Informatik B

Vorlesung 11
Collection Framework

Rückblick

- **`Set`**
- **`Map`**
- **`List`**
- Foreach-Schleife
- Typsicherheit (Generics)

Das Interface Collection

- Das Interface **`Collection`** ist eines der Basisinterfaces im Collection Framework
- Das Interface erweitert das Interface **`Iterable`**: **`interface java.util.Collection<E> extends Iterable`**
- In dem Interface werden Methodenköpfe zum Einfügen, Löschen und Manipulieren einer Collection vorgegeben
- Auffallend ist die Verwendung von generischen Typen

Methodenübersicht

- **`boolean add(E o)`** (optional) Fügt der Collection ein Element hinzu und liefert **`true`**, falls das Element eingefügt werden konnte, sonst **`false`**
- **`boolean addAll(Collection<? extends E> c)`** Fügt die Elemente der Collection **`c`** der Collection hinzu
- **`void clear()`** (optional) Löscht alle Elemente der Collection
- **`boolean contains(Object o)`** **`true`**, falls die Collection ein inhaltlich gleiches Element enthält
- **`boolean containsAll(Collection<?> c)`** **`true`**, falls die Collection alle Elemente der Collection **`c`** enthält
- **`boolean isEmpty()`** **`true`**, falls die Collection keine Elemente enthält
- **`Iterator<E> iterator()`** Liefert ein typsicheren **`Iterator`**

Methodenübersicht

- **`boolean remove(Object o)`** (optional) Entfernt das angegebene Objekt aus der Collection
- **`boolean removeAll(Collection<?> c)`** (optional) Entfernt alle Objekte der Collection **`c`** aus der Collection
- **`boolean retainAll(Collection<?> c)`** (optional) Entfernt alle Objekte, die nicht in der Collection **`c`** vorkommen
- **`int size()`** Gibt die Anzahl der Elemente zurück
- **`Object[] toArray()`** Gibt ein Array mit allen Elementen zurück
- **`<T> T[] toArray(T[] a)`** Gibt ein Array mit allen Elementen zurück und verwendet das als Argument übergebene Array als Zielcontainer, wenn es groß genug ist, ansonsten wird ein Array passender Größe angelegt, dessen Laufzeittyp a entspricht
- **`boolean equals(Object o)`** Prüft, ob das angegebene Objekt ebenfalls ein Container ist und die gleichen Elemente enthält wie dieser Container
- **`int hashCode()`** Liefert den Hash-Wert der Collection, bei Änderungen am Inhalt der Collection wird der Hash-Wert geändert, da er von allen Elementen der Collection abhängt

`Set`

- Ein **`Set`** ist eine (im Prinzip) ungeordnete Sammlung von Elementen
- Jedes Element darf nur einmal vorkommen
- Für Mengen sieht die Java-Bibliothek die Schnittstelle **`java.util.Set`** vor
- Implementierende Klassen sind unter anderem:
  - **`HashSet`**: Schnelle Mengenimplementierung durch Hashing-Verfahren (Dahinter steckt eine **`HashMap`**)
  - **`TreeSet`**: Mengen werden durch balancierte Binärbäume realisiert, die eine Sortierung ermöglichen
  - **`LinkedHashSet`**: Schnelle Mengenimplementierung unter Beibehaltung der Einfügereihenfolge
  - **`EnumSet`**: Eine spezielle Menge ausschließlich für Enum-Objekte
  - **`CopyOnWriteArraySet`**: Schnelle Datenstruktur für viele lesende Operationen

HashSet

- Ein **`java.util.HashSet`** verwaltet die Elemente in einer schnellen hashbasierten Datenstruktur
- Dadurch sind Elemente schnell einsortiert und schnell zu finden
- Falls eine Sortierung vom **`HashSet`** nötig ist, müssen die Elemente nachträglich umkopiert und dann sortiert werden
- **`HashSet()`** Erzeugt ein neues **`HashSet`**-Objekt mit **`16`** freien Plätzen und einem Füllfaktor von **`0,75`**
- **`HashSet(Collection<? extends E> c)`** Erzeugt ein neues Set aus der Menge gegebener Elemente
- **`HashSet(int initialCapacity)`** Erzeugt ein neues **`HashSet`** mit einer gegebenen Anzahl freier Plätze und dem Füllfaktor von **`0,75`**
- **`HashSet(int initialCapacity, float loadFactor)`** Erzeugt ein neues leeres **`HashSet`** mit einer Startkapazität und einem gegebenen Füllfaktor
- Die Startgröße ist für die Performance wichtig
- Ist die Größe zu klein gewählt, muss die Datenstruktur bei neu hinzugefügten Elementen vergrößert werden

