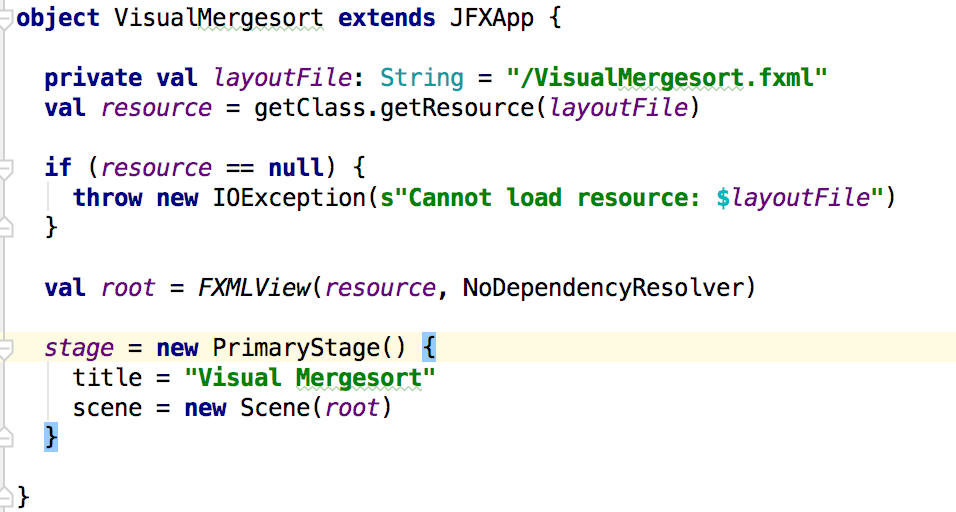
code

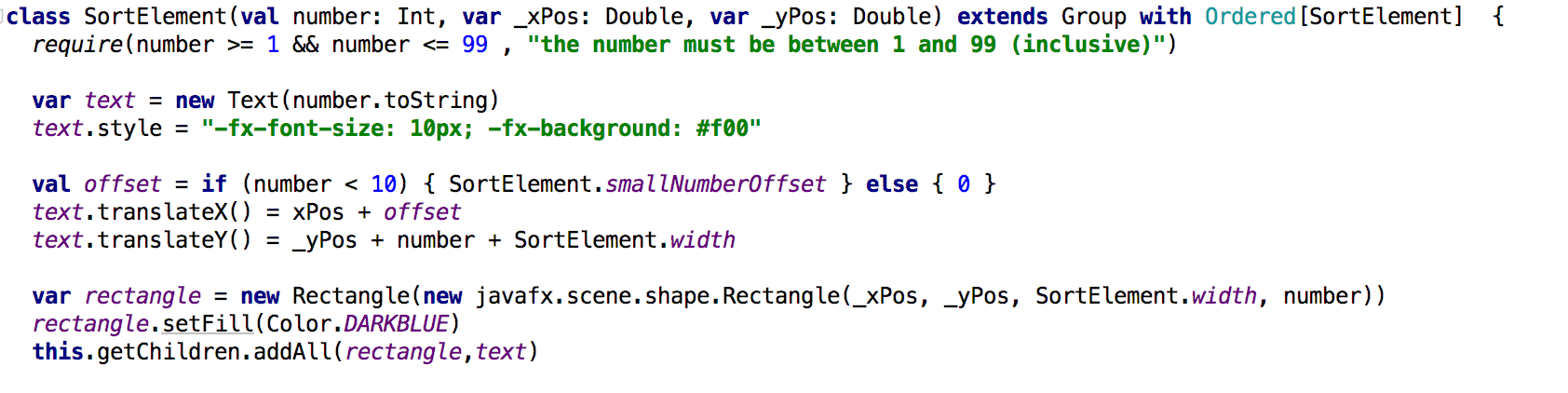
Das Objekt VisualMergesort

Das Objekt VisualMergesort ist ein Singleton-Objekt der anonymen Klasse VisualMergesort, welches garantiert einzigartig ist. Das bedeutet, dass keine weiteren Objekte gleicher Art erzeugt werden können. „VisualMergesort“ erbt von JFXApp und definiert somit den Startpunkt unserer Applikation.

In diesem Objekt wird das in FXML vordefinierte Layout unserer GUI in die Stage der Applikation geladen.

Die Klasse SortElement

Die Klasse SortElement ist für die Objekte verantwortlich, die durch den Algorithmus auf der Zeichenfläche sortiert werden sollen. Die zu sortierenden Objekte sind zusammengesetzte Gruppen und bestehen aus einem Rechteck sowie einem darunter liegenden Text, welcher den Wert des Objektes angibt. Die Höhe des Rechtecks verhält sich proportional zu diesem Wert.

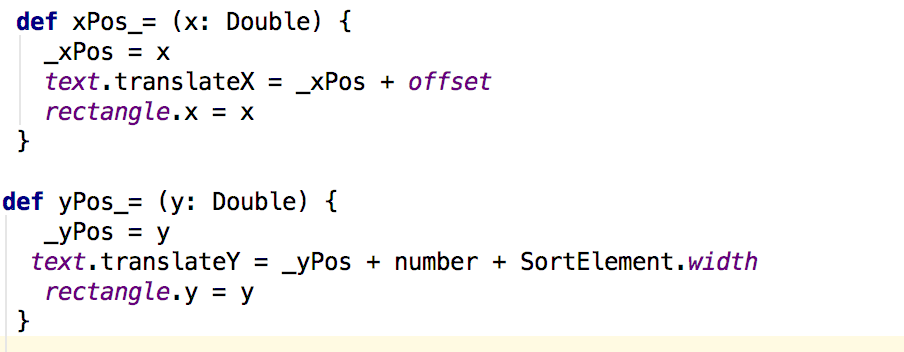
Der Text benötigt, um mittig unter dem Rechteck zu stehen, noch einen offset, falls der Wert einstellig ist.

