Table of Contents for Programming Scala, 2nd Edition

Chapter 8. Object-Oriented Programming in Scala

Scala is a functional programming language, but it is also an object-oriented programming language like Java, Python, Ruby, Smalltalk, and others. I've waited until now to explore Scala's "OO side" for two reasons.

First, I wanted to emphasize that functional programming has become an essential skill set for modern problems, a skill set that may be new to you. When you start with Scala, it's easy to use it as a "better Java," a better object-oriented language, and neglect the power of its functional side.

Second, a common architectural approach that Scala promotes is to use FP for *programming in the small* and OOP for *programming in the large*. Using FP for implementing algorithms, manipulating data, and managing state in a principled way is our best way to minimize bugs, the amount of code we write, and the risk of schedule delays. On the other hand, Scala's OO model provides tools for designing composable, reusable *modules*, which are essential for larger applications. Hence, Scala gives us the best of both worlds.

I've assumed you already understand the basics of object-oriented programming, such as Java's implementation. If you need a refresher, see Robert C. Martin's *Agile Software Development: Principles, Patterns, and Practices* or Bertrand Meyer's comprehensive introduction, *Object-Oriented Software Construction* (both by Prentice Hall). If you aren't familiar with *design patterns*, see *Design Patterns: Elements of Reusable Object-Oriented Software*, by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, known as the "Gang of Four" (Addison-Wesley).

In this chapter, we'll quickly review what we've already seen and fill in other details concerning Scala's terminology for OOP, including the mechanics of declaring classes and deriving one class from another, the notion of *value classes*, and how constructors work for Scala classes. The next chapter will dive into *traits* and then we'll spend a few chapters filling in additional details on Scala's object model and standard library.

Class and Object Basics

Classes are declared with the keyword `class`, while *singleton* objects are declared with the `object` keyword. For this reason, I use the term *instance* to refer to objects in general terms, even though *instance* and *object* are usually synonymous in most OO languages.

To prevent creation of *derived* classes from a class, prefix the declaration with the `final` keyword.

Use `abstract` to prevent instantiation of the class, such as when it contains or inherits member declarations (fields, methods, or types) without providing concrete definitions for them. Even when no members are undefined, `abstract` can still be used to prevent instantiation.

An instance can refer to itself using the `this` keyword. Although it's common to see `this` used in Java code, it's actually somewhat rare in Scala code. One reason is that constructor boilerplate is absent in Scala. Consider the following Java code:

```
// src/main/java/progscala2/basicoop/JPerson.java
package progscala2.basicoop;

public class JPerson {
  private String name;
  private int    age;

  public JPerson(String name, int age) {
    this.name = name;
    this.age  = age;
  }

  public void   setName(String name) { this.name = name;
}
  public String getName()            { return this.name;
}

  public void setAge(int age) { this.age = age;  }
  public int  getAge()        { return this.age; }
}
```

Now compare it with the following equivalent Scala declaration, in which all the boilerplate disappears:

```
class Person(var name: String, var age: Int)
```

Prefixing a constructor argument with a `var` makes it a mutable *field* of the class, also call an *instance variable* or *attribute* in different OO languages. Prefixing a constructor argument with a `val` makes it an immutable field. Using the `case` keyword infers the `val` keyword and also adds additional methods, as we've seen:

```
case class ImmutablePerson(name: String, age: Int)
```

Note that the state of an instance is the union of all the values currently represented by the instance's fields.

The term *method* refers to a function that is tied to an instance. In other words, its argument list has an "implied" `this` argument. Method definitions start with the `def` keyword. Scala will "lift" an applicable method into a function when a function argument is needed for another method or function.

Like most statically typed languages, Scala allows *overloaded methods*. Two or more methods can have the same name as long as their full *signatures* are unique. The signature includes the enclosing type name, method name, and the list of argument types (the names don't matter). In the JVM, different return types alone are not sufficient to distinguish methods.

However, recall from Working Around Erasure that the JVM prevents some methods from being truly distinct, because of *type erasure* of the type parameters for *higher-kinded* types, i.e., types with type parameters, like `List[A]`. Consider the following example:

```
scala> object C {
         def m(seq: Seq[Int]): Unit = println(s"Seq[Int]: $seq"
     | )
         def m(seq: Seq[String]): Unit = println(s"Seq[String]: $seq"
     | )
     | }
<console>:9: error: double definition:
method m:(seq: Seq[String])Unit and
method m:(seq: Seq[Int])Unit at line 8
have same type after erasure: (seq: Seq)Unit
         def m(seq: Seq[String]): Unit = println(s"Seq[String]: $seq"
)
             ^
```

The type parameters `Int` and `String` are erased in the byte code.

Unlike Java, member *types* can be declared using the `type` keyword. These types provide a complementary mechanism to type parameterization, as we saw in Abstract Types Versus Parameterized Types. They are often used as aliases for more complex types, to aid readability. Are type members and parameterized types redundant mechanisms? No, but we'll have to wait until Comparing Abstract Types and Parameterized Types to explore that question.

The term *member* refers to a field, method, or type in a generic way. Unlike in Java, a field and method can have the same name, but only if the method has an argument list:

```
scala> trait Foo {
          val x:
     |   Int
          def x:
     |   Int
     |   }
<console>:9: error: value x is defined twice
  conflicting symbols both originated in file '<console>'
         def x: Int
             ^

scala> trait Foo {
          val x:
     |   Int
          def x(i: Int):
     |   Int
     |   }
defined trait Foo
```

Type names must be unique.

Scala does not have *static* members, like Java. Instead, an `object` is used to hold members that span instances, such as constants.

If an `object` and a `class` have the same name and are defined in the same file, they are called *companions*.

Recall from Chapter 1, that when an `object` and a `class` have the same name *and* they are defined in the same

file, they are *companions*. For case classes, the compiler automatically generates a companion object for you.

Reference Versus Value Types

Java syntax models how the JVM implements data. First, there is a set of special *primitives*: `short`, `int`, `long`, `float`, `double`, `boolean`, `char`, `byte`, and the keyword `void`. They are stored on the stack or CPU registers for better performance.

