Es gibt 2 Möglichkeiten Daten in mehreren Threads gemeinsam zu nutzen:

1. Daten werden als **Parameter** an den Konstruktor der Thread-Klasse übergeben.

   a) Wird ein Objekt einer Klasse an einen Thread übergeben, wird es als (Referenz-) Attribut gespeichert und steht damit in dem Thread mit Vollzugriff (Lesen und Schreiben) zur Verfügung. Der Zugriff auf die Attribute des Objektes erfolgt dabei wie in der OOSE üblich über die Methoden, die in der Klasse des übergebenen Objektes zur Verfügung stehen (z.B. Getter und Setter).

   b) Wird ein Attribut von einem Objekt nur als Wert (eines primitiven Datentyps) übergeben (Call-by-value), muss dieses Attribut in der zugehörigen Klasse des Objektes als statisches (Klassen-) Attribut angelegt sein, damit der gemeinsam zu nutzende Wert auch wirklich nur einmal (als Speicherreferenz) in allen Objekten der Thread-Klasse zur Verfügung steht.

2. Die Thread-Klassen werden als **innere Klasse** der Anwendung deklariert. Damit haben Sie Zugriff auf alle Attribute der äußeren Klasse.
   Bei der GUI-Programmierung verwendet man solche Konstrukte sehr häufig. Für erste Übungen sollte man das erst mal nicht so machen.

Ein Beispiel (ist gleichzeitig auch eine Programmierübung) :

(Zugehöriges UML-Klassendiagramm siehe nächste Seite)

Monika ist ein Mensch und Rainer ist ein Mensch. Gemeinsam sind sie ein Paar und die **2 Menschen haben ein gemeinsames Konto**. Beide wollen auf das Konto zugreifen, also Geld einzahlen und Geld abheben.

Zu Beginn eines Monats zahlt jede(r) einen Betrag auf das Konto ein. Im Laufe des Monats hebt jede(r) evtl. mehrmals Geld ab. Monika und Rainer haben vereinbart, das Konto nicht zu überziehen, d. h. bevor eine(r) etwas abhebt, wird der Kontostand geprüft.

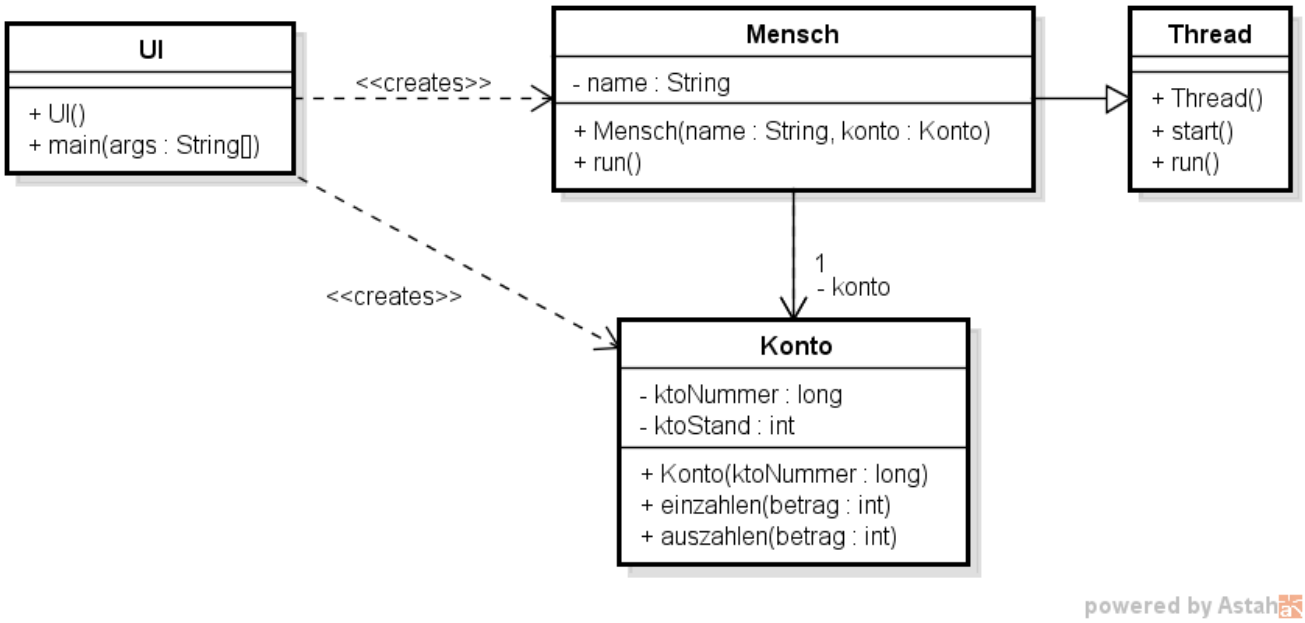
Programmiertechnisch heißt das:

Monika und Rainer sind Objekte der Klasse `Mensch`. Die Klassen `Mensch` erbt von `Thread`, die erzeugten `Mensch`-Objekte „Rainer“ und „Monika“ sind also in eigenen Threads parallel aktiv. Sie nutzen ihr Konto, ein Objekt der Klasse `Konto`, gemeinsam, jede/r zahlt auf ihr gemeinsames Konto ein und hebt ab.

