ESR Consortium
MicroUI-1.4.1

Micro User Interface
Profile Specification

ESR002

Reference: ESR-SPE-002-MicroUI
Version: 1.4.1
Rev: I

DEFINITIONS

"ESR" means the Specification, including any modifications and upgrades, where these terms have been stated or referred to, and made available to You by ESR Consortium, including without limitation, texts, drawing, codes,and examples.

"ESR Consortium" means the non-profit entity, registered in France in accordance with the French law of 1901.

"You" means the legal entity or entities represented by the individual executing this Agreement.

READ ONLY RIGHTS

Subject to the terms and conditions contained herein, ESR Consortium grants to You a non-exclusive, non-transferable, worldwide, and royalty-free license to view and read the ESR solely for purposes of Your internal evaluation.

GENERAL TERMS

THIS DOCUMENTATION IS PROVIDED "AS IS", WITHOUT WARRANTIES OF ANY KIND, EITHER EXPRESS OR IMPLIED.

THE READING OF THE ESR AND ALL CONSEQUENCES ARISING THEREOF IS YOUR SOLE RESPONSIBILITY. ESR CONSORTIUM SHALL NOT BE LIABLE TO YOU FOR ANY LOSS OR DAMAGE CAUSED BY, ARISING FROM, DIRECTLY OR INDIRECTLY, OR IN CONNECTION WITH THE ESR.

COPYRIGHT

ESR consortium does not have any right on this ESR. You are free to use this ESR to make any clean room implementations or derivative work as long as You doesn't not claim that Your work is compliant with the ESR. Compliance tests are available from the ESR Consortium.

MISCELLANEOUS

This Agreement shall be governed by, and interpreted in accordance with French Law. In no event shall this Agreement be construed against the drafter.

This Agreement contains the entire understanding between the parties concerning its subject matter and supersedes any other agreement or understanding, whether written or oral, which may exist or have existed between the parties on the subject matter hereof.

THIS PUBLICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THE PUBLICATION.

ESR CONSORTIUM MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN ANY ESR PUBLICATION AT ANY TIME.

Trademarks

Java™ is Sun Microsystems' trademark for a technology for developing application software and deploying it in cross-platform, networked environments. When it is used in this documentation without adding the ™ symbol, it includes implementations of the technology by companies other than Sun.

Java™,all Java-based marks and all related logos are trademarks or registered trademarks of Sun Microsystems Inc, in the United States and other Countries.

Contents

Tables

Illustrations

1 PREFACE TO MICROUI, ESR002

This document defines the *Micro User Interface profile specification v 1.4.1 , MicroUI 1.4.1*. Although [B-ON 1.2] is not mandatory, it is highly recommended.

1.1 Who should use this specification

This specification is targeted at the following audiences:

- Implementors of the MicroUI profile specification,
- Application developers designing Embedded HMIs, and targeting MicroUI,
- Virtual machines providers deploying technology for Embedded Human to Machine Interface Devices (eHMId).

1.2 How this specification is organized

This specification is organized as follow:

- Introduction: Explains what MicroUI is and why it has been designed. It presents the main advantages and general perspectives of MicroUI.
- Basic Concepts: Aims at making the reader familiar with the fundamental MicroUI notions and vocabulary.
- Architecture: Explains choices made about concurrency, the drawing scheme, fonts management, image rendering and sound playing.

1.3 Comments

Your comments about MicroUI are welcome. Please send them by electronic mail to the following address : `comments@e-s-r.net` with MicroUI in the subject.

1.4 Glossary

HMI : Human to Machine Interface

eHMId : Embedded Human to Machine Interface Device

1.5 Related Literature

B-ON 1.2: ESR Consortium, Beyond - ESR001, 2009
DSGN: Eric Gamma, Richard Helm, Ralph Johnson & John Vlissides, Design Patterns : Elements of reusable object-oriented software, 1997
MVPTL: Mike Potel, MVP: Model View Presenter, 1996,
http://www.wildcrest.com/Potel/Portfolio/mvp.pdf
Unicode: Unicode Consortium, The Unicode Standard, Version 6.1, 2012
PORTER-DUFF: Thomas Porter and Tom Duff, Computer Graphics V18 N3, 1984

1.6 Document conventions

In this document references to methods of a java class are written as `ClassName.methodName(args)`. This applies to both static and instance methods. Where the method is static this will be made clear in the accompanying text.

1.7 Implementation notes

The MicroUI Profile specification does not include any implementation considerations. MicroUI implementors are free to use whatever techniques they deem appropriate to implement the specification, with (or without) collaboration of any virtual machine provider. MicroUI experts have taken great care not to mention any special virtual machines, nor any of their special features, in order to encourage fair competing implementations.

2 INTRODUCTION

The goal of this specification is to define an enhanced architecture and the associated API required to enable an open, third-party, application development environment for Embedded HMI Devices, or eHMId. Such devices typically have some form of display, some input sensors and potentially some sound rendering capabilities. This specification spans a potentially wide set of devices. MicroUI experts agreed to limit the set of APIs specified to those only required to achieve large portability and successful deployment of embedded HMI devices. These include a User Interface based on some inputs/outputs and displays that can be alpha numeric or graphic.

Illustration 2-1: Examples of Embedded HMI Devices

2.1 Architecture

This specification defines a high-level specification for User Interface designers. MicroUI system-level implementation is outside the scope of this document. Illustration 2-2 depicts the different layers of a common eHMId running an application based on MicroUI. OEM-specific native code is not binary portable to other eHMIds, whereas the Java part is. MicroUI is designed for eHMIds that may have several screens to drive, and several kind of input sensors (generating different sorts of events).

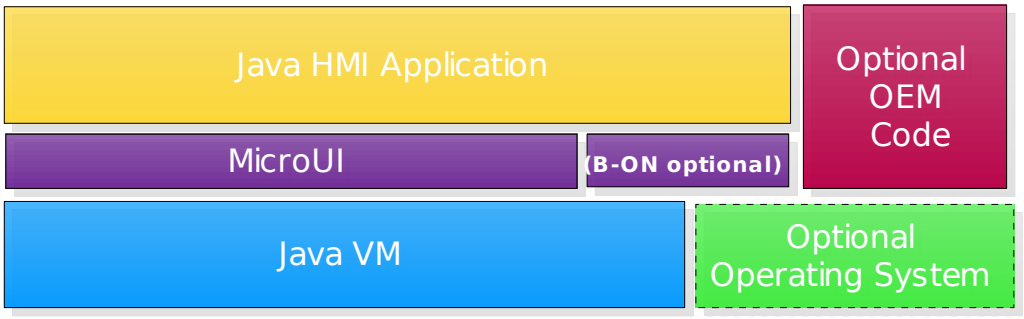


Illustration 2-2: MicroUI architecture

2.2 Requirements

The term MUST indicates that the associated definition is an absolute requirement, whereas MAY indicates that the item is optional. SHOULD indicates a highly recommended requirement.

The MicroUI profile specification assumes that eHMIds may have limited processing power, memory, and display size.

Although this specification defines minimal requirements, devices with more resources may also benefit from MicroUI special care in employing resources to their bests advantages.

2.2.1 Hardware

eHMIds MUST have the following minimum characteristics:

- **Display**: optional (several displays are permitted),
  - Display size: any
  - Display type: graphic or alpha-numeric
    - Graphic: depth is 1-bit or more,
    - Alpha numeric: at least one line, with one character
- **Input**: optional
  - Any user-input mechanisms: buttons, rotary switches, keyboards, multi touch screens, mouse-like-pointers, etc ...
- **Output**: optional
  - Any sound mechanisms that provide the ability to play tones (via hardware and/or software): buzzer, PWM, etc ...
  - Any kind of LEDs.
- **Memory**:
  - About 30 kilobytes of non-volatile memory if the whole MicroUI implementation is required by an application.
  - For graphical screens, at least one display size of volatile memory for each graphical screen. For a 128x128 monochrome display, this is 2 kilobytes of ram. For a 320x240 16-bit color display, it is 150 kilobytes of ram.

Typical small hardware platforms suitable for MicroUI implementations range from 8-bit to 32-bit running as low as 8Mhz, with less than 256KB of flash, less than 32KB of RAM, a SPI connection to a LCD controller physically controlling the screen, a set of buttons, a set of LEDs, and a buzzer. Of course, more typical powerful systems can run MicroUI too, for example driving an OpenGL graphic hardware accelerator.

2.2.2 Software

The MicroUI profile specification makes minimal assumptions about the system software of the eHMId. These requirements are as follows:

- A Java virtual machine. The kernel does not need to support an OS/RTOS - the virtual machine may be bare metal (i.e. the device boots directly in Java).
- Optionally, a [B-ON 1.2] library. B-ON defines a mechanism for immutable Java objects : read-only persistent objects in non-volatile memory, and immortal read/write objects in volatile memory.
- Optionally, a minimal capability to write to a bit-mapped graphics display.
- Optionally, a mechanism to capture user input from any of the input mechanisms.
- Optionally, a minimal capability to write to some sampled tone-oriented sound support.

2.2.3 Specification

This section sets out the requirements of this specification. Compliant MicroUI 1.4.1 implementations:

- MUST include all packages, classes, and interfaces of the MicroUI API.
- MUST support the UTF-8 character encoding.
- MUST adhere to the details of the specification as contained in the remainder of this document, with particular attention to those items marked with MUST.

2.3 Scope

2.3.1 Why MicroUI ?

MicroUI, *Micro User Interface*, aims at providing the minimal cornerstone for the quick construction of advanced, portable and user-friendly applications for a wide and heterogeneous range of cost effective devices with just-what-is-needed resources.

MicroUI has many notable characteristics that makes it a very attractive solution for embedded software development. MicroUI serves as a very robust foundation for implementing complex widget and/or windowing systems.

2.3.2 A MicroUI application is portable

The MicroUI profile comes on top of a Java virtual machine. As a result, any HMI designed with MicroUI benefits from the binary portability of the Java technology: the very same binary code will run unchanged on any device that provides a MicroUI implementation.

In addition, the very low constraints put on the hardware characteristics allow a MicroUI application to be used on a wide range of systems, enabling a high capitalization of the software.

2.3.3 MicroUI is designed for embedded devices

MicroUI is designed to target embedded systems with different kinds of resources, such as memory, screen sizes and execution speed. MicroUI supports different kinds of screens differing in size, resolution, colors available, etc. Its design allows one application to target and to drive more than one screen, depending on hardware capacities.

2.3.4 Size and flexibility are what drive MicroUI

MicroUI provides a high-level generic framework for the creation and use of widgets. The final application only loads the required widgets, which results in a limited memory footprint.

The main asset of MicroUI is probably its flexibility to fit customer needs at minimal cost: rapid design of HMIs without jeopardizing the Bill Of Material of the device.

The list of widgets that can be created using MicroUI is potentially infinite. Every company can easily defines its own set of widgets with specific look and feel in synergy with corporate or product-line graphic charters.

2.3.5 MicroUI is easy to learn, to use, and to design with

MicroUI takes its roots from established patterns such as MVC [DSGN] and MVP [MVPTL]. These architectures are highly mature and well known frameworks: it allows anyone to use MicroUI with minimal learning cost.

At the heart of MVC is a clear division between domain objects that model our perception of the real world, and presentation objects that are the graphic UI elements we see on the screen. Domain objects, referred to as models, work without reference to the presentation: they should be able to support simultaneously multiple presentations. Model objects are completely unaware of the UI.

Thanks to its base in established UI architectures well known to the vast majority of object-oriented programmers, MicroUI is a straightforward framework to design with. Once the small number of concepts and class names have been (re-)introduced, it only takes a couples of hours to design state-of-the-art widgets, from which to build highly attractive and convenient HMIs for any particular device.

In Java, the life cycle of objects is fully managed by the runtime environment (i.e. the Java virtual machine). MicroUI adheres to that principle: it means that software using MicroUI does not need to deal with freeing objects even if they are "system" resources[1] (e.g. Image, Font, ...). This is automatically done by the Java virtual machine.

2.4 Portability and logical capabilities

MicroUI is a Java framework, which implies maximizing the binary independence of MicroUI applications from the hardware on which they run: binary portability. An application may run without recompiling or changing a single bit of its binary code on several hardware platforms that offer similar rendering capabilities and similar I/O.

It is the responsibility of the implementers of MicroUI for a specific Java virtual machine on specific hardware to offer the set of I/O and rendering capabilities the application needs to run, according to its specification.

MicroUI allows several displays to be targeted at the same time. Let's assume that an application uses two displays, `D1` and `D2` . Some information will be displayed on `D1` and other on `D2` . The

1 For example, the toolkit SWT does not follow that rule. It requires explicit deallocation of objects that are known to be "resource objects" (Font, Image, Region, ...) by calling the dispose() method explicitly on such objects.

point is that it assumes two "logical" displays. The way these two logical displays are effectively provided to the application by the MicroUI+JVM combination is completely transparent to the application. There can be two real physical displays, or one physical display split in two regions by the underlying LCD driver that lies within the Java virtual machine.

The same considerations apply for I/O. Button management provides a typical example. Let's consider an application assuming an `OK` button. The capabilities of MicroUI allow the platform to translate button presses into logical command events. The application can then refer only to this logical `OK` command.

A MicroUI implementation MAY include a facility for the configuration of the mapping between hardware events and MicroUI events. The details of such a facility are outside the scope of this specification and are implementation specific. If no such facility is provided then the mappings will be fixed.

3 MICROUI BASIC CONCEPTS

3.1 Models & Listeners

The MicroUI framework uses the Observer pattern [DSGN] to provide loose coupling between objects. The most general case involves a model and one or more listeners. When the model changes, the listeners react. It is the responsibility of the model to know when it has changed.

MicroUI provides a `Listener` interface and a `Model` class. The class `Model` implements:

- `changed()`, `changed(Object)`, `changed(int)`: when a model changes (i.e. receives one of the three `changed(...)` messages), it sends the message `performAction` with the related arguments to all its listeners.
- `addListener(Listener)`, `removeListener(Listener)`: add/remove an object that observes the model it is attached to.

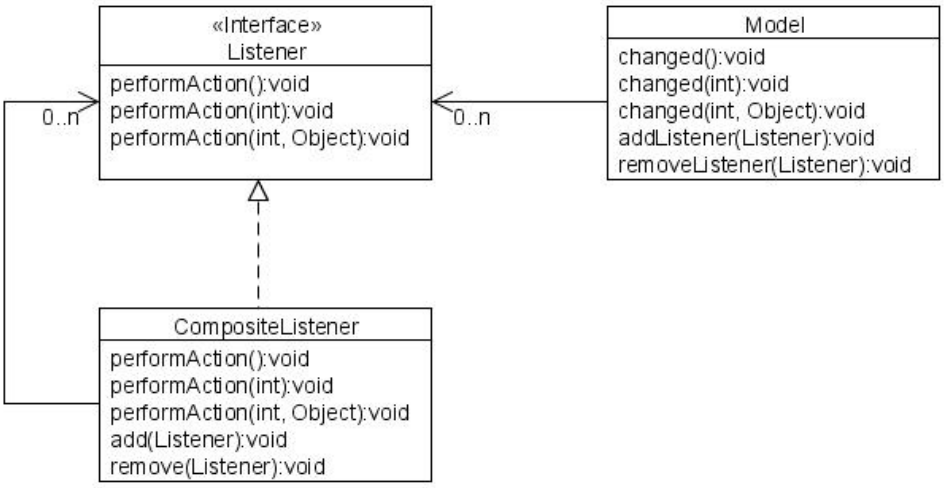


Illustration 3-1: Listeners and Models

Listeners react to their model's changes by executing one of the three methods: `performAction() ,performAction(Object) ,performAction(int) .`

MicroUI provides users with two generic default models: `IntHolder` and `ObjectHolder` . Both change when the wrapped value is set to a different one: `value(int)` , `value(Object)` .

The MicroUI framework also makes extensive use of the `Listener` interface by itself, without explicit `Model` objects: an object acts as the model but is not an instance of that class or a subclass. The object acting as the model makes explicit calls to the listener's `performAction` methods. In most cases the object acting as the model can only be connected to a single `Listener` , unlike real `Model` objects that may have many listeners. To make things easier in these cases the framework provides a `CompositeListener` class that implements `Listener` and propagates calls to `performAction` to zero or more connected listeners.

3.2 Events

MicroUI is an event-based framework. User inputs generate events, and the processing of these events updates the underlying application and affects displays and other outputs.

Events in MicroUI are represented by a single `int` value, allowing for a rich event mechanism compatible with scarce resources. An event has a type, a 8-bit value that forms the most significant byte of the `int`, followed by 8-bits which identify the generator of the event, followed by 16-bits of data, as described in Illustration 3-2.

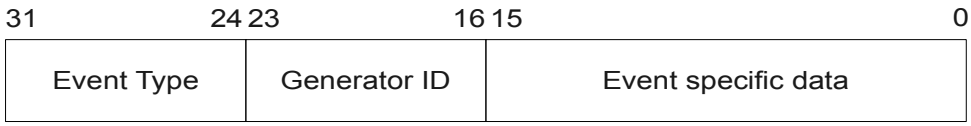


Illustration 3-2: MicroUI int-based event format

The `Event` class, which cannot be instantiated, provides a number of static methods to assist with the creation and processing of event values. `Event.getType(int)`, `Event.getGeneratorID(int)` and `Event.getData(int)` return the values of the three fields of the event passed as the argument.

The first 16 event types `[0x00..0x0f]` are reserved for MicroUI built-in events. The other values may be used freely by applications.. MicroUI-1.4.1 defines 6 public built-in events, which are representative of most often encountered input sensors categories, as described in Table 3-1.

Event Type	Name	Description	Data defined?	Section reference
0	COMMAND	Application-level events	YES	4.12.1
1	BUTTON	2 states buttons events	YES	4.12.2
2	KEYBOARD	Input letter events	NO	4.12.3
3	POINTER	Pointing device events	NO	4.12.4
4	KEYPAD	Standard keypads events	NO	4.12.5
5	STATE	State device events	YES	4.12.6
6	POINTER_BUTTON	Pointing device associated with buttons events	YES	4.12.7

Table 3-1: MicroUI built-in public events

The generator id field of an event contains either the id of an event generator, as discussed in section 3.3, or the value `0xFF` if no generator is associated with the event.

The format and meaning of the 16-bit data field in the event is defined in this specification for some of the built-in event types. The table indicates which event types have a fixed definition for the data field. See the referenced section for details. For all other events the data format and meaning is defined by the creator of the event.

3.3 Event generators

An *event generator* is an object that generates MicroUI events, for example as a result of hardware input events. The MicroUI framework contains a class for each of the five built-in event types, as shown in Illustration 3-3.

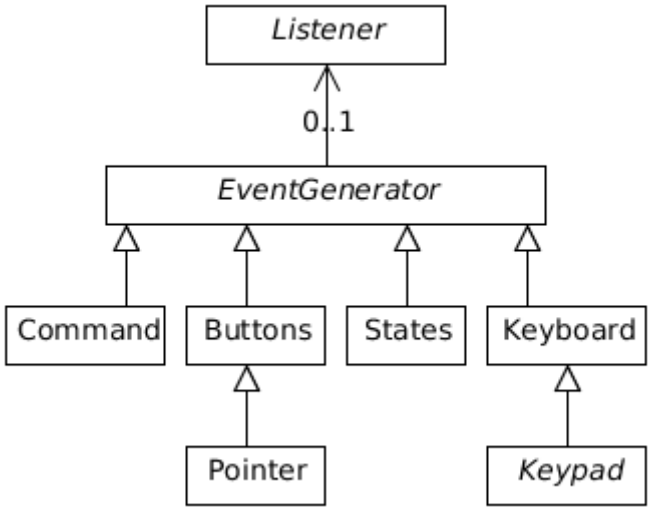


Illustration 3-3: Built-in event generators

The application may create its own event generators. An event generator is associated with a specific event type, which can be retrieved using `EventGenerator.eventType()`.

MicroUI defines a global pool of `EventGenerator` called the s*ystem-pool.* Each generator in the system-pool is allocated a unique id and the system-pool maps ids to generators. Where an event contains a generator id (that is, it has a generator id value other than `0xFF`) the pool can be used to retrieve the `EventGenerator` that generated it. The method `Event.getGenerator(int)` returns the generator object associated with the event.

At startup, the pool holds all event generators provided by the system (the choice of which generators are provided is implementation and platform specific, and related to hardware resources). An application can, if it wishes, add its own `EventGenerator` instances to the pool using `addToSystemPool` in order to get a generator id.

It is an integrity requirement of the MicroUI framework that for any event `e`, such that `Event.getGeneratorID(e) != 0xFF`, the expression

```
Event.getGenerator(e).eventType() == Event.getType(e)
```

must be true. Note that by construction, the system-pool cannot have more than 254 event generators registered[2].

3.3.1 Event handling

An event generator is normally associated with a `Listener` that handles the events it generates. When an event occurs the listener's `performAction(int)` method is called. Note that an application must call `EventGenerator.setListener(Listener)` to set the listener, and also note than an event generator may have only one listener, which may be a `CompositeListener`.

3.3.2 Generation requests

The built-in event generators each provide a method by which an application (or another part of the MicroUI framework) can request that an event be generated. These methods are all named `send`. This facility is useful for translating hardware-specific events into command events, and for creating simulated events, such as using a touch-pad press to simulate a button press[3].

