Object Oriented Programming - Lecture 9

Diana Borza - diana.borza@ubbcluj.ro

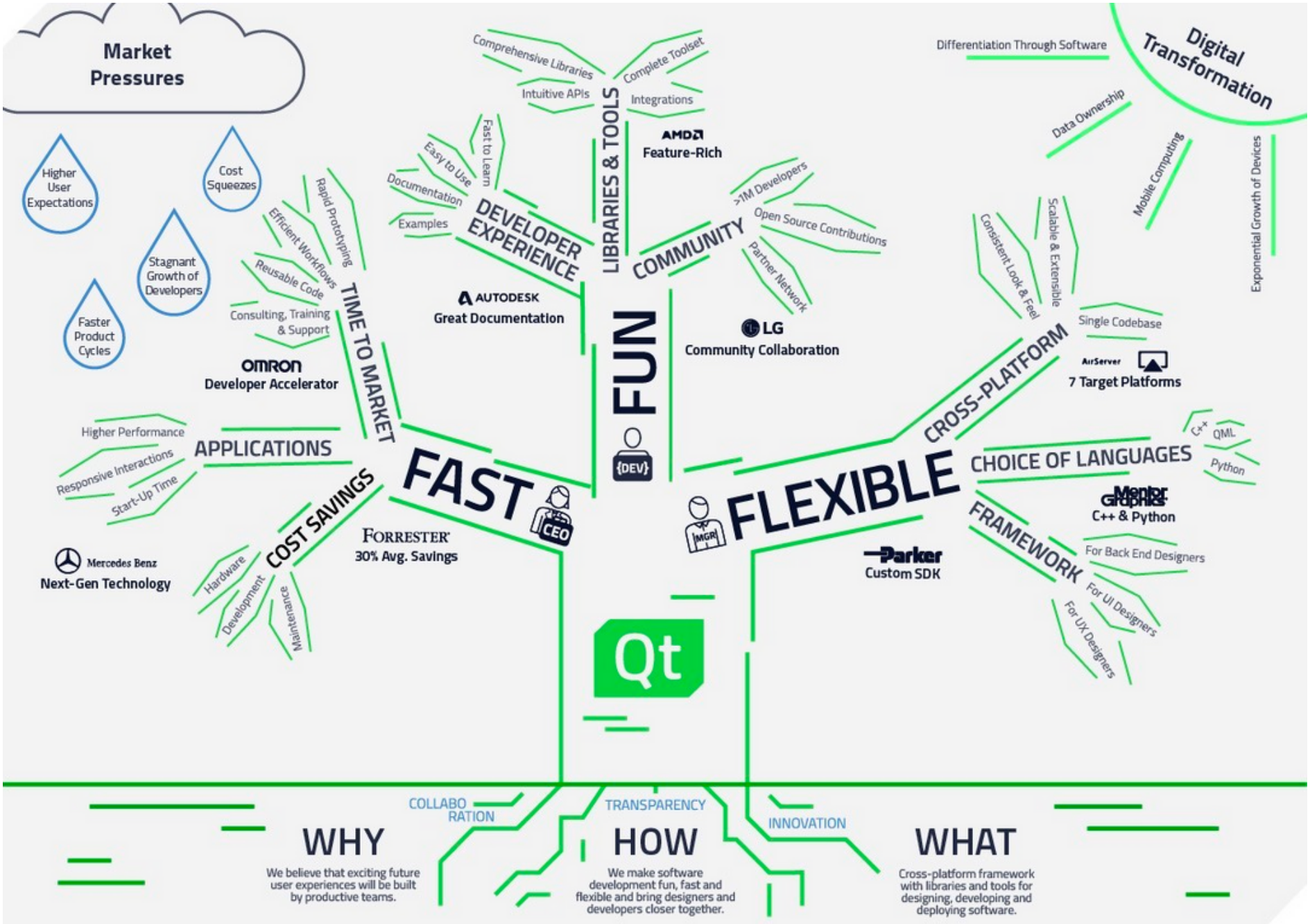
May 15, 2022

Content

- Qt programming framework
- Qt GUI components
- Layouts
- Common pattern to build a GUI in Qt

Qt programming framework

- pronounced "cute" :)
- complete software development framework, it comprises a set of highly intuitive and *modularized* C++ classes;
- a toolkit for creating graphical user interfaces
- it is **cross-platform** - you can use the same code to run your application on various software and hardware platforms such as Linux, Windows, macOS, Android or embedded systems.
- Language bindings are available in C#, Java, Python(PyQt, Qt for Python), Ada, Pascal, Perl, PHP(PHP-Qt), Ruby(RubyQt).
- Qt is available under GPL v3, LGPL v2 and commercial license.


