# 10 Exchanger und BlockingQueue

Sollten Threads miteinander kommunizieren, so können sie sich z. B. gegenseitig referenzieren und entsprechende Methoden aufrufen. Diese Art der Kommunikation erfordert aber einen abgestimmten Ablauf. Neben dieser methodenbasierten, synchronen Kommunikation gibt es auch die Möglichkeit einer nachrichtenbasierten. Hierbei können ein oder mehrere Threads Nachrichten (*messages*) in eine spezielle Datenstruktur (häufig eine *FIFO-Queue*) stellen, die von einem oder mehreren anderen Teilnehmern ausgelesen und abgearbeitet werden. Das Senden und die Abarbeitung geschehen hierbei asynchron.

Ein Spezialfall ist der synchrone Austausch von Daten zwischen zwei Threads. Das Konzept funktioniert wie bei einem gewöhnlichen Tauschgeschäft. Die beiden Teilnehmer treffen sich und tauschen dabei ihre Gegenstände. Sie müssen dabei ggf. aufeinander warten.

Java bietet für die synchrone, nachrichtenbasierte Kommunikation die Klasse `Exchanger` und für die asynchrone verschiedene Implementierungen des Interface `BlockingQueue` an.

## 10.1 Exchanger

Ein `Exchanger` entspricht einem synchronen Austauschkanal (oft auch als *Rendezvous-Punkt* bezeichnet) zwischen zwei Teilnehmern. Über die `exchange`-Methode können die beiden Beteiligten zeitgleich ihre Objekte austauschen. Ist einer der Teilnehmer noch nicht bereit, muss der andere warten. Man kann mit dem Konzept eine spezielle Variante des Erzeuger-Verbraucher-Musters (*producer consumer pattern*) mit genau einem Erzeuger und einem Verbraucher realisieren. Anders als die Lösung mit der `BlockingQueue` (siehe unten) wartet der Erzeuger ggf. so lange, bis sein Produkt abgenommen wird.

Abbildung 10-1 zeigt schematisch die Arbeitsweise. Thread 1 hat seine Aufgabe erledigt und die auszutauschenden Daten in ein Objekt abgelegt (*data1*). Er ruft die `exchange`-Methode auf und wartet, bis ein anderer mit ihm das Objekt tauscht. Thread 2 übernimmt von Thread 1 dessen Objekt,

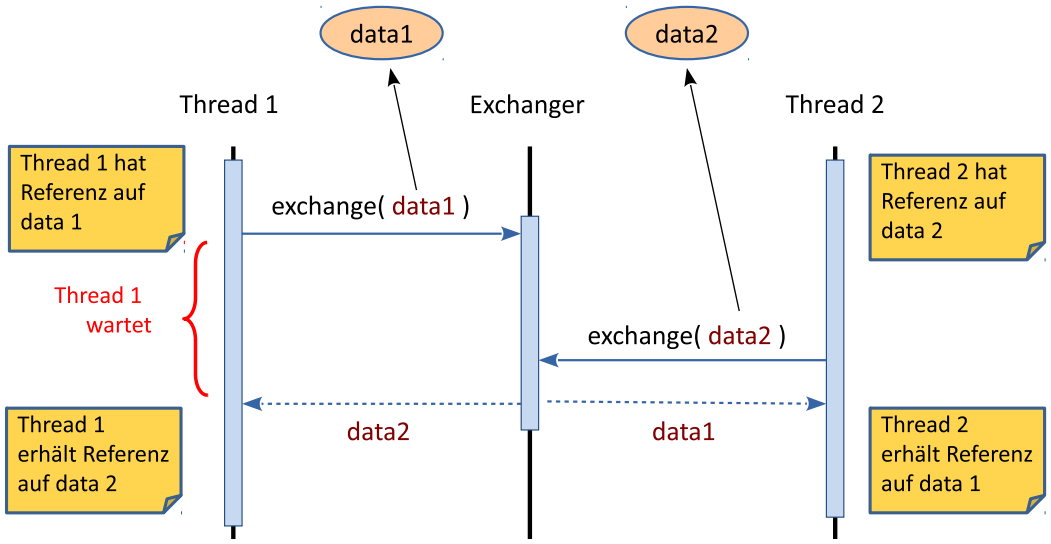


**Abbildung 10-1:** Die Funktionsweise des Exchanger

Thread 1 erhält im Gegenzug eines (vom selben Typ) von Thread 2. Danach können beide unabhängig weiterarbeiten.

Zur Veranschaulichung betrachten wir ein Beispiel. Der Erzeuger (Klasse `RandomStringProducer`) füllt ein Array mit Zufallsstrings, das er über einen `Exchanger` mit dem Array des Verbrauchers (Klasse `RandomStringConsumer`) austauscht. Der Empfänger wertet daraufhin die Häufigkeit der vorkommenden Buchstaben aus.

Codebeispiel 10.1 zeigt die Implementierung der Klasse `RandomStringProducer`. Ein `RandomStringProducer`-Objekt erhält über das Konstruktor-Argument Zugriff auf einen `Exchanger` (❶). Nachdem ein internes String-Array gefüllt wird, wird es an den `Exchanger` (❷) übergeben. Im Gegenzug erhält der Erzeuger ein String-Array zurück, dessen Inhalt er überschreibt. Das Ende des Austauschs wird durch die Übergabe von `null` signalisiert (❸).

```
public class RandomStringProducer implements Runnable
{
  private final Exchanger<String[]> exchanger = null;
  private String[] data = new String[100];

  public RandomStringProducer(Exchanger<String[]> exchanger)      ❶
  {
     this.exchanger = exchanger;
  }


  public void run()
  {
```

```
    try
    {
      for (int j = 0; j < 10; j++)
      {
        for (int i = 0; i < data.length; i++)
        {
          data[i] = Util.getRandomString(100);
        }

        // Rendezvous-Punkt
        data = exchanger.exchange(data);                          ❷
      }

      // Signalisiert das Ende des Austauschs
      exchanger.exchange(null);                                   ❸
    }
    catch (InterruptedException exce)
    {
      exce.printStackTrace();
    }
  }
}
```

**Codebeispiel 10.1:** Erzeuger-Klasse mit einem Exchanger

Die Erzeugung von Zufallsstrings kann wie folgt geschehen:

```
public class Util
{
  public static final String BUCHSTABEN                           ❶
                    = "abcdefghijklmnopqrstuvwxyzüöäß";

  public static String getRandomString(int len)
  {
    StringBuilder sb = new StringBuilder(len);
    for (int i = 0; i < len; i++)
    {
      int idx = ThreadLocalRandom.current()                       ❷
                                 .nextInt(BUCHSTABEN.length());
      sb.append(BUCHSTABEN.charAt(idx));
    }
    return sb.toString();
  }
}
```

Das Attribut `BUCHSTABEN` enthält den verwendeten Zeichenvorrat (❶). Es werden dann daraus zufällig Buchstaben ausgewählt (❷).

