- Ganzzahlen, gleichgültig, welche Größe sie haben und ob sie negative Zahlen beinhalten, haben den Wert 0.
- Fließkommazahlen, gleichgültig welcher Genauigkeit, haben den Wert 0.0.
- ID-Variablen (Zeiger auf Objekte) haben den Wert nil bzw. Nil, wenn sie über den Typen Class auf Klassenobjekte zeigen.
- Andere Zeiger (C-Zeiger) haben den Wert NULL.
- Für Strukturen und C-Felder gelten diese Regeln für jede Komponente.

Ferner wird bereits in der Allokation das erzeugte Instanzobjekt für den Empfang von Nachrichten aufgebaut, erhält also eine Klasse. Dies ist deshalb wichtig, weil das Objekt als nächstes die Initialisierungsnachricht empfangen muss. (In unseren bisherigen Beispielen `–init`.)

`+alloc` selbst übernimmt genau genommen nicht diese Aufgaben, sondern schickt an das Klassenobjekt die Nachricht `+allocWithZone:`. Hierbei wird als Zone `nil` übergeben, was Standardzone bedeutet.

GRUNDLAGEN

»Zones« sind in Cocoa Speicherbereiche, die zusammenhängen. Sie müssen sich nicht um diese Zones kümmern, da Cocoa und OS X selbst eine Standardzone erzeugen.

Es besteht in aller Regel kein Bedarf, die `+alloc`…-Implementierungen von `NSObject` zu verändern.

4.1.2 Initialisierung

Nachdem das Objekt mit `+alloc` erstellt wurde, ist die nächste Nachricht immer eine Initialisierungsnachricht. Diese beginnt mit `init`. Sie sollten (mit sollten im Sinne von dürfen) auch keine andere Methode als einen Initialisierer so bezeichnen. Ja, ich weiß, init ist so ein schönes Wort. Ihnen fällt sicherlich aber im entscheidenen Falle ein anderes ein.

Die Initialisierung dient dazu, die neu erzeugte Instanz mit sinnvollen Werten zu befüllen. Bisher stehen dort ja nur 0-en, was vielleicht nicht zulässig ist. Stellen Sie sich nur vor, jedes Instrument müsste einen Namen haben.

Designated-Initializer

Bisher hatten wir lediglich die Nachricht `–init` verwendet. Eine entsprechende Methode wurde von uns nicht in den abgeleiteten Klassen `Instrument`, `Guitar` und `Piano` implementiert. Dadurch wurde sie von `NSObject` geerbt. Da diese Methode nichts macht, bleibt es dabei, dass alle Instanzvariablen auf 0-en stehen. Das bedeutet also, dass die Eigenschaft `age` 0 ist, die Eigenschaft `price` 0 und die Eigenschaft `name` `nil`. Wir können das mal testen. Dazu ändern Sie bitte main.m:

```
int main (int argc, const char * argv[])
{
   @autoreleasepool {
       Instrument *instrument = [[Instrument alloc] init];
       NSLog( @"%@ %ld %ld", instrument.name, instrument.age, instrument.price );
   }
   return 0;
}
```

Als Ausgabe erhalten wir entsprechend:

```
>… (null) 0 0
```

Das *(null)* bedeutet übrigens bei Objekten `nil`. Deshalb hatten wir auch häufig als erstes dem Instrument einen Namen gegeben.

Aus diesem Grunde bietet man neben dem von `NSObject` ererbten Initialisierer `–init` weitere an, die die Erstellung einer Instanz erleichtern. Das machen wir jetzt auch. Zunächst geben wir dem Header-Instrument eine neue Methode:

```
@interface Instrument : NSObject <Play>
@property (copy)   NSString  *name;
@property NSInteger  age;
@property NSInteger  price;

#pragma mark Creation
- (id)initWithName:(NSString*)name age:(NSInteger)age price:(NSInteger)price;

#pragma mark Aging
- (void)growOld;
@end
```

Diese Methode muss freilich dann auch implementiert werden.

```
- (id)initWithName:(NSString*)name age:(NSInteger)age price:(NSInteger)price
{
   // Zunaechst Initialisierung der Superklasse ausfuehren
   self = [super init];

   // Bei Erfolg ...
   if( self ) {
```

```
        // ... eigene Instanzvariablen setzen
        self.name = name;
        self.age = age;
        self.price = price;
    }
    return self;
}
@end
```

