

2007/09/27～28

Ruby/Tk講習会 at 都城

永井 秀利 (nagai@ai.kyutech.ac.jp)
九州工業大学情報工学部知能情報工学科

はじめに

- ◇ 今回の講習会で説明できるのはRuby/Tkの機能の内の極一部に過ぎません。
- ◇ この資料の他、2004/12/11に行ったRuby/Tk講習会の参考資料ファイルも置いてありますから、そちらも参照してください。
- ◇ Rubyのソースに添付されたRuby/Tkのサンプルも参考になるはずです。サンプルであると同時に、そのままライブラリとして使用できるファイル(クラス)もありますから、必要に応じて自由に利用してください。
- ◇ できればRuby/Tkのライブラリファイルも覗いてみてください。もしかすると新たな機能の発見があるかもしれません。
- ◇ Ruby/Tkについての質問があれば、お気軽にruby-listメーリングリストに流してください。

Rubyについて(1)

- ◇ オブジェクト指向スクリプト言語
- ◇ メソッド呼び出しは<クラス名>.<メソッド名>(...)
 - ◇ 曖昧でなければ、引数指定の(,)は省略可能
- ◇ オブジェクト生成は一般に<クラス名>.new(...)
 - ◇ コンストラクタはinitializeという名前のメソッド
- ◇ 説明時のメソッド記法
 - ◇ クラスメソッド ... <クラス名>.<メソッド名>
 - ◇ インスタンスメソッド ... <クラス名>#<メソッド名>

Rubyについて(2)

- ◇ Rubyの変数は「付箋紙」のようなものだと考えること
 - ◇ 変数への代入はコピーではなく、オブジェクトに対して付箋を張り付けるようなもの
 - ◇ 変数には型がないので、数値でも文字列でも、どのようなオブジェクトでも代入可能
- ◇ 変数のスコープ(有効範囲)は変数名で識別
 - ◇ ローカル変数 ... 英小文字または _ で始まる
 - ◇ 定数(クラス名も同じ) ... 英大文字で始まる
 - ◇ グローバル変数 ... \$で始まる
 - ◇ インスタンス変数 ... @で始まる
 - ◇ クラス変数 ... @@で始まる

Rubyについて (3)

- ◇ self ... 処理中に「自分自身」を指す変数(代入不可)
- ◇ クラスの定義 ... 親クラスは一つだけ

```
class クラス < 親クラス
  def self.クラスメソッド(...)
    ...
  end

  def インスタンスメソッド(...)
    ...
  end
end
```
- ◇ モジュール ... クラスに取り込めるメソッドの集まり

```
module モジュール
  # クラスと同様にメソッドを定義
end
```

Rubyについて (4)

- ◇ Mix-in ... モジュールの機能の取り込み

```
class クラス
  # 定義をインスタンスメソッドとして取り込み
  include モジュール

  # 定義をクラスメソッドとして取り込み
  extend モジュール
end
```
- ◇ クラス(定数)定義のパス

```
class CLASS1
  class CLASS2
    CONST_VAR = 0
  end
end
```

⇒ CLASS1::CLASS2とかCLASS1::CLASS2::CONST_VARとかでアクセス

Rubyについて (5)

- ◇ ブロック
 - ◇ 実行手続きのまとまり
 - ◇ ブロックが書かれた位置のスコープを保持

```
x = 1
object.method(...) { p x } # ブロックの外の x と同一
```
 - ◇ ブロック引数に値を渡して呼び出すことが可能
- ◇ 手続きオブジェクト (Proc)
 - ◇ ブロックと同様に、定義された位置のスコープを保持

```
class HOGE
  def self.hoge(proc_obj)
    proc_obj.call(2, 3)
  end
end

x = 1
HOGE.hoge(proc{|a, b| p a*x + b}) #=> 5 を出力
```

Rubyについて (6)

- ◇ Object#instance_eval
 - ◇ レシーバのオブジェクトがselfとなる環境で、与えられたブロックを評価
 - ◇ ローカル変数のスコープも保持していることに注意

```
x = 1
p @foo # この時点の self (≠ object) の
      # インスタンス変数 @foo

object.instance_eval {
  p @foo # object のインスタンス変数 @foo
  p x    # ブロックの外の x と同一
}

# do ~ end によるブロックの書式
object.instance_eval do
  p @foo # object のインスタンス変数 @foo
  p x    # ブロックの外の x と同一
end
```

Ruby/Tkについて

- ◇ Ruby/Tkとは
 - ◇ Tcl/Tkを、GUIライブラリとしてRubyに組み込んだもの
 - ◇ Tcl/Tkのスクリプトや拡張ライブラリのほぼすべてをそのまま動かすことができる
- ◇ Tcl/Tkとは
 - ◇ Tcl : 軽量なスクリプト言語(元々は組み込み用に開発)
 - ◇ Tk : Tcl用のGUIツールキット
 - ◇ シンプルで簡単にGUIを作成できることで人気
 - ◇ 基本的で汎用性が高く高機能の少数のウィジェット集合
 - ◇ 「足りなければ組み立てる」が基本思想
 - 差分プログラミングが適するが、標準のTcl/Tkはオブジェクト指向ではない



Rubyによるオブジェクト指向の殻でよりお手軽に!!

用語解説 (1)

- ◇ ウィジェット(widget)
 - ◇ Window Gadgetの略
 - ◇ GUIの部品で、なんらかの「機能」を持つ
 - 画像などは「機能」を特には持たないためウィジェットではない
- ◇ ウィジェットの親子関係
 - ◇ ルート(root)ウィジェットを根とした木構造を成す
 - ウィジェット階層, ウィジェットツリー
 - ◇ 木構造上の位置をファイルパスのように示したものがウィジェットパスで、ウィジェットの識別名になる
 - ◇ ウィジェットを破壊すれば、その子孫も破壊される
- ◇ ジオメトリマネージャ
 - ◇ ウィジェットの配置を司るもの
 - ◇ ウィジェットは、ジオメトリマネージャの管理下に置かれて初めて画面に表示される
 - ◇ 管理方法により複数種が存在

用語解説 (2)

- ◇ イベント
 - ◇ 画面(ウィンドウシステム)上の「出来事」すべて
 - クリック, マウスの移動, キーボードからの入力, フォーカス設定, ウィンドウのリサイズ等々
- ◇ バインド (バインディング)
 - ◇ 各種のイベントに対して、ウィジェットがどのように振る舞うかを規定すること, またはその規定のこと
- ◇ イベントループ
 - ◇ イベントの発生を待ち、それを処理するためのもの
 - ◇ アイドルタスク (再描画等, 緊急性のないもの) も処理
 - ◇ 通常は、ルートウィジェットが破壊されたときに終了

Hello, World!!

- ◇ 次の内容のファイル (hello0.rb) を作成

```
require 'tk'
TkLabel.new(:text=>'Hello, World!!').pack
Tk.mainloop
```

- ◇ コマンドプロンプトで `ruby hello0.rb` を実行

各行の説明

- ◇ `require 'tk'`
 - ◇ Ruby/Tkのライブラリの読み込み
 - ◇ `root`ウィジェットが自動的に生成される
 - ◇ 適切なTcl/Tkのライブラリが存在しない場合はエラー
- ◇ `TkLabel.new(:text=>'Hello, World!!').pack`
 - ◇ `root`ウィジェットの子供として`label`ウィジェットを生成 (詳しくは後述)
 - ◇ `text`属性の値を' Hello, World!!'に設定
 - ◇ `pack`ジオメトリマネージャを使って配置
- ◇ `Tk.mainloop`
 - ◇ イベントループの起動
 - ◇ `root`ウィジェットが存在する限り終了しない

labelウィジェット (TkLabelクラス)

- ◇ 文字列または画像(あるいはそれらの複合)を表示するためのウィジェット
- ◇ 設定可能な属性(オプション)の例
 - ◇ `text` :: 表示する文字列の指定
 - ◇ `font` :: 文字列表示に用いるフォントの指定
 - ◇ `bitmap` :: 表示するbitmapデータの指定
 - ◇ `image` :: 表示する画像データの指定
 - ◇ `compound` :: 文字列と画像との複合表示方法の指定

