### Was ist ein Prozess?

Wird ein Programm gestartet, startet das Betriebssystem dieses Programm als Prozess im Rechner. Zum Prozess gehören ein Teil des Arbeitsspeichers und CPU-Ressourcen. Anweisungen in einem Prozess werden grundsätzlich nacheinander abgearbeitet.

Dies ist ärgerlich, wenn eine Anweisung auf externe Ereignisse wartet (wie in der Demo-Anwendung auf die Verbindung eines Clients und seine Nachrichten). Dann sind z. B. Eingabeaktionen an der GUI blockiert.

Eine Lösung dazu sind Threads, die auch als Nebenläufigkeit bezeichnet werden. Sie erlauben eine Entkopplung solcher blockierenden Anweisungen.

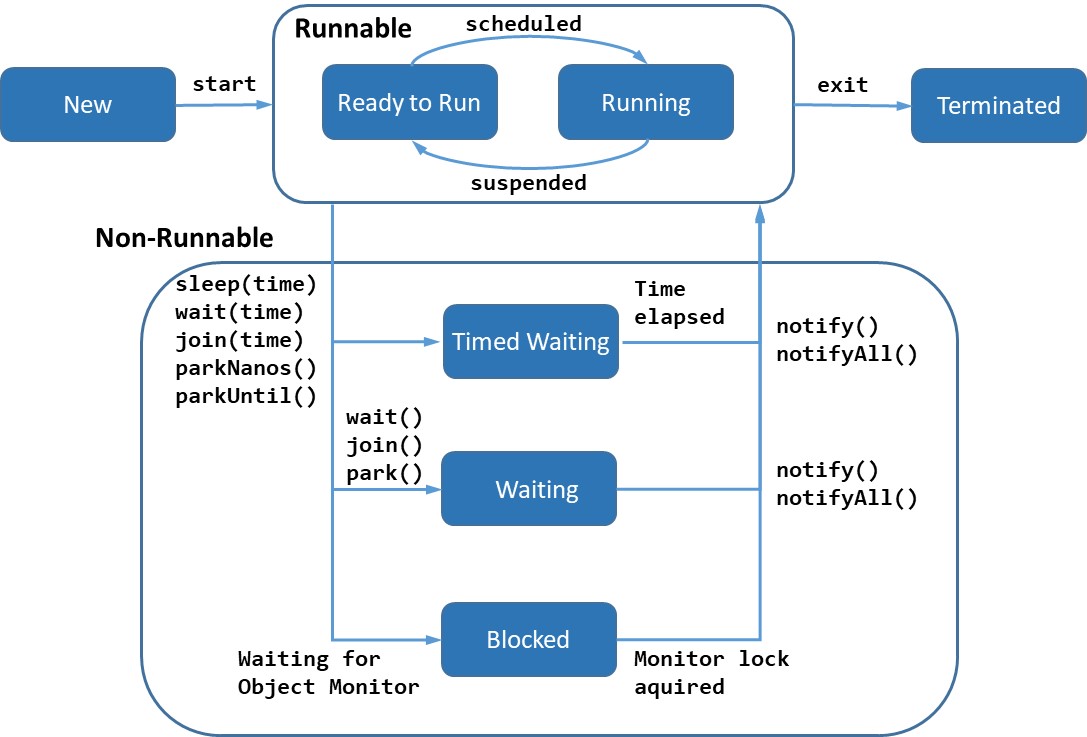
### Prozesse und Threads

Prozesse und Threads unterscheiden sich vor allem im Umfang der ihnen zur Verfügung stehenden Ressourcen des Rechners.

Ein Prozess bekommt alles was zu seiner Ausführung benötigt, u. a. einen abgeschirmten Speicherbereich, bekommt Zugriff auf Rechnerkomponenten wie Laufwerke, Netzwerkkarte, Prozessorkerne usw. Verwaltet werden die verschiedenen laufenden Prozesse auf dem Rechner durch den Prozess-Scheduler des Betriebssystems.

Threads sind nun kleine eigenständige Ausführungseinheiten, nur eben innerhalb seines Prozesses, zu denen sie gehören.

Mit Hilfe von Threads können wir Nebenläufigkeit (Scheinparallelität und echte Parallelität) innerhalb eines Prozesses erreichen. Wir können uns Threads als eine Unterteilung eines Prozesses in Einheiten vorstellen, die dann nebeneinander bzw. nebenläufig (concurrent) auf den zu diesem Prozess gehörenden Ressourcen ausgeführt werden (Multithreading).

Mit Multithreading können wir mehrere Rechnerkerne besser ausnutzen und vor allem Oberflächen bedienbar halten.

Ein Hauptproblem ist, dass schwer auffindbare Fehler auftreten können. Übertreiben wir es mit dem Multithreading, kann das Ganze nach hinten losgehen und der Aufwand der Verwaltung sind so hoch, dass der potenzielle Leistungsgewinn zunichtegemacht wird.

