Creating Generics Classes

We'll cover the following

Often the classes we create deal with specific types; for example, a `Book` may have `title` of type `String` and a list of `Authors`, a `Publisher`, and so on. Sometimes though, especially when working with collections, we may not want to tie the members of a class to specific types. For example, a list of `Books` vs. a list of `Authors` are both lists. The list should be general enough to support members of either of those types, or other types, but specific enough to disallow, for example, adding an `Author` to a list of `Books`. Generics classes are used to create such generalized lists with type safety. Kotlin support for generics is much like the support in Java in many ways, but variance and constraints are declared differently; we discussed these in the context of generic functions in Generics: Variance and Constraints of Parametric Types.

Generic classes

We frequently use generic classes, but, as application programmers, we tend not to create these types of classes as often as we create nongeneric classes. Still, it's useful to learn how to create a generic class. We'll achieve that goal with an example.

Kotlin has a class named `Pair` that holds two objects of two different types. We'll create here a `PriorityPair` that will hold a pair of objects of the same type, but based on ordering, with the larger object first and the smaller one second. We'll use a `compareTo()` method, of the `Comparable<T>` interface, to determine the objects' ordering.

Before we jump into code, let's think through the features of the class we're about to create. Once we create an instance, there's no need to modify the members that are part of the `PriorityPair`, thus there's no need for any mutable operations.

are part of the PriorityPair, thus there's no need for any mutable operations. Since the objects will be ordered, the parametric class needs to be constrained to

implement the `Comparable<T>` interface. The class permits only read and not write, so we may be tempted to mark the parametric type with the out annotation, like the way class `Pair<out A, out B>` is defined in Kotlin. But since the properties of our class will be passed to the `compareTo()` method of `Comparable<T>`, we can't annotate with the `out` keyword. That's enough thinking—let's jump into the code now.

Here's the `PriorityPair` class with parametrized type `T` constrained to implement `Comparable<T>`:

```kotlin
class PriorityPair<T: Comparable<T>>(member1: T, member2: T) {
  val first: T
  val second: T

  init {
    if (member1 >= member2) {
      first = member1
      second = member2
    } else {
      first = member2
      second = member1
    }
  }

  override fun toString() = "${first}, ${second}"
}
```

prioritypair.kts

Following the class name, the parametric type and the constraint are specified within the angle brackets `<>`. Since there's only one constraint, we didn't need the `where` clause and instead used the concise colon `:` syntax we saw in Generics: Variance and Constraints of Parametric Types. The class receives two primary constructor parameters, `member1` and `member2`, of the parametric type `T`. Two properties—first and second—are defined as `val`, that is immutable, but their initialization is postponed to determine the ordering of the members.

Within the `init` block we initialize `first` and `second` based on the order of priority of the two members passed to the constructor, where the ordering is done using the `compareTo()` method of the Comparable interface. Instead of using the `compareTo()` method directly, we're using the `>=` operator—see Overloading Operators. Finally, the `toString()` method returns a `String` with the properties `first` and `second`.

Creating generic instances

Let’s use this generic class to create two instances:

```
println(PriorityPair(2, 1))      //2, 1
println(PriorityPair("A", "B"))  //B, A
```

prioritypair.kts

An instance of `PriorityPair<T>` may be created using any type that implements the `Comparable<T>` interface to specialize the parametric type. A generic class builds on the syntax of a regular class with the added complexity of constraints and variance specifications. When designing a generic class, we have to do a lot more testing to ensure that the generics work properly for the specialized types.

Looking at the `PriorityPair` class, we haven’t implemented the `equals()` and `hashCode()` methods. We could implement those methods in minutes if needed, but you don’t have to write all the code manually all the time.

Even though Kotlin makes it easier to define properties and create methods in classes, it can generate some common things, as we’ll see in the next lesson.