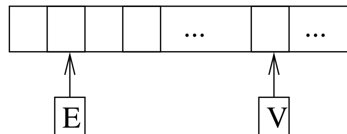
### 7.2.5 Zusammenfassung

Programmiersprachen müssen zur nebenläufigen Programmierung folgendes bereitstellen:

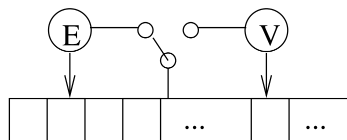
1. Konstrukte zur Erzeugung neuer Prozesse (Ada: `task`, C/Unix: `fork`, Go: `go`, …)
2. Konstrukte zur Kommunikation/Synchronisation zwischen Prozessen

Es folgt eine Veranschaulichung von Synchronisationsmechanismen für einen Puffer (hierbei ist E ein Erzeuger und V ein Verbraucher):

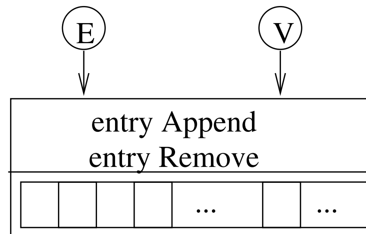
#### Direktzugriff ohne Synchronisation



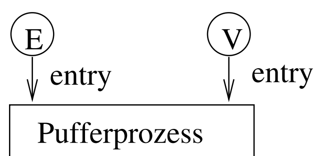

#### Semaphor



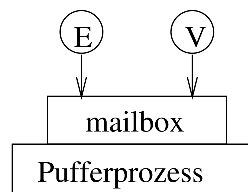

#### Monitor




#### Rendezvous




#### Nachrichtenaustausch



## 7.3 Nebenläufige Programmierung in Java

Java wurde ursprünglich als Programmiersprache für das Internet eingeführt und enthält daher Möglichkeiten zur nebenläufigen Programmierung:

1. Zum Arbeiten mit Prozessen gibt es die Standardklasse `Thread`. Dies bedeutet, dass man eigene Prozesse durch Klassen definieren kann, die als Unterklasse von `Thread` eingeführt werden.

2. Für die Synchronisation wird ein Monitor-ähnliches Konzept auf der Ebene von Klassen verwendet: jedes Objekt kann als Monitor agieren, wenn alle Attribute `private` und alle Methoden `synchronized` sind.

Wir wollen uns im folgenden diese Aspekte genauer anschauen.

### 7.3.1 Prozesse in Java: Threads

Analog zu anderen Basisklassen wie `String` gibt es eine Klasse `Thread`, deren Objekte ablaufbare Prozesse sind. Diese entsprechen nicht unbedingt Betriebssystem-Prozessen, da die JVM Threads selbst verwalten kann. Wichtige Methoden der Klasse `Thread` sind:

`start()` startet einen neuen Thread, wobei in diesem die Methode `run()` ausgeführt wird.

`run()` ist die Methode, die ein Thread ausführt. Diese wird üblicherweise in Unterklassen redefiniert. Wenn die Ausführung von `run()` terminiert, wird der Thread beendet.

`sleep(long ms)` stoppt den Thread für `ms` Millisekunden.

**Beispiel 7.8** (Threads in Java)**.** Wir betrachten einen Druckprozess

```
class PrintThread extends Thread {
  public void run () {
    <Anweisungen zum Drucken>
  }
}
```

Starten des Druckprozesses:

```
new PrintThread().start();
```

Eine weitere Möglichkeit zur Erzeugung von Threads ist die Implementierung des Interfaces `Runnable`, was insbesondere dann sinnvoll ist, wenn die eigene Klasse schon von einer anderen Klassen erbt und daher nicht auch eine Unterklasse von `Thread` sein kann. Das Interface `Runnable` enthält nur die Methode `run()`. Um dies zu nutzen, kann man einen Thread auch mit dem Konstruktor `Thread(Runnable target)` erzeugen:

```
class PrintSpooler implements Runnable {
    public void run() { ... };
}
```

Starten des Druckprozesses:

```
new Thread(new PrintSpooler()).start();
```

### 7.3.2 Synchronisation in Java

Die Synchronisation in `Java` ist in der Klasse `Object` verankert: Jedes Objekt kann zur Synchronisation verwendet werden und enthält zu diesem Zweck eine Sperre (lock). Auf die Sperre kann man allerdings nicht explizit zugreifen, sondern hierfür gibt es eine Synchronisationsanweisung:

```
synchronized (expr) statement
```

Die Bedeutung dieser Anweisung ist:

1. Werte `expr` zu einem Objekt `o` aus.
2. Falls `o` nicht gesperrt ist, sperre `o`, führe `statement` aus, entsperre `o`.
3. Falls `o` vom gleichen Prozess gesperrt ist, führe `statement` aus.
4. Falls `o` von einem anderen Prozess gesperrt ist, warte, bis `o` entsperrt ist.