Ein Thread kennt mehrere Phasen (siehe oben), deren Übergänge durch die angegebenen Methoden erreicht werden können. Zunächst betrachten wir aber:

* Nach dem Start ist der Thread Runnable, er wird meist in die Running Phase treten, in der er seine Aufgaben abarbeitet.
* Durch Konkurrenz und Kommunikation mit anderen Threads kann es vorkommen, dass er warten muss oder einem anderen Thread Vortritt gibt: Er wechselt in die Phase waiting.
* Wir erinnern uns an unseren Receiver und erkennen, dass die Verbindung mit dem Client in einem separaten Thread ausgelagert werden muss.
* Ist ein Thread beendet, terminiert er. Auf die gezielte vorzeitige Beendigung gehen wir später noch ein.

Wie geht nun die Umwandlung eines Sequenziellen in ein nebenläufiges Programm? Dazu sehen wir das folgende Beispiel-Programm an, das die rechts stehende Ausgabe erzeugt:

Runner1 0

Runner1 1

Runner1 2

Runner1 3

Runner1 4

Runner1 5

Runner1 6

Runner1 7

Runner2 0

Runner2 1

Runner2 2

Runner2 3

Runner2 4

Runner2 5

Runner2 6

Runner2 7

**class Runner1 {**

**public void startRunner1() {**

**for (int i=0;i<=8;i++) {**

**System.out.println("Runner1 "+i);**

**}**

**}**

**}**

**class Runner2 {**

**public void startRunner2() {**

**for (int i=0;i<=8;i++) {**

**System.out.println("Runner2 "+i);**

**}**

**}**

**}**

**public class RunnerExample {**

**public static void main(String[] args) {**

**Runner1 runner1 = new Runner1();**

**Runner2 runner2 = new Runner2();**

**runner1.startRunning();**

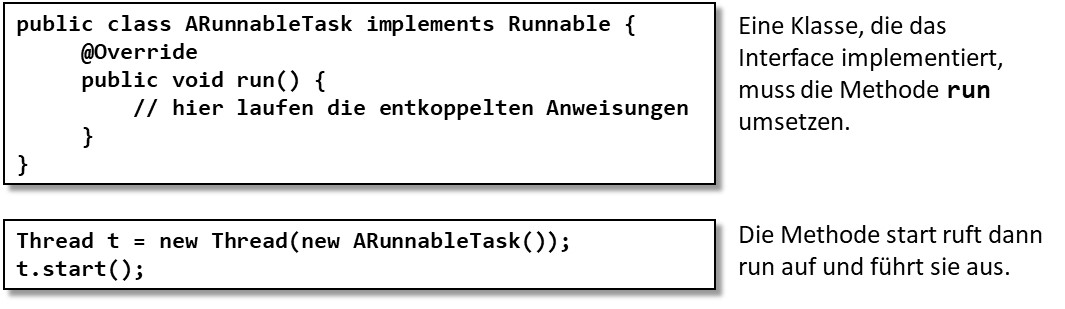
**runner2.startRunning();**

**}**

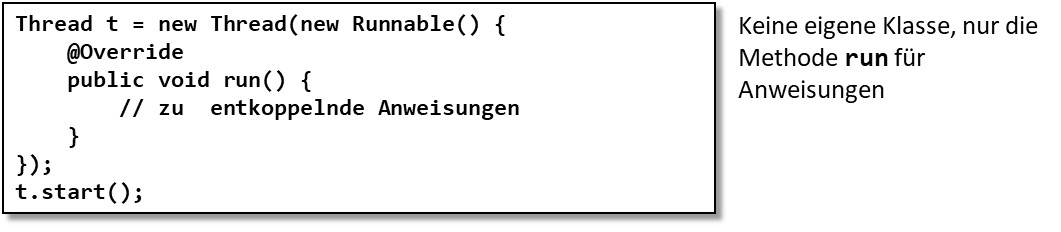
**}**

Wir erkennen klar, die zuerst gestartete for-Schleife wird abgearbeitet, dann die zweite Schleife. Um die beiden Zählschleifen nun nebeneinander laufen zu lassen, bietet Java zwei Mittel an: die Klasse Thread zum Ableiten oder das Interface Runnable.

Wie funktioniert das genau? Wir brauchen eine Klasse, die das Interface implementiert. In der dann zu realisierenden Methode können die Anweisungen, die sonst zu einem Blockieren führen, eingebettet werden. Die Klasse, die das Interface implementiert, muss dann einem Thread als Parameter übergeben werden und mit start() wird dann die run()-Methode aufgerufen (das kann nicht direkt erfolgen, schließlich muss der oben erwähnte Kontext erst gebildet werden). Hinweis: Die Klassendarstellung von Thread ist nicht vollständig, sondern beschränkt sich auf den gezeigten Code.



Wollen wir keine eigene Klasse erstellen, weil wir nur einen Methodenaufruf einbetten wollen, so können wir auch eine anonyme Klasse einbetten.



