Was sind Threads?

Moderne Betriebssysteme wie Windows haben die besondere Fähigkeit, mehrere Programme parallel ablaufen zu lassen. Jedes Programm, das Sie starten, wird auf der Task-Leiste des Betriebssystems (in der Standardeinstellung am unteren Rand des Bildschirms) durch eine Schaltfläche repräsentiert und in einem separaten Fenster ausgeführt. Windows bezeichnet diese parallel vom Rechner verwalteten Programme als **Anwendungen**. Auch wenn immer nur ein Anwendungsfenster aktiv und damit im Vordergrund sein kann, sind doch alle Programme tätig. So kann Ihnen z.B. ein auf die Task-Leiste minimiertes E-Mail-Programm den Eingang neuer E-Mails durch ein akustisches Signal oder in Dialogfenster anzeigen, wenn Sie gerade einen Text in einem Textverarbeitungsprogramm erstellen. Die neueren Windows-Betriebssysteme verfügen über ein Systemprogramm, mit dem man sich die Liste gerade aktiver Anwendungen ebenfalls ansehen kann: den Windows **Task-Manager**. Sie erreichen dieses Programm durch einen Klick mit der rechten Maustaste auf eine freie Stelle der Task-Leiste und wählen im aufklappenden Kontextmenü den Eintrag „Task-Manager“. Auf der Registerkarte „Anwendungen“ sehen Sie genau die Programme, die sich auch auf der Task-Leiste eingefunden haben.

Liste der gestarteten Anwendungen im Windows Task-Manager

Mit dieser Auflistung sehen Sie aber nur die, Spitze des Eisbergs‘. Einen ersten Hinweis auf weitere Aktivitäten des Rechners meldet die Statusleiste des Task-Managers links unten mit der Angabe: „Prozesse: 45“ (Da wird bei Ihnen wahrscheinlich eine andere Zahl stehen). Schalten Sie auf die Registerkarte „Prozesse“ um, so sehen Sie, um welche Prozesse es sich hier handelt. In der Liste finden sich auch die Programme aus der Registerkarte „Anwendungen“, allerdings nun mit dem Namen ihrer exe-Datei (Word z. B. als „winword.exe“, der Browser mit „firefox.exe“ usw.) Darüber hinaus zeigt sich hier eine Vielzahl von Programmen, die das Betriebssystem am Laufen hält, ohne dass sie alle in einem Windows-Fenster sichtbar werden. Vornehmlich sind dies Dienste des Betriebssystems oder Anwendungsprogramme, die als Service (Dienst) eingerichtet wurden – z. B. ein laufender Apache Webserver. Ein Anwendungsprogramm kann sich aber auch aus mehreren Prozessen zusammensetzen.

Ein **Prozess** ist ein gestartetes Programm, das im Arbeitsspeicher des Rechners ausgeführt wird. Objektorientiert gesehen könnte man auch von der „**Instanz**“ eines Programms sprechen, das über den allein für diese Instanz reservierten Arbeitsspeicher verfügt.

All die im Task-Manager aufgelisteten Prozesse führt der Rechner parallel aus – so scheint es jedenfalls. Doch im Normalfall verfügt ein PC lediglich über einen Prozessor / eine **CPU** („Central Processing Unit“) und kann daher auch nur ein einziges Programm von der CPU ‚gleichzeitig‘ ausführen lassen. In einem ausgetüftelten System wechselt jedoch die CPU ständig in sehr kurzen Zeitabständen den unterstützten Prozess und teilt auf diese Weise rund um Rechenressourcen zu. Man nennt diesen Prozesswechsel auch **Kontextumschaltung**. So entsteht der Eindruck, dass die diversen Prozesse tatsächlich parallel laufen. Die Kontextumschaltung dient auch einer guten Auslastung der CPU-Kapazitäten.

Nehmen wir als Beispiel einen Prozess zur Verarbeitung von Daten, die zunächst von einem Laufwerk gelesen oder auf ein Laufwerk geschrieben werden müssen. Aktivitäten, die sich allein im Arbeitsspeicher abspielen, sind viel schneller als I/O-Aktivitäten. Der Prozess muss also häufig warten, bis die benötigten Daten geladen bzw. geschrieben sind. In dieser Zeit können andere Prozesse gute Arbeit leisten, was die Kontextumschaltung ermöglicht. Mit den Prozessen haben Sie aber noch immer nicht den vollständigen Umfang (scheinbar) parallel laufender Aktivitäten auf Ihrem Rechner sichtbar gemacht.

