# Any and Nothing Classes

We'll cover the following

- Any, the base class
- Nothing is deeper than void

Some methods like `equals()` and `toString()` are pervasive. In Java you'd expect to find them in the `Object` base class. In Kotlin, these methods are included in the `Any` class, along with a number of other methods that are useful on instances of any class. Any is Kotlin's counterpart of Java's `Object` class, except `Any` has a lot of special methods that come in through extension functions. Kotlin also has a class named `Nothing` that serves to stand in as type when a function literally is expected to return *nothing*—this is useful for type-checking methods when one or more branches is expected to return nothing. Nothing in Java is equivalent to Kotlin's `Nothing`. In this section you'll learn about the facilities offered by the ubiquitous `Any` and the purpose of `Nothing`.

## Any, the base class #

All classes in Kotlin inherit from `Any`, which maps to `Object` in Java. If a function will take objects of different types as a parameter, then you can specify its type as `Any`. Likewise, if you can't put your finger on a specific type to return, you may return `Any`. The `Any` class gives you the maximum—often too much—flexibility from the type point of view, so use it sparingly.

The purpose of `Any` isn't to let us define variables, parameters, or return types as `Any`, though occasionally we may want to, but to provide some common methods that are available on all Kotlin types. For example, methods like `equals()`, `hashCode()`, and `toString()` may be called on any type in Kotlin because those methods are implemented in `Any`.

Even though `Any` maps to `Object` in Java in the bytecode, they're not identical. Also, `Any` offers some special methods through extension functions. For example, the `to()` extension function that we saw in Using Pair and Triple, is an excellent

the `to()` extension function that we saw in Using Pair and Triple, is an excellent example. Since creating a `Pair` of different objects is such a common operation

and `Pair` is used widely with helper functions to create maps, Kotlin decided to make the `to()` method, which creates a `Pair` of objects of any type, universally available on objects of every single type.

Likewise, executing a block of code in the context of an object can remove a lot of verbose and repetitive code. To facilitate this, `Any` has extension functions like `let()`, `run()`, `apply()`, and `also()`—we'll explore these in Fluency with Any Object. By using these methods you can remove a lot of clutter. These are also useful for creating highly fluent internal DSLs in Kotlin, as you'll see in Chapter 14, Creating Internal DSLs.

## Nothing is deeper than void #

In languages like Java we use `void` to indicate that a method returns nothing. In Kotlin we use `Unit`, instead, to tell us when functions, which are expressions, return nothing useful. But there are situations where a function truly returns nothing...nada; that's where the `Nothing` class comes in. The class `Nothing` has no instances and it represents a value or result that will never exist. When used as a return type of a method it means that the function never returns—the function call will only result in an exception.

One unique capability of `Nothing` is that it can stand in for anything—that is, `Nothing` is substitutable for any class, including `Int`, `Double`, `String`, and so on. For example, take a look at the following code:

```
fun computeSqrt(n: Double): Double {
    if(n >= 0) {
        return Math.sqrt(n)
    } else {
        throw RuntimeException("No negative please")
    }
}
```

The `if` part returns a `Double`, while the `else` part throws an exception. The exception part is represented by the type `Nothing`. Cumulatively, the compiler can determine the return type of the `if` expression, in this case, to be a `Double` type. Thus the sole purpose of `Nothing` is to be able to help the compiler verify that the integrity of types in a program is sound.

In the next lesson, we'll learn how to deal with null values.