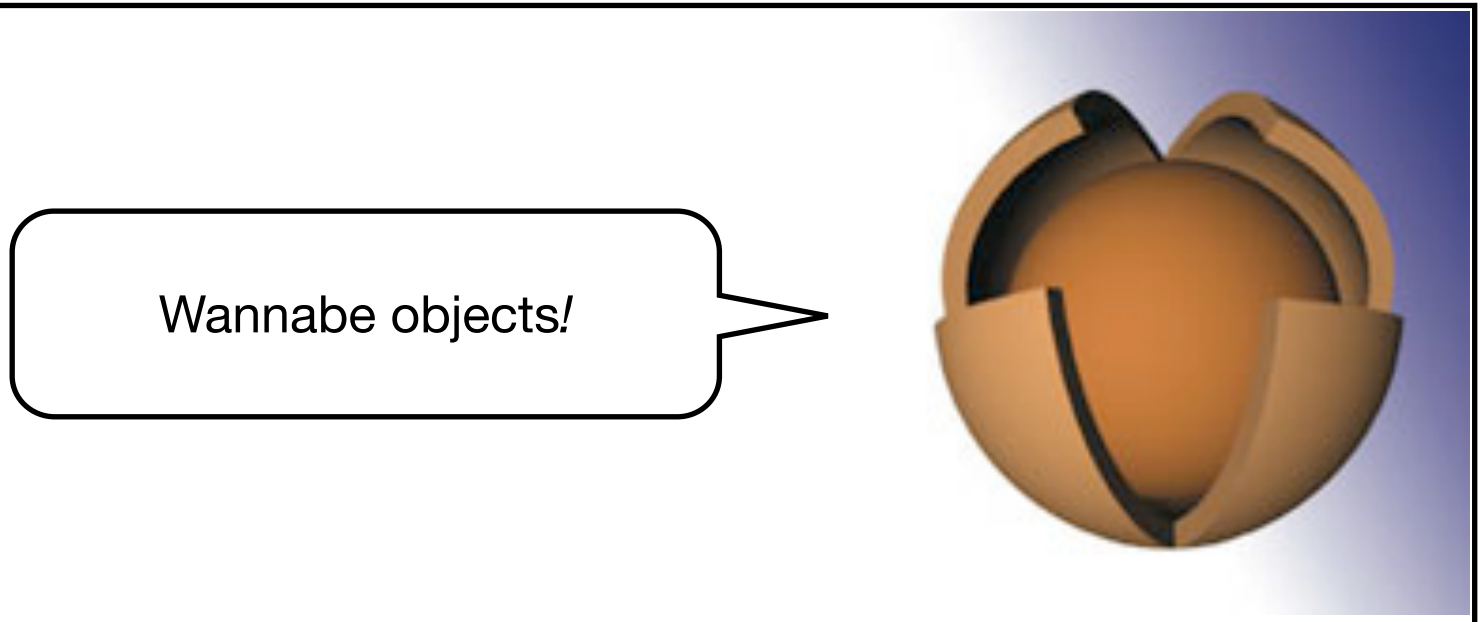


Verbunde und Datenkapseln

OOPM, Ralf Lämmel

Das Konzept des Datentyps

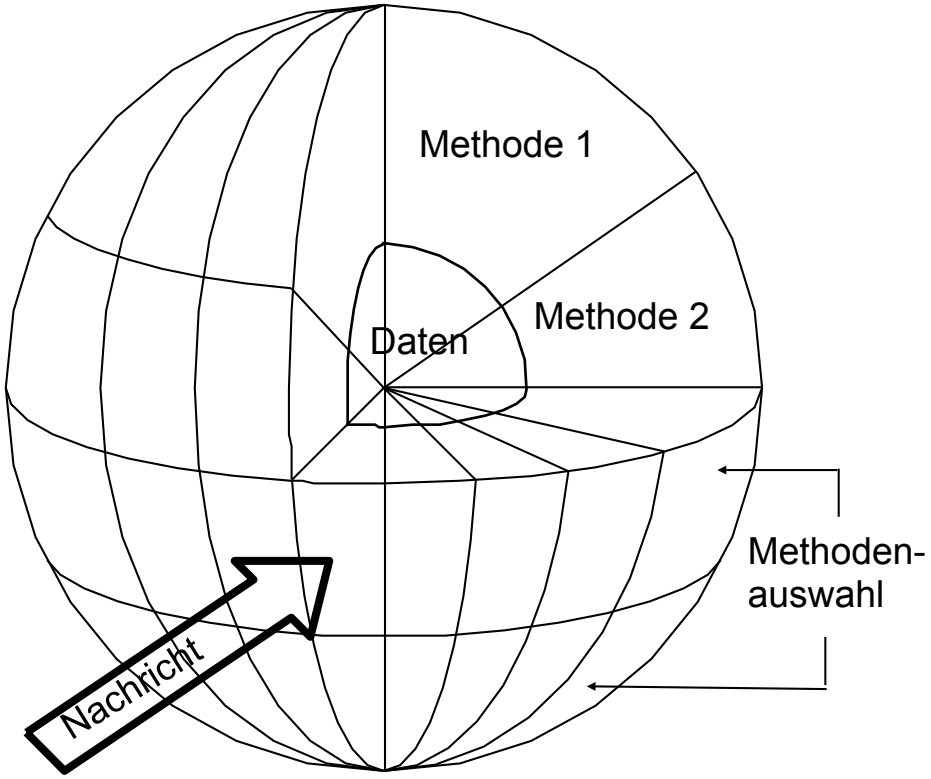
- **Datentyp = Wertebereich + Operationen**
- Vordefinierte Datentypen:
  - int, float, double, ...: +, -, *, ...
  - boolean: !, &&, ||, ...
  - String - Methoden:
    - concat: Verkettung
    - indexOf: Suche von Teilstring
    - toLowerCase: Konvertierung nach Kleinschreibung
    - ...

Verbunde und Datenkapseln unterstützen die Definition gewisser (einfacher) Datentypen, wie wir bald sehen werden.

Bedarf an programmdefinierten Datentypen

- Benötigte Datentypen mögen fehlen:
  - Float und double gibt es - wo ist "***complex***"?
  - Wo ist ein Datentyp für ***Bankkonten***?
- Existierende Datentypen mögen inadäquat sein:
  - Wo sind ***erweiterbare Felder***?
  - Wie speichert man *dünn besiedelte* Matrizen?

Datenkapseln (“einfache Objekte”)



z.B.:
Bankkonten

Arten von Operationen

- Prefix:

  -a

- Infix:

  a-b

- Statische Methoden:

  MyMath.*minus*(*a*,*b*)

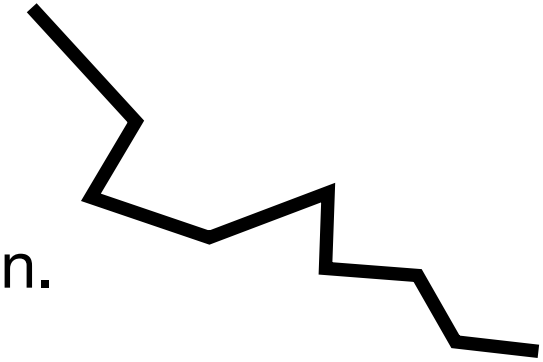
- Instanz-Methoden:

  a.*minus*(*b*)

Verwandt zu prozeduraler, funktionaler und modularer Programmierung

Verwendung mit ***Datenkapseln***; Vorhut der OO Programmierung

Beispiel: ein Datentyp für Zickzack-Linien



- Eine **Zickzack-Linie** besteht aus vielen Punkten.
- Ein **Punkt** besteht aus x- und y-Koordinate.
- Denkbare Operationen für solche Linien:
  - Einfügen/Löschen/Bewegen von Punkten
  - Berechnen der Länge der Linie
  - ...

Wie können wir die Koordinaten eines Punktes gruppieren?

