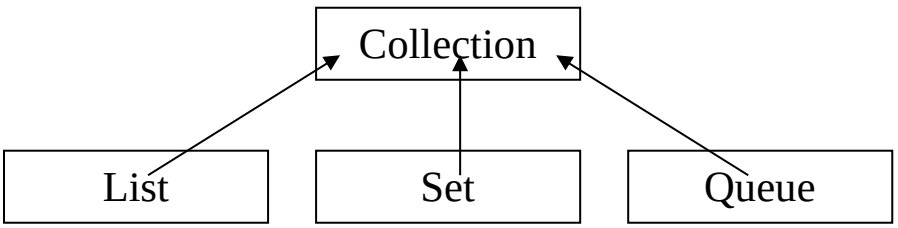
Collections-Framework (im Package java.util)

Das Collections-Framework beinhaltet Datenstrukturen, um Daten zusammenhängend zu speichern. Im Unterschied zu einem Array passt sich die Größe dieser Datenstrukturen der Anzahl der gespeicherten Daten an.

Es gibt unterschiedliche Datenstrukturen für unterschiedliche Anwendungsfälle. Die wichtigsten sind nachfolgend:→ Liste (List), Menge (Set), Schlange (Queue)

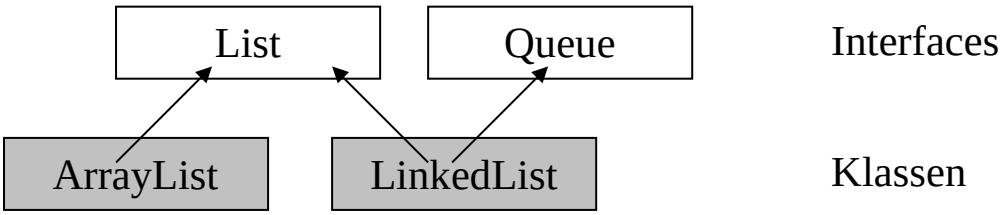
List	erlaubt Duplikate (Objekte gleichen Inhalts) kann sortiert sein Zugriff sequentiell oder wahlweise
Set	Duplikate sind verboten Menge im mathematischen Sinn
Queue	Liste, Zugriff nach dem FIFO-Prinzip (First In, First Out) Zugriff nur sequentiell, nicht wahlweise

Es gibt verschiedene Interfaces mit folgender Hierarchie:



Die Interfaces *List*, *Set* und *Queue* sind vom Interface *Collection* abgeleitet.

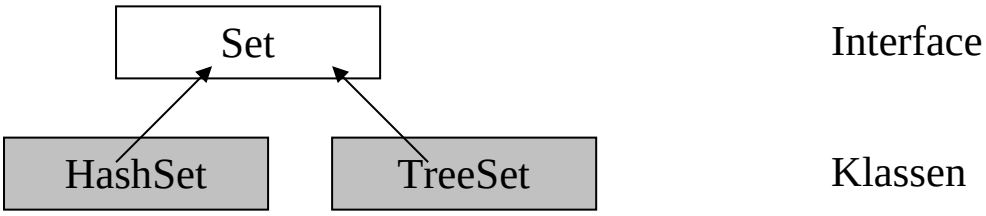
Es gibt zwei unterschiedliche Listen: ArrayList und LinkedList.



Eine ArrayList ist ein dynamisches Array, dessen Größe sich an die Anzahl der gespeicherten Elemente anpasst. Die Klasse *ArrayList* implementiert nur das Interface *List*.

Eine LinkedList ist eine doppelt verkettete Liste. Die Klasse *LinkedList* implementiert sowohl das Interface *List* als auch das Interface *Queue*.

Es gibt auch zwei unterschiedliche Sets: HashSet und TreeSet.



Auf die Vor- und Nachteile der Klassen *ArrayList*, *LinkedList*, *HashSet* und *TreeSet* und wie man damit arbeiten kann kommen wir später zu sprechen.

Generische Datentypen

In Java gibt es die Möglichkeit, generische Datentypen (generische Klassen) zu verwenden.