TreeSet

- Die Klasse **`java.util.TreeSet`** verfolgt eine andere Implementierungsstrategie als ein **`HashSet`**
- Ein **`TreeSet`** verwaltet die Elemente immer sortiert
- Intern werden die Elemente in einem balancierten Binärbaum gehalten
- Speichert **`TreeSet`** ein neues Element, so fügt **`TreeSet`** das Element automatisch sortiert in die Datenstruktur ein
- Das kostet etwas mehr Zeit als ein **`HashSet`**, dafür ist diese Sortierung dauerhaft
- Daher ist es auch nicht zeitaufwändig, alle Elemente geordnet auszugeben
- Die Suche nach einem einzigen Element ist aber etwas langsamer als im **`HashSet`**
- Der Begriff langsamer muss jedoch relativiert werden, da die Suche logarithmisch ist und daher nicht wirklich langsam
- Beim Einfügen und Löschen muss bei bestimmten Konstellationen eine Reorganisation des Baumes in Kauf genommen werden, was die Einfüge-/Löschzeit verschlechtert

TreeSet

- **`TreeSet()`**: Erzeugt ein neues leeres **`TreeSet`**
- **`TreeSet(Collection<? extends E> c)`**: Erzeugt ein neues **`TreeSet`** aus der gegebenen Collection
- **`TreeSet(Comparator<? super E> c):`** Erzeugt ein leeres **`TreeSet`** mit einem gegebenen **`Comparator`**, der für die Sortierung der internen Datenstruktur die Vergleiche übernimmt
  - Ein **`Comparator`** implementiert das Interface **`Comparator`**
  - Er vergleicht zwei Objekte der „Größe“ nach miteinander
- **`TreeSet(SortedSet<E> s)`** Erzeugt neues **`TreeSet`** und übernimmt alle Elemente von **`s`** und auch die Sortierung von **`s`**

`TreeSet`

- **`TreeSet`** implementiert **`SortedSet`** und damit die folgenden Methoden:
  - **`E first()`** Liefert das kleinste Element
  - **`E last()`** Liefert das größte Element
  - **`Comparator<? super E> comparator()`** Liefert den mit der Menge verbundenen **`Comparator`**
  - **`SortedSet<E> headSet(E toElement)`** Liefert eine Teilmenge von Elementen, die echt kleiner als **`toElement`** ist
  - **`SortedSet<E> tailSet(E fromElement)`** Liefert eine Teilmenge mit Elementen, die größer oder gleich **`fromElement`** sind
  - **`SortedSet<E> subSet(E fromElement, E toElement)`** Liefert eine Teilmenge im gewünschten Bereich

`TreeSet`

- Anders als **`HashSet`** liefert der **`Iterator`** beim **`TreeSet`** die Elemente aufsteigend sortiert
- Davon profitieren auch die beiden **`toArray`**-Methoden, denn sie nutzen den **`Iterator`**, und erzeugen daher ein sortiertes Array
- Durch die interne sortierte Speicherung gibt es zwei ganz wichtige Bedingungen:
  - Die Elemente müssen sich vergleichen lassen
  - Die Elemente müssen vom gleichen Typ sein
- Beispiel: **`treeset1`**

TreeSet

- Das **TreeSet** nutzt zur Einordnung einen externen **Comparator** bzw. die **compareTo()**-Methode, wenn die Elemente **Comparable** sind
- Gibt die Vergleichsmethode **0** zurück, so sind die Elemente gleich und gleiche Elemente sind in der Menge nicht erlaubt
- Beispiel: **Point**-Objekte
  - Gegeben seien zwei **Point**-Objekte (1,100) und (100,1)
  - Die **Point**-Objekte vergleichen sich anhand des Abstandes zum Ursprung
  - Laut **equals()** sind die Punkte nicht gleich
  - Der Abstand zum Ursprung ist gleich (compareTo())
  - Dies führt dazu, dass nur ein Punkt in das **TreeSet** kommt, da die Implementierung nicht auf **equals()** basiert
- Beispiel: **treeset2**

`LinkedHashSet`

- Ein **`LinkedHashSet`** vereint die Reihenfolgentreue einer Liste und die hohe Performance für Mengenoperationen vom **`HashSet`**
- Dabei bietet die Klasse keine Listen-Methoden wie **`first()`** oder **`get(index)`**, sondern ist eine Implementierung ausschließlich der Set-Schnittstelle, in der der **`Iterator`** die Elemente in der Einfüge-Reihenfolge liefert
- Beispiel: **`linkedhashset1`**