Anstelle der Synchronisation einzelner Anweisungen/Blöcke kann man auch (strukturierter!) Methoden synchronisieren:

```
synchronized τ method(...) {
  Rumpf
}
```

Dies entspricht dann folgender Definition:

```
τ method(...) {
  synchronized(this) { Rumpf }
}
```

Der Rumpf der Methode `method` wird somit nur ausgeführt, falls man die Sperre auf dem Objekt `o` bekommt oder schon hat.

Damit können wir einen Monitor-orientierten Stil realisieren, wenn wir folgende Prinzipien beachten (dies ist auch empfehlenswert wegen der guten Programmstruktur):

- alle Attribute sind als `private` deklariert
- alle nicht `private`-Methoden sind `synchronized`

Dies hat dann den Effekt, dass nur ein Prozess gleichzeitig Attribute eines Objektes verändern kann.

An dem Puffer-Beispiel haben wir gesehen, dass auch die **Suspension** von Prozessen notwendig ist, falls bestimmte Bedingungen nicht erfüllt sind. Zu diesem Zweck hat in Java jedes Objekt eine *Prozesswarteschlange*, die zwar nicht direkt zugreifbar ist, aber die mit folgenden Methoden beeinflusst werden kann:

`o.wait()` suspendiert den aufrufenden Prozess und gibt gleichzeitig die Sperre auf das Objekt `o` frei.

`o.notify()` aktiviert einen(!) Prozess, der mit `o.wait()` vorher suspendiert wurde. Dieser Prozess muss sich dann wieder um die frei gegebene Sperre bewerben. Falls er diese erhält, kann dieser arbeiten.

`o.notifyAll()` aktiviert alle diese Prozesse.

Eine typische Anwendung dieser Methoden innerhalb von synchronisierten(!) Objektmethoden sieht wie folgt aus:

- `wait()` in `while`-Schleife, bis die Bedingung zur Ausführung erfüllt ist:

```
synchronized void doWhen() {
    while (!<bedingung>) wait();
    <Hier geht es richtig los>
}
```

- `notify()` durch Prozess, der den Objektzustand so verändert, dass ein wartender Prozess evtl. etwas machen und daher aktiviert werden kann.

```
synchronized void change() {
    <Zustandsaenderung>
    notify(); // auch: notifyAll()
}
```

Wichtig: Die Reihenfolge ist beim Aktivieren nicht festgelegt (d.h. es ist keine wirkliche Warteschlange), daher sollte `notifyAll()` verwendet werden, falls möglicherweise mehrere Prozesse warten und es relevant ist, welcher aufgeweckt wird.

**Beispiel 7.9** (Synchronisierter Puffer). Einen *synchronisierter Puffer* könnten wir in Java wie folgt implementieren:

```
class Buffer {
  private int n;          // Pufferlaenge
  private int[] contents; // Pufferinhalt
  private int num, ipos, opos = 0;
```

```
  public Buffer (int size) {
    n = size;
    contents = new int[size];
  }

  public synchronized void append (int item)
      throws InterruptedException {
    while (num==n) wait();
    contents[ipos] = item;
    ipos = (ipos+1)%n;
    num++;
    notify(); // oder: notifyAll()
  }

  public synchronized int remove () throws InterruptedException {
    while (num==0) wait();
    int item = contents[opos];
    opos = (opos+1)%n;
    num--;
    notify(); // oder: notifyAll()
    return item;
  }
}
```

Prozesse, die für einige Zeit inaktiv sind (die z. B. mittels `sleep()` oder `wait()` auf Ereignisse warten), können die Ausnahme `InterruptedException` werfen, falls sie durch einen Interrupt unterbrochen werden. Aus diesem Grund muss die `InterruptedException` bei Benutzung von `sleep()` oder `wait()` auch behandelt oder deklariert werden. Üblicherweise kann die Behandlung dieser Ausnahmen durch direkte Beendigung erfolgen (s. u.), aber man könnte auch noch offene Dateien schließen oder ähnliche „Aufräumaktionen“ durchführen.
Ein Erzeuger für einen Puffer könnte dann so aussehen:

```
class Producer extends Thread {
  private Buffer b;

  public Producer (Buffer b) { this.b = b; }

  public void run() {
    try {
      for (int i = 1; i <= 10; i++) {
        System.out.println("into buffer: " + i);
        b.append(i);
        sleep(5);
```

```
      }
    }
    catch (InterruptedException e)
      { return; }  // terminate this thread
  }
}
```

Ein Verbraucher für einen Puffer könnte die folgende Form haben:

```
class Consumer extends Thread {
  private Buffer b;

  public Consumer (Buffer b) { this.b = b; }

  public void run() {
    try {
      for (int i = 1; i <= 10; i++) {
        System.out.println("from buffer: " + b.remove());
        sleep(20);
      }
    }
    catch (InterruptedException e)
      { return; } // terminate this thread
  }
}
```

Das Starten der Erzeugers und Verbrauchers kann wie folgt passieren:

```
Buffer b = new Buffer(4);
new Consumer(b).start();
new Producer(b).start();
```