В.В. Подбельский

Иллюстрации к курсу лекций по дисциплине «Программирование»

C#_19

Generics

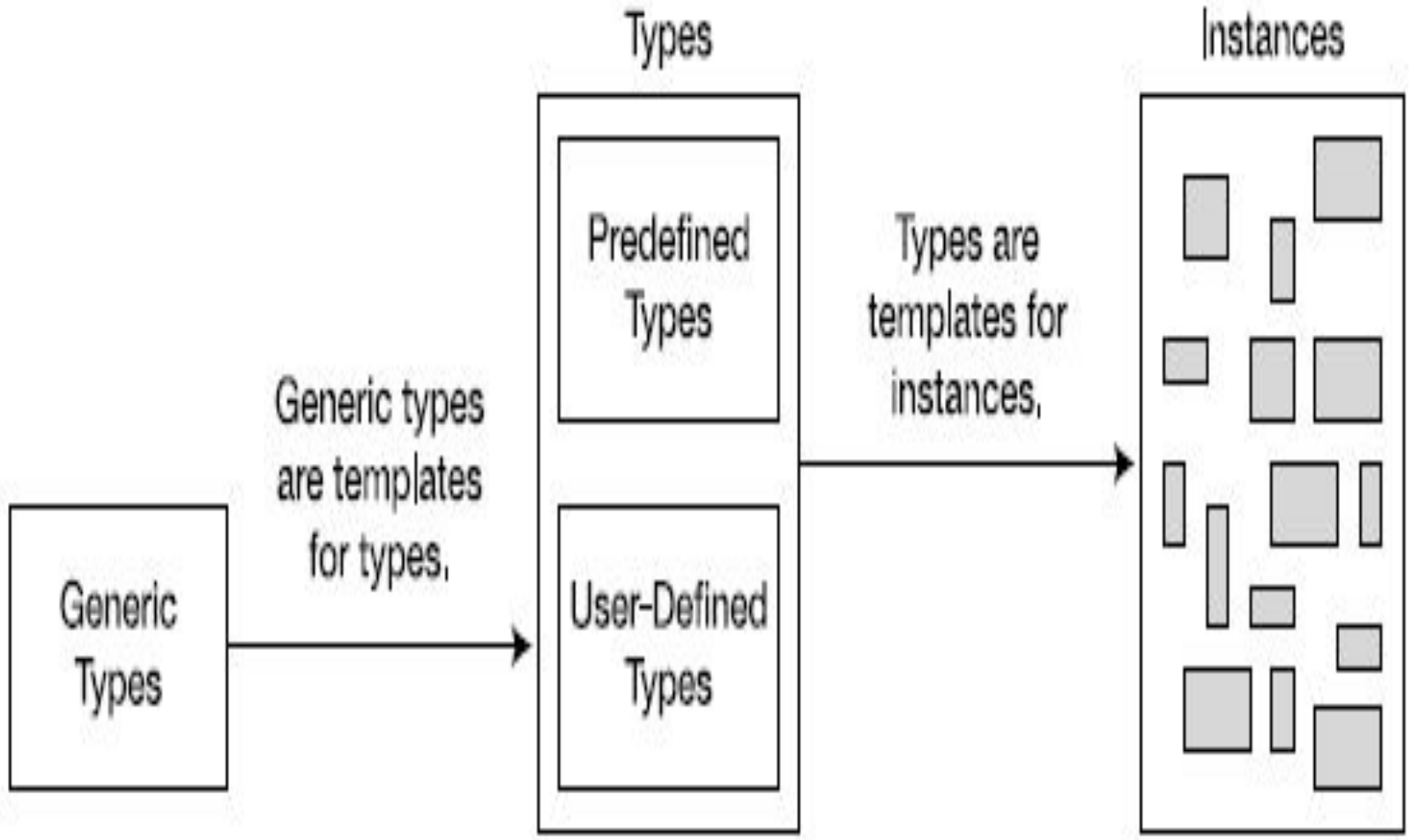
Использованы материалы пособия Daniel Solis, Illustrated C# 2008.