Codebeispiel 10.2 zeigt die Implementierung des Verbrauchers. Auch er erhält Zugriff auf einen `Exchanger`. Am Rendezvous-Punkt tauscht er sein String-Array mit dem Erzeuger aus (❷). Erhält er eine `null`-Referenz, beendet er seine Arbeit (❸) und gibt sein Ergebnis zurück (Future-Pattern).

```
public class RandomStringConsumer implements Callable<int[]>
{
  private Exchanger<String[]> exchanger = null;
  private String[] data = new String[100];

  public RandomStringConsumer(Exchanger<String[]> exchanger)      ❶
  {
    this.exchanger = exchanger;
  }

  public int[] call()
  {
    try
    {
      int[] charFrequency
           = new int[Util.BUCHSTABEN.length()];

      while (true)
      {
        // Rendezvous-Punkt
        data = exchanger.exchange(data);                          ❷

        if (data == null)                                         ❸
        {
           break;
        }
        else
        {
          for (int i = 0; i < data.length; i++)
          {
            String str = data[i];
            for (int j = 0; j < str.length(); j++)
            {
              char c = str.charAt(j);
              int pos = Util.BUCHSTABEN.indexOf(c);
              if (pos != -1)
                charFrequency[pos]++;
            }
          }
        }
      }
      return charFrequency;
    }
    catch (Exception exce)
    {
      exce.printStackTrace();
      return null;
    }
  }
}
```

**Codebeispiel 10.2:** Verbraucher-Klasse mit einem Exchanger

Codebeispiel 10.3 zeigt die Verwendung der einzelnen Komponenten. Ein `Exchanger` mit entsprechender Typisierung wird erzeugt (❶) und an den Erzeuger und Verbraucher als Konstruktor-Argument übergeben. Danach wird der Erzeuger in einem eigenen Thread gestartet (❷). Der Verbraucher wird ebenfalls nebenläufig gestartet, wobei die Rückgabe über ein `Future`-Objekt organisiert wird (❸). Durch den Aufruf der `get`-Methode wird auf das Ergebnis der »Häufigkeitsanalyse« gewartet (❹).

```
ExecutorService executor = Executors.newFixedThreadPool(2);

Exchanger<String[]> exchanger = new Exchanger<String[]>();          ❶

RandomStringProducer producer = new RandomStringProducer(exchanger);
executor.submit(producer);                                          ❷

RandomStringConsumer consumer = new RandomStringConsumer(exchanger);
Future<int[]> result = executor.submit(consumer);                   ❸

int[] frequency = result.get();                                     ❹
executor.shutdown();

for(int i=0; i < frequency.length; i++)
{
  System.out.println( Util.BUCHSTABEN.charAt(i)  + " : "
                      + frequency[i]);
}
```

**Codebeispiel 10.3:** Das Hauptprogramm für das Exchanger-Beispiel

## 10.2 Queues

Warteschlangen (*Queues*) sind oft benutzte Datenstrukturen. Warteschlangen sollten immer dann zum Einsatz kommen, wenn mehr Anfragen bzw. Anforderungen pro Zeiteinheit an ein System gesendet werden, als es in derselben Zeit verarbeiten kann. Diese Situation tritt z. B. beim Erzeuger-Verbraucher-Muster auf. Java stellt verschiedene Varianten von Warteschlangen zur Verfügung, die in Multithreaded-Anwendungen eingesetzt werden können. Abbildung 10-2 zeigt die Interface-Hierarchie. Neben der `BlockingQueue` existieren noch die Varianten `TransferQueue` und `BlockingDeque` (`Deque`, *double ended queue*).

Die wichtigsten Methoden von `Queue` und ihrer Erweiterung `BlockingQueue` für das Einfügen und Entnehmen von Elementen sind in Tabelle 10-1 aufgelistet. Darunter sind auch die folgenden Methoden, generisch typisiert durch `E`:

- `boolean offer(E e)`: Fügt ein Element am Ende der Queue ein. Die Rückgabe gibt an, ob die Operation erfolgreich war. Die Rückgabe ist insbesondere bei platzbeschränkten Queues wichtig. Durch `false` wird signalisiert, dass das Element nicht aufgenommen werden konnte.
- `boolean offer(E e, long timeout, TimeUnit unit)`: Die Wirkung ist wie bei `offer(E e)` mit dem Unterschied, dass die maximale Wartezeit durch die beiden letzten Parameter spezifiziert wird.
- `E poll()`: Entnimmt ein Element vom Anfang der Queue. Liefert `null`, falls kein Element vorhanden ist.
- `E poll(long timeout, TimeUnit unit)`: Die Wirkung ist wie bei `poll()`, wobei hier die maximale Wartezeit angegeben wird.
- `void put(E e)`: Fügt ein Element in die Queue ein und wartet ggf., bis ein entsprechender Platz in der Queue vorhanden ist.
- `E take()`: Entnimmt ein Element vom Anfang der Queue und wartet (blockiert) ggf., bis ein Element vorhanden ist.

Blockierende Methoden, also `take`, `put` und alle Methoden mit einer Wartezeitangabe, werfen bei einer Unterbrechung eine `InterruptedException`.

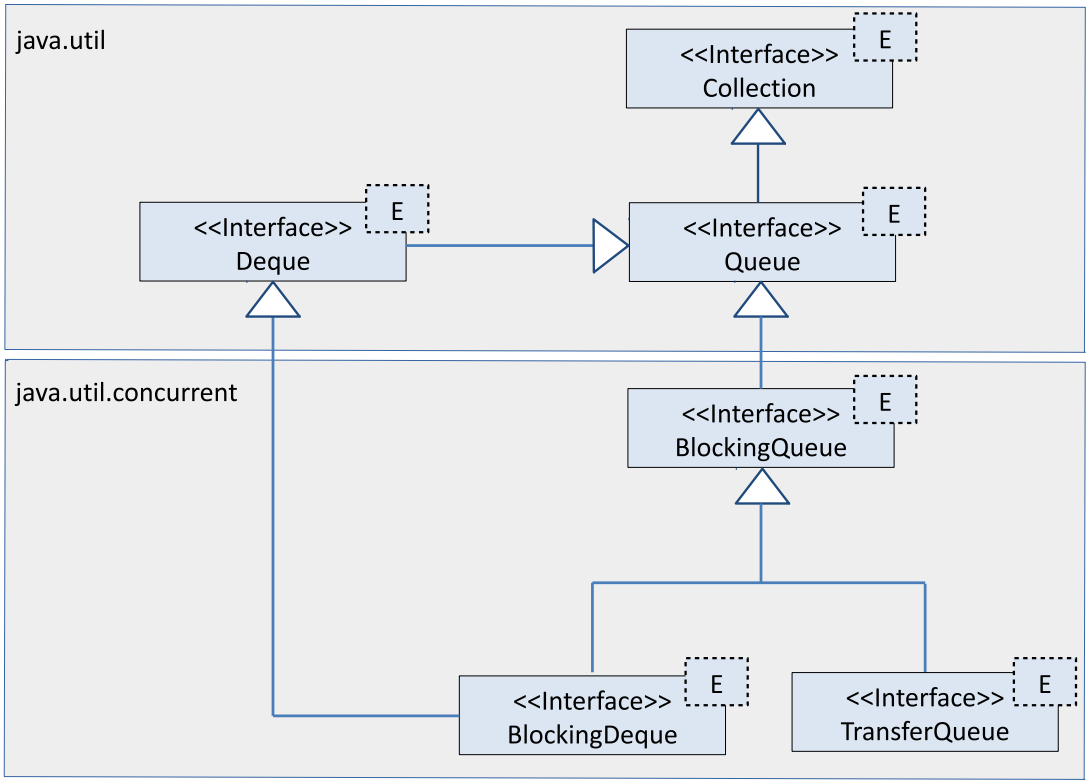


**Abbildung 10-2:** Hierarchie der Queue-Klassen

| | Mit Exception | Nicht blockierend | Blockierend | Timeout |
|---|---|---|---|---|
| Einfügen | `add(e)` | `offer(e)` | `put(e)` | `offer(e, time, unit)` |
| Auslesen | `remove()` | `poll()` | `take()` | `poll(time, unit)` |
| Überprüfen | `element()` | `peek()` | nicht def. | nicht def. |

**Tabelle 10-1:** Wichtige Methoden von `BlockingQueue`

Für die `BlockingQueue` gibt es je nach Einsatzzweck verschiedene Implementierungen (vgl. Abb. 10-3). Mit Ausnahme von `PriorityQueue` und deren Verwandten werden Elemente immer am Ende der Queue eingefügt, entnommen werden sie immer am Beginn (FIFO-Prinzip).

