**3: Functional Data Types**

In this chapter…

**3.1 Simple data types**

The types we have used have been the primitive types such as **Int** and **Boolean**. In Chapter 2 we introduced function types (to describe higher order functions), function composition, function currying, etc. We can also use *type synonyms* to define more convenient and descriptive names as in:

typealias Point = Pair<Double, Double>

typealias Quantity = Int

typealias Username = String

typealias ProductCode = String

typealias ValidateOrder = (UnvalidatedOrder) -> ValidatedOrder

Here, the type name **Point** is a synonym for a **Pair** of **Double**s. A type alias in Kotlin does not introduce a new type: they are equivalent to the corresponding underlying types. In your code the Kotlin compiler will expand **Point** to Pair<Double, Double>. Type synonyms are purely for making code more readable.

Example 01a introduces the **Point** and **Line** typealias. Both are intended to increase the clarity of the code by emphasising that, for example, a **Pair** of **Point**s is used to represent a geometric **Line**. The **val** binding for **zero** in function **main** demonstrates how **Pair** and **Point** as synonyms. They can be used interchangeably. However, the declaration for functions **slope** and **length** demonstrate a consequence of using typealiases. The expression line.second.second refers to the y coordinate of the line end.

**Example 01a**: *Typealias*

package example01a

import kotlin.test.\*

import kotlin.math.sqrt

typealias Point = Pair<Double, Double>

typealias Line = Pair<Point, Point>

fun isAtOrigin(p: Point): Boolean = (p.first == 0.0) && (p.second == 0.0)

fun slope(line: Line): Double = (line.second.second - line.first.second) / (line.second.first - line.first.first)

fun length(line: Line): Double {

val xDist: Double = line.second.first - line.first.first

val yDist: Double = line.second.second - line.first.second

return sqrt(xDist \* xDist + yDist \* yDist)

}

fun main(args: Array<String>) {

val zero: Point = Pair(0.0, 0.0) // synonyms

val point43: Point = Point(4.0, 3.0)

val line43: Line = Line(zero, point43)

assertEquals(true, isAtOrigin(zero))

assertEquals(0.75, slope(line43))

assertEquals(5.0, length(line43))

}

We have demonstrated that a useful consequence of type aliases is that common types can be given aliases that more clearly communicate the intent. We see this in the above **typealias** examples. However, we cannot use type aliases to constrain the actual values. A value binding for **Username** will accept any **String** value:

val name: Username = “google.com”

Type aliases are invisible to the underlying type system. They are not checked by the compiler or visible at run-time using reflection. However, they are convenient to the programmer. They enable us to express the domain of some part of the program in a convenient way.

To better reflect the semantics of **Point** and **Line** types we are better using classes. Classes are at the centre of object-oriented programming. They define a *blueprint* from which instances are created. A class in Kotlin is used to represent some abstraction in the problem domain. A class declares the *state* (data) and the *behaviour* of objects defined by that class. Hence a Kotlin class describes both the *properties* and *functions* for that class. The properties specify the state information maintained by objects of the class. The functions define the behaviours we can expect from the objects.

Example 01b revisits the previous example using the **Point** and **Line** classes. The **Point** class has two properties **x** and **y** introduced into the *class constructor*. Both are tagged as **val** bindings so the class is immutable. One of the fundamental axioms of functional programming that has been embraced by the object oriented programming community is that all values are *immutable*: once initialised a variable cannot be modified. In fact, the term *variable* is quite inappropriate because it connotes the notion of change (or variability). For this reason we prefer the term *binding* (or the term *reference*) where an identifier is *bound* to (or references) an object. Specifically, once this binding is made on an *immutable reference* then no other binding to that identifier can occur.

