Object Oriented Software Engineering (COMP2003)

Lecture 5: Event-Driven Programming

Updated: 16th April, 2018

Discipline of Computing
School of Electrical Engineering, Computing and Mathematical Sciences (EECMS)

Copyright © 2018, Curtin University
CRICOS Provide Code: 00301J

Outline

Event-Driven Programming

- Event-Driven Programming (EDP) is perhaps as important as Object Orientation (OO).
- OO and EDP are both software design paradigms.
  - They are compatible, but independent.
  - EDP can be done in virtually any language.

Events

- When you first learn programming, you're taught that:
  - The program begins at `main()`.
  - `main()` calls other functions or methods.
  - In turn, they call *more* functions/methods, etc.
  - This is basically the way to break up a program.
- However, many programs spend most of their time waiting for *events*:
  - Button presses (and similar user-related signals).
  - Arrival of network data.
  - Timer expiry.
- Often you know *what* to do, but not *when* to do it.
  - You need to ask something else to figure out the "when"...
  - ...and "call back" at the right time.
- A hierarchy of method calls doesn't capture this very well.

Large-Scale Decoupling

- "*Events*" are how we often talk about communication between high level things.
- Using events helps minimise coupling between packages, components, etc.
- In MVC:
  - The view generates user-input events that the controller must handle.
  - The model generates update events that the view must handle (by re-displaying itself).
- Libraries also generate various events that your application can choose to handle.
- In networking/distributed applications, an event happens when data is received from a remote location.

Event Handling

- Events involve:
  1. An *event source* or *subject* that generates the events; e.g., a timer, or a GUI button.
  2. An *event handler*, *observer* or *listener* that needs to know when an event happens.
- Neither one knows anything about the other (other than their existence).
  - This is the decoupling at work.
- They interact as follows:
  1. You *register* zero or more event handlers with the event source.
  2. When an event happens, the event source calls all event handlers (if any).
  3. You can later *un-register* event handlers if needed.

Inversion of Control

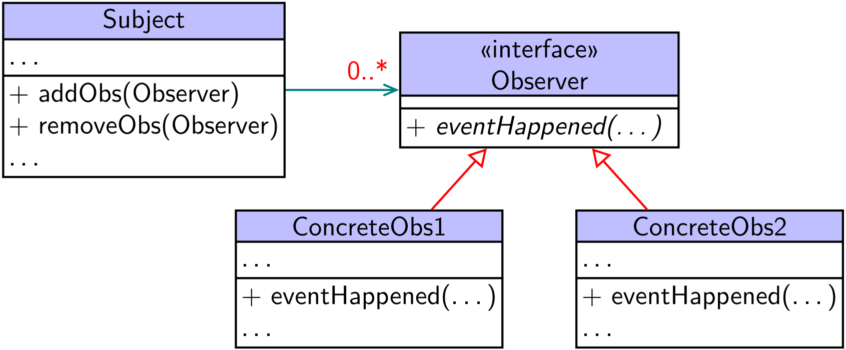
- Often your whole application is made up of event-handlers.
  - All your code is simply *responding* to something.
  - (Apart from the initialisation in `main()`.)
- We call this situation "inversion of control".
  - "Control" refers to deciding when things happen.
  - "Inversion" means that *you* don't decide this anymore. Your code simply reacts.
- Very common style of programming.
  - Most GUI, web and mobile apps are written this way.

Exceptions and Events

- Event handlers *cannot* generally throw exceptions.
  - The event source doesn't know anything about the event handler.
  - So it cannot meaningfully respond when something goes wrong.
- So, your event handler *must* deal with it.
  - Or call on another object to deal with it.
- *However...* one way of handling exceptions is to treat their occurrence *as* an event!
  - So your `catch` block can become an event source all of its own.
  - And some other event handler will pre-register itself to handle these events.
  - We'll get back to this.

