Speicherdiagramm, Grundelemente 1

Variable mit einfachem Datentyp

Java:

```
double x = 56.32;
```

Speicherdiagramm:

x : double | 56.32

Referenz (Variable mit Klasse als Datentyp)

Java:

```
Konto k = null;
```

Speicherdiagramm:

Objekt

Java:

```
class Konto {
  int saldo = 100000;
  double zins = 0.25;
}
...
new Konto();
```

Speicherdiagramm:

Konto-Objekt

saldo : int | 100000

zins : double | 0.25

Array

Java:

```
new int[] = {10, 234};
```

Speicherdiagramm:

int-Array

[0] | 10

[1] | 234

```
new Konto[2];
```

Konto-Array

[0]

[1]

Speicherdiagramm, kombinierte Elemente

Referenz auf Objekt

Java:

```
Konto k = new Konto();
```

Speicherdiagramm:

Referenz auf Array

Java:

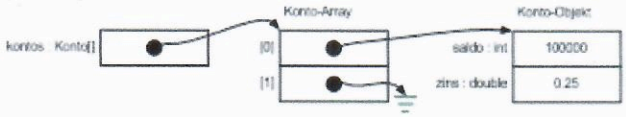
```
int[] i = {10, 234};
```

Speicherdiagramm:

Java:

```
Konto[] kontos = new Konto[];
kontos[0] = new Konto();
```

Speicherdiagramm:

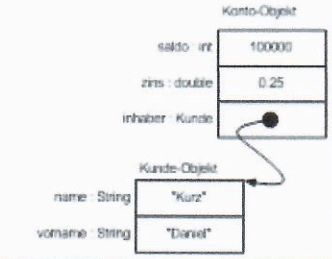


Referenz von einem Objekt auf ein anderes Objekt

Java:

```
class Konto {
  ...
  Kunde inhaber;
}

class Kunde {
  String name;
  String vorname;
}
...
new Konto(
  new Kunde("Kurz", "Daniel")
);
```

Speicherdiagramm:



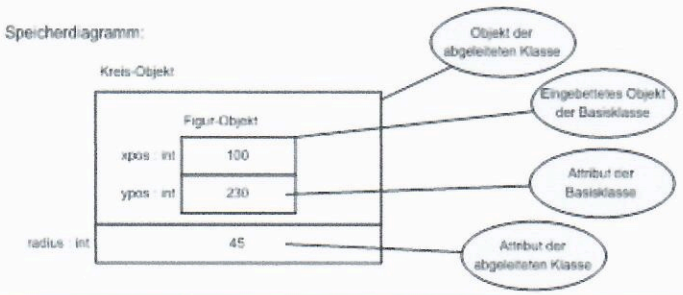
Speicherdiagramm, Grundelemente 2

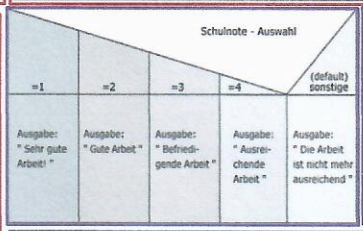
Objekt von abgeleiteter Klasse

Java:

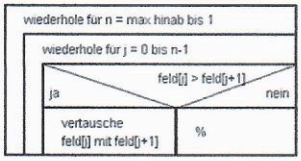
```
class Figur {
  int xpos;
  int ypos;
}

class Kreis extends Figur {
  int radius;
}
...
new Kreis(100, 230, 45);
```

Speicherdiagramm:





Bubble-Sort




```
class MultiDimArrayDemo {
    public static void main(String[] args) {
        String[][] names = {
            {"Mr. ", "Mrs. ", "Ms. "},
            {"Smith", "Jones"}
        };
        // Mr. Smith
        System.out.println(names[0][0] + names[1][0]);
        // Ms. Jones
        System.out.println(names[0][2] + names[1][1]);
    }
}
```

The output from this program is:

```
Mr. Smith
Ms. Jones
```

```
int[] meinArray = {1,2,3,4,5};
for(int i=0; i<meinArray.length; i++) {
        System.out.println(meinArray[i]);
}
```

Random ZUFALL = new Random();
int anzahlX = ZUFALL.nextInt(4)+1;
a.equals(b) verwenden
nicht ==

```
public class Konto {
    // Instanzvariablen
    private double saldo;
    private double zinssatz = 1.0;
    private Kunde inhaber;

    // Konstruktor
    public Konto(double zinssatz, Kunde inhaber) {
        this.zinssatz = zinssatz;
        this.inhaber = inhaber;
    }

    // Instanzklassen
    public void einzahlen(double betrag) {
        this.saldo += betrag;
    }

    public void verzinsen(int tage) {
        double zins = this.saldo * this.zinssatz * tage / 365;
        this.saldo += zins;
    }


    // Getter und Setter
    public double getSaldo() {
        return saldo;
    }
    public void setSaldo(double saldo) {
        this.saldo = saldo;
    }
```

