KAPITEL 4

Referenztypen

Referenztypen halten Referenzen auf Objekte und verschaffen Ihnen die Möglichkeit, auf Objekte zuzugreifen, die irgendwo im Speicher festgehalten werden. Der Speicherort ist für den Programmierer irrelevant. Alle Referenztypen sind Unterklassen von `java.lang.Object`. Tabelle 4-1 führt die fünf Java-Referenztypen auf.

Tabelle 4-1: Referenztypen

Referenztyp	Kurzbeschreibung
Annotation	Ein Mittel, Metadaten (Daten über Daten) mit Programmelementen zu verbinden.
Array	Eine Datenstruktur unveränderlicher Größe, die Datenelemente des gleichen Typs speichert.
Class	Bietet Vererbung, Polymorphie und Kapselung. Modelliert üblicherweise etwas aus dem wahren Leben und besteht aus einer Sammlung von Werten, die Daten festhalten, und einem Satz von Methoden, die auf diesen Daten operieren.
Enumeration	Eine Referenz auf eine Menge von Objekten, die eine Auswahl aufeinander bezogener Optionen repräsentieren.
Interface	Stellt eine öffentliche API und wird von Java-Klassen »implementiert«.

Elementare Typen und Referenztypen im Vergleich

In Java gibt es zwei Typkategorien: Referenztypen und elementare Typen. Tabelle 4-2 zeigt einige der entscheidenden Unterschiede zwischen beiden auf. Weitere Details finden Sie in Kapitel 3.

Tabelle 4-2: Referenztypen und elementare Typen im Vergleich

Referenztypen	Elementare Typen
Unbeschränkte Anzahl von Referenztypen, da diese durch die Nutzer definiert werden.	Besteht aus dem Typ `boolean` und den numerischen Typen `char`, `byte`, `short`, `int`, `long`, `float` und `double`.
Der Speicherort speichert eine Referenz auf die Daten.	Der Speicherort speichert die tatsächlich von einem elementaren Typ festgehaltenen Daten.
Wird ein Referenztyp einem anderen Referenztyp zugewiesen, weisen beide auf das gleiche Objekt.	Wird ein elementarer Wert einer anderen Variablen des gleichen Typs zugewiesen, wird eine Kopie erstellt.
Wird ein Objekt an eine Methode übergeben, kann die aufgerufene Methode den Inhalt des übergebenen Objekts ändern, aber nicht die Adresse des Objekts.	Wird ein elementarer Wert an eine Methode übergeben, wird nur eine Kopie übergeben. Die aufgerufene Methode hat keinen Zugriff auf den ursprünglichen elementaren Wert und kann ihn deswegen nicht ändern. Die aufgerufene Methode kann nur den kopierten Wert ändern.

Vorgabewerte

Vorgabewerte sind die Werte, die in Java Instanzvariablen zugewiesen werden, wenn nicht explizit ein Initialisierungswert gesetzt wurde.

Instanzvariablen und lokale Variablen

Instanzvariablen (also jene, die auf Klassenebene deklariert werden) haben den Vorgabewert `null`. `null` referenziert nichts.

Lokale Variablen (jene, die in einer Methode deklariert werden) haben keinen Vorgabewert, nicht einmal den Wert `null`. Deswegen sollten Sie lokale Variablen immer initialisieren. Wird eine nicht initialisierte lokale Variable auf einen Wert geprüft (gilt auch für eine Prüfung auf `null`), führt das zu einem Compilerfehler.

Obgleich Objektreferenzen mit dem Wert `null` kein Objekt auf dem Heap referenzieren, können auf `null` gesetzte Objekte in Code referenziert werden, ohne dass das zu Compiler- oder Laufzeitfehlern führt:

```
LocalDate birthdate = null;
// Lässt sich kompilieren.
if (birthdate == null) {
  System.out.println(birthdate);
}
$ null
```

Der Aufruf einer Methode auf einer Referenzvariablen, die `null` ist, oder die Verwendung des Punktoperators auf dem Objekt führt zu einer `java.lang.NullPointerException`:

```
final int MAX_LENGTH = 20;
String partyTheme = null;
/*
 * java.lang.NullPointerException wird ausgelöst,
 * da partyTheme null ist
 */
if (partyTheme.length() > MAX_LENGTH) {}
```

Arrays

Arrays erhalten immer einen Vorgabewert, unabhängig davon, ob sie als Instanzvariablen oder als lokale Variablen deklariert sind. Arrays, die deklariert, aber nicht initialisiert sind, erhalten den Vorgabewert `null`.

