20.6.0 Task

Stellen Sie sich die folgende Situation vor: In einem Programm werden als Teilaufgabe in einer Prozedur P zeit-intensive Berechnungen ausgeführt. Das bedeutet, dass der Interpreter während der Ausführung von P gebunden ist. In der Zeit der Ausführung von P betritt er die Event-Schleife nicht, die als Ruhemodus des Interpreters angesehen werden kann. Dort wird auf externe Ereignisse gewartet, wie zum Beispiel ein Klick auf einen Button. Das bedeutet – während P ausgeführt wird – dass die GUI des Prozesses eingefroren ist!

Um trotzdem zeitnah auf ein Ereignis reagieren zu können, bieten sich zwei Möglichkeiten an:

- Sie forcieren an geeigneten Stellen im Quelltext den Eintritt in die Event-Schleife mit der Wait-Anweisung.
- Sie lagern die Prozedur P in einen Task als Hintergrund-Prozess aus. Damit wird die Bearbeitung von P von einer anderen Instanz des Interpreters in einem anderen Prozess ausgeführt. Ihr "Haupt-Interpreter" (welcher die GUI des Prozesses betreut) ist im Ruhemodus und kann auf Benutzer-Eingaben und die aus dem Task eintreffenden Ergebnisse *sofort* reagieren.

Auch in Gambas werden zum Beispiel in der Komponente *gb.form* Tasks verwendet, um eine Datei-Vorschau (Preview) zu generieren. Der Clou ist folgender: Während die Datei-Vorschau in einem Task erzeugt wird, kann das Hauptprogramm noch mit dem Benutzer interagieren. Es befindet sich nicht in einer zeitaufwändigen Routine und die Event-Schleife im Hauptprogramm läuft normal.

- Die Klasse *Task* existiert in der Komponente *gb* und ist somit Teil des Interpreters.
- Die Begriffe Task und Hintergrund-Prozess werden in diesem Kapitel synonym verwendet – klar abgegrenzt vom Task-Objekt als Instanz einer selbst zu schreibenden Task-Klasse, welche von der Klasse *Task (gb)* über die Instruktion 'INHERITS Task' erbt.

Ein Task stellt eine Kopie des Elternprozesses dar (Fork → http://de.wikipedia.org/wiki/Fork_%28Unix%29) und läuft als eigenständiger Prozess mit eigener Prozessnummer völlig unabhängig vom aufrufenden Elternprozess. Übergebene Variablen z.B. können deshalb im Task eigenständig verändert werden, ohne das der übergeordnete (Eltern-)Prozess etwas davon mitbekommt. Das gleiche gilt natürlich auch in der anderen Richtung.

In diesem Kapitel werden Ihnen neben den Eigenschaften, Methoden und Events der Klasse Task (gb) Projekte vorgestellt, die deren Verwendung zeigen.

20.6.0.1 Eigenschaften

Die Klasse *Task* hat drei Eigenschaften:

Eigenschaft	Datentyp	Beschreibung
Handle	Integer	Gibt die Prozess-Nummer (PID) des Hintergrund-Prozesses (Task) zurück.
Running	Boolean	Gibt True zurück, wenn der angegebene Hintergrund-Prozess ausgeführt wird.
Value	Variant	Gibt den Funktionswert der Main()-Funktion des Hintergrund-Prozesses zurück. Wenn im Hintergrund-Prozess ein Fehler auftrat, so wird ein Fehler ausgelöst, der über *Task.Value* ausgelesen und ausgewertet werden kann.

Tabelle 20.6.0.1.1 : Eigenschaften der Klasse Task

- Nachdem der Funktionswert der Main()-Methode zurückgegeben wurde, wird der Hintergrund-Prozess automatisch beendet und Sie können den Funktionswert (Datentyp Variant) im Event *Task_Kill()* über die Eigenschaft *Task.Value* auslesen.
- Ein Fehler im Hintergrund-Prozess löst auch das Event *Task_Kill()* aus und nicht – was ja nahe läge – das Event *Error(Data AS String)*, denn ein Fehler im Hintergrund-Prozess beendet den Hintergrund-Prozess auch sofort.
- Fehler-Informationen können auch über die Eigenschaft *Task.Value* ausgelesen werden.