Hinter den meisten Prozessen verstecken sich nämlich ‚Teilprozesse‘, die bestimmte Teilaufgaben des Programms selbstständig ausführen. Man nennt diese Teilprozesse **Threads**. Der wesentliche Unterschied zwischen Threads und Prozessen liegt in der Zuordnung von Arbeitsspeicher. Während jeder Prozess über einen allein für ihn reservierten Arbeitsspeicher verfügt, laufen die Threads eines Prozesses nebeneinander innerhalb des Speicherbereichs dieses Prozesses. Man spricht daher auch von **Nebenläufigkeiten**. Jeder Prozess ist allerdings auch selbst zugleich ein Thread.

Beispiele für Threads

1. Häufig werden Sie sich für Threads entscheiden, wenn Sie grafische Oberflächen mit hohem Kommunikationsaufwand zwischen Benutzer und Programm programmieren. Da kann es passieren, dass die Anwendung eine zeitintensive Aktivität durchführt – z. B. das Laden größerer Dateien über ein Netzwerk und derweil die Benutzeroberfläche blockiert. Also werden Sie den Ladevorgang in einen Thread separieren, damit die Benutzeroberfläche sofort auf weitere Eingaben reagieren kann.

2. Ein anderes Beispiel sind Serveraktivitäten. Ein Druck-Server soll für mehrere Clients Druckaufträge abwickeln, ein Fileserver verschiedenen Clients Dateien liefern oder ein Chat-Server viele Chat-Teilnehmer mit Informationen versorgen. In all diesen Fällen wird man für jeden Client einen separaten Thread bereitstellen. Ein solches Beispiel werden Sie in der letzten Lektion dieses Lernhefts programmieren.

Doch zunächst müssen Sie die beiden Wege kennenlernen, wie Sie unter Java Threads programmieren können.

Projektarbeit: Ein einfacher Thread zur Zeitsteuerung

Ein Thread soll bestimmte Programmabläufe abarbeiten. In diesem Abschnitt werden wir Threads vorrangig zur Zeitsteuerung benutzen. Die Programmierung einer **Uhr** zeigt schnell die Aufgabe einer Zeitsteuerung: Eine Uhr soll im Sekundenabstand die Zeitanzeige aktualisieren. Dabei ist es prinzipiell egal, ob man eine solche Uhr grafisch als Analoguhr mit Ziffernblatt und Zeigern gestaltet oder als digitale Uhr, die mit Ziffern die aktuelle Zeit anzeigt. Das Programmierproblem besteht darin, die Zeit von der einen zur nächsten Sekunde zu überbrücken.

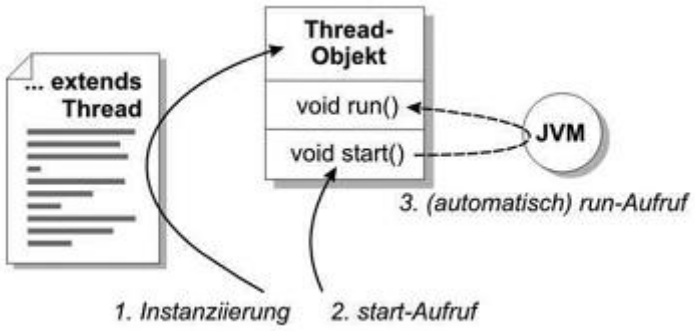
Diese Überbrückungsaufgabe ist deshalb so gut für Thread-Programmierung geeignet, weil das Uhrenprogramm nur jede Sekunde etwas tun soll, in der Zwischenzeit aber keine Rechnerressourcen beanspruchen muss. In dieser Zwischenzeit kann die CPU andere Prozesse auf dem Rechner mit ihren Ressourcen bedienen.
Wenn wir ein Thread-Objekt auf die übliche Weise erzeugen Thread uhr = new Thread(); dann haben wir zwar ein Thread-Objekt, doch es bleibt erst einmal unklar, wie wir diesem Thread-Objekt beibringen sollen, vernünftig im Sekundentakt eine Zeitanzeige zu erneuern. Denn in der Klasse „java.lang.Thread“ finden Sie nirgends eine Methode, die diese spezielle Aufgabe bereits löst. Wir müssen daher die Klasse „Thread“ in einer eigenen Klasse *erweitern*, um in dieser erbenden Klasse das Nötige festzulegen. In der erbenden Klasse wird dann die von „Thread“ ererbte Methode public void **run**() überschrieben.