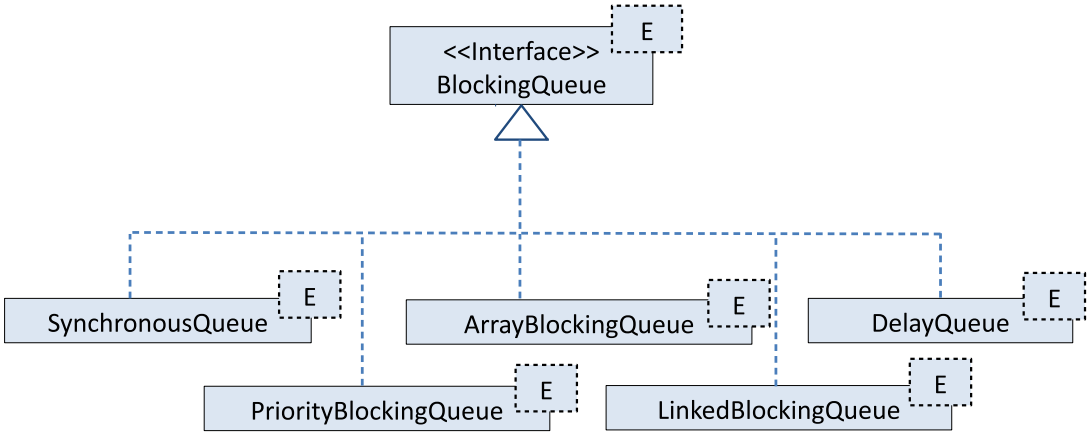


**Abbildung 10-3:** Implementierungen des Interface `BlockingQueue`

**ArrayBlockingQueue<E>** ist eine Queue mit einer festen Größe (Kapazität). Intern wird ein klassischer beschränkter Ringpuffer verwendet.

**LinkedBlockingQueue<E>** existiert sowohl als kapazitätsbeschränkte als auch als unbeschränkte Queue. Der Name deutet auch darauf hin, dass sie mithilfe einer (doppelt) verketteten Liste implementiert ist.

**DelayQueue<E>** kann nur Objekte aufnehmen, deren Klasse das Interface `Delayed` implementiert. Für die interne Organisation werden die Methoden `compareTo` und `getDelay` verwendet.

**PriorityBlockingQueue<E>** sortiert mithilfe der `compareTo`-Methode bzw. mit dem explizit angegebenen `Comparator`-Objekt ihre verwalteten Elemente.

**SynchronousQueue<E>** ist eine blockierende Queue, bei der die beiden beteiligten Threads aufeinander warten müssen. Zu bemerken ist, dass eine `SynchronousQueue` keine Kapazität hat. Die Kommunikation der beiden Partner muss wie bei `Exchanger` synchron stattfinden. Einige Methoden des `BlockingQueue`-Interface (wie `poll`, `peek` etc.) haben keine Bedeutung (sie liefern zum Beispiel immer `null` zurück).

Mithilfe einer Queue kann das Erzeuger-Verbraucher-Muster sehr einfach realisiert werden.

### Hinweis

Das Interface `Queue` erweitert `Collection` und bietet daher auch dessen Methode `add` zum Aufnehmen eines Elements in den Container an. Im Unterschied zu `offer` wird `add` eine `IllegalException` auslösen, wenn das Einfügen nicht möglich ist. Da in der Praxis Queues mit einer festen Größe der Normalfall sind, sollte die Methode `offer` bevorzugt werden. Analog löst `remove` im Vergleich zu `poll` im Falle einer leeren Queue eine `NoSuchElementException` aus.

## 10.3 Das Erzeuger-Verbraucher-Muster

Das Erzeuger-Verbraucher-Muster (*producer consumer pattern*) entkoppelt zwei Tasks, die nebenläufig ausgeführt werden. Der erste Task stellt dabei Daten für den zweiten zur weiteren Verarbeitung zur Verfügung, wobei der Austausch über eine Queue stattfindet (vgl. Abb. 10-4).

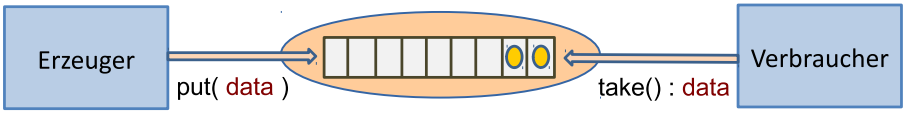


**Abbildung 10-4:** Funktionsweise des Erzeuger-Verbraucher-Musters

Durch die Verwendung einer Queue kann die Reihenfolge der Verarbeitung der Daten erhalten bleiben. In der einfachsten Variante, wie in Abbildung 10-4, ist das Muster nicht skalierbar, d.h., die Queue kann sich als Flaschenhals erweisen.

Als Anwendungsbeispiel betrachten wir eine Variante der obigen Häufigkeitsanalyse von Buchstaben bei Zufallsstrings. Zur Kommunikation kommt statt eines `Exchanger` eine `ArrayBlockingQueue` zum Einsatz.

Das Ende wird durch ein definiertes Stopp-Token (*poison pill*) signalisiert (❶). Der Unterschied zur obigen Implementierung ist, dass der Erzeuger im Normalfall nicht warten muss. Er wird nur bei einer vollen Queue blockiert. Der Verbraucher muss blockiert pausieren, wenn es kein Element zu entnehmen gibt.

```
public class RandomStringProducer implements Runnable
{
  private final BlockingQueue<String> queue = null;
  private final String endToken = null;

  public RandomStringProducer(BlockingQueue<String> queue,
                              String endToken)
  {
    this.endToken = endToken;
    this.queue = queue;
  }

  public void run()
  {
    try
    {
      for (int i = 0; i < 2000; i++)
      {
        queue.put(Util.getRandomString(200000));
      }
      queue.put( endToken );                                    ❶
    }
    catch (InterruptedException ex)
    {
      ex.printStackTrace();
    }
  }
}
```

**Codebeispiel 10.4:** Ein Erzeuger für Zufallsstrings

Listing 10.5 zeigt die Implementierung des Verbrauchers. Er liest die Zufallsstrings aus der Queue aus und analysiert die Häufigkeiten der Kleinbuchstaben, bis er auf das Stopp-Token stößt (❶).

```
public class RandomStringConsumer implements Callable<int[]>
{
  private final BlockingQueue<String> queue = null;
  private final String endToken = null;

  public RandomStringConsumer(BlockingQueue<String> queue, String
      endToken)
  {
    this.queue = queue;
    this.endToken = endToken;
  }
```

```
  public int[] call()
  {
    try
    {
      int[] charFrequency = new int[Util.BUCHSTABEN.length()];

      String str = null;
      while (true)
      {
        str = queue.take();
        if (str.equals(endToken))  break;                    ❶

        count(charFrequency, str);
      }

      return charFrequency;
    }
    catch (Exception ex)
    {
      ex.printStackTrace();
      return null;
    }
  }

  private static void count(int[] charFrequency, String str)
  {
    for (int i = 0; i < str.length(); i++)
    {
      char c = str.charAt(i);
      int pos = Util.BUCHSTABEN.indexOf(c);
      if (pos != -1)
        charFrequency[pos]++;
    }
  }
}
```

**Codebeispiel 10.5:** Ein Verbraucher zur Ermittlung der Buchstabenhäufigkeiten

Codebeispiel 10.6 zeigt die Verwendung der beiden Klassen. Das Beispiel benutzt den `ExecutorService`, über den der Erzeuger und der Verbraucher gestartet werden. Die beiden erhalten als Argumente die `BlockingQueue` und das Stopp-Token (❶). Da der Verbraucher einen Wert zurückliefert, wird hier ein `Future` eingesetzt (❷).

```
final String END_TOKEN = "_STOPP";

ExecutorService executor = Executors.newFixedThreadPool(2);
BlockingQueue<String> queue = new ArrayBlockingQueue<String>(1000);

RandomStringProducer producer
         = new RandomStringProducer(queue, END_TOKEN);       ❶
executor.submit(producer);
```

```
RandomStringConsumer consumer
         = new RandomStringConsumer(queue, END_TOKEN);          ❶
Future<int[]> result = executor.submit(consumer);               ❷

// Ergebnis holen und ausgeben
int[] freq = result.get();
executor.shutdown();

for (int i = 0; i < freq.length; i++)
{
  System.out.println(Util.BUCHSTABEN.charAt(i) + " : " + freq[i]);
}
```

**Codebeispiel 10.6:** Das Hauptprogramm für das Erzeuger-Verbraucher-Beispiel

## 10.4 Varianten

Die Einsatzmöglichkeiten einer `BlockingQueue` sind sehr vielfältig. Neben dem klassischen Erzeuger-Verbraucher-Muster gibt es verschiedene Varianten bzw. Ausbauformen, von denen im Folgenden einige vorgestellt werden.