Observer Pattern

- Object Oriented + Event-Driven = Observer Pattern.
- An event source class is a *subject* – it generates events.
- An event handler is an *observer* – it needs to receive events.
- They go together a bit like the Strategy pattern:



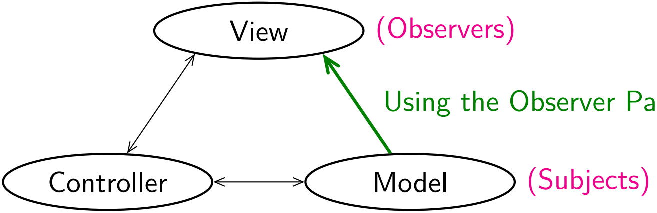
Observer Pattern – the Basics

- The subject (event source) associates with zero-or-more observers (event handlers).
- When an event occurs, the subject calls *all* its observers, and (possibly) provides event details.
  - The subject doesn't care what the observers do.
  - Observers just do their own thing with the information provided.
- Observers have a common interface.
  - The subject doesn't really know anything about its observers.
  - (Except what's in the interface.)
- The subject's observers are added and removed by another class.
  - An observer might choose to add/remove itself.

Observers vs Strategies

- The Observer and Strategy patterns can look similar in UML, but they are quite distinct.
- In the Strategy Pattern:
  - A strategy class "works for" its context. It:
    - Is *owned* by the context (i.e. aggregation).
    - Performs a service needed by the context, and (generally) provides information back to it.
  - Strategies are *selected*, based on the situation.
- In the Observer Pattern:
  - Simply one-way communication with minimal coupling.
  - Observers each have their own individual responsibilities.
  - Subjects don't care what observers do, or even how many there are (even if it's zero).
  - There's no selection. Subjects call all registered observers.

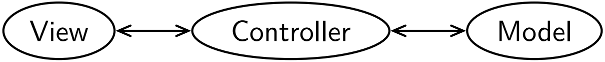
Observers in MVC



- Model $\rightarrow$ View communication is a key problem solved by the Observer Pattern.
- Various parts of the view *observe* specific parts of the model.
- When part of the model is updated, we call that an "event".
- Any observers (parts of the view) are notified, so they can update the information they display.

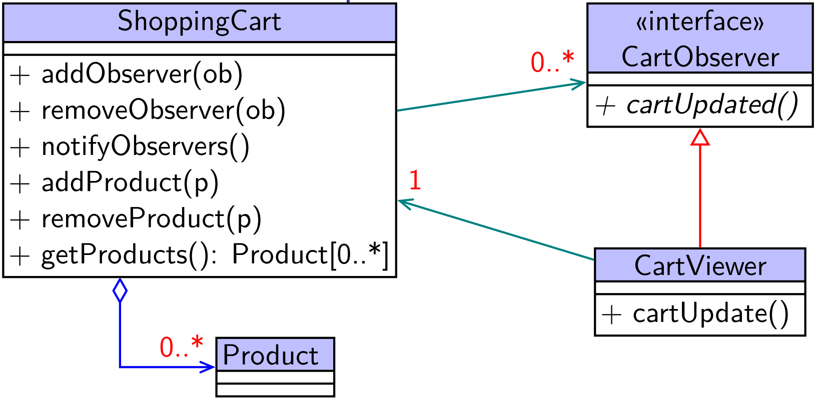
Everything Via the Controller?

- In some descriptions of MVC, the model and view don't communicate directly at all.
  - Everything goes through the controller.



- In console-based programs and server-side web applications:
  - The view is built-up step-by-step, in a very specific order.
  - So, parts of the view can't independently update themselves.
  - Must be coordinated by a controller.
- In GUI applications, the view is more independent.
  - See previous slide.
  - Parts of the UI *can* update themselves without interfering with anything else.
  - So, no need for the controller to get directly involved.
  - (The controller still updates the model in the first place, so it's always *indirectly* involved.)