Market Pressures
Higher User Expectations
Cost Squeezes
Stagnant Growth of Developers
Faster Product Cycles
Digital Transformation
Differentiation Through Software
Data Ownership
Mobile Computing
Exponential Growth of Devices
LIBRARIES & TOOLS
Comprehensive Libraries
Complete Toolset
Intuitive APIs
Integrations
AMD
Feature-Rich
DEVELOPER EXPERIENCE
Fast to Learn
Easy to Use
Documentation
Examples
AUTODESK
Great Documentation
COMMUNITY
>1M Developers
Open Source Contributions
Partner Network
LG
Community Collaboration
TIME TO MARKET
Rapid Prototyping
Efficient Workflows
Reusable Code
Consulting, Training & Support
OMRON
Developer Accelerator
FUN
{DEV}
CROSS-PLATFORM
Consistent Look & Feel
Scalable & Extensible
Single Codebase
AirServer
7 Target Platforms
APPLICATIONS
Higher Performance
Responsive Interactions
Start-Up Time
COST SAVINGS
Hardware
Development
Maintenance
FORRESTER
30% Avg. Savings
FAST
CEO
FLEXIBLE
MGR
CHOICE OF LANGUAGES
C++
QML
Python
Mentor Graphics
C++ & Python
FRAMEWORK
For Back End Designers
For UI Designers
For UX Designers
Parker
Custom SDK
Mercedes Benz
Next-Gen Technology
Qt
COLLABO RATION
TRANSPARENCY
INNOVATION
WHY
We believe that exciting future user experiences will be built by productive teams.
HOW
We make software development fun, fast and flexible and bring designers and developers closer together.
WHAT
Cross-platform framework with libraries and tools for designing, developing and deploying software.

Qt modules

Qt for Application Development, Commercial Licensing

Applications written in Qt

https://en.wikipedia.org/wiki/Qt_(software)

- Ableton Live
- Adobe Photoshop Album
- Adobe Photoshop Elements
- Autodesk Maya
- Autodesk 3ds Max
- Bitcoin Core
- Bitcoin ABC
- CryEngine V editor
- DaVinci Resolve
- Google Earth
- Mathematica
- Moonlight Stream
- Stellarium
- Subsurface
- Teamviewer
- Telegram
- VirtualBox OS virtualization software
- VLC media player
- XnView MP

Organizations using Qt

- AMD
- Blizzard Entertainment
- BMW, Crytek, Daimler
- European Space Agency
- DreamWorks
- Huawei, Microsoft
- LG, Panasonic, Philips
- Lucasfilm, Luxoft
- Robert Bosch GmbH
- Samsung, Siemens
- Tesla, Volvo
- German Air Traffic Control
- Walt Disney Animation Studios

QApplication, QCoreApplication, QGUIApplication I

- Qt applications use an **event loop**.
- An **event loop** is an *infinite* loop that works in the background of your application and handles events incoming from your OS (mouse events, timers, network events, paint events, hardware events etc.), as well as internal communication (signals and slots).
- QCoreApplication - base class. Use it in command line applications.
- QGuiApplication - base class + GUI capabilities. Use it in QML applications.
- QApplication - base class + GUI + support for widgets. Use it in QtWidgets applications.

Qt event loop

- In Qt, the event loop starts when you call the exec() function on a QCoreApplication class or subclass;
- When a Qt application is running, the event loop waits for user input, then events are generated and sent to the widgets of the application.
- The loop is terminated when any of the functions exit() or quit() is called.

QApplication I

- QApplication's main areas of responsibility are:
  - It initializes the application with the user's desktop settings;
  - It performs event handling, i.e. it receives events from the underlying window system and dispatches them to the relevant widgets;
  - It parses common command line arguments and sets its internal state accordingly;
  - It defines the application's look and feel;
  - It provides localization of strings that are visible to the user;
  - It keeps track about the application's windows;
  - It manages the application's mouse cursor handling.
- QApplication **MUST** be created before any other widget objects.

QApplication II

- The QApplication class manages the GUI application's control flow and main settings.
- QApplication contains the main event loop, where all events from the window system and other sources are processed and dispatched.
- There is there is precisely one QApplication object, no matter whether the application has 0, 1, 2 or more windows QApplication object, no matter how many windows the application has.
- The QApplication object is accessible through the instance() function.
- Useful functions from QApplication:
  - - applicationDirPath() - returns the directory that contains the application executable.
  - - applicationFilePath() - returns the file path of the application executable.