Um die von Scala vordefinierten Getter und Setter der Variablen „xPos“ und „yPos“ zu überschreiben, werden diese in „\_xPos“ und

„\_yPos“ umbenannt. Danach werden zunächst die Getter definiert:

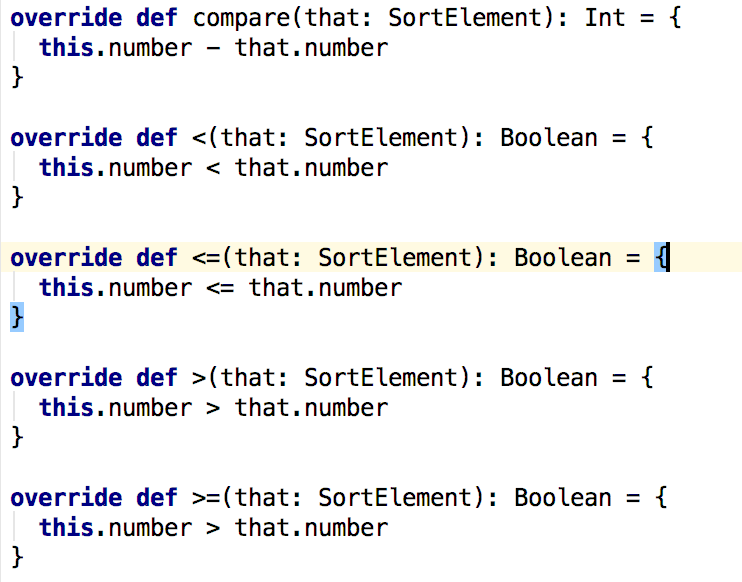


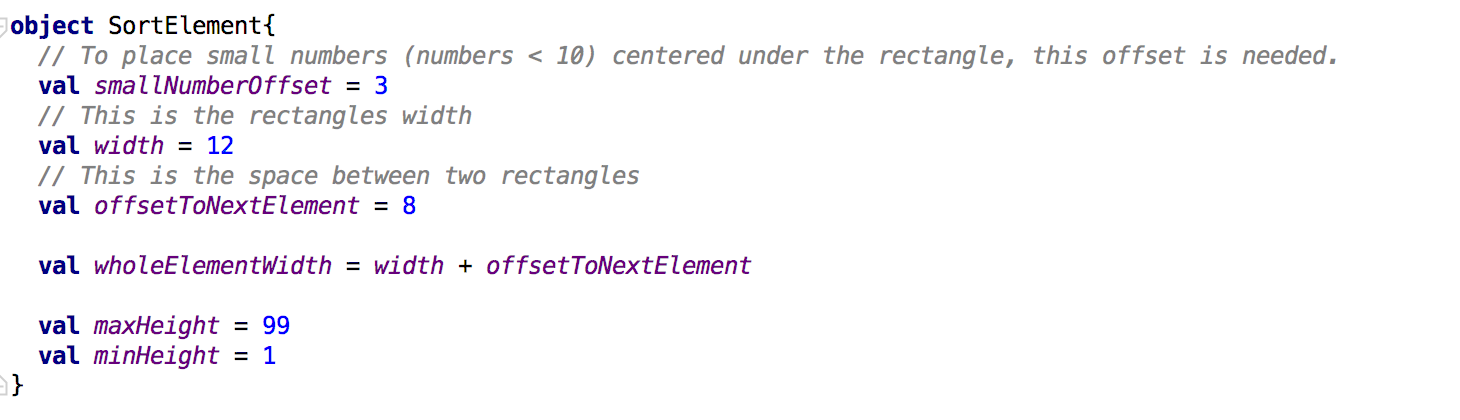
Dieser Code definiert zwei simple Methoden „xPos“ und „yPos“, welche die Variablen „\_xPos“ und „\_yPos“ zurückgeben. Da bei Scala der letzte Ausdruck einer Methode auch gleichzeitig der Rückgabewert ist und geschweifte Klammern nicht benötigt werden, falls die Methode nur aus einem Ausdruck besteht, sind diese und das Return-Statement nicht vorhanden.

Als nächstes werden die Setter neu definiert:

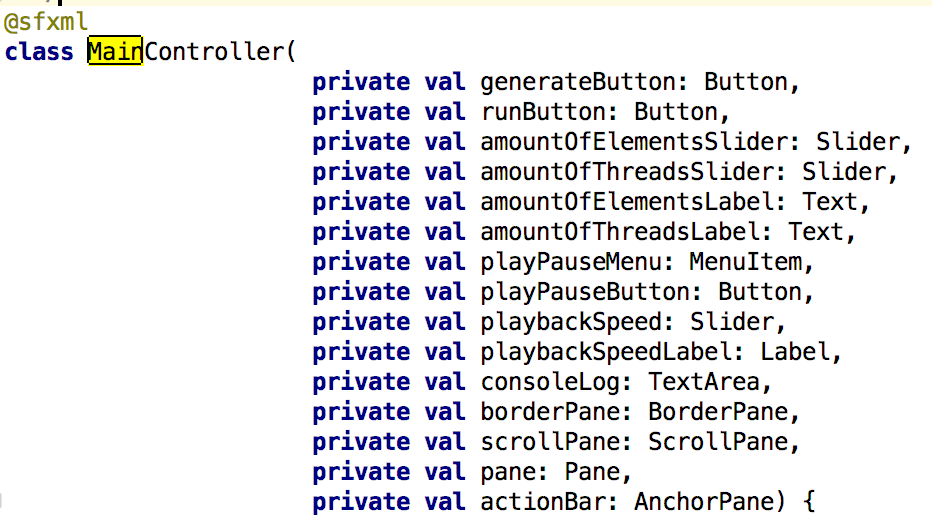
Der Name dieser Methoden ist „xPos\_“ und „yPos\_“. Der Unterstrich dient in Scala als spezielles Zeichen und ist in diesem Fall als Platzhalter zu verstehen, der beim Aufrufen der Methoden durch ein Leerzeichen ersetzt werden kann. Somit können diese Methoden im folgenden Code sowohl mit „xPos =„ bzw. „yPos =„ als auch mit „xPos\_=„ bzw. „yPos\_=„ aufgerufen werden.

(Quelle für Literatur: <http://dustinmartin.net/getters-and-setters-in-scala/>)

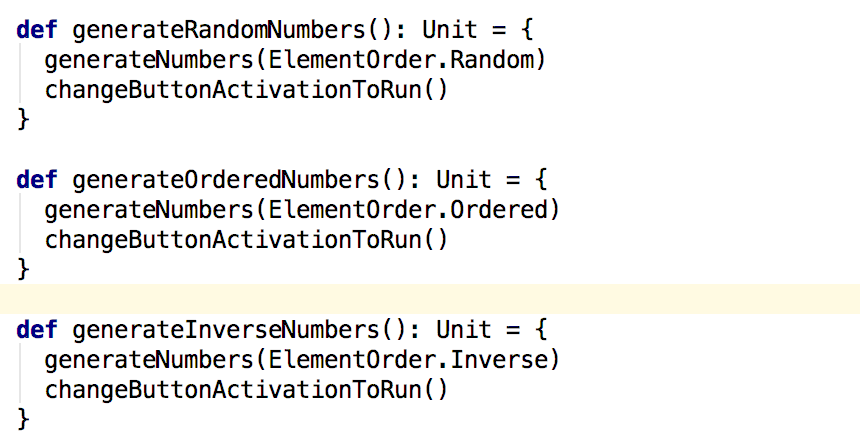
Um die Objekte beim Sortieren durch den Mergesort miteinander vergleichen zu können, werden die Methoden des Trait „Ordered“ implementiert. Dieser Trait entspricht dem Interface „Comparable“ in Java. Zusätzlich zu der Methode „compare“ werden verschiedene Vergleichsoperatoren für das Objekt SortElement neu definiert:

Zusätzlich zu der Klasse SortElement wird noch ein Singleton-Object „SortelEment“ angelegt, welches alle vordefinierten Werte beinhaltet, die von der Klasse SortElement benötigt werden. Die Klasse SortElement kann auf alle Felder des Objektes zugreifen und diese für Erstellung neuer Objekte benutzen.