Die `UI`-Klasse erzeugt ein `Konto`-Objekt und die beiden `Mensch`-(Thread-) Objekte Rainer und Monika in eigenen Threads. Das `Konto` -Objekt wird bei der Erzeugung der (Thread-) Objekte an beide Thread-Objekte übergeben.

In der `run()` - Methode der „Menschen“ passiert (**wieder als Simulation**) Folgendes:

Es wird zuerst ein Betrag eingezahlt (Monatsanfang) und dann wiederholt (n-mal) von beiden Menschen Beträge vom Konto abgehoben (im Verlauf des Monats). Vor jedem Abheben wird der Kontostand geprüft, es wird nur abgehoben, wenn der Kontostand > 0 ist.



Aufgabe 1:

Implementieren Sie die Klasse `Mensch`, die von der Klasse `Thread` erbt. Testen Sie die UI-Anwendung. Sie finden Vorlagen für die Klassen `UI` und Hilfen zur Implementierung der `run()`-Methode der Klasse `Mensch` im **Material 1.**

Aufgabe 2:

Dokumentieren Sie den Ablauf in einem Sequenzdiagramm, d. h. das Erzeugen der Mensch-Objekte und das Einzahlen und Abheben vom Konto.
Details dazu und eine Vorlage finden Sie im **Material 2**.

Aufgabe 3:

Beispiel für eine Ausgabe:

```
Monika Kontostand: 0
Monika hat 500 Euro eingezahlt, neuer Kontostand: 500
Rainer Kontostand: 0
Monika will 72 Euro abheben.
Rainer hat 500 Euro eingezahlt, neuer Kontostand: 1000
Rainer will 63 Euro abheben.
Rainer hat 63 Euro abgehoben, neuer Kontostand: 865
Monika hat 72 Euro abgehoben, neuer Kontostand: 928
Rainer will 86 Euro abheben.
Rainer hat 86 Euro abgehoben, neuer Kontostand: 779
Monika will 60 Euro abheben.
Monika hat 60 Euro abgehoben, neuer Kontostand: 719
```

Die ersten Zeilen scheinen in der falschen Reihenfolge zu sein, woran kann das liegen?

Kommt es vor, dass ein negativer Kontostand erreicht wird? Wenn ja, woran kann das liegen?

Lesen Sie weiter auf der nächsten Seite!

Probleme beim Zugriff auf gemeinsame Daten – Kritische Abschnitte synchronisieren

Da Threads ohne Übergabe von Objekten / Daten intern oft ihre eigenen Daten verwalten - sie haben alle eigene lokale Variablen - kommen sie sich für den Fall nicht in die Quere. Auch wenn mehrere Threads gemeinsame Daten nur lesen, ist das unbedenklich. **Schreiboperationen sind jedoch kritisch**. Wenn sich zehn Nutzer einen Drucker teilen, der die Ausdrucke nicht als unteilbare Einheiten bündeln würde, lässt sich leicht ausmalen, wie das Ergebnis aussieht. Seiten, Zeilen oder gar einzelne Zeichen aus verschiedenen Druckaufträgen werden bunt gemischt ausgedruckt.

Zusammenhängende Programmblöcke, die nicht unterbrochen werden dürfen und besonders geschützt werden müssen, sind sogenannte **kritische Abschnitte**.

Einfaches Beispiel:
```
public void foo() { i++; } // der Wert der Variable i soll um 1 erhöht werden
```

Zuerst wird `i` gelesen und auf dem Stack[1] abgelegt. Danach wird die Konstante 1 auf den Stack gelegt, und anschließend zu `i` addiert. Das Ergebnis steht wiederum auf dem Stack und wird zurück in `i` geschrieben.