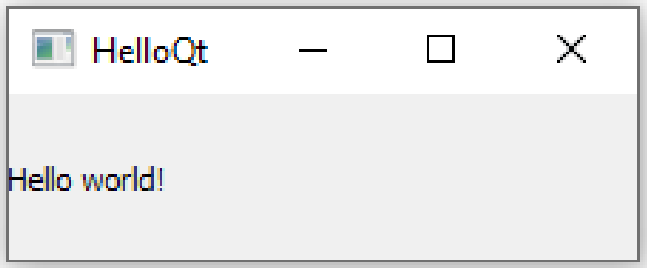

A User Interface Is Like
A Joke. If You Have To
Explain It, It's Not
That Good.
Martin LeBlanc, Iconfinder

Hello world! The cute version :)

```cpp
#include <QLabel>
#include <QApplication>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv); // creating a QApplication, that handles the event loop.
    // !!! The QApplication object must be created before any widget Object !!!
    QLabel helloLabel{"Hello world!"}; // creating a new label, which will contain the string "Hello world!"
    helloLabel.show(); // show the label

    return a.exec(); // start the application's event loop
}
```



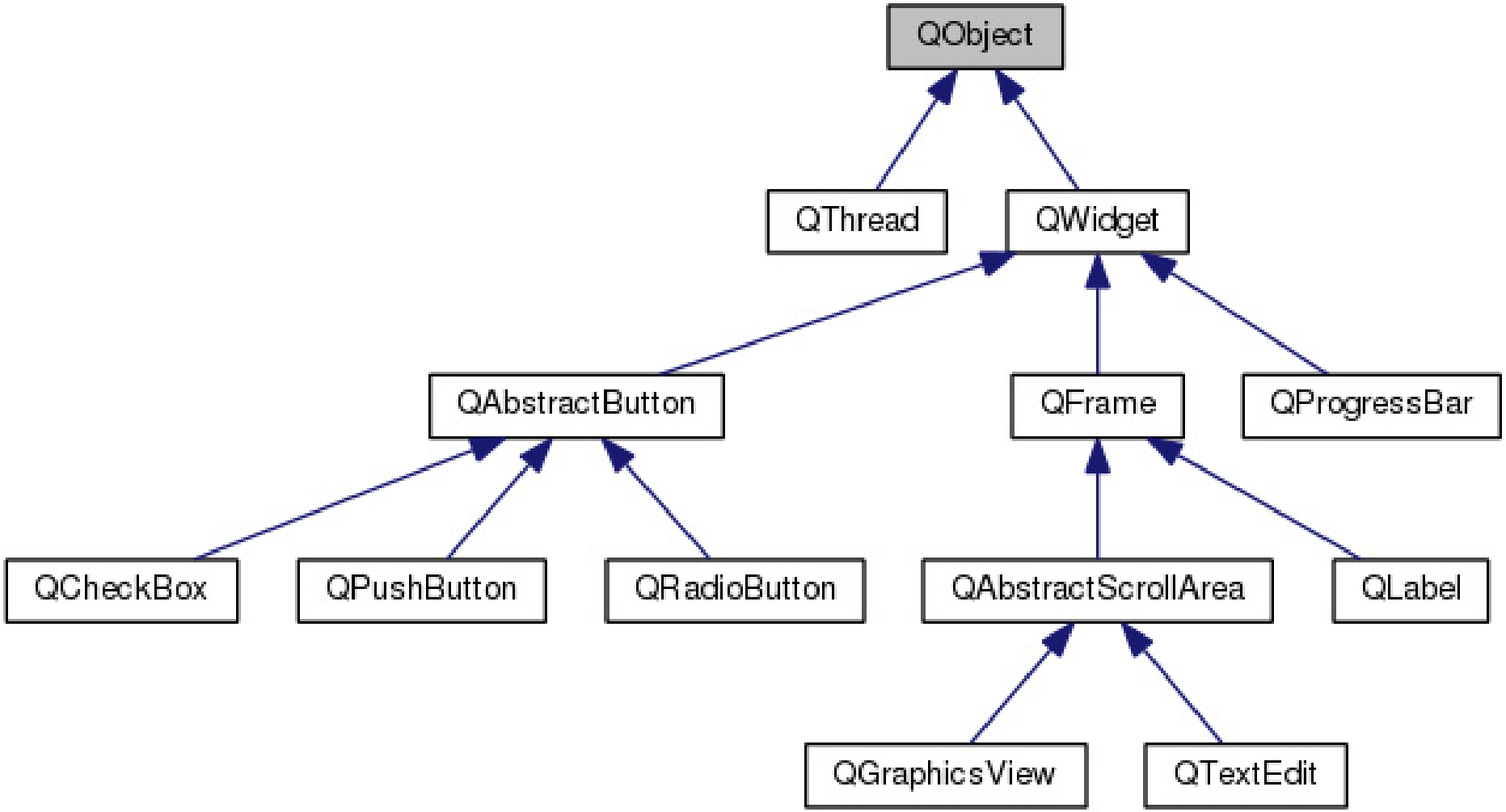
Widgets I

- A widget is the basic building block for graphical user interface (GUI) applications built with Qt. E.g.: buttons, labels, list views, etc.
- A GUI component (widget) can be placed on the user interface window or can be displayed as an independent window.
- A widget that is not embedded in a parent widget is called a *window*.
- Widows provide the screen space upon which the user interface is built.

