Encapsulation

Effective Programming in Scala

A Case For Abstraction Barriers

Consider the following situation. We want to implement a program that exposes data read from a database as JSON documents.

The part of the program that exposes the data calls the part of the program that reads from the database and serializes the result into JSON.

Problem: if the database connection is widely available to any part of the program, it might be easy to mess up with the state of the system! Instead, we would like to restrict access to the database connection, so that we can increase our confidence that the code that uses it is correct.

A Case For Abstraction Barriers (2)

This leads to the following architecture:

The HTTP server can't access the underlying `connection` of the `DatabaseAccess` module. It can only use the public `readData()` operation.

Classes

```
class DatabaseAccess(connection: Connection):
  def readData(): List[Data] = ...

class HttpServer(databaseAccess: DatabaseAccess):
  ...
  databaseAccess.readData()
  ...

def main() =
  val connection: Connection          = ...
  val databaseAccess: DatabaseAccess = DatabaseAccess(connection)
  val httpServer: HttpServer          = HttpServer(databaseAccess)
  ...
```

Class Definitions

```
class DatabaseAccess(connection: Connection):
  def readData(): List[Data] =
    ...
    connection...
    ...
```

This code defines a **type** `DatabaseAccess` and a **constructor** of the same name. The type `DatabaseAccess` has one method, `readData`.

By contrast with case classes, constructor parameters of "simple" classes are **private**: they can be accessed only from the class body.

This highlights one difference between case classes and simple classes: the former achieve aggregation whereas the latter achieve **encapsulation**.

Private Members

```
class DatabaseAccess(connection: Connection):
  def readData(): List[Data] = ...
```

Class members are public by default: a user of the class `DatabaseAccess` can call its `readData` operation. It is also possible to define private members by qualifying them as such:

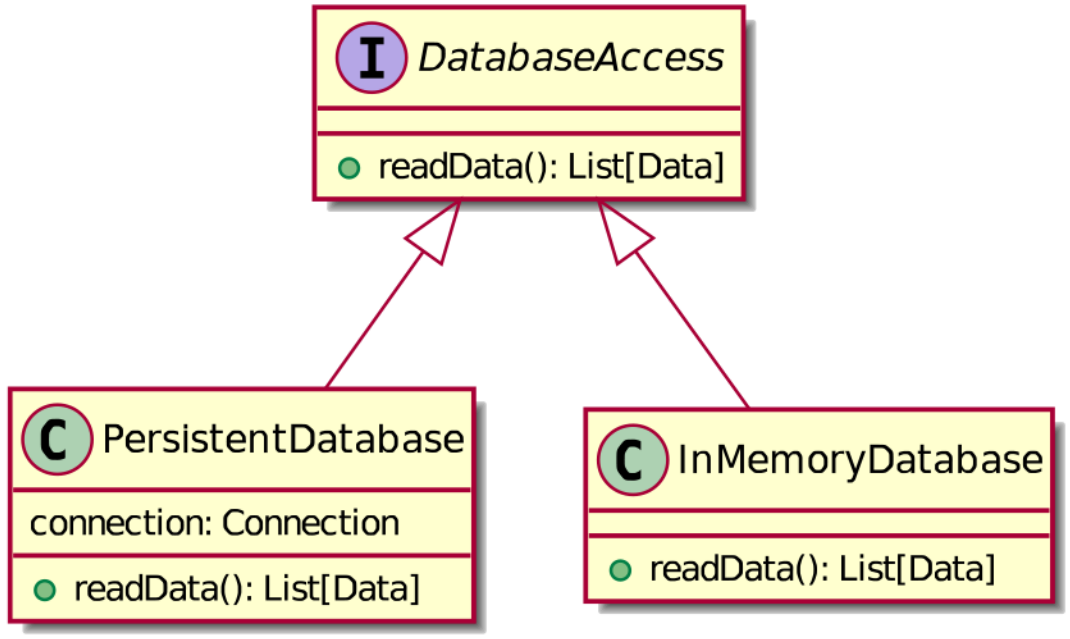
```
class DatabaseAccess(connection: Connection):
  private def decodeTableRow(row: TableRow): Data = ...
  def readData(): List[Data] = ...
```

The operation `decodeTableRow` can only be called from the inside of the class `DatabaseAccess`.

Abstracting Over Implementations

Sometimes we want to have several implementations of the same type.

For instance, we may want to use a persistent database for production, but an in-memory database for (some) tests:



Traits

```
trait DatabaseAccess:
  def readData(): List[Data]

class PersistentDatabase(connection: Connection) extends DatabaseAccess:
  def readData(): List[Data] = ...

class InMemoryDatabase extends DatabaseAccess:
  def readData(): List[Data] = ...
```

Traits and Abstract Methods

```
trait DatabaseAccess:
  def readData(): List[Data]
```

This code defines a **type** `DatabaseAccess` but **no constructor**. The type `DatabaseAccess` has one **abstract method**, `readData`.

