opentext™

# Team Developer 7.1 Multithreading

Parallele Prozesse in TD basierenden Anwendungen

Roadshow April, 2018 | Helmut Reimann

# Agenda


- Was ist Multithreading?
- Team Developer 7.1 Implementation
- Debugging
- Beispiel

# Was sind Threads?


- Threads sind ein Mechanismus um Programmecode – Sequenzen parallel innerhalb einer Anwendung auszuführen
- User Interface (UI) vs. Worker Threads
  - UI Bibliotheken wie MFC und WPF beinhalten definierte Threads um Anwender Interaktionen (Events) auszuführen
  - Diese Threads nennt man UI Threads, sie sind innerhalb einer Anwendung verantwortlich für die Interaktion zwischen Anwender und Windows-Subsystem
    - Wenn dieser Thread blockiert ist, dann ‘steht’ die Anwendung (meist) temporär (“Keine Rückmeldung…”)
  - Worker Threads erlauben die Programmierung von lang laufenden Code Sequenzen im Hintergrund, ohne den aktuellen UI Thread zu blockieren
  - Worker Threads haben keine Verbindung zu UI Elementen, daher müssen andere Mechanismen gefunden werden, um mit der Anwendungsoberfläche zu kommunizieren.
- Dispatcher
  - Funktionieren unterschiedlich in Win32 und WPF, grundsätzlich bieten ein Dispatcher die Möglichkeit eine Kommunikation zwischen Hintergrund (Worker) Prozess und UI zu programmieren.

# Implementierung von Threading in TD


- Das Arbeiten mit Threads kann sehr kompliziert sein!
- Oftmals bieten ‘low level libraries’ eine sehr ausgeprägte Funktionalität und können somit fast alle Anforderungen abdecken, aber sind sehr schwierig mit dem Anwender Interface (Anwendungsoberfläche) zu verknüpfen.
- Ziel für TD:
  - Einfache Implementierung und einfache Handhabung; Evtl. nicht den vollen Funktionsumfang….
  - Nutzen eines Event-Modells wie in WPF um die Kommunikation zwischen Hintergrund Prozess und UI bereitzustellen.
  - Einfache Übergabe von Informationen vom Hintergrund Prozess an den UI für Fortschritt, Prozess Ende, Fehler Handling usw.

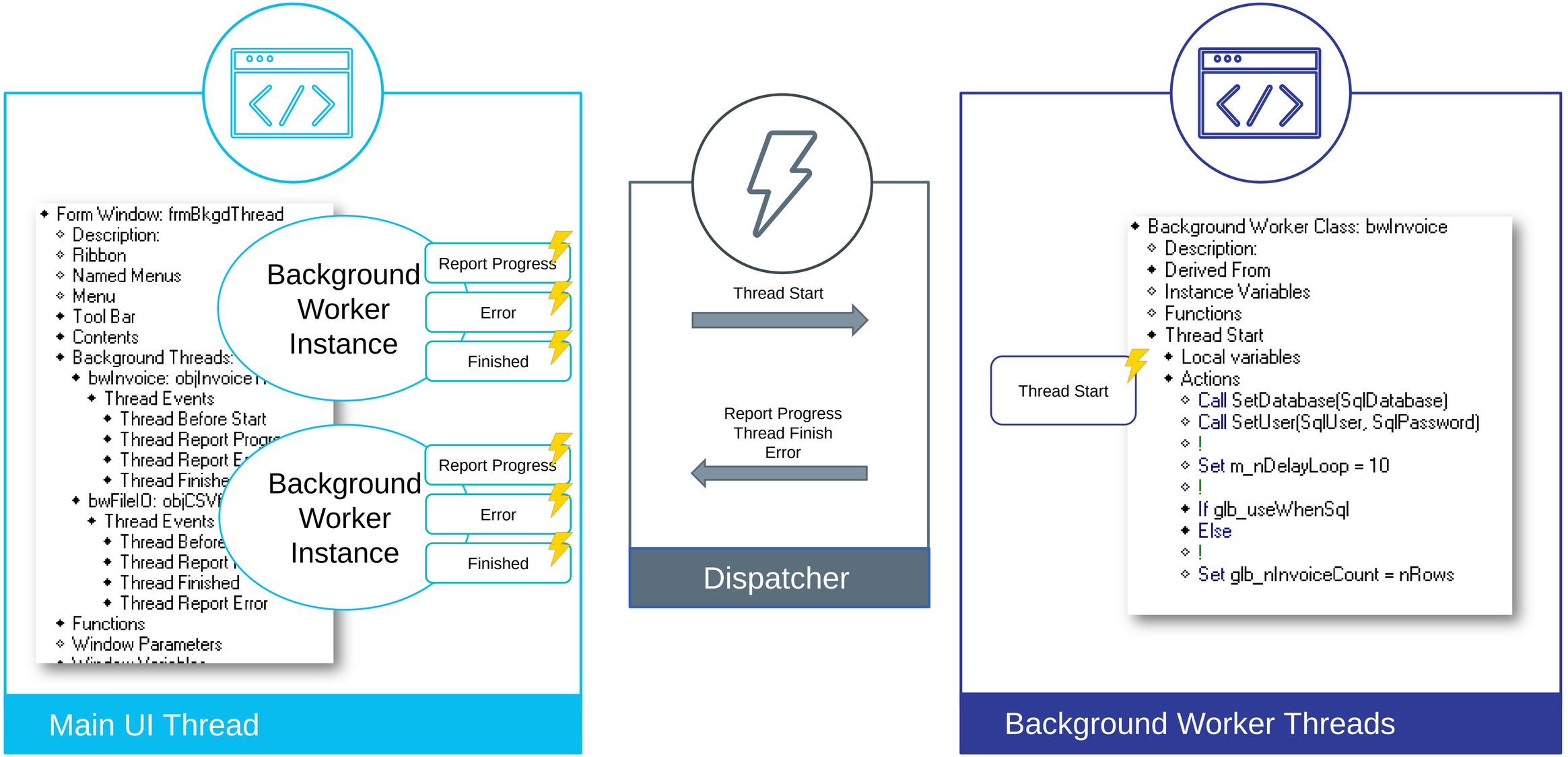
# Vorteile und Anforderung TD Multithreading

