INF2611 Visual Programming II Notes

October 18, 2023

Contents

Unit 1

Advanced Widgets

1.1 System Clock Time in LCD

LCD-like digits are displayed using the LCD Number widget, an instance of the `QLCDNumber` class.

QLCDNumber Methods

- `setMode(modeType) -> None`, where `modeType` is `Hex`, `Dec`, `Oct`, or `Bin`. `Dec` is default.
- `display() -> None`
- `value() -> int`

1.1.1 Timers

Used to perform repetitive tasks. Uses an instance of `QTimer` class. Connect the `timeout()` signal of `QTimer` to the slot that performs the desired task.

Timeout Signals

- `start(n)`: sets the timer to generate a timeout signal at n millisecond intervals.
- `setSingleShot(True)`: sets the timer to generate a timeout signal only once.
- `singleShot(n)`: sets timer to generate timeout signal only once after n milliseconds.

1.1.2 System Clock Time

Use `QTime` to get system clock time, and measure span of elapsed time. Time returned is in 24-hour format.

QTime Methods

- `currentTime()`
- `hour()`
- `minute()`
- `seconds()`
- `msec()`
- `addSecs()`
- `addMSecs()`
- `secsTo()`
- `msecsTo()`

1.2 Calendars and Dates

Calendars are displayed using the `QCalendarWidget` class. By default, it displays the current month and year. Days displayed in abbreviated forms, and weekends marked in red. Week numbers displayed, Sunday is first column.

QCalendarWidget Properties

- `minimumDate`
- `maximumDate`
- `selectionMode`: Set to `NoSelection` to prevent user from selecting date.
- `verticalHeaderFormat`: Set to `NoVerticalHeader` to remove week numbers.
- `gridVisible`
- `HorizontalHeaderFormat`
  - `SingleLetterDayNames`
  - `ShortDayNames`
  - `LongDayNames`
  - `NoHorizontalHeader`

QCalendarWidget Methods

- `selectedDate()`
- `monthShown()`
- `yearShown()`
- `setFirstDayOfWeek()`
- `selectionChanged()`

1.2.1 QDate

Date selected in `QCalendarWidget` returned as a `QDate` object. Contains a calendar date with year, month, and day in Gregorian calendar. Current date read from system clock.

QDate Methods

- `currentDate()`
- `setDate()`
- `year()`
- `month()`
- `day()`
- `dayOfWeek()`
- `addDays()`
- `addMonths()`
- `addYears()`
- `daysTo()`
- `daysInMonth()`
- `daysInYear()`
- `isLeapYear()`
- `toPyDate()`

Date Formats

d	Day as a number, no leading zero.	**MM**	Month as a number, leading zero.
dd	Day as a number, leading zero.	**MMM**	Month in abbreviated form.
ddd	Day in abbreviated form.	**MMMM**	Month in long form.
dddd	Day in long form.	**yy**	Year as two digits.
M	Month as a number, no leading zero.	**yyyy**	Year as four digits.

1.2.2 QDateEdit

Display the date a user selects in a Calendar widget. Used for displaying and editing dates.

QDateEdit Properties

- `minimumDate`
- `maximumDate`

QDateEdit Methods

- `setDate()`
- `setDisplayFormat()`

If an invalid date format is specified, the format will not be set.

1.3 Combo Boxes

Used to display a pop-up list. Uses the `QComboBox` class. Both texts and pixmaps can be displayed.

QComboBox Methods

- `setItemText()`
- `removeItem()`
- `clear()`
- `currentText()`
- `setCurrentIndex()`
- `count()`
- `setMaxCount()`
- `setEditable()`
- `addItem()`
- `addItems()`
- `itemText()`
- `currentIndex()`

QComboBox Signals

- `currentIndexChanged()`
- `activated()`
- `highlighted()`
- `editTextChanged()`

1.4 Tables

To display contents in a table, use a `QTableWidget`. The items displayed in a Table Widget are instances of the `QTableWidgetItem` class.

If a table uses a custom data model, use the `QTableView` class.

QTableWidget Methods

- `setRowCount()`
- `setColumnCount()`
- `rowCount()`
- `columnCount()`
- `clear()`
- `setItem()`

QTableWidgetItem Methods

- `setFont()`
- `setCheckState()`
- `checkState()`

1.5 Graphics

`QGraphicsView` is used for viewing and managing 2D graphical items. It displays a scene which is a container for graphic items. A scene is created using `QGraphicsScene`, and items using `QGraphicsItem`.

Add items to a scene using `addItem()`, and remove them using `removeItem()`. To add the scene to a view, n order to display it, use the `setScene()` function for `QGraphicsView`.