3.4 Pumps

MicroUI provides a generic and extensible data pump framework. A pump is a software device that actively reads data from a data source and processes it. The framework provides an abstract class `Pump` that can be extended to create specific data processors. The main use of pumps is to read data from input devices and generate events.

Each instance of the `Pump` class holds a thread in order to read and dispatch data items. Each data item is a single 32 bit `int`. The default implementation of Pump repeatedly reads data using `read()` and sends it to the `execute(int)` method. Developers must extend the `Pump` class and implement `read` and `execute` methods. `Pump` is reliable: if an exception occurs, the pump catches the exception and calls the `crash(Throwable)` hook method.

MicroUI provides another more sophisticated pump implementation based on a buffer queue, called `FIFOPump`. It implements the `Listener` interface, so that data can be added using the `performAction(int)` method. The default implementation of the `read()` method returns the oldest data in the queue or blocks until data is available. Developers may directly extend the `FIFOPump` class to implement `execute` methods. The queue will expand to hold as many data items as the value returned by the `getMaxSize()` method. By default when the queue is full, the new data overwrites the oldest data. This behavior can be changed (new data can be dropped) by overriding `dropOnFull()` method and returning `true` .

3.5 Screens

The class `Screen` is the common superclass for all kind of screens. MicroUI specifies two kind of screens : graphical screens (`Display` class) and alphanumeric screens (`AlphanumericDisplay` class). The class `Screen` defines screen characteristics such as :

- width (`getWidth()`)
- height (`getHeight()`)
- color or monochrome display (`isColor (), getNumberOfColors()`)
- number of colors of the display (`getNumberOfColors()`)

2 Typical devices have less than 10 registered event generators.
3 This facility is intensively used by automated test suites

Such characteristics are provided by the platform and cannot be changed by the user's application. Typical applications should inquire about such characteristics and adjust themselves accordingly.

It also defines optional facilities:

- *contrast*: `hasContrast()` returns true if the screen contrast intensity can be changed. (`getContrast()`, `setContrast(int)`)
- *backlight*: `hasBacklight()` returns true if the screen backlight intensity can be changed. (`getBacklight()`, `setBacklight(int)`)
- *backlight color*: `hasBacklightColor()` returns true if the screen backlight color can be changed. Color interpretation is the same as for the drawing colors. The implementation SHOULD make its best effort to select one of the nearest available colors. (`getBacklightColor()` , `setBacklightColor(int)`)

Events handled by a screen are redirected to a `Listener` instance called the *event handler* . The way events are handled is screen specific. For graphical screens, the listener is built-in to the MicroUI implementation and cannot be changed. For an `AlphanumericDisplay a` listener has to be set by the application.

4 MICROUI ARCHITECTURE

4.1 Graphical display

4.1.1 Display characteristics

A graphical display is also sometimes called pixelated display. A graphical display is managed by an instance of the `Display` class. `Display` objects are pre-configured by the implementation and cannot be created by the application. The whole set of available `Display` objects may be retrieved using `Display.getAllDisplays()` . The length of the resulting array is the number of graphical displays available on the platform. MicroUI also defines a *default* display, `Display.getDefaultDisplay()` , which remains the same during the entire application execution. It may be `null` if no graphical display is available (see 4.2 , alpha numeric display).

A `Display` object displays `Displayable` objects. At any time a display is displaying at most one `Displayable` object. At any time, a display may be asked for its displayable (`getDisplayable()`), which may be `null` if no displayable is currently shown on that display.

Drawing primitives are provided by the display's `GraphicsContext` objects. Every display has a default `GraphicsContext` object which is automatically provided by the platform. Other `GraphicsContext` objects may be created for the same `Display` object using `getNewGraphicsContext()`(See section 4.1.7).

4.1.2 Displays FIFO events queue

Each `Display` instance manage a list of serialized events that are derived from user interaction. The event generation and its related processing are asynchronous: events are FIFO-queued. Methods issuing such events return instantly: the thread that has made the call is not blocked. Each `Display` is associated with a runtime process which performs the appropriate processing of the event as soon as possible and in order. It ensures that the processing of a previous event will have been completed before the next one is started. The delay between an event is issued and its processing is implementation dependent.

The following is the list of events processed by a display:

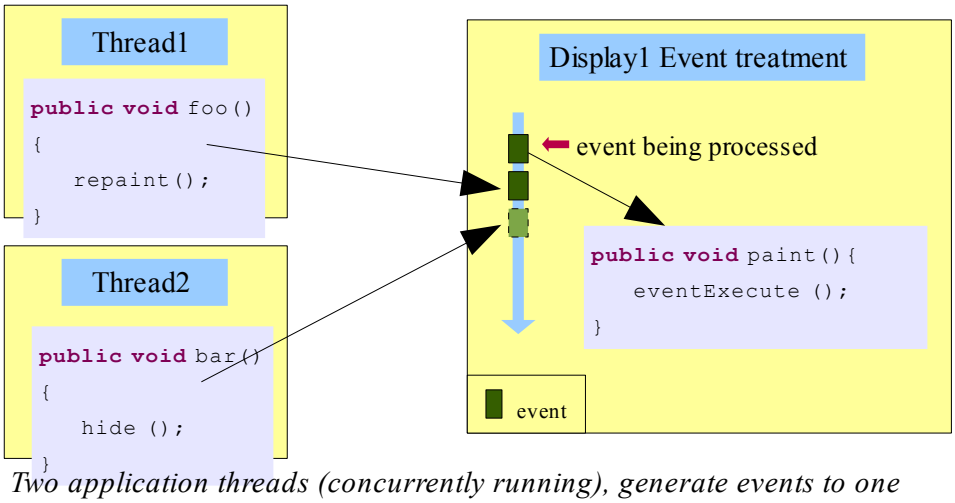
- *MicroUI events*: all kinds of public events as described in section 3.2, most likely issued by input sensors such as all available haptic sensors: buttons, mouse, touchpad, etc... `Display.handleEvent(int)` issues these events. When processed, the event is sent to the current `Displayable's` listener, through `Listener.performAction(int)`. Adding an event to a display's queue may be done at any time. An event is lost if the display has no displayable or if the displayable has no listener.
- *repaint events*: issued by both the MicroUI implementation and applications on different occasions, typically when something needs to be redrawn. `Displayable.repaint()` and `ComponentView.repaint()` trigger such events.
- *anonymous events*: an application may wish that a piece of code be executed before or after a particular event. This is done by generating an anonymous event and specifying a `Runnable` object : `Display.callSerially(Runnable)`. The `runnable's run()` method will be called when the anonymous event is processed.
- `Displayable` *show* and *hide events*: these are issued when a `Displayable` object is placed on a display (or removed from a display) using `Displayable.show()` and `Displayable.hide()`.
- `FlyingImage` *show* and *hide events*: these are issued when a `FlyingImage` object is placed on a display (or removed from a display) using `FlyingImage.show()` and `FlyingImage.hide()`.

The events of multiple displays are not globally serialized, in other words, each display serializes only its own events, which means that two displays may treat their events at different rates.

MicroUI uses regular thread priority to define the priority at which event processing occurs: `Display.setPriority(int)` . By default, the priority is the Java thread default priority of the virtual machine MicroUI is running on. Different displays may have different priorities.

MicroUI provides two ways to synchronize with event processing.

- `Display.waitForEvent(int)` adds the specified event to the display queue and blocks the current thread until this event has been processed.
- `Display.waitForEvent()` blocks the current thread until the previous event added to the display event queue (by the current thread or another thread) has been processed.



Two application threads (concurrently running), generate events to one display. These events are serialized in that display's event FIFO queue. Meanwhile, one previous event executes its processing.

Illustration 4-1: Display events serialization

4.1.3 Event producers and event consumers

MicroUI is a based on the producers-consumers pattern : some threads generates events that are asynchronously treated by display pump threads (note that most devices have only one display).

The MicroUI implementation MUST allow the setting of a maximum execution time for any event processing[4]. When an event takes too much time to execute, that processing is interrupted and the default behavior is to log an error (see section 4.11). The maximum execution time of a display can be queried using `maxExecTimeMillis() : 0` means no limit.

The maximum number of events that may be pending for each display is platform dependent. Even though, no generated event MUST be lost, that is, event generators have the guaranty that all events they generate will not be dropped. This implies that display threads (the consumers) may take precedence over threads that generate events (producers) when the display maximum pending events buffer is full. There is only one exception to this rule : events generated from display threads (consumers) may be dropped only when the pending events buffer is full. MicroUI experts encourages MicroUI implementations to start giving precedence to the consumers before the pending event buffer of a display is completely full.

4.1.4 Displayable

4.1.4.1 Visibility

As mentioned in section 4.1, `Display` objects display `Displayable` objects. In other words, a `Displayable` object can be placed on a `Display` object.

When a displayable is on a display, it is said to be shown, or visible: `Displayable.isShown()`.

The rendering of the displayable depends on the `Display`'s capabilities (color, size, etc). Thus a `Displayable` object is created for a specific display (`Displayable(Display)`). At any time, a displayable may be asked for its display (`Displayable.getDisplay()`), which cannot be `null`.

4 This is also named as “running under a watch-dog”.

In order to become visible (resp. invisible), the application needs to send `show()` (resp. `hide()`) to a displayable. A show event results in a `Displayable.showNotify()`, whereas a hide event results in a `Displayable.hideNotify()`. When a displayable replaces a previously visible displayable on a particular display, the previous displayable will be first be hidden : the `hideNotify()` method is called before the `showNotify()` (this does not issue a new event).

4.1.4.2 Rendering

When a `Display` issues a repaint for a `Displayable` (`Displayable.repaint()`), the whole `Displayable` gets repainted. `The Displayable subclasses` can implements the rendering of the `Displayable` overriding the `paint(GraphicsContext)` method (section 4.1.7 for more information about drawing).

4.1.4.3 Event Handling

Many interactions with a display are possible in MicroUI. A `Display` object can be notified about certain events and so can the displayable. When an event is processed by the display, the displayable object receive the event. The `Displayable` subclasses can handle this event by overriding the `performAction(int)` method.

4.1.5 Viewable

A `Viewable` is a `Displayable` that holds a `ComponentView`, and a `Listener`. The `ComponentView` defines what is drawn on the display, and the `Listener` defines how events are handled.

When a `Display` issues a repaint for a `Viewable` (`Viewable.repaint()`), the whole set of views of the `Viewable` gets repainted at the same time (this does not generate an event per view, but only one event).

4.1.5.1 Event Listener

The `Viewable` object delegates event management to its `Listener` instance, if any has been set for the `Viewable`.

`Listener` defines `performAction(int)`, which should be implemented to handle the events and their interactions with the application. By default a `Viewable` has a `null` listener. In order to react to events, a viewable must be given a `Listener` instance explicitly : `Viewable.setListener(Listener)`.

4.1.6 Views

A `Viewable` holds a `ComponentView` which represents a paintable rectangular area. From now, the term *view* means a `ComponentView` instance.

A view can be connected to at most one `Displayable` at a time using `Displayable.setComponentView(ComponentView)`. If the view is already connected to a `Displayable`, it is detached from the previous one.

The abstract class `ComponentView` has two subclasses in the MicroUI framework: the abstract class `View`, representing a specific area of the display, and the concrete class `CompositeView`, which holds a set of `ComponentView` and is discussed in the next section.

4.1.6.1 Encapsulation

A `CompositeView` is a `ComponentView` and holds a set of `ComponentView` (composite pattern [DSGN]). `CompositeView.add(ComponentView)` adds the given `ComponentView` to the set of views (see section 4.1.7 for a full discussion on the coordinate system). Although one may create any `ComponentView` to put on a `Viewable`, it is common to create a default one with `Viewable.newCompositeView()` which returns a composite covering the full `Viewable` area and which is set as the `Viewable`'s `ComponentView`.

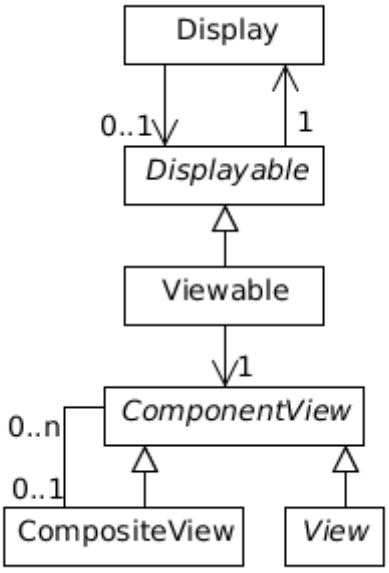


Illustration 4-2: View structure

Illustration 4-3 shows a `Viewable` that has *compositeView1* composed of 3 views: *aView1*, *aView2* and *compositeView2* composed of 4 views: *aViewA*, *aViewB*, *aViewC*, *aViewD*.

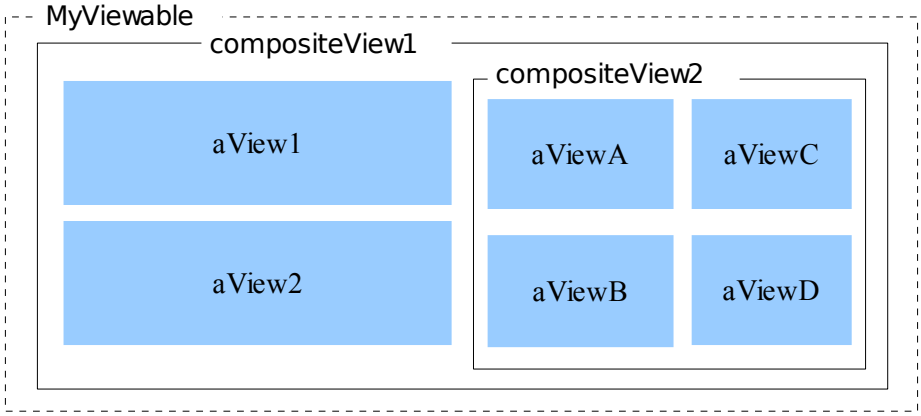


Illustration 4-3: Example of nested views

A `ComponentView` object may be removed from the set of views it is in: `CompositeView.remove(ComponentView)` ; it may also be moved around and re-dimensioned: `ComponentView.update(int,int,int,int)`. A view is aware of its relative coordinates from its container: `getX()`, `getY()`, its absolute coordinates relative to the display: `getAbsoluteX()`,

`getAbsoluteY()`, and its size: `getWidth()`, `getHeight()`. Absolute coordinates of a view will only be affected in the following cases:

- Direct coordinates updates: `update(int,int,int,int)`, `updateLocation(int,int)`, `updateSize(int,int)`
- Connection to a `Viewable`: `Viewable.setComponentView(ComponentView)`
- Add or remove from a `CompositeView`
- If view is added to a `CompositeView` and parent view coordinates are affected (recursive)

The abstract class `ComponentView` must be subclassed to define visible zones on a `Viewable` object by implementing the `paint(GraphicsContext)` method.

A `ComponentView` MUST NOT be connected twice in the hierarchy of a `Displayable` or shared with an other `Viewable`. In such situation, an `IllegalArgumentException` is thrown by `CompositeView.add(ComponentView)` and `Viewable.setComponentView(ComponentView)`.

4.1.6.2 Views and Models

A `View` object is a `Listener` and may be associated with a `Model`. It will be notified to perform some action when its model changes: `View.performAction()`. The application can set the model for a `View`: `View.setModel(Model)`. The view is then automatically added to the model's listeners in order to be notified when the model changes.

`View` implements the `Listener` interface, and the default implementation for its `performAction()` is to trigger a repaint event:

```
public class View extends ComponentView implements Listener {
    public void performAction(){
        repaint();
    }
    ...
}
```

4.1.6.3 Painting

`ComponentView` defines visible zones of a `Viewable` which as a result can be repainted. Repaint event processing results in the call of the `ComponentView.paint(GraphicsContext)` method.

When repainting a `CompositeView` object, the background area is filled if `CompositeView.fillBackground(boolean)` has been set to true previously. Then all contained `ComponentView` objects are repainted from back to top. By default, this order is the order in which `ComponentView` have been added to the `CompositeView`. The first added `ComponentView` is on back, and the last added `ComponentView` is on top. The default order can be modified using `CompositeView.arrange(ComponentView, int)` which allow a view to be moved to front/back[5].

4.1.6.4 Rendering

A `GraphicsContext` object provides graphical 2D rendering API. Every `Display` has an implicit default `GraphicsContext`, which is passed as the argument to the `paint` method when a repaint event is handled (section 4.1.2).

5 BRING_TO_FRONT, BRING_FORWARD, SEND_BACKWARD and SEND_TO_BACK are the available action constants.

In order to perform rendering operations, a `GraphicsContext` object holds a color, a font, a translation, and a clip rectangle. All rendering operations use these specified values:

- *a drawing color*, a 24-bit value interpreted as: `0xRRGGBB` , that is, the least significant 8 bits gives the blue color, the next eight bits the green value and the following ones the red color. The high order bits are ignored.
- *a clipping region*, in order to restrict rendering activities to a smaller rectangle than the `ComponentView` area, each graphic context has a rectangle, the clipping region. Rendering operations have effect only within that rectangle. It is legal to specify a clip rectangle whose width or height is zero or negative: in that case, the region is considered as empty.
- *a font*, the font used to perform string and character drawing (see font section 4.3).
- *a translation*, defines the origin of the `GraphicsContext` relative to the upper left corner of the `ComponentView` object. The initial translation is `(0, 0)` .

`GraphicsContext` offers the following drawing API:

- *points*: `drawPixel`, `readPixel`
- *lines*: `drawLine`, `drawHorizontalLine`, `drawVerticalLine`
- *polygons*: `drawRect`, `drawRoundRect`, `fillRect`, `fillRoundRect`, `drawPolygon`, `fillPolygon`
- *arcs*: `drawCircle`, `fillCircle`, `drawEllipse`, `fillEllipse`, `drawArc`, `fillArc`
- *line style*: `setStrokeStyle`, `getStrokeStyle`.
- *text*: `drawChar`, `drawChars`, `drawString`, `drawSubstring`, `setEllipsis`, `getEllipsis`
- *fonts*: `getFont`, `setFont`
- *images and regions*: `drawRegion`, `drawImage`, `drawDeformedImage`, `copyArea`, `get A RGB`, `drawARGB`
- *colors*: `getColor`, `setColor`, `getDisplayColor`
- *clipping*: `clipRect , setClip, getClipX, getClipY, getClipWidth, getClipHeight`
- *origin translation*: `translate, getTranslateX, getTranslateY`

`ExplicitFlush`, a subclass of `GraphicsContext`, adds the ability to manually control when redrawn parts of the screen are made visible: the `flush` method (see 4.1.7.4).

4.1.7 Drawing

4.1.7.1 Coordinates

The origin of the coordinate system is at the upper left corner (Illustration 4-4). The X-axis is horizontal and the Y-axis is vertical downwards.

One X-unit is exactly one pixel width, and one Y-unit is exactly one pixel height. A coordinate does not map a pixel, but rather the location between pixels. Therefore, the top left pixel matches a region of "one square unit", a rectangle, that lies between coordinates `(0,0)`, `(0,1)`, `(1,1)`, `(1,0)`.

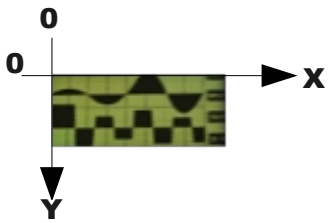


Illustration 4-4: Coordinate system

4.1.7.2 Serialized View Drawing

Many events may be added to a display's event queue during application execution. Asynchronous behavior implies that some may be out-of-date at the time they are treated, for instance a repaint event of a view, which is no longer visible. Others may be redundant, coming just after exactly the same event.

The display's event queue MUST ensure that successive repaint events are not processed separately, but are instead processed as only one repaint event. For example, if three repaint events of a same view appear successively in a display's event queue, as soon as the first is finished being handled, the three have to be "removed" from the queue.

In addition, when successive show-hide events are placed in a display's event queue, only the last show-hide event MUST be processed.

4.1.7.3 Direct Drawing

It is possible to draw on a display without using the `ComponentView`'s mechanism (`repaint()` and `paint()` methods). In other words, drawing can take place at any time, bypassing the regular highly optimized event serialization mechanism (see 4.1.2). If this is done there is no interaction with the current views parameters (color, clipping, etc.). This mechanism should be used with caution, for instance, to draw an abnormal message such as a warning.

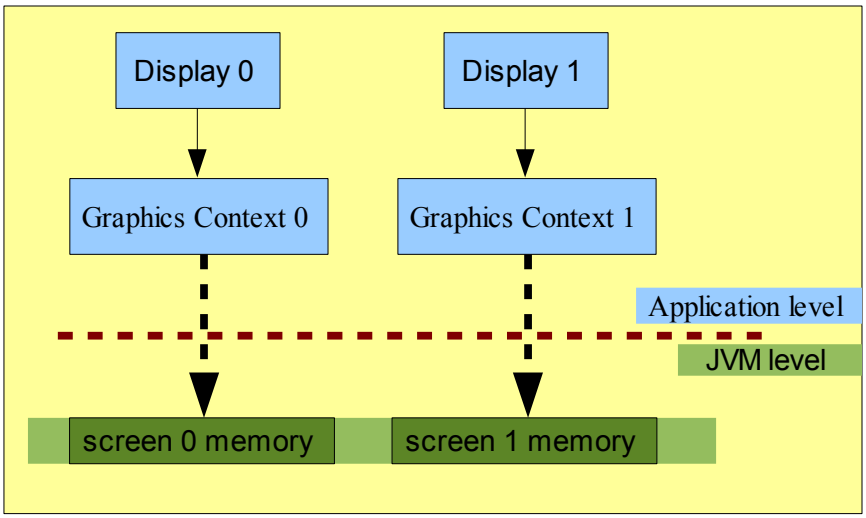


Illustration 4-5: Default display architecture

At startup, a `GraphicsContext` is defined for each `Display`. It is the standard graphics context to perform draw actions on a screen memory.

