

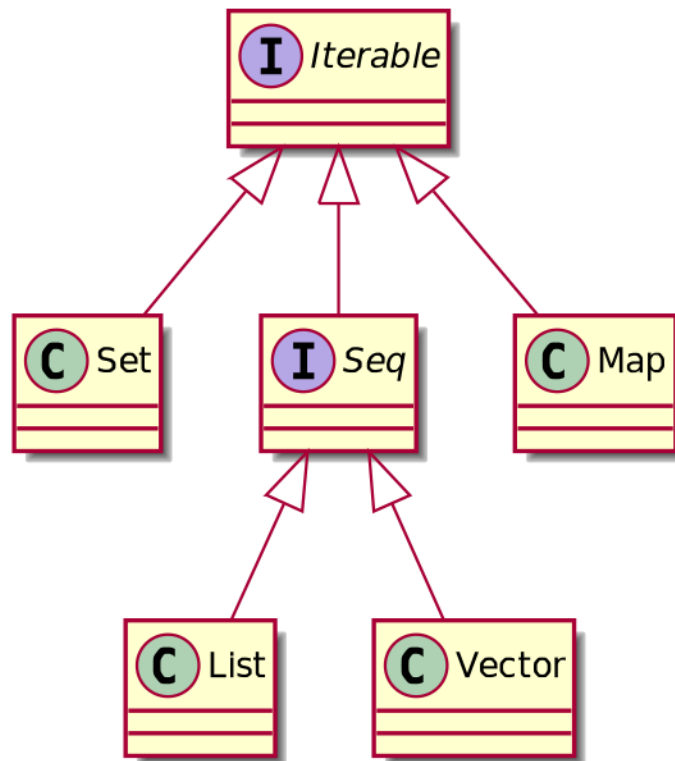


Extending and Refining Classes

Effective Programming in Scala

Standard Collections

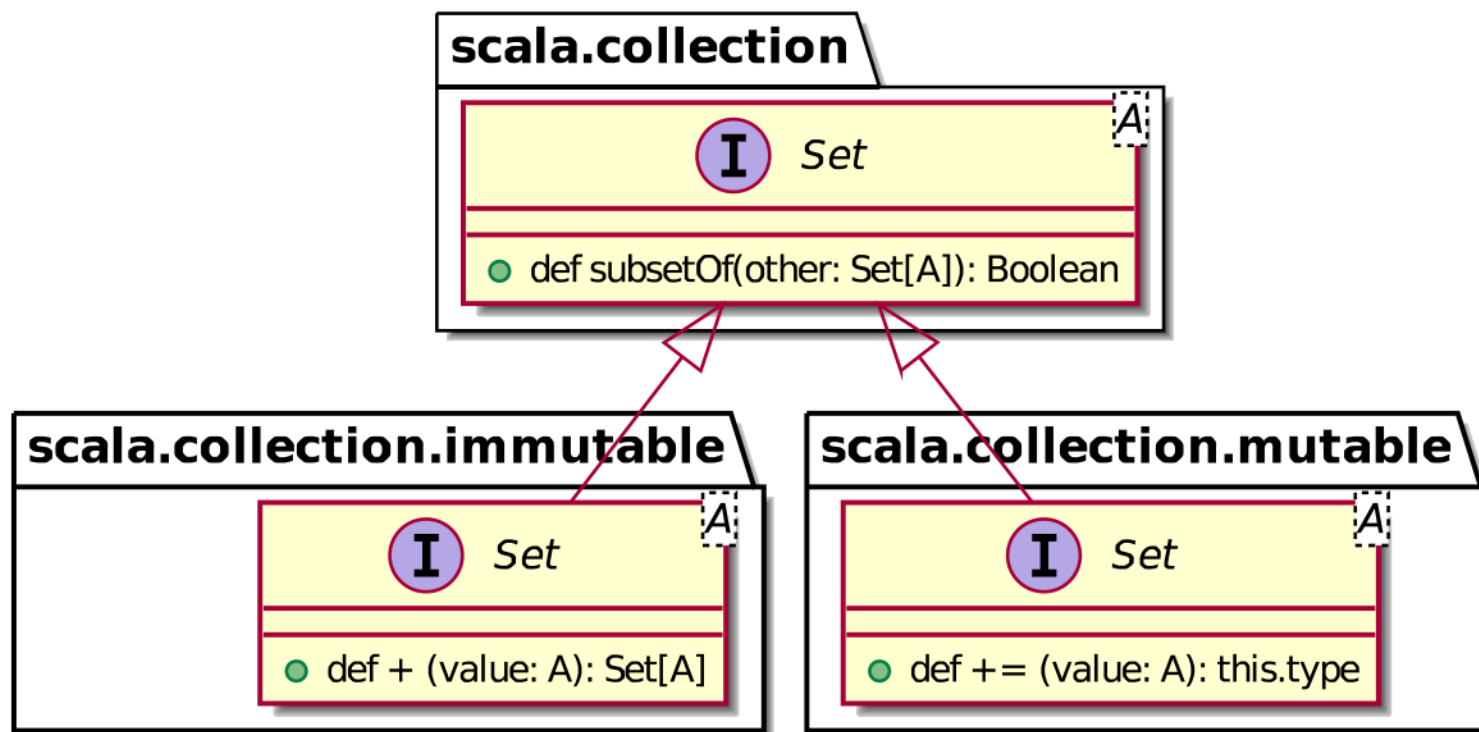
Traits and classes can form rich hierarchies. An example is the standard collections:



