Prog1 CheatSheet #HSR HS 2012

Emanuel Duss (@mindfuckup, eduss@hsr.ch)

Java

Allgemein

Codierung: UTF16
Grundsätzlich nur Call-by-Value
Operatoren werden von links nach rechts abgearbeitet.

Reservierte Wörter

```
abstract, assert, boolean, break, byte,
case, catch, char, class, const, continue,
default, do, double, else, enum, extends,
final, finally, float, for, goto, if,
implements, import, instaceof, int,
interface, long, native, new, package,
private, protected, public, return, short,
static, strictfp, super, switch,
synchronized, this, throw, throws,
transistent, try, void, volatile, while
```

Operatoren (nach Priorität)

[]	Array-Index
()	Methodenaufruf
.	Komponentenzugriff
++ --	Inkrement / Dekrement
+ -	Vorzeichen
~	bitweises Komplement
!	Logische Negation
(type)	Typ-Umwandlung
new	Erzeugung
* / %	Multiplikation, Division, Rest
+ -	Addition, Subtraktion
+	Stringverkettung
<<	Linksshift
>>	Vorzeichenbehafteter Rechtsshift
>>>	Vorzeichenloser Rechtsshift
< <= > >=	Vergleich
instanceof	Typüberprüfung eines Objekts
==	Gleichheit
!=	Ungleichheit
& &&	bitweises / logisches UND
^	bitweises Exclusiv- ODER
||	bitweises/logisches ODER
? :	Bedingung
=	Wertzuweisung
*= /= += -=	Kombinierte Wertzuweisung (Auszug)

If-Else

```
if (a) { x = b; } else { x = c; }
x = a ? b : c
```

Switch Case

```
switch (i) {
  case 5:
    break;
  case 23:
    break;
  default:
    // alles andere }
```

Datentypen / Variabeln

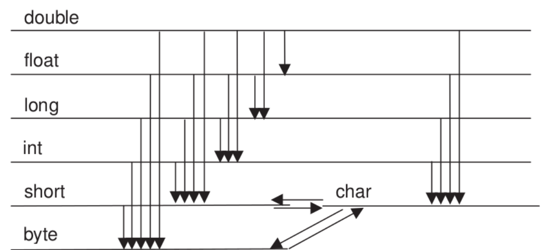
Einfache Datentypen und Default-Wert

boolean (true, false); FALSE
char (16 Bit, 0 bis 65535) ; '\u0000'
byte (8 Bit Ganzzahl, -2^7 bis +2^7 -1); 0
short (16 Bit Ganzzahl, -2^15 bis + 2^15 -1); 0
int (32 Bit Ganzzahl-2^31 bis +2^31 -1); 0
long (64 Bit Ganzzahl -2^63 bis +2^63 -1); 0
float (Gleitpunkt, -3.4E38 bis + 3.4E38); 0.0f
double (Gleitpunkt, -1.7E308 bis +1.7E308); 0.0d

Initialisierung

Methoden: Variablen immer initialisieren!
Klassen: Nicht unbedingt; Sonst Defaultwerte

Casting



Zu breiterem OK; zu schmalerem Casten.

Wrapper Klassen (int ⟶ Integer, ...)

```
String name = "Hagbard";
String vorname = new String("Celine");
```

Auto-Boxing: Compiler packt int in Iteger ein und aus.

Arrays (Index 0 bis i.length()-1;)

```
int[] i = new int[10]; i[0] = 23;
long[] l = {5,23}; l[1] = 42;
int[][] zd = new int[2][3]; zd[1][2] = 5;
int[][] zd = {{0}, {1,2}, {2,3,4}};
i.equals(l) // Nicht mit == vergleichen
```

Modifikatoren

	Variable	Methode	Konstr	Klasse	Interf
abstract		✓		✓	✓
final	✓	✓		✓	
private	✓	✓	✓		
protected	✓	✓	✓		
public	✓	✓	✓	✓	✓
static	✓	✓		✓	✓

final: Variable: Konstante; Methode: Nicht überschreibbar; Klasse: nicht vererbbar; Konstantenklasse: alles final
private: Nur in eigener Klasse sichtbar
protected: Nur in Pakage und Subklassen sichtbar
public: In allen Klassen sichtbar
(Default): Alle Klassen im selben Package
abstract: Methoden: kein Rumpf, MUSS von Subklasse implementiert werden; Klasse nicht instanzierbar
static: Nur einmal pro Klasse, nicht jedes Objekt