### 10.4.1 Pipeline von Erzeugern und Verbrauchern

Erzeuger und Verbraucher können auf vielfältige Art und Weise zusammengefügt werden. Abbildung 10-5 zeigt den Aufbau einer Verarbeitungskette. Das mittlere Glied ist hier sowohl Verbraucher als auch Erzeuger und hat den Zugang zu zwei `Queue`-Objekten.

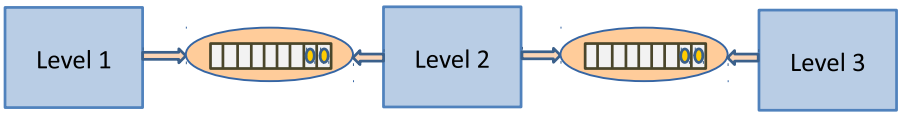


**Abbildung 10-5:** Kette aus Erzeugern und Verbrauchern

Abbildung 10-6 zeigt, wie das Muster »vertikal« skaliert werden kann. An eine `Queue` können sowohl mehrere Erzeuger als auch Verbraucher angeschlossen werden. Hierbei ist nun zu beachten, dass am Ende der Verarbeitung ggf. so viele *End-Tokens* in die `Queue` geschrieben werden, wie Verbraucher vorhanden sind. Wenn die Teilnehmer dynamisch hinzugefügt oder entfernt werden, empfiehlt es sich, dies über eine zentrale Klasse zu steuern, sodass bei Bedarf die richtige Anzahl von *End-Tokens* gesendet werden kann.

Aus den beiden obigen Varianten lassen sich nun auch komplexere Strukturen aufbauen. Abbildung 10-7 zeigt eine Verarbeitungskette, bei der der mittlere Verarbeitungsschritt *skaliert* ist. Dies ist dann sinnvoll, wenn

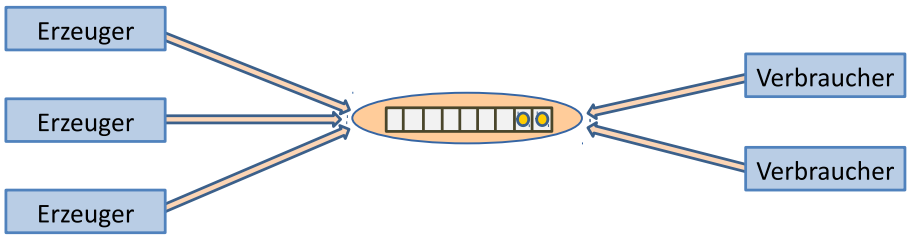


**Abbildung 10-6:** Skalierungsmöglichkeiten durch den Einsatz von parallelen Erzeugern und Verbrauchern

blockierende Aufrufe, wie z. B. Netzwerk-Requests vorkommen. Durch die *vertikale* Skalierung kann der Durchsatz erhöht werden. Man beachte, dass dadurch die Reihenfolge der Ergebnisse nicht mehr mit der der Aufträge übereinstimmt. Bei vielen Anwendungen stellt das keine Limitierung dar. Bei Bedarf kann aber z. B. durch die Verwendung von eindeutigen Auftrags-IDs am Ende die ursprüngliche Reihenfolge wieder hergestellt werden.

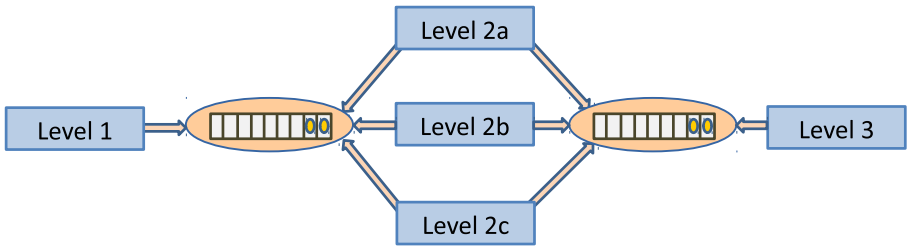


**Abbildung 10-7:** Komplexes Netzwerk aus Erzeugern und Verbrauchern

### 10.4.2 Erzeuger-Verbraucher-Muster mit Empfangsbestätigung

Die `TransferQueue` erweitert die `BlockingQueue` im Wesentlichen um die Methode `transfer(E elem)` (vgl. Abb. 10-8). Sie kann nur von einem Erzeuger benutzt werden. Die `transfer`-Methode kehrt erst dann zurück, wenn das übergebene Element abgeholt wird. Neben der `transfer`-Methode existieren noch die nicht blockierenden Varianten `tryTransfer(E elem)` und `tryTransfer(E elem, long timeout, TimeUnit unit)`, die über eine boolesche Rückgabe signalisieren, ob das übergebene Objekt von einem (wartenden) Verbraucher entnommen wurde. Wartet kein Abholer darauf (Rückgabe `false`), wird das Element nicht in die `Queue` gelegt.

Mit `hasWaitingConsumer` bzw. `getWaitingConsumerCount` kann ein Erzeuger abfragen, ob aktuell Interessenten an der `TransferQueue`

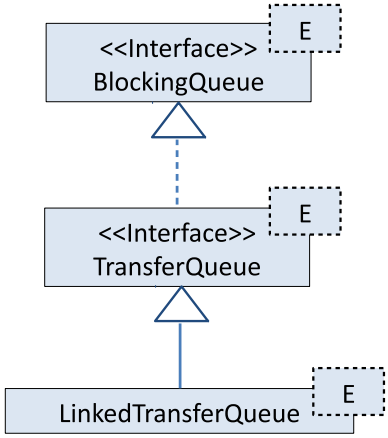


**Abbildung 10-8:** Klassenhierarchie für `TransferQueue`

warten. Entsprechend verwendet ein Verbraucher, je nach Anwendung, die Methode `take` bzw. `poll`.

### 10.4.3 Erzeuger-Verbraucher-Muster mit Work-Stealing

Bei einer `Deque` (*double ended queue*) können Elemente sowohl am Anfang als auch am Ende eingefügt bzw. entnommen werden. So gibt es z. B. neben `put` und `take` jeweils die Methoden `putFirst` und `putLast` bzw. `takeFirst` und `takeLast`, wobei `put` äquivalent zu `putLast` und `take` zu `takeFirst` ist.