Javabuch:
Variablen: 13, 26
Datentypen: 13
Operatoren: 17
Ausdrücke: 17
Arrays: 74, 147, 183
Klassen: 33,36
Objekte: 35, 38
Methoden: 36,40
Parameter: 40
Referenzen: 38, 60

```
    System.out.println("Der Kunde " + konto2.getInhaber().getName() + " hat ein Konto");

    konto2.einzahlen(720.50);
    konto2.verzinsen(750);

    System.out.println("Der Saldo des Kontos ist " + konto2.getSaldo() + " CHF");

    konto2.einzahlen(100);

    System.out.println("Der Saldo des Kontos ist " + konto2.getSaldo() + " CHF");

    // Aufgabe 6
    String a = new String("hallo");
    String b = new String("hallo");
    // Referenz (Ort) des Objektes vergleichen
    System.out.println(a==b);
    // Wert des Objektes vergleichen
    System.out.println(a.equals(b));
}
```

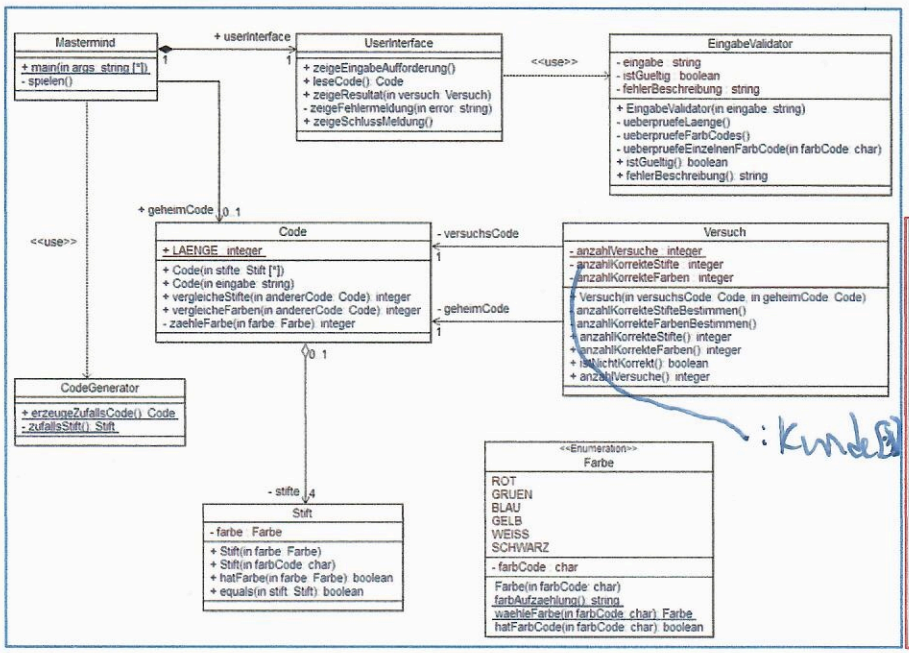
Array[0] = null;

```
Color black = new Color();
```

```
private ArrayList<Color> allColors = new ArrayList<Color>();
```

Array.length

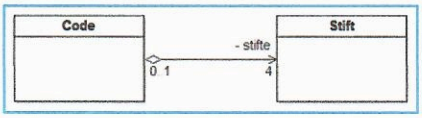
.add("1st Name(1half)").remove(id)

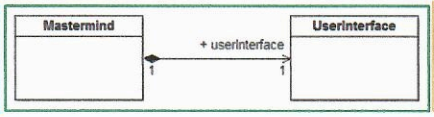


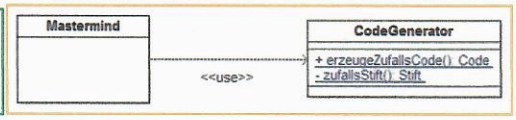
Sicherheit
– = private
+ = public
~ = package
# = protected

Mirio Eggmann

Java	UML
`class A { B myBObj; //Referenz auf B-Objekt } class B {}`	A — myBObj — B (0..1)
`class A { B[] myBObjs; //Referenz auf B-Array } class B {}`	A — myBObj — B (*)
`class A { Vector<B> myBObjs; //Vector mit B-Obj. } class B {}`	A — myBObj — B (*)
`class A { Vector<B> myBObjs; } class B { A myAObj; }`	A — myBObj (*) — B; myAObj (0..1); oder: A — myAObj (0..1) — myBObj (*) — B








```
static org.junit.Assert.*;

public class KontoTest {
    private Konto konto;
    private Kunde kunde = new Kunde("Eggmann", "Mirio", "12344556");
    private Konto konto2;

    @Before
    public void setUp() {
        konto = new Konto(1.0, kunde);
    }

    @Test
    public void testKonto() {
        assertNull(konto2);
        assertNotNull(konto);
        assertNotNull(konto.getInhaber());
        assertEquals(1.0, konto.getZinssatz(), 0);
    }
```

Delegation

Wir haben, ausgehend vom Code in Listing 1, durch mehrmaliges Hinzufügen von weiteren Klassen und Anwenden von Delegation eine klassenbasierte Lösung erhalten, welche das Single-Responsibilty-Principle respektiert und dadurch viel übersichtlicher, lesbarer und damit besser wartbar ist.