- Implementiert für Win32/Win64 und .NET
- Anwendung arbeitet weiter, auch wenn aufwendige Prozesse im Hintergrund parallel laufen.
- Anwender kann trotz laufenden Hintergrund Prozessen weiter mit der Anwendung arbeiten auch bei
  - Komplexen Datenbank Operationen
  - Komplizierte Datei Transaktionen (z.B. XML-Schnittstellen)
  - Verarbeiten von Mails
- Einfache TD OOP Implementierung
- Einfach zu debuggen

# Neue Background Worker Klasse


- Background Worker Class ist ein neuer Klassen-Typ
- Diese Klassen können Functional Classes erben
- Diese Klassen besitzen eine “Thread Start“ Outline Sektion:
  - Dieser Code wird dann im parallelen Prozess ausgeführt.
- Es wird eine Instanz der Background Worker Class im Window in der neuen Thread Section eingefügt
  - Diese Instanz beinhaltet eine Event Sektion, um eine Kommunikation zwischen Hintergrundprozess und Oberfläche zu ermöglichen.
  - Innerhalb diesen Events, kann z.B. eine Progress Bar angesprochen werden.
  - Für die Kommunikation wurden neue SAL-API Funktionen implementiert.

# Architektur

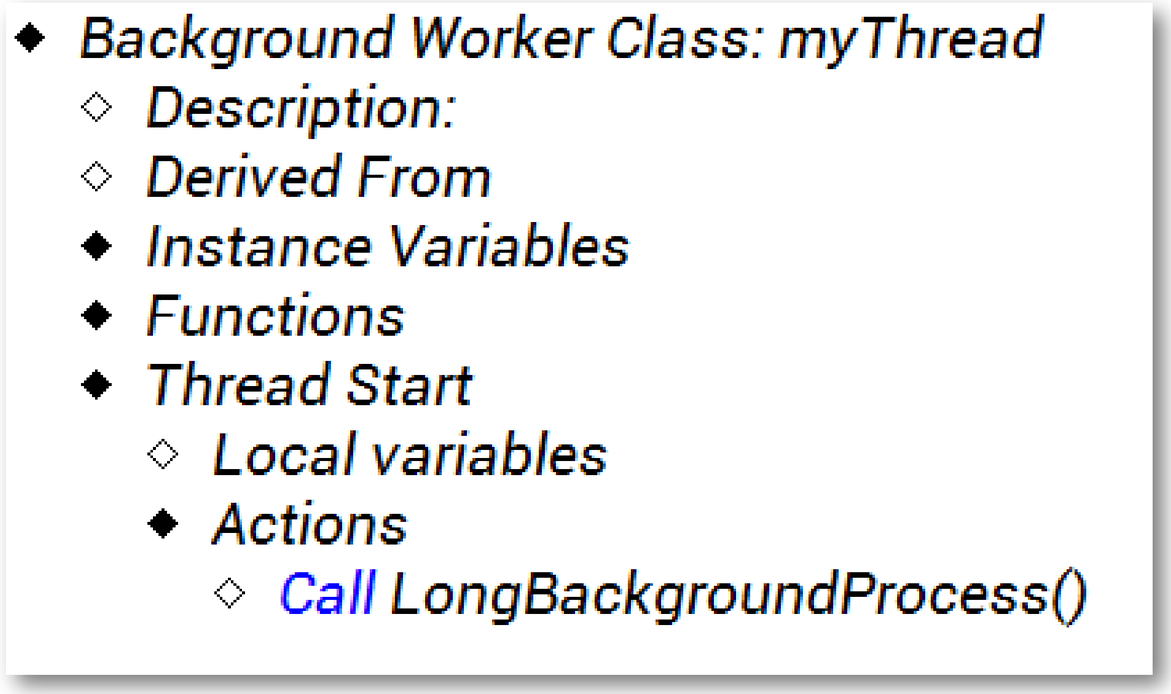


# Implementieren von Hintergrund Prozessen


- Erstellen einer Background Worker Class
- Einfügen der Background Thread Instanz einer Background Worker Class in das entsprechende Fenster
- Innerhalb des Window Objekts
  - Start des Background Thread mit ***SalBackgroundWorkerStart(oBkgdWorker)***
  - Der Thread wird gestartet und erzeugt Events in der Thread Instanz

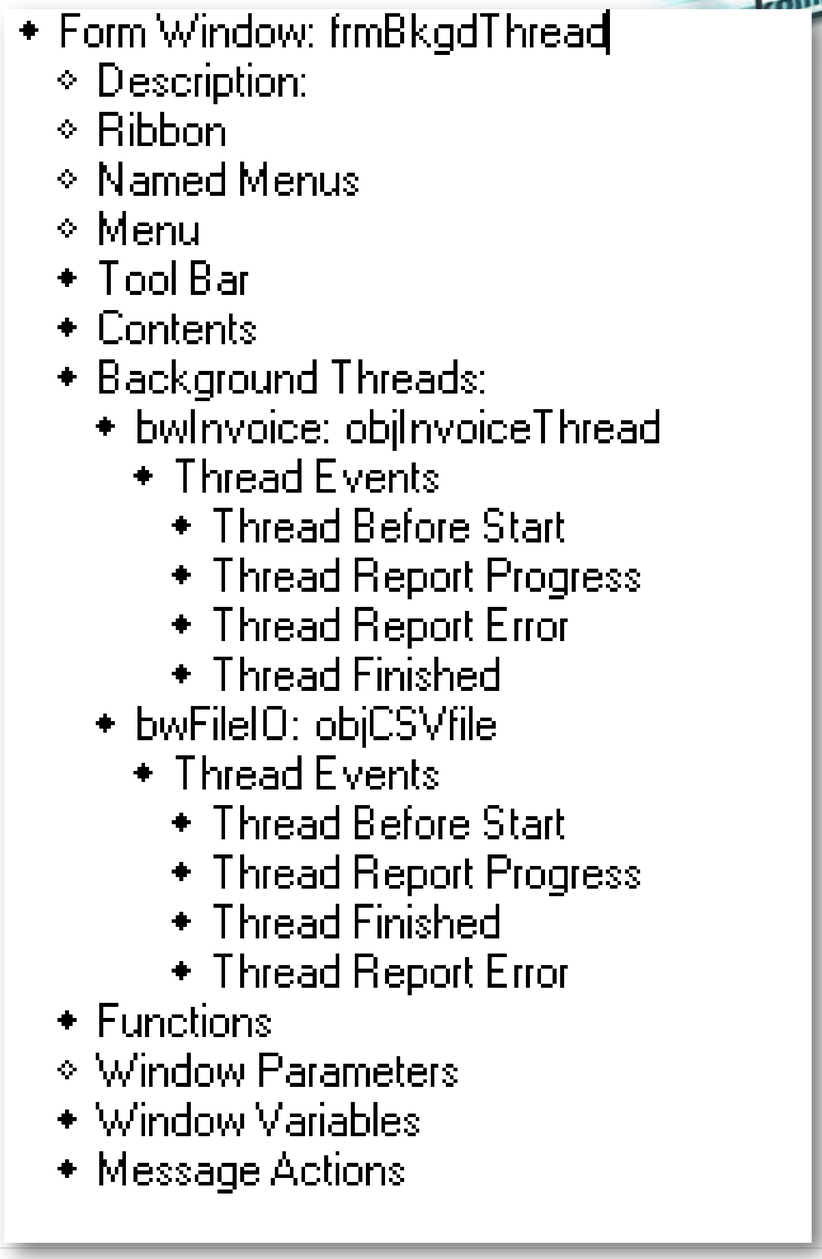
# Background Worker Klasse

- Thread Class Definition
  - Instanz Variablen zum Initialisieren der Klasse
    - Übergabe von Werten aus der Anwendung
  - Thread Code (Actions –Sektion)
    - Lang laufende Datenbank Prozesse
    - Datei Verarbeitung
    - Einlesen von Daten aus Schnittstellen
    - Etc.
  - Keine UI Interaktion!



