|  |
| --- |
| Circle Language Spec: Events |

## Events

### Introduction

Events are notification calls. Call receivers subscribe to an event. The call sender will call the receivers whenever it feels like it. Another name for events that the world adopts, is *‘the observer pattern’*. The caller decides what the message looks like. The receiver has to supply a command, that has an interface, specified by the caller.

The concept of events is nothing more than a combination of constructs: an array of command references, a command interface and supporting the command interface, registering the command inside the array of command references and then calling all the commands in the array at specific times.

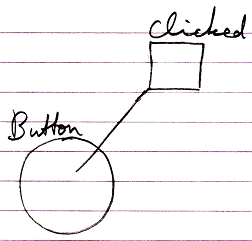
Even though it is just a combination of some other constructs, it is such an important concept, that the new computer language explicitly defines it as a sepate construct, following the footsteps of other programming environments, that did this as well.

The documentation will also show how events look in the diagram notation of the new computer language.

### Prime Event Example: Button Clicked

The key example that explains the purpose of events is a button click. A programmer defines the command to run when the user presses a button. The button click is implemented using the Events construct, so that a programmer can easily stick a procedure under a Button Clicked event without having to worry how the procedure really got invoked.

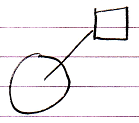
The notation of that is as follows.



### Main Concepts

#### Main Notation of Event

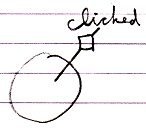
The main notation of an event is as follows:



It shows an object with an event procedure / event command / event implementation associated with it.

#### Event Connector

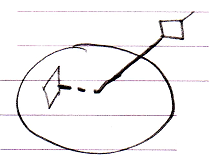
When an event procedure is not filled in, you will only see the event as a connector:



The reason for displaying it like this is that connectors usually look like the result of the system call, but then as a loose end.

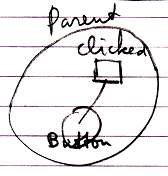
#### Event Raising

The object can raise the event. If you only see the interface of the object you will usually not see this, but if you can see the implementation of the object, its raising the event will look as follows:



#### Call to a Parent

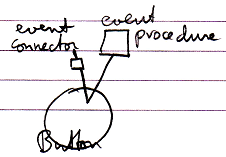
An event is a way for an object to call its parent. So this is what that looks like:



This is the reason why events are displayed like this in the diagram: it is like a child calling a parent.

#### Event Implementation & Connector

When an event procedure is present, the connector to the event procedure is still available so that other procedures can be tied to the event:

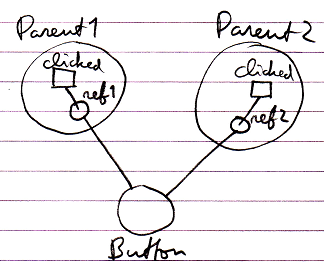


### Event Situations

Several situations are possible in case of events. Events are *multi-cast*, meaning that multiple commands in multiple objects can run in response to the event. Also, a child object is not the only object you can pick up events from. You can also pick up events from deeper objects.

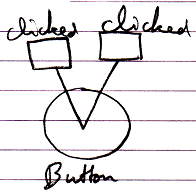
#### Multi-Cast Events

An event can be picked up by multiple parent objects when they both have a reference to the event-casting object and they both define an event command:



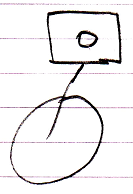
#### Multi-Cast in Same Parent

Multiple commands can be bound to an event, even when they both are defined inside the same parent object:



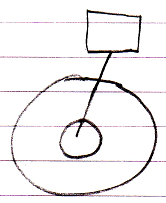
#### Event with Parameters

Events can have parameters. Here is an example of an event with a parameter.



#### Event From Deeper Object

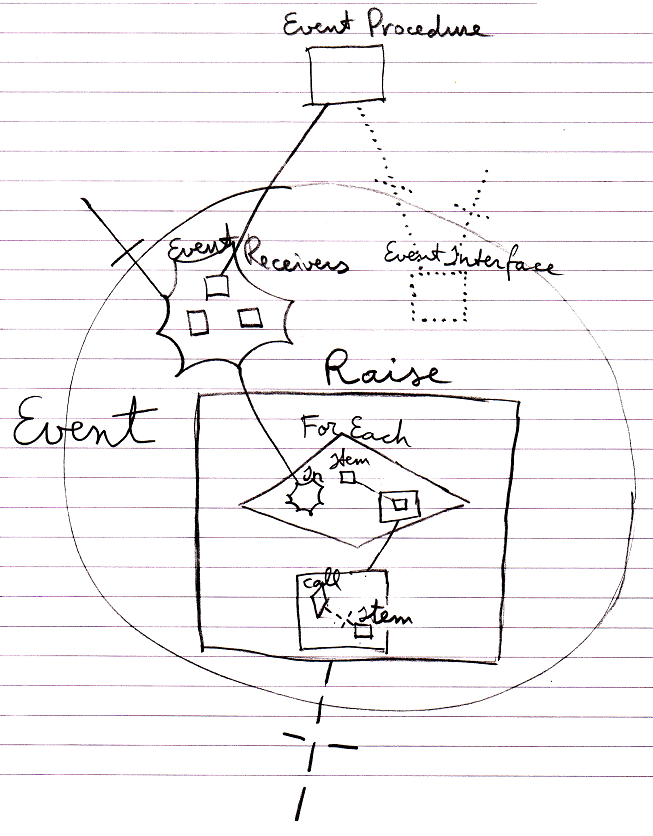
A child object is not the only object you can pick up events from. You can just as easily pick up events from a deeper object.



