

# DAS COLLECTIONS-FRAMEWORK

## Komplexere Datenstrukturen und der bewusste Umgang mit Ausnahmefehlern

Struktur und Inhalt des Kurses wurden 2012 von Markus Heckner entwickelt. Im Anschluss haben Alexander Bazo und Christian Wolff Änderungen am Material vorgenommen. Die aktuellen Folien wurden von Alexander Bazo erstellt und können unter der [MIT-Lizenz](#) verwendet werden.

# AKTUELLER SEMESTERFORTSCHRITT (WOCHE 12)

Kursabschnitt	Themen		
Grundlagen	Einführung	Einfache Programme erstellen	Variablen, Klassen & Objekte
	Kontrollstrukturen & Methoden	Arrays & komplexe Schleife	
Klassenmodellierung	Grundlagen der Klassenmodellierung	Vererbung & Sichtbarkeit	
Interaktive Anwendungen	Event-basierten Programmierung	String- & Textverarbeitung	
Datenstrukturen	Listen, Maps & die Collections	Speicherverwaltung	Umgang mit Dateien
Software Engineering	Debugging	Planhaftes Vorgehen bei der Softwareentwicklung	
	Qualitätsaspekte von Quellcode		

## DIE OOP-WEIHNACHTSCHALLENGE 2019/20

Vor der Weihnachtspause haben wir Ihnen eine Aufgabe gestellt:

**"Entwerfen Sie ein weihnachtliches *Jump and Run*-Spiel".**

Alle Einreichungen wurden vom OOP-Team gespielt und bewertet.

Vielen Dank für die Teilnahme!

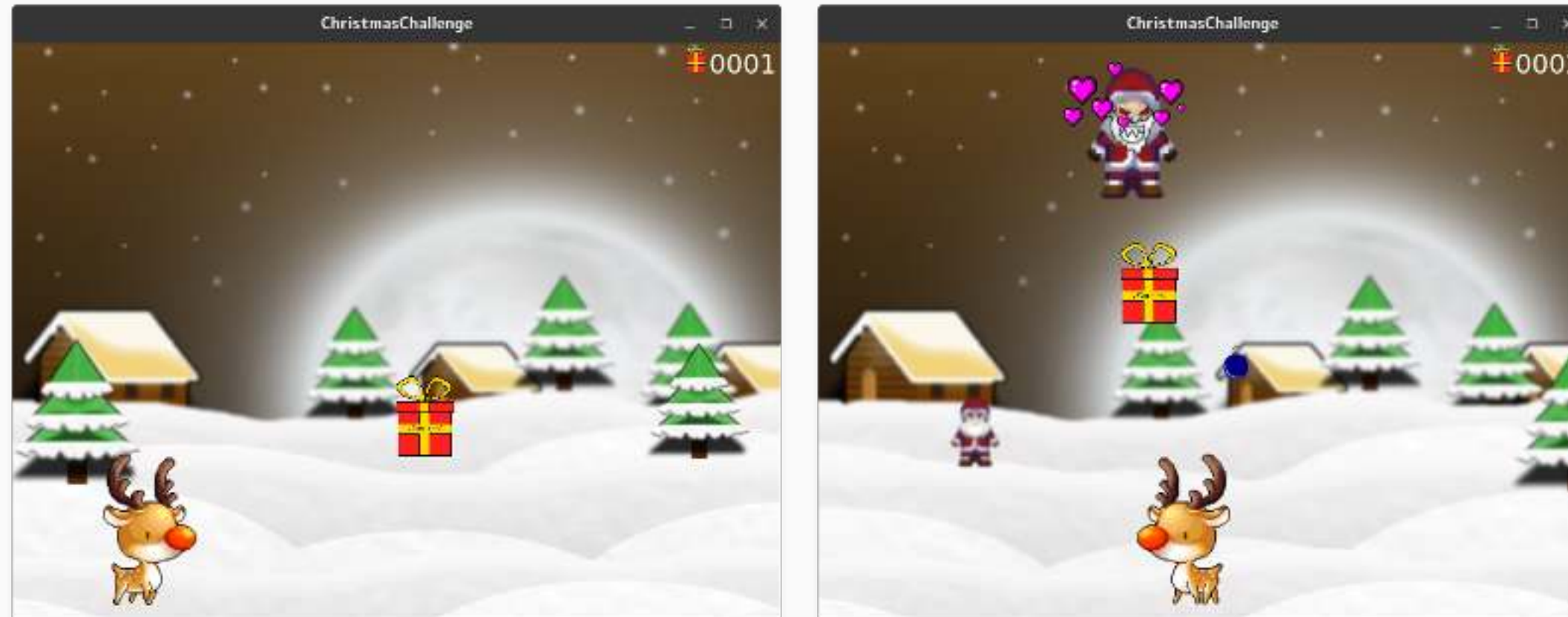
**Dies sind die Preisträger.**

### 3. PLATZ: "THE DAY AFTER CHRISTMAS EVE" (ELINA MAIER)



Verhelfen Sie *Santa* zum wohlverdienten Feierabendwein ...

## 2. PLATZ: "RUDOLPH VS KRAMPUS" (JANNIK WIESE)



Sammeln Sie Geschenke um im *Boss Fight* gegen Krampus zu bestehen ...