Das Konzept des Verbunds

Vergl. Felder

- ***Verbund*** (Struktur, engl.: Record): Datenstruktur bestehend aus einer endlichen, ungeordneten Gruppe von benannten Datenelementen potentiell *verschiedener* Typen.
- ***Verbundtyp***: Deklaration eines Schemas für Verbünde.
- Java's *Klassendeklarationen* decken Verbundtypen ab.
- Beispiel:

```
public class Point {

  public double x, y;

}
```

Siehe package data.zigzag

Beispiel: Kaufauftragspositionen

```
public class Position {
  public int number;
  public String product;
  public int quantity;
  public float price;
}
```

Klassisch heißen die Elemente eines Verbunds Komponenten. In einer OO-Sprache nennt man sie eher Attribute.

Operationen für Verbunde

- Attributzugriff (“.”)
- Anforderung (“new”)
- Vergleich mit “null”
- Statische Methoden
- ...

Variablen von einem Verbundtyp mögen auf “***null***” zeigen -- in Analogie zu Feldtypen. Solche Variablen speichern in der Tat ***Zeiger auf Verbunde***.

```
public static void main(String[] args) {
   Point p1 = new Point();
   p1.x = 3;
   p1.y = 4;
   Point p2 = new Point();
   p2.x = 6;
   p2.y = 8;
   System.out.println(distance(p1, p2)); // prints 5
}
```

Verwendung von Verbunden

- Verbunde werden ähnlich wie andere Werte verwendet:
  - als Argumente für Methoden,
  - als Ergebnisse von Methoden,
  - als Elemente in Feldern,
  - als Werte lokaler Variablen,
  - etc.

Verwendung von Verbunden

- Verbunde werden ähnlich wie andere Werte verwendet:
  - **als Argumente für Methoden**,
  - als Ergebnisse von Methoden,
  - als Elemente in Feldern,
  - als Werte lokaler Variablen,
  - etc.

```
public static double distance(Point p1, Point p2) {
   double deltax = p1.x - p2.x;
   double deltay = p1.y - p2.y;
   return Math.sqrt(deltax*deltax + deltay*deltay);
}
```

Verwendung von Verbunden

- Verbunde werden ähnlich wie andere Werte verwendet:
  - als Argumente für Methoden,
  - als Ergebnisse von Methoden,
  - **als Elemente in Feldern**,
  - als Werte lokaler Variablen,
  - etc.

Wir finden die erste freie Stelle durch eine “null” im Feld.

```
public static double length(Point[] line) {
   double result = 0;
   for (int i=1; i<line.length && line[i] != null; i++)
      result += distance(line[i-1],line[i]);
   return result;
}
```

Vom Verbund zur Datenkapsel

- **Ein *Verbund* gruppiert Datenelemente.**
- **Eine *Datenkapsel* gruppiert zusätzlich Operationen.**
- Typischerweise wird Geheimhaltung verlangt.

  (Nur die Operationen sind von außen sichtbar.)

  - Im Englischen:
    - Encapsulation
    - Information hiding

Beispiel: Punktkapseln (öffentlich)

```
public class Point {
   public double x, y;
   public double distanceTo(Point p) {
      double deltax = x - p.x;
      double deltay = y - p.y;
      return Math.sqrt(deltax*deltax + deltay*deltay);
   }
}
```

Siehe package data.zigzag

Deklarationsform für Datenkapseln (vorläufig)

```
public class Datentypsname {

  public Typ1 Komponente1;
  ...
  public TypN KomponenteN;

  public Result1 Operation1(... Argumente1 ...) { ... };
  ...
  public ResultM OperationM(... ArgumenteM ...) { ... };

}
```

Beispiel: Punktkapseln (verschlossen)

```
public class Point {
   private double x, y;
   public double getX() { return x; }
   public double getY() { return y; }
   public void setX(double newx) { x = newx; }
   public void setY(double newy) { y = newy; }
   public double distanceTo(Point p) {
      double deltax = x - p.getX();
      double deltay = y - p.getY();
      return Math.sqrt(deltax*deltax + deltay*deltay);
   }
}
```

Beispiel: Punktkapseln (verschlossen)

```
public class Point {
    private double x, y;
    public double getX() { return x; }
    public double getY() { return y; }
    public void setX(double newx) { x = newx; }
    public void setY(double newy) { y = newy; }
    public double distanceTo(Point p) {
        double deltax = x - p.x;
        double deltay = y - p.y;
        return Math.sqrt(deltax*deltax + deltay*deltay);
    }
}
```

Ok, da gleiche Klasse.