### Explicit Implementation of Event Object

The concept of events is nothing more than a combination of constructs: an array of command references, a command interface and supporting the command interface, registering the command inside the array of command references and then calling all the commands in the array at specific times.

Explicitly drawing out the event concept using constructs mentioned above, the picture would look like this:



That is the way Event objects are internally implemented.

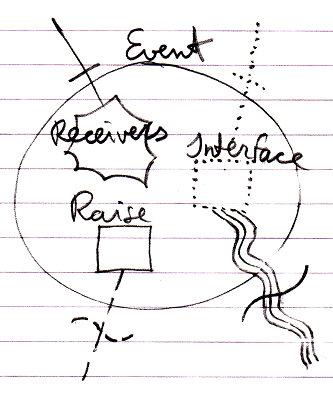
What you see is the registered event implementation (at the top), the collection of event receivers, the event interface, the Raise command and its implementation that loops through the event receiver commands and calls each of them.

Note that even though the concept is that parents can pick up messages from children, it is not the *parents* that are registered, but the *event procedures* that are registered.

Even though the event concept is just a combination of other constructs, it is such an important concept, that the new computer language explicitly defines it as a sepate construct with its own notation.

### Explicit Interface of Event Object

The previous article showed how the implementation of an event object explicitly looks. The interface of an Event object, so the part that other objects interact with, looks as follows:

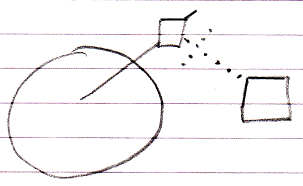


The connectors at the top of the picture are the members that the event *receiver* will interact with: it will *register* its command as a Receiver and it will *implement* the command Interface.

The connectors at the bottom of the picture are the members that the event *sender* will interact with: it will *call* the Raise command of the event and it will also define the *data* that will form the command Interface of the event.

### Event Interface Reference

An event command will always implement an interface specified by the Event object. But if you want to express a command’s implementing the event interface without actually being tied to the event, it would look like this:



So the event connector is there and the command’s interface reference is tied to the event connector.

### Interaction Events

Apart to system events, the next most commonly used group of events may very well be the *interaction events*.

Interaction events are events that occur when a user for instance clicks a button, or double clicks or presses a keyboard key.

This type of event is more of a *controls concept* than a coding concept, but they are mentioned here anyway, because they are such a common use of events

Below you will find a simplified overview of the most common interaction events.

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

### Black Boxed Events

Details about black boxes are covered in the chapter *Black Boxes*.

However, two situations are explicitly denoted here:

- Making events Private

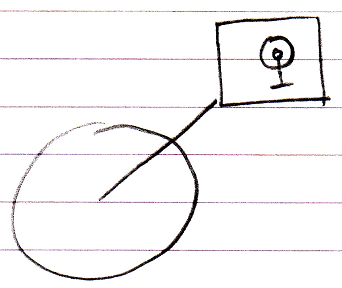
- Accessing the Privates of an event parameter

Just like any other member of an object, events can also be made Private. In that case only Friend objects can pick up this event. Below you will find the involved notations. Details are not covered. More information about black boxing can be found in the *Black Boxing* chapter.

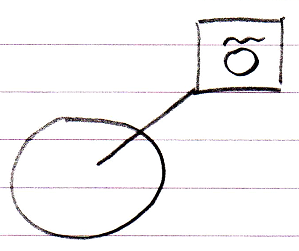
|  |  |  |
| --- | --- | --- |
| Public Event Connector | Friend Event Connector | Private Event Connector |
|  |  |  |
|  |  |  |
| Public Event Connection | Friend Event Connection |  |
|  |  | – |

There are two reasons why the Public event connector does not get an access mark: Public is sort of the default, and the event connection is outward so it does not need an access symbol. The others do need an access mark, because it is the only way to express the access modification.

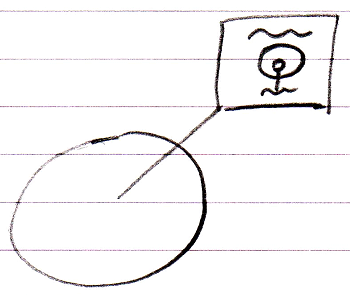
The second black boxing situation to be explicitly mentioned is accessing the Privates of an event parameter. By default you only get to access the Public members of an event parameter. This is what makes it the black box principle.



To be able to access the **Private** members of the event parameter, the event procedure must be declared Friend of the event parameter.



Then the event procedure can access the Private members of the event parameter:



The Friend declaration is an agreement between the event sender and the class of the event parameter. The Friend declaration is part of the event interface, defined by the event sender. So the event receiver can not just declare itself Friend of the event parameter whenever it feels like it.

### Event Alternatives

An event is a means for a child to call the parent. The Events concept works well for this. But there are alternatives for a child to call a parent. A simple command reference would do as well, only this does not allow multi-cast events.

Another alternative is that the child defines an interface, that the parent must support, the parent implements the interface, and the child calls upon the interface of the parent. If you want this to work in a multi-cast situation, you have to program the multi-cast functionality yourself pretty much the same way as an event object implements it (see *Explicit Implementation of Event Object*). Just using standard events might be an easier solution.