Обобщения

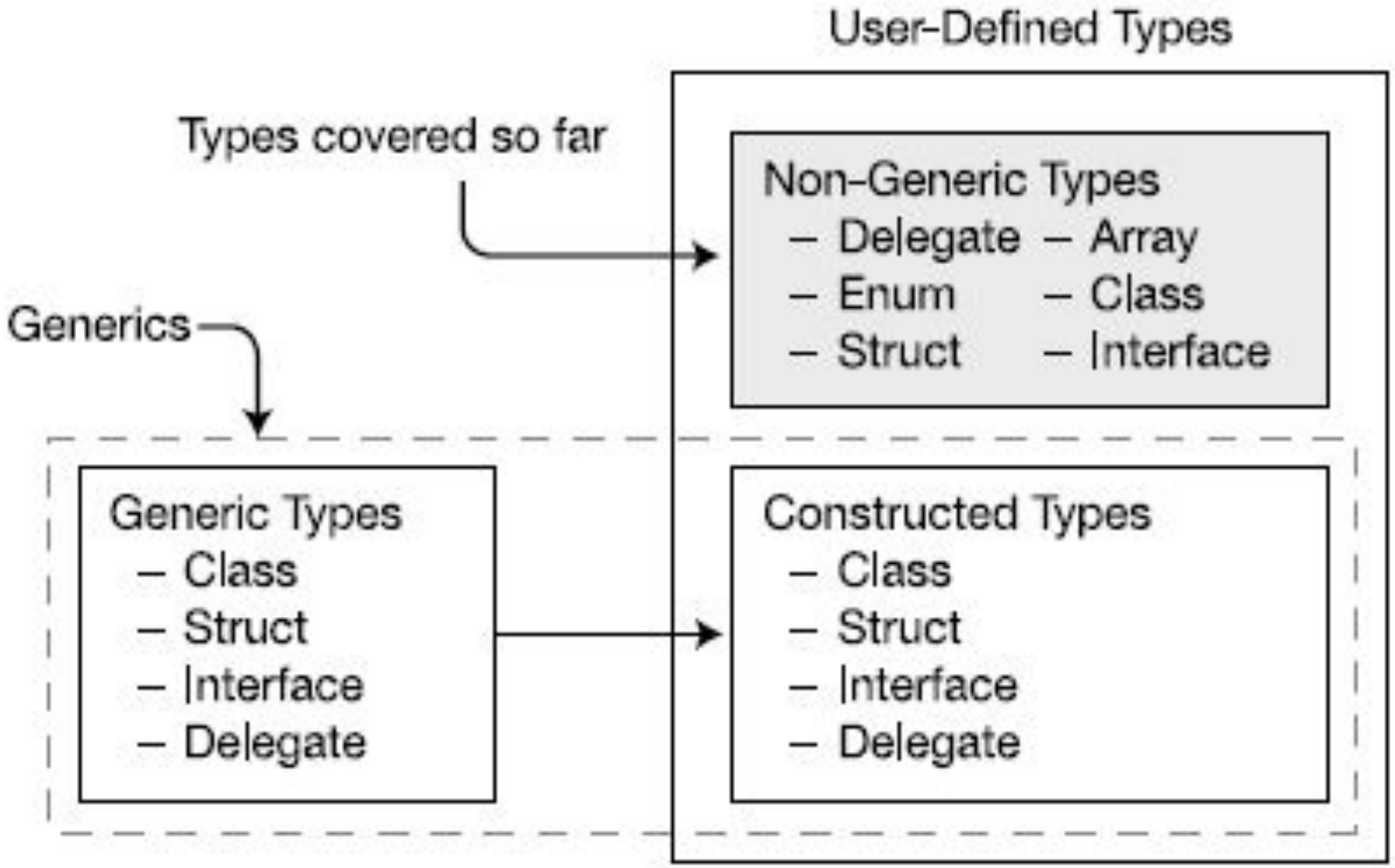
19.1. Стек для чисел

```
class MyFloatStack      // Stack for floats
{
int StackPointer = 0;
float [] StackArray;        // Array of float
  ↑             float
float                   ↓
public void Push( float x )   // Input type: float
{...  }
     float
       ↓
public float Pop()          // Return type: float
{...}
...
}
```

19-2. Generic types are templates for types



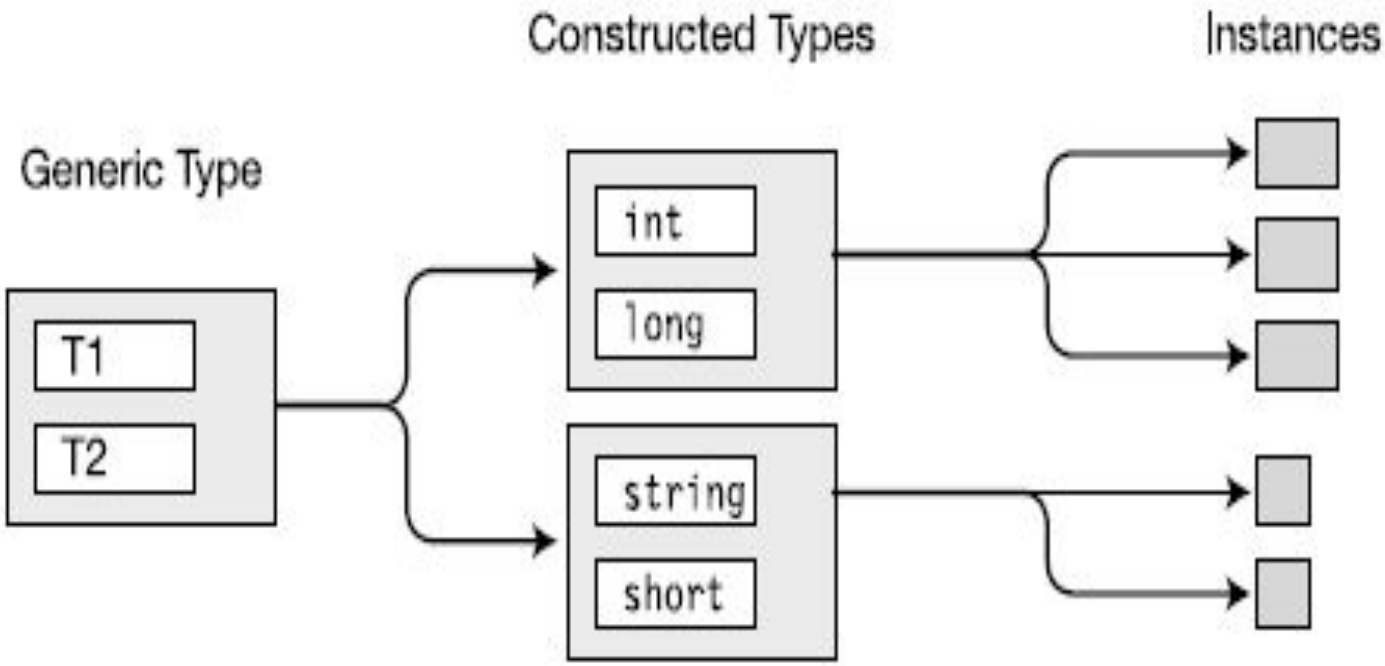
19-3. Generics and user-defined types



19-4. Continuing with the Stack Example

```
class MyStack <T>
{
    int StackPointer = 0;
    T [] StackArray;
    public void Push(T x ) {...}
    public T Pop() {...}
    ...
}
```

19-5. Creating instances from a generic type



① Declare generic type.

