Class: **AdventureActor**

Superclass: AdventureRobot
Category: Adventure-Acting
Instance variables: behaviorBlock script reactionDelay thinkDelay partner biography pocket obscenityResponses nonsenseResponses timeoutResponses scriptTraceEnabled

I'm an interactive autonomous character in the adventure. I follow a state machine called a script. My script responds to events generated by cues.

Instance Variables:
behaviorBlock <BlockClosure> that my process uses for its behavior. This is in an instance variable so that I can change its value as I am paused/resumed.
script <AdventureScript> that I'm following.
reactionDelay <Delay> between my turns.
*thinkDelay <Delay> from the time I notice that *some* cue has triggered to the time I actually find out what cue has triggered and react to it. Note that I wait this delay without turn synchronization, so the rest of the game (and the player) is not held up. That means that I may have noticed cue "A" triggering, but by the time I try to do something about it, cue "B" has triggered, and I react to it instead. So, for example, a player may say something interesting, which I notice, then leave. Since "disappearance" cues are generally tested first, I would notice the speech, but react to the disappearance.*
partner <AdventurePlayer> that I'm interacting with. This is set by the cue that first notices the player.
biography <String> The story of my life.
pocket <IdentityDictionary> of AdventureThing's. I can use this to hold anything, indexing its contents with a Symbol. The contents of #pocket are invisible, and completely under my control. Messages such as #adventure: and #preSave are forwarded to the contents of pocket, so it should contain AdventureThing's or AdventureProperty's.
obscenityResponses <AdventureRandomValueHolder> of String's to say when I hear an obscenity.
nonsenseResponses <AdventureRandomValueHolder> of String's to say when I hear something I don't understand.
timeoutResponses <AdventureRandomValueHolder> of String's to say when I hear nothing when I'm expecting to hear something.
scriptTraceEnabled <Boolean> True if I am tracing my script's transitions.

behavior

bootBehaviorBlock
```
^
[self
    executeTurn:
        [self currentState allCues: #reset.
        self currentState allCues: #activate.
        self script boot].
self behaviorBlock: self restartBehaviorBlock.
self behaviorBlock value]
```

currentState
```
^self script currentState
```

reactToAnyTriggeredCue
```
| triggered |
(triggered := self currentState triggeredCue) notNil
    ifTrue:
        [self script event: triggered machineEvent.
        self currentState allCues: #reset]
```

restartBehaviorBlock
```
^[
[self reactionDelay wait.
(self executeTurn: [self somethingTriggered])
    ifTrue:
        [self thinkDelay wait.
        self
            executeTurn:
                [self currentState allCues: #deactivate.
                self reactToAnyTriggeredCue.
                self currentState allCues: #activate]]] repeat]
```

somethingTriggered
```
^self currentState triggeredCue notNil
```

initialize-release

id: idSymbol name: nameString description: descriptionString biography: bioString
```
self id: idSymbol; name: nameString; description: descriptionString; biography: bioString
```

initialize
```
super initialize.
self isMoveable: false.
self behaviorBlock: self bootBehaviorBlock
```

forwarding

performAndForward: aSymbol performFirst: aBoolean
```
self
    performAndForward: aSymbol
    performFirst: aBoolean
    to: self pocket
```

performAndForward: aSymbol with: aParameter performFirst: aBoolean
```
self
    performAndForward: aSymbol
    with: aParameter
    performFirst: aBoolean
    to: self pocket
```

accessing

behaviorBlock
```
^behaviorBlock
```

behaviorBlock: aBlock
```
behaviorBlock := aBlock
```

```
biography
	^biography

biography: aString
	biography := aString

nonsenseResponses
	^nonsenseResponses isNil
		ifTrue: [nonsenseResponses := AdventureRandomValueHolder withAll: #(
				'Say what?'
				'Speak up, you''re mumbling!'
				'I missed that.'
				'Pardon me?'
				'English as a second language is a bitch, huh?!'
				'Could you re-phrase that?'
				'No comprendo.'
			)]
		ifFalse: [nonsenseResponses]

nonsenseResponses: aRandomValueHolder
	nonsenseResponses := aRandomValueHolder

obscenityResponses
	^obscenityResponses isNil
		ifTrue: [obscenityResponses := AdventureRandomValueHolder withAll: #(
				'Shame shame shame!'
				'You''re going straight to hell!'
				'Why I outta wash your mouth out with soap!'
				'Wow. Such imaginative language.'
				'How rude!'
				'Don''t use that kind of language with me!'
				'Shocking!'
			)]
		ifFalse: [obscenityResponses]

obscenityResponses: aRandomValueHolder
	obscenityResponses := aRandomValueHolder

partner
	^partner

partner: aPlayer
	partner := aPlayer

pocket
	^pocket isNil
		ifTrue: [pocket := IdentityDictionary new]
		ifFalse: [pocket]

pocket: anIdentityDictionary
	pocket := anIdentityDictionary

reactionDelay
	^reactionDelay isNil
		ifTrue: [reactionDelay := Delay forMilliseconds: 500]
```

```
        ifFalse: [reactionDelay]

reactionDelay: aDelay
    reactionDelay := aDelay

script
    ^script isNil
        ifTrue: [script := self buildScript]
        ifFalse: [script]

script: aScript
    script := aScript

scriptTraceEnabled
    ^scriptTraceEnabled isNil
        ifTrue: [scriptTraceEnabled := false]
        ifFalse: [scriptTraceEnabled]

scriptTraceEnabled: aBoolean
    scriptTraceEnabled := aBoolean

thinkDelay
    ^thinkDelay isNil
        ifTrue: [thinkDelay := Delay forMilliseconds: 1000]
        ifFalse: [thinkDelay]

thinkDelay: aDelay
    thinkDelay := aDelay

timeoutResponses
    ^timeoutResponses isNil
        ifTrue: [timeoutResponses := AdventureRandomValueHolder withAll: #(
                'Speak up!'
                'Don''t be shy.'
                'I can''t hear you.'
                'Please say something!'
                'Talk to me!'
                'Say something, already!'
                'Aren''t you paying attention?'
                'Don''t make me repeat myself.'
            )]
        ifFalse: [timeoutResponses]

timeoutResponses: aRandomValueHolder
    timeoutResponses := aRandomValueHolder
```