## 1. PLATZ: "SANTA CAN JUMP" (DAVID RING)



Meistern Sie Hindernisse und Fallen auf mehreren Karten ...

# RÜCKBLICK

- Funktionale Vollständigkeit ist ein wichtiger, aber nicht der einzige Teil guter Software: Hohe Codequalität ist wichtig!
- *Bugs* lassen sich nicht vollständig vermeiden: Wir benötigen Strategien zum Identifizieren und Beheben dieser Probleme.
- *Printlining* und der *Debugger* sind zwei Strategien zum systematischen Beheben von Programmfehlern.
- Moderne Entwicklungsumgebungen unterstützen sich beim Beheben von Fehlern und beim Steigern der Codequalität.

# DAS PROGRAMM FÜR HEUTE

- Der Umgang mit unvermeidbaren *Exceptions*
- Komplexere Datenstrukturen: *Maps* und das *Collections-Framework*
- Praktischer Umgang mit *HashMaps*



# AUSBLICK AUF DIE NÄCHSTEN WOCHEN

- In den Übungen dieser Woche bitten wir um Ihre Hilfe: Wir möchten anonyme Interaktionsdaten über den Umgang mit der IntelliJ-IDE sammeln.
- In der nächsten Woche werden Vorlesung und Übung evaluiert: Bitte bringen Sie Laptop oder Smartphone mit.

**Hinweis:** Vergessen Sie nicht die **Klausuranmeldung** in Flexnow vom 1. bis zum 9. Februar!

# DER UMGANG MIT *EXCEPTIONS*

# JAVA KOMMUNIZIERT FEHLER ÜBER EXCEPTIONS

[illegible]

## EXCEPTIONS VS ERRORS

Wenn Java auf einen Fall stößt, in dem es nicht normal weiterarbeiten kann, wird eine *Exception* verursacht: die relevante Methode wirft (*throws*) eine *Exception* (Ausnahme). Eine Ausnahme ist, im Unterschied zu einem (*terminal*) *error*, dabei eine zwar außergewöhnliche, aber in der Regel korrigierbare Situation während des Programmablaufs:

- Java verlangt von Programmen bzw. den ProgrammierInnen, mit bestimmten Ausnahmen umgehen zu können.
- *Exceptions* sind eine Art Notfallplan: Wenn alles gut geht, läuft das Programm wie gehabt weiter. Wenn etwas schief geht, kümmert sich das Programm um diese Ausnahme.

## DER UMGANG MIT EXCEPTIONS

**Exceptions abfangen und verarbeiten:** Verschieden Stellen des eigenen oder fremden Code können Ausnahmen auslösen. Dies wird über entsprechende Syntaxelemente kommuniziert. Nutzen wir diese Stellen, *müssen wir im aufrufenden Code einen Plan für das Abfangen dieser Fehlermöglichkeit definieren.*

**Exceptions erzeugen:** In unserem eigenen Code können wir *Exceptions* einsetzen, um Ausnahmefälle zu definieren und Möglichkeiten zum Umgang mit diesen anzubieten.

**Hinweis:** Der übermäßige Einsatz von *Exceptions* ist nicht unumstritten: Häufig bietet es sich an, Fehlersituationen direkt zu verarbeiten und über entsprechende Rückgabewerte zu kommunizieren, statt die Fehlerbehandlung über *Exceptions* nach Außen zu verlagern.

## TYPISCHE FÄLLE FÜR "KRITISCHE" STELLEN

- **Arbeit mit Dateien:** Was passiert, wenn die gewünschte Datei nicht vorhanden ist?
- **Netzwerkkommunikation:** Was passiert, wenn keine Internetverbindung aufgebaut werden kann?
- **Nutzereingaben:** Was passiert, wenn zur Laufzeit nicht-kompatible Eingaben getätigt werden?

**Hinweis:** Im eigenen Code können wir die vorgegebenen Fehlerfälle erweitern oder ergänzen, um die *Exceptions* an den konkreten Anwendungsfall anzupassen, z.B. in dem statt einer `FileNotFoundException`-Meldung ein inhaltlich deutlicherer `ConfigurationFileNotFoundException`-Hinweis wird.



## KRITISCHE STELLEN AUSPROBIEREN

Stellen im Quellcode, die unter Umständen einen Fehler oder *Exception* verursachen könnten, müssen in Java entsprechend gekennzeichnet werden. Die *Runtime* versucht dann, den Code auszuführen. Ob eine Methode/Klasse eine *Exception* auslösen könnte, steht in der Dokumentation bzw. wird Ihnen von *IntelliJ* mitgeteilt:

### Scanner

```
public Scanner(File source)
    throws FileNotFoundException
```

Constructs a new Scanner that produces values scanned from the specified file. Bytes from the file are converted into characters using the underlying platform's default charset.

