# 1. Aufgabe

Erstellen Sie eine Java-Klasse Bruch. Die zwei Attribute *zaehler* und *nenner* vom Typ *int* enthält. Es soll zwei Konstruktoren für die Klasse Bruch geben:

- ein Konstruktor mit einem Parameter, für den Zähler (der Nenner soll mit dem Wert 1 initialisiert werden).
- ein Konstruktor mit zwei Parametern für die beiden Attribute

Schreiben Sie für die beiden Attribute jeweils eine get-Methode; sowie eine Methode toString, die die Werte der beiden Attribute *zaehler* und *nenner* getrennt durch das Zeichen „/" zurückliefert:

- *public String toString()*.

Implementieren Sie außerdem noch die folgenden Methoden:

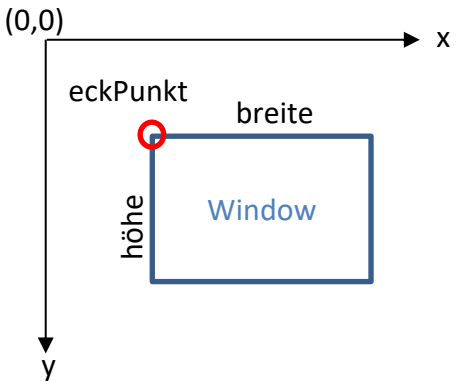
- *public Bruch multipliziere (int n)*: Diese Methode soll einen Bruch mit der ganzen Zahl n multiplizieren.
- *public Bruch multipliziere (Bruch b)*: Diese Methode soll zu einem Bruch einen zweiten Bruch multiplizieren.
- *public Bruch dividiere (int n)*: Diese Methode soll einen Bruch durch eine ganze Zahl dividieren.
- *public Bruch dividiere (Bruch b)*: Diese Methode soll einen Bruch durch einen zweiten Bruch dividieren.
- *public Bruch kuerze()*: der größte gemeinsame Teiler wird durch den euklidischen Algorithmus bestimmt, Zähler und Nenner entsprechend gekürzt. Den Algorithmus sollte man durch eigene Recherche finden.

Die Methoden sollen als Funktionen ausgelegt werden, die das Ergebnis der Methode durch Rückgabe eines Ergebnisobjekts (neuer Bruch) darstellen. Ausnahme: *kuerze()* – diese Funktion soll eine Referenz auf das Ausgangsobjekt zurück liefern. Die überladenen Methoden *multipliziere* und *dividiere* liefern ein ungekürztes Ergebnis.
Testen Sie Ihre Klasse über eine separate kleine Testklasse (mit *main*-Methode), die mehrere Objekte der Klasse Bruch erzeugt, die programmierten Methoden aufruft und die Ergebnisse (zusammen mit den jeweiligen Operanden) darstellt.

# 2. Aufgabe

Gegeben ist die folgende Klasse Window mit Attributen und Methoden in UML-Notation:

| **Window** |
| --- |
| `eckPunkt: Punkt`<br>`breite: double`<br>`höhe: double` |
| `vergrößern(int faktor)`<br>`setBreite(double w)`<br>`setHöhe(double h)`<br>`addPunkt(double x, double y)`<br>`getLinkerObererPunkt()`<br>`getRechterUntererPunkt()` |

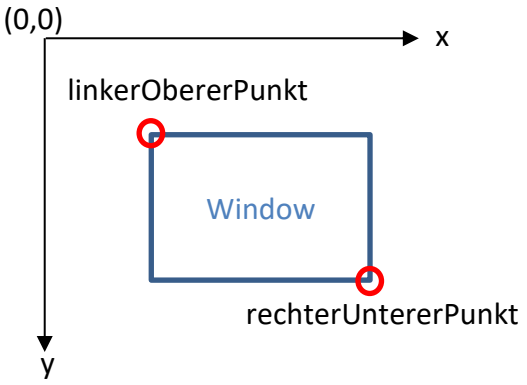



Das Attribut `eckPunkt` definiert hier den linkeren oberen Punkt des Windows. Mit den Methoden `setBreite`, `setHöhe` und `setPunkt` werden die Attribute auf die entsprechenden Werte gesetzt. Mit `vergrößern` kann das `Window` um einen entsprechenden Faktor vergrößert werden.