script building

```
addDefaultSpeechHandlersTo: aState
    self addObscenityHandlerTo: aState.
    self addNonsenseHandlerTo: aState

addNonsenseHandlerTo: aState
    self
        addNonsenseHandlerTo: aState
        successBlock: [:aScript :anEvent | self say: self nonsenseResponses value]
```

```
        successState: nil
```

addNonsenseHandlerTo: aScriptState successBlock: sBlockOrNil successState: sStateOrNil

```
    aScriptState addEntryBlock: [:aScript |
        self nonsenseResponses reset].

    aScriptState
        addCue: (
            AdventureSpeechCue
                machineEvent: (MachineEvent named: #AdventureActorNonsense)
                actor: self
                matcher: '*'
        ).

    aScriptState
        add: (
            (MachineTransitions forEvent: #AdventureActorNonsense)
                add: (
                    MachineTransition new
                        actionBlock: sBlockOrNil;
                        nextState: sStateOrNil
                )
        )
```

addObscenityHandlerTo: aState

```
    self
        addObscenityHandlerTo: aState
        successBlock: [:aScript :anEvent | self say: self obscenityResponses value]
        successState: nil
```

addObscenityHandlerTo: aState successBlock: sBlockOrNil successState: sStateOrNil

```
    aState addEntryBlock: [:aScript |
        self obscenityResponses reset].

    aState addCue: (
        AdventureObscenitySpeechCue
            machineEvent: (MachineEvent named: #AdventureActorObscenity)
            actor: self
    ).

    aState add: (
        (MachineTransitions forEvent: #AdventureActorObscenity)
            add: (
                MachineTransition new
                    actionBlock: sBlockOrNil;
                    nextState: sStateOrNil
            )
    ).
```

addPlayerAppearanceHandlerTo: aScriptState successBlock: sBlockOrNil successState: sStateOrNil

```
    aScriptState
        addCue: (
```

```
            AdventurePlayerAppearanceCue
                machineEvent: (MachineEvent named: #AdventureActorPlayerAppearance)
                actor: self
        ).

    aScriptState
        add: (
            (MachineTransitions forEvent: #AdventureActorPlayerAppearance)
                add: (
                    MachineTransition new
                        actionBlock: sBlockOrNil;
                        nextState: sStateOrNil
                )
        )
```

addPlayerDisappearanceHandlerTo: aScriptState successBlock: sBlockOrNil successState: sStateOrNil

```
    aScriptState
        addCue: (
            AdventurePlayerDisappearanceCue
                machineEvent: (MachineEvent named: #AdventureActorPlayerDisappearance)
                actor: self
        ).

    aScriptState
        add: (
            (MachineTransitions forEvent: #AdventureActorPlayerDisappearance)
                add: (
                    MachineTransition new
                        actionBlock: sBlockOrNil;
                        nextState: sStateOrNil
                )
        )
```

addPlayerPresenceHandlerTo: aScriptState successBlock: sBlockOrNil successState: sStateOrNil

```
    aScriptState
        addCue: (
            AdventurePlayerPresenceCue
                machineEvent: (MachineEvent named: #AdventureActorPlayerPresence)
                actor: self
        ).

    aScriptState
        add: (
            (MachineTransitions forEvent: #AdventureActorPlayerPresence)
                add: (
                    MachineTransition new
                        actionBlock: sBlockOrNil;
                        nextState: sStateOrNil
                )
        )
```

addQuestionHandlerTo: aState questions: qCollection answers: aCollection disappearanceBlock: dBlockOrNil disappearanceState: dStateOrNil failureBlock: fBlockOrNil failureState: fStateOrNil

```
	"Add a question handler using common defaults."

	self
		addQuestionHandlerTo: aState
		questions: qCollection
		answers: aCollection
		timeoutResponseHolder: self timeoutResponses
		nonsenseResponseHolder: self nonsenseResponses
		obscenityResponseHolder: self obscenityResponses
		chances: self defaultChances
		timeout: self defaultTimeout
		disappearanceBlock: dBlockOrNil
		disappearanceState: dStateOrNil
		failureBlock: fBlockOrNil
		failureState: fStateOrNil
```