#### Parameters:

source - A file to be scanned

#### Throws:

FileNotFoundException - if source is not found

Quelle: [Oracle Java Dokumentation](#)

## TRY & CATCH

Kritische Stellen werden explizit über einen try-Block gekennzeichnet. Für den *Fall der Fälle* wird eine mögliche Fehlerbehandlung im catch-Block formuliert:

```
1 // Erstellt einen StringJoiner (StringBuilder mit automatisch eingefügten Trennzeichen)
2 // Hier wird nach jedem "Eintrag" ein plattformspezifischer Zeilenumbruch eingefügt
3 StringJoiner out = new StringJoiner(System.getProperty("line.separator"));
4 try {
5     // Beim Erstellen des Scanners mit der übergebenen Datei kann eine Ausnahme
6     // auftreten, wenn die angegebene Datei nicht gefunden wird.
7     Scanner in = new Scanner(new File("input.txt"));
8     while (in.hasNext()) {
9         out.add(in.nextLine());
10    }
11 // Tritt beim Ausführen des Codes einer der vorgesehenen Fehler auf, wird dieser Code
12 // ausgeführt.
13 } catch (FileNotFoundException e) {
14     e.printStackTrace();
15 }
```

## DAS EXCEPTIONS-OBJEKT

```
1 try {  
2     // Dangerous stuff: z.B. Datei öffnen, lesen, schließen  
3 } catch (Exception e) {  
4     // Umgang mit der möglichen Fehlersituation  
5 }
```

**Hinweis:** *Exceptions* werden als Objekte kommuniziert (Superklasse *Exception*), die Informationen zum jeweiligen Fehler enthalten.

## DER BESONDERE FEHLER: RUNTIME EXCEPTIONS

**Hinweis:** Bestimmte *Exceptions* (solche, die von der Klasse `RuntimeException` abgeleitet werden) behandeln häufig auftretende Fehler und müssen nicht explizit durch `try ... catch` abgefangen werden (der Code würde sonst sehr unübersichtlich werden). Die Fehler treten trotzdem auf und können zu Abstürzen führen.

```
public class IndexOutOfBoundsException  
    extends RuntimeException
```

Thrown to indicate that an index of some sort (such as to an array, to a string, or to a vector) is out of range.

Applications can subclass this class to indicate similar exceptions.