All other types are called *reference types*, because all instances of them are allocated on the heap and variables that refer to these instances actually refer to the corresponding heap locations. There are no "structural" types whose instances can live on the stack, as in C and C++, although this capability is being considered for a future version of Java. Hence the name *reference type* is used to distinguish these instances from primitive values. Instances of these types are created with the `new` keyword.

Scala has to obey the JVM's rules, of course, but Scala refines the clear distinction between primitives and reference types.

All reference types are subtypes of `AnyRef`. `AnyRef` is a subtype of `Any`, the root of the Scala type hierarchy. All value types are subtypes of `AnyVal`, which is also a subtype of `Any`. These are the only two subtypes of `Any`. Note that Java's root type, `Object`, is actually closest to `AnyRef`, not `Any`.

Instances of reference types are still created using the `new` keyword. Like other methods with no arguments, we can drop the parentheses when using a constructor that takes no arguments (called a *default constructor* in some languages).

Scala follows Java conventions for literal values of the number types and `Strings`. For example, `val name = "Programming Scala"` is equivalent to `val name = new String("Programming Scala")`.

However, Scala also adds a literal syntax for tuples, `(1,2,3)`, which is equivalent to `new Tuple3(1,2,3)`. We've also seen how language features make it easy to implement custom literal syntax conventions without compiler support, such as `1 :: 2 :: Nil` for `List`s and `Map("one" -> 1, "two" -> 2)` for `Map`s.

It's common for instances of reference types to be created using objects with `apply` methods, which function as *factories*. (These methods must use `new` internally or an available literal syntax.) Because companion objects with such `apply` methods are generated automatically for case classes, instances of case classes are usually created this way.

The types `Short`, `Int`, `Long`, `Float`, `Double`, `Boolean`, `Char`, `Byte`, and `Unit` are called *value types*. They correspond to the JVM *primitives* `short`, `int`, `long`, `float`, `double`, `boolean`, `char`, `byte`, and the `void` keyword, respectively. All value types are subtypes of `AnyVal` in Scala's object model. `AnyVal` is the second of the two subtypes of `Any`.

"Instances" of value types are not created on the heap. Instead, the JVM primitive values are used instead and they are stored in registers or on the stack. Instances are always created using literal values, e.g., `1`, `3.14`, `true`. The literal value for `Unit` is `()`, but we rarely use that value explicitly.

In fact, there are no public constructors for these types, so an expression like `val i = new Int(1)` won't compile.

Hence, Scala minimizes the use of "boxed" reference types, giving us the best of both worlds, the performance of primitives without boxing with object semantics in source code.

This uniformity of syntax allows us to declare parameterized collections of value types, like `List[Int]`. In contrast, Java requires the boxed types to be used, like `List<Integer>`. This complicates library code. It's common in Java *Big Data* libraries to have a long list of custom collection types specialized for each of the primitive types, or perhaps just `long` and `double`. You'll see a class dedicated to vectors of `longs`, a class dedicated to vectors of `doubles`, and so forth. The "footprint" of the library is larger and the implementation can't exploit code reuse as well. (There are still issues with boxing and collections that we'll explore in Specialization for Value Types.)

Value Classes

As we've seen, it's common for Scala to introduce wrapper types to implement *type classes*, also called *extension methods* (see Type Class Pattern). Unfortunately, wrappers around value types effectively turn them into reference types, defeating the performance optimization of using primitives.

Scala 2.10 introduced a solution, called *value classes*, and a tandem feature called *universal traits*. These types impose limits on what can be declared, but in exchange, they don't result in heap allocations for the wrappers:

```
// src/main/scala/progscala2/basicoop/ValueClassDollar.sc

class Dollar(val value: Float) extends AnyVal {
  override def toString = "$%.2f".format(value)
}

val benjamin = new Dollar(100)
// Result: benjamin: Dollar =
$100.00
```

To be a valid value class, the following rules must be followed:

1. The value class has one and only one public `val` argument (as of Scala 2.11, the argument can also be nonpublic).
2. The type of the argument must not be a value class itself.
3. If the value class is parameterized, the `@specialized` annotation can't be used.
4. The value class doesn't define secondary constructors.
5. The value class defines only methods, but no other `val`s and no `var`s.
6. However, the value class can't override `equals` and `hashCode`.
7. The value class defines no nested `trait`s, `class`es, or `object`s.
8. The value class cannot be subclassed.
9. The value class can only inherit from *universal traits.*
10. The value class must be a top-level type or a member of an object that can be referenced.[]

That's a long list, but the compiler provides good error messages when we break the rules.

At compile time the type is the outer type, `Dollar` in this example. The runtime type is the wrapped type, e.g., `Float`.

Usually the argument is one of the `AnyVal` types, but it doesn't have to be. If we wrap a reference type, we still benefit from not allocating the wrapper on the heap, as in the following implicit wrapper for `String`s that are phone numbers:

```
// src/main/scala/progscala2/basicoop/ValueClassPhoneNumber.sc

class USPhoneNumber(val s: String) extends AnyVal {

  override def toString = {
    val digs = digits(s)
    val areaCode  = digs.substring(0,3)
    val exchange  = digs.substring(3,6)
                                          // "subscriber
    val subnumber = digs.substring(6,10)  number"
     "($areaCode) $exchange-
    s$subnumber"
  }

  private def digits(str: String): String = str.replaceAll("""\D""", "")
}

val number = new USPhoneNumber("987-654-3210")
// Result: number: USPhoneNumber = (987) 654-
3210
```

A value class can be a `case` class, but the many extra methods and the companion object generated are less likely to be used and hence more likely to waste space in the output class file.

A *universal trait* has the following properties:

1. It derives from `Any` (but not from other universal traits).
2. It defines only methods.
3. It does no initialization of its own.