Observers in MVC: Example

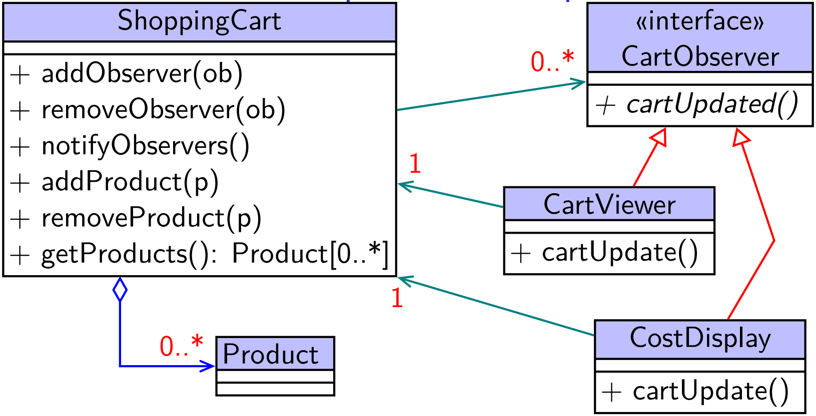


1. When we add/remove a product from `ShoppingCart`, we call `notifyObservers()`.
2. `notifyObservers()` calls `cartUpdated()`, for each observer.
3. `CartViewer` (the only observer in this case) uses its "back reference" to call `getProducts()`, and displays the result.

Observers in MVC: Subject Code

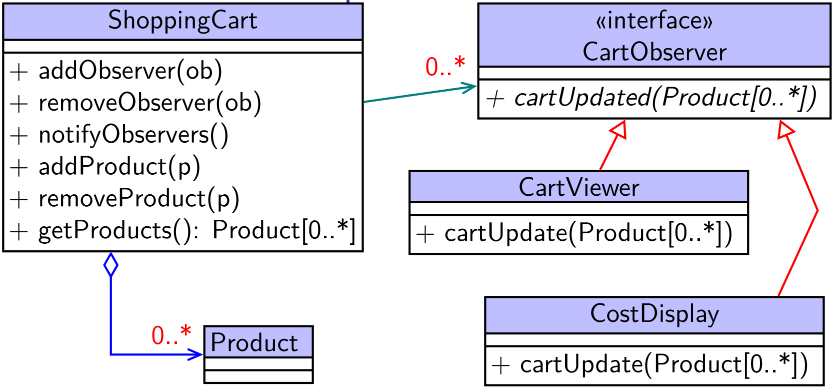
```
public class ShoppingCart // The 'subject'
{
    private Set<Product> products = new HashSet<>();
    private Set<CartObserver> obs = new HashSet<>();

    public void addObserver(CartObserver ob) { obs.add(ob); }
    public void addProduct(Product p)
    {
        products.add(p);
        notifyObservers(); // <-- also in removeProduct()
    }
    public void notifyObservers()
    {
        for(CartObserver ob : obs ) { ob.cartUpdated(); }
    }
    ...
}
```

Observers in MVC: Example with Multiple Observers



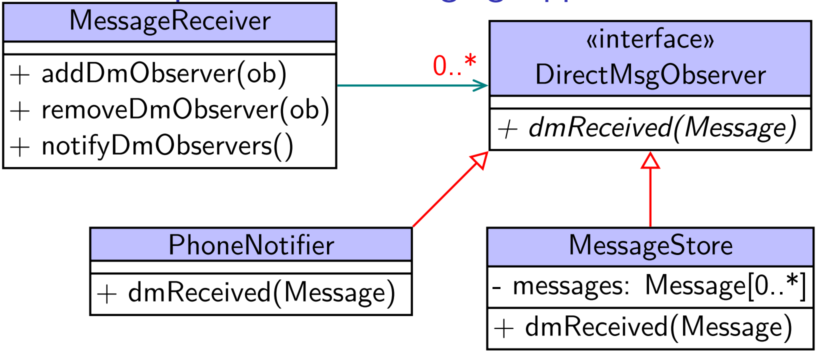
- `notifyObservers()` calls *all* registered observer objects:
  - Instances of `CartViewer` and/or `CostDisplay`.
  - In theory, we could have *multiple* instances of each class too (though less likely in this situation).