addQuestionHandlerTo: aState questions: qCollection answers: aCollection timeoutResponseHolder: tHolder nonsenseResponseHolder: nHolder obscenityResponseHolder: oHolder chances: cInteger timeout: tInteger disappearanceBlock: dBlockOrNil disappearanceState: dStateOrNil failureBlock: fBlockOrNil failureState: fStateOrNil

```
"Timeout is in seconds. All blocks here are state machine actionBlock's, so they take two parameters:
[:aScript :anEvent]. Event names are prefixed here to avoid possible conflict with user-supplied event
names in answers."

| questions retryCount |
questions := (AdventureRandomValueHolder withAll: qCollection).
retryCount := AdventureRetryCount new.

"Blocks"

	aState
		addEntryBlock: [:machine |
			retryCount resetTo: cInteger - 1.
			self say: questions value].

"Cues"

	aState
		"Begin cues in order"
		addCue: (
			AdventurePlayerDisappearanceCue
				machineEvent: (MachineEvent named:
					#AdventureActorPlayerDisappearance)
				actor: self
		).

	aCollection do: [:anAnswer |
		aState addCue: anAnswer cue].

	aState
		addCue: (
			AdventureObscenitySpeechCue
				machineEvent: (MachineEvent named: #AdventureActorObscenity)
				actor: self
		);
```

```
        addCue: (
            AdventureSpeechCue
                machineEvent: (MachineEvent named: #AdventureActorNonsense)
                actor: self
                matcher: '*'
        );

        addCue: (
            AdventureTimeoutCue
                machineEvent: (MachineEvent named: #AdventureActorTimeout)
                seconds: tInteger
        ).
        "End cues in order"

"Transitions"

    aState
        add: (
            (MachineTransitions forEvent: #AdventureActorPlayerDisappearance)
                add: (
                    MachineTransition new
                        actionBlock: dBlockOrNil;
                        nextState: dStateOrNil
                )
        ).

    aCollection do: [:anAnswer |
        aState add: (
            (MachineTransitions forEvent: anAnswer cue machineEvent name)
                add: (
                    MachineTransition new
                        actionBlock: anAnswer actionBlock;
                        nextState: anAnswer nextState
                )
        )
    ].

    aState
        add: (
            (MachineTransitions forEvent: #AdventureActorTimeout)
                "Begin transitions in order"
                add: (
                    MachineTransition new
                        guardBlock: [:aScript :anEvent | retryCount decrementAndTest];
                        actionBlock: fBlockOrNil;
                        nextState: fStateOrNil
                );

                add: (
                    MachineTransition new
                        actionBlock: [:aScript :anEvent |
                            self say: tHolder value]
                )
                "End transitions in order"
        );
```

```
        add: (
            (MachineTransitions forEvent: #AdventureActorObscenity)
                "Begin transitions in order"
                add: (
                    MachineTransition new
                        guardBlock: [:aScript :anEvent | retryCount decrementAndTest];
                        actionBlock: fBlockOrNil;
                        nextState: fStateOrNil
                );

                add: (
                    MachineTransition new
                        actionBlock: [:aScript :anEvent |
                            self say: oHolder value]
                )
                "End transitions in order"
        );

        add: (
            (MachineTransitions forEvent: #AdventureActorNonsense)
                "Begin transitions in order"
                add: (
                    MachineTransition new
                        guardBlock: [:aScript :anEvent | retryCount decrementAndTest];
                        actionBlock: fBlockOrNil;
                        nextState: fStateOrNil
                );

                add: (
                    MachineTransition new
                        actionBlock: [:aScript :anEvent |
                            self say: nHolder value]
                )
                "End transitions in order"
        )
```

addTimeoutHandlerTo: aScriptState seconds: anInteger successBlock: sBlockOrNil successState: sStateOrNil

```
    aScriptState
        addCue: (
            AdventureTimeoutCue
                machineEvent: (MachineEvent named: #AdventureActorTimeout)
                seconds: anInteger
        ).

    aScriptState
        add: (
            (MachineTransitions forEvent: #AdventureActorTimeout)
                add: (
                    MachineTransition new
                        actionBlock: sBlockOrNil;
                        nextState: sStateOrNil
                )
        )
```

addWaitHandlerTo: waitState successBlock: sBlockOrNil successState: sStateOrNil disappearanceBlock: dBlockOrNil disappearanceState: dStateOrNil seconds: anInteger

```
	"Add to waitState so I wait anInteger seconds, then make the transition
	to sStateOrNil, executing sBlockOrNil (:aScript :anEvent). If my partner
	disappears, execute aDisappearanceBlockOrNil (:aScript :anEvent)
	and change to dStateOrNil."

	waitState
		"Begin cues in order"
		addCue: (
			AdventurePlayerDisappearanceCue
				machineEvent: (MachineEvent named: #AdventureActorPlayerDisappearance)
				actor: self
		);

		addCue: (
			AdventureTimeoutCue
				machineEvent: (MachineEvent named: #AdventureActorTimeout)
				seconds: anInteger
		).
		"End cues in order"

	waitState
		add: (
			(MachineTransitions forEvent: #AdventureActorTimeout)
				add: (
					MachineTransition new
						actionBlock: sBlockOrNil;
						nextState: sStateOrNil
				)
		);

		add: (
			(MachineTransitions forEvent: #AdventureActorPlayerDisappearance)
				add: (
					MachineTransition new
						actionBlock: dBlockOrNil;
						nextState: dStateOrNil
				)
		)
```

buildScript

```
	"This script does nothing."

	| myScript |
	myScript := AdventureScript named: #actorDefaultScript.
	myScript add: (AdventureScriptState named: #idle actor: self).
	^myScript
```

defaultChances

```
	^3
```

defaultTimeout

```
	^16
```

menu items

```
addDebugMenuItemsFor: aPlayer to: aCollection
    self singleSteppingEnabled ifFalse: [aCollection add: (AdventureMenuItem
                label: 'enable single stepping'
                block: [self enableSingleStepping]
                type: #debug)].
    aCollection add: (self scriptTraceEnabled
            ifTrue: [AdventureMenuItem
                    label: 'disable script trace'
                    block: [self disableScriptTrace]
                    type: #debug]
            ifFalse: [AdventureMenuItem
                    label: 'enable script trace'
                    block: [self enableScriptTrace]
                    type: #debug]).
    super addDebugMenuItemsFor: aPlayer to: aCollection

addMenuItemsFor: aPlayer to: aCollection
    aCollection add: (AdventureMenuItem label: 'read biography' block: [AdventureActorBiographyViewer
openOn: self]).
    super addMenuItemsFor: aPlayer to: aCollection
```

script tracing

```
disableScriptTrace
    self script retractInterestIn: #transition for: self.
    self scriptTraceEnabled: false

enableScriptTrace
    self script
        expressInterestIn: #transition
        for: self
        sendBack: #traceScriptTransition:.
    self scriptTraceEnabled: true

traceScriptTransition: anArray
    (anArray at: 1)
        traceTransitionFrom: (anArray at: 2)
        to: (anArray at: 3)
        triggeredBy: (anArray at: 4)
        on: Transcript
```

