Event dispatching & handling

The `pyglet.event` module provides a framework for uniformly dispatching and handling events. For our purposes, an "event dispatcher" is an object that has events it needs to notify other objects about, and an "event handler" is some code (a function or method) that can be registered (or "attached") to receive those events.

Event Dispatchers are created by subclassing the `EventDispatcher` base class. Many of pyglet's built-in modules, such as `pyglet.window`, `pyglet.media`, `pyglet.app`, `pyglet.text`, `pyglet.input`, `pyglet.gui` and others make use of this pattern. You can also reuse this in your own classes easily.

Event handlers are simply functions or methods that are written to accept the same arguments as the dispatched event. Event handlers can be registered or unregistered during runtime. More than one handler can be registered to receive the same events, which is described in the following sections. Event dispatchers can *optionally* have default handlers for some of their events. Your own handlers can replace these entirely, or just be added on.

Setting event handlers

For an example, lets look at the `Window` class. `Window` subclasses `EventDispatcher` and, being a Window, has a variety of different events which it dispatches. For instance, the `pyglet.window.Window.on_resize()` event. Every time a resizeable Window is resized (and once when first created), this event is dispatched with two parameters: `(width, height)`. Therefore, an event handler for this event should be written to accept these two values. For example:

```
def on_resize(width, height):
    pass
```

There are a few different ways in which event handlers can be attached to recieve them. The simplest way is to directly attach the event handler to the corresponding attribute on the object. This will completely replace the default event handler:

;

```
window = pyglet.window.Window()

def on_resize(width, height):
    # Set some custom projection

window.on_resize = on_resize
```

Sometimes replacing the default handler is desired, but not in all cases. For example the default *Window.on_resize* handler is responsible for setting up a orthographic 2D projection for drawing graphics. If you replace it entirely, you must also handle setting the projection yourself.

Another way to replace a default event handler is when subclassing pyglet objects. This is common do do with Window class, as shown in Subclassing Window. If your methods have the same name as the default event, they will be replaced:

```
class MyWindow(pyglet.window.Window):
    def on_resize(self, width, height):
        # set a custom projection there
```

You can of course still call the default handler with *super()*, and then add your custom code before/after that:

```
class MyWindow(pyglet.window.Window):

    def on_resize(self, width, height):
        super().on_resize(width, height)
        # do something else
```

The event decorator

Instead of replacing default handlers, you can just also add an additional handler. pyglet provides a shortcut using the `event` decorator. Your custom event handler will run, followed by the default event handler:

```
window = window.Window()

@window.event
def on_resize(width, height):
    print(f"Window was resized to: {width}x{height}")
```

or if your handler has a different name, pass the event name to the decorator:

```
@window.event('on_resize')
def my_resize_handler(width, height):
    pass
```

In most simple cases, the `event` decorator is most convienent. One limitation of using the decorator, however, is that you can only add one additional event handler. If you want to add multiple additional event handlers, the next section describes how to accomplish that.

Stacking event handlers

It is often convenient to attach more than one event handler for an event. `EventDispatcher` allows you to stack event handlers upon one another, rather than replacing them outright. The event will propagate from the top of the stack to the bottom, but can be stopped by any handler along the way by returning *pyglet.event.EVENT_HANDLED*.

To push an event handler onto the stack, use the `push_handlers()` method:

```
def on_key_press(symbol, modifiers):
    if symbol == key.SPACE:
        fire_laser()

window.push_handlers(on_key_press)
```

One use for pushing handlers instead of setting them is to handle different parameterisations of events in different functions. In the above example, if the spacebar is pressed, the laser will be fired. After the event handler returns control is passed to the next handler on the stack, which on a `Window` is a function that checks for the ESC key and sets the `has_exit` attribute if it is pressed. By pushing the event handler instead of setting it, the application keeps the default behaviour while adding additional functionality.