**Since:**

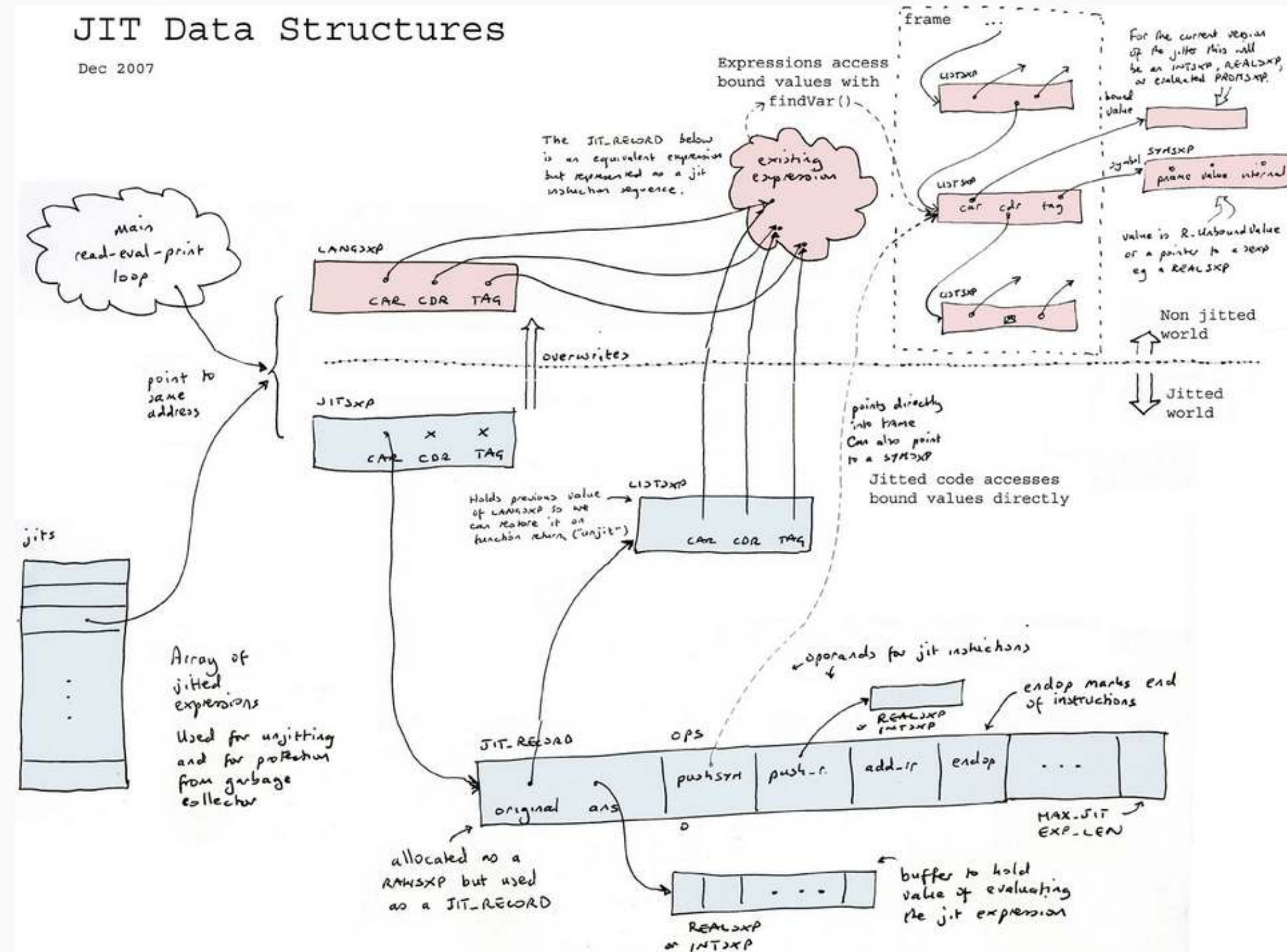
JDK1.0

**See Also:**

[Serialized Form](#)

Quelle: [Oracle Java Dokumentation](#)

# KOMPLEXE DATENSTRUKTUREN



Quelle: <http://www.milbo.users.sonic.net/ra/dstruct.html>

## EIN KURZER RÜCKBLICK AUF INTERFACES

1. Interfaces werden wie Klassen in Dateien mit der Endung `.java` definiert und bestehen aus einer Liste von öffentlichen Konstanten und Methodensignaturen (!).
2. Bei der Definition einer Klasse können über das `implements`-Schlüsselwort ein oder mehrere Interfaces angegeben werden: Die Klasse *implementiert* dann diese Interfaces.
3. Implementiert eine Klasse ein Interface, müssen innerhalb der Klasse alle im Interface enthaltene Methoden mit der korrekten Signatur und einem frei wählbaren Rumpf definiert werden.
4. Implementiert eine Klasse ein Interface, können Instanzen der Klasse als Instanzen der Interface behandelt werden.



# INTERFACES IN DER GRAPHICSAPP

## Scalable

Line, Image, Ellipse und Rect implementieren das Scalable-Interface. Alle Objekte sind dadurch über die `scale`-Methode skalierbar. Die eigentliche Implementierung ist jeweils unterschiedlich, die öffentliche Interaktion ist aber die gleiche.

## Resizable

Ellipse, Image und Rect implementieren das Resizable-Interface. Die Größe von allen Objekte kann über die Methode `setSize`

# INTERFACES ALS BEISPIEL FÜR POLYMORPHISMUS IN JAVA

Polymorphismus (altgriechisch für *Vielgestaltigkeit*) ist ein Programmier-Konzept, das es erlaubt, den Inhalt einer Variable – abhängig von seiner aktuellen Verwendung – als Wert unterschiedlicher Typen zu benutzen.

Scalable und Resizable stellen abstrakte, allgemeinen Datentype dar. Alle Instanzen von Klassen, die die Interfaces implementieren, können als solche verwendet werden. Zugänglich sind dann aber nur die jeweiligen Interface-Methoden!

```
1 Scalable obj;  
2 obj = new Rect(0, 0, 100, 100);  
3 obj.scale(2, 2);  
4  
5 obj = new Ellipse(0, 0, 100, 100);  
6 obj.scale(2, 2);
```

**Hinweis:** Zur Laufzeit wird die jeweils passende Implementierung der Methode genutzt, also die, die in der Klasse der aktuell in obj gespeicherten Instanz definiert ist.

# KOMPLEXERE DATENSTRUKTUREN

# KOMPLEXERE ANWENDUNGEN ERFORDERN KOMPLEXERE DATENSTRUKTUREN UND -MODELLE

The image displays two side-by-side screenshots of a mobile application interface, illustrating a search function. Both screenshots show a window titled 'App' with a standard Android-style title bar (minimize, maximize, close buttons).

**Left Screenshot:** The app has an orange header bar with the text 'Enter Name to search'. Below the header, there is a list of five search results, each displayed in a blue box with white text. The results are:

- Martin Brockelmann  
0941 943 3524
- Raphael Wimmer  
0941 943 3170
- Alexander Bazo  
0941 943 4958
- Patricia Böhm  
0941 943 5099
- Victoria Böhm  
0941 943 5099

**Right Screenshot:** The app has the same orange header bar, but the text is 'Bö'. Below the header, there is a list of two search results, each displayed in a blue box with white text. The results are:

- Patricia Böhm  
0941 943 5099
- Victoria Böhm  
0941 943 5099

# DATEIEN SPEICHERN ... UND WIEDERFINDEN IN EINEM TELEFONBUCH (1/2)

```
1 public class Entry {
2
3     private String name;
4     private String number;
5
6     public Entry(String name, String number) {
7         this.name = name;
8         this.areaCode = areaCode;
9         this.number = number;
10    }
11
12    public String getName() {
13        return name;
14    }
15
16    public String getNumber() {
17        return number;
18    }
19
20 }
```

## DATEIEN SPEICHERN ... UND WIEDERFINDEN IN EINEM TELEFONBUCH (2/2)

```
1 ArrayList<Entry> phonebook = new ArrayList<Entry>();
2
3 public Entry getEntryByName(String name) {
4     for(Entry entry: phonebook) {
5         if(entry.name.equals(name)) {
6             return entry;
7         }
8     }
9     return null;
10 }
```