OOP: Objektorientierte Programmierung

Methoden

Überladen: Gleicher Name mit anderer Signatur.
Überschreiben: @override Methode der Oberklasse
Neu: private der Oberklasse neu definieren

Konstruktor

Methode mit selbem Namen wie Klasse. Zuerst wird der Konstruktor der Basisklasse aufgerufen. Konstruktor der eigenen Klasse: this(); Konstruktor der Oberklasse: super() (muss erster Aufruf im Konstruktor sein).

Klassenmethoden (static)

```
Klasse.drive(); Klasse.SPEED;
```

Vererbung

Vererbung: Subklasse kann nur von einer Klasse erben.

```
public class Car extends Vehicle { ... }
```

Polymorphie

Method Overriding (@override): Mehrere Varianten für gleichen Methodennamen.

```
Vehicle v = new Car();
```

Hat eine Referenz den statischen Typ der Superklasse (Vehicle), wird dennoch die überschriebene Methode des dynamischen Typs (Car) aufgerufen.

```
if (v instanceof Car) { ...}
```

Dynamische Typprüfung.

Interface

Nur Methodenköpfe; nicht instanzierbar

```
interface i1 { int E = 5; foo();}
interface i2 { int E = 23; bar();}
class Klasse implements i1, i2{
  void foo() { syso(i1.E); }
  void bar() { // MUSS impl. sein! }}
```

Methoden public abstract; Variabeln: public static final
Polymorphie: Objekte als Typ der Schnittstelle.
Interface kann Interface erben.

Rekursion

```
long factorial(int number) {
if (number <= 0) { // Abbruchbedingung
  return 1;
} else {
  return number * factorial(number - 1); }
}
```

Exceptions

Eigene Exception schreiben

```
class FnordException extends Exception {
  FnordException(String message) {
    super("Meine Exception " + message);}}
```

Exception werfen

```
String foo() throws FnordException {
throw new FnordException("Fuuuu"); }
```

Exception abfangen in public static void main:

```
try { ... }
catch(FnordException|FooException e) {
  e.getMessage(); }
catch(Execption e) { e.printStackTrace (); 
} final { // immer }
```

Collections

@override

HashSet/HashMap: equals() und hashcode()
TreeSet/TreeMap: equals() und compareTo()

LinkedList + Iterator (s/LinkedList/ArrayList/g)

```
List<String> ll = new LinkedList<>();
mylist.add("foo");
Iterator<String> it = ll.iterator();
while (it.hasNext()){ syso(it.next()); }
```

Methoden: add(„Foo“), get(0), set(23, „Bar“), size(), remove(„Bar“), contains(„Fuuuu“), add(6, „yay“); addLast(„Foo“), addFirst(„Foo“), removeFirst(), removeLast()

TreeSet (Keine Duplikate, sortiert)

```
Set<String> ts = new TreeSet<>();
ts.add("Foo"); ts.add("Bar");
ts.add("Bar"); // Wird nicht mehr eingef.
for (String s : treeSet){ syso(s);}
```

HashSet (Keine Duplikate, nicht sortiert)

```
Set<String> hs = new HashSet<>();
hs.add("Foo");hs.add("Bar");
hs.add("Bar");
for (String s : hashSet){ syso(s);}
```

TreeMap (Duplikate, sortiert)

```
Map<Integer, String> tm = new TreeMap<>();
tm.put(2, "Foo"); tm.put(1, "Bar");
tm.put(3, "Bar"); tm.put(3, "Bar");
for (Integer i : tm.keySet()) {
  syso(i + " ist " + tm.get(i));}
for (String s : tm.values()) { syso(s);}
```

HashMap (Duplikate, nicht sortiert)

```
Map<Integer, String> hm = new HashMap<>();
hm.put(2, "Foo"); hm.put(1, "Bar");
hm.put(3, "Bar"); hm.put(3, "Bar");
for (Integer i : hm.keySet()) {
  syso(i + " ist " + hm.get(i));}
```

