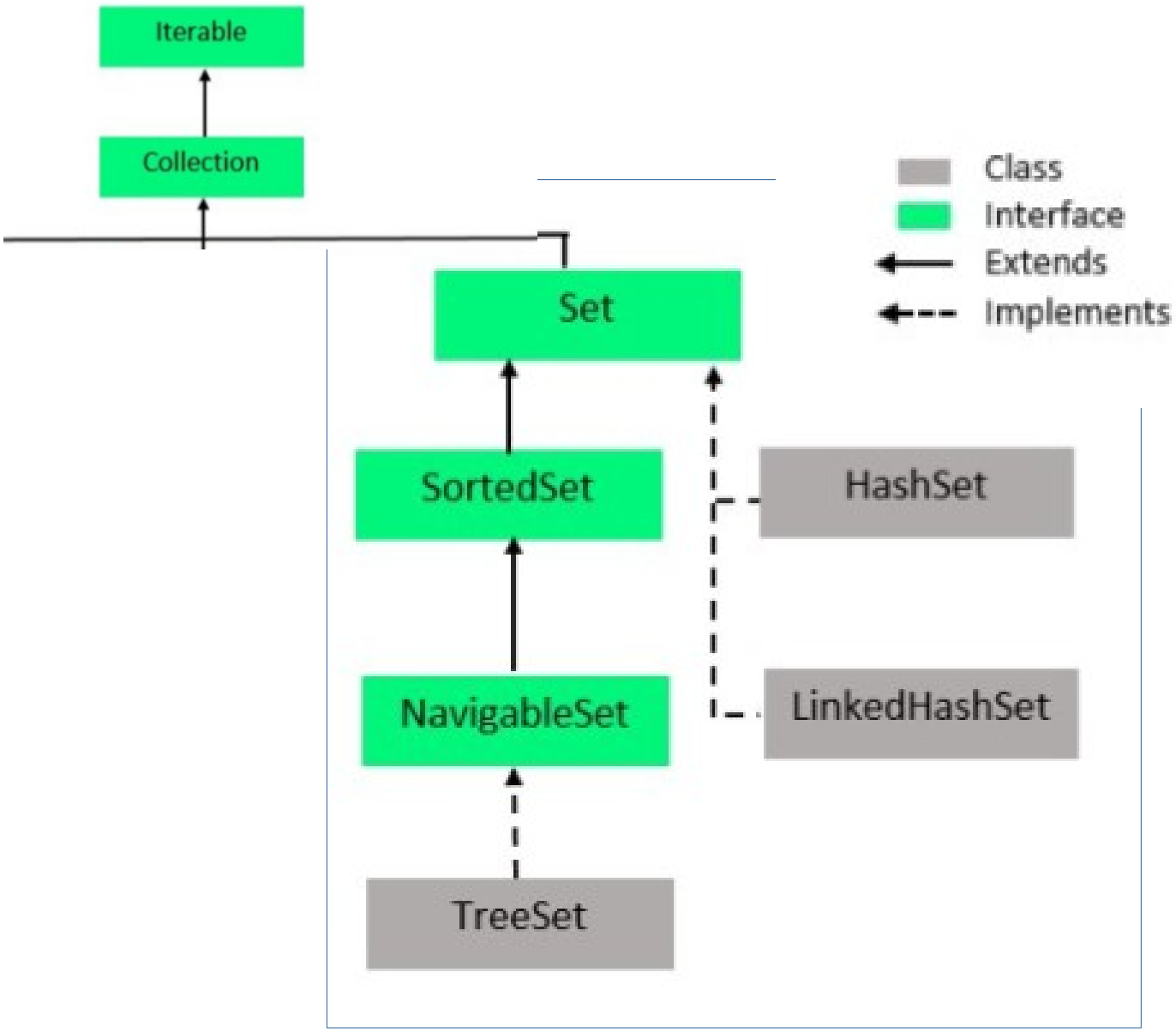

Ein **Set** ist eine weitere grundlegende Datenstruktur im **Java Collections Framework**, die sich auf die Speicherung **einzigartiger** Elemente konzentriert. Im Gegensatz zu **Lists**, die eine geordnete Sammlung von Elementen sein können (mit möglichen Duplikaten), erlaubt ein **Set** keine **doppelten Werte**.

Also Hauptfunktion ist: Keine Duplikate, keine garantierte Reihenfolge (außer bei speziellen Set-Implementierungen). Ideal für die Speicherung von **einzigartigen Elementen**.

Grundkonzepte von Sets:

1. Eindeutige Elemente:

- Ein **Set** speichert nur **eindeutige** Elemente. Das bedeutet, dass **keine Duplikate** vorhanden sein dürfen.

