Java I – Vorlesung 6
Referenz-Datentypen

7.6.2004

Referenzen
this, super und null
Typkonvertierung von Referenztypen
Finale Methoden und Klassen

Datentypen in Java

- In Java gibt es zwei Arten von Datentypen:
  - primitive Datentypen
    (int, char, double, ...)
  - Referenz-Datentypen
    - » Klassen
    - » Interfaces
    - » Arrays
- Warum heißen Referenz-Datentypen so?

Variablen und Datentypen

- Warum hat eine Variable einen Datentyp?
- Variable entspricht einem Stück Speicher
- Datentyp gibt an, wie der Speicherbereich als Wert interpretiert wird:
  - int: lies 4 Bytes als Binärzahl mit Vorzeichen
  - double: lies 4 Bytes als IEEE-Zahl

Referenzen

- Bei einem Referenz-Datentyp liegt im Speicher der Variable nur eine Referenz.
- Referenzen zeigen auf Objekte oder Arrays irgendwo anders im Speicher.
- Ein Objekt kann von mehreren verschiedenen Referenzen referenziert werden.
- Objekt hat seine eigenen Methoden. Datentyp der Variable gibt an, welche zur Compilezeit verfügbar sind.

Referenzen: Ein Beispiel

```
int a = 3;
int b = a;

b = 4;
System.out.println(a);  // -> 3
```


Referenzen: Ein Beispiel

```
class MutableInteger {
  private int x;

  public void get() { return x; }
  public void set(int val) { x = val; }
}

// ...

MutableInteger a = new MutableInteger();
MutableInteger b = a;

a.set(3);
b.set(4);
System.out.println(a.get());  // -> 4
```

Zuweisungen und Methodenaufrufe

- Bei Zuweisungen und Methodenaufrufen wird nicht das Objekt kopiert, sondern nur die Referenz!
- Danach referieren beide Variablen auf das gleiche Objekt im Speicher.
- new-Ausdrücke erzeugen neue Objekte. Ihr Wert ist eine Referenz auf das neue Objekt.

Methodenaufrufe

- Bei Methodenaufrufen heißt dieses Verhalten "call by reference".
- "call by value" (primitive Datentypen): Kopiere nur Wert, Änderungen bleiben lokal.

Call by Value

```
void setTo27(int x) {
  x = 27;
}

// ...

int zahl = 3;

setTo27(zahl);

System.out.println(zahl);  // -> 3
```


Call by Reference

```
void setTo27(MutableInteger x) {
  x.set(27);
}

// ...

MutableInteger zahl = new MutableInteger();

zahl.set(3);
setTo27(zahl);

System.out.println(zahl.get());  // -> 27
```

Warum Referenzen?

- Referenzen sind effizienter zu kopieren als das ganze Objekt.
- Einheitliche Handhabung von Objekten als Referenzen vereinfacht das Sprachdesign.
- In anderen Sprachen (C++) kann man Objekte by value und by reference übergeben.

this

- In jeder nichtstatischen Methode ist die Referenz "this" definiert. Sie zeigt auf das Objekt, für das die Methoden aufgerufen wurde.
- Methodenaufrufe und Feldselektionen ohne Punkt gehören implizit zu this.
- In statischen Methoden darf this nicht verwendet werden.

super

- In jeder nichtstatischen Methode ist eine Referenz super definiert. Sie referiert auf das aktuelle Objekt (genau wie this), aber als Objekt der Basisklasse.
- Aufrufe super.meth(...) sind in Wirklichkeit Aufrufe von meth(...) für die Referenz super.
- Konstruktor-Aufruf super(...) hat damit genau genommen nichts zu tun.
- In statistischen Methoden kein super.

Laufzeit- und Compilezeit-Typen

- Jeder Ausdruck hat einen Compilezeit-Typ, der von den Typen der Variablen bestimmt wird.
  - für statische Methodenaufrufe und Feldselektion
- Jeder Ausdruck von Referenztyp hat einen Laufzeit-Typ, der vom Objekt abhängt, auf das referiert wird.
  - für nicht-statische Methoden

instanceof

- Man kann den Laufzeit-Typ mit dem Operator instanceof herausfinden:
  ```
  expr instanceof reftype
  ```
- Ausdruck evaluiert zu `true`, wenn Laufzeit-Typ von `expr` spezieller als `reftype` ist.
- Typische Verwendung:
  - Klasse A hat abgeleitete Klassen B und C
  - Compilezeit-Typ von `expr` ist A
  - Ist Laufzeit-Typ von `expr` B oder C?

Typkonvertierung

- Man kann den Compilezeit-Typ eines Ausdrucks mit einer Typkonvertierung (Cast) verändern:
  (NeuerTyp) expr
- Es gibt erweiternde und verengende Typkonvertierungen.
- Für primitive Datentypen haben wir die Konvertierungen schon gesehen.

Erweiternde Konvertierung von Referenztypen

- Es gibt eine Hierarchie von Referenz-Datentypen:
  - Abgeleitete Klassen sind spezieller als Basisklassen
  - Implementierende Klassen sind spezieller als die implementierten Interfaces.
- Erweiternde Typkonvertierung (upcast) ist Konvertierung von speziellerem in allgemeineren Typ.
- Wird in Zuweisung, Aufruf automatisch gemacht.