To draw on a specific display its `GraphicsContext` has to be retrieved first. All a available displays can be retrieved with the static method `Display.getAllDisplays()`.

A new graphics context can be retrieved from a display with either `Display.getNewGraphicsContext()` or `Display.getNewExplicitFlush()`. The returned graphics context works on the same screen memory as all other `GraphicsContexts` associated with the display (as shown in Illustration 4-6). This new `GraphicsContext` has its own parameters such as clip, font and color, and does not interfere with the display's default graphics context. `Display.getNewGraphicsContext()` returns an object of the `GraphicsContext` class whereas `Display.getNewExplicitFlush()` returns an object of the `ExplicitFlush` class (see 4.1.7.4).

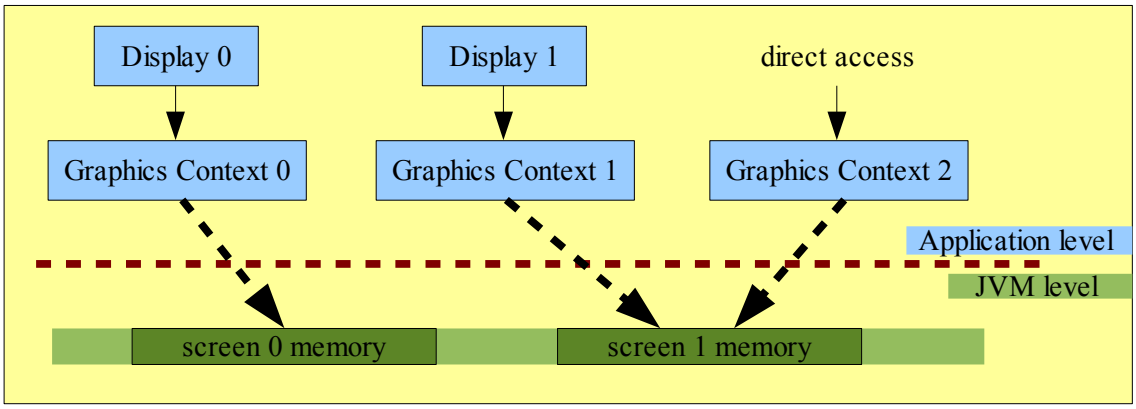


Illustration 4-6: Displays and graphics context direct access

Using this direct drawing mechanism does not ensure drawings will be performed before, during or after the current `ComponentView`'s `paint()`.

4.1.7.4 Controlling when drawing effects are visible

MicroUI specifies three policies for when the effects of drawing operations become visible on screens:

- When `GraphicsContext` instances are used within the event serialization system (see 4.1.2), the drawing effects take place at the end of the processing of the `repaint` event. If the ends of processing `repaint` events occur at a rate that is above 100 Hertz, some drawing effects may not be visible.
- When `GraphicsContext` instances are used for direct drawing (using `Display.getNewGraphicsContext()` to bypass the highly optimized events serialization system), the drawing effects take place after each drawing primitive.
- When `ExplicitFlush` instances are used for direct drawing (using `Display.getNewExplicitFlush()`), the drawing action effects take place by explicitly calling `ExplicitFlush.flush()` method. No effects will be visible until the `flush()` method executes. Whatever is the rate of the calls to the `flush()` method, all (explicit) flushes must be done.

4.1.7.5 Drawing algorithms

4.1.7.5.1 Colors

MicroUI offers a primitive to set the current drawing color. As soon as a color is set, all basic drawing actions use the current color (`drawLine`, …). The API `setColor(int rgbColor)` replaces the current color by the given `rgbColor`. A RGB color has three components:

- bits 0 to 7: BLUE component
- bits 8 to 15: GREEN component
- bits 16 to 23: RED component
- bits 24 to 31: *not used*

Application can retrieve standard colors in `Colors` interface. Examples:

- `Colors.WHITE == 0x00FFFFFF`
- `Colors.RED == 0x00FF0000`

The current color can be retrieved by the application using the method `getColor()`.

4.1.7.5.2 Filters

The API `setFilter(int rgbColor, int operator)` does not change the current color but modifies the rendering applying a filter operation on the current color. Given `rgbColor` value is interpreted as a 24-bits RGB. `operator` is a combination between a binary operator and two operators:

Binary operators are:

- `PLUS`: performs a saturated addition between each color component (R, G and B) and each corresponded filter component. If the result is higher than `0xFF`, the final component is `0xFF`. The final color becomes so lighter.
- `MINUS`: performs a saturated substraction between each color component (R, G and B) and each corresponded filter component. If the result is lower than `0x00`, the final component is `0x00`. The final color becomes so darker.
- `OR`: performs an '`OR`' logical operation on each color component (R, G and B) and each corresponded filter component
- `AND`: performs an '`AND`' logical operation on each color component (R, G and B) and each corresponded filter component
- `XOR`: performs a '`XOR`' logical operation on each color component (R, G and B) and each corresponded filter component

Unary operators are:

- `INV_COLOR`: inverts the source color (current RGB color) before performing the filter operation
- `INV_RESULT`: performs the filter operation and invert the resulted color.

By default, no filter is set. As soon as a filter is set, all new drawing actions (as `drawLine`, `drawImage`...) will use the current filter operation.

4.1.7.5.3 Polygons

MicroUI offers some primitives to draw and fill complex polygons. According to the list of points, the polygons can be convex or concave. It is also possible to draw figures like drawings of Illustration 4-7 (P *n* represent the order of the points to give to the draw or fill polygon method).

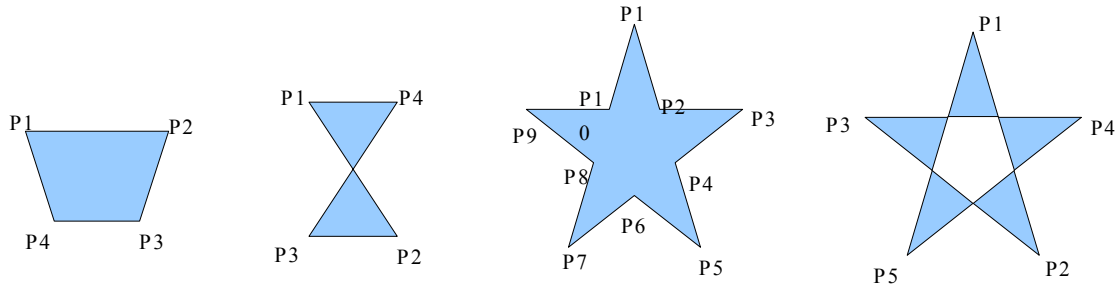


Illustration 4-7: Example of polygons

4.1.7.5.4 Deformed images

The `drawDeformedImage` method provides the ability to draw a deformed image on a graphics context. A deformed image is an image on which modifications are done like perspective, symmetry, rotation, enlargement, reducing, ... `GraphicsContext.drawDeformedImage()` method controls the modification through the image's four corners.

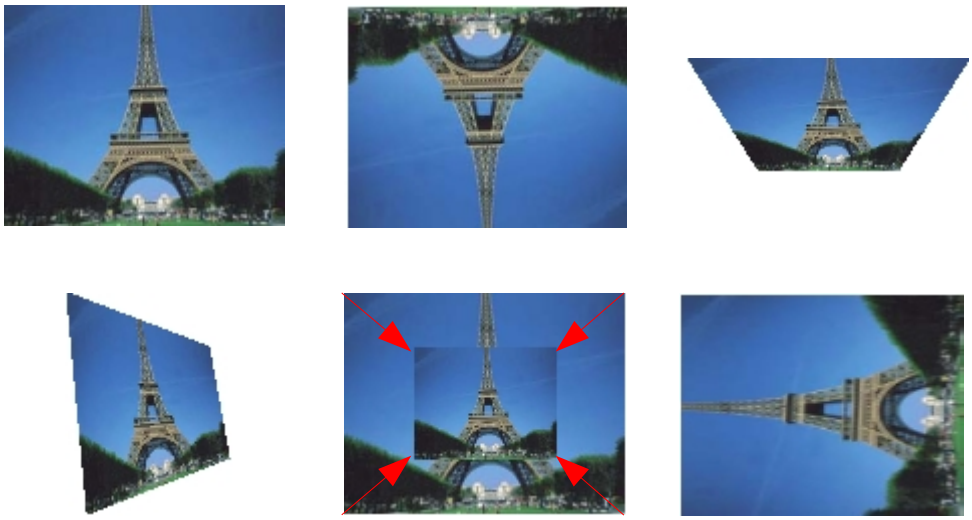


Illustration 4-8: Example of deformed images

4.2 Alphanumeric display

4.2.1 Display characteristics

Each available alphanumeric screen is represented by a specific `AlphaNumeric Display` object. `AlphaNumeric Display` objects are pre-configured by the implementation and cannot be created by the application.

The whole set of `AlphaNumeric Display` objects may be retrieved using `AlphaNumeric Display.getAllDisplays()`. The length of the resulting array is the precise number of alphanumeric displays available on the platform. MicroUI also defines a *default* display, `AlphaNumeric Display.getDefaultDisplay()`, which remains the same during the entire application execution. It may be `null` if no alphanumeric display is available (see 4.1 , display).

An `AlphanumericDisplay` object allows writing characters in a rectangular cell grid with at least one line and one column. `Screen.getWidth()` gives the number of columns and `Screen.getHeight()` gives the number of lines. A cursor can be shown. `showCursor` shows a static cursor whereas `blinkCursor` shows a blinking cursor. `hideCursor` hides cursor or does nothing if the cursor is already hidden. `setCursorPosition` sets its `(column,line)` position. At startup the cursor is hidden and its position is at the upper left corner `(0,0)`. The way the cursor is shown and blinks depends on the alphanumeric display hardware. On most common displays, non-blinking cursor underlines the current character, and blinking cursor repeatedly shows and hides the current character.

Alphanumeric displays are simple displays compared to graphical displays (see 4.1). Whereas graphical displays render `Displayable` objects and have a notion of serialized events, alphanumeric displays are "tiny" resources and therefore directly hold the rendering methods and the event handling object.

4.2.2 Text Rendering

Text rendering is performed by the methods `drawChar`, `drawChars`, `drawInt` [6] , `drawString` and `drawSubstring`. Characters drawn to an `AlphaNumericDisplay` are not Unicode characters but are font dependent characters (see section 4.3.3 for Unicode character conversion). Characters are drawn with the current color of the `AlphaNumericDisplay` object (if it has one). The coordinate system is: the upper left corner of the display is the "first" character at position `0,0`; positive towards the right within columns, and positive downwards within lines. The cursor position is moved to just after the end of the last drawn character. The method `clear` clears the display.

4.2.3 Event Handling

Each alphanumeric display can hold at most one `Listener` object. By default, this listener is `null`. In order to react to events, an alpha numeric display must be given a `Listener` instance explicitly: `AlphaNumericDisplay.setEventHandler(Listener)`. The event processing occurs immediately in the thread that initiated the event and waits for full completion of the event processing: `AlphaNumericDisplay.handleEvent(int)`.

4.2.4 Flush

`AlphaNumericDisplay` may hold a hardware or software underlying text buffer. All drawing operations (`draw...`, `clear`) are not guaranteed to be rendered until `flush` is executed. By default, all drawing operations implicitly call `flush`. This default behavior may be disabled (`setImplicitFlush(false)`). In this case, the application should explicitly call `flush` to ensure the screen is updated. Use of the explicit flush mechanism is recommended when several drawing primitives should be performed in sequence.

6 There is a variant of drawInt that right-aligns the int and always clears a specified amount of digits.

4.2.5 Text scrolling

Alphanumeric displays may provide one or more areas with continuously scrolling of displayed characters: the direction and speed of scrolling are determined when setting up the area (`UP`, `DOWN`, `LEFT`, `RIGHT`). Once set, the visual effect is that a string scrolls continuously. Illustration 4-9 shows all possible scroll combinations. MicroUI uses regular Java thread priority to define the priority of the scrolling processing thread : `AlphaNumericDisplay.setPriority(int)` . By default, the priority is the Java thread default priority of the virtual machine MicroUI is running on.

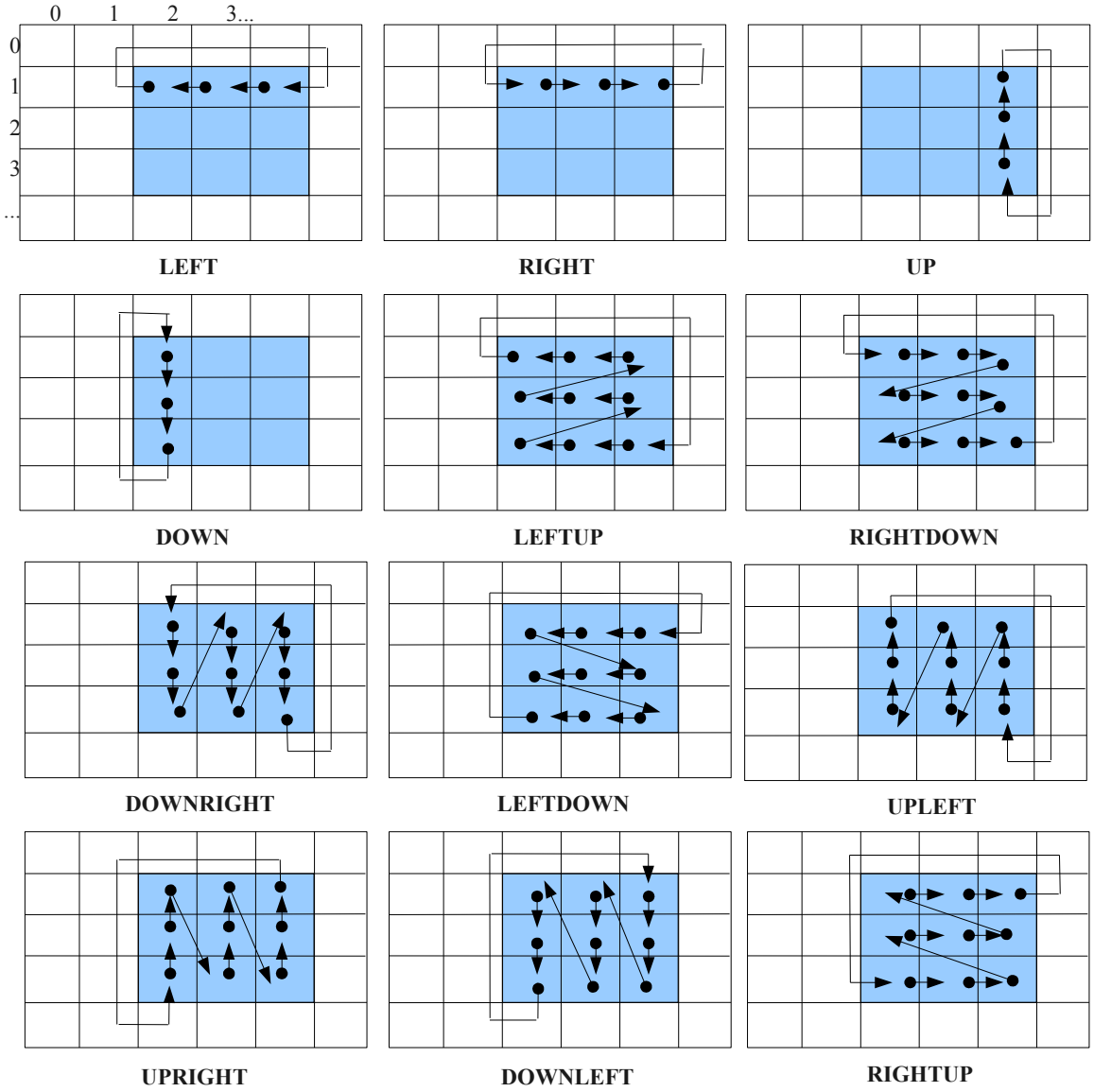


Illustration 4-9: AlphanumericDisplay scrolling combinations

Each active scroll area is identified with an unique positive `int` id. Basic scroll area management is available:

- *creation*: `newScroll` returns the id of the new created scroll area. `-1` is returned if the area could not be created due to some reasons: parameters out of bounds, the area has already some characters that are scrolling.
- *starting*: `startScroll` activates the scrolling effect of a scroll area. Starting a scroll area that does not exist throws an `IllegalArgumentException`. Starting a scroll area that is already started has no effect.
- *stopping*: `stopScroll` stops the scroll area animation. Stopping a scroll area that does not exist throws an `IllegalArgumentException`. Stopping a scroll area that is already stopped has no effect.
- *setting*: `setScrollString`, `setScrollChars`, `setScrollSubstring` assigns to a scroll a sequence of characters to display. This string replaces the default string that would have been the characters that were in place when the scroll was created. If the string is larger than the scroll area, the string is "scrolled" within the scroll area in order to have the string fully displayed. `setScrollWait` updates scroll speed in milliseconds. `0` or negative milliseconds means scroll should be repeatedly updated (full speed).
- *destroy*: `removeScroll` removes the scroll area from active scrolls and automatically frees any related resources. Its id may be reused by next created scrolls.

4.3 Fonts

4.3.1 Overview

A `Font` defines how text is rendered on a screen. MicroUI fonts conform to the Basic Multilingual Plane of the Unicode Standard [Unicode], which specifies a numeric value (code point) and a name for each of its characters. In MicroUI, a unicode script [Unicode] (a set of contiguous code point ranges) is defined by an integer, called one of the identifiers of the font. MicroUI defines the very same 70 scripts as [Unicode] (see complete list in section 5.2).

A MicroUI `Font` has one or more identifiers, a style and a descriptor. It may also have a name called its descriptor (a Java string).

An identifier is one of the predefined identifiers or one specified by the font designer. For instance, when a font holds the `LATIN` identifier, that means the font is able to render all Latin characters [Unicode]. If the font also holds the `ARABIC` identifier, that means the font is also able to render Arabic characters. A font can also hold other special identifiers that provide a useful way to caracterize a specific font. For instance, a font that contains some special characters as arrows, smileys, can be tagged by the font's creator with a special identifier (implementation dependent) that can be shared among all business units of a company.

Style is a combination of attributes (`STYLE_BOLD`, `STYLE_ITALIC` or `STYLE_UNDERLINED`). The default style is `STYLE_PLAIN`. Descriptor is a helpful string that describes the font.

MicroUI requires that all unicode characters[7] used within an application are renderable in some way. This means that the implementation must be able to find a graphical representation for every character. Unknown characters may be rendered with an empty square or a square with the 4 hexadecimal digits inside (see Illustration 4-10).

MicroUI defines no particular font format, but a few font characteristics:

- `getStyle()`: get style attributes. One may query a font about its style: `isPlain()`, `isItalic()`, `isBold()`, `isUnderlined()`.

7 Unicodes of the BMP [Unicode] ranges from \u0000 to \uFFFF.

- `getIdentifiers()`, the font hold an array of identifiers (see section 5.2).
- `supportIdentifier(int)`: returns true when given identifier is supported by the font.
- `getDescriptor()`: a string which describe the font. It is an optional parameter useful to specify the font. The descriptor can be null.
- `isMonospaced()` : returns true when all font characters have the same width.
- `charWidth(char)`, `charsWidth(char[],int,int)`, `stringWidth(String)`, `subStringWidth(String,int,int)`, the horizontal width to draw the received chars, including inter-character spacing necessary for proper positioning of subsequent chars. In the case of a mono-spaced font (`isMonospaced()`), `charWidth(char)` always returns the same value whatever the character.

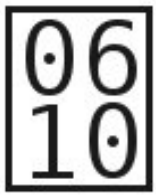


Illustration 4-10

4.3.2 Display Fonts

Display fonts are fonts that can be rendered on a graphical display. Display fonts are not related to a display: they can be rendered on all displays. All available fonts may be retrieved using `Display Font.getAllFonts()` . The default font is returned by `DisplayFont .getDefaultFont()`. `GraphicsContext` instances are initialized with the default font.

An application can get a specific font using the method `Display Font.getFont(int identifier, int height, int style)`. If no available font exactly matches the request, the system will attempt to provide the closest one. The implementation MUST use the following rules in order to determine a suitable font:

1. A suitable font must support the specified identifier. If there is no available font with a matching identifier return the default font (`null` if there is no default font).
2. From within the fonts that support the specified identifier, select the font that is the closest in height to the specified height. If there are two or more fonts equally close in height to the specified height select them all.
3. From within the fonts selected in the previous rule, pick the font or fonts that match the most style flags.
4. If more than one font is identified by the previous rule, the choice of font to return is implementation dependent. It MAY be selected on the basis of which font will render at the highest quality. For instance a font with a built-in italic style may be selected prior to a font that is drawn in italic.

In addition to the default font characteristics, display font also provides:

- `getHeight()`, `getBaselinePosition()`: respectively the total height of the font and its baseline (number of pixels from the top to the baseline of the font).