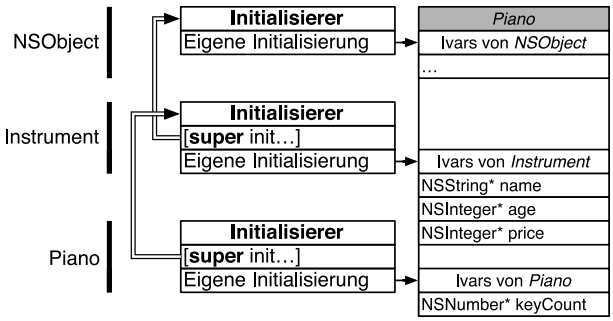
TIPP

Ich implementiere die Initialisierer immer am Ende einer Klasse. Dies hat den Vorteil, dass ich sämtliche bestehenden Methoden davor stehen habe, welche deshalb von mir benutzt werden können. Umgekehrt wird so gut wie nie eine andere Methode einen Initialisierer nutzen. Die Initialisierungsnachricht kommt ja von außen. Und deshalb befinden sich Initiailisierer ja auch im Header. Nach der Regel, dass Methoden erst benutzt werden dürfen, nachdem sie deklariert wurden, verhindere ich so Kollisionen. Ich nannte das bereits einmal Bottom-Up.

Gehen wir das mal durch:

```
self = [super init];
```

Auffallend ist hier, dass das `super` zum ersten Mal von uns verwendet wird. Aufgepasst, jetzt kommt eine goldene Regel: Wenn sich eine Instanz einer Subklasse initialisiert, muss sie zunächst der Basisklasse die Gelegenheit geben, sich ebenfalls zu initialisieren. Es könnte ja sein, dass in `NSObject` bereits Instanzvariablen definiert wurden, die ebenfalls auf vernünftige Werte vorgesetzt werden müssen. Auch wird das klar, wenn man bedenkt, dass wir ja noch Subklassen haben, nämlich `Guitar` und `Piano`. Auch diese werden gleich Initialisierer erhalten. – Und auch diese müssen dann `Instrument` die Gelegenheit geben, sich zu initialisieren. Denn nur jede Klasse für sich weiß, was bei der Initialiserung ihrer Instanzen geschehen muss. Die Initialisierung läuft also die Klassenhierarchie hoch. Da jedoch diese Nachricht die erste in jedem Initialisierer ist, wird die Initialisierung ausgehend von der obersten Klasse nach unten durchlaufen.



Von oben nach unten: Jeder initialisiert die von ihm definierten Instanzvariablen.

Für Umsteiger von anderen Programmiersprachen ist es auch außerordentlich ungewöhnlich, dass die Initialisierung einen Rückgabewert hat. Und der wird dann auch noch an `self` zugewiesen. `self` war ja der Objektkontext. Das bedeutet also nichts anderes, als dass, während die Initialisierungsmethode läuft, das Instanzobjekt ausgetauscht werden kann. Tatsächlich gibt es dafür Anwendungsfälle, die allerdings fortgeschrittene Technologien betreffen. Für Sie ist es nur wichtig zu wissen, dass es diesen Rückgabewert gibt und dass er an `self` zugewiesen werden muss.

Kommen wir zum if: Jeder Initialisierer ist berechtigt, die Initialisierung als gescheitert zu markieren. Er muss dann den Wert `nil` zurückliefern. Der Objektkontext existiert dann also nicht mehr, und die Initialisierung muss abgebrochen werden. Wenn `self` `nil` ist, wird unser If-Zweig nicht ausgeführt. (Bei Objektzeigern bedeutet `nil` so viel wie unwahr.)

In den If-Zweig, wenn also die Initialisierung funktionierte, setzen wir dann unsere Instanzvariablen auf vernünftige Werte vor.