20.6.0.2 Methoden

Die Klasse *Task* verfügt nur über zwei Methoden:

- Task.Stop: Der aktive Task als Hintergrund-Prozess wird beendet!
- Task.Wait: Warten auf das Ende des aktiven Hintergrund-Prozesses.

20.6.0.3 Ereignisse

In der Klasse *Task* sind diese drei Ereignisse deklariert:

Ereignis	Beschreibung
Read(Data As String)	Das Ereignis wird ausgelöst, wenn der Task Daten (Datentyp String) auf seine Standard-Ausgabe ausgibt. Normalerweise werden die Daten zeilenweise ausgegeben; es sei denn, der Task verwendet die FLUSH-Anweisung ohne Argument. Die Task-Standard-Ausgabe kann auch benutzt werden, um Task-Status-Informationen an den übergeordneten (Eltern-)Prozess zu senden.
Error(Data As String)	Das Ereignis wird ausgelöst, wenn der Task Daten auf seine Fehler-Ausgabe ausgibt.
Kill()	Das Ereignis wird ausgelöst, wenn der aktive Task als Hintergrund-Prozess beendet wurde.

Tabelle 20.6.0.3.1 : Ereignisse der Klasse Task

- Das Event *Error(Data AS String)* wird nur dann ausgelöst, wenn im Hintergrund-Prozess Daten, zum Beispiel mit dem ERROR-Befehl, in die Standard-Fehler-Ausgabe geschrieben wurden!

Anders als es das Event *Task_Kill()* vermuten lässt, wird das Ereignis *nicht* ausgelöst, wenn das Task-Objekt zerstört wird, sondern wenn der Task-Prozess beendet wird. *Beendet* bedeutet hier:

- Der Task wurde durch die Methode *Task.Stop* gestoppt oder
- nach dem Ende der Main()-Methode wird der Task automatisch beendet.

Zu betonen ist der Unterschied von Task als Hintergrund-Prozess – in dem der Task-Code ausgeführt wird – und Task-Objekt, das im übergeordneten Gambas-Prozess existiert. Das Task-Objekt erlaubt es Ihnen mit dem Task zu kommunizieren, solange der Task existiert. Das Task-Objekt aber 'überlebt' das Ende des Hintergrund-Prozesses, damit Sie nach dessen Ende noch den Funktionswert der Main()-Funktion (Datentyp Variant) über die Eigenschaft *Task.Value* oder andere Eigenschaften auslesen können.

20.6.0.4 Task als Hintergrund-Prozess in Gambas

Um eine Aufgabe im Hintergrund in einem Task bearbeiten zu lassen müssen Sie:

- zuerst eine Klasse mit frei wählbarem Klassen-Namen erzeugen, welche die Klasse Task erbt → Task-Klasse,
- dann in der Task-Klasse eine öffentliche Main()-Methode ohne Argument definieren, welche die zu bearbeitende Aufgabe beschreibt und
- danach im (Haupt-)Programm eine Instanz der Task-Klasse erzeugen, um eine neue Aufgabe zu starten. Geben Sie zusätzlich einen geeigneten Ereignis-Namen an, wenn Sie zum Beispiel das *Read(Data AS String)*-, das *Error(Data AS String)*- oder das *Kill()*-Ereignis abfangen und darauf reagieren wollen.

Die Frage, ob mit dem Erzeugen eines Task-Objekts notwendig auch ein Task als Hintergrund-Prozess gestartet wird, kann so beantwortet werden:

- Wird ein Task-Objekt erzeugt, so wird sofort der Start eines assoziierten Hintergrund-Prozesses *geplant*. Der Task wird allerdings erst während der nächsten Ausführung der Event-Schleife tatsächlich gestartet.
- Innerhalb des Tasks wird die Main()-Methode der Task-Klasse ausgeführt. Wenn der Funktionswert der Main()-Funktion zurückgegeben wurde, dann wird der Task automatisch beendet! Das Task-Objekt im Haupt-Prozess bleibt jedoch erhalten!

