20. Generics

20.1 Generic Class Declarations

A generic class declaration is a declaration of a class that requires type parameters to be supplied in order to form actual types.

A class declaration may optionally define type parameters.

class-declaration:
attributes$_{opt}$ *class-modifiers*$_{opt}$ `class` *identifier* *type-parameter-list*$_{opt}$ *class-base*$_{opt}$
type-parameter-constraints-clauses$_{opt}$ *class-body* `;`$_{opt}$

A class declaration may not supply *type-parameter-constraints-clauses* (§20.7) unless it also supplies a *type-parameter-list*.

A class declaration that supplies a *type-parameter-list* is a generic class declaration. Additionally, any class nested inside a generic class declaration or a generic struct declaration is itself a generic class declaration because type parameters for the containing type must be supplied to create a constructed type.

Generic class declarations follow the same rules as normal class declarations except where noted and particularly with regard to naming, nesting, and the permitted access controls. Generic class declarations may be nested inside nongeneric class declarations.

A generic class is referenced using a ***constructed type*** (§20.5). Given the generic class declaration

```
class List<T> {}
```

some examples of constructed types are `List<T>`, `List<int>` and `List<List<string>>`. A constructed type that uses one or more type parameters, such as `List<T>`, is called an ***open constructed type***. A constructed type that uses no type parameters, such as `List<int>`, is called a ***closed constructed type***.

Generic types may not be "overloaded"; that is, the identifier of a generic type must be uniquely named within a scope in the same way as ordinary types.

```
class C {}
class C<V> {}          // Error, C defined twice
class C<U,V> {}        // Error, C defined twice
```

However, the type lookup rules used during unqualified type name lookup (§20.9.3) and member access (§20.9.4) do take the number of type parameters into account.

20.1.1 Type Parameters

Type parameters may be supplied on a class declaration. Each type parameter is a simple identifier that denotes a placeholder for a type argument that is supplied to create a constructed type. A type parameter is a formal placeholder for a type that will be supplied later. By contrast, a type argument (§20.5.1) is the actual type that is substituted for the type parameter when a constructed type is referenced.

type-parameter-list:
    < *type-parameters* >

type-parameters:
    type-parameter
    type-parameters , *type-parameter*

type-parameter:
    attributes$_{opt}$ *identifier*

Each type parameter in a class declaration defines a name in the declaration space (§3.3) of that class. Thus, it cannot have the same name as another type parameter or a member declared in that class. A type parameter cannot have the same name as the type itself.

The scope (§3.7) of a type parameter on a class includes the *class-base*, *type-parameter-constraints-clauses*, and *class-body*. Unlike members of a class, it does not extend to derived classes. Within its scope, a type parameter can be used as a type.

type:
    value-type
    reference-type
    type-parameter

Because a type parameter can be instantiated with many different actual type arguments, type parameters have slightly different operations and restrictions than other types. These include the following.

- A type parameter cannot be used directly to declare a base class or interface (§20.1.3).
- The rules for member lookup on type parameters depend on the constraints, if any, applied to the type parameter. They are detailed in §20.7.2.

- The available conversions for a type parameter depend on the constraints, if any, applied to the type parameter. They are detailed in §20.7.4.
- The literal `null` cannot be converted to a type given by a type parameter, except if the type parameter is constrained by a class constraint (§20.7.4). However, a default value expression (§20.8.1) can be used instead. In addition, a value with a type given by a type parameter *can* be compared with `null` using `==` and `!=` (§20.8.4).
- A `new` expression (§20.8.2) can only be used with a type parameter if the type parameter is constrained by a *constructor-constraint* (§20.7).
- A type parameter cannot be used anywhere within an attribute.
- A type parameter cannot be used in a member access or type name to identify a static member or a nested type (§20.9.1, §20.9.4).
- In unsafe code, a type parameter cannot be used as an *unmanaged-type* (§18.2).

As a type, type parameters are purely a compile-time construct. At runtime, each type parameter is bound to a runtime type that was specified by supplying a type argument to the generic type declaration. Thus, the type of a variable declared with a type parameter will, at runtime, be a closed type (§20.5.2). The runtime execution of all statements and expressions involving type parameters uses the actual type that was supplied as the type argument for that parameter.

20.1.2 The Instance Type

Each class declaration has an associated constructed type, the ***instance type***. For a generic class declaration, the instance type is formed by creating a constructed type (§20.4) from the type declaration, with each of the supplied type arguments being the corresponding type parameter. Because the instance type uses the type parameters, it is only valid where the type parameters are in scope: inside the class declaration. The instance type is the type of `this` for code written inside the class declaration. For nongeneric classes, the instance type is simply the declared class. The following shows several class declarations along with their instance types.

```
class A<T>                          // instance type: A<T>
{
    class B {}                      // instance type: A<T>.B

    class C<U> {}                   // instance type: A<T>.C<U>
}

class D {}                          // instance type: D
```

20.1.3 Base Specification

The base class specified in a class declaration may be a constructed class type (§20.5). A base class may not be a type parameter on its own, but it may involve the type parameters that are in scope.

```
class Extend<V>: V {}          // Error, type parameter used as base class
```

A generic class declaration may not use `System.Attribute` as a direct or indirect base class.

The base interfaces specified in a class declaration may be constructed interface types (§20.5). A base interface may not be a type parameter on its own, but it may involve the type parameters that are in scope. The following code illustrates how a class can implement and extend constructed types.

```
class C<U,V> {}

interface I1<V> {}

class D: C<string,int>, I1<string> {}

class E<T>: C<int,T>, I1<T> {}
```

The base interfaces of a generic class declaration must satisfy the uniqueness rule described in §20.3.1.

Methods in a class that override or implement methods from a base class or interface must provide appropriate methods of specialized types. The following code illustrates how methods are overridden and implemented. This is explained further in §20.1.10.

```
class C<U,V>
{
    public virtual void M1(U x, List<V> y) {...}
}

interface I1<V>
{
    V M2(V x);
}

class D: C<string,int>, I1<string>
{
    public override void M1(string x, List<int> y) {...}

    public string M2(string x) {...}
}
```

20.1.4 Members of Generic Classes

All members of a generic class may use type parameters from any enclosing class, either directly or as part of a constructed type. When a particular closed constructed type (§20.5.2) is used at runtime, each use of a type parameter is replaced with the actual type argument supplied to the constructed type. For example

```
class C<V>
{
    public V f1;
    public C<V> f2 = null;
```

```
        public C(V x) {
        this.f1 = x;
        this.f2 = this;
    }
}

class Application
{
    static void Main() {
        C<int> x1 = new C<int>(1);
        Console.WriteLine(x1.f1);            // Prints 1

        C<double> x2 = new C<double>(3.1415);
        Console.WriteLine(x2.f1);            // Prints 3.1415
    }
}
```

Within instance function members, the type of `this` is the instance type (§20.1.2) of the declaration.

Apart from using type parameters as types, members in generic class declarations follow the same rules as members of nongeneric classes. Additional rules that apply to particular kinds of members are discussed in the following sections.

20.1.5 Static Fields in Generic Classes

A static variable in a generic class declaration is shared amongst all instances of the same closed constructed type (§20.5.2) but is not shared amongst instances of different closed constructed types. These rules apply regardless of whether the type of the static variable involves any type parameters.

For example

```
class C<V>
{
    static int count = 0;

    public C() {
        count++;
    }

    public static int Count {
        get { return count; }
    }
}


class Application
{
    static void Main() {
        C<int> x1 = new C<int>();
        Console.WriteLine(C<int>.Count);     // Prints 1
```

```
        C<double> x2 = new C<double>();
        Console.WriteLine(C<int>.Count);  // Prints 1

        C<int> x3 = new C<int>();
        Console.WriteLine(C<int>.Count);  // Prints 2
    }
}
```

20.1.6 Static Constructors in Generic Classes

Static constructors in generic classes are used to initialize static fields and perform other initialization for each different closed constructed type that is created from a particular generic class declaration. The type parameters of the generic type declaration are in scope and can be used within the body of the static constructor.

A new closed constructed class type is initialized the first time that either:

- An instance of the closed constructed type is created
- Any of the static members of the closed constructed type are referenced

To initialize a new closed constructed class type, first a new set of static fields (§20.1.5) for that particular closed constructed type is created. Each of the static fields is initialized to its default value (§5.2). Next, the static field initializers (§10.4.5.1) are executed for those static fields. Finally, the static constructor is executed.

Because the static constructor is executed exactly once for each closed constructed class type, it is a convenient place to enforce runtime checks on the type parameter that cannot be checked at compile time via constraints (§20.7). For example, the following type uses a static constructor to enforce that the type parameter is a reference type.

```
class Gen<T>
{
    static Gen() {
        if ((object)T.default != null) {
            throw new ArgumentException("T must be a reference type");
        }
    }
}
```

20.1.7 Accessing Protected Members

Within a generic class declaration, access to inherited protected instance members is permitted through an instance of any class type constructed from the generic class. Specifically, the rules for accessing `protected` and `protected internal` instance members specified in §3.5.3 are augmented with the following rule for generics.

- Within a generic class `G`, access to an inherited protected instance member `M` using a *primary-expression* of the form `E.M` is permitted if the type of `E` is a class type constructed from `G` or a class type inherited from a class type constructed from `G`.

In the example

```
class C<T>
{
    protected T x;
}

class D<T>: C<T>
{
    static void F() {
        D<T> dt = new D<T>();
        D<int> di = new D<int>();
        D<string> ds = new D<string>();
        dt.x = T.default;
        di.x = 123;
        ds.x = "test";
    }
}
```

the three assignments to x are permitted because they all take place through instances of class types constructed from the generic type.

20.1.8 Overloading in Generic Classes

Methods, constructors, indexers, and operators within a generic class declaration can be overloaded; however, overloading is constrained so that ambiguities cannot occur within constructed classes. Two function members declared with the same names in the same generic class declaration must have parameter types such that no closed constructed type could have two members with the same name and signature. When considering all possible closed constructed types, this rule includes type arguments that do not currently exist in the current program but could be written. Type constraints on the type parameter are ignored for the purpose of this rule.