② Create constructed types by supplying actual types.

③ Create instances from the constructed types.

19-6. Declaring a Generic Class

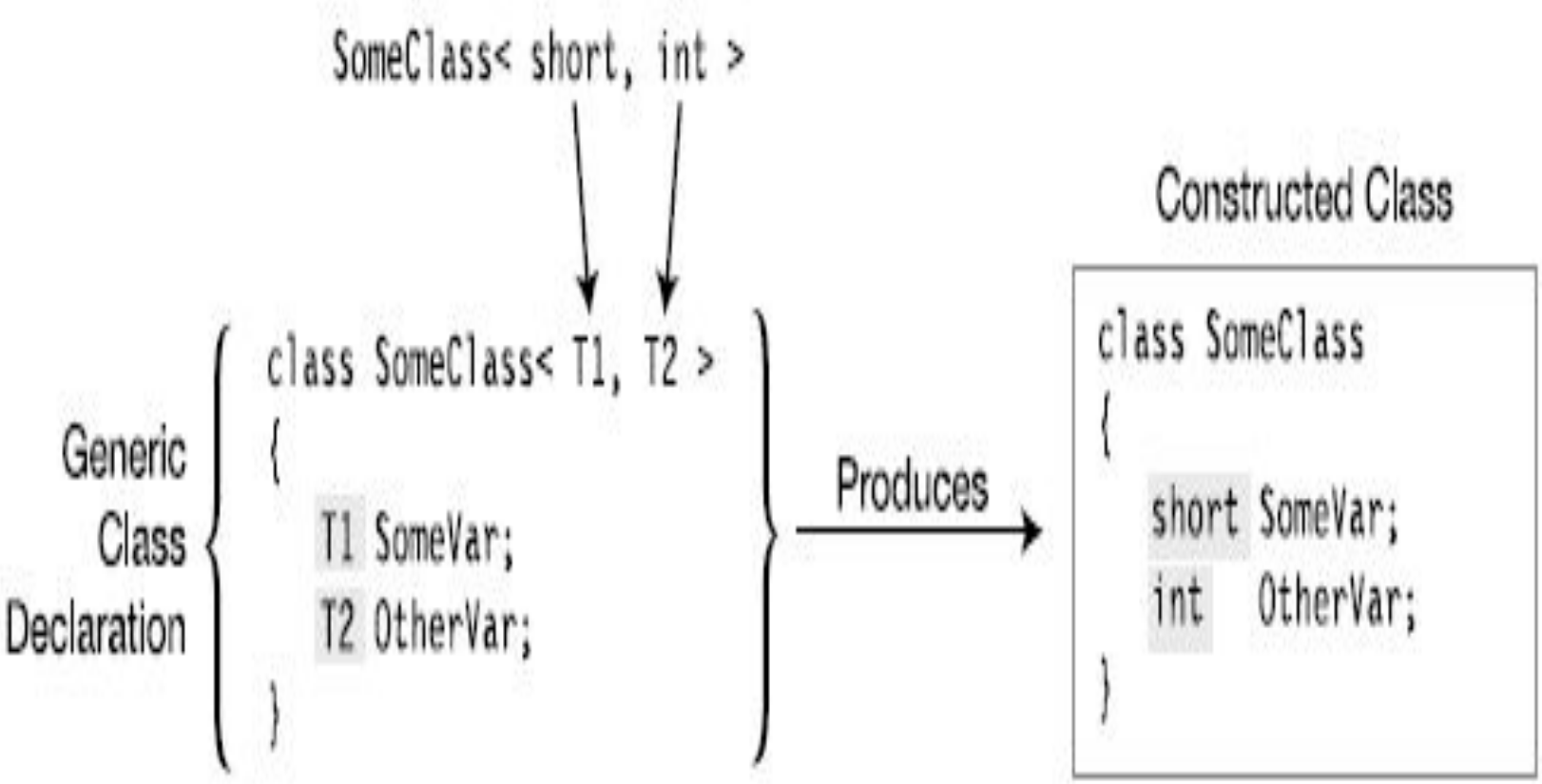
Type parameters

↓

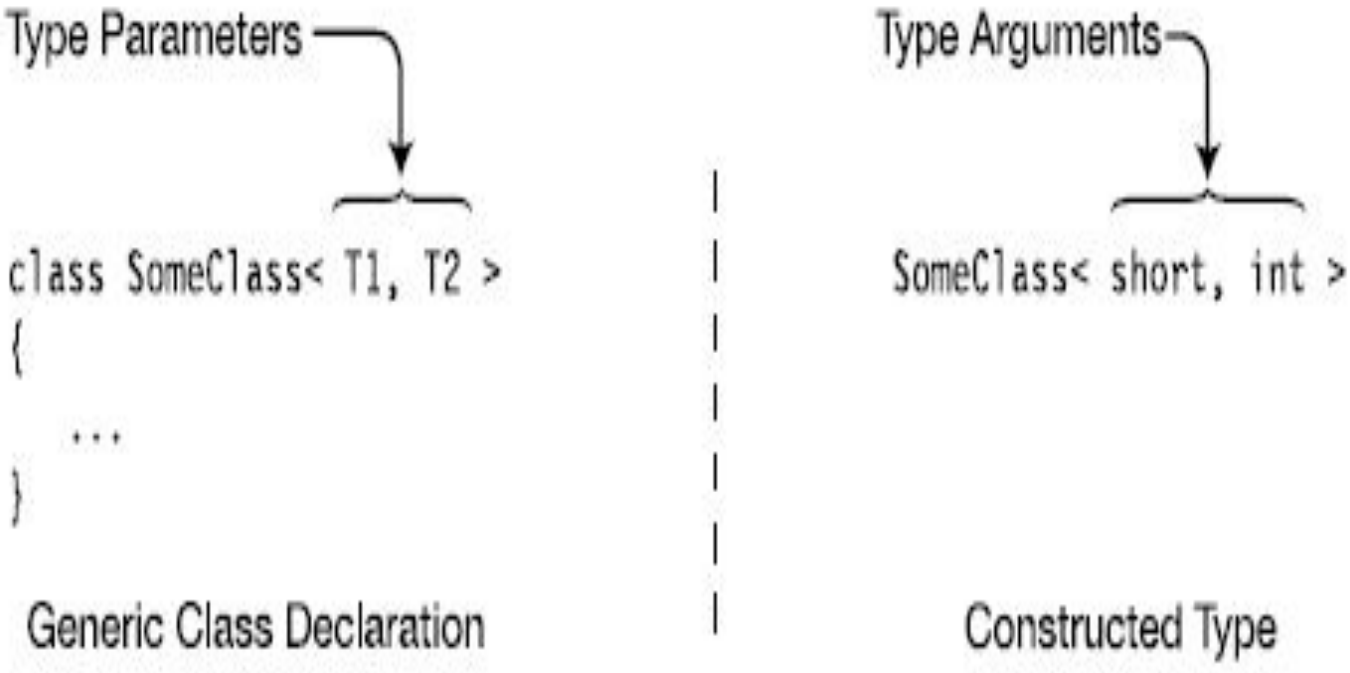
```
class SomeClass < T1, T2 >
{ Normally, types would be used in these positions.
    ↓                   ↓
public T1 SomeVar = new T1();
public T2 OtherVar = new T2();
}       ↑                   ↑
```

Normally, types would be used in these positions.

19-7. Creating a Constructed Type