4.3.3 Alphanumeric Fonts

Alphanumeric fonts are fonts that can be rendered on an `AlphaNumericDisplay`. They are specific to one and only one `AlphaNumericDisplay`. All available fonts for an `AlphaNumericDisplay` may be retrieved from `AlphaNumericDisplay.getAllFonts()`. The default font is returned by `AlphaNumericDisplay.getDefaultFont()`. `AlphaNumericDisplay.setFont(AlphaNumericFont)` sets the font used to draw subsequent characters.

Unicode characters are converted to font characters. For example, the font provided by a DIP204 alpha numeric display will convert the character `0x007B` (defined in ISO/IEC 10646 specification as `{`) into `0x00FD`.

Alphanumeric display hardware often offers the possibility to create some font characters. For example, DIP204-compliant alphanumeric displays allow the 8 first characters to be defined. A MicroUI implementation MAY provide a means for the developer (at compile-time or at runtime) to define such extra characters.

4.4 Images

MicroUI allows images to be included within an application GUI. Images are rectangular and can be rendered on a graphical display. Rendering is highly dependent on the targeted display capabilities. For example, a `16-bit, 64x64` image will be partially and poorly rendered on a monochrome `32x32` screen.

4.4.1 Characteristics

Images are instances of the `ej.microui.io.Image` class. MicroUI allows the creation of mutable images (writable images) and the loading of immutable images from data in a specified image format (see 5.3). Images are created for one and only one display and can only be shown on their targeted display. When an image could not be created (unsupported image format, image creation not supported, ...) a `MicroUIException` instance is thrown.

Once created, an image can be drawn on its target display using `GraphicsContext` drawing methods such as `drawImage()`, `drawDeformedImage()` (see 4.1.6.4). An image provides access to its characteristics:

- *width*: `getWidth()` returns pixel width
- *height*: `getHeight()` returns pixel height
- *data*: `getARGB(int[], int, int, int, int, int, int)` fills a linear region of the `int` array with the content of image's rectangular region (`0xARGB` color)

Images resources are automatically garbage collected: there MUST BE NO `close()` / `dispose()` / `...` method to invoke when an `Image` instance is no longer needed.

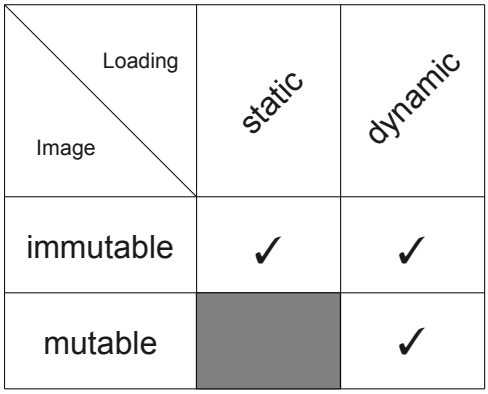


Illustration 4-11: Image creation policies

4.4.2 Colors and physical color display representation

Colors are a 32-bit quantity in MicroUI (0x00RRGGBB), even though graphical displays do not always support this full true color range. When reading pixels from a graphical context (using either `getARGB` or `getDisplayColor` methods) that has not true color capabilities, such as a 16-bit display, each primary MicroUI color lower bits are valued with zero.

For example, on a 16-bit display that represents colors with 5 bits for Red, 6 bits for Green and 5 bits for Blue, a 16-bit color `11111-111111-11111` (in 10 radix : 31-63-31) is translated into a 32-bit color with a boolean value representation of `00000000-11111000-11111100-11111000` (in 10 radix `0-248-252-248`).

4.4.3 Mutable images

Mutable images are created using `Image.createImage(int width, int height)`, `Image.createImage(Display, int, int)`. The created image is held in an off-screen buffer and can be modified after retrieving a `GraphicsContext` instance using `Image.getGraphicsContext()`. Only mutable images can get a `GraphicsContext` instance, otherwise an `IllegalArgumentException` is thrown if the image is immutable (`Image.isMutable()` returns `false`)

4.4.4 Immutable images

Immutable images can be loaded:

- *from a byte array (dynamic)*: `createImage(byte[], int, int, int)`, `createImage(Display, byte[], int, int, int)`
- *froim an InputStream (dynamic)*: `createImage(InputStream, int)`, `createImage(Display, InputStream, int)`
- *from another image (dynamic)*: `createImage(Image)`, `createImage(Image, int, int, int, int)`
- *from a resource (static)* : `createImage(String, int)`, `createImage(Display, String, int)`

An `IOException` is thrown when data cannot be decoded according to the indicated image format.

4.4.5 Collision detection

MicroUI provides facilities for collision checks between two image regions.

- `Image.collidesWith(int x0, int y0, int w, int h, Image image, int posX, int posY, int ix0, int iy0, int iw, int ih, boolean checkTransparency)`: If pixel-level detection is used (`checkTansparency` is `true`), a collision is detected only if opaque pixels collide. That is, an opaque pixel in the image would have to collide with an opaque pixel in given image for a collision to be detected. Only those pixels within the images' collision rectangles are checked. If pixel-level detection is not used (`checkTansparency` is `false`), this method simply checks if this image's collision rectangle intersects.
- `Image.collidesWith(int x0, int y0, int w, int h, Image image, int posX, int posY, int ix0, int iy0, int iw, int ih, int checkColor)`: If pixel-level detection is used (`checkColor` is a RGB color), a collision is detected only if neither of the colliding pixels are the same color as `checkColor`. Only those pixels within the images' collision rectangles are checked. If pixel-level detection is not used (`checkColor` is `-1`), this method simply checks if this image's collision rectangle intersects.

4.4.6 Specification

A MicroUI implementation:

- MAY NOT support the creation of mutable images
- MAY NOT support loading of immutable images from dynamic data (`Image.createImage()` with an `InputStream`, a byte array or an other image).
- MUST support loading of immutable images from a resource name. The way the resource is loaded is implementation dependent. It can be loaded dynamically at runtime or preprocessed for the display target at compile-time.
- MUST be able to load images that are in monochrome BMP format (`BMP_MONOCHROM`).
- MAY support other image formats such as PNG images (`PNG`).

Basically, an implementation is MicroUI compliant if it only supports loading of immutable images from resources in monochrome BMP format. This may be the case of implementations on highly memory constrained devices that do not provide runtime image decoders and do not allow runtime offscreen buffer allocation.

4.5 Transparency

All `GraphicsContext` destinations (mutable images or display) consist entirely of fully opaque pixels. Only immutable images may contain transparency information, called alpha level. Since an `Image` is associated to a display, the number of supported alpha levels is display specific and can be retrieved using `Display.getNumberOfAlphaLevels()`. The minimum number returned value is `2`, meaning that displays must at least support full opacity and full transparency with no blending. Alpha level is `255` for fully opaque pixels, `0` for fully transparent pixels, and between `0` and `255` for semitransparent pixels.

For all rendering operations, source pixels are always combined with destination pixels using the *Source Over Destination* rule [PORTER-DUFF]. One of its properties is that compositing any pixel with a fully opaque destination pixel always results in a fully opaque destination pixel. This has the effect of confining full and partial transparency to immutable images, which may only be used as the source for rendering operations.

When creating an image from source data, a fully opaque pixel in the source data must always result in a fully opaque pixel in the new image (Alpha level 255), and a fully transparent pixel in the source data must always result in a fully transparent pixel in the new image (Alpha level 0). A semitransparent pixel data should be converted to same alpha level, if available, or the next lower available alpha level. Therefore a semitransparent pixel is converted to a fully transparent pixel on a display that supports only 2 alpha levels.

`GraphicsContext.setColor(int)` only deals with `RGB` color so that drawing primitives always generate fully opaque pixels (the highest eight bits are skipped). `Image.getARGB(...)` allows the application to retrieve image pixels with alpha level.

4.6 Flying Images

A `FlyingImage` holds an image to be displayed at the top level in the rendering depth of a display. A `FlyingImage` may be used in collaboration with a `Pointer` event generator in order to display a cursor image at a pointing device position (see section 4.12.4).

The image held by the `FlyingImage` associates the `FlyingImage` with a specific display (since an `Image` is created for a specific display). Several `FlyingImage` objects may be associated with a `Display`. The location of the flying image can be changed using `FlyingImage.setLocation(int, int)` and retrieved using `getX()` and `getY()`.

A `FlyingImage` is in either the *show* or *hide* state. When created, a `FlyingImage` is in *hide* state. Only instances that are in *show* state may be drawn. State can be changed at any time using `FlyingImage.show()`, `FlyingImage.hide()`. These actions are serialized with display events, so applications need to check `FlyingImage.isShown()` to ensure the new state has been handled by the display (`isShown()` can return `false` right after a call to `show()`).

The flying images of a display are drawn:

- after all drawings originating from the `Displayable.paint(GraphicsContext)` system calls on current displayable.
- during an explicit call to `ExplicitFlush.flush()` (see 4.1.7.4)
- subsequently to an explicit call to `FlyingImage.repaint()`. This action is serialized with display events.

Flying images are repainted together according to the following process:

- Restore the display area of the last position for each `FlyingImage` in the reverse order they have been set to *show* state. The last restored area is the area of the first flying image set in *show* state (it is restored above all the other drawings).
- Draw the underlying image on the display area, in the order they have been set to *show* state. The last drawn flying image is the last set in *show* state (it is drawn above all the other drawings).

`FlyingImage` follows `Image` life-cycle policy. It will be garbage collected when unreferenced. Garbage collecting a flying image that was in the *show* state will first set the flying image set to *hide*, then it will leave the display unchanged.

4.7 LEDs

LEDs are a basic output mechanism. MicroUI provides a small and efficient set of methods to interact with LEDs. In order to support the philosophy that LEDs are "tiny" resources, a LED is identified simply by an `int` id, and LEDs are manipulated using static methods on the class `Leds`.

- `Leds.getNumberOfLeds()` returns the available number of LEDs. The range of valid LED ids is 0 to (`Leds.getNumberOfLeds() - 1`).
- `Leds.setLedIntensity(int ledId, int intensity)` controls the intensity of the specified LED. If the id is invalid (out of range) the method has no effect.
- `Leds.ledOn(int ledId)` turns on a given LED. It is a synonym of `Leds.setLedIntensity(ledId, MAX_INTENSITY)`.
- `Leds.ledOff(ledId)` tuns off a given LED. It is a synonym of `Leds.setLedIntensity(ledId, MIN_INTENSITY)`.

If a LED does not handle intensity, any valid intensity different from `MIN_INTENSITY` turns the LED on.

4.8 Sound system

MicroUI provides basic sound rendering for devices which have the ability to play sounds. `AudioOut.playTone(int tone, int duration, int volume)` allows a tone at a specified volume and for the given duration to be played. The `tone` is a value between `0` and `127` (see Table 4-1). The value to use for a tone with `freq` as frequency can be retrieved from the following formulas:

```
SEMITONE_CONST = 1/(ln( 2^(1/12) )) = 17.31234049066755
tone = ln(freq/8.176) * SEMITONE_CONST
```

The `duration` is the time in milliseconds the sound is played, `0` or negative value for infinite duration. The `volume` is a value between `0` and `100` which represents a relative amount of volume. Global volume can be adjusted by `AudioOut.setMasterVolume(int)`: from `0` (mute) to `100` (initial value, maximum available platform volume).

MicroUI provides multi-tone sound rendering: `AudioOut.playTones(byte[])`. The array of bytes is a list of commands and tones to be rendered by the sound system. The format is described in Table 4-2.

	Tones											
Octave	C	C#	D	D#	E	F	F#	G	G#	A	A#	B
-1	0	1	2	3	4	5	6	7	8	9	10	11
0	12	13	14	15	16	17	18	19	20	21	22	23
1	24	25	26	27	28	29	30	31	32	33	34	35
2	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59
4	60	61	62	63	64	65	66	67	68	69	70	71
5	72	73	74	75	76	77	78	79	80	81	82	83
6	84	85	86	87	88	89	90	91	92	93	94	95
7	96	97	98	99	100	101	102	103	104	105	106	107
8	108	109	110	111	112	113	114	115	116	117	118	119
9	120	121	122	123	124	125	126	127				

Table 4-1: Tones list

```
// Notes:
// un: n is the number of bytes to specify an integer value, in big
// endian format (MSB,...,LSB)

// HEADER
Bytes ::= Version BPM Resolution NbBlocks Blocks NbCmds Commands
Version ::= u1          // currently only 1 is available
BPM ::= u2              // beat per minute
Resolution ::= u1       // number of duration per beat
NbBlocks ::= u4
NbCmds ::= u4

// Block: atomic sequence of tones, described by an unique id
Blocks ::= Blocks Block
Block ::= BlockID Length BlockNext
BlockID ::= u4
Length ::= u4           // length of tones and durations

// Sequence of tones and durations
// The Duration of the block defines the semantic of the Tones bytes.
// If Duration is 0, the Tones bytes are organized as a sequence of two
// u1 representing one tone and its duration. If Duration is greater
// than zero, the sequence of Tones bytes are just the tones, all
// with the same Duration.
BlockNext ::=
       0  TonesAndDurations  // explicit duration for each tone
     | u1 Tones              // global duration same for each tone
TonesAndDurations ::= Tone Duration
Duration ::= u1
Tones ::= Tones Tone
Tone ::= u1

// Playing commands
Commands ::= Commands Command
Command ::= C_PLAY | C_BPM | C_AUTOREPEAT | C_VOLUME
C_PLAY ::= 0 BlockID    // Play a block
C_BPM ::= 1 BPM         // Adjust current beat per minute
C_AUTOREPEAT ::= 2      // Restart after the last command (play loop)
C_VOLUME ::= 3 u1       // Adjust current volume (range [0..100]).
                        // By default, volume is 50
```

Table 4-2: Multi-tone byte array grammar

Multi-tone executions can be paused and resumed (`AudioOut.pause(true/false)`). `playTone` functions return immediately. `AudioOut.stopTone()` tells the system to stop the basic sound rendering as soon as it can if some sound was currently playing. Calling a `playTone` function automatically stops the playing of any current tones.

An application can synchronize its execution with sound execution by registering a listener (`AudioOut.setListener(Listener)`). `Listener.performAction(int)` is called each time a sound is being played and the argument is the index in the byte array of the current multi-tone execution (only when tone is played from `playTone(byte[])`).

```
byte C = 60, D = 62, E = 64;
byte[] music = new byte[]{
    //header
    1, //version
    0, 60, //beat per minute
    2, //resolution (one duration is (60/60)/2 second, i.e. 1/2 second)
    0, 0, 0, 1, //number of blocks
    //blocks of tones
    0, 0, 0, 0, //block ID 0
    0, 0, 0,12, //length of the block
    1, //duration of all the tones within the block
    C, C, C, D, E, D, C, E, D, D, C,
    //commands
    3, 100, //set volume to its maximum
    0, 0, 0, 0, 0 //play block 0
}
```

Table 4-3: Example of `playTones` byte array encoding

4.9 Startup and termination

4.9.1 MicroUI startup

The MicroUI implementation may or may not start automatically. Until it is started, display rendering and event processing will not take place. A system property named `ej.microui.autostart` allows the behavior of the implementation to be defined. If this property is not defined, the MicroUI will start automatically with the application. If this property is set to `false`, MicroUI will not start until the method `MicroUI.start()` is invoked.

4.9.2 MicroUI termination

An application may want to stop MicroUI, thus stopping all rendering and event processing operations. This can be done with `MicroUI.stop()`. This method acts asynchronously: all pending events will be processed before stopping MicroUI.

4.10 System Properties

The MicroUI specification defines a set of properties, described in Table 4-4.

Property	Description
`ej.microui.autostart`	*Optional*. When `false`, MicroUI should not start until there is an explicit call to `MicroUI.start()` (see 4.9.1)
`ej.microui.vendor`	*Optional*. The name of MicroUI library provider
`ej.microui.vendor.url`	*Optional*. The web site of the MicroUI library provider.
`ej.microui.version`	*Optional*. The MicroUI version that is supported by the implementation: three numbers separated with `'.'` (an example is `1.4.1`)

Table 4-4: System Properties

4.11 Error management

MicroUI defines two ways to notify users of erroneous behavior :

- a minimal error-log service: this allows information to be received from the MicroUI implementation when something goes wrong (for example, during event processing). `MicroUI.errorLog(boolean)` is used to switch the service on or off. The error-log service is off at MicroUI start-up. The precise logged information and the way information is logged is fully MicroUI implementation dependent.
- a basic alert mechanism : `MicroUI.beep()` allows a "`beep`" to be generated: the meaning of the beep is platform dependent. On a system with sound capabilities, it is reasonable to expect a short sound. This basic alert is accessible from the application code and from the MicroUI implementation (it is common for instance to get a `beep` when a display event queue is full).

4.12 Built-in events and event generators

4.12.1 COMMAND

4.12.1.1 Event format

Commands are application-level events. They are not directly related to input events but are generated by the platform or the application to indicate that the application should carry out some processing. Commands are typical application-level effects of input events. The advantage of using commands rather than specific input events in an application is that the application can be more portable: it is not tied to specific input devices. The format of command events is shown in Illustration 4-12.

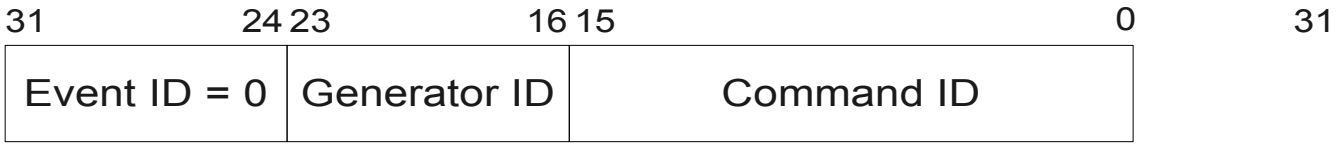


Illustration 4-12: Command event format

A set of basic commands frequently required by applications is predefined, and can be accessed in the MicroUI framework as constants defined in the `Command` class (Table 4-5). An application can define its own specific commands using other ids up to `65535`.

Command ID	Command name
0x00	ESC
0x01	BACK
0x02	UP
0x03	DOWN
0x04	LEFT
0x05	RIGHT

0x06	SELECT
0x07	CANCEL
0x08	HELP
0x09	MENU
0x0A	EXIT
0x0B	START
0x0C	STOP
0x0D	PAUSE
0x0E	RESUME
0x0F	COPY
0x10	CUT
0x11	PASTE
0x12	CLOCKWISE
0x13	ANTICLOCKWISE
0x14	PREVIOUS
0x15	NEXT
0x16	DISPLAY

Table 4-5: Predefined commands

4.12.1.2 Event generator

`ej.microui.Command` is an event generator that generates command events. A command event is generated using `Command.send(int commandId)`. This allows the generation of commands from within MicroUI without relying on an underlying input event format.

4.12.2 BUTTON

4.12.2.1 Event format

The data of buttons event is composed of an action and a button id (Illustration 4-13), so it allows at most 256 buttons to have at most 256 different actions. MicroUI defines 6 basic actions described in Table 4-6.

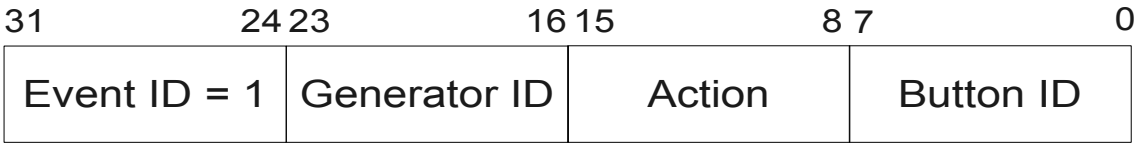


Illustration 4-13: Buttons event format

Action ID	Action name	Description and usage
0x00	PRESSED	Button pressed.
0x01	RELEASED	Button released. Usually sent after its corresponding PRESSED event.
0x02	LONG	Button pressed during a long amount of time. Usually sent once between a PRESSED and a RELEASED event. The delay before generating this event is implementation dependent.
0x03	REPEATED	Button pressed during a long amount of time. Usually sent periodically between a PRESSED event and a RELEASED event. The period delay to generate this event is implementation dependent.
0x04	CLICKED	May be automatically generated after a PRESSED event for buttons that supports extended features (see below)
0x05	DOUBLE_CLICKED	May be automatically generated after two consecutive PRESSED events for buttons that supports extended features. The maximum delay between two PRESSED events to generate a DOUBLE_CLICK event can be configured by application when enabling DOUBLE_CLICK feature for a button (see below)

Table 4-6: Basic button actions

4.12.2.2 Event generator

A Buttons event generator is usually associated with a group of physical buttons and generates events relating to them.

The Buttons class also contains a number of static helper methods that return information extracted from an event:

- int buttonID(int event), int action(int event) to get the button id and the action associated with this event
- isPressed(int event), isReleased(int event), isLong(int event), isRepeated(int event), isClicked(int event), isDoubleClicked(int event) to check which kind of action is associated with the event

A button event can be generated using Buttons.send(int buttonId, int action).

4.12.2.3 Extended features

For a specified subset of buttons the Buttons generator holds the elapsed time since the last event occurrence for that button and supports the optional generation of CLICK and DOUBLE_CLICK events. An application can determine whether a button supports these extended features using Buttons.supportsExtendedFeatures(int id). If a button supports them, the CLICK and DOUBLE_CLICK event generation features are disabled by default.