`List`

- Eine **`List`** ist eine sequentielle Abfolge von Elementen
- In einer **`List`** dürfen Objekte mehrfach vorkommen
- Die Klassen **`ArrayList`**, **`LinkedList`** und **`Vector`** implementieren das Interface **`List`**
- **`Vector`** ist im Gegensatz zu **`ArrayList`** und **`LinkedList`** synchronisiert und ist daher vor konkurrierenden Threadzugriffen geschützt
- Eine Liste gibt über **`iterator()`** einen speziellen **`ListEnumerator`** zurück

List

- Die Liste sieht nicht direkt vor mit Bereichen zu arbeiten
- Stattdessen gibt es die Möglichkeit eine Sicht auf eine eingeschränkte Subliste zu erhalten
- Die Methode **`subList(fromIndex, toIndex)`** liefert eine solche Subliste zurück
- Um einen Bereich zu löschen, kann man also mit einer View auf einer Liste arbeiten: **`subList(fromIndex, toIndex).clear()`**
- Die **`subList()-`**Technik deckt gleich noch ein paar andere Operationen ab, für die es keine speziellen Range-Varianten gibt, zum Beispiel **`indexOf(),`** also die Suche in einem Teil der Liste
- Beispiel: **`list1`**

List

- Achtung: Es ist in der Regel keine gute Idee eine Liste zu sich selbst hinzuzufügen
- ```
  List l = new ArrayList();
  l.add(42);
  l.add(l);
  System.out.println(l);
  ```
- Obiger Quellcode führte in früheren Javaversionen zu einem Stack-Overflow
- Mittlerweile ist das Problem zwar behoben, dennoch ist es meist nicht sinnvoll in einer Collection die Collection selbst einzufügen
- Werden generische Typen verwendet ist dies ohnehin meist nicht möglich
- Beispiel: **`list2`**
  ```

`ArrayList`

- Um ein **`ArrayList`**-Objekt zu erzeugen, existieren drei Konstruktoren:
  - **`ArrayList()`**
    - Eine leere Liste mit einer Anfangskapazität von zehn Elementen wird angelegt
    - Werden mehr als zehn Elemente eingefügt, muss die Liste sich vergrößern
  - **`ArrayList(int initialCapacity)`**
    - Eine Liste mit **`initialCapacity`** freien Elementen wird angelegt
  - **`ArrayList(Collection<? extends E> c)`**
    - Kopiert alle Elemente der Collection c in das neue **`ArrayList`**-Objekt

`ArrayList`

- Die Klasse **`ArrayList`** verwaltetet zwei Größen:
  - Zum einen die Anzahl der gespeicherten Elemente
  - Zum anderen die interne Größe des Felds
- Ist die Kapazität des Felds größer als die Anzahl der Elemente, können noch Elemente aufgenommen werden, ohne dass die Liste etwas unternehmen muss
- Die Anzahl der Elemente in der Liste liefert die Methode **`size()`**
