EVENTS-FIRST PROGRAMMING IN APP INVENTOR

Franklyn Turbak
Computer Science Department
Wellesley College
fturbak@wellesley.edu

Mark Sherman and Fred Martin
Department of Computer Science
University of Massachusetts Lowell
{msherman, fredm}@cs.uml.edu

David Wolber
Computer Science Department
University of San Francisco
wolber@usfca.edu

Shaileen Crawford Pokress
MIT Media Lab
Massachusetts Institute of Technology
shaileen@media.mit.edu

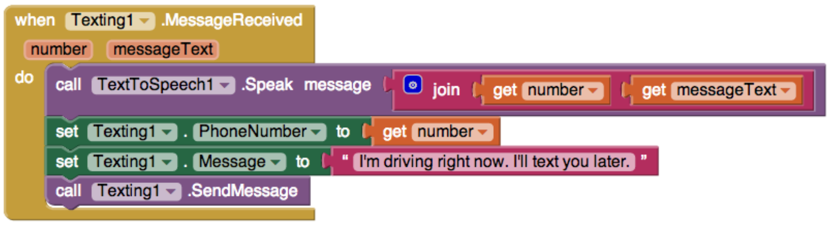
ABSTRACT

Events are a central concept in computer science. They are becoming more important with the prevalence of mobile and web platforms that use event-based programming. Yet, events are typically taught late in the CS curriculum—e.g., in a web programming or operating systems course. We have introduced events to CS0 students from day one using MIT App Inventor, a blocks-based programming environment that enables students to create apps for Android devices. This paper presents the system's event-based model, along with typical coding problems and best-practice approaches for solving them. We advocate for increasing early emphasis on events in the CS curriculum.

INTRODUCTION

The programming model in introductory CS courses is typically based on executing a main program with a single entry point or evaluating expressions in an interpreter. Events and user interfaces are left for later. Event handling, even for simple button clicks, can be remarkably complex (e.g., specifying a listener in Java).

With MIT App Inventor, our CS0 students are immediately introduced to an event-based processing model as they build mobile apps [1]. Their first programs involve specifying how their app should respond to events related to device features, such as touching the screen, shaking the device, changing its location, receiving a text, etc.



```
when Texting1 .MessageReceived
  number  messageText
do  call TextToSpeech1 .Speak message  join  get number
                                             get messageText
    set Texting1 . PhoneNumber to  get number
    set Texting1 . Message to  " I'm driving right now. I'll text you later. "
    call Texting1 .SendMessage
```

Fig. 1: An App Inventor program that speaks and responds to a text message.

Being able to program real-world apps is highly motivating to students and is key to providing a successful first engagement, which ultimately helps to broaden and diversify the pool of new coders [2]. For example, Fig. 1 shows an event handler for an app that receives a text, speaks the message, and sends a reply to it. From simple initial apps like this, students are motivated to try even more complex apps—interactive games, apps that process web data, music recording and playback apps, and many more. Unlike the usual introduction to computing, novices experience the power and the complexities of asynchronous, event-based programming from the beginning.

An events-first approach has been successfully used for many years in Williams College introductory CS courses, where beginners program responses to mouse events in graphics programs during their very first week [3,4,5]. Java libraries hide complexities of the Java event model, whose details are revealed as these courses progress towards traditional Java GUI programming. App Inventor extends this events-first approach by providing primitive event-handler blocks for all types of events that can occur on a mobile device, enabling CS0 students to program responses to general GUI events, sensor events (e.g. device shaking), and social events (e.g., incoming texts) on their very first day. This expands the range of apps students can build and increases their motivation.

In this paper, we explain event-based programming in App Inventor, focusing on a model for events and design patterns for programming with events. We also discuss event-based programming in other languages, particularly Scratch. Space constraints limit our discussion; see the companion technical report for more details [6].

THE APP INVENTOR EVENT MODEL

Programming effectively requires having a good model of a "notional machine"—an explanation of the underlying system that executes the code [7]. One of many challenges that novices face in learning programming is developing such a model. In our experience, it is helpful to explicitly introduce this model when teaching App Inventor.

An App Inventor app consists of a collection of components and a program that specifies the behavior of the components. Components include visible items in the user interface (e.g., buttons, images, and text boxes) and non-visible items used in the app (e.g., camera, GPS sensor). The program is written in a visual blocks-based language in which the programmer connects code fragments that are shaped like puzzle pieces.

