Widget personalizado QWidget

Hasta ahora hemos trabajado ejemplos muy sencillos en una ventana principal con un solo widget, pero ahora vamos a dar un paso adelante y a implementar varios widgets en el mismo espacio. Si vamos a utilizar varios widgets necesitamos organizarlos y precisamente para eso existen los layouts, que se traducirían en español como disposiciones.

Hace un tiempo encontré una forma excelente de ilustrar el funcionamiento de los layouts. Consiste en crear una clase base con un fondo coloreado, así veremos exactamente el espacio que ocupan los layouts de una forma muy visual.

Así que vamos a preparar un widget personalizado para visualizar nuestros layouts, lo vamos a llamar `caja` y lo heredaremos de una simple label:

```
from PySide6.QtWidgets import QApplication, QLabel, QMainWindow
import sys

class Caja(QLabel):
    def __init__(self, color):
        super().__init__()
        self.setStyleSheet(f"background-color:{color}")
```

Me váis a permitir adelantarme un poco y utilizar una hoja de estilo para otorgar un color de fondo a nuestra caja mediante el método `setStyleSheet`. Este método lo estudiaremos a fondo en la unidad de tematización.

Vamos a crear una ventana principal básica usando esta caja como widget central:

```
from PySide6.QtWidgets import QApplication, QLabel, QMainWindow
import sys


class Contenedor(QLabel):
    def __init__(self, color):
        super().__init__()
        self.setStyleSheet(f"background-color:{color}")


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        caja = Contenedor("green")
        self.setCentralWidget(caja)
```

```
if __name__ == "__main__":
    app = QApplication(sys.argv)
    window = MainWindow()
    window.show()
    sys.exit(app.exec_())
```

Listo, en la siguiente lección vamos a experimentar creando contenedores y organizándolos en diferentes disposiciones.

Layouts básicos QVBoxLayout y QHBoxLayout

Existen dos tipos de disposición básica para organizar elementos vertical u horizontalmente.

Empecemos por el primer tipo:

```
from PySide6.QtWidgets import QApplication, QMainWindow, QLabel,
QVBoxLayout
import sys


class Caja(QLabel):
    def __init__(self, color):
        super().__init__()
        self.setStyleSheet(f"background-color:{color}")


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        # empezamos creando un layout vertical
        layout = QVBoxLayout()

        # le añadimos una caja verde
        layout.addWidget(Caja("green"))

        # probamos a establecerlo como widget central
        self.setCentralWidget(layout)


if __name__ == "__main__":
    app = QApplication(sys.argv)
```

```
    window = MainWindow()
    window.show()
    sys.exit(app.exec_())
```

Al ejecutarlo veremos que nos devuelve un error:

```
TypeError: 'PySide6.QtWidgets.QMainWindow.setCentralWidget' called
with wrong argument types:
PySide6.QtWidgets.QMainWindow.setCentralWidget(QVBoxLayout)
```

Nos está indicando que no se permite utilizar un layout como widget central. Eso es porque los layouts no son widgets, no heredan de la clase `QWidget`. La forma de manejar esto es crear un `dummy widget` para asignarle el layout y usarlo como widget central:

```
from PySide6.QtWidgets import ..., QWidget  # edited

class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        layout = QVBoxLayout()
        layout.addWidget(Contenedor("green"))

        # creamos un dummy widget para hacer de contenedor
        widget = QWidget()

        # le asignamos el layout
        widget.setLayout(layout)

        # establecemos el dummy widget como widget central
        self.setCentralWidget(widget)
```

Como podemos observar el layout contiene la caja verde, que a su vez se encuentra dentro del dummy widget asignado como widget principal. La diferencia más notable es que un layout tiene espacios y márgenes, por eso la caja no ocupa todo el espacio.

Vamos a añadir más cajas para ver cómo organiza el espacio automáticamente:

```
# le añadimos unas cuantas cajas
layout.addWidget(Caja("green"))
layout.addWidget(Caja("blue"))
layout.addWidget(Caja("red"))
```

¿Véis como el layout vertical reparte equitativamente el espacio entre los widgets que contiene?

Vamos a cambiar a un layaout horizontal para ver cómo se reparten los objetos:

- `QVBoxLayout` -> `QHBoxLayout`
- `layout = QVBoxLayout()` -> `layout = QHBoxLayout()`

Es exactamente lo mismo, pero en esta ocasión todo se organiza horizontalmente.

