【2C】Delphi/C++テクニカルセッション

「正しいGUIの作り方」
- 重い処理でGUIが固まるのを防ぐ -

エンバカデロ・テクノロジーズ　エヴァンジェリスト
高橋　智宏

EMBARCADERO
TECHNOLOGIES.

GUIが固まる

アジェンダ

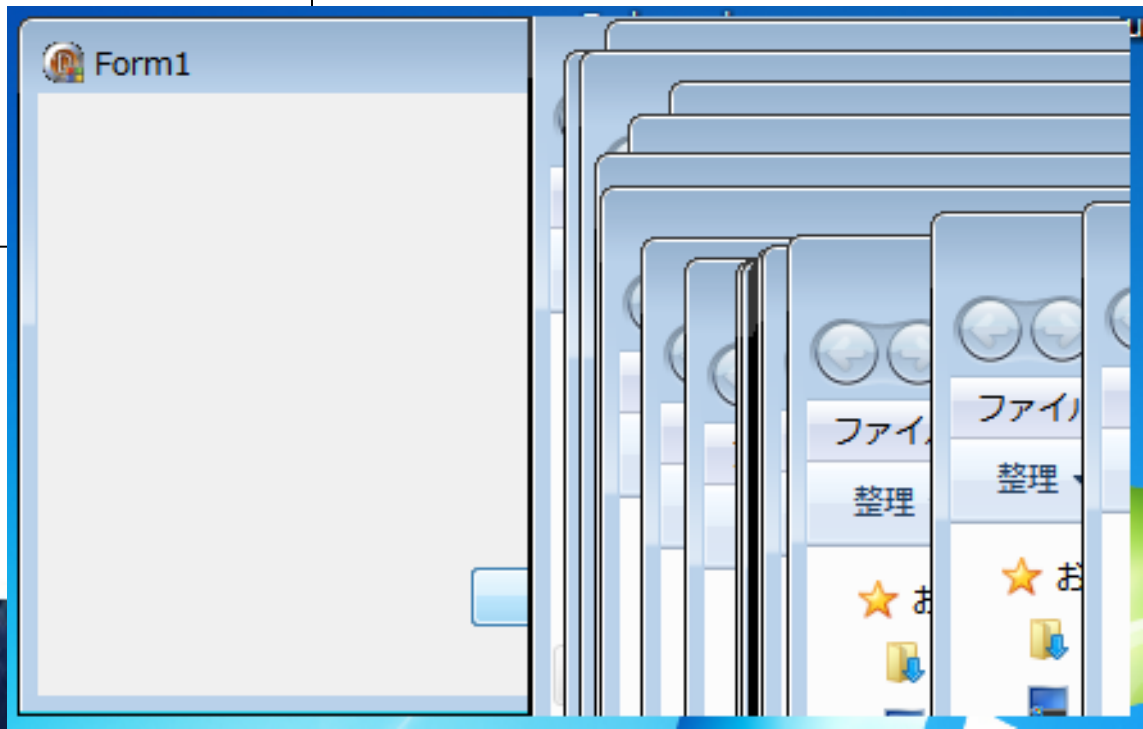
- Delphi言語で説明しますが、C++Builderでも全く同じです

- いちばん良くないパターン
  - ボタンを押すとGUIが長時間固まったまま
- シングルスレッドのままで頑張るパターン
  - プログレスバーなどで状況を知らせる
  - [キャンセル]ボタンを追加したい
  - でも、限界がありますよね?
- マルチスレッドに対応させるパターン
  - ワーカースレッド内からGUIを更新するパターン
  - GUIスレッドからワーカースレッドをポーリングするパターン

長時間固まったまま

```
procedure OneSecondMethod;
begin
  Sleep(1000);
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  i: Integer;
  org_cursor: TCursor;
begin
  org_cursor := Screen.Cursor;
  Screen.Cursor := crHourGlass;
  for i := 1 to 100 do
  begin
    OneSecondMethod();
  end;
  Screen.Cursor := org_cursor;
end;
```