CLICK can be activated using enableClick(boolean state, int id). When enabled, a CLICK event is generated subsequently to a PRESSED event. DOUBLE_CLICK can be activated using enableDoubleClick(boolean state, int delta, int). When activated, a DOUBLE_CLICK event is generated subsequently to a PRESSED event if the time between the previous PRESSED event is less or equal than delta time. When both CLICK and DOUBLE_CLICK

are activated on a button, then when a `DOUBLE_CLICK` event is going to be generated, a `CLICK` event is generated right before.

The current state can be retrieved using `boolean clickEnabled(int id) / boolean doubleClickEnabled(int id)`. `Buttons.elapsedTime(int id)` returns the elapsed time since the last event occurred on this button.

`Buttons` generator(s) provided by the implementation are not required to support these extended features for all of their buttons. Is it implementation dependent. An application may create its own `Buttons` generator using `Buttons()` and `Buttons(int n)` constructors. The second constructor specifies that extended features are required for button ids from `0` to `n-1`, whereas the first one does not configure any buttons with extended features.

4.12.3 KEYBOARD

The format of the data field of a keyboard event is not defined in this specification; it is implementation specific (Illustration 4-14).

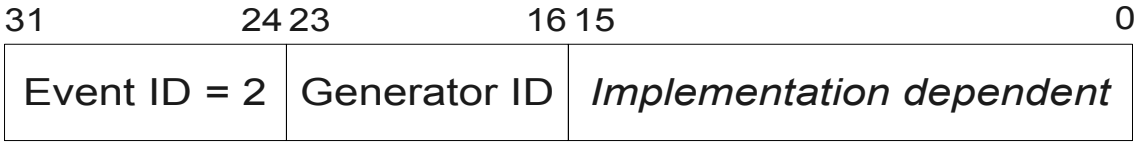


Illustration 4-14: Keyboard event format

A `Keyboard` event generator allows key combinations to generate a key code. A `Keyboard` generates the low-level events `KEY_DOWN` and `KEY_UP` and the high-level event `TEXT_INPUT`. The low-level events may be turned on as they are off by default (`Keyboard.onlyTextInput(boolean)`).

The event action (either `Keyboard.KEY_DOWN`, `Keyboard.KEY_UP` or `Keyboard.TEXT_INPUT`) can be retrieved using the method `Keyboard.action(int event)` and the key code can be retrieved using `Keyboard.nextChar(int event)`. For low level events, this returns a `Keyboard` specific key code. This code may be an unicode for keys that represent a character. For the `TEXT_INPUT` high level event, `nextChar` returns the generated unicode character. For example, if low-level events are enabled pressing the `Q` key on a PC/AT US keyboard using a US keyboard layout mapping will produce:

- `KEY_DOWN` with `Q` as key code
- `TEXT_INPUT` with `q` as unicode character
- `KEY_UP` with `Q` as key code

If a `SHIFT` key is pressed while the same `Q` key is pressed, the following keyboard events will be produced:

- `KEY_DOWN` with `SHIFT` as key code (`Keyboard` specific)
- `KEY_DOWN` with `Q` as key code
- `TEXT_INPUT` with `Q` as unicode character
- `KEY_UP` with `SHIFT` as key code
- `KEY_UP` with `Q` as key code

`Keyboard` may hold an internal buffered event queue. `Keyboard.reset()` flushes all pending key codes (pending key codes are all key codes that have been generated but that the application has not yet retrieved using `nextChar`). The application can get the policy used by the keyboard when its event queue is full. `Keyboard.dropOnFull()` returns `false` if the new event overwrites the oldest event, `true` if the new event is dropped.

A keyboard event can be generated using `Keyboard.send(int type, char keycode)`.

4.12.4 POINTER

The data of pointer event is composed as a buttons event of an action and a data (Illustration 4-15), so it allows at most 256 different actions. The first 6 actions are defined by buttons. Pointer defines another 4 actions described in Table 4-7.

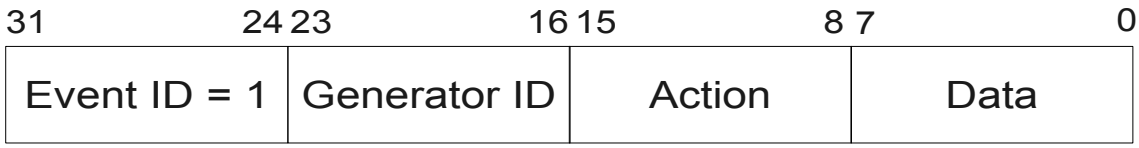


Illustration 4-15: Pointer event format

Action ID	Action name	Description and usage
`0x06`	`MOVED`	Pointer moved. Data is 0.
`0x07`	`DRAGGED`	Pointer moved and one of the buttons is pressed. Data is 0.
`0x08`	`ENTERED`	Pointer entered an object. Data is implementation dependent.
`0x09`	`EXITED`	Pointer exited an object. Data is implementation dependent.

Table 4-7: Basic pointer actions

A `Pointer` event generator reports the position of a pointing device as an x, y position within an area called *pointer area*. The size of the pointer area is set when the `Pointer` is constructed and cannot be modified (`Pointer.getAbsoluteWidth()`, `Pointer.getAbsoluteHeight()`). Coordinates are clipped to the pointer area.

The `Pointer` can be asked for its last absolute position, expressed in terms of the pointer area with which it was constructed (`getAbsoluteX()`, `getAbsoluteY()`). It can also be asked for scaled coordinates (`getX()`, `getY()`). The *scaled area* is set using `setScale(int, int)`, `setScale(Display)` methods. By default there is no scaling. It is also possible to specify, using `setOrigin(int, int)`, an offset to be applied to the scaled coordinates returned. For example, if the origin is set to be `(20, 30)` then the `x` position returned will be the absolute `x` position – `20`, and the y position will be the absolute `y` position – `30`. By default there is no offset. If both scaling and origin adjustment are specified then the origin offset is first applied to the absolute position then the scaling is applied. Illustration 4-17 shows an example of coordinates system configuration.

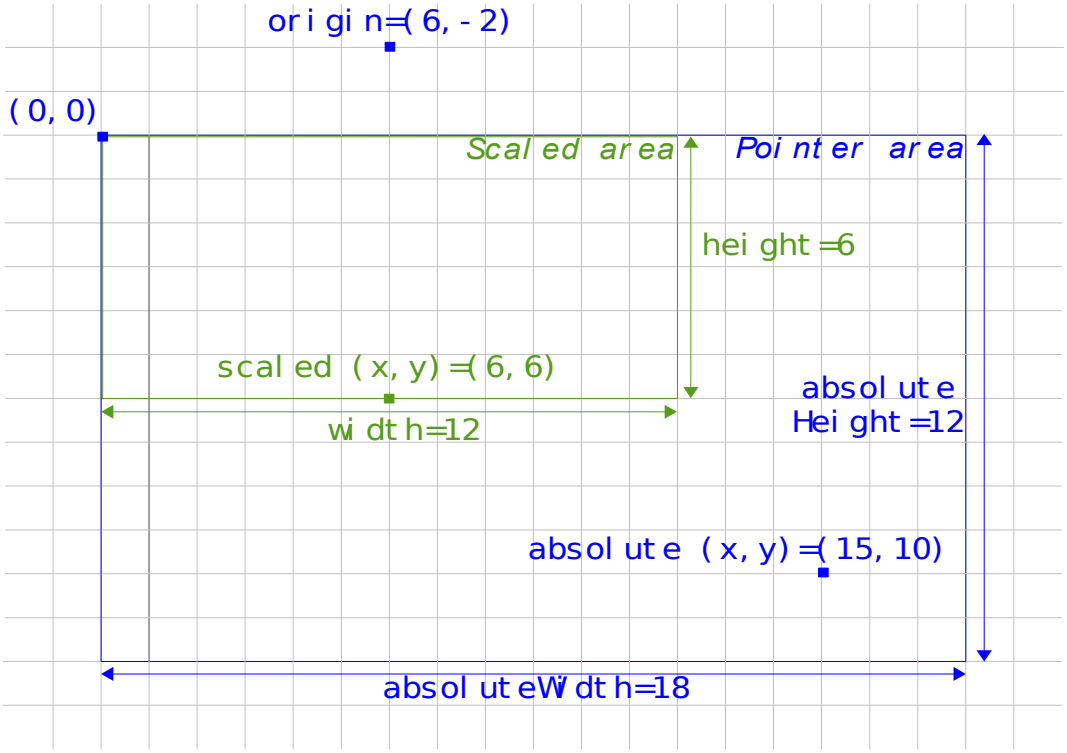


Illustration 4-16: Example of the `Pointer` coordinate system

A flying image can be associated with the pointer using `Pointer.setFlyingImage(FlyingImage)` method. When the pointer moves, the location of the flying image is automatically updated by the generator with scaled area coordinates.

A `Pointer` moving event can be generated using `Pointer.move(int x, int y)`, the listener will receive a `DRAGGED` event if at least one of the buttons is pressed, `MOVED` otherwise.

4.12.5 KEYPAD

The format of the data field of a keypad event is not defined in this specification; it is implementation specific (Illustration 4-17).

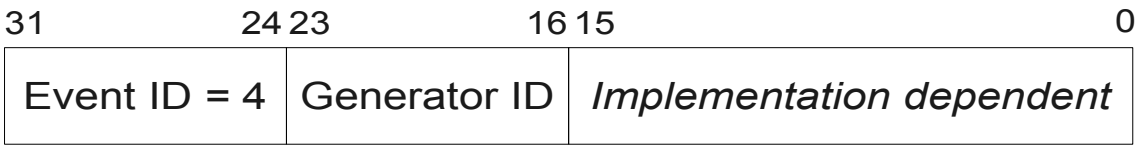


Illustration 4-17: Keypad event format

A `Keypad` is a `Keyboard` that defines an event generator for 12-key keypads. It follows the ETSI ES 202 130 mapping, which takes into account ETSI, ITU-T, CEN and ISO/IEC specifications and recommendations. Also see ISO/IEC 10646. The key mapping is defined in `Table 33` and `Table 63` of ETSI ES 202 130 (v1.1.1). Key mappings from `1` to `9` are specified. In addition, the next three keys have extended mappings defined as:

- key 10: `'*'` : this key is only used to switch from one mode to another

- key 11: `' '`, `'+'`, `'0'` in order
- key 12: `'\n'`, `'#'` in order

`Keypad` sends low-level `Keyboard` events with basic code of the key (`'0'`, ... , `'9'`, `'#'` or `'*'`) and high level `TEXT_INPUT` events with next key code mapping until key is validated (key codes are scrolled in order, circularly). A key is validated when no new key has been pressed before the validation delay or if another physical key of the keypad is pressed. The delay starts when the key is pressed, so a key may be validated even if it is not yet released. When a key is validated, `Keypad` sends `KEY_VALIDATED` event. The delay for key validation can be modified at any time using `Keypad.setDelay(int)`. Keypad uses 4 different modes (Table 4-8) to filter the letters that are scrolled. The mode can be changed using `Keypad.setMode(int)`.

For example, assuming that low-level events are enabled (see Keyboard 4.12.3), pressing the `'2'` key twice rapidly and then waiting a little amount of time after validation delay will generate:

- `KEY_DOWN` with `'2'`
- `TEXT_INPUT` with `'a'`
- `KEY_UP` with `'2'`
- `KEY_DOWN` with `'2'`
- `TEXT_INPUT` with `'b'`
- `KEY_UP` with `'2'`
- `KEY_VALIDATED` (after validation delay expired)

Pressing the `'2'` key and then the `'3'` key will generate:

- `KEY_DOWN` with `'2'`
- `TEXT_INPUT` with `'a'`
- `KEY_UP` with `'2'`
- `KEY_VALIDATED`
- `KEY_DOWN` with `'3'`
- `TEXT_INPUT` with `'d'`
- `KEY_UP` with `'3'`
- `KEY_VALIDATED` (after validation delay expired)

Mode	Description
`NUM`	Only digits are selected
`ALPHA`	Digits and letters are selected
`CAP`	Only capital letters and digits are selected
`CAP1`	Same as `CAP`, but must switch to `ALPHA` mode after the first character is validated.

Table 4-8: Keypad selection modes

A keypad event can be generated using `Keyboard.send(int type, char keycode)`.

4.12.6 STATE

4.12.6.1 Event format

The data of `STATE` event is composed of a state value and a state id (Illustration 4-18), so it allows at most 256 states to have at most 256 different state values.

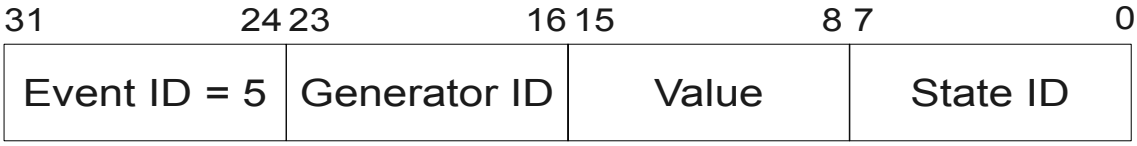


Illustration 4-18: States event format

4.12.6.2 Event generator

A `States` event generator is usually associated to a group of physical devices holding a position (switch, rotary wheel encoder, ...) and allows to generate events relating to them. A state has a unique ID between `0` and `States.nbStates()-1`. `States.currentValue(int)` allows to retrieve the current value of a state, and `States.nbValues(int)` gives the total number of values allowed for this state.

A `STATE` event can be generated using `States.send(int stateId, int newValue)`. The given value is stored to be the new current value for the given state and sends the event to the registered listener if any.

4.12.7 POINTER_BUTTON (Deprecated)

4.12.7.1 Event format

The data of pointer buttons event is composed of an action and a button id (Illustration 4-19), so it allows at most 256 buttons to have at most 256 different actions. MicroUI defines 6 basic actions described in Table 4-6.

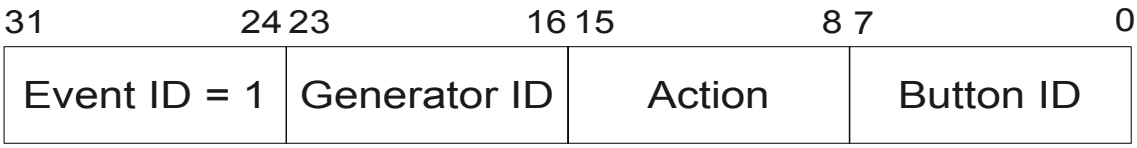


Illustration 4-19: PointerButton event format

4.12.7.2 Event generator

A `PointerButtons` event generator is usually associated to a group of physical buttons that are linked to a `Pointer`.
When an event is sent by this generator, the related `Pointer` instance can be retrieved in order to get the `(x,y)` coordinates of the button action.
A pointer button event can be generated using `PointerButtons .send(int buttonId, int action)`.

4.13 Thread-safe framework

A MicroUI implementation MUST be thread-safe in the sense that:

- any method can be called several times from several threads concurrently on the same receiver without jeopardizing the integrity of that receiver.
- the user application may synchronize on any object without causing a dead-lock with the implementation of MicroUI.

This definition allows the number and the size of critical sections to be minimized, without compromising the robustness of the framework. However it does not require that the outcome of concurrent access is the same as if the accesses had been sequential. For example, if two threads try simultaneously to add a listener to the same `Displayable` object, the MicroUI implementation does not have to ensure that both additions will be effective. Nevertheless, the MicroUI implementation must ensure that the `Displayable` object, after the listener additions, is in a safe and consistent state.

Because any MicroUI implementation must not lock on user's objects, the user is free to synchronize its application design according to its need without risking mysterious deadlocks.

5 APPENDIX

5.1 Class diagram

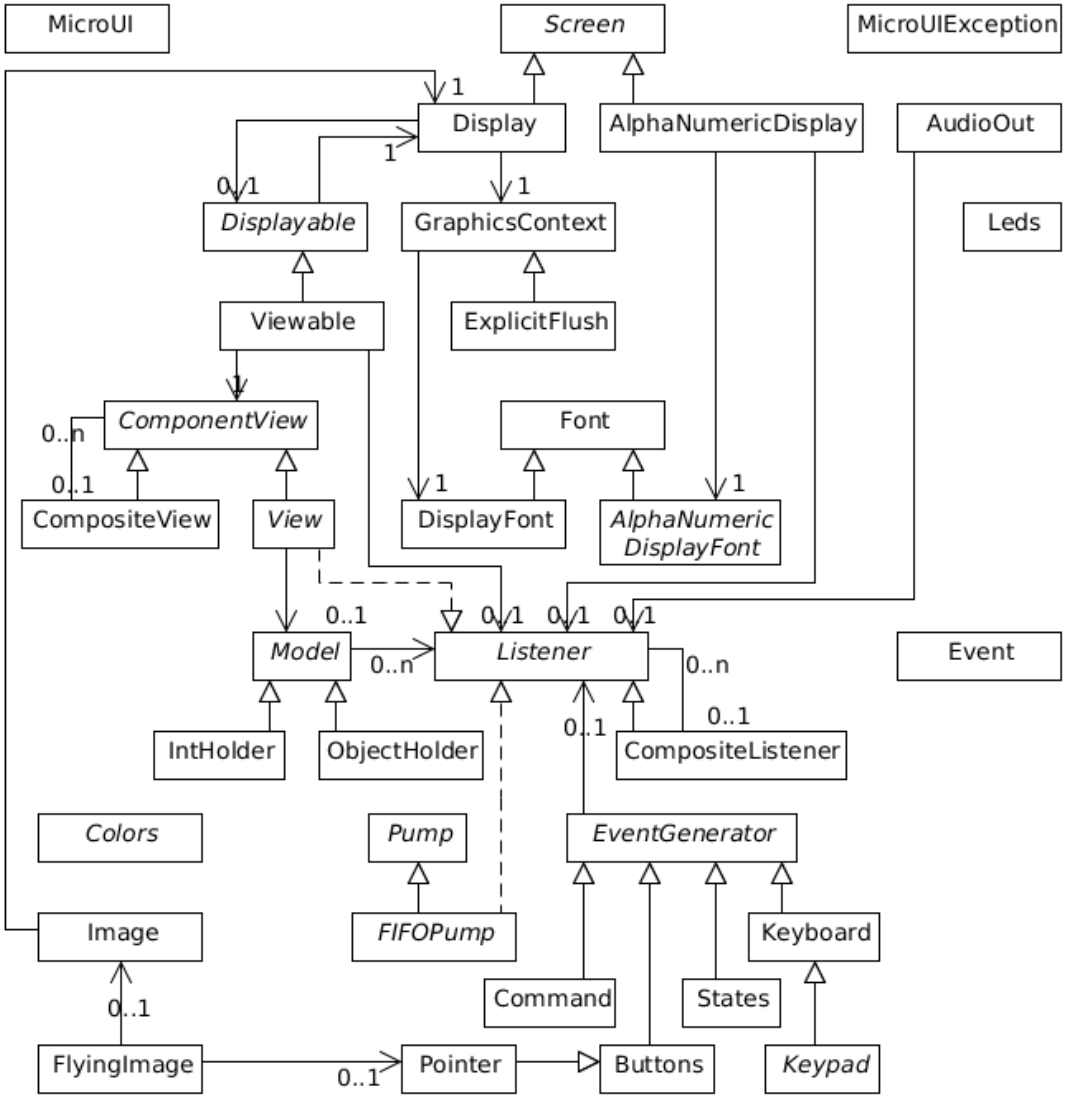


Illustration 5-1: MicroUI class diagram

5.2 Font identifiers

Table 5-1 describes MicroUI font identifiers, based on Unicode scripts [Unicode].

Unicode script name	MicroUI identifier	Unicode script name	MicroUI identifier	Unicode script name	MicroUI identifier
ARABIC	1	KHMER	32	INHERITED	63
ARMENIAN	2	LAO	33	SUNDANESE	64
BALINESE	3	LATIN	34	LEPCHA	65

BENGALI	4	LIMBU	35	OL_CHIKI	66
BOPOMOFO	5	MALAYALAM	36	VAI	67
BRAILLE	6	MONGOLIAN	37	SAURASHTRA	68
BUGINESE	7	MYANMAR	38	KAYAH_LI	69
BUHID	8	NEW_TAI_LUE	39	REJANG	70
CANADIAN_ABORIGINAL	9	NKO	40	CHAM	71
CHEROKEE	10	OGHAM	41	TAI_THAM	72
COPTIC	11	ORIYA	42	TAI_VIET	73
unused	12	unused	43	SAMARITAN	74
unused	13	PHAGS_PA	44	LISU	75
CYRILLIC	14	unused	45	BAMUM	76
unused	15	RUNIC	46	JAVANESE	77
DEVANAGARI	16	unused	47	MEITEI_MAYEK	78
ETHIOPIC	17	SINHALA	48	BATAK	79
GEORGIAN	18	SYLOTI_NAGRI	49	MANDAIC	80
GLAGOLITIC	19	SYRIAC	50		
unused	20	TAGALOG	51		
GREEK	21	TAGBANWA	52		
GUJARATI	22	TAI_LE	53		
GURMUKHI	23	TAMIL	54		
HAN	24	TELUGU	55		
HANGUL	25	THAANA	56		
HANUNOO	26	THAI	57		
HEBREW	27	TIBETAN	58		
HIRAGANA	28	TIFINAGH	59		
KANNADA	29	unused	60		
KATAKANA	30	YI	61		
unused	31	COMMON	62		

Table 5-1: MicroUI fonts identifiers based on Unicode scripts

5.3 Image formats

Table 5-2 describes MicroUI image formats.

