

Lecture 23

Chapter 15 Event-driven Programming

15.1 Basics of Event-Driven Programming

15.2 Event Handling in our Graphics Module

Introduction

In this chapter we will examine another important style known as **event-driven programming**.

In this paradigm, an executing program waits passively for external **events** to occur, and then responds appropriately to those events.

15.1 Basics of Event-Driven Programming

Example 1:

```
fname=raw_input('Please, enter the file name:')
```

! when writing a program we have to forecast the precise opportunities for user interaction, and the user must follow this script.

Example 2:

```
paper=Canvas()  
cue = paper.wait()  
ball=Circle(10,cue.getMouseLocation())  
ball.setFillColor('red')  
paper.add(ball)
```

15.1 Basics of Event-Driven Programming

From now on: the sequential way of thinking is not as meaningful. Such programs should be described through **event handling**.

The program declares various events that should be available to the user, and provides explicit code that should be followed to handle each individual type of event when triggered.

This piece of code is called **event handler**.

15.1 Basics of Event-Driven Programming

Event Handlers

- Usually implemented as a stand-alone function (called **callback function**), or as an instance of a specially defined class
- A separate event handler for each kind of event

The callback function should be registered in advance as a handler for a particular kind of event. Sometimes it is described as registering to **listen** for an event, thus handlers are sometimes called **listeners**.

With object-oriented programming, event handling is usually implemented through an *event-handling class*.

Advantage: a handler can maintain state information to coordinate the responses for a series of events.

15.1 Basics of Event-Driven Programming

The event loop

The design of event-driven software is quite different from our traditional flow-driven programming.

When software executes, initialization is performed in traditional fashion (create one or more windows, set up appropriate items) + event handlers are declared and registered.

Then, execution may reach a point where next task is simply wait for the user to do something.

Our software should be ready to handle any number of predefined events triggered in arbitrary order. Usually it is done with **event loop** - infinite loop that does nothing, yet when an event occurs, it stops and looks for an appropriately registered handler (if it is not found the event is ignored).

15.1 Basics of Event-Driven Programming

The event loop

When a handler is called, the flow of control is temporarily ceded to the handler.

Once the handler completes its task, the default continuation is to re-enter the event loop (except when we want to 'quit' event loop).

15.1 Basics of Event-Driven Programming

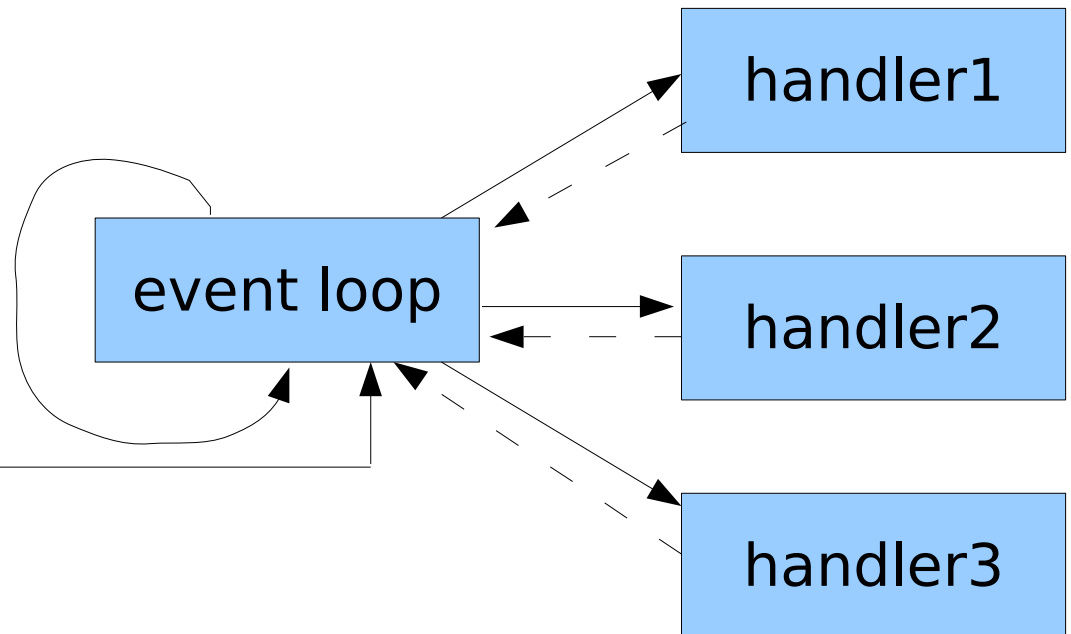
The event loop

When a handler is called, the flow of control is temporarily ceded to the handler.