19-8. Type parameters versus type arguments



19-9. Creating Variables and Instances

```
MyNonGenClass myNGC = new MyNonGenClass ();
Constructed class          Constructed class
      ↓                          ↓
SomeClass<short, int> mySc1 = new SomeClass<short, int>();
var mySc2 = new SomeClass<short, int>();
```

19-10.

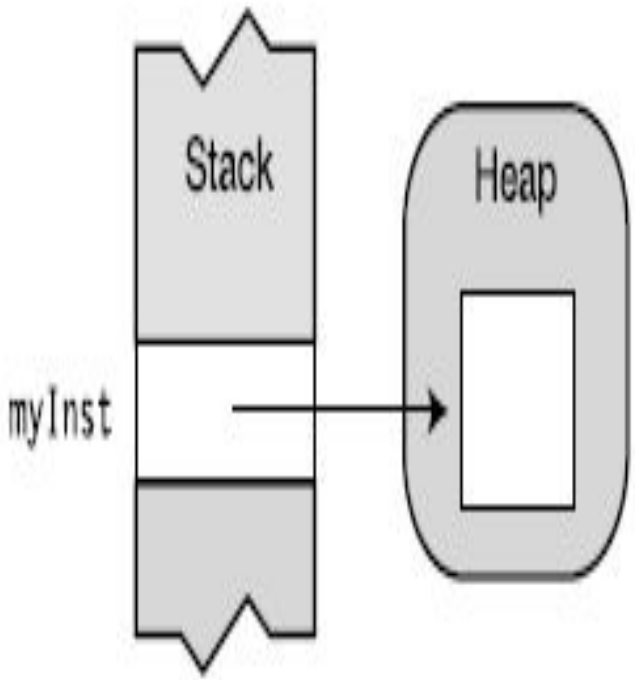
```
class SomeClass< T1, T2 >
{
   public T1 SomeVar;
   public T2 OtherVar;
}
   ...
SomeClass< short, int > myInst;

myInst = new SomeClass< short, int >( );
```