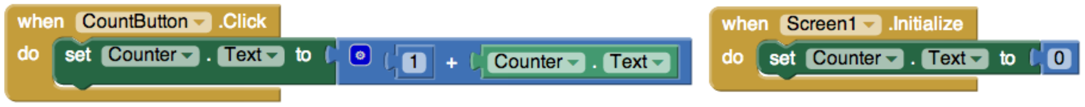

```
when CountButton .Click
do  set Counter . Text to  1 + Counter . Text

when Screen1 .Initialize
do  set Counter . Text to 0
```

Fig. 2: Blocks for an App Inventor counter program.

In App Inventor, all computation is initiated by event handlers associated with components. For example, Fig. 2 shows a program that increments the number in the `Counter` label when the `CountButton` is pressed and resets this label to 0 whenever `Screen1` of the app is visited. The `when CountButton.Click` block is an event handler that executes the event body code (labelled `do`) whenever `CountButton` is pressed. The `Initialize` event handler for a screen is the only thing in App Inventor that resembles a main entry point to a program. However, it rarely does anything beyond initializing the state of the screen's program. The main work of the app is done by all the other event handlers associated with the screen. The app's "program" is its set of event handlers!

Some event handlers have parameters that can be referenced in the event body. In Fig. 1, the `number` parameter is the sender's phone number from the incoming text message and `messageText` is the actual text of the message.

App Inventor has a single-threaded processing model in which only one event handler can be executing at a time. If an event handler executes for a long time, the entire application will appear to be frozen as the system waits for that handler to complete. During this time, new events are queued, and their corresponding handlers will be carried out in order as prior ones complete. However, certain system actions within an event handler—e.g., playing sounds or initiating web requests—are executed as parallel threads by the underlying Android operating system.

App Inventor's notional machine might surprise readers familiar with Scratch [8]. Scratch is also an event-based blocks programming language, but has a multi-threaded model in which multiple events can be executing at the same time. In the Scratch program in Fig. 3, when the green flag is clicked, a meowing sound is played repeatedly. If the sprite image on the screen is then clicked, the sprite moves 5 steps every tenth of a second while the meow sound continues to play. The executions of the play sound and move commands are interleaved, even though they are in different event bodies.

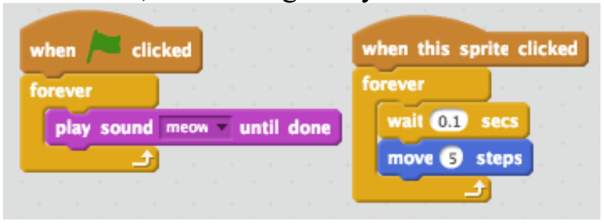


Fig. 3: A Scratch program involving meow sounds and a moving sprite.

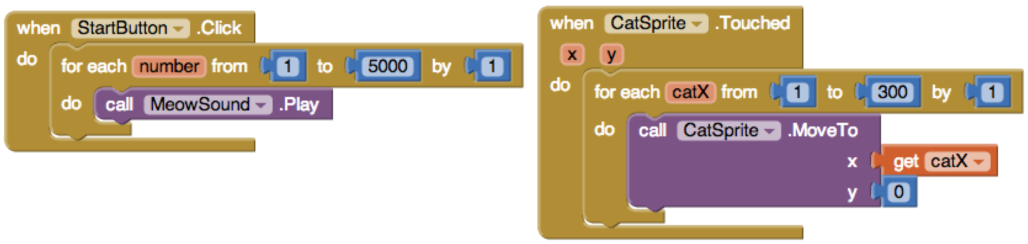
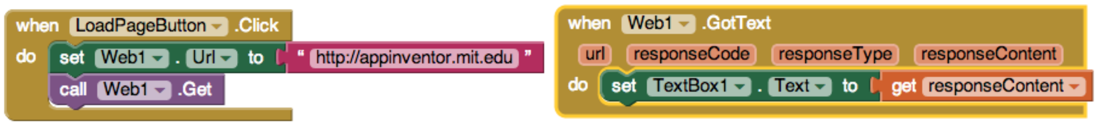

when StartButton .Click
do for each number from 1 to 5000 by 1
do call MeowSound .Play
when CatSprite .Touched
x y
do for each catX from 1 to 300 by 1
do call CatSprite .MoveTo
x get catX
y 0

Fig. 4: An App Inventor program involving meow sounds and a moving sprite.

Fig. 4 shows a similar App Inventor program using finite loops rather than infinite ones. If `StartButton` is clicked and then `CatSprite` is touched immediately afterward, the meow sounds will play to completion before the sprite moves at all. Unlike Scratch, there is no interleaving between actions within loop bodies in different events in App Inventor. (We discuss later how timers can be used to achieve interleaving.) When the sprite does move, it jumps to its final position without any animation; the single-threaded nature of App Inventor prevents updates to the user interface while an event is running. And despite being in a loop that executes 5000 times, the meow sound repeats only about a dozen times. Why? When `MeowSound.Play` is called, the sound plays for a minimum interval, during which other calls to `MeowSound.Play` are ignored.

