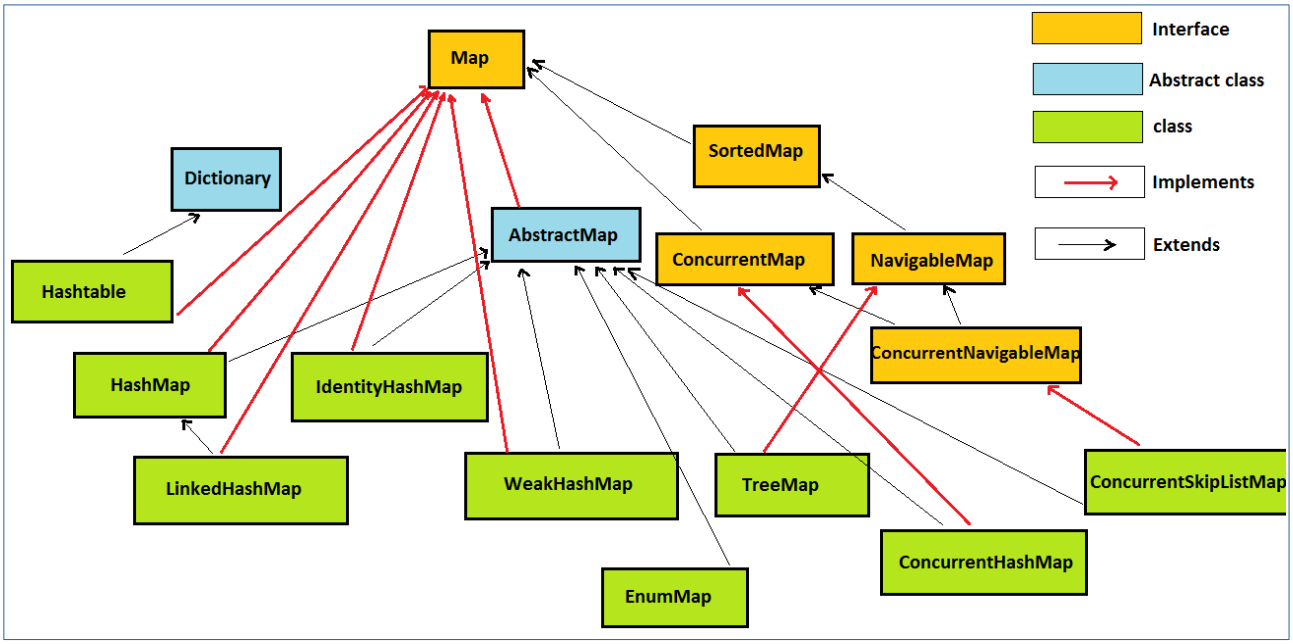
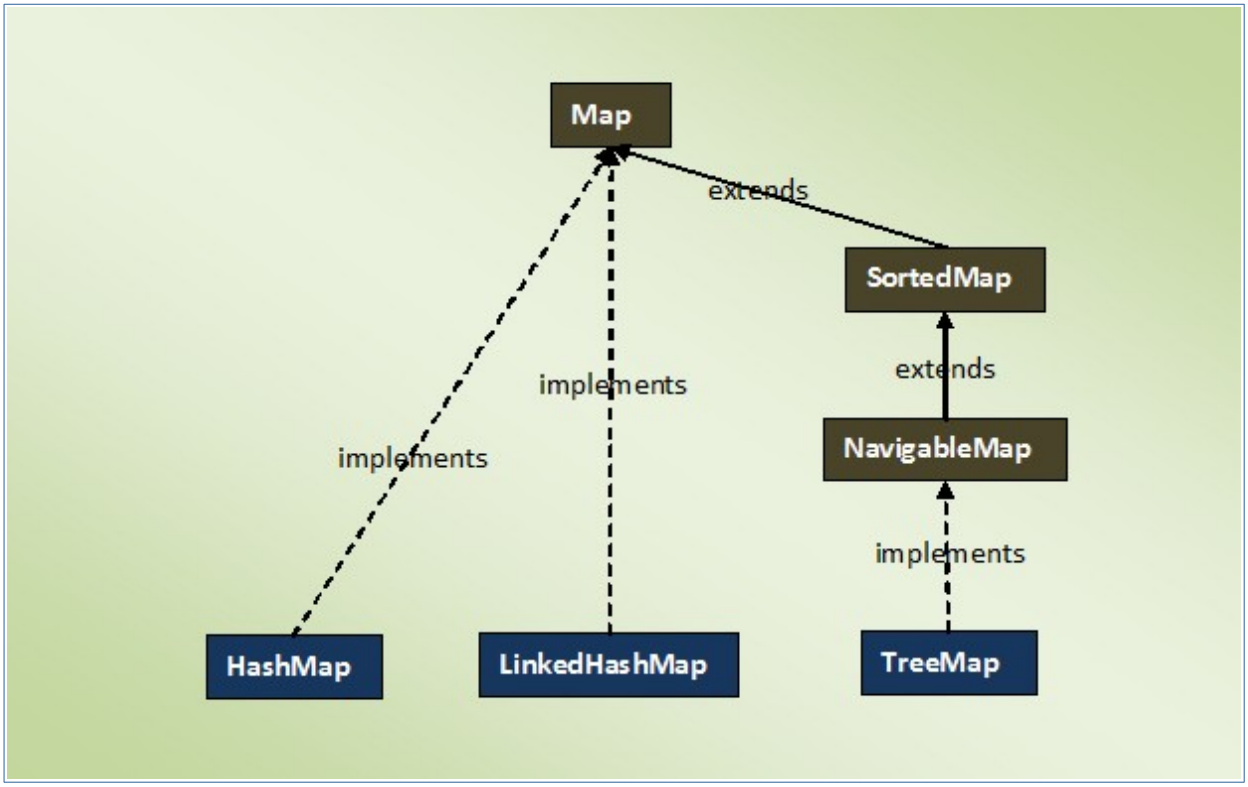
Tabellen:



Maps sind eine grundlegende Datenstruktur im **Java Collections Framework**, die auf der Idee basiert, **Schlüssel-Wert-Paare** zu speichern. Anders als Listen oder Sets, bei denen es nur um einzelne Werte geht, speichern **Maps** eine Zuordnung zwischen einem **Schlüssel** und einem **Wert**.



Grundkonzepte von Maps:

1. Schlüssel-Wert-Paare:

- Eine **Map** speichert Daten in **Schlüssel-Wert-Paaren**. Jeder **Schlüssel** ist **eindeutig**, das heißt, jeder Schlüssel kann nur einmal in der Map vorhanden sein. Jedoch kann ein Wert mehrfach vorkommen.
- **Beispiel**: Eine Map kann die Zuordnung zwischen **Benutzernamen** (Schlüssel) und **E-Mail-Adressen** (Wert) darstellen. Jeder Benutzername ist eindeutig, aber verschiedene Benutzer können theoretisch dieselbe E-Mail-Adresse haben.

2. Schlüssel (Key):

- Der **Schlüssel** wird verwendet, um den zugehörigen Wert zu identifizieren und darauf zuzugreifen. In einer Map ist jeder Schlüssel **eindeutig**, das bedeutet, zwei gleiche Schlüssel können nicht existieren.

3. Wert (Value):

- Der **Wert** ist die Information, die mit dem Schlüssel verknüpft ist. Die Werte müssen nicht eindeutig sein, das heißt, mehrere Schlüssel können denselben Wert speichern.

4. Kein fester Index:

- Im Gegensatz zu Listen (die ihre Elemente basierend auf einem Index speichern), **speichern Maps ihre Elemente basierend auf Schlüsseln**. Es gibt also **keine direkte Indizierung** der Elemente, wie bei einer Liste.

5 .Null-Schlüssel und Null-Werte:

- **Einige Map-Implementierungen erlauben** null als Schlüssel oder Wert, während andere dies nicht tun. Beispielsweise **kann eine `HashMap` einen null-Schlüsse**l und beliebig viele null-Werte speichern, während ein**e `TreeMap` keinen null-Schlüssel zulässt.**

6. Schnelle Zugriffsgeschwindigkeit:

- Maps bieten normalerweise eine sehr schnelle **Zugriffszeit** auf die gespeicherten Werte, oft **O(1)** bei Hash-basierten Implementierungen wie `HashMap`, oder **O(log n)** bei baumbasierten Implementierungen wie `TreeMap`.

Hashtable: (Ungeordnete Map)

- speichert die Schlüssel-Wert-Paare in **keiner bestimmten Reihenfolge**
- verwendet eine **Hash-Funktion** um den Schlüssel auf eine Position in einem internen Array zu mappen
- Hash-Funktion wandelt den Schlüssel in einen Hashcode um, der als Index im Array verwendet wird.

1. Schlüssel-Wert-Paare:

- Jeder Eintrag in einer Hashtable besteht aus einem **Schlüssel** und einem **Wert**. Die **Werte werden über ihre Schlüssel angesprochen.**

2. Hashing:

- Durch **Hashing** wird ein Schlüssel in einen **Index** in einem Array umgewandelt, wodurch der Zugriff auf den Wert in konstanter **Zeit (O(1))** möglich ist, sofern keine Kollisionen auftreten.

3. Kollisionen:

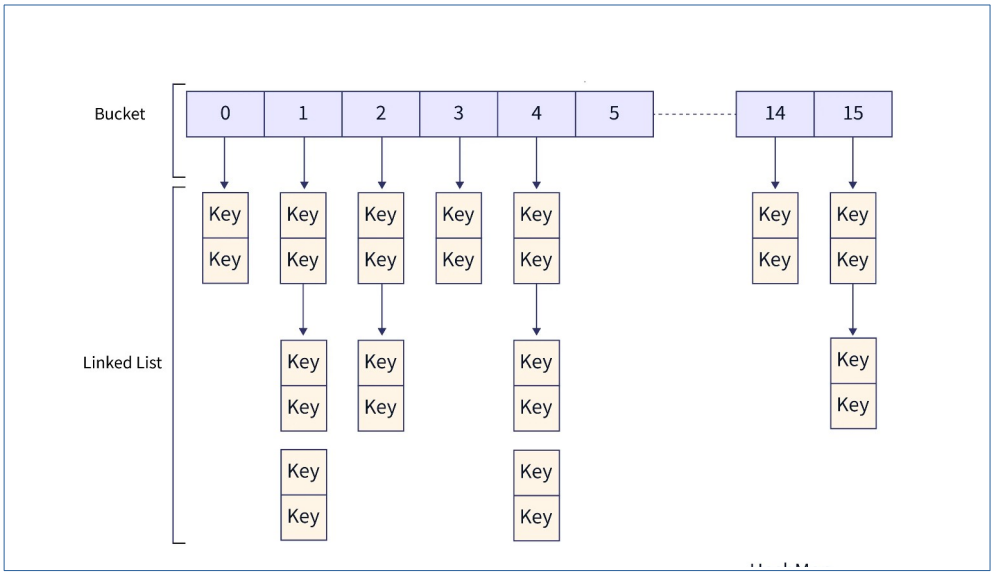
- Eine **Kollision** tritt auf, wenn zwei verschiedene Schlüssel denselben Hashcode erzeugen und dadurch auf denselben Index im Array zugreifen wollen. Um Kollisionen zu handhaben, verwenden Hashtables üblicherweise **Verkettung** (Chaining) oder **offene Adressierung**.

4. Null-Werte:

- In einer `Hashtable` können weder **null-Schlüssel** noch **null-Werte** gespeichert werden.

5. Zeitkomplexität:

- Einfügen, Löschen und Suchen in einer Hashtable haben eine durchschnittliche Zeitkomplexität von **O(1)**, da der Zugriff durch Hashing effizient ist.



Properties:

- die klasse **Properties** erbt von der Klasse **Hashtable**
- speicher auch mi**t Key und Value**
- **kann nur Strings speichern**

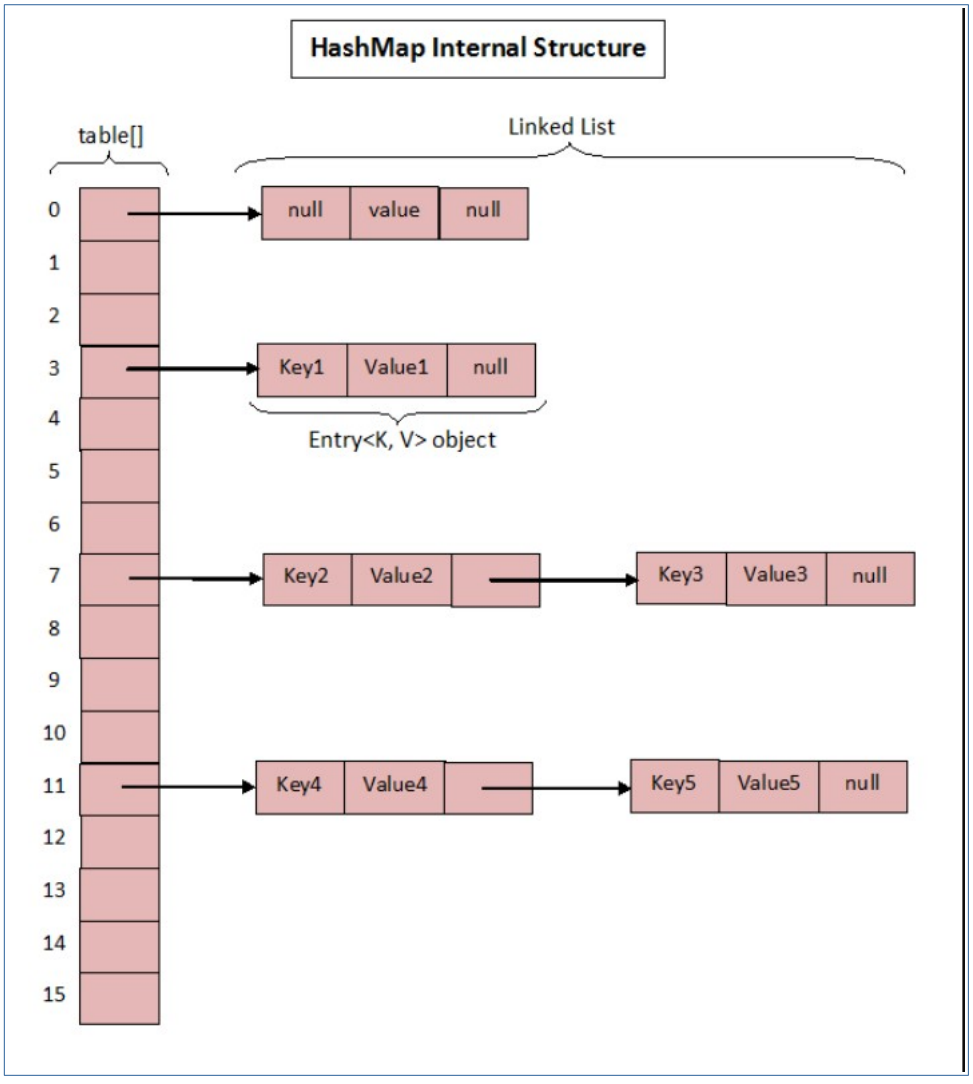
HashMap: (Ungeordnete Map)

1. **Schlüssel-Wert-Paare:**
   - Ähnlich wie bei der Hashtable speichert die HashMap Schlüssel-Wert-Paare, bei denen jeder Schlüssel eindeutig ist. Und speichern die Schlüssel **in keiner bestimmten Reihenfolge**

2. **Null-Schlüssel und Null-Werte:**
   - Im Gegensatz zur `Hashtable` erlaubt eine `HashMap` **einen null-Schlüssel** und beliebig viele **null-Werte**.

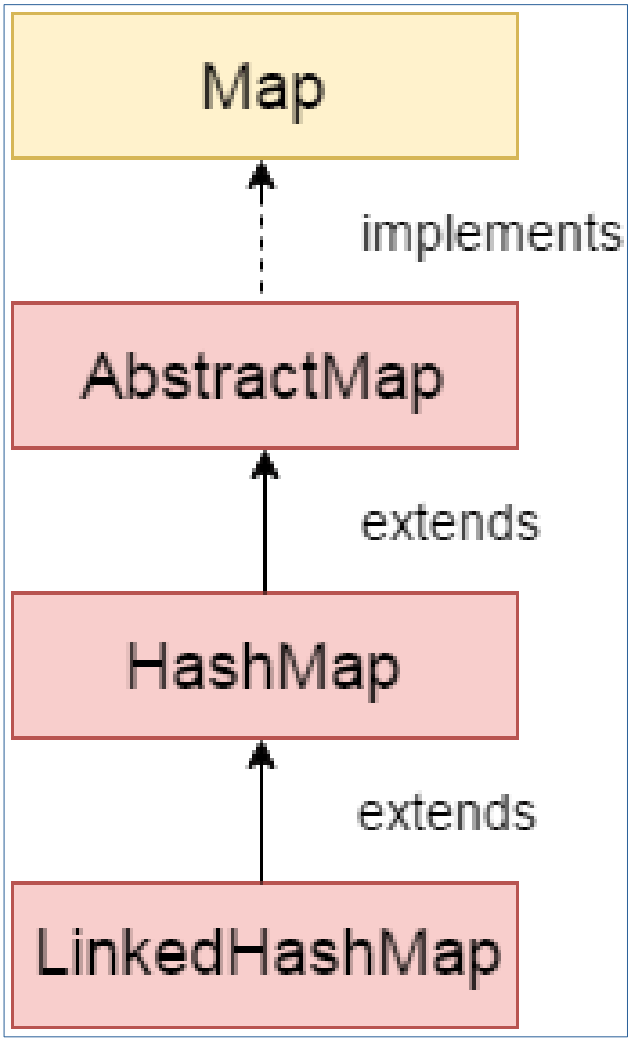
3. **Zeitkomplexität:**
   - Wie bei der `Hashtable` sind die Operationen **`put`, `get`, und `remove`** durchschnittlich in **O(1)**, solange das Hashing gut funktioniert und die Anzahl der Kollisionen gering ist.

4. **Kollisionen:**
   - **Wie bei der Hashtable werden Kollisionen durch Verkettung oder offene Adressierung gelöst.**



LinkedHashMap: (Geordnete Map)

- erbt von der Klasse **HashMap**
- kann **Null** Werte speichern
- Seine Elemente sind **nicht Indexed**
- Ein **LinkedHashSet** ist fast wie ein **HashSet**, hat aber eine zusätzliche Eigenschaft: Es **merkt sich**, in welcher Reihenfolge die Elemente eingefügt wurden.
- Also sie behält die **Einfügereihenfolge** oder eine **sortierte Reihenfolge** der Schlüssel bei.
- gibt es die Elemente immer in der gleichen Reihenfolge zurück
    - Z.B:
        - Wenn du die Zahlen 10, 20 und 30 hinzufügst, wirst du sie immer in der Reihenfolge 10, 20, 30 erhalten – auch nach dem Durchlaufen oder erneuten Anzeigen.
- **Das LinkedHashSet ist etwas langsamer als ein normales HashSet,**



TreeMap : (Geordnete Map)

- speichert die Schlüssel in einer **sortierten Reihenfolge**, entweder durch die natürliche Reihenfolge der Schlüssel oder durch einen benutzerdefinierten **Comparator**
- Verwendet einen balancierten binären Suchbaum (Red-Black Tree), daher haben Einfüge-, Lösch- und Suchoperationen eine Zeitkomplexität von **O(log n)**.
- `TreeMap` ist langsamer als `HashMap`, da sie eine Zeitkomplexität von **O(log n)** für die meisten Operationen hat.

Zusammenfassung der Eigenschaften von Maps

Map-Typ	Reihenfolge	Synchronisiert	Null-Schlüssel	Null-Werte	Zeitkomplexität
HashMap	Keine	Nein	Ja	Ja	O(1)
LinkedHashMap	Einfügereihenfolge	Nein	Ja	Ja	O(1)
TreeMap	Sortiert	Nein	Nein	Ja	O(log n)
Hashtable	Keine	Ja	Nein	Nein	O(1)
ConcurrentHashMap	Keine	Ja	Nein	Nein	O(1)
WeakHashMap	Keine (schwache Ref.)	Nein	Ja	Ja	O(1)