You can prevent the remaining event handlers in the stack from receiving the event by returning a true value. The following event handler, when pushed onto the window, will prevent the escape key from exiting the program:

```
def on_key_press(symbol, modifiers):
    if symbol == key.ESCAPE:
        return True

window.push_handlers(on_key_press)
```

You can push more than one event handler at a time, which is especially useful when coupled with the `pop_handlers()` function. In the following example, when the game starts some additional event handlers are pushed onto the stack. When the game ends (perhaps returning to some menu screen) the handlers are popped off in one go:

```
def start_game():
    def on_key_press(symbol, modifiers):
        print('Key pressed in game')
        return True

    def on_mouse_press(x, y, button, modifiers):
        print('Mouse button pressed in game')
        return True

    window.push_handlers(on_key_press, on_mouse_press)

def end_game():
    window.pop_handlers()
```

Note that you do not specify which handlers to pop off the stack – the entire top "level" (consisting of all handlers specified in a single call to `push_handlers()`) is popped.

You can apply the same pattern in an object-oriented fashion by grouping related event handlers in a single class. In the following example, a `GameEventHandler` class is defined. An instance of that class can be pushed on and popped off of a window:

```
class GameEventHandler:
    def on_key_press(self, symbol, modifiers):
        print('Key pressed in game')
        return True

    def on_mouse_press(self, x, y, button, modifiers):
        print('Mouse button pressed in game')
        return True

game_handlers = GameEventHandler()

def start_game()
    window.push_handlers(game_handlers)

def stop_game()
    window.pop_handlers()
```

In order to prevent issues with garbage collection, the `EventDispatcher` class only holds weak references to pushed event handlers. That means the following example will not work, because the pushed object will fall out of scope and be collected:

```
dispatcher.push_handlers(MyHandlerClass())
```

Instead, you must make sure to keep a reference to the object before pushing it. For example:

```
my_handler_instance = MyHandlerClass()
dispatcher.push_handlers(my_handler_instance)
```

Creating your own event dispatcher

pyglet provides the `Window`, `Player`, and other event dispatchers, but exposes a public interface for creating and dispatching your own events.

The steps for creating an event dispatcher are:

1. Subclass `EventDispatcher`
2. Call the `register_event_type()` class method on your subclass for each event your subclass will recognise.
3. Call `dispatch_event()` to create and dispatch an event as needed.

In the following example, a hypothetical GUI widget provides several events:

```
class ClankingWidget(pyglet.event.EventDispatcher):
    def clank(self):
        self.dispatch_event('on_clank')

    def click(self, clicks):
        self.dispatch_event('on_clicked', clicks)

    def on_clank(self):
        print('Default clank handler.')

ClankingWidget.register_event_type('on_clank')
ClankingWidget.register_event_type('on_clicked')
```


Event handlers can then be attached as described in the preceding sections:

```
widget = ClankingWidget()

@widget.event
def on_clank():
    pass

@widget.event
def on_clicked(clicks):
    pass

def override_on_clicked(clicks):
    pass

widget.push_handlers(on_clicked=override_on_clicked)
```

The `EventDispatcher` takes care of propagating the event to all attached handlers or ignoring it if there are no handlers for that event.

There is zero instance overhead on objects that have no event handlers attached (the event stack is created only when required). This makes `EventDispatcher` suitable for use even on light-weight objects that may not always have handlers. For example, `Player` is an `EventDispatcher` even though potentially hundreds of these objects may be created and destroyed each second, and most will not need an event handler.

Implementing the Observer pattern

The Observer design pattern, also known as Publisher/Subscriber, is a simple way to decouple software components. It is used extensively in many large software projects; for example, Java's AWT and Swing GUI toolkits and the Python `logging` module; and is fundamental to any Model-View-Controller architecture.

`EventDispatcher` can be used to easily add observerable components to your application. The following example recreates the *ClockTimer* example from *Design Patterns* (pages 300-301), though without needing the bulky `Attach`, `Detach` and `Notify` methods:

```
# The subject
class ClockTimer(pyglet.event.EventDispatcher):
    def tick(self):
        self.dispatch_event('on_update')

ClockTimer.register_event_type('on_update')

# Abstract observer class
class Observer:
    def __init__(self, subject):
        subject.push_handlers(self)

# Concrete observer
class DigitalClock(Observer):
    def on_update(self):
        pass

# Concrete observer
class AnalogClock(Observer):
    def on_update(self):
        pass

timer = ClockTimer()
digital_clock = DigitalClock(timer)
analog_clock = AnalogClock(timer)
```

The two clock objects will be notified whenever the timer is “ticked”, though neither the timer nor the clocks needed prior knowledge of the other. During object construction any relationships between subjects and observers can be created.