Generic parameter type may appear as the type of a method parameter.

```
class Queu[+T] {
  def append(x:T) =
  ...
}
```

If T supports the same operations as U and all of T's operation have less strict requirements and provide more than the corresponding operations in U.

Then T is a subtype of U and you can substitute a T where a U is required.

Old versions remain available even after extensions or modifications.

```
trait OutputChannel[-T] {
  def write(x:T)
}
```

It makes sense that `OutputChannel` of `AnyRef` is a subtype of `OutputChannel` of `String` because a class that can write out an `AnyRef` can certainly write out `Strings`. The converse is not true.

A functional `Queue`.

```
class Queue[+T] {
  dep append[U>:T] (x:U) :
    Queue[U] = ...
}
```

This allows the adding of `Queue[Apple]` and an `Orange` to make a new `Queue[Fruit]`.

Before Java introduced generics, it was the only way to support method signatures that operated on all types of arrays.

```
void sort(Object[]a, Comparator cmp)
```

Scala lets you cast an array of T to an array of any supertype of T.

They appear before formal parameter.

```
MyClass[-T] contravariant
MyClass[T] nonvariant
MyClass[+T] covariant
```

- Efficiency, type of elements added may be checked against store type.
- Storage is not covariant, resulting in runtime `ArrayStoreExceptions` the compiler cannot catch.

```
class MyClass private (
  val param1: Int
  val param2: String
)
```

Escaping the variance checker's usual prohibition on reassignable fields of a covariant parameter.

Correctness, by classifying all positions in a class as positive, negative, or neutral.

Nonvariants can be used at all three, covariants at positive only, and contravariants at negative only.

If the var is of the covariant type generated, setter and getter have methods with the covariant parameter.