Ein generischer Datentyp ist allgemein: Collection<E>, was besagt, dass in dieser speziellen Collection nur Objekte vom Datentyp E gespeichert werden dürfen.

Beispiel: ArrayList

```
ArrayList<String> arrList;
arrList = new ArrayList<String>();

for (int i = 1; i <= 10; i++) {
        arrList.add("Obj" + i);
        }

arrList.add(new Integer(12));       // Fehlermeldung des Compilers
```

Mittels generischer Datentypen schreiben wir im obigen Beispiel vor, dass in der ArrayList *arrList* nur Objekte vom Datentyp *String* gespeichert werden dürfen. Im obigen Beispiel ist *add* eine Methode der Klasse *ArrayList* und fügt der Liste ein Objekt vom Datentyp *String* zu. Dazu später mehr.

Ich kann nur dringend empfehlen, generische Datentypen zu verwenden. Das Speichern von Objekten unterschiedlichen Datentyps in einer Collection kann zu größeren Problemen führen.

Das Interface Collection

Das Interface *Collection* beinhaltet Basisfunktionalitäten, die allen Datenstrukturen (allen Collections) zur Verfügung stehen. Wichtige Methoden sind:

boolean add(E o)	→ fügt Objekt o vom Datentyp E zu → true, wenn Collection geändert
void clear()	→ löscht alle Elemente der Collection

boolean contains(Object o)	→ true, wenn o in der Collection vorhanden ist → die Klasse von o muss *equals* implementiert haben
int hashCode()	→ Hashcode der Collection
boolean isEmpty()	→ true, wenn die Collection leer ist
Iterator<E> iterator()	→ Iterator-Objekt (später mehr dazu)
boolean remove(Object o)	→ entfernt o, wenn vorhanden
int size()	→ Anzahl der Elemente in der Collection
Object[] toArray()	→ Array mit allen Objekten der Collection

Kleine Exkursion zur Arbeit mit dem Application Programming Interface (API)

Das API ist eine Sammlung von vielen Java-Klassen, die in unterschiedlichen Packages organisiert sind und von Java zur Verfügung gestellt werden.

Nehmen wir mal an, wir wollen wissen, welche Methoden im Interface *Collection* vorhanden sind.

1. Wir geben "Java API" in Google ein
2. Wir drücken den Link "Java Platform, Standard Edition 7 API …"
3. Wir scrollen in der Box "All Classes", bis wir *Collection* finden
4. Wir drücken auf *Collection*
5. Im rechten Fenster erhalten wir eine Beschreibung des Interfaces *Collection*. Unter "Method Summary" finden wir eine Liste mit den vorhandenen Methoden.
6. Falls uns eine Methode besonders interessiert, clicken wir den Name der Methode an und erhalten Details zu der Methode. Zum Beispiel ist hier beschrieben, dass die Methode *contains(Object o)* die equals-Methode von o benutzt.

Wenn Sie mit einer Java-Klasse arbeiten (z.B. mit *String* oder *ArrayList*), schauen Sie bitte ins API, welche Methoden diese Klasse zur Verfügung stellt. Sie sparen sich eventuell eine Menge eigenen Code. Im Falle von *ArrayList* sollten Sie auch in den Interfaces *List* und *Collection* nachschauen.

Wenn Sie nicht ganz sicher sind, was eine Methode einer von Java zur Verfügung gestellten Klasse macht, schauen Sie bitte ins API. Es kam in der Vergangenheit öfter vor, dass Studierende die Methode *contains* benutzt haben, ohne sicherzustellen, dass die equals-Methode sinnvoll vorhanden war.