Para modificar los márgenes del layout se utiliza el método `setContentsMargins` pasándole por orden los píxeles a la izquierda, arriba, derecha y abajo:

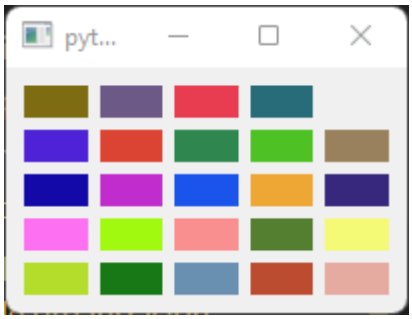
```
# modificamos los márgenes
layout.setContentsMargins(0,0,0,0)
```

Y para quitar el espaciado entre los widgets utilizaremos `setSpacing` con 0 píxeles:

```
# modificamos el espaciado
layout.setSpacing(0)
```

Sobra decir que estamos usando nuestra caja para visualizar el espacio de cada widget, pero en la vida real estaríamos añadiendo etiquetas, campos de texto y otros widgets para diseñar formularios o lo que necesitemos.

Layout en cuadrícula QGridLayout



El layout en cuadrícula se basa en crear un único layout compuesto de filas y columnas. Primero se crea la cuadrícula y luego se rellena cada hueco o celda haciendo referencia a ella con índices que empiezan valiendo cero:

```
from PySide6.QtWidgets import (
    QApplication, QMainWindow, QLabel, QGridLayout, QWidget)
import sys


class Caja(QLabel):
    def __init__(self, color):
        super().__init__()
        self.setStyleSheet(f"background-color:{color}")


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        # creamos un layout en cuadrícula
        cuadricula = QGridLayout()

        # añadimos widgets en las celdas usando los índices
        cuadricula.addWidget(Caja("orange"), 0, 0)
        cuadricula.addWidget(Caja("purple"), 1, 1)
        cuadricula.addWidget(Caja("magenta"), 2, 2)
        cuadricula.addWidget(Caja("gray"), 2, 0)
        cuadricula.addWidget(Caja("red"), 0, 2)
```

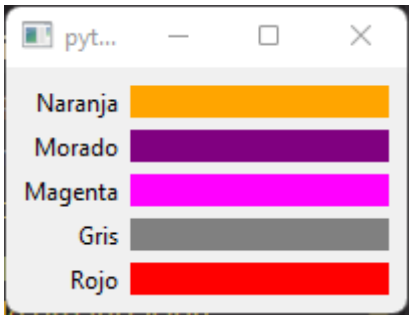
```
        # cremos el widget dummy y le asignamos el layout horizontal
        widget = QWidget()
        widget.setLayout(cuadricula)

        self.setCentralWidget(widget)


if __name__ == "__main__":
    app = QApplication(sys.argv)
    window = MainWindow()
    window.show()
    sys.exit(app.exec_())
```

El tamaño de la cuadrícula vendrá determinado automáticamente por los mayores índices con un widget, lo que generará huecos vacíos si no los rellenemos explícitamente.

Vamos a hacer un experimento para generar dinámicamente una cuadrícula con cajas de colores aleatorios a partir de dos bucles for:

```
import random

# bucles for para generar una cuadrícula
for fila in range(5):
    for columna in range(5):
        # añadimos una caja de color aleatorio
        color = str(hex(random.randint(0, 16777215)))  # int(0xFFFFFF)
        cuadricula.addWidget(Caja(f"#{color[2:]}"), fila, columna)
```

Layout en formulario QFormLayout



Si lo que necesitamos es una estructura para manejar un formulario podemos usar un `QFormLayout` que nos permite añadir etiquetas y widgets en fila de una forma más cómoda que las cuadrículas:

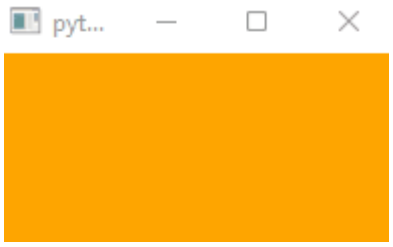
```
from PySide6.QtWidgets import (
    QApplication, QMainWindow, QLabel, QFormLayout, QWidget)
import sys


class Caja(QLabel):
    def __init__(self, color):
        super().__init__()
        self.setStyleSheet(f"background-color:{color}")
```

```
class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        # creamos un layout en formulario
        formulario = QFormLayout()

        # añadimos widgets con etiquetas en filas
        formulario.addRow("Campo 1", Caja("orange"))
        formulario.addRow("Campo 2", Caja("purple"))
        formulario.addRow("Campo 3", Caja("magenta"))
        formulario.addRow("Campo 4", Caja("gray"))
        formulario.addRow("Campo 5", Caja("red"))

        # cremos el widget dummy y le asignamos el layout
        widget = QWidget()
        widget.setLayout(formulario)

        self.setCentralWidget(widget)


if __name__ == "__main__":
    app = QApplication(sys.argv)
    window = MainWindow()
    window.show()
    sys.exit(app.exec_())
```

Dependiendo del sistema operativo el formulario se visualizará de forma diferente con el objetivo de respetar la integración, pero es posible cambiar la alineación de las etiquetas y los widgets manualmente:

```
# configuraciones extra
formulario.setLabelAlignment(Qt.AlignRight)
formulario.setFormAlignment(Qt.AlignHCenter | Qt.AlignVCenter)
```