# Hintergrund Prozesse

- Hinzufügen einer Instanz der Background Workers Class zu einem Form/Dialog Fenster
- Die Thread Instanzen beinhalten die Thread Events
  - Diese Events geben dem Entwickler die Möglichkeit ein Feedback vom Prozess an die Form/Dialog zu senden um z.B. die Progress Bar / Statuszeile upzudaten
- ***SalBackGroundWorkerStart(ThreadInstance)*** startet den Background Prozess (Thread)

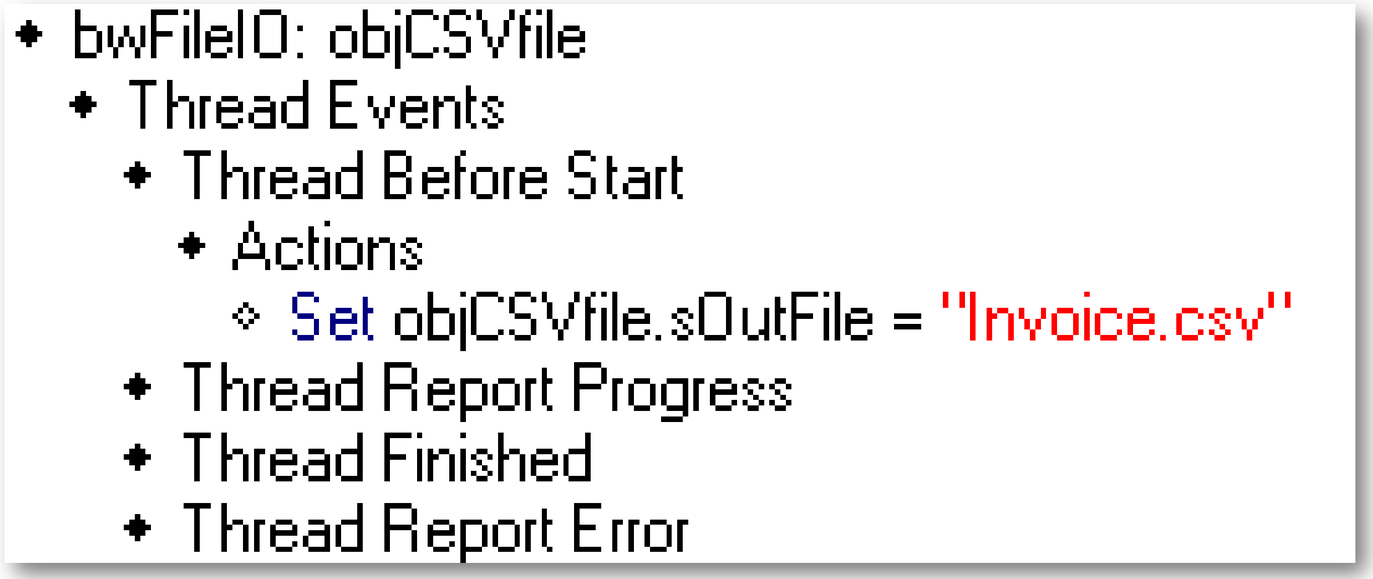


# Thread Events

- ***Thread Before Start***
  - Der Code innerhalb des Thread startet erst, wenn der Event ‘Thread Before Start’ abgearbeitet ist. Setzen der Instanzvariablen, Übergabe von Variablen, Initialisierung.
  - **Synchroner Event**
- ***Thread Report Progress***
  - Gesendet mit der API Funktion ***SalBackgroundWorkerReportProgress(nProgress, sMessage)***
  - Sendet *nProgress* für einen prozentualen Fortschritt aus dem Background Prozess als Wert und den Message String.
  - **Asynchroner Event**, der Hintergrund Prozess läuft weiter. Der Event Code läuft nur dann, wenn die GUI ohne Interaktion mit dem Anwender ist.
- ***Thread Finished***
  - Wird ausgelöst, wenn der Hintergrund Prozess beendet ist. Z.B.: Update der GUI mit Anzeige der Werte.
  - **Asynchroner Event**
- ***Thread Error***
  - Wird ausgelöst, wenn ein unbehandelter Fehler im Hintergrund Prozess ausgelöst wird.  Es ist sehr wichtig, dass keine Messagebox andere GUI Interaktion ausgelöst wird. Der Fehler sollte z.B. im Thread Error Event behandelt werden!
  - **Asynchroner Event**