Wenn zwei Threads A und B die Methode `foo()` parallel ausführen, kann folgende Situation eintreten:

Thread A holt sich den Wert von `i` in den internen Speicher (seinen Stack), wird dann aber unterbrochen. Er kann das um 1 erhöhte Resultat nicht wieder `i` zuweisen.
Nach der Unterbrechung von A kommt Thread B an die Reihe. Auch er liest den Wert von `i`, kann aber `i + 1` berechnen und das Ergebnis in `i` ablegen. Dann ist B beendet, und Thread A kommt wieder dran.
Jetzt steht in `i` das von Thread B um 1 erhöhte `i`. Thread A addiert nun 1 zu dem gespeicherten alten Wert von `i` und schreibt dann nochmals denselben Wert wie Thread B zuvor. Insgesamt wurde die Variable `i` nur um 1 erhöht, obwohl zweimal inkrementiert werden sollte. Jeder Thread hat für sich gesehen das korrekte Ergebnis berechnet.

Solche **kritischen Abschnitte** können mit dem Schlüsselwort **`synchronized` geschützt werden**. Im einfachsten Fall markiert dieser sogenannte Modifizierer **`synchronized`** eine vollständige Methode.
Arbeitsweise:
Betritt ein Thread A eine synchronisierte Methode eines Objekts O und versucht anschließend Thread B eine synchronisierte Methode des gleichen Objekts O aufzurufen, muss der nachfolgende Thread B so lange warten, bis A wieder aus dem synchronisierten Teil austritt. Das geschieht, wenn der erste Thread A die Methode verlässt, denn mit dem Verlassen einer Methode – oder auch einer Ausnahme – gibt die JVM automatisch den Lock frei.

```
public synchronized void foo() { i++; } // löst das Problem
```

Damit wird die Methode `foo()` am Stück ausgeführt und kann nicht unterbrochen werden. So wird garantiert, dass die Variable `i` nicht von jemand anderem überschrieben werden kann.

Aufgabe 4:
Im Beispiel „Monika und Rainer und ihr gemeinsames Konto“: Synchronisieren Sie das Einzahlen und das Abheben der Geldbeträge auf das gemeinsame Konto. Überprüfen Sie dann anhand der Ausgaben, ob damit das gemeinsame Konto immer (also nach den Ein- und Auszahlen) konsistente Daten hat (= in der Konsole anzeigt)

[1] Der Stack ist ein Speicherbereich, in dem ein Thread Variablen mit ihren Werten zwischenspeichert.

Material 1 zur Aufgabe 1:
Klasse `UI`:

```
public class UI {
  public UI() {
     Konto konto = new Konto(123456); // Objekte erzeugen
     new Mensch("Monika",konto);
     new Mensch("Rainer",konto);
  }
  public static void main(String[] args) {
    new UI();
  }
}
```

Klasse `Konto`: Siehe UML-Klassendiagramm.

Klasse `Mensch`:
Einige wichtige (aus Platzgründen nicht optimal formatierte) Programmcodeausschnitte aus der Klasse `Mensch`. Nicht abgedruckter Programmcode ist aus dem UML-Klassendiagramm abzuleiten und zu implementieren:

```
public class Mensch extends Thread {
  …
    public Mensch(String name, Konto konto) {
    …
  }
  public void run() {
    Random zufallsgenerator = new Random();
    int betrag;
    konto.einzahlen(500);  // Monatsanfang, Geld einzahlen

    System.out.println(name + " hat 500 Euro eingezahlt, "
    + "neuer Kontostand: " + konto.getKtoStand()+'\n');

    for (int i = 1; i < 21; i++){ // während des Monats, Geld abheben, z.B. 20 mal
      // Betrag zwischen 50 und 100 als Zufallszahl ermitteln
      betrag = zufallsgenerator.nextInt(50) + 50;

      System.out.print(name + " will " + betrag + " Euro abheben.");

      // prüfen, ob genug Geld auf dem Konto ist
      if (konto.getKtoStand() > betrag) {
        konto.auszahlen(betrag);
        System.out.println("-> " + name + " hat " + betrag + " Euro abgehoben, "
                           + "neuer Kontostand: " + konto.getKtoStand()
                           + " (Abhebung Nr. " + i +")");
      }
      else{
        System.out.println(" Nicht genug Guthaben. (Abhebung Nr. " + i +")");
        konto.einzahlen(250);  // Konto auffüllen, Geld einzahlen
        System.out.println(name + " hat 250 Euro eingezahlt, "
        + "neuer Kontostand: " + konto.getKtoStand() + "\n");
      }
      try {
        Thread.sleep(zufallsgenerator.nextInt(5000)); // Zeit einstellbar
      }
      catch(Exception e) {e.printStackTrace();}
    }
  }
}
```

Material 2 zur Aufgabe 2: Sequenzdiagramm

Ergänzen Sie das Sequenzdiagramm für den `main()`-Methodenaufruf der Klasse `UI`. Im Sequenzdiagramm soll auch der Aufruf der `start()`-Methode, die ja dann die `run()`-Methode startet UND auch die Sequenz für die `run()`-Methode sowohl für das Objekt `monika` als auch für das Objekt `rainer` enthalten sein.