Hier existiert eine ewige Diskussion, ob auch im `–init…` die Accessoren benutzt werden sollen, wie ich es im Beispiel mache. Das Allerwichtigste an dieser Diskussion: Es spielt fast nie eine Rolle. Ich mache es so aus den folgenden Gründen:

- Zugriffe mit Accessoren sind gut (gekapselt), direkte Zugriffe sind böse. Das gilt unbestritten in jeder anderen Methode. (Mit Ausnahme des spiegelbildlichen `–dealloc`, wozu wir noch kommen.) Wieso also nicht im `–init…`?
- Der Anwender einer Klasse erwartet, dass Parameter genau so die Instanzvariablen setzen wie es beim Setzen der Eigenschaften mit den Accessoren der Fall ist. Das führt zu unterschiedlichem Code, je nach dem ob eine `copy`-, retain- oder `weak`-Property betroffen ist. Dieselbe Information wird also an zwei Stellen (Property, Initialisierung) abgehandelt. Das ist grundsätzlich nachteilig. Und wieso sollte man nicht Accessoren verwenden, wenn das Verhalten wie bei Accessoren sein soll?

Um es klar zu sagen: Apple empfiehlt das unmittelbare Setzen der Instanzvariablen. Begründet wird dies nicht. Eine häufig gehörte Begründung ist allerdings, dass theoretisch eine Subklasse die Accessoren überschrieben haben kann. Dann würde der Aufruf von `–init…` (super) in der Subklasse dazu führen, dass eine Subklassenmethode – nämlich der überschriebene Setter – ausgeführt wird, bevor die Subklasse ihr eigenes `–init…` durchlaufen hat. Das stimmt. Nur gilt dies grundsätzlich für jede Methode, die in `–init…` benutzt wird. Es ist einfach ein Umstand, den man kennen sollte. Dann halte ich es für ungefährlich. Übrigens: Auch Apple hält sich nicht immer an die eigene Empfehlung.

Wenn man aber lieber die Instanzvariablen direkt, also ohne Accessoren, setzt, dann gelten folgende Regeln:

Hängt an der Instanzvariable eine Eigenschaft, die mit `copy` spezifiziert wurde, so ist eine Kopie zu bilden:

```
_name = [name copy];
```

Bedenken Sie hier im Hinblick auf die Synthetisierung, dass die Instanzvariable einen führenden Unterzug (_) hat. Ansonsten: Kopien besprechen wir noch. Aber der Code sollte klar sein und kann jedenfalls so übernommen werden.

Hängt an der Instanzvariable eine Eigenschaft, die mit `weak` spezifiziert wurde (äußerer Verweis), muss der Parameter unmittelbar zugewiesen werden. Beispiel für `name`:

```
_name = name;
```

Ist es eine `strong`-Eigenschaft (äußerer Verweis), so kommt es auf die verwendete Speicherverwaltung an: Bei automatischer Speicherverwaltung erfolgt eine Zuweisung wie bei `weak`. Bei manueller Speicherverwaltung lautet die Zuweisung:

```
_name = [name retain];
```

Ist der Typ der Instanzvariable ein C-Typ, so ist wieder einfach zuzuweisen. Als Beispiel kann `age` dienen:

```
_age = age.
```

Sie sehen schon: Hier spielen mit `-copy`, `-retain` und einfacher Zuweisung ganz andere Themen in die Initialisierung hinein. Diese werden eigentlich von den Settern abgehandelt. Genau das meinte ich oben mit dem zweiten Nachteil.

Wie dem auch sei: Ich verwende Setter, wie Sie es im Beispielcode von oben sehen.

Am Ende der Methode geben wir dann die initialisierte Instanz mit `return` zurück. Sollte die Initialisierung in `–init` (`super`) gescheitert sein, so steht `self` auf `nil`, so dass `nil` zurückgegeben wird.

Wir probieren jetzt unseren neuen Initialiserer in main() aus:

```
int main (int argc, const char * argv[])
{
   @autoreleasepool {
      Instrument *instrument
      = [[Instrument alloc] initWithName:@"Gitarre" age:0 price:1000];
      NSLog( @"%@ %ld %ld", instrument.name, instrument.age, instrument.price );
   }
   return 0;
}
```

Starten Sie das Programm. Die Ausgabe im Log sollte entsprechend lauten.