Observers in MVC: Example with Parameters



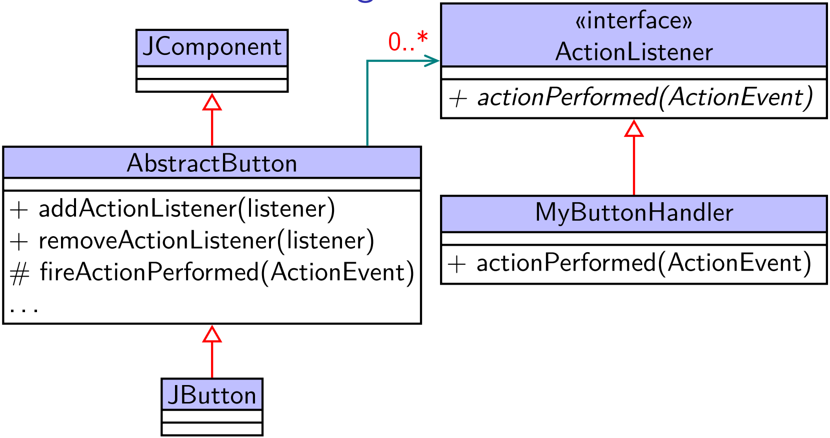
- Instead of back references, the subject can pass parameters.
  - In this case, the entire updated product list.
- Which is best? Depends on the situation.
  - Do all observers need the same information?
  - Is the information simple enough to encode in parameters?

Another Example – Text Messaging App



- Say `MessageReceiver` is responsible for receiving text messages from the network. It passes these onto its observers.
- `PhoneNotifier` displays a popup and/or sounds a ringtone.
- `MessageStore` saves the message for later viewing.
- The `Message` class contains the text, and the sender.

“Listeners” in Java’s Swing GUI



- When using Swing, you inherit from various “listener” (observer) interfaces, like `ActionListener`.
- You then get notified of button presses, and other GUI events.

Exception Observers

- The Observer Pattern could be used in error handling:

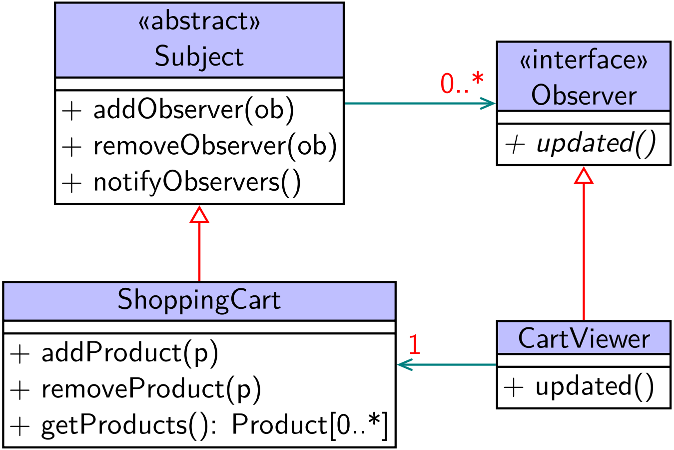
```
try { ... }
catch(MyException e)
{
    notifyErrorObservers(e);
}
```

```
public interface ErrorObserver
{
    void errorHappened(MyException e);
}
```

- Sometimes you can't throw the exception upwards.
  - The caller cannot always deal with it.
- So, you can get observers to respond instead.

Reuse by Inheritance

- Some sources also show inheritance on the subject side.
- Be aware of it, but it's not a universally good idea.