Idiom: Getter und Setter

- Gestatte teilweisen Zugriff auf Repräsentation.
- Getter
  - Methode für lesenden Attributzugriff
  - Namenskonvention: get...
- Setter
  - Methode für schreibenden Attributzugriff
  - Namenskonvention: set...

Getter und Setter müssen nicht direkt zu Attributen korrespondieren. Zusätzliche Aktionen sowie die Übersetzung zur tatsächlichen Repräsentation mögen stattfinden.

Deklarationsform für Datenkapseln

```
public class Datentypsname {

  private Typ1 Komponente1;
  ...
  private TypN KomponenteN;

  public Result1 Operation1(... Argumente1 ...) { ... };
  ...
  public ResultM OperationM(... ArgumenteM ...) { ... };

}
```

Sprachkonstruktion der Konstruktoren

- Spezielle Methoden zur Initialisierung von Datenkapseln

```
public class Point {
   private double x, y;
   public Point(double x, double y) {
      setX(x);
      setY(y);
   }
   public void setX(double newx) { x = newx; }
   public void setY(double newy) { y = newy; }
  ...
}
```

- Aufruf durch “new”

```
public static void main(String[] args) {
   Point p = new Point(3,4);
   ...
}
```

Der Zeiger “this”

- Spezieller Zeiger auf aktuelle(n) Verbund/Kapsel

```
public double getX() { return x; }
public double getY() { return y; }
public void setX(double x) { this.x = x; }
public void setY(double y) { this.y = y; }
```

- Anwendungsfälle:
  - Auflösung von Namenskonflikten
  - Dokumentation (Klarheit)
  - “this” als Methodenargument

Zwischenzusammenfassung

Terminologie

* Eine ***Datenkapsel*** ist eine Datenstruktur in einem Programm welche grundlegendere Daten (wie z.B. *Kontendaten*) zusammen mit Operationen über diesen Daten (etwa *Einzahlung*) gruppiert.

* Ein ***Verbund*** ist eine Datenkapsel ohne besondere Operationen. (Ein Verbund kann immerhin dazu verwendet werden, um mehrere Datenpartikel zu gruppieren. Zum Beispiel kann damit ein Punkt aus zwei Koordinaten gebildet werden.)

Implementation von Datentypen durch Datenkapseln

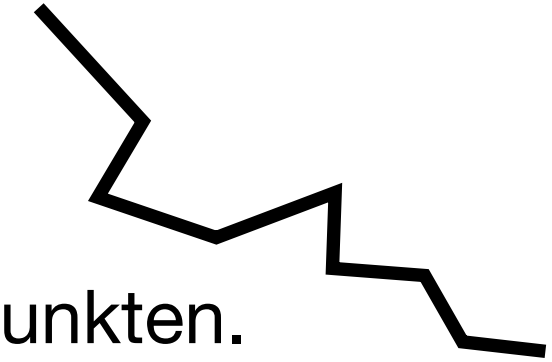
- Implementationsschritte
  - Festlegung einer *Repräsentation* des Wertebereichs
  - *Implementation* von Operationen
  - *Kapselung* von Daten und Operationen
- Verwendung des Datentyps im Programm
  - Erzeugung von Kapseln
  - Anwendung von Operationen

Zwischenzusammenfassung: klassenbasierte Implementation von Datentypen (in Java)

- Abstraktionsmechanismus *Klasse* (**class**)
  - Datenrepräsentation als *Attribute*
  - Operationen als *Instanzmethoden*
  - Instanziierung mit **new**
  - Operationsanwendung als *Methodenaufruf*

Mehr Beispiele

Beispiel: ein Datentyp für Zickzack-Linien



- Eine Zickzack-Linie besteht aus vielen Punkten.
- Ein Punkt besteht aus x- und y-Koordinate.
- Denkbare Operationen für solche Linien:
  - Einfügen/Löschen/Bewegen von Punkten
  - Berechnen der Länge der Linie
  - ...

