Qt Framework Cheatsheet

A quick reference guide to the Qt framework, covering essential classes, functions, and concepts for developing cross-platform applications.

Core Concepts

Signals and Slots

Qt's signal and slot mechanism facilitates communication between objects. A signal is emitted when a particular event occurs, and a slot is a function that is called in response to a signal.

- `signals:` keyword declares signals within a class.
- `slots:` keyword declares slots within a class.
- `connect()` function establishes the connection between a signal and a slot.

Example:

```
// Define a signal
signals:
    void buttonClicked();

// Define a slot
public slots:
    void handleButtonClicked();

// Connect the signal and slot
connect(button, &QPushButton::clicked, this,
&MyClass::handleButtonClicked);
```

Qt's signals and slots provide a type-safe way to implement callbacks, reducing the risk of runtime errors.

Meta-Object System

The Meta-Object System (MetaObject) provides information about the objects at runtime.

- `Q_OBJECT` macro is mandatory in any class that uses signals and slots or other meta-object features.
- `QMetaObject` class provides access to meta-information about a class.
- Allows for dynamic property access and invocation of methods.

Example:

```
class MyClass : public QObject {
    Q_OBJECT
public:
    MyClass(QObject *parent = nullptr) :
QObject(parent) {}
};
```

Object Model

Qt's object model is based on a hierarchical object tree. `QObject` is the base class for all Qt objects that support object hierarchies, signals, and slots.

- Parent-child relationships manage object lifetimes.
- Deleting a parent object will also delete its children.
- `QObject::parent()` returns the parent of an object.

Example:

```
QObject *parent = new QObject();
QObject *child = new QObject(parent);

// When parent is deleted, child is also
deleted.
delete parent;
```

Common Classes

QWidgets

`QPushButton`	A button that the user can click.
`QLabel`	Displays text or an image.
`QLineEdit`	A single-line text editor.
`QTextEdit`	A multi-line text editor with rich text support.
`QComboBox`	A combo box widget (drop-down list).
`QCheckBox`	A checkbox widget.
`QSlider`	A horizontal or vertical slider.
`QProgressBar`	Displays the progress of a task.

Layout Managers

`QVBoxLayout`	Arranges widgets vertically.
`QHBoxLayout`	Arranges widgets horizontally.
`QGridLayout`	Arranges widgets in a grid.
`QFormLayout`	Arranges widgets in a two-column form.

Data Handling

Containers

`QList<T>`	A dynamically-sized array.
`QVector<T>`	Provides contiguous storage.
`QMap<Key, Value>`	A key-value storage.
`QSet<T>`	Stores unique values.
`QStringList`	A list of strings.

String Handling

`QString` is Qt's string class.

- It provides support for Unicode.
- It is implicitly shared.
- It offers many manipulation methods.

Common QString Methods:

- `QString::append()`
- `QString::prepend()`
- `QString::toLower()`
- `QString::toUpper()`
- `QString::trimmed()`
- `QString::split()`

Networking

Network Classes

`QTcpSocket`	Provides a TCP socket.
`QTcpServer`	Listens for incoming TCP connections.
`QUdpSocket`	Provides a UDP socket.
`QNetworkRequest`	Represents a network request.
`QNetworkAccessManager`	Manages network requests.

HTTP Operations

Qt simplifies HTTP operations using `QNetworkAccessManager` and related classes.

- `get()`
- `post()`
- `put()`
- `deleteResource()`

Example:

```
QNetworkAccessManager *manager = new QNetworkAccessManager(this);
connect(manager, &QNetworkAccessManager::finished, this, [=](QNetworkReply
*reply) {
    if (reply->error()) {
        qDebug() << "Error:" << reply->errorString();
    } else {
        qDebug() << reply->readAll();
    }
    reply->deleteLater();
});

manager->get(QNetworkRequest(QUrl("http://example.com")));
```