Widgets II

- Windows visually separate applications from each other and usually provide a window decoration (show a title bar, allows the user to resize, position, etc).
- The Widgets module in Qt uses inheritance.
- All widgets inherit from QWidget, which is derived from QObject.

Widgets III


QObject
QThread
QWidget
QAbstractButton
QFrame
QProgressBar
QCheckBox
QPushButton
QRadioButton
QAbstractScrollArea
QLabel
QGraphicsView
QTextEdit

Parenting system in Qt I

- Widgets use the Qt parenting system:
  - Any object that inherits from QObject can have a parent and children.
  - When an object is destroyed, all of its children are destroyed as well.
  - All QObject have methods that allow searching the object's children : the children() method from QObject.
  - Child widgets in a QWidget automatically appear inside the parent widget.

Parenting system in Qt II

- When QObject are created on the heap (i.e., created with new), a tree can be constructed from them in any order, and later, the objects in the tree can be destroyed in any order.
- When any QObject in the tree is deleted, if the object has a parent, the destructor automatically removes the object from its parent.
- If the object has children, the destructor automatically deletes each child.
- No QObject is deleted twice, regardless of the order of destruction.

QLabel

- QLabel is used for displaying text or an image.
- No user interaction functionality is provided.
- The visual appearance of the label can be configured in various ways, and it can be used for specifying a focus mnemonic key for another widget.
- You can set a "buddy" on a QLabel: when the user presses the shortcut key indicated by this label, the keyboard focus is transferred to the label's buddy widget.
- Is defined in the header <QLabel>.