Im Programmblock der überschriebenen Methode „public void run()“ legen Sie *immer* für einen Thread fest, was er tun soll.

Diese Verfahrensweise ähnelt dem Umgang mit typischen Applet-Methoden, die Sie ebenfalls in einer von „java.applet.Applet“ erbenden Klasse ausprogrammieren, um ein bestimmtes Verhalten in einem bestimmten Abschnitt der Lebenszeit eines Applets festzulegen. Ebenso ähnlich wie bei den typischen Applet-Methoden „init“, „start“, „stop“ und „destroy“ werden Sie diese run-Methode nie selbst aufrufen! Für den run-Aufruf sorgt ein interner Mechanismus der **JVM** (Java Virtual Machine), der immer dann loslegt, wenn Sie einen Thread starten, was über den Aufruf der ebenso einfach gestrickten Thread-Methode public void **start**() erfolgt.

Die Programmierung eines Threads umfasst damit vier Stufen:

1. Sie definieren eine von „java.lang.Thread“ erbende Klasse.
2. In dieser Klasse überschreiben Sie die run-Methode und regeln dort, was der Thread tun soll.
3. Sie bilden eine Instanz der von „Thread“ erbenden Klasse, also Ihr Thread-Objekt.
4. Sie rufen für dieses Thread-Objekt die start-Methode auf, um den Thread zum Laufen zu bringen. Dies führt automatisch dazu, dass die Virtuelle Maschine die run-Methode des Threads ausführt.



Wie man einen Thread „zum Laufen“ bringt.

Code 1: Die drei Aktionen zur Laufzeit sind in der Kommentierung hervorgehoben:

```
import java.util.Date;
import java.text.SimpleDateFormat;

public class Konsolenuhr extends Thread {

   private Date date;
   String timeFormat="HH:mm:ss";
   SimpleDateFormat sdfDatum;

   public static void main(String[] args) {
      // 1. Thread-Objekt erzeugen
      Konsolenuhr uhr=new Konsolenuhr();
      // 2. Thread starten
      uhr.start();
}
   // 3. Aufruf von "start" führt zur Ausführung von "run"
   public void run(){
      while(true) {
         try {
            Thread.sleep(1000);
         } catch(InterruptedException ie) {}
         date=new Date();
         sdfDatum=new SimpleDateFormat(timeFormat);
         // '\r' (Carriage Return) bewirkt Rücksprung
         // an den Zeilenanfang:
         System.out.print('\r'+sdfDatum.format(date));
      }
   }
}
```

Eine einfache Digitaluhr zur Ausgabe auf Konsole (CMD)

Was tut dieser Thread in seiner run-Methode?

Ein zweiter Weg zum Thread

Die Konsolenuhr aus Code 1 beschäftigt sich ausschließlich mit der Zeitausgabe. Stellen Sie sich nun vor, Sie wollten eine solche Uhr nicht in Form einer simplen Konsolenapplikation laufen lassen, sondern in einem Frame bzw. JFrame oder in Form eines Applets. Dann bekommen Sie ein Problem: Diese Klasse müsste von „java.awt.Frame“, „javax.swing.JFrame“ oder „java.applet.Applet“ erben, um ein Frame bzw. JFrame oder ein Applet zu sein, sie kann dann aber nicht mehr von „java.lang.Thread“ erben, denn **Mehrfachvererbung** ist in Java nicht zulässig.

Müssen wir daraus schließen, dass Frame- bzw. JFrame- oder Applet-Klassen niemals als Thread ausgebildet werden können?

Im Grunde gilt diese Überlegung auch für die Klasse „Thread“. Denn woher bezieht diese Klasse eigentlich ihre Fähigkeit, Thread zu sein? An sich ist „Thread“ keine besonders außergewöhnliche Klasse. Sehen Sie sich deren Klassendeklaration an: „Thread“ erbt ganz einfach von der Stammklasse aller Java-Klassen, der Klasse „java.lang.Object“.