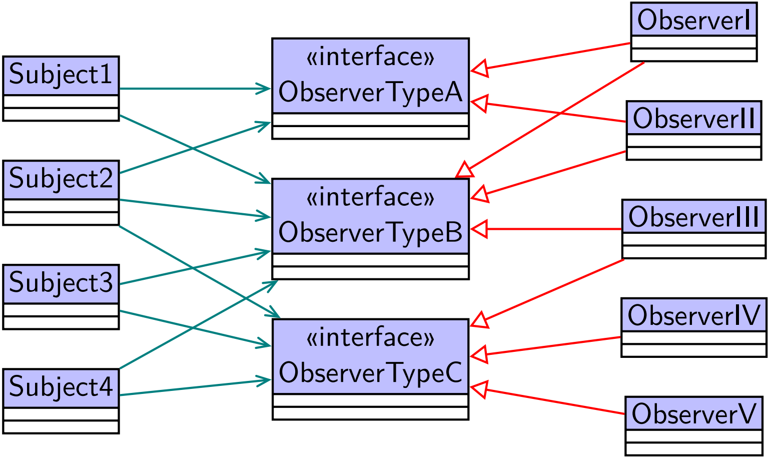


Reuse by Inheritance

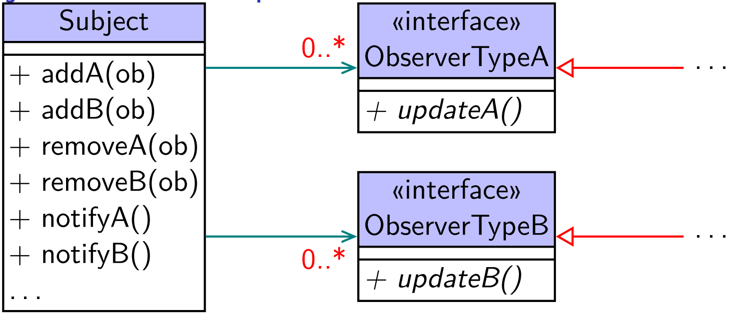
- The subject's add/remove/notifyObserver methods are basically always the same.
  - They're not complicated, but perhaps we'd like to reuse them.
  - So, put them in a common subject superclass.
  - Subclasses acquire them automatically, and simply call `notifyObservers()` when needed.
- *BUT...* this isn't always going to work:
  - What if `ShoppingCart` needs to extend something else?
    - Okay in C++/Python (multiple inheritance), but not Java/C#.
  - What if we want different kinds of events?
    - With a common superclass, there can only be one kind.
    - Real situations can be more complex.
- Besides, this doesn't save much anyway.
  - `addObserver()` and `removeObserver` are one-liners.
  - `notifyObservers()` is a two-liner.

Multiple Everything

- Both subjects and observers may need to deal with multiple event types.