2. Ungeordnete Sammlung:

- hat in der Regel **keine festgelegte Reihenfolge**. Das bedeutet, dass die Elemente nicht in der Reihenfolge gespeichert werden

3. Null-Werte:

- Die meisten Set-Implementierungen, wie **HashSet**, erlauben **null** als Element. Allerdings kann ein Set nur **einen null-Wert** speichern, da null als ein einzigartiges Element betrachtet wird.

4. Effiziente Suche:

- Sets bieten normalerweise sehr schnelle **Such- und Einfügeoperationen**. In der Regel hat das Einfügen oder Prüfen, ob ein Element vorhanden ist, eine Zeitkomplexität von **O(1)** (bei Hash-basierten Implementierungen) oder **O(log n)** (bei baum-basierten Implementierungen).

5. Keine feste Reihenfolge (außer bei speziellen Set-Implementierungen):

- In den meisten Set-Implementierungen ist die Reihenfolge der Elemente **nicht festgelegt** (z.B. bei `HashSet`). Allerdings gibt es spezielle Set-Implementierungen, wie **LinkedHashSet** oder **TreeSet**, die eine bestimmte Reihenfolge aufrechterhalten:
- **`LinkedHashSet`:** Beibehaltung der **Einfügereihenfolge**.
- **`TreeSet`:** Speicherung der Elemente in **sortierter Reihenfolge**

6. Keine direkte Indizierung:

- Im Gegensatz zu Listen bietet ein Set **keinen direkten Indexzugriff** auf seine Elemente. Es gibt keine Methode, um auf das `n`-te Element zuzugreifen, weil die Reihenfolge in einem Set keine Rolle spielt (außer bei speziellen Implementierungen wie **`LinkedHashSet`** oder **`TreeSet`**).

Wichtige Set-Typen:

1. HashSet:

- **HashSet** ist die häufigste Set-Implementierung. Es speichert die Elemente **ungeordnet** und verwendet **Hashing**, um die Elemente effizient zu speichern.
- Die Einfüge- und Suchoperationen haben in der Regel eine Zeitkomplexität von **O(1)**.
- `HashSet` erlaubt **einen null-Wert**.
- Da es keine Reihenfolge beibehält, können die Elemente in einer beliebigen Reihenfolge ausgegeben werden.

2. LinkedHashSet:

- **LinkedHashSet ist wie ein `HashSet`, behält jedoch zusätzlich die Reihenfolge der Einfügungen bei.** Das bedeutet, wenn du Elemente in einer bestimmten Reihenfolge hinzufügst, wirst du sie auch in dieser Reihenfolge erhalten, wenn du das Set durchläufst.
- Die Leistung ist ähnlich wie bei einem `HashSet` (ungefähr **O(1)** für Einfügen und Suchen), aber es gibt einen geringen zusätzlichen Speicheraufwand, um die Einfügereihenfolge zu verfolgen.

3. TreeSet:

- **TreeSet** ist eine Implementierung des **SortedSet**-Interfaces und speichert die Elemente in **sortierter Reihenfolge**. Die Sortierung kann entweder nach der natürlichen Reihenfolge der Elemente erfolgen (z.B. numerische oder alphabetische Ordnung) oder durch einen benutzerdefinierten **Comparator**.
- Die Einfüge- und Suchoperationen haben eine Zeitkomplexität von **O(log n)**, da die Daten in einem **Red-Black-Baum** organisiert sind.
- `TreeSet` erlaubt **keine null-Werte**. Einfügen eines null-Wertes führt zu einer Ausnahme.

5. EnumSet:

- **EnumSet** ist eine spezialisierte Set-Implementierung für die Arbeit mit **Enums**. Es ist extrem effizient, da es die Werte des Enums direkt in einem kompakten, bitweisen Format speichert.
- `EnumSet` ist sehr schnell und hat eine sehr geringe Speicheranforderung, eignet sich aber nur für Enums.

Anwendungsfälle von Sets:

1. Vermeidung von Duplikaten:

- Sets werden häufig verwendet, um sicherzustellen, dass eine Sammlung von Elementen **keine Duplikate** enthält. Dies kann in vielen Szenarien nützlich sein, z.B. bei der Verwaltung von **IDs**, **Benutzernamen** oder **E-Mail-Adressen**, wo doppelte Werte nicht erlaubt sind.

Set-Typ	Reihenfolge	Null-Werte erlaubt	Zeitkomplexität (Einfügen/Suchen)	Besonderheiten
HashSet	Keine	Ja	O(1)	Ungeordnete, aber schnelle Sammlung
LinkedHashSet	Einfügereihenfolge	Ja	O(1)	Beibehaltung der Einfügereihenfolge
TreeSet	Sortierte Reihenfolge	Nein	O(log n)	Elemente sortiert (natürlich oder durch Comparator)
EnumSet	Sortierte Reihenfolge	Nein	O(1)	Sehr effizient für Enum-Werte
ConcurrentSkipListSet	Sortierte Reihenfolge	Nein	O(log n)	Thread-sicher, unterstützt mehrere Threads