Chapter 9 – Advanced Widgets

LCD Display

To display system clock time in LCD format, need to know how to do following:
Display LCD digits (`QLCDNumber` class)
Use Timers (`QTimer` class)
Fetch and measure system clock time (`QTime` class)

Table 9.1. Methods Provided by QLCDNumber

Method	Use
`setMode()`	Used to change the base of the numbers. Available options:
	`Hex` for displaying hexadecimal digits.
	`Dec` for displaying decimal digits.
	`Oct` for displaying octal digits.
	`Bin` for displaying binary digits.
`display()`	To display the specified content as LCD digits.
`value()`	Returns the numerical value displayed by the LCD Number widget.

Using Timers

To perform a repetitive task, you use a timer. A timer is an instance of the `QTimer` class. To use timers in an application, you just need to create an instance of `QTimer` and connect its `timeout()` signal to the slot that performs the desired task. A `timeout()` signal can be controlled by these methods:
`start(n)`: Initiates 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 the timer to generate a `timeout()` signal only once after `n` milliseconds.

Fetching and Measuring System Clock Time

To fetch the system clock time and measure a span of elapsed time, you use the `QTime` class. The time returned by this class is in 24-hour format.

Table 9.2. Methods Supported by QTime

Method	Description
`currentTime()`	Fetches the system's clock time and returns it as a `QTime` object.
`hour()`	Returns the number of hours.
`minute()`	Returns the number of minutes.
`seconds()`	Returns the number of seconds.
`msec()`	Returns the number of milliseconds.
`addSecs()`	Returns the time after adding a specified number of seconds.
`addMSecs()`	Returns the time after adding a specified number of milliseconds.
`secsTo()`	Returns the number of seconds between two times.
`msecsTo()`	Returns the number of milliseconds between two times.

Displaying Calendar

To display a monthly calendar, you use the Calendar widget, which is an instance of the `QCalendarWidget` class. By default, the Calendar widget displays the current month and year, the days are displayed in abbreviated form (Sun, Mon, Tue), and Saturdays and Sundays are marked in red. The week numbers are displayed, and the first column day is Sunday.

Table 9.4. Methods Provided by QCalendarWidget

Method	Description
`selectedDate()`	Returns the currently selected date.
`monthShown()`	Returns the currently displayed month.
`yearShown()`	Returns the currently displayed year.
`setFirstDayOfWeek()`	Used to set the day in the first column.
`selectionChanged()`	Emitted when the user selects a date other than the currently selected date. The date can be selected using the mouse or keyboard.

QDate Class

For working with dates, you use an instance of the `QDate` class. A `QDate` object contains a calendar date with the year, month, and day in the Gregorian calendar.

Table 9.5. Methods Provided by the QDate Class

Method	Use
currentDate()	Returns the system date as a QDate object.
setDate()	Sets a date by specifying the year, month, and day.
year()	Returns the year from the specified date object.
month()	Returns the month from the specified date object.
day()	Returns the day from the specified date object.
dayOfWeek()	Returns the day of the week from the specified date object.
addDays()	Adds the specified number of days to the specified date and returns new date.
addMonths()	Adds the specified number of months to the specified date and returns new date.
addYears()	Adds the specified number of years to the specified date and returns new date.
daysTo()	Returns the number of days between two dates.
daysInMonth()	Returns the number of days in the specified month.
daysInYear()	Returns the number of days in the specified year.
isLeapYear()	Returns true if the specified date is in a leap year.
toPyDate()	Returns the date as a string. The format parameter determines the format of the result string.

Using the Date Edit Widget

For displaying and editing dates, you use the Date Edit widget, which is an instance of the QDateEdit class. Properties used to configure the Date Edit widget:

minimumDate: This property is used to define the minimum date that can be set to the widget.

maximumDate: This property is used to define the maximum date that can be set to the widget.

Table 9.6. Methods Provided by the QDateEdit Class

Method	Description
setDate()	Used to set the date to be displayed in the widget.
setDisplayFormat()	Used to specify the string format that you want to apply to the date displayed in the Date Edit widget. Formats with their outputs are these:

Using Combo Box

To display a pop-up list (also known as a Combo Box), you use the `QComboBox` class.

Table 9.7. Methods Provided by QComboBox

Method	Use
`setItemText()`	Used to change the item in the Combo Box.
`removeItem()`	Used to remove an item.
`clear()`	Used to remove all items.
`currentText()`	Returns the text of the current item.
`setCurrentIndex()`	Used to set the current item.
`count()`	Returns the number of items in the Combo Box.
`setMaxCount()`	Used to set the maximum number of items.
`setEditable()`	Used to allow editing in the Combo Box.
`addItem()`	Used to add an item to the Combo Box with specified text. The item is appended to the list.
`addItems()`	Used to add each of the strings in the text to the Combo Box. Each item is appended to the list.
`itemText()`	Returns the text at the specified index in the Combo Box.
`currentIndex()`	Returns the index of the current item in the Combo Box. An empty Combo Box or a Combo Box with no current item selected returns −1 as the index.