Format constant name				
APP1	FPX	M	PDB	SHTML
ART	FRACTAL	M2V	PDF	STEGANO
AVI	FTP	MAP	PFA	SUN

AVS	G	MAT	PFB	SVG
B	G3	MATTE	PGM	TEXT
BIE	GIF	MIFF	PICON	TGA
BIM8	GIF87	MNG	PICT	TIF
BMP	GRADIENT	MONO	PIX	TIFF
BMP_MONOCHROM	GRANITE	MPC	PLASMA	TILE
C	GRAY	MPEG	PM	TIM
CAPTION	H	MPG	PNG	TTF
CMYK	HDF	MPR	PNM	TXT
CMYKA	HISTOGRAM	MPRI	PPM	UIL
CUT	HTM	MSL	PREVIEW	UYVY
DCM	HTML	MTV	PS	VDA
DCX	HTTP	MVG	PS2	VICAR
DIB	ICB	NETSCAPE	PS3	VID
DPS	ICM	NULL	PSD	VIFF
DPX	ICO	O	PTIF	VST
EPDF	ICON	OTB	PWP	WBMP
EPI	IPTC	P7	R	WPG
EPS	JBG	PAL	RAS	X
EPS2	JBIG	PALM	RGB	XBM
EPS3	JP2	PBM	RGBA	XC
EPSF	JPC	PCD	RLA	XCF
EPSI	JPEG	PCDS	RLE	XPM
EPT	JPG	PCL	ROSE	XV
FAX	K	PCT	SCT	XWD
FILE	LABEL	PCX	SFW	Y
FITS	LOGO		SGI	YUV

Table 5-2: MicroUI supported image formats

6 JAVA SPECIFICATION

Package ej.microui

Interface Summary		Page
Colors	The interface `Colors` provides useful constants to handle RGB colors format.	*45*
Listener	The classes which implements this interface have to perform an action when one of the performAction() methods is called.	*71*

Class Summary		Page
Command	`Command` is an event generator that generates application-level events.	*49*
CompositeListener	`CompositeListener` holds an array of Listener.	*55*
Event	MicroUI features int-based events, allowing for a rich event mechanism compatible with scare resources.	*57*
EventGenerator	`EventGenerator`s generate int-based events (see Event).	*61*
FIFOPump	A `FIFOPump` is a `Pump` which holds a FIFO (First IN - First OUT) queue of data waiting to be processed.	*65*
IntHolder	An `IntHolder` is a generic model that wraps one signed 32-bit int.	*69*
MicroUI	The `MicroUI` class offers basic services in the MicroUI implementation.	*72*
Model	`Model` is an abstract class which represents the subject of a View.	*75*
ObjectHolder	An `ObjectHolder` is a generic model that wraps one object and notifies its listeners with the method Model.changed(int, Object), passing `0` and the new object as arguments, when the wrapped object is replaced by another one.	*78*
Pump	A `Pump` holds a `Thread` in order to execute actions.	*80*

Exception Summary		Page
MicroUIException	Thrown when a MicroUI implementation error has occurred.	*74*

Interface Colors

ej.microui

`public interface` **`Colors`**

The interface `Colors` provides useful constants to handle RGB colors format.

RGB colors format is as follow:
| color's red level (`8-bit`) | color's green level (`8-bit`) | color's blue level (`8-bit`) |

Field Summary		Page
int	**BLACK** The black RGB color constant.	*46*
int	**BLUE** The blue RGB color constant.	*46*
int	**CYAN** The cyan RGB color constant.	*46*
int	**GRAY** The gray RGB color constant.	*46*
int	**GREEN** The green RGB color constant.	*46*
int	**LIME** The lime RGB color constant.	*46*
int	**MAGENTA** The magenta RGB color constant.	*47*
int	**MAROON** The maroon RGB color constant.	*47*
int	**NAVY** The navy RGB color constant.	*47*
int	**OLIVE** The olive RGB color constant.	*47*
int	**PURPLE** The purple RGB color constant.	*47*
int	**RED** The red RGB color constant.	*47*
int	**SILVER** The silver RGB color constant.	*48*
int	**TEAL** The teal RGB color constant.	*48*
int	**WHITE** The white RGB color constant.	*48*
int	**YELLOW** The yellow RGB color constant.	*48*

Field Detail

BLACK

```
public static final int BLACK
```

The black RGB color constant.

The value `0x000000` is assigned to `BLACK`.

BLUE

```
public static final int BLUE
```

The blue RGB color constant.

The value `0x0000ff` is assigned to `BLUE`.

CYAN

```
public static final int CYAN
```

The cyan RGB color constant.

The value `0x00ffff` is assigned to `CYAN`.

GRAY

```
public static final int GRAY
```

The gray RGB color constant.

The value `0x808080` is assigned to `GRAY`.

GREEN

```
public static final int GREEN
```

The green RGB color constant.

The value `0x008000` is assigned to `GREEN`.

LIME

```
public static final int LIME
```

The lime RGB color constant.

The value `0x00ff00` is assigned to `LIME`.

MAGENTA

```
public static final int MAGENTA
```

The magenta RGB color constant.

The value `0xff00ff` is assigned to `MAGENTA`.

MAROON

```
public static final int MAROON
```

The maroon RGB color constant.

The value `0x800000` is assigned to `MAROON`.

NAVY

```
public static final int NAVY
```

The navy RGB color constant.

The value `0x000080` is assigned to `NAVY`.

OLIVE

```
public static final int OLIVE
```

The olive RGB color constant.

The value `0x808000` is assigned to `OLIVE`.

PURPLE

```
public static final int PURPLE
```

The purple RGB color constant.

The value `0x800080` is assigned to `PURPLE`.

RED

```
public static final int RED
```

The red RGB color constant.

The value `0xff0000` is assigned to `RED`.

SILVER

```
public static final int SILVER
```

The silver RGB color constant.

The value `0xc0c0c0` is assigned to `SILVER`.

TEAL

```
public static final int TEAL
```

The teal RGB color constant.

The value `0x008080` is assigned to `TEAL`.

WHITE

```
public static final int WHITE
```

The white RGB color constant.

The value `0xffffff` is assigned to `WHITE`.

YELLOW

```
public static final int YELLOW
```

The yellow RGB color constant.

The value `0xffff00` is assigned to `YELLOW`.

Class Command

ej.microui

```
java.lang.Object
  └
    ej.microui.EventGenerator
      └
        ej.microui.Command
```

```
public class Command
extends EventGenerator
```

`Command` is an event generator that generates application-level events. Unlike io generators that are related to some hardware input format, the command generator defines its own input format. Basically input and output event are the same. This allows the generation of commands from within MicroUI without relying on an underlying input event format.

This class defines constants for a set of basic commands. The commands defined in this class are typical application-level effects of input events. The advantage of using commands rather than specific input events in an application is that the application can be more portable: it is not tied to specific input devices.

Field Summary		Page
static int	**ANTICLOCKWISE** The "anti-clockwise" command constant.	53
static int	**BACK** The "back" command constant.	50
static int	**CANCEL** The "cancel" command constant.	51
static int	**CLOCKWISE** The "clockwise" command constant.	53
static int	**COPY** The "copy" command constant.	52
static int	**CUT** The "cut" command constant.	53
static int	**DISPLAY** The "display" command constant.	53
static int	**DOWN** The "down" command constant.	51
static int	**ESC** The "escape" command constant.	50
static int	**EXIT** The "exit" command constant.	52
static int	**HELP** The "help" command constant.	51
static int	**LEFT** The "left" command constant.	51
static int	**MENU** The "menu" command constant.	52
static int	**NEXT** The "next" command constant.	53
static int	**PASTE** The "paste" command constant.	53

static int	**PAUSE** The "pause" command constant.	*52*
static int	**PREVIOUS** The "previous" command constant.	*53*
static int	**RESUME** The "resume" command constant.	*52*
static int	**RIGHT** The "right" command constant.	*51*
static int	**SELECT** The "select" command constant.	*51*
static int	**START** The "start" command constant.	*52*
static int	**STOP** The "stop" command constant.	*52*
static int	**UP** The "up" command constant.	*51*

Constructor Summary	*Page*
Command() Creates a new command `EventGenerator`.	*54*

Method Summary		*Page*
int	**getEventType**() Gets the `Command` event generator's type.	*54*
void	**send**(int command) Sends the given command to the event generator's listener	*54*

Methods inherited from class ej.microui.EventGenerator
addToSystemPool, eventType, get, get, get, getID, getListener, removeFromSystemPool, setListener

Field Detail

ESC

```
public static final int ESC
```

The "escape" command constant.
The value `0x0000` is assigned to `ESC`.

BACK

```
public static final int BACK
```

The "back" command constant.
The value `0x0001` is assigned to `BACK`.

UP

```
public static final int UP
```

The "up" command constant.
The value `0x0002` is assigned to `UP`.

DOWN

```
public static final int DOWN
```

The "down" command constant.
The value `0x0003` is assigned to `DOWN`.

LEFT

```
public static final int LEFT
```

The "left" command constant.
The value `0x0004` is assigned to `LEFT`.

RIGHT

```
public static final int RIGHT
```

The "right" command constant.
The value `0x0005` is assigned to `RIGHT`.

SELECT

```
public static final int SELECT
```

The "select" command constant.
The value `0x0006` is assigned to `SELECT`.

CANCEL

```
public static final int CANCEL
```

The "cancel" command constant.
The value `0x0007` is assigned to `CANCEL`.

HELP

```
public static final int HELP
```

The "help" command constant.
The value `0x0008` is assigned to `HELP`.

MENU

```
public static final int MENU
```

The "menu" command constant.
The value `0x0009` is assigned to `MENU`.

EXIT

```
public static final int EXIT
```

The "exit" command constant.
The value `0x000A` is assigned to `EXIT`.

START

```
public static final int START
```

The "start" command constant.
The value `0x000B` is assigned to `START`.

STOP

```
public static final int STOP
```

The "stop" command constant.
The value `0x000C` is assigned to `STOP`.

PAUSE

```
public static final int PAUSE
```

The "pause" command constant.
The value `0x000D` is assigned to `PAUSE`.

RESUME

```
public static final int RESUME
```

The "resume" command constant.
The value `0x000E` is assigned to `RESUME`.

COPY

```
public static final int COPY
```

The "copy" command constant.
The value `0x000F` is assigned to `COPY`.

CUT

```
public static final int CUT
```

The "cut" command constant.
The value `0x0010` is assigned to `CUT`.

PASTE

```
public static final int PASTE
```

The "paste" command constant.
The value `0x0011` is assigned to `PASTE`.

CLOCKWISE

```
public static final int CLOCKWISE
```

The "clockwise" command constant.
The value `0x0012` is assigned to `CLOCKWISE`.

ANTICLOCKWISE

```
public static final int ANTICLOCKWISE
```

The "anti-clockwise" command constant.
The value `0x0013` is assigned to `ANTICLOCKWISE`.

PREVIOUS

```
public static final int PREVIOUS
```

The "previous" command constant.
The value `0x0014` is assigned to `PREVIOUS`.

NEXT

```
public static final int NEXT
```

The "next" command constant.
The value `0x0015` is assigned to `NEXT`.

DISPLAY

```
public static final int DISPLAY
```

The "display" command constant.
The value `0x0016` is assigned to `DISPLAY`.

Constructor Detail

Command

```
public Command()
```

Creates a new command `EventGenerator`.

Method Detail

getEventType

```
public int getEventType()
```

Gets the `Command` event generator's type. Default value is Event.COMMAND.

Overrides:
getEventType in class EventGenerator
Returns:
the command generator's type

---

send

```
public void send(int command)
```

Sends the given command to the event generator's listener

Parameters:
`command` - the command to be sent

Class CompositeListener

ej.microui

```
java.lang.Object
  └
    ej.microui.CompositeListener
```

All Implemented Interfaces:

Listener

```
public class CompositeListener
extends Object
implements Listener
```

`CompositeListener` holds an array of Listener. Note that `CompositeListener` is also a `Listener`. When `CompositeListener` is asked to perform an action, it asks its listeners to perform the action.

Constructor Summary	*Page*
CompositeListener`()` Creates a `CompositeListener` with no listener.	*55*

Method Summary		*Page*
`void`	**add** (Listener `listener`) Adds a `Listener` to the list of listeners.	*56*
`void`	**performAction**`()` Sends a performAction to each listener	*55*
`void`	**performAction**`(int value)` Sends the `value` to each listener	*56*
`void`	**performAction**`(int value, Object object)` Sends the `value` and the `object` to each listener	*56*
`void`	**remove** (Listener `listener`) Removes a `Listener` from the list of listeners.	*56*

Constructor Detail

CompositeListener

```
public CompositeListener()
```

Creates a `CompositeListener` with no listener.

Method Detail

performAction

```
public void performAction()
```

Sends a performAction to each listener

Specified by:
performAction in interface Listener

---

performAction

```
public void performAction(int value)
```

Sends the `value` to each listener

Specified by:
performAction in interface Listener
Parameters:
`value` - the value to perform on.

---

performAction

```
public void performAction(int value,
                          Object object)
```

Sends the `value` and the `object` to each listener

Specified by:
performAction in interface Listener
Parameters:
`value` - the value to perform on.
`object` - the object given by the model.

---

add

```
public void add(Listener listener)
```

Adds a `Listener` to the list of listeners.

Parameters:
`listener` - the `Listener` to add.
Throws:
`IllegalArgumentException` - if `listener` is null.

---

remove

```
public void remove(Listener listener)
```

Removes a `Listener` from the list of listeners. If the listener is not in the list, nothing is done.

Parameters:
`listener` - the `Listener` to be removed.
Throws:
`NullPointerException` - if `listener` is null.

Class Event

ej.microui

```
java.lang.Object
  └
    ej.microui.Event
```

```
public class Event
extends Object
```

MicroUI features int-based events, allowing for a rich event mechanism compatible with scare resources. The `Event` class provides EventGenerator classes with event constants and helper methods to build and analyse events.

An event has a type, a 8-bit figure that forms the most significant byte of the int-event, followed by 8-bits which is the generator id quantity, and followed by 16-bit of data.

event : type (8-bit) + generatorID (8-bit) + data (16-bit)

The very first 16 types [0x00..0x0f], some of which are defined by constants in this class, are MicroUI reserved. An application may create as many as 240 different kind of events.

See Also:
EventGenerator

Field Summary		Page
static int	**BUTTON** The BUTTON event type.	58
static int	**COMMAND** The COMMAND event type.	58
static int	**KEYBOARD** The KEYBOARD event type.	58
static int	**KEYPAD** The KEYPAD event type.	58
static int	**POINTER** The POINTER event type.	58
static int	**POINTER_BUTTON** **Deprecated.**	59
static int	**STATE** The STATE event type.	58

Method Summary		Page
static int	**buildEvent**`(int type, EventGenerator gen, int data)` Builds an event from a given type, an eventGenerator and data.	59
static int	**getData**`(int event)` Returns the event's data issued by a generator.	59
static EventGenerator	**getGenerator**`(int event)` Gets a converter out of an event assuming the event has been generated by an EventGenerator that has been previously added to the system pool.	60
static int	**getGeneratorID**`(int event)` Returns the event's generator id.	60

static int	**getType**(int event) Returns the type of an event.	59

Field Detail

COMMAND

```
public static final int COMMAND
```

The COMMAND event type.

The value 0x00 is assigned to COMMAND.

BUTTON

```
public static final int BUTTON
```

The BUTTON event type.

The value 0x01 is assigned to BUTTON.

KEYBOARD

```
public static final int KEYBOARD
```

The KEYBOARD event type.

The value 0x02 is assigned to KEYBOARD.

POINTER

```
public static final int POINTER
```

The POINTER event type.

The value 0x03 is assigned to POINTER.

KEYPAD

```
public static final int KEYPAD
```

The KEYPAD event type.

The value 0x04 is assigned to KEYPAD.

STATE

```
public static final int STATE
```

The STATE event type.

The value `0x05` is assigned to `STATE`.

POINTER_BUTTON

```
public static final int POINTER_BUTTON
```

Deprecated.

The POINTER_BUTTON event type.

The value `0x06` is assigned to `POINTER_BUTTON`.

Since:
1.3.2

Method Detail

buildEvent

```
public static int buildEvent(int type,
                             EventGenerator gen,
                             int data)
```

Builds an event from a given type, an eventGenerator and data.

Parameters:
`type` - the type of the event to build
`gen` - the generator associated with the event
`data` - the data of the event to build
Returns:
the event as an `int`

---

getType

```
public static int getType(int event)
```

Returns the type of an event.

Parameters:
`event` - an event
Returns:
event's type as an `int`

---

getData

```
public static int getData(int event)
```

Returns the event's data issued by a generator.

Parameters:
`event` - an event

Returns:
event's data as an `int`

---

getGeneratorID

```
public static int getGeneratorID(int event)
```

Returns the event's generator id.

Parameters:
`event` - an event
Returns:
event's generator as an `int`

---

getGenerator

```
public static EventGenerator getGenerator(int event)
```

Gets a converter out of an event assuming the event has been generated by an EventGenerator that has been previously added to the system pool.

Parameters:
`event` - an event
Returns:
the associated EventGenerator
Throws:
`NullPointerException` - if the generator does not exist (most likely because the event is not an EventGenerator related event).
See Also:
EventGenerator.addToSystemPool()

Class EventGenerator

ej.microui

```
java.lang.Object
  └
    ej.microui.EventGenerator
```

Direct Known Subclasses:
Buttons, Command, Keyboard, States

```
abstract public class EventGenerator
extends Object
```

EventGenerators generate int-based events (see Event). They are responsible for holding data that cannot be sent within the event. EventGenerators are state machines that convert serialized (mostly external) integers to MicroUI events.

There is a system pool that holds generators. A generator in the system pool has an ID between 0 and 254, otherwise ID is 0xFF. The advantage of putting a generator in the system pool is that it can then be looked-up using get(int) and get(Class, int).

See Also:
Event

Constructor Summary	Page
EventGenerator() Creates a new EventGenerator.	62

Method Summary		Page
int	**addToSystemPool**() Adds the generator in the system pool.	62
int	**eventType**() **Deprecated.** *use getEventType()*	63
static EventGenerator	**get**(int id) Gets a generator from its id	62
static EventGenerator[]	**get**(Class clazz) Gets all generators whose class is clazz from the system pool.	63
static EventGenerator	**get**(Class clazz, int fromIndex) Gets a generator whose class is clazz from the system pool starting the search from fromIndex.	63
abstract int	**getEventType**() Gets the event type associated with the event generator	64
int	**getID**() Gets the generator's unique id.	63
Listener	**getListener**() Gets the generator's listener	63
void	**removeFromSystemPool**() Removes the generator from the system generators pool.	62
void	**setListener**(Listener listener) Sets the EventGenerator's listener.	62

Constructor Detail

EventGenerator

```
public EventGenerator()
```

Creates a new `EventGenerator. As soon as a new EventGenerator is added to the system pool it has a valid id.`

Method Detail

addToSystemPool

```
public int addToSystemPool()
```

Adds the generator in the system pool. The generator can only be added once as a system generator. When added, its id is set.

Returns:
the `EventGenerator`'s id in the system pool.

Throws:
`RuntimeException` - if the maximum number of `EventGenerators` added to the system pool has been reached

---

removeFromSystemPool

```
public void removeFromSystemPool()
```

Removes the generator from the system generators pool. If the generator is not in the pool, nothing is done.

---

setListener

```
public void setListener(Listener listener)
```

Sets the `EventGenerator`'s listener. It replaces the old one and can be null.

Parameters:
`listener` - the new listener

---

get

```
public static EventGenerator get(int id)
```

Gets a generator from its id

Parameters:
`id` - the generator's id

Returns:
the associated `EventGenerator`.

Throws:
`IndexOutOfBoundsException` - if the generator does not exist (most likely because the event is not an EventGenerator related event).

---

get

```
public static EventGenerator get(Class clazz,
                                 int fromIndex)
```

Gets a generator whose class is `clazz` from the system pool starting the search from `fromIndex`. If class `clazz` is not an `EventGenerator` or if nothing is found, returns `null`.

Parameters:
`clazz` - the `EventGenerator`'s class to return
`fromIndex` - index from which starting to search
Returns:
the `EventGenerator` or null.

---

get

```
public static EventGenerator[] get(Class clazz)
```

Gets all generators whose class is `clazz` from the system pool. If class `clazz` is not an `EventGenerator` or if nothing is found, returns an empty array.

Parameters:
`clazz` - the `EventGenerator`'s class to return
Returns:
the array of `EventGenerator`.

---

getID

```
public int getID()
```

Gets the generator's unique id. Up to 254 `EventGenerator`s can be installed as MicroUI system event generator.

Returns:
the generator's id as an `int`.

---

getListener

```
public Listener getListener()
```

Gets the generator's listener

Returns:
the generator's `Listener`.