Die Klasse MainController

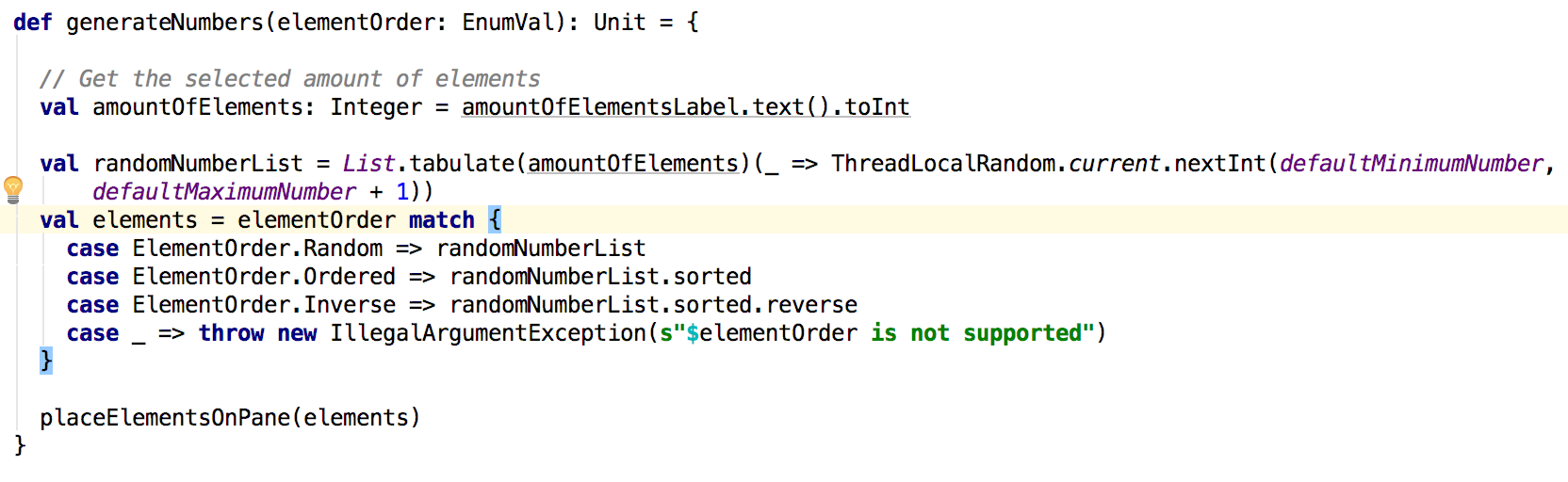
Die Klasse MainController ist, wie der Name schon hergibt, der Controller unserer Applikation. Hier werden alle Interaktionen des Benutzers mit der GUI entgegengenommen und verarbeitet. Um dies zu implementieren, werden alle im FXML-Layout vordefinierten Buttons, Slider und andere Eingabemöglichkeiten in diese Klasse importiert und deren Methoden definiert. Hierzu dient die über der Klasse stehende Annotation „@sfxml“ . Die Objekte lassen sich über eine im FXML-Code festgelegte ID vom Controller ansprechen. Diese ID’s müssen für die Klasse übernommen und am Anfang festgelegt werden:

Erzeugen der Elemente und Initialisierung der Anwendung durch den Benutzer

Um den Visual-Mergesort betriebsbereit zu machen, hat der Benutzer die Möglichkeit, aus drei verschiedenen Möglichkeiten auszuwählen, um die zu sortierenden Elemente auf der Zeichenfläche zu platzieren. Im Menü lässt sich über die vorhandenen Schaltflächen auswählen, ob die Elemente vom System zufällig verteilt, vorsortiert oder invertiert erzeugt werden sollen. Hierzu dient ein Enum, das je nach betätigtem Button mit dem entsprechenden Wert an die Methode „generateNumbers“ übergeben wird.

Nachdem die Elemente erzeugt wurden, ist die Anwendung betriebsbereit und der Run-Button wird aktiviert.

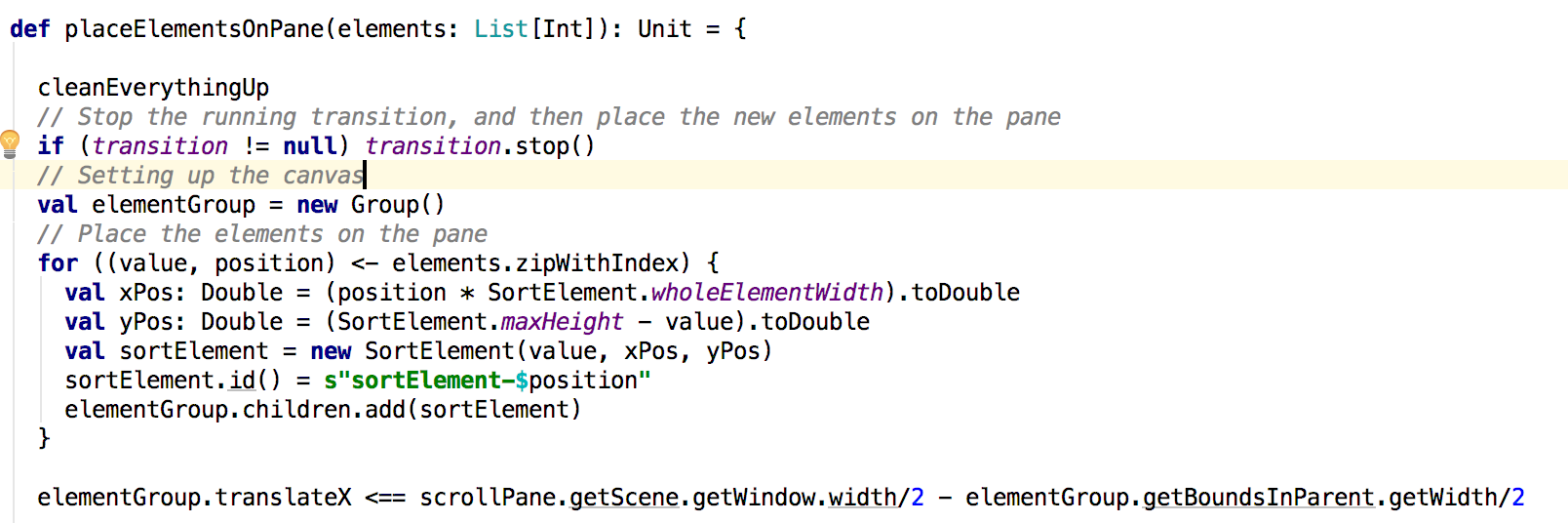
In der Methode „generateNumbers“ wird eine Liste mit Zufallszahlen zwischen dem Minimalwert „defaultMinimumNumber“ und dem Maximalwert „defaultMaximumNumber“ generiert, die so viele Elemente enthält, wie über den Slider in der Applikation eingestellt wurde. Der Unterstrich dient in diesem Fall als Platzhalter für jedes Element in der Liste. Je nachdem, welcher Button zuvor betätigt wurde, wird die Liste anschließend optional noch sortiert oder invertiert.