@override equals()

Wenn equals() muss man auch hashCode() überschr.

```
@Override
public boolean equals(Object obj) {
if (this == obj) { return true; }
if (obj == null) { return false; }
if (!(obj instanceof Person)) {
  return false;}
```

```
Person other = (Person) obj;
if (name == null) {
  if (other.name != null) {return false;}}
else if (!name.equals(other.name))
{ return false; }
return true; }
```

@override hashcode()

```
@Override
public int hashCode() {
  final int prime = 31;
  int result = 1;
  result = prime * result + ((name ==
null) ? 0 : name.hashCode());
  result = prime * result + ((vorname ==
null) ? 0 : vorname.hashCode());
  return result; }
```

Comparable / Comparator

Comparable Interface (@override compareTo())

```
class Fruit implements Comparable<Fruit>{
  private String name;
  private Integer price;
  Fruit(String name, Integer price) {
    this.name = name;
    this.price = price; }
  @Override
  public int compareTo(Fruit f) {
  if (getName().equals(f.getName())) {
     return
getPrice().compareTo(f.getPrice());
  } else { return
getName().compareTo(f.getName());}}}
```

Comparator Interface (@override compare())

```
class FruitComparator implements
Comparator<Fruit> {
  @Override
  public int compare(Fruit f1, Fruit f2) {
  if (f1.getName().equals(f2.getName())) {
     return
f1.getPrice().compareTo(f2.getPrice());
    } else { return
f1.getName().compareTo(f2.getName()); }}}
```

Anwenden

```
Comparator<Fruit> c1 = new
FruitComparator();
Collections.sort(l1, c1); // Comparator
Collections.sort(l1);     // Comparable
```

Enummeratoren

```
enum Muenze {
  EINER (1, Muenzfarbe.KUPFER),
  FUENFER (5, Muenzfarbe.GOLD),
  FRANKEN (100, Muenzfarbe.SILBER);
  private int wert;
  private Muenzfarbe farbe;
  Muenze(int wert, Muenzfarbe farbe){
    this.wert = wert;
    this.farbe = farbe; }
  @Override
  public String toString() {
    return String.format("%s, Wert: %d,
Farbe: %s",
    name(), wert, farbe); }
```

Enum simple

```
enum Snafu { DISCORDIA, ERIS }
Snafu goettin = Snafu.ERIS;
```

Input/Output und Streams

Byte Streams: 8 Bit Data (InputStream, OutputStream)
Character Stream: 16 Bit Text (Reader/Writer)

Character Stream

```
try (FileReader reader = new
FileReader("quotes.txt")) {
int value = reader.read();
while (value >= 0) { // -1 = end of file
  char c = (char)value; // use character
  value = reader.read(); // 16-bit
char } }
```

Nach schreiben wieder lesen:

```
try (FileWriter writer = new
FileWriter("test.txt", true)) { // append
writer.write("Hello!"); // String schr.
writer.write('\n'); }// Char schreiben
```

Serialisieren von Objekten (Bsp. Klasse Foo)

8 Bit Data Byte Stream (InputStream, OutputStream)

```
class Foo implements Serializable {
  private static final long
serialVersionUID = 1L; // Versionsnummer
  List<String> list = new LinkedList<>();
  public void add(String name) { // ... }
```

Schreiben

```
@SuppressWarnings("resource")
public void save(String filename) {
try { OutputStream fos = new
FileOutputStream(filename);
  ObjectOutputStream stream = new
ObjectOutputStream(fos);
  stream.writeObject(list);
} catch (IOException e) { // ... } }
```

Lesen

```
@SuppressWarnings({ "unchecked",
"resource" })
public void load(String filename)  {
try { InputStream fis = new
FileInputStream(filename);
  ObjectInputStream stream = new
ObjectInputStream(fis);
  list = (List<String>)
stream.readObject();}
catch (IOException |ClassNotFoundException
e) { //... }}}
```

Lesen / Schreiben

```
Foo foo = new Foo();
foo.add("Foo"); foo.add("Bar");
foo.save("/tmp/foo.bin");
foo.load("/tmp/foo.bin");
```

