1. Einführung in die objektorientierte Programmierung (OOP)

1.1 Objekte:

Ein Computerprogramm ist oft ein vereinfachendes Abbild einer realen Situation. Das objektorientierte Denkschema geht davon aus, das gesamte System bestehe aus **Objekten**. Die Software-Objekte sind dabei Abstraktionen der realen Welt, bei denen nur die für die Problemstellung wesentlichen Bestandteile im Software-Modell nachgebildet werden.
Jedes Objekt in der realen Welt verfügt über Attribute (Eigenschaften), die seinen momentanen Zustand beschreiben (die Ampel ist rot, die Stoppuhr ist gestoppt, das Gerät ist aus...) und zeigt gewisse Verhaltensweisen (Ampel schaltet auf grün...).
Bei den Objekten des Software-Modells werden

- die **Attribute** in **Variablen** festgehalten,
- die möglichen Aktionen oder Reaktionen durch **Methoden** (Prozeduren oder Funktionen) beschrieben.

1.2 Klassen und ihre Instanzen:

Eine **Klasse** beschreibt die gemeinsamen Attribute und Methoden einer Menge von Objekten, sie ist gewissermaßen der abstrakte Bauplan der Objekte.

Eine Klasse wird in der Regel in einer eigenen Unit deklariert und implementiert. Ein konkretes Objekt dieser Klasse „aus Fleisch und Blut" nennt man eine **Instanz** dieser Klasse. Instanzen der Klasse werden erst in einem Anwendungsprogramm verwendet. Dies hat den Vorteil, dass ganz verschiedene Anwendungsprogramme auf die Unit zugreifen können, in der die Klasse implementiert ist, und Instanzen dieser Klasse verwenden können.
Ein Beispiel ist die in der Lazarus-Unit `ExtCtrls` implementierte Klasse `TShape`, mit der Rechtecke und Ellipsen erzeugt werden können. Instanzen der Klasse `TShape` bekommen in Anwendungsprogrammen zunächst die von Lazarus vorgegebenen Standardnamen `Shape1`, `Shape2` usw. Sie haben alle dieselben Attribute (z.B `width`, `height`), die jedoch bei verschiedenen Instanzen verschiedene Werte annehmen können.

1.3 Unterrichtsziel

In Zukunft werden Sie nicht mehr nur kleine Anwendungsprogramme schreiben; vielmehr sollen Sie, als Vorstufe dazu, für die jeweilige Anwendung passende Klassen modellieren und implementieren. Das Anwendungsprogramm, in dem Instanzen dieser Klassen verwendet werden, ist erst der zweite Schritt.

1.4 Deklaration von Klassen

Eine Klasse wird unter **`type`** deklariert:

Beispiele: `type TForm1 = class(TForm)` Automatisch definierte Klasse beim Gestalten eines Formulars
`type TAuto = class(TObject)` Selbstdefinierte Klasse

Allgemein: `type TKlasse = class(TVorfahrklasse)` siehe 1.5 Vererbung

Zwar ist es nicht zwingend vorgeschrieben, aber dringend zu empfehlen, den Namen einer selbstdefinierten Klasse mit einem **großen T** (wie „Typ") beginnen zu lassen, um sie deutlich von den Instanzen dieses Typs zu unterscheiden.

In der Typdeklaration der Klasse sind nur die Attribute und die Kopfzeilen der Methoden angegeben. Die eigentliche Programmierung der Methoden erfolgt erst unter **`implementation`**.

```
type
  TForm1 = class(TForm)
    Btstart: TButton;
    Label1: TLabel;
    procedure FormCreate(Sender: TObject);
    procedure BtstartClick(Sender: TObject);
  private
    augenzahl:integer;
    procedure wuerfeln;
  end;
```

1.5 Vererbung und der „Urahn" TObject.

In diesen Beispielen baut jede neu definierte Klasse auf einer bereits in Lazarus implementierten **Vorfahr**klasse auf, von der sie alle Attribute und Methoden erbt. Sie kann diese verändern und eigene, neue Attribute und

Methoden hinzufügen. Die so definierte Nachfolgerklasse „kann“ alles, was die Vorfahrklasse kann, aber eben noch mehr.

Beispiel: Die in Lazarus implementierte Klasse `TForm`, die Vorfahrklasse aller Formulare, verfügt über die Eigenschaft `color`, mit der die Hintergrundfarbe festgelegt werden kann. Dies ist dann auch beim **Nachkommen** `TForm1` möglich; dagegen gibt es in `TForm1` beispielsweise einen Button `BtStart` und eine Prozedur `BtStartClick`, die in `TForm` nicht bekannt sind.

