Reified Type Parameters

We'll cover the following

- Why reified type parameters are needed
- Applying reification

Why reified type parameters are needed

When using generics in Java we sometimes get into smelly situations where we have to pass `Class<T>` parameters to functions. This becomes necessary when the specific parametric type is needed in a generic function but the type details are lost due to Java's type erasure. Kotlin removes this smell with reified type parameters.

To get a clear understanding of reification, we'll first explore some verbose and unpleasant code where we pass the class details as a parameter to a function, and then we'll refactor the code to use type reification.

Suppose we have a base class `Book` and a couple of derived classes, like so:

```
// reifiedtype.kts
abstract class Book(val name: String)
class Fiction(name: String) : Book(name)
class NonFiction(name: String) : Book(name)
```

Here's a list that includes different kinds of books:

```
// reifiedtype.kts
val books: List<Book> = listOf(
  Fiction("Moby Dick"), NonFiction("Learn to Code"), Fiction("LOTR"))
```

The list contains a mixture of `Fiction` and `NonFiction` instances in the `List<Book>`. Now, suppose we want to find the first instance of a particular type, either a `Fiction` or a `NonFiction` from the list. We may write a function like this in Kotlin, much like how we'd write it in Java:

```
fun <T> findFirst(books: List<Book>, ofClass: Class<T>): T {
    val selected = books.filter { book -> ofClass.isInstance(book) }
    if(selected.size == 0) {
        throw RuntimeException("Not found")
    }
    return ofClass.cast(selected[0])
}
```

Since the parameteric type `T` will be erased when the code is compiled to bytecode, we can't use `T` within the function to perform operations like `book is T` or `selected[0] as T`. As a work-around, both in Java and Kotlin, we pass the type of the object we desire as a parameter, like `ofClass: Class<T>` in this example. Then within the code, we use the `ofClass` to perform type checking and type cast, which makes the code verbose and messy. This approach also burdens the user of the function, like so:

```
println(findFirst(books, NonFiction::class.java).name) //Learn to Code
```

Each time the function is called, programmers have to pass the runtime type information as an additional argument. That's smelly both on the caller side and the callee side. And such effort makes the code error prone.

Applying reification

Thankfully, we have a much better alternatively in Kotlin—reified type parameters.

Kotlin still has to work with the limitations of type erasure—at runtime the parametric type isn't available. However, Kotlin permits us to use the parametric type within the function when the parametric type is marked as `reified` and the function itself is marked as `inline`. We'll discuss the benefits of `inline` in Inlining Functions with Lambdas, but for now, let's simply say that inlined functions are expanded at compile time, thus removing a function call overhead.

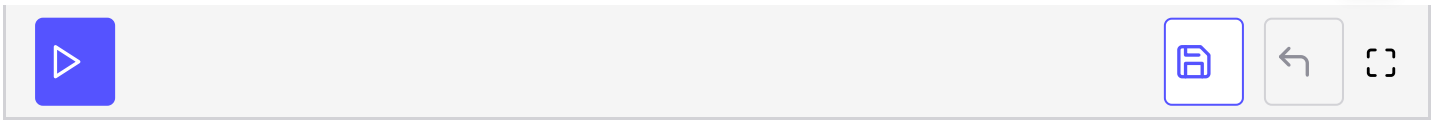
Let's refactor the `findFirst()` function to use reified.

```
inline fun <reified T> findFirst(books: List<Book>): T {
  val selected = books.filter { book -> book is T }

  if(selected.size == 0) {
    throw RuntimeException("Not found")
  }

  return selected[0] as T
}
```

We marked the parametric type `T` as `reified` and removed the `Class<T>` parameter. Within the function we use `T` to perform type checking and for casting. Since the function is marked as `inline`—we can use `reified` only for `inline` functions—the body of the function will be expanded at the site of a function call. Thus, the type `T` will be replaced in the expanded code with the actual type that is known at compile time.

In addition to removing the noise in the function, this feature also benefits the callers of the functions that use reified type parameters. Let's rewrite the call to the `findFirst()` function, like so:

```
println(findFirst<NonFiction>(books).name) //Learn to Code
```



reifiedtype.kts

Reified type parameters are useful to reduce clutter and also to alleviate potential errors in code. Reified type parameters eliminate the need to pass extra class information to functions, help to write code with safe casts, and customize the return type of functions with compile-time safety.

Keep your eyes open for functions that use reified type parameters as you work along with Kotlin code—for example, functions like `listOf<T>()` and `mutableListOf<T>()` of the Kotlin standard library you saw in Chapter 6, Using Collections, and `parse<T>` of the Klaxon library, as you'll see in Chapter 17, Asynchronous Programming.

QUIZ

Which other keyword has to be used with `reified` to make it work?

Retake Quiz

The next lesson concludes the discussion for this chapter.