

## Grundlagen von Java


### Aufgabe 1: Verständnisfragen


Welche der folgenden Aussagen sind korrekt? Kreuzen Sie alle zutreffenden an. (Können Sie Ihre Antworten auch begründen?)


**Jein** a) Konstruktormethoden dienen der Erzeugung von Objekten. (Erzeugung durch new, Konstruktor = Initialisierung)

**JA** c) Jede Klasse besitzt mindestens einen Konstruktor.

**JA** e) Als private deklarierte Attribute können nur innerhalb der eigenen Klasse gelesen werden.

 b) Der Programmierer muss immer einen Konstruktor erstellen.

 d) Nur Konstruktormethoden können überladen werden.

 f) In als public deklarierten Methoden darf nicht auf die Attribute zugegriffen werden.

### Aufgabe 2: Fehler

Klaus Schussel hat im folgenden Programm eine Unmenge an Fehlern eingebaut. Können Sie Ihm helfen, die Fehler zu finden und zu korrigieren?

```
public class A {  
    public String msg;  
    private int len;  
  
    private A(String msg) {  
        len =  
            this.msg.length();  
        msg = msg;  
    }  
  
    public int f(int i) {  
        return i + len;  
    }  
    public String f(int i) {  
        return i + msg;  
    }  
}  
  
private void g(int i) {  
    int value;  
    if (i >= 0)  
        value = i;  
    len = value;  
    return result;  
}  
  
class B {  
    private static void  
    main(String[] args) {  
        A a = new A();  
        g(17);  
        String msg = a.f(0);  
    }  
}
```

```
public class A_Aufg2_Korrigiert {  
    public String msg;  
    private int len;
```

```
// private A_Aufg2_Korrigiert(String msg) {  
    // Konstruktor i.a. public (ausser bei Singleton-Pattern)  
    public A_Aufg2_Korrigiert(String msg) {  
        //len = this.msg.length(); // erst NACH Zuweisung an this.msg,  
        // ODER this weglassen, Sonst:  
        // Laufzeitfehler: NullPointerException  
        this.msg = msg; // statt: msg = msg; (Sinnlose Anweisung)  
        len = this.msg.length(); // Alternativ: len = msg.length();  
    }  
  
    public int f(int i) {  
        return i + len;  
    }  
    // public String f(int i) { // Compilerfehler: 2 x Funktion f mit  
    // Argument int  
    public String f(String i) {  
        return i + msg; // wäre VOR Korrektur von Parametertyp i  
    } // Compilerfehler: int + String undefiniert  
  
    // private void g(int i) { // Fehler 1: Rückgabotyp void, aber return  
    // mit Rückgabewert  
    // Fehler 2: Methode g soll in Klasse B  
    // verwendet werden --> nicht private  
  
    public int g(int i) {  
        int value;  
        if (i >= 0)  
            value = i;  
        else value = 0; // bugfix: setze Default-Wert, damit lokale  
        // Variable value stets initialisiert  
        len = value; // Compilerfehler: Variable value evtl.  
        // un-initialisiert  
    } // Compilerfehler: Variable result  
    // unbekannt --> sollte value sein  
    return result;  
    return value;  
    }  
}
```

// Klasse B sinnvollerweise in eigene Datei "B.java" packen  
// (sonst Programm, d.h. main-Methode nicht direkt aufrufbar (aus Eclipse))  
class B {  
 // private static void main(String[] args) { // main-Methode public,  
 // damit als Programm startbar  
 public static void main(String[] args) {  
 // A\_Aufg2\_Korrigiert a = new A\_Aufg2\_Korrigiert(); // new A();  
 // Compilerfehler: Default-Konstruktor in Klasse A nicht  
 // vorhanden  
 // --> Lösung: Default-Konstruktor hinzufügen (manuell)  
 // oder Parameter übergeben  
 A\_Aufg2\_Korrigiert a = new A\_Aufg2\_Korrigiert("Hallo");  
 // eine mögliche Korrektur  
 // g(17); // == Aufruf: this.g(17);  
 // --> Compilerfehler: Methode g gibt es nicht in Klasse B  
 a.g(17); // Methode g aus Klasse A (für Objekt a) aufrufen  
 // String msg = a.f(0); // Typfehler nach Korrektur Klasse A:  
 // f(int) liefert int, f(String) liefert String  
 String msg = "" + a.f(0); // Rückgabotyp int in String umwandeln,  
 // ALTERNATIV: msg = a.f("0"); (String  
 // übergeben --> andere Methode  
 // f(String) aufrufen statt f(int) !

```
}  
}
```