**Example 01b**: *Classes*

package example01b

import kotlin.test.\*

import kotlin.math.sqrt

class Point(val x: Double, val y: Double)

class Line(val start: Point, val end: Point)

fun isAtOrigin(p: Point): Boolean = (p.x == 0.0) && (p.y == 0.0)

fun slope(line: Line): Double = (line.end.y - line.start.y) / (line.end.x - line.start.x)

fun length(line: Line): Double {

val xDist: Double = line.end.x - line.start.x

val yDist: Double = line.end.y - line.start.y

return sqrt(xDist \* xDist + yDist \* yDist)

}

fun main(args: Array<String>) {

val zero: Point = Point(0.0, 0.0)

val point43: Point = Point(4.0, 3.0)

val line43: Line = Line(zero, point43)

assertEquals(true, isAtOrigin(zero))

assertEquals(0.75, slope(line43))

assertEquals(5.0, length(line43))

}

Example 01b defines the simple class **Point** that has a *constructor* of two property *parameters*, both of type **Double**. Such a constructor declared immediately in the class header is called the *primary constructor*. To create an instance of the class **Point** one simply calls its primary constructor as if it were a regular function:

val point43: Point = Point(4.0, 3.0)

Class properties allow for a very natural syntax when getting their value. Each of the properties in Example 01b has a *backing field* that stores the value, and one *accessor function*: a getter. The **get** function is implicitly defined and is called when accessing a property. For example, the property access **point43.y** invokes the **get** function defined for the **y** property which retrieves the value of its backing field.

The full syntax for an immutable property can include a user-defined **get** function. If we define a custom getter, it will be called every time we access the property (this allows us to implement a computed property). Examples of this will be shown later.

**3.1.1 Immutable classes**

Immutable classes make a host of worrisome things in Kotlin disappear. One of the benefits of switching to a functional mindset is the realisation that tests exist to check that changes occur successfully in code. In other words, testing's purpose is to validate mutation — and the more mutation you have, the more testing is required to make sure you get it right. If you isolate the places where changes occur by severely restricting mutation, you create a much smaller space for errors to occur and have fewer places to test.

Embracing immutability is high on the list of ways to progress to a functional style. Although building immutable objects in Kotlin requires a bit more up-front complexity, the downstream simplification offered by this abstraction easily offsets the effort.

Immutability is one of the keys to functional programming. It minimises the parts that change and so make it easier to reason about those parts. Immutable classes and objects in object oriented programming remove many problematic issues. As we shall discuss in a later chapter large amounts of testing seeks to ensure that changes to variables occur successfully. The less mutation we have, the less testing we have to perform.

**3.2 Algebraic data types**

In functional programming an *algebraic data type* (or ADT) is a composite type formed by combining other types. The term algebraic refers to the fact that ADTs, like mathematical algebras, include a set of basic types and operations that apply to those types to create new types. The two common categories of ADT are *product type* and *sum type*. Product types supports having one or more value in a single structure. Sum types are types where a value must be one of a fixed set of options. Many data structures in everyday programming can be built using just these two mechanisms. Later, we shall see examples of these.

A *product type* is a combination of one or more component types. Product types are found in most programming languages. In Kotlin a product type is realised with a *class*. The **Point** and **Line** classes from Example 01b are examples of product types. Other examples are:

class Book(val isbn: String, val title: String, val publicationDate: LocalDate, val pages: Int)

class Contact(val id: String, val phone: String, val email: String)

class Patient(val id: String, val name: String, val dob: LocalDate, val gender: Boolean)

class User(val name: String, val password: String, val email: String)

datatype Book = Book of String \* String \* LocalDate \* Int

Example 02 illustrates using the **Patient** datatype. The example includes the **age** function.

**Example 02**: *Patient class*

package example02

import java.time.LocalDate

import java.time.Period

import kotlin.test.\*

class Patient(val id: String, val name: String, val dob: LocalDate, val gender: Boolean) {

fun age(): Int {

val today: LocalDate = LocalDate.now()

val period = Period.between(dob, today)

return period.years

} // age

} // Patient

fun main(args: Array<String>) {

val p: Patient = Patient("ABC123", "KenBarclay", LocalDate.of(1995, 1, 1), true)

assertEquals(28, p.age())

}

**3.2.1 Classes**

Classes are at the centre of object-oriented programming. They define a *blueprint* from which instances are created. A class in Kotlin is used to represent some abstraction in the problem domain. As discussed above, a class declares the *state* (data) and the *behaviour* of objects defined by that class. Hence a Kotlin class describes both the *properties* and *functions* for that class. The properties specify the state information maintained by objects of the class. The functions define the behaviours we can expect from the objects.

