Programmierkurs Java

Dr. Dietrich Boles

Aufgaben zu UE25-KlassenUndADT
(Stand 28.09.2012)

Aufgabe 1:

Sie haben in der Vorlesung die Klasse `Stack` (Stapel) kennen gelernt. Kennzeichen eines Stacks ist, dass dieser nach dem Prinzip „Last-In-First-Out“ arbeitet, d.h. zugegriffen werden kann immer nur auf das "jüngste" Element eines Stacks.

In dieser Aufgabe sollen Sie eine Klasse `Queue` (Warteschlange) implementieren, die nach dem Prinzip „First-In-First-Out“ arbeitet, d.h. zugegriffen werden kann immer nur auf das "älteste" Element einer Queue, d.h. das Element, das am längsten in der Queue abgespeichert ist.

Aufgaben: Implementieren Sie eine Klasse `Queue` mit folgenden Methoden:

- Ein Konstruktor, dem als Parameter die maximale Anzahl an zu speichernden Elementen mitgeteilt wird.
- Eine Methode `isEmpty`, die genau dann true liefert, wenn die Queue leer ist.
- Eine Methode `isFull`, die genau dann true liefert, wenn die Queue voll ist.
- Eine Methode `put`, die einen `int`-Wert in der Warteschlange speichert.
- Eine Methode `get`, die den `int`-Wert der Queue liefert (und ihn aus der Queue entfernt), der am längsten in der Queue gespeichert ist.

Aufgabe 2:

In dieser Aufgabe sollen Sie eine Klasse *PriorityQueue* (Prioritätswarteschlange) implementieren. Eine Prioritätswarteschlange ist eine Struktur, in der man Elemente abspeichern kann und von der jeweils das größte der abgespeicherten Elemente als nächstes geliefert und entfernt werden kann.

Aufgaben: Implementieren Sie eine Klasse *PriorityQueue* zum Abspeichern von int-Werten. Die Klasse soll folgende Methoden besitzen:

- Ein Konstruktor, dem als Parameter die maximale Anzahl an zu speichernden Elementen mitgeteilt wird.
- Eine Methode `isEmpty`, die genau dann true liefert, wenn die Queue leer ist.
- Eine Methode `isFull`, die genau dann true liefert, wenn die Queue voll ist.
- Eine Methode `insert`, die ein als Parameter übergebenes Element vom Typ int in der Warteschlange speichert.
- Eine Methode `remove`, die den aktuell größten int-Wert der PriorityQueue liefert und ihn aus der Queue entfernt.

Aufgabe 3:

Implementieren Sie in Java eine Klasse `Konto`, die ein Bankkonto realisiert. Ein Konto wird dabei repräsentiert durch einen Kontostand sowie einen eingeräumten Kreditrahmen. Die Klasse soll folgende Methoden zur Verfügung stellen:

1. Einen Konstruktor zum Initialisieren eines neuen (leeren) Kontos.
2. Einen Copy-Konstruktor zum Initialisieren eines Kontos mit einem bereits existierenden Konto.
3. Eine Methode zum Klonieren eines Konto-Objektes.
4. Eine Methode zum Überprüfen der Wertegleichheit zweier Konto-Objekte.
5. Eine Methode, die den aktuellen Kontostand als String-Objekt zurückliefert.
6. Eine Methode zum Einzahlen eines bestimmten Geldbetrages auf ein Konto. Dabei soll gelten: Wenn der Kontostand einmal den Wert von 10000 überschreitet, wird dem Konto im Folgenden ein Kreditrahmen von 3000 eingeräumt.
7. Eine Methode zum Abheben eines bestimmten Geldbetrages von einem Konto.
8. Eine Methode, die den aktuellen Kontostand als Wert liefert
9. Eine Methode zum Überweisen eines bestimmten Geldbetrages von einem Konto auf ein anderes

- Schreiben Sie weiterhin ein Programm zum Testen (= Aufruf aller Methoden) der Klasse.

