**Making functions easier to call**

**fun <T> joinToString(**

collection: Collection<T>,

separator: String,

prefix: String,

postfix: String): String {

**val** result = StringBuilder(prefix)

**for** ((index, element) **in** collection.withIndex()) {

**if** (index > 0) result.append(separator)

result.append(element)

}

result.append(postfix)

**return** result.toString()

}

**Named arguments**

**val list = listOf(1, 2, 3)**

joinToString(list, ", ", "(", ")")

joinToString(list, separator = ",", prefix = "(", postfix = ")")

**Default parameter values**

**fun <T> joinToString(**

collection: Collection<T>,

separator: String = ", ",

prefix: String = "",

postfix: String = ""

): String

When using the regular call syntax, you can omit only trailing arguments. If you use named arguments, you can omit some arguments from the middle of the list and specify only the ones you need.

joinToString(list, prefix = "# ")

when you call a Kotlin function with default parameter values from Java. you can annotate it with

@JvmOverloads . This instructs the compiler to generate Java overloaded functions, omitting each of the parameters one by one, starting from the last one.

**Getting rid of static utility classes**

In Kotlin, you don’t need to create classes that don’t contain any state or any instance methods and that act as containers for a bunch of static methods. A perfect example is the Collections class in the JDK. You can place functions directly at the top level of a source file, outside of any class. Such functions are still members of the package declared at the top of the file, and you still need to import them if you want to call them from other packages.

Create a file called join.kt with the following contents:

**package** strings

**fun** joinToString(...): String { ... }

let’s look at the Java code that would compile to the same class:

/\* Java \*/

**package** strings;

**public** **class** JoinKt {

**public** static String joinToString(...)

{ ... }

}

You can see that the name of the class generated by the Kotlin compiler corresponds to the name of the file containing the function.

To change the name of the generated class that contains Kotlin top-level

functions, you add a @JvmName annotation to the file. Place it at the

beginning of the file, before the package name:

@file:JvmName("className")

**TOP-LEVEL PROPERTIES**

Just like functions, properties can be placed at the top level of a file. If you want to expose a constant to Java code as a public static final field, to make its usage more natural, you can mark it with the const modifier.

**const** **val** UNIX\_LINE\_SEPARATOR = "\n"

This gets you the equivalent of the following Java code:

**public static final** String UNIX\_LINE\_SEPARATOR = "\n";

**Extension functions and properties**

Conceptually, an extension function is a simple thing: it’s a function that can be called

as a member of a class but is defined outside of it.

**fun** String.lastChar(): Char = **this**.get(**this**.length - 1)

"Hello".lastChar()

In the body of an extension function, you use this as you’d use it in a method. And, as in a regular method, you can omit it.

**fun** String.lastChar(): Char = get(length – 1)

**Imports and extension functions**

When you define an extension function, it doesn’t automatically become available across

your entire project. Instead, it needs to be imported.

**import** package.lastChar

or

**import** package.\*

You can change the name of the class or function you’re importing using the **as** keyword:

**as** keyword in an import statement is the only way to resolve the conflict.

**import** package.lastChar **as last**

"Kotlin".last()

**Calling extension functions from Java**

That makes using extension functions from Java pretty easy: you call the static

method and pass the receiver object instance. Just as with other top-level functions, the name of the Java class containing the method is determined from the name of the file where the function is declared. Let’s say it was declared in a StringUtil.kt file.

You can’t override an extension function.If the class has a member function with the same signature as an extension function, the member function always takes precedence.

**Extension properties**

Extension properties provide a way to extend classes with APIs that can be accessed using the property syntax, rather than the function syntax. Even though they’re called properties, they can’t have any state: it’s not possible to add extra fields to existing instances of Java objects.

**val** String.lastChar: Char

**get**() = get(length – 1)

The getter must always be defined, because there’s no backing field and therefore no default getter implementation. Initializers aren’t allowed for the same reason.