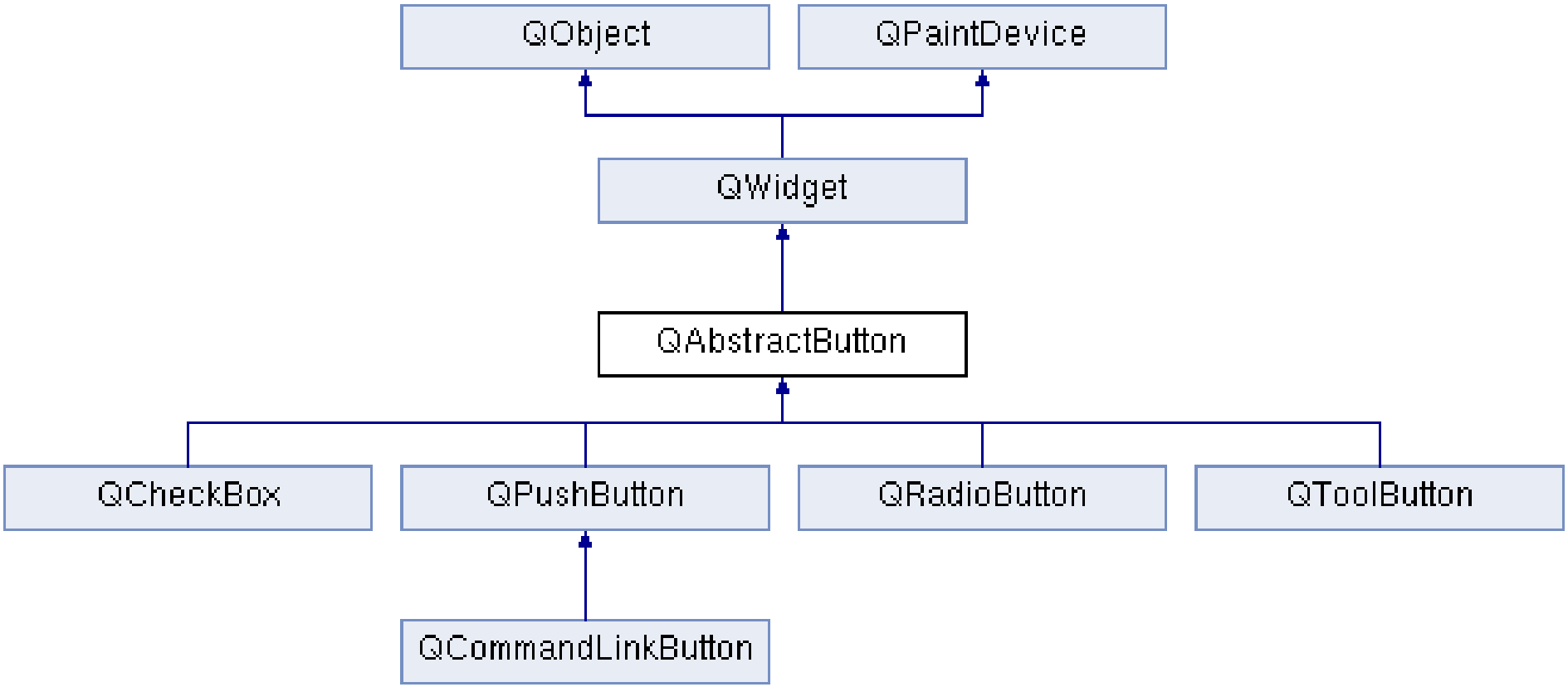
QLabel buddy - mnemonics in Qt

- Applications invariably need to define accelerator keys for actions → shortcuts.
- Microsoft's recommended choice: an emboldened letter plus the ALT key.
- In Qt, the buddy mechanism is only available for QLabels that contain text in which one character is prefixed with an ampersand, '&'.
- This character is set as the shortcut key.
- By pressing ALT + the shortcut key, the QLabels's buddy widget gets keyboard focus.

QLineEdit

- QLineEdit is an one-line text editor.
- A line edit allows the user to enter and edit a single line of plain text with a useful collection of editing functions, including undo and redo, cut and paste, and drag and drop.
- It is defined in the header <QLineEdit>
- A related class is QTextEdit; it is a widget that is used to *edit* and *display* both plain and rich text.

Buttons in Qt I



Buttons in Qt II

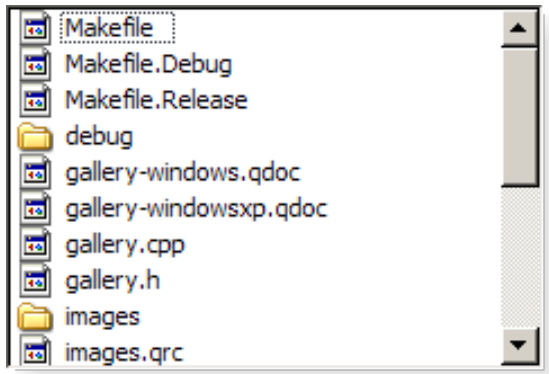
- The QAbstractButton class is the abstract base class of button widgets, providing functionality common to buttons. Its subclasses handle user actions, and specify how the button is drawn.
- QAbstractButton provides support for both *push buttons* (QPushButton, QToolButton) and *checkable* (QRadioButton, QCheckBox) (toggle) buttons.
- Any button can display a label containing text and an icon.

QPushButton

- The QPushButton widget provides a command button.
- Push (click) a button to command the computer to perform some action.
- Push buttons display a textual label, and optionally a small icon. A shortcut key can be specified by preceding the preferred character with an ampersand.
- Is defined in the header $<$ *QPushButton* $>$.

QListWidget

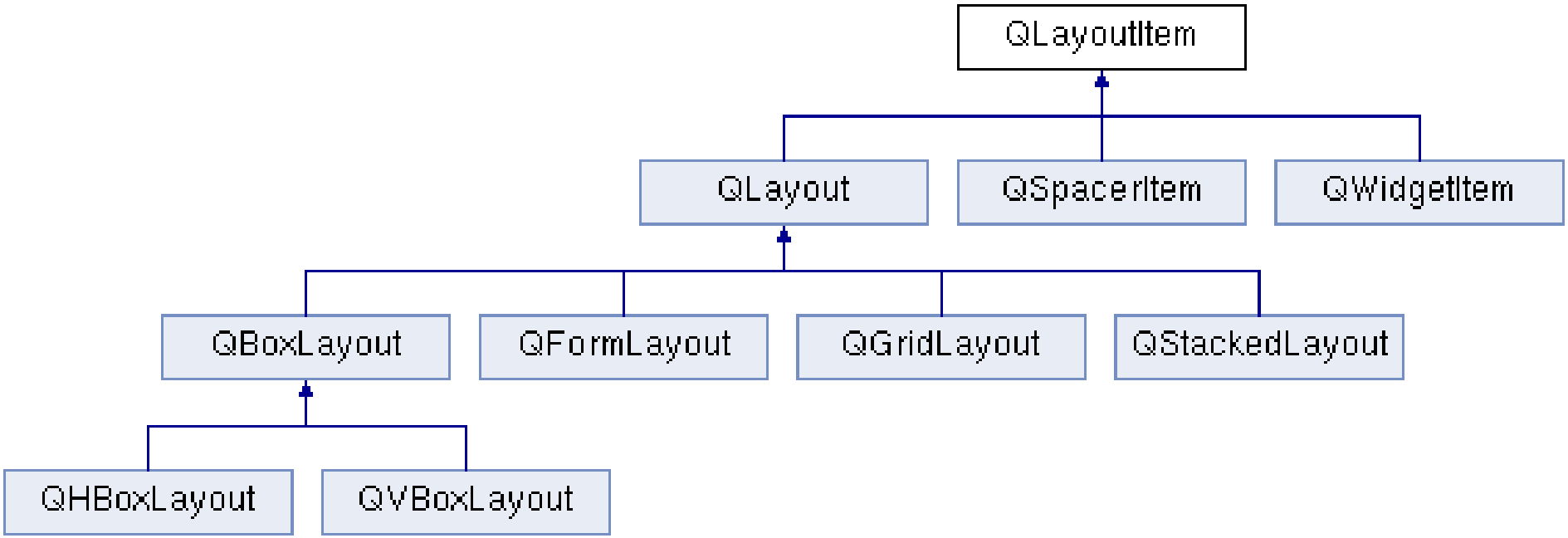
- The QListWidget widget provides an item-based list widget.
- The widget presents a list of items to the user. QListWidget uses an internal model to manage each item in the list (QListWidgetItem).
- There are two ways to add items to the list:
  1. they can be constructed with the list widget as their parent widget;
  2. they can be constructed with no parent widget and added to the list later.
- Is defined in the header <QListWidget>.