The following examples show overloads that are valid and invalid according to this rule.

```
interface I1<T> {...}

interface I2<T> {...}

class G1<U>
{
    long F1(U u);              // Invalid overload, G<int> would have two
    int F1(int i);             // members with the same signature

    void F2(U u1, U u2);       // Valid overload, no type argument for U
    void F2(int i, string s); // could be int and string simultaneously

    void F3(I1<U> a);          // Valid overload
    void F3(I2<U> a);

    void F4(U a);              // Valid overload
    void F4(U[] a);
}
```

```
class G2<U,V>
{
    void F5(U u, V v);         // Invalid overload, G2<int,int> would have
    void F5(V v, U u);         // two members with the same signature

    void F6(U u, I1<V> v);     // Invalid overload, G2<I1<int>,int> would
    void F6(I1<V> v, U u);     // have two members with the same signature

    void F7(U u1, I1<V> v2);   // Valid overload, U cannot be V and I1<V>
    void F7(V v1, U u2);       // simultaneously

    void F8(ref U u);          // Invalid overload
    void F8(out V v);
}

class C1 {...}

class C2 {...}

class G3<U,V> where U: C1 where V: C2
{
    void F9(U u);              // Invalid overload, constraints on U and V
    void F9(V v);              // are ignored when checking overloads
}
```

20.1.9 Parameter Array Methods and Type Parameters

Type parameters may be used in the type of a parameter array. For example, given the declaration

```
class C<V>
{
    static void F(int x, int y, params V[] args);
}
```

the following invocations of the expanded form of the method

```
C<int>.F(10, 20);
C<object>.F(10, 20, 30, 40);
C<string>.F(10, 20, "hello", "goodbye");
```

correspond exactly to the following.

```
C<int>.F(10, 20, new int[] {});
C<object>.F(10, 20, new object[] {30, 40});
C<string>.F(10, 20, new string[] {"hello", "goodbye"} );
```

20.1.10 Overriding and Generic Classes

Function members in generic classes can override function members in base classes, as usual. If the base class is a nongeneric type or a closed constructed type, then any overriding function member cannot have constituent types that involve type parameters. However, if the base class is an open constructed type, then an overriding function member can use type parameters in its declaration. When determining the overridden base member, the members of the base classes must be determined by substituting type arguments, as

described in §20.5.4. Once the members of the base classes are determined, the rules for overriding are the same as for nongeneric classes.

The following example demonstrates how the overriding rules work in the presence of generics.

```
abstract class C<T>
{
    public virtual T F() {...}

    public virtual C<T> G() {...}

    public virtual void H(C<T> x) {...}
}

class D: C<string>
{
    public override string F() {...}      // Ok

    public override C<string> G() {...}   // Ok

    public override void H(C<T> x) {...}  // Error, should be C<string>
}

class E<T,U>: C<U>
{
    public override U F() {...}           // Ok

    public override C<U> G() {...}        // Ok

    public override void H(C<T> x) {...}  // Error, should be C<U>
}
```

20.1.11 Operators in Generic Classes

Generic class declarations may define operators, following the same rules as normal class declarations. The instance type (§20.1.2) of the class declaration must be used in the declaration of operators in a manner analogous to the normal rules for operators, as follows.

- A unary operator must take a single parameter of the instance type. The unary ++ and -- operators must return the instance type.
- At least one of the parameters of a binary operator must be of the instance type.
- Either the parameter type or the return type of a conversion operator must be the instance type.

The following shows some examples of valid operator declarations in a generic class.

```
class X<T>
{
    public static X<T> operator ++(X<T> operand) {...}

    public static int operator *(X<T> op1, int op2) {...}

    public static explicit operator X<T>(T value) {...}
}
```

For a conversion operator that converts from a source type S to a target type T, when the rules specified in §10.9.3 are applied, any type parameters associated with S or T are considered to be unique types that have no inheritance relationship with other types, and any constraints on those type parameters are ignored.

In the example

```
class C<T> {...}

class D<T>: C<T>
{
    public static implicit operator C<int>(D<T> value) {...}    // Ok

    public static implicit operator C<T>(D<T> value) {...}      // Error
}
```

the first operator declaration is permitted because, for the purposes of §10.9.3, T and int are considered unique types with no relationship. However, the second operator is an error because C<T> is the base class of D<T>.

Given the previous example, it is possible to declare operators that, for some type arguments, specify conversions that already exist as predefined conversions. In the example

```
struct Nullable<T>
{
    public static implicit operator Nullable<T>(T value) {...}

    public static explicit operator T(Nullable<T> value) {...}
}
```

when type object is specified as a type argument for T, the second operator declares a conversion that already exists (an implicit, and therefore also an explicit, conversion exists from any type to type object).

In cases where a predefined conversion exists between two types, any user-defined conversions between those types are ignored. Specifically

- If a predefined implicit conversion (§6.1) exists from type S to type T, all user-defined conversions (implicit or explicit) from S to T are ignored.
- If a predefined explicit conversion (§6.2) exists from type S to type T, any user-defined explicit conversions from S to T are ignored. However, user-defined implicit conversions from S to T are still considered.

For all types but object, the operators declared by the Nullable<T> type do not conflict with predefined conversions. For example

```
void F(int i, Nullable<int> n) {
    i = n;                          // Error
    i = (int)n;                     // User-defined explicit conversion
    n = i;                          // User-defined implicit conversion
    n = (Nullable<int>)i;           // User-defined implicit conversion
}
```

However, for type `object`, predefined conversions hide the user-defined conversions in all cases but one:

```
void F(object o, Nullable<object> n) {
    o = n;                          // Pre-defined boxing conversion
    o = (object)n;                  // Pre-defined boxing conversion
    n = o;                          // User-defined implicit conversion
    n = (Nullable<object>)o;        // Pre-defined unboxing conversion
}
```

20.1.12 Nested Types in Generic Classes

A generic class declaration can contain nested type declarations. The type parameters of the enclosing class may be used within the nested types. A nested type declaration may contain additional type parameters that apply only to the nested type.

Every type declaration contained within a generic class declaration is implicitly a generic type declaration. When writing a reference to a type nested within a generic type, the containing constructed type, including its type arguments, must be named. However, from within the outer class, the inner type can be used without qualification; the instance type of the outer class can be implicitly used when constructing the inner type. The following example shows three different correct ways to refer to a constructed type created from `Inner`; the first two are equivalent.

```
class Outer<T>
{
    class Inner<U>
    {
        static void F(T t, U u) {...}
    }

    static void F(T t) {
        Outer<T>.Inner<string>.F(t, "abc");   // These two statements have
        Inner<string>.F(t, "abc");            // the same effect

        Outer<int>.Inner<string>.F(3, "abc"); // This type is different

        Outer.Inner<string>.F(t, "abc");      // Error, Outer needs type arg
    }
}
```

Although it is bad programming style, the type parameters in a nested type can hide a member or type parameter declared in the outer type.

```
class Outer<T>
{
    class Inner<T>     // Valid, hides Outer's T
    {
        public T t;    // Refers to Inner's T
    }
}
```

20.1.13 Application Entry Point

The application entry point method (§3.1) may not be in a generic class declaration.

20.2 Generic Struct Declarations

Like a class declaration, a `struct` declaration may optionally define type parameters.

struct-declaration:
    attributes$_{opt}$ *struct-modifiers*$_{opt}$ `struct` *identifier* *type-parameter-list*$_{opt}$ *struct-interfaces*$_{opt}$
        type-parameter-constraints-clauses$_{opt}$ *struct-body* `;`$_{opt}$

The rules for generic class declarations apply equally to generic struct declarations, except where the exceptions noted in §11.3 for *struct-declaration*s apply.

20.3 Generic Interface Declarations

Interfaces may also optionally define type parameters.

interface-declaration:
    attributes$_{opt}$ *interface-modifiers*$_{opt}$ `interface` *identifier* *type-parameter-list*$_{opt}$
        interface-base$_{opt}$ *type-parameter-constraints-clauses*$_{opt}$ *interface-body* `;`$_{opt}$

An interface that is declared with type parameters is a generic interface declaration. Except where noted, generic interface declarations follow the same rules as normal interface declarations.

Each type parameter in an interface declaration defines a name in the declaration space (§3.3) of that interface. The scope (§3.7) of a type parameter on an interface includes the *interface-base*, *type-parameter-constraints-clauses*, and *interface-body*. Within its scope, a type parameter can be used as a type. The same restrictions apply to type parameters on interfaces as apply to type parameters on classes (§20.1.1).

Methods within generic interfaces are subject to the same overload rules as methods within generic classes (§20.1.8).

20.3.1 Uniqueness of Implemented Interfaces

The interfaces implemented by a generic type declaration must remain unique for all possible constructed types. Without this rule, it would be impossible to determine the correct method to call for certain constructed types. For example, suppose a generic class declaration were permitted to be written as follows.

```
interface I<T>
{
    void F();
}

class X<U,V>: I<U>, I<V>                    // Error: I<U> and I<V> conflict
{
    void I<U>.F() {...}
    void I<V>.F() {...}
}
```

Were this permitted, it would be impossible to determine which code to execute in the following case.

```
I<int> x = new X<int,int>();
x.F();
```

To determine if the interface list of a generic type declaration is valid, the following steps are performed.

- Let `L` be the list of interfaces directly specified in a generic class, struct, or interface declaration `C`.
- Add to `L` any base interfaces of the interfaces already in `L`.
- Remove any duplicates from `L`.
- If any possible constructed type created from `C` would, after type arguments are substituted into `L`, cause two interfaces in `L` to be identical, then the declaration of `C` is invalid. Constraint declarations are not considered when determining all possible constructed types.

In the class declaration `X` above, the interface list `L` consists of `I<U>` and `I<V>`. The declaration is invalid because any constructed type with `U` and `V` being the same type would cause these two interfaces to be identical types.