Aufgabe 4:

Eine **Bitmenge** ist eine Datenstruktur, in der einzelne Bits gesetzt (1), andere nicht gesetzt sind (0). Beispielsweise repräsentiert die Bitfolge 10011000, dass die Bits 1, 4 und 5 gesetzt und alle anderen Bits nicht gesetzt sind.

Implementieren Sie in Java eine Klasse `BitSet`, welche eine Bitmenge als Abstrakten Datentyp realisiert. Die Länge der Bitfolgen soll dabei auf 8 festgesetzt sein, d.h. jedes Objekt der Klasse `BitSet` repräsentiert eine Bitfolge mit 8 Bits! Auf Objekten vom Datentyp `BitSet` sollen dabei folgende Funktionen ausführbar sein:

a) Initialisieren einer leeren Bitmenge, d.h. kein Bit ist gesetzt (Default-Konstruktor)

b) Clonieren einer Bitmenge

c) Konvertieren einer Bitmenge in ein String-Objekt (Bitfolge)

d) Überprüfen auf Wertegleichheit zweier Bitmengen (zwei Bitmengen sind wertegleich, wenn in beiden Mengen exakt dieselben Bits gesetzt sind)

e) Setzen eines einzelnen Bits der Bitmenge. Das zu setzende Bit wird dabei als `int`-Wert übergeben

f) Löschen eines einzelnen Bits der Bitmenge. Das zu löschende Bit wird dabei als `int`-Wert übergeben

g) Ausführung eines logischen `ANDs` (Konjunktion) mit einer anderen Bitmenge

h) Ausführung eines logischen `ORs` (Disjunktion) mit einer anderen Bitmenge

i) Ausführung eines logischen `NOTs` (Negation) auf einer Bitmenge

Schreiben Sie weiterhin ein Programm zum Testen

Aufgabe 5:

Implementieren Sie eine Klasse Cardinal, die das Rechnen mit Natürlichen Zahlen (1, 2, 3, ...) ermöglicht!

Die Klasse Cardinal soll folgende Methoden besitzen:

a) einen Konstruktor, der ein Cardinal-Objekt mit einem übergebenen int-Wert initialisiert

b) einen Copy-Konstruktor, der ein Cardinal-Objekt mit einem bereits existierenden Cardinal-Objekt initialisiert

c) eine Methode clone, die ein Cardinal-Objekt klont

d) eine Methode equals, die zwei Cardinal-Objekte auf Wertegleichheit überprüft

e) eine Methode toString, die eine String-Repräsentation des Cardinal-Objektes liefert

f) eine Klassenmethode sub, die zwei Cardinal-Objekte voneinander subtrahiert

g) eine Methode sub, die vom aufgerufenen Cardinal-Objekt ein übergebenes Cardinal-Objekt subtrahiert

Aufgabe 6:

Implementieren Sie eine Klasse RGBFarbe, die den Umgang mit Farben realisiert! Farben sollen dabei durch ihren sogenannten RGB-Wert repräsentiert werden, der die Farbanteile der Farbe bzgl. Rot, Grün und Blau angibt. Die Farbanteile werden durch Werte zwischen 0 (kein) und 255 (maximal) festgelegt.

Beispiele: Reines kräftiges Rot hat den RGB-Wert (255,0,0), Schwarz hat den RGB-Wert (0,0,0), Weiss hat den RGB-Wert (255,255,255) (Mischen aller Farben), kräftiges Gelb hat den RGB-Wert (255,255,0) (Mischen von Rot und Grün), ein mittelkräftiger Orange-Ton hat den Wert (255, 89, 10).