Listen

Das Interface List

Das Interface *List* stellt weitere Methoden für die abgeleiteten Klassen *ArrayList* und *LinkedList* zur Verfügung. Einige wichtige sind:

void add (int index, E element)
→ Hinzufügen von element an der Position index

E get(int index)
→ Element an der Position index

int indexOf(Object o)
→ Index des ersten Vorkommens
→ nutzt die equals-Methode
→ nicht vorhanden: Rückgabewert = -1

E set(int index, E element)
→ Ersetzen des Elementes an index durch element
→ Rückgabewert: ersetztes Element

Arbeiten mit Listen

Wie schon weiter oben beschrieben, ist eine *ArrayList* ein dynamisches Array, das seine Größe automatisch an die Anzahl der in der *ArrayList* gespeicherten Objekte anpasst.

Beispiel:

```
import java.util.*;

public class Iteration {
	public static void main(String[] args) {
		ArrayList<String> arrList;
		arrList = new ArrayList<String>();
		for (int i = 1; i <= 10; i++) {
			arrList.add("Obj" + i);
			}

		// Liste durchlaufen und jedes 2. Objekt löschen

		if (arrList.contains("Obj8")) {
			System.out.println("Objekt 8 enthalten");
			}
		}
	}
```

Frage: Wozu brauche ich neben der *ArrayList* noch eine *LinkedList*?

Antwort: Wenn man oft Objekte in die Liste einfügen oder aus der Liste löschen muss.

Der Grund ist, dass eine Einfüge- oder Lösch-Operation bei einer *ArrayList* recht zeitaufwendig ist. Angenommen, wir haben eine Liste mit 14 Elementen und wollen hinter dem zweiten Element ein neues Objekt einfügen.

Hinter den Kulissen passiert dann folgendes:

Alle Elemente der Liste hinter dem zweiten Element werden um eine Stelle nach rechts verschoben und ein neues Element eingefügt.

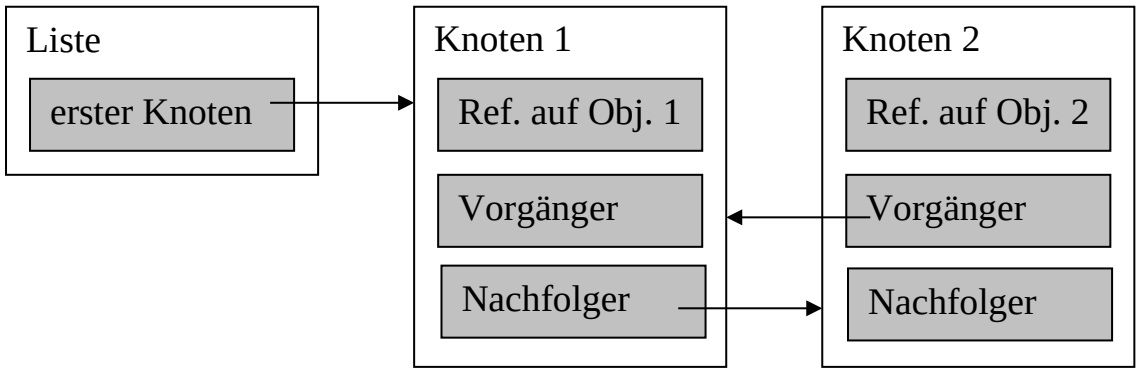
Danach wird das einzufügende Objekt im neuen Listenelement gespeichert.

Bei größeren Listen kann dieser Prozess aufgrund der zahlreichen Verschiebe-Operationen viel Zeit in Anspruch nehmen.

Soll ein Element aus der Liste gelöscht werden, haben wir mitunter viele Verschiebe-Operationen nach links.

Wenn wir also damit rechnen müssen, dass oft neue Objekte in der Liste gespeichert oder Objekte aus der Liste gelöscht werden, sollten wir anstelle einer *ArrayList* eine *LinkedList* verwenden.

Im Unterschied zur *ArrayList*, die ein dynamisches Array ist, ist eine *LinkedList* eine zweifach verkettete Liste.