Many App Inventor component methods that require significant processing time are paired with *callback event handlers* that are triggered when the action initiated by the method completes. This allows other events to be handled while the system is waiting for the result of the method. For example, the `Web` component's `Get` method initiates an HTTP GET request; when a response is received, the `GotText` callback event handler is triggered with the response information. There are roughly two dozen such method/event pairs in App Inventor. Fig. 5 shows an example in which clicking `LoadPageButton` causes the `Web1` component to request a web page. When this page is returned, `Web1`'s `GotText` handler displays the HTML for the page in a text box.

Handlers paired with component actions serve the same role as callback functions and listeners in other models of event-based programming. For instance, in the JavaScript



```
when LoadPageButton .Click
do  set Web1 . Url to " http://appinventor.mit.edu "
    call Web1 .Get
```

```
when Web1 .GotText
 url  responseCode  responseType  responseContent
do  set TextBox1 . Text to  get responseContent
```

Fig. 5: App Inventor program that displays the HTML of a web page in a text box.

jQuery library, HTTP GET requests are initiated via the `get` method, which takes (1) the desired URL and (2) a callback function that specifies what to do with the data returned from the URL. E.g., this code requests a web page and displays its HTML in an alert box:

```
 $.get("http://appinventor.mit.edu", function (page) { alert(page); });
```

In this example, the callback `function (page) { … }` plays the same role as the Web component's `GotText` event handler.

Callbacks also handle events in Java GUI programming, in the form of objects implementing the `ActionListener` interface that are registered with GUI components. The Java code for specifying the behavior of even a simple counter button (like Fig. 2) is notably complex. In contrast, such simple behaviors can be taught in the very first week with App Inventor. In general, App Inventor's paired methods/event handlers hide numerous complexities in the underlying Android operating system and allow complex behaviors, such as web requests, to be taught to students in CS0 or CS1.

LESSONS LEARNED

In practice there are many challenges associated with using events to create App Inventor apps that work as desired. Here, we summarize some of the key lessons we have learned about event-based programming in App Inventor while helping introductory students implement apps they have designed for various projects.

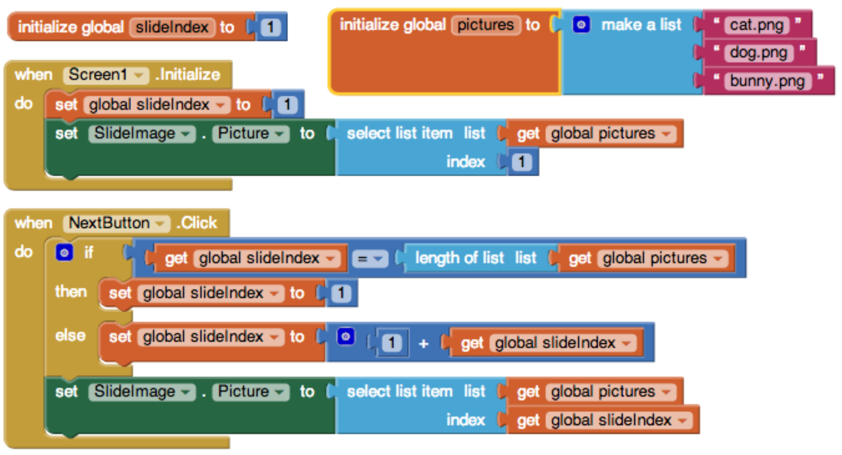
Global State Machines

In App Inventor, many iterative processes that would be expressed with loops in other languages are expressed with an event that performs a single step of the iteration each time it is triggered. State variables that would be local in the loop approach need to be global in the event-based solution.

Consider a simple slideshow app that lets the user navigate through a list of pictures using a **Next** button, wrapping back to the first picture when the last picture is reached. Fig. 6 shows the App Inventor code for this app. The code uses two global variables: (1) `pictures`, which holds a list of the pictures; and (2) `slideIndex`, which holds the index of the picture currently displayed by the `SlideImage` component. For simplicity, we'll assume that `pictures` contains a fixed nonempty list of pictures, but it could be populated by another process (e.g., taking pictures with the camera).