20.3.2 Explicit Interface Member Implementations

Explicit interface member implementations work with constructed interface types in essentially the same way as with simple interface types. As usual, an explicit interface member implementation must be qualified by an *interface-type*, indicating which interface is being implemented. This type may be a simple interface or a constructed interface, as in the following example.

```
interface IList<T>
{
    T[] GetElements();
}

interface IDictionary<K,V>
{
    V this[K key];

    void Add(K key, V value);
}

class List<T>: IList<T>, IDictionary<int,T>
{
    T[] IList<T>.GetElements() {...}

    T IDictionary<int,T>.this[int index] {...}

    void IDictionary<int,T>.Add(int index, T value) {...}
}
```

20.4 Generic Delegate Declarations

A delegate declaration may include type parameters.

delegate-declaration:
attributes$_{opt}$ *delegate-modifiers*$_{opt}$ `delegate` *return-type identifier type-parameter-list*$_{opt}$
`(` *formal-parameter-list*$_{opt}$ `)` *type-parameter-constraints-clauses*$_{opt}$ `;`

A delegate that is declared with type parameters is a generic delegate declaration. A delegate declaration may not supply *type-parameter-constraints-clauses* (§20.7) unless it also supplies a *type-parameter-list*. Generic delegate declarations follow the same rules as normal delegate declarations, except where noted. Each type parameter in a generic delegate declaration defines a name in a special declaration space (§3.3) that is associated with that delegate declaration. The scope (§3.7) of a type parameter on a delegate declaration includes the *return-type*, *formal-parameter-list*, and *type-parameter-constraints-clauses*.

Like other generic type declarations, type arguments must be given to form a constructed delegate type. The parameter types and return type of a constructed delegate type are formed by substituting, for each type parameter in the delegate declaration, the corresponding type argument of the constructed delegate type. The resulting return type and parameter types are used for determining what methods are compatible (§15.1) with a constructed delegate type. For example

```
delegate bool Predicate<T>(T value);

class X
{
    static bool F(int i) {...}
```

```
    static bool G(string s) {...}

    static void Main() {
        Predicate<int> p1 = F;
        Predicate<string> p2 = G;
    }
}
```

Note that the two assignments in the previous `Main` method are equivalent to the following longer form.

```
static void Main() {
    Predicate<int> p1 = new Predicate<int>(F);
    Predicate<string> p2 = new Predicate<string>(G);
}
```

The shorter form is permitted because of method group conversions, which are described in §21.9.

20.5 Constructed Types

A generic type declaration, by itself, does not denote a type. Instead, a generic type declaration is used as a "blueprint" to form many different types, by way of applying ***type arguments***. The type arguments are written within angle brackets (< and >) immediately following the name of the generic type declaration. A type that is named with at least one type argument is called a ***constructed type***. A constructed type can be used in most places in the language that a type name can appear.

type-name:
    namespace-or-type-name

namespace-or-type-name:
    identifier type-argument-list$_{opt}$
    namespace-or-type-name . *identifier type-argument-list*$_{opt}$

Constructed types can also be used in expressions as simple names (§20.9.3) or when accessing a member (§20.9.4).

When a *namespace-or-type-name* is evaluated, only generic types with the correct number of type parameters are considered. Thus, it is possible to use the same identifier to identify different types as long as the types have different numbers of type parameters and are declared in different namespaces. This is useful when mixing generic and nongeneric classes in the same program.

```
namespace System.Collections
{
    class Queue {...}
}
```

```
namespace System.Collections.Generic
{
    class Queue<ElementType> {...}
}

namespace MyApplication
{
    using System.Collections;
    using System.Collections.Generic;

    class X
    {
        Queue q1;          // System.Collections.Queue
        Queue<int> q2;     // System.Collections.Generic.Queue
    }
}
```

The detailed rules for name lookup in these productions is described in §20.9. The resolution of ambiguities in these productions is described in §20.6.5.

A *type-name* might identify a constructed type even though it does not specify type parameters directly. This can occur where a type is nested within a generic class declaration, and the instance type of the containing declaration is implicitly used for name lookup (§20.1.12).

```
class Outer<T>
{
    public class Inner {...}

    public Inner i;        // Type of i is Outer<T>.Inner
}
```

In unsafe code, a constructed type cannot be used as an *unmanaged-type* (§18.2).

20.5.1 Type Arguments

Each argument in a type argument list is simply a *type*.

type-argument-list:
    < *type-arguments* >

type-arguments:
    type-argument
    type-arguments , *type-argument*

type-argument:
    type

Type arguments may in turn be constructed types or type parameters. In unsafe code (§18), a *type-argument* may not be a pointer type. Each type argument must satisfy any constraints on the corresponding type parameter (§20.7.1).

20.5.2 Open and Closed Types

All types can be classified as either ***open types*** or ***closed types***. An open type is a type that involves type parameters. More specifically

- A type parameter defines an open type.
- An array type is an open type if and only if its element type is an open type.
- A constructed type is an open type if and only if one or more of its type arguments is an open type.

A closed type is a type that is not an open type.

At runtime, all of the code within a generic type declaration is executed in the context of a closed constructed type that was created by applying type arguments to the generic declaration. Each type parameter within the generic class is bound to a particular runtime type. The runtime processing of all statements and expressions always occurs with closed types, and open types occur only during compile-time processing.

Each closed constructed type has its own set of static variables, which are not shared with any other closed constructed types. Because an open type does not exist at runtime, there are no static variables associated with an open type. Two closed constructed types are the same type if they are constructed from the same type declaration, and corresponding type arguments are the same type.

20.5.3 Base Classes and Interfaces of a Constructed Type

A constructed class type has a direct base class, just like a simple class type. If the generic class declaration does not specify a base class, the base class is `object`. If a base class is specified in the generic class declaration, the base class of the constructed type is obtained by substituting, for each *type-parameter* in the base class declaration, the corresponding *type-argument* of the constructed type. Given the generic class declarations

```
class B<U,V> {...}
class G<T>: B<string,T[]> {...}
```

the base class of the constructed type `G<int>` would be `B<string,int[]>`.

Similarly, constructed class, struct, and interface types have a set of explicit base interfaces. The explicit base interfaces are formed by taking the explicit base interface declarations on the generic type declaration and substituting, for each *type-parameter* in the base interface declaration, the corresponding *type-argument* of the constructed type.

The set of all base classes and base interfaces for a type is formed, as usual, by recursively getting the base classes and interfaces of the immediate base classes and interfaces. For example, given the generic class declarations

```
class A {...}
class B<T>: A {...}
class C<T>: B<IComparable<T>> {...}
class D<T>: C<T[]> {...}
```

the base classes of `D<int>` are `C<int[]>`, `B<IComparable<int[]>>`, `A`, and `object`.

20.5.4 Members of a Constructed Type

The noninherited members of a constructed type are obtained by substituting, for each *type-parameter* in the member declaration, the corresponding *type-argument* of the constructed type.

For example, given the generic class declaration

```
class Gen<T,U>
{
    public T[,] a;
    public void G(int i, T t, Gen<U,T> gt) {...}
    public U Prop { get {...} set {...} }
    public int H(double d) {...}
}
```

the constructed type `Gen<int[],IComparable<string>>` has the following members.

```
public int[,][] a;
public void G(int i, int[] t, Gen<IComparable<string>,int[]> gt) {...}
public IComparable<string> Prop { get {...} set {...} }
public int H(double d) {...}
```

Note that the substitution process is based on the semantic meaning of type declarations and is not simply textual substitution. The type of the member `a` in the generic class declaration `Gen` is "two-dimensional array of `T`," so the type of the member `a` in the previous instantiated type is "two-dimensional array of one-dimensional array of `int`," or `int[,][]`.

The inherited members of a constructed type are obtained in a similar way. First, all the members of the immediate base class are determined. If the base class is itself a constructed type, this may involve a recursive application of the current rule. Then, each of the inherited members is transformed by substituting, for each *type-parameter* in the member declaration, the corresponding *type-argument* of the constructed type.

```
class B<U>
{
    public U F(long index) {...}
}
```

```
class D<T>: B<T[]>
{
    public T G(string s) {...}
}
```

In the previous example, the constructed type `D<int>` has a noninherited member `public int G(string s)` obtained by substituting the type argument `int` for the type parameter `T`. `D<int>` also has an inherited member from the class declaration `B`. This inherited member is determined by first determining the members of the constructed type `B<T[]>` by substituting `T[]` for `U`, yielding `public T[] F(long index)`. Then, the type argument `int` is substituted for the type parameter `T`, yielding the inherited member `public int[] F(long index)`.

20.5.5 Accessibility of a Constructed Type

A constructed type `C<`T_1`, ..., `T_N`>` is accessible when all its parts `C`, T_1, ..., T_N are accessible. For instance, if the generic type name `C` is `public` and all of the *type-arguments* T_1, ..., T_N are accessible as `public`, then the constructed type is accessible as `public`; but if either the *type-name* or one of the *type-arguments* has accessibility `private`, then the accessibility of the constructed type is `private`. If one *type-argument* has accessibility `protected`, and another has accessibility `internal`, then the constructed type is accessible only in this class and its subclasses in this assembly.

More precisely, the accessibility domain for a constructed type is the intersection of the accessibility domains of its constituent parts. Thus, if a method has a return type or argument type that is a constructed type where one constituent part is `private`, then the method must have an accessibility domain that is `private`; see §3.5.

20.5.6 Conversions

Constructed types follow the same conversion rules (§6) as do nongeneric types. When applying these rules, the base classes and interfaces of constructed types must be determined as described in §20.5.3.

No special conversions exist between constructed reference types other than those described in §6. In particular, unlike array types, constructed reference types do not exhibit "co-variant" conversions. This means that a type `List<B>` has no conversion (either implicit or explicit) to `List<A>` even if `B` is derived from `A`. Likewise, no conversion exists from `List<B>` to `List<object>`.

The rationale for this is simple: If a conversion to `List<A>` is permitted, then apparently one can store values of type `A` into the list. But this would break the invariant that every object in a list of type `List<B>` is always a value of type `B`, or else unexpected failures may occur when assigning into collection classes.