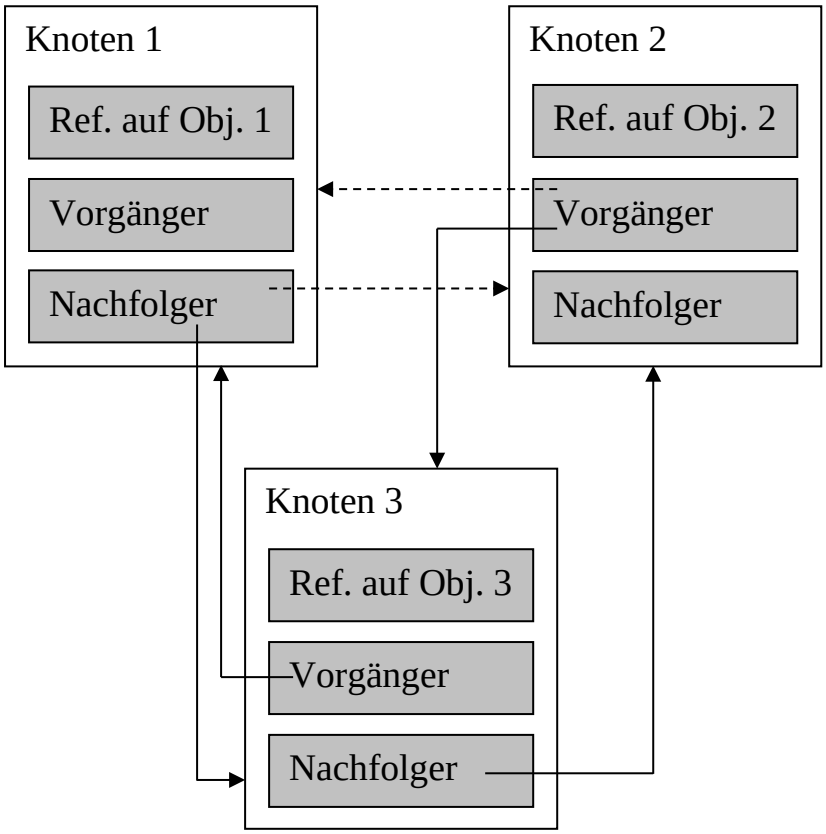


In einer Liste vom Typ LinkedList werden Objekte in sogenannten Knoten gespeichert. Neben der Referenz auf das gespeicherte Objekt enthält ein Knoten noch einen Verweis auf den Vorgängerknoten und einen Verweis auf den Nachfolgerknoten (nur der Startknoten der Liste beinhaltet keine Objektreferenz und

keinen Verweis auf einen Vorgängerknoten, sondern nur einen Verweis auf den Knoten 1 der Liste).

Wenn jetzt ein neues Objekt in der Liste eingefügt oder aus der Liste gelöscht werden soll, müssen jetzt keine Elemente mehr verschoben, sondern nur Verweise geändert werden, was deutlich schneller geht.

Die folgende Grafik veranschaulicht dies für den Einfüge-Prozess. Zwischen Objekt 1 und Objekt 2 soll ein Objekt 3 eingefügt werden.



Die gestrichelten Verweise spiegeln den Zustand vor dem Einfügen wider und existieren jetzt nicht mehr. Insbesondere bei Listen mit vielen Elementen und vielen Einfüge- und Löschoperationen sollte man eine *LinkedList* einer *ArrayList* vorziehen. Spielen Einfüge- und Löschoperationen eine eher geringe Rolle, ist wegen der einfacheren inneren Struktur und des damit verbundenen geringeren Verwaltungsaufwandes eine *ArrayList* vorzuziehen.

In der softwaretechnischen Umsetzung macht es keinen Unterschied, ob man eine *ArrayList* oder eine *LinkedList* benutzt, wie das folgende vergleichende Beispiel zeigt.