20.6.0.5 Task-Priorität

Sie können die Priorität eines Hintergrund-Prozesses über die Eigenschaft *Application.Priority* (Datentyp Integer) ändern → Kapitel 20.11 Klasse Application (gb). Die Standard-Priorität hat den Wert 0.

Um die Priorität eines Hintergrund-Prozesses zu erhöhen (Werte -1 bis -20) sind Root-Rechte erforderlich, die Sie für das Herabsetzen der Priorität (Werte 1 bis 19) nicht benötigen.

```
' Gambas class file

Inherits Task

Public Function Main() As Variant
  Dim aTagesListe As String[]

  Application.Priority = 10 ' Prozess-Priorität für den Task
  aTagesListe = Split("Sonntag,Montag,Dienstag,Mittwoch,Donnerstag,Freitag,Samstag", ",")
  Return aTagesListe[WeekDay(Now())]

End ' Function Main()
```

20.6.0.6 Interaktive Prozess-Kommunikation

Die Datenübertragung zwischen einem Task als Hintergrund-Prozess und dem übergeordneten (Eltern-) Prozess erfolgt in der Klasse Task (gb) nur uni-direktional – also in eine Richtung:

(A) Programm → Task

In dieser Richtung können Sie dem Task einmalig *Argumente als Startwerte* mitgeben. Dazu müssen Sie in der Task-Klasse – hier *MyTask.class* – öffentliche Variablen definieren, um diesen im Programm unmittelbar nach dem Erzeugen des Tasks geeignete (Start-)Werte zuzuweisen.

MyTask.class:

```
Public iWaitTime As Integer ' Start-Argument für die Wartezeit zwischen den drei Anzeigen
```

FMain.class:

```
Private hTask As MyTask

Public Sub btnTaskStart_Click()
  If hTask = NULL then hTask = New MyTask As "MyTask" ' = Task-Klassen-Name
  hTask.iWaitTime = 3 ' Wertzuweisung für globale Variable in der Klasse MyTask
  …

End ' btnTaskStart_Click()
```

Einen anderen Ansatz für die Übergabe von Argumenten sehen Sie hier:

MyTask.class:

```
Public Sub _new({Matrix} As Matrix, Row As Integer)
  $hMatrix = {Matrix}
  $iRow = Row
End ' _new(..)
```

FMain.class:

```
Public Sub Main()
  Dim iRow, iCol As Integer
  Dim hDeterminante As TaskMinors

  ...
  For iCol = 0 To $hMatrix.Width - 1
    hDeterminante = New TaskMinors($hMatrix, iRow) As "TaskMinors"
    hDeterminante.Tag = iRow
    Inc $iTasks ' Zähler für die gestarteten Tasks erhöhen
  Next

  ...
End ' Main()
```

(B) Task → Programm

Die Art der Datenübertragung wird dadurch bestimmt, ob nur einmalig ein (Funktions-)Wert aus dem Task zurückgegeben wird oder ob *permanent* Daten vom Task an den Haupt-Prozess übermittelt werden (getaktet, zufällig). Benötigen Sie Daten aus dem laufenden Task, so müssen Sie diese über den PRINT- oder ERROR-Befehl in der Main()-Methode an die Standard-Ausgaben geschickten Daten im Event *Task_Read(Data As String)* oder im Event *Task_Error(Data As String)* auslesen. Beachten Sie: Im zweiten Fall sind nur Daten vom Daten-Typ *String* zugelassen.

In den vorgestellten Projekten finden Sie die unterschiedlichen Umsetzungen der Inter-Prozess-Kommunikation (inter-process communication, IPC).

Für eine permanente Datenübertragung Task → Programm von Daten mit nativem Daten-Typ hat Tobias Boege Serialisierungs- und Deserialisierungsfunktionen entwickelt, die in einem speziellen Projekt vorgestellt werden.