MetaClass: AdventureActor class

instance creation

```
id: idSymbol name: nameString description: descriptionString biography: bioString
    ^(self new) id: idSymbol; name: nameString; description: descriptionString; biography: bioString
```

Class: **AdventureActorCue**

Superclass: AdventureCue
Category: Adventure-Acting
Instance variables: actor

I'm an asbstract superclass for cues that need to know the actor they'll be cueing.

Instance Variables:
actor <AdventureActor> that I'll be cueing. This is set at my creation time.

accessing

actor
^actor

actor: anActor
actor := anActor

MetaClass: AdventureActorCue class

instance creation

machineEvent: aMachineEvent actor: anActor
^(self new) machineEvent: aMachineEvent; actor: anActor

Class: **AdventureAffirmativeSpeechCue**

Superclass: AdventureHardWiredSpeechCue
Category: Adventure-Acting

I trigger when the player answers in the affirmative.

MetaClass: AdventureAffirmativeSpeechCue class

resources

buildMatcher
    ^AdventureWordMatcher withAll: #('yes' 'yup' 'yeah')

Class: **AdventureAlwaysCue**

Superclass: AdventureCue
Category: Adventure-Acting

I am always triggered. I'm used to kick-start scripts.

polling

isTriggered
    ^true

MetaClass: AdventureAlwaysCue class

Class: **AdventureANDCue**

Superclass: AdventureLogicalCue
Category: Adventure-Acting

I trigger on the logical AND of my cues.

logical operation

```
logicalOperationOn: aBoolean1 with: aBoolean2
    ^aBoolean1 & aBoolean2
```

MetaClass: AdventureANDCue class

Class: **AdventureAnswer**

Superclass: Object
Category: Adventure-Acting
Instance variables: cue actionBlock nextState

I'm an answer as defined by an AdventureActor's question. I know what speech cue I must match, and the nextState and actionBlock fields that should be used in the question's transition.

Instance Variables:
cue <AdventureSpeechCue> The cue that's tested by the question. The cue's eventName can be any symbol, subject to the restriction that symbols in the cues of a collection of AdventureAnswer's should be unique.
actionBlock <BlockClosure> This and nextState are used to make the transition when my #cue returns true from #isTriggered.
nextState <AdventureScriptState>

accessing

actionBlock
^actionBlock

actionBlock: aBlockOrNil
actionBlock := aBlockOrNil

cue
^cue

cue: aSpeechCue
cue := aSpeechCue

nextState
^nextState

nextState: aStateOrNIl
nextState := aStateOrNIl

initialize-release

cue: aCue actionBlock: aBlockOrNil nextState: aStateOrNil
self cue: aCue;
actionBlock: aBlockOrNil;
nextState: aStateOrNil

MetaClass: AdventureAnswer class

instance creation

```
cue: aCue actionBlock: aBlockOrNil nextState: aStateOrNil
    ^self new
        cue: aCue
        actionBlock: aBlockOrNil
        nextState: aStateOrNil
```

Class: **AdventureComponentCue**

Superclass: AdventureEventCue
Category: Adventure-Acting
Instance variables: componentId

I trigger when I get an event with matching event type from the component with a component id that matches my #componentId.

Instance Variables:
componentId *<Symbol> Id of the adventure component that I match. That component must be the source of the event for me to trigger.*

events

matches: anEvent
```
"Answer whether anEvent matches my criteria."

^(super matches: anEvent)
    & (self componentId == anEvent source id)
```

accessing

componentId
```
^componentId
```

componentId: aSymbol
```
componentId := aSymbol
```

printing

printDetailsOn: aStream
```
super printDetailsOn: aStream.
self componentId printOn: aStream.
aStream space
```

MetaClass: AdventureComponentCue class

instance creation

machineEvent: aMachineEvent actor: anActor eventType: eSymbol componentId: cSymbol
```
^(self new) machineEvent: aMachineEvent; actor: anActor; eventType: eSymbol; componentId: cSymbol
```

Class: **AdventureCue**

Superclass: Object
Category: Adventure-Acting
Instance variables: machineEvent

I'm the abstract superclass for all cues.

Instance Variables
machineEvent <MachineEvent> I don't use this; I merely store it so that when an actor polls me, it knows what event to drop into it script state machine.

Subclasses must implement methods for
polling
isTriggered

polling

isTriggered
self subclassResponsibility

accessing

machineEvent
^machineEvent

machineEvent: aMachineEvent
machineEvent := aMachineEvent

activation-deactivation

activate
"Become active. This is the hook to connect to any part of the environment that interests me."
"super activate."