ArrayList

```
import java.util.*;

public class Iteration {
	public static void main(String[] args) {
		ArrayList<String> arrList;
		arrList = new ArrayList<String>();
		for (int i = 1; i <= 10; i++) {
			arrList.add("Obj" + i);
			}

		// Liste durchlaufen und jedes 2. Objekt löschen

		if (arrList.contains("Obj8")) {
			System.out.println("Objekt 8 enthalten");
			}
		}
	}
```

LinkedList

```
import java.util.*;

public class Iteration {
	public static void main(String[] args) {
		LinkedList<String> liList;
		liList = new LinkedList<String>();
		for (int i = 1; i <= 10; i++) {
			liList.add("Obj" + i);
			}

		// Liste durchlaufen und jedes 2. Objekt löschen

		if (liList.contains("Obj8")) {
			System.out.println("Objekt 8 enthalten");
			}
		}
	}
```

Listen sequenziell durchlaufen

Es gibt sowohl in einer ArrayList als auch in einer LinkedList eine komfortable Möglichkeit, sequentiell durch die Liste zu laufen.

Sowohl in der Klasse *ArrayList* als auch in der Klasse *LinkedList* gibt es eine Methode *iterator()*, die als Ergebniswert ein Objekt der Klasse *Iterator* zurückgibt.

```
import java.util.*;

public class Iteration {
	public static void main(String[] args) {
		ArrayList<String> arrList;
		arrList = new ArrayList<String>();
		for (int i = 1; i <= 10; i++) {
			arrList.add("Obj" + i);
			}

		Iterator<String> iter = arrList.iterator();

		// Liste durchlaufen und jedes 2. Objekt löschen

		if (arrList.contains("Obj8")) {
			System.out.println("Objekt 8 enthalten");
			}
		}
	}
```

Mittels Methoden des Iterator-Objektes kann die Liste durchlaufen und z.B. auch Objekte gelöscht werden. Direkt nach Aufruf der Methode iterator() befindet man sich am Anfang der Liste. Man kann sich das so vorstellen, als stünde ein gedachter Positionszeiger vor dem ersten Listenelement.

Liste durchlaufen und jedes 2. Objekt löschen:

```
int x = 0;
String s;
while (iter.hasNext()) {
	s = iter.next();
	if ((x%2) == 0) {
		iter.remove();
		x++;
		}
	}
```

Die while-Schleife wird solange durchlaufen, solange es noch ein nächstes Listenelement gibt. Ob es noch ein nächstes Listenelement gibt, wird mit iter.hasNext() geprüft. Wenn es ein nächstes Listenelement gibt, holt man sich den Inhalt dieses Elementes mit iter.next(). Zu Beginn holt man sich natürlich den Inhalt des ersten Listenelementes, weil der gedachte Positionszeiger vor dem ersten

Listenelement steht. Nachdem man sich den Inhalt des Elementes geholt hat, springt der gedachte Positionszeiger um eine Stelle weiter.

Der Ergebniswert von *iter.next()* ist ein Objekt vom Datentyp *String*. In diesem kleinen Beispiel wäre es ausreichend gewesen, wenn wir statt *s = iter.next()* auch nur *iter.next()* geschrieben hätten, weil wir mit dem String selbst nichts tun. Mit *if ((x%2) == 0)* überprüfen wir, ob x geradzahlig oder ungeradzahlig ist. Wenn x eine gerade Zahl ist, löschen wir das Element mit *iter.remove()*. *iter.remove()* löscht das Element in der Liste, das zuletzt mit *iter.next()* aus der Liste geholt wurde.

Nachfolgend das komplette Beispiel:

```
import java.util.*;

public class Iteration {
	public static void main(String[] args) {
		ArrayList<String> arrList;
		arrList = new ArrayList<String>();
		for (int i = 1; i <= 10; i++) {
			arrList.add("Obj" + i);
			}

		Iterator<String> iter = arrList.iterator();

		int x = 0;
		while (iter.hasNext()) {
			iter.next();
			if ((x%2) == 0) {
				iter.remove();
				x++;
				}
			}
		if (arrList.contains("Obj8")) {
			System.out.println("Objekt 8 enthalten");
			}
		iter = arrList.iterator();  //Positionszeiger wieder auf Anfang
		}
	}
```