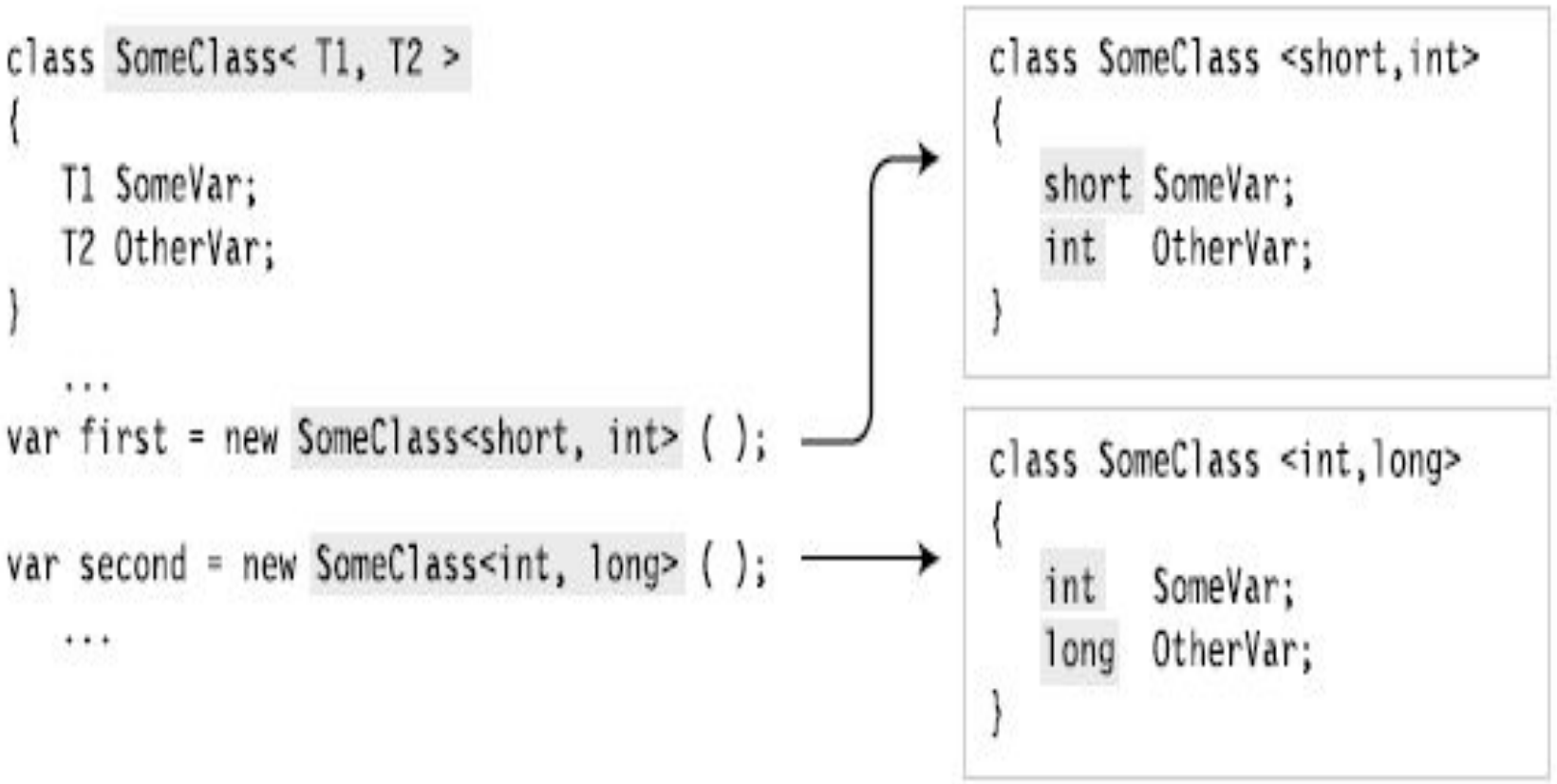
Generic Class Declaration

Allocate class variable

Allocate instance



19-11. *Two constructed classes created from a generic class*



19-12. Constraints on Type Parameters

```
class Simple<T>
{
static public bool LessThan(T i1, T i2)
{
return i1 < i2; // Error
}
...
}
```

19-13. Where Clauses

Type parameter ↓ Constraint list ↓

where *TypeParam* : *constraint*, *constraint*, ...

↑ Colon

nbounded ↓ ↓ With constraints

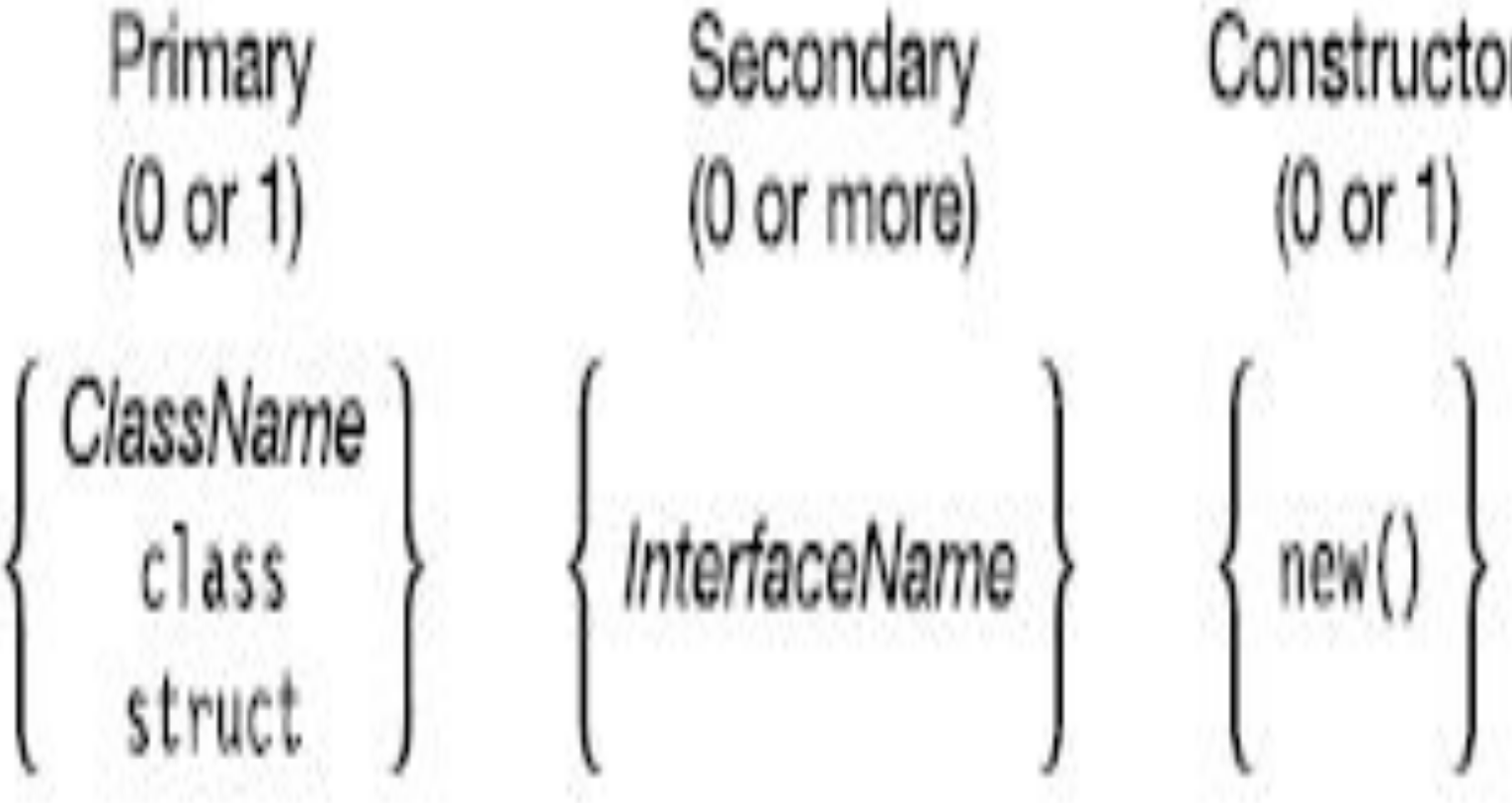
No separators

```
class MyClass < T1, T2, T3 >    ↓
    where T2: Customer      // Constraint for T2
    where T3: Icomparable // Constraint for T3
{                                ↑
...                        No separators
}
```