### Aufgabe 3: Java Klassendefinition

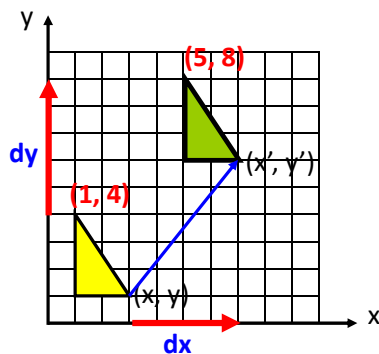
Für Spiele und andere grafische Anwendungen müssen wir Objekte auf der Karte positionieren und ggf. bewegen können. Dazu ist u.a. deren Position erforderlich.

**Teil 1.** Erstellen Sie eine Klasse **Point**, welche die Position bzw. Koordinaten im zweidimensionalen Raum realisiert.

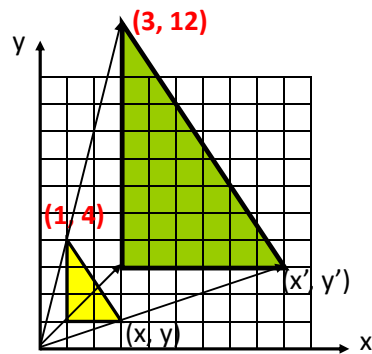
Als **Attribute** sind die jeweiligen **x**- bzw. **y**-Koordinaten zu speichern, Dabei sollen die internen Werte der Koordinaten von außen nicht direkt zugreifbar sein.

Die Klasse Position soll ferner die folgenden **Methoden** bereitstellen:

- **Konstruktormethoden** zum Setzen der Koordinaten. Dabei soll auch ein Default-Konstruktor mit dem Koordinatenursprung (0,0) angeboten werden.
- **get-Methoden** zum Lesen der jeweiligen x- und y-Koordinaten.
- **public String toString()** soll eine textuelle Repräsentation des Objektes zurückliefern.
- **void move(Position shift)** soll die Koordinaten entsprechend der anderen Position shift verschieben:  $\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} x + dx \\ y + dy \end{pmatrix}$



- **void scale(int factor)** skaliert die Koordinaten:  $\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \alpha \bullet x \\ \alpha \bullet y \end{pmatrix}$



- **Point moveTo(Point newPos)** soll die Koordinaten zum anderen Punkt newPos verschieben:  $\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} newPosX \\ newPosY \end{pmatrix}$
- **double distance()** liefert den Abstand  $\sqrt{x^2 + y^2}$  des Punktes vom Koordinatenursprung (0,0).
- **double distance(Position other)** liefert den Abstand der beiden Positionen voneinander.
- **boolean equals(Position other)** zum Vergleich zweier Objekte.

**Teil 2:** Testen Sie Ihr Programm.

Was würde beispielsweise der folgende Test ergeben?

```
Position p = new Position(1, 4);
p.scale(2);
p.move( new Position(2, -5) );
Position q = new Position();
System.out.println("p = " + p);
System.out.println("q = " + q);
System.out.println("Distance = " + p.distance(q) );
System.out.println("p.equals(q) = " + p.equals(q) );
System.out.println("p.equals(null) = " + p.equals(null) );
```