# Thread Before Start

- Initialisierung bevor der Thread Code startet
  - Setzen von Instanz Variablen usw.
- Thread startet nach der Ausführung des Codes

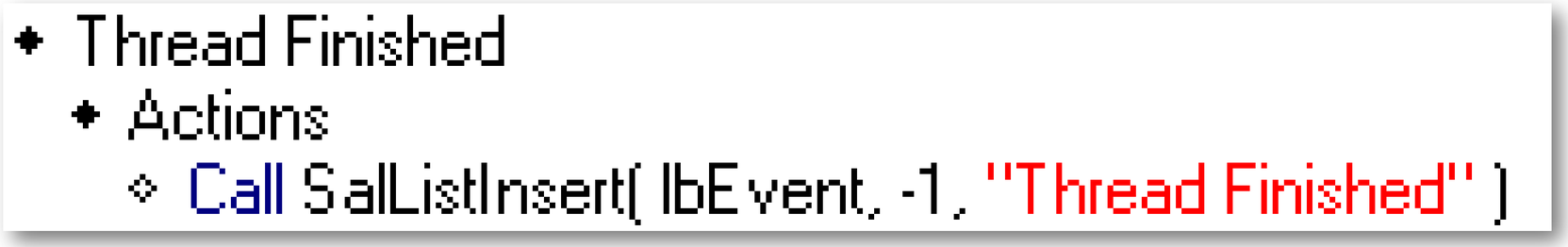


# Thread Report Progress

- Ermitteln des Fortschritts des Hintergrund Prozesses
- Ein numerischer Wert und ein String werden vom Hintergrund Prozess zurückgegeben
- Update der Progress Bar und/oder Anzeige einer Meldung

# Thread Finished

- Der Hintergrund Prozess wurde beendet:
  - Enable Buttons, Ribbon Bar Änderungen, Anzeige der ermittelten Daten usw.
- Wenn ein unbehandelter Fehler auftritt und der *Thread Error* Event wurde ausgelöst, dann wird *Thread Finished* nicht ausgelöst!



# Thread Error

- Wenn zur Laufzeit eines Hintergrund Prozesses ein unbehandelter Fehler auftritt, dann wird *Thread Error* vom Prozess gesendet
  - *nError* – Ein optionaler Fehler Code, z.B. wenn ein SQL Error ausgelöst wird
  - *strError* – Fehler Meldung als Text
- Wenn das Event ausgelöst wird, so darf keine Message Box ausgegeben werden!
- Wenn Thread Error ausgelöst wird, dann wird kein *Thread Finished* ausgelöst!

```
Thread Report Error
  Event Parameters
    Number: nError
    String: strError
  Actions
    Set mlError = mlError || "SQL Error: " || SalNumberToStrX(nError, 0)
    Set mlError = mlError || "
      " || strError || "
      "
```

# Neue Threading API


- Neue API für die Ausführung von Hintergrund Prozessen:
  - ***SalBackgroundWorkerStart(oBkgdWorker)*** (Main)
  - ***SalBackgroundWorkerReportProgress(nProgress, sMessage)*** (Thread)
  - ***SalBackgroundWorkerIsBusy(objBkgdWorker)*** (Main)
  - ***SalBackgroundWorkerCancel(objBkgdWorker)*** (Main)
  - ***SalBackgroundWorkerIsCanceled( )*** (Thread)
  - ***SalBackgroundWorkerAnyRunning(hWnd)*** (Main)

# SalBackgroundWorkerStart()

- **SalBackgroundWorkerStart(oBkgdWorker)**
- Startet eine Background Worker Class. Es wird  der *On Thread Start*  Event ausgelöst, dann wird der Background Thread gestartet und der Code in *Thread Start*  wird durchlaufen (Code in Background Worker Class)
- *oBkgdWorker* - Eine Background Worker Instanz die gestartet werden soll
- *Returns Boolan* - FALSE wenn der Prozess bereits läuft

