**Nullable types**

**fun** strLen(s: String) = s.length

Calling strLen with an argument that may be null isn’t allowed and will be flagged as error at compile time.

If you want to allow the use of this function with all arguments, including those thatcan be null, you need to mark it explicitly by putting a question mark after the type name.

**fun** strLen(s: String?) = s.length

Once you have a value of a nullable type, the set of operations you can perform on it is restricted.

You can’t assign it to a variable of a non- null type:

**val** x :String? = **null**

**var** y: String = x

You can’t pass it as a parameter to a function expecting a non- null argument:

By adding the check for null, the code now compiles.

**fun** strLen(s: String?): Int =

**if** (s != **null**) s.length **else** 0

**Safe call operator: ?.**

One of the most useful tools in Kotlin’s arsenal is the safe-call operator: ?. . It allows you to combine a null check and a method call into a single operation. For example, the expression s?.toUpperCase() is equivalent to the following, more cumbersome one: if (s != null) s.toUpperCase() else null .

**val** allCaps: String? = s?.toUpperCase()

allCaps may be null.

**Elvis operator: ?:**

**fun** foo(s: String?) {

**val** t: String = s ?: ""

}

**fun** strLenSafe(s: String?): Int = s?.length ?: 0

If "s" is null, the result is an empty string.

**Safe casts: as?**

The as? operator tries to cast a value to the specified type and returns null if the

value doesn’t have the proper type.

**val** otherPerson = x **as?** Person

**val** otherPerson = x **as?** Person ?: **throw** IllegalArgumentException("Not Person")

**Not-null assertions: !!**

It’s represented by a double exclamation mark and converts any value to a non- null type.

**val** sNotNull: String = s!!

**The let function**

The let function makes it easier to deal with a nullable argument that should be passed to a function that expects a non- null parameter. All the let function does is turn the object on which it’s called into a parameter of the lambda.

**val** email: String? = "emailaddress"

email?.let { email -> sendEmailTo(email) }

email?.let { sendEmailTo(it) }

**Late-initialized properties**

**class** MyService {

**fun** performAction(): String = "foo"

}

**class** MyTest {

**lateinit** **var** myService: MyService

// this initialise the variable

@Before

**fun** setUp() {

myService = MyService()

}

// accessing the variballe

**fun** testAction() {

myService.performAction()

}

}

Note that a late-initialized property is always a var , because you need to be able to change its value outside of the constructor. But you no longer need to initialize it in a constructor, even though the property has a non- null type. You can accesses the property without extra null checks.

**Extensions on nullable types**

**fun** String?.isNullOrBlank(): Boolean =

**this** == **null** || **this**.isBlank()

When you declare an extension function for a nullable type, that means you can call this function on nullable values; and **this** in a function body can be null , so you have to check for that explicitly.

Note that the let function we discussed earlier can be called on a nullable receiver as well, but it doesn’t check the value for null . If you invoke it on a nullable type without using the safe-call operator, the lambda argument will also be nullable:

person.let { sendEmailTo(it) }

you have to use the safe-call operator ?.

person?.let { sendEmailTo(it) }

**Nullability of type parameters**

**fun** <T> printHashCode(t: T) {

println(t?.hashCode())

}

In the printHashCode call, the inferred type for the type parameter T is a nullable type, Any? . Therefore, the parameter t is allowed to hold null , even without a question mark after T .

To make the type parameter non- null , you need to specify a non- null upper bound for it. That will reject a nullable value as an argument:

**fun** <T : Any> printHashCode(t: T) {

println(t.hashCode())

}