^self

deactivate
"Become inactive. This is the hook to disconnect from anything I connected to in #activate."
"super deactivate"

^self

resetting

reset
"Prepare myself to be activated and deactivated in a cycle."

```
    ^self
```

printing

```
printDetailsOn: aStream
    "Extend in subclasses to write instance details to aStream."
    "super printDetailsOn: aStream"

    ^self

printOn: aStream
    super printOn: aStream.
    aStream nextPut: $(.
    self printDetailsOn: aStream.
    aStream nextPut: $)
```

initialize-release

```
initialize
    ^self
```

MetaClass: AdventureCue class

instance creation

```
machineEvent: aMachineEvent
    ^self new machineEvent: aMachineEvent

new
    ^super new initialize
```

Class: **AdventureEventCue**

Superclass: AdventureActorCue
Category: Adventure-Acting
Instance variables: isTriggered eventType

I trigger if I get an event that matches my eventType.

Instance Variables
isTriggered <Boolean> True if an event has triggered me.
eventType <Symbol> Type of event that I cue on.

accessing

eventType
^eventType

eventType: aSymbol
eventType := aSymbol

isTriggered
^isTriggered

isTriggered: aBoolean
isTriggered := aBoolean

activation-deactivation

activate
super activate.
self isTriggered: false.
self actor container
expressInterestIn: self eventType
for: self
sendBack: #examineEvent:

deactivate
super deactivate.
self actor container retractInterestIn: self eventType for: self

events

examineEvent: anEvent
"Examine anEvent. If it matches my criteria, trigger myself."

(self matches: anEvent)
ifTrue: [self triggerFrom: anEvent]

matches: anEvent
"Answer whether anEvent matches my criteria. At this point, I know that

```
    the event type matches my own. Subclasses may redefine this for
    more selective criteria."

    ^true

triggerFrom: anEvent
    ""
    "super triggerFrom: anEvent"

    self isTriggered: true
```

printing

```
printDetailsOn: aStream
    super printDetailsOn: aStream.
    self eventType printOn: aStream.
    aStream space
```

MetaClass: AdventureEventCue class

instance creation

```
machineEvent: aMachineEvent
    self error: 'Must specify an actor'

machineEvent: aMachineEvent actor: anActor eventType: aSymbol
    ^(self new) machineEvent: aMachineEvent; actor: anActor; eventType: aSymbol
```

Class: **AdventureHardWiredSpeechCue**

Superclass: AdventureSpeechCue
Category: Adventure-Acting

I'm an abstract superclass for speech cues with hard-wired #matcher's. To subclass me, a subclass must define the class method #buildMatcher. When the subclass creates an instance, in will build and cache the matcher (which is readonly).

Class Instance Variables:
matcher <AdventureWordMatcher | String> Cached matcher. All instances use this one matcher.

Subclasses must implement methods for
class protocol
resources
buildMatcher

initialize-release

initialize
```
super initialize.
self matcher: self class matcher
```

MetaClass: AdventureHardWiredSpeechCue class

Instance variables: matcher

resources

buildMatcher
```
self subclassResponsibility
```

accessing

matcher
```
^matcher isNil
    ifTrue: [matcher := self buildMatcher]
    ifFalse: [matcher]
```

Class: **AdventureLogicalCue**

Superclass: AdventureCue
Category: Adventure-Acting
Instance variables: cues

I'm an abstracl class for cues that perform logical operations on other cues. My subclasses must implement #logicalOperationOn:with: to do the actual operation.

Instance Variables:
cues <OrderedCollection> The collection of cues. This may contain any number of cues. If it contains zero cues, I always answer false. If it contains one cue, I answer the isTriggered value of my single cue. Otherwise (>1), I perform my logical operation on my cues in sequence.

Subclasses must implement methods for
logical operation
logicalOperationOn:with:

polling

isTriggered
```
| answer |
self cues isEmpty ifTrue: [^false].
answer := (self cues at: 1) isTriggered.
2 to: self cues size do: [:i |
    answer := self logicalOperationOn: answer with: (self cues at: i) isTriggered].
^answer
```

logical operation

logicalOperationOn: aBoolean1 with: aBoolean2
```
self subclassResponsibility
```

accessing

cues
```
^cues isNil
    ifTrue: [cues := OrderedCollection new]
    ifFalse: [cues]
```

cues: anOrderedCollection
```
cues := anOrderedCollection
```

activation-deactivation

activate
```
super activate.
self cues do: [:aCue | aCue activate]
```

deactivate
```
super deactivate.
```

```
    self cues do: [:aCue | aCue deactivate]
```

resetting

```
reset
    super reset.
    self cues do: [:aCue | aCue reset]
```

printing

```
printDetailsOn: aStream
    super printDetailsOn: aStream.
    self cues
        do:
            [:aCue |
            aCue printOn: aStream.
            aStream space]
```

MetaClass: AdventureLogicalCue class

instance creation

```
with: aCue1 with: aCue2
    ^self withAll: (Array with: aCue1 with: aCue2)

with: aCue1 with: aCue2 with: aCue3
    ^self withAll: (Array
            with: aCue1
            with: aCue2
            with: aCue3)

with: aCue1 with: aCue2 with: aCue3 with: aCue4
    ^self withAll: (Array
            with: aCue1
            with: aCue2
            with: aCue3
            with: aCue4)

withAll: aCollection
    | newCue |
    newCue := self new.
    newCue cues addAll: aCollection.
    ^newCue
```

Class: **AdventureNegativeSpeechCue**

Superclass: AdventureHardWiredSpeechCue
Category: Adventure-Acting

I trigger when the player answers in the negative.