Table 9.8. Signals Generated by QComboBox

Signal	Description
`currentIndexChanged()`	The signal is emitted if the index of the Combo Box is changed (through user interaction or via program), and a new item is selected.
`activated()`	The signal is emitted when the index is changed by user interaction.
`highlighted()`	The signal is emitted when the user highlights an item in the Combo Box.
`editTextChanged()`	The signal is emitted when the text of an editable Combo Box is changed.

Displaying a Table

To display contents in a row and column format, you use a Table widget, which is an instance of the `QTableWidget` class.

Table 9.9. Methods Provided by QTableWidget

Method	Use
`setRowCount()`	Used to specify the number of rows in the Table widget.
`setColumnCount()`	Used to specify the number of columns in the Table widget.
`rowCount()`	Returns the number of rows in the table.
`columnCount()`	Returns the number of columns in the table.
`clear()`	Clears the table.
`setItem()`	Sets the item for a given row and column of the table.

Displaying Items in the Table

The items displayed in the Table widget are instances of the `QTableWidgetItem` class.

Table 9.10. Methods Provided by QTableWidgetItem

Method	Use
`setFont()`	Used to set the font for the text label of the Table Item.
`setCheckState()`	Used to check or uncheck a Table Item.
`checkState()`	Used to determine if the Table Item is checked or not.

Displaying Web Pages

To view and edit web pages, you use a QWebView widget, that represents an instance of `QWebView class`. It is the main widget component of the `QtWebKit` web-browsing module.

Table 9.11. QWebView Methods for Displaying Web Pages

Method	Use
`load()`	Loads the specified URL and displays it through QWebView widget. The view remains unchanged until enough data is downloaded to display.
`setUrl()`	Same as `load()` method.
`setHtml()`	To view HTML content.

Table 9.12. Signals Generated by QWebView While Loading Web Pages

Signal	Description
`loadStarted()`	Emitted when the view begins loading.
`loadProgress()`	Emitted whenever an element of Web View completes loading, such as an embedded image, video, or script.
`loadFinished()`	Emitted when the view is loaded completely.

Chapter 10 – Menus and Toolbars

Dock Widget

A Dock widget is created with the `QDockWidget` class. A Dock widget can be used to create detachable tool palettes or widget panels. They can be closed or docked in the Dock area around the central widget inside `QMainWindow` or floated as a top-level window on the desktop. Allowable dock areas are `LeftDockWidgetArea`,

RightDockWidgetArea, TopDockWidgetArea, and BottomDockWidgetArea, where TopDockWidgetArea is below the toolbar.

Table 10.2. Properties of a Dock Widget

Property	Description
DockWidgetClosable	If selected, the Dock widget can be closed.
DockWidgetMovable	If selected, the Dock widget can be moved between dock areas.
DockWidgetFloatable	If selected, the Dock widget can be detached from the main window and floated as an independent window.
DockWidgetVerticalTitleBar	If selected, the Dock widget displays a vertical title bar on its left side.
AllDockWidgetFeatures	If selected, automatically selects the DockWidgetClosable, DockWidgetMovable, and DockWidgetFloatable properties, allowing the Dock widget to be closed, moved, or floated.
NoDockWidgetFeatures	If selected, the Dock widget cannot be closed, moved, or floated.

Converting a Tab Widget

Tool Box: A Tool Box is an instance of the QToolBox class and provides a column of tabbed widget items, one above the next.

Stacked Widget: A Stacked widget is an instance of QStackedWidget and provides a stack of widgets where only one widget is visible at a time.

Chapter 11 – Multiple Documents and Layouts

Multiple-Document Interface

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, a toolbar, and a central QWorkspace widget.

To implement an MDI, you will use an MdiArea widget, which is an instance of the QMdiArea class. The MdiArea widget provides an area where child windows (also called *subwindows*) are displayed. It arranges subwindows in a cascade or tile pattern. The subwindows are instances of QMdiSubWindow.