Example 02 defines the simple class **Patient** that has a *constructor* of four *parameters*, two of type **String**, one of type **LocalDate** and one of type **Boolean**. Such a constructor declared immediately in the class header is called the *primary constructor*. To create an instance of the class **Patient** one simply calls its primary constructor as if it were a regular function:

val p: Patient = Patient(“ABC123”, "Ken Barclay", LocalDate.of(2020, 1, 1), true)

More usually, a class also has *member functions* to define the behaviours of instances. In the simple **Account** class shown in Example 03 we introduce the functions **credit** and **debit** to support making a deposit and making a withdrawal. The function **credit**, for example, defines that its behaviour is to add the value of the *formal parameter* **amount** to the **balance** property. Since the properties for **Account** cannot be modified, a new **Account** object is returned by member function **credit**.

**Example 03**: *Member functions*

package example03

import kotlin.test.\*

class Account(val number: String, val balance: Int) {

fun credit(amount: Int): Account =

Account(number, balance + amount)

fun debit(amount: Int): Account =

if (balance >= amount)

Account(number, balance - amount)

else

this

}

fun main(args: Array<String>) {

val acc: Account = Account("ABC123", 1200)

val acc1: Account = acc.credit(200) // balance now 1400

assertEquals(1400, acc1.balance)

val acc2: Account = acc1.debit(900) // balance now 500

val acc3: Account = acc2.debit(700) // balance remains at 500

assertEquals(500, acc3.balance)

}

The function **debit** makes the withdrawal conditional on there being sufficient funds. If there are sufficient funds available then a new **Account** object is returned with the amount removed from the initial balance. If there are insufficient funds we return a reference to the object the function has been called upon using the **this** keyword.

**3.3 Immutability**

One of the fundamental axioms of functional programming that has been embraced by the object oriented programming community is that all values are *immutable*: once initialised an object cannot be modified. In fact, the term *variable* is quite inappropriate because it connotes the notion of change (or variability). For this reason we prefer the term *binding* (or the term *reference*) where an identifier is *bound* to (or references) an object. Specifically, once this binding is made on an *immutable reference* then no other binding to that identifier can occur.

An extension of this is an immutable object's state cannot change after its construction. The constructor is the only function that mutates the object's state. We have no other functions defined on the class that will mutate the state of the object. So the only way to mutate an object is to create another with the new values. This is the essence of class **Account** in Example 03.

Immutability is one of the keys to functional programming. It minimises the parts that change and so make it easier to reason about those parts. Immutable classes and objects in object oriented programming remove many problematic issues. As we shall discuss in a later chapter large amounts of testing seeks to ensure that changes to variables occur successfully. The less mutation we have, the less testing we have to perform.

In the book Effective Java by Joshua Bloch the author makes the compelling recommendation:

"Classes should be immutable unless there's a very good reason to make them mutable....If a class cannot be made immutable, limit its mutability as much as possible."

This is the philosophy we wish to follow in this book. We will seek to maximise opportunities to make our classes immutable. Further, we will aim to use immutable binding references so that an identifier only ever references one object.

A modern OO programming language like Kotlin has mechanisms such as scope and visibility to make it easy to modify state in a controlled manner. However state is so fundamental to OO that it requires great effort to manage it.

Immutable classes make a host of worrisome things in Kotlin disappear. One of the benefits of switching to a functional mindset is the realisation that tests exist to check that changes occur successfully in code. In other words, testing's purpose is to validate mutation — and the more mutation you have, the more testing is required to make sure you get it right. If you isolate the places where changes occur by severely restricting mutation, you create a much smaller space for errors to occur and have fewer places to test.

Embracing immutability is high on the list of ways to progress to a functional style. Although building immutable objects in Kotlin requires a bit more up-front complexity, the downstream simplification forced by this abstraction easily offsets the effort.