Generics

```
class Foo<T> { private T name; ... };
class Bar<T, U>{ private T name; private U
einkommen; ... }
```

static Variable geht nicht; da nur einmal pro Klasse

Generischer Typ

```
Foo<String> = new Foo<>(); // Diamond
Bar<String, Integer> = new Bar<>();
```

Generische Methoden

```
class Test { public <T> T foo(T x, T y)
{ ... } ... }
this.<String>foo("ja", "nein")
```

Beispiel Generics

```
public class Fooo {
  public static void main(String[] args) {
    Frucht f1 = new Frucht("Manderine");
    Frucht f2 = new Frucht("Erdberefein");
    Fleisch f3 = new Fleisch("Schinken");
    Fleisch f4 = new Fleisch("Steak");
    Fach<Frucht> f5 = new Fach<>();
    f5.add(f1); }}
class Fach<T> {
  List<T> l1 = new LinkedList<>();
  public void add(T object) {
    l1.add(object); }}
class Frucht { String name;
  Frucht(String s) { name = s; }}
class Fleisch { String name;
  Fleisch(String s) { name = s; }}
```

Nested Classes

Wegen Sichtbarkeit, Gruppierung

Innere

```
Äussere a1 = new Äussere();
Äussere.Innere a2 = a1.new Innere();
```

Statische innere Klasse

```
Äussere.Innere foo = Äussere.Innere();
```

Lokale innere Klasse

Innerhalb Methode; Kann nur auf final Variablen der Äusseren zugreifen.

Anonyme innere Klasse

Direkt in Methodenkopf beim Aufruf

```
Collections.sort(personList, new
Comparator<Person>() {
public int compare(Person p1, Person p2) {
... my comparison logic ... } } );
```

Packages

Definition zuoberst im File

```
package foo.bar.fnord;
```

Verwenden der Klassen aus foo.bar (jedoch nicht aus foo.bar.fnord):

```
import foo.bar.*
```

UnitTesting

assertEquals(expected, actual)	actual eq. expected
assertSame(expected, actual)	actual == expected
assertNotSame(expected, actual)	expected != actual
assertTrue(condition)	condition
assertFalse(condition)	!condition
assertNull(value)	value == null
assertNotNull(value)	value != null
fail()	Immer verletzt

→ Oder mit Message (String) als erstes Argument.

Positive Tests (optional mit Timeout)

```
public class MultiSetTest {
private static final String ELEM_NAME =
"elem";
private static final int DEFAULT_TIMEOUT =
2000;
@Test(timeout = DEFAULT_TIMEOUT)
public void testSingleOccurrence() {
  MultiSet<String> set = new MultiSet<>();
  set.add(ELEM_NAME);
  assertTrue("contained",
set.contains(ELEM_NAME));
  assertEquals("set size", 1, set.size());
set.remove(ELEM_NAME);
  assertFalse("removed",
set.contains(ELEM_NAME));
  assertEquals("original size", 0,
set.size()); } }
```

Negative Tests

```
@Test(expected =
IllegalArgumentException.class)
public void testAddNullElementFails() {
MultiSet<Double> set = new MultiSet<>();
set.add(null); }
```

UML

+public; -private, static; #protected; ~package;
→zugehörig; <<interface>>; *AbstrakteKlasse*

Sonstiges

ASCII Tabelle ($ man 7 ascii)

```
   30  40  50  60  70  80  90  100  110  120
  ---------------------------------------
0:      (   2   <   F   P   Z   d    n    x
1:      )   3   =   G   Q   [   e    o    y
2:      *   4   >   H   R   \   f    p    z
3:  !   +   5   ?   I   S   ]   g    q    {
4:  "   ,   6   @   J   T   ^   h    r    |
5:  #   -   7   A   K   U   _   i    s    }
6:  $   .   8   B   L   V   `   j    t    ~
7:  %   /   9   C   M   W   a   k    u    DEL
8:  &   0   :   D   N   X   b   l    v
9:  '   1   ;   E   O   Y   c   m    w
```

Hex: 1 = 0x31; A = 0x41; a = 0x61

If-Schleifen

Vgl. → http://if-schleife.de

#Prüfung

Viel Glück an der Prog1 Prüfung && have fun 0x21 ;-)

Last Change

2013-01-17