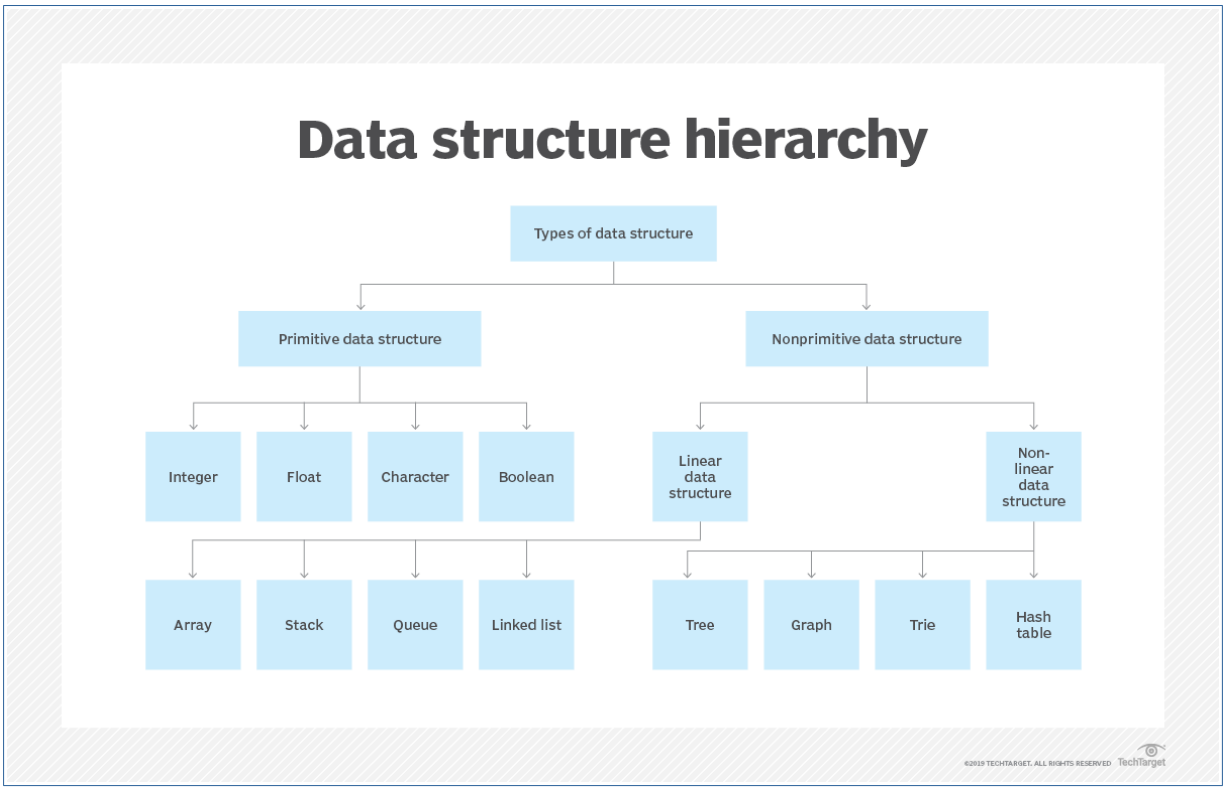

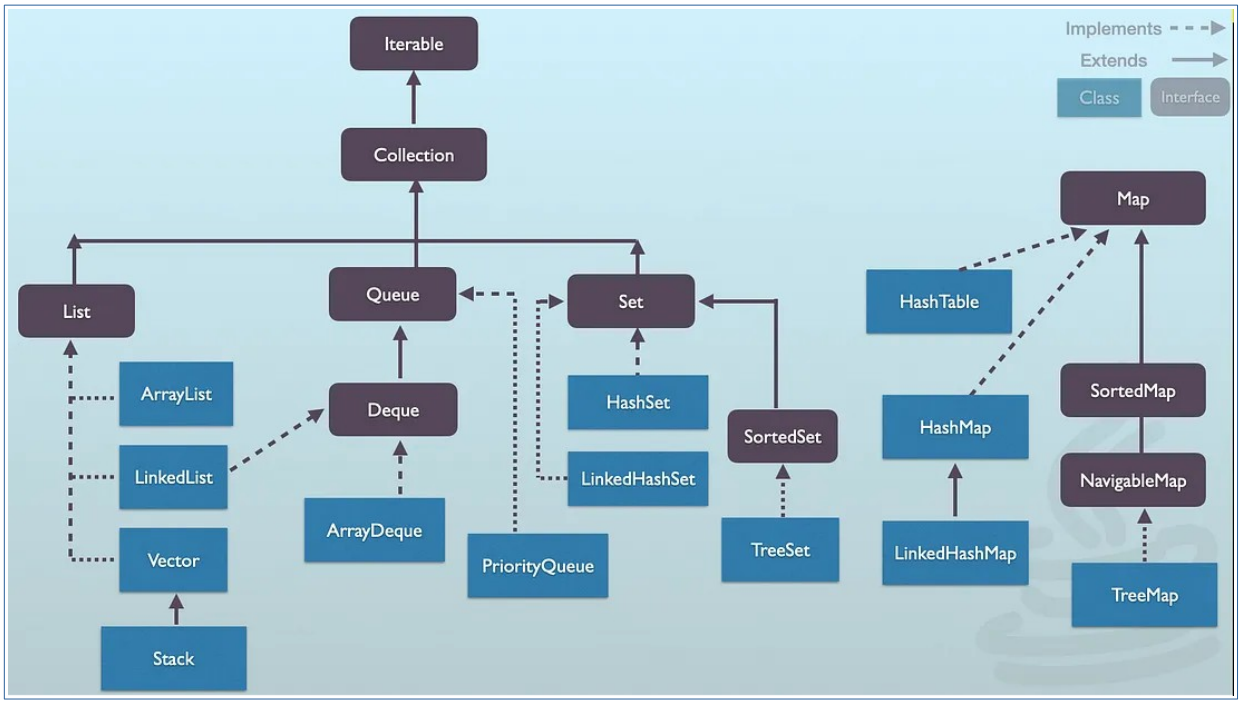

List vs. Set vs. Map

- **List**: Speichert **geordnete Elemente** und erlaubt **Duplikate**. Zugriff erfolgt über **Indizes**.
- **Set**: Speichert **einzigartige Elemente**, keine Duplikate. Keine direkte Indizierung.
- **Map**: Speichert **Schlüssel-Wert-Paare**, wobei die Schlüssel eindeutig sind, aber die Werte dupliziert werden können.

Lists Zusammenfassung:

Eine **List** ist eine **geordnete Sammlung**, die **Duplikate** erlaubt und **Index-basierten Zugriff** bietet. Sie ist ideal für Szenarien, in denen die Reihenfolge der Elemente wichtig ist und der Zugriff auf spezifische Positionen erforderlich ist. Es gibt verschiedene List-Implementierungen, die sich in ihrer Leistung und Flexibilität unterscheiden:

- **ArrayList** ist ideal für schnellen Zugriff über den Index, aber langsamer beim Einfügen/Entfernen in der Mitte.
- **LinkedList** ist effizient beim Hinzufügen und Entfernen von Elementen, insbesondere am Anfang oder Ende, aber langsamer beim Zugriff über den Index.
- **Vector** bietet die gleichen Vorteile wie `ArrayList`, ist aber synchronisiert und daher für Multithreading-Anwendungen geeignet.
- **Stack** ist eine spezielle Art von `Vector`, die nach dem **LIFO-Prinzip** arbeitet.

Listen sind besonders nützlich in Programmen, bei denen die Elemente in einer bestimmten Reihenfolge gespeichert und häufig über ihren Index angesprochen werden müssen.

Sets Zusammenfassung:

Ein **Set** ist eine Sammlung von **einzigartigen** Elementen, die typischerweise **keine feste Reihenfolge** haben.

- Es gibt verschiedene Implementierungen:
    - **HashSet** für schnelle und ungeordnete Mengen.
    - **LinkedHashSet** für Mengen mit Einfügereihenfolge.
    - **TreeSet** für sortierte Mengen.
    - **EnumSet** für Mengen von Enum-Werten.
    - **ConcurrentSkipListSet** für thread-sichere, sortierte Mengen.
- Sets bieten schnelle Zugriffszeiten, sind jedoch nicht geeignet, wenn du die Reihenfolge der Elemente auf Basis von Indizes steuern musst (wie bei einer Liste). Sie sind jedoch ideal, um **Duplikate zu verhindern** und für **Mengenoperationen** wie Vereinigung oder Schnittmenge.

Queue Zusammenfassung:

Eine **Queue** ist eine Sammlung, die typischerweise nach dem **FIFO-Prinzip** (First In, First Out) arbeitet.

- Es gibt verschiedene Implementierungen:
    - **PriorityQueue** für prioritätsbasierte Warteschlangen, bei denen die Reihenfolge durch eine natürliche Ordnung oder einen Comparator bestimmt wird.
    - **ArrayDeque** für doppelendige Warteschlangen, die sowohl FIFO als auch LIFO unterstützen.
    - **LinkedList** kann ebenfalls als Queue verwendet werden und bietet doppelt verkettete Warteschlangen.
- Queues sind ideal, um **Daten in der Reihenfolge ihrer Ankunft** zu verarbeiten, bieten jedoch keinen direkten Indexzugriff auf Elemente.
- Sie sind nützlich für Szenarien wie **Aufgabenverwaltung** oder **Prozesswarteschlangen**, bei denen die Reihenfolge wichtig ist, aber **Duplikate** erlaubt sind.

Map Zusammenfassung:

Eine **Map** speichert **Schlüssel-Wert-Paare**, wobei die **Schlüssel eindeutig** sind und auf ihre zugehörigen Werte zugreifen lassen.

- Es gibt verschiedene Implementierungen:
    - **HashMap** für schnelle, ungeordnete Schlüssel-Wert-Zuordnungen.
    - **LinkedHashMap** für Maps, die die **Einfügereihenfolge** der Schlüssel beibehalten.
    - **TreeMap** für **sortierte** Schlüssel-Wert-Zuordnungen basierend auf der natürlichen Reihenfolge oder einem Comparator.
    - **Hashtable** für thread-sichere Maps, die keine null-Schlüssel oder null-Werte erlauben.
    - **ConcurrentHashMap** für hochperformante, **thread-sichere** Maps in Mehr-Thread-Umgebungen.
- Maps bieten schnellen Zugriff auf Werte über die **Schlüssel**, sind aber nicht geeignet, wenn die Reihenfolge oder ein direkter Indexzugriff erforderlich ist.
- Sie sind ideal für **Datenzuordnungen**, bei denen du Schlüssel verwenden musst, um Werte schnell abzurufen, wie z.B. bei **Wörterbüchern** oder **Datenbanken**.