To make a Kotlin class immutable we must:

* Make all the properties **val**. When a property is defined as **val** you must either initialise it at the declaration or in the constructor function. Kotlin then only auto-generates the getter function.
* Make the class **final** so that a class cannot inherit from it (see next chapter), otherwise, the subclass might include a mutating function. Kotlin classes are final by default so there is no extra work for us to do here.
* Do not include any mutating functions other than the constructor. This means that setter functions will not be part of the class definition.
* Return values from functions must only be immutable objects. Otherwise, even if the object reference to the returned value is **val**, it would be possible to change the object.
* Provide a parameterised constructor with formal parameters to initialise the properties. More importantly there should be no default constructors unless all the state is properly initialised.

**3.4 Composition of product types**

Algebraic data types are composable which aids domain modelling. We can create complex models by mixing types in different combinations. A very simple example of composing types appeared earlier in which the **Line** class is composed of two **Point**s.

Working with our datatype declarations we can quickly sketch the types involved in payments made to a business for services provided:

Class CardNumber(val number: String)

Class CardType(val type: String)

Class CreditCard(val type: CardType, val number: CardNumber)

The types **CardNumber** and **CardType** are simple wrapper types. The **CreditCard** type is composed of the **CardType** and the **CardNumber**.

Example 04 introduces the classes **Name** and **Address** and uses them to compose the **Person** and **Customer** classes.

**Example 04**: *Composing types*

package example04

import kotlin.test.\*

data class Name(val firstName: String, val lastName: String)

data class Address(val street: String, val city: String, val postCode: String)

data class Person(val name: Name, val address: Address)

fun isAtSameCity(p: Person, q: Person): Boolean =

(p.address.city == q.address.city)

data class Id(val id: String)

data class Customer(val id: Id, val name: Name, val shippingAddress: Address, val billingAddress: Address)

fun isAtSameLocation(customer: Customer): Boolean =

(customer.shippingAddress == customer.billingAddress)

fun main(args: Array<String>) {

val c1: Customer = Customer(id = Id("ABC123"),

name = Name("Bob", "The Butcher"),

shippingAddress = Address("Princes Street", "Edinburgh", "EH1 1XY"),

billingAddress = Address("George Street", "Edinburgh", "EH22 2PQ")

)

val c2: Customer = Customer(id = Id("ABC123"),

name = Name("Bill", "The Baker"),

shippingAddress = Address("Baker Street", "Glasgow", "G1 1EA"),

billingAddress = Address("Baker Street", "Glasgow", "G1 1EA")

)

assertEquals(false, isAtSameLocation(c1))

assertEquals(true, isAtSameLocation(c2))

val p1 = Person(

Name("Ken", "Barclay"),

Address("Princes Street", "Edinburgh", "EH1 1AB")

)

val p2 = Person(

Name("John", "Savage"),

Address("York Place", "Edinburgh", "EH2 2XY")

)

assertEquals(true, isAtSameCity(p1, p2))

}

**3.5 Sum types**

Algebraic data types are powerful in representing the data model in a concise and transparent way. In many cases, they are defined as a closed set of possible values under one, common interface. The values of a *sum type* are typically grouped into several classes, called *variants*. A value of a variant type is usually created with a *value constructor*. Each variant has its own constructor, which takes a specified number of arguments with specified types (i.e. a product type). *Enumerated types* are a special case of sum types in which the constructors take no arguments, as exactly one value is defined for each constructor.

For example, a pet can be of at most one species at a time, although one might have as many house animals as needed. The **Pet** type can act as an enumeration here, containing a finite set of possible members. This makes algebraic data types a clever way of describing the entities in the application. Here is our **Pet** type:

enum class Pet {

DOG,

CAT

}

The algebra represented here is a *sum* (alternative) of the type’s members — either a **DOG** or a **CAT**. The next example might represent the rendering of a line in a graphics program:

enum class LineStyle {

SOLID,

DASHED,

DOTTED

}

The sum and product types can be freely mixed. With a few lines we can define a set of types that might be used in a payment application:

enum class CardType {

MASTERCARD,

VISA

}

class CardNumber(val number: String)

class CredtCard(val type: CardType, val number: CardNumber)

An *enum class* is a class whose fields consist of a fixed set of constants denoted by identifiers. In Kotlin an enum class is introduced using the **enum** keyword. For example a days-of-the-week enum class would be declared as:

enum class Day {

SUNDAY,

MONDAY,

TUESDAY,

WEDNESDAY,

THURSDAY,

FRIDAY,

SATURDAY

}

Enum types are especially useful for representing a fixed set of constants such as in our sum types. Examples might include the day-of-the-week example shown, the planets of our solar system, some fixed choice of colors, the coins of some monetary system or a set of volume settings on an audio device.