Secondary-Initializer

Unser erster Initialisierer hatte einen Parameter für jede Eigenschaft. Das führte dazu, dass er sehr lang ist. Wir mussten ja sogar auf eine zweite Zeile ausweichen. Dabei ist das vielleicht gar nicht nötig. `age` ist eine Eigenschaft, die ja mit 0 ganz gut vorgesetzt wird.

Aus diesem Grunde kann es für den Nutzer einer Klasse bequem sein, weitere Initialisierer zu haben, die weniger Parameter bekommen und die ausgelassenen mit sinnvollen Werten vorsetzen. Erweitern wir die Klasse `Instrument` um einen solchen:

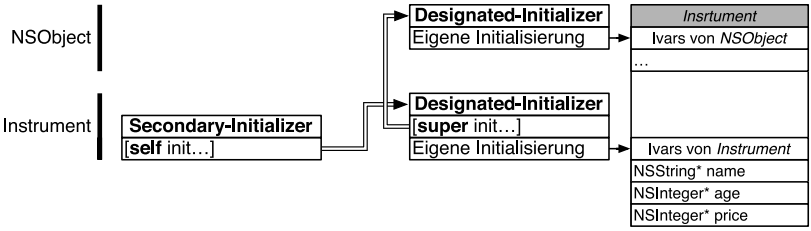
```
@interface Instrument : NSObject <Play>
…
#pragma mark Creation
- (id)initWithName:(NSString*)name age:(NSInteger)age price:(NSInteger)price;
- (id)initWithName:(NSString*)name price:(NSInteger)price;
…
@end
```

Jetzt müsste man also noch einmal einen solche Initialisierer ebenfalls implementieren. Also noch einmal die gesamte Arbeit von vorhin? Nein, das brauchen wir nicht. Eigentlich macht unser großer Initialisierer ja schon alles. Daher machen wir es uns jetzt einfach:

```
- (id)initWithName:(NSString *)name price:(NSInteger)price
{
    return [self initWithName:name age:0 price:price];
}
@end
```

Sie sehen: In diesem Falle wird einfach der größere Initialisierer ausgeführt und dabei der fehlende Parameter eingesetzt.

Moment, höre ich Sie sagen: Aber müssen wir nicht zuerst den Initialisierer der Basisklasse aufrufen? Ja, das tun wir aber auch. Denn bevor `–initWithName:age:price:` die eigenen Instanzvariablen setzt, ruft dieser Initialisierer ja den Initialisierer der Basisklasse auf. Die Kette lautet also: Vom Secondary-Initializer zum Designated-Initializer mit `self`, vom Designated-Initializer der Subklasse zum Designated-Initializer der Basisklasse mit super.



Vom Secondary zum Designated geht es mit self horizontal.

Und jetzt kann ich auch die unterschiedliche Bezeichnung in den Überschriften aufklären: Ein Secondary-Initializer ist einer, der einen eigenen Designated-Initializer aufruft. Ein Designated-Initializer ist einer, der einen Designated-Initialisierer der Basisklasse aufruft. Es stellen sich zwei Fragen:

Wie erfahre ich, was der Designated-Initializer der Basisklasse ist? Das steht in der Dokumentation. Man kann es aber recht leicht erkennen. Es ist nahe zu immer derjenige, der die meisten Parameter hat. Denn wenn der Secondary-Initializer den Designated-Initializer nutzt, kann der Secondary-Initializer jedenfalls nicht ohne Weiteres mehr Parameter haben. Umgekehrt mache ich es ebenso bei eigenen Klassen: Designated-Initializer ist in der Regel derjenige, der die meisten Parameter bekommt. Hiermit erpare ich mir Tipparbeit – und anderen, wie wir gleich sehen werden.

Der zweite Punkt ist, dass ich oben von »einem« Desiganted-Initializer gesprochen habe. Kann es davon auch mehrere geben? Ja, es gibt recht häufig einen weiteren mit dem Namen `–initWithCoder:`, der eine Instanz aus einer Datei erzeugt. (Das ist hier etwas verkürzt. Wir beschäftigen uns damit noch im Kapitel über die Modelschicht.) Grundsätzlich darf eine Klasse so viele Designated-Initializer bestimmen, wie sie will. Es bedeutet nur mehr Aufwand. Von dem Falle des Coders aber mal abgesehen, existiert bei den allermeisten Klassen jedoch nur einer und zwar der jenige, der die meisten Parameter nimmt.