- Die Liste vergrößert sich automatisch, falls mehr Elemente aufgenommen werden, als ursprünglich am Platz vorgesehen waren
- Diese Operation heißt Resizing
- Dabei spielt die Größe **`initialCapacity`** für effizientes Arbeiten eine wichtige Rolle. Sie sollte passend gewählt sein

`ArrayList`

- Wenn das Array zehn Elemente fasst, nun ein elftes eingefügt werden soll, muss das Laufzeitsystem einen neuen Speicherbereich reservieren und jedes Element des alten Felds in das neue kopieren
- Das kostet Zeit
- Schon aus diesem Grund sollte der Konstruktor **`ArrayList(int initialCapacity)`** gewählt werden, weil dieser eine Initialgröße festsetzt
- Falls kein Wert voreingestellt wurde, so werden zehn Elemente angenommen
- In vielen Fällen ist dieser Wert zu klein.

`ArrayList`

- Die Größe eines neuen Feldes kann nach unterschiedlichen Strategien festgelegt werden
- Ein **`Vector`** (arbeitet nach dem Prinzip der **`ArrayList`** ist lediglich synchronisiert) verwendet die Verdopplungsmethode
- Wird er vergrößert, so ist das neue Feld doppelt so groß wie das alte
- Dies ist eine Vorgehensweise, die für kleine und schnell wachsende Felder eine clevere Lösung darstellt, großen Feldern jedoch schnell zum Verhängnis werden kann
- Die **`ArrayList`** verdoppelt nicht die Größe, sie nimmt die neue Größe mal 1,5

ArrayList

- Die interne Größe des Arrays kann mit **`ensureCapacity()`** geändert werden
- Ein Aufruf von **`ensureCapacity(int minimumCapacity)`** bewirkt, dass die Liste insgesamt mindestens **`minimumCapacity`** Elemente aufnehmen kann, ohne dass ein Resizing nötig wird
- Ist die aktuelle Kapazität der Liste kleiner als **`minimumCapacity`**, so wird mehr Speicher angefordert
- Die **`ArrayList`** verkleinert die aktuelle Kapazität nicht, falls sie schon höher als **`minimumCapacity`** ist

`LinkedList`

- Eine verkettete Liste hat neben den gegebenen Operationen aus dem Interface **`List`** weitere Hilfsmethoden:
  - **`addFirst()`**
  - **`addLast()`**
  - **`getFirst()`**
  - **`getLast()`**
  - **`removeFirst()`**
  - **`removeLast()`**
- Beispiel: **`list3`**
- **`LinkedList()`** Eine neue leere Liste.
- **`LinkedList(Collection<? extends E> c)`** Kopiert alle Elemente der Collection c in die neue verkettete Liste

`ArrayList` vs. `LinkedList`

- **`ArrayList`** speichert Elemente in einem Array (wie es auch **`Vector`** macht)
- **`LinkedList`** speichert die Elemente in einer verketteten Liste