Nehmen wir unsere beiden Klassen Runner1 und Runner2 und lassen sie das Interface implementieren, schreiben wir die Schleife der Übersichtlichkeit halber in die run()-Methode, die wir ja umsetzen müssen. Zudem starten wir dann natürlich auch mit der start()-methode des Threads (wir rufen also die Methode mit der Schleife nicht mehr direkt auf).

**class Runner1 implements Runnable {**

**@Override**

**public void run() {**

**for (int i=0;i<=8;i++) {**

**System.out.println("Runner1 "+i);**

**}**

**}**

**}**

**class Runner2 {**

**@Override**

**public void run() {**

**for (int i=0;i<=10;i++) {**

**System.out.println("Runner2 "+i);**

**}**

**}**

**}**

**public class RunnerExample {**

**public static void main(String[] args) {**

**Runner1 runner1 = new Runner1();**

**Runner2 runner2 = new Runner2();**

**Thread runnerThread1 = new Thread(runner1);**

**Thread runnerThread2 = new Thread(runner2);**

**runnerThread1.start();**

**runnerThread2.start(); }**

**}**

Runner1 0

Runner2 0

Runner1 1

Runner2 1

Runner1 2

Runner2 2

Runner2 3

Runner1 3

Runner2 4

Runner1 4

Runner1 5

Runner2 5

Runner1 6

Runner2 6

Runner1 7

Runner2 7

Wir sehen, dass die beiden Methoden nun abwechselnd ihre Ausgaben erzeugen. Beide Threads konkurrieren um die Ausgabe (nur jeweils einer kann ausgeben), aber sie laufen quasi gleichzeitig und müssen nur immer auf die Freigabe der Ausgabe warten.

Eine Alternative zur Verwendung des Runnable Interfaces ist die Verwendung von Vererbung. Dazu leiten wir die Klassen Runner1 und Runner2 von der Klasse Thread ab.

Runner1 0

Runner2 0

Runner1 1

Runner2 1

Runner2 2

Runner1 2

Runner2 3

Runner2 4

Runner1 3

Runner2 5

Runner1 4

Runner2 6

Runner1 5

Runner2 7

Runner1 6

Runner1 7

**class Runner1 extends Thread {**

**// gleich wie oben**

**}**

**class Runner2 extends Thread {**

**// gleich wie oben**

**}**

**public class RunnerExample {**

**public static void main(String[] args) {**

**Runner1 runner1 = new Runner1();**

**Runner2 runner2 = new Runner2();**

**runner1.start();**

**runner2.start(); }**

**}**

Betrachtet man die Ausgabe, so sieht man ein anderes Problem der Nebenläufigkeit: wir können nur ungefähr vorhersagen, wann ein Thread drankommt.

Zusammenfassend: Die Implementierung des Runnable wird allgemein als das bessere Verfahren angesehen, da nicht die Thread-Klasse verändert werden muss (Trennung von spezifischem Verhalten und Systemabbildung).

Zusätzlich stehen ab Java 5 mit dem java.util.concurrent-Package weitere Klassen und Interfaces zur Verfügung. So können mit dem Callable-Interface Parameter und Rückgabewerte verwendet werden, benötigen aber zur Ausführung die neue Klasse ExecutorService. Hierzu wird jedoch auf die weiterführende Literatur verwiesen.

Was passiert, wenn wir nach dem Start der beiden Threads weitere Anweisungen (z.B. eine Ausgabe) in der main einfügen?

Fertig

Runner1 0

Runner1 1

Runner1 2

Runner1 3

Runner1 4

Runner1 5

...

**public class RunnerExample {**

**public static void main(String[] args) {**

**Runner1 runner1 = new Runner1();**

**Runner2 runner2 = new Runner2();**

**runner1.start();**

**runner2.start();**

**System.out.println("Fertig");**

**}**

**}**

Das hätten wir erwarten können. Die Threads sollen ja eben den Ablauf des Hauptprogramms nicht unterbrechen. Wollen wir nun warten, bis die gestarteten Threads fertig sind, können wir unseren Thread (die main) durch den Aufruf der join()-Methode zum Warten bewegen. Der aktuelle Thread (main) wartet, bis der aufgerufene Thread fertig ist. Alternativ kann man der join()-Methode noch eine Wartezeit in Millisekunden mitgeben, nach deren Ablauf der aufrufende Thread auch wieder dran kommt.

Runner1 0

Runner1 1

Runner1 2

...

Runner1 6

Runner1 7

Runner1 8

Fertig

**public class RunnerExample {**

**public static void main(String[] args) {**

**Runner1 runner1 = new Runner1();**

**runner1.start();**

**try {**

**runner1.join();**

**} catch (InterruptedException e) {}**

**System.out.println("Fertig");**

**}**

**}**

**join(): It will put the current thread on wait until the thread on which it is called is dead. If thread is interrupted then it will throw InterruptedException.**

Damit wird der aufrufende Thread angehalten, bis der oder anderen Threads geendet haben.