---

**Lösung (Ausgabe):**

```
p = (4.0, 3.0)
q = (0.0, 0.0)
Distance = 5.0
p.equals(q) = false
p.equals(null) = false
```

---

**Lösung: Point.java**

```
/**
 * Klasse Point in Java.
 * Sie realisiert Punkte (d.h. Positionen bzw. Koordinaten) im
 * zweidimensionalen Raum.
 *
 * <p>
 * Für Spiele und andere grafische Anwendungen müssen wir Objekte auf der
 * Karte positionieren und ggf. bewegen können. Dazu ist u.a. deren
 * Position erforderlich.
 *
 * <p>
 * Diese Klasse dient dem Einstieg in Java - wir werden später diese Klasse
 * noch modifizieren und anpassen.
 *
 * <p>
 * Die Kommentare sind als <b>javadoc<b>-Kommentare ausgeführt:
 * <br>
 *         javadoc Point.java
 * <br>
 * erzeugt eine HTML-Seite mit der Beschreibung der Methoden
 * (Schnittstellenbeschreibung wie die Java API, siehe unter
 * http://docs.oracle.com/javase/6/docs/api/index.html)
 *
 *
 * @author Thomas Nitsche
 * @version 1.0
 */
public class Point {

    // Attribute -----
    /**
     * Attribute:
     * x und y enthalten die Koordinaten des Punktes.
     *
     * <p>
     * Wegen der Deklaration als private sind die Attribute von außen
     * nicht direkt zugreifbar.
     * Sie können von außen nur über die getX() und getY()-Methoden
     * gelesen werden und durch move() geändert werden.
     */
    private double x, y;

    // Konstruktoren -----
    /**
     * Konstruktor setzt die Koordinaten auf (x, y)
     *
     * @param x        die x-Koordinate
     * @param y        die y-Koordinate
     */
    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }

    /**
     * Konstruktor setzt die Position auf den Nullpunkt (0, 0)
     */
    public Point () {
```

```
        this(0, 0);          // Aufruf Konstruktor: Point(0, 0);
        /* Alternative:
        this.x = 0;
        this.y = 0;
        */
    }

// get-Methoden -----
/**
 * get-Methode getX() liefert die x-Koordinate.
 *
 * @return x-Koordinate des Punktes.
 */
public double getX() {
    return this.x;
}
/**
 * get-Methode getY() liefert die y-Koordinate.
 *
 * @return y-Koordinate des Punktes.
 */
public double getY() {
    return this.y;
}

// String-Darstellung -----
/**
 * Liefert eine textuelle Darstellung des Punktes (als String).
 *
 * <p>
 * Die Methode toString() ist in Java bereits vordefiniert.
 * Da jedoch die Default-Implementierung lediglich den Klassennamen
 * und die Adresse des Objektes zurückliefert (und damit nicht sehr
 * nützlich ist), wird im Regelfall die toString-Methode in jeder
 * selbst definierten Klassen überschrieben, um eine sinnvolle
 * textuelle Darstellung der Objekte zu liefern.
 *
 * @return String-Darstellung des Objektes.
 */
@Override public String toString() {
    return "(" + this.getX() + ", " + this.getY() + ")";
}

// Funktionen -----
/**
 * move verschiebt die Koordinaten entsprechend der anderen Position
 * shift.
 *
 * <p>
 * Es gilt für die geänderte Position:
 * <br>
 *  $x' = x + \text{shift.x}$ 
 * <br>
 *  $y' = y + \text{shift.y}$ 
 *
 * <p>
 * Falls jemand mit der verschoben Position weiterarbeiten will,
 * geben wir dieses Objekt zurück.
 *
 * @param shift    der Verschiebe-Vektor