Beispiel: Erweiternde Typkonvertierung

```
class A {
  void meth() { ... }

  static void test(Object x) {
    // ... x ist irgendein Object ...
    // x.meth() wäre hier illegal
  }
}

A var = new A();
Object varAsObject = var; // bei Zuweisung
A.test(var);              // bei Aufruf
A.test( (Object) var );   // geht auch
```

Verengende Typkonvertierung

- Verengende Typkonvertierungen (downcasts) zwischen Referenztypen: vor allem von allgemeineren zu spezielleren Typen (ex. andere Fälle).
- Wenn Konvertierung fehlschlägt, wird zur Laufzeit ClassCastException geworfen.
- Es empfiehlt sich, vor unsicheren Downcasts instanceof zu verwenden.

Beispiel: instanceof und Downcast

```
class A {
  void meth() { ... }

  static void test(A x) {
    x.meth();
    // ...
  }
}

Object var = new A();

A.test(var);            // geht nicht!

if( var instanceof A )
  A.test( (A) var );    // geht
```

null

- Es gibt speziellen Ausdruck `null`: Die "Null-Referenz".
- `null` ist vom "Null-Typ". Das ist ein Referenztyp, der erweiternd in jeden anderen Referentyp konvertiert werden kann.
- Wesentliche Verwendung: als Fehlerwert für Methoden, die Referenztypen zurückgeben.

Beispiel: null

```
Map<String,String> map =
  new HashMap<String,String>();

map.put("abcd", "efgh");

String x = map.get("foo");
if( x == null )
  System.out.println("foo nicht in map!");
```

clone

- Um doch eine Kopie eines Objekts anzulegen, verwendet man die Methode `clone()`:

```
objexpr.clone()
```

- Methode `clone()` darf nur für Klassen aufgerufen werden, die das (leere) Interface `Cloneable` implementieren.
- Standard-Implementierung kopiert jedes einzelne Feld (ggf. als Referenz) des Objekts: "shallow copy".

Überschreiben von clone

- Kann clone() überschreiben, muss aber (leider) immer Object zurückgeben!
- Eine eigene Implementierung der clone-Methode erzeugt ein neues Objekt (den Klon) und gibt Referenz auf dieses Objekt zurück.

Beispiel: clone

```
class MutableInteger implements Cloneable {
  // ...
}

MutableInteger zahl = new MutableInteger();
zahl.set(3);

MutableInteger z2 =
    (MutableInteger) zahl.clone();

System.out.println(z2.get());  // -> 3

setTo27(z2);
System.out.println(z2.get());  // -> 27
System.out.println(zahl.get());  // -> 3
```


equals

- Die Methode equals() stellt fest, ob zwei Objekte gleich sind:
  `obj1.equals(obj2)`
- Standard-Implementierung (aus Klasse Object) gibt true zurück, wenn obj1 und obj2 Referenzen auf gleiches Objekt sind.
- Kann equals-Methode in meinen eigenen Klassen überschreiben.
- JCF-Klassen (z.B. für Map) vergleichen Schlüssel mit ihren equals-Methoden.

hashCode

- Die Klassen HashMap und HashSet verwenden intern die Methode hashCode ihrer Schlüssel-Objekte.
- Es muss sichergestellt sein, dass Objekte, die bzgl. equals gleich sind, auch den gleichen hashCode haben.

final

- Manchmal will man verbieten, dass eine Methode überschrieben wird. Man kann das verbieten, indem man sie `final` deklariert.
- Wenn man eine Klasse `final` deklariert, kann nicht von ihr abgeleitet werden. Insbesondere wird nie eine Methode überschrieben.
- Zweck:
  - andere Programmierer können mir nicht "in die Karten schauen"
  - Effizienzgründe

Beispiel: final

```
class MutableInteger {
  // ...
  final void set(int x) { ... }
}

class A extends MutableInteger {
  // ...
  void set(int x) { ... }   // verboten!

  void set(double x) { ... }
  // ok, da nicht überschrieben
}
```


Beispiel: final

```
final class MutableInteger {
  // ...
}

class A extends MutableInteger {  // verboten!
  // ...
}
```

Finale Variablen

- Als Modifikator bei einer Variablendeklaration bedeutet `final`, dass dieser Variable nur einmal ein Wert zugewiesen werden kann.
- Zuweisungs-Statement wird vom Compiler abgewiesen, wenn Variable an diesem Punkt nicht garantiert noch keinen Wert hat.
- Zweck:
  - Dokumentation von Variable als Konstante
  - Effizienzgründe

Beispiel: Finale Variablen

```
int irgendwas() {
  final int einmal;

  einmal = 27;  // ok
  einmal = 28;  // jetzt verboten
}
```

Beispiel: Finale Variablen

```
int irgendwas(boolean flag) {
  final int einmal;

  if( flag )
    einmal = 27;

  einmal = 28;  // jetzt verboten
}
```


Beispiel: Finale Variablen als Konstanten

```
class EinigeKonstanten {
  public static final int EINS = 1;
  public static final int ZWEI = 2;
}
```

Zusammenfassung

- Arrays, Klassen und Interfaces sind Referenzdatentypen.
- Typkonvertierung
- Final: Ableitung verbieten