Letztlich geht es hier um die Qualität unserer Programme. Allerdings ist es nicht immer möglich, im ersten Anlauf gleich die beste Lösung zu finden. Deshalb muss man beim Schreiben von Programmen den bestehenden Code immer wieder überprüfen und bei Bedarf optimieren. Man spricht in diesem Zusammenhang auch von *Refactoring*.

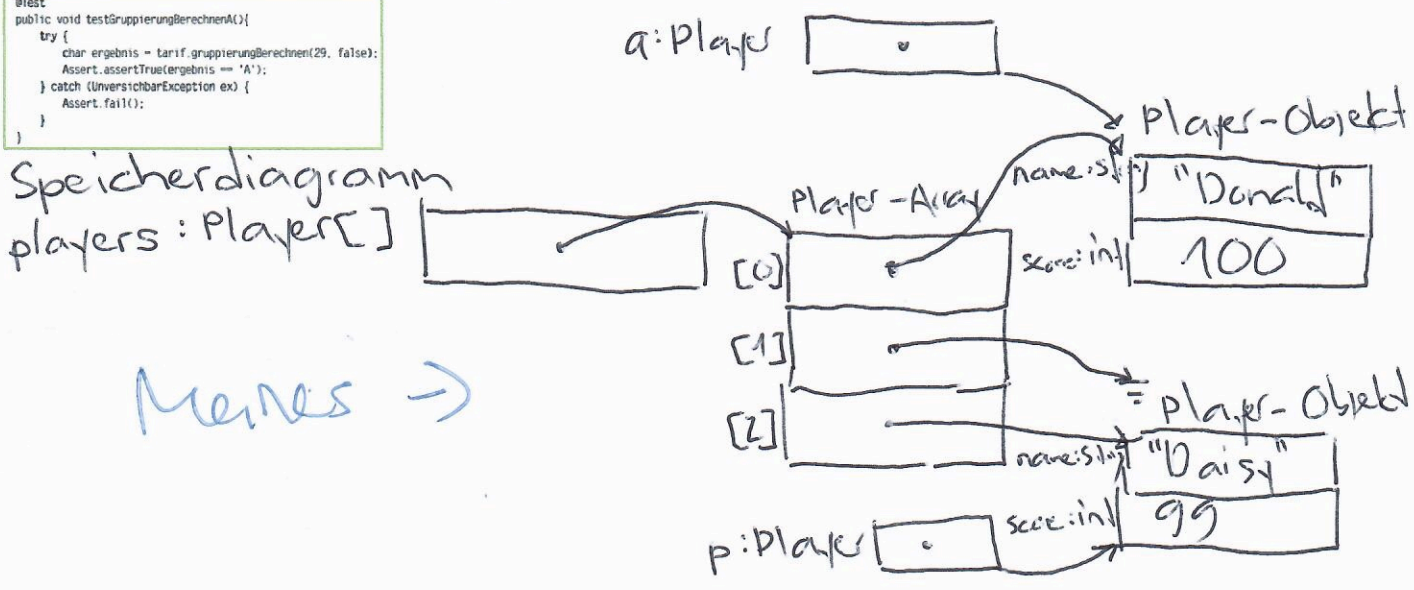
```
@Test
public void testEinzahlen() {
    assertEquals(1.0, konto.getZinssatz(), 0);
    assertEquals(0.0, konto.getSaldo(), 0);
    konto.einzahlen(500.54);
    assertEquals(500.54, konto.getSaldo(), 0);

    // Zum testen von weiteren asserts
    konto.einzahlen(100.0);
    assertTrue(600.54 == konto.getSaldo());

    assertFalse(0 == konto.getSaldo());

    konto2 = new Konto(2.0, kunde);

    assertSame(konto.getInhaber(), konto2.getInhaber());
    assertNotSame(konto, konto2);
}

    void testVerzinsen() {
    assertEquals(0.0, konto.getSaldo(),0);
    konto.einzahlen(500.75);
    assertEquals(500.75, konto.getSaldo(), 0);
    konto.verzinsen(365);
    assertEquals(1001.5, konto.getSaldo(), 0);
}

@After
public void waehreAuchNochMoeglich() {
}
```

Asserts

assertEquals(Object exp, Object act)
(float exp, float act, float delta)
assertFalse(boolean condition)
assertNotNull(Object object)
assertNotSame(Object unexp, Object act)
assertNull(Object object)
assertSame(Object exp, Object act) → selbe Obj referenz
assertTrue(boolean condition)
fail(String message)

```
@Test
public void testGruppierungBerechnenA(){
    try {
        char ergebnis = tarif.gruppierungBerechnen(29, false);
        Assert.assertTrue(ergebnis == 'A');
    } catch (UnversichbarException ex) {
        Assert.fail();
    }
}
```

Speicherdiagramm



Memes →