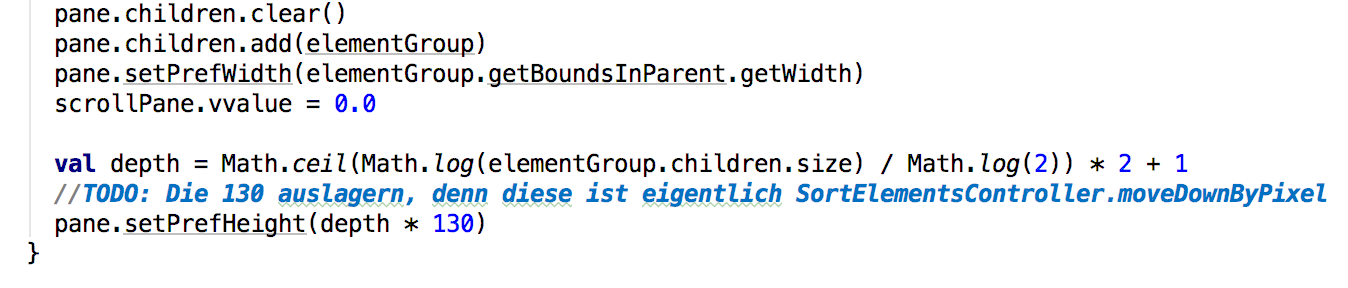


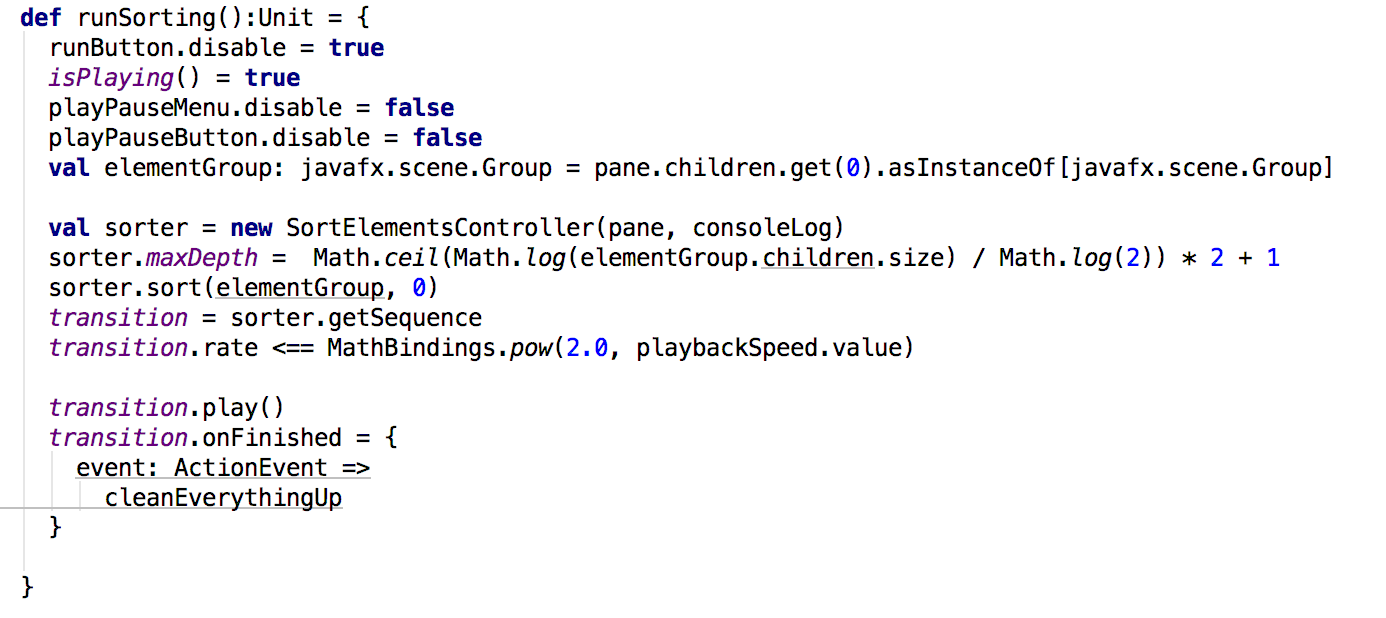
Zum Schluss wird die fertige Liste and die Methode placeElementsOnPane übergeben.

Die Methode placeElementsOnPane erzeugt für jeden Wert in der übergebenen Liste ein „SortElement“ und fasst alle generierten Objekte in einer Gruppe zusammen. Ist die Applikation bei Aufruf der Methode am Laufen, wird sie gestoppt.