# SalBackgroundWorkerReportProgress()

- **SalBackgroundWorkerReportProgress(nProgress, sMessage)**
- Löst den Event Thread Report Progress in der GUI aus. Dieser wird aus dem Thread Code gesendet
  - Hinweis: Wird nur angezeigt, wenn die GUI keine Anwender Interaktion durchführt („Idle“)
- *nProgress* – Fortschrittswert
- *sMessage* – Meldung zum Fortschritt
- *Returns Boolean* - FALSE wenn der Prozess nicht läuft

# SalBackgroundWorkerIsBusy()

- **SalBackgroundWorkerIsBusy(objBkgdWorker)**
- Abfragen, ob der Prozess noch läuft
- *oBkgdWorker* – Name der Background Worker Instanz
- *Returns Boolean* - TRUE wenn dieser Prozess noch läuft

# SalBackgroundWorkerCancel()

- **SalBackgroundWorkerCancel(objBkgdWorker)**
- Abbrechen eines Hintergrund Prozesses
  - Hinweis: Es wird nicht ein sofortiges beenden den Prozesses erwirkt, es wird ein Flag gesetzt, das der Hintergrundprozess regelmäßig abfragen sollte. Wenn das Flag gesetzt ist, dann sollte die Anwendung den Hintergrundprozess beenden
- *oBkgdWorker* - Ein Background Worker Instanz
- *Returns Boolean* - FALSE wenn der Prozess abgebrochen wurde

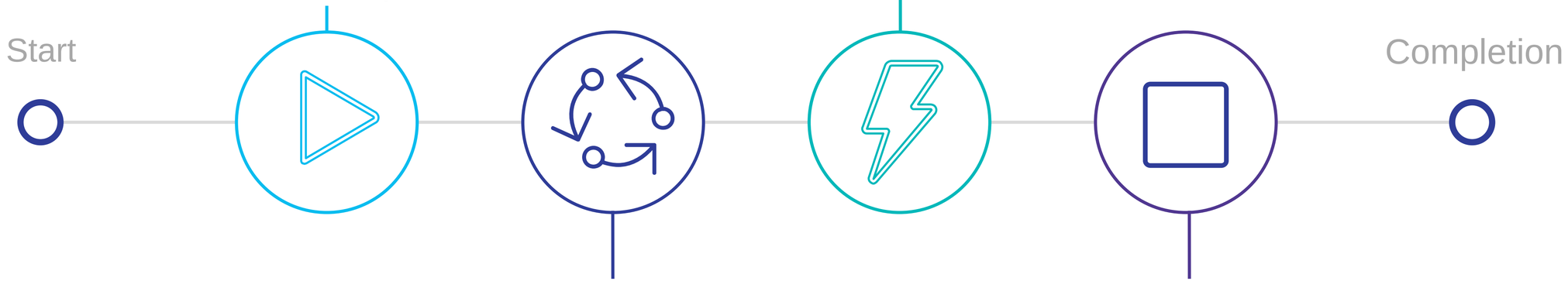
# SalBackgroundWorkerIsCanceled()

- **SalBackgroundWorkerIsCanceled()**
- Überprüfen, ob *SalBackgroundWorkerCancel* aufgerufen. Wenn ja: der Anwender muss den Prozess beenden.
  - Beispiel: dieser Check wird bei jeder Schleife im Datenbankzugriff ausgeführt. Wenn *True* zurückgegeben wird: Prozess ‚sauber' beenden und Rückkehr zur Hauptanwendung.
- *Returns Boolean* - TRUE wenn *SalBackgroundWorkerCancel* vom Hintergrundprozess ausgelöst wurde

# SalBackgroundWorkerAnyRunning()

- **SalBackgroundWorkerAnyRunning(hWnd)**
- Überprüft, ob ein Hintergrundprozess für eine Form / Dialog gestartet wurde. Gibt TRUE oder FALSE zurück
- Anwender kann gewarnt werden, wenn ein Fenster mit laufenden Hintergrundprozessen geschlossen wird
- Wenn *hWndNULL* als Parameter übergeben wird, dann wird diese Funktion die gesamte Anwendung auf laufende Prozesse überprüfen
  - Sinnvoll für CleanUp beim On App Exit

# Thread Lifecycle




## Initialization

TD spawns a new Worker Thread.  Calls Thread Start event on form synchronously to give you ability to initialize properties for that run (file out name etc.)

