Setup:

1. IntelliJ starten
2. IntelliJ auf eigenen Desktop, vollbild und praesentations modus
3. Terminal und step 1 ausfuehren
4. Systemproperties Desktop offnen und arragemenet auswaehlen

|  |  |
| --- | --- |
| Beschreibung | Aktion |
| Leeres projekt | **./step 1** |
| Package erstellen | Cmd + N -> **package eightyDays.java** |
| Person implementieren | Cmd + N -> class **Person**  Felder name, firstname  Consturctor  Getter/Setter  **Equals/HashCode** |
| Bank implementieren | Cmd + N -> class **Bank**  Feld name  Consturctor  Getter/Setter  **addPartner** (beim hinzuefugen der **gleichen person** soll die **gleiche UUID** geliefert werden)  public UUID addPartner(Person pPerson) {  if (persons.containsValue(pPerson)) {  for(UUID id : persons.keySet()) {  if (persons.get(id).equals(pPerson)) {  return id;  }  }  }  UUID newId = UUID.*randomUUID*();  persons.put(newId, pPerson);  return newId; }  **getPartner**  public Person getPerson(UUID pId) {  return persons.get(pId); } |
| Tests ausfuehren | Tests vorbereteitet  .**/step 2**  Run Ctrl + Shift + R |
| Scala implementieren  Package erstellen | Package erstellen  Cmd + N -> package eightyDays.scala |
| Person implementieren | Cmd + N -> scala class **Person**  Case class  Felder name, firstname |
| Bank implementieren | Cmd + N -> scala class **Bank**  Feld **val name**  Feld **var partners**, hier will ich **mutable** state  **var** *persons* = *Map*[UUID, Person]()  Methode **addPartner** (bei find mit entry => entry.\_2 dann wieder loeschen \_.\_2)  **def** addPartner(pPerson: Person) = *persons* .find(\_.\_2 == pPerson)  .map(\_.\_1)  .getOrElse {  **val** newID = UUID.*randomUUID*()  *persons* = *persons* + (newID -> pPerson)  newID  }  Methode **person**  **def** person(pId: UUID) = *persons* get pId |
| Tests | **./step 3**  Tests zeigen: **Person** kein equals/hashCode, da dieser **generiert** ist  Tests ausfuehren |
| immutability | Scala favorisiert **immutability**. Map, name ist immutable  Wieso? |
| **Slide Immutability** | Immutability hat gegenüber Mutability folgende Vorteile. Da sich Daten nicht ändern, kann ich diese **ohne Gefahr sharen**. D.h. immutable objects sind **inherent thread-safe**. Es ist shared state, aber nicht shared mutable state.  Dies erlaubt auch, dass ich die Objekte **cachen/zwischenspeichern** kann.  Immutability ermöglich auch die Realisierung der **Wert-Semantik**. Der Unterschied zwischen einem Objekt und einem Wert liegt in der Identität. Ein Objekt hat eine Identität, ein Wert nicht. Beispiel BigDecimal/String/Integer in Java implementiert für einen bestimmten Wertbereich ein **flyweight-Pattern**. Dies kann sich positiv auf die **Performance**, resp. **Ressoucen**-Bedarf auswirken. Das **Erstellen von Objekten ist atomar**. Die sind Objekte **nach der Erstellung** per Definition immer **valide/gültig**. Wenn etwas schiefgeht, dann im Constructor bei der Erstellung. Dies **vereinfacht** die **Fehlersuche**. Die **Komplexität** ist geringer, da ich die **Fälle**, bei denen sich Daten **verändern**, **nicht implementieren** muss. Klassen sind **einfacher** zu erstellen, zu testen und zu benutzen.  Es dauert einige Zeit wenn man jahrelange mit mutable objects gearbeitet hat bis man diese Praxis ablegt und mit immutability arbeitet. |
| Java immutable machen | Person **final** und **setter weg**  Bank **final** und **setter weg** |
| Java functional machen | Bank **addPartner** umschreiben  War **imperativ** und mit **Boilerplate**. Neu **functional**, d.h. mit **Stream** und **high-order Functions**.  (Block mit Alt + Cursor markieren und mit Alt + Shift + Cursor bewegen)  public UUID addPartner(Person pPerson) {  return persons.entrySet().stream()  .filter(entry -> entry.getValue().equals(pPerson))  .findFirst()  .map(Map.Entry::getKey)  .orElseGet(() -> {  UUID newID = UUID.*randomUUID*();  persons.put(newID, pPerson);  return newID;  }); }  Bank **getPartner** mit **Optional** (**Consumer** soll sich nicht mit **null befassen**, sondern es wird immer etwas geliefert. Ist **Null-Object-Pattern**). Empty ist None  public Optional<Person> getPerson(UUID pId) {  return Optional.*ofNullable*(persons.get(pId)); } |
| Tests ausfuehren | Sind noch gruen. Nichts gebrochen (./step 4 definiert den Stand) |
| **Slide Accounts mit Constraints** | **Slides** |