Anschließend wird die Zeichenfläche von allen darin befindlichen Objekten gesäubert und die neu generierte Gruppe wird hinzugefügt:

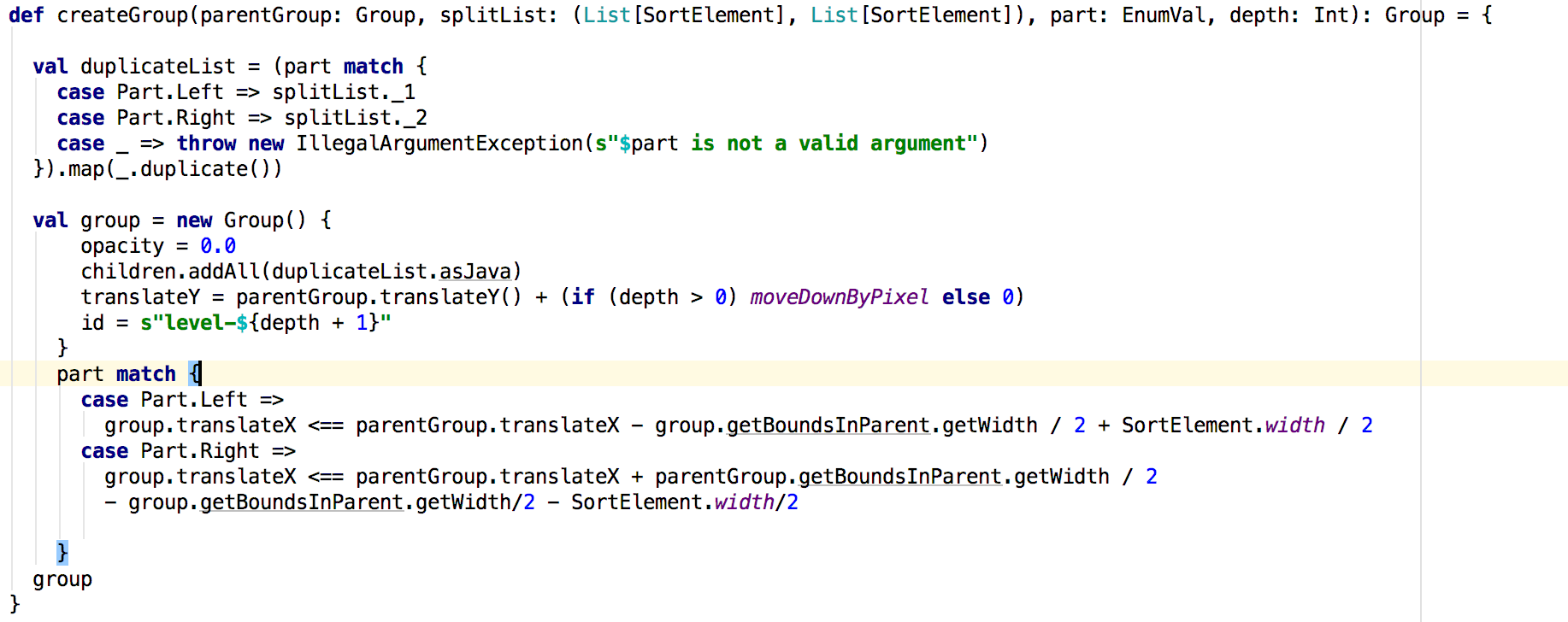
Basierend auf der Anzahl der neu erzeugten Elemente, wird die Größe der Zeichenfläche angepasst. Dazu wird vorerst die maximale Tiefe der Gruppen definiert. Bei jedem Split und bei jedem Merge, wandern die gesplitteten oder zusammengesetzten Gruppen eine Ebene nach unten. Da sich eine Gruppe bei jedem Split so lange in zwei Teile teilt, bis sich nur noch ein einziges Element in der Gruppe befindet und genauso viele Merges wie Splits durchgeführt werden, lautet die Formel für die Anzahl an Splits und Merges „log(Basis2)(n) \* 2“. Allerdings befindet sich von Anfang an schon die vom System generierte Gruppe auf der Zeichenfläche. Diese muss noch hinzugerechnet werden. Somit ergibt sich für die maximale Tiefe log(Basis2)(n)\*2 +1.



Wurde die zu sortierende Menge auf der Zeichenfläche platziert, kann der Benutzer auf „Run“ klicken, um den Algorithmus zu starten. Dadurch wird die Methode „runSorting“ ausgeführt.

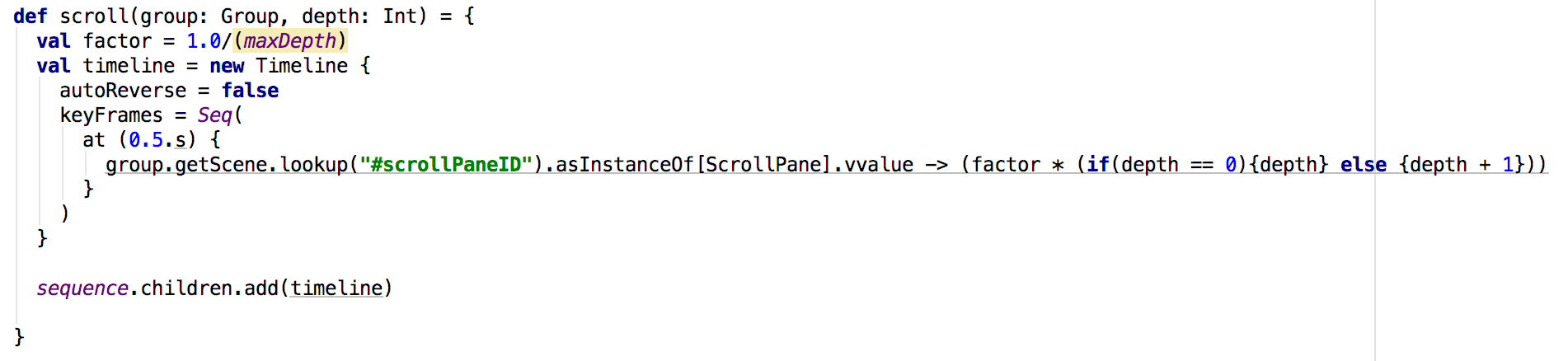
Durch die Methode „runSorting“ wird der Run-Button deaktiviert und der Algorithmus zum Sortieren der Elemente durchgeführt, welcher die vorher generierte Gruppe als Eingangsmenge übernimmt. Anschließend wird die Animation, die den Algorithmus visualisiert, gestartet.

Die Implementierung der Logik für den Algorithmus befindet sich in der Klasse SortElementsController.