MetaClass: AdventureNegativeSpeechCue class

resources

buildMatcher
```
^AdventureWordMatcher withAll: #('no' 'nope' 'negative' 'nix' )
```

Class: **AdventureNeverCue**

Superclass: AdventureCue
Category: Adventure-Acting

I am never triggered. I'm useful for debugging.

polling

isTriggered
    ^false

MetaClass: AdventureNeverCue class

Class: **AdventureObscenitySpeechCue**

Superclass: AdventureHardWiredSpeechCue
Category: Adventure-Acting

I trigger when the player utters an obscenity.

MetaClass: AdventureObscenitySpeechCue class

resources

buildMatcher

```
^AdventureWordMatcher withAll: #(
    "You should be ashamed, reading source code like this!"
    'fuck*' 'shit*' '*damn' 'hell' 'bitch' 'asshole')
```

Class: **AdventureORCue**

Superclass: AdventureLogicalCue
Category: Adventure-Acting

I trigger on the logical OR of my cues.

logical operation

```
logicalOperationOn: aBoolean1 with: aBoolean2
    ^aBoolean1 | aBoolean2
```

MetaClass: AdventureORCue class

Class: **AdventurePartnerCue**

Superclass: AdventurePlayerCue
Category: Adventure-Acting

I trigger on an event from my actor's #partner.

events

```
matches: anEvent
    ^(super matches: anEvent)
        & (anEvent source == self actor partner)
```

MetaClass: AdventurePartnerCue class

Class: **AdventurePlayerAppearanceCue**

Superclass: AdventurePlayerCue
Category: Adventure-Acting

I trigger when a player appears. When I trigger, I set my actor's #partner to the source of the triggering event.

events

triggerFrom: anEvent
```
super triggerFrom: anEvent.
self actor partner: anEvent source
```

initialize-release

initialize
```
super initialize.
self eventType: #appearanceEvent
```

MetaClass: AdventurePlayerAppearanceCue class

Class: **AdventurePlayerCue**

Superclass: AdventureEventCue
Category: Adventure-Acting

I trigger when my eventType matches, and the source of the event was a player.

events

matches: anEvent
```
^(super matches: anEvent)
    & (anEvent source isKindOf: AdventurePlayer)
```

MetaClass: AdventurePlayerCue class

Class: **AdventurePlayerDisappearanceCue**

Superclass: AdventurePartnerCue
Category: Adventure-Acting

I trigger when my actor's #partner disappears. The test for actor's partner is not necessary at present, because it's not a multi-user game.

initialize-release

initialize
```
super initialize.
self eventType: #disappearanceEvent
```

MetaClass: AdventurePlayerDisappearanceCue class

Class: **AdventurePlayerPresenceCue**

Superclass: AdventureActorCue
Category: Adventure-Acting

I trigger when there is a player in my container. When I trigger, I set the #partner of my actor. I'm useful for situations when the actor is in the adventure's startingPlace, and you want to bind it to the player. (In that case, an AdventurePlayerAppearanceCue won't work, because the player never makes a transition into the container--its presence is one of the initial conditions.)

Since I make no attempt to suppress polling when I have already found a player, I assume that isTriggered will only be called once. If that assumption is not valid, I'll burn CPU time.

polling

isTriggered

```
"Search my actor's container for a player."

self actor container contents do: [:aThing |
    (aThing isKindOf: AdventurePlayer)
        ifTrue:
            [self actor partner: aThing.
            ^true]].
^false
```

MetaClass: AdventurePlayerPresenceCue class

Class: **AdventurePrecedenceCue**

Superclass: AdventureCue
Category: Adventure-Acting
Instance variables: levels cueHasTriggered

I am a cue wrapper. I organize cues into ordered levels. For #isTriggered purposes, levels are evaluated from low to high. Within a level, cues are evaluated in order of addition. After any cue has triggered, evaluation stops. If I'm polled with #isTriggered again, I simply return true--cues aren't re-polled. When sent #machineEvent, I answer the machineEvent of the cue that triggered.

Instance Variables:
levels <Array> Holds the precedence levels. Each level is an OrderedCollection of cues to test.
cueHasTriggered <Boolean> True if any of my contents has triggered.

polling

isTriggered
```
self cueHasTriggered ifTrue: [^true].
self levels do: [:aLevel | aLevel do: [:aCue | aCue isTriggered
        ifTrue:
            [self cueHasTriggered: true.
            self machineEvent: aCue machineEvent.
            ^true]]].
^false
```

activation-deactivation

activate
```
super activate.
self allCues: #activate
```

deactivate
```
super deactivate.
self allCues: #deactivate
```

resetting

reset
```
super reset.
self allCues: #reset.
self cueHasTriggered: false
```

initialize-release

forLevels: anInteger
```
self levels: (Array new: anInteger).
1 to: anInteger do: [:i | self levels at: i put: OrderedCollection new]
```

accessing

levels
```
^levels
```

levels: anArray
```
levels := anArray
```

private

allCues: aSelector
```
"Perform aSelector on all cues in all levels."

self levels do: [:aLevel | aLevel do: [:aCue | aCue perform: aSelector]]
```

allCues: aSelector with: aParameter
```
"Perform aSelector on all cues in all levels passing aParameter."

self levels do: [:aLevel |
    aLevel do: [:aCue |
        aCue perform: aSelector with: aParameter]]
```

private-accessing

cueHasTriggered
```
^cueHasTriggered isNil
    ifTrue: [cueHasTriggered := false]
    ifFalse: [cueHasTriggered]
```

cueHasTriggered: aValue
```
cueHasTriggered := aValue
```

adding

atLevel: anInteger add: aCue
```
"Add aCue at level atInteger. Answer aCue."

^(self levels at: anInteger)
    add: aCue
```

MetaClass: AdventurePrecedenceCue class

instance creation

forLevels: anInteger
```
^self new forLevels: anInteger
```

Class: **AdventureRetryCount**

Superclass: Object
Category: Adventure-Acting
Instance variables: counter

I'm a retry counter. Set me initially with #resetTo:. Tell me there's been an attempt with #decrement. Test to see if I've been exhausted with #exhausted.