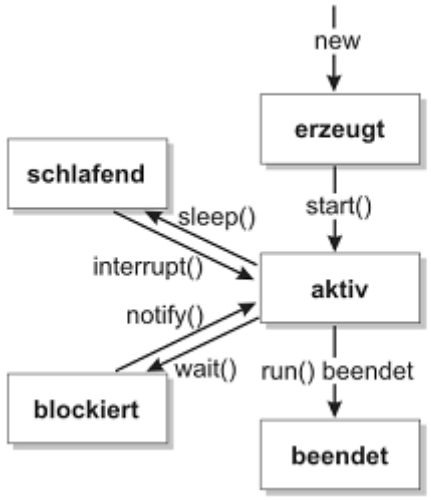
Wenn Sie nun noch einmal Ihren Blick über Code 1.1 streifen lassen und nach Besonderheiten in dieser Klasse Ausschau halten, so fällt insbesondere der Mechanismus auf, dass die Virtuelle Maschine nach einem Aufruf der Thread-Methode „start“ automatisch dafür sorgt, dass die run-Methode aufgerufen wird. Damit haben Sie bereits ein Kernstück der Thread-Funktionalität: Ein Thread benötigt immer eine **run**-Methode, in der sein Verhalten beschrieben ist. Auch die Klasse „Thread“ kennt diese run-Methode, die in der Methode Summary der API-Dokumentation aufgeführt ist.

```
import java.util.Date;
import java.text.SimpleDateFormat;
import java.awt.Font;

public class AppletUhr extends java.applet.Applet
implements Runnable {

   private Date date;
   private String timeFormat="HH:mm:ss";
   private SimpleDateFormat sdfDatum;
   private Font fett;

   public void init() {
      fett=new Font("Monospaced", Font.BOLD, 16);
   }
   public void run(){
      // ...
   }
   public void paint(java.awt.Graphics g) {
      date=new Date();
      g.setFont(fett);
      sdfDatum=new SimpleDateFormat(timeFormat);
      g.drawString(sdfDatum.format(date),20,30);
   }
}
```

Grundstruktur eines Applets mit Thread-Funktionalität

Zustände von Threads

Ursprünglich hatte Sun Microsystems das Verhalten von Threads sehr ähnlich zum Verhalten von Applets aufgebaut. Beide Klassen kennen nämlich nicht nur die bereits mehrfach angesprochene start-Methode, sondern auch eine **stop-Methode** mit identischer Deklaration. Man stellte sich vor, dass Threads je nach Bedarf gestartet und gestoppt werden können, also mal etwas tun und mal ihre Arbeit einstellen. Seit dem JDK ist diese stop-Methode der Klasse „Thread“ als „**deprecated**“ („veraltet“ oder auch schärfer: „missbilligt“) bezeichnet worden. Man hat erkannt, dass der abrupt erfolgende Stopp der Thread-Arbeit zu inkonsistenten Zuständen führen kann. Sie sollten daher die Arbeit eines Threads nicht in der Form threadObjekt.stop() abbrechen. Gleiches gilt für das Methodenpärchen „void **suspend**()“ und „void **resume**()“, mit dem man die Arbeit eines Threads einstellen und wieder aufnehmen konnte. Auch diese Methoden gelten nun als „deprecated“. Damit reduzieren sich die noch gültigen Zustände eines Threads im Wesentlichen auf jene fünf, die in Abb. 1.5 dargestellt sind:



Zustände eines Threads

Der Lebenszyklus eines Threads beginnt mit seiner Erzeugung durch Aufrufeines Thread-Konstruktors zusammen mit dem Operator „new“.

1. Der Thread beginnt seine Arbeit mit dem Aufruf seiner **start**-Methode.
2. Die Arbeit eines Threads ist beendet, wenn seine **run**-Methode abgearbeitet ist.
3. Darüber hinaus kann man einen Thread durch Aufruf der **sleep**-Methode schlafen legen – das haben Sie bereits ausprobiert. Aus dem Schlaf kann ein Thread gezielt durch Aufruf der **interrupt**-Methode aufgeweckt werden.
4. Schließlich kann die Arbeit eines Threads durch Aufruf der **wait**-Methode zeitweise blockiert werden. Ein Aufruf der **notify**-Methode macht ihm die Fortsetzung seiner Arbeit möglich. Mit diesem Methodenpärchen wird sich Lektion 3 dieses Hefts intensiver beschäftigen.