The behavior of conversions and runtime type checks is illustrated as follows.

```
class A {...}

class B: A {...}

class Collection {...}

class List<T>: Collection {...}

class Test
{
    void F() {
        List<A> listA = new List<A>();
        List<B> listB = new List<B>();

        Collection c1 = listA;        // Ok, List<A> is a Collection
        Collection c2 = listB;        // Ok, List<B> is a Collection

        List<A> a1 = listB;           // Error, no implicit conversion
        List<A> a2 = (List<A>)listB;  // Error, no explicit conversion
    }
}
```

20.5.7 The System.Nullable <T> Type

The `System.Nullable<T>` generic struct type defined in the .NET Base Class Library represents a value of type `T` that may be null. The `System.Nullable<T>` type is useful in a variety of situations, such as to denote nullable columns in a database table or optional attributes in an Extensible Markup Language (XML) element.

An implicit conversion exists from the null type to any type constructed from `System.Nullable<T>`. The result of such a conversion is the default value of `System.Nullable<T>`. In other words, writing this

```
Nullable<int> x = null;
Nullable<string> y = null;
```

is the same as writing the following.

```
Nullable<int> x = Nullable<int>.default;
Nullable<string> y = Nullable<string>.default;
```

20.5.8 Using Alias Directives

Using aliases may name a closed constructed type but may not name a generic type declaration without supplying type arguments. For example

```
namespace N1
{
    class A<T>
    {
        class B {}
    }
```

```
    class C {}
}

namespace N2
{
    using W = N1.A;        // Error, cannot name generic type

    using X = N1.A.B;      // Error, cannot name generic type

    using Y = N1.A<int>;   // Ok, can name closed constructed type

    using Z = N1.C;        // Ok
}
```

20.5.9 Attributes

An open type may not be used anywhere inside an attribute. A closed constructed type can be used as the argument to an attribute but cannot be used as the *attribute-name* because `System.Attribute` cannot be the base type of a generic class declaration.

```
class A: Attribute
{
    public A(Type t) {...}
}

class B<T>: Attribute {}          // Error, cannot use Attribute as base

class List<T>
{
    [A(typeof(T))] T t;           // Error, open type in attribute
}

class X
{
    [A(typeof(List<int>))] int x; // Ok, closed constructed type

    [B<int>] int y;               // Error, invalid attribute name
}
```

20.6 Generic Methods

A generic method is a method that is generic with respect to certain types. A generic method declaration names, in addition to normal parameters, a set of type parameters that are provided when using the method. Generic methods may be declared inside class, struct, or interface declarations, which may themselves be either generic or nongeneric. If a generic method is declared inside a generic type declaration, the body of the method can refer to both the type parameters of the method and the type parameters of the containing declaration.

class-member-declaration:

…

generic-method-declaration

struct-member-declaration:

…
generic-method-declaration

interface-member-declaration:

…
interface-generic-method-declaration

Generic methods are declared by placing a type parameter list following the name of the method.

generic-method-declaration:
generic-method-header method-body

generic-method-header:
attributes$_{opt}$ *method-modifiers*$_{opt}$ *return-type member-name type-parameter-list*
(*formal-parameter-list*$_{opt}$) *type-parameter-constraints-clauses*$_{opt}$

interface-generic-method-declaration:
attributes$_{opt}$ `new`$_{opt}$ *return-type identifier type-parameter-list*
(*formal-parameter-list*$_{opt}$) *type-parameter-constraints-clauses*$_{opt}$;

The *type-parameter-list* and *type-parameter-constraints-clauses* have the same syntax and function as in a generic type declaration. The type parameters declared by the *type-parameter-list* are in scope throughout the *generic-method-declaration* and may be used to form types throughout that scope including the *return-type*, the *method-body*, and the *type-parameter-constraints-clauses* but excluding the *attributes*.

The name of a method type parameter cannot be the same as the name of an ordinary parameter to the same method.

The following example finds the first element in an array, if any, that satisfies the given test delegate. Generic delegates are described in §20.4.

```
public delegate bool Test<T>(T item);

public class Finder
{
    public static T Find<T>(T[] items, Test<T> test) {
        foreach (T item in items) {
            if (test(item)) return item;
        }
        throw new InvalidOperationException("Item not found");
    }
}
```

A generic method may not be declared `extern`. All other method modifiers are valid on a generic method.

20.6.1 Generic Method Signatures

For the purposes of signature comparisons, any type parameter constraints are ignored, as are the names of the type parameters, but the number of type parameters is relevant, as are the ordinal positions of type parameters in left-to-right ordering. The following example shows how method signatures are affected by this rule.

```
class A {}

class B {}

interface IX
{
    T F1<T>(T[] a, int i);          // Error, both declarations have the same
    void F1<U>(U[] a, int i);       // signature because return type and type
                                    // parameter names are not significant

    void F2<T>(int x);              // Ok, the number of type parameters is part
    void F2(int x);                 // of the signature

    void F3<T>(T t) where T: A;     // Error, constraints are not
    void F3<T>(T t) where T: B;     // considered in signatures
}
```

Overloading of generic methods is further constrained by a rule similar to that which governs overloaded methods in a generic type declaration (20.1.8). Two generic methods declared with the same names and same number of type arguments must have parameter types such that no list of closed type arguments, when applied to both methods in the same order, yield two methods with the same signature. Constraints are not considered for the purposes of this rule. For example

```
class X<T>
{
    void F<U>(T t, U u) {...}      // Error, X<int>.F<int> yields two methods
    void F<U>(U u, T t) {...}      // with the same signature
}
```

20.6.2 Virtual Generic Methods

Generic methods can be declared using the `abstract`, `virtual`, and `override` modifiers. The signature matching rules described in §20.6.1 are used when matching methods for overriding or interface implementation. When a generic method overrides a generic method declared in a base class or implements a method in a base interface, the constraints given for each method type parameter must be the same in both declarations, where method type parameters are identified by ordinal positions, left to right.

```
abstract class Base
{
    public abstract T F<T,U>(T t, U u);

    public abstract T G<T>(T t) where T: IComparable;
}
```

```
class Derived: Base
{
    public override X F<X,Y>(X x, Y y) {...}     // Ok

    public override T G<T>(T t) {...}            // Error
}
```

The override of `F` is correct because type parameter names are permitted to differ. The override of `G` is incorrect because the given type parameter constraints (in this case, none) do not match those of the method being overridden.

20.6.3 Calling Generic Methods

A generic method invocation may explicitly specify a type argument list, or it may omit the type argument list and rely on type inference to determine the type arguments. The exact compile-time processing of a method invocation, including a generic method invocation, is described in §20.9.5. When a generic method is invoked without a type argument list, type inference takes place as described in §20.6.4.

The following example shows how overload resolution occurs after type inference and after type arguments are substituted into the parameter list.

```
class Test
{
    static void F<T>(int x, T y) {
        Console.WriteLine("one");
    }

    static void F<T>(T x, long y) {
        Console.WriteLine("two");
    }

    static void Main() {
        F<int>(5, 324);       // Ok, prints "one"
        F<byte>(5, 324);      // Ok, prints "two"
        F<double>(5, 324);    // Error, ambiguous

        F(5, 324);            // Ok, prints "one"
        F(5, 324L);           // Error, ambiguous
    }
}
```

20.6.4 Inference of Type Arguments

When a generic method is called without specifying type arguments, a ***type inference*** process attempts to infer type arguments for the call. The presence of type inference allows a more convenient syntax to be used for calling a generic method and allows the programmer to avoid specifying redundant type information. For example, given the method declaration

```
class Util
{
    static Random rand = new Random();

    static public T Choose<T>(T first, T second) {
        return (rand.Next(2) == 0)? first: second;
    }
}
```

it is possible to invoke the `Choose` method without explicitly specifying a type argument.

```
int i = Util.Choose(5, 213);              // Calls Choose<int>

string s = Util.Choose("foo", "bar");     // Calls Choose<string>
```

Through type inference, the type arguments `int` and `string` are determined from the arguments to the method.

Type inference occurs as part of the compile-time processing of a method invocation (§20.9.5) and takes place before the overload resolution step of the invocation. When a particular method group is specified in a method invocation, and no type arguments are specified as part of the method invocation, type inference is applied to each generic method in the method group. If type inference succeeds, then the inferred type arguments are used to determine the types of arguments for subsequent overload resolution. If overload resolution chooses a generic method as the one to invoke, then the inferred type arguments are used as the actual type arguments for the invocation. If type inference for a particular method fails, that method does not participate in overload resolution. The failure of type inference, in and of itself, does not cause a compile-time error. However, it often leads to a compile-time error when overload resolution then fails to find any applicable methods.

If the supplied number of arguments is different from the number of parameters in the method, then inference immediately fails. Otherwise, type inference first occurs independently for each regular argument that is supplied to the method. Assume this argument has type `A`, and the corresponding parameter has type `P`. Type inferences are produced by relating the types `A` and `P` according to the following steps.

- Nothing is inferred from the argument (but type inference succeeds) if any of the following are true.
  - `P` does not involve any method type parameters.
  - The argument is the `null` literal.
  - The argument is an anonymous method.
  - The argument is a method group.

- If `P` is an array type and `A` is an array type of the same rank, then replace `A` and `P` respectively with the element types of `A` and `P` and repeat this step.
- If `P` is an array type and `A` is not an array type of the same rank, then type inference fails for the generic method.
- If `P` is a method type parameter, then type inference succeeds for this argument, and `A` is the type inferred for that type parameter.
- Otherwise, `P` must be a constructed type. If, for each method type parameter M_X that occurs in `P`, exactly one type T_X can be determined such that replacing each M_X with each T_X produces a type to which `A` is convertible by a standard implicit conversion, then inference succeeds for this argument and each T_X is the type inferred for each M_X. Method type parameter constraints, if any, are ignored for the purpose of type inference. If, for a given M_X, no T_X exists or more than one T_X exists, then type inference fails for the generic method (a situation where more than one T_X exists can only occur if `P` is a generic interface type and `A` implements multiple constructed versions of that interface).

If all of the method arguments are processed successfully by the previous algorithm, then all inferences that were produced from the arguments are pooled. This pooled set of inferences must have the following properties.

- Each type parameter of the method must have had a type argument inferred for it. In short, the set of inferences must be ***complete***.
- If a type parameter occurred more than once, then all of the inferences for that type parameter must infer the same type argument. In short, the set of inferences must be ***consistent***.

If a complete and consistent set of inferred type arguments is found, then type inference is said to have succeeded for the given generic method and argument list.