Here a refined version of `USPhoneNumber` that mixes in two universal traits:

```
// src/main/scala/progscala2/basicoop/ValueClassUniversalTraits.sc

trait Digitizer extends Any {
  def digits(s: String): String = s.replaceAll("""\D""", "")          // ❶
}

trait Formatter extends Any {                                          //
❷
  def format(areaCode: String, exchange: String, subnumber: String): String =
     "($areaCode) $exchange-
    s$subnumber"
}

class USPhoneNumber(val s: String) extends AnyVal
    with Digitizer with Formatter {

  override def toString = {
    val digs = digits(s)
    val areaCode = digs.substring(0,3)
    val exchange = digs.substring(3,6)
    val subnumber  = digs.substring(6,10)
    format(areaCode, exchange, subnumber)                              // ❸
  }
}

val number = new USPhoneNumber("987-654-3210")
// Result: number: USPhoneNumber = (987) 654-
3210
```

❶ `Digitizer` is a trait that contains the `digits` method we originally had in `USPhoneNumber`.

❷ `Formatter` formats the phone number the way we want it.

❸ Use `Formatter.format`.

`Formatter` actually solves a design problem. We might like to specify a second argument to `USPhoneNumber` for a format string or some other mechanism for configuring the actual format produced by `toString`, because there are many popular format conventions. However, we're only allowed to pass one argument to `USPhoneNumber`, but we can mix in universal traits to do the configuration we want!

However, universal traits do sometimes trigger instantiation (i.e., heap allocation of an instance) by Scala, due to limitations of the JVM. Here's a summary of the circumstances requiring instantiation:

1. When a value class instance is passed to a function expecting a universal trait implemented by the instance. However, if a function expects an instance of the value class itself, instantiation isn't required.
2. A value class instance is assigned to an `Array`.
3. The type of a value class is used as a type parameter.

For example, when the following method is called with a `USPhoneNumber`, an instance of it will have to be allocated:

```
def toDigits(d: Digitizer, str: String) = d.digits(str)
...
val digs = toDigits(new USPhoneNumber("987-654-3210"), "123-Hello!-456")
// Result: digs: String =
123456
```

Also, when the following parameterized method is passed a `USPhoneNumber`, an instance of `USPhoneNumber` will have to be allocated:

```
def print[T](t: T) = println(t.toString)
print(new USPhoneNumber("987-654-3210"))
// Result: (987) 654-
3210
```

To summarize, value classes provide a low-overhead technique for defining extension methods (type classes) and for defining types with meaningful domain names (like `Dollar`) that exploit the type safety of the underlying value.

Note

The term *value type* refers to the `Short`, `Int`, `Long`, `Float`, `Double`, `Boolean`, `Char`, `Byte`, and `Unit` types Scala has had for a long time. The term *value class* refers to the new construct for defining custom classes that derive from `AnyVal`.

For more information on the implementation details of value classes, see SIP-15: Value Classes. *SIP* stands for *Scala Improvement Process*, the community mechanism for proposing new language and library features.

Parent Types

Derivation of *child* or *derived* types from a *parent* or *base* type is a core principle of most object-oriented languages. It's a mechanism for reuse, encapsulation, and polymorphic behavior (behavior that varies depending on the instance's actual type in a type hierarchy).

Like Java, Scala supports single inheritance, not multiple inheritance. A child (or derived) class can have one and only one parent (or base) class. The sole exception is the root of the Scala class hierarchy, `Any`, which has no parent.

We've already seen several examples of parent and child classes. Here are snippets of one of the first we saw, in Abstract Types Versus Parameterized Types, which demonstrates the use of type members. Here are the most important details again:

```
abstract class BulkReader {
  type In
  val source: In
                    // Read source and return a
  def read: String  String
}

class StringBulkReader(val source: String) extends BulkReader {
  type In = String
  def read: String = source
}

class FileBulkReader(val source: java.io.File) extends BulkReader {
  type In = java.io.File
  def read: String = {...}
}
```

As in Java, the keyword `extends` indicates the parent class, in this case `BulkReader`. In Scala, `extends` is also used when a class inherits a trait as its parent (even when it mixes in other traits using the `with` keyword). Also, `extends` is used when one trait is the child of another trait or class. Yes, traits can inherit classes.

If we don't `extend` a parent class, the default parent is `AnyRef`.

Constructors in Scala

Scala distinguishes between a *primary constructor* and zero or more *auxiliary constructors*, also called *secondary constructors*. In Scala, the primary constructor is the entire body of the class. Any parameters that the constructor requires are listed after the class name. `StringBulkReader` and `FileBulkReader` are examples.

Let's revisit some simple case classes, `Address` and `Person`, that we saw in Chapter 5 and consider enhancements using secondary constructors:

```
//
src/main/scala/progscala2/basicoop/PersonAuxConstructors.scala
package progscala2.basicoop

case class Address(street: String, city: String, state: String, zip: String) {

  def this(zip: String) =                                              // ❶
    this("[unknown]", Address.zipToCity(zip), Address.zipToState(zip), zip)
}

object Address {

  def zipToCity(zip: String)  = "Anytown"                              // ❷
  def zipToState(zip: String) = "CA"
}

case class Person(
  name: String, age: Option[Int], address: Option[Address]) {          // ❸

  def this(name: String) = this(name, None, None)                      // ❹

  def this(name: String, age: Int) = this(name, Some(age), None)

  def this(name: String, age: Int, address: Address) =
    this(name, Some(age), Some(address))

  def this(name: String, address: Address) = this(name, None, Some(address))
}
```

❶

A secondary constructor that takes just a zip code argument. It calls helper methods to infer the city and state, but it can't infer the street.

❷

Helper functions that look up the city and state from the zip code (or at least they pretend to do that).

❸

Make the person's age and address optional.

❹

Provide convenient auxiliary constructors that let the user specify some or all of the values.

Note that an auxiliary constructor is named `this` and it must call the primary constructor or another auxiliary constructor as its first expression. The compiler also requires that the constructor called is one that appears *earlier* in the source code. So, we must order secondary constructors carefully in our code.

By forcing all construction to go through the primary constructor (eventually), code duplication is minimized and initialization logic for new instances is always uniformly applied.