Unlike sealed traits, "simple" traits can have an unbounded number of implementations.

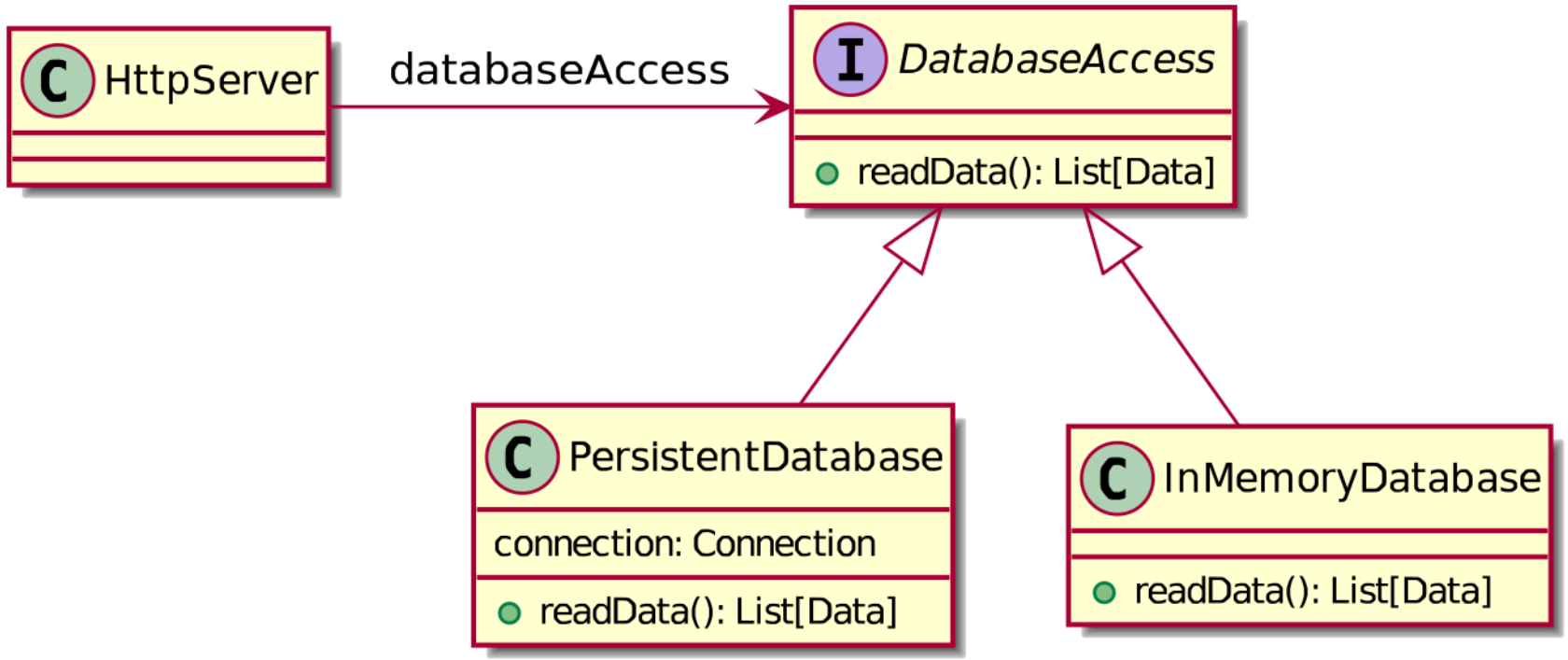
Abstract Methods

```
trait DatabaseAccess:
  def readData(): List[Data]
```

Classes that extend the trait `DatabaseAccess` have to implement the method `readData`:

```
class InMemoryDatabase extends DatabaseAccess
      ^
error: class InMemoryDatabase needs to be abstract. Missing implementation for:
  def readData: List[Data] // inherited from trait DatabaseAccess
```

Depending on Interfaces vs Implementations



Late Binding

```
class HttpServer(databaseAccess: DatabaseAccess):
  ...
  databaseAccess.readData()
  ...
```

The actual implementation of `readData` will depend on the concrete class passed to the `HttpServer` constructor.

Protected Members

Public members are visible from the outside of a class or trait, private members are visible from the inside of a class or trait only.

Protected members are visible from the inside of a trait or class, and from the inside of its descendants (but not from the outside).

Secret Sauce

With a combination of protected and abstract members we can make a dish while delaying to a later point the secret sauce brought by a chef.

```
trait Chef:
  protected def secretSauce(sauce: Sauce): Sauce
  def makeDish(meat: Meat, vegetables: Vegetables): Dish =
    ...
    secretSauce(...)
    ...

object JamieOliver extends Chef:
  protected def secretSauce(sauce: Sauce): Sauce = ...
```

Summary

Define abstraction barriers with classes and traits, which encapsulate implementation details.

Use private members to restrict the visibility of members to the inside of the class or trait definition.

Use abstract methods to delay the implementation of operations to concrete classes.