Layout apilado QStackedLayout



Otra disposición que da mucho juego es apilar los widgets usando un `QStackedLayout`:

```
from PySide6.QtWidgets import (
    QApplication, QMainWindow, QLabel, QStackedLayout, QWidget)
import sys


class Caja(QLabel):
    def __init__(self, color):
        super().__init__()
```

```
        self.setStyleSheet(f"background-color:{color}")


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        # creamos un layout apilado
        layout = QStackedLayout()

        # Añadimos varios widgets unos sobre otros
        layout.addWidget(Caja("orange"))
        layout.addWidget(Caja("magenta"))
        layout.addWidget(Caja("purple"))
        layout.addWidget(Caja("red"))

        # creamos el widget dummy y le asignamos el layout apilado
        widget = QWidget()
        widget.setLayout(layout)

        self.setCentralWidget(widget)


if __name__ == "__main__":
    app = QApplication(sys.argv)
    window = MainWindow()
    window.show()
    sys.exit(app.exec_())
```

El problema de este layout es que necesita controladores.

Vamos a definir un evento que capture cuando presionamos las flechas del teclado para poder alternar entre los widgets. Los eventos ya existen en el widget, lo que haremos es sobreescribir su comportamiento:

```
from PySide6.QtCore import Qt  # nuevo

def keyPressEvent(self, event):
    # detectamos la flecha presionada
    if event.key() == Qt.Key_Right:
        print("Flecha derecha presionada")
    elif event.key() == Qt.Key_Left:
        print("Flecha izquierda presionada")
    # continuamos con el evento por defecto
    event.accept()
```

Ahora utilizaremos el método `setCurrentIndex` del layout para controlar el widget que se muestra teniendo en cuenta que el índice empieza valiendo 0 y al tener 4 widgets su valor máximo será 3. Este numero máximo podemos conseguirlo contando los elementos del layout con su método `count`.

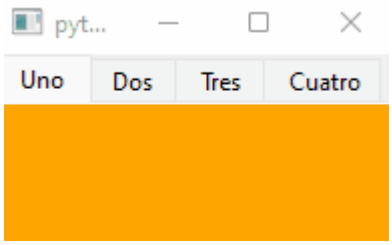
Al presionar la flecha derecha incrementaremos el índice y con la izquierda lo decrementaremos. Para generar un efecto infinito si el índice es menor que cero lo estableceremos al máximo, si es mayor que el máximo lo estableceremos a cero:

```
# necesitamos crear un accesor para usar el layout desde el evento
self.layout = layout
```

```
def keyPressEvent(self, event):
    # recuperamos el índice
    indice = self.layout.currentIndex()
    # buscamos el indice máximo del layout contando cuantos widgets
tiene
    indice_maximo = self.layout.count() - 1

    # dependiendo de la flecha presionada sumamos o restamos
    if event.key() == Qt.Key_Right:
        indice += 1
    elif event.key() == Qt.Key_Left:
        indice -= 1

    # rectificamos el índice para generar el efecto infinito
    if indice > indice_maximo:
        indice = 0
    if indice < 0:
        indice = indice_maximo

    # finalmente establecemos el nuevo índice
    self.layout.setCurrentIndex(indice)

    # continuamos con el evento por defecto
    event.accept()
```

En este experimento hemos introducido los `eventos`, pero podríamos haber utilizado unos botones para cambiar de índice sin problema.

Layout con pestañas QTabWidget



El último tipo de disposición que veremos es con pestañas utilizando un `QTabWidget`, se trata de una variante del apilado con un control más visual. Esta variante sí hereda de la clase `QWidget` y por tanto no requiere un dummy widget:

```
from PySide6.QtWidgets import (
    QApplication, QMainWindow, QLabel, QTabWidget)
import sys


class Caja(QLabel):
    def __init__(self, color):
        super().__init__()
        self.setStyleSheet(f"background-color:{color}")


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
```

```
        # creamos un layout de pestañas
        tabs = QTabWidget()

        # Añadimos varios widgets como pestañas con nombres
        tabs.addTab(Caja("orange"), "Uno")
        tabs.addTab(Caja("magenta"), "Dos")
        tabs.addTab(Caja("purple"), "Tres")
        tabs.addTab(Caja("red"), "Cuatro")

        # asignamos las pestañas como widget central
        self.setCentralWidget(tabs)


if __name__ == "__main__":
    app = QApplication(sys.argv)
    window = MainWindow()
    window.show()
    sys.exit(app.exec_())
```

Algunas opciones interesantes de este widget es que podemos modificar la posición de las pestañas:

```
tabs.setTabPosition(QTabWidget.West)  # West, East, North, South
```

O hacer que las pestañas se puedan arrastrar para cambiar el orden:

```
tabs.setMovable(True)
```

Y con esto acabamos el repaso de los layouts esenciales de Qt.