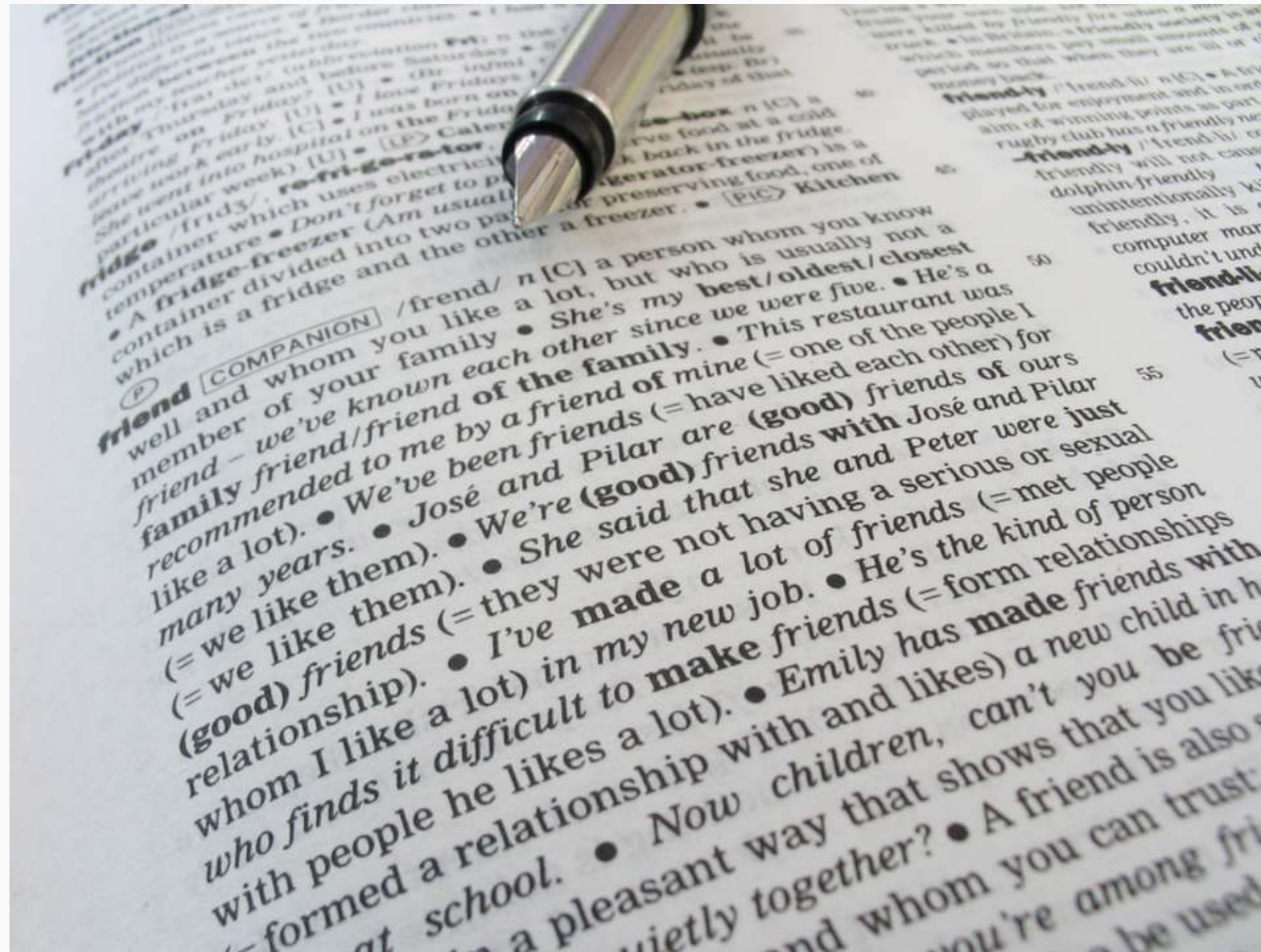
Geht das besser?



# DATENSTRUKTUREN MIT SCHLÜSSELN: TELEFONBÜCHER



# DATENSTRUKTUREN MIT SCHLÜSSELN: WÖRTERBÜCHER



In einem Wörterbuch werden Definitionen zu Wörtern zugeordnet.

# MAPS IN JAVA

*Map* ist ein Interface, dass Methoden für Datenstrukturen mit besonderen Aufgaben vorgibt:

- Speichern von mehreren Werten (Objekten) als einzelne `value`
- Zuordnung eines `key` zu jedem `value`
- Ermöglichen des Zugriffs auf gespeicherter Werte über den `key`
- Schlüssel und Werte müssen Objekte sein!



## MAPS IN JAVA: WÖRTERBUCH

Die Implementierung eines Wörterbuch kann in Java einfach über das Map-Interface oder eine entsprechende, vorgegebene Datenstruktur erfolgen:

- Als key werden die einzelnen Wörter, z.B. als Strings verwendet.
- Den value stellen die Definition des Worts, z.B. als String, Liste von Strings oder komplexe Objekte dar.
- Durch die Verknüpfung von Wort und Definition kann bei bekanntem Schlüssel einfach nach der zugehörigen Definition gesucht werden.