Gut, da uns die 0 beim Alter ganz gelegen kommt, verbessern wir wieder das Hauptprogramm:

```
int main (int argc, const char * argv[])
{
   @autoreleasepool {
      Instrument *instrument
      = [[Instrument alloc] initWithName:@"Gitarre" price:1000];
      NSLog( @"%@ %ld %ld", instrument.name, instrument.age, instrument.price );
   }
   return 0;
}
```

Sollte auch funktionieren.

Man kann sich jetzt freilich weitere Secondary-Initializer schreiben. Allerdings erscheint es durchaus sinnvoll, wenn Name und Preis gleich angegeben werden. In der Regel will man das ja setzen. Ich verzichte daher hier darauf, weitere Secondary-Initializer anzulegen.

Ererbte Initialisierer

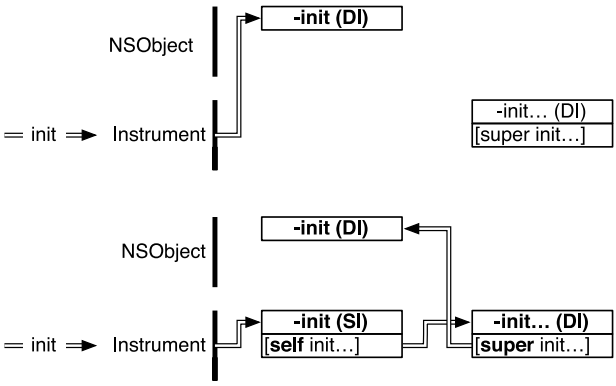
Na ja, so ganz kann ich nicht darauf verzichten. Immerhin hat unsere Klasse `Instrument` noch `–init` von `NSObject` geerbt. Und das ist jetzt wirklich misslich: Wird diese Nachricht an das neu erzeugte Objekt gesendet, so wird `–init` von `NSObject` ausgeführt, und unsere

Initialisierer kommen überhaupt nicht zum Laufen. Das gilt es zu ändern. Dabei sind zwei Fälle zu unterscheiden:

Wechsel des Designated-Initializers

Häufig stellt sich die Situation dann, wenn der Designated-Initializer gewechselt wird. So ist es auch hier: Der Designated-Initializer von `NSObject` ist `–init`. Es ist ja der einzige Initialisierer. Der Designated-Initializer von `Instrument` ist `–initWithName:age:price:`, also ein anderer. In diesen Fällen ist der Designated-Initializer der Subklasse zu überschreiben, und zwar so, dass er nun als – neuer – Secondary-Initializer den eignen Designated-Initializer aufruft:

```
- (id)init
{
    return [self initWithName:@"Any Instrument" age:0 price:0];
}
@end
```



Bei einem Wechsel des DI muss dieser überschrieben werden (unten), da sonst die eigene Klasse nicht initialisiert wird (oben).

Übernahme von Secondary-Initializern

Dies gilt übrigens nicht für die Secondary-Initializer. Da `NSObject` über keine solche verfügt, schauen wir uns das auf einer Ebene tiefer an. Geben Sie zunächst `Piano` einen neuen Designated-Initializer:

```
@interface Piano : Instrument
@property (copy) NSNumber* keyCount;

- (id)initWithName:(NSString *)name
               age:(NSInteger)age
             price:(NSInteger)price
          keyCount:(NSNumber*)keyCount;
@end
```

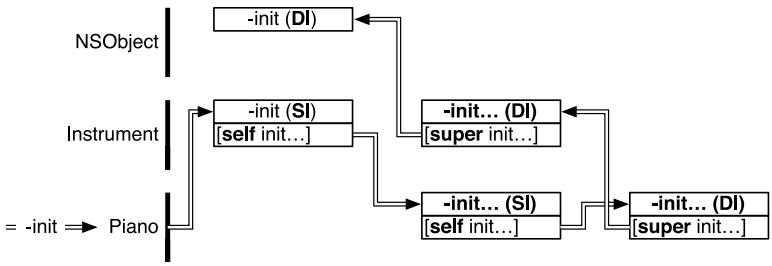
Den wir entsprechend implementieren:

```
@implementation Piano
- (id)initWithName:(NSString *)name
               age:(NSInteger)age
             price:(NSInteger)price
          keyCount:(NSNumber *)keyCount
{
    self = [super initWithName:name age:age price:price];
    if( self ) {
        self.keyCount = keyCount;
    }
    return self;
}
@end
```