Beispiel: Zickzack-Linien

```
public class Line {
   private Point[] line = new Point[42];
   public void add(Point p) {
      int i;
      for (i=0; i<line.length && line[i]!=null; i++) {}
      line[i] = p;
   }
   public double length() {
      double result = 0;
      for (int i=1; i<line.length && line[i] != null; i++)
         result += line[i-1].distanceTo(line[i]);
      return result;
   }
   ...
}
```

Siehe package data.zigzag

Beispiel: Ein Bankkonto

- Ein Bankkonto ist determiniert durch ...
  - den Kontostand
  - und eventuell den Kreditrahmen.
- Bankkonten erlauben Operationen für ...
  - das Einzahlen von Geld,
  - das Auszahlen von Geld,
  - u.a.

Ein Datentyp für Bankkonten

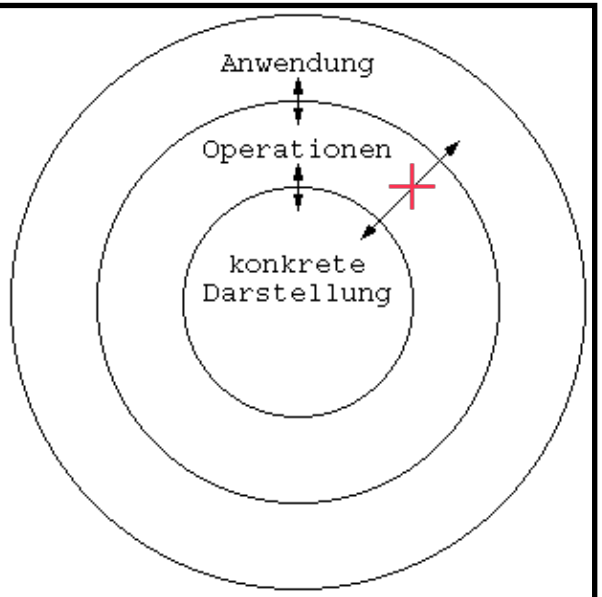
```
public class Account {
   public float balance;
   public void deposit(float amount) {
      balance += amount;
   }
   public void withdraw(float amount) {
      balance -= amount;
   }
}
```

Während es für Punkte noch wenig wichtig war, ob deren Attribute geheim gehalten werden, erscheint es für Konten sehr wichtig, den Zugriff zu reglementieren.

Warum?

Siehe package data.account

Geheimhaltung



- **Die konkrete Darstellung sollte geheim sein wenn ...**
  - direkter Zugriff zu *inkonsistenten Änderungen* führen kann,
  - der Zugriff überwacht werden soll (etwa für “Logging”),
  - die Darstellung eine *niedrige Abstraktionsebene* benutzt,
  - Aufrufer unnötig abhängig von Darstellung werden könnten.

Datenkapselung von Bankkonten (Variante mit Geheimhaltung)

```
public class Account {
   private float balance;
   public float getBalance() {
      return balance;
   }
   public void deposit(float amount) {
      balance += amount;
   }
   public void withdraw(float amount) {
      balance -= amount;
   }
}
```

Die Invariante für ein nicht negatives Konto wird noch nicht sichergestellt.

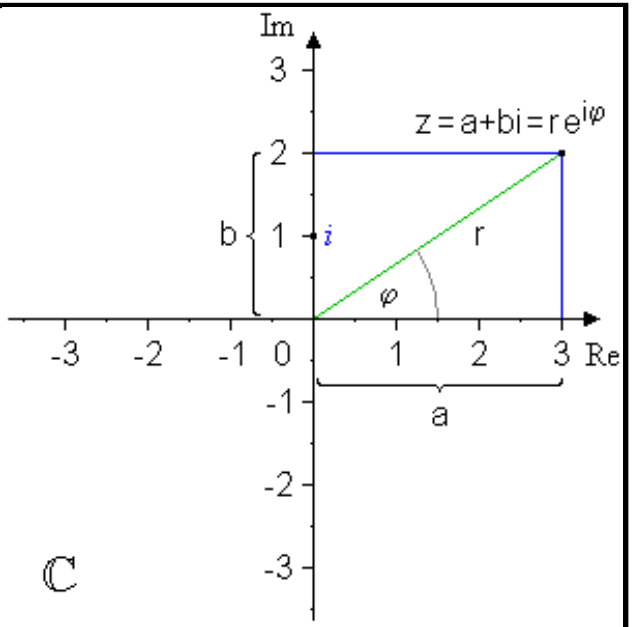
Datenkapselung von Bankkonten (Variante mit Konsistenzsicherung)

```
public class Account {
   private float balance;
   public float getBalance() { return balance; }
   public void deposit(float amount) {
      if (amount <= 0)
         return;
      balance += amount;
   }
   public float withdraw(float amount) {
      if (amount <= 0 || balance < 0)
         return 0;
      float result = 0;
      result = amount > balance ? balance : amount;
      balance -= result;
      return result;
   }
}
```