Im folgenden Code wird zwar das Array `gameList1` initialisiert, nicht aber die einzelnen Werte darin. Das bedeutet, dass die Objektreferenzen den Wert `null` haben. Die Objekte müssen dem Array noch hinzugefügt werden:

```
/*
 * Die deklarierten Arrays gameList1 und
 * gameList2 werden per Vorgabe auf null initialisiert.
 */
Game[] gameList1;
Game gameList2[];

/*
 * Das folgende Array wurde initialisiert, aber
 * die Objektreferenzen sind weiter null, weil
 * das Array keine Objekte enthält.
 */
gameList1 = new Game[10];
```

```
/*
 * Fügen wir der Liste nun ein Game-Objekt hinzu,
 * sodass sie ein Objekt enthält.
 */
gameList1[0] = new Game();
```

Mehrdimensionale Arrays sind in Java eigentlich Arrays mit Arrays. Sie können mit dem new-Operator initialisiert werden oder indem ein Initialisierungsausdruck in geschweiften Klammern angegeben wird. Mehrdimensionale Arrays können gleichförmig oder unregelmäßig sein:

```
// anonymes Array
int twoDimensionalArray[][] = new int[6][6];
twoDimensionalArray[0][0] = 100;
int threeDimensionalArray[][][] = new int[2][2][2];
threeDimensionalArray[0][0][0] = 200;
int varDimensionArray[][] = {{0,0},{1,1,1}, {2,2,2,2}};
varDimensionArray[0][0] = 300;
```

Anonyme Arrays ermöglichen die Erstellung neuer Arrays mit Werten an beliebiger Stelle im Code:

```
// Beispiele für die Verwendung anonymer Arrays
int[] luckyNumbers = new int[] {7, 13, 21};
int totalWinnings = sum(new int[] {3000, 4500, 5000});
```

Umwandlung von Referenztypen

Ein Objekt kann in den Typ seiner Oberklasse (erweitert) oder in den Typ seiner Unterklassen (eingeengt) umgewandelt werden.

Der Compiler prüft Umwandlungen bei der Kompilierung, und die *Java Virtual Machine* (JVM) prüft sie zur Laufzeit.

Erweiternde Umwandlung

- Eine Erweiterung wandelt eine Unterklasse implizit in ihre Elternklasse (Oberklasse) um.
- Erweiternde Umwandlungen lösen keine Laufzeit-Exceptions aus.
- Es ist kein expliziter Cast erforderlich:

```
String s = new String();
Object o = s; // erweiternd
```

Einengende Umwandlung

- Eine Einengung wandelt einen allgemeineren Typ in einen spezifischeren Typ um.
- Eine Einengung ist die Umwandlung einer Oberklasse in eine Unterklasse.
- Es ist ein expliziter Cast erforderlich. Sie casten ein Objekt auf ein anderes Objekt, indem Sie den Typ des Objekts, auf das gecastet werden soll, in Klammern unmittelbar vor dem Objekt angeben, das gecastet werden soll.
- Unzulässige Einengungen führen zu einer `ClassCastException`.
- Einengung kann zum Verlust von Daten oder von Genauigkeit führen.

Objekte können nicht in einen nicht verwandten Typ umgewandelt werden – d. h. in einen anderen Typ als den Typ einer ihrer Ober- oder Unterklassen. Der Versuch führt bereits bei der Kompilierung zu einem »nicht konvertierbaren Typenfehler«. Hier sehen Sie ein Beispiel für eine Umwandlung, die zu einem Compilerfehler aufgrund »nicht konvertierbarer Typen« führt:

```
Object o = new Object();
String s = (Integer) o; // Compilerfehler
```

Umwandlungen zwischen elementaren Typen und Referenztypen

Die automatische Umwandlung zwischen elementaren Typen und Referenztypen wird als *Autoboxing* respektive *Unboxing* bezeichnet. Mehr Informationen finden Sie in Kapitel 3.