a) Vervollständigen Sie die Klassendefinition und implementieren Sie eine vollständige Java Klasse. Beachten Sie, dass zusätzlich eine Klasse Punkt notwendig ist. Schreiben Sie ein Testprogramm, das ein oder mehrere Window-Objekte (unterschiedlicher Koordinaten) erzeugt und geben Sie die Koordinaten der Eckpunkte (links oben und rechts unten) mit *System.out.printf* als formatierte Tabelle auf dem Bildschirm aus.

b) Im Rahmen der Weiterentwicklung der Software muss die Klasse wie folgt geändert werden:

| **Window** |
| --- |
| `linkerObererPunkt:Punkt`<br>`rechterUntererPunkt : Punkt` |
| `vergrößern(int faktor)`<br>`setBreite(double w)`<br>`setHöhe(double h)`<br>`addPunkt(double x, double y)`<br>`getLinkerObererPunkt()`<br>`getRechterUntererPunkt()` |




Passen Sie ihre Implementierung aus a) der neuen Definition von `Window` an.

c) Erklären Sie an diesem Beispiel die Datenkapselung und den damit verbundenen Vorteil für
- den Anwender (also das Testprogramm) der Klasse
- den Entwickler der Klasse

d) Im Rahmen weiterer Anpassungen wird die Klasse nochmals modifiziert:
- „setBreite“ und „setHöhe“ sollen abgekündigt werden. Statt dessen soll eine Methode „void addDim(double width, double hight)“ verwendet werden; das ursprüngliche Testprogramm muss aber noch laufen.
- „getLinkerObererPunkt“ und „getRechterUntererPunkt“ werden ebenfalls abgekündigt. Neu hinzu kommt dafür eine Funktion „eckPunkte“:
  `public Punkt[] eckPunkte();`
- Erstellen Sie dazu ein weiteres Testprogramm.

# 3. Aufgabe

Schreiben Sie eine Klasse „**Datum**" mit den Instanzvariablen **month, day** und **year:**

- Die Klasse soll folgende Datumsformate unterstützen:
  YYYY-MM-DD (ISO-Norm)
  14 Juni, 1996 (DIN-Norm)
  DD.MM.YYYY (optional)
- Benutzen Sie überladene Konstruktoren um **Datum** Objekte zu erzeugen, die mit den angegebenen Formaten initialisiert werden. **month, day** und **year** werden jeweils durch einen Parameter repräsentiert.
- Erstellen Sie eine Funktion *ausgabe(int variante)* die eine der drei Varianten als String zurückgibt. Beispielsweise soll das ISO-Norm Datum folgendermaßen abgerufen werden können: *ausgabe(Datum.ISO_Norm)*.
- Überprüfen Sie die Korrektheit des eingegebenen Datums. Setzen Sie bei Falscheingaben das heutige Datum.
- Schreiben Sie eine Klasse „**DatumTest**" mit einer **main**-Methode, um ihre Klasse zu testen.

# 4. Aufgabe

Entwerfen Sie eine Klasse „Fahrkartenautomat“ die folgenden Methoden breit stellen sollte:

- **Fahrpreis setzen (**Boolesche Funktion**)**: es kann ein beliebiger Fahrpreis gesetzt werden. Der Automat verfügt über folgende Münzarten: 2 €, 1 €, 50c, 20c, 10c, 5c. Es können also nur Preise gesetzt werden, die ein Vielfaches von 5c sind. Rückgabewert ist **true**, wenn der Fahrpreis automatenverträglich ist.
- **Geld eingeben (**Boolesche Funktion**)**: das Guthaben zum Lösen einer Fahrkarte kann in einem oder mehreren Schritten eingegeben werden. Rückgabewert ist **true**, wenn der Geldbetrag automatenverträglich ist.
- **Ausgabebereitschaft/Status** (Boolesche Funktion): Der Fahrkartenautomat ist ausgabebereit (Rückgabe: **true**), wenn das Guthaben >= dem Fahrpreis ist.
- **Stornieren**: es wird keine Fahrkarte gelöst. Das eingegebene Guthaben wird zurückgegeben.
- **Fahrkarte lösen**: wenn der Automat ausgabebereit ist, wird die Fahrkarte gelöst und eventuell Wechselgeld ausgegeben.
- **Eingabeprüfung (private)**: der Automat kann intern prüfen ob Geldbeträge automatenverträglich sind – zweistellig, durch 5c teilbar.

Erstellen Sie dazu eine Klasse *Fahrkartenkauf* zur Steuerung der Prozesse um den Fahrkartenautomaten. Diese Klasse soll also Dialogmöglichkeiten anbieten, um Fahrpreise zu setzen, Geld einzugeben, die Eingabe zu stornieren, den aktuellen Status anzuzeigen (Fahrpreis, eingegebenes Guthaben), die Fahrkarte zu lösen. Geldrückgaben sollen entsprechend der verfügbaren Münzarten durchgeführt werden. Möglicher Beispieldialog:

```
Fahrkarte lösen (f)
Geld eingeben   (g)
stornieren      (s)
Status anzeigen (a)
ende            (e)
Fahrpreis       (p)> p
Fahrkartenpreis> 6.35
Fahrkarte lösen (f)
Geld eingeben   (g)
stornieren      (s)
Status anzeigen (a)
ende            (e)
Fahrpreis       (p)> g
Einzahlung> 10
Fahrkarte lösen (f)
Geld eingeben   (g)
stornieren      (s)
Status anzeigen (a)
ende            (e)
Fahrpreis       (p)> f
*** Fahrkarte gelöst ***
   Wechselgeld: 1*2€; 1*1€; 1*50c; 1*10c; 1*5c
Fahrkarte lösen (f)
Geld eingeben   (g)
stornieren      (s)
Status anzeigen (a)
ende            (e)
Fahrpreis       (p)>
```

# 5. Aufgabe

Ein Roboter kann durch Kommandos gesteuert werden. Die Kommandos haben folgenden Aufbau:
*Objektbezeichnung Aktionsbezeichnung Parameter 1 Parameter 2*

- Objekte: Körper, Schulter, Arm, Hand, Finger
- Aktionen:
  - Rotieren (nur beim Körper und bei der Hand)
  - Beugen (nur bei der Schulter, dem Arm und der Hand)
  - Schließen (nur beim Finger)
- Parameter 1: Absolut oder Relativ (Indikator ob die Winkelveränderung absolut oder relativ sein soll).
- Parameter 2: Gradangabe (mit oder ohne Vorzeichen)

**Folgende Kommandos sind erlaubt:**

- KRA, KRR Grad: –135º bis +135º (absolut)
- SBA, SBR Grad: –45º bis +45º (absolut)
- ABA, ABR Grad: –45º bis +45º (absolut)
- HBA, HBR Grad: –45º bis +45º (absolut)
- HRA, HRR Grad: –90º bis +180º (absolut)
- FSA, FSR Grad: 0º bis +180º (absolut)

Der aktuelle Winkel der Objekte kann durch die Kommandos geändert werden, muss aber immer im jeweils vorgegebenen Bereich für das Rotieren, Beugen und Schließen bleiben.