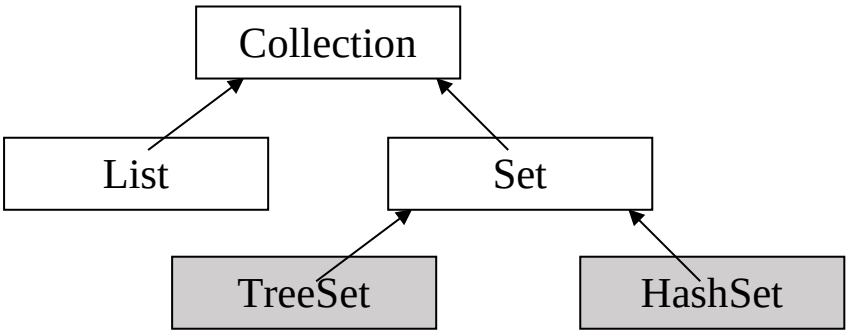
Methoden der Klasse Iterator:

boolean hasNext()
→ prüft, ob ein weiteres Element vorhanden ist

E next()
→ holt das aktuelle Element und setzt den Positionszeiger vor das nächste Element
→ kein nächstes Element
⇒ NoSuchElementException

void remove
→ entfernt das Element, das beim letzten Aufruf von next() zurück gegeben wurde, aus der Liste

Sets

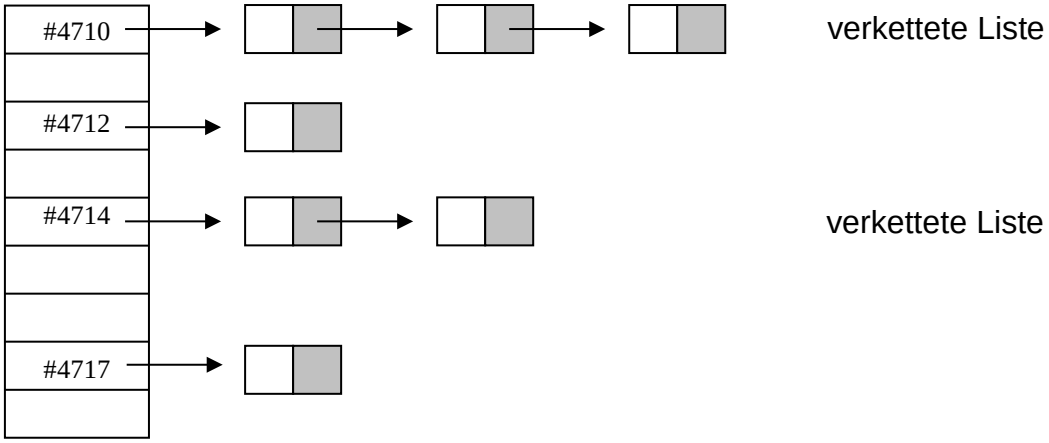


Sets sind Mengen im mathematischen Sinn

- Sets dürfen keine doppelten Elemente enthalten
- Reihenfolge der Elemente für Iterator kann anders sein als die des Sets

HashSet (Hash-Tabellen)

Ein HashSet ist eine Tabelle, in der man indexbasiert über den Hashcode eines Objektes sehr schnell auf das zugehörige Objekt zugreifen kann. Haben mehrere unterschiedliche Objekte denselben Hashcode (man spricht von Kollisionen), werden diese Objekte über eine verkettete Liste miteinander verknüpft.



Dass unterschiedliche Objekte zufällig denselben Hashcode haben können, lässt sich an einem einfachen Beispiel schnell erkennen.

> Eine Klasse Rechteck soll nur die Attribute Laenge und Breite haben. Ein Objekt R1 hat eine Laenge von 9 und eine Breite von 5. Ein Objekt 2 hat eine Laenge von 5 und eine Breite von 9. Beide Objekte sind im Sinne der equals-Methode verschieden, haben aber denselben Hashcode.

Wenn mehrere Objekte denselben Hashcode haben, macht dies den Zugriff auf ein Objekt natürlich langsamer, da zusätzlich zum schnellen Zugriff über den Hashcode noch die verkettete Liste durchsucht werden muss.