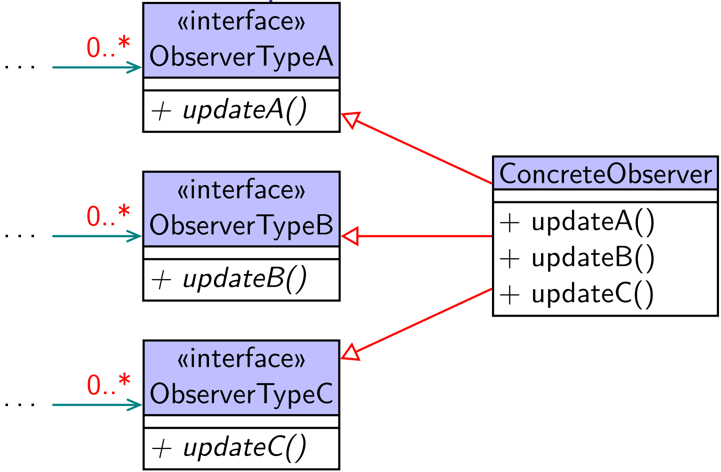


Subjects with Multiple Events



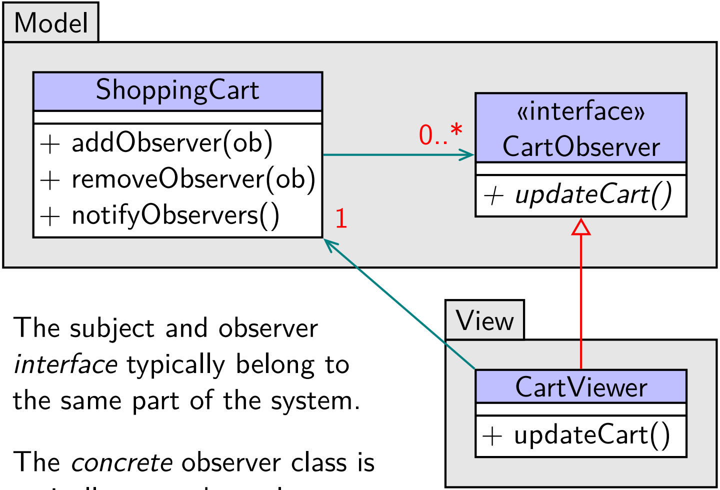
- Each observer interface represents an event type.
- We *could* just have one interface, with both `updateA()` and `updateB()`.
- But that forces the observers to handle both types of events, and they might not need to.
  - But it's still an option to consider.

Observers with Multiple Events



- An observer can observe multiple different event types.
- The “update” methods must be different!

Communication Across Packages



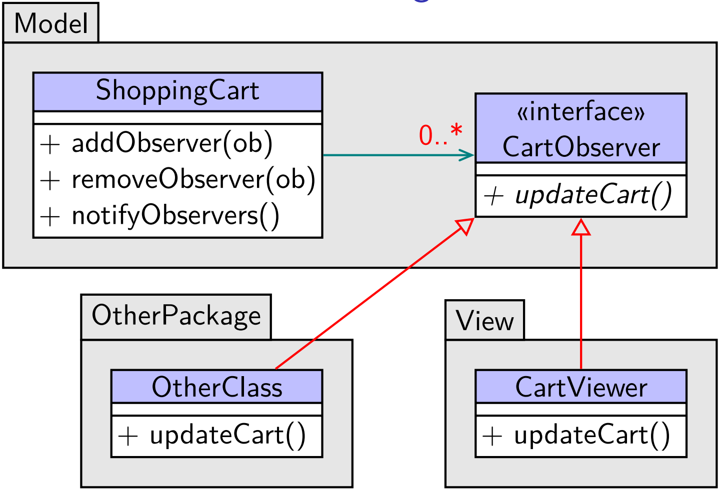
The subject and observer *interface* typically belong to the same part of the system.

The *concrete* observer class is typically somewhere else.

Communication Across Packages

- Why does the interface go with the *subject* (and not with the concrete observer)?
- A matter of minimising coupling.
- The Observer Pattern means the model can update the view *without knowing anything about the view*.
- This wouldn't be true if the interface was part of the view.
  - In that case, `ShoppingCart` would have a direct reference into the view.
- Consider the previous slide:
  - The view only has *outgoing* links to the model.
  - The model does not reference the view at all.

Communication Across Packages



- You could also have concrete observers across multiple packages.

Inner Class Observers

- As described so far, the observer pattern is already quite good at decoupling.
- BUT we can go one better.
- In many case, the observer class itself is responsible for setting up the relationship.
  - e.g. `CartViewer` could call `ShoppingCart.addObserver(this)`, passing a reference to itself to the subject.
- In this case, instead of *being* an observer, `CartViewer` could *contain* it.
  - e.g. `CartViewer` could have a nested or inner class `CartViewer.Observer`.
- But first we'll see the simple case. . .