Selbstdefinierte Klassen sind oft Nachkommen der in Lazarus implementierten Klasse **`TObject`**, der Klasse aller Lazarus-Objekte. Sämtliche bereits in Lazarus vorhandene Klassen sind Nachkommen von `TObject`, oft über mehrere „Generationen“ hinweg.

Auch selbstdefinierte Klassen müssen nicht direkte Nachkommen von TObject sein, sondern können ebensogut von anderen, auch von selbstdefinierten Klassen abgeleitet sein:

```
type TMercedes = class(TAuto)
```

2. Erstes Beispiel: Konstruktion der Klasse TStoppuhr

2.1 Vorüberlegung: Eigenschaften und Methoden einer Stoppuhr

Methoden: Was soll man mit einer Stoppuhr tun können?
Attribute: Welche Variablen sind dazu nötig?

Diese Überlegungen führen auf die Prozeduren `starten`, `stoppen`, `reset` und die Funktion `Laufzeit`. An Variablen braucht man `tstart`, `tstop:longint` (Zeitpunkt, zu dem die Uhr gestartet bzw. gestoppt wird) und eine boolsche Variable `aktiv`, die angibt, ob die Uhr gerade läuft oder nicht.

2.2 Sichtbarkeit und Geheimnisprinzip...

Für die Methoden und Attribute einer Klasse gibt es verschiedene Stufen der Sichtbarkeit (**Schutzklassen**):

- **private**: nur in der Unit sichtbar, in der die Klasse deklariert und implementiert wird.
- **protected**: außerdem sichtbar in Units, in denen Nachkommenklassen implementiert werden.
- **public**: außerdem in allen Units sichtbar, die eine Instanz der Klasse verwenden. Unter **public** wird man also die Attribute und Methoden deklarieren, auf die das Anwendungsprogramm Zugriff haben muss.

Es gilt das **Geheimnisprinzip**:
Grundsätzlich sollte so wenig wie möglich der in einem Objekt gespeicherten Information nach außen hin sichtbar sein. Dadurch wird die Wartungsfreundlichkeit und die Stabilität eines Programms erhöht.

2.3 ...angewandt bei TStoppuhr

Der Anwendungsprogrammierer darf zwar wissen, *was* er mit seiner Stoppuhr tun kann, aber nicht, *wie* sie intern funktioniert.

Daher empfiehlt sich (nur in diesem Beispiel, keinesfalls immer) folgendes Vorgehen:
Alle in 2.1 aufgeführten Methoden werden der Schutzklasse **public** zugeordnet, da der Anwendungsprogrammierer darauf Zugriff haben muss, um beispielsweise die Uhr zu stoppen.

Die Variablen dagegen kommen in die Schutzklasse **protected**. Denn um etwa der Startzeit einen Wert zuzuweisen, werden Sie die Windows-Funktion `gettickcount` verwenden, die angibt, wieviele Millisekunden seit dem Booten vergangen sind – für den Anwendungsprogrammierer eine völlig überflüssige Information. Andererseits sollte man die Variablen nicht unter **private** verstecken. Die Möglichkeit, einen Nachkommen von `TStoppuhr` zu programmieren, sollte man sich immer offen halten; und dort sollten die Variablen bekannt sein.

2.4 Der Konstruktor create

Zu den Methoden einer Klasse gehört **immer** auch eine besondere Prozedur, die nicht mit dem Schlüsselwort **`procedure`**, sondern mit **`constructor`** eingeleitet wird. Der Name dieser Prozedur ist immer `create`; je nach Situation mit oder ohne Parameter; deklariert wird sie unter **public**.

Zweck des Konstruktors:

- Beim Aufruf im Anwendungsprogramm: Erschaffung eines Objekts dieser Klasse, d.h. Bereitstellung von Speicherplatz.
- Bei der Implementierung der Klasse: Initialisierung von Variablen.

Damit ist den Methoden in 2.1 noch **`constructor`** `create` hinzuzufügen.

Anmerkung: Bei den von Lazarus automatisch erzeugten Objekten (Formulare, Buttons, Shapes...) dürfen Sie den Konstruktor weder deklarieren noch aufrufen, da dies bereits intern geschieht.