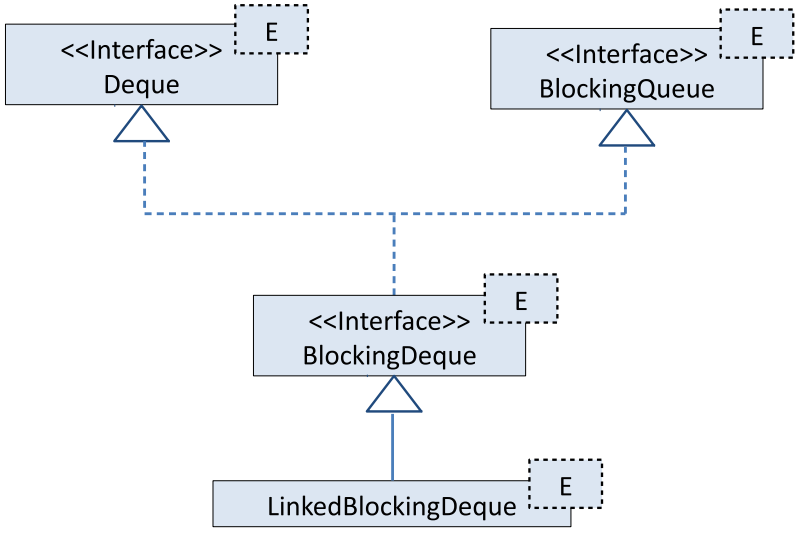


**Abbildung 10-9:** Klassenhierarchie für `BlockingDeque`

Ein wichtiges Anwendungsgebiet für `Deque`-Warteschlangen ist die Implementierung von *Work-Stealing*-Verfahren. Als Beispiel betrachten wir die

Ermittlung der Anzahl von Dateien in einem Verzeichnis inklusive der Unterverzeichnisse. Die Arbeit soll parallel von vier Tasks übernommen werden (vgl. Abb. 10-10). Statt einer rekursiven Variante verwenden wir eine stackbasierte, wobei jeder Task eine eigene `Deque` als Stack benutzt.

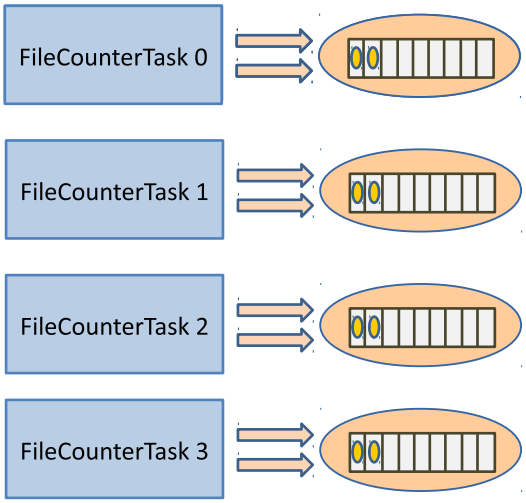


**Abbildung 10-10:** Zählen von Dateien mit vier parallelen Tasks

Bei dem Verfahren wird der Inhalt eines Verzeichnisses ausgelesen, die Anzahl der Datei gezählt und gefundene Unterverzeichnisse auf den Stack abgelegt (Codebeispiel 10.7). Ein Task verwaltet seine zu untersuchenden `File`-Objekte in einer `BlockingDeque`. Solange Elemente vorhanden sind, liest er das entsprechende Verzeichnis aus. Hat er ein Unterverzeichnis gefunden, legt er ein entsprechendes `File`-Objekt auf seinen Stack (❶). Ist das Verzeichnis abgearbeitet, prüft er, ob noch weitere zu untersuchen sind (❷).

```
file = this.workDeque[this.nr].pollFirst();
...
// Prüfe, ob in der dem Task zugeordneten Queue
// ein Element vorhanden ist
while (file != null)
{
  File[] files = file.listFiles();
  for (File f : files)
  {
    if (f.isDirectory())
      this.workDeque[this.nr].offerFirst(f);                    ❶
    else
      count++;
  }
  file = this.workDeque[this.nr].pollFirst();                   ❷
}
```

**Codebeispiel 10.7:** Verwaltung der Verzeichnisse auf einem Stack

Die Idee des *Work-Stealing* ist, dass ein Task, wenn er keine Elemente mehr auf seinem Stack hat, bei anderen nachschaut, ob diese noch Aufgaben zu bearbeiten haben. Falls ja, holt er sich eines, meist vom Ende der Queue, und bearbeitet es.

Codebeispiel 10.8 zeigt dieses Verfahren. Wenn sein eigener Stack leer ist (`file == null`), durchläuft der Task der Reihe nach die Stacks der anderen (❶) und prüft, ob dort noch `File`-Objekte auf die Bearbeitung warten (❷). Falls ja, holt er sich eines und bearbeitet es (❸). Wenn er Unterverzeichnisse findet, legt er die korrespondierenden `File`-Objekte dann wieder auf seinen eigenen Stack.

```
// Suche "Victim-Queue", Strategie Round-Robin
for (int i = 1; i < len; i++)                                    ❶
{
  int victimQueue = (this.nr + i) % len;
  if (this.workDeque[victimQueue].isEmpty() == false)            ❷
  {
    // Hole Item aus der Victim-Queue
    file = this.workDeque[victimQueue].pollLast();               ❸
    if(file != null)
      break;
  }
}
```

**Codebeispiel 10.8:** Work-Stealing

Der Code ist noch unvollständig bzw. noch nicht korrekt. Ein Problem ist das sichere koordinierte Beenden des Zählvorgangs. Die Tasks dürfen erst dann beendet werden, wenn keiner mehr aktiv ist. Die Bedingung, dass alle Stacks leer sind, reicht nicht, da ja ein Task gerade noch einen Auftrag bearbeiten könnte und dieser dann wieder viele neue Aufgaben produziert.

Zum koordinierten Beenden der Tasks kann ein *Terminierungsmonitor* benutzt werden, der im Wesentlichen einer Zählvariablen entspricht. Codebeispiel 10.9 zeigt eine Implementierung mit einem `AtomicInteger`-Objekt.

```
class TerminationMonitor
{
  private final AtomicInteger count;

  TerminationMonitor()
  {
    this.count = new AtomicInteger(0);
  }

  void setActive(boolean active)
  {
    if (active)
      count.getAndIncrement();
```

```
      else
        count.getAndDecrement();
    }

    boolean isTerminated()
    {
      return count.get() == 0;
    }
  }
```

**Codebeispiel 10.9:** Zähler zum koordinierten Beenden der Tasks

Codebeispiel 10.10 zeigt die komplette Implementierung. Zu Beginn der `call`-Methode bzw. nach der Abarbeitung seines eigenen Stacks signalisiert jeder Task seinen Zustand (❶,❷). Bei Bedarf sucht er bei den anderen nach Aufgaben. Hat er eine gefunden und sie an sich genommen, ist er wieder aktiv (❸) und verarbeitet sie. Erst wenn kein Task mehr aktiv ist, beenden sich alle (❹).

```
public class FileCountTask implements Callable<Integer>
{
  private static final FileFilter fileFilter = new FileFilter()
  {
    public boolean accept(File f)
    {
      return f.isDirectory() || f.isFile();
    }
  };

  private final int nr;
  private final BlockingDeque<File>[] workDeque;
  private final TerminationMonitor barrier;

  private FileCountTask(int nr,
                        BlockingDeque<File>[] workQueues,
                        TerminationMonitor barrier)
  {
    this.nr = nr;
    this.workDeque = workQueues;
    this.barrier = barrier;
  }

  @Override
  public Integer call() throws Exception
  {
    int len = this.workDeque.length;
    int count = 0;
    File file = null;

    this.barrier.setActive(true);                                    ❶
```

```
    // Hole Elemente aus der dem Thread zugeordneten Queue
    file = this.workDeque[this.nr].pollFirst();
    while (true)
     {
      // Prüfe, ob in der dem Task zugeordneten Queue
      // Elemente vorhanden sind
      while (file != null)
      {
        File[] files = file.listFiles(fileFilter);
        for (File f : files)
        {
         if (f.isDirectory())
          {
            this.workDeque[this.nr].offerFirst(f);
          }
          else
          {
            count++;
          }
        }
        file = this.workDeque[this.nr].pollFirst();
      }
      // Queue ist jetzt leer
      this.barrier.setActive(false);                                    ❷
      // Work-Stealing-Procedure
      while (file == null)
      {
        // Wenn es nur einen Task gibt, ist Work-Stealing sinnlos
        if (len == 1) break;

         // Suche "Victim-Queue", Strategie Round-Robin
        for (int i = 1; i < len; i++)
        {
          int victimQueue = (this.nr + i) % len;
          this.barrier.setActive(true);                                 ❸
          // Hole Element aus der Victim-Queue
          file = this.workDeque[victimQueue].pollLast();
          if (file != null)
          {
            break;              // Element war vorhanden
          }
          this.barrier.setActive(false);
        }
        // Alle Elemente sind abgearbeitet
        if (this.barrier.isTerminated())                                ❹
        {
          return count;
        }
      }
    }
  }
}
```

**Codebeispiel 10.10:** Stackbasierter Task zur Ermittlung der Dateianzahl

Man beachte, dass bei dieser Implementierung ein nach Arbeit suchender Task ständig die anderen Queues abfragt. Eine Alternative wäre die Verwendung von `pollLast` mit Timeout.

