussion of generics. In the lecture about collections, from Chapter 44 to Chapter 48, we will make heavy use of generic types.