Hier ist nichts Neues zu erkennen: Wir haben einen Designated-Initializer, der den Designated-Initializer der Basisklasse nutzt. Klar ist auch, dass wir einen Wechsel des Designated-Initializers haben, wir also entsprechend weiterleiten müssen:

```
- (id)initWithName:(NSString *)name age:(NSInteger)age price:(NSInteger)price
{
    return [self initWithName:name
                          age:age
                        price:price
                     keyCount:[NSNumber numberWithInt:42]];
}
@end
```

Bis hierhin ist das also nur Übung von Bekanntem. Aber es existieren in der Basisklasse `Instrument` ja auch noch die Secondary-Initialiser `–init` und `–initWithName:price:`. Besteht bei denen nicht ebenfalls das Problem, dass die Initialisierung an Subklasse vorbei läuft? Nein, wenn wir uns das einmal in der Graphik anschauen:



Die Secondary-Initializer führen zu überschriebenen Designated-Initializer.

Der Trick liegt also darin, dass ein Secondary-Initializer ja ohnehin einen Designated-Initializer ausführt. Da ber Designated-Initializer in Subklassen bei einem Wechsel überschrieben wird, führt dies zur Ausführung des Codes in der Subklasse. Alles okay. Auch hier ist es also so, dass die Anzahl der Designated-Initializer den Aufwand bestimmt.

TIPP

Es kann dennoch gewünscht sein, auch einen Secondary-Initializer in einer Subklasse zu überschreiben. In `Piano` so zu überschreiben, dass als Standardname nicht @"Any Instrument", sondern @"Any Piano" gesetzt wird. Aber einen Zwang dazu gibt es eben nicht.

Beachten Sie bitte in der Graphik noch einen wichtigen Punkt, der unabhängig von der Initialisierung interessant ist: Es wird die Nachricht `–init` an eine Piano-Instanz gesendet. Da `Piano` `–init` nicht implementiert, rutscht die Ausführung in `–init` (`Instrument`) hoch. Diese Implementierung leitet die Nachricht an `–initWithName:age:price:` (`self`) weiter. Diese Methode gibt es zweimal. Es wird die Variante von `Piano` gewählt, obwohl die Nachricht in `Instrument` steht. Das liegt daran, dass `self` auf eine Instanz von `Piano` zeigt. Es kommt also – wie auch bei allen anderen Nachrichten – bei einer Nachricht an `self` nicht darauf an, von welcher Klasse die Nachricht aus erfolgt, sondern darauf, welche Klasse der Empfänger der Nachricht hat. Und das ist eben in dem Beispiel eine Instanz von `Piano`.

Übernahme des Designated-Initializers

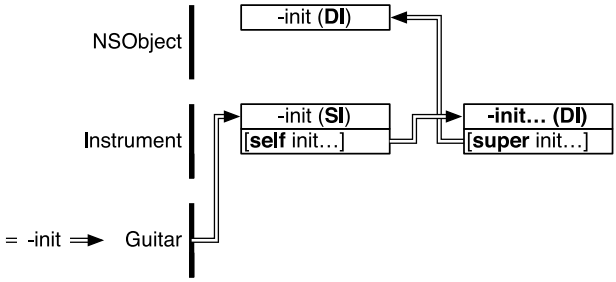
Es ist aber übrigens auch nicht zwingend notwendig, den Designated-Initializer zu überschreiben. Man kann auch den der Basisklasse in einer Subklasse verwenden. Das ist dann nahe liegend, wenn in der Subklasse keine neuen Instanzvariablen auftauchen, die es zu initialisieren gilt oder aber diese Instanzvariablen durchaus mit 0 starten sollen, so dass man sich einen größeren Designated-Initializer erspart.