If the generic method was declared with a parameter array (§10.5.1.4), then type inference is first performed against the method in its normal form. If type inference succeeds, and the resultant method is applicable, then the method is eligible for overload resolution in its normal form. Otherwise, type inference is performed against the method in its expanded form (§7.4.2.1).

20.6.5 Grammar Ambiguities

The productions for *simple-name* and *member-access* in §20.9.3 and §20.9.4 can give rise to ambiguities in the grammar for expressions. For example, the statement

```
F(G<A,B>(7));
```

could be interpreted as a call to `F` with two arguments, `G < A` and `B > (7)`. Alternatively, it could be interpreted as a call to `F` with one argument, which is a call to a generic method `G` with two type arguments and one regular argument.

If an expression could be parsed in two different valid ways, where > can be either interpreted as all or part of an operator or as ending a *type-argument-list*, the token immediately following the > is examined. If it is one of the following

```
(   )   ]   >   :   ;   ,   .   ?
```

then the > is interpreted as the end of a *type-argument-list*. Otherwise, the > is interpreted as an operator.

20.6.6 Using a Generic Method with a Delegate

An instance of a delegate can be created that refers to a generic method declaration. The exact compile-time processing of a delegate creation expression, including a delegate creation expression that refers to a generic method, is described in §20.9.6.

The type arguments used when invoking a generic method through a delegate are determined when the delegate is instantiated. The type arguments can be given explicitly via a *type-argument-list* or determined by type inference (§20.6.4). If type inference is used, the parameter types of the delegate are used as argument types in the inference process. The return type of the delegate is *not* used for inference. The following example shows both ways of supplying a type argument to a delegate instantiation expression.

```
delegate int D(string s, int i);

delegate int E();

class X
{
    public static T F<T>(string s, T t) {...}

    public static T G<T>() {...}

    static void Main() {
        D d1 = new D(F<int>);      // Ok, type argument given explicitly
        D d2 = new D(F);           // Ok, int inferred as type argument

        E e1 = new E(G<int>);      // Ok, type argument given explicitly
        E e2 = new E(G);           // Error, cannot infer from return type
    }
}
```

In the previous example, a nongeneric delegate type was instantiated using a generic method. It is also possible to create an instance of a constructed delegate type (§20.4) using a generic method. In all cases, type arguments are given or inferred when the delegate instance is created, and a *type-argument-list* may not be supplied when a delegate is invoked (§15.3).

20.6.7 No Generic Properties, Events, Indexers, or Operators

Properties, events, indexers, and operators may not themselves have type parameters (although they can occur in generic classes and use the type parameters from an enclosing class). If a property-like construct is required that must itself be generic, a generic method must be used instead.

20.7 Constraints

Generic type and method declarations can optionally specify type parameter constraints by including *type-parameter-constraints-clauses* in the declaration.

type-parameter-constraints-clauses:
    type-parameter-constraints-clause
    type-parameter-constraints-clauses *type-parameter-constraints-clause*

type-parameter-constraints-clause:
    `where` *type-parameter* `:` *type-parameter-constraints*

type-parameter-constraints:
    class-constraint
    interface-constraints
    constructor-constraint
    class-constraint `,` *interface-constraints*
    class-constraint `,` *constructor-constraint*
    interface-constraints `,` *constructor-constraint*
    class-constraint `,` *interface-constraints* `,` *constructor-constraint*

class-constraint:
    class-type

interface-constraints:
    interface-constraint
    interface-constraints `,` *interface-constraint*

interface-constraint:
    interface-type

constructor-constraint:
    `new ( )`

Each type parameter constraints clause consists of the token `where`, followed by the name of a type parameter, followed by a colon and the list of constraints for that type parameter. There can be only one `where` clause for each type parameter, but the `where` clauses may be listed in any order. Similar to the `get` and `set` tokens in a property accessor, the `where` token is not a keyword.

The list of constraints given in a `where` clause may include any of the following components in this order: a single class constraint, one or more interface constraints, and the constructor constraint `new()`.

If a constraint is a class type or an interface type, that type specifies a minimal "base type" that every type argument used for that type parameter must support. Whenever a constructed type or generic method is used, the type argument is checked against the constraints on the type parameter at compile time. The type argument supplied must derive from or implement all of the constraints given for that type parameter.

The type specified as a *class-constraint* must satisfy the following rules.

- The type must be a class type.
- The type must not be `sealed`.
- The type must not be one of the following special types: `System.Array`, `System.Delegate`, `System.Enum`, or `System.ValueType`.
- The type must not be `object`. Because all types derive from `object`, such a constraint would have no effect if it were permitted.
- At most, one constraint for a given type parameter can be a class type.

The type specified as an *interface-constraint* must satisfy the following rules.

- The type must be an interface type.
- The same type may not be specified more than once in a given `where` clause.

In either case, the constraint may involve any of the type parameters of the associated type or method declaration as part of a constructed type and may involve the type being declared, but the constraint may not be a type parameter alone.

Any class or interface type specified as a type parameter constraint must be at least as accessible (§10.5.4) as the generic type or method being declared.

If the `where` clause for a type parameter includes a constructor constraint of the form `new()`, it is possible to use the `new` operator to create instances of the type (§20.8.2). Any type argument used for a type parameter with a constructor constraint must have a parameterless constructor (see §20.7.1 for details).

The following are examples of possible constraints.

```
interface IPrintable
{
    void Print();
}
```

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

interface IKeyProvider<T>
{

    T GetKey();
}

class Printer<T> where T: IPrintable {...}

class SortedList<T> where T: IComparable<T> {...}

class Dictionary<K,V>
    where K: IComparable<K>
    where V: IPrintable, IKeyProvider<K>, new()
{
    ...
}
```

The following example is in error because it attempts to use a type parameter directly as a constraint.

```
class Extend<T,U> where U: T {...}// Error
```

Values of a constrained type parameter type can be used to access the instance members implied by the constraints. In the example

```
interface IPrintable
{
    void Print();
}

class Printer<T> where T: IPrintable
{
    void PrintOne(T x) {
        x.Print();
    }
}
```

the methods of `IPrintable` can be invoked directly on `x` because `T` is constrained to always implement `IPrintable`.

20.7.1 Satisfying Constraints

Whenever a constructed type is used or a generic method is referenced, the supplied type arguments are checked against the type parameter constraints declared on the generic type or method. For each `where` clause, the type argument `A` that corresponds to the named type parameter is checked against each constraint as follows.

- If the constraint is a class type or an interface type, let `C` represent that constraint with the supplied type arguments substituted for any type parameters that appear in the constraint. To satisfy the constraint, it must be the case that type `A` is convertible to type `C` by one of the following:
  - An identity conversion (§6.1.1)
  - An implicit reference conversion (§6.1.4)
  - A boxing conversion (§6.1.5)
  - An implicit conversion from a type parameter `A` to `C` (§20.7.4).
- If the constraint is `new()`, the type argument `A` must not be `abstract` and must have a public parameterless constructor. This is satisfied if one of the following is true.
  - `A` is a value type (as described in §4.1.2, all value types have a public default constructor).
  - `A` is a class that is not `abstract` and `A` contains a public constructor with no parameters.
  - `A` is not `abstract` and has a default constructor (§10.10.4).

A compile-time error occurs if one or more of a type parameter's constraints are not satisfied by the given type arguments.

Because type parameters are not inherited, constraints are never inherited either. In the following example, `D` must specify a constraint on its type parameter `T` so that `T` satisfies the constraint imposed by the base class `B<T>`. In contrast, class `E` need not specify a constraint because `List<T>` implements `IEnumerable` for any `T`.

```
class B<T> where T: IEnumerable {...}
class D<T>: B<T> where T: IEnumerable {...}
class E<T>: B<List<T>> {...}
```

20.7.2 Member Lookup on Type Parameters

The results of member lookup in a type given by a type parameter `T` depends on the constraints, if any, specified for `T`. If `T` has no constraints or only the `new()` constraint, then member lookup on `T` returns the same set of members as member lookup on `object`. Otherwise, the first stage of member lookup (§20.9.2) considers all the members in each of the types that are constraints for `T`. After performing the first stage of member lookup for each of the type constraints of `T`, the results are combined, and then hidden members are removed from the combined results.

Before the advent of generics, member lookup always returned either a set of members declared solely in classes or a set of members declared solely in interfaces and possibly the type `object`. Member lookup on type parameters changes this somewhat. When a type parameter has both a class constraint and one or more interface constraints, member lookup can return a set of members, some of which were declared in the class, and others of which were declared in an interface. The following additional rules handle this case.

- During member lookup (§20.9.2), members declared in a class other than `object` hide members declared in interfaces.
- During overload resolution of methods (§7.5.5.1) and indexers (§7.5.6.2), if any applicable member was declared in a class other than `object`, all members declared in an interface are removed from the set of considered members.

These rules only have effect when doing binding on a type parameter with both a class constraint and an interface constraint. Informally, members defined in a class constraint are always preferred over members in an interface constraint.

20.7.3 Type Parameters and Boxing

When a struct type overrides a virtual method inherited from `System.Object` (`Equals`, `GetHashCode`, or `ToString`), invocations of the virtual method through an instance of the struct type does not cause boxing to occur. This is true even when the struct is used as a type parameter and the invocation occurs through an instance of the type parameter type. For example

```
using System;

struct Counter
{
    int value;

    public override string ToString() {
        value++;
        return value.ToString();
    }
}

class Program
{
    static void Test<T>() where T: new() {
        T x = new T();
        Console.WriteLine(x.ToString());
        Console.WriteLine(x.ToString());
        Console.WriteLine(x.ToString());
    }

    static void Main() {
        Test<Counter>();
    }
}
```

The output of the program is

```
1
2
3
```

Although it is never recommended for `ToString` to have side effects, the example demonstrates that no boxing occurred for the three invocations of `x.ToString()`.

Boxing never implicitly occurs when accessing a member on a constrained type parameter. For example, suppose an interface `ICounter` contains a method `Increment` that can be used to modify a value. If `ICounter` is used as a constraint, the implementation of the `Increment` method is called with a reference to the variable that `Increment` was called on, never a boxed copy.