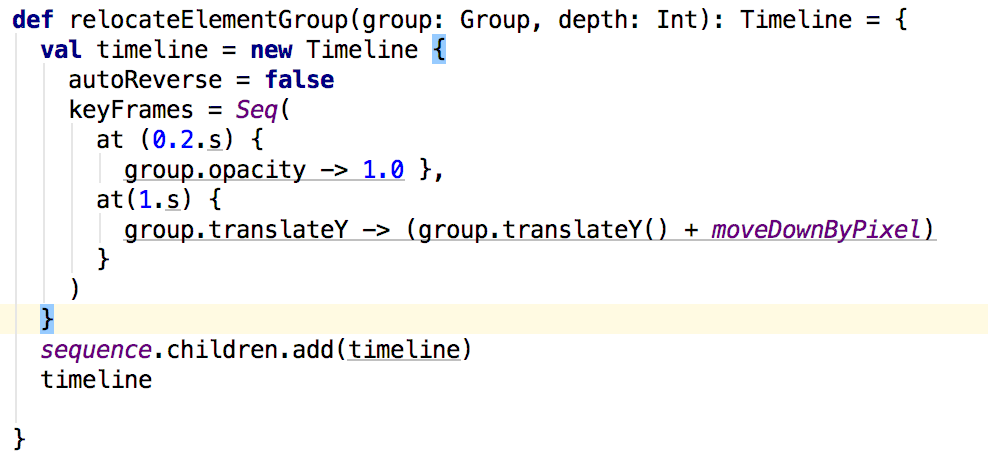
Wie schon erwähnt, wird bei unserer Implementierung des Mergesort-Algorithmus nicht mit einer einzigen Gruppe gearbeitet, sondern mit vielen Listen, die nicht überschrieben werden. Das dient dazu, die schon gesplitteten und gemergten Elemente auf der Zeichenfläche unangetastet zu lassen. Um dies zu realisieren, müssen Duplikate der Eingangslisten erstellt werden, mit denen weitergearbeitet werden kann, ohne die schon gezeichneten Objekte zu beeinflussen. Man braucht also eine zweite Liste, welche Objekte mit äquivalenten Werten enthält. Für diese Aufgabe existiert die Methode „createGroup“, welche in unserer „sort“ - Methode aufgerufen wird.

Die Methode createGroup erzeugt gleich zu Beginn ein Duplikat der eingegangenen splitList und zwar, abhängig vom mitgegebenem Enum, entweder von der linken oder der rechten Hälfte und speichert dieses in der Value „duplicateList“. Die „map“-Methode, die hier verwendet wird, erzeugt für jedes SortElement-Objekt in einer SplitList ein Objekt mit gleichen Werten und gibt dieses an die neue Liste weiter. Der Unterstrich steht in diesem Fall also für jedes Element. Nachdem alle Elemente der „duplicateList“ zu einer Gruppe hinzugefügt wurden, wird diese Gruppe, relativ zu der Gruppe, aus der sie entstanden ist, ausgerichtet. Durch das verwendete Binding „<==„ wird die Positionierung stets angepasst, falls sich die Position der Elterngruppe ändern sollte.

Implementierung des Autoscrolls

Um dem Benutzer den Fokus auf den Algorithmus zu erleichtern, haben wir es für sinnvoll erachtet, einen automatischen Scrollmechanismus zu implementieren, welcher stets das Geschehen zum Mittelpunkt der Szene macht. Hierbei machen wir uns noch einmal die zuvor für die Größe der Zeichenfläche genutzte Tiefe der Gruppenstruktur zu Nutze.

Zuerst wird, abhängig von der maximalen Tiefe unserer Ausgangsgruppe, ein Faktor berechnet. Dieser Faktor dient dazu, die Zeichenfläche in genau so viele Teile zu untergliedern, dass für jeden Tiefenwert ein Bereich entsteht. Bei Aufruf der Methode „scroll“ wird eine Animation erzeugt, die dazu dazu dient, die Applikation in genau 0.5 Sekunden automatisch zu dem Bereich zu bewegen, welcher der Tiefe der übergebenen Gruppe zugeordnet ist. Zum Schluss wird die generierte Animation unserer Hauptsequenz hinzugefügt.

Nachdem ein Split in der Methode „sort“ durchgeführt wurde, muss die neue Gruppe mit den duplizierten Elementen noch richtig animiert werden. Hierfür ist die Methode „relocateElementGroup“ verantwortlich.

Die Methode „relocateElementGroup“ erzeugt eine Animation, welche die übergebene Gruppe in 0.2 Sekunden sichtbar macht und sie binnen einer Sekunde von der Elterngruppe um einen festgelegten Wert nach unten bewegt.

Mergesort:

Der erstmals 1945 durch John von Neumann vorgestellte Mergesort ist ein Sortieralgotithmus, der nach dem Paradigma divide-and-conquer arbeitet. Bei diesem Prinzip wird das eigentliche, große Problem so lange rekursiv in kleinere Probleme unterteilt, bis diese lösbar sind. Im Anschluss wird aus allen Teillösungen die Endlösung rekonstruiert. Die genaue Funktionsweise des Mergesort erfolgt in zwei Schritten: Im ersten Schritt wird die ursprüngliche Liste in zwei Hälften zerlegt, die jeweils wieder in einer Liste gespeichert werden. Dieser Schritt wird, zusammen mit dem nachfolgenden Schritt, so lange rekursiv fortgesetzt, bis sich nur noch ein Element in jeder Liste befindet. Im zweiten Schritt werden die Hälften sortiert und zu einer Menge zusammengefügt, bis sich irgendwann wieder die Gesamtmenge mit allen enthaltenen Ursprungselementen ergibt. Hierbei werden immer die ersten Elemente der beiden Hälften verglichen, wobei das jeweils kleinere in die zusammengefügte Menge wandert.

Laufzeit:

Da sich die Größe der Liste bei jedem Merge verdoppelt, werden log(n) (n = Anzahl der zu sortierenden Elemente) Mergeschritte benötigt, um das Ergebnis vollständig zusammenzusetzen. Bei beispielsweise 8 Elementen ergeben sich insgesamt log2(8) = 3 Schritte.

Jeder Mergeschritt benötigt wiederum n Schritte, um die Elemente der beiden Listen zu sortieren, da hierzu jedes einzelne Elemente betrachtet und eingeordnet werden muss

Die gesamte Laufzeit beträgt also O(n logn). Im Vergleich zu anderen Sortieralgorithmen wie Bubblesort (Worst-Case-Laufzeit: O(n^2)) und Quicksort (Worst-Case-Laufzeit: O(n^2)) ist der Mergesort bei größeren Datenmengen sehr effizient, da dessen Laufzeiten im Best-Case- und im Worst-Case-Szenario kaum Unterschiede aufweisen.