The `Screen1.Initialize` event sets the `slideIndex` to 1 (the first index in an App Inventor list) and displays the initial picture. Clicking `NextButton` increments the global `slideIndex` variable before displaying the picture at that index. There is a special case when `slideIndex` is the last index in `pictures` (in which case it is set to 1).

Although global variables are problematic in many software projects, here they are necessary and appropriate. The `pictures` and `slideIndex` variables must be global because otherwise there is no other way to share their state between different invocations of the `Screen.Initialize` and `NextButton` handlers. Using global variables or component properties (components are also global) to communicate state information

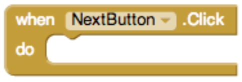


```
initialize global slideIndex to 1

initialize global pictures to make a list "cat.png"
                                          "dog.png"
                                          "bunny.png"

when Screen1 .Initialize
do  set global slideIndex to 1
    set SlideImage . Picture to select list item list get global pictures
                                                 index 1

when NextButton .Click
do  if get global slideIndex = length of list list get global pictures
    then set global slideIndex to 1
    else set global slideIndex to 1 + get global slideIndex
    set SlideImage . Picture to select list item list get global pictures
                                                 index get global slideIndex
```

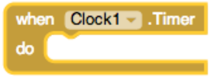
Fig. 6: Blocks for the manually advancing slideshow app.

between event handler invocations is a common pattern in event-based programming in App Inventor. They effectively act like registers in a state-based abstract machine.

Timers

App Inventor's `Clock` component provides a `Timer` event that fires repeatedly with a specified interval as long as the component is enabled. The manually advancing slideshow presented above can be changed to an automatically advancing one simply by replacing 

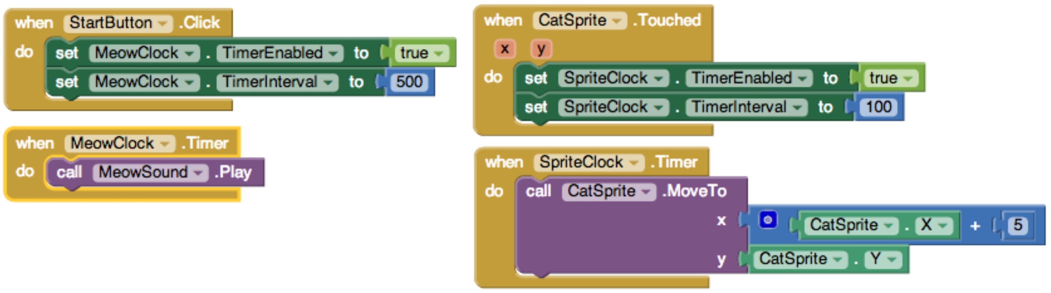
```
when NextButton .Click
do
```

by the clock event handler block 

```
when Clock1 .Timer
do
```

.

Timers are used when a component needs to be regularly updated, such as animating a sprite or repeatedly playing a sound. For example, Fig. 7 shows an App Inventor program for the "play a meow sound repeatedly" and the "move a sprite smoothly across the screen" behaviors that were programmed quite differently in the Scratch programs first shown in Fig. 3. The timer interval for `MeowClock` is chosen to be



```
when StartButton .Click
do  set MeowClock . TimerEnabled to true
    set MeowClock . TimerInterval to 500

when MeowClock .Timer
do  call MeowSound .Play

when CatSprite .Touched
  x  y
do  set SpriteClock . TimerEnabled to true
    set SpriteClock . TimerInterval to 100

when SpriteClock .Timer
do  call CatSprite .MoveTo
                   x CatSprite . X + 5
                   y CatSprite . Y
```

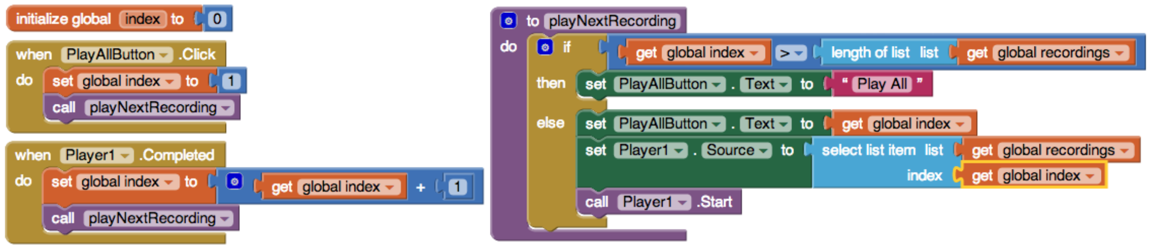
Fig 7: Expressing the interleaved Scratch events from Fig. 3 in App Inventor.

about the duration of the meow sound clip. The meow sound and animation are interleaved because each timer event handler holds onto the single thread of control for only a short duration, and timer events that happen to fire during this period are queued.

