Die String Klasse in Java

Die String Klasse in Java ist eine Klasse, die es ermöglicht, Strings zu erstellen und zu manipulieren. Sie ist eine Klasse, die in Java bereits vordefiniert ist und muss nicht importiert werden.
Beispiel:

```
String s = "Hallo Welt";
```

Man kann Strings auch mit dem `new` Operator erstellen, dies ist aber nicht zu empfehlen, da es zu Problemen mit dem == Operator kommen kann.
Beispiel:

```
String s = new String("Hallo Welt");
```

Um zwei Strings zusammen zufügen, kann der + Operator verwendet werden.
Beispiel:

```
String s = "Hallo" + "Welt";
```

Methoden

Methode	Beschreibung
`length()`	Gibt die Länge des Strings zurück.
`charAt(int index)`	Gibt das Zeichen an der Stelle `index` zurück.
`substring(int beginIndex, int endIndex)`	Gibt den Teilstring von `beginIndex` bis `endIndex` zurück.
`substring(int beginIndex)`	Gibt den Teilstring von `beginIndex` bis zum Ende des Strings zurück.
`indexOf(String s)`	Gibt den Index des ersten Vorkommens von `s` zurück.
`lastIndexOf(String s)`	Gibt den Index des letzten Vorkommens von `s` zurück.
`startsWith(String s)`	Gibt `true` zurück, wenn der String mit `s` beginnt.
`endsWith(String s)`	Gibt `true` zurück, wenn der String mit `s` endet.
`contains(String s)`	Gibt `true` zurück, wenn der String `s` enthält.
`equals(String s)`	Gibt `true` zurück, wenn der String gleich `s` ist.
`equalsIgnoreCase(String s)`	Gibt `true` zurück, wenn der String gleich `s` ist, ohne auf Groß- und Kleinschreibung zu achten.
`compareTo(String s)`	Gibt einen Wert kleiner als 0 zurück, wenn der String lexikographisch vor `s` kommt.
`compareToIgnoreCase(String s)`	Gibt einen Wert kleiner als 0 zurück, wenn der String lexikographisch vor `s` kommt, ohne auf Groß- und Kleinschreibung zu achten.
`toUpperCase()`	Gibt den String in Großbuchstaben zurück.
`toLowerCase()`	Gibt den String in Kleinbuchstaben zurück.
`trim()`	Gibt den String ohne Leerzeichen am Anfang und Ende zurück.
`replace(String oldString, String newString)`	Gibt den String zurück, in dem alle Vorkommen von `oldString` durch `newString` ersetzt wurden.
`split(String regex)`	Gibt ein String Array zurück, das den String an den Stellen, an denen `regex` vorkommt, aufteilt.
`startsWith(String prefix)`	Gibt `true` zurück, wenn der String mit `prefix` beginnt.
`endsWith(String suffix)`	Gibt `true` zurück, wenn der String mit `suffix` endet.
`valueOf(int i)`	Gibt den String zurück, der die Zahl `i` repräsentiert.

Fachbegriffe Bäume

- **Wurzel**: Der oberste Knoten (ohne Vorgänger)
- **Knoten**: In einem Knoten werden die Daten gespeichert (immer die Wurzel eines Teilbaums und können Nachfolger besitzen)
- **Kanten**: Verbindung zwischen zwei Knoten
- **Tiefe**: Anzahl der Kanten von der Wurzel bis zum jeweiligen Knoten.
- **Grad eines Baumes**: Anzahl der maximalen Nachfolger eines Baumes.
- **Blatt**: Knoten ohne Nachfolger.
- **Teilbaum**: Jeder Teil eines Baumes ist ebenfalls ein Baum und zwar der sogenannte Teilbaum.
- **Pfad**: Weg über Kanten des Baumes, die zu einem bestimmten Knoten führen.
- **Ebene (Niveau)**: Alle Knoten, deren Pfad zur Wurzel gleichlang sind, befinden sich auf einer Ebene.

Binäre Bäume (BinaryTrees)

Jeder Knoten darf maximal zwei Nachfolger besitzen.

Binäre Suchbäume (BinarySearchTrees)

Jeder Knoten darf maximal zwei Nachfolger besitzen. Es darf nicht mehrmals den selben Inhalt eines Knotens im Baum geben. Die Inhalte müssen sortiert sein, hierbei befinden sich im linken Teilbaum kleinere Elemente und im rechten Teilbaum ausschließlich größere Elemente.

Entarteter Baum

Alle Knoten des Baumes haben entweder nur Nachfolger nach links oder rechts.

Traversierungsverfahren

Pre-Order:
Zunächst wird die Wurzel betrachtet, anschließend der linke Teilbaum und zuletzt der rechte Teilbaum:

```
public void traversePreOrder(BinaryTree<Integer> b) {
    System.out.println(b.getContent());
    if(!b.getLeftTree().isEmpty()) {
        traversePreOrder(b.getLeftTree());
    }
    if(!b.getRightTree().isEmpty()) {
        traversePreOrder(b.getRightTree());
    }
}
```

In-Order:
Zunächst wird der linke Teilbaum betrachtet, anschließend die Wurzel und zuletzt der rechte Teilbaum:

```
public void traverseInOrder(BinaryTree<Integer> b) {
    if(!b.getLeftTree().isEmpty()) {
        traverseInOrder(b.getLeftTree());
    }
    System.out.println(b.getContent());
    if(!b.getRightTree().isEmpty()) {
        traverseInOrder(b.getRightTree());
    }
}
```

Post-Order:
Zunächst wird der linke Teilbaum betrachtet, anschließend der rechte Teilbaum und zuletzt die Wurzel:

```
public void traversePostOrder(BinaryTree<Integer> b) {
    if(!b.getLeftTree().isEmpty()) {
        traversePostOrder(b.getLeftTree());
    }
    if(!b.getRightTree().isEmpty()) {
        traversePostOrder(b.getRightTree());
    }
    console.println(b.getContent());
}
```

In einen Binären Suchbaum einfügen

Pseudocode:

```
Inhalt einfügen (Inhalt neuer Inhalt)
falls (neuer Inhalt != null) dann
  falls (baum leer) dann
    fülle den baum mit neuerInhalt
  sonst
    falls (neuerInhalt < wurzelInhalt) dann
      linkerTeilbaum.einfügen(neuerInhalt)
    sonst
      falls(neuerInhalt > wurzelinhalt) dann
        rechterTeilbaum.einfügen(neuerInhalt)
```

Löschen aus einem Binären Suchbaum

Pseudocode:

```
  entferne (Inhalt exInhalt)
  falls (exInhalt != null und baum nicht leer) dann
    falls (exInhalt == wurzelInhalt) dann
      falls (baum hat keine Teilbäume) dann
        leere den baum
      sonst
        falls (rechter Teilbaum ist leer) dann
          rücke linken Teilbaum an die Position des zu entfernenden Knotens
        sonst
          falls (linker Teilbaum ist leer) dann
            rücke rechten Teilbaum an die Position des zu entfernenden
            Knotens
          sonst
            setze Maximum des linken Teilbaums an die Position des zu
            löschenden Knotens
            entferne (Maximum des linken Teilbaums)
    sonst
      falls (exInhalt < wurazelInhalt) dann
        linkerTeilbaum.entferne(exInhalt)
      sonst
        rechterTeilbaum.entferne(exInhalt)
```