The auxiliary constructor for `Address` is a good example of a method that does something nontrivial, rather than just provide convenient alternative invocations, like `Person`'s auxiliary constructors.

This file is compiled by `sbt`, so we can use the types in the following script:

```
// src/main/scala/progscala2/basicoop/PersonAuxConstructors.sc
import progscala2.basicoop.{Address, Person}

                     "1 Scala
val a1 = new Address(Lane"          , "Anytown", "CA", "98765")
// Result: Address(1 Scala
Lane,Anytown,CA,98765)

val a2 = new Address("98765")
// Result:
Address([unknown],Anytown,CA,98765)

           "Buck
new Person(Trends1"       )
// Result: Person(Buck
Trends1,None,None)

           "Buck
new Person(Trends2"       , Some(20), Some(a1))
// Result: Person(Buck
Trends2,Some(20),
//           Some(Address(1 Scala
Lane,Anytown,CA,98765)))


           "Buck
new Person(Trends3"       , 20, a2)
// Result: Person(Buck
Trends3,Some(20),
//
Some(Address([unknown],Anytown,CA,98765)))

           "Buck
new Person(Trends4"       , 20)
// Result: Person(Buck
Trends4,Some(20),None)
```

This code works well enough, but actually there are a few issues with it. First, `Person` now has a lot of boilerplate for the auxiliary constructors. We already know that we can define method arguments with default values and the user can name the arguments when calling the methods.

Let's reconsider `Person`. First, let's add default values for `age` and `address` and assume that it's not "burdensome" for the user to specify `Some(…)` values:

```
//
src/main/scala/progscala2/basicoop/PersonAuxConstructors2.sc
import progscala2.basicoop.Address

                     "1 Scala
val a1 = new Address(Lane"         , "Anytown", "CA", "98765")
val a2 = new Address("98765")

case class Person2(
  name: String,
  age: Option[Int] = None,
  address: Option[Address] = None)

            "Buck
new Person2(Trends1"      )
// Result: Person2 = Person2(Buck
Trends1,None,None)

            "Buck
new Person2(Trends2"      , Some(20), Some(a1))
// Result: Person2(Buck
Trends2,Some(20),
//           Some(Address(1 Scala
Lane,Anytown,CA,98765)))

            "Buck
new Person2(Trends3"      , Some(20))
// Result: Person2(Buck
Trends3,Some(20),None)

            "Buck
new Person2(Trends4"      , address = Some(a2))
// Result: Person2(Buck
Trends4,None,
//
Some(Address([unknown],Anytown,CA,98765)))
```

The user of `Person` writes a little more code, but the reduced maintenance burden on the library developer is an important benefit. Trade-offs…

Let's decide we really prefer to maximize the user-friendly options. The second issue with our implementation is that the user has to create instances with `new`. Perhaps you noticed that the examples used `new` to construct instances.

Try removing the `new` keywords and see what happens. Unless you're invoking the primary constructor, you get a compiler error.

Warning

The compiler does not automatically generate `apply` methods for secondary constructors in case classes.

However, if we overload `Person.apply` in the companion object, we can have our convenient "constructors" and avoid the requirement to use `new`. Here is our final implementation of `Person`, called `Person3`:

```
// src/main/scala/progscala2/basicoop/PersonAuxConstructors3.scala
package progscala2.basicoop3
import progscala2.basicoop.Address

case class Person3(
  name: String,
  age: Option[Int] = None,
  address: Option[Address] = None)

object Person3 {

  // Because we are overloading a normal method (as opposed to
  constructors),
  // we must specify the return type annotation, Person3 in this
  case.
  def apply(name: String): Person3 = new Person3(name)

  def apply(name: String, age: Int): Person3 = new Person3(name, Some(age))

  def apply(name: String, age: Int, address: Address): Person3 =
    new Person3(name, Some(age), Some(address))

  def apply(name: String, address: Address): Person3 =
    new Person3(name, address = Some(address))
}
```

Note that overloaded methods like `apply` that aren't constructors must have an explicit return type annotation.

Finally, here is a script that uses the final types:

```
//
src/main/scala/progscala2/basicoop/PersonAuxConstructors3.sc
import progscala2.basicoop.Address
import progscala2.basicoop3.Person3

                     "1 Scala
val a1 = new Address(Lane"         , "Anytown", "CA", "98765")
val a2 = new Address("98765")

        "Buck
Person3(Trends1"      )
// Primary
// Result: Person3(Buck
Trends1,None,None)

        "Buck
Person3(Trends2"      , Some(20), Some(a1))
// Primary
// Result: Person3(Buck
Trends2,Some(20),
//           Some(Address(1 Scala
Lane,Anytown,CA,98765)))

        "Buck
Person3(Trends3"      , 20, a1)
// Result: Person3(Buck
Trends3,Some(20),
//           Some(Address(1 Scala
Lane,Anytown,CA,98765)))

        "Buck
Person3(Trends4"      , Some(20))
// Primary
// Result: Person3(Buck
Trends4,Some(20),None)

        "Buck
Person3(Trends5"      , 20)
// Result: Person3(Buck
Trends5,Some(20),None)

        "Buck
Person3(Trends6"      , address = Some(a2))
// Primary
// Result: Person3(Buck
Trends6,None,
//
Some(Address([unknown],Anytown,CA,98765)))

        "Buck
Person3(Trends7"      , address = a2)
// Result: Person3(Buck
Trends7,None,
//
Some(Address([unknown],Anytown,CA,98765)))
```

All examples with the `Primary` comment call the primary `apply` method generated automatically as part of the `case` class. The other examples without the comment call one of the other `apply` methods.

In fact, it's not all that common to define auxiliary constructors in Scala code, because alternative techniques generally work better for minimizing boilerplate while still providing users with flexible construction options. Instead, make judicious use of Scala's support for named and optional parameters, and use overloaded `apply` "factory" methods in objects.

Fields in Classes

We started the chapter with a reminder that the primary constructor arguments become instance fields if they are prefixed with the `val` or `var` keyword. For case classes, `val` is assumed. This convention greatly reduces source-code boilerplate, but how does it translate to byte code?