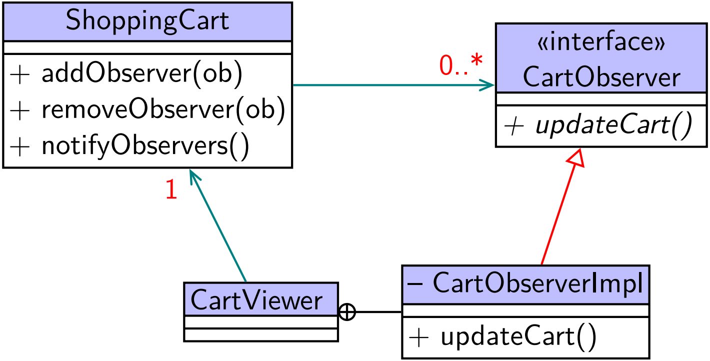
Observer Initialisation

```
public class CartViewer implements CartObserver
{
    private ShoppingCart cart; // Back reference to subject
    ...

    public void setup() // The observer sets itself up here
    {
        cart.addObserver(this);
    }

    @Override
    public void cartUpdated()
    {
        Set<Product> products = cart.getProducts();
        ... // display products
    }
}
```

Using an Inner Class Observer

```java
public class CartViewer
{
    private ShoppingCart cart;
    ...
    public void setup() // Set up an inner class observer
    {
        cart.addObserver(new CartObserverImpl());
    }

    private class CartObserverImpl implements CartObserver
    {
        @Override public void cartUpdated()
        {
            Set<Product> products = cart.getProducts();
            ... // display products
        }
    }
}
```

Inner Class Observers in UML



- `CartViewer` and `CartObserverImpl` are the *same source file*.
- `CartObserverImpl` in particular is not visible to the outside world.
- `CartViewer` uses it internally to receive updates.

Inner vs Ordinary Nested Classes (Implementation Issue)

- Lecture 3 discusses nested classes, and *briefly* introduces inner classes.
- Java's inner classes *don't* have the `static` keyword.
  - Ordinary nested classes *are* `static`.
- An inner class has an implied association to its containing class.
  - Inner class *objects* are each linked to an outer class object.
  - This means you can access outer class fields and methods from inside the inner class.
  - On a previous slide, we access the outer class's "`cart`" field.
  - If needed, we can write "`OuterClass.this`" to refer to the linked outer class object, from within the inner class.
- Alternatively we *can* just use an ordinary nested class.
  - Inside the nested class, just declare a field of the outer class.
  - Makes the association explicit, but achieves the same effect.

Local Classes

- Java, C++ and Python support *local classes*.
  - An entire class defined *inside* a function/method/constructor.
  - Used sparingly, and only for *extremely simple* classes.
  - If in a method or constructor, it can access the outer class's fields.
- Local classes can (with some limitations) access the function/method/constructor's local variables.
  - In Java, these have to be `final` (or "effectively final").
- Why?
  - An additional layer of information hiding.
- Why not?
  - It could blow-out the size of your methods.
  - Don't have large local classes.

Local Classes – Example

```
public class CartViewer
{
    private ShoppingCart cart;
    ...
    public void setup() // Set up an local class observer
    {
        class CartObserverImpl implements CartObserver
        {
            @Override public void cartUpdated()
            {
                Set<Product> products = cart.getProducts();
                ... // display products
            }
        }
        cart.addObserver(new CartObserverImpl());
    }
}
```

Anonymous Classes (Java Only)

- In Java, you can define and instantiate a local class in one go, without even giving it a name – an "anonymous class".

```
cart.addObserver(
    new CartObserver()
    {
        @Override public void cartUpdated()
        {
            Set<Product> products = cart.getProducts();
            ... // display products
        }
    }
);
```

- This defines a nameless class that implements `CartObserver`.

Anonymous Classes – Discussion

- Anonymous class definitions are *expressions*.

```
obj = new TheClass() { ... };
```

```
someMethod(new TheClass() { ... });
```

- Each of these defines and instantiates a class that *extends* or *implements* `TheClass` (which must already exist).
  - i.e. you get an instance of a *subclass* of `TheClass`.
- Within `{ ... }` you can:
  - Override methods defined in `TheClass`.
  - Define new non-abstract methods.
  - Define new fields.
- Restrictions:
  - Constructors are not allowed. (The compiler will generate one.)
  - Subclassing an anonymous class is impossible.