```
using System;

interface ICounter
{
    void Increment();
}

struct Counter: ICounter
{
    int value;

    public override string ToString() {
        return value.ToString();
    }

    void ICounter.Increment() {
        value++;
    }
}

class Program
{
    static void Test<T>() where T: new(), ICounter {
        T x = new T();
        Console.WriteLine(x);
        x.Increment();                // Modify x
        Console.WriteLine(x);
        ((ICounter)x).Increment();    // Modify boxed copy of x
        Console.WriteLine(x);
    }

    static void Main() {
        Test<Counter>();
    }
}
```

The first call to `Increment` modifies the value in the variable `x`. This is not equivalent to the second call to `Increment`, which modifies the value in a boxed copy of `x`. Thus, the output of the program is

```
0
1
1
```

20.7.4 Conversions Involving Type Parameters

The conversions that are allowed on a type parameter `T` depend on the constraints specified for `T`. All type parameters, constrained or not, have the following conversions.

- An implicit identity conversion from `T` to `T`.
- An implicit conversion from `T` to `object`. At runtime, if `T` is a value type, this is executed as a boxing conversion. Otherwise, it is executed as an implicit reference conversion.
- An explicit conversion from `object` to `T`. At runtime, if `T` is a value type, this is executed as an unboxing conversion. Otherwise, it is executed as an explicit reference conversion.
- An explicit conversion from `T` to any interface type. At runtime, if `T` is a value type, this is executed as a boxing conversion. Otherwise, it is executed as an explicit reference conversion.
- An explicit conversion from any interface type to `T`. At runtime, if `T` is a value type, this is executed as an unboxing conversion. Otherwise, it is executed as an explicit reference conversion.

If the type parameter `T` has the interface type `I` specified as a constraint, the following additional conversions exist.

- An implicit conversion from `T` to `I`, and from `T` to any base interface type of `I`. At runtime, if `T` is a value type, this is executed as a boxing conversion. Otherwise, it is executed as an implicit reference conversion.

If the type parameter `T` has the class type `C` specified as a constraint, the following additional conversions exist:

- An implicit reference conversion from `T` to `C`, from `T` to any class `C` is derived from, and from `T` to any interface `C` implements
- An explicit reference conversion from `C` to `T`, from any class `C` is derived from to `T`, and from any interface `C` implements to `T`

- An implicit user-defined conversion from `T` to `A`, if an implicit user-defined conversion exists from `C` to `A`
- An explicit user-defined conversion from `A` to `T`, if an explicit user-defined conversion exists from `A` to `C`
- An implicit reference conversion from the null type to `T`

An array type with element type `T` has the usual conversions to and from `object` and `System.Array` (§6.1.4, §6.2.3). If `T` has a class type `C` specified as a constraint, then additionally

- An implicit reference conversion exists from an array type A_T with element type `T` to an array type A_U with element type `U`, and an explicit reference conversion exists from A_U to A_T, if both the following are true:
  - A_T and A_U have the same number of dimensions.
  - `U` is one of: `C`, a class `C` is derived from, an interface `C` implements, an interface `I` that is specified as a constraint on `T`, or a base interface of `I`.

The previous rules do not permit a direct explicit conversion from an unconstrained type parameter to a noninterface type, which may be surprising. The reason for this rule is to prevent confusion and make the semantics of such conversions clear. For example, consider the following declaration.

```
class X<T>
{
    public static long F(T t) {
        return (long)t;           // Error, explicit conversion not permitted
    }
}
```

If the direct explicit conversion of `t` to `int` were permitted, one might easily expect that `X<int>.F(7)` would return `7L`. However, it would not because the standard numeric conversions are only considered when the types are known to be numeric at compile time. In order to make the semantics clear, the previous example must instead be written as follows.

```
class X<T>
{
    public static long F(T t) {
        return (long)(object)t;   // OK, conversions permitted
    }
}
```

20.8 Expressions and Statements

The operation of some expressions and statements is modified with generics. This section details those changes.

20.8.1 Default Value Expression

A default value expression is used to obtain the default value (§5.2) of a type. Typically, a default value expression is used for type parameters because it may not be known if the type parameter is a value type or a reference type. (No conversion exists from the `null` literal to a type parameter.)

primary-no-array-creation-expression:
    …
    default-value-expression

default-value-expression:
    primary-expression `.` `default`
    predefined-type `.` `default`

If a *primary-expression* is used in a *default-value-expression*, and the *primary-expression* is not classified as a type, then a compile-time error occurs. However, the rule described in §7.5.4.1 also applies to a construct of the form `E.default`.

If the left-hand side of a *default-value-expression* evaluates at runtime to a reference type, the result is `null` converted to that type. If the left-hand side of a *default-value-expression* evaluates at runtime to a value type, the result is the *value-type*'s default value (§4.1.2).

A *default-value-expression* is a constant expression (§7.15) if the type is a reference type or a type parameter that has a class constraint. In addition, a *default-value-expression* is a constant expression if the type is one of the following value types: `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal`, or `bool`.

20.8.2 Object Creation Expressions

The type of an object creation expression can be a type parameter. When a type parameter is specified as the type in an object creation expression, both of the following conditions must hold or a compile-time error occurs.

- The argument list must be omitted.
- A constructor constraint of the form `new()` must have been specified for the type parameter.

Execution of the object creation expression occurs by creating an instance of the runtime type that the type parameter has been bound to and invoking the default constructor of that type. The runtime type may be a reference type or a value type.

20.8.3 The typeof Operator

The `typeof` operator can be used on a type parameter. The result is the `System.Type` object for the runtime type that was bound to the type parameter. The `typeof` operator can also be used on a constructed type.

```
class X<T>
{
    public static void PrintTypes() {
        Console.WriteLine(typeof(T).FullName);
        Console.WriteLine(typeof(X<X<T>>).FullName);
    }
}

class M
{
    static void Main() {
        X<int>.PrintTypes();
    }
}
```

The previous program will print the following.

```
System.Int32
X<X<System.Int32>>
```

The `typeof` operator cannot be used with the name of a generic type declaration without specifying the type arguments.

```
class X<T> {...}

class M
{
    static void Main() {
        Type t = typeof(X);    // Error, X requires type arguments
    }
}
```

20.8.4 Reference Equality Operators

The reference type equality operators (§7.9.6) may be used to compare values of a type parameter `T` if `T` is constrained by a class constraint.

The use of the reference type equality operators is slightly relaxed to allow one argument to be of a type parameter `T` and the other argument to be `null`, even if `T` has no class constraint. At runtime, if `T` is a value type, the result of the comparison is `false`.

The following example checks whether an argument of an unconstrained type parameter type is `null`.

```
class C<T>
{
    void F(T x) {
        if (x == null) throw new ArgumentNullException();
        ...
    }
}
```

The `x == null` construct is permitted even though `T` could represent a value type, and the result is simply defined to be `false` when `T` is a value type.

20.8.5 The is Operator

The `is` operator operates on open types largely following the usual rules (§7.9.9). If either the compile-time type of `e` or `T` is an open type, then a dynamic type check on the runtime types of `e` and `T` is always performed.

20.8.6 The as Operator

The `as` operator can be used with a type parameter `T` as the right-hand side only if `T` has a class constraint. This restriction is required because the value `null` might be returned as a result of the operator.

```
class X
{
    public T F<T>(object o) where T: Attribute {
        return o as T;         // Ok, T has a class constraint
    }

    public T G<T>(object o) {
        return o as T;         // Error, unconstrained T
    }
}
```

In the current specification for the `as` operator (§7.9.10), for the expression `e as T` the final bullet point states that if no explicit reference conversion is available from the compile-time type of `e` to `T`, a compile-time error occurs. With generics, this rule changes slightly. If either the compile-time type of `e` or `T` is an open type, then no compile-time error occurs in this case; instead, a runtime type check occurs.

20.8.7 Exception Statements

The usual rules for `throw` (§8.9.5) and `try` (§8.10) statements apply when used with open types.

- The `throw` statement can be used with an expression whose type is given by a type parameter only if that type parameter has `System.Exception` (or a subclass thereof) as a class constraint.
- The type named in a `catch` clause may be a type parameter only if that type parameter has `System.Exception` (or a subclass thereof) as a class constraint.

20.8.8 The lock Statement

The `lock` statement may be used with an expression whose type is given by a type parameter. If the runtime type of the expression is a value type, the locking will have no effect (because the boxed value could not have any other references to it).

20.8.9 The using Statement

The `using` statement (§8.13) follows the usual rules: The expression must be implicitly convertible to `System.IDisposable`. If a type parameter is constrained by `System.IDisposable`, then expressions of that type may be used with a `using` statement.

20.8.10 The foreach Statement

Given a `foreach` statement of the form

```
foreach (ElementType element in collection) statement
```

If the `collection` expression is a type that does not implement the collection pattern but does implement the constructed interface `System.Collections.Generic.IEnumerable<T>` for exactly one type `T`, then the expansion of the `foreach` statement is