Häufig wird das Verhalten eines Threads in der run-Methode durch eine endlos laufende while-Schleife beschrieben, wenn der Thread dauerhaft etwas Bestimmtes tun soll, wie z. B. unser Uhren-Thread. Da stellt sich natürlich die Frage, wie man jemals einen solchen Thread beenden soll, wenn einerseits die run-Methode nie von sich aus zu einem Ende findet und andererseits ein stop-Aufruf nunmehr verboten (bzw. nicht gerne gesehen) ist. Eine Lösung sollte so aufgebaut werden, dass eine while-Schleife in der runMethode nach Möglichkeit immer eine **Gültigkeitsbedingung** prüft. Wenn diese Bedingung nicht mehr zutrifft, soll die Abarbeitung der while-Schleife – und damit auch die run-Methode – enden. Im Beispiel des Uhren-Applets könnte man sich wünschen, dass der Uhren-Thread seine Arbeit einstellt, sobald das Appletviewer- bzw. Browser-Fenster nicht mehr sichtbar ist, weil es entweder auf die Task-Leiste minimiert, oder weil die HTMLSeite mit dem Uhren-Applet verlassen wurde. In diesen Fällen ruft der Browser die stop-Methode des Applets auf. Das kann man sich zunutze machen und in der Applet-stop-Methode die Referenz auf das Applet-Objekt löschen: uhr=null; Mit dieser Maßnahme allein wird der Thread noch nicht beendet. Die while-Schleife kann man aber von der Bedingung abhängig machen, dass sie nur ausgeführt wird, solange das Thread-Objekt ungleich „null“ ist:

while(uhr != null) Das ermöglicht dem Thread, jedenfalls den aktuellen Durchlauf in der whileSchleife geordnet zu Ende zu bringen, ohne dass er erneut in die Abarbeitung der while-Schleife einsteigt. Damit endet dann auch die run-Methode (vergleichen Sie noch einmal Abb. 1.5) und der Thread erreicht den Zustand „beendet“. Nun soll die Uhr aber wieder laufen, wenn das Applet mit Thread in einer Anwendung erneut sichtbar wird. Nachdem wir zuvor das Thread-Objekt auf „null“ gesetzt haben, muss es nun erneut erzeugt werden, damit der Lebenszyklus wieder von vorne beginnen kann. Sie verlagern deshalb die Thread-Erzeugung in die startMethode des Applets, sodass das Applet den Lebenszyklus des Threads nun wie folgt steuert:

```
public void start(){
   if(uhr==null){
      uhr=new Thread(this);
      uhr.start();
   }
}
public void stop(){
   uhr=null;
}
public void run(){
   while(uhr != null) {
      ...
   }
}
```

Übernehmen Sie die beschriebenen Änderungen in den Code von „AppletUhr“.

Ihr Applet-Thread wird nun auf eine nicht-„deprecated“-Weise gestartet und gestoppt.

Zusammenfassung:

Auf jedem Multitasking-Computer lassen sich mehrere Anwendungen (Programme) parallele treiben. Ein Programm kann sich in mehrere Prozesse aufspalten. Neben jenen Programmen, die in separaten Fenstern tätig sind, arbeiten auf einem Multitasking-Computer weitere Prozesse im Hintergrund. Ein Prozess ist immer auch ein Thread, kann seine Arbeit aber auch von mehreren nebenläufig in einem gemeinsamen Speicherbereich aktiven Threads ausführen lassen. Tatsächlich kann ein Rechner mit einer CPU immer nur einen Prozess bedienen. Der Eindruck von Multitasking, Multiprozessing bzw. Multithreading entsteht, weil die CPU mit hoher Geschwindigkeit ständig Kontextumschaltungen vornimmt und so alle Prozesse und Threads mit Rechenzeit bedient.
Für die Programmierung eines Threads mit Java gibt es zwei Möglichkeiten:

- Sie beerben die Klasse „java.lang.Thread“ und überschreiben deren runMethode. Dann kann diese Klasse direkt als Thread-Objekt instanziiert und gestartet werden.
- Alternativ schreiben Sie eine Klasse, die das Interface „java.lang.Runnable“ implementiert. Dies zwingt zur Implementierung der run-Methode, in der Sie das Verhalten des Threads programmieren. Das Thread-Objekt selbst wird aus der Klasse „java.lang.Thread“ instanziiert. Als „Runnable target“ ist in diesem Fall ein Objekt der Klasse anzugeben, in der Sie die run-Methode programmiert haben.

Ein Thread beendet seine Arbeit, wenn seine run-Methode abgearbeitet ist. Zwischendurch kann er schlafen gelegt und wieder aufgeweckt bzw. in Wartestellung gebracht und wieder aktiviert werden. Das sind die wesentlichen Zustände, die ein Thread in seinem Lebenszyklus annehmen kann.