Example 05a demonstrates some fundamentals of a credit card type enumeration. The enum declaration is introduced with the **enum** keyword, has a type identifier (**CardType**), and comprises a list of identifiers. Since an enum is a value type we can assign a value directly (CardType.GOLD).

**Example 05a**: *Enum class*

package example05a

import kotlin.math.abs

import kotlin.test.\*

enum class CardType {

SILVER,

GOLD,

PLATINUM

}

fun creditLimit(cardType: CardType): Int =

when (cardType) {

CardType.SILVER -> 10000

CardType.GOLD -> 20000

CardType.PLATINUM -> 30000

} // creditLimit

fun creditDifference(ct1: CardType, ct2: CardType): Int =

abs(creditLimit(ct1) - creditLimit(ct2))

fun main(args: Array<String>) {

assertEquals(10000, creditLimit(CardType.SILVER))

assertEquals(20000, creditLimit(CardType.GOLD))

assertEquals(30000, creditLimit(CardType.PLATINUM))

assertEquals(0, creditDifference(CardType.SILVER, CardType.SILVER))

assertEquals(10000, creditDifference(CardType.SILVER, CardType.GOLD))

assertEquals(20000, creditDifference(CardType.SILVER, CardType.PLATINUM))

assertEquals(20000, creditDifference(CardType.PLATINUM, CardType.SILVER))

}

Example 05b introduces an enum type for a **Compass** bearing, defined with a property for the cardinal points NORTH, etc. Here, each enum constant is declared with a value representing the degree measure for that constant. These values are passed to the enum type primary constructor. The constructor of an enum type is implicitly private and, therefore, cannot be directly invoked. In this enum type we also introduce the function **minus** to obtain the angular change between two enum values. The member function **minus** is tagged as an **operator** so it can be used with the corresponding symbol. We see a usage in the assert statement.

**Example 05b**: *Defining enum constants with properties*

package example05b

import kotlin.test.\*

enum class Compass(val degrees: Int) {

NORTH(0),

EAST(90),

SOUTH(180),

WEST(270);

operator fun minus(c: Compass): Int = (this.degrees - c.degrees)

}

fun main(args: Array<String>) {

val east: Compass = Compass.EAST

assertEquals(180, Compass.WEST - east)

}

**3.5.1 Sealed classes and inheritance**

Typically the value constructors of a sum type will take a number of arguments with specified types (i.e. a product type). For example, in a library application books and journal may be borrowed by library members. We capture these publications with the sealed class declaration:

sealed class Publication

class Book(val author: Author, val title: Title, val isbn: ISBN) : Publication()

class Journal(val title: Title, val month: Month, val year: Year, val article: Article) : Publication()

}

In this the book has an author, a title and an ISBN book number. A journal has a title, the month and year of the issue and the name of an article.

Sea*led classes* are used in Kotlin for representing restricted class hierarchies, when a value can have one of the types from a limited set, but cannot have any other type. They are, in a sense, an extension of enum classes: the set of values for an enum type is also restricted, but each enum constant exists only as a single instance, whereas a subclass of a sealed class can have multiple instances which can contain state.

A *sealed class* is a class hierarchy in which all the subclasses must appear in the same file. Further, if we pattern match on a type that is a sealed class then the Kotlin compiler will ensure that all cases are covered.

Example 06a introduces the sealed class **Shape** derived from the specification:

sealed class Shape {

class Circle(val radius: Double) : Shape()

class Rectangle(val length: Double, val height: Double) : Shape()

class Triangle(val sideA: Double, val sideB: Double, val sideC: Double) : Shape()

}

The three classes **Circle**, **Rectangle** and **Triangle** are nested in the declaration for **Shape**. For more complex examples it may be better to declare them alongside the **Shape** class in the same file.