I consider myself to be exhausted when #decrement has been sent more than #resetTo: number of times.

Instance Variables:
counter <Integer> Counts the number of tries down toward -infinity. If this is negative, I'm exhausted.

private-accessing

counter
^counter

counter: anInteger
counter := anInteger

counting

decrement
self counter: self counter - 1

decrementAndTest
self counter: self counter - 1.
^self exhausted

exhausted
^self counter < 0

resetTo: anInteger
self counter: anInteger

MetaClass: AdventureRetryCount class

Class: **AdventureScript**

Superclass: Machine
Category: Adventure-Acting
Instance variables: actor

I'm a Machine, specialized for use with adventures. I know the actor that is executing me. This way, generic code (#actionBlock:, for example) can talk to my actor.

Instance Variables:
actor <AdventureActor> that is executing me.

accessing

actor
```
^actor
```

actor: anActor
```
actor := anActor
```

printing

printDetailsOn: aStream
```
super printDetailsOn: aStream.
aStream nextPutAll: self actor printString; space
```

MetaClass: AdventureScript class

instance creation

named: aSymbol actor: anActor
```
"Answer an instance of the receiver named aSymbol, that will be
executed by anActor. Specifying the actor is non-optional."

^(self new) name: aSymbol; actor: anActor
```

Class: **AdventureScriptState**

Superclass: MachineState
Category: Adventure-Acting
Instance variables: levels

I am a kind of MachineState. I'm augmented with an Array of OrderedCollections of AdventureCue's. I organize my cues into levels for the purposes of #isTriggered evaluation. By convention, I only have two levels (1 and 2.) Level 1 is reserved for scripts. Level 2 is reserved for automatic (at instance creation time) cues that are added by my subclasses. Within a level, order of evaluation of cues is in order of addition. The first cue added is tested (#isTriggered) first, and so on.

I have to hold the cues that are associated with me as a state, so that the actor can use them to become cognizant of the appropriate things in a given state.

In general, priority levels are reserved in the following way:

Level	*Add Cue With*	*Function*
1	*addCue:*	*cues in scripts*
2	*addDefaultCue:*	*default cues added by a subclass of my class, or a subclass of AdventureActor.*

Instance Variables:
levels *<Array>* *Holds the cue priority levels. Each level is an OrderedCollection of cues to test.*

private-accessing

levels
```
^levels
```

levels: aCollection
```
levels := aCollection
```

private-initialize-release

defaultLevels
```
"Answer the default number of priority levels."

^2
```

forLevels: anInteger
```
self levels: (Array new: anInteger).
1 to: anInteger do: [:i | self levels at: i put: OrderedCollection new]
```

initialize-release

actor: anActor
```
self forLevels: self defaultLevels.
self addDefaultsFor: anActor
```

addDefaultsFor: anActor

```
    "Augment in subclasses if desired. This is where defaults should be
    applied to me. By 'defaults', I mean cues, transitions to handle their
    events, and state entry, reentry, and exit blocks. Note that event names
    should be prefixed with the class name (or something guaranteed
    unique) to avoid conflicts. Note that the call to #super may be made
    *after* subclass defaults are added, if order is important."
    "super addDefaultsFor: anActor."

    ^self
```

private-adding

```
atLevel: anInteger add: aCue
    "Add aCue at level atInteger. Answer aCue."

    ^(self levels at: anInteger)
        add: aCue
```

adding

```
addCue: aCue
    self atLevel: 1 add: aCue

addDefaultCue: aCue
    self atLevel: 2 add: aCue
```

cues

```
allCues: aSelector
    "Perform aSelector on all cues in all levels."

    self levels do: [:aLevel | aLevel do: [:aCue | aCue perform: aSelector]]

allCues: aSelector with: aParameter
    "Perform aSelector on all cues in all levels, passing aParameter."

    self levels do: [:aLevel |
        aLevel do: [:aCue |
            aCue perform: aSelector with: aParameter]]

triggeredCue
    "Answer the first cue to trigger or nil. Test cues in order, level by level."

    self levels do: [:aLevel |
        aLevel do: [:aCue |
            aCue isTriggered ifTrue: [^aCue]]].
    ^nil
```

MetaClass: AdventureScriptState class

instance creation

named: aSymbol
self error: 'Must specify an actor'

named: aSymbol actor: anActor
"Answer an instance of myself named aSymbo for actor anActor. anActor is specified so that any default cues I create in addDefaultCuesAndEventsFor: will have an actor if they need it."

^(self new) name: aSymbol; actor: anActor

Class: **AdventureSentence**

Superclass: Object
Category: Adventure-Acting
Instance variables: string asWords

I'm useful for dividing a string into its component words, and matching a String pattern against it.