Für eine sinnvolle Implementierung der Methode zur Berechnung des Hashcodes eines Objektes gelten folgende Anforderungen:

1. Die Berechnung des Hashcodes sollte schnell gehen. Andernfalls würde schon dieser Schritt den Zugriff auf ein Objekt in der Hashtabelle verzögern (um auf ein Objekt über den Hashcode zuzugreifen, muss dieser erst berechnet werden).

2. Der Algorithmus zur Berechnung des Hashcodes sollte möglichst wenig Kollisionen erzeugen, wobei die Hashtabelle trotzdem nicht zu viel Speicherplatz benötigen sollte.

3. Die Methode zur Berechnung des Hashcodes muss konsistent zur equals-Methode sein. Sind zwei Objekte im Sinne der equals-Methode gleich, müssen sie denselben Hashcode haben.

Die Anleitung zur Implementierung der Methode *hashCode()* entspricht diesen Anforderungen (siehe dazu den Beispielinhalt "Die Superklasse Object").

Die Vorgehensweise zur Nutzung von HashSet ist die folgende:

1. Ein HashSet-Objekt über den Konstruktor *HashSet(int initialCapacity, float loadFactor)* erzeugen. *initialCapacity* sollte etwa 1,5 mal so groß sein wie die erwartete Anzahl der zu speichernden Objekt und sollte eine Primzahl sein. *loadFactor* sollte auf den Wert 0,75 gesetzt werden. Dies bedeutet, dass, falls die Hashtabelle zu 75 % gefüllt ist, sie automatisch in eine doppelt so große Tabelle umgespeichert wird.

2. In eigenen Klassen, deren Objekte in einem HashSet gespeichert werden sollen, sollte die Methode *hashCode()* überschrieben werden (und dann natürlich auch die equals-Methode).

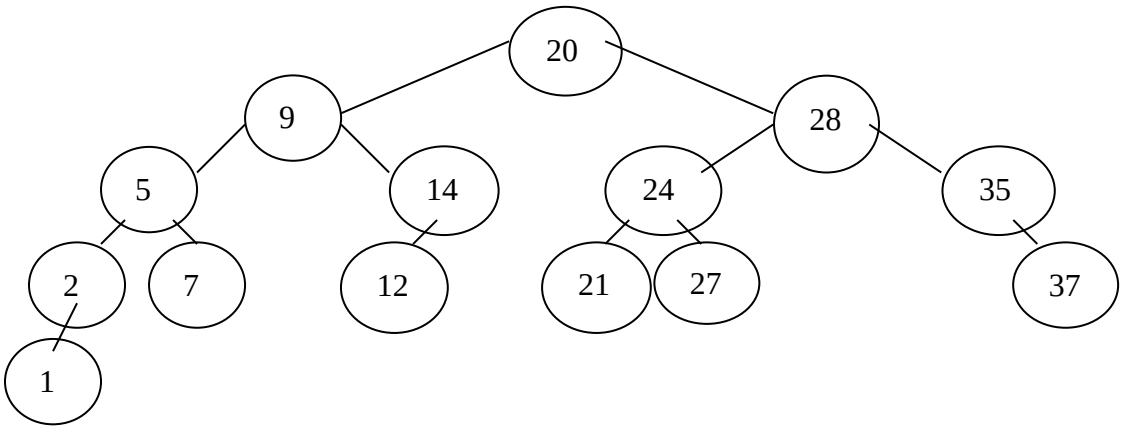
3. Objekte speichern und/oder wieder darauf zugreifen.

Beispiel:

```
class Person {
        private String Nachname;
        private String Vorname;
        . . .
        public int hashCode() {
                int hc = 17;
                int mult = 59;
                return hc + mult*getNachname().hashCode() +
                                mult*getVorname().hashCode();
                }
        // equals-Methode
        . . .
        }

class HashPerson {
        public static void main(String[] args) {
                HashSet<Person> personSet;
                personSet = new HashSet<Person>(193, 0.75);

                Person personA;
                personA = new Person("Duck", "Donald");
                personSet.add(personA);

                Person personB;
                personB = new Person("Mouse", "Mickey");
                personSet.add(personB);
                . . .

                Iterator iter = personSet.iterator();
                while (iter.hasNext()) {
                        System.out.println("Hashcode: " + iter.next().hashCode());
                        }
                . . .
                }
        }
```

Bemerkung: Den Iterator gibt es auch für Sets.