Once the handler completes its task, the default continuation is to re-enter the event loop (except when we want to 'quit' event loop).

Main program

```
win = Canvas()
win.addHandler(handler1)
win.addHandler(handler2)
win.addHandler(handler3)
```



Flow of control in an event-driven program

15.1 Basics of Event-Driven Programming

Threading

Multithreaded programming allows the main program continue executing, even while the event loop is monitoring and responding to events, i.e. the flow control is not ceded to the event loop.

The main routine and this event loop run simultaneously as separate **threads** of the program.

Threading can be supported by the programming language and the underlying operating system.

In reality, the threads are sharing the CPU, each given small alternating time slices in which to execute.

15.2 Event Handling in our Graphics Module

Without knowing it, every time we use `cs1graphics` package, an event loop is running concurrently with the rest of our program.

Every time we click on the canvas's window or typed on the keyboard the event loop is informed.

Let's see how do we play with clicks before we proceed with event handling.

See program [clicks.py](#)

15.2 Event Handling in our Graphics Module

The **EventHandler** class

is a base class for creating new event Handlers.

Member Functions:

```
def __init__(self)
```

Create a new event handler.

Children of this class must call this constructor.

```
def handle(self, event)
```

Handle an event.

Child classes must override this method, but do not need to call it.

`event` is an instance of **Event** class, used to describe the particular event that occurred (will explore it in next class).



15.2 Event Handling in our Graphics Module

A very simple handler:

```
class BasicHandler(EventHandler):  
    def handle(self, event):  
        print 'Event Triggered'
```

What does it do?

15.2 Event Handling in our Graphics Module

A very simple handler:

```
class BasicHandler(EventHandler):  
    def handle(self, event):  
        print 'Event Triggered'
```

What does it do?

Prints Event Triggered each time an event is detected.

15.2 Event Handling in our Graphics Module

Example 1:

```
class BasicHandler(EventHandler):
    def handle(self, event):
        print 'Event Triggered'

def main():

    simple = BasicHandler()
    paper=Canvas(700,600,'light yellow','no Title')
    paper.addHandler(simple) # registering handler
simple

main()
```

See what will be happening (run [example1.py](#))

15.2 Event Handling in our Graphics Module

Example 1:

```
class BasicHandler(EventHandler):
    def handle(self, event):
        print 'Event Triggered'

def main():

    simple = BasicHandler()
    paper=Canvas(700,600,'light yellow','no Title')
    paper.addHandler(simple) # registering handler
simple

main()
```

See what will be happening (run [example1.py](#))

Every time a key is pressed (while the canvas is active) or we click on the canvas, we see **Event Triggered** in the Python Shell

15.2 Event Handling in our Graphics Module

Example 2:

```
class BasicHandler(EventHandler):
    def handle(self,event):
        print 'Event Triggered'

def main():
    paper=Canvas(700,600,'light yellow','no Title')
    button=Rectangle(60,20,Point(300,300))
    button.setFillcolor('green')
    paper.add(button)

    simple = BasicHandler()
    button.addHandler(simple)

main()
```

See what will be happening now (run [example2.py](#))

15.2 Event Handling in our Graphics Module

Example 2:

```
class BasicHandler(EventHandler):
    def handle(self,event):
        print 'Event Triggered'

def main():
    paper=Canvas(700,600,'light yellow','no Title')
    button=Rectangle(60,20,Point(300,300))
    button.setFillColor('green')
    paper.add(button)

    simple = BasicHandler()
    button.addHandler(simple)

main()
```

See what will be happening now (run [example2.py](#))

Event Triggered is displayed when we click on the green rectangle and when we press a key (on the keyboard) while the mouse cursor is over the green rectangle .

15.2 Event Handling in our Graphics Module

Example 3:

```
class TallyHandler(EventHandler):
    def __init__(self, textObj):
        EventHandler.__init__(self)
        self._count=0
        self._text=textObj
        self._text.setMessage(str(self._count))

    def handle(self, event):
        self._count += 1
        self._text.setMessage(str(self._count))
        print 'Event Tiggered. Count:',self._count

def main():
    paper=Canvas(700,600,'light yellow','no Title')
    score=Text('',12,Point(300,300))
    paper.add(score)

    referee=TallyHandler(score)
    paper.addHandler(referee)

main()
```

See what will be happening now (run [example3.py](#))