```

```
* @return gibt das eigene (modifizierte) Objekt nach Verschiebung
* zurück.
*/
public Position move(Point shift) {
    if (shift != null) {
        this.x = this.x + shift.x; // ändere das x-Attribut
                                   // vom eigenen Objekt
        this.y += shift.y;         // so geht's auch (mit +=)
    }
    return this; // wir geben das eigene modifizierte Objekt zurück
}
```

```
public Position moveTo(Point newPos) {
    if (newPos != null) {
        this.x = newPos.x; // ändere das x-Attribut
                           // vom eigenen Objekt
        this.y = newPos.y;
    } else {
        this.x = 0;
        this.y = 0;
    }
    return this; // wir geben das eigene modifizierte Objekt zurück
}
```

```
/**
 * scale skaliert die Koordinaten.
 *
 * <p>
 * Es gilt für die geänderte Position:
 * <br>
 *  $x' = \text{factor} * x$ 
 * <br>
 *  $y' = \text{factor} * y$ 
 *
 * <p>
 * Falls jemand mit der skalierten Position weiterarbeiten will,
 * geben wir dieses Objekt zurück.
 *
 * @param factor der Skalierungs-Faktor
 * @return gibt das eigene (modifizierte) Objekt nach Skalierung
 * zurück.
 */
public Point scale(double factor) {
    this.x = factor * this.x;
    this.y *= factor;

    return this; // wir geben das eigene modifizierte Objekt zurück
}
```

```
/**
 * distance liefert den Abstand des Punktes vom Koordinatenursprung
 * (0, 0).
 *
 * Ergebnis:
 * <br>
 *  $\text{Quadratwurzel}(x^2 + y^2)$ 
 *
 * @return Abstand vom Punkt (0, 0)
 */
```

```
public double distance() {
    return Math.sqrt(this.x * this.x + this.y * this.y);
    //alternativ:
    //return Math.sqrt(Math.pow(this.x, 2) + Math.pow(this.y, 2));
    // oder: x^2 + y^2
}

/**
 * distance - liefert den Abstand des Punktes vom einem anderen
 * Punkt, d.h. den Abstand von "this - other".
 *
 * @param other      Vergleichspunkt
 * @return Abstand vom anderen Punkt
 */
public double distance(Point other) {
    Point diff = new Point(this.x - other.x,
                           this.y - other.y);

    // Alternative:
    // (dafür muss jedoch die Klasse Point das Interface
    // "Cloneable" implementieren, d.h. schreibe
    // class Position implements Cloneable { ... }
    // try {
    //     // Kopiere (clone) other,
    //     // berechne dann "this - other"
    //     // mittels: (-1)*other + this --> scale(-1) und move(this)
    //     diff = ((Position) other.clone()).scale(-1).move(this);
    // } catch (CloneNotSupportedException e) {
    //     e.printStackTrace();
    // }

    // diff = this - other
    return diff.distance();

    //alternativ:
    //return Math.sqrt(Math.pow(this.x - other.x, 2)
    //                  + Math.pow(this.y - other.y, 2));
}

/**
 * equals - vergleicht zwei Position-Objekte miteinander.
 *
 * @param other      die Vergleichs-Position
 * @return true, wenn beide Positionen gleich sind, d.h. gleiche
 *         Koordinaten besitzen.
 */
public boolean equals(Point other) {
    // Prüfe zunächst, ob other überhaupt ein Objekt (!= null) ist
    if (other == null)
        return false;

    // jetzt ist other != null --> keine NullPointerException
    // möglich bei Zugriff auf other
    // Vergleiche nun die Attribute von other mit dem eigenen
    // Objekt (this)
    return (this.x == other.x && this.y == other.y);

    // Anmerkung: Besser mit epsilon-Umgebung wegen Rundungsfehlern
    // arbeiten:
    // double epsilon = 1e-6;
}
```



```
        //          return (Math.abs(this.x - other.x) < epsilon
        //          && ...);
    }

    /**
     * Hauptprogramm zum Testen der Methoden.
     *
     * @param args      Kommandozeilenparameter
     */
    public static void main(String[] args) {
        Point p = new Point(1, 4);    // --> p = (1, 4)
        p.scale(2);                    // --> p = (2, 8)
        p.move( new Point(2, -5) );   // --> p = (4, 3)
        Point q = new Point();        // --> q = (0, 0)
        System.out.println("p = " + p);
        System.out.println("q = " + q);
        System.out.println("Distance = " + p.distance(q) ); // --> 5
        System.out.println("p.equals(q) = " + p.equals(q) );
                                                // --> false
        System.out.println("p.equals(null) = " + p.equals(null) );
                                                // --> false
    }
}
```

---

## Zusatz-Aufgabe 4: Schiffe versenken

Falls Sie noch Zeit und Lust haben, können Sie Ihr SchiffeVersenken1D-Programm von Übung 1 um eine Klasse **Ship** zur Repräsentation von Schiffen (oder ähnlichen Spielobjekten) erweitern. Diese besitzt intern (als Attribute)

- einen Namen
- ihre momentane Position
- die Größe (d.h. Ausdehnung) in x- bzw. y-Richtung

sowie die folgenden Methoden:

- ein geeigneter **Konstruktor**
- **get-Methoden** (zum Lesen des Namens, der Position und Größe)
- eine **toString()**-Methode
- eine Testfunktion **boolean overlaps(Ship other)** zur Überprüfung, ob sich zwei Schiffe gegenseitig rammen würden, d.h. sich deren Positionen partiell überschneiden.

