|  |  |  |
| --- | --- | --- |
| Aktion | Java | Scala |
| Folie: Domain-Model mit Bank, Identification und Partner / Person |  |  |
| J: Package ‚eightydays’ erstellen |  |  |
| J: Java Class ‚Partner’ erstellen mit Feld ‚Name’  Konstruktor, Getter+Setter, Equals + Hashcode |  | Warum machst du Setter? |
|  | Damit ich den Namen setzten kann |  |
|  |  | Wir möchten Immutability. Namen kannst du beim erstellen der Instanz mit Konstruktor definieren. |
|  | Wieso? |  |
| Folie: Warum Immutability wichtig ist |  |  |
|  | Wie verändere ich dann meine Daten? |  |
|  |  | Daten einer Instanz werden nicht geändert. Wir erstellen eine neue Instanz mit den veränderten Daten. Anstelle einen Wert zu verändern, erstellen wir eine Instanz mit dem neuen Wert. |
| J: Feld ‚Name’ wird final, Setter wird gelöscht | (Aha) |  |
| J: Java Class ‚Person’ extends ‚Partner’ erstellen mit Feld ‚Firstname’  Konstruktor, Getter, Equals + Hashcode |  |  |
| J: Java Class ‚LegalEntity’ extends ‚Partner’ erstellen mit Feld ‚form  Konstruktor, Getter, Equals + Hashcode |  |  |
| J: Java Class ‚Identifikation’ erstellen mit Feld ‚UUID’  Konstruktor, Getter, Equals + Hashcode |  |  |
| J: Java Class ‚Bank’ mit Felder ‚name’ und HashMap ‚partners’  Konstruktor, Optional getPartner(id), Equals + Hashcode |  |  |
|  | Testfälle dürfen natürlich nicht fehlen, da ich mit Hashcode und Equals Verhalten programmiert habe. |  |
| * Step 2 |  |  |
| Tests ausführen und zeigen, dass sie erfolgreich sind |  |  |
|  | Wie erstellt man jetzt solche Klassen in Scala? |  |
| S: Package ‚eightydays’ erstellen |  |  |
| S: Scala Class ‚Partner’ erstellen mit Feld ‚Name’ |  | Fertig! |
|  | Du bist doch noch nicht fertig! Wo ist das Feld ‚name’? Wo ist der Construcotr? Wo ist der getter, hashCode und Equals? |  |
|  |  | Das was bei Java von Hand geschrieben oder mit der IDE generiert wird, macht bei Scala der Compiler für mich. Bei einer case-Klasse werden u.A. getter, hashCode und equals unter anderem automatisch erstellt. |
|  |  | Ein weiterer Vorteil davon ist, dass ich mir viel Redundanz erspare. Ein Beispiel ist das der Konstruktor zwingend gleich heissen muss wie die Klasse selbst.  Ebenfalls muss das Feld name nur einmal geschrieben werden. Bei Java ist es ein Mehrfaches, was die Wartbarkeit und Fehleranfälligkeit erhöhen kann. |
|  |  | Ich habe in Scala auch die Einschränkung nicht, dass die Datei gleich heissen muss wie die die Klasse. D.h. ich kann in einem File mehrere Elemente definieren und so zusammengehörende Fachlichkeit auch zusammen definieren. |
| S: Scala Class Person mit firstName, LegalEntity mit form und Identification mit number |  |  |
|  | Du hast da den Typ von Number vergessen | Nein, was liefert mir die Identification für einen Wert zurück? |
|  | Einen Typ ‚UUID’? |  |
|  |  | Genau. Und das weiss der Compiler auch! Er leitet den Typ aus einer gegebenen Expression ab. |
|  |  | Übrigens: In Scala ist fast alles eine Expression. D.h. auch ein If-Statement oder ein for- oder while-Loop liefert einen Wert zurück. |
| S: Scala Class Bank mit partners (private[this]) |  |  |
|  | Was ist das [this] bei private? |  |
|  |  | Ein Unterschied zu Java ist, dass in Scala alles per default einen public-Scope hat. Wir haben ja alle Daten als Immutable, daher ist das kein Problem.  Bei Java ist es ja package-Scope. |
|  |  | Das hier ist ein private-Scope nicht nur für die Klasse, sondern für die Instanz einer Klasse. Nur diese hat hierauf Zugriff. Etwas, was in Java nicht möglich ist. |
| S: Scala Class Bank mit partner() = partners get id |  |  |
|  | Wo sind die Punkte bei der Implementierung von partner? |  |
|  |  | In Scala können neben den Semikolons unter bestimmten Umständen die Punkte oder auch die Klammern weggelassen werden.  Dies ermöglicht zum Beispiel die einfache Erstellung von internen DSLs. |
|  | Die Funktion gibt aber nichts zurück, da fehlt doch noch das Return? |  |
|  |  | Auch das Return selbst ist in Scala optional.  Normalerweise ist das Resultat, also der Rückgabewert einer Funktion, die letzte Expression. |
|  | Das ist etwas gewöhnungsbedürftig.  Da sehe ich nicht mehr auf den ersten Blick, wann was und wie aus einer Methode zurückgegeben wird! |  |
|  |  | Ein wesentlicher Vorteil von Scala gegenüber Java ist, dass ich mit Scala den Code auf das Wesentliche, auf die Essenz reduzieren kann.  Ein Return-Statement ist nichts Anderes als ein Goto, welches ans Ende einer Funktion springt und den dahinter folgenden Wert zurückliefert. Oder kurz: Return ist Boilerplate-Code.  Scala ist minimalistisch. Das beste Beispiel dazu ist, wie wir bereits gesehen haben, die Definition einer Case-Klasse, reduziert auf eine Zeile Code. |
|  | Das leuchtet mir ein.  Und was ist mit Testfällen? |  |
| * Step 3 |  |  |
| Test ausführen und Resultat zeigen. |  | Da ich mit Scala das ganze Java-Universum verwenden kann, kann ich natürlich auch auf JUnit zurückgreifen.  Bei Scala gibt es aber auch noch andere Testframeworks, z.B. ScalaTest. |
|  |  | ScalaTest kennt verschiedene Test Stile.  Neben dem „Function Style“, ähnlich dem jUnit, gibt es auch strukturierte Tests, wie z.B. WordSpec, welcher ein BDD Stil implementiert, also Behaviour-Driven Development. |
|  | Bei JUnit hatte ich immer das Problem, dass ich meine Testfälle nicht strukturieren konnte.  ScalaTest löst das für mich schön elegant. |  |
|  |  | Stimmt. Aber lass uns das Domain Model weiter implementieren. |
| Folie: Domain-Model mit Accounts |  |  |
| * Step 4 |  |  |
|  | <Code erklären> |  |
|  | Das Design, resp. die Implementierung der verschiedenen Kontotypen ist etwas schwierig. |  |
| Folie: AccountTypen und Regeln, die orthogonal zur Hierarchie sind | Die Regeln, Berechnungen sind Querschnittsfunktionen, die unabhängig von der Vererbungshierarchie benutzt werden.  Um die Anforderungen sauber (meint ohne Redundanz und unter Beachtung des open-closed prinzip) zu implementieren, habe ich mich fuer den Ansatz mit einem Decorator auf den Methoden withdraw und desposite entschieden. So kann ich bei jeder Klasse bestimmen, welche Regeln angewendet werden und dies unabhängig von der Basis-Klasse. |  |
| * Step 5 |  |  |
| Tests ausführen, Code erklären. Saving und dann Current | <erklären>  Zusätzlicher Code für die Implementierung des Pattern. |  |
|  |  | Das funktioniert, aber das ist technischer Boilerplate-Code, welcher nicht fachlich getrieben ist.  Mit Scala geht das einfacher. |
| * Step 6 (Account, Sub-Klassen und Traits) |  |  |
|  |  | <erklären>   * Account   + Type Amount ist alias auf BigDecimal   + ‚balance’ berechnet Saldo mit Hilfe der Bookings |
|  | Moment, da ist ein Fehler!  Wie kannst Du ein Amount zu einer Booking mit einem + addieren? |  |
|  |  | Das ist etwas Scala-Magie. Hier wird ähnlich den Case-Klassen durch den Compiler Code generiert, welcher diese Konvertierung ermöglicht.  In Java ist das übrigens das gleiche wie mit dem Autoboxing un Unboxing von primitiven Typen.  In Scala kann ich solche Konvertierungen selbst definieren. |
|  |  | Diese implizite Konvertierungen sind eine sehr mächtige Möglichkeit in Scala.  Daneben können auch Methoden, Klassen und Parameter implizit sein, was mir ganz neue Möglichkeiten eröffnet  Ich als Entwickler muss daher wissen, was der Compiler hier für mich macht. Dadurch steigt aber auch die Anforderung an mich als Entwickler. |
|  | Aber wie weiss ich, welche implizite Methode da aufgerufen wird? Ich sehe das dem Code ja nicht direkt an. |  |
|  |  | Das ist in der Tat so. Da muss ich mich auf die Hilfe von IDEs verlassen.  Sie hilft mir in diesem Fall und zeigt mir eine implizite Konvertierung an <grauer Unterstrich und Ctrl + Q bei booking>.  In der Entwicklung selbst gilt es einen Mittelweg zwischen den Möglichkeiten der Sprache und der Verständlichkeit des Codes zu finden.  Je mächtiger eine Sprache ist, desto mehr muss ich mir das bewusst sein. |
|  |  | * Account   + Bei withdraw habe ich einen Default Value für den Parameter ‚valuta’ definiert. Das ist das Equivalent zu einer überladenen Methode in Java. * Package ‚fee’ mit Trait ‚LowBalancePerBooking’ ist ein MixIn, welches die Methode ‚post’ aus Account überschreibt und dadurch die Funktion erweitert. In diesem Trait könnte man auch neue Funktionalität hinzufügen. In unserem Fall hier, ist das aber nicht nötig.   (geht auf val anstelle von def beim trait?) |
|  | Das ist doch das gleiche wie die Default Methoden in Interfaces bei Java 8.  https://dzone.com/articles/interface-default-methods-java  https://kerflyn.wordpress.com/2012/07/09/java-8-now-you-have-mixins/ |  |
|  |  | Die Motivation bei Java ist eine andere. Sie werden auch ‚Public Defender Methods’ genannt. Da geht es um Interface Evolution und nicht um die eigentliche Modularisierung von Fachlichkeit bzw. von Komponenten. |
|  |  | Scala definiert auch eine strikte Reihenfolge, wie gleiche Methoden bei mehreren vorhandenen Traits nacheinander aufgerufen werden. Dies löst das bekannte Diamond-Problem bei mehrfach Vererbung.  <http://stackoverflow.com/questions/16410298/what-are-the-differences-and-similarties-between-scala-traits-vs-java-8-interfa> |
|  |  | * Current: hier wird ein Account definiert, welches zusätzlich Limited und LowBalancedPerbook beinhaltet. Dadurch wird beim ‚post’ zusätzliche Fachlichkeit ausgeführt. Fee und Limiten werden als zusätzliche Constructor-Parameter übergeben. * Saving ist das gleiche, aber mit NoOverdraw anstatt LowBalancedPerBooking |
| Tests ausführen |  |  |
|  | Weiter mit den Methoden des Zusammenspiels Partner/Account mit Bank |  |