## ABER: MAP IST "NUR" EIN INTERFACE

Irgendeine Klasse muss das Map-Interface implementieren, um die gewünschte Funktionalität bereitzustellen. Ein Beispiel dafür ist die HashMap-Klasse:

- Die Klasse HashMap implementiert Map.
- Die *HashMap* ist generisch formuliert (Vgl. `ArrayList`) d.h. die Datentypen für Wert und Schlüssel (!) müssen bei Objekterzeugung angegeben werden: Ein Typ für `key`, ein Typ für `value`.
- *HashMaps* sind schnell: Zeiten für das Ablegen und Auslesen von Elementen sind konstant niedrig (unabhängig von der Größe der *HashMap*).
- *HashMaps* wissen nichts über die Reihenfolge ihres Inhalts.

## HASHMAPS: ERSTELLEN

```
1 HashMap<String, String> dict = new HashMap<String, String>();  
2  
3 HashMap<String, Entry> phonebook = new HashMap<String, Entry>();
```

**Hinweis:** HashMaps sind Objekte, die über einen Konstruktor erzeugt werden und über ein Set an Methoden verwendet werden!



# HASHMAPS: DATEN EINFÜGEN

```
1 dict.put("Regensburg", "Eine Stadt in der Oberpfalz.")
2
3 Entry entry = new Entry("Alexander Bazo", "1234");
4 phonebook.put(entry.getName(), entry);
5
6 // Achtung: Das ist kein gutes Beispiel, da wir auch einfach die Nummer im
7 // bereits gespeicherten Eintrag ändern könnten ;)
8 Entry newEntry = new Entry("Alexander Bazo", "4958");
9 // Gibt den ursprünglich assoziierten Wert zurück
10 Entry oldEntry = phonebook.put(newEntry.getName(), newEntry);
```

**Hinweis:** Die Methode `put` speichert den übergebenen Wert (zweiter Parameter) in der *Map* und ordnet ihm den übergebenen Schlüssel (erster Parameter) zu. Falls der Schlüssel bereits einem anderen Wert zugeordnet war, wird dieser überschrieben und die Methode gibt den ursprünglich assoziierten Wert zurück.

## HASHMAPS: DATEN AUSLESEN

```
1 String infoRegensburg = dict.get("Regensburg");  
2  
3 Entry entry = phonebook.get("Alexander Bazo");
```

**Hinweis:** Die Methode `get` liest den Wert aus der *Map* aus, der mit dem übergebenen Schlüssel assoziiert ist. Falls kein entsprechender Eintrag gefunden wird, wird `null` zurückgegeben. Die Rückgabe entspricht dem bei Initialisierung festgelegtem Wert.

## HASHMAPS: WEITERE METHODEN

Daten entfernen	<code>map.remove(key)</code>
Alle Daten entfernen	<code>map.clear()</code>
Alle Daten auslesen	<code>map.values</code> (Gibt eine <i>Collections</i> aller Werte(!) zurück)
Vorhandensein von Schlüsseln oder Werten prüfen	<code>map.containsKey(key)</code> bzw. <code>map.containsValue(value)</code>
Länge bestimmen	<code>map.size()</code>

## CAVEAT: DER UMGANG MIT HASHMAPS (1/2)

HashMaps erlauben den schnellen Zugriff auf die gespeicherten Werte, wenn der jeweilige Schlüssel bekannt ist. Sie eignen sich gut für *look up*-Operationen. Ein systematischer Zugriff auf alle Daten z. B. in sortierter Reihenfolge ist nicht wirklich möglich.

- Verwenden Sie ausschließlich *Immutableables* als Schlüssel
- Vermeiden Sie es, Schlüssel-Objekte zu "verlieren"

## CAVEAT: DER UMGANG MIT HASHMAPS (2/2)

- Überschreiben Sie bei selbst-erstellten Schlüssel-Klassen unbedingt die hashCode- und equals-Funktion, um vollständige Kontrolle über die Verwendung als Schlüssel zu erhalten
- Wenn hashCode nicht überschrieben wird, wird die *Default*-Implementierung verwendet: In der Regel wird dabei die Speicheradresse ge-hasht
- Strings können dagegen unproblematisch als Schlüssel verwendet werden

# COLLECTIONS

## DAS JAVA COLLECTIONS FRAMEWORK (1/2)

*A collections framework is a unified architecture for representing and manipulating collections. All collections frameworks contain the following.*

**Interfaces:** *These are abstract data types that represent collections. Interfaces allow collections to be manipulated independently of the details of their representation. In object-oriented languages, interfaces generally form a hierarchy.*

## DAS JAVA COLLECTIONS FRAMEWORK (2/2)

**Implementations:** *These are the concrete implementations of the collection interfaces. In essence, they are reusable data structures.*