Referenztypen an Methoden übergeben

Folgendes passiert, wenn ein Objekt als Variable an eine Methode übergeben wird:

- Es wird eine Kopie der Referenzvariablen übergeben, nicht das eigentliche Objekt.
- Der Aufrufer und die aufgerufenen Methoden halten identische Kopien der Referenz.
- Der Aufrufer sieht alle Änderungen, die die aufgerufene Methode am Objekt vornimmt. Dass Änderungen am ursprünglichen Objekt vorgenommen werden, kann verhindert werden, indem der aufgerufenen Methode eine Kopie des Objekts übergeben wird.
- Die aufgerufene Methode kann die Adresse des Objekts nicht ändern, aber sie kann den Inhalt des Objekts ändern.

Das folgende Beispiel illustriert die Übergabe von elementaren Typen und Referenztypen an Methoden und zeigt die Auswirkungen auf die Typen, wenn sie von der aufgerufenen Methode geändert werden:

```
void roomSetup() {
  // Übergabe einer Referenz
  Table table = new Table();
  table.setLength(72);
  // Length wird geändert
  modTableLength(table);

  // Übergabe eines elementaren Werts
  // Wert von chairs wird nicht geändert
  int chairs = 8;
  modChairCount(chairs);
}

void modTableLength(Table t) {
  t.setLength(36);
}

void modChairCount(int i) {
  i = 10;
}
```

Referenztypen vergleichen

Referenztypen können in Java verglichen werden. Vergleichsoperatoren und die `equals`-Methode können in Vergleichen eingesetzt werden.

Die Vergleichsoperatoren verwenden

Die Vergleichsoperatoren != und == werden genutzt, um Speicherorte zweier Objekte im Speicher zu vergleichen. Wenn die Speicheradressen der verglichenen Objekte gleich sind, werden die beiden Objekte als gleich betrachtet. Diese Operatoren werden nicht eingesetzt, um den Inhalt von zwei Objekten zu vergleichen.

Im folgenden Beispiel haben `guest1` und `guest2` die gleiche Speicheradresse, es wird also `"Gleicher Gast"` ausgegeben:

```
String guest1 = new String("name");
String guest2 = guest1;
if (guest1 == guest2)
  System.out.println("Gleicher Gast");
```

Im nächsten Beispiel sind die Speicheradressen nicht gleich, es wird also `"Anderer Gast"` ausgegeben:

```
String guest1 = new String("name");
String guest2 = new String("name");
if (guest1 != guest2)
  System.out.println("Anderer Gast");
```

Die equals()-Methode verwenden

Zum Vergleichen des Inhalts von zwei Klassenobjekten kann die `equals()`-Methode der Klasse `Object` genutzt oder überschrieben werden. Wenn die `equals()`-Methode überschrieben wird, sollte ebenfalls die `hashCode()`-Methode überschrieben werden, und zwar wegen der Kompatibilität mit Hash-basierten Collections wie `HashMap()` und `HashSet()`.

Standardmäßig nutzt die `equals()`-Methode für Vergleiche einfach den ==-Operator. Diese Methode muss überschrieben werden, wenn sie wirklich nützlich sein soll.

Möchten Sie die Werte vergleichen, die in zwei Instanzen der gleichen Klasse enthalten sind, sollten Sie eine selbst definierte `equals()`-Methode verwenden.

Strings vergleichen

In Java gibt es zwei Möglichkeiten, zu prüfen, ob zwei Strings gleich sind, doch die Definition von »gleich« ist dabei jeweils eine andere:

- Die `equals()`-Methode vergleicht zwei Strings zeichenweise, um ihre Gleichheit zu prüfen. Das ist nicht die Standardimplementierung von `equals()`, die die Klasse `Object` stellt, es ist eine überschreibende Implementierung, die die Klasse `String` stellt.
- Der ==-Operator prüft, ob zwei Objekte auf die gleiche Instanz eines Objekts verweisen.

Hier ist ein Programm, das zeigt, wie Strings mit der `equals()`-Methode und dem ==-Operator verglichen werden (weitere Informationen zur Auswertung von Strings finden Sie in Abschnitt »Stringliterale« auf Seite 34 in Kapitel 2):

```
class MyComparisons {

  // Einen String dem Pool hinzufügen.
  String first = "chairs";
  // Den String aus dem Pool verwenden.
  String second = "chairs";
  // Einen neuen String erstellen.
  String third = new String ("chairs");


  void myMethod() {

    /*
     * Obwohl viele das nicht erwarten würden,
     * wird das zu true ausgewertet. Überrascht?
     */
    if (first == second) {
      System.out.println("first == second");
    }

    // Das wird zu true ausgewertet.
    if (first.equals(second)) {
      System.out.println("first equals second");
    }
    // Das wird zu false ausgewertet.
    if (first == third) {
      System.out.println("first == third");
    }
```

```
    // Das wird zu true ausgewertet.
    if (first.equals(third)) {
      System.out.println("first equals third");
    }
  } // Ende myMethod()
} //Ende Klasse
```

Objekte der Klassen `StringBuffer` und `StringBuilder` sind veränderlich. Objekte der Klasse `String` sind unveränderlich.