Die Klasse RGBFarbe soll folgende Methoden besitzen:

h) einen Konstruktor, der ein RGBFarbe-Objekt mit drei übergebenen int-Werten, die die Farbanteile für Rot, Grün und Blau repräsentieren, initialisiert

i) einen Copy-Konstruktor, der ein RGBFarbe-Objekt mit einem bereits existierenden RGBFarbe-Objekt initialisiert

j) eine Methode equals, die zwei RGBFarbe-Objekte auf Wertegleichheit überprüft

k) eine Methode mischen, die das aufgerufene RGBFarbe-Objekt mit einem übergebenen RGBFarbe-Objekt mischt. Realisieren Sie das Mischen durch das Bilden des Mittelwertes der jeweiligen Farbanteile der beiden Objekte

l) eine Klassenmethode mischen, die ein neues RGBFarbe-Objekt durch das Mischen zweier existierender RGBFarbe-Objekte erzeugt. Realisieren Sie das Mischen durch das Bilden des Mittelwertes der jeweiligen Farbanteile der beiden Objekte

Schreiben Sie ein kleines Testprogramm für die Klasse RGBFarbe, das die beiden Methoden mischen testet!

Aufgabe 7:

Implementieren Sie eine Klasse *StringTokenizer*. Die Klasse *StringTokenizer* erlaubt es, Strings in Token aufzubrechen, die durch spezielle Delimiter-Charakter voneinander getrennt sind.

Beispiel:

```
String           = „hurra,_ich_werde__die_Klausur_bestehen“
Delimiter-String = „_,“
Tokenfolge   = „hurra“ „ich“ „werde“ „die“ „Klausur“ „bestehen“
```

Noch ein Beispiel:

```
String           = „zu4dumm,7ich789bin6heute9 nicht4fit“
Delimiter-String = “0123456789“
Tokenfolge    = „zu“ „dumm,“ „ich“ „bin“ „heute“ „ nicht“ „fit“
```

Die Klasse StringTokenizer soll folgende Methoden besitzen:

1. einen Konstruktor, der ein StringTokenizer-Objekt mit einem String und einem Delimiter-String initialisiert
2. einen Copy-Konstruktor, der ein StringTokenizer-Objekt mit einem bereits existierenden StringTokenizer-Objekt initialisiert
3. eine Methode clone, die ein StringTokenizer-Objekt klont
4. eine Methode equals, die zwei StringTokenizer-Objekte vergleicht
5. eine Methode countToken, die die Anzahl an Token im String zurückliefert
6. eine Methode getAllToken, die alle Token des Strings in einem String-Array zurückliefert

Schreiben Sie ein geeignetes (!) Testprogramm für die Klasse StringTokenizer.

Sie können natürlich die Methoden der Klasse java.lang.String nutzen:

```
public class String {
  public String();
  public String(String str);
  public boolean equals(Object str);
  public char charAt(int index); // „dibo“.charAt(0) == ‘d‘
  public int indexOf(char ch);   // „dibo“.indexOf(‚i‘) == 1
                                 // „dibo“.indexOf(‚a‘) == -1
  public int indexOf(char ch, int fromIndex);
  public int indexOf(String str);
  public int indexOf(String str, int fromIndex);
  public int length();
  public String substring(int beginIndex, int endIndex);
                               // „dibo“.substring(0, 2) -> „di“
}
```

Aufgabe 8:

Implementieren Sie eine Klasse *BigCardinal*, die beliebig große natürliche Zahlen (also keine Einschränkung auf 32 oder 64 Bit) repräsentiert. Tipp: Speichern Sie die zu repräsentierende Zahl intern in einem String.

Die Klasse *BigCardinal* soll folgende Methoden besitzen:

1. einen Konstruktor, der ein BigCardinal-Objekt mit einem String initialisiert
2. einen Konstruktor, der ein BigCardinal-Objekt mit einem int-Wert initialisiert
3. einen Copy-Konstruktor, der ein BigCardinal-Objekt mit einem bereits existierenden BigCardinal-Objekt initialisiert
4. eine Methode *add*, die zu dem aufgerufenen BigCardinal-Objekt ein anderes als Parameter übergebenen BigCardinal-Objekt addiert
5. eine Methode *sub*, die von dem aufgerufenen BigCardinal-Objekt ein anderes als Parameter übergebenes BigCardinal-Objekt subtrahiert
6. eine Methode *compareTo*, die das aufgerufene BigCardinal-Objekt mit einem anderen als Parameter übergebenen BigCardinal-Objekt vergleicht und die Werte –1, 0 oder 1 liefert, je nachdem, ob der Wert des aufgerufenen BigCardinal-Objektes kleiner, gleich oder größer ist als der Wert des als Parameter übergebenen BigCardinal-Objektes
7. eine Methode *toString*, die die natürliche Zahl als String zurückliefert.