```
IEnumerator<T> enumerator = ((IEnumerable<T>)(collection)).GetEnumerator();
try {
    while (enumerator.MoveNext()) {
        ElementType element = (ElementType)enumerator.Current;
        statement;
    }
}
finally {
    enumerator.Dispose();
}
```

20.9 Revised Lookup Rules

Generics modify some of the basic rules used to look up and bind names. The following sections restate all the basic name lookup rules, taking generics into account.

20.9.1 Namespace and Type Names

The following replaces §3.8.

Several contexts in a C# program require a *namespace-name* or a *type-name* to be specified. Either form of name is written as one or more identifiers separated by . tokens.

namespace-name:
    namespace-or-type-name

type-name:
    namespace-or-type-name

namespace-or-type-name:
    identifier type-argument-list$_{opt}$
    namespace-or-type-name . *identifier type-argument-list*$_{opt}$

A *namespace-name* is a *namespace-or-type-name* that refers to a namespace. Following resolution as described below, the *namespace-or-type-name* of a *namespace-name* must refer to a namespace, or otherwise a compile-time error occurs. No type arguments can be present in a *namespace-name* (only types can have type arguments).

A *type-name* is a *namespace-or-type-name* that refers to a type. Following resolution as described below, the *namespace-or-type-name* of a *type-name* must refer to a type, or otherwise a compile-time error occurs.

The meaning of a *namespace-or-type-name* is determined as follows.

- If the *namespace-or-type-name* is of the form `I` or of the form `I<`A_1`, ..., `A_N`>`, where `I` is a single identifier and `<`A_1`, ..., `A_N`>` is an optional type argument list.
  - If the *namespace-or-type-name* appears within the body of a generic method declaration and if that declaration includes a type parameter of the name given by `I` and no type argument list was specified, then the *namespace-or-type-name* refers to that type parameter.
  - Otherwise, if the *namespace-or-type-name* appears within the body of a type declaration, then for each instance type `T` (§20.1.2), starting with the instance type of that type declaration and continuing with the instance type of each enclosing class or struct declaration (if any)
    - If the declaration of `T` includes a type parameter of the name given by `I` and no type argument list was specified, then the *namespace-or-type-name* refers to that type parameter.

  - Otherwise, if `I` is the name of an accessible member in `T` and if that member is a type with a matching number of type parameters, then the *namespace-or-type-name* refers to the type `T.I` or the type `T.I<`A_1`, ..., `A_N`>`. Note that nontype members (constants, fields, methods, properties, indexers, operators, instance constructors, destructors, and static constructors) and type members with a different number of type parameters are ignored when determining the meaning of a *namespace-or-type-name*.

- Otherwise, for each namespace `N`, starting with the namespace in which the *namespace-or-type-name* occurs, continuing with each enclosing namespace (if any) and ending with the global namespace, the following steps are evaluated until an entity is located.
  - If `I` is the name of a namespace in `N` and no type argument list was specified, then the *namespace-or-type-name* refers to that namespace.
  - Otherwise, if `I` is the name of an accessible type in `N` with a matching number of type parameters, then the *namespace-or-type-name* refers to that type constructed with the given type arguments.
  - Otherwise, if the location where the *namespace-or-type-name* occurs is enclosed by a namespace declaration for `N`
    - If the namespace declaration contains a *using-alias-directive* that associates the name given by `I` with an imported namespace or type and no type argument list was specified, then the *namespace-or-type-name* refers to that namespace or type.
    - Otherwise, if the namespaces imported by the *using-namespace-directive*s of the namespace declaration contain exactly one type with the name given by `I` and a matching number of type parameters, then the *namespace-or-type-name* refers to that type constructed with the given type arguments.
    - Otherwise, if the namespaces imported by the *using-namespace-directive*s of the namespace declaration contain more than one type with the name given by `I` and a matching number of type parameters, then the *namespace-or-type-name* is ambiguous and an error occurs.
- Otherwise, the *namespace-or-type-name* is undefined, and a compile-time error occurs.

- Otherwise, the *namespace-or-type-name* is of the form `N.I` or of the form `N.I<`A_1`, ..., `A_N`>`, where `N` is a *namespace-or-type-name*, `I` is an identifier, and `<`A_1`, ..., `A_N`>` is an optional type argument list. `N` is first resolved as a *namespace-or-type-name*. If the resolution of `N` is not successful, a compile-time error occurs. Otherwise, `N.I` or `N.I<`A_1`, ..., `A_N`>` is resolved as follows.
  - If `N` refers to a namespace and if `I` is the name of a nested namespace in `N` and no type argument list was specified, then the *namespace-or-type-name* refers to that nested namespace.
  - Otherwise, if `N` refers to a namespace and `I` is the name of an accessible type in `N` with a matching number of type parameters, then the *namespace-or-type-name* refers to that type constructed with the given type arguments.
  - Otherwise, if `N` refers to a (possibly constructed) class or struct type and `I` is the name of an accessible type nested in `N` with a matching number of type parameters, then the *namespace-or-type-name* refers to that type constructed with the given type arguments.
  - Otherwise, `N.I` is an invalid *namespace-or-type-name*, and a compile-time error occurs.

20.9.2 Member Lookup

The following replaces §7.3.

A member lookup is the process whereby the meaning of a name in the context of a type is determined. A member lookup may occur as part of evaluating a *simple-name* (§20.9.3) or a *member-access* (§20.9.4) in an expression.

A member lookup of a name `N` in a type `T` is processed as follows.

- First, a set of accessible members named `N` is determined.
  - If `T` is a type parameter, then the set is the union of the sets of accessible members named `N` in each of the types specified as a class constraint or interface constraint for `T`, along with the set of accessible members named `N` in `object`.
  - Otherwise, the set consists of all accessible (§3.5) members named `N` in `T`, including inherited members and the accessible members named `N` in `object`. If `T` is a constructed type, the set of members is obtained by substituting type arguments as described in §20.5.4. Members that include an `override` modifier are excluded from the set.

- Next, members that are hidden by other members are removed from the set. For every member `S.M` in the set, where `S` is the type in which the member `M` is declared, the following rules are applied.
  - If `M` is a constant, field, property, event, or enumeration member, then all members declared in a base type of `S` are removed from the set.
  - If `M` is a type declaration, then all nontypes declared in a base type of `S` are removed from the set, and all type declarations with the same number of type parameters as `M` declared in a base type of `S` are removed from the set.
  - If `M` is a method, then all nonmethod members declared in a base type of `S` are removed from the set, and all methods with the same signature as `M` declared in a base type of `S` are removed from the set.
- Next, interface members that are hidden by class members are removed from the set. This step only has an effect if `T` is a type parameter and `T` has both a class constraint and one or more interface constraints. For every member `S.M` in the set, where `S` is the type in which the member `M` is declared, the following rules are applied if `S` is a class declaration other than `object`.
  - If `M` is a constant, field, property, event, enumeration member, or type declaration, then all members declared in an interface declaration are removed from the set.
  - If `M` is a method, then all nonmethod members declared in an interface declaration are removed from the set, and all methods with the same signature as `M` declared in an interface declaration are removed from the set.
- Finally, having removed hidden members, the result of the lookup is determined.
  - If the set consists of a single member that is not a type and not a method, then this member is the result of the lookup.
  - Otherwise, if the set contains only methods, then this group of methods is the result of the lookup.
  - Otherwise, if the set contains only type declarations, then this group of type declarations in the result of the lookup.
  - Otherwise, the lookup is ambiguous, and a compile-time error occurs.

For member lookups in types other than type parameters and interfaces, and member lookups in interfaces that are strictly single inheritance (each interface in the inheritance chain has exactly zero or one direct base interface), the effect of the lookup rules is simply that derived members hide base members with the same name or signature. Such single-inheritance lookups are never ambiguous. The ambiguities that can possibly arise from member lookups in multiple-inheritance interfaces are described in §13.2.5.

20.9.3 Simple Names

The following replaces §7.5.2.

A *simple-name* consists of an identifier, optionally followed by a type parameter list.

simple-name:
identifier *type-argument-list*$_{opt}$

A *simple-name* of the form `I` or of the form `I<`A_1`, ..., `A_N`>`, where `I` is an identifier and `<`A_1`, ..., `A_N`>` is an optional type argument list, is evaluated and classified as follows.

- If the *simple-name* appears within a *block* and if the *block*'s (or an enclosing *block*'s) local variable declaration space (§3.3) contains a local variable or parameter with the name given by `I`, then the *simple-name* refers to that local variable or parameter and is classified as a variable. If a type argument list was specified, a compile-time error occurs.
- If the *simple-name* appears within the body of a generic method declaration and if that declaration includes a type parameter with the name given by `I`, then the *simple-name* refers to that type parameter. If a type argument list was specified, a compile-time error occurs.
- Otherwise, for each instance type `T` (§20.1.2), starting with the instance type of the immediately enclosing class, struct, or enumeration declaration and continuing with the instance type of each enclosing outer class or struct declaration (if any)
  - If the declaration of `T` includes a type parameter of the name given by `I`, then the *simple-name* refers to that type parameter. If a type argument list was specified, a compile-time error occurs.
  - Otherwise, if a member lookup (§20.9.2) of `I` in `T` produces a match
    - If `T` is the instance type of the immediately enclosing class or struct type and the lookup identifies one or more methods, the result is a method group with an associated instance expression of `this`. If a type argument list was specified, it is used in calling a generic method (§20.6.3).
    - If `T` is the instance type of the immediately enclosing class or struct type, if the lookup identifies an instance member, and if the reference occurs within the *block* of an instance constructor, an instance method, or an instance accessor, the result is the same as a member access (§20.9.4) of the form `this.I`. If a type argument list was specified, a compile-time error occurs.
    - Otherwise, the result is the same as a member access (§20.9.4) of the form `T.I` or `T.I<`A_1`, ..., `A_N`>`. In this case, it is a compile-time error for the *simple-name* to refer to an instance member.

- Otherwise, for each namespace `N`, starting with the namespace in which the *simple-name* occurs, continuing with each enclosing namespace (if any) and ending with the global namespace, the following steps are evaluated until an entity is located.
  - If `I` is the name of a namespace in `N` and no type argument list was specified, then the *simple-name* refers to that namespace.
  - Otherwise, if `I` is the name of an accessible type in `N` with a matching number of type parameters, then the *simple-name* refers to that type constructed with the given type arguments.
  - Otherwise, if the location where the *simple-name* occurs is enclosed by a namespace declaration for `N`
    - If the namespace declaration contains a *using-alias-directive* that associates the name given by `I` with an imported namespace or type and no type argument list was specified, then the *simple-name* refers to that namespace or type.
    - Otherwise, if the namespaces imported by the *using-namespace-directive*s of the namespace declaration contain exactly one type with the name given by `I` and a matching number of type parameters, then the *simple-name* refers to that type constructed with the given type arguments.
    - Otherwise, if the namespaces imported by the *using-namespace-directive*s of the namespace declaration contain more than one type with the name given by `I` and a matching number of type parameters, then the *simple-name* is ambiguous and an error occurs.
- Otherwise, the name given by the *simple-name* is undefined, and a compile-time error occurs.

20.9.4 Member Access

The following replaces §7.5.4.

A *member-access* consists of a *primary-expression* or a *predefined-type*, followed by a `.` token, followed by an *identifier*, optionally followed by a *type-argument-list*.

member-access:
    primary-expression `.` *identifier type-argument-list*$_{opt}$
    predefined-type `.` *identifier type-argument-list*$_{opt}$

predefined-type: one of