---

eventType

```
public int eventType()
```

Deprecated. *use getEventType()*

---

getEventType

```
public abstract int getEventType()
```

Gets the event type associated with the event generator

Returns:
the event type

Class FIFOPump

ej.microui

```
java.lang.Object
  └
    ej.microui.Pump
      └
        ej.microui.FIFOPump
```

All Implemented Interfaces:

Listener, Runnable

```
abstract public class FIFOPump
extends Pump
implements Listener
```

A `FIFOPump` is a `Pump` which holds a FIFO (First IN - First OUT) queue of data waiting to be processed. A `FIFOPump` is a `Listener` and fills its FIFO at each call of the `performAction()` method. The method `read()` returns the oldest data stored into the FIFO.

By default, the FIFO grows indefinitely. Subclasses must override the method getMaxSize() to specify their own policy. When the FIFO is full and cannot grow, `FIFOPump` calls the dropOnFull() method. By default this method returns `false`, which means the new data will overwrite the oldest data stored into the FIFO. To discard the new data the subclass must override this method in order to return `true`.

Subclasses must implement Pump.execute(int) to specify how queued data is to be processed.

Fields inherited from class ej.microui.Pump
DEFAULT_PRIORITY

Constructor Summary	Page
FIFOPump() Creates a new `FIFOPump` without specify the FIFO's default size.	66
FIFOPump(int defaultSize) Creates a new `FIFOPump` specifying the FIFO's default size.	66

Method Summary		Page
boolean	**dropOnFull**() Subclasses should override this method to specify their policy.	66
void	**fifoDropNewEvent**() Drop event at the the beginning index	67
void	**fifoDropOldestEvent**() Drop oldest event in the eventsQueue	67
void	**full**(int event) Called when queue is full, with the event that has not been added.	67
int	**getMaxSize**() Returns the maximum size of the FIFO.	66
void	**performAction**() The default behavior is to do nothing.	68
void	**performAction**(int value) This method adds the `value` to the FIFO.	67

void	**performAction**(int value, Object object) The default behavior is identical to performAction(value).	*68*
int	**read**() Returns the oldest data or waits for it.	*67*

Methods inherited from class ej.microui.Pump
addToSystemPool, crash, execute, getName, getPriority, isStopping, removeFromSystemPool, run, setPriority, start, stop

Constructor Detail

FIFOPump

```
public FIFOPump()
```

Creates a new FIFOPump without specify the FIFO's default size. The effect is identical to
`new FIFOPump(5);`

FIFOPump

```
public FIFOPump(int defaultSize)
```

Creates a new FIFOPump specifying the FIFO's default size.

Parameters:
defaultSize - the FIFO's default size.
Throws:
IllegalArgumentException - if defaultSize = 0

Method Detail

getMaxSize

```
public int getMaxSize()
```

Returns the maximum size of the FIFO. The sub-classes must override this method to specify their policy. By default this method returns -1, that means the FIFO can grow indefinitely. This method must be called by the MicroUI framework implementation as soon as the FIFO is full.

Returns:
the maximum size of the FIFO

---

dropOnFull

```
public boolean dropOnFull()
```

Subclasses should override this method to specify their policy. By default this method returns false, which means the oldest data are overwritten by new data. If it returns true new data are dropped when the pump is full.

Returns:
true to drop the new data or false to overwrite the oldest data.

---

read

```
public int read()
```

Returns the oldest data or waits for it. This method blocks until data is available.

Overrides:
read in class Pump
Returns:
the oldest data added to the FIFO

---

performAction

```
public void performAction(int value)
```

This method adds the `value` to the FIFO. When the FIFO is full, getMaxSize() is called to know if the FIFO must grow or not. If not, `dropOnFull()` is called to know which event to drop.

Specified by:
performAction in interface Listener
Parameters:
`value` - the new data to be added to the FIFO

---

full

```
public void full(int event)
```

Called when queue is full, with the event that has not been added. By default, one event is dropped. If dropOnFull() returns true it calls fifoDropNewEvent(), otherwise fifoDropOldestEvent() is called.

Since:
1.3.2

---

fifoDropOldestEvent

```
public void fifoDropOldestEvent()
```

Drop oldest event in the eventsQueue

See Also:
performAction(int)

---

fifoDropNewEvent

```
public void fifoDropNewEvent()
```

Drop event at the the beginning index

See Also:
performAction(int)

---

performAction

```
public void performAction(int value,
                          Object object)
```

The default behavior is identical to `performAction(value)`.

Specified by:
performAction in interface Listener

Parameters:
`value` - the new data to be added to the FIFO
`object` - dropped by default

---

performAction

```
public void performAction()
```

The default behavior is to do nothing.

Specified by:
performAction in interface Listener

Class IntHolder

ej.microui

```
java.lang.Object
  └
    ej.microui.Model
      └
        ej.microui.IntHolder
```

```
public class IntHolder
extends Model
```

An `IntHolder` is a generic model that wraps one signed 32-bit int. It notifies its listeners with the method Model.changed(int) when the wrapped int is changed.

Constructor Summary	Page
IntHolder`()` Creates an instance with `0` as default value.	69
IntHolder`(int value)` Creates an instance with the given default value.	69

Method Summary		Page
`int`	**get**`()` Returns the wrapped int	70
`void`	**set**`(int value)` Sets a new int.	70

Methods inherited from class ej.microui.Model
addListener, addListeners, changed, changed, changed, getAllListeners, hasListeners, removeAllListeners, removeListener

Constructor Detail

IntHolder

```
public IntHolder()
```

Creates an instance with `0` as default value.

IntHolder

```
public IntHolder(int value)
```

Creates an instance with the given default value.

Since:

1.3.2

Method Detail

get

```
public int get()
```

Returns the wrapped int

Returns:
the wrapped `int`

---

set

```
public void set(int value)
```

Sets a new int. If the new wrapped int is different from the existing one, listeners are notified.

Parameters:
`value` - new int to be set

Interface Listener

ej.microui

All Known Implementing Classes:
CompositeListener, FIFOPump, View

```
public interface Listener
```

The classes which implements this interface have to perform an action when one of the performAction() methods is called. This interface is used by the MicroUI Event mechanism to send a MicroUI Event to a listener.

For reasons of efficiency it is preferred to notify listeners using an int rather than an object. It is for that reason that this interface does not contain a `performAction(Object)` method.

Method Summary		Page
void	**performAction**() Performs action.	71
void	**performAction**(int value) Performs action according `value`.	71
void	**performAction**(int value, Object object) Performs action according `value` and `object`.	71

Method Detail

performAction

```
void performAction()
```

Performs action.

---

performAction

```
void performAction(int value)
```

Performs action according `value`.

Parameters:
`value` - the value to perform on.

---

performAction

```
void performAction(int value,
                   Object object)
```

Performs action according `value` and `object`.

Parameters:
`value` - the value to perform on.
`object` - the object given by the model.

Class MicroUI

ej.microui

```
java.lang.Object
  └
    ej.microui.MicroUI
```

```
public class MicroUI
extends Object
```

The `MicroUI` class offers basic services in the MicroUI implementation.
MicroUI may start automatically with the application or explicitly by calling start(). Whether or not start is automatic depends on a property.
MicroUI may also be stopped with stop().

Error logging may be activated with errorLog(boolean) and a simple alarm mechanism may be activated with beep().

Method Summary		Page
static void	**beep**() Generates a basic alert mechanism.	73
static void	**errorLog**(boolean show) Sets whether error log information is shown.	73
static void	**start**() Starts MicroUI.	72
static void	**stop**() Stops MicroUI.	72

Method Detail

start

```
public static void start()
```

Starts MicroUI.
It implies starting event serialization as well as rendering mechanisms.
It also starts all the pumps in the pumps pool.

See Also:
Pump.addToSystemPool()

---

stop

```
public static void stop()
```

Stops MicroUI.
It implies stopping any potential event serialization as well as rendering mechanisms.
It also stops all the pumps in the pumps pool.

See Also:
Pump.addToSystemPool()

---

errorLog

```
public static final void errorLog(boolean show)
```

Sets whether error log information is shown. The way log errors are shown is implementation dependent. By default logging is disabled.

Parameters:

`show` - if true activates error log

---

beep

```
public static void beep()
```

Generates a basic alert mechanism. Result is implementation dependent.

Class MicroUIException

ej.microui

```
java.lang.Object
  └
    java.lang.Throwable
      └
        java.lang.Exception
          └
            java.lang.RuntimeException
              └
                ej.microui.MicroUIException
```

```
public class MicroUIException
extends RuntimeException
```

Thrown when a MicroUI implementation error has occurred.

Constructor Summary	Page
MicroUIException() Constructs an MicroUIException with no detail message.	74
MicroUIException(String message) Constructs an MicroUIException with the specified detail message.	74

Constructor Detail

MicroUIException

```
public MicroUIException()
```

Constructs an MicroUIException with no detail message.

MicroUIException

```
public MicroUIException(String message)
```

Constructs an MicroUIException with the specified detail message.

Parameters:

`message` - the exception message

Class Model

ej.microui

```
java.lang.Object
  └
    ej.microui.Model
```

Direct Known Subclasses:
IntHolder, ObjectHolder

```
abstract public class Model
extends Object
```

`Model` is an abstract class which represents the subject of a View. It can hold several listeners and notify them when changes occur.

This class is thread-safe regarding its internal state. Although all methods may be called concurrently, it is the responsibility of the caller to manage synchronization. For instance, two concurrent `addListener(...)` calls may produce the following result: one of the two added listeners is dropped but nevertheless it produces a coherent result, i.e. it does not cause exceptions to be thrown.

See Also:
Listener

Constructor Summary	Page
Model()	76

Method Summary		Page
void	**addListener**(Listener listener) Adds the listener to the model.	76
void	**addListeners**(Listener[] listeners) Adds the `listeners` to the model.	76
void	**changed**() Notify all model's listeners.	77
void	**changed**(int value) Notify all model's listeners, using an int.	77
void	**changed**(int value, Object object) Notify all listeners of the model, using one object and one int value.	77
Listener[]	**getAllListeners**() Returns an array containing all the listeners of the model.	77
boolean	**hasListeners**() Indicates whether the model has some listeners or not.	77
void	**removeAllListeners**() Removes all listeners from the model.	76
void	**removeListener**(Listener listener) Removes the listener from the model.	76

Constructor Detail

Model

```
public Model()
```

Method Detail

addListener

```
public void addListener(Listener listener)
```

Adds the listener to the model. Caller has to manage synchronization.

Parameters:
`listener` - the new listener to add
Throws:
`IllegalArgumentException` - if listener is null

---

addListeners

```
public void addListeners(Listener[] listeners)
```

Adds the `listeners` to the model. If called concurrently this method does not ensure that `listeners` will be effectively added to the list of listeners. Caller has to manage synchronization.

Parameters:
`listeners` - the new listeners to be added
Throws:
`IllegalArgumentException` - if one of the listeners is null

---

removeListener

```
public void removeListener(Listener listener)
```

Removes the listener from the model. If the listener is not a model's listener, this method has no effect. If called concurrently this method does not ensure that `listener` will be effectively removed from the list of listeners. Caller has to manage synchronization.

Parameters:
`listener` - the listener to be removed

---

removeAllListeners

```
public void removeAllListeners()
```

Removes all listeners from the model.

---

hasListeners

```
public boolean hasListeners()
```

Indicates whether the model has some listeners or not.

Returns:
`true` if the model has at least one listener, `false` otherwise.

---

getAllListeners

```
public final Listener[] getAllListeners()
```

Returns an array containing all the listeners of the model.

Returns:
an array `Listener[]` with all the listeners of the model

---

changed

```
public void changed()
```

Notify all model's listeners.

---

changed

```
public void changed(int value)
```

Notify all model's listeners, using an int.

Parameters:
`value` - the int to send to the listeners

---

changed

```
public void changed(int value,
                    Object object)
```

Notify all listeners of the model, using one object and one int value.

Parameters:
`value` - the int to send to the listeners
`object` - the object to send to the listeners

Class ObjectHolder

ej.microui

```
java.lang.Object
  └
    ej.microui.Model
      └
        ej.microui.ObjectHolder
```

```
public class ObjectHolder
extends Model
```

An `ObjectHolder` is a generic model that wraps one object and notifies its listeners with the method Model.changed(int, Object), passing `0` and the new object as arguments, when the wrapped object is replaced by another one.

Constructor Summary	Page
ObjectHolder`()` Creates an instance with `null` as default value.	78
ObjectHolder`(Object value)` Creates an instance with the given default value.	78

Method Summary		Page
`Object`	**get**`()` Returns the wrapped object.	79
`void`	**set**`(Object object)` Sets a new wrapped object.	79

Methods inherited from class ej.microui.Model
addListener, addListeners, changed, changed, changed, getAllListeners, hasListeners, removeAllListeners, removeListener

Constructor Detail

ObjectHolder

```
public ObjectHolder()
```

Creates an instance with `null` as default value.

ObjectHolder

```
public ObjectHolder(Object value)
```

Creates an instance with the given default value.

Since:
1.3.2

Method Detail

get

```
public Object get()
```

Returns the wrapped object.

Returns:
the `Object` wrapped object

---

set

```
public void set(Object object)
```

Sets a new wrapped object. If the new wrapped object is different from the existing one, listeners are notified.

Parameters:
`object` - the wrapped object

Class Pump

ej.microui

```
java.lang.Object
  └
    ej.microui.Pump
```

All Implemented Interfaces:
Runnable

Direct Known Subclasses:
FIFOPump

```
abstract public class Pump
extends Object
implements Runnable
```

A `Pump` holds a `Thread` in order to execute actions. The default implementation of a pump is a loop that repeatedly gets a new action using the abstract method read() and processes it via the method execute(int).

The subclass is responsible for implementing the methods `read()` and `execute(int)`. If an error occurs in these methods the exception is caught by the `Pump` and crash(Throwable) is called.

The `Pump`'s thread priority can be changed using the method setPriority(int), by default, it is `5`.

A `Pump` can be added to a system pool in order to start the `Pump`'s thread at the same time as MicroUI framework (and to stop it too).

Field Summary		Page
static int	**DEFAULT_PRIORITY** The default `Pump`'s thread priority value.	81

Constructor Summary	Page
Pump() Creates a new pump.	81

Method Summary		Page
void	**addToSystemPool**() Adds the pump to the system pool.	81
protected void	**crash**(Throwable e) Called when an error occurred during run().	83
abstract void	**execute**(int data) Process the data previously returned by read().	83
String	**getName**() Names the pump thread The subclasses have to override this method to specify a convenient name	83
int	**getPriority**() Returns the `Pump` thread's priority.	82
boolean	**isStopping**() Indicates if the Pump's thread must stop as soon as possible.	82
abstract int	**read**() Returns the next data to process.	83

void	**removeFromSystemPool**() Removes the pump from the system pool.	*81*
void	**run**() The Pump's Runnable run method.	*83*
void	**setPriority**(int priority) Changes the Pump thread's priority.	*82*
void	**start**() Starts the Pump's thread.	*82*
void	**stop**() Indicates the Pump's thread must stop as soon as possible.	*82*

Field Detail

DEFAULT_PRIORITY

```
public static final int DEFAULT_PRIORITY
```

The default Pump's thread priority value.

The value 5 is assigned to DEFAULT_PRIORITY.

Constructor Detail

Pump

```
public Pump()
```

Creates a new pump.

Method Detail

addToSystemPool

```
public void addToSystemPool()
```

Adds the pump to the system pool. This pump will be started at the same time as MicroUI framework. If MicroUI framework is already running, the pump's thread is immediately started.

See Also:
MicroUI

---

removeFromSystemPool

```
public void removeFromSystemPool()
```

Removes the pump from the system pool. When MicroUI framework stops, this pump will not be stopped automatically. The pump's holder is responsible to stop the pump by calling the method stop().

---

setPriority

```
public void setPriority(int priority)
```

Changes the `Pump` thread's priority.

Parameters:
`priority` - the new priority.
Throws:
`IllegalArgumentException` - If the priority is not in the range `Thread.MIN_PRIORITY` to `Thread.MAX_PRIORITY`.

---

getPriority

```
public int getPriority()
```

Returns the `Pump` thread's priority.

Returns:
the `Pump` thread's priority.

---

start

```
public void start()
```

Starts the `Pump`'s thread. If the thread is already running, this method has no effect. This method must be called by the MicroUI framework implementation at MicroUI startup if the pump has been added to the `Pump` system list.

Throws:
`IllegalArgumentException` - if the Pump is stopping.
See Also:
addToSystemPool()

---

stop

```
public void stop()
```

Indicates the `Pump`'s thread must stop as soon as possible. If the thread is not running, this method has no effect. This method must be called by the MicroUI framework implementation at MicroUI shut down if the pump has been added to the `Pump` system list.

See Also:
addToSystemPool()

---

isStopping

```
public boolean isStopping()
```

Indicates if the Pump's thread must stop as soon as possible. This may be used by implementors to check the state of the Pump in their read() or execute(int) implementations.

Returns:
true if stop() has been called previously, false otherwise.

---

read

```
public abstract int read()
```

Returns the next data to process. This method should block until data is available.

Returns:
data to be processed

---

execute

```
public abstract void execute(int data)
```

Process the data previously returned by read().

Parameters:
`data` - the data

---

run

```
public void run()
```

The `Pump`'s `Runnable run` method. Application must not call this method directly and use `start()` method instead.

Override this method if you want to do something other than loop repeatedly calling read() and execute(int).

Specified by:
`run` in interface `Runnable`

---

crash

```
protected void crash(Throwable e)
```

Called when an error occurred during run().
The default behavior is to stop the pump and to send an error log.

---

getName

```
public String getName()
```

Names the pump thread The subclasses have to override this method to specify a convenient name

Package ej.microui.io

Class Summary		Page
AlphaNumericDisplay	An `AlphaNumericDisplay` object represents an implementation screen, and there is a display for each implementation screen.	85
AlphaNumericDisplayFont	Represents a font that can be displayed on an `AlphaNumericDisplay`.	103
AudioOut	The `AudioOut` class allows sounds to be played.	106
Buttons	A Buttons event generator is usually associated to a group of physical buttons and allow to generate events relating to them.	111
ComponentView	`ComponentView` is an abstract class that represents a visible area.	119
CompositeView	`CompositeView` is a container class of `ComponentView`s.	124
Display	A `Display` object represents a pixelated screen in the platform, and there is a display for each such pixelated screen.	129
Displayable	`Displayable` is an abstract class which defines the very objects that can be shown on a `Display`.	137
DisplayFont	A `DisplayFont` defines how text is rendered on a Display.	140
ExplicitFlush	An `ExplicitFlush` is a `GraphicsContext` where flushing data to the screen must be done explicitly by the application.	144
FlyingImage	The `FlyingImage` class defines an image to be displayed at the top level in the rendering depth of a display.	145
Font	A `Font` defines how text is rendered on a screen.	148
GraphicsContext	The `GraphicsContext` class offers basic drawing facilities, to render lines, rectangles, polygons, arcs and text.	171
Image	An `Image` object holds graphical display data.	197
Keyboard	A Keyboard event generator allows key combinations to generate a key code.	238
Keypad	Keypad is a Keyboard that defines an event generator for 12-key keypads.	242
Leds	This class is used to manage all LEDs available on the platform.	248
Pointer	A `Pointer` event generator represents a pointing device that is usually associated to a group of physical buttons.	251
PointerButtons	**Deprecated.**	259
Screen	A `Screen` represents an hardware screen with its characteristics: width, height, number of supported colors, backlight etc.	261
States	A States event generator is usually associated to a group of physical devices holding a position (switch, rotary wheel encoder, ...)	265
View	`View` is an `abstract` class which extends `ComponentView`.	269
Viewable	`Viewable` is a `Displayable` subclass that contains a hierarchy of `ComponentView` for its rendering.	272

Class AlphaNumericDisplay

ej.microui.io

```
java.lang.Object
  └
    ej.microui.io.Screen
      └
        ej.microui.io.AlphaNumericDisplay
```

```
public class AlphaNumericDisplay
extends Screen
```

An `AlphaNumericDisplay` object represents an implementation screen, and there is a display for each implementation screen. Available alpha numeric displays can be retrieved with the method getAllDisplays(). A default alpha numeric display is defined in most MicroUI implementations and can be fetched with the method getDefaultDisplay().

An alpha numeric display is able to render a alpha numeric character: text can be drawn using drawChar(), drawChars(), drawString() or drawSubstring().

At startup, the cursor is hidden.

By default, all drawing operations (draw and clear) implicitly flush the content to the screen. Applications can disable the implicit flush and manually call flush().