**Example 06a**: *Sealed class*

package example06a

import kotlin.test.\*

import kotlin.math.PI

import kotlin.math.abs

import kotlin.math.sqrt

sealed class Shape {

data class Circle(val radius: Double) : Shape()

data class Rectangle(val length: Double, val height: Double) : Shape()

data class Triangle(val sideA: Double, val sideB: Double, val sideC: Double) : Shape()

} // Shape

fun area(shape: Shape): Double =

when (shape) {

is Shape.Circle -> PI \* shape.radius \* shape.radius

is Shape.Rectangle -> shape.length \* shape.height

is Shape.Triangle -> {

val semiPerimeter: Double = 0.5 \* (shape.sideA + shape.sideB + shape.sideC)

sqrt(semiPerimeter \* (semiPerimeter - shape.sideA) \* (semiPerimeter - shape.sideB) \* (semiPerimeter - shape.sideC))

}

} // area

fun assertApproximate(expected: Double, actual: Double, precision: Double): Unit {

assertTrue(abs(expected - actual) < precision)

}

fun main(args: Array<String>) {

val circle: Shape.Circle = Shape.Circle(2.0)

val rectangle: Shape.Rectangle = Shape.Rectangle(3.0, 4.0)

val triangle: Shape.Triangle = Shape.Triangle(3.0, 4.0, 5.0)

assertApproximate(12.566, area(circle), 0.001)

assertApproximate(12.0, area(rectangle), 0.001)

assertApproximate(6.0, area(triangle), 0.001)

}

In this chapter we introduce the *inheritance* relationship that may exist between classes. It is widely used in object oriented applications and brings to our designs and programs a powerful feature unique to object orientation.

Inheritance (also known as *specialisation*) is a way to form new classes using classes that have already been defined. The former, known as *derived classes*, inherit properties and behaviours of the latter, which are referred to as *base classes*. The terms *parent class* and *child class* are also used in this context. Inheritance is intended to help reuse existing code with little or no modification.

Inheritance is also called *generalisation*. For instance, an account is a generalisation of both checking account and saving account. We say that account is an *abstraction* of checking account and saving account. Conversely, we can say that checking accounts are accounts i.e. a checking account *is-a* account. They inherit all the features common to all accounts, such as the account number property or the deposit function and therefore can substitute for where an account is expected.

The declaration:

class Circle(val radius: Double) : Shape()

introduces the class **Circle** as a derived class of the **Shape** class. In Kotlin to declare the subclass/superclass relationship, the name of the *superclass* is placed after a colon in the class header for the *subclass*. Further, we must initialise the properties of the superclass by calling its primary constructor. Had the superclass **Shape** included properties, then parameters from the **Circle** primary constructor are passed along to the **Shape** class primary constructor:

class Circe(...) : Shape(...)

Further, the **Shape** class is described as an *abstract class* when used in a sealed class hierarchy. Instances of an abstract class cannot be created.

Consider a bank in which its customers open various checking accounts and savings accounts. Each checking account is given a unique account number, as well as a balance and the permitted amount by which the account may be overdrawn. Each savings account has an account number and balance and the amount of interest earned by the account.

Example 06b includes the classes **Account**, **CurrentAccount** and **SavingAccount**. We can think of a **CheckingAccount** and a **SavingAccount** as special kinds of **Account**. The **Account** class has the features common to both the **CheckingAccount** and **SavingAccount** class, namely, the account number and the balance. The **CheckingAccount** class is then related to the **Account** class by *inheritance*. The **SavingAccount** class is also related to the **Account** class by inheritance. The **Account** class is usually referred to as the *superclass* (or base class) and the **CheckingAccount** (and **SavingAccount**) class as the *subclass* (or derived class).

