ON THIS PAGE

Generic Parameters and Arguments

This chapter describes parameters and arguments for generic types, functions, and initializers. When you declare a generic type, function, subscript, or initializer, you specify the type parameters that the generic type, function, or initializer can work with. These type parameters act as placeholders that are replaced by actual concrete type arguments when an instance of a generic type is created or a generic function or initializer is called.

For an overview of generics in Swift, see Generics.

Generic Parameter Clause

A *generic parameter clause* specifies the type parameters of a generic type or function, along with any associated constraints and requirements on those parameters. A generic parameter clause is enclosed in angle brackets (<>) and has the following form:

```
<generic parameter list>
```

The *generic parameter list* is a comma-separated list of generic parameters, each of which has the following form:

```
type parameter: constraint
```

A generic parameter consists of a *type parameter* followed by an optional *constraint*. A *type parameter* is simply the name of a placeholder type (for example, `T`, `U`, `V`, `Key`, `Value`, and so on). You have access to the type parameters (and any of their associated types) in the rest of the type, function, or initializer declaration, including in the signature of the function or initializer.

The *constraint* specifies that a type parameter inherits from a specific class or conforms to a protocol or protocol composition. For example, in the generic function below, the generic parameter `T: Comparable` indicates that any type argument substituted for the type parameter `T` must conform to the `Comparable` protocol.

```
func simpleMax<T: Comparable>(_ x: T, _ y: T) -> T {
    if x < y {
        return y
    }
    return x
}
```

Because `Int` and `Double`, for example, both conform to the `Comparable` protocol, this function accepts arguments of either type. In contrast with generic types, you don't specify a generic argument clause when you use a generic function or initializer. The type arguments are instead inferred from the type of the arguments passed to the function or initializer.

```
simpleMax(17, 42) // T is inferred to be Int
simpleMax(3.14159, 2.71828) // T is inferred to be Double
```

Generic Where Clauses

You can specify additional requirements on type parameters and their associated types by including a generic `where` clause right before the opening curly brace of a type or function's body. A generic `where` clause consists of the `where` keyword, followed by a comma-separated list of one or more *requirements*.

```
where requirements
```

The *requirements* in a generic `where` clause specify that a type parameter inherits from a class or conforms to a protocol or protocol composition. Although the generic `where` clause provides syntactic sugar for expressing simple constraints on type parameters (for example, `<T: Comparable>` is equivalent to `<T> where T: Comparable` and so on), you can use it to provide more complex constraints on type parameters and their associated types. For example, you can constrain the associated types of type parameters to conform to protocols. For example, `<S: Sequence> where S.Iterator.Element: Equatable` specifies that `S` conforms to the `Sequence` protocol and that the associated type `S.Iterator.Element` conforms to the `Equatable` protocol. This constraint ensures that each element of the sequence is equatable.

You can also specify the requirement that two types be identical, using the `==` operator. For example, `<S1: Sequence, S2: Sequence> where S1.Iterator.Element == S2.Iterator.Element` expresses the constraints that `S1` and `S2` conform to the `Sequence` protocol and that the elements of both sequences must be of the same type.

Any type argument substituted for a type parameter must meet all the constraints and requirements placed on the type parameter.

You can overload a generic function or initializer by providing different constraints, requirements, or both on the type parameters. When you call an overloaded generic function or initializer, the compiler uses these constraints to resolve which overloaded function or initializer to invoke.

For more information about generic `where` clauses and to see an example of one in a generic function declaration, see Generic Where Clauses.

> GRAMMAR OF A GENERIC PARAMETER CLAUSE
>
> *generic-parameter-clause* → **<** generic-parameter-list **>**
> *generic-parameter-list* → generic-parameter | generic-parameter **,** generic-parameter-list
> *generic-parameter* → type-name
> *generic-parameter* → type-name **:** type-identifier
> *generic-parameter* → type-name **:** protocol-composition-type
>
> *generic-where-clause* → **where** requirement-list
> *requirement-list* → requirement | requirement **,** requirement-list
> *requirement* → conformance-requirement | same-type-requirement
>
> *conformance-requirement* → type-identifier **:** type-identifier

> *conformance-requirement* → type-identifier **:** protocol-composition-type
> *same-type-requirement* → type-identifier **==** type

Generic Argument Clause

A *generic argument clause* specifies the type arguments of a generic type. A generic argument clause is enclosed in angle brackets (<>) and has the following form:

```
<generic argument list>
```

The *generic argument list* is a comma-separated list of type arguments. A *type argument* is the name of an actual concrete type that replaces a corresponding type parameter in the generic parameter clause of a generic type. The result is a specialized version of that generic type. The example below shows a simplified version of the Swift standard library's generic dictionary type.

```
struct Dictionary<Key: Hashable, Value>: Collection,
  ExpressibleByDictionaryLiteral {
    /* ... */
}
```

The specialized version of the generic `Dictionary` type, `Dictionary<String, Int>` is formed by replacing the generic parameters `Key: Hashable` and `Value` with the concrete type arguments `String` and `Int`. Each type argument must satisfy all the constraints of the generic parameter it replaces, including any additional requirements specified in a generic `where` clause. In the example above, the `Key` type parameter is constrained to conform to the `Hashable` protocol and therefore `String` must also conform to the `Hashable` protocol.

You can also replace a type parameter with a type argument that is itself a specialized version of a generic type (provided it satisfies the appropriate constraints and requirements). For example, you can replace the type parameter `Element` in `Array<Element>` with a specialized version of an array, `Array<Int>`, to form an array whose elements are themselves arrays of integers.

```
let arrayOfArrays: Array<Array<Int>> = [[1, 2, 3], [4, 5,
```

```
    6], [7, 8, 9]]
```

As mentioned in Generic Parameter Clause, you don't use a generic argument clause to specify the type arguments of a generic function or initializer.

GRAMMAR OF A GENERIC ARGUMENT CLAUSE

generic-argument-clause → **<** generic-argument-list **>**
generic-argument-list → generic-argument | generic-argument **,** generic-argument-list
generic-argument → type

‹ Patterns Summary of the Grammar ›