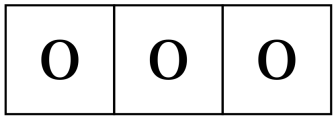
Man unterscheidet im Wesentlichen zwei große Untertypen von Listenimplementierungen. Die, die sich auf Felder (arrays) stützen und die, die eine „verzeigerte“ Struktur aufweisen (linked lists). Da Felder bereits aus der zehnten Jahrgangsstufe bekannt sind, ist dieser Ansatz der naheliegenderer und wir beginnen mit ihm. Für den Moment wollen wir uns ein Feld als reservierten Speicherplatz vorstellen, im Folgenden symbolisiert durch eine Aneinanderreihung an Quadraten. Aktuell soll das Feld noch vollständig leer sein.

In Java werden Felder allerdings standardmäßig mit 0 bzw. null intialisiert, wenn keine weiteren Vorgaben gemacht werden. **Beispiel**:

```
int[] array = new int[3];
```

Die Variable array zeigt nun zu einem Feld, das 3 mal den Eintrag 0 enthält, in Java-Schreibweise: {0,0,0}



Neben primitiven Datentypen wie int können auch Listen von beliebigen Objekten gebildet werden. Nehmen wir an, wir haben eine Klasse „Mensch “bereits erzeugt, dann wird

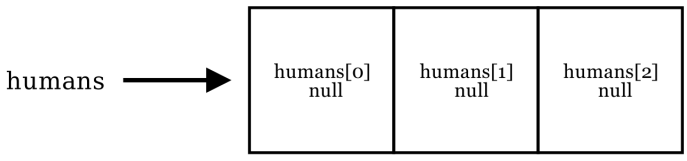
```
Human[] humans = new Human[5];
```

ein Feld mit 5 Einträgen null erzeugen, auf das durch die Variable humans zugegriffen wird.
Wir können auf beliebige Einträge eines zuvor erzeugten Feldes zugreifen, wenn wir Zugriff auf die Variable haben, die den Anfang der Liste speichert, z.B.:

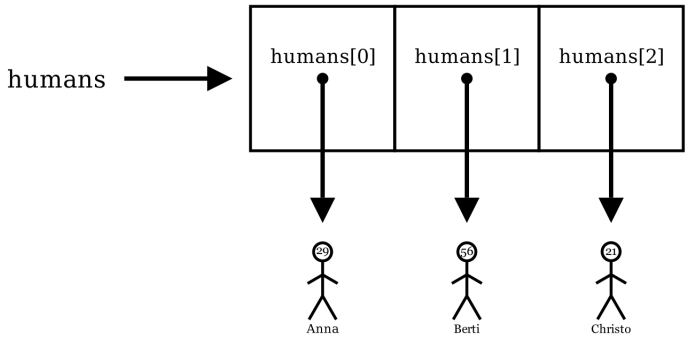
```
int[] numbers = {5,3,42,17,-2};
int thirdEntry = numbers[2];
System.out.println(thirdEntry);
```

Der obige Code wird die Zahl 42 auf die Konsole schreiben. Dieses Beispiel soll daran erinnern, dass in der Informatik bei 0 zu zählen begonnen wird. Auf den für uns dritten Eintrag wird also mit der „2“ zugegriffen.

Versuchen wir stattdessen, z.B. mit humans[0] auf den ersten Menschen zuzugreifen, werden wir nur null erhalten,da dieses Feld aktuell nur null-Einträge enthält.



Haben wir die Methode hintenAnfügen() (push()) implementiert und drei Menschen Anna, Berti und Christo erzeugt und zur Warteschlange hinzugefügt:



Dabei stehen die Strichys (Germanistik-Exkurs) ebenfalls wieder für eine bestimmte Menge an Daten, die (ggf. an ganz anderer physikalischer Stelle) im Speicher abgelegt sind.
Grundsätzlich bietet das Feld an sich in seiner üblichen Implementierung in Java bereits alle Eigenschaften, die wir von einer Liste verlangen würden. Wenn unsere Warteschlange benutzt wird, soll aber nur am Anfang entnommen und am Ende angefügt werden können. Es ist also mehr Kontrolle nötig.
Beginnen wir damit unsere Warteschlange als neue Klasse zu definieren. Da es unser Ziel ist eine Schlange von Menschen zu modellieren, brauchen wir zunächst eine Klasse, die einen Menschen beschreibt.

Human
- name: String - age: int

Mensch
- name: String - alter: int

Bevor es mit der Implementierung weitergeht bietet sich an dieser Stelle eine Wiederholung der wichtigsten drei Diagrammarten an, die in der Oberstufe Verwendung finden:

- Klassendiagramm
- Objektdiagramm
- Sequenzdiagramm

⇒ Exkurs im **Anhang**

```
    public class Human() {
        private String name;
        private int age;

        public Human(String name, int age){
            this.name = name;
            this.age = age;
        }
    }
```

Unsere Menschen haben aktuell nur die Attribute Name und Alter und außer dem Konstruktor sind keine weiteren Methoden vorhanden. Unser Ziel ist es eine Warteschlange der folgenden Struktur zu bauen:

Queue
- queue: Human[]
+ Queue() + push(h: Human): void + pop(): Human

Warteschlange
- warteschlange: Mensch[]
+ Warteschlange() + hintenAnfügen(Mensch): void + vorneEntfernen(): Mensch

```
public class MyListArray(){
    private Human[] queue;
    private int count;

    public MyListArray() {
        queue = new Human[5];
        count = 0;
    }
}
```

Der Konstruktor erzeugt ein neues Feld der Größe 5 und setzt die derzeitige Belegung auf 0. Die Anzahl der Menschen in der Warteschlange muss nicht zwangsläufig in einem Attribut gespeichert werden, da es aber eine Größe ist, die häufiger von Nutzen ist, wäre eine ständige Berechnung unnötiger Aufwand.
Um die Warteschlange auch sinnvoll verwenden zu können müssen weitere Methoden ergänzt werden. Wir beginnen mit der hintenAnfügen() Methode. Da wir spezifizieren müssen, was wir anfügen wollen folgt für die Signatur:

```
public void push(Human humanToAdd)
```

Ein erster - naiver - Ansatz zur Befüllung wäre der folgende:

```
public void push(Human humanToAdd) {
    for(int i = 0; i < queue.length; i++) {
        if(queue[i] == null) {
            queue[i] = humanToAdd;
            break;
        }
    }
}
```

Aufgabe 1: Begründen Sie kurz, warum diese Implementierung ineffizient ist.

Eine effizientere Lösung besteht darin, das Attribut *anzahl* zu verwenden, dadurch vereinfacht sich die Implementierung zu:

```
public class MyListArray(){
    private Human[] queue;
    private int count;

    public MyListArray() {
        queue = new Human[5];
        count = 0;
    }

    public void push(Human humanToAdd) {
        queue[count] = humanToAdd;
        count++;
    }
}
```

Aufgabe 2: Begründen Sie, warum auch diese Implementierung noch problematisch ist. Lösen Sie das Problem!

Aktuell kann unsere Liste nur fünf Elemente enthalten. Versucht man ein sechstes Element anzuhängen, so erhält man als Nachricht **„java.lang.ArrayIndexOutOfBoundsException“**. Es wird also versucht an eine Position der Liste einzufügen, die nicht existiert.
Ist die Liste voll, muss das Feld vergrößert werden. Da dies eine Funktionalität ist, die wir eventuell noch an anderer Stelle brauchen könnten, ist es sinnvoll dafür eine eigene Methode anzulegen. Sie erzeugt ein neues, größeres Feld und kopiert die Einträge aus dem alten Feld in das neue Feld. Anschließend wird neu verzeigert.
Erinnerung: Die Länge eines Feldes ist in einem öffentlichen Attribut des Felds gespeichert, auf das direkt zugegriffen wird, *length* ist deswegen kein Methodenaufruf in untenstehendem Code, sondern bezieht sich auf dieses Attribut. Keine Klammern!

```
public void push(Human humanToAdd) {
    if(count == queue.length) {
        enlargeArray();
    }
    queue[count] = humanToAdd;
    count++;
}

private void enlargeArray() {
    Human[] newQueue = new Human[queue.length + 5];
    for(int i = 0; i < queue.length; i++){
        newQueue[i] = queue[i];
    }
    queue = newQueue;
}
```

Um die Methode effizienter zu gestalten, wird das Feld um 5 Plätze vergrößert. Man möchte zu große Felder vermeiden (insbesondere mit leeren Einträgen!), um nicht unnötig Speicherplatz zu verschwenden (die Implementierung der ArrayList, die von Java direkt angeboten wird verwendet z.B. eine Vergrößerung um 5 Plätze, für Details zu dieser „vorgebauten“ Struktur siehe Java Doc ArrayList).
Als nächstes muss noch das vorderste Element der Warteschlange entfernt werden können. Spätestens hier zeigt sich, dass das Feld für diese Anwendungsform der Liste nur schlecht geeignet ist. Wenn das vorderste Element entfernt wird, müssen alle folgenden Elemente einen Platz nach vorne geschoben werden, ansonsten entsteht mit der Zeit ein großer Block an leeren Einträgen im Feld. Dieser Kopiervorgang nimmt vergleichsweise viel Zeit und Leistung in Anspruch. Eine mögliche Implementierung der Methode sieht folgendermaßen aus:

```
public Human pop() {
    if(queue[0] == null) {
        return null;
    }
    Human toReturn = queue[0];
    for(int i = 0; i < count - 1; i++){
        queue[i] = queue[i+1];
    }
    queue[count-1] = null;
    count--;
    return toReturn;
}
```

Die Methode gibt eine Referenz auf das entfernte Objekt zurück, sofern es existiert. Wird diese Referenz nicht in einer neuen Variable gespeichert bei Anwendung der Methode, so wird sie vom garbage collector aufgeräumt und ist vollständig verschwunden. Alternativ könnte auch eine äquivalente Methode mit der Signatur

```
public void pop()
```

definiert werden. In diesem Fall wird der erste Mensch aus der Schlange entfernt und das Objekt geht „verloren“.
Visualisierung der Anfügen und Entfernen-Methoden: Youtube - push - pop

Übungsblock
Die folgenden Aufgaben steigen grob im Schwierigkeitsgrad an und erweitern die Funktionalität der Liste sukzessive. Die Lösungen finden sich kommentiert auf den darauf folgenden Seiten. Versuchen Sie die beschriebenen Methoden zunächst selbst zu implementieren!

Aufgabe 3: Schreiben Sie eine Methode schreibeListe (printList()), die die Warteschlange in einer sinnvollen Weise auf der Konsole sichtbar macht.
Hinweis: Ergänzen Sie eine passende Methode in der Klasse Mensch (Human)!

Aufgabe 4: Schreiben Sie eine Methode menschAnPosition(int position) (humanAtPosition(int position)), die eine Referenz zum Menschen an der angegebenen Position zurückgibt. Beachten Sie Grenz- bzw. Spezialfälle!

Aufgabe 5: Schreiben Sie eine Methode sucheMenschInSchlange(Mensch m) (searchHumanInQueue(Human h)), die die Position des Menschen in der Schlange zurückgibt.
Hinweis: Überschreiben Sie die allgemeine equals() Methode in der Klasse Mensch.

Aufgabe 6: Schreiben Sie eine Methode enthält(Mensch m) (contains(Human h)), die wahr zurückgibt, wenn der der entsprechende Mensch in der Schlange ist, anderfalls falsch.
Hinweis: Verwenden Sie die equals-Methode aus Aufgabe 5

Aufgabe 7: Schreiben Sie eine Methode entferneAnPosition(int position) (removeAtPosition(int position)), die den Menschen an der gegebenen Stelle entfernt und die übrigen nach vorne rutschen lässt.
Schreiben Sie zwei Versionen dieser Methode, eine, die den Menschen nur entfernt und eine, die eine Referenz zu diesem Menschen zurückgibt.

Aufgabe 8 - für Experten: Schreiben Sie eine Methode zusammenfügen(MeineListeFeld zweiteWarteschlange) (concatenate(MyListArray secondQueue)), die eine zusammengefügte Liste zurückgibt.

Aufgabe 9 - für Experten: Schreiben Sie eine Methode fügeSortiertHinzu(Mensch m) (appendSorted(Human h)), die die Warteschlange nicht von hinten auffüllt, sondern die Menschen an einer bestimmten Stelle einsortiert.
Hinweis: Dazu muss eine Vergleichsmethode in der Klasse Mensch definiert werden. Z.B. könnte nach Alter, oder auch nach Namen oder nach einer Kombination von beidem sortiert werden.

Aufgabe 10 - frei: Überlegen Sie sich eigene, sinnvolle weitere Methoden für die Warteschlangen-Implementierung.