ウィジェット生成の基本形

- ◇ 基本形

```
widget_class.new(parent, arg, ..., options){ ... }
```

 - `parent` :: 親となるウィジェットオブジェクト (省略可. 省略時の親はルートウィジェット)
 - `arg, ...` :: ウィジェットクラス固有の引数 (必要とするのは極一部のウィジェットのみ)
 - `options` :: <属性名> => <属性値> の Hash
<属性名> はシンボルまたは文字列で与える
 - ブロック :: 生成したウィジェットオブジェクトにおいて `instance_eval`で評価
(1個のブロック引数を取ることもでき, その場合は生成したウィジェットオブジェクトが代入される)

生成+属性指定+packの例

- ◇ newの引数のみ

```
TkLabel.new(:text=>'Hello, World!!').pack
```
- ◇ ブロックの利用

```
TkLabel.new{
  text 'Hello, World!!'
  self.text('Hello, World!!')
}.pack
```
- ◇ ブロック引数の利用

```
TkLabel.new{|obj|
  obj.text 'Hello, World!!'
}.pack
```
- ◇ newの引数とブロックとの併用

```
TkLabel.new(:text=>'Hello, World!!'){ pack }
```

引数での指定とブロック内での操作

- ◇ 引数での指定を適用してウィジェットを生成した後にブロックを評価
- ◇ newメソッドの引数でしか指定できない属性も存在
 - ◇ 一部にウィジェット生成時に固定される属性が存在
 - ◇ 引数での指定は一般にウィジェット生成と同時に適用されるのに対し、ブロック内での操作は既に存在するウィジェットの属性変更になるため

ウィジェット属性の指定

- ◇ ウィジェットオブジェクトwidgetへの属性指定
 - ◇ widget.configure(属性=>val, ...) : 戻り値はwidget
 - ◇ widget.configure(属性, val) : 戻り値はwidget
 - ◇ widget.属性名メソッド(val) : 戻り値はwidget
 - ◇ widget.属性名メソッド = val : 戻り値はval
 - ◇ widget[属性] = val : 戻り値はval

※ 属性はシンボルまたは文字列で与える

(例) widgetのtext属性の値を 'foo' にしたい場合
widget.configure(:text=>'foo')
または widget.configure('text'=>'foo')
widget.configure(:text, 'foo')
または widget.configure('text', 'foo')
widget.text('foo')
widget.text = 'foo'
widget[:text] = 'foo' または widget['text'] = 'foo'

ウィジェット属性の参照 (1)

- ◇ ウィジェットオブジェクトwidgetの属性参照
 - ◇ widget.cget(属性)
 - ◇ widget.属性名メソッド
 - ◇ widget[属性]
指定した属性の現在値を返す
- ◇ widget.configinfo
widgetに有効なすべての属性について、以下のいずれかの情報(配列)を含むような配列を返す
 - ◇ [属性名, オプションデータベース名, データベースクラス, デフォルト値, 現在値]
 - ◇ [属性の短縮名, 属性の完全名]
- ◇ widget.configinfo(属性)
指定した属性について、configinfoと同じ情報を返す

ウィジェット属性の参照 (2)

- ◇ widget.current_configinfo
widgetに有効なすべての属性について、
{属性名=>現在値, ... }というHashを返す
 - ◇ widget.current_configinfo(属性)
指定した属性について、{属性名=>現在値}という
1要素のみのHashを返す
- (例) widgetのtext属性の情報を得たい場合
widget.cget(:text)または widget.cget('text')
widget.textまたはwidget[:text]またはwidget['text']
widget.configinfoが返す配列から検索
widget.configinfo(:text)
またはwidget.configinfo('text')
widget.current_configinfoが返すHashから検索
widget.current_configinfo(:text)
またはwidget.current_configinfo('text')

ウィジェットの破壊/非表示

- ◇ 不要になったウィジェットwidgetの破壊
`widget.destroy`
`Tk.destroy(widget)`
- ◇ 一時的に表示しないようにするだけなら破壊は不要
→ ジオメトリマネージャの管理から除外すればよい

(例) packジオメトリマネージャで管理していたウィジェットwidgetを管理から除外する
`widget.pack_forget`
`widget.unpack`
`Tk.pack_forget(widget)`
`Tk.unpack(widget)`
`TkPack.forget(widget)`

試してみよう！

- ◇ hello0.rbを書き換えて、属性の参照や変更を試そう
- ◇ TkLabel.newの行をもう1行増やすとどうなる？
- ◇ Tcl/Tkのlabelウィジェットのマニュアルを見よう
 - ◇ 属性の情報などはTcl/Tkのマニュアルを見るのが確実
 - ◇ 「Ruby/Tk講習会 '04/12/11 -- 参考資料」というファイル(seminar2004/seminar-text.sjis)に「Tcl/Tkのマニュアルの活用方法」という章があるので参照すること
- ◇ 日本語(ShiftJIS)を使うのであれば要注意！
 - ◇ `ruby -Ks hello0.rb` ということにオプションを付けなければSyntax Errorになる可能性大
 - ◇ ファイルを分けるのも一つの手段
(`sjis-test-main.rb`と`sjis-text.rb`による例を参照)

irbtkw.rbw(irbtkw2.rb)を使ってみよう

- ◇ 裏(別のthread上で)ですでにTkのイベントループが動いているirb
- ◇ Ruby/Tkをインタラクティブに操作可能
- ◇ exitコマンドの他、rootウィジェットの破壊でも終了
- ◇ コンソール部分もRuby/Tkで作られており、コマンドプロンプトでirbを動かした時に生じるthread切替えのトラブルを回避可能
- ◇ irbtkw2.rbwでは環境に合わせて\$KCODEも自動設定
- ◇ ちょっと反応が悪いことがあります(^_^;

フォント

- ◇ フォントを指定するための属性
 - ◇ family : フォントファミリー名
 - ◇ size : 正の値ならポイントサイズ指定, 負の値ならピクセルサイズ指定
 - ◇ weight : 'normal' or 'bold'
 - ◇ slant : 'roman' or 'italic'
 - ◇ underline, overstrike : それぞれの有無
- ◇ 指定形式
 - ◇ 配列での指定 : [family, size]
または [family, size, style, ...]
styleはシンボルまたは文字列で、次のいずれか
normal/bold/roman/italic/underline/overstrike
 - ◇ Hashでの指定 : { 属性=>値, ... }
underline, overstrikeの値はtrueかfalse

フォントを変更してみよう！

- ◆ TkFont.familiesメソッドで、利用可能なフォントファミリーの一覧が確認可能
- ◆ labelウィジェットのfont属性を操作してみよう
- ◆ 指定されたフォントが存在しない場合、システムに依存したデフォルトのフォントが用いられる
- ◆ フォント変更を繰り返したり、複数のウィジェットで共通にフォント管理をするなら、TkFontオブジェクトの利用を推奨
 - Ruby/Tk講習会 '04/12/11 -- 参考資料」
(seminar2004/seminar-text.sjis)の
「フォントオブジェクト」の項と
その項のサンプルスクリプトとを参照

イメージオブジェクト

- ◆ 組み込みのbitmap以外はオブジェクトの生成が必要
- ◆ TkBitmapクラス：ビットマップ(2値画像)イメージ
 - ◆ Tcl/Tkの'bitmap'のマニュアルを参照
- ◆ TkPhotoImageクラス：フルカラーイメージ
 - ◆ Tcl/Tkの'photo'のマニュアルを参照
 - ◆ 標準ではPPM/PGM, GIFをサポート
 - ◆ Tcl/TkのImg拡張の導入で対応フォーマット増加
(Img拡張はActiveTclに含まれる)
require 'tkextlib/tkimg'に成功なら利用可能
- ◆ サンプルスクリプトphoto-def.rbの定義を参考に、生成したTkPhotoImageオブジェクトをlabelウィジェットのimage属性に指定して表示させてみよう！さらに、compound属性を使って文字列と画像とを同時に表示させてみよう！

ウィジェットデモを試そう！

- ◆ Tkのウィジェットの利用例を示すデモ
- ◆ 'widget'というファイルが各デモのランチャー
- ◆ デモスクリプトのソースを表示し、それを変更して再実行することが可能
 - ◆ オリジナルのファイルを変更するわけではないので、ファイルを壊してしまう心配はほとんどない
- ◆ 再実行の影響が他に及ばないように細工を行っているため、各デモの実行はかなり遅くなっている
 - ◆ 本来の速度が知りたければ、細工なしの旧バージョンを試してみよう
(Ruby1.8.2に添付のものがseminar2004の下にある)
- ◆ ウィジェットデモ以外にも単独で動くサンプルが多数あるので試してみよう！

GUIプログラミングのポイント

- ◆ ウィジェットの選択
- ◆ 属性の設定
- ◆ ウィジェットの配置
- ◆ バインディング

大事なのは操作性の維持や利用の状況に対する想像力

- ◆ サンプルスクリプトでの比較
(とりあえず中身は気にしなくてよい)
 - ◆ entry-bind-a.rb と entry-bind-b.rb
 - ◆ pack-order-a.rb と pack-order-b.rb

buttonウィジェット (TkButtonクラス)

- ◇ 最も利用頻度の高いウィジェットかも？
- ◇ クリックすると登録された手続きを実行する
- ◇ 登録する手続きはcommand属性に与える

```
TkButton.new(:command=>proc{ ... })
TkButton#command{ ... }      ... など
```

(注) Tcl/Tk側からのRuby側の呼び出し(コールバック)に関わる手続きは、TkCallbackEntryのサブクラスのオブジェクトに登録され、管理される。TkCallbackEntry#cmdにより登録手続きを取り出すことができる。

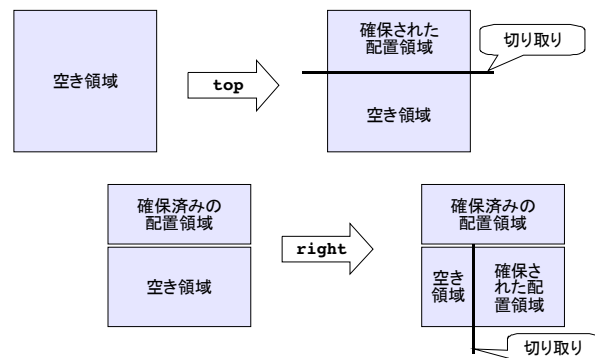
- ◇ hello0.rbのTkLabelをTkButtonに変更し、ボタンのクリック時に“Hello, World!!”と出力するようにしてみよう！

packジオメトリマネージャ

- ◇ Tkで(多分)最も頻繁に使われるジオメトリマネージャ
- ◇ 配置の直接指定ではなく、主に論理構造を指定する
- ◇ 「空き領域モデル」という考え方に基づいて自動配置
 - ◇ 表示領域
 - ◇ ウィジェットの表示に使われる領域
 - ◇ 領域の大きさは一般にウィジェットのサイズに等しい
 - ◇ 配置領域
 - ◇ 表示領域のために切り出された領域
 - ◇ 配置領域サイズ >= 表示領域サイズ
 - ◇ 空き領域
 - ◇ 未使用の領域
 - ◇ 矩形の空き領域を直線で切り分けて、矩形の配置領域を切り出す → 残りの空き領域も矩形

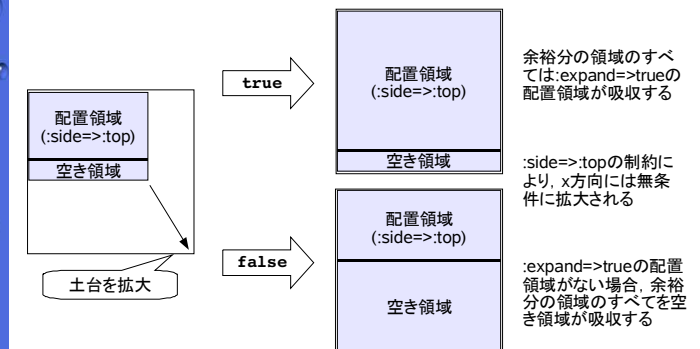
配置領域の確保

- ◇ sideオプションにより切り取る「側」を指定
 - ◇ 指定値: top/bottom/left/right (文字列またはシンボルで指定) デフォルトはtop(上側)



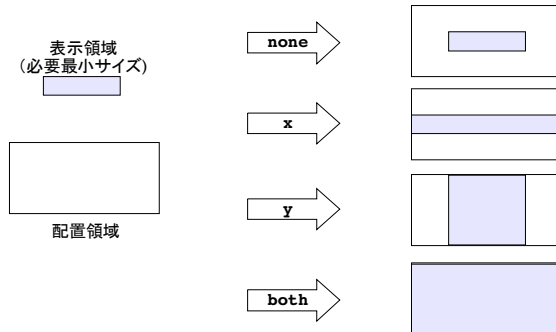
配置領域の拡張

- ◇ expandオプションにより、土台のサイズに余裕があるときに配置領域の大きさを拡張するかどうかを指定
 - ◇ 指定値: true/false デフォルトはfalse(拡張しない)



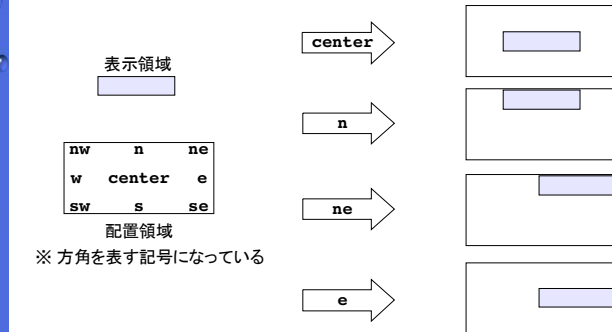
表示領域の拡張

- ◇ fillオプションにより、配置領域 > 表示領域の時に表示領域を拡張するかどうかを指定
- ◇ 指定値：none/x/y/both (文字列またはシンボルで指定)
デフォルトはnone(拡張しない)



表示領域の配置

- ◇ anchorオプションにより、配置領域 > 表示領域の時に配置領域内のどこに表示領域を配置するかを指定
- ◇ 指定値：center/n/ne/e/se/s/sw/w/nw
デフォルトは(center) (文字列またはシンボルで指定)



余白の制御

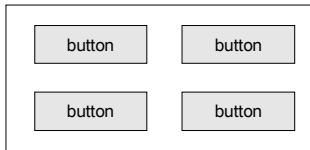
- ◇ padx, pady : 表示領域の外側に余白を確保
- ◇ ipadx, ipady : 表示領域の内側に余白を確保
 - ◇ 配列で二つの値を与えた場合は、それぞれ左(上)側, 右(下)側の余白指定となる
 - ◇ 表示領域内側の余白をどのように使うかは、配置されたウィジェット次第
- ◇ pack-pad-sample.rbを動かして、効果を見てみよう！

配置領域が確保できないときは？

- ◇ 土台の縮小などで領域が不足するときの処理は
packの順序に依存 ⇒ 先のものから優先表示
 - ◇ 均等に縮小したりはしない
 - ◇ 優先順位が高いものはフルサイズで表示
 - ◇ 残り領域が不十分な場合は、そのすべてを使ってできるだけ大きく表示
 - ◇ 残り領域が0になれば、一切表示せずに無視
- ※ 前に試した pack-order-a.rb と pack-order-b.rb との違いを思い出そう！

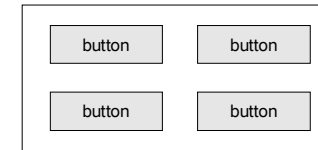
縦横配置

- ◇ packで次の図のような配置がうまくできるか？



縦横配置

- ◇ packで次の図のような配置がうまくできるか？



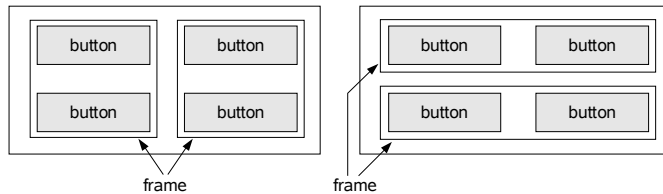
- ◇ 単一の土台の上では不可能！
 - ◇ 一つの土台の上で縦方向と横方向の配置を混ぜると失敗するケースが多い（他のジオメトリマネージャを使う方法は後述）



frameウィジェット(TkFrameクラス)の出番！

frameウィジェット (TkFrameクラス)

- ◇ 他の(複数の)ウィジェットを配置するための土台とすることを主目的としたウィジェット
- ◇ ウィジェットのまとまりを作るために用いられる(新しい複合ウィジェットクラスを作る際にも土台となる)
- ◇ 配置の際のスペーサとして使われることもある



※ frameウィジェットを使って積木のように配置する

土台へのサイズの伝播

- ◇ 通常は中身のサイズに応じて土台の大きさを自動調整
→ 時折、それでは困るケースも．．．

(例) labelウィジェットの文字列を変更する度に
ウィンドウサイズが変更されて困るとか．．．

- ◇ 次の設定をすることで、それ以降、内容物のサイズに影響されなくなる

＜土台のウィジェット＞.pack_propagate(false)

radiobuttonとcheckbox

- ◇ radiobuttonウィジェット(TkRadioButtonクラス)
 - ◇ 組になったradiobuttonの中から一つだけを選択可能
 - ◇ 同じTcl変数を割り当てたウィジェットが組になる
 - ◇ 選択時、ボタンごとに設定した値をTcl変数に代入
- ◇ checkboxウィジェット(TkCheckButtonクラス)
 - ◇ チェックのON/OFFを状態設定
 - ◇ ON/OFFそれぞれの状態での値を設定可能
 - ◇ 状態に応じた値を割り当てられたTcl変数に代入

↓
いずれのウィジェットも、
状態を**Tcl/Tk上の変数**に保存

↓
Rubyからはどうやってアクセスするの？

TkVariableクラス

- ◇ RubyとTcl/Tk上の変数との間の架け橋
(例) `v = TkVariable.new`
`TkRadioButton.new(:variable=>v,`
`:text=>'FOO',`
`:value=>'foo')`
- ◇ Tcl/Tk上の値はすべて文字列となることに注意！
- ◇ アクセスメソッド群
 - ◇ `value, value=` : 文字列としての読み書き
 - ◇ `numeric, numeric=` : 数値としての読み書き
 - ◇ `bool, bool=` : 真偽値としての読み書き
 - ◇ `list, list=` : Tcl/Tkのリスト(RubyではArray)としての読み書き
- ◇ `trace`を設定して、値の書き換え時などに特定の手続きを呼び出すようにすることも可能

TkVariableの対応演算子

- ◇ 利用可能な演算子

`&, |, +, -, *, /, %, **, =~, ==, <=>`

- ◇ 次の例をirb上で試してみよう！

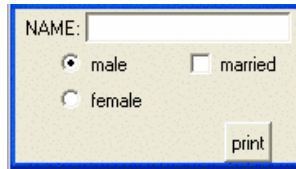
```
v = TkVariable.new(3) # 初期値を"3"に設定
v + 4                 #==> 7
v / 2.0               #==> 1.5
v.numeric *= 2        #==> 6
v.value               #==> "6"
v.numeric             #==> 6
v.value += 'aa'       #==> "6aa"
v.value              #==> "6aa"
v.value *= 3          #==> "6aa6aa6aa"
v.value              #==> "6aa6aa6aa"
v.numeric             #==> ERROR!! (例外発生)
```

entryウィジェット(TkEntryクラス)

- ◇ 1行入力を行うためのウィジェット
- ◇ 内容の読出 : `TkEntry#get`または`TkEntry#value`
- ◇ 内容の設定 : `TkEntry#set`または`TkEntry#value=`
- ◇ `textvariable`属性にTkVariableオブジェクトを与えることで、TkVariableオブジェクトを通じて間接的にアクセスすることも可能

練習問題

- ◇ 次の図のようなものを作ってみよう！



(printボタンのクリックで、入力した情報を出力)

- ◇ 使うもの

TkLabel, TkEntry, TkRadioButton, TkCheckBox, TkFrame, TkVariable

textウィジェット (TkTextクラス)

- ◇ テキスト表示を主目的とした多機能ウィジェット
- ◇ 行と文字位置とで管理するジオメトリマネージャ
- ◇ イメージやウィジェットの埋め込みも可能
- ◇ そのままエディタとして使えるほどの編集機能を持つ
- ◇ テキストの一部に様々な属性を設定可能

→ 機能の実例はウィジェットデモを参照

- ◇ 多機能過ぎて説明しきれないので、本講習会では省略
- ◇ 「Ruby/Tk講習会 '04/12/11 参考資料」に若干の説明あり

(例)

```
require 'tk'
t = TkText.new.pack
TkButton.new(:text=>'output',
             :command=>proc{print t.value}).pack
Tk.mainloop
```

scrollbarウィジェット (TkScrollbarクラス)

- ◇ 単独で用いられることは非常に稀
- ◇ 通常は、スクロール可能な他のウィジェットと組み合わせて用いる
- ◇ スクロール可のウィジェットには次のメソッドが存在
 - ◇ xscrollbar(scr)
引数で与えたscrollbarウィジェットscrを
横方向のスクロールバーとして使うように設定する
 - ◇ yscrollbar(scr)
引数で与えたscrollbarウィジェットscrを
縦方向のスクロールバーとして使うように設定する

scrollbarウィジェットの利用例

- ◇ textウィジェットと連携させてみよう！

- ◇ 縦スクロールのみ

```
scr = TkScrollbar.new.pack(:side=>:right, :fill=>:y)
TkText.new{
  yscrollbar scr
}.pack
```

- ◇ 縦横スクロール

```
xscr = TkScrollbar.new.pack(:side=>:bottom, :fill=>:x)
yscr = TkScrollbar.new.pack(:side=>:right, :fill=>:y)
TkText.new(:wrap=>:none){
  xscrollbar xscr
  yscrollbar yscr
}.pack
```

機能してはいるものの、スクロールバーの配置が格好悪い！

↓
gridジオメトリマネージャの出番

gridジオメトリマネージャ

- ◇ 行(row)と列(column)で配置を管理 ← 「表」の配置
- ◇ 指定方法
 - ◇ 行単位で指定していく方法
Tk.grid(widget1, widget2, ..., options)
 - ◇ ウィジェットごとに行と列の位置(セル)を指定する方法
widget.grid(:row=>行, :column=>列, opt=>val, ...)
- ◇ セル(== 配置領域)内での配置設定
→ stickyオプションで指定
 - ◇ セル内での表示領域の位置と拡張とに関する設定
 - ◇ packのfill, expand, anchorを合わせたようなもの
↓
stickyオプションの指定値と効果は,
grid-sticky-test.rb を参照

gridの利用例

- ◇ 縦横スクロールバー付きテキストウィジェット

```
xscr = TkScrollbar.new.grid(:row=>1, :column=>0,  
                             :sticky=>:ew)  
  
yscr = TkScrollbar.new.grid(:row=>0, :column=>1,  
                             :sticky=>:ns)  
  
TkText.new(:wrap=>:none){  
  xscrollbar xscr  
  yscrollbar yscr  
}.grid(:row=>0, :column=>0, :sticky=>:news)
```
- ◇ ボタンの縦横配置

```
Tk.grid(TkButton.new(:text=>'button 0,0'),  
        TkButton.new(:text=>'button 0,1'))  
  
Tk.grid(TkButton.new(:text=>'button 1,0'),  
        TkButton.new(:text=>'button 1,1'))
```

ジオメトリマネージャの組み合わせ

- ◇ **一つの土台の上で、複数種のジオメトリマネージャを使うことは厳禁!**
- ◇ 組み合わせて管理したい場合は、frameウィジェットを使って管理単位を作り、管理単位ごとに一つのジオメトリマネージャを用いること
- ◇ 「組み合わせないと実現できない」と思った場合に使うべきもの
 - ◇ placeジオメトリマネージャ
 - ◇ canvasウィジェット上の埋め込みウィンドウ配置
... など

バインディング

- ◇ イベント発生に対しての動作を(コールバック)を規定する(コールバック手続きをバインドすること)
- ◇ **GUIの機能/操作性に関する要の一つ**
→ 良質なGUI作成のためには避けて通れない道
- ◇ 多くのウィジェットは、それぞれに独自のバインディングをデフォルトで持つ
(例) buttonウィジェット
→ マウスのボタン1(左ボタン)が離されたならば、command属性に設定された手続きを実行する
entryウィジェット
→ キー入力があったら、対応する文字を挿入する
↓
個々のウィジェットの一般的機能としては十分でも、GUI全体の反応/動作として考えた場合は不足/不十分

バインドの定義

◇ bindメソッドを使う

```
(例) require 'tk'
      label = TkLabel.new(:text=>'hoge').pack
      label.bind('ButtonRelease-1'){puts 'hoge'}
      Tk.mainloop
```

◇ バインディングを行いたい対象をtargetとすると、

- ◇ target.bind(イベントシーケンス, 手続き)
- ◇ target.bind(イベントシーケンス){ 手続き }
- ◇ target.bind(イベントシーケンス, 手続き, 引数, ...)
- ◇ target.bind(イベントシーケンス, 引数, ...){ 手続き }

↓

「バインディング『対象』? 単にウィジェットじゃないの?」
「イベント『シーケンス』って何? どう書くの?」
「引数? 何を渡すの? 何でそんなものが必要なの?」

... こうした疑問がバインディング理解のポイント!

イベントシーケンス

◇ 手続きを割り当てる「状況」の指定

◇ 指定形式

- ◇ イベントパターン
適合するイベントの種類の指定
- ◇ イベントパターンの配列
イベントパターンに適合するイベントが、配列の順に連続して発生した状況を指定
- ◇ 仮想イベント
イベントシーケンスの集合に対して付けた別名で、その集合のいずれが発生してもこの仮想イベントが発生した状況とみなす
(TkVirtualEventクラス, TkNamedVirtualEventクラス)

↓
Tcl/Tkで定義済みの仮想イベントをオブジェクト化する場合などに利用

イベントパターン (1)

◇ 部分省略可能な次の形式の文字列

MOD-...-MOD-TYPE-DETAIL

◇ TYPE : イベントの種類

Activate	Destroy	Map
ButtonPress, Button	Enter	MapRequest
ButtonRelease	Expose	Motion
Circulate	FocusIn	MouseWheel
CirculateRequest	FocusOut	Property
Colormap	Gravity	Reparent
Configure	KeyPress, Key	ResizeRequest
ConfigureRequest	KeyRelease	Unmap
Create	Leave	Visibility
Deactivate		

◇ DETAIL : イベントの詳細

例えば、キー入力イベントなら押されたキーの名称、
マウスクリックであれば押されたボタンの番号

イベントパターン (2)

◇ MOD : イベントの修飾子

Control	Button1, B1	Mod1, M1	Meta, M
Shift	Button2, B2	Mod2, M2	Alt
Lock	Button3, B3	Mod3, M3	Double
Extended	Button4, B4	Mod4, M4	Triple
	Button5, B5	Mod5, M5	Quadruple

◇ イベントパターンの例

‘a’, ‘KeyPress-a’
キーボードの‘a’というキーを押した

‘KeyRelease’
キーボードのキー(どのキーかは問わない)が離された

‘1’, ‘ButtonPress-1’
マウスのボタン1(左ボタン)を押した
(数値のみはキーではなくボタン番号と見なされる)

‘Control-Shift-Alt-B3-Double-KeyPress-Return’
Controlキー, Shiftキー, Altキー, マウスの右ボタンを同時に押しながら, Returnキーを2回連続で叩いた

‘B1-Enter’, ‘Button1-Enter’
マウスの左ボタンを押したまま状態で, マウスカーソルがウィジェット上に進入した

コールバック手続きに渡される引数

- ◇ 発生したイベントについての詳細な情報も必要
 - ◇ イベントが発生したウィジェットは？
 - ◇ イベント発生時のマウスポインタの座標は？
 - ◇ 入力されたキーのキーシンボルは？
 - ... などなど
 - ◇ bindメソッドの「引数」で指定する
 - ◇ bind(evseq){ ... }のように「引数」指定がない場合
→ TkEvent::Eventオブジェクトが渡される
 - ◇ 1個以上の「引数」が指定されている場合
→ 指定に応じた個数と種類で情報が渡される
- ※ イベントタイプに依存した詳細情報の種類は、Tcl/Tkのbindのマニュアルを参照すること

TkEvent::Eventオブジェクト

- ◇ すべての詳細情報の情報を含んだオブジェクト
- ◇ 有効な情報か無効な情報かは、イベントタイプに依存
- ◇ bindメソッドの引数が登録する手続きに依存しない点は手軽だが、不必要な情報まで処理してしまう分、遅い

(例) `proc{|ev| ... }`

ev.widget : イベントが発生したウィジェット
ev.x : イベントが発生したX座標(ウィジェット左上が原点)
ev.y : イベントが発生したY座標(ウィジェット左上が原点)
ev.rootx : スクリーン上のX座標(スクリーン左上が原点)
ev.rooty : スクリーン上のY座標(スクリーン左上が原点)
ev.char : キーボードで押されたキーの文字
ev.keysym : キーボードで押されたキーのキーシンボル
... など

イベント詳細情報の個別指定

- ◇ bindメソッドの「引数」→ 基本的に文字列で指定
(空白を含む場合はsplitしたそれぞれが引数として扱われる)
- ◇ %置換が可能な'%'+1文字
→ TkEvent::Eventオブジェクトと同様の型で返す
 - %W ⇒ ev.widget
 - %x ⇒ ev.x
 - %y ⇒ ev.y
 - %X ⇒ ev.rootx
 - %Y ⇒ ev.rooty
 - %K ⇒ ev.keysym ... など
- ◇ 引数文字列中の'%'+1文字
→ 可能なら%置換が実施される
(置換文字列はTcl/Tk上の情報表現文字列となる)
- ◇ その他
→ Tcl/Tk上の表現に変換した文字列が渡される

イベント詳細情報指定引数の例

(例) `tgt.bind(seq,'%W','%xW','%x %y',123){|a,b,c,d,e| ... }`

'%W' ⇒ a ::= イベントが発生したウィジェットオブジェクト
'%xW' ⇒ b ::= 'x'+ウィジェットのパス文字列
'%x %y' ⇒ '%x' ⇒ c ::= イベント発生時のX座標
⇒ '%y' ⇒ d ::= イベント発生時のY座標
123 ⇒ e ::= 文字列"123"

※ 先に出てきたlabelウィジェットへのバインディング例を少し修正して、この例を試してみよう！

コールバックの疑問

- ◇ 以下を実行して、ボタンをクリックするとどうなる？
結果を予想した上で試してみよう！

```
frame = TkFrame.new(Tk.root).pack
b = TkButton.new(frame, :text=>'foo',
                  :command=>proc{p :foo}).pack
b.bind('ButtonRelease-1'){p :bind}
frame.bind('ButtonRelease-1'){p :frame}
Tk.root.bind('ButtonRelease-1'){p :root}
```

コールバックの疑問

- ◇ 以下を実行して、ボタンをクリックするとどうなる？
結果を予想した上で試してみよう！

```
frame = TkFrame.new(Tk.root).pack
b = TkButton.new(frame, :text=>'foo',
                  :command=>proc{p :foo}).pack
b.bind('ButtonRelease-1'){p :bind}
frame.bind('ButtonRelease-1'){p :frame}
Tk.root.bind('ButtonRelease-1'){p :root}
```

- ◇ `:bind`, `:foo`, `:root`の順に出力されたはず
→ でも、なぜ???

- ◇ なぜ1回のイベントに複数の手続きがよばれるのか？
- ◇ なぜバインディングが上書きされないのか？
- ◇ デフォルトのバインディングは禁止できないのか？
- ◇ なぜrootウィジェットのバインディングまで実行を？
- ◇ なぜframeウィジェットについては実行されない？

バインドタグリスト

- ◇ 次の命令を実行してみると,
`p TkButton.new.bindtags`

次のような配列が返されるはず

```
[ #<TkButton:0xb7bf7540 @path=".w00000">,
  TkButton,
  #<TkRoot:0xb7bdfe7c @path=".">,
  #<TkBindTag: all> ]
```

↓

これが**バインドタグリスト**と呼ばれるもの

||

バインディングの適用の可否や順序を制御する機構

バインドタグ

- ◇ バインディングの対象 = バインドタグ
- ◇ バインドタグになるもの
 - ◇ ウィジェットオブジェクト
 - ◇ ウィジェットクラス
 - ◇ TkBindTagオブジェクト
 - ◇ 文字列 (←指定は可能だが、扱いに手間を要するので非推奨)
- ◇ コールバックの実行
 - ◇ バインドタグリストの順に、バインドタグに登録されたバインディングを実行
 - ◇ 各バインドタグにおいて実行されるのは最大でも1個
 - ◇ 選択ルールは「より詳細なシーケンス指定ほど優先」で、さらに「より後から定義した方が優先」となる

デフォルトのバインドタグリスト

- ◇ 標準でウィジェットが持つバインドタグリスト
 - ウィジェットオブジェクト (自分自身)
 - ウィジェットクラス (自分自身のクラス)
 - ウィンドウを形成する最上位のウィジェット (例えばrootウィジェット)
 - TkBindTag::ALL ('all' という文字列に対応するTkBindTagオブジェクト)
- ◇ つまり、先にbuttonウィジェットで試した例の出力は
 - ◇ バインドタグリストの先頭であるウィジェットオブジェクトにバインドしたコールバック
 - ◇ 2番目の要素であるウィジェットクラスのコールバック (ウィジェットクラスのデフォルトのコールバック)
 - ◇ 3番目の要素であるrootウィジェットにバインドしたコールバックの順に実行がなされた結果ということ

バインドタグリストの操作

- ◇ バインドタグリストは**bindtags**=メソッドで変更可
(例) `widget.bindtags = [widget, widget.class]`
 - ◇ ウィジェットクラスをリストから除くと、そのクラス特有のバインディングが働かなくなる
 - ◇ バインディングを行ったTkBindTagオブジェクト追加したり除いたりすれば、複数のバインディングをまとめて付与したり停止したりが容易になる
- ◇ 特定のウィジェットだけで、ウィジェットクラスのバインディングの一部だけを無効化することは可能か？
→ バインドタグリストの順次実行を強制的に打ち切ることでほぼ実現可能
 - ◇ コールバック中で捕捉されないbreakを実行する (あるいはTk.callback_breakを実行する)
 - ◇ break実行後、残りのバインドタグリストは無視される

キーボードイベントとフォーカス

- ◇ フォーカス
 - ◇ キーボードに関するイベントが送られる先を規定
 - ◇ ウィジェットの**focus**メソッドで設定可能
- ◇ フォーカスを適切にコントロールすることで、ユーザにストレスを与えることなく、操作をうまく誘導することができる

練習問題

- ◇ 4個のentryウィジェットを生成する。
各entryウィジェットにマウスカーソルが進入したら、そのウィジェットにフォーカスが設定されるようにする。
- ◇ 扱うべきイベント: Enter
- ◇ すべてのentryウィジェットに同様のバインディングを行うことを考えて、記述はできるだけ少なく済まそう！
(例えば、'%W'を使うと...))

listboxウィジェット (TkListboxクラス)

- ◆ 項目リストの表示や操作を行うためのウィジェット
- ◆ リストのインデックスは0から始まる数値で指定
- ◆ 特殊なインデックス(文字列またはシンボルで指定)
 - ◇ end : 末尾の項目の次の位置
 - ◇ active : 位置カーソルが存在する項目
 - ◇ anchor : セレクションのアンカー位置
 - ◇ @x, y : 座標(x, y)の位置を含む項目

listboxウィジェットのメソッド例

- ◆ リスト内容の変更
 - TkListbox#inset(index, val, ...) : 項目挿入
 - TkListbox#delete(first [, last]) : 指定範囲の項目消去
 - TkListbox#clear : 全項目の消去
 - TkListbox#value=(array) : 全項目の内容を一度に置き換え
- ◆ 情報獲得
 - TkListbox#get(index) : 1 項目の内容(文字列)を得る
 - TkListbox#get(first, last) : 指定範囲の内容を配列で得る
 - TkListbox#value : 全項目の内容を配列で得る
 - TkListbox#nearest(y) : y座標に最も近い項目のインデックス
 - TkListbox#see(index) : 項目が表示されるようにスクロール
- ◆ 項目ごとの属性操作
 - TkListbox#itemcget, TkListbox#itemconfigure, TkListbox#itemconfiginfo, TkListbox#current_itemconfiginfo : 第1引数に項目のインデックスを与える以外はconfigure等に類似

練習問題

- ◆ 1 個のlabelウィジェットと1 個のlistboxウィジェット (scrollbarウィジェットと関連付けてスクロール操作を可能にする)と1 個のentryウィジェットを持つGUIを作る。

entryウィジェットに入力してReturn(Enter)キーを叩くと、その文字列をlistboxウィジェットの末尾に加える。なお、加えた直後はその項目(つまりリスト末尾)が表示されるようにする。

さらに、listboxウィジェットの項目をクリックすると、labelウィジェットの表示文字列がクリックした項目の文字列に変更されるようにする。
- ◆ 扱うべきイベント :
ButtonPress-1, Return
- ◆ 使うメソッドの例 :
TkListbox#yscrollbar, TkListbox#nearest, TkListbox#get, TkListbox#insert, TkListbox#see

タイマー処理

- ◆ 一定時間後あるいは一定時間毎に処理を行いたい場合
 - ◇ **コールバック手続き中でのsleepは絶対にダメ!**
→ イベントループ自体がその間停止してしまう
= GUIの動作が停止してしまう
- ◆ Ruby/Tk上のタイマー処理を利用するのが安全
- ◆ ただし、タイマー処理の時間精度の甘さには注意!
 - ◇ 時間指定は「およそ」であって、保証はされていない
 - ◇ 同じ処理を繰り返しても、時間間隔が同じとは限らない
 - ◇ リアルタイム性についての保証は全くない

タイマー処理の種類

- ◇ Tk.after(ミリ秒) { ... }
指定時間後にタイマー割り込みで手続きを実行(1回限り)するように設定する (このメソッド自体はすぐに終了する).
- ◇ TkTimer.new(ミリ秒, 回数, 手続き, ...).start
指定の時間間隔で指定されたすべての手続きを順次実行する処理を指定回数(-1 なら無限回)ループする.
- ◇ TkRTimer.new(ミリ秒, 回数, 手続き, ...).start
処理コストは大きくなるが, 時間精度をTkTimerクラスよりも若干改善したクラス.
例えば時計アプリなどで, 処理時間のずれの蓄積で時計が狂ってしまう危険を極めて小さくすることができる.

練習問題

- ◇ タイマー処理で, 次のようなものを作ってみよう!
 - ◇ クリックされると10秒後に終了する自爆ボタン
 - ◇ 何もしないうちに終了するだけならTk.afterで十分
 - ◇ 可能なら, カウントダウンの表示をしよう!
→ この場合は, TkRTimerクラスの利用が楽
 - ◇ 一定時間間隔で点滅しつづけるlabelウィジェット
 - ◇ 消えた状態にするには, 文字列の表示色(foreground)を背景色(background)と同じに指定すればよい
 - ◇ Tk.afterの中で次のタイマー処理を設定する方法や, TkTimerクラスを利用する方法がある
 - ◇ クリックされると3回点滅するlabelウィジェット
 - ◇ 消す手続きと戻す(点ける)手続きとを3回繰り返すと考えれば, TkTimerクラスの利用が楽

toplevelウィジェット(TkToplevelクラス)

- ◇ マルチウィンドウアプリケーションを作るには必須のウィジェットクラス
- ◇ rootウィジェットとは別のウィンドウを生成する
- ◇ 機能的にはrootウィジェットと同じ
(独自のメニューバーを持つこともできる)
- ◇ バインドタグリストにおける「ウィンドウを形成する最上位のウィジェット」となり得るのは, rootウィジェットとtoplevelウィジェットのみ

練習問題

- ◇ 次のようなGUIを作ってみよう!
 - rootウィジェット上のボタンを左クリックすると, 新しいウィンドウ(toplevelウィジェット)を生成する.
 - 生成した新しいウィンドウ上のボタンをクリックすると, ウィンドウを破棄する.

menuウィジェット (TkMenuクラス)

- ◇ メニュー項目の種類
 - ◇ コマンド : ボタンウィジェットに類似
 - ◇ チェックボタン : チェックボタンウィジェットに類似
 - ◇ ラジオボタン : ラジオボタンウィジェットに類似
 - ◇ カスケード : サブメニューを呼び出す
 - ◇ セパレータ : 項目の分離線を表示
 - ◇ (ティアオフ) : メニューの切り離しを可能にする
(通常の項目とは異なる特殊な項目)
- ◇ TkMenu#addメソッドにより項目を一つずつ追加する
- ◇ 項目ごとの属性操作
TkMenu#entrycget, TkMenu#entryconfigure,
TkMenu#entryconfiginfo, TkMenu#current_entryconfiginfo
第1引数に項目のインデックスを与える以外はconfigure等に類似

メニュースペック (TkMenuSpecモジュール)

- ◇ メニュー項目を一つずつ作っていくのは少々面倒！
- ◇ Ruby/Tkには「メニュースペック」と呼ぶ形式が存在
 - ◇ メニューの構造を配列を使って定義したもの
 - ◇ TkMenu.new_menuspecメソッドにより、メニュースペックで示された構造のメニューを構築可能
 - ◇ TkRootまたはTkToplevelウィジェットのadd_menubarメソッドを用いれば、メニュースペックに示された構造のメニューバーを簡単に構築可能

メニュースペックの書式

- ◇ menuspec ::= [menuinfo, menuinfo, ...]
- ◇ menuinfo ::= [btninfo, iteminfo, iteminfo, ...]
- ◇ btninfo ::= [text, underline, configs]
- ◇ iteminfo ::= cmd_ent | check | radio | cascade | separator
- ◇ cmd_ent ::= [label, command,
underline, accelerator, configs]
- ◇ check ::= [label, TkVar_obj,
underline, accelerator, configs]
- ◇ radio ::= [label, [TkVar_obj,value],
underline, accelerator, configs]
- ◇ cascade ::= [label, [iteminfo, ...],
underline, accelerator, configs]
- ◇ separator ::= '---'

メニューバーの構築

- ◇ メニュースペックを用いた構築例
menubar-test.rb
- ◇ Windowsでは、ウィジェット配置よりも先にメニューバーを設置しなければ、表示が不適切になる場合あり
- ◇ WindowsのSystem Menuにメニュー項目を追加したい場合は:menu_name=>'system'という属性を設定する
System Menu : タイトルバーを右クリックしたときに
表示されるメニュー
- ◇ System Menuへの項目追加は、メニュースペックの末尾で行うこと (それ以降の定義は無視されてしまう)

組込みダイアログ

- ◇ Tk.messageBox : tk_messageBoxコマンドのwrapper
 - ◇ メッセージとボタンとを表示して、押されたボタンの情報を得るような組み込みダイアログ
 - ◇ アイコン : error/info/question/warning
 - ◇ ボタン構成 : abortretryignore/ok/okcancel/retrycancel/yesno/yesnocancel
- ◇ Tk.getOpenFile : tk_getOpenFileコマンドのwrapper
 - ◇ 開くファイルを選択する場合のための組み込みダイアログ
 - ◇ 存在しないファイルの指定時はエラーメッセージを表示
- ◇ Tk.getSaveFile : tk_getSaveFileコマンドのwrapper
 - ◇ 保存ファイルを選択する場合のための組み込みダイアログ
 - ◇ 既存のファイルの指定時は上書き確認ダイアログを表示
- ◇ Tk.chooseDirectory : tk_chooseDirectoryコマンドのwrapper
 - ◇ ディレクトリを選択するための組み込みダイアログ
- ◇ Tk.chooseColor : tk_chooseColorコマンドのwrapper
 - ◇ 色を指定に合わせた組み込みダイアログ

TkDialogクラス

- ◇ Tk.messageBoxよりも少しだけ自由度の高いダイアログを作りたい場合に用いる
- ◇ 属性設定
 - 属性は以下のメソッドを再定義して必要情報を返すようにするか、newの際に同名属性(default_buttonのみ'default')で指定する
 - ◇ title : ダイアログのタイトル文字列
 - ◇ message : ダイアログに表示するメッセージ文字列
 - ◇ message_config : メッセージ文字列の属性を指定するHash
 - ◇ msgframe_config : メッセージのフレームの属性を指定するHash
 - ◇ bitmap : 表示するビットマップ
 - ◇ bitmap_config : ビットマップの属性を指定するHash
 - ◇ default_button : デフォルトのボタン番号か名前(nilなら設定なし)
 - ◇ buttons : 表示するボタンのラベルの配列
 - ◇ btnframe_config : ボタン群のフレームの属性を指定するHash
 - ◇ button_configs : ボタンごとの属性設定に必要な情報を得るためのProc/Array/Hashのいずれか(nilなら設定なし)
 - ◇ prev_command : ダイアログ表示直前に実行する手続き
- ※ 具体的にはライブラリのソース(<ruby-lib>/tk/dialog.rb)を参照

練習課題

- ◇ 以前に出たサンプル「縦横スクロールバー付きのtextウィジェット」を参考にしつつ、ファイル入出力のためのメニューを追加しよう！
- ◇ ファイル選択には、組込みダイアログを使えばよい
Tk.getOpenFile, Tk.getSaveFile
 - ◇ 選択されたファイル名を返す
 - ◇ キャンセルなどの場合は空文字列を返す
- ◇ textウィジェットの内容の読み書きは、
TkText#value, TkText#value=
を使えばよい
- ◇ スクリプトを終了するためのメニュー項目も加えよう

ウィンドウマネージャとの連携

- ◇ Tk::Wmモジュール
 - ◇ ウィンドウマネージャへの指示および情報参照
 - ◇ TkRootクラスとTkToplevelクラスとはinclude済み
- ◇ メソッド例
 - title, title(str), title=str : ウィンドウタイトルの参照/設定
 - withdraw : ウィンドウの非表示を指示
 - iconify : ウィンドウのアイコン化を指示
 - deiconify : ウィンドウの表示を指示(非表示/アイコン化からの復帰)
 - overrideredirect, overrideredirect(mode), overrideredirect=mode
 - : ウィンドウに枠を付けるかどうか(ウィンドウマネージャの管理下に置かないようにするかどうか)の参照/設定。
 - 管理下に置かないならtrue, 置くならfalseを設定する。
 - (OSによっては、状態変更には一旦非表示にする必要あり)
 - geometry, geometry(geom_str), geometry=geom_str
 - : ウィンドウのサイズや位置の参照/設定
 - maxsize, maxsize(x,y), maxsize=[x,y] (minsizeも同様)
 - : ウィンドウの最大/最小サイズの参照/設定
 - resizable, resizable(xmode,ymode), resizable=[xmode,ymode]
 - : ウィンドウの縦横のサイズをユーザが変更可能かを参照/設定。
 - trueであれば変更可能

不意のウィンドウ消去の防止

- ◇ ありがちなトラブル
「あ、間違えてウィンドウを消しちゃった！(T_T)」を防ぎたい
→ ウィンドウマネージャプロトコルの管理が必要
- ◇ プロトコルの種類
 - ◇ WM_DELETE_WINDOW : ウィンドウの消去を要求
 - ◇ WM_TAKE_FOCUS : 入力フォーカスの割り当て
 - ◇ WM_SAVE_YOURSELF : 状態保存を要求
- ◇ Tk::Wm#protocolメソッドで、ウィンドウマネージャプロトコルの管理が可能

Tk::Wm#protocolメソッド

- ◇ 利用方法
 - ◇ Tk::Wm#protocol(プロトコル名, 手続き)
Tk::Wm#protocol(プロトコル名){ ... }
: 指定されたプロトコルのハンドラとして手続きを登録。
プロトコル名は文字列かシンボルで与える。
前者の形式で手続きに空文字列を与えた場合には、プロトコルハンドラの消去となる。
 - ◇ Tk::Wm#protocol(プロトコル名)
: プロトコルに対して登録されたハンドラ(TkCallbackEntryオブジェクト)を返す。
 - ◇ Tk::Wm#protocol
: ハンドラが登録されたプロトコル名のリストを返す。
 - ◇ Tk::Wm#protocols
: 登録済みのすべてのハンドラを{プロトコル=>ハンドラ}の形式のHashで返す。
 - ◇ Tk::Wm#protocols(プロトコル=>手続き, ...)
: 複数のプロトコルに対する登録を一度に指定して行う。
手続きが空文字列なら、そのプロトコルについてはハンドラの消去となる。

Tk::Wm#attributes

- ◇ OS (ウィンドウシステム)に固有のウィンドウ属性を参照/操作するためのメソッド
 - ◇ attributes : 有効な{属性=>値}のHashを返す
 - ◇ attributes(属性) : 属性値を参照
 - ◇ attributes(属性, 値) : 属性値を設定
 - ◇ attributes(属性=>値, ...) : 複数の属性値を一度に設定
- ◇ Windows XPにおける固有属性の例
 - ◇ alpha
ウィンドウの透明度レベルを設定する。
0.0(全透過)〜1.0(非透過)の実数値を設定する。
 - ◇ transparentcolor
ウィンドウの透明色を設定する。
指定された色で描画されたピクセルは全透過となる。
空文字列を指定した場合は「透明色なし」となる。
- ◇ irbtkw.rbwを使ってalpha属性とtransparentcolor属性とを操作してみよう！

非矩形半透過ウィンドウのGUIを作ってみよう！

- ◇ ウィンドウの枠をなくす
→ TkRoot#overrideredirect
- ◇ 枠無しウィンドウを移動(ドラッグ)可能にする
→ TkRoot#geometry
widget.bind('1') { 起点を記録 }
widget.bind('B1-Motion') { ウィンドウ移動 }
- ◇ ウィンドウの半透過設定
→ TkRoot#alpha
- ◇ ウィンドウの透明部分の設定
→ TkRoot#attributes(:transparentcolor, 透明色)
widget.background = 透明色
- ◇ ウィンドウの形状の描画
→ labelウィジェットのイメージ表示を利用する方法や
canvasウィジェット(TkCanvasクラス)を利用する方法など

canvasウィジェット (TkCanvasクラス)

- ◇ 図形要素も扱える極めて高機能なウィジェット
 - ◇ すべての図形要素(キャンバスアイテム)について、描画や操作だけでなく、バインディングを行うことも可能
 - ◇ 他のウィジェットを埋め込んで座標で管理することが可能
== 座標で管理するジオメトリマネージャ
 - ◇ 表示範囲外への描画や表示範囲のスクロールが可能
 - ◇ 描画した内容をPostscript形式で出力することが可能
- ◇ キャンバスアイテムの種類
 - ◇ TkArcクラス : 円弧
 - ◇ TkBitmapクラス : ビットマップ
 - ◇ TkImageクラス : イメージ
 - ◇ TkLineクラス : 線
 - ◇ TkOvalクラス : 楕円
 - ◇ TkPolygonクラス : 多角形
 - ◇ TkRectangleクラス : 矩形
 - ◇ TkTextクラス : テキスト
 - ◇ TkWindowクラス : 埋め込みウィンドウ

キャンバスアイテムの生成

- ◇ ウィジェットオブジェクト生成と類似
クラス.new(canvas, 座標情報, ... , オプション)
- ◇ 座標情報
 - ◇ どれだけの数の座標情報を必要とするかは、キャンバスアイテムの種類に依存
 - ◇ 座標の指定形式
(フラットに展開されるので、以下のどの形式でも可能)
 - ◇ x0, y0, x1, y1, ...
 - ◇ [x0, y0, x1, y1, ...]
 - ◇ [x0, y0], [x1, y1], ...
 - ◇ [[x0, y0], [x1, y1], ...]

3個のボタンを持つ楕円形のGUIにしてみよう！

- ◇ ウィンドウの形状の描画部分では以下を参考にして、非矩形半透過のGUIを作ってみよう！
- ◇ キャンバス生成
TkCanvas.new(:width=>楕円幅, :height=>楕円高さ,
:background=>透過色,
:highlightthickness=>0)
- ◇ 楕円の描画
TkOval(canvas, 左上角x, 左上角y, 右下角x, 右下角y,
:outline=>枠の色, :fill=>塗りつぶし色)
- ◇ ボタンウィジェットの埋め込み
TkWindow(canvas, x座標, y座標, :window=>ボタン)
- ※ 作例はnon-rect-win.rb

操作可能範囲のコントロール

- ◇ グラブを設定する
→ グラブを設定したウィジェットとその子孫のウィジェット以外のウィジェットの操作を禁止する
||
期待しない操作が行われることを回避
- ◇ 独自のダイアログを作成する場合によく用いられる
(例)パラメータ指定のダイアログを開いている状態
(値が未確定)で、パラメータを必要とする操作
が行われることを禁止する
- ◇ メソッド例
 - ◇ target.grab : targetにローカルグラブを設定
 - ◇ target.grab_release : targetからグラブを解放
 - ◇ target.grab_current : targetと同じウィンドウ上で、
現在グラブが設定されている
ウィジェットを返す

グラフの操作

- ◇ 重要なのはグラフ解放のタイミング
 - ◇ ウィンドウの破壊を待つ解放
 - ◇ ダイアログのようなモーダルウィンドウで有効な手段
 - ◇ `win.wait_destroy`を実行 == `win`が破壊されるまで停止
 - ◇ グラフが不要になった時点で`win`を破壊
 - ==> `win.grab` → `win.wait_destroy` → `win.grab_release`
という流れ
- ◇ TkVariableへの値書き込みを待つ解放
 - ◇ ウィジェットを破壊したくない(再利用したいなど)場合に有効な手段
 - ◇ `TkVariable#wait`を実行
 - == 変数(`TkVariable`)への書き込みが行われるまで停止
 - ◇ グラフが不要になった時点で変数への値設定(書き込み)
- ==> `win.grab` → `var.wait` → `win.grab_release` という流れ
- ◇ その他
確実に解放タイミングを捉えられる方法を考えること！

複合ウィジェットのクラス化

- ◇ 複数のウィジェットを組み合わせて構築したまとまりを一つのウィジェットクラスにするなら？
(例) scrollbar付きのlistboxを一つのウィジェットとして扱う
 - ◇ ウィジェットオブジェクトのメソッド呼び出し等の対象
 - ◇ ジオメトリマネージャ等の配置関係のメソッド
 - `@epath`に設定されたウィジェットパスが対象
 - ◇ その他の一般的なメソッド呼び出し
 - `@path`に設定されたウィジェットパスが対象
 - ◇ TkCompositeモジュール
 - ◇ 複合ウィジェットクラスの定義を助けるモジュール
 - ◇ `include`により、`initialize`の再定義や、複合ウィジェットの定義を助けるためのいくつかのメソッドの定義を実施
 - ◇ `initialize_composite`というメソッドを再定義して、その中で複合ウィジェットを構築するようにする
- ※「Ruby/Tk講習会 '04/12/11 参考資料」の説明を参照

Tcl/Tk拡張の利用

- ◇ 世の中には多種多様なTcl/Tk拡張ライブラリが存在
 - ほぼすべてをRuby/Tkから利用可能
- ◇ 一部のTcl/Tk拡張については、Ruby/Tkの一部としてすぐに使えるようなwrapperライブラリが標準添付
 - ◇ Tcl/Tk拡張のwrapperライブラリは自動では読み込まれないので、`require 'tkextlib/<ライブラリ>'`の実行が必要
 - ◇ `<ruby-lib>/tkextlib/pkg_checker.rb`を実行すれば、現在の環境で使える状態にあるものの確認が可能
 - ◇ Rubyのソースの`ext/tk/sample/tkextlib`の下にはいくつかのTcl/Tk拡張のデモスクリプトがあるので試してみるとよい
- ◇ wrapperライブラリがないものは？
 - ◇ Tcl/Tkのコマンドを直接呼んでやればよい
 - ◇ `Tk.tk_call(token, ...)`
各引数をTclのコマンド行の構成要素としてTcl/Tk側を呼ぶ
 - ◇ `Tk.ip_eval(tcl_script)`
引数の文字列をTclスクリプトとしてTcl/Tk上で実行する

おわりに

- ◇ Tcl/Tkは古くから存在しますが、現在も進化が進行中です。
 - ◇ 古くから存在するということは、連携できるライブラリの多さにも繋がっています。
 - ◇ 現在開発中のTcl/Tk8.5には既に複数の新機能が追加済みです。
 - ◇ 計算機の発達で速度が問題になるケースは少なくなりましたが、もちろん現時点で不得手な分野は存在します。
- ◇ 新しいGUIライブラリのように特定目的に対応する多種多様なウィジェットは持っていませんが、シンプルであるが故に、数行程度のお手軽GUIから高度で繊細なコントロールまで、必要に応じて作成可能です。
- ◇ Ruby/TkではTcl/Tkにオブジェクト指向の殻を被せてますから、ライブラリデザインの古さはかなりカバーできています。
- ◇ 入口が広く奥が深いので、深いところにある魅力に触れる前に逃げてしまう人が多いのも事実です。
- ◇ どうしても特殊なウィジェットが必要ななら、世の中に多種多様に存在するTcl/Tk拡張から探してみましょう。

あなたはRuby/Tk (Tcl/Tk) の実力と魅力とを
十分に引き出せるでしょうか？