Actually, Scala just does implicitly what Java code does explicitly. There is a private field created internal to the class and the equivalent of "getter" and "setter" accessor methods are generated. Consider this simple Scala class:

```
class Name(var value: String)
```

Conceptually, it is equivalent to this code:

```
class Name(s: String) {
  private var _value: String = s                          //
❶

  def value: String = _value
//  ❷

  def value_=(newValue: String): Unit = _value = newValue  //
❸
}
```

❶

Invisible field, declared mutable in this case.

❷

The "getter" or reader method.

❸

The "setter" or writer method.

Note the convention used for the `value_=` method name. When the compiler sees a method named like this, it will allow client code to drop the `_`, effectively enabling infix notation as if we were setting a bare field in the object:

```
scala> val name = new Name("Buck")
name: Name = Name@2aed6fc8

scala> name.value
res0: String = Buck

scala> name.value_=("Bubba")
name.value: String = Bubba

scala> name.value
res1: String = Bubba

scala> name.value = "Hank"
name.value: String = Hank

scala> name.value
res2: String = Hank
```

If we declare a field immutable with the `val` keyword, the writer method is not synthesized, only the reader method.

You can follow these conventions yourself, if you want to implement custom logic inside reader and writer methods.

We can pass constructor arguments to noncase classes that aren't intended to become fields. Just omit both the `val` and `var` keywords. For example, we might pass an argument needed to construct an instance, but we want to discard it afterwards.

Note that the value is still in the scope of the class body. As we saw in earlier examples of implicit conversion classes, they referred to the argument used to construct the instances, but most of them did not declare the argument to be a field of the instance. For example, recall our `Pipeline` example from Phantom Types:

```
object Pipeline {
  implicit class toPiped[V](value:V) {
    def |>[R] (f : V => R) = f(value)
  }
}
```

While `toPiped` refers to `value` in the `|>` method, `value` is not a field. Whether or not the constructor arguments are declared as fields with `val` or `var`, the arguments are visible in the entire class body. Hence they can be used by members of the type, such as methods. Compare with constructors as defined in Java and most other OO languages. Because the constructors themselves are methods, the arguments passed to them are not visible outside those methods. Hence, the arguments must be "saved" as fields, either public or hidden.

Why not just always make these arguments fields? A field is visible to clients of the type (that is, unless it's declared `private` or `protected`, as we'll discuss in Chapter 13). Unless these arguments are really part of the logical state exposed to users, they shouldn't be fields. Instead, they are effectively private to the class body.

The Uniform Access Principle

You might wonder why Scala doesn't follow the convention of the JavaBeans Specification that reader and writer methods for a field `value` are named `getValue` and `setValue`, respectively. Instead, Scala chooses to follow the *Uniform Access Principle*.

As we saw in our `Name` example, it appears that clients can read and write the "bare" `value` field without going through accessor methods, but in fact they are calling methods. On the other hand, we could just declare a field in the class body with the default public visibility and then access it as a bare field:

```
class Name2(s: String) {
  var value: String = s
}
```

Now `value` is public and the accessor methods are gone.

Let's try it:

```
scala> val name2 = new Name2("Buck")
name2: Name2 = Name2@303becf6

scala> name2.value
res0: String = Buck

scala> name2.value_=("Bubba")
name2.value: String = Bubba

scala> name2.value
res1: String = Bubba
```

Note that user "experience" is identical. The user's code is agnostic about the implementation, so we are free to change the implementation from bare field access to accessor methods when necessary; for example, if we want to add some sort of validation on writes or lazily construct a result on reads, for efficiency. Conversely, we can replace accessor methods with public visibility of the field, to eliminate the overhead of a method call (though the JVM will probably eliminate that overhead anyway).

Therefore, the Uniform Access Principle has an important benefit in that it minimizes how much client code has to know about the internal implementation of a class. We can change that implementation without forcing client code changes, although a recompilation is required.

Scala implements this principle without sacrificing the benefits of access protections or the occasional need to perform additional logic besides just reading or writing a value.

Note

Scala doesn't use Java-style getter and setter methods. Instead, it supports the Uniform Access Principle, where the syntax for reading and writing a "bare" field looks the same as the syntax for calling methods to read and write it, indirectly.

However, sometimes we need JavaBeans-style accessor methods for interoperability with Java libraries. We can annotate classes with the `scala.reflect.BeanProperty` or the `BooleanBeanProperty` annotation. See JavaBean Properties for more details.

Unary Methods

We saw how the compiler lets us define an assignment method `foo_=` for field `foo`, then use it with the convenient

`myinstance.foo = value` syntax. There's one other kind of operator we haven't seen how to implement, *unary operators*.

An example is negation. If we implement a complex number class, how would we support the negation of some instance `c`, i.e., `-c`? Here's how:

```
// src/main/scala/progscala2/basicoop/Complex.sc

case class Complex(real: Double, imag: Double) {
  def unary_- : Complex = Complex(-real, imag)
// ❶
  def -(other: Complex) = Complex(real - other.real, imag - other.imag)
}

val c1 = Complex(1.1, 2.2)
val c2 = -c1                          // Complex(-1.1, 2.2)
val c3 = c1.unary_-                   // Complex(-1.1, 2.2)
val c4 = c1 - Complex(0.5, 1.0)       // Complex(0.6, 1.2)
```

❶

The method name is `unary_X`, where `X` is the operator character we want to use, `-` in this case. Note the space between the `-` and the `:`. This is necessary to tell the compiler that the method name ends with `-` and not `:`! For comparison, we also implement the usual minus operator.

Once we've defined a unary operator, we can place it *before* the instance, as we did when defining `c2`. We can also call it like any other method, as we did for `c3`.