Allgemeine Implementierung in Scala

def sort(list: List[SortElement]) {

if (list.size > 1 ){

val firstListLength = (list.size / 2.0).ceil.toInt

val splitList = list.splitAt(firstListLength)

val left = splitList.\_1

val right = splitList.\_2

sort(left)

sort(right)

merge(list, left, right)

}

}

Die Methode „sort“ nimmt eine Liste von Elementen entgegen und teilt diese in der Mitte in zwei weitere Listen auf. Durch „list.splitAt“ wird ein Tupel erzeugt, welches in der Value splitList gespeichert wird. Mit Hilfe eines Unterstrichs kann entweder auf das erste Feld oder auf das zweite Feld des Tupels zugegriffen werden. Nachdem die ursprüngliche Liste gesplittet wurde, wird auf dem linken Teil rekursiv die Methode „sort“ aufgerufen. Dies sorgt zunächst dafür, dass die linke Liste so lange in zwei Teile aufgeteilt wird, bis sich nur noch ein einziges Element in dieser Liste befindet und die Abbruchbedingung der Rekursion greift. Zu diesem Zeitpunkt befindet sich im rechten Teil der Liste auch nur ein Element und es kommt zur ersten Zusammensetzung zweier sortierter Listen. Diese Methodik wird nun rekursiv für alle Teile der Liste durchgeführt, bis die zwei zu Anfang aufgeteilten Listen sortiert sind und es zum letzten Merge kommt.

Merge-Methode:

def merge(resultList:List[SortElement], leftList:List[SortElement], rightList:List[SortElement]) {

val leftSize: Int = leftList.size

val rightSize: Int = rightList.size

val totalSize: Int = leftSize + rightSize

var i = 0

var j = 0

for (k <- 0 until totalSize) {

if(i < leftSize && j < rightSize){

if (leftList(i) < rightList(j)){

resultList = resultList.updated(k, leftList(i))

i = i + 1

} else {

resultList = resultList.updated(k, rightList(j))

j = j + 1

}

}

else if(i> = leftSize && j< rightSize){

resultList = resultList.updated(k, rightList(j))

j=j+1

} else {

resultList = resultList.updated(k, leftList(i))

i=i+1

}

}

}

Diese Methode nimmt zwei vorsortierte Listen an und setzt alle Elemente dieser Teillisten zu einer einzigen, sortierten Liste zusammen. Hierzu wird für jeden Index „k“ der Ergebnisliste bestimmt, welches Element aus den beiden Listen an dieser Stelle einsortiert wird. Vorausgesetzt, die Ergebnisliste enthält noch nicht alle Elemente mindestens einer Teilliste, werden bei jedem Schritt die jeweils kleinsten Elemente der beiden Teillisten miteinander verglichen, die sich noch nicht in der Ergebnisliste befinden. Das kleinere wird an den aktuellen Index k der Ergebnisliste gesetzt und der Zeiger i bzw. j der Teilliste wird inkrementiert, um auf das nächstgrößere Element zu zeigen. An dieser Stelle macht man es sich zu Nutze, dass die beiden Teillisten vorsortiert sind, da das nächstgrößere Element mit dem Hochzählen des Indexes i bzw. j der Teillisten automatisch ausgewählt wird. In diesem Codebeispiel existiert also eine einzige Liste, die fortlaufend aktualisiert wird, bis sie das Ergebnis beinhaltet. Zur Visualisierung des Algorithmus in unserer Applikation werden wir später für jedes Zwischenergebnis, welches gesplittet oder zusammengesetzt wurde, eine eigene Gruppe anlegen.

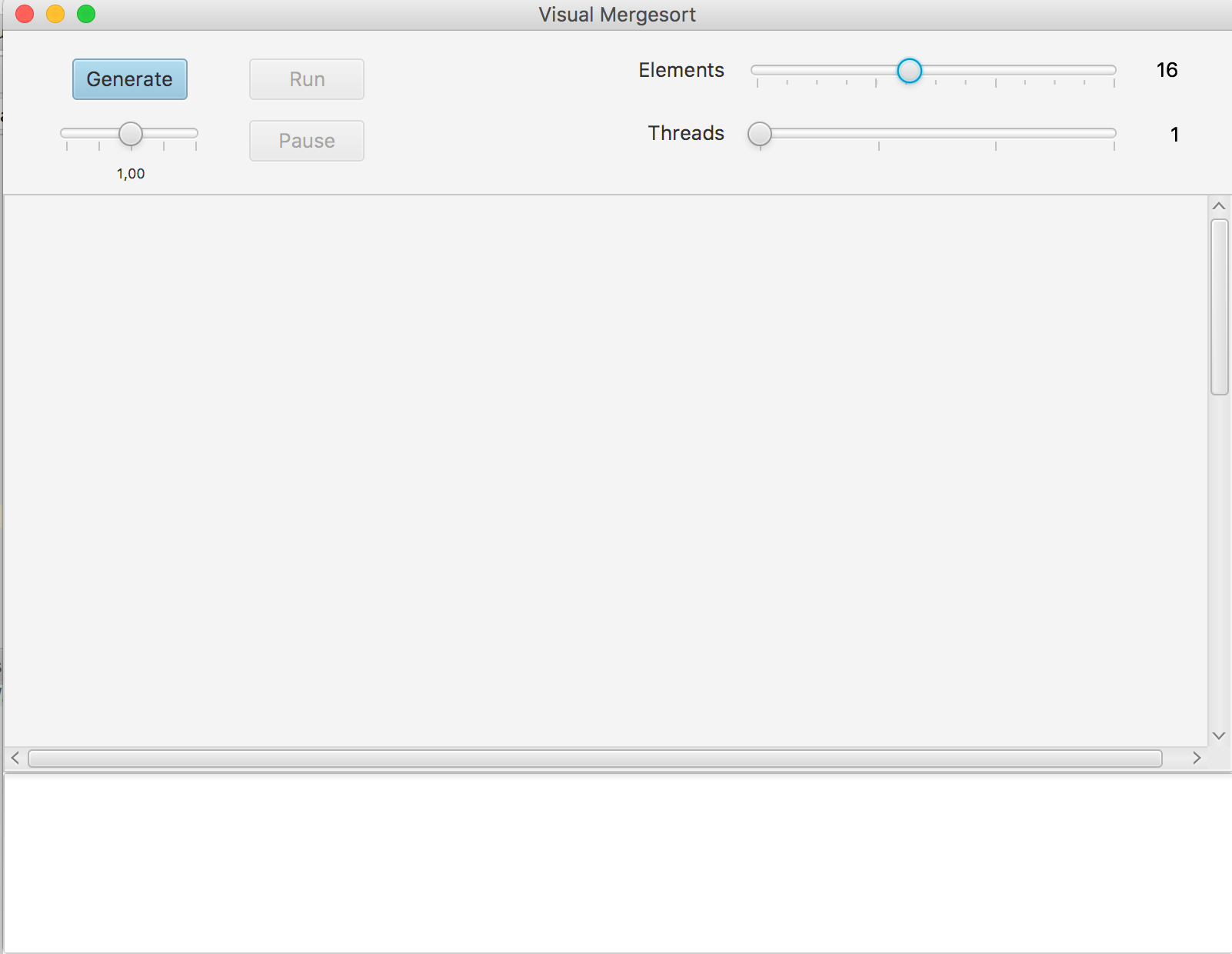
Motivation?