Schreiben Sie ein geeignetes Testprogramm für die Klasse *BigCardinal*!

Sie können die Methoden der Klasse `java.lang.String` nutzen, wenn Sie String-Objekte für die Realisierung der internen Datenstruktur wählen:

```
public class String {
  public String(String str);
  public boolean equals(Object str);
  public char charAt(int index); // „dibo".charAt(0) == 'd'
  public int length();
  public String substring(int beginIndex, int endIndex);
                              // „dibo".substring(0, 2) -> „di"
  public String concat(String str)
                             // „di".concat(„bo") -> „dibo"
}
```

Aufgabe 9:

Implementieren Sie in Java eine Klasse Gerade, welche den Umgang mit (geometrischen) Geraden realisiert (kartesisches Koordinatensystem). Auf Objekten vom Datentyp Gerade sollen dabei folgende Funktionen ausführbar sein:

a) Initialisieren einer Geraden mit Anfangs- und Endpunkt (Konstruktor)
b) Initialisieren einer Geraden mit einer bereits existierenden Geraden, d.h. anschließend sind die beiden Geraden wertegleich (siehe Teilaufgabe (e)) (Copy-Konstruktor)
c) Clonieren einer Geraden, d.h. initialisieren einer neuen Geraden mit einer bereits existierenden Geraden, d.h. anschließend sind die beiden Geraden wertegleich (siehe Teilaufgabe (e))
d) Konvertieren einer Geraden in ein String-Objekt (geeignete Darstellung wählen)
e) Überprüfen auf Wertegleichheit zweier Geraden (zwei Geraden sind wertegleich, wenn sie denselben Anfangs- und Endpunkt besitzen)
f) Addition zweier Geraden (nur möglich, wenn Endpunkt der ersten und Anfangspunkt der zweiten Geraden identisch sind)
g) Skalierung der Geraden mit einem bestimmten Faktor
h) Lieferung der Steigung der Geraden

Aufgabe 10:

Implementieren Sie in Java eine Klasse Roman, die den Umgang mit römischen Zahlen (mit Werten zwischen 1 und 3999) ermöglicht. Im (für diese Aufgabe definierten) römischen Zahlensystem steht das Symbol I für 1, V für 5, X für 10, L für 50, C für 100, D für 500 und M für 1000. Symbole ergeben hintereinander geschrieben die römische Zahl. Symbole mit größeren Werten stehen dabei

normalerweise vor Symbolen mit niedrigeren Werten. Der Wert einer römischen Zahl wird in diesem Fall berechnet durch die Summe der Werte der einzelnen Symbole. Falls ein Symbol mit niedrigerem Wert vor einem Symbol mit höherem Wert erscheint (es darf übrigens jeweils höchstens **ein** niedrigeres Symbol **einem** höheren Symbol vorangestellt werden), errechnet sich der Wert dieses Teils der römischen Zahl als die Differenz des höheren und des niedrigeren Wertes. Die Symbole I, X, C und M dürfen bis zu dreimal hintereinander stehen; die Symbole V, L und D kommen immer nur einzeln vor. Die Umrechnung von Dezimalzahlen in römische Zahlen ist übrigens nicht eineindeutig. So lässt sich z.B. die Dezimalzahl 1990 römisch darstellen als MCMXC bzw. MXM.