Validating Input

What if we want to validate the input arguments to ensure that the resulting instances have a valid state? `Predef` defines a useful set of overloaded methods called `require` that are useful for this purpose. Consider this class that encapsulates US zip codes. Two forms are allowed, a five-digit number and a "zip+4" form that adds an additional four digits. This form is usually written "12345-6789". Also, not all numbers correspond to real zip codes:

```
// src/main/scala/progscala2/basicoop/Zipcode.scala
package progscala2.basicoop

case class ZipCode(zip: Int, extension: Option[Int] = None) {
  require(valid(zip, extension),
// ❶
     "Invalid Zip+4 specified:
    s$toString"                          )

  protected def valid(z: Int, e: Option[Int]): Boolean = {
    if (0 < z && z <= 99999) e match {
      case None    => validUSPS(z, 0)
      case Some(e) => 0 < e && e <= 9999 && validUSPS(z, e)
    }
    else false
  }

  /** Is it a real US Postal Service zip code?
  */
  protected def validUSPS(i: Int, e: Int): Boolean = true            //
❷

  override def toString =
// ❸
    if (extension != None) s"$zip-${extension.get}" else zip.toString
}

object ZipCode {
  def apply (zip: Int, extension: Int): ZipCode =
    new ZipCode(zip, Some(extension))
}
```

❶

Use the `require` method to validate input.

❷

A real implementation would check a USPS-sanctioned database to verify that the zip code actually exists.

❸

Override `toString` to return the format people expect for zip codes, with proper handling of the optional four-digit extension.

Here is a script that uses it:

```
// src/main/scala/progscala2/basicoop/Zipcode.sc
import progscala2.basicoop.ZipCode

ZipCode(12345)
// Result: ZipCode =
12345

ZipCode(12345, Some(6789))
// Result: ZipCode = 12345-
6789

ZipCode(12345, 6789)
// Result: ZipCode = 12345-
6789

try {
                    // Invalid Zip+4 specified: 0-
  ZipCode(0, 6789)  6789
} catch {
  case e: java.lang.IllegalArgumentException => e
}

try {
  ZipCode(12345, 0)
// Invalid Zip+4 specified: 12345-
0
} catch {
  case e: java.lang.IllegalArgumentException => e
}
```

One very good reason for defining domain-specific types like `ZipCode` is the ability to do validation of values once, during construction, so that users of `ZipCode` instances know that no further validation is required.

There are also `ensuring` and `assume` methods in `Predef` for similar purposes. We'll explore more uses for `require` and these two *assertion* methods in Better Design with Design by Contract.

Although we discussed validation in the context of construction, we can call these assertion methods inside any methods. However, an exception is the class bodies of value classes. The assertion checks can't be used there, otherwise a heap allocation would be required. However, `ZipCode` can't be a value class anyway, because it takes a second constructor argument.

Calling Parent Class Constructors (and Good Object-Oriented Design)

The primary constructor in a derived class must invoke one of the parent class constructors, either the primary constructor or an auxiliary constructor. In the following example, `Employee` is a subclass of `Person`:

```
// src/main/scala/progscala2/basicoop/EmployeeSubclass.sc
import progscala2.basicoop.Address

                    // This was Person2 previously, now
case class Person(  renamed.
  name: String,
  age: Option[Int] = None,
```

```
  address: Option[Address] = None)

class Employee(                                                    //
❶
  name: String,
  age: Option[Int] = None,
  address: Option[Address] = None,
  val title: String = "[unknown]",                                 // ❷
  val manager: Option[Employee] = None) extends Person(name, age, address) {

  override def toString =                                          //
❸
     "Employee($name, $age, $address, $title,
    s$manager)"
}

                    "1 Scala
val a1 = new Address(Lane"         , "Anytown", "CA", "98765")
val a2 = new Address("98765")

val ceo = new Employee("Joe CEO", title = "CEO")
// Result: Employee(Joe CEO, None, None, CEO,
None)

             "Buck
new Employee(Trends1"      )
// Result: Employee(Buck Trends1, None, None, [unknown],
None)

             "Buck
new Employee(Trends2"      , Some(20), Some(a1))
// Result:  Employee(Buck Trends2,
Some(20),
//            Some(Address(1 Scala Lane,Anytown,CA,98765)), [unknown],
None)

             "Buck
new Employee(Trends3"      , Some(20), Some(a1), "Zombie Dev")
// Result:  Employee(Buck Trends3,
Some(20),
//            Some(Address(1 Scala Lane,Anytown,CA,98765)), Zombie Dev,
None)

             "Buck
new Employee(Trends4"      , Some(20), Some(a1), "Zombie Dev", Some(ceo))
// Result:  Employee(Buck Trends4,
Some(20),
//            Some(Address(1 Scala Lane,Anytown,CA,98765)), Zombie
Dev,
//            Some(Employee(Joe CEO, None, None, CEO,
None)))
```

❶

`Employee` is declared a regular class, not a `case` class. We'll explain why in the next section.

❷

The new fields, `title` and `manager`, require the `val` keyword because `Employee` isn't a `case` class. The other arguments are already fields, from `Person`. Note that we also call `Person`'s primary constructor.

❸

Override `toString`. Otherwise, `Person.toString` would be used.

In Java, we would define constructor methods and call `super` in them to invoke the parent class initialization logic. In Scala, we implicitly invoke the parent class constructor through the `ChildClass(…) extends ParentClass(…)` syntax.

Note

Although `super` can be used to invoke overridden methods, as in Java, it cannot be used to invoke a superclass constructor.

Good Object-Oriented Design: A Digression

This code *smells*. The declaration of `Employee` mixes `val` keywords or no keywords in the argument list. But deeper problems lurk behind the source code.

We can derive a noncase class from a case class or the other way around, but we can't derive one case class from another. This is because the autogenerated implementations of `toString`, `equals`, and `hashCode` do not work properly for subclasses, meaning they ignore the possibility that an instance could actually be a derived type of the case class type.

This is actually by design; it reflects the problematic aspects of subclassing. For example, should an `Employee` instance and a `Person` instance be considered equal if both have the same name, age, and address? A more flexible interpretation of object equality would say yes, while a more restrictive version would say no. In fact, the mathematical definition of equality requires commutative behavior: `someEmployee == somePerson` should return the same result as `somePerson == someEmployee`. The more flexible interpretation would break associativity, because you would never expect an `Employee` instance to think it's equal to a `Person` instance that is not an `Employee`.