E**xample 06b**: *Bank accounts*

package example06b

import kotlin.test.\*

sealed class Account(val number: String, val balance: Int) {

abstract fun isOverdrawn(): Boolean

class CheckingAccount(number: String, balance: Int, val overdraftLimit: Int) : Account(number, balance) {

val totalSpend: Int = balance + overdraftLimit

override fun isOverdrawn(): Boolean = (balance < -overdraftLimit)

}

class SavingAccount(number: String, balance: Int, val interestRate: Double) : Account(number, balance) {

val interest: Int = (balance \* interestRate/100.0).toInt()

override fun isOverdrawn(): Boolean = (balance < 0)

}

} // Account

fun main(args: Array<String>) {

val checkingAccount: Account.CheckingAccount = Account.CheckingAccount("ABC123", 1200, 200)

val savingAccount: Account.SavingAccount = Account.SavingAccount("DEF456", 2000, 4.0)

assertEquals(1400, checkingAccount.totalSpend)

assertEquals(80, savingAccount.interest)

val cAccount: Account.CheckingAccount = Account.CheckingAccount("GHI789", -300, 200)

assertEquals(false, checkingAccount.isOverdrawn())

assertEquals(false, savingAccount.isOverdrawn())

assertEquals(true, cAccount.isOverdrawn())

}

Note that since the **CheckingAccount** class will inherit the **number** and **balance** properties from the **Account** class, then the **number** and **balance** parameters are not qualified as **val**s in the **CheckingAccount** primary constructor, otherwise we would be re-declaring the properties (see the example listing). The same is true for the **SavingAccount** class. A number of restrictions apply to data classes. The primary constructor of a data class must have at least one parameter and be annotated as **val** or **var**. Consequently the subclasses **CheckingAccount** and SavingAccount are not declared as data classes.

In Kotlin all the features declared in a superclass are inherited by a subclass. This means that the **CheckingAccount** class need only declare those functions and properties required by it. In this case it is the additional **overdraftLimit** and **totalSpend** properties and the **isOverdrawn** member function. In more complex examples this would represent a significant saving in effort.

In Kotlin all the features declared in a superclass are inherited by a subclass. This means that if the **CheckingAccount** class did not define the **toString** function then the one defined in the **Account** class would be inherited and used by all **CheckingAccount** objects. However, the member functions of a subclass can refer to all the properties of its class and of the superclass.

With Kotlin, a function inherited by a subclass can be *redefined* to have a different behaviour. When we override in a subclass then the keyword **override** is necessary, otherwise Kotlin will report a compile error. Equally, it is an error to annotate a subclass feature with **override** when no such feature exists in the superclass.

It is often useful to be able to define a class that only acts as a basis for establishing others. There is no intention to make an instance of it. It is a way of guaranteeing that all descendants share a common set of features. This kind of class is referred to as an *abstract class*. We noted above that our sealed classes are automatically abstract classes.

It is common for an abstract class to include *deferred functions* i.e. one for which no function definition is given. This usually arises because the class is too abstract to determine how the function should be implemented. The inclusion of a deferred function in an abstract class infers that subclasses must provide an implementation if they are to represent concrete classes from which instances can be created. In effect, the inclusion of a deferred function imposes a *protocol* on subclasses that must be respected if a concrete class is required. A deferred function in Kotlin is known as an *abstract function* and is qualified with the **abstract** keyword. In Example 06b the abstract sealed class **Account** includes an abstract function entitled **isOverdrawn.**

**3.5.2 Pattern matching and the when expression**

Building algebraic types is only half the story; to do anything useful with them we need to tear them down and look inside them. Values of algebraic types are analyzed with *pattern matching*, which identifies a value by its constructor and extracts the data it contains.

Now we can use Kotlin’s *when expression* to handle the result. The when expression is akin to pattern matching. While not as powerful as pattern matching in some other languages, Kotlin’s when expression is one of the language’s most important features. Function **area** in Example 07a demonstrates using the when expression. In function **area**:

fun area(shape: Shape): Double =

when (shape) {

is Shape.Circle -> PI \* shape.radius \* shape.radius

is Shape.Rectangle -> shape.length \* shape.height

is Shape.Triangle -> {

val semiPerimeter: Double = 0.5 \* (shape.sideA + shape.sideB + shape.sideC)

sqrt(semiPerimeter \* (semiPerimeter - shape.sideA) \* (semiPerimeter - shape.sideB) \* (semiPerimeter - shape.sideC))

}

} // area

many features are at work.