Beispiele:

```
 3999 = MMMCMXCIX
   48 = XLVIII
  764 = DCCLXIV
 1234 = MCCXXXIV
  581 = DLXXXI
```

Implementieren Sie folgende Methoden:

1. Konvertieren eines String-Objektes, das eine römische Zahl repräsentiert, in einen int-Wert. Sie können davon ausgehen, dass das String-Objekt eine gültige römische Zahl repräsentiert.
2. Konvertieren eines int-Wertes in einen String, der eine römische Zahl repräsentiert
3. Initialisieren einer römischen Zahl mit einem String, der eine römische Zahl repräsentiert (Konstruktor)
4. Initialisieren einer römischen Zahl mit einem int-Wert (Konstruktor)
5. Initialisieren einer römischen mit einer bereits existierenden römischen Zahl, d.h. anschließend sind die beiden römischen Zahlen wertegleich (Copy-Konstruktor)
6. Clonieren einer römischen Zahl, d.h. initialisieren einer neuen römischen Zahl mit einer bereits existierenden römischen Zahl, d.h. anschließend sind die beiden römischen Zahlen wertegleich
7. Umwandeln einer römischen Zahl in ein String-Objekt
8. Überprüfen auf Wertegleichheit zweier römischer Zahlen
9. Addition zweier römischer Zahlen

Aufgabe 11:

In einem Programm soll ein abstrakter Datentyp *Menge* verwendet werden. Dieser soll es ermöglichen, mit Mengen im mathematischen Sinne umzugehen. Eine Menge soll dabei eine beliebig große Anzahl von int-Werten aufnehmen können, jeden int-Wert aber maximal einmal.

Schreiben sie eine Klasse *Menge*, welche diesen ADT implementiert.

Auf dem Datentypen Menge sollen folgende Funktionen möglich sein:

a) Erzeugen einer neuen leeren Menge.

b) Erzeugen einer neuen Menge mit einer bereits existierenden (Copy-Konstruktor)

c) Überprüfen auf Gleichheit zweier Mengen

d) Hinzufügen eines int-Wertes zu einer Menge.

e) Entfernen eines int-Wertes aus einer Menge.

f) Überprüfung, ob ein bestimmter int-Wert in der Menge enthalten ist.

g) Schnittmengenbildung zweier Mengen.

h) Vereinigung zweier Mengen.

i) Differenzbildung zweier Mengen.

Schreiben Sie weiterhin ein kleines Testprogramm für die Klasse *Menge*.

Aufgabe 12:

Implementieren Sie eine ADT-Klasse Intervall. Diese soll den Umgang mit mathematischen Intervallen ermöglichen. Ein Intervall im Sinne dieser Aufgabe besteht dabei aus einer Untergrenze (double) und einer Obergrenze (double) und beinhaltet alle Zahlen zwischen der Untergrenze inklusive und der Obergrenze inklusive. Ist die Untergrenze größer als die Obergrenze ist das Intervall leer.

Die Klasse Intervall soll folgende Methoden bereitstellen:

- Einen Default-Konstruktor (erzeugt ein leeres Intervall)
- Einen Konstruktor, dem die Untergrenze und die Obergrenze des Intervalls als Parameter übergeben werden
- Einen Copy-Konstruktor
- Eine Methode clone zum Klonieren eines Intervalls (Überschreiben der Methode clone der Klasse Object)
- Eine Methode equals, mit der zwei Intervalle auf Gleichheit überprüft werden können. Zwei Intervalle sind dabei gleich, wenn sie beide leer sind oder dieselben Zahlen beinhalten.
- Eine Methode toString, die eine String-Repräsentation eines Intervalls liefert (Überschreiben der Methode toString der Klasse Object). Die String-Repräsentation soll dabei folgende Gestalt haben: „[]“ für ein leeres Intervall und „[u, o] “ für nicht leere Intervalle, wobei u die Unter- und o die Obergrenze darstellen.
- Eine boolesche Methode enthaelt, die überprüft, ob das aufgerufene Intervall ein als Parameter übergebenen Intervall komplett enthält. Dabei gilt: Ein leeres Intervall ist in jedem Intervall enthalten. Ein leeres Intervall enthält nur leere Intervalle. Für nicht leere Intervalle *a* und *b* enthält *a b* genau wenn, wenn alle Zahlen von *b* auch in *a* enthalten sind.
- Eine statische Methode schnittmenge, die zwei Intervalle als Parameter übergeben bekommt und die Schnittmenge dieser Intervalle als neues

Intervall-Objekt liefert. Dabei gilt: Die Schnittmenge eines leeren Intervalls mit einem anderen Intervall ist ein leeres Intervall. Die Schnittmenge zweier nicht leerer Intervalle ist ein Intervall, das alle Zahlen umfasst, die in beiden Intervallen enthalten sind.