Enumerationen vergleichen

`enum`-Werte können mit `==` oder mit der `equals()`-Methode verglichen werden. Beide liefern das gleiche Ergebnis. Bei Enumerationstypen wird der `==`-Operator häufiger verwendet.

Referenztypen kopieren

Wenn Referenztypen kopiert werden, wird entweder eine Kopie der Referenz auf das Objekt erstellt oder die tatsächliche Kopie des Objekts, wodurch ein neues Objekt entsteht. Zweiteres wird in Java als *Klonen* bezeichnet.

Die Referenz auf ein Objekt kopieren

Wenn eine Referenz auf ein Objekt kopiert wird, erhält man ein Objekt, auf das zwei Referenzen zeigen. Im folgenden Beispiel wird `closingSong` eine Referenz auf das Objekt zugewiesen, auf das `lastSong` zeigt. Alle Änderungen, die an `lastSong` vorgenommen werden, werden von `closingSong` gespiegelt (wie umgekehrt auch):

```
Song lastSong = new Song();
Song closingSong = lastSong;
```

Objekte klonen

Klonen führt zu einer neuen Kopie des Objekts, nicht bloß zu einer Kopie einer Referenz auf ein Objekt. Klonen steht Klassen stan-

dardmäßig nicht zur Verfügung. Beachten Sie, dass das Klonen in der Regel äußerst komplex ist, Sie sollten deswegen aus folgenden Gründen als Alternative einen Kopierkonstruktor erwägen:

- Wenn eine Klasse klonbar sein soll, muss sie das Interface `Cloneable` implementieren.
- Die geschützte Methode `clone()` ermöglicht es Objekten, sich selbst zu klonen.
- Soll ein Objekt ein anderes Objekt als sich selbst klonen, muss die `clone()`-Methode überschrieben und von dem Objekt, das geklont wird, öffentlich gemacht werden.
- Beim Klonen muss ein Cast verwendet werden, da `clone()` den Typ `object` liefert.
- Klonen kann eine `CloneNotSupportedException` auslösen.

Flaches und tiefes Klonen

Java kennt zwei Arten des Klonens: flaches und tiefes Klonen.

Beim flachen Klonen werden die elementaren Werte und die Referenzen in dem Objekt, das geklont wird, kopiert. Es werden keine Kopien der Objekte erstellt, auf die diese Referenzen zeigen.

Im folgenden Beispiel wird `leadingSong` der Wert in `length` zugewiesen, da das ein elementarer Typ ist. Zudem werden `leadingSong` die Referenzen in `title`, `artist` und `year` zugewiesen, weil das Referenztypen sind:

```
Class Song {
  String title;
  Artist artist;
  float length;
  Year year;
  void setData() {...}
}
Song firstSong = new Song();
try {
  // Mit Klonen ein Kopie erstellen.
  Song leadingSong = (Song)firstSong.clone();
} catch (CloneNotSupportedException cnse) {
  cnse.printStackTrace();
} // Ende
```

Beim tiefen Klonen erstellt das geklonte Objekt eine Kopie aller seiner Objektfelder und geht dabei rekursiv alle Objekte durch, die es referenziert. Methoden zum tiefen Klonen müssen vom Programmierer definiert werden, da die Java-API keine bereitstellt. Alternativen zum tiefen Klonen sind Serialisierung und Kopierkonstruktoren. (Kopierkonstruktoren werden der Serialisierung häufig vorgezogen.)

Speicherallozierung und die Garbage Collection von Referenztypen

Wird ein neues Objekt erstellt, wird Speicher alloziert. Wenn es keine Referenzen auf ein Objekt gibt, kann der von diesem Objekt beanspruchte Speicher während des Garbage-Collection-Vorgangs zurückgewonnen werden. Mehr Informationen zu diesem Thema finden Sie in Kapitel 11.