Table 11.1. Methods Provided by QMdiArea

Method	Use
`subWindowList()`	Returns a list of all subwindows in the MDI area arranged in the order set through the `WindowOrder()` function.
`WindowOrder()`	Used to specify the criteria for ordering the list of child windows returned by `subWindowList()`. Following are the available options: `CreationOrder`: The windows are returned in the order of their creation. This is the default order. `StackingOrder`: The windows are returned in the order in which they are stacked, with the topmost window last in the list. `ActivationHistoryOrder`: The windows are returned in the order in which they were activated.
`activateNextSubWindow()`	Sets the focus to the next window in the list of child windows. The current window order determines the next window to be activated.
`activatePreviousSubWindow()`	Sets the keyboard focus to the previous window in the list of child windows. The current window order determines the previous window to be activated.
`cascadeSubWindows()`	Arranges subwindows in cascade fashion.
`tileSubWindows()`	Arranges subwindows in tile fashion.
`closeAllSubWindows()`	Closes all subwindows.
`setViewMode()`	Sets the view mode of the MDI area. The subwindows can be viewed in two view modes, SubWindow view and Tabbed view: SubWindow view: Displays subwindows with window frames (default). You can see the content of more than one subwindow if arranged in tile fashion. It is also represented by a constant value 0. Tabbed view: Displays subwindows with tabs in a tab bar. Only one subwindow's content can be seen at a time. It is also represented by a constant value 1.

Table 12.1. Data Types in MySQL

Data Type	Stores
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

QSqlDatabase Class

To integrate and access databases in PyQt, you use the `QSqlDatabase` class. To represent connection to a database, an instance of `QSqlDatabase` is used.

Table 12.2. Methods of the QSqlDatabase Class

Method	Use
addDatabase()	Used to specify the database driver of the database to which you want to establish connection. It is through the database drivers that the database is accessed. **Driver types:** QDB2: IBM DB2 Driver QMYSQL: MySQL Driver QOCI: Oracle Call Interface Driver QODBC: ODBC Driver (includes Microsoft SQL Server) QPSQL: PostgreSQL Driver QSQLITE: SQLite version 3 or above
setHostName()	Used to specify the hostname.
setDatabaseName()	Used to specify the name of the database that you want to work with.
setUserName()	Used to specify the name of the authorized user through whom you want to access the database.
setPassword()	Used to specify the password of the authorized user to access the database.
open()	Opens the database connection using the current connection attributes. The method returns a Boolean true or false value, depending on whether the connection to the database is successfully established or not.
lastError()	Used to display error information that may occur while opening with database through open() function.

Displaying Rows

To display the rows fetched from the database table, you will use a Table View widget. To create a model, you need to create an instance of the `QSqlTableModel` class.

Table 12.3. Methods of QSqlTableModel

Method	Use
`setTable()`	Used to specify the database table you want the model to work with.
`setEditStrategy()`	Applies the strategy for editing the database table. The available strategies are these: `OnFieldChange`: All modifications made in the model will be applied immediately to the database table. `OnRowChange`: All modifications made to a row will be applied to the database table on moving to a different row. `OnManualSubmit`: All modifications will be cached in the model and applied to the database table when `submitAll()` is called. Also, modifications that are cached can be cancelled or erased without applying to the database by calling `revertAll()`.
`select()`	Used to populate the model with the information of the database table specified with `setTable()`.

`QSqlQueryModel` class: Provides a read-only model based on the specified SQL query.

`setQuery()`: Used to specify the SQL query.

`record(int)`: Used to access individual records (rows) from the specified database table.

`record.value("column_name")`: Used to retrieve the value of the specified column of the current row of the database table.

`submit()`: Submits the currently edited row; applies the modifications to the underlying database table if the edit strategy is set to `OnRowChange` or `OnFieldChange`.

Recall that if the edit strategy is set to `OnRowChange`, all the modifications done to a row in the model will be applied to the database table on moving on to a different row. If the edit strategy is set to `OnFieldChange`, all modifications to the model will be applied to the database table. If the edit strategy is set to `OnManualSubmit`, all modifications will be cached in the model and applied to the database table when `submitAll()` is called. Also, all cached modifications will be cancelled without being applied to the database if `revertAll()` is called.

`submitAll()`: Used to submit all pending changes to the database table if the edit strategy is set to `OnManualSubmit`. The method returns true if the modifications are successfully applied to the database table; otherwise it returns false.

`lastError()`: Used to display detailed error information.

`revertAll()`: Used to cancel all the pending editing of the current database table if the model's editing strategy is set to `OnManualSubmit`.

`revert()`: Used to cancel the editing of the current row if the model's strategy is set to `OnRowChange`.

`insertRow()`: Inserts an empty row after the specified position in an open database table. If the specified position is a negative value, the row will be inserted at the end.

`removeRow()`: Removes the row at the specified index from an open database table. You need to call `submitAll()to apply the changes to the database table if the edit strategy is set to OnManualSubmit.`
`setFilter()`: Used to specify the filter condition for the database table. If the model is already populated with rows of the database table, the model repopulates the model with the filtered rows.