**Aufgabe:**

- Schreiben Sie ein Java-Konsolprogramm.
- Das Programm prüft, ob eingegebene Kommandos der vorgegebenen Syntax entsprechen und die resultierenden Winkel im erlaubten Bereich liegen. Die erlaubten Winkelbereiche sind in der obigen Tabelle angegeben.
- Lesen Sie die Kommandoteile einzeln ein, so dass anschließend die folgenden Variablen initialisiert sind: Objekt, Aktion, Parameter1, Parameter2.
- Geben Sie bei einem gültigen Kommando als Ausgabe eine Kommandonummer aus, die der folgenden Zuordnung entspricht:
  - KRA = 1, KRR = 2, SBA = 3 bis FSR = 12.
- Geben Sie eine aussagekräftige Fehlermeldung aus, wenn die Kommandosyntax und/oder -semantik verletzt wurde.
- Es können beliebig viele Kommandos hintereinander eingegeben werden.

**Mögliches Dialogbeispiel:**

```
Befehl> ABR
Winkel> 37
KommandoNr: 6
Befehl> ABR
Winkel> 14
außerhalb des Winkelbereiches: von -45 bis 45, aktueller Winkel: 37
Befehl> ABC
Winkel> 123
C: falsche Positionierungsangabe

Befehl>
```

# 6. Aufgabe

Schreiben Sie ein Programm, das die Ausführungszeiten der Operation + für String-Objekte und der Methode append für StringBuffer-Objekte vergleicht. Es soll dabei ein zunächst leerer String bzw. String-Buffer in einer Schleife jeweils um ein oder mehrere Zeichen verlängert werden. Die gewünschte Anzahl der Schleifendurchgänge sollte eingelesen werden. Mögliches Dialogbeispiel:

```
Anzahl der Durchläufe> 10000
String-Operation +: 10000 Durchläufe in 121 Millisekunden
StringBuffer-Operation append: 10000 Durchläufe in 1 Millisekunden
```

# 7. Aufgabe

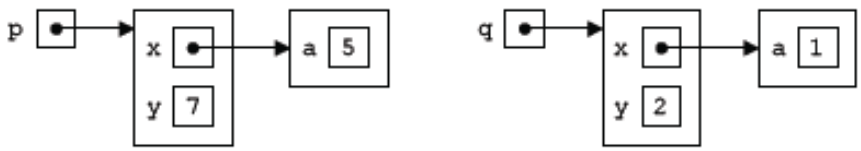
Geben sind folgende Klassen[1]:

```
public class IntKlasse {
  public int a;
  public IntKlasse (int a) {
    this.a = a;
  }
}
```

```
public class RefIntKlasse {
  public IntKlasse x;
  public double y;
  public RefIntKlasse(int u, int v) {
    x = new IntKlasse(u);
    y = v;
  }
}
```

```
public class KlassenTest {
  public static void copy1 (RefIntKlasse f, RefIntKlasse g) {
    g.x.a = f.x.a;
    g.y   = f.y;
  }
  public static void copy2 (RefIntKlasse f, RefIntKlasse g) {
    g.x = f.x;
    g.y = f.y;
  }
  public static void copy3 (RefIntKlasse f, RefIntKlasse g) {
    g = f;
  }
  public static void main (String args[]) {
    RefIntKlasse p = new RefIntKlasse(5,7);
    RefIntKlasse q = new RefIntKlasse(1,2); // Ergibt das Ausgangsbild
    // HIER FOLGT NUN EINE KOPIERAKTION:
         //***
  }
}
```

Der Start von Klassentest ergibt folgendes Ausgangsbild/Ergebnis (mit Referenzen und Werten):




Welche Bilder würden sich ergeben, wenn unmittelbar vor //*** einer der folgenden Aufrufe bzw. Anweisungen stehen würde:

- copy1(p, q);
- copy2(p, q);
- copy3(p, q);
- q = p;

Zeichnen Sie jeweils die Bilder der Referenzen und Werte, die sich für den entsprechenden Aufruf bzw. die entsprechende Anweisung ergeben würden.