19-14. Constraint Types and Order

Constraint Type	Description
ClassName	Only classes of this type, or classes derived from it, can be used as the type argument.
class	Any reference type, including classes, arrays, delegates, and interfaces, can be used as the type argument.
struct	Any value type can be used as the type argument.
InterfaceName	Only this interface, or types that implement this interface, can be used as the type argument.
new()	Any type with a parameterless public constructor can be used as the type argument. This is called the *constructor constraint*.

19-15. ***If a type parameter has multiple constraints, they must be in this order.***



19-16. Правила задания ограничений

```
class SortedList<S>
    where S: IComparable<S> { ... }
class LinkedList<M,N>
    where M : IComparable<M>
    where N : ICloneable { ... }
class MyDictionary<KeyType, ValueType>
    where KeyType : IEnumerable,
            new() { ... }
```

19-17. Generic Structs

```
struct PieceOfData<T> // Generic struct
{
    public PieceOfData(T value) { _Data = value; }
    private T _Data;
    public T Data
    {
    get { return _Data; }
    set { _Data = value; }
    }
}
```

19-18. Generic Structs

```
class Program    {
static void Main()     Constructed type
{                  ↓
var intData = new PieceOfData<int>(10);
var stringData = new PieceOfData<string>("Hi there.");
                   ↑
               Constructed type
Console.WriteLine("intData = {0}", intData.Data);
Console.WriteLine("stringData = {0}", stringData.Data);
}
}
```

19-19. Generic Interfaces

Type parameter

↓

```
interface IMyIfc<T>            // Generic interface
{   T ReturnIt(T inValue);     }
```

Type parameter Generic interface

↓ ↓

```
class Simple<S> : IMyIfc<S>   // Generic class
{
public S ReturnIt(S inValue)     // Implement interface
{ return inValue; }
}
```

19-20. Generic Interfaces

```
class Program{
static void Main(){
    var trivInt = new Simple<int>();
    var trivString = new Simple<string>();
    Console.WriteLine("{0}", trivInt.ReturnIt(5));
    Console.WriteLine("{0}", trivString.ReturnIt("Hi there."));
}
}
```

This code produces the following output:

5

Hi there.

19-21. Using Generic Interfaces

```
interface IMyIfc<T>                    // Generic interface
{   T ReturnIt(T inValue);     }
```

Two different interfaces from the same generic interface

↓ ↓

```
class Simple : IMyIfc<int>, IMyIfc<string> // Non-generic class
{
public int ReturnIt(int inValue) /          / Implement int interface
{ return inValue; }
public string ReturnIt(string inValue) // Implement string interface
{ return inValue; }
}
```

19-22. Using Generic Interfaces

```
class Program    {
static void Main()     {
    Simple trivInt = new Simple();
    Simple trivString = new Simple();
    Console.WriteLine("{0}", trivInt.ReturnIt(5));
    Console.WriteLine("{0}", trivString.ReturnIt("Hi there."));
}
}
```

This code produces the following output:

```
5
Hi there.
```

19-23. Duplicate interface

```
interface IMyIfc<T> {    T ReturnIt(T inValue);   }

class Simple<S> : IMyIfc<int>, IMyIfc<S>   {// Error!
public int ReturnIt(int inValue)        // Implement first interface.
   {return inValue;   }
public S ReturnIt(S inValue)     // Implement second interface,
   {return inValue;   }    // but if it's int, it would be
                           // the same as the one above.
}
```

19-24. Generic Delegates

Type parameters

↓

```
delegate R MyDelegate<T, R>( T value );
```

↑ Return type

↑ Delegate formal parameter

19-25. Generic delegate and matched delegate methods

```
delegate void MyDelegate<T>(T value);  // Generic delegate

class Simple {
static public void PrintString(string s)
    {   Console.WriteLine(s);}
static public void PrintUpperString(string s)
    {   Console.WriteLine("{0}", s.ToUpper());   }
}
```