2.5 Aufgabe: Definieren Sie die Klasse TStoppuhr in einer eigenen Unit

1. Wählen Sie nach dem Start von Lazarus **„Datei/Neue Unit".** Das Projekt enthält jetzt auch eine Unit2.
   „Projekt/Projekt speichern unter...": Benennen Sie Unit1 in uHaupt, Unit2 in uStoppuhr um; Projekt1 bleibt unverändert. Speichern Sie alles in einen neuen **Ordner** „01Stoppuhr"**.**
   Beginnen Sie mit der Unit uStoppuhr:

2. Da wir die Funktion `gettickcount` (liefert Millisekunden seit dem Booten) brauchen, die in einer Unit abgelegt ist, müssen wir gleich am Anfang des interface-Teils mit **`uses`** die entsprechende Bibliothek `Windows` einbinden:
   `uses` `windows;`

3. Tragen Sie anschließend die Klassendeklaration ein:

```
type
  TStoppuhr = class(TObject)
  protected
    tstart, tstop:longint;
    aktiv:boolean;
  public
    constructor create;
    procedure starten;
    procedure stoppen;
    procedure reset;
    function Laufzeit:real; //Zeitraum in Sekunden zwischen Starten und Stoppen der Uhr
  end;
```

4. Unter **`implementation`** erfolgt die Kodierung der einzelnen Methoden. Der Klassentyp, also `TStoppuhr`, muss getrennt durch einen Punkt, dem Namen der Prozedur oder der Funktion vorangestellt werden. (Wie z.B. in **`procedure`** `TForm1.Button1Click(Sender:TObject);`)

   a) Wir beginnen mit dem Konstruktor:

```
constructor TStoppuhr.create;
begin
  inherited create;
  //Initialisierungen
  tstart:=0;
  tstop:=0;
  aktiv:=false;
end;
```

   Die Anweisung **`inherited`** `create` **darf nicht** im Konstruktor **fehlen**. Sie sorgt dafür, dass der Vorfahr (hier: `TObject`) der Nachkommenklasse `TStoppuhr` alle seine Methoden und Attribute vererbt, so dass sie auch dort bekannt sind.
   Da eine neu erzeugte Stoppuhr zunächst steht, wird der Status `aktiv` mit `false` initialisiert.

   b) Als nächstes folgt die Prozedur `starten`:

```
procedure TStoppuhr.starten;
begin
  if not aktiv
  then begin
    tstart:=gettickcount;
    aktiv:=true;
  end;
end;
```

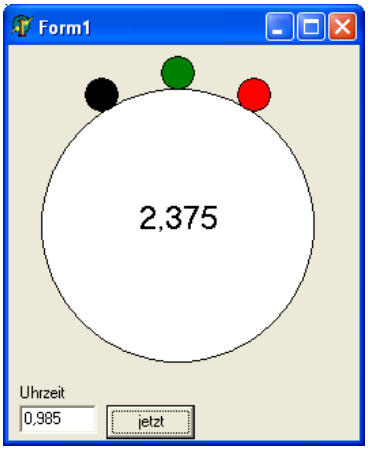
   Die Uhr kann nur gestartet werden, wenn sie gerade nicht läuft.

   c) Erstellen Sie entsprechend die Prozedur `stoppen`.

   d) Die Prozedur `reset` initialisiert alle Zeiten mit 0, falls die Uhr gerade nicht läuft. (Andernfalls hat sie keine Auswirkung).

   e) Implementieren Sie die Funktion `Laufzeit`. Vergessen Sie dabei nicht, die Differenz aus `tstop` und `tstart` von Millisekunden in Sekunden umzuwandeln.

2.6 Aufgabe: uHaupt – Anwendung für die Klasse TStoppuhr



1. Das Formular: Vier kreisförmige Shapekomponenten (Register *Additional*); die drei kleinen Kreise mit den Namen `ShReset` (schwarz), `ShStart` (grün), `ShStop` (rot). In der Mitte des weißen Kreises befindet sich das ebenfalls weiß gefärbte Label `llzeit`.
   Das Labelededit „Uhrzeit“ und der Button „Jetzt“ sind für spätere Erweiterungen des Programms gedacht.

2. Fügen Sie in der Unit `uHaupt` der **`uses`**-Liste noch `uStoppuhr` hinzu, damit Sie die Klasse `TStoppuhr` (hier nur mit ihren unter **`public`** deklarierten Methoden) verwenden können.

3. Im **`private`**-Bereich des Formulars deklarieren Sie die Variable `uhr1:TStoppuhr;`

4. **Wichtig:** Damit die Stoppuhr überhaupt verwendet werden kann, muss sie zunächst **erzeugt** werden. Dies erfolgt häufig beim Erzeugen des Formulars. (Ereignis `OnCreate`) Tragen Sie in der zugehörigen Prozedur `FormCreate` die folgende Anweisung ein:

   ```
   uhr1:= TStoppuhr.create;
   ```

5. Für die drei Knöpfe der Uhr können Sie das `Mousedown`-Ereignis (Objektinspektor!) der Klasse TShape verwenden.

   ```
   procedure TForm1.ShStartMouseDown(Sender: TObject;...);
   begin
     uhr1.starten;
     llzeit.caption:='Uhr läuft!';
   end;
   ```

   Programmieren Sie entsprechend das Stoppen und Zurücksetzen der Uhr; geben Sie unmittelbar nach dem Stoppen die Laufzeit aus.