```
bool    byte   char   decimal  double  float  int     long
object  sbyte  short  string   uint    ulong  ushort
```

A *member-access* of the form `E.I` or of the form `E.I<`A_1`, ..., `A_N`>`, where `E` is a *primary-expression* or a *predefined-type*, `I` is an *identifier*, and `<`A_1`, ..., `A_N`>` is an optional *type-argument-list*, is evaluated and classified as follows.

- If `E` is a namespace and `I` is the name of a nested namespace in `E` and no type argument list was specified, then the result is that namespace.
- If `E` is a namespace and `I` is the name of an accessible type in `E` with a matching number of type parameters, then the result is that type constructed with the given type arguments.
- If `E` is a *predefined-type* or a *primary-expression* classified as a type, if `E` is not a type parameter, and if a member lookup (§20.9.2) of `I` in `E` produces a match, then `E.I` is evaluated and classified as follows.
  - If `I` identifies one or more type declarations, then determine the type declaration with the same number of type parameters (possibly zero) as were supplied in the *type-argument-list*, if present. The result is that type constructed with the given type arguments. If no type declaration has a matching number of type parameters, a compile-time error occurs.
  - If `I` identifies one or more methods, then the result is a method group with no associated instance expression. If a type argument list was specified, it is used in calling a generic method (§20.6.3).
  - If `I` identifies a static property, a static field, a static event, a constant, or an enumeration member, and if a type argument list was specified, a compile-time error occurs.
  - If `I` identifies a `static` property, then the result is a property access with no associated instance expression.
  - If `I` identifies a `static` field
    - If the field is `readonly` and the reference occurs outside the static constructor of the class or struct in which the field is declared, then the result is a value, namely the value of the static field `I` in `E`.
    - Otherwise, the result is a variable, namely the static field `I` in `E`.
  - If `I` identifies a `static` event
    - If the reference occurs within the class or struct in which the event is declared, and the event was declared without *event-accessor-declarations* (§10.7), then `E.I` is processed exactly as if `I` was a static field.
    - Otherwise, the result is an event access with no associated instance expression.
  - If `I` identifies a constant, then the result is a value, namely the value of that constant.

  - If `I` identifies an enumeration member, then the result is a value, namely the value of that enumeration member.
  - Otherwise, `E.I` is an invalid member reference, and a compile-time error occurs.
- If `E` is a property access, indexer access, variable, or value, the type of which is `T`, and a member lookup (§7.3) of `I` in `T` produces a match, then `E.I` is evaluated and classified as follows.
  - First, if `E` is a property or indexer access, then the value of the property or indexer access is obtained (§7.1.1) and `E` is reclassified as a value.
  - If `I` identifies one or more methods, then the result is a method group with an associated instance expression of `E`. If a type argument list was specified, it is used in calling a generic method (§20.6.3).
  - If `I` identifies an instance property, an instance field, or an instance event, and if a type argument list was specified, a compile-time error occurs.
  - If `I` identifies an instance property, then the result is a property access with an associated instance expression of `E`.
  - If `T` is a *class-type* and `I` identifies an instance field of that *class-type*
    - If the value of `E` is `null`, then a `System.NullReferenceException` is thrown.
    - Otherwise, if the field is `readonly` and the reference occurs outside an instance constructor of the class in which the field is declared, then the result is a value, namely the value of the field `I` in the object referenced by `E`.
    - Otherwise, the result is a variable, namely the field `I` in the object referenced by `E`.
  - If `T` is a *struct-type* and `I` identifies an instance field of that *struct-type*
    - If `E` is a value, or if the field is `readonly` and the reference occurs outside an instance constructor of the struct in which the field is declared, then the result is a value, namely the value of the field `I` in the struct instance given by `E`.
    - Otherwise, the result is a variable, namely the field `I` in the struct instance given by `E`.
  - If `I` identifies an instance event
    - If the reference occurs within the class or struct in which the event is declared, and the event was declared without *event-accessor-declarations* (§10.7), then `E.I` is processed exactly as if `I` was an instance field.
    - Otherwise, the result is an event access with an associated instance expression of `E`.
- Otherwise, `E.I` is an invalid member reference, and a compile-time error occurs.

20.9.5 Method Invocations

The following replaces the part of §7.5.5.1 that describes compile-time processing of a method invocation.

The compile-time processing of a method invocation of the form `M(A)`, where `M` is a method group (possibly including a *type-argument-list*), and `A` is an optional *argument-list*, consists of the following steps.

- The set of candidate methods for the method invocation is constructed. For each method `F` associated with the method group `M`
  - If `F` is nongeneric, `F` is a candidate when
    - `M` has no type argument list, and
    - `F` is applicable with respect to `A` (§7.4.2.1).
  - If `F` is generic and `M` has no type argument list, `F` is a candidate when
    - Type inference (§20.6.4) succeeds, inferring a list of type arguments for the call, and
    - Once the inferred type arguments are substituted for the corresponding method type parameters, the parameter list of `F` is applicable with respect to `A` (§7.4.2.1), and
    - The parameter list of `F`, after substituting type arguments, is not the same as an applicable nongeneric method, possibly in expanded form (§7.4.2.1), declared in the same type as `F`.
  - If `F` is generic and `M` includes a type argument list, `F` is a candidate when
    - `F` has the same number of method type parameters as were supplied in the type argument list, and
    - Once the type arguments are substituted for the corresponding method type parameters, the parameter list of `F` is applicable with respect to `A` (§7.4.2.1).
- The set of candidate methods is reduced to contain only methods from the most derived types: For each method `C.F` in the set, where `C` is the type in which the method `F` is declared, all methods declared in a base type of `C` are removed from the set.
- If the resulting set of candidate methods is empty, then no applicable methods exist, and a compile-time error occurs. If the candidate methods are not all declared in the same type, the method invocation is ambiguous, and a compile-time error occurs (this latter situation can only occur for an invocation of a method in an interface that has multiple direct base interfaces, as described in §13.2.5).

- The best method of the set of candidate methods is identified using the overload resolution rules of §7.4.2. If a single best method cannot be identified, the method invocation is ambiguous, and a compile-time error occurs. When performing overload resolution, the parameters of a generic method are considered after substituting the type arguments (supplied or inferred) for the corresponding method type parameters.
- Final validation of the chosen best method is performed.
  - The method is validated in the context of the method group: If the best method is a static method, the method group must have resulted from a *simple-name* or a *member-access* through a type. If the best method is an instance method, the method group must have resulted from a *simple-name*, a *member-access* through a variable or value, or a *base-access*. If neither of these requirements is true, a compile-time error occurs.
  - If the best method is a generic method, the type arguments (supplied or inferred) are checked against the constraints (§20.7.1) declared on the generic method. If any type argument does not satisfy the corresponding constraint(s) on the type parameter, a compile-time error occurs.

Once a method has been selected and validated at compile time by the previous steps, the actual runtime invocation is processed according to the rules of function member invocation described in §7.4.3.

20.9.6 Delegate Creation Expressions

The following replaces the part of §7.5.10.3 that describes compile-time processing of a delegate creation expression.

The compile-time processing of a *delegate-creation-expression* of the form `new D(E)`, where `D` is a *delegate-type* and `E` is an *expression*, consists of the following steps.

- If `E` is a method group
  - A single method is selected corresponding to a method invocation (§20.9.5) of the form `E(A)`, with the following modifications.
    - The parameter types and modifiers (`ref` or `out`) of `D` are used as the argument types and modifiers of the argument list `A`.
    - Conversions are not considered in applicability tests and type inferencing. In instances where an implicit conversion would normally suffice, types are instead required to be identical.
    - The overload resolution step is not performed. Instead, the set of candidates must include exactly one method that is compatible (§15.1) with `D` (following substitution of type parameters with type arguments), and this method becomes the one to which the newly created delegate refers. If no matching method exists, or if more than one matching method exists, a compile-time error occurs.

  - If the selected method is an instance method, the instance expression associated with `E` determines the target object of the delegate.
  - The result is a value of type `D`, namely a newly created delegate that refers to the selected method and target object.
- Otherwise, if `E` is a value of a *delegate-type*
  - `D` and `E` must be compatible (§15.1); otherwise, a compile-time error occurs.
  - The result is a value of type `D`, namely a newly created delegate that refers to the same invocation list as `E`.
- Otherwise, the delegate creation expression is invalid, and a compile-time error occurs.

20.10 Right-Shift Grammar Changes

The syntax for generics uses the < and > characters to delimit type parameters and type arguments (similar to the syntax used in C++ for templates). Constructed types sometimes nest, as in `List<Nullable<int>>`, but there is a subtle grammatical problem with such constructs: The lexical grammar will combine the last two characters of this construct into the single token >> (the right shift operator), rather than producing two > tokens, which the syntactic grammar would require. Although a possible solution is to put a space between the two > characters, this is awkward and confusing and does not add to the clarity of the program in any way.

In order to allow these natural constructs and to maintain a simple lexical grammar, the >> and >>= tokens are removed from the lexical grammar and replaced with *right-shift* and *right-shift-assignment* productions.

operator-or-punctuator: one of

```
{       }       [       ]       (       )       .       ,       :       ;
+       -       *       /       %       &       |       ^       !       ~
=       <       >       ?       ++      --      &&      ||      ==      ->
!=      <=      >=      +=      -=      *=      /=      %=      &=      |=
^=      <<      <<=
```

right-shift:

```
>       >
```

right-shift-assignment:

```
>       >=
```

Unlike other productions in the syntactic grammar, no characters of any kind (not even white space) are allowed between the tokens in the *right-shift* and *right-shift-assignment* productions.

The following productions are modified to use *right-shift* and *right-shift-assignment*.

shift-expression:
additive-expression
shift-expression << *additive-expression*
shift-expression *right-shift* *additive-expression*

assignment-operator:
=
+=
−=
*=
/=
%=
&=
|=
^=
<<=
right-shift-assignment

overloadable-binary-operator:
+
−
*
/
%
&
|
^
<<
right-shift
==
!=
>
<
>=
<=