Diese einfachen Konten bestehen nur aus einer Komponente (dem Kontostand). Es ist ein trivialer Verbund, aber eine interessante Datenkapsel.

Ist dieser Teil der Bedingung notwendig?

Beispiel: Komplexe Zahlen



http://de.wikipedia.org/wiki/Komplexe_Zahl

- ***Optionen für Repräsentationen***
  - Kartesische Form:

    Real- und Imaginärteil

    Präferenz für heute
  - Polarform:

    Betrag und Winkel
- Operationen für Komplexe Zahlen
  - Addition, Substraktion, Multiplikation, Division
  - Wurzel, Exponentation, ...

Operationen für Komplexe Zahlen

* Addition

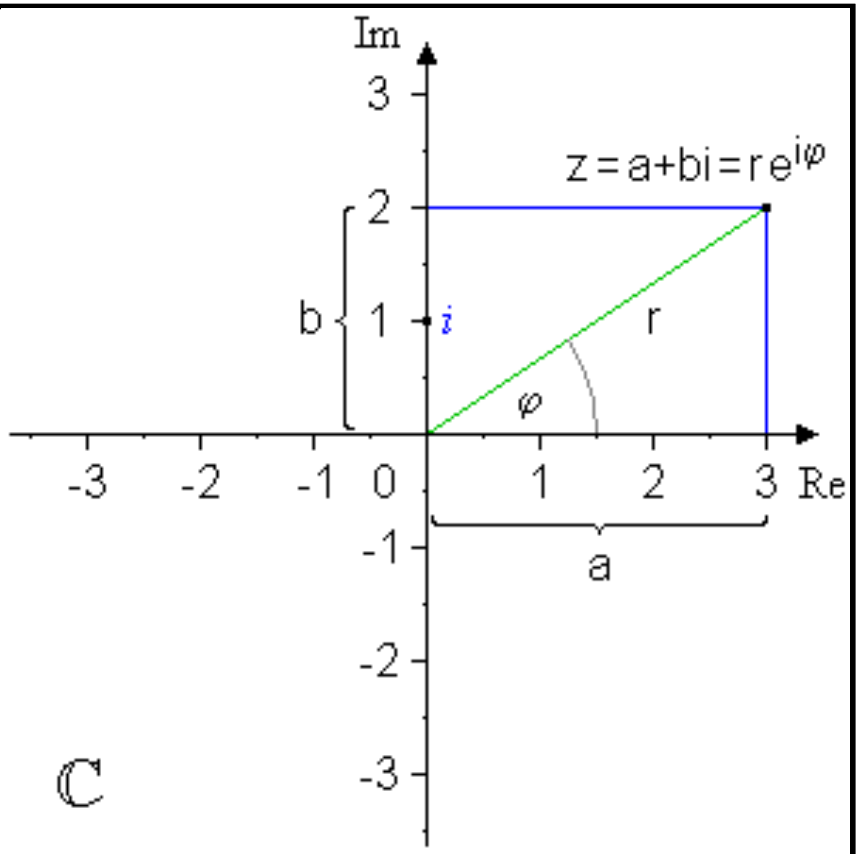
$$(a + b\,\mathrm{i}) + (c + d\,\mathrm{i}) = (a + c) + (b + d)\,\mathrm{i}$$

* Subtraktion

$$(a + b\,\mathrm{i}) - (c + d\,\mathrm{i}) = (a - c) + (b - d)\,\mathrm{i}$$

* ...