20.6.0.7 Projekt 1

Die Aufgabe, die im Projekt 1 zu bearbeiten ist, klingt wenig spektakulär: Mit dem Programmstart soll eine analoge Uhr angezeigt werden und parallel dazu wird über einen Task aus dem aktuellen Datum der Wochentag berechnet und angezeigt. Da Sie einmalig den Funktionswert der Main()-Funktion vom Task – der sein einziger Lebenszweck ist – benötigen, können Sie die Eigenschaft *Task.Value* auslesen, nachdem der Hintergrund-Prozess beendet wurde.

Der Quelltext für die Task-Klasse *DayTask.class* wird vollständig angegeben, während von der Klasse *FMain.class* nur relevante Abschnitte vorgestellt werden:

```
' Gambas class file

Inherits Task

Public Function Main() As Variant
  Dim aTagesListe As String[]

  aTagesListe = Split("Sonntag,Montag,Dienstag,Mittwoch,Donnerstag,Freitag,Samstag", ",")
  Return aTagesListe[WeekDay(Now())] ' Rückgabe Wochentag

End ' Function Main()
```

In dieser Klasse wird aus dem aktuellen Datum der Wochentag berechnet und als Funktionswert von der Main-Methode zurückgegeben.

Im Programm wird ein Task in der Prozedur TaskRun() erzeugt:

```
' Gambas class file

Private $hTask As DayTask

Public Sub Form_Open()
  ...
' Task erzeugen: Task-Objekt und Task-Prozess
  TaskRun()

End ' Form_Open()

Private Sub TaskRun()

' Ein neues Task-Objekt erzeugen - Objekt-Name = Objekt-Event-Name: DayTask
  If $hTask = Null Then $hTask = New DayTask As "DayTask"
  Repeat
    Wait 0.001
  Until $hTask <> Null

End ' TaskRun()

Public Sub DayTask_Kill()
  Dim DayOfWeek As String
  Dim sErrorMessage As String

' Prozess-Rückgabewert sichern. Alternative: Last.Value
```

```
  Try DayOfWeek = $hTask.Value
  If Not Error Then
     lblDayOfWeek.Text = DayOfWeek
  Else
     sErrorMessage = "Fehler!" & gb.NewLine
     sErrorMessage &= Error.Where & gb.NewLine
     sErrorMessage &= Error.Text
     Message.Error(sErrorMessage)
     lblDayOfWeek.Text = "Task-Fehler"
  Endif ' ERROR ?

End ' DayTask_Kill()
...
```

Es wird die analoge Uhr angezeigt und darunter der im Task zum aktuellen Datum einmalig berechnete Wochentag – sofern kein Fehler im Task auftrat:

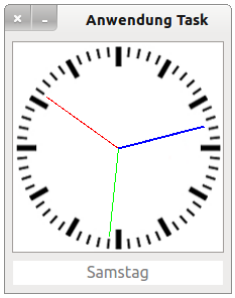

Abbildung 20.6.0.7.1: Anzeige von Uhrzeit und Wochentag

Einen Fehler können Sie recht einfach erzeugen, indem Sie statt des korrekten Separators *Komma* zum Beispiel ein *Semikolon* verwenden:

```
  aTagesListe = Split("Sonntag,Montag,Dienstag,Mittwoch,Donnerstag,Freitag,Samstag", ";")
```

Die Anzeige ändert sich im unteren Teil, weil in der Prozedur 'Public Sub DayTask_Kill()' der Fehler erkannt und dokumentiert wird:

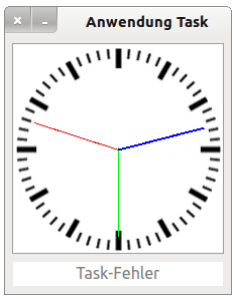

Abbildung 20.6.0.7.2: Anzeige von Uhrzeit und Fehler-Meldung

Sie können im letzten Fall noch einmal sehr deutlich erkennen, dass mit dem Erzeugen des Task nach der Gabelung in zwei unabhängige Prozesse die Uhr völlig unabhängig vom Task und seinem Rückgabewert läuft.