GRUNDLAGEN

Soll eine Instanzvariable mit dem Wert 0 initialisiert werden, so ist es untunlich, dies explizit in einem Initialisierer zu machen. Der Allocator setzt den Wert bereits auf 0.

Ganz, ganz zufällig haben wir sogar einen solchen Fall: `Guitar`. Hier wurde ja nichts hinzugefügt. Also gibt es auch keine Notwendigkeit, eigene Instanzvariablen vorzusetzen. Also gibt es auch keine Notwendigkeit für einen eigenen Designated-Initializer.

Funktioniert das auch? Schauen wir uns das mal graphisch an:



Ist kein eigener Designated-Initializer vonnöten, kann die Arbeit der Basisklasse überlassen werden.

Wird an eine Instanz von `Piano` die Nachricht `–init` geschickt, so findet sich in der Klasse `Piano` dafür keine Methode. Also wird es zu `Instrument` hoch gereicht. Das Ganze landet bei `–init` (`Instrument`). Diese Methode sendet jetzt wieder an `self` die Nachricht `–initWithName:age:price:`, was dazu führt, dass bei `Guitar` nach einer entsprechenden Methode gesucht wird. Wieder kein Treffer und daher hoch zu `Instrument`.

Jetzt wird es etwas knifflig: `–initWithName:age:price:` schickt eine init-Nachricht nicht an `self`, sondern an `super`. Was heißt das? `self` zeigt ja immer noch auf eine Instanz von `Guitar`. So betrachtet würde `super` bedeuten, dass die Nachricht an `Instrument`, die Superklasse von `Guitar`, geschickt wird. Das wäre aber misslich, denn da kommen wir gerade her. Die Nachricht würde durch die Initialisierer kreisen und zwar bis in alle Ewigkeit.

Und hier gibt es eine wichtige Regel, die ich bereits im Kapitel 3 ansprach: Bei einer Nachricht, die an den Empfänger `super` gesendet wird, kommt es nicht darauf an, welche Klasse der Empfänger hat, hier also `Guitar`. Vielmehr ist es entscheidend, von wo die Nachricht geschickt wird. Also gerade anders, als ich das vor wenigen Zeilen noch mit `self` erläutert habe. Weil diese Nachricht von `Instrument` gesendet wird, landet sie bei `NSObject`, also der Superklasse von `Instrument`. Dass wir es gerade mit einer Instanz von `Piano` zu tun haben, ist hier unbeachtlich. Man nennt das statische Bindung:

GRUNDLAGEN

Wenn die Superklasse keine entsprechende Methode bietet, wird die Nachricht freilich weiter nach oben durchgereicht. Statische Bindung heißt hier also nicht, dass es keine Polymorphie mehr gäbe. Es heißt eben nur, dass das `super` nicht auf den Empfänger der Nachricht zur Laufzeit bezogen ist, sondern auf den Absender, der bereits zur Übersetzungszeit feststeht.

Auch hier kann es natürlich sinnvoll sein, den Designated-Initializer der Basisklasse zu überschreiben. Erforderlich ist es nicht.

Zusammenfassung

Weil die Erläuterungen recht komplex waren und verschiedene Szenarien betrafen, ist es an der Zeit, das Ganze wieder zu einer Quintessenz zu reduzieren:

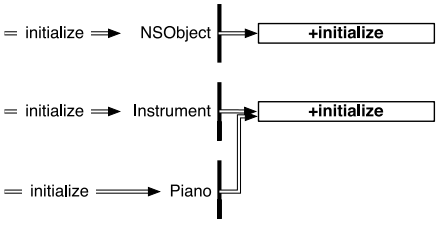
- Jede Klasse hat mindestens einen, jedoch möglichst wenige parameterreiche Designated-Initializer. Diese führen zunächst mit super den Designated-Initializer der Basisklasse auf und initialisieren dann ihre eigenen Instanzvariablen.
- Jede Klasse kann parameterärmere Secondary-Initializer haben, die mit `self` ihren Designated-Initializer ausführen, wobei sie eine Parametrisierung mit Standardwerten vornehmen.
- Eine Subklasse muss den Designated-Initializer selbst implementieren, wenn sie neue Instanzvariablen einführt, die sie auf andere Werte als die 0-Werte des Allocators setzen möchte. Ansonsten kann sie den Designated-Initializer der Basisklasse mitbenutzen.
- Eine Subklasse muss den Designated-Initializer der Basisklasse überschreiben, wenn sie den Designated-Initializer ändert. In der neuen Implementierung muss eine Weiterleitung von dem alten an den neuen Designated-Initializer mittels `self` erfolgen.