Instance Variables:
string <String> The string from which I parse. When this is set, asWords is emptied.
asWords <OrderedCollection> of String's. Cached copy of my string parsed into words.

accessing

asWords
```
^asWords isNil
    ifTrue: [asWords := self buildAsWords]
    ifFalse: [asWords]
```

asWords: aCollection
```
asWords := aCollection
```

string
```
^string
```

string: aString
```
string := aString.
self asWords: nil
```

comparing

matchUsing: aMatcher
```
"aMatcher may be anything that implements the comparison: 'match:
aString', such as a String or an AdventureStringPatternCollection."

self asWords do: [:aWord | (aMatcher match: aWord)
        ifTrue: [^true]].
^false
```

private

buildAsWords
```
^self parseWordsWithoutDelimitingPunctuationFrom: self string
```

copyRemoveDelimitingPunctuationFrom: aString
```
^self copyRemoveLeadingPunctuationFrom:
    (self copyRemoveTrailingPunctuationFrom: aString)
```

copyRemoveLeadingPunctuationFrom: aString
```
^aString isEmpty ifFalse: [(self isPunctuation: aString first)
        ifTrue: [^self copyRemoveLeadingPunctuationFrom:
```

```
                (aString copyFrom: 2 to: aString size)]
            ifFalse: [aString]]
        ifTrue: [aString]

copyRemoveTrailingPunctuationFrom: aString
    ^aString isEmpty ifFalse: [(self isPunctuation: aString last)
            ifTrue: [^self copyRemoveTrailingPunctuationFrom:
                (aString copyFrom: 1 to: aString size - 1)]
            ifFalse: [aString]]
        ifTrue: [aString]

isPunctuation: aCharacter
    ^aCharacter isAlphaNumeric not & aCharacter isSeparator not

parseWordsWithoutDelimitingPunctuationFrom: aString
    | coll stream |
    coll := OrderedCollection new.
    stream := nil.
    aString do: [:char | stream isNil
            ifTrue: [char isSeparator
                ifFalse:
                    [stream := String new writeStream.
                    stream nextPut: char]]
            ifFalse: [char isSeparator
                ifTrue:
                    [coll add: (self copyRemoveDelimitingPunctuationFrom:
                        stream contents).
                    stream := nil]
                ifFalse: [stream nextPut: char]]].
    stream notNil ifTrue: [coll add: (self copyRemoveDelimitingPunctuationFrom:
            stream contents)].
    ^coll
```

MetaClass: AdventureSentence class

instance creation

```
asWords: aString
    ^(self new string: aString) asWords

string: aString
    ^self new string: aString
```

Class: **AdventureSpeechCue**

Superclass: AdventurePartnerCue
Category: Adventure-Acting
Instance variables: matcher

I trigger when my #matcher matches a speech event from my actor's #partner. My matching behavior is completely controlled by #matcher.

Instance Variables:
matcher *<String | AdventureWordMatcher> The matcher against which a speech event must match for me to trigger.*

accessing

matcher
```
^matcher
```

matcher: aMatcher
```
"aMatcher may be anything that implements the comparison: 'match:
aString', such as a String."

matcher := aMatcher
```

events

matches: anEvent
```
^(super matches: anEvent)
    & (self matcher match: anEvent string)
```

initialize-release

initialize
```
super initialize.
self eventType: #speechEvent
```

MetaClass: AdventureSpeechCue class

instance creation

machineEvent: aMachineEvent actor: anActor matcher: aMatcher
```
^(self new) machineEvent: aMachineEvent; actor: anActor; matcher: aMatcher
```

machineEvent: aMachineEvent actor: anActor matchingAllWords: aCollection
```
^(self new)
    machineEvent: aMachineEvent;
    actor: anActor;
```

```
matcher: (AdventureWordMatcher withAll: aCollection)
```

Class: **AdventureTimeoutCue**

Superclass: AdventureCue
Category: Adventure-Acting
Instance variables: seconds timestamp

I trigger when the timestamp now is greater than the timestamp at which I am set to expire. The greater-than test assures that I will wait the amount of time the user specified, plus up to 1 second, but never less. I remember my delay time in seconds. I re-set #timestamp whenever #reset is sent to me.

I expect to be reset first, then repeatedly polled with #isTriggered.

Instance Variables:
seconds *<Time> My delay in seconds.*
timestamp *<AdventureTimestamp> Timestamp at which I trigger. I maintain this only for efficiency.*

accessing

seconds
```
^seconds
```

seconds: aTime
```
seconds := aTime
```

timestamp
```
^timestamp
```

timestamp: aTime
```
timestamp := aTime
```

resetting

reset
```
super reset.
self timestamp: (AdventureTimestamp now addTime: self seconds)
```

polling

isTriggered
```
^AdventureTimestamp now > self timestamp
```

MetaClass: AdventureTimeoutCue class

instance creation

machineEvent: aMachineEvent seconds: anInteger
```
^(self new) machineEvent: aMachineEvent; seconds: (Time fromSeconds: anInteger)
```

Class: **AdventureTimestamp**

Superclass: Time
Category: Adventure-Acting

I'm an absolute date and time, represented in hours:minutes:seconds format. Of course, my hours component if much greater than 24, because it contains the date too.

MetaClass: AdventureTimestamp class

instance creation

now

```
| now |
now := Time dateAndTimeNow.
^(now at: 2)
      addTime: (now at: 1)
```

Class: **AdventureWordMatcher**

Superclass: OrderedCollection
Category: Adventure-Acting

I match a collection of "matchers" against a string. Before I do the comparisons, I divide the string into words, and strip punctuation from the start and end of the words. If any of my contents matches the input string, then I match the string. A "matcher" is any object that implements the message 'match: aString' and answers a boolean. A matcher may be an ordinary String (wildcarded or not), or something more interesting.

comparing

```
match: aString
    | sentence |
    sentence := AdventureSentence string: aString.
    self do: [:aMatcher | (sentence matchUsing: aMatcher)
            ifTrue: [^true]].
    ^false
```

MetaClass: AdventureWordMatcher class