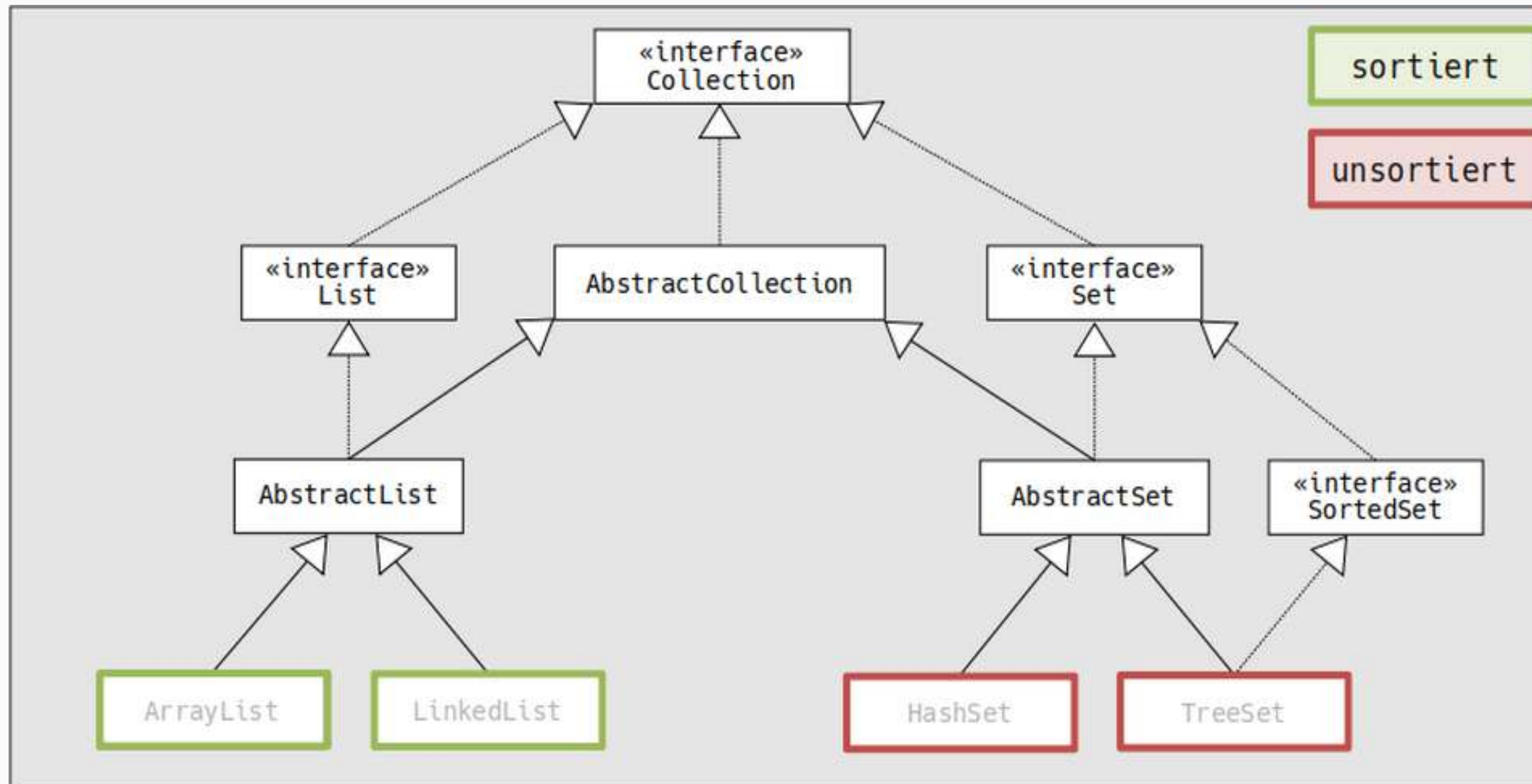
**Algorithms:** *These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces. The algorithms are said to be polymorphic: that is, the same method can be used on many different implementations of the appropriate collection interface. In essence, algorithms are reusable functionality.*

Quelle:

<http://docs.oracle.com/javase/tutorial/collections/intro/index.html>



# HIERARCHIEN IM COLLECTION-FRAMEWORK



Alle Klassen, die eines der dargestellten Interfaces implementieren, sind **Collections**.

## METHODEN DES COLLECTION-FRAMEWORKS

<code>boolean add(E e)</code>	<i>Appends the specified element to the collection. Returns true if this collection changed as a result of the call.</i>
<code>boolean remove(E e)</code>	<i>Removes the first occurrence of the specified element from this list, if it is present; the value is true, if a match is found.</i>
<code>void clear()</code>	<i>Removes all values from the collection.</i>
<code>int size()</code>	<i>Returns the number of values in the collection.</i>
<code>boolean contains(Object o)</code>	<i>Returns true if this collection contains the specified element.</i>
<code>boolean isEmpty()</code>	<i>Returns true if this collection contains no elements.</i>
<code>Iterator iterator()</code>	<i>Returns an iterator that allows clients to step through the values in the collection.</i>

Quelle: [Oracle Java Dokumentation](#)