## Notify

During execution, SalBackgroundWorkerReportProgress can be used to send information to UI thread on progress

## Execution

TD calls Thread Start on the worker class, executes SAL code

## Error/Finished

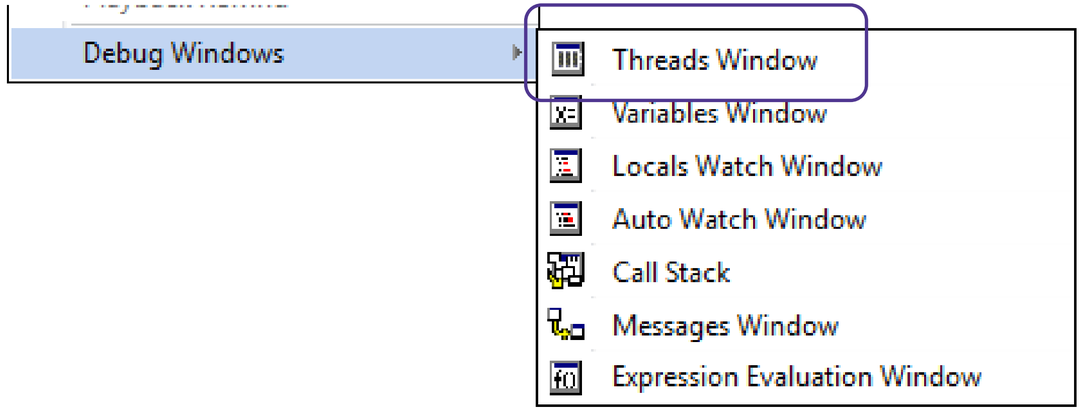
When SAL code returns from Thread Start, Finished is raised. Or if error is thrown, Error is raised