- Die Verkettung wird mit einem Hilfsobjekt der Klasse **`Entry`** für jedes Listen-Element erreicht
- Je nach Einsatzgebiet, muss ausgewählt werden, ob die **`LinkedList`** oder die **`ArrayList`** verwendet werden sollte
- Da **`ArrayList`** intern ein Array benutzt, ist der Zugriff auf ein spezielles Element über die Position in der Liste sehr schnell
- Eine **`LinkedList`** muss aufwändiger durchsucht werden, und dies kostet Zeit

`ArrayList` vs. `LinkedList`

- Die verkettete Liste ist deutlich im Vorteil, wenn Elemente mitten in der Liste gelöscht oder eingefügt werden
- Es muss nur die Verkettung der Hilfsobjekte an einer Stelle verändert werden
- Bei einem **`ArrayList`**-Objekt als Container bedeutet dies viel Arbeit, es sei denn, das Element muss am Ende gelöscht oder eingefügt werden
  - Zum einen müssen alle nachfolgenden Listen-Elemente beim Einfügen und Löschen im Array verschoben werden
  - Zum anderen kann beim Einfügen das verwendete Array-Objekt zu klein werden
  - Dann bleibt, wie beim **`Vector`**, nichts anderes übrig, als ein neues Array-Objekt anzulegen und alle Elemente zu kopieren
- Beispiel: **`listvergleich1`**

`ListIterator`

- **`ListIterator`** ist eine Erweiterung von Iterator
- Diese Schnittstelle fügt noch Methoden hinzu, damit an der aktuellen Stelle auch Elemente eingefügt werden können
- Mit einem **`ListIterator`** lässt sich rückwärts laufen und auf das vorhergehende Element zugreifen

ListIterator

- **`boolean hasPrevious(), boolean hasNext()`** Liefert **`true`**, wenn es ein vorhergehendes/nachfolgendes Element gibt
- **`E previous(), E next()`** Liefert das vorangehende/nächste Element der Liste und **`NoSuchElementException`**, wenn es das Element nicht gibt
- **`int previousIndex(), int nextIndex()`** Liefert den Index des vorhergehenden/nachfolgenden Elements
  - Geht **`previousIndex()`** vor die Liste, so liefert die Methode **`–1`**, geht **`nextIndex()`** hinter die Liste, liefert die Methode die Länge der gesamten Liste
- **`void remove()`** (optional) Entfernt das letzte von **`next()`** oder **`previous()`** zurückgegebene Element
- **`void add(E o)`** (optional) Fügt ein neues Objekt in die Liste ein
- **`void set(E o)`** (optional) Ersetzt das Element, das **`next()`** oder **`previous()`** als letztes zurückgaben
- Beispiel: **`listiterator1`**

Map

- Bei einer Map handelt es sich um einen Assoziativspeicher