Actually, the problem of `equals` is even worse here, because `Employee` doesn't override the definitions of `equals` and `hashCode`. We're effectively treating all `Employee` instances as `Person` instances.

That's dangerous for small types like this. It's inevitable that someone will create a collection of employees, where they will try to sort the employees or use an employee as a key in a hash map. Because `Person.equals` and `Person.hashCode` will get used, respectively, anomalous behavior will occur when we have two people named John Smith, one of whom is the CEO while the other works in the mail room. The occasional confusion between the two will happen just often enough to be serious, but not often enough to be easily repeatable for finding and fixing the bug!

The real problem is that we are subclassing *state*. That is, we are using inheritance to add additional state contributions, `title` and `manager` in this case. In contrast, subclassing *behavior* with the *same* state fields is easier to implement robustly. It avoids the problems with `equals` and `hashCode` just described, for example.

Of course, these problems with inheritance have been known for a long time. Today, good object-oriented design

favors *composition over inheritance*, where we compose units of functionality rather than build class hierarchies.

As we'll see in the next chapter, *traits* make composition far easier to use than Java interfaces, at least before Java 8. Hence, the examples in the book that aren't "toys" won't use inheritance that adds *state*. Such inheritance hierarchies are also very rare in production-quality Scala libraries, fortunately.

Hence, the Scala team could have made a choice to implement subclass-friendly versions of `equals`, `hashCode`, and `toString`, but that would have added extra complexity to support a bad design choice. Case classes provide convenient, simple domain types, with pattern matching and decomposition of instances of these types. Supporting inheritance hierarchies is not their purpose.

When inheritance is used, the following rules are recommended:

1. An abstract base class or trait is subclassed one level by concrete classes, including case classes.
2. Concrete classes are never subclassed, except for two cases:
    a. Classes that mix in other behaviors defined in `trait`s (see Chapter 9). Ideally, those behaviors should be *orthogonal*, i.e., not overlapping.
    b. Test-only versions to promote automated unit testing.
3. When subclassing seems like the right approach, consider partitioning behaviors into traits and mix in those traits instead.
4. Never split logical state across parent-child type boundaries.

By "logical" state in the last bullet, I mean we might have some private, implementation-specific state that doesn't affect the externally visible, logical behavior of equality, hashing, etc. For example, our library might include special subtypes of our collections that add private fields to implement caching or logging behaviors (when a mixin trait for such features is not a good option).

So, what about `Employee`? If subclassing `Person` to create `Employee` is bad, what should we do instead? The answer really depends on the context of use. If we're implementing a Human Resources application, do we need a separate concept of `Person` or can `Employee` just be the base type, declared as a `case` class? Do we even need *any* types for this at all? If we're processing a result set from a database query, is it sufficient to use tuples or other containers to hold the fields returned from the query? Can we dispense with the "ceremony" of declaring a type altogether?

Let's just suppose we really need separate concepts of `Person` and `Employee`. Here's one way I would do it:

```
// src/main/scala/progscala2/basicoop/PersonEmployeeTraits.scala
package progscala2.basicoop2                                          //
❶

case class Address(street: String, city: String, state: String, zip: String)

object Address {
  def apply(zip: String) =                                            //
❷
    new Address(
      "[unknown]", Address.zipToCity(zip), Address.zipToState(zip), zip)

  def zipToCity(zip: String)  = "Anytown"
  def zipToState(zip: String) = "CA"
}

trait PersonState {
// ❸
  val name: String
  val age: Option[Int]
  val address: Option[Address]

  // Some common methods declared/defined
  here?
}

case class Person(
// ❹
  name: String,
  age: Option[Int] = None,
  address: Option[Address] = None) extends PersonState

trait EmployeeState {
// ❺
  val title: String
  val manager: Option[Employee]
}

case class Employee(
// ❻
  name: String,
  age: Option[Int] = None,                                            //
❼
  address: Option[Address] = None,
  title: String = "[unknown]",
  manager: Option[Employee] = None)
extends PersonState with EmployeeState
```

❶

Use a different package because earlier versions of some of these types are in package `oop`.

❷

Previously, `Address` had an auxiliary constructor. Now we use a second factory method.

❸

Define a `trait` for the state we *want* a person to have. You could pick a naming convention you like better than `PersonState`.

❹

When we just have `Person` instances, use this case class, which implements `PersonState`.

❺

Use the same technique for `Employee`, although it's less useful to declare a separate trait and case class for `Employee`. Still, consistency has its merits. The drawback is the extra "ceremony" we've introduced with separate traits and case classes.

❻

The `Employee` case class.

❼

Note that we have to define the default values twice for the fields shared between `Person` and `Employee`. That's a slight disadvantage (unless we actually need that flexibility).

Note that `Employee` is no longer a subclass of `Person`, but it is a subclass of `PersonState`, because it mixes in that trait. Also, `EmployeeState` is not a subclass of `PersonState`. Figure 8-1 is a class diagram to illustrate the relationships:

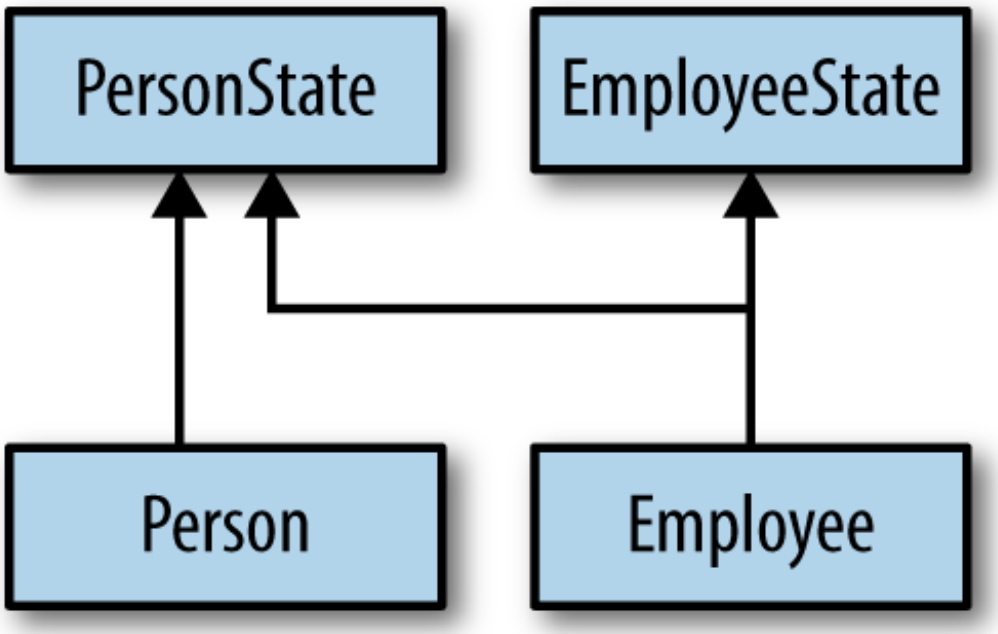

Figure 8-1. Class diagram for PersonState, Person, EmployeeState, and Employee

Note that both `Person` and `Employee` mix in traits, but `Employee` doesn't subclass another concrete class.

Let's try creating some objects:

```
// src/main/scala/progscala2/basicoop/PersonEmployeeTraits.sc
import progscala2.basicoop.{ Address, Person, Employee }

                           "1 Scala
val ceoAddress  = Address(Lane"          , "Anytown", "CA", "98765")
// Result: ceoAddress: oop2.Address = Address(1 Scala
Lane,Anytown,CA,98765)

val buckAddress = Address("98765")
// Result: buckAddress: oop2.Address =
Address([unknown],Anytown,CA,98765)

val ceo = Employee(
  name = "Joe CEO", title = "CEO", age = Some(50),
  address = Some(ceoAddress), manager = None)
// Result: ceo: oop2.Employee = Employee(Joe
CEO,Some(50),
//            Some(Address(1 Scala
Lane,Anytown,CA,98765)),CEO,None)

                        "Jane
val ceoSpouse = Person(Smith"      , address = Some(ceoAddress))
// Result: ceoSpouse: oop2.Person = Person(Jane
Smith,None,
//            Some(Address(1 Scala
Lane,Anytown,CA,98765)))

val buck = Employee(
  name = "Buck Trends", title = "Zombie Dev", age = Some(20),
  address = Some(buckAddress), manager = Some(ceo))
// Result: buck: oop2.Employee = Employee(Buck
Trends,Some(20),
//             Some(Address([unknown],Anytown,CA,98765)),Zombie
Dev,
//             Some(Employee(Joe
CEO,Some(50),
//               Some(Address(1 Scala
Lane,Anytown,CA,98765)),CEO,None)))

                         "Ann
val buckSpouse = Person(Collins"     , address = Some(buckAddress))
// Result: buckSpouse: oop2.Person = Person(Ann
Collins,None,
//
Some(Address([unknown],Anytown,CA,98765)))
```

You'll notice I used named arguments for several declarations. When a constructor or other method takes a lot of arguments, I like using named arguments to make it clear what each argument means. It also helps avoid bugs when several arguments have the same type and it would be easy to switch values. Of course, you should try to avoid these risks by keeping the number of arguments small and making their types unique.

Now that I've whetted your appetite for traits, the next chapter explores them in depth. But first, we have one final topic to cover.

Nested Types

Scala lets us nest type declarations and definitions. For example, it's common to define type-specific exceptions and other useful types in an object. Here is a sketch of a possible database layer:

```
//
src/main/scala/progscala2/basicoop/NestedTypes.scala

object Database {
// ❶
  case class ResultSet(/*...*/)
// ❷
  case class Connection(/*...*/)
// ❸

  case class DatabaseException(message: String, cause: Throwable) extends
    RuntimeException(message, cause)

  sealed trait Status
// ❹
  case object Disconnected extends Status
  case class  Connected(connection: Connection)  extends Status
  case class  QuerySucceeded(results: ResultSet) extends Status
  case class  QueryFailed(e: DatabaseException)  extends Status
}

class Database {
  import Database._

  def connect(server: String): Status = ???
// ❺
  def disconnect(): Status = ???

  def query(/*...*/): Status = ???
}
```

❶

A simplified interface to databases.

❷

Encapsulate query *result sets*. We elided the details that we don't care about for this sketch.

❸

Encapsulate connection pools and other information.

❹

Use a sealed hierarchy for the status; all allowed values are defined here. Use `case object`s when instances don't actually carry any additional state information. These objects behave like "flags" indicating a state.

❺

The `???` is an actual method defined in `Predef`. It simply throws an exception. It is used to mark methods as unimplemented. It is a relatively recent introduction to the library.

Tip

Consider using `case object` when a case class doesn't have any fields representing additional state information.

The `???` method is very handy for providing a placeholder implementation of a method when code is under development. The code compiles, but you can't call the method!

There is one "hole" I've found with `case object`s. Consider this session:

```
scala> case object Foo
defined object Foo

scala> Foo.hashCode
res0: Int = 70822

scala> "Foo".hashCode
res1: Int = 70822
```

Apparently, the generated `hashCode` for the `case object` simply hashes the object's name. The object's package is ignored as are any fields in the object. This means that `case object`s are risky in contexts where a strong `hashCode` implementation is needed.

Warning

Avoid using `case object`s in contexts where a strong `hashCode` is required, such as keys for hash-based maps and sets.

Recap and What's Next

We filled in the details for the basics of Scala's object model, including constructors, inheritance, and nesting of types. We also digressed at times on the subject of good object-oriented design, in Scala or any language.

We also set the stage for diving into `trait`s, Scala's enhancement to Java interfaces, which provide a powerful tool for composing behaviors from constituent parts, without resorting to inheritance and its drawbacks. In the next chapter we'll complete our understanding of traits and how to use them to solve various design problems.