Codebeispiel 10.11 zeigt die Verwendung von `FileCountTask`-Objekten.

```
// Anzahl der parallelen Tasks
final int WORKER = 4;

// Startverzeichnis
final File root = new File(".....");


TerminationMonitor barrier = new TerminationMonitor();

// Erzeugen der Queues für die Worker
@SuppressWarnings("unchecked")
BlockingDeque<File>[] queues = new LinkedBlockingDeque[WORKER];
for (int i = 0; i < WORKER; i++)
{
  queues[i] = new LinkedBlockingDeque<>();
}
// Gebe Startverzeichnis dem ersten Worker
queues[0].offerFirst(root);

// Starten der Worker
List<FileCountTask> worker = new ArrayList<>();
for (int i = 0; i < WORKER; i++)
{
  worker.add(new FileCountTask(i, queues, barrier));
}
ExecutorService threadpool = Executors.newFixedThreadPool(WORKER);
List<Future<Integer>> futures = threadpool.invokeAll(worker);

// Sammeln der Ergebnisse
int count = 0;
for (Future<Integer> f : futures)
{
  count += f.get();
}
System.out.println("Anzahl der Dateien : " + count);

threadpool.shutdown();
```

**Codebeispiel 10.11:** Starten des Zählvorgangs

Neben der Implementierung von *Work-Stealing*-Verfahren kann eine `Deque` auch z. B. für die Implementierung von *Undo-Redo*-Mechanismen, Browser-Histories oder Palindromprüfungen eingesetzt werden.

### Hinweis

Die hier vorgestellten Varianten sind denen von *Messaging-Systemen* sehr ähnlich, deren Einsatzgebiete sowohl komplexe Systemlandschaften als auch die Integration verschiedener Anwendungen umfassen (vgl. [24]).

## 10.5 Zusammenfassung

Mit einem `Exchanger`-Objekt können zwei Threads synchron Daten austauschen. Für allgemeine Erzeuger-Verbraucher-Anwendungen mit asynchroner Kommunikation stehen verschiedene Queue-Klassen zur Verfügung. Das Blockieren und die beschränkte Kapazität garantieren, dass die Queue-Länge nicht unendlich wächst. Dadurch wird auch der Erzeuger ggf. gebremst. Auf eine Queue dürfen mehrere Threads sowohl lesend als auch schreibend zugreifen. Dadurch ist das Erzeuger-Verbraucher-Muster sehr flexibel einsetz- und skalierbar.

# 11 CountDownLatch und CyclicBarrier

Im Alltag kommt es oft vor, dass Teilnehmer aufeinander warten müssen, bevor sie mit der nächsten Aktion weitermachen können. Eine Reisegruppe muss z.B. auf einen Museumsführer warten, bevor sie Einlass in die Ausstellung erhält. Im Allgemeinen handelt es sich hier um einen sogenannten *Rendezvous-Punkt*, an dem sich die Teilnehmer treffen, bevor weitere Aktivitäten durchgeführt werden.

Analog kommt es auch bei der nebenläufigen Programmierung vor, dass Threads auf ein bestimmtes Ereignis warten müssen (bis z. B. bestimmte Initialisierungen beendet sind), bevor sie mit ihrer Arbeit weitermachen können. Java bietet für solche Synchronisationsaufgaben drei verschiedene Hilfsklassen an: `CountDownLatch`, `CyclicBarrier` und `Phaser`. In diesem Kapitel besprechen wir `CountDownLatch` und `CyclicBarrier` und im nächsten den `Phaser`.

## 11.1 CountDownLatch

Ein `CountDownLatch`-Objekt realisiert eine einfache Schranke, an der beliebig viele Threads warten können. Bei der Erzeugung wird ihm ein Startwert für einen internen Zähler mitgegeben, der mit der `countDown`-Methode erniedrigt werden kann. Sobald der Wert null erreicht wird, öffnet sich die Schranke und alle daran wartenden Threads können weiter laufen.

Möchte ein Thread an der Schranke auf das Startsignal warten, so ruft er an dem `CountDownLatch` dessen `await`-Methode auf. Ein `CountDownLatch` entspricht somit im übertragenen Sinn einer Startlinie, wie man sie von verschiedenen Rennen kennt. Das Setzen des Zählers auf null entspricht dem Startschuss für die Wartenden. Danach können auch verspätet ankommende Threads die Linie ohne Verzögerung passieren (vgl. Abb. 11-1).

In Tabelle 11-1 sind die Methoden eines `CountDownLatch`-Objekts aufgelistet. Man beachte, dass die beiden `await`-Methoden unterbrechbar sind, d.h. eine `InterruptedException` werfen können. Ein `CountDownLatch` führt kein Buch über die Anzahl der an ihm wartenden Threads. Man bezeichnet `await` als *wait-only*- und `countDown` als *signal-only*-Methode.

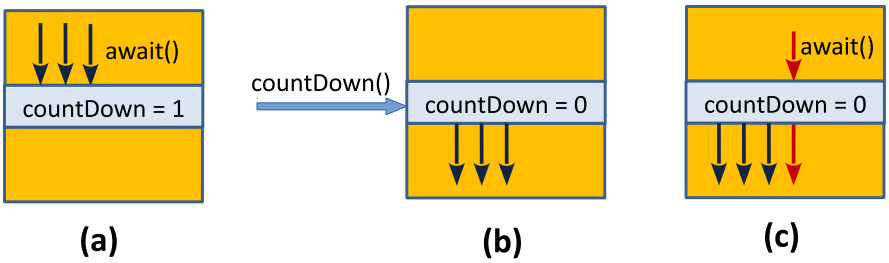


**Abbildung 11-1:** Wirkung von `CountDownLatch`

| Methode | Beschreibung |
|---|---|
| `void await()` | Hält den aufrufenden Thread an, solange der interne Zähler größer null ist und der Thread nicht unterbrochen wurde (`interrupted`). |
| `boolean await( long timeout, TimeUnit unit)` | Wie `await()` aber mit einem Timeout. Ist die übergebene Zeitspanne verstrichen, wird der Programmfluss fortgesetzt und die Methode liefert `false` zurück, ansonsten wird `true` zurückgegeben. |
| `void countDown()` | Zählt den internen Zähler um eins herunter. |
| `long getCount()` | Liefert den Stand des internen Zählers. |

**Tabelle 11-1:** Einige Methoden der `CountDownLatch`-Klasse

## Hinweis

Ein `CountDownLatch` kann nur einmal verwendet werden. Der interne Zählerstand kann nicht wieder hochgezählt bzw. zurückgesetzt werden.

Der Aufruf von `countDown` blockiert den Aufrufer nicht. Ein `CountDownLatch` ist in diesem Sinne »vorrückbar« (*advanceable*).