Initialisierung von Klassenobjekten

Wesentlich einfacher ist die Initialisierung von Klassenobjekten. Zunächst ist zu erwähnen, dass diese fast nie notwendig ist. Klassenobjekte speichern ja keine Daten, haben also keine Instanzvariablen. Wenn dennoch etwas getan werden soll, so kann das optional in der Methode `+initialize` erfolgen. Zwei wichtige Unterschiede: Zum einen hat diese Methode keinen Rückgabewert. Es ist also nicht möglich, bei der Initialisierung das Klassenobjekt auszutauschen. Ist auch irgendwie klar: Die Herstellung wird ja vom System vorgenommen.

Zum anderen darf nicht eine initialize-Nachricht an `super` gesendet werden. Das macht bereits das System, indem es beginnend mit der Wurzelklasse alle Klassenobjekte anspricht. Letztlich läuft es also darauf hinaus, dass in `+initialize` einfach der Code untergebracht werden muss, den man unterbringen will. Frei von allem Ballast.

Es gibt nur eine Besonderheit zu beachten: Da das System selbsttätig `+initialize` für jede Klasse aufruft, erfolgt dies immer für eine Basisklasse und für deren Subklassen. Implementieren die Subklassen nicht `+initialize`, so wird diese Nachricht wieder an die Basisklasse durchgereicht. Die erhält dann die Nachricht ein weiteres Mal.



Klassenobjekte erhalten automatische Initialisierungsnachrichten, und zwar auch dann, wenn sie +initialize nicht implementieren.

Wenn man sicherstellen möchte, dass die eigene Methode `+initialize` nicht mehrfach ausgeführt wird, so kann man das mit einer Abfrage bewerkstelligen. Ein Beispiel für `Instrument`:

```
+ (void)initialize
{
   if( self == [Instrument class] ) {
      // Initialisierungscode für Instrument
   }
}
```

Das funktioniert, weil, sofern die initialize-Nachricht an das Klassenobjekt von `Piano` geschickt wird, `self` auf das Klassenobjekt von `Piano` zeigt und daher die Bedingung in dem If nicht erfüllt ist.

Tatsächlich benötigt man derlei aber selten und vor allem immer seltener. Der frühere Standardfall für Key-Value-Observing ist weggefallen, wie wir noch dort besprechen werden.

HILFE

Sie können das Projekt in diesem Zustand als Projekt Objective-C-14 von der Webseite herunterladen.

4.1.3 Convenience-Allocators und +new…

Sie haben es schon bemerkt: Die Nachricht zur Erzeugung lautet `alloc`. Danach muss eine Initialisierungsnachricht verschickt werden. Man kann das dann ja doch zusammenfassen in einer Nachricht und so dem Nutzer einer Klasse noch mehr Komfort bieten. Dabei gibt es aus Gründen der Speicherverwaltung zwei Methodengruppen:

- Die `new`-Methoden kombinieren `+alloc` und `–init`… Im Bereich Speicherverwaltung werden Sie allerdings sehen, dass eine weitere wichtige Nachricht unmittelbar auf die Initialisierung folgt.
- Die Convenience-Allocators nehmen daher auch diese Nachricht, es handelt sich um autorelease, auf.

Die Unterscheidung ist vor allem dann wichtig, wenn Sie mit manueller Speicherverwaltung arbeiten, da dann die autorelease-Nachricht explizit im Code stehen muss. Dies führte in den letzten Jahren dazu, dass viele Convenience-Allocatoren für die bestehenden Klassen geschrieben wurden. Mit der neuen automatischen Speicherverwaltung werden diese Speicherverwatungsnachrichten durch den Compiler erzeugt, so dass der Unterschied jedenfalls auf den ersten Blick gar nicht auffällt. Der Satz an Convenience-Allocatoren ist