Zusammenhänge



$\mathbb{C}$

http://de.wikipedia.org/wiki/Komplexe_Zahl

$$a = r \cdot \cos\varphi$$

$$b = r \cdot \sin\varphi$$

$$r = |z| = \sqrt{a^2 + b^2}$$

Winkel im Bogenmaß (Radiant) im Bereich -π bis + π

$$\varphi = \arg(z) = \begin{cases} \arctan\frac{b}{a} & \text{für } a > 0, b \text{ beliebig} \\ \arctan\frac{b}{a} + \pi & \text{für } a < 0, b \geq 0 \\ \arctan\frac{b}{a} - \pi & \text{für } a < 0, b < 0 \\ \pi/2 & \text{für } a = 0, b > 0 \\ -\pi/2 & \text{für } a = 0, b < 0 \\ \text{unbestimmt} & \text{für } a = 0, b = 0 \end{cases}$$

```
public class Complex {
   private double re;
   private double im;
   public Complex(double re, double im) {
      this.re = re;
      this.im = im;
   }
   public double getRe() { return re; }
   public double getIm() { return im; }
   public double getModulus() { return Math.hypot(re, im); }
   public double getRadianAngle() { return Math.atan2(im, re); }
   public double getAngle() { return getRadianAngle() * 180 / Math.PI; }
   public Complex addTo(Complex c) {
      return new Complex(
         this.getRe() + c.getRe(),
         this.getIm() + c.getIm());
   }
}
```

$$a = r \cdot \cos\varphi$$

$$b = r \cdot \sin\varphi$$

$$r = |z| = \sqrt{a^2 + b^2}$$

$$\varphi = \arg(z) = \begin{cases} \arctan\frac{b}{a} & \text{für } a > 0, b \text{ beliebig} \\ \arctan\frac{b}{a} + \pi & \text{für } a < 0, b \geq 0 \\ \arctan\frac{b}{a} - \pi & \text{für } a < 0, b < 0 \\ \pi/2 & \text{für } a = 0, b > 0 \\ -\pi/2 & \text{für } a = 0, b < 0 \\ \text{unbestimmt} & \text{für } a = 0, b = 0 \end{cases}$$

Übungsaufgabe:
mehr Operationen!

Siehe package data.complex

Beispiel: Erweiterbare Felder

- Stärken von normalen Feldern
  - Index-basierter, schneller Zugriff
- Schwächen von normalen Feldern
  - Größe muss bei Feldkonstruktion bekannt sein
- Mögliche Verbesserung
  - Datentyp mit wachsender Feldgröße

Operationen von Feldern

	Klassische Felder	**Erweiterbare Felder**
Anforderung	... new int[42] ...	... new IntArray() ...
Abfrage der Länge	... myArray.length ...	... myArray.getLength() ...
Lesender Zugriff	... = ... myArray[i] ...;	... = ... myArray.getAt(i) ...
Schreibender Zugriff	myArray[i] = ...;	myArray.setAt(i,...);

Erweiterbare Int-Felder

```
public class IntArray {
   private int[] a = new int[0];
   private int length = 0;
   private void resizeIfNecessary(int i) { ... }
   public int getLength() { return length; }
   public int getCapacity() {   return a.length; }
   public int getAt(int i) {
      resizeIfNecessary(i);
      return a[i];
   }
   public void setAt(int i, int x) {
      resizeIfNecessary(i);
      a[i] = x;
   }
}
```

Siehe package data.array

Dynamische Erweiterung

```
private void resizeIfNecessary(int i) {
   if (i>= length)
      length = i+1;
   if (i>=a.length) {
      int[] b = new int[2* length];
      for (int j=0; j<a.length; j++)
         b[j] = a[j];
      a = b;
   }
}
```

Siehe package data.array

Zusammenfassung

- Verbunde gruppieren heterogene Komponenten.
- Datenkapsel = Verbund mit Operationen
- Prinzip der Geheimhaltung

Beginn eines wiederkehrenden Beispiels:
das ***Bankkonto***

Ausblick

- Spezifikation von Programmen
- Testen von Programmen
- Verifikation von Programmen
- Komplexität von Programmen