# ITERATION ÜBER COLLECTIONS

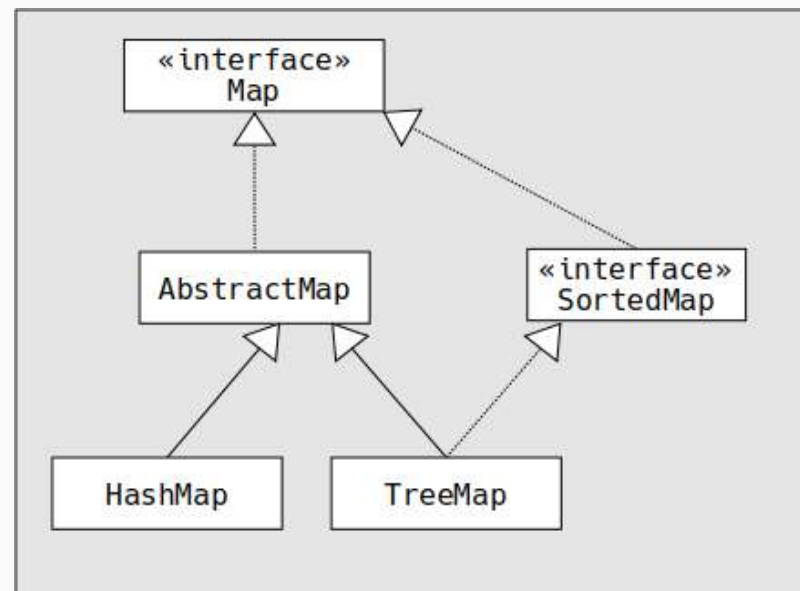
Iteratoren erlauben das schrittweise "Durchgehen" von (unsortierten) Elementmengen. Der konkrete Iterator wird von der Komponente bereitgestellt, die die Daten verwaltet (z.B. der *ArrayList* oder der *HashMap*)

- Der Iterator ist eine Alternative zur for-Schleife
- Alle *Collections* implementieren die Methode `iterator`
- Die `iterator`-Methode gibt den Datenbestand der jeweiligen *Collections* als iterierbare Elementliste zurück

# ITERATION ÜBER COLLECTIONS: ARRAYLIST

```
1 // Erstellen und Befüllen der Liste
2 ArrayList<String> nameList = new ArrayList<String>();
3 nameList.add("Bob");
4 nameList.add("Alice");
5
6 // Erzeugen des Iterators
7 Iterator<String> nameIterator = nameList.iterator();
8 // Verarbeitung aller Elemente, die vom Iterator zurückgegeben werden
9 while (nameIterator.hasNext()) {
10     System.out.println(nameIterator.next());
11 }
```

# MAPS SIND NICHT TEIL DES COLLECTION-FRAMEWORKS



*HashMaps* sind keine *Collections*, da das Interface *Collection* nicht implementiert wird. Der Sinn von einer *Interface*-basierten Gruppierung von Datenstrukturen zeigt sich hier noch deutlicher: Durch das einheitliche *Interface* sind beide Varianten austauschbar verwendbar. Der Unterschied liegt in der jeweiligen Implementierung der Datenspeicherung und des entsprechenden, internen Zugriffs.

## ITERATION ÜBER COLLECTIONS: KEYSETS VON HASHMAP




HashMaps sind nicht Teil der *Collections*-Hierarchie und erzeugen keinen Iterator für die gespeicherten Daten. HashMaps können aber ein Set mit allen ihrer *Keys* zurückgeben, das dann wiederum einen Iterator erzeugen kann (Sets sind Teil der Collections-Hierarchie!).

```
1 Set<String> nameSet = phoneBook.keySet();
2 Iterator<String> nameIterator = nameSet.iterator();
3 while (nameIterator.hasNext()) {
4     String name = nameIterator.next();
5     Entry entry = phonebook.get(name);
6     System.out.println("Name: " + entry.getName());
7     System.out.println("Number: " + entry.getNumber());
8 }
```

# ZUSAMMENFASSUNG

- Vorhersehbare, potenzielle Fehler werden in Java über Exceptions abgefangen.
- Kritische Stellen, solche an denen möglicherweise Exceptions erzeugt werden können, werden im Code durch `try ... catch`-Konstrukte gesichert.
- HashMaps speichern Schlüssel-Wert Paare, Werte können über einen Key aus einer *HashMap* abgefragt werden.
- Alle Klassen der *Collections*-Hierarchie (d.h. alle Klassen, die das Interface *Collection* implementieren) stellen einen Iterator zur Verfügung, mit dem sich alle Datensätze der *Collection* iterieren lassen.

## **VIELEN DANK FÜR IHRE AUFMERKSAMKEIT. WENN SIE MÖCHTEN, SEHEN WIR UNS IM ANSCHLUSS IN DER ZENTRALÜBUNG!**

Fragen oder Probleme? In allgemeinen Angelegenheiten und bei Fragen zur Vorlesung wenden Sie sich bitte an Alexander Bazo ([alexander.bazo@ur.de](mailto:alexander.bazo@ur.de) ). Bei organisatorischen Fragen zu den Studienleistungen und Übungsgruppen schreiben Sie bitte Florin Schwappach ([florin.schwappach@ur.de](mailto:florin.schwappach@ur.de) ). Bei inhaltlichen Fragen zu den Übungen, Beispielen und Studienleistungen schreiben Sie uns unter [mi.oop@mailman.uni-regensburg.de](mailto:mi.oop@mailman.uni-regensburg.de) .



## QUELLEN

Eric S. Roberts, *The art and science of Java: an introduction to computer science, New international Edition*, 1. Ausgabe, Pearson, Harlow, UK, 2014.