Eine typische Anwendung für diesen Mechanismus ist das gleichzeitige Loslaufen mehrerer Threads. Im Codebeispiel 11.1 wird ein Laufwettkampf simuliert. Ein `CountDownLatch` übernimmt hier die Rolle der Startlinie. Die jeweiligen Läufer werden durch Objekte der Klasse `Athlet` repräsentiert (❶). Sie erhalten über den Konstruktor den Zugriff auf ein `CountDownLatch`-Objekt, dessen `await` zu Beginn der `run`-Methode von jedem Thread aufgerufen wird (❷).

```
public class CountDownDemo1
{
  static class Athlet implements Runnable                          ❶
  {
    private String name;
    private CountDownLatch latch;

    public Athlet(String name, CountDownLatch latch)
    {
      this.name  = name;
      this.latch = latch;
    }

    @Override
    public void run()
    {
      System.out.println(name + " ist bereit ....");
      try
      {
        // Warte auf das Startsignal
        latch.await();                                             ❷
        TimeUnit.MILLISECONDS.sleep(
                ThreadLocalRandom.current().nextInt(1000));
        System.out.println(name + " ist am Ziel ");
      }
      catch (InterruptedException ex)
      {
        System.out.println("Wettkampf abgebrochen!");
      }
    }
  }

  public static void main(String[] args) throws InterruptedException
  {
    ExecutorService executor = Executors.newFixedThreadPool(3);

    CountDownLatch startlinie = new CountDownLatch(1);             ❸
    Athlet a1 = new Athlet("Carl Lewis", startlinie);
    Athlet a2 = new Athlet("Maurice Greene", startlinie);
    Athlet a3 = new Athlet("Usain Bolt", startlinie);

    executor.execute(a1);
    executor.execute(a2);
    executor.execute(a3);

    TimeUnit.MILLISECONDS.sleep(500);
    System.out.println("Los!");
    startlinie.countDown();                                        ❹

    executor.shutdown();
  }
}
```

**Codebeispiel 11.1:** CountDownLatch mit dem Startsignal vom `main`-Thread

Der Startschuss kommt im Codebeispiel 11.1 nach der festgelegten Wartezeit vom `main`-Thread (❹). Kommt einer der Teilnehmer verspätet an, wird darauf keine Rücksicht genommen. Möchte man sicherstellen, dass alle Läufer (Threads) erst bei Vollzähligkeit loslaufen, kann man den `CountDownLatch` (❸) mit der Anzahl der zu synchronisierenden Threads initialisieren:

```
CountDownLatch startlinie = new CountDownLatch(3);
```

Jeder Thread erniedrigt dann den Zähler, bevor er an die Startlinie kommt.

```
latch.countDown();
latch.await();
```

Der zuletzt ankommende setzt den Zähler dadurch auf null und alle laufen los.

Wird ein an einem `CountDownLatch` wartender Thread durch `interrupt` unterbrochen, so hat das keinerlei Auswirkungen auf die Schranke bzw. auf die anderen daran wartenden Threads. Der `CountDownLatch` kann ohne Einschränkung weiter benutzt werden, da er keinerlei Information über die an ihm wartenden Threads hat. Die im nächsten Abschnitt besprochene `CyclicBarrier` reagiert in einem solchen Fall völlig anders.

## 11.2 CyclicBarrier

Eine `CyclicBarrier` ist ein weiteres Synchronisationsmittel, das dem `CountDownLatch` in mancher Hinsicht ähnlich ist. Eine `CyclicBarrier` entspricht ebenfalls einer Synchronisationsschranke (Barriere), wobei hier eine im Vorfeld festgelegte Anzahl von Threads ankommen muss, damit sich die Schranke öffnet. Bevor alle loslaufen, kann optional noch eine `Runnable`-Aktion ausgeführt werden. Die Barriere ist im Unterschied von `CountDownLatch` zyklisch, weil sie mehrmals verwendet werden kann.

Die Klasse `CyclicBarrier` besitzt zwei Konstruktoren, wobei jeweils die Anzahl der zu koordinierenden Threads festgelegt wird:

- `CyclicBarrier(int parties)`: Erzeugt eine `CyclicBarrier`, die `parties` Threads synchronisieren kann.
- `CyclicBarrier(int parties, Runnable barrierAction)`: Erzeugt eine `CyclicBarrier`, die `parties` Threads synchronisieren kann. Beim »Schalten« der Barriere wird vor dem Loslaufen der Threads das Runnable `barrierAction` ausgeführt.

Mit dem `await`-Aufruf kann sich ein Thread an der Barriere registrieren. Er kann dann erst weiterlaufen, wenn die vorher festgelegte Anzahl von Teilnehmern (Anzahl der Aufrufe von `await`) erreicht wurde. Die `await`-Methode ist eine sogenannte *signal-wait*-Methode. Sie signalisiert die Ankunft an einer Barriere und wartet anschließend auf die Freischaltung. Aus diesem Grund ist eine `CyclicBarrier` nicht vorrückbar (*not advanceable*).

Abbildung 11-2 und Abbildung 11-3 zeigen das zyklische Vorgehen schematisch.

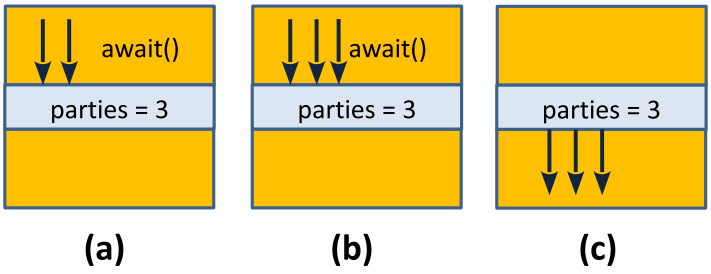


**Abbildung 11-2:** Koordination von drei Threads mit einer `CyclicBarrier`

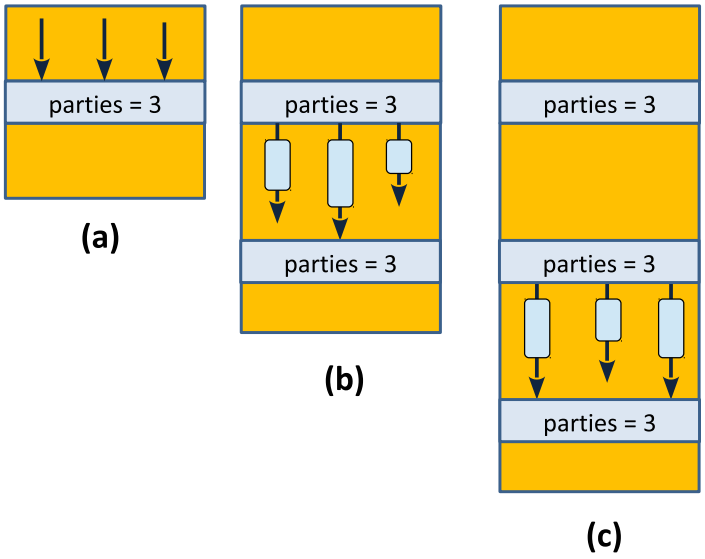


**Abbildung 11-3:** Zeitlicher Ablauf an einer `CyclicBarrier`

Tabelle 11-2 gibt einen Überblick über die Methoden der Klasse. Der Umgang mit Fehlern ist bei der `CyclicBarriere` komplizierter als bei dem `CountDownLatch`, da sie an eine feste Anzahl von Threads gebunden ist und ein `await`-Aufruf ihren internen Wartezähler ändert. Die Barriere kann auf vielfältige Art gebrochen werden:

- Wenn auf einem wartenden Thread `interrupted` aufgerufen wird, führt das zum Bruch der Barriere. Alle anderen wartenden erhal-

ten dadurch eine `BrokenBarrierException` und verlassen den *wait*-Zustand.
- Wird `reset` auf der Barriere aufgerufen, erhalten ebenfalls alle wartenden Threads eine `BrokenBarrierException`.
- Wenn einer der wartenden Threads eine `TimeoutException` erhält, weil er `await(long timeout, TimeUnit unit)` aufgerufen hat, führt das zu einem Bruch der Barriere.