 First we were able to check the type of **result** using Kotlin’s **is** operator. By checking the type, Kotlin is able to *smartcast* the value of **shape** for us for each case. So if **shape** is a **Circle** we can access the radius as if it were typed Shape.Circle. The next we did with the **when** expression is to exhaust all the possibilities for the **Shape** sealed class type. In the example there are no other possible types, so the compiler knows we have covered all cases. This represents an important feature when modelling our application domains. The program is:

**Example 07a**: *when expression*

package example07a

import java.lang.Math.PI

import java.lang.Math.sqrt

import kotlin.test.\*

sealed class Shape {

class Circle(val radius: Double) : Shape()

class Rectangle(val length: Double, val height: Double) : Shape()

class Triangle(val sideA: Double, val sideB: Double, val sideC: Double) : Shape()

}

fun area(shape: Shape): Double =

when (shape) {

is Shape.Circle -> PI \* shape.radius \* shape.radius

is Shape.Rectangle -> shape.length \* shape.height

is Shape.Triangle -> {

val semiPerimeter: Double = 0.5 \* (shape.sideA + shape.sideB + shape.sideC)

sqrt(semiPerimeter \* (semiPerimeter - shape.sideA) \* (semiPerimeter - shape.sideB) \* (semiPerimeter - shape.sideC))

}

}

fun main(args: Array<String>) {

val circle = Shape.Circle(2.0)

val rectangle = Shape.Rectangle(3.0, 4.0)

val triangle = Shape.Triangle(3.0, 4.0, 5.0)

assertEquals(12.566370614359172, area(circle))

assertEquals(12.0, area(rectangle))

assertEquals(6.0, area(triangle))

}

Consider a domain model of university students. Students are categorised as either undergraduate students or postgraduate students. Undergraduate students can take their studies either full time or part time which might have different durations or different fee structures. Postgraduate studies are always full time. We capture this specification with the **Student** sealed class:

To represent the hierarchy of **Student** correctly we make the **Undergraduate** class a sealed class with two concrete subclasses **FTUndergraduate** and **PTUndergraduate**. This is shown in Example 07b. Now our **when** expression will see each **Undergraduate** subclass as unique branches that must be exhausted, as in function **isFulltime**.

**Example 07b**: *Nested sealed classes*

package example07b

import kotlin.test.\*

sealed class Student(val matriculation: String, val name: String) {

}

package example07b

import kotlin.test.\*

sealed class Student(val matriculation: String, val name: String) {

class Postgraduate(matriculation: String, name: String, val projectTitle: String) : Student(matriculation, name)

sealed class Undergraduate(matriculation: String, name: String, val courseTitle: String, val duration: Int) : Student(matriculation, name) {

class FTUndergraduate(matriculation: String, name: String, courseTitle: String, duration: Int) : Undergraduate(matriculation, name, courseTitle, duration)

class PTUndergraduate(matriculation: String, name: String, courseTitle: String, duration: Int) : Undergraduate(matriculation, name, courseTitle, duration)

} // Undergraduate

} // Student

typealias Postgraduate = Student.Postgraduate

typealias FTUndergraduate = Student.Undergraduate.FTUndergraduate

typealias PTUndergraduate = Student.Undergraduate.PTUndergraduate

fun isFulltime(student: Student): Boolean =

when (student) {

is Student.Postgraduate -> true

is Student.Undergraduate.FTUndergraduate -> true

is Student.Undergraduate.PTUndergraduate -> false

} // isFulltime

fun main(args: Array<String>) {

val pg: Postgraduate = Postgraduate("ABC123", "Ken Barclay", "ADTs in Kotlin")

val ftug: FTUndergraduate = FTUndergraduate("DEF456", "John Savage", "Computing", 3)

val ptug: PTUndergraduate = PTUndergraduate("GHI789", "Jessie Kennedy", "Database", 5)

assertEquals(true, isFulltime(pg))

assertEquals(true, isFulltime(ftug))

assertEquals(false, isFulltime(ptug))

}

Note also the **typealias** to provide alternative names for existing types. Here we use the **typealias** to provide abbreviations for longer type names. Deliberately, in function **isFulltime** we have used the fully qualified names in the **when** expression. In function main we have shown how the abbreviations make the code more compact.