# Thread Debugging

## Debuggen von Hintergrund Prozessen

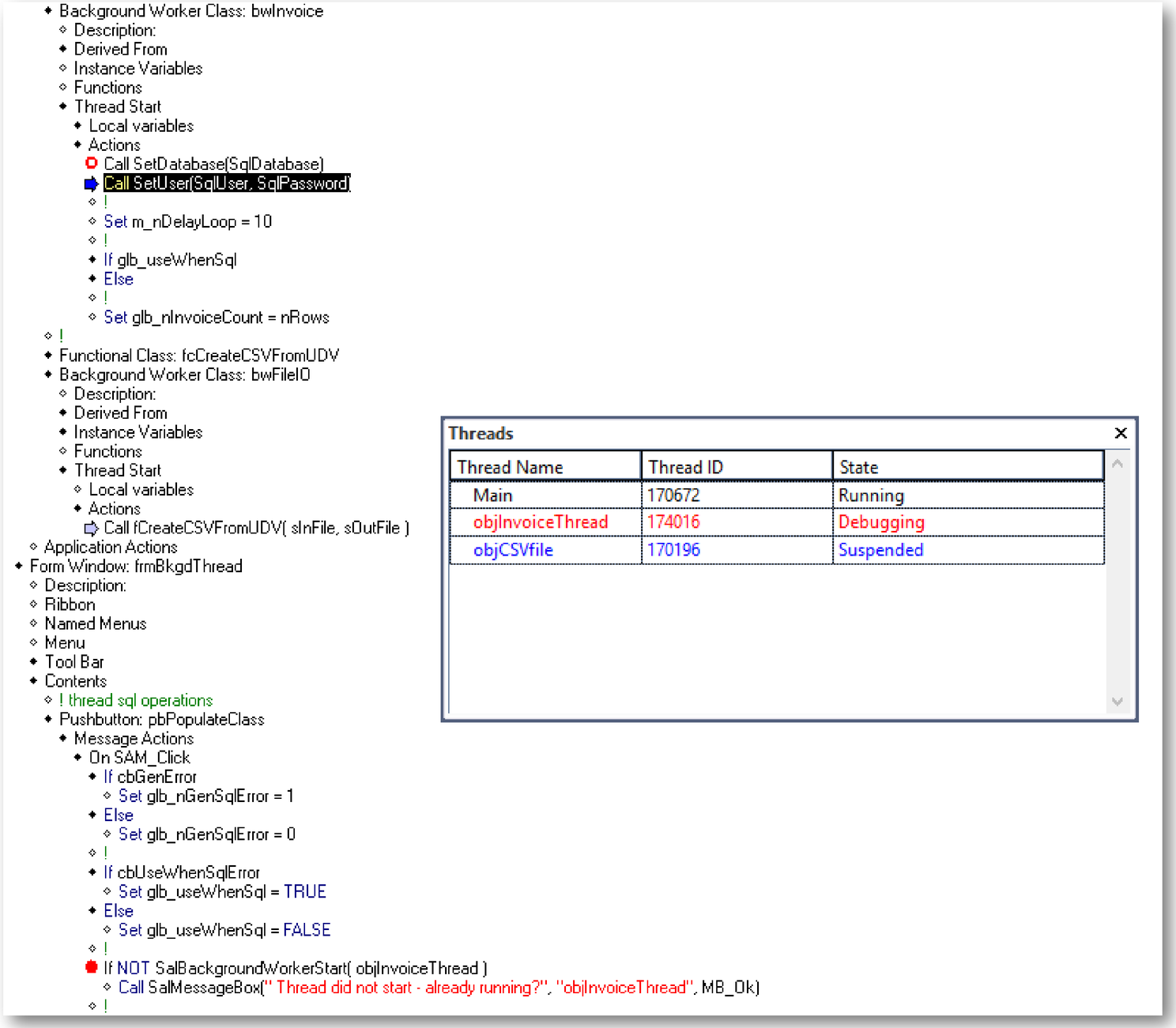
# Thread Debugging

- TD 7.1 beinhaltet neue Möglichkeiten zum debuggen von Hintergrund Prozessen
- Neues Debug Fenster “Threads”



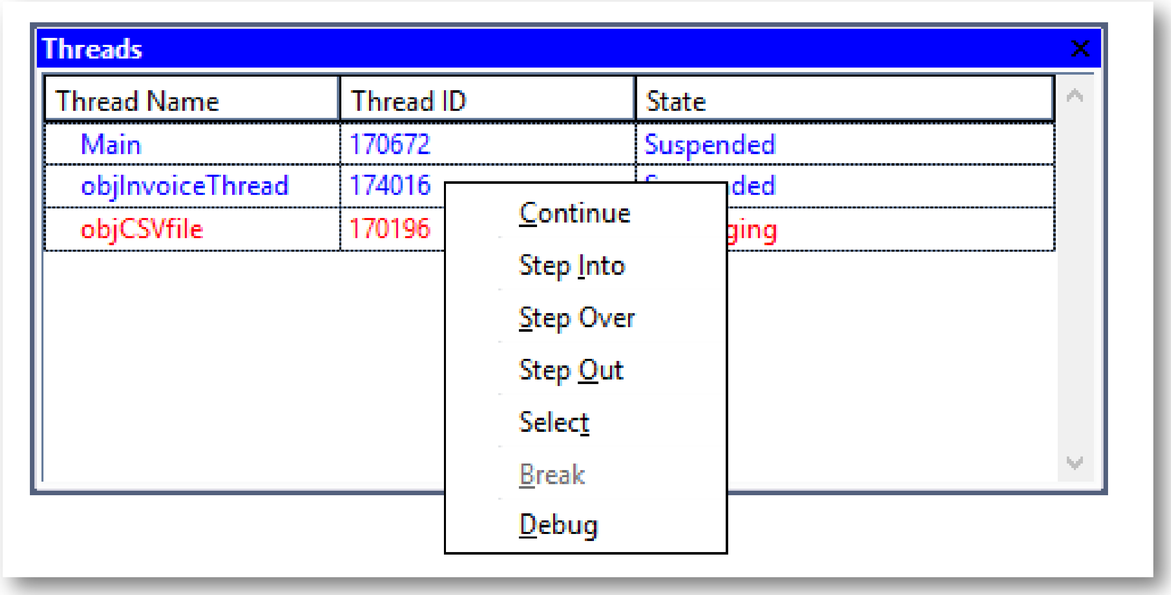
# Threads Window

- Erlaubt die Möglichkeit zwischen einzelnen Prozessen zu wählen
- Übersicht über Prozess Status:
  - *Running* – Thread führt Code aus
  - *Suspended* – Steht am Breakpoint, ist aber nicht aktiv(z.B. ein anderer Thread wird gedebugged)
  - *Debugging* – Der aktive Thread wird gedebugged
  - Hinweis: Unterschiedliche Darstellung der aktiven Zeilen (Mehrere Zeilen können aktiv sein, dunkle Farbe zeigt aktive Position und Thread an)



# Threads Window

- *Right-click* öffnet ein Kontext Menü um folgende Aktionen auszuführen:
- *Continue/Step:* Debuggen des ausgewählten Threads
- *Select:* Zeigt die aktuelle Code Zeile und der der Debugger wird gestoppt. Prozess wird angehalten.
- *Break:* Wenn der Thread im laufenden Modus ist: Debugger an aktueller Zeile anhalten
- *Debug:* Macht aus einem Suspended Thread einen aktiven Thread, zeigt entsprechend die Debug Informationen des entsprechenden Threads



# Demo

## Threading & TD Demo

opentext™

# Danke!

twitter.com/ot_gupta

facebook.com/opentextgupta

linkedin.com/company/opentextgupta

**opentext.com/gupta**