[1] D. Ratz, J. Scheffler, D. Seese, J. Wiesenberger, Grundkurs Programmieren in Java 8, 7. Auflage, 2014

# Fragen zum Wiederholen/ Selbststudium/ Ausprobieren

## Klassen und Objekte

### 1. Frage

**Welche dieser Aussagen über Konstruktoren trifft zu?**

a) Gibt es in einer Klasse keinen Konstruktor, so wird automatisch ein Standardkonstruktor durch den Compiler erstellt.
b) Ein Konstruktor hat immer den Rückgabewert void.
c) Ein Konstruktor hat immer den Zugriffsbezeichner private.
d) Es darf nur einen Konstruktor geben.
e) Ein Konstruktor darf keinen Parameter haben.
f) Ein Konstruktor hat immer den Rückgabewert String.
g) Ein Konstruktor hat immer den Zugriffsbezeichner protected.

### 2. Frage

**Welche der untenstehenden Codefragmente beinhaltet eine korrekte Implementierung eines Konstruktors?**

a) public class Car{ Car void( ) { }; }
b) public class Car{ Car ( ) { }; }
c) public class Car{ car ( ) { }; }
d) public class Car{ Car (String anzahl) { }; }
e) public class Car{ Car String( ) { }; }
f) public class Car{ int Car ( ) { }; }

# Überladen von Methoden

## 1. Frage

**Bei welcher der überladenen Methoden kommt es zu einem Kompilierfehler?**

```
public class A{
      public long methodA(long a, int b) { return 0; }     Zeile 1
      public long methodA(long a, long b) { return 0; }    Zeile 2
      public long methodA(long a) { return 0; }            Zeile 3
      public long methodA(int a) { return 0; }             Zeile 4
      public long methodA( ) { return 0; }                 Zeile 5
}
```

a) Zeile 1
b) Zeile 2
c) Zeile 3
d) Zeile 4
e) Zeile 5
f) Keine dieser Möglichkeiten.

## 2. Frage

**Bei welcher der überladenen Methoden kommt es zu einem Kompilierfehler?**

```
public class A{
      public long methodA(long a, long b) { return 0; }    Zeile 1
      public int methodA(long a, long b) { return 0; }     Zeile 2
      public long methodA(long a, int b) { return 0; }     Zeile 3
      public long methodA(int a, long b) { return 0; }     Zeile 4
      public long methodA( ) { return 0; }                 Zeile 5
}
```

a) Zeile 1
b) Zeile 2
c) Zeile 3
d) Zeile 4
e) Zeile 5
f) Keine dieser Möglichkeiten.

## 3. Frage

**Bei welcher der überladenen Methoden kommt es zu einem Kompilierfehler?**

```
public class A{
      public int methodA(long a, int b) { return 0; }      Zeile 1
      public int methodA(long x, int y) { return 0; }      Zeile 2
      public int methodA(int b, long a) { return 0; }      Zeile 3
      public long methodA(int a) { return 0; }             Zeile 4
      public long methodA( ) { return 0; }                 Zeile 5
}
```

a) Zeile 1
b) Zeile 2
c) Zeile 3
d) Zeile 4
e) Zeile 5
f) Keine dieser Möglichkeiten.

## 4. Frage

**Welche der überladenen Methoden können an der gekennzeichneten Stelle eingefügt werden, ohne dass es zu einem Kompilierfehler kommt?**

```
public class A{
      public double methodA(double a) { return 0.0; }
      //Einfügen einer weiteren Methode
}
```

a) public long methodA(double a) { return 0; }
b) public double methodA(long a) { return 0.0; }
c) public double methodA(double a, double b) { return 0.0;}
d) public double methodA(float a) { return 0.0;}
e) Keine dieser Möglichkeiten.

# 5. Frage

**Welche der überladenen Methoden können an der gekennzeichneten Stelle eingefügt werden, ohne dass es zu einem Kompilierfehler kommt?**

```
public class A{
      public double methodA(double a, double b) { return 0.0;}
      //Einfügen einer weiteren Methode
}
```

a) public long methodA(double x, double y) { return 0; }
b) public double methodA(long a) { return 0.0; }
c) public int methodA(double a, double b) { return 0; }
d) public double methodA(double a) { return 0.0; }
e) Keine dieser Möglichkeiten.