Unit 2

Menus and Toolbars

2.1 Menus

A menu bar has several menus, which contain several entries, which may include submenu entries. A menu entry can be checkable.

An application can have several toolbars, but only one menu bar.

Toolbar vs Menu Bar

A toolbar displays icons instead of text to represent the task it can perform.

When editing the text for a menu or submenu entry, if you add an ampersand character (&) before any character, that character will be underlined in the menu, and will work as a shortcut key.

2.1.1 Action Editor

Action

An operation that the user initiates through the user interface. Can be initiated by selecting a toolbar button, selecting a menu entry, or pressing a shortcut key.

In Qt, an action is an instance of the `QAction` class. These can be assigned to a menu or a toolbar button.

Action States

Normal The icon's image or pixmap when the user is not interacting with the action, and is in enabled mode.
Disabled The icon's pixmap when the action is in disabled mode.
Active The icon's pixmap when the action is enabled and the user is interacting with it.
Selected The icon's pixmap when the action is selected.

The signal used to connect a `QAction` to a slot is the `triggered()` signal.

Signal and Slot Syntax Differences

PyQt4
```
self.connect(self.ui.actionName,
QtCore.SIGNAL('signalName()'), self.slotFunction)
```

PyQt5
```
self.ui.actionName.signalName.connect(self.slotFunction)
```

2.2 Creating a Toolbar

Toolbars typically use icons instead of text to indicate actions a user can perform. These icons are normally added to a program using a resource file.

Resource Files

A resource file is used to add icons and other resources to an application. All resources added to a resource file need to have a prefix. A **prefix** is a section or category name given to a resource.

Toolbar buttons are usually created with actions.

2.3 Tab Widget

Used to display information in chunks. Enables one to split information into small sections, and display each section when the Tab button is selected.

Style Sheet Editor

Apply styles to widgets to customise its appearance. You can add resources, gradients, colours, and fonts.

The location of the tabs can be set using `tabPosition`, which can be North, South, East, or West.

2.4 Converting a Tab Widget

To change a Tab Widget to another type, right click and select “Morph Into”. It can be changed to a Tool Box, or a Stacked Widget.

Tool Box

Instance of the `QToolBox` class, provides a column of tabbed widget items, one above the next. Widgets of the current tab are displayed below it.

Stacked Widget

Instance of `QStackWidget`. Provides a stack of widgets, where only one widget is visible at a time.

By default, does not have a way to switch pages, so use another widget, such as a Combo Box or List Widget to set the page. Every widget in a Stacked Widget has an index number.

Unit 3

Multiple Documents and Layouts

3.1 Multiple Document Interface (MDI)

Applications that provide one document per main window are said to be SDI (single-document interface) applications. A multiple-document interface (MDI) consists of a main window containing a menu bar, toolbar, and a central `QWorkspace` widget. The central workspace displays and manages several child windows.

To implement an MDI, use an instance of `QMdiArea`. This widget provides an area where child windows (**subwindows**) are displayed. It arranges subwindows in a **cascade** or **tile** pattern. The subwindows are instances of `QMdiSubWindow`. They are rendered within a frame that has a title, and buttons to show, hide and maximise it.

QMdiArea Methods

- `subWindowList()`
- `windowOrder()`
  - `CreationOrder` (Default)
  - `StackingOrder`
  - `ActivationHistoryOrder`
- `activateNextSubWindow()`
- `activatePreviousSubWindow()`
- `cacadeSubWindows()`
- `tileSubWindows()`
- `closeAllSubWindows()`
- `setViewMode()`
  - `SubWindow View`: (Default) Displays subwindows with window frames. Represented by 0.
  - `Tabbed View`: Displays subwindows with tabs in a tab bar. Represented by 1.

3.2 Layouts

Layout

Used to arrange and manage the widgets that make up a user interface within its container.

Each widget has a recommended size defined in its `sizeHint` property. When windows are resized, widgets in a layout are resized to meet their size hint.

To avoid excessive spreading of widgets when the window size is increased, use **spacers**.

3.2.1 Horizontal Layout

Lays widgets next to each other in a row.

Group Box

Used to represent information that is related in some way. An instance of `QGroupBox`. Appears in a frame with a title.

Child widgets within a Group Box can be aligned and enabled or disabled collectively with a CheckBox.

QGroupBox Properties

checkable Display a checkbox in Group Box's title. Child widgets enabled only when checkbox is checked. By default, GroupBoxes are not checkable.

flat Space consumed by GroupBox is reduced.

QGroupBox Methods

- `isCheckable()`
- `isChecked()`
- `setChecked()`

Generates a `clicked()` signal when the checkbox is selected, or when its shortcut key is pressed.