Timers are also used in situations where an event handler needs to stall for a specified duration while letting other handlers run. Modifying the `TimerInterval` property of the `SpriteClock` in Fig. 7 is equivalent to modifying the `wait` argument in Fig 3. Busy waits should be avoided because the single-threaded event model means that no other handlers can make progress while a loop executes.

Callback Event Handlers

A component method with an associated callback handler can be invoked multiple times in a program. Correctly handling all these invocations with a single callback handler can be tricky. For example, consider the program in Fig. 8. When the **Play All**



```
initialize global index to 0

when PlayAllButton.Click
do  set global index to 1
    call playNextRecording

when Player1.Completed
do  set global index to  get global index + 1
    call playNextRecording

to playNextRecording
do  if   get global index > length of list list  get global recordings
    then set PlayAllButton.Text to " Play All "
    else set PlayAllButton.Text to get global index
         set Player1.Source to select list item list  get global recordings
                                                index get global index
         call Player1.Start
```

Fig 8: A program that plays all of the sound recordings in a list.

button is clicked, it plays all the sound files in the global `recordings` list in order, one after the other. During this process, the label on the button changes to be the index of the current recording. The `Player1.Completed` callback event, which is triggered when the current sound file is done playing, is used in conjunction with the global `index` variable to guarantee that the current sound file plays to completion before the next sound file in the sequence is played. This is similar to the `NextButton.Click` and `Clock1.Timer` examples from above, except that the user presses the button to play the first recording, and the `Player1.Completed` callback event effectively "presses the button" for the next recording when the previous one is finished.

This pattern of using a callback event handler in conjunction with additional state can be used to synchronize otherwise asynchronous actions and to demultiplex multiple invocations of a callback event handler. See [6] for more details.

LOOKING AHEAD

We see several ways to improve the way that users are introduced to App Inventor's event-based notional machine. App Inventor does allow "watching" a code block, which displays a bubble showing a result as the block is executed. This could be extended to better visualize the internal state of the program, along the lines of tools like Python Tutor [9]. Such tools provide step-by-step processing and a visualization of the hidden data on the run-time stack, all in a fashion more palatable to beginners than typical debuggers. App Inventor could provide similar stepping facilities, and also provide a representation of the event queue and a programmer-controlled run-time clock, so users could see how events work in slow motion.

Programs like those in Fig. 8 could be simplified if App Inventor had a callback mechanism that could maintain the state associated with a particular method invocation in the callback closure. Many examples of global state could be made more local with callback closures. But such a mechanism would have to be designed with novices in mind.

Finally, we need to improve how event-based programming is taught in App Inventor. We currently teach the above strategies for programming with events largely on an ad hoc basis: when students encounter problems with events in their projects, we guide them through a solution process. We plan to develop a set of standard examples with associated instructional materials that explain App Inventor's notional machine for events and present design patterns for addressing typical challenges that arise in event-based programming in App Inventor.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant Numbers 1225680, 1225719, 1225745, 1225976, and 1226216

REFERENCES

[1] Wolber, D., Abelson, H., Spertus, E., and Looney, L., *App Inventor*, O'Reilly, 2011.
[2] Wagner, A., Gray, J., Corley, J., and Wolber, D., Using App Inventor in a K-12 Summer Camp, *SIGCSE '13*, 621--626.
[3] Bruce, K.. Danyluk, A., and Murtagh, T., Event-driven Programming is Simple Enough for CS1, *ITiCSE '01*, 1--4.
[4] Bruce, K., Danyluk, A., and Murtagh, T., *Java: An Eventful Approach*, Pearson, 2006.
[5] Murtagh, T., Weaving CS into CS1: A Doubly Depth-first Approach, *SIGCSE '07*, 336--340.
[6] Turbak, F., Sherman, M., Martin, F., Wolber, D., and Pokress, S., Events-First Programming in App Inventor, MIT Center For Mobile Learning Technical Report MIT-CML-TR-2014-001.
[7] du Boulay, B., Some Difficulties of Learning to Program, *Journal of Educational Computing Research*, 2, (1), 57–73, 1986.
[8] Resnick, M., Maloney, J., Monroy-Hernandez, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B., and Kafai, Y., Scratch: Programming for All, *CACM*, 52, (11), 60-67, Nov. 2009.
[9] Guo, P., Online Python Tutor: Embeddable Web-based Program Visualization for CS Education, *SIGCSE '13*, 579--584.