Aufgabe 13:

Hintergrund:
Die komplexen Zahlen erweitern den Zahlenbereich der reellen Zahlen derart, dass auch Wurzeln negativer Zahlen berechnet werden können. Dies gelingt durch Einführung einer neuen Zahl `i` als Lösung der Gleichung $x^2 = -1$. Diese Zahl `i` wird auch als *imaginäre Einheit* bezeichnet.

Komplexe Zahlen werden meist in der Form `a + b*i` dargestellt, wobei `a` und `b` reelle Zahlen (Typ `double`) sind und `i` die imaginäre Einheit ist. Man nennt `a` den *Realteil* und `b` den *Imaginärteil* von `a + b*i`.

Die Addition zweier komplexer Zahlen ist folgendermaßen definiert: `(a + bi) + (c + di) = (a+c) + (b + d)i`

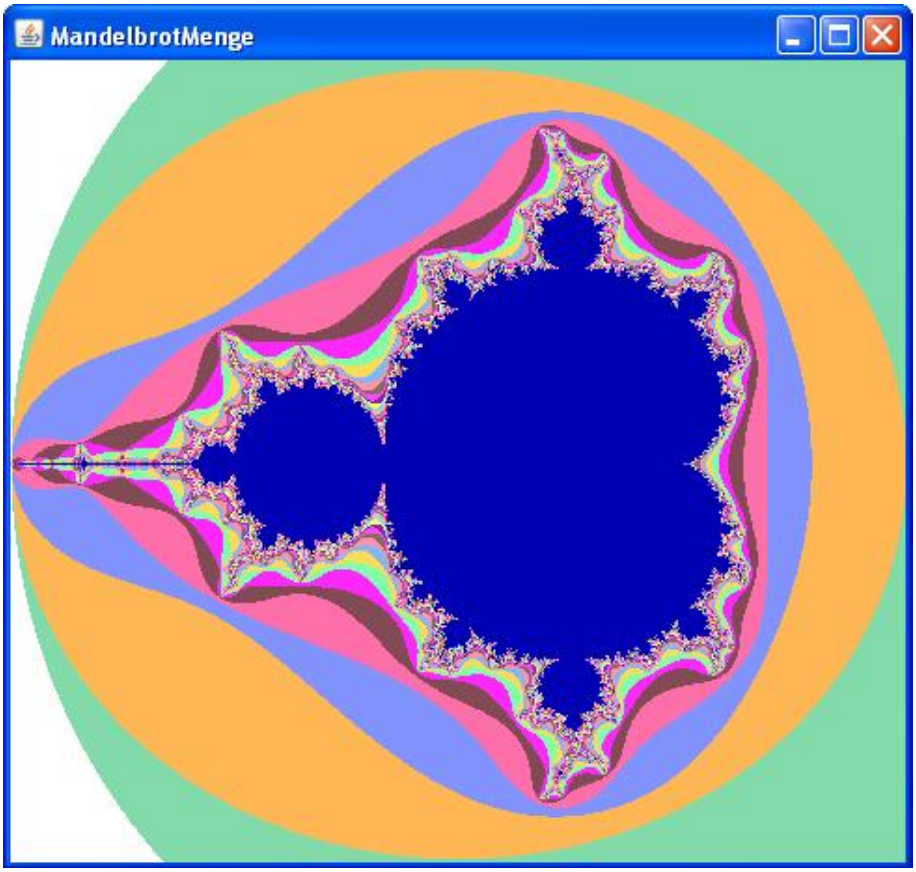
Die Multiplikation zweier komplexer Zahlen ist folgendermaßen definiert: `(a + bi) * (c + di) = (ac - bd) + (ad + bc)i`