Functions as Objects

- Python treats functions (and methods) as objects.
  - You can pass them by reference to other functions/methods:

```
def f1(): print("Hello!")
def f2(callback):
    callback() # Call the parameter function

f2(f1) # Pass the 1st function to the 2nd
```

- If you remember, standard C does this too:

```
void f2(void(*callback)(void)) {
    (*callback)();
}
...
f2(&f1);
```

- This is a useful callback mechanism – no need for a class.

Closures

Consider this:

```
def getFunction(x):
    def multiplyBy(y): # Nested function
        return x * y
    return multiplyBy  # Return nested function

f = getFunction(3)
print(f(4)) # Prints 12
```

- getFunction() seems to create and return another function.
  - When we call getFunction(3), the result is a function that multiplies something by 3.
  - When we call *that* function with 4, we get 12.
- The returned function multiplyBy() has access to x even after getFunction() has returned.
  - It becomes a "closure" – a function with hidden state.

Closures as Callbacks

- Sometimes, you'd like to provide parameters to a callback "in advance".
  - Perhaps the callback (as before) requires 2 values, *but*...
  - The thing calling it only has one value to supply.
- Closures can provide a simple solution to this.
- They allow a callback function to "remember" things that the caller itself isn't supposed to know.
  - Separation of concerns again!
  - A little bit like `static` local variables in C, but more flexible.

Lambdas in Python

- A lambda is a nameless function, created by a special expression.
- Python has supported them for some time:

```
callback = lambda x, y: x * y
...
print(callback(3, 4)) # Prints 12
```

- A lambda can be used as an *extremely small* callback.
  - Minimal code, avoids clutter.
  - Don't use it if your callback is complex, though.

Lambdas in Java 1.8

- Java 1.8 now has lambdas too:

```
public interface Callback
{
    int call(int x, int y);
}
...
Callback c = (x, y) -> x * y;
...
System.out.println(c.call(3, 4));
```

- The expression above creates an object of type `Callback`.
  - Sort of a compressed version of an anonymous class.
  - The compiler “infers” the various types involved.

Lambdas in C++11

- C++ 2011 has a (surprisingly) sane approach:

```
auto callback = [](int x, int y){ return x * y; };
...
std::cout << callback(3, 4);
```

  - In C++11, the auto keyword causes the compiler to automatically figure out the correct datatype.
- This creates a *functor* – an object that behaves as if it was a function.
  - It automatically overloads the function call operator "()".
- In all languages, lambdas are just "syntactic sugar".
  - You never actually *need* them, *but*...
  - They can reduce the amount of code.

Timers

- Making things happen *at a particular time* is also a classic example of event-driven programming.
  - Deliberate delays and timeouts.
  - Playback of video or audio at a particular speed.
  - Scheduling things to happen at a particular time-of-day; e.g. a calendar notification.
- Most languages know how to time things.
  - You could do it yourself, but it's messy and often inefficient.
- But having something else do it interrupts the normal flow of control; e.g.
  - Your code (e.g. `main()`) sets up the timer.
  - The timer calls your *other code*.
  - You never see what happens in between!

Timers in Java – Example

- Say we want to make an animation. We can get some code to run every $\frac{1}{30}$th of a second like this:

```
import java.util.*;
...
double frameRate = 30.0;
Animator anim = new TimerTask()
{
    @Override
    public void run() { ... } // Update animation
};
Timer timer = new Timer();
timer.schedule(anim, 0.0, 1000.0 / frameRate);
```

- Timer calls `TimerTask.run()`, which you must override.
- *Not* the observer pattern though.

Timers in Python

- Just for completeness, we could do this in Python:

```
from threading import Timer
frameRate = 30.0

def animate():
    ... # Update animation

def time():
    animate()
    t = Timer(1.0 / frameRate, time)
    t.start() # Call time() after 1/30 sec delay.

time()
```

- Python's `Timer` only generates *one* timer event (not repeated ones), so we must reset it each time.