Field Summary		Page
static int	**ANCHOR_CENTER** The "center" anchor.	90
static int	**ANCHOR_LEFT** The "left" anchor.	90
static int	**ANCHOR_RIGHT** The "right" anchor.	90
static int	**SCROLL_DOWN** The "down" scrolling direction.	88
static int	**SCROLL_DOWN_LEFT** The "down left" scrolling direction and anchor.	89
static int	**SCROLL_DOWN_RIGHT** The "down right" scrolling direction and anchor.	90
static int	**SCROLL_LEFT** The "left" scrolling direction and anchor.	88
static int	**SCROLL_LEFT_DOWN** The "left down" scrolling direction and anchor.	89
static int	**SCROLL_LEFT_UP** The "left up" scrolling direction and anchor.	88
static int	**SCROLL_RIGHT** The "right" scrolling direction and anchor.	88
static int	**SCROLL_RIGHT_DOWN** The "right down" scrolling direction and anchor.	89
static int	**SCROLL_RIGHT_UP** The "right up" scrolling direction and anchor.	89
static int	**SCROLL_UP** The "up" scrolling direction.	88

static int	**SCROLL_UP_LEFT** The "up left" scrolling direction and anchor.	*89*
static int	**SCROLL_UP_RIGHT** The "up right" scrolling direction and anchor.	*89*

Method Summary		*Page*
void	**blinkCursor**() Shows and blinks the cursor.	*94*
void	**clear**() Clear the screen.	*95*
void	**drawChar**(char character, int x, int y, int anchor) Draws a character using the current color.	*99*
void	**drawChars**(char[] data, int offset, int length, int x, int y, int anchor) Draws some characters using the current color.	*98*
void	**drawInt**(int val, int x, int y, int anchor) Draws a string that represents an integer using the current color.	*99*
void	**drawInt**(int val, int x, int y, int nbDigits, int anchor) Draws a string that represents an integer using the current color.	*99*
void	**drawString**(String str, int x, int y, int anchor) Draws the string using the current color.	*97*
void	**drawSubstring**(String str, int offset, int len, int x, int y, int anchor) Draws the string from `offset` to `offset+length` using the current color.	*98*
void	**flush**() Forces any buffered characters to be sent to the display.	*102*
static AlphaNumericDisplay[]	**getAllDisplays**() Returns all available alpha numeric displays.	*90*
AlphaNumericDisplayFont[]	**getAllFonts**() Returns an array containing all available fonts in the `AlphaNumericDisplay`.	*101*
int	**getBacklight**() Returns the backlight of the screen.	*93*
int	**getBacklightColor**() Returns the current backlight color.	*93*
int	**getColor**(int rgbColor) Returns the current color: a 32-bits value interpreted as: `0x00RRGGBB`, that is, the eight less significant bits give the blue color, the next eight bits the green value and the next eight bits the red color.	*100*
int	**getContrast**() Returns the contrast of the screen.	*92*
int	**getCursorPositionX**() Returns the x coordinate of the cursor position.	*94*
int	**getCursorPositionY**() Returns the y coordinate of the cursor position.	*94*
static AlphaNumericDisplay	**getDefaultDisplay**() Returns the default alpha numeric display of the system.	*90*
AlphaNumericDisplayFont	**getDefaultFont**() Returns the default font of the `AlphaNumericDisplay`.	*101*
AlphaNumericDisplayFont	**getFont**(int identifier, int style) Returns an `AlphaNumericDisplayFont` matching the requested characteristics as close as possible.	*101*

int	**getHeight**() Each alpha numeric display has a fixed number of lines.	*91*
int	**getMaximumNumberOfScrolls**() An alpha numeric display has got a limited number of scrolls.	*94*
int	**getNumberOfColors**() Returns the number of available colors for this screen.	*91*
int	**getWidth**() Each alpha numeric display has a fixed number of columns.	*91*
void	**handleEvent**(int event) Inject a MicroUI event to be handled by the event generator associated with this AlphaNumericDisplay (if any).	*100*
boolean	**hasBacklight**() Tells whether the screen has a backlight.	*92*
void	**hideCursor**() Hides the cursor.	*93*
boolean	**isColor**() Tells whether the screen offers color.	*91*
int	**newScroll**(int x0, int y0, int x1, int y1, int direction, int waitInMillisecond) A scroll area is an area where the characters scroll continuously across the display.	*95*
void	**removeScroll**(int id) Remove a scroll area referenced by its id.	*95*
void	**setBacklight**(int backlight) Sets the backlight level of the screen.	*92*
void	**setBacklightColor**(int rgbColor) Sets the current backlight color, if it is allowed by implementation.	*93*
void	**setColor**(int rgbColor) Sets the screen's current color.	*100*
void	**setContrast**(int contrast) Sets the contrast of the screen.	*92*
void	**setCursorPosition**(int x, int y) Sets the cursor position.	*94*
void	**setEventHandler**(Listener handler) Sets the event handler.	*100*
boolean	**setFont**(AlphaNumericDisplayFont font) Set font as the current font.	*101*
void	**setImplicitFlush**(boolean state) Enable/disable implicit flush after next drawing operations.	*102*
void	**setPriority**(int priority) Sets the priority of the scroll processing	*95*
void	**setScrollChars**(int id, char[] data, int offset, int length) Sets the data to be displayed for the scroll area referenced by its id.	*96*
void	**setScrollString**(int id, String str) Sets the text to be displayed for the scroll area referenced by its id.	*95*
void	**setScrollSubString**(int id, String str, int offset, int length) Sets the data to be displayed for the scroll area referenced by its id.	*96*
void	**setScrollWait**(int id, int waitInMillisecond) Sets the speed for the scroll area referenced by its id.	*97*
void	**showCursor**() Shows the cursor.	*93*

void	**startScroll**(int id) Starts the scroll area referenced by its id.	*97*
void	**stopScroll**(int id) Disables the scroll area referenced by its id.	*97*
void	**switchBacklight**(boolean on) Switches on or off the backlight of the screen.	*92*

Methods inherited from class ej.microui.io.Screen
getEventHandler

Field Detail

SCROLL_UP

```
public static final int SCROLL_UP
```

The "up" scrolling direction.

The value `0x01` is assigned to `SCROLL_UP`.

SCROLL_DOWN

```
public static final int SCROLL_DOWN
```

The "down" scrolling direction.

The value `0x02` is assigned to `SCROLL_DOWN`.

SCROLL_LEFT

```
public static final int SCROLL_LEFT
```

The "left" scrolling direction and anchor.

The value `0x03` is assigned to `SCROLL_LEFT`.

SCROLL_RIGHT

```
public static final int SCROLL_RIGHT
```

The "right" scrolling direction and anchor.

The value `0x04` is assigned to `SCROLL_RIGHT`.

SCROLL_LEFT_UP

```
public static final int SCROLL_LEFT_UP
```

The "left up" scrolling direction and anchor.

The value `0x05` is assigned to `SCROLL_LEFT_UP`.

SCROLL_LEFT_DOWN

`public static final int` **`SCROLL_LEFT_DOWN`**

The "left down" scrolling direction and anchor.

The value `0x06` is assigned to `SCROLL_LEFT_DOWN`.

SCROLL_RIGHT_UP

`public static final int` **`SCROLL_RIGHT_UP`**

The "right up" scrolling direction and anchor.

The value `0x07` is assigned to `SCROLL_RIGHT_UP`.

SCROLL_RIGHT_DOWN

`public static final int` **`SCROLL_RIGHT_DOWN`**

The "right down" scrolling direction and anchor.

The value `0x08` is assigned to `SCROLL_RIGHT_DOWN`.

SCROLL_UP_LEFT

`public static final int` **`SCROLL_UP_LEFT`**

The "up left" scrolling direction and anchor.

The value `0x09` is assigned to `SCROLL_UP_LEFT`.

SCROLL_UP_RIGHT

`public static final int` **`SCROLL_UP_RIGHT`**

The "up right" scrolling direction and anchor.

The value `0x0a` is assigned to `SCROLL_UP_RIGHT`.

SCROLL_DOWN_LEFT

`public static final int` **`SCROLL_DOWN_LEFT`**

The "down left" scrolling direction and anchor.

The value 0x0b is assigned to SCROLL_DOWN_LEFT.

SCROLL_DOWN_RIGHT

```
public static final int SCROLL_DOWN_RIGHT
```

The "down right" scrolling direction and anchor.

The value 0x0c is assigned to SCROLL_DOWN_RIGHT.

ANCHOR_LEFT

```
public static final int ANCHOR_LEFT
```

The "left" anchor.

The value 0x00 is assigned to ANCHOR_LEFT.

ANCHOR_RIGHT

```
public static final int ANCHOR_RIGHT
```

The "right" anchor.

The value 0x01 is assigned to ANCHOR_RIGHT.

ANCHOR_CENTER

```
public static final int ANCHOR_CENTER
```

The "center" anchor.

The value 0x02 is assigned to ANCHOR_CENTER.

Method Detail

getDefaultDisplay

```
public static AlphaNumericDisplay getDefaultDisplay()
```

Returns the default alpha numeric display of the system. It can be null if no alpha numeric display is defined.

Returns:
the default alpha numeric display

---

getAllDisplays

```
public static AlphaNumericDisplay[] getAllDisplays()
```

Returns all available alpha numeric displays. If no alpha numeric display is defined it returns a empty array.

Returns:
all available displays as an array of AlphaNumericDisplay as `AlphaNumericDisplay[]`.

---

getWidth

```
public int getWidth()
```

Each alpha numeric display has a fixed number of columns.

Overrides:
getWidth in class Screen
Returns:
the width: the number of characters per line. It is at least 1.

---

getHeight

```
public int getHeight()
```

Each alpha numeric display has a fixed number of lines.

Overrides:
getHeight in class Screen
Returns:
the height: the number of characters per column. It is at least 1.

---

isColor

```
public boolean isColor()
```

Tells whether the screen offers color.

Overrides:
isColor in class Screen
Returns:
`true` if screen has color

---

getNumberOfColors

```
public int getNumberOfColors()
```

Returns the number of available colors for this screen. The number of colors is 1 for monochrome screens.

Overrides:
getNumberOfColors in class Screen
Returns:
the number of colors.

---

setContrast

```
public void setContrast(int contrast)
```

Sets the contrast of the screen. `contrast` value range is 0-100.

Overrides:
setContrast in class Screen
Parameters:
`contrast` - the new value of the contrast.

---

getContrast

```
public int getContrast()
```

Returns the contrast of the screen.

Overrides:
getContrast in class Screen
Returns:
the current contrast of the screen (range 0-100)

---

hasBacklight

```
public boolean hasBacklight()
```

Tells whether the screen has a backlight.

Overrides:
hasBacklight in class Screen
Returns:
`true` if screen has a backlight

---

switchBacklight

```
public void switchBacklight(boolean on)
```

Switches on or off the backlight of the screen.

Overrides:
switchBacklight in class Screen
Parameters:
`on` - if `true`, switch on the backlight; otherwise switch off the backlight

---

setBacklight

```
public void setBacklight(int backlight)
```

Sets the backlight level of the screen. `backlight` value range is 0-100

Overrides:
setBacklight in class Screen

Parameters:
`backlight` - the new value of the backlight

---

getBacklight

```
public int getBacklight()
```

Returns the backlight of the screen.

Overrides:
getBacklight in class Screen
Returns:
the backlight of the screen (value range is 0-100)

---

setBacklightColor

```
public void setBacklightColor(int rgbColor)
```

Sets the current backlight color, if it is allowed by implementation.

Overrides:
setBacklightColor in class Screen
Parameters:
`rgbColor` - the color to set

---

getBacklightColor

```
public int getBacklightColor()
```

Returns the current backlight color. Returned value is interpreted as a 24-bit RGB color, where the eight less significant bits matches the blue component, the next eight bits matches the green component and the next eight bits matches the red component. By default, this method returns 0 and sub-classes should overwrite this default behavior.

Overrides:
getBacklightColor in class Screen
Returns:
the color of the backlight

---

showCursor

```
public void showCursor()
```

Shows the cursor.

---

hideCursor

```
public void hideCursor()
```

Hides the cursor.

---

blinkCursor

```
public void blinkCursor()
```

Shows and blinks the cursor.

---

setCursorPosition

```
public void setCursorPosition(int x,
                              int y)
```

Sets the cursor position. Caution: this position can be modified by a drawing on the display (scrolling included)

Parameters:
`x` - the horizontal position
`y` - the vertical position
Throws:
`IllegalArgumentException` - if the point (x, y) is outside the display.

---

getCursorPositionX

```
public int getCursorPositionX()
```

Returns the x coordinate of the cursor position.

Returns:
x coordinate

---

getCursorPositionY

```
public int getCursorPositionY()
```

Returns the y coordinate of the cursor position.

Returns:
y coordinate

---

getMaximumNumberOfScrolls

```
public int getMaximumNumberOfScrolls()
```

An alpha numeric display has got a limited number of scrolls. When the limit is reached the application cannot create a new scroll. It has to remove an old scroll calling removeScroll(int) method before.
This method returns the maximum number of srolls this display can open simultaneously.

---

newScroll

```
public int newScroll(int x0,
                     int y0,
                     int x1,
                     int y1,
                     int direction,
                     int waitInMillisecond)
```

A scroll area is an area where the characters scroll continuously across the display. The direction and speed of scrolling are determined when setting the zone. The speed can be redefined by the setScrollWait(int, int) method.

Parameters:

`x0` - x coordinate top left corner from the scrolling area
`y0` - y coordinate top left corner from the scrolling area
`x1` - x coordinate bottom right corner from the scrolling area
`y1` - y coordinate bottom right corner from the scrolling area
`direction` - direction (see scrolling direction constants in this class)
`waitInMillisecond` - refresh period of the scroll

Returns:

the scroll area unique id as an integer. `-1` indicates creation failure.

---

removeScroll

```
public void removeScroll(int id)
```

Remove a scroll area referenced by its id. If the given id does not represent a valid scroll area, nothing occurs.

Parameters:

`id` - to be removed

Throws:

`IllegalArgumentException` - if `id` is an invalid scroll id.

---

setPriority

```
public void setPriority(int priority)
```

Sets the priority of the scroll processing

Parameters:

`priority` - the new priority of scroll processing

---

clear

```
public void clear()
```

Clear the screen. Note that clearing the screen doesn't stop the scrolls.

---

setScrollString

```
public void setScrollString(int id,
                            String str)
```

Sets the text to be displayed for the scroll area referenced by its id.

Parameters:
`id` - the scroll area identifier
`str` - the string to be scrolled
Throws:
`NullPointerException` - if str is null
`MicroUIException` - if the limit of MicroUI's implementation capacity is reached. In this case, reduce string size or remove another scroll.
`IllegalArgumentException` - if `id` is an invalid scroll id.

---

setScrollSubString

```
public void setScrollSubString(int id,
                               String str,
                               int offset,
                               int length)
```

Sets the data to be displayed for the scroll area referenced by its id. The data starts from `offset` to `offset+length`.

Parameters:
`id` - the scroll area identifier
`str` - the string to be scrolled
`offset` - index of the first character in the string to draw
`length` - number of characters to draw from `offset`
Throws:
`StringIndexOutOfBoundsException` - if `offset` and `length` do not specify a valid range within `str`
`NullPointerException` - if `str` is `null`
`MicroUIException` - if the limit of MicroUI's implementation capacity is reached. In this case, reduce string size or remove another scroll.
`IllegalArgumentException` - if `id` is an invalid scroll id.

---

setScrollChars

```
public void setScrollChars(int id,
                           char[] data,
                           int offset,
                           int length)
```

Sets the data to be displayed for the scroll area referenced by its id.

Parameters:
`id` - the scroll area identifier
`data` - the array of characters to be drawn
`offset` - offset of the first character to draw in `data`
`length` - the number of characters to draw from `offset`
Throws:
`ArrayIndexOutOfBoundsException` - if `offset` and `length` do not specify a valid range within `data`
`NullPointerException` - if `data` is `null`
`MicroUIException` - if the limit of MicroUI's implementation capacity is reached. In this case, reduce string size or remove another scroll.
`IllegalArgumentException` - if `id` is an invalid scroll id.

---

setScrollWait

```
public void setScrollWait(int id,
                          int waitInMillisecond)
```

Sets the speed for the scroll area referenced by its id.

Parameters:
`id` - the scroll area identifier
`waitInMillisecond` - the scroll wait between two scroll states
Throws:
`IllegalArgumentException` - if `id` is an invalid scroll id.

---

stopScroll

```
public void stopScroll(int id)
```

Disables the scroll area referenced by its id.

Parameters:
`id` - the scroll identifier
Throws:
`IllegalArgumentException` - if `id` is an invalid scroll id.

---

startScroll

```
public void startScroll(int id)
```

Starts the scroll area referenced by its id. If no data has been set on the scroll area, the scroll will use the screen data on the scroll area. Each time this scroll is updated, the cursor's position is reset to (0,0)

Parameters:
`id` - the scroll identifier
Throws:
`IllegalArgumentException` - if `id` is an invalid scroll id.
`MicroUIException` - if the limit of MicroUI's implementation capacity is reached. In this case, reduce string size or remove another scroll.

---

drawString

```
public final void drawString(String str,
                             int x,
                             int y,
                             int anchor)
```

Draws the string using the current color.
The text anchor's point is at position `(x,y)`. Position constants may be given to specify the precise location of the text around the anchor point. Cursor position is moved to just after the end of the text drawn.

Parameters:
`str` - the string to draw
`x` - the x coordinate of the anchor point
`y` - the y coordinate of the anchor point
`anchor` - specifies how the text is positioned around the anchor point

Throws:

`NullPointerException` - if str is null
`IllegalArgumentException` - if anchor is not a valid value

---

drawSubstring

```
public void drawSubstring(String str,
                          int offset,
                          int len,
                          int x,
                          int y,
                          int anchor)
```

Draws the string from `offset` to `offset+length` using the current color.
The text anchor point is at position `(x,y)`. Position constants may be given to specify the precise location of the text around the anchor point. Cursor position is moved to just after the end of the text drawn.

Parameters:

`str` - the string to draw
`offset` - index of the first character in the string to draw
`len` - number of characters to draw from `offset`
`x` - the x coordinate of the anchor point
`y` - the y coordinate of the anchor point
`anchor` - specifies how the text is positioned around the anchor point

Throws:

`StringIndexOutOfBoundsException` - if `offset` and `length` do not specify a valid range within `str`
`IllegalArgumentException` - if anchor is not a valid value
`NullPointerException` - if `str` is `null`

---

drawChars

```
public void drawChars(char[] data,
                      int offset,
                      int length,
                      int x,
                      int y,
                      int anchor)
```

Draws some characters using the current color.
The text anchor point is at position `(x,y)`. Position constants may be given to specify the precise location of the text around the anchor point. Cursor position is moved to just after the end of the text drawn.

Parameters:

`data` - the array of characters to draw
`offset` - offset of the first character to draw in `data`
`length` - the number of characters to draw from `offset`
`x` - the x coordinate of the anchor point
`y` - the y coordinate of the anchor point
`anchor` - specifies how the text is positioned around the anchor point

Throws:

`ArrayIndexOutOfBoundsException` - if `offset` and `length` do not specify a valid range within `data`
`IllegalArgumentException` - if anchor is not a valid value
`NullPointerException` - if `data` is `null`

---

drawChar

```
public void drawChar(char character,
                     int x,
                     int y,
                     int anchor)
```

Draws a character using the current color.
The text anchor point is at position `(x,y)`. Position constants may be given to specify the precise location of the character around the anchor point. Cursor position is moved to just after the end of the character drawn.

Parameters:

`character` - the character to draw
`x` - the x coordinate of the anchor point
`y` - the y coordinate of the anchor point
`anchor` - specifies how the character is positioned around the anchor point

Throws:

`IllegalArgumentException` - if anchor is not a valid value

---

drawInt

```
public void drawInt(int val,
                    int x,
                    int y,
                    int anchor)
```

Draws a string that represents an integer using the current color.
The integer anchor point is at position `(x,y)`. Position constants may be given to specify the precise location of the character around the anchor point. Cursor position is moved to just after the end of the int drawn.

Parameters:

`val` - the integer to draw
`x` - the x coordinate of the anchor point
`y` - the y coordinate of the anchor point
`anchor` - specifies how the text is positioned around the anchor point

Throws:

`IllegalArgumentException` - if anchor is not a valid value

---

drawInt

```
public void drawInt(int val,
                    int x,
                    int y,
                    int nbDigits,
                    int anchor)
```

Draws a string that represents an integer using the current color. `nbDigits` specifies the number of characters to render the integer. If the integer needs less characters than `nbDigits`, the string will be drawn right-justified, with leading spaces. If the integer needs more characters than `nbDigits`, this method will draw only the integer's least significant part.
The integer anchor point is at position `(x,y)`. Position constants may be given to specify the precise location of the character around the anchor point. Cursor position is moved to just after the end of the int drawn.

Parameters:

`val` - the integer to draw
`x` - the x coordinate of the anchor point
`y` - the y coordinate of the anchor point
`nbDigits` - the number of characters to render the integer

`anchor` - specifies how the text is positioned around the anchor point

Throws:

`IllegalArgumentException` - if anchor is not a valid value

---

setEventHandler

`public void` **`setEventHandler`**`(Listener handler)`

Sets the event handler. It replaces the old one if any.

Note that by default an AlphaNumericDisplay has no event handler.

Parameters:

`handler` - the new event handler or null.

---

handleEvent

`public void` **`handleEvent`**`(int event)`

Inject a `MicroUI` event to be handled by the event generator associated with this AlphaNumericDisplay (if any).

Parameters:

`event` - an event in the MicroUI format

---

setColor

`public void` **`setColor`**`(int rgbColor)`

Sets the screen's current color.
Given value `rgbColor` is interpreted as a 24-bit RGB color, where the eight less significant bits match the blue component, the next eight more significant bits match the green component and the next eight more significant bits match the red component.
Even if some screens have several colors, those colors might be global, that is to say, changing the color for one character will set the color of all characters.

Parameters:

`rgbColor` - the color to set

---

getColor

`public int` **`getColor`**`(int rgbColor)`

Returns the current color: a 32-bits value interpreted as: `0x00RRGGBB`, that is, the eight less significant bits give the blue color, the next eight bits the green value and the next eight bits the red color.

Returns:

current color

---

getAllFonts

`public` AlