Datenstruktur S C H L A N G E

Eine ***Schlange*** ist in der Informatik eine Datenstruktur, bei der man an einem Ende etwas einreihen und am anderen Ende (also Anfang) wieder entnehmen kann. Die Elemente innerhalb der Schlange werden in genau der Reihenfolge entnommen, wie man eingefügt hat. Dieses Prinzip nennt man in der Informatik ***FIFO*** (**F**irst **I**n **F**irst **O**ut). Die Kapazität der Schlange kann begrenzt sein, muss aber nicht.

Aufgabe 1:

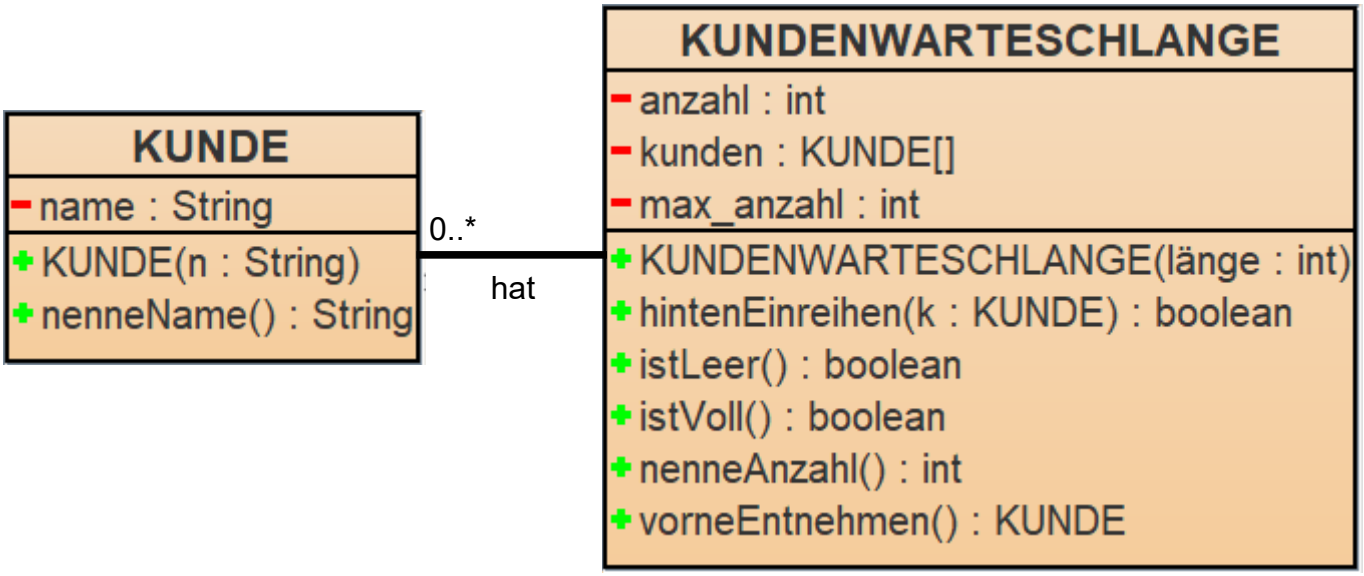
Welche Attribute braucht eine Datenstruktur SCHLANGE?

Welche Methoden benötigt eine Datenstruktur SCHLANGE, um das Prinzip FIFO umzusetzen?

→ Wie funktionieren diese Methoden? → Rollenspiel

Realisierung durch ein Array

Unser erster Versuch wird die Schlange mit Hilfe eines Arrays abbilden. Hier haben wir es also mit einer Schlange von begrenzter Kapazität zu tun. Hierzu erstellen wir zuerst eine Klasse KUNDE und danach eine Klasse KUNDENWARTESCHLANGE nach folgenden Vorlagen:



Aufgabe 2:

a) Öffne das BlueJ-Projekt **Datenstrukturen fester Länge Vorlage.zip** und deklariere in der Klasse KUNDENWARTESCHLANGE die Attribute gemäß des Klassendiagramms.

b) Initialisiere die Attribute im Konstruktor für eine neue, leere Warteschlange.

c) Implementiere die Methoden nenneAnzahl(), istLeer() und istVoll() und überlege dir, wie man diese Information aus den Attributen gewinnen kann.

d) Schreibe anschließend die Methode hintenEinreihen(Kunde k). Überlege dir, wie man „hinten im Array“ den ersten freien Platz finden kann. Falls das Array voll ist, soll false zurückgegeben werden.

e) Schreibe zum Schluss die Methode vorneEntnehmen(). Entnommen wird immer der Kunde ganz vorne auf Platznummer 0, aber die restlichen Kunden sollen dann aufrutschen! Das heißt, dass der Kunde an Index 1 nach dem Entnehmen am Index 0 stehen soll, der Kunde am Index 2 dann auf Index 1, usw...

Einschub: Softwareentwicklung

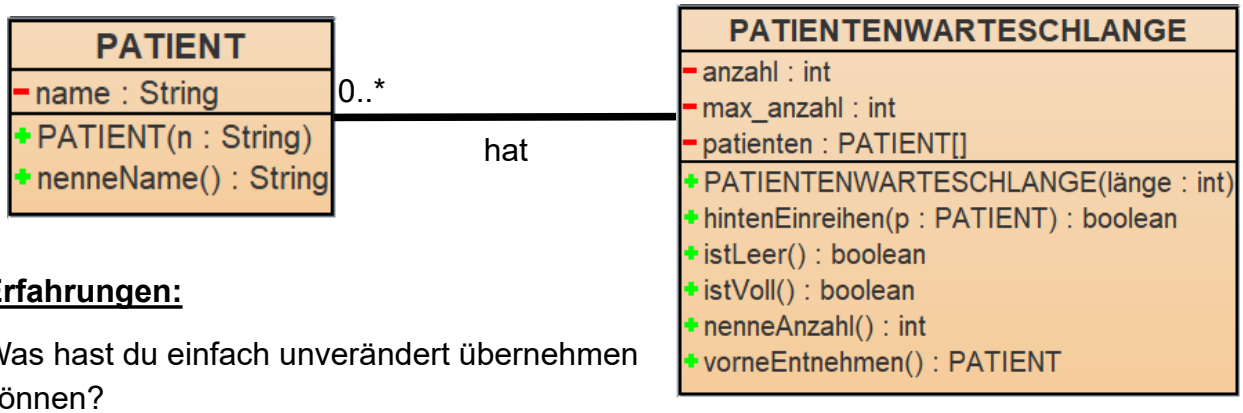
Du sollst im Rahmen des Informatikunterrichts der 12. Jahrgangsstufe eines über das Schuljahr verteilten **Softwareentwicklungspraktikums** sollst du nach und nach verschiedene Aspekte kennenlernen und guten von schlechtem Code unterscheiden lernen. Der ***Entwurf des Softwaredesign*** ist dabei von größter Bedeutung.

Ein Aspekt hierbei ist die ***Wiederverwendbarkeit des Codes***. Schafft man es, so zu modellieren und später zu implementieren, dass man bei einem ähnlichen Problem möglichst viel des vorherigen Codes unverändert wiederverwenden kann, so kann die Entwicklung weiterer (ähnlicher) Softwareprodukte erheblich schneller ablaufen.

Dies wird erreicht, indem man den ***Code modular aufbaut.*** Dies hat auch den Vorteil, dass das gesamte Softwareprodukt **kompatibel** ist und man im Falle einer späteren Änderung des Modells ***einzelne Klassen einfach austauschen kann.***

Aufgabe 3:

Du sollst herausfinden, wie geeignet unser bisheriges Konzept einer Warteschlange bzgl. einer Wiederverwendung in einem ähnlichen Sachzusammenhang ist.
Nutze ***Rechtsklick → Duplicate***, um aus dem KUNDEN eine Klasse PATIENT und aus der KUNDENWARTESCHLANGE eine PATIENTENWARTESCHLANGE zu erstellen:



Erfahrungen:

Was hast du einfach unverändert übernehmen können?

Wo waren kleinere Änderungen nötig?

Was musstest du quasi vollständig neu implementieren?

Verbesserungsmöglichkeiten:

Welcher Teil des Warteschlangenprogramms muss allgemeiner gefasst werden, sodass die Schlange vielseitiger einsetzbar ist?

→ Die Klasse PATENTWARTESCHLANGE kann nach dieser Übung wieder gelöscht werden!

Der Code der Warteschlange ist deshalb ein Code insbesondere deshalb nicht vernünftig wiederverwendet werden konnte, weil die Struktur der SCHLANGE zu fest an die konkrete Klasse KUNDE gebunden war.

Allgemein ist es schlecht, mehrere verschiedene Anliegen in einer Klasse umzusetzen. Es gibt grundsätzlich die ***Prämisse***: **Jede Klasse hat ihre eigene Aufgabe**. So soll die ***Trennung zwischen Struktur*** (SCHLANGE) ***und Inhalt*** (KUNDE, PATIENT, ...) erlangt werden, die wieder einen modularen Aufbau und die Wiederverwendbarkeit gewährleistet.

So sollte sich die SCHLANGE nur um das FIFO-Prinzip kümmern, also um das hinten Einreihen, das vorne Entnehmen und das Aufrücken. Was in der Datenstruktur SCHLANGE gepuffert wird, muss für die Datenstruktur SCHLANGE unerheblich sein.

Aus Sicht eines KUNDEN ist es allerdings unabdingbar, über eine Fähigkeit zu verfügen, sich in einer SCHLANGE zurecht zu finden, denn kein Kunde wird um eine Einreihung in eine SCHLANGE herum kommen.

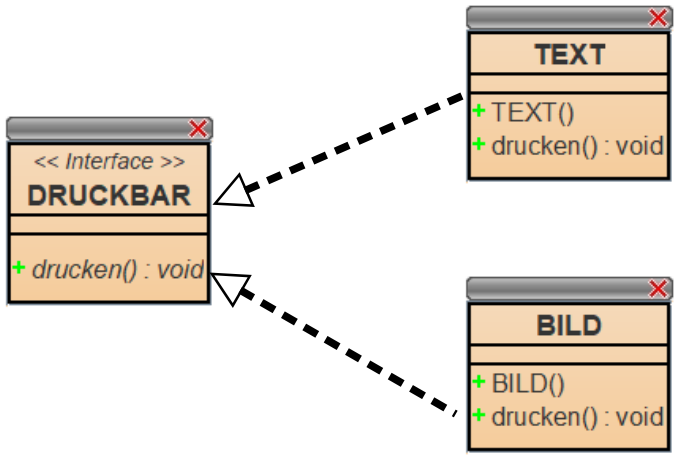
Da aber auch jeder Patient oder jeder Druckauftrag in einer SCHLANGE verwaltet werden können sollte, müssen auch diese über die Fähigkeit des Einreihens in eine SCHLANGE verfügen. All diese Klassen, deren Objekte in eine SCHLANGE eingereiht werden können sollen, haben also ***eine gemeinsame Fähigkeit***. Gemeinsamkeiten werden durch ***Vererbung*** oder durch ***Interfaces*** abgebildet.

Nun haben aber KUNDE, PATIENT und DRUCKAUFTRAG so absolut gar nichts gemeinsam, so dass sich keine sinnvolle Vererbungshierarchie ergibt. Daher bietet sich hier das ***Interface*** an, da sie eh nur eine Fähigkeit gemeinsam haben: sich in eine SCHLANGE einreihen zu lassen.

Interfaces und Referenzen

Ein Interface ist eine ***Schnittstelle***, die Klassen mit dem Interface immer vollständig implementieren müssen. Konkret beinhaltet ein Interface (z.B. DRUCKBAR, ANSTELLBAR, … usw.) ***Methoden(-deklarationen)***, die die Klassen mit diesem Interface für sich selbst entsprechend passend implementieren müssen. So ist sichergestellt, dass jede Klasse mit diesem Interface diese Methoden (z.B. drucken(), … usw.) enthält und diese damit auch aufgerufen werden können.

Im Klassendiagramm wird dies folgendermaßen dargestellt:




```
public interface DRUCKBAR{
    public void drucken();
}
```

```
public class BILD implements DRUCKBAR {

    public void drucken(){
        //TODO
    }
}
```

Der Sinn davon ist , dass jede Klasse diese Methode drucken() entsprechend passend für sich implementiert und diese Implementierungen sich unterscheiden können.

Zudem definiert das Interface (hier: DRUCKBAR) ***einen neuen Datentyp***. Referenzen dieses Typs können auf **alle Objekte referenzieren**, deren Klassen dieses Interface implementieren.

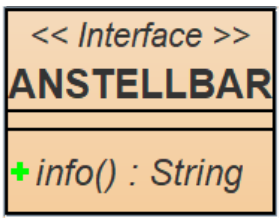
```
DRUCKBAR d1 = new BILD();
DRUCKBAR d2 = new TEXT();
```

Es gibt aber **keine Objekte vom Typ DRUCKBAR**, da DRUCKBAR ein Interface ist und keine Klasse. → nur von Klassen können Objekte erzeugt werden!

Aufgabe 4:

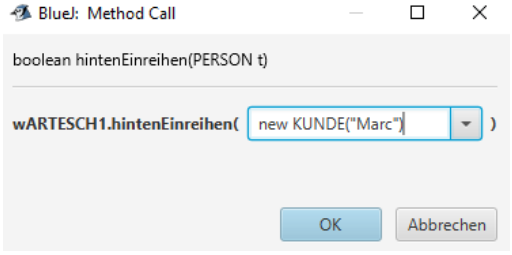
Erweitere dein Projekt **Warteschlange**, um das Interface **ANSTELLBAR** und gib den Interface die Methode **public String info();**, sodass anstellbare Objekte eine Info/Auskunft (z.B. Namen oder Patientenleiden) über sich zurückgeben kann.

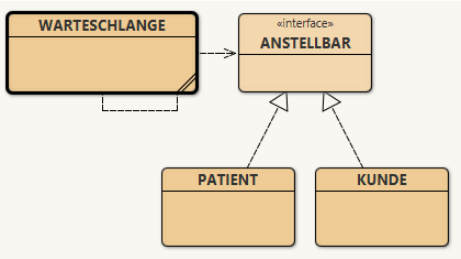
Lass KUNDE und PATIENT das Interface ANSTELLBAR implementieren und ergänze die durch das Interface erzwungene Methode info().



Ändere die Datentypen in der Klasse WARTESCHLANGE auf **ANSTELLBAR**, sodass die Schlange anstellbare Objekte verwalten kann.

Teste, ob die Schlange jetzt noch allgemeiner einsetzbar ist.





Was fällt dir beim Testen der Schlange mit verschiedenen Objekten gleichzeitig (KUNDE, PATIENT) auf? Besitzt die Schlange Möglichkeiten, die so nicht sein sollten?

Zusammenfassung: wann nimmt man was?

Während eine klassische Vererbung (extends KLASSE) ein „ich bin ein“ zum Ausdruck bringt (ein DACKEL ist ein HUND), so verwendet man ein Interface (implements INTERFACE) anstatt eine Oberklasse immer dann, wenn man ein „ich kann ... tun“ zum Ausdruck bringen möchte.

Interfaces	***beschreiben also eine oder mehrere Methoden, die nötig sind, um eine Fähigkeit „nachzurüsten“.***
Oberklassen	***hingegen bringen zum Ausdruck, dass man eine Spezialisierung einer allgemeineren Spezies darstellt.***

Generische Datentypen

Jetzt muss man nur noch der SCHLANGE beibringen, dass sie beliebige Objekte aufnehmen darf (bereits erledigt), aber ***sobald man ein Objekt eines Typ aufgenommen wurde, darf die SCHLANGE nur noch Objekte diesen einen Typs zulassen***. Dies realisiert man in Java mit Hilfe von ***Generischen Datentypen***.

- Ein generischer Datentyp ist ein ***Platzhalter für einen noch unbekannten Datentyp***.

- Der Platzhalter wird in ***dreieckige Klammern*** geschrieben.
  z.B. ***<T>***
- Der Platzhalter muss direkt nach dem Klassennamen erwähnt werden.
  z.B. ***public class SCHLANGE<T>***
- Innerhalb der dreieckigen Klammern kann man **optional** auch noch angeben, dass die unbekannte Klasse von einer anderen Klasse erben oder ein bestimmtes Interface implementieren soll.
  z.B. ***<T extends ANSTELLBAR>***
- Beim Aufruf des Konstruktors einer solchen Klasse muss man dann den konkreten Typ in spitzen Klammern mit angeben. Bei der Instanziierung der Klasse zur Laufzeit (wenn das Programm ausgeführt wird) wird sie an den angegebenen Datentyp (hier: KUNDE) gebunden.
  z.B. ***new SCHLANGE<KUNDE>()***

Aufgabe 5:

Erweitere dein Projekt ***Warteschlange***, um den generischen Datentyp.

Einige Code-Ausschnitte, bei denen generische Typen mit Interfaces Verwendung finden:

```
public class WARTESCHLANGE<T extends ANSTELLBAR> {
  …
  ANSTELLBAR[ ] elemente;
```

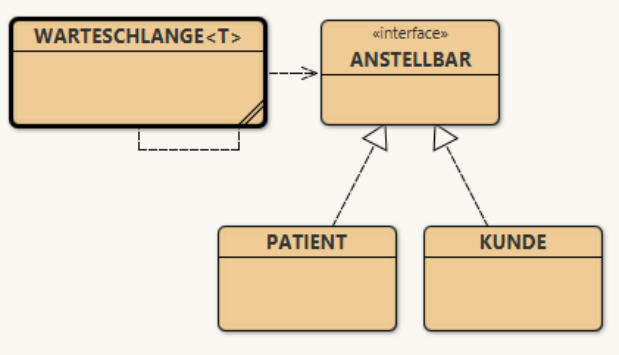
Es gibt in Java kein generisches Array!

```
  …

  public boolean hintenEinreihen(T t) {
     …
  }
  public T vorneEntnehmen() {
       if ( !istLeer() ) {
          T t = (T) elemente[0];
```

Cast (Datentypumwandlung der Referenz) nötig, weil die Referenz elemente[0] vom Typ ANSTELLBAR ist, aber die Rückgabereferenz vom Typ T sein muss

```
          for (int i=0 ; i<elemente.length-1 ; i=i+1) {
             elemente[i] = elemente[i+1];
          }
          elemente[anzahl-1] = null;
          anzahl = anzahl – 1;
          return t;
       } else {   return null;  }}
```



Datentypumwandlung in Java (Cast)

Unter ***implizitem Casting bei Referenzen*** versteht man das Zuweisen ***eines spezielleren Objekt einer allgemeineren Referenz***.
→ funktioniert ohne weiteres Zutun

```
TIER t = new KATZE();
HUND h2 = new SCHÄFERHUND();
```

Unter ***explizitem Casting bei Referenzen*** versteht man die Datentypänderung einer ***allgemeineren Referenz auf eine speziellere***, um auf alle Methoden des Objekts zuzugreifen zu können. Dies funktioniert nur, wenn das Objekt auch entsprechend des spezielleren Datentyp ist.

```
TIER t = new KATZE();
KATZE k = (KATZE) t;
k.schnurren();
```

Ein ***expliziter Cast*** wird erzeugt über: **(*NEUERDATENTYP*)** Objektbezeichner/Referenz

Wichtig: Diese Datentypumwandlung funktioniert nur bei Referenzen, um Zugriff auf die zusätzlichen Attribute und Methoden der Unterklasse zu erhalten. Objekte können ihren Datentyp nicht verändern! → Aus einem Hund kann man keine Katze machen!

Beispiele:

Das allgemeinere Referenz DATENELEMENT soll zur spezielleren CD werden.

→ funktioniert nur, falls das Objekt ursprünglich als CD oder HOERBUCH instanziiert wurde. Sonst käme es zu einer ClassCastException nicht beim Kompilieren sondern während der Laufzeit des Programms, eben dann, wenn der Fehler auftritt.

```
DATENELEMENT d1 = new BUCH(1,"BUCH_autor","BUCH_titel");
DATENELEMENT d2 = new DVD(2,"DVD_titel",12, 120.2);
DATENELEMENT d3 = new CD(3,"CD_titel",80.3);
DATENELEMENT d4 = new HOERBUCH(4,"HOERBUCH_titel",3.2,"HOERBUCH_sprecher");

BUCH buch1 = (BUCH)d1;
DVD dvd1 = (DVD)d2;
CD cd1 = (CD)d3;
HOERBUCH hb1 = (HOERBUCH)d4;

System.out.println(buch1.nenneTitel());
System.out.println(dvd1.nenneFSK());
System.out.println(cd1.nenneLaenge());
System.out.println(hb1.nenneSprecher());
```

CD cd = new CD(4,“CD_Titel“,80.3);

HOERBUCH hoerbuch = (HOERBUCH) cd; → ***ClassCastException***

Ausnahme bei Zahlen/primitiven Datentypen:

Casting bei primitiven Datentypen (int, double, char, boolean) hängt an der Interpretation der Codierung des jeweiligen Datentyps.

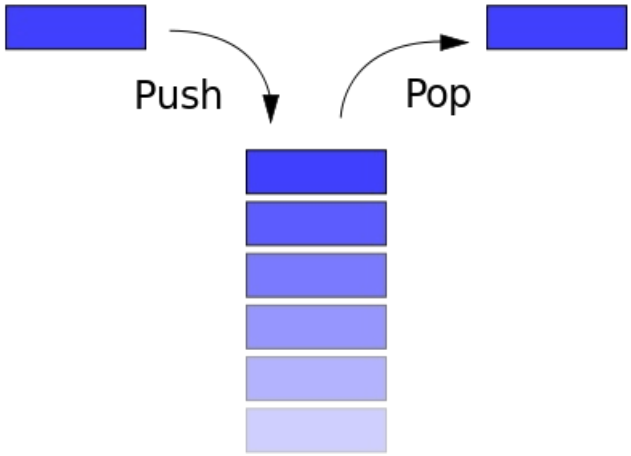
double zahl = 5.9876;

int d = (int) zahl; → d = 5

(Falls eine Fließkommazahl zu einer Ganzzahl gecastet wird, gehen die Nachkommastellen verloren. Diese Information kann eine Ganzzahl nicht speichern. → keine ClassCastException, denn Zahlen sind primitive Datentypen und keine Klasse)

Datenstruktur S T A P E L

Man nennt den ***Stack*** auch ***Keller*** oder ***Stapel***. Hierbei handelt es sich um eine Datenstruktur, die tatsächlich einem Stapel Papier oder einem Kartenstapel gleicht. Man legt jedes neue Element oben drauf. Möchte man nun wieder ein Element entfernen, so muss man das oberste nehmen. Es kommt also das zuletzt hinein gelegte Element als erstes wieder heraus. Das Verfahren dieser Datenstruktur nennt man auch ***LIFO*** (**L**ast **I**n **F**irst **O**ut). Ein Stack kann z.B. dazu verwendet werden, um die Reihenfolge von Elementen umzukehren.



Realisierung durch ein Array

Betrachten wir den Stack zunächst wieder als durch ein Array realisiert, so stellen wir fest, dass der Stack nichts anderes ist als unsere SCHLANGE, nur dass man die Elemente nicht am anderen sondern am selben Ende (vorne) wieder herausnimmt.

Deshalb könnten wir den Stack realisieren, indem wir unsere bisherige Datenstruktur SCHLANGE schlichtweg um eine Methode **vorneEinreihen()** erweitern.

Diese Realisierung widerspricht aber dem Prinzip: **Jede Klasse hat genau eine Aufgabe.**

Aufgabe 6:

Dupliziere deine Klasse WARTESCHLANGE → STAPEL und lösche die Methode hintenEinreihen(…) und implementiere dafür die Methode vorneEinreihen(…)!
Füge zwei KUNDEN in den STAPEL ein und entnehme sie anschließend wieder.
Kommen sie in umgekehrter Reihenfolge wieder heraus?