- Es werden immer Schlüssel/Wert-Paare abgespeichert
- Ein Assoziativspeicher arbeitet nur in einer Richtung schnell
- Wenn etwa im Fall eines Wörterbuches ein Wort mit einer Erklärung assoziiert wurde, kann die Datenstruktur die Frage nach einer Erklärung schnell beantworten
- In die andere Richtung dauert es wesentlich länger, weil hier keine Verknüpfung besteht

Map

- Java implementiert assoziative Speicher auf zwei unterschiedliche Arten:
  - Eine übliche und sehr schnelle Implementierung ist die Hash-Tabelle (engl. *hashtable*), die in Java durch **`java.util.HashMap`** implementiert ist
  - Daneben existiert die Klasse **`java.util.TreeMap`**, die etwas langsamer im Zugriff ist, doch dafür alle Schlüssel in einem Baum sortiert abspeichert

Map

- Mit der Methode **`put(K key, V value)`** können Schlüssel/Wert-Paare eingefügt werden
- Das erste Argument ist der Schlüssel und das zweite Argument der mit dem Schlüssel zu assoziierende Wert
- Der Schlüssel und der Wert können **`null`** sein
- Falls sich zu einem Schlüssel schon ein Eintrag in der Hash-Tabelle befindet, wird dieser überschrieben und der vorherige Wert zum Schlüssel zurückgegeben
- Ist der Schlüssel neu, liefert **`put()`** den Rückgabewert **`null`**
- Für Objekte, die als Schlüssel in einer Hash-Tabelle dienen sollen, müssen die Methoden **`hashCode()`** und **`equals()`** in geeigneter Weise (der Bedeutung oder Semantik des Objekts entsprechend) untereinander konsistent implementiert sein

Map

- Um wieder ein Element auszulesen, wird die Methode **`get(Object key)`** verwendet
- Das Argument identifiziert das zu findende Objekt über den Schlüssel, in dem das Objekt herausgesucht wird, das den gleichen Hashwert besitzt und im Sinne von **`equals()`** gleich ist
- Wenn das Objekt nicht vorhanden ist, ist die Rückgabe **`null`**
- Allerdings kann auch **`null`** der mit einem Schlüssel assoziierte Wert sein, denn **`null`** ist als Wert erlaubt

Map

- Neben **`get()`** kann auch noch mit einer anderen Methode das Vorhandensein eines Schlüssels getestet werden:
  - **`containsKey()`** überprüft, ob ein Schlüssel in der Tabelle vorkommt, und gibt dann ein **`true`** zurück
- Im Gegensatz zu **`get()`** und **`containsKey()`**, die das Auffinden eines Werts bei gegebenem Schlüssel erlauben, lässt sich auch nur nach den Werten (also ohne Schlüsselangabe) suchen
- Dies ist allerdings wesentlich langsamer, da alle Werte der Reihe nach durchsucht werden müssen
- Die Klasse bietet hierzu **`containsValue()`** an

Map

- Zum Löschen eines Elements gibt es **`remove()`** und zum Löschen der gesamten Map die Methode **`clear()`**
  - **`V remove(Object key)`** Löscht den Schlüssel und seinen zugehörigen Wert
  - Wenn der Schlüssel nicht in der Hash-Tabelle ist, so macht die Methode nichts
  - Der Wert zum Schlüssel wird zurückgegeben
  - **`void clear()`** Löscht die Einträge der Hash-Tabelle

Map

- Mit **`size()`** lässt sich die Anzahl der Wert-Paare in der Hash-Tabelle erfragen
- **`isEmpty()`** entspricht einem **`size() == 0`**, gibt also **`true`** zurück, falls die Hash-Tabelle keine Elemente enthält
- **`toString()`** liefert eine Zeichenkette, die eine Repräsentation der Hash-Tabelle zurückgibt
- Die Stringrepräsentation der Hash-Tabelle liefert jeden enthaltenen Schlüssel, gefolgt von einem Gleichheitszeichen und dem zugehörigen Wert

HashMap

- Die Klasse **`HashMap`** eignet sich ideal dazu, viele Elemente unsortiert zu speichern und sie über die Schlüssel schnell wieder verfügbar zu machen
- Es gibt zwei Konstruktoren:
  - **`HashMap()`**
    - Erzeugt eine neue Hash-Tabelle
  - **`HashMap(Map<? extends K,? extends V> m)`**
    - Erzeugt eine neue Hash-Tabelle aus einer anderen **`Map`**

Typsicherheit bei einer **`Map`**

- Soll eine **`Map`** typsicher erzeugt werden, muss sowohl für den Key, als auch für den Value ein Typ angegeben werden
- Z.B.:
```
Map<String, Integer> map = new
HashMap<String, Integer>();
```
- Beispiel: **`map1`**

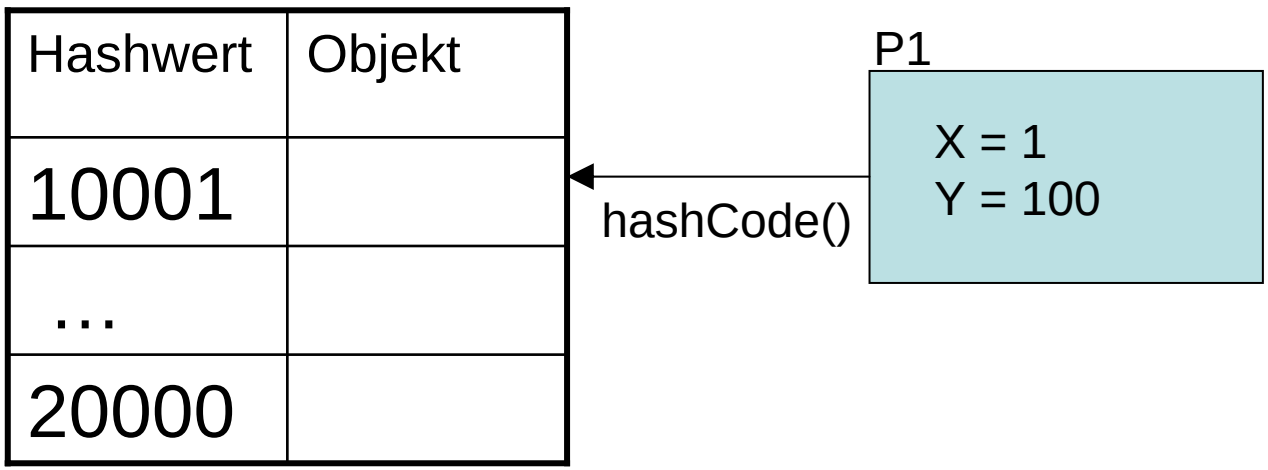
`equals/hashCode`

- Wenn Objekte in eine **`Map`** eingefügt werden, stellt sich die Frage, wonach die Identität zweier Keys überprüft wird
- Zunächst wird der Hashwert des Schlüssel-Objektes ermittelt, ist dieser gleich wird zusätzlich auf **`equals()`** überprüft
- Erst wenn **`equals()`** **`true`** liefert ist klar, dass der Key bereits enthalten ist
- Dies macht deutlich, dass die Implementierung von **`equals()`** und **`hashCode()`** bei eigenen Klassen wichtig ist
- Vor allem müssen die Methoden korrespondierend implementiert werden:
  - Liefert **`equals()`** für zwei Objekte **`true`**, muss auch der HashCode gleich sein

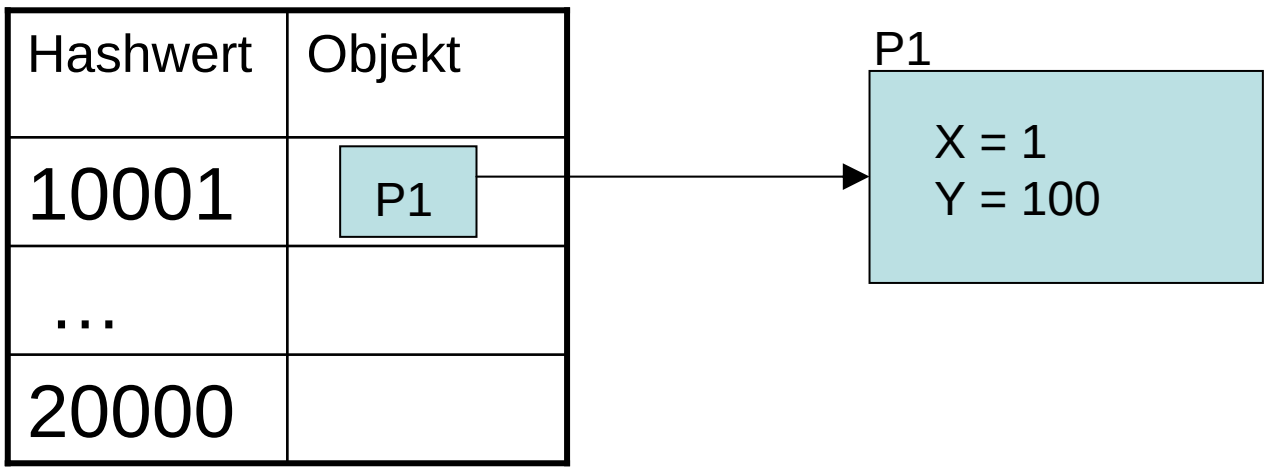
`equals/hashCode`

- Die verwendeten Keys sollten *immutable* sein
- Sind sie dies nicht, kann das Objekt verändert werden und der **`HashCode`** ändert sich
- Dadurch kann das Element nicht mehr gefunden werden und liegt als Leiche in der **`Map`**
- Beispiel:
  - Ein Punkt P(1,100) mit Hashwert 10001 wird als Key in das entsprechende Bucket eingefügt
  - Die Koordinaten werden auf (100,100) geändert, wodurch der Hashwert sich auf 20000 ändert
  - Das Element liegt nun im falschen Bucket
  - Wird nach P(100,100) gesucht ist das Objekt nicht dort
  - Wird nach P(1,100) gesucht, liegt im Bucket zwar ein Element, es ist aber nicht **`equals()`**
- **Achtung**: Bei einer **`TreeMap`** wird (analog zum **`TreeSet`**) mittels **`compareTo()`** die Gleichheit überprüft
- Beispiel: **`map2`**

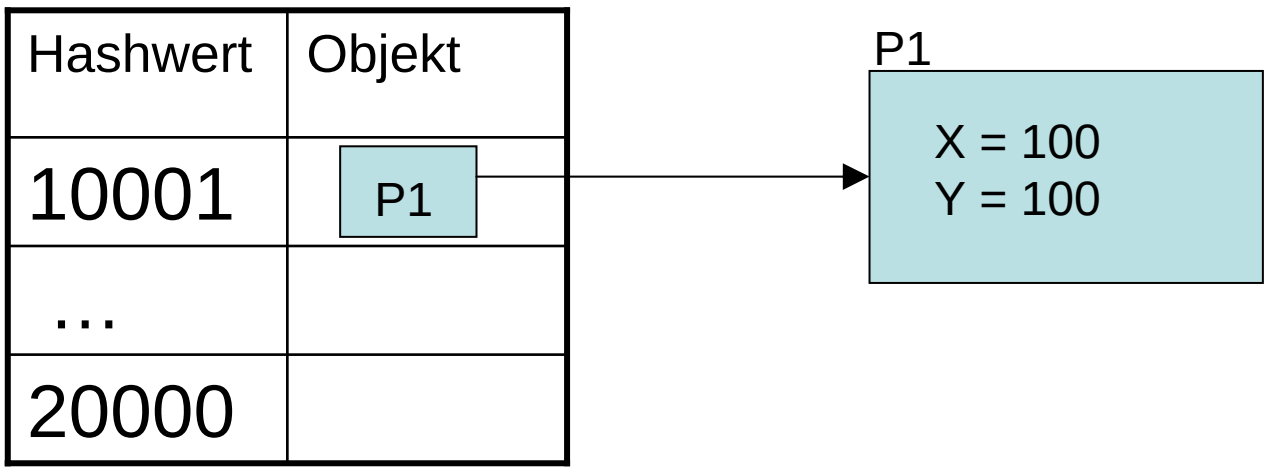
equals/hashCode



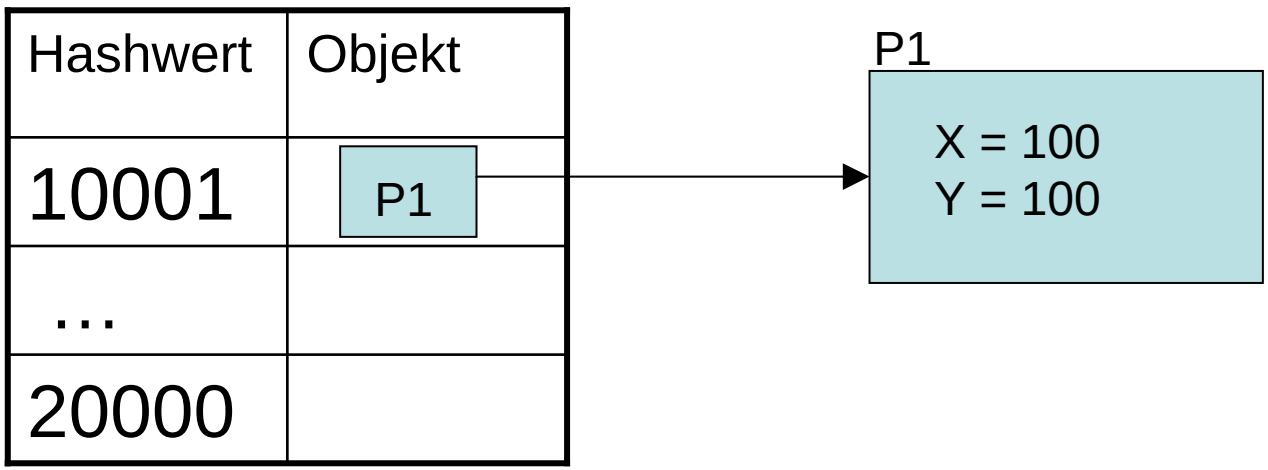
equals/hashCode



equals/hashCode



equals/hashCode



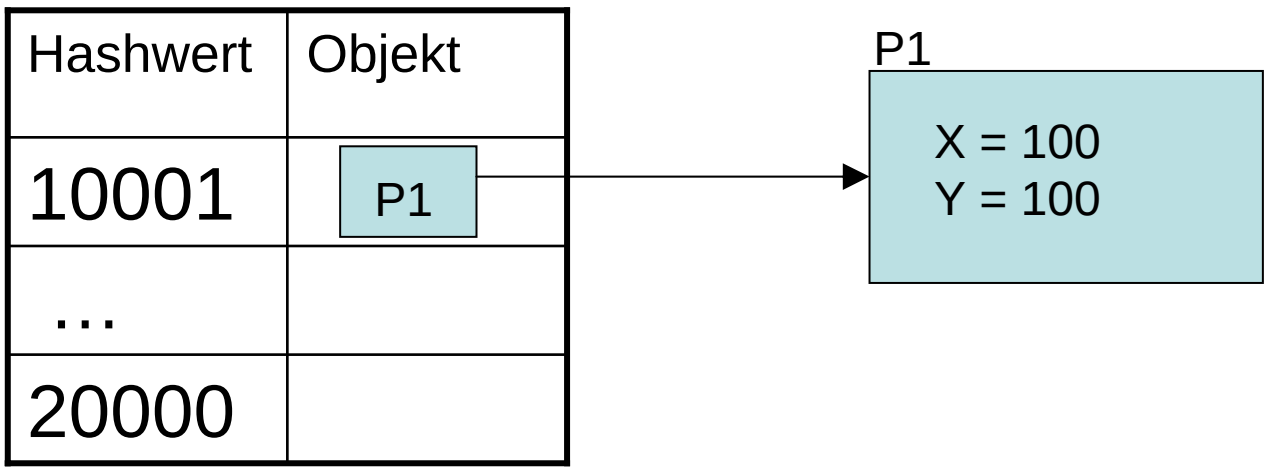
Ist P1 enthalten?

containsKey() bildet den Hashwert von P1

Der Hashwert ist 20000

Im Bucket 20000 ist das Objekt nicht

equals/hashCode



Ist P2 = new Point(1,100) enthalten?

containsKey() bildet den Hashwert von P2
Der Hashwert ist 10001
Im Bucket 10001 ist ein Objekt enthalten
Das Objekt ist aber nicht equals/compareTo
=> Das Objekt ist nicht in der Map

Collection-View einer **`Map`**

- Die **`Map`**-Klassen implementieren nicht das Interface **`Iterable`**
- Daher kann eine Instanz der Klasse **`HashMap`** nicht mit einer foreach-Schleife durchlaufen werden, wie es bei den anderen Collections der Fall ist
- Eine Map kann jedoch auf drei Arten Collection-Sammlungen zurückgeben:
  - **`keySet()`** liefert eine Menge der Schlüssel
  - **`values()`** liefert eine Collection der Werte
  - **`entrySet()`** liefert ein Set mit speziellen **`Map.Entry`**-Objekten. Die **`Map.Entry`** speichern gleichzeitig den Schlüssel sowie den Wert

Collection-View einer **`Map`**

- Ein **`Map.Entry`** Objekt besteht aus einem Schlüssel/Wert-Paar
- Mit den drei Collection-Views ist es möglich die Elemente in einer foreach-Schleife zu durchlaufen
- Dabei wird keine bestimmte Reihenfolge eingehalten, es sei denn es wird eine **`TreeMap`** verwendet und die enthaltenen Objekte sind **`Comparable`** oder es gibt einen entsprechenden **`Comparator`**
- Beispiel: **`map3`**

Rückblick

- Interface Collection
- **`Set`**
  - **`HashSet`**
  - **`TreeSet`**
- **`List`**
  - **`ArrayList`**
  - **`LinkedList`**
  - **`ListIterator`**
- **`Map`**

Ausblick

- Die Klasse Collections
- Generics
- Visitorpattern