Layout management

- The Qt layout system provides a way to automatically arrange child widgets within a widget to ensure that they make good use of the available space.
- Qt includes a set of layout management classes that are used to describe how widgets are laid out in an application's user interface.
- These layouts automatically position and resize widgets when the amount of space available for them changes, ensuring that they are consistently arranged and that the user interface as a whole remains usable.

Qt Layouts I


QLayoutItem
QLayout
QSpacerItem
QWidgetItem
QBoxLayout
QFormLayout
QGridLayout
QStackedLayout
QHBoxLayout
QVBoxLayout

Qt Layouts II

- When a layout is set on a widget, it takes charge of the following tasks:
  - Positioning of child widgets;
  - Sensible default sizes for windows;
  - Sensible minimum sizes for windows;
  - Resize handling
  - Automatic updates when contents change:
    - Font size, text or other contents of child widgets.
    - Hiding or showing a child widget.
    - Removal of child widgets.

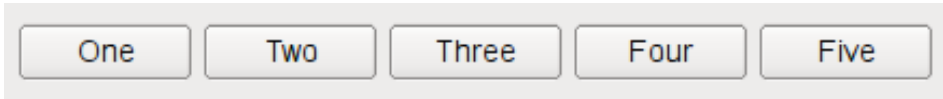
QHBoxLayout - horizontal layout

- QHBoxLayout is used to line up widgets **horizontally**.

```
QWidget *window = new QWidget;
QPushButton *button1 = new QPushButton("One");
QPushButton *button2 = new QPushButton("Two");
QPushButton *button3 = new QPushButton("Three");
QPushButton *button4 = new QPushButton("Four");
QPushButton *button5 = new QPushButton("Five");

QHBoxLayout *layout = new QHBoxLayout;
layout->addWidget(button1);
layout->addWidget(button2);
layout->addWidget(button3);
layout->addWidget(button4);
layout->addWidget(button5);

window->setLayout(layout);
window->show();
```



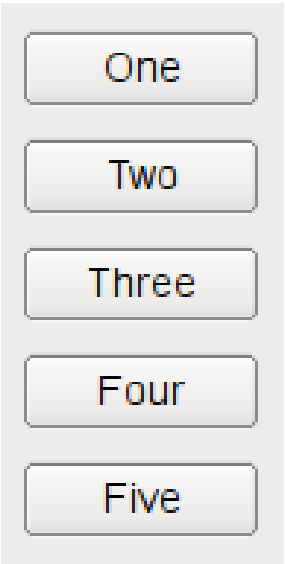
QVBoxLayout - vertical layout

- QVBoxLayout is used to line up widgets **vertically**.

```
QWidget *window = new QWidget;
QPushButton *button1 = new QPushButton("One");
QPushButton *button2 = new QPushButton("Two");
QPushButton *button3 = new QPushButton("Three");
QPushButton *button4 = new QPushButton("Four");
QPushButton *button5 = new QPushButton("Five");

QVBoxLayout *layout = new QVBoxLayout;
layout->addWidget(button1);
layout->addWidget(button2);
layout->addWidget(button3);
layout->addWidget(button4);
layout->addWidget(button5);

window->setLayout(layout);
window->show();
```



QFormLayout

- QFormLayout manages forms of input widgets and their associated labels. It lays out its children in a two-column form.
- The left column consists of labels and the right column consists of "field" widgets (line editors, spin boxes, etc.).

```
QWidget *window = new QWidget;
QPushButton *button1 = new QPushButton("One");
QLineEdit *lineEdit1 = new QLineEdit();
QPushButton *button2 = new QPushButton("Two");
QLineEdit *lineEdit2 = new QLineEdit();
QPushButton *button3 = new QPushButton("Three");
QLineEdit *lineEdit3 = new QLineEdit();

QFormLayout *layout = new QFormLayout;
layout->addRow(button1, lineEdit1);
layout->addRow(button2, lineEdit2);
layout->addRow(button3, lineEdit3);

window->setLayout(layout);
window->show();
```