Aufgabe:
Implementieren Sie eine ADT-Klasse `Komplex` für die Handhabung von komplexen Zahlen in Java. Die Klasse `Komplex` soll folgende 12 Methoden definieren:

1. Einen Default-Konstruktor, der die neu erzeugte komplexe Zahl mit einem geeigneten Standardwert initialisiert
2. Einen Konstruktor, dem als Parameter Werte für den Realteil und den Imaginärteil der neu erzeugten komplexen Zahl übergeben werden.
3. Einen Copy-Konstruktor
4. Eine Methode `istGleich`, die überprüft, ob das aufgerufene Komplex-Objekt wertegleich einem als Parameter übergebenen Komplex-Objekt ist.
5. Eine Methode `makeString`, die eine geeignete String-Repräsentation des Komplex-Objektes liefert.
6. Eine Methode `getRealTeil`, die den Realteil des Komplex-Objektes liefert.
7. Eine Methode `getImaginaerTeil`, die den Imaginärteil des Komplex-Objektes liefert.
8. Eine Klassenmethode `add`, die zwei Komplex-Objekte als Parameter übergeben bekommt und die Summe der beiden komplexen Zahlen als Komplex-Objekt liefert (entspricht `c1 + c2`).
9. Eine Methode `add`, die ein Komplex-Objekt als Parameter übergeben bekommt und dieses zum aufgerufenen Komplex-Objekt addiert (entspricht `c1 += c2`).
10. Eine Klassenmethode `mult`, die zwei Komplex-Objekte als Parameter übergeben bekommt und das Produkt der beiden komplexen Zahlen als Komplex-Objekt liefert (entspricht `c1 * c2`).

11. Eine Methode `mult`, die ein Komplex-Objekt als Parameter übergeben bekommt und dieses zum aufgerufenen Komplex-Objekt multipliziert (entspricht `c1 *= c2`).

Testen:
Im StudIP steht ein Testprogramm zur Verfügung, mit dem Sie testen können, ob Ihre Klasse (zumindest teilweise) korrekt ist. Wenn sich das mit Ihrer Klasse `Komplex` kompilieren lässt und (zumindest teilweise) korrekt ist, sollte folgendes Fenster mit der so genannten *Mandelbrot-Menge* auf dem Bildschirm erscheint (in Farbe):



Aufgabe 14:

Implementieren Sie eine ADT-Klasse `Binaerzahl` für die Handhabung von nicht-negativen Binärzahlen in Java. Die Klasse `Binaerzahl` soll folgende Methoden definieren:

1. Einen Default-Konstruktor, der die neu erzeugte Binärzahl mit einem geeigneten Standardwert initialisiert.
2. Einen Konstruktor, dem als Parameter ein String übergeben wird, der eine Binärzahl repräsentiert (bspw. repräsentiert der String „1011“ die Binärzahl 1011, d.h. den int-Wert 11).
3. Einen Copy-Konstruktor.

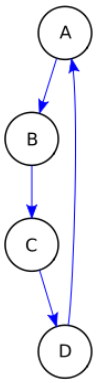
4. Eine Methode `clone`, die ein `Binaerzahl`-Objekt liefert, das wertegleich dem aufgerufenen `Binaerzahl`-Objekt ist ((Überschreiben der von der Klasse `Object` geerbten Methode `clone`).
5. Eine Methode `equals`, die überprüft, ob das aufgerufene `Binaerzahl`-Objekt wertegleich einer als Parameter übergebenen Binärzahl ist (Überschreiben der von der Klasse `Object` geerbten Methode `equals`).
6. Eine Methode `toString`, die eine geeignete String-Repräsentation des aufgerufenen `Binaerzahl`-Objektes liefert (Überschreiben der von der Klasse `Object` geerbten Methode `toString`).
7. Eine Klassenmethode `add`, die zwei `Binaerzahl`-Objekte als Parameter übergeben bekommt und die Summe der beiden Binärzahlen als `Binaerzahl`-Objekt liefert (entspricht `b1 + b2`).
8. Eine Methode `add`, die ein `Binaerzahl`-Objekt als Parameter übergeben bekommt und dieses zum aufgerufenen `Binaerzahl`-Objekt addiert (entspricht `b1 += b2`).