TreeSet (Bäume)



Ein TreeSet ist eine Datenstruktur mit folgenden Eigenschaften:

- Die Objekte werden in einem binären Baum gespeichert
- etwas langsamer als HashSet, aber schneller als ArrayList bzw. LinkedList
- Elemente liegen sortiert vor
- jeder Elternknoten hat höchstens 2 Kindknoten
  (binärer Baum)
- linker Kindknoten < Elternknoten
- rechter Kindknoten > Elternknoten

Daten suchen oder hinzufügen $\Rightarrow$ wenige Vergleiche (passiert hinter den Kulissen)

Beispiel: Element 8 einfügen

a) $8 < 20 \Rightarrow$ linker Kindknoten
b) $8 < 9 \Rightarrow$ linker Kindknoten
c) $8 > 5 \Rightarrow$ rechter Kindknoten
d) $8 > 7 \Rightarrow$ rechter Kindknoten (einfügen)

Insbesondere bei selbst geschriebenen Klassen muss die Java-Laufzeitumgebung wissen, wie und wonach sie auf > oder < vergleichen soll. Wenn man zum Beispiel Rechtecke in einem TreeSet speichern will, muss die Laufzeitumgebung wissen, welche Informationen sie für den Vergleich nutzen soll (Breite oder Länge oder Fläche oder Umfang oder was auch immer). Dies teilt man der Laufzeitumgebung mit, indem man in der Klasse Rechteck das generische Interface *Comparable* implementiert und anschließend die Methode *compareTo* überschreibt.

```
public class Rechteck implements Comparable<Rechteck> {
        private int Breite;
        private int Laenge;
        . . .
        public int compareTo(Rechteck r) {
                return this.getFlaeche().compareTo(r.getFlaeche());
                }
        . . .
        }
```

Die Laufzeitumgebung von Java würde in diesem Fall z.B. beim Einfügen eines neuen Rechtecks die notwendigen Vergleiche mit den vorhandenen Rechtecken, die im TreeSet gespeichert sind, mittels der Fläche durchführen.

Ein weiteres Beispiel:

```
public class Bier implements Comparable<Bier> {
        private String Name;
        private String Herkunft;
        private float Inhalt;
        . . .

        public int compareTo(Bier b) {
                // checken,ob this.getName() != null und b.getName() != null
                return this.getName().compareTo(b.getName());
                }
        . . .
        }
```

Hinweis:

Will man ein Array von Objekten sortieren, kann man dies mit der Methode *java.util.Arrays.sort(Object[] a)* tun. Die Methode *sort* gibt es auch für Arrays von einfachen Datentypen wie *int* oder *double*.

Die Methode sort verwendet den Quicksort-Algorithmus und ist daher sehr schnell. Auch diese Methode muss wissen, wonach sie sortieren soll, was größer und was kleiner ist. Ein Array von Objekten einer eigenen Klasse lässt sich daher nur sortieren, wenn diese eigene Klasse das generische Interface *Comparable* implementiert und die Methode *compareTo* überschreibt.

Weitere Methoden von TreeSet (zusätzlich zu Collection bzw. Set)

E first()
→ liefert das erste (kleinste) Element des TreeSets

E last()
→ liefert das letzte (größte) Element des TreeSets

SortedSet<E> headSet(E toElement)
→ liefert den Teil des TreeSet, dessen Elemente kleiner sind als toElement

SortedSet<E> tailSet(E fromElement)
→ liefert den Teil des TreeSet, dessen Elemente größer sind als fromElement

SortedSet<E> subSet(E fromElement, E toElement)
→ liefert den Teil des TreeSet zwischen fromElement und toElement