19-26. Generic Delegate

```
class Program    {
static void Main( )    {
var myDel =                    // Create inst of delegate
     new MyDelegate<string>(Simple.PrintString);
    myDel += Simple.PrintUpperString;     // Add a method.
    myDel("Hi There."); // Call delegate
}
}
```

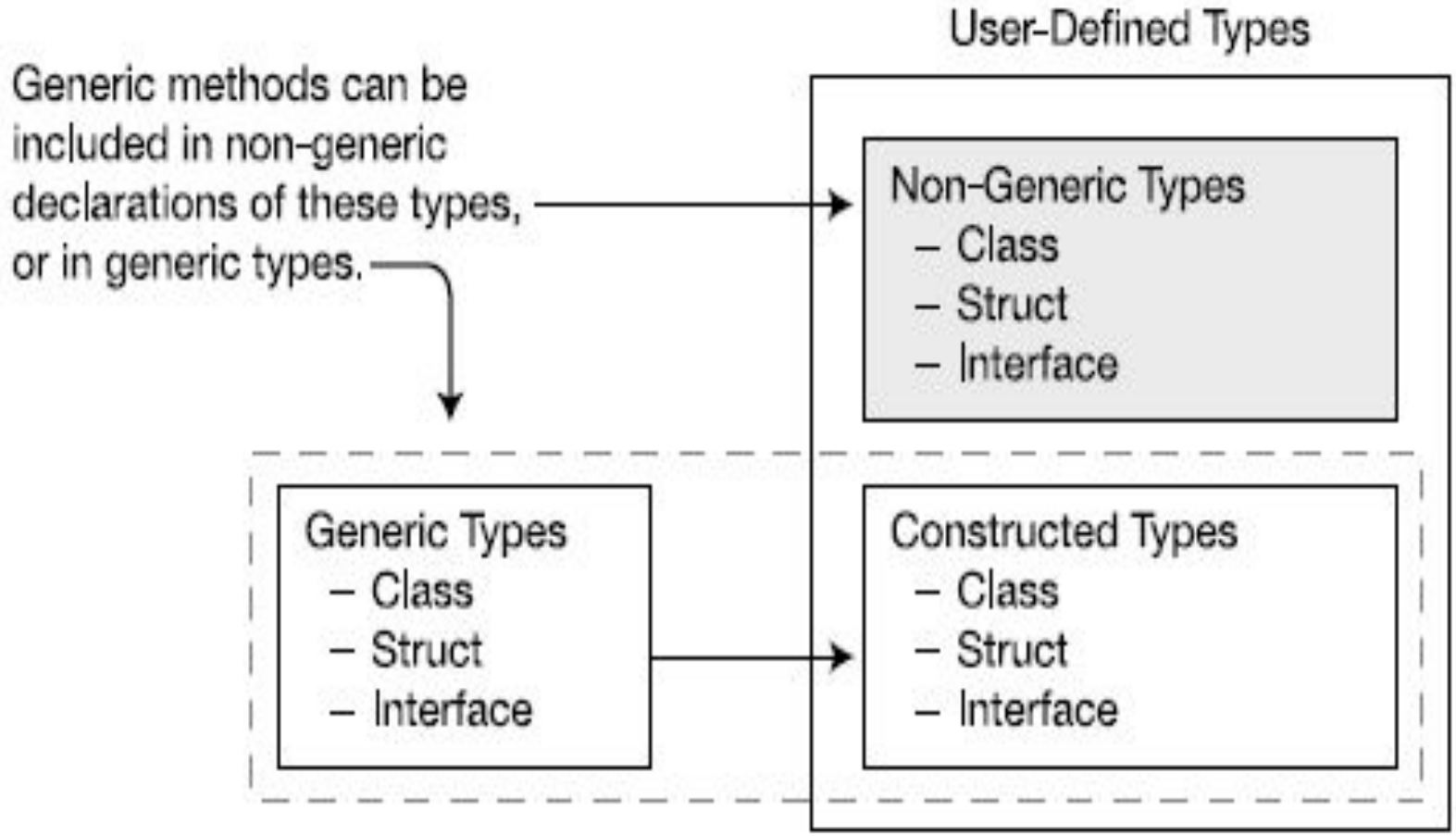
This code produces the following output:

```
Hi There.
HI THERE.
```

19-27. Generic Delegate

```
public delegate TR Func<T1, T2, TR>(T1 p1, T2 p2);
class Simple {
    static public string PrintString(int p1, int p2)
    {int total = p1 + p2;    return total.ToString();   }
    }
class Program    {
static void Main()     {
var myDel = new Func<int, int, string>(Simple.PrintString);
Console.WriteLine("Total: {0}", myDel(15, 13));
}
}
```

19-28. Generic Methods



19-29. Declaring a Generic Method

Type parameter list Constraint clauses

↓ ↓

```
public void PrintData<S, T> ( S p ) where S: Person
{
    ….
}
```

↑

Method parameter list

19-30. Invoking a Generic Method

Объявление обобщенного метода:

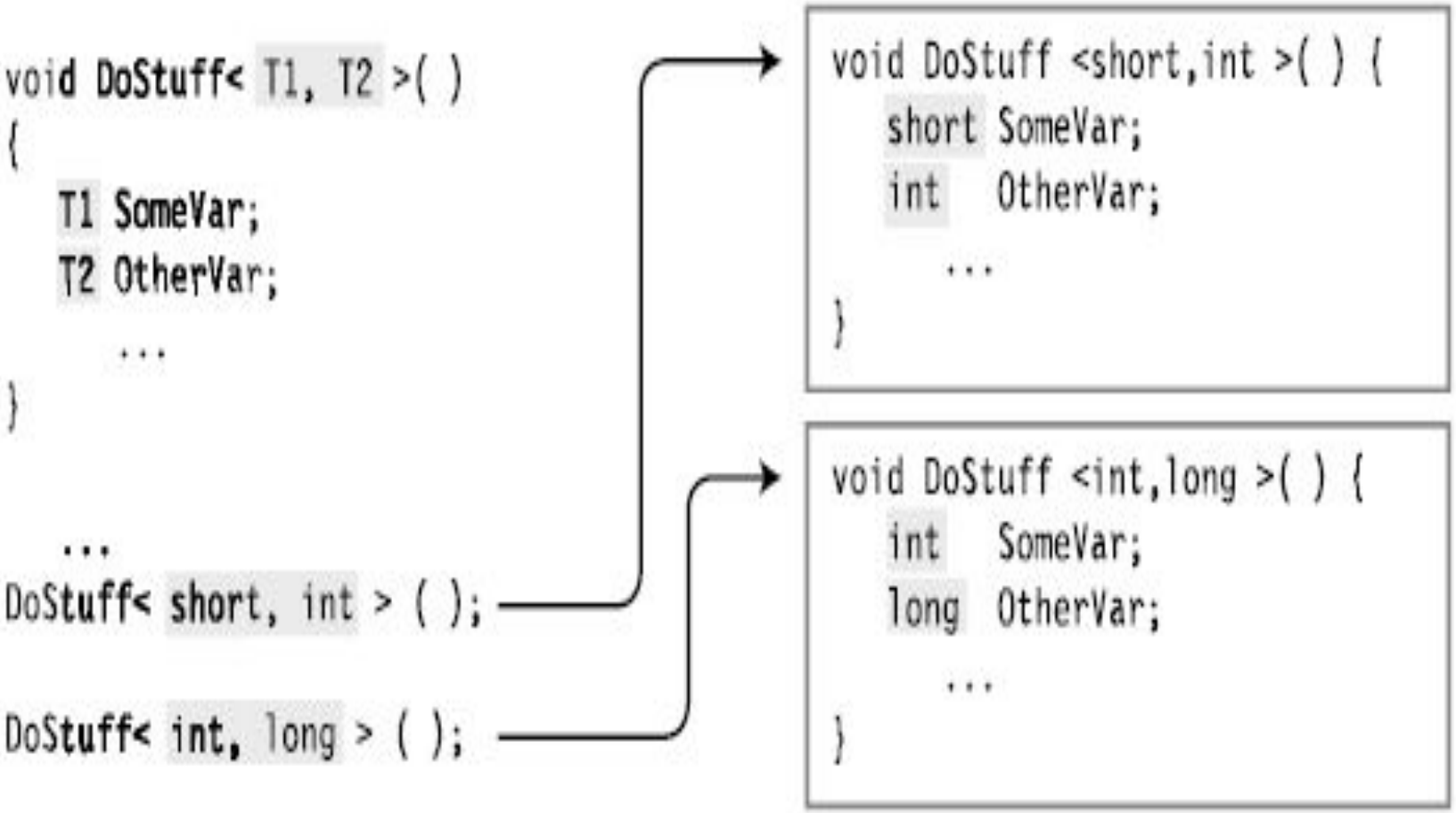
```
void MyMethod<T1, T2>()  {
    T1 someVar;
    T2 otherVar;
    …
}
```

Type arguments

↓

```
MyMethod<short, int>();
MyMethod<int, long >();
```

19-31. A generic method with two instantiations



19-32. Inferring Types

```
public void MyMethod <T> (T myVal) { ... }
```

↑ ↑

Both are of type T

```
int MyInt = 5;
MyMethod <int> (MyInt);
```

↑ ↑

Both are ints

MyMethod(MyInt);

19-33. Example of a Generic Method

```
class Simple {     // Non-generic class
static public void ReverseAndPrint<T>(T[] arr) // Generic method
{
Array.Reverse(arr);
foreach (T item in arr) // Use type argument T.
Console.Write("{0}, ", item.ToString());
Console.WriteLine("");
}
}
```

19-34. Example of a Generic Method

```
class Program    {
static void Main()     {
    var intArray = new int[] { 3, 5, 7, 9, 11 };
    var stringArray = new string[] { "first", "second", "third" };
    var doubleArray = new double[] { 3.567, 7.891, 2.345 };
    Simple.ReverseAndPrint<int>(intArray); // Invoke method
    Simple.ReverseAndPrint(intArray); // Infer type and invoke
    Simple.ReverseAndPrint<string>(stringArray);
    Simple.ReverseAndPrint(stringArray);
    Simple.ReverseAndPrint<double>(doubleArray);
    Simple.ReverseAndPrint(doubleArray);
}
    }
```

19-35. Extension Methods with Generic Classes

```
static class ExtendHolder   {
public static void Print<T>(this Holder<T> h)      {
 T[] vals = h.GetValues();
 Console.WriteLine("{0},\t{1},\t{2}", vals[0], vals[1], vals[2]);   }
}
class Holder<T> {
T[] Vals = new T[3];
public Holder(T v0, T v1, T v2)
{ Vals[0] = v0; Vals[1] = v1; Vals[2] = v2; }
public T[] GetValues() { return Vals; }
}
```

19-36. Example

```
class Program    {
static void Main(string[] args) {
    var intHolder = new Holder<int>(3, 5, 7);
    var stringHolder = new Holder<string>("a1", "b2", "c3");
    intHolder.Print();
    stringHolder.Print();
    }
}
```

This code produces the following output:

3, 5, 7

a1, b2, c3

19-37.