6. Beim Beenden einer Anwendung sorgt Lazarus zwar dafür, dass erzeugte Objekte wieder gelöscht werden. Für einen **guten Programmierstil** sollte der Programmierer aber erzeugte Objekte, die nicht mehr benötigt werden, mit der **Methode free** löschen und damit den Speicher wieder freigeben.
   Löschen Sie das Objekt Uhr1, wenn das Formular geschlossen wird (Formular-Ereignis `OnClose`):

   ```
   Uhr1.free;
   ```

2.7 Erweiterte Anwendung: uHaupt – Eine zweite Stoppuhr

Diese soll die gesamte Laufzeit des Programms messen. Sie wird also sofort beim Erzeugen des Formulars gestartet und beim Schließen wieder gestoppt. Die Ausgabe der Laufzeit erfolgt über ein Meldefenster: `Showmessage('...')`.

1. Führen Sie unter **`private`** noch eine Variable `uhr2` ein.
2. `Uhr2` muss unter `Formcreate` erzeugt und sofort gestartet werden.
3. In der Prozedur `FormClose` wird sie gestoppt und die Laufzeit angezeigt.
4. Löschen Sie auch `Uhr2` beim Schließen des Formulars; beachten Sie, dass nach `uhr2.free;` keine weiteren `uhr2` betreffenden Anweisungen mehr möglich sind. (Andernfalls produzieren Sie prächtige Systemabstürze.)

2.8 Änderung: Ständiges Aktualisieren des Zeitlabels → Property istaktiv.

Anstelle einer Meldung „Zeit läuft!“ o.ä. soll jetzt die Zeit nach dem Anklicken von `ShStart` jede Millisekunde im Zeitlabel aktualisiert werden. Beim Stoppen bleibt sie stehen. Dies erfordert kleine Ergänzungen:

1. `uStoppuhr`: Eine neue, unter **`public`** zu deklarierende Funktion `jetztzeit:real` gibt, falls die Uhr läuft, die seit dem Starten vergangene Zeit in Sekunden an; andernfalls (d.h bei `aktiv = false`) den Wert Null.

2. `uStoppuhr` soll so verändert werden, dass man vom Hauptprogramm aus auf die Variable `aktiv` zugreifen kann. Dies ist bisher nicht möglich, da `aktiv` unter **`protected`** deklariert ist.
   1. Möglichkeit: `aktiv` wird unter **`public`** deklariert: Sehr schlecht, da die Variable `aktiv` im Hauptprogramm nur gelesen, aber keinesfalls durch einen direkten Schreibzugriff verändert werden darf.
   2. Möglichkeit: Unter **`public`** eine Funktion `istaktiv:boolean` deklarieren, die nur den Wert der Variablen `aktiv` zurückgeben soll. Möglich, aber lästige Schreibarbeit unter **`implementation`**. Am besten ist die
   3. Möglichkeit: Unter **`public`** lediglich folgendes einfügen: **`property`** `istaktiv:boolean` **`read`** `aktiv;`
   Das bedeutet, dass das Hauptprogramm bloß die Eigenschaft `istaktiv` der Uhr „sieht“, um z.B mit der Anweisung **`if`** `uhr1.istaktiv...` die interne Variable `aktiv` zu lesen. Diese jedoch ist nach wie vor für das Hauptprogramm unsichtbar und kann von dort aus nicht durch Schreibzugriff verändert werden.

3. `uHaupt`, Prozedur `ShStartMousedown`: Um ständig die Zeit zu aktualisieren, fügen Sie eine `While`-Schleife ein und verwenden Sie für die Einstiegsbedingung die Property `istaktiv` und im Rumpf die Funktion `jetztzeit`. Damit das Programm funktioniert, muss die Schleife auch die Anweisung

   ```
   Application.Processmessages;
   ```

   enthalten. Diese prüft bei jedem Schleifendurchlauf, ob inzwischen ein neues Ereignis wie etwa das Anklicken von `ShStop`, also Stoppen der Uhr, eingetreten ist, infolgedessen die Schleife verlassen wird.

4. Bei Klick auf den bisher nicht verwendeten Button „Jetzt“ wird im Editfenster „Uhrzeit“ ebenfalls die „Jetztzeit“ ausgegeben.

   Tipp: Wer die Zeit immer mit drei Dezimalen ausgeben will, muss anstelle einer simplen `floattostr`-Anweisung **Formatstrings** verwenden:

   ```
   llzeit.Caption:=format('%4.3f', [uhr1.jetztzeit]);
   ```