---

### Lösung: Ship.java

```
public class Ship {  
  
    // Attribute  
    private String name;  
    private int width, height;    // Größe: X x Y  
    private Position pos;        // nur int-Werte verwendet  
    // Verwendung der Positionen (für Schiffeversenken):  
    // Position (0,0) = links oben,  
    // belegte Positionen eines Schiffes:  
    // - X-Positionen: pos.x .. pos.x + width - 1  
    // - Y-Positionen: pos.y .. pos.y + width - 1  
  
    // Konstruktoren  
    public Ship(String name, int width, int height, Position pos) {  
        this.name = name;  
        this.width = width;  
        this.height = height;  
        this.pos = pos;  
    }  
    public Ship(String name, int width, int height, int posX, int posY) {  
        this(name, width, height, new Position(posX, posY));  
    }  
  
    // get-Methoden  
    public String getName() {  
        return this.name;  
    }  
  
    public Position getPosition() {  
        return this.pos;  
    }  
}
```

```
}

public int getWidth() {
    return this.width;
}

public int getHeight() {
    return this.height;
}

// String-Darstellung: Name(width x height, pos = (posX, posY))
@Override public String toString() {
    return this.getName()
        + "(" + this.getWidth() + " x " + this.getHeight()
        + ", pos = " + this.getPosition().toString() + ")";
    // liefert double-Werte!
    + ", pos = (" + this.getXPos() + ", " + this.getYPos()
    + ")";
}

// Test-Funktion
public boolean overlaps(Ship other) {
    if (other == null)
        return false;
    return isWithinOrTouchesShip(this, other)
        || isWithinOrTouchesShip(other, this);
}

// Hilfsfunktionen
private boolean isWithinOrTouchesShip(Ship s1, Ship s2) {
    return (s1.getLeftMostPos() <= s2.getXPos()
        && s2.getXPos() <= s1.getRightMostPos() + 1)
        && (s1.getUpperMostPos() <= s2.getYPos()
        && s2.getYPos() <= s1.getLowerMostPos() + 1);
    // Beachte: Berührung der Schiffe ist verboten,
    // deshalb rightMostPos + 1!
}

int getXPos() {
    return (int) this.getPosition().getX();
}

int getLeftMostPos() {
    return (int) this.getXPos();
}

int getRightMostPos() {
    return (int) this.getPosition().getX() + this.getWidth() - 1;
}

int getYPos() {
    return (int) this.getPosition().getY();
}

int getUpperMostPos() {
    return (int) this.getYPos();
}

int getLowerMostPos() {
    return (int) this.getPosition().getY() + this.getHeight() - 1;
}
}
```

**Lösung: ShipTest.java**

```
public class ShipTest {

    public static void main(String args[]) {
        // Schiffe(Nr.) mit Ihren Positionen/Ausdehnungen:
        //      |0123456789
        //      +-----+
        //      0|      3333
        //      1|      3333
        //      2|  1  33444
        //      3| 12   444
        //      4|  1

        Ship ship1 = new Ship("S1", 1, 3, new Position(2,2));
        Ship ship2 = new Ship("S2", 1, 1, 3,3);
        Ship ship3 = new Ship("S3", 4, 3, 5,0);
        Ship ship4 = new Ship("S4", 3, 2, 7,2);

        System.out.println("ship1 = " + ship1);
        System.out.println("Reflexivitätstest: ship1.overlaps(ship1) = "
            + ship1.overlaps(ship1));
        System.out.println("ship2 = " + ship2);
        System.out.println("ship1.overlaps(ship2) = "
            + ship1.overlaps(ship2));
        System.out.println("Symmetrietest: ship2.overlaps(ship1) = "
            + ship2.overlaps(ship1));
        System.out.println("ship3 = " + ship3);
        System.out.println("ship1.overlaps(ship3) = "
            + ship1.overlaps(ship3));
        System.out.println("ship2.overlaps(ship3) = "
            + ship2.overlaps(ship3));
        System.out.println("ship4 = " + ship4);
        System.out.println("ship1.overlaps(ship4) = "
            + ship1.overlaps(ship4));
        System.out.println("ship2.overlaps(ship4) = "
            + ship2.overlaps(ship4));
        System.out.println("ship3.overlaps(ship4) = "
            + ship3.overlaps(ship4));
        System.out.println("ship4.overlaps(ship3) = "
            + ship4.overlaps(ship3));
    }
}
```