Schon vor der Themenauswahl für die Projektarbeiten waren wir von der Programmiersprache Scala begeistert. Um diese Programmiersprache zu erlernen, suchten wir nach einem Thema, welches sinnvollerweise in Scala implementiert werden könnte. Mit den Neuerungen von ScalaFX kam uns die Aufgabe zur Visualisierung eines parallelisierbaren Sortieralgorithmus sehr entgegen und verleitete uns schließlich zu unserer Entscheidung für dieses Thema.

Bedienungsanleitung

Im Normalfall liegt der Visual-Megesort als Datei mit der Endung .jar vor. Die Dateiendung JAR kennzeichnet Archive, die mehrere Java-Dateien und deren Metainformationen enthalten. Um die Datei ausführen zu können, muss die „Java Runtime Environment“ installiert sein. Die Laufzeitumgebung kann gegebenenfalls kostenlos im Internet heruntergeladen werden.

Startet man die Applikation, so öffnet sich folgendes Fenster:

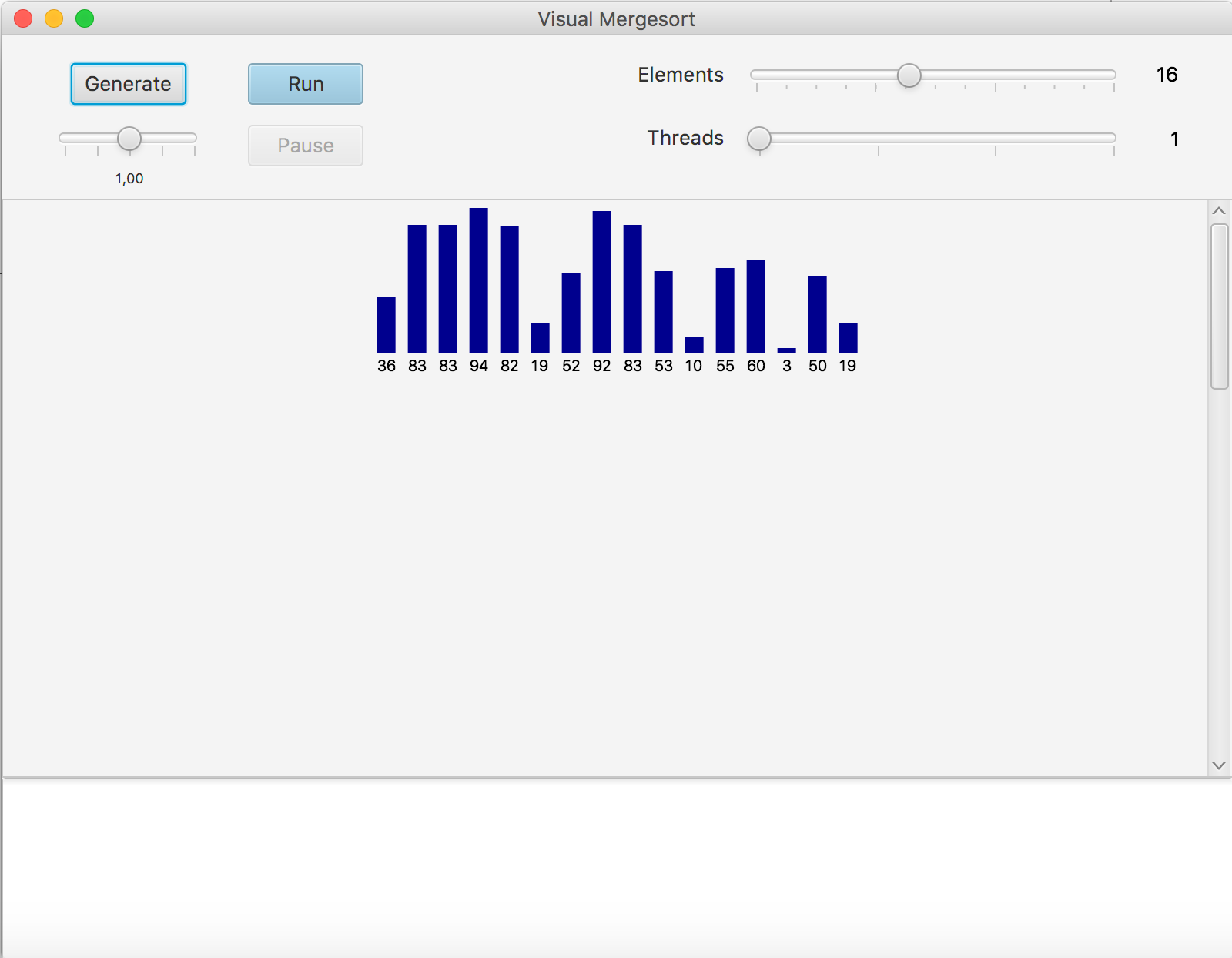
Über ENTER oder das Klicken auf „Generate“ können direkt die über den Slider voreingestellten 16 Zufallselemente in willkürlicher Reihenfolge generiert werden. Alternativ kann über die Menüleiste „File -> Generate Random Data“ eine andere Reihenfolge ausgewählt werden. Um einen schnellen Start mit der gewünschten Menge zu ermöglichen, existieren folgende Shortcuts, welche optional verwendet werden können:

STRG + b -> Generiert eine Menge von Zufallszahlen in willkürlicher Reihenfolge (default)

STRG + o -> Generiert eine Menge von Zufallszahlen in vorsortierter, aufsteigender Reihenfolge

STRG + i -> Generiert eine Menge von Zufallszahlen, welche absteigend sortiert ist.

STRG + u -> Sowohl die Menge der Zahlen als auch die Reihenfolge kann über den Benutzer manuell eingegeben werden.

Wurde eine dieser Optionen und die Anzahl der Elemente über den Slider ausgewählt, werden diese auf der Zeichenfläche platziert:

Mit ENTER, einem Klick auf „Run“ oder dem optionalen Shortcut STRG + r wird der Sortieralgorithmus gestartet. Über den Slider mit der Signatur „Threads“ kann die Anzahl der für den Algorithmus verwendeten Threads eingestellt werden.

Wurde die Animation gestartet, kann diese jederzeit in ihrer Geschwindigkeit über den Regler unter dem Generate-Button variiert oder über den Button Pause bzw. Play komplett pausiert bzw. fortgesetzt werden.