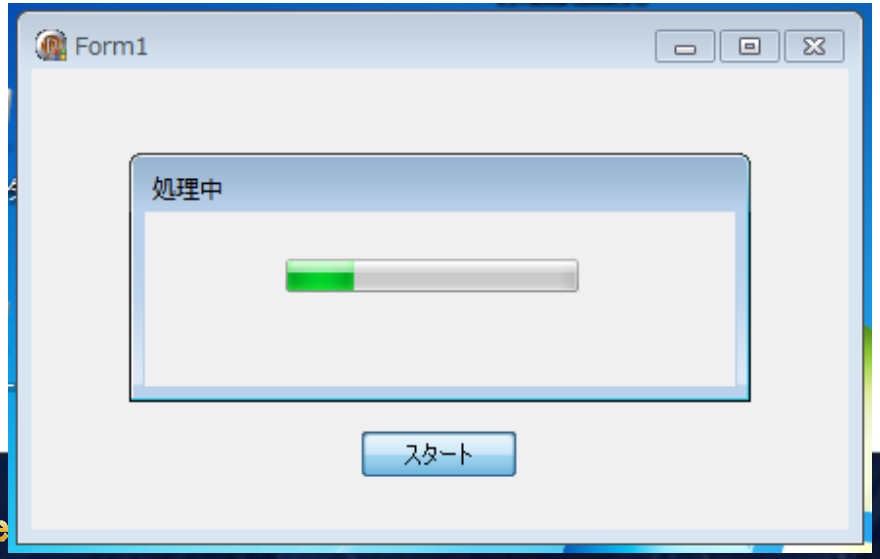
サーバーへの呼び出しだったら
かかる時間の予測は難しいです



状況を表示

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i: Integer;
begin
  ProgressForm.Show();
  for i := 1 to 100 do
  begin
    OneSecondMethod();
    ProgressForm.UpdateProgress(i);
  end;
  ProgressForm.Close;
end;
```

```
procedure TProgressForm.UpdateProgress(pos: Integer);
begin
  Label1.Caption    := IntToStr(pos) + '%';
  ProgressBar1.Position := pos;
end;
```

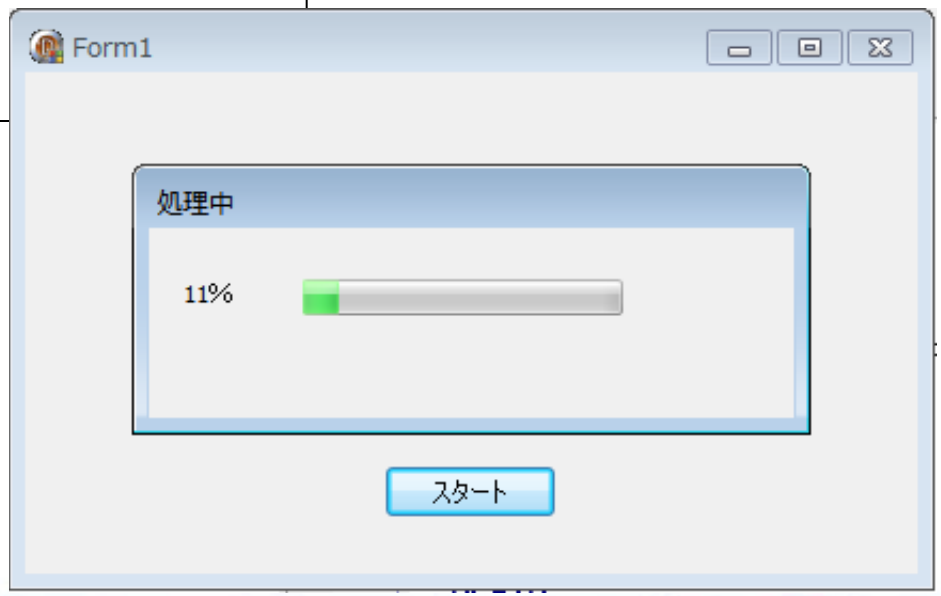


何かが変!?

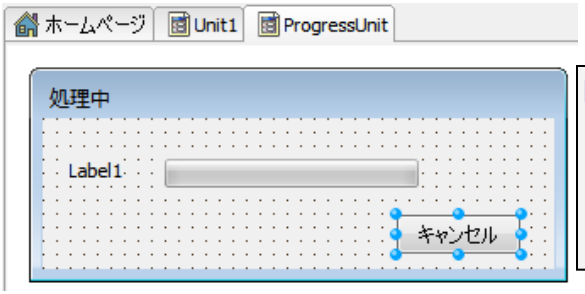
とりあえずの対応策

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i: Integer;
begin
  Self.Enabled := False;
  ProgressForm.Show();
  for i := 1 to 100 do
  begin
    OneSecondMethod();
    ProgressForm.UpdateProgress(i);
    Application.ProcessMessages();
  end;
  Self.Enabled := True;
  ProgressForm.Close;
end;
```

でも、まだ何かが変!?



さらに[キャンセル]ボタンを用意する




```
procedure TProgressForm.CancelButtonClick(Sender: TObject);
begin
  CancelButton.Enabled := False;
  Form1.CancelRequested := True; // TForm1を参照????
end;
```

```
type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: T
  public
    CancelRequested: Boolean;
  end;
```

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i: Integer;
begin
  CancelRequested := False;
  Enabled := False;
  ProgressForm.CancelButton.Enabled := True;
  ProgressForm.Show();
  for i := 1 to 100 do
  begin
    OneSecondMethod();
    ProgressForm.UpdateProgress(i);
    Application.ProcessMessages();
    if CancelRequested then
      Break;
  end;
  Enabled := True;
  ProgressForm.Close;
end;
```

キャンセルボタンの
反応が鈍い…

シングルスレッドでの限界

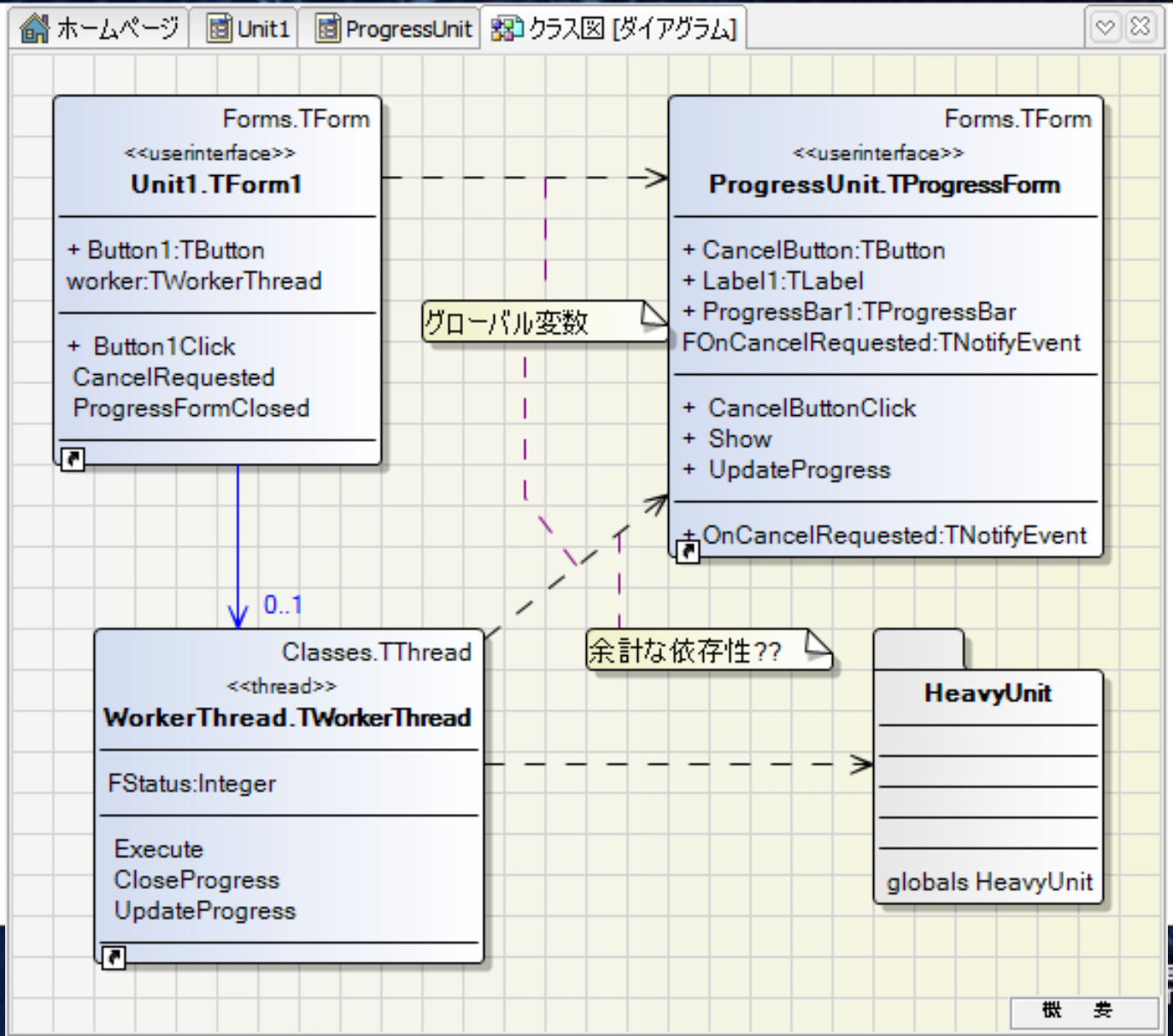
- Application.ProcessMessages(); の呼び出しに依存する
  - 外部のライブラリ利用時など、あるメソッドの呼び出しからリターンまでにかかる時間が予測できないとき
  - for/whileループが書けないビジネスロジックのとき
  - しかも、ウィンドウメッセージの処理順序に副作用が起こる可能性あり

ページ | Unit1 | HeavyUnit

```
uses Windows;
//uses Windows, Forms;

procedure HeavyBusinessMethod1; // 1秒のハズが10秒
begin
  Sleep(1000 * 10);
end;

procedure HeavyBusinessMethod2; // 1秒間のビジーループ
var
  s, n: Cardinal;
  i: Integer;
begin
  i := 0;
  s := GetTickCount();
  while True do
  begin
    Inc(i);
    n := GetTickCount();
    if (n - s) >= 1000 then
      Break;
    //対策???
    //Application.ProcessMessages();
  end;
end;
```

マルチスレッド

ワーカースレッド内からGUIを更新



ワーカースレッド内からGUIを更新(続き)

- TThreadクラスのSynchronizeメソッドを利用する
  - ワーカースレッド内からフォームを参照せざるを得ない

ムページ | Unit1 | ProgressUnit | WorkerThread | クラス図 [ダイアグラム

```
procedure TWorkerThread.UpdateProgress;
begin
  ProgressForm.UpdateProgress(FStatus);
end;

procedure TWorkerThread.CloseProgress;
begin
  ProgressForm.Close;
end;

procedure TWorkerThread.Execute;
var
  i: Integer;
begin
  Synchronize(UpdateProgress);
  for i := 1 to 100 do
  begin
    if Terminated then // キャンセル要求?
      Break;
    HeavyUnit.HeavyBusinessMethod1();
    FStatus := i;
    Synchronize(UpdateProgress);
  end;

  Synchronize(CloseProgress);
end;
```

ワーカースレッド内からGUIを更新(続き)

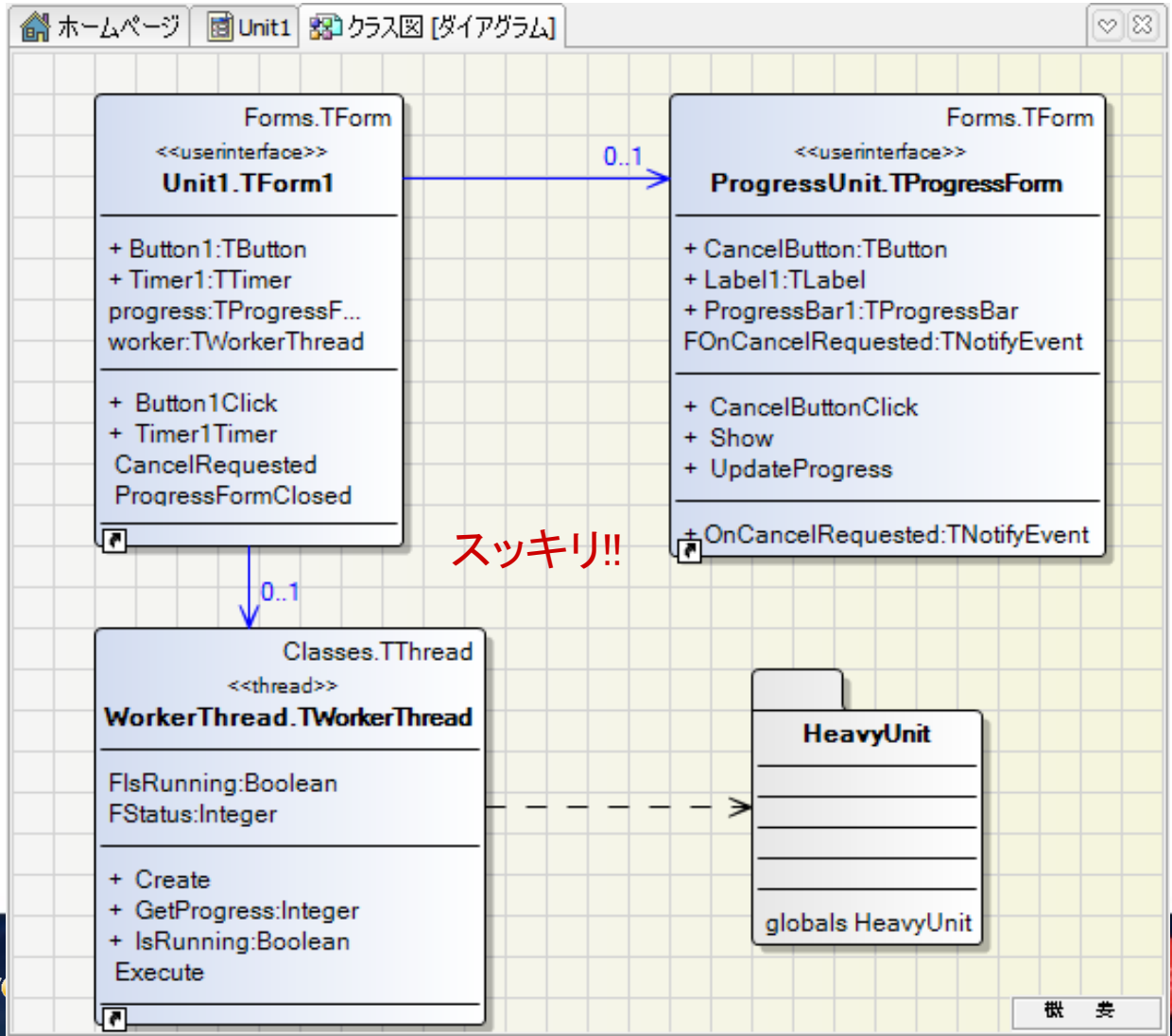
- ワーカースレッドのライフサイクル管理が結構難しい…
  - TThreadの終了待ち合わせ
  - TThreadインスタンスの明示的な破棄

  etc…

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Enabled := False;
  ProgressForm.OnCancelRequested := CancelRequested;
  ProgressForm.OnClose := ProgressFormClosed;
  ProgressForm.Show();

  worker := TWorkerThread.Create(True);
  worker.FreeOnTerminate := True; // 自動解放
  worker.Resume;
end;

procedure TForm1.CancelRequested(Sender: TObject);
begin
  worker.Terminate; // キャンセル要求だけ
end;

procedure TForm1.ProgressFormClosed(Sender: TObject; v
begin
  Enabled := True;
end;
```

GUIスレッドからワーカースレッドをポーリング

- TTimerで定期的にワーカースレッドに状況を問い合わせてGUIを更新



GUIスレッドからワーカースレッドをポーリング(続き)

- ワーカースレッドのクラスにステータスを追加
  - 例: スレッド自体の状態を示す IsRunning
  - 例: 処理の進捗状況を返す GetProgress

```
type
  TWorkerThread = class(TThread)
  private
    FStatus: Integer;
    FIsRunning: Boolean;
  protected
    procedure Execute; override;
  public
    constructor Create(CreateSuspended: Boolea
    function GetProgress: Integer;
    function IsRunning: Boolean;
  end;

implementation

uses HeavyUnit; // ProgressUnitは、もはや不要

constructor TWorkerThread.Create(CreateSuspend
begin
  inherited;
  FStatus := 0;
  FIsRunning := True;
end;
```

スッキリ!!

```
procedure TWorkerThread.Execute;
var
  i: Integer;
begin
  for i := 1 to 100 do
  begin
    if Terminated then // キャンセル要求?
      Break;
    HeavyUnit.HeavyBusinessMethod1();
    FStatus := i; // 厳密には排他制御が必要
  end;
  FIsRunning := False; // 厳密には排他制御が必要
end;

function TWorkerThread.GetProgress: Integer;
begin
  Result := FStatus; // 厳密には排他制御が必要
end;

function TWorkerThread.IsRunning: Boolean;
begin
  Result := FIsRunning; // 厳密には排他制御が必要
end;
```

GUIスレッドからワーカースレッドをポーリング(続き)

- GUIスレッドですべてを管理
  - ワーカースレッドのライフサイクル
  - 進捗状況・キャンセルボタンの処理や、サブフォームのライフサイクル

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Enabled := False;
  progress := TProgressForm.Create(nil);
  progress.OnCancelRequested := CancelRequested;
  progress.OnClose := ProgressFormClosed;
  progress.Show();

  worker := TWorkerThread.Create(True);
  worker.FreeOnTerminate := False; // 自動解放しない
  worker.Resume;
end;

procedure TForm1.CancelRequested(Sender: TObject);
begin
  worker.Terminate; // キャンセル要求だけ
end;
```

```
cedure TForm1.ProgressFormClosed(Sender: TObject; var A
in
nabled := True;
;

cedure TForm1.Timer1Timer(Sender: TObject);
in
f not Assigned(worker) or not Assigned(progress) then
 Exit;

if worker.IsRunning then
begin
  progress.UpdateProgress(worker.GetProgress());
end
else
begin
  worker.WaitFor;
  FreeAndNil(worker);
  progress.Close;
  FreeAndNil(progress);
end;
end;
```

まとめ

時間のかかる処理には

- 可能な限り、進捗状況をユーザーに知らせる
- 可能な限り、[キャンセル]ボタンなどで中止できるようにする
- 可能な限り、マルチスレッドを使おう
  - GUIとビジネスロジックの分離
  - GUIの更新は、GUIスレッドで
  - もちろん、スレッド間で共有される変数は、正しく排他制御してください

Q & A