Es empfiehlt sich, gebrochene Barrieren nicht weiter zu verwenden, sondern mit einer neu erzeugten und initialisierten den Ablauf fortzusetzen.

| Methode | Beschreibung |
|---|---|
| `int await()` | Der Thread wartet, bis alle `parties` an der Barriere angekommen sind. Als Rückgabe erhält man die Anzahl der noch zu erwartenden Threads. |
| `int await(long timeout, TimeUnit unit)` | Wie `await()` aber mit einem Timeout. Ist die Zeitspanne verstrichen, wird eine `TimeoutException` geworfen und die Barriere wird gebrochen. Im Normalfall erhält man als Rückgabe die Anzahl der noch zu erwartenden Threads. |
| `int getParties()` | Liefert die Anzahl der notwendigen Threads. |
| `int getNumberWaiting()` | Liefert die Anzahl der aktuell wartenden Threads. |
| `boolean isBroken()` | Gibt an, ob die Barriere gebrochen wurde. |
| `void reset()` | Setzt die Barriere auf den Initialzustand zurück. Eventuell wartende Threads werden durch eine `BrokenBarrierException` unterbrochen. |

**Tabelle 11-2:** Einige Methoden einer `CyclicBarrier`

Eine `CyclicBarrier` kann dann bevorzugt eingesetzt werden, wenn sich eine Aufgabe in mehrere nebenläufig ausführbare Schritte unterteilen lässt. Sobald alle Teilprobleme gelöst sind, können z. B. deren Ergebnisse in einer angegebenen `Runnable`-Aktion zu einer Gesamtlösung zusammengeführt werden.

Das Codebeispiel 11.2 zeigt die Verwendung einer `CyclicBarrier`. Hier liefern sich die drei Athleten einen Wettkampf, der aus drei aufein-

anderfolgenden Rennen besteht. Der Eintritt in jede Runde erfolgt synchronisiert (❶) an einer `CyclicBarrier`. Sie wird für 3 Threads (`taskCount`) konfiguriert (❷). Zusätzlich wird ihr noch ein `Runnable`-Objekt zugewiesen (❸), das ausgeführt wird, wenn alle drei ankommen. Danach wird die Barriere für den nächsten Lauf freigeschaltet (vgl. Abb. 11-4).

```
public class CyclicBarrierDemo
{
  static class Athlet implements Runnable
  {
    private final String name;
    private final CyclicBarrier barrier;

    public Athlet(String name, CyclicBarrier barrier)
    {
      this.name    = name;
      this.barrier = barrier;
    }

    @Override
    public void run()
    {
      System.out.println(name + " ist bereit ....");
      try
      {
        int time = 0;
        for(int i=0; i < ROUND; i++)
        {
          // Warte auf Mitläufer für eine neue Runde
          barrier.await();                                          ❶
          int lauf = ThreadLocalRandom.current().nextInt(1000);
          TimeUnit.MILLISECONDS.sleep( lauf );
          time += lauf;
        }

        // Warte auf das Ende des Wettkampfs
        barrier.await();
        System.out.println(name + " ist am Ziel : "
                                    + time + " Gesamtzeit");
      }
      catch (InterruptedException | BrokenBarrierException ex)
      {
        System.out.println("Wettkampf abgebrochen!");
      }
    }
  }

  private static final int ROUND = 3;

  public static void main(String[] args)
  {
    final int taskCount = 3;
```

```
    ExecutorService executor = Executors.newFixedThreadPool(taskCount);

    CyclicBarrier barrier = new CyclicBarrier(taskCount,          ❷
          new Runnable()                                          ❸
          {
             private int count = 1;

             @Override
             public void run()
             {
               if( count <= ROUND )
               {
                 System.out.println(
                      "==> Starte in die Runde " + count++);
               }
             }
           });

    Athlet a1 = new Athlet("Carl Lewis", barrier);
    Athlet a2 = new Athlet("Maurice Greene", barrier);
    Athlet a3 = new Athlet("Usain Bolt", barrier);
    executor.execute(a1);
    executor.execute(a2);
    executor.execute(a3);

    executor.shutdown();
  }
}
```

**Codebeispiel 11.2:** Anwendung einer `CyclicBarrier`

## Hinweis

Eine gebrochene Barriere kann theoretisch weiterverwendet werden, indem die `reset`-Methode aufgerufen wird. Häufig ist es aber sehr umständlich, alle Threads wieder korrekt zu synchronisieren, insbesondere, wenn zum Zeitpunkt des `reset`-Aufrufs noch nicht alle angekommen sind. Wird eine Barriere gebrochen, sollte man sie folglich nicht weiterverwenden, sondern besser eine neu erzeugte einsetzen.

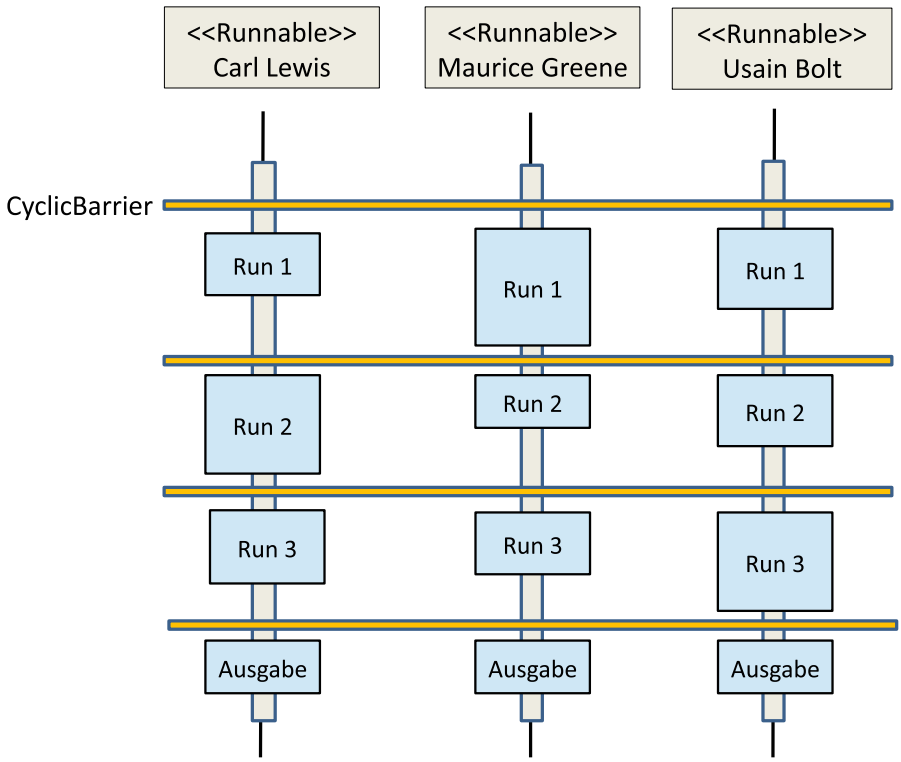


**Abbildung 11-4:** Ein Wettkampf mit drei Runden, koordiniert durch eine `CyclicBarrier`

## 11.3 Zusammenfassung

Mit den Klassen `CountDownLatch` und `CyclicBarrier` können mehrere Threads synchronisiert koordiniert werden.

Um ein einfaches Start- oder Stoppsignal für eine unbestimmte Anzahl von Threads zu realisieren, kann die Klasse `CountDownLatch` verwendet werden. Durch das Herunterzählen eines internen Zählers kann man die wartenden Threads koordiniert weiterlaufen lassen. Das Herunterzählen kann von »außen«, z. B. vom `main`-Thread, erfolgen oder durch die Threads selbst.

Wartet man auf eine bestimmte Anzahl von Threads und muss insbesondere die auszuführende Arbeit periodisch wiederholt werden, so ist der Einsatz von `CyclicBarrier` passender. Eine `CyclicBarrier` schaltet dann, wenn eine vorher festgelegte Anzahl von Beteiligten an ihr warten. Ihr kann zusätzlich ein `Runnable`-Objekt zugeordnet werden, das vor jeder Öffnung ausgeführt wird.