**var** StringBuilder.*lastChar*: Char

**get**() = get(length - 1)

**set**(value: Char) {

**this**.setCharAt(length - 1, value)

}

"Kotlin".***lastChar***

***Varargs***

When you call a function to create a list, you can pass any number of arguments to it:

**fun** listOf<T>(**vararg** values: T): List<T> { ... }

**val** list = listOf(2, 3, 5, 7, 11)

When calling the function Kotlin requires you to explicitly unpack the array. Technically, this feature is called using a spread operator, but in practice it’s as simple as putting the \* character before the corresponding argument.

**val** list = listOf("args: ", \*args)

The spread operator lets you combine the values from an array and some fixed values in a single call.

**Pairs**

**val** map = mapOf(1 to "one", 7 to "seven", 53 to "fifty-three")

The word to in this line of code isn’t a built-in construct, but rather a method invocation of a special kind, called an infix call.

Infix calls can be used with regular methods and extension functions that have one required parameter. To allow a function to be called using the infix notation, you need to mark it with the infix modifier. Here’s a simplified version of the declaration of the to function:

**infix** **fun** infixFuntion(other: Any) = Pair(**this**, other)

**infix** **fun** Any.to(other: Any) = Pair(**this**, other)

Note that you can assign a pair of elements to two variables directly:

**val** (number, name) = 1 to "one"

**Working with strings and regular expressions**

**Splitting strings**

The one that takes a regular expression requires a value of Regex type, not String . This ensures that it’s always clear whether a string passed to a method is interpreted as plain text or a regular expression.

"12.345-6.A".split("\\.|-".toRegex())

The other overload of the split extension function in Kotlin takes an arbitrary number of delimiters as plain-text strings:

"12.345-6.A".split(".", "-")

"12.345-6.A".split('.', '-')

**Triple-quoted strings**

The purpose of triple-quoted strings is not only to avoid escaping characters. Such a string literal can contain any characters, including line breaks but you can’t use special characters like \n .

so the Windows-style path

"C:\\Users\\yole\\kotlin-book"

can be written as

"""C:\Users\yole\kotlin-book"""

**local functions and extensions**

use local functions to fix a fairly common case of code duplication.

**class** User(**val** id: Int, **val** name: String, **val** address: String)

**fun** saveUser(user: User) {

**if** (user.name.isEmpty()) {

**throw** IllegalArgumentException(

"Cannot save user ${user.id}: Name is empty")

}

**if** (user.address.isEmpty()) {

**throw**

IllegalArgumentException(

"Cannot save user ${user.id}: Address is empty")

}

// Save user to the database

}

But if you put the validation code into a local function, you can get rid of the duplication and still maintain a clear code structure.

**class** User(**val** id: Int, **val** name: String, **val** address: String)

**fun** saveUser(user: User) {

**fun** validate(user: User,

value: String,

fieldName: String) {

**if** (value.isEmpty()) {

**throw** IllegalArgumentException(

"Cannot save user ${user.id}: $fieldName is empty")

}

}

validate(user, user.name, "Name")

validate(user, user.address, "Address")

// Save user to the database

}

because local functions have access to all parameters and variables of the enclosing function. Let’s take advantage of that and get rid of the extra User parameter:

**class** User(**val** id: Int, **val** name: String, **val** address: String)

**fun** saveUser(user: User) {

**fun** validate(value: String, fieldName: String) {

**if** (value.isEmpty()) {

**throw** IllegalArgumentException(

"Can't save user ${user.id}: " +

"$fieldName is empty")

}

}

validate(user.name, "Name")

validate(user.address, "Address")

// Save user to the database

}

To improve this example even further, you can move the validation logic into an extension function of the User class:

**class** User(**val** id: Int, **val** name: String, **val** address: String)

**fun** User.validateBeforeSave() {

**fun** validate(value: String, fieldName: String) {

**if** (value.isEmpty()) {

**throw** IllegalArgumentException(

"Can't save user $id: empty $fieldName")

}

}

validate(name, "Name")

validate(address, "Address")

}

**fun** saveUser(user: User) {

user.validateBeforeSave()

// Save user to the database

}