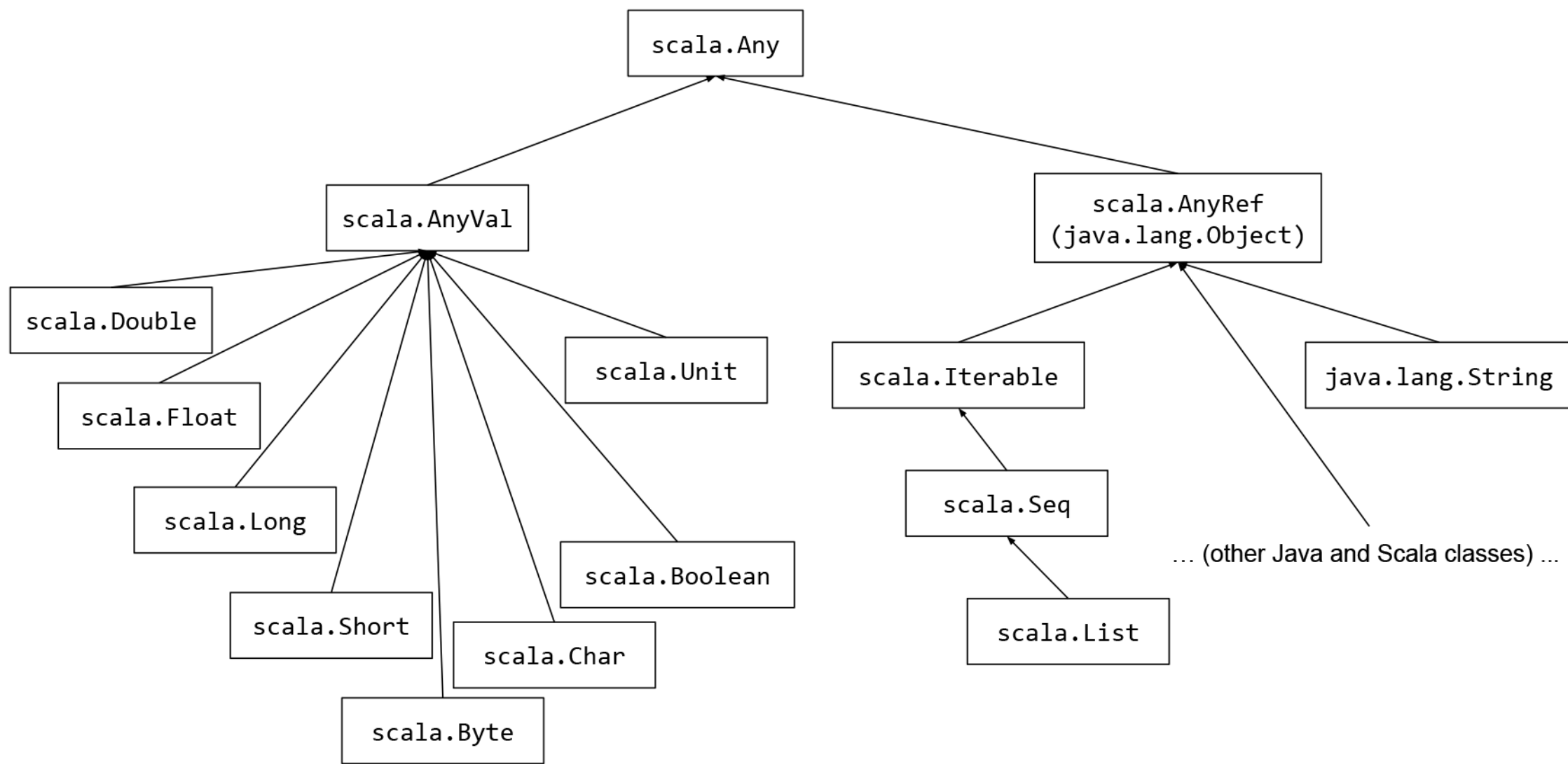
Each subtype may introduce more specific methods.

Standard Collections (2)

Furthermore, each collection type comes in two variants, immutable and mutable. Both variants have a more general parent type. For example, with Set:



Scala's Class Hierarchy



Top Types

At the top of the type hierarchy we find:

Any the base type of all types

Methods: '==', '!=', 'equals', 'hashCode', 'toString'

AnyRef The base type of all reference types;
Alias of 'java.lang.Object'

AnyVal The base type of all primitive types.

Mix-Ins

A class or object definition can extend several traits:

```
trait Logging:  
  val logger = ...
```

```
class InMemoryDatabase extends DatabaseAccess, Logging:  
  def readData(): List[Data] =  
    ...  
    logger.debug("...")  
    ...
```

Overriding Members

You can **override** the implementation of inherited members. Here is an example taken from the library ScalaTest:

```
import org.scalatest.BeforeAndAfterAll

trait Resources extends Suite, BeforeAndAfterAll:
  val resources = acquire(...)

  override def afterAll(): Unit =
    resources.close()
    super.afterAll()
```

final Members

You can prevent a member from being overridden by qualifying it with final.

For example, to guarantee that no subclasses can break the behavior of a method:

```
trait HttpClient:  
  
  def send(request: Request): Response  
  
  /** Shorthand for sending a GET request */  
  final def get(uri: String): Response =  
    send(Request("GET", uri))
```


this

Within a class definition, you can use the identifier `this` to refer to the instance being defined:


```
trait HttpClient:  
  
  def send(request: Request): Response  
  
  /** Shorthand for sending a GET request */  
  final def get(uri: String): Response =  
    this.send(Request("GET", uri))
```

Advanced Object Oriented Programming

In this course, we have introduced traits with abstract methods as a way to define interfaces, and classes that extend these traits as a way to provide concrete implementations.

Scala supports other variations around traits and classes, such as abstract classes, traits parameters, or secondary constructors.

These constructs have subtle differences or specificities, which are out of the scope of this course. If you want to know more, please head over to the reference documentation at <https://docs.scala-lang.org/>.



Summary

Types form a hierarchy whose top type is `Any`.

Multiple traits can be mixed to a class or an object definition.

From within a class definition, you can refer to:

- ▶ the instance being defined with the identifier `this`,
- ▶ the parent implementation of a method `m` with `super.m`.

You can override inherited members, unless they are `final`.