**Ausgabe: ShipTest.java**

```
ship1 = S1(1 x 3, pos = (2, 2))
Reflexivitätstest: ship1.overlaps(ship1) = true
ship2 = S2(1 x 1, pos = (3, 3))
ship1.overlaps(ship2) = true
Symmetrietest: ship2.overlaps(ship1) = true
ship3 = S3(4 x 3, pos = (5, 0))
ship1.overlaps(ship3) = false
ship2.overlaps(ship3) = false
ship4 = S4(3 x 2, pos = (7, 2))
ship1.overlaps(ship4) = false
ship2.overlaps(ship4) = false
ship3.overlaps(ship4) = true
ship4.overlaps(ship3) = true
```

## Zusatz-Aufgabe 5 (für Java-Experten): Java-Rätsel – Strings/Konstrukturen

a) Rätsel (Strings): Was gibt das folgende Programm aus – und warum?

```
public class Lachen {  
    public static void main(String[] args) {  
        System.out.print("H" + "a");  
        System.out.print('H' + 'a');  
    }  
}
```

### Erwartete Ausgabe:

HaHa

### Tatsächliche Ausgabe:

Ha169

### Erklärung:

- Der Operator **+** ist definiert für Strings und für Zahlen, nicht jedoch für Zeichen (**char**).
- Folglich werden bei der zweiten Ausgabe im Ausdruck **'H' + 'a'** die Zeichen **'H'** und **'a'** in ein **int** (d.h. deren ASCII-Code) umgewandelt und anschließend addiert, d.h. **'H' + 'a' = 72 + 97 = 169**.
- Die Ausgabe ist folglich **Ha169**.
- Hinweis:
  - Um die Ausgabe **HaHa** Ausgabe zu erhalten, müssen folglich die Zeichen nach String umgewandelt werden. Dies kann wie folgt realisiert werden:

```
StringBuffer sb = new StringBuffer();  
Sb.append('H');  
Sb.append('H');  
System.out.print(sb);
```

- Einfacher ist es jedoch durch Konkatenation (+) mit einem Leerstring:

```
System.out.print("" + 'H' + 'a');
```

b) Rätsel (Strings): Was gibt das folgende Programm aus – und warum?

```
public class Abc {  
    public static void main(String[] args) {  
        String letters = "ABC";  
        char [] numbers = {'1', '2', '3'};  
        System.out.println(letters + " easy as " + numbers);  
    }  
}
```

**Erwartete Ausgabe:**

ABC easy as 123

**Tatsächliche Ausgabe:<sup>1</sup>**

ABC easy as [C@19f953d

**Erklärung:**

- `numbers` ist ein Array.
- Arrays sind in Java Objekte, und die Default-String-Repräsentation eines Objektes ist sein Klassenname, gefolgt von der Adresse des Objektes (bzw. "null" für null-Objekte), sofern die `toString`-Methode nicht überschrieben wird.
- Die korrekte Ausgabe wird erreicht durch Umwandlung des Arrays in einen String:

```
System.out.println(letters + " easy as " +  
                    String.valueOf(numbers));
```

- Alternativ können Sie auch durch zwei Aufrufe von `System.out.print` die überladene `println`-Funktion für `char[]`-Argumente verwenden:

```
System.out.print(letters + " easy as ");  
System.out.println(numbers);
```

---

<sup>1</sup> Bzw. eine ähnliche Ausgabe mit anderer Adresse.

- c) Rätsel (Konstruktor): Die folgende Klasse besitzt zwei überladene Konstruktoren. Die main-Methode ruft einen Konstruktor auf – aber welchen?  
Was gibt das Programm aus? Ist es überhaupt ein korrektes Java-Programm?

```
public class Confusing {  
    private Confusing(Object o) {  
        System.out.println("Object");  
    }  
  
    private Confusing(double[] dArray) {  
        System.out.println("double array");  
    }  
  
    public static void main(String[] args) {  
        new Confusing(null);  
    }  
}
```

**Ausgabe:**

double array

**Erklärung:**

- Java verwendet im Falle von überladenen Methoden stets diejenige Methode, die am speziellsten ist.
- Hier haben wir einen Konstruktor, der ein Object –Objekt als Argument bekommt, sowie einen Konstruktor, der ein Feld (double Array) als Argument besitzt.

- Da jedes Array auch ein Objekt ist, ist die Methode

Confusing(double[] dArray)

spezieller.

- Da null sowohl ein gültiges (leeres) Objekt als auch ein gültiges (leeres) Array repräsentieren kann, wird hier die spezielle Methode aufgerufen, d.h. der Konstruktor Confusing(double[] dArray) wird hier aufgerufen.