3.2.2 Vertical Layout

Arrange widgets vertically, in a column one below another.

3.2.3 Other Layouts

Other layouts include GridLayout, and FormLayout.

Unit 4

Database Handling

4.1 Creating a Database

Database

A collection of information that is organised so that it can be easily accessed, managed, and updated.

Stores tables, indexes, foreign key constraints, primary key constraints, and other necessary components.

A database table consists of rows and columns. Each column contains a single pieces of information, and a row is a collection of columns that contains complete information of an object, item, or entity.

Syntax for Creating a Database

```
create database database_name;
```

MySQL Data Types

- **smallint, mediumint, int, bigint** Integer values
- **float** Single-precision floating-point values
- **double** Double-precision floating-point values
- **char** Fixed-length strings up to 255 characters
- **varchar** Variable-length strings up to 255 characters
- **tinyblob, blob, mediumblob, longblob** Large blocks of binary data
- **tinytext, text, mediumtext, longtext** Long blocks of text data
- **date** Date values
- **time** Time values or durations
- **datetime** Combined date and time values

Connecting to SQL server

```
import MySQLdb
conn = MySQLdb.connect(host="localhost", user="user_name",
                       passwd="password", db="db_name")
cursor = conn.cursor()
```

MySQLdb Methods

MySQLdb.connect()
Connect to database server. (Returns database connection, normally variable is called conn)Four parameters: host name, username, password, database name. Hostname specifies the location of the MySQL database server. For remote, specify the IP address. For local, use `localhost`.

conn.cursor()
Returns the cursor object from the connection. Used to traverse records from the result set.

cursor.execute()
Execute an SQL query.

conn.commit()
Apply modifications to a database table.

conn.rollback()
Cancels all modifications applied to the database table.

cursor.close()
Close cursor.

conn.close()
Disconnect database connection.

Some MySQL Commands

- `use database_name;`
- `show tables;`
- `describe table_name;`

SQL INSERT

Used to insert rows into a table.

```
INSERT INTO table_name (table_column1, table_column2)
VALUES (value1, value2);
```

4.1.1 Fetching Rows from a Table

Use the `SELECT` SQL statement, via `execute()` on a cursor object. A `resultset` object is created that contains the tows from the database that satisfy the query.

Result Set Cursor Methods

While a result set is created, you access it using the cursor. There are two possible methods:

`cursor.fetchone()` Fetches the next row in result set.
`cursor.fetchall()` Fetches the remaining rows in result set, or all, if none have been fetched.

You also use `SELECT` to search for a specific value in the table.

```
SELECT * FROM table_name where attribute=value;
```

You use UPDATE to change information, if it is found.

```
SELECT * from table_name where attribute=value;
UPDATE table_name set attribute=value2 where attribute=value;
```

You use DELETE to remove a row from a table.

```
SELECT * from table_name where attribute=value;
DELETE from table_name where attribute=value;
```

4.2 Database Maintenance Through GUI Programs

To integrate and access databases in PyQt, you use the `QSqlDatabase` class.

QSqlDatabase Methods

- **`addDatabase()`** Specify the database driver of the database to which you want to establish a connection.
  - **QDB2**: IBM DB2
  - **QMYSQL**: MySQL
  - **QOCI**: Oracle Call Interface
  - **QODBC**: ODBC (includes Microsoft SQL Server)
  - **QPSQL**: PostgreSQL
  - **QSQLITE**: SQLite version 3 or above
- **`setHostName()`**
- **`setDatabaseName()`**
- **`setUserName()`**
- **`setPassword()`**
- **`open()`** Opens the database connection using the current connection attributes. Returns either `True` or `False`, depending on whether the connection to the database is successfully established or not.
- **`lastError()`** Display error information that may occur while opening a connection with the database through the `open()` function.

4.2.1 Displaying Rows

Use a `QTableView` with a custom model.

Model

A mirror image of the database table that the user can use to navigate and edit if required.

The model used is an instance of the `QSqlTableModel` class.

QSqlTableModel

Provides a model that can be set to display information of a database table. Also makes it easy to navigate the model, and set editing strategy for the underlying database tables.

Methods

`setTable()` Specify database table for the model.
`setEditStrategy()` Applies the strategy for editing the database table.

- **OnFieldChange** All modifications made in the model applied immediately to the database table.
- **OnRowChange** All modifications made to a row applied to the database table when moving to a different row.
- **OnManualSubmit** All modifications cached in the model and applied to the database table when `submitAll()` is called. The modifications that have been cached can be cancelled or erased by called `revertAll()`

`select()`