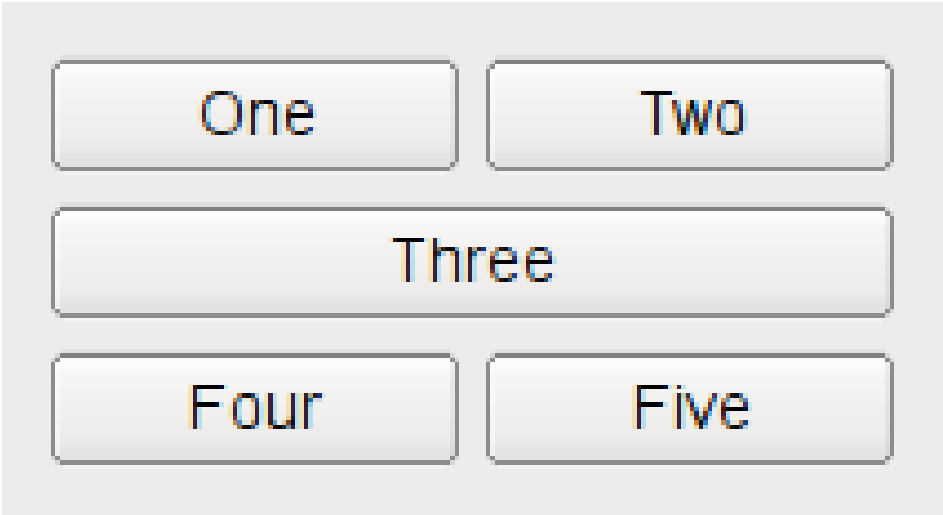
QGridLayout

- QGridLayout lays out widgets in a grid. QGridLayout takes the space made available to it , divides it up into rows and columns, and puts each widget it manages into the correct cell.
- Normally, each managed widget is put into a cell of its own It is also possible for a widget to occupy multiple cells (row and column spans).

```
QWidget *window = new QWidget;
QPushButton *button1 = new QPushButton("One");
QPushButton *button2 = new QPushButton("Two");
QPushButton *button3 = new QPushButton("Three");
QPushButton *button4 = new QPushButton("Four");
QPushButton *button5 = new QPushButton("Five");

QGridLayout *layout = new QGridLayout;
layout->addWidget(button1, 0, 0);
layout->addWidget(button2, 0, 1);
layout->addWidget(button3, 1, 0, 1, 2);
layout->addWidget(button4, 2, 0);
layout->addWidget(button5, 2, 1);

window->setLayout(layout);
window->show();
```



Why use layouts?

- They provide a consistent behavior across different screen sizes and styles.
- Layout managers handle resize operations.
- They automatically adapt to different fonts and platforms.
- If the user changes the systems font settings, the applications forms will respond immediately, resizing themselves if necessary.
- They automatically adapt to different languages. If the applications user interface is translated to other languages, the layout classes take into consideration the widgets translated contents to avoid text truncation.
- If a widget is added to or removed from a layout, the layout will automatically adapt to the new situation (the same thing happens when applying the show() or hide() functions for a widget).
- Layouts can be combined together.

Absolute positioning

- As an alternative to layouts, you can use absolute positioning: i.e. "manually" setting the position of a widget.
- An absolute position can be specified for a widget using the function QWidget::setGeometry(x, y, width, height), which builds a rectangle using the given parameters (x and y positions, width and height).
- Absolute positioning disadvantages
  - If the window is resized, the widgets with absolute positions remain unchanged.
  - Some text may be truncated (large font or change in the labels).
  - The positions and sizes must be calculated manually (error-prone, hard to maintain).

Resources and resource files (.qrc)

- Qt resource system is a platform-independent mechanism for **storing binary files IN THE APP'S EXECUTABLE**.
- Normally, apps always need a certain set of files (icons, translation files, etc.) and you don't want to run the risk of losing the files.
- The resources associated with an application are specified in a *.qrc* file (*xml* file that lists files on the disk and optionally assigns them a resource name that the application must use to access the resource).
- By default, resources are accessible in the application under the same file name as they have in the xml tree, with a **:/** prefix: e.g. ":/images/cut.png"

```
<!DOCTYPE RCC><RCC version="1.0">
<qresource>
    <file>images/copy.png</file>
    <file>images/cut.png</file>
    <file>images/new.png</file>
    <file>images/open.png</file>
    <file>images/paste.png</file>
    <file>images/save.png</file>
</qresource>
</RCC>
```