Aufgabe 15:

Ein *Graph* ist in der Graphentheorie eine abstrakte Struktur, die eine Menge von Objekten zusammen mit den zwischen diesen Objekten bestehenden Verbindungen repräsentiert. Die mathematischen Abstraktionen der Objekte werden dabei *Knoten* (auch Ecken) des Graphen genannt. Die paarweisen Verbindungen zwischen Knoten heißen *Kanten* (manchmal auch Bögen). Die Kanten können gerichtet oder ungerichtet sein. Häufig werden Graphen anschaulich gezeichnet, indem die Knoten durch Punkte und die Kanten durch Linien dargestellt werden (aus Wikipedia).

Anschauliche Beispiele für Graphen sind ein Stammbaum oder das U-Bahn-Netz einer Stadt. Bei einem Stammbaum stellt jeder Knoten ein Familienmitglied dar und jede Kante ist eine Verbindung zwischen einem Elternteil und einem Kind, sodass es sich insbesondere um einen Baum handelt. In einem U-Bahn-Netz stellt jeder Knoten eine U-Bahn-Station dar und jede Kante eine direkte Zugverbindung zwischen zwei Stationen.

In *gerichteten Graphen* werden Kanten statt durch Linien durch Pfeile gekennzeichnet, wobei der Pfeil vom ersten (Ausgangsknoten) zum zweiten Knoten (Endknoten) zeigt. Dies verdeutlicht, dass jede Kante des Graphen nur in eine Richtung durchlaufen werden kann.



Aufgabe:
Implementieren Sie eine ADT-Klasse `Graph`, die gerichtete Graphen repräsentiert. Knoten sollen durch Namen in Form von Strings repräsentiert werden. Die Klasse soll folgende Methoden bereitstellen:

- Einen Default-Konstruktor.
- Einen Konstruktur, dem über einen VarArgs-Parameter beliebig viele Knoten übergeben werden können.
- Eine Methode zum Hinzufügen eines weiteren Knotens zum Graphen.
- Eine Methode zum Hinzufügen einer gerichteten Kante zum Graphen. Der Ausgangsknoten der Kante wird dabei als erster, der Endknoten der Kante als zweiter Parameter übergeben.
- Eine Methode, die überprüft (und das Ergebnis der Überprüfung als Wert liefert), ob es eine direkte Verbindung (also eine Kante) zwischen zwei als Parameter übergebenen Knoten gibt. Der Ausgangsknoten der Kante wird dabei als erster, der Endknoten der Kante als zweiter Parameter übergeben.
- Eine Methode, die überprüft (und das Ergebnis der Überprüfung als Wert liefert), ob es eine (nicht-unbedingt direkte) Verbindung zwischen zwei als Parameter übergebenen Knoten gibt. Der Ausgangsknoten der Kante wird dabei als erster, der Endknoten der Kante als zweiter Parameter übergeben. Eine Verbindung existiert dabei genau dann, wenn der angegebene Endknoten entlang einer Menge an Kanten ausgehend vom Ausgangsknoten erreicht werden kann. In der obigen Abbildung gibt es also bspw. eine Verbindung zwischen Knoten „A“ und „D“, und zwar via „B“ und „C“.

Überlegen Sie sich eine adäquate Datenstruktur zur Speicherung von Knoten und Kanten.

Testen Sie die Ihre Klasse `Graph` bspw. mit folgendem Programm. Es sollte die Ausgaben „true“ und „false“ produzieren:

```
public static void main(String[] args) throws Exception {
     Graph graph = new Graph("1", "2", "3", "4", "5");
     graph.addEdge("1", "2");
     graph.addEdge("2", "3");
     graph.addEdge("3", "2");
     graph.addEdge("3", "4");
```

```
        graph.addEdge("1", "5");
        System.out.println(graph.existsConnection("1", "4"));
        System.out.println(graph.existsConnection("2", "5"));
    }
```