| Naive Java Implementierung | **./step 5**  Den Code habe ich vorbereitet.  Account zeigen   * Liste der Buchungen * Buchung zeigen -> Value und Valuta, ist immutable * Withdraw and deposit   + Prüft den Betrag > 0   + Post aufrufen, soll/haben mit positiv, resp. negativen Betrag * **Post ist abstract** und muss von der konkreten Sub-Klasse überschrieben werden. Grund: Immutable und es muss eine neue Instance mit der zusätzlichen Buchung der konkreten Klasse erzeugt werden.   Zwei konkrete Ausprägungen: Current und Saving (**Alt+Cmd+B**)  Saving zuerst   * WithdrawUtil.noOverdraw anschauen * WithdrawUtil.limited anschauen * Wenn obigen Code keine Exception, dann neues Saving-Account mit der zuästzlichen buchung   Current   * Limited wie bei Saving * Etwas komplizierter, es soll, wenn der Saldo eine bestimmte Limite unterschreitet, eine zusätzliche Buchung für die Gebühr gemacht werden. Die Prüfung der Limite, wie auch die Buchung ist FeeUtil.feeIfLowBalance implementiert. |
| Tests ausführen | Tests zeigen und ausführen (**Shift+Ctrl+R**) |
| Scala implementation | Wie implementiere ich dies in **Scala**?  Den Code habe ich vorbereitet **(./step 6** ausführen)  Account zeigen   * Die Klasse macht das **gleiche**, wie bei Java mit einem Unterschied. **Post ist implementiert**. Dies war bei Java auf den konkreten Sub-Klassen.   Saving zeigen   * Saving hat die Constraints, dass ich das Konto nicht überziehen kann und auch eine Rückzugslimite habe. Diese beiden Aspekte sind als Traits realisiert, welche ich als MixIns benutze.   NoOverdraw zeigen   * Dieser Trait überschreibt die Methode withdraw und fügt den zusätzlichen check ein (das if).   Limited zeigen   * Wie auch NoOverdraw überschreibt dieser Trait die Methode withdraw. Auch hier wird ein zusätzlicher Check bezüglich der Rückzugslimite gemacht. Wenn der Betrag unter der Limite ist, wird super.withdraw aufgerufen.   Zurück zu Saving: Da die traits beide die gleiche Methode withdraw überschreiben, werden diese durch Scala linearisiert und in einer Sequenz ausgeführt. |
| Tests ausführen | Auch hier habe ich Tests. Fachlich machen die genau das gleiche, wie bei Java (Shift+Ctrl+R). |
| Zurück zu Java | Wie sieht jetzt eine Implementierung in Java aus, wenn ich dies ähnlich machen möchte? D.h. ich möchte meine Cross-Cutting Concerns nicht in Utility-Klassen haben, sondern eher so was wie Traits, resp MixIns.  Den Code habe ich vorbereitet (./step 7).  AccountTest zeigen   * Die Tests sind identisch wie bei der naiven Implementierung.   Account zeigen   * Dies ermöglicht, dass ich post jetzt auch in der Basis Klasse Account implementieren kann.   Saving zeigen   * Anders als zuvor wird jetzt withDraw überschrieben. Hier mache ich eine **function composition, eine Komposition von Funktionen**. Nämliche den Check dass kein Überziehen möglich ist, den Check der Rückzugslimite und schlussendlich das Buchen in der Basis-Klasse. Die Linearisierung, welche in Scala durch den Kompiler durchgeführt wurde, muss ich hier selber machen.   Limited zeigen   * Hier wird jeweils der predessor/Vorgänger als Parameter beim Konstruktor mitübergeben. * Die Methode build macht nicht den Check, sondern liefert eine Methode, die den Check ausführt   NoOverdraw   * Die Methode build macht nicht den Check, sondern liefert eine Methode, die den Check ausführt   Current zeigen   * Hier wird auch Limited benutzt * Zusätzlich wird noch die Funktion von LowBalancePerBooking benutzt   LowBalancePerBooking zeigen   * Hier wird, wenn der Saldo unter der Schwelle ist, eine zusätzliche Gebühr belastet * Auch hier ist es eine High-order Function, d.h. eine Funktion, die eine Function liefert |
| Test ausführen | Die Tests sind unverändert und sind grün (Shift+Ctrl+R) |
| Unterschiede | Auf den ersten Blick sind die Implementierung in Scala und Java sehr ähnliche, aber nicht gleich. Ein wesentlicher Unterschied ist, dass bei der Java Implementierung die konkrete Klasse wissen muss, welche Methoden der Basis-Klasse durch den Cross-Cutting-Concern, wie NoOverdraw oder LowBalancePerBooking erweitert wird. Weil hier dann eine Delegation stattfinde. Die Scala Implementierung mit Traits hat diesen Nachteil nicht.  In Scala ist die Implementierung weniger invasiv als in Java. Scala folgt hier dem Open-Closed-Principle von Betrand Meyer. |

Moegliche Erweiterungen fuer JUGS:

- statement vs expression -> alles muss in einer Zeile gehen

- pattern matching -> exterm starke sache -> dekomposition bei verschatelten strukturen

- curring und partial applied functions -> keine stming data mehr

- implicity conversions -> artikel den ich heute gelesen habe als basis nehmen