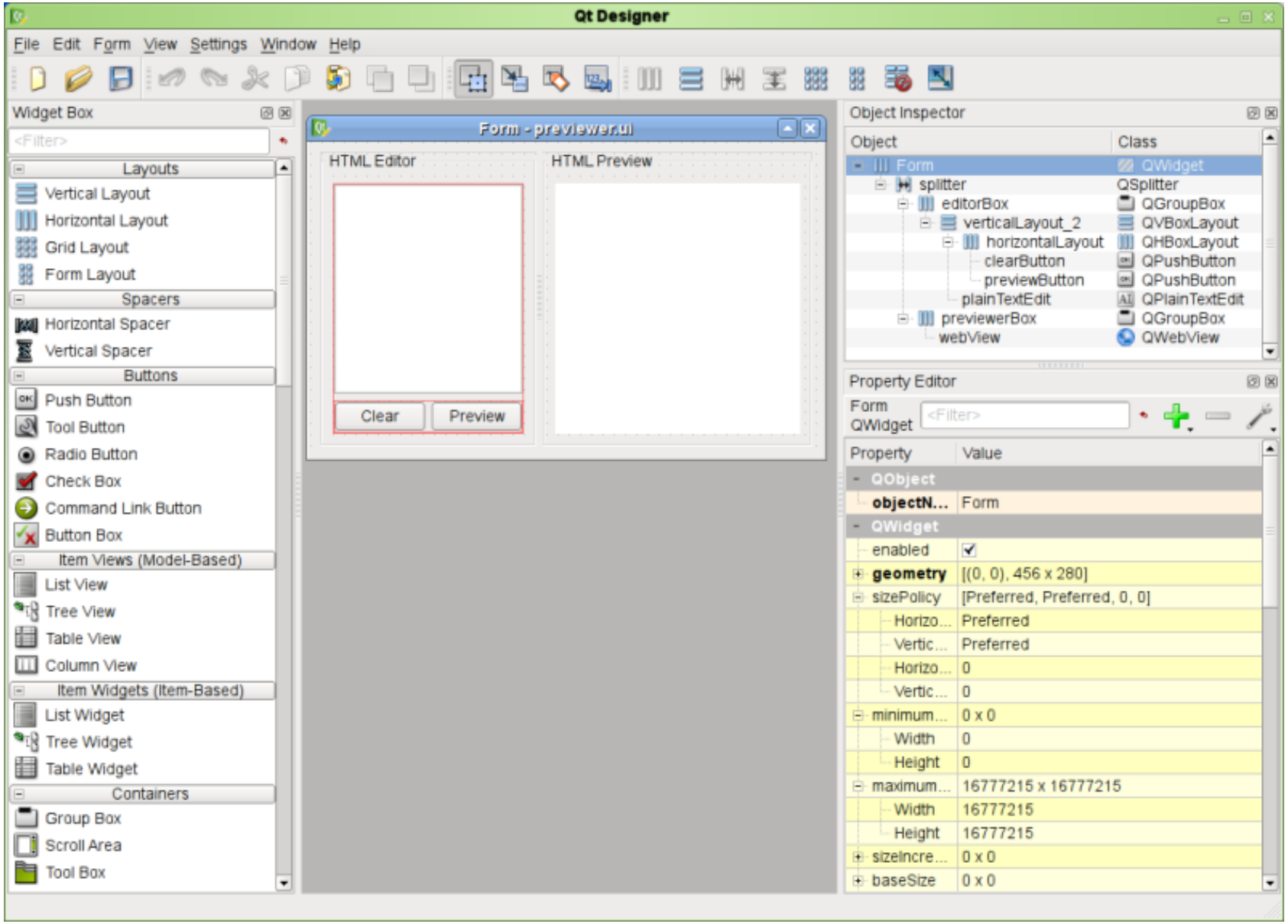
Qt Designer I

- Tutorials on how to use QtDesigner: https://doc.qt.io/qt-5/qtdesigner-manual.html
- Qt Designer is the Qt tool for designing and building graphical user interfaces (GUIs) with Qt Widgets.
- You can compose and customize your windows or dialogs in a *what-you-see-is-what-you-get* (WYSIWYG) manner, and test them using different styles and resolutions.
- Objects can be dragged from the widget box and dropped on the form.
- Object properties can be modified interactively.

Qt Designer II

- Using the Qt Designer can be faster than hand-coding the interface.
- One can experiment with different designs quickly.
- A **.ui** file is created, representing the widget tree of the form in *xml* format.
- The User Interface Compiler (uic) can then be used to create a corresponding C++ header file.

Qt Designer III



When should we implement the UI programmatically?

- When the elements in the dialog must change dynamically.
- When we want to use custom widgets.
- How?
  1. Create a new class, by inheriting from QWidget.
  2. Implement the GUI.
  3. Show the newly created widget.

Common pattern to build a GUI in Qt

1. Instantiate the required Qt widgets.
2. Set properties for these, if necessary.
3. Add the widgets to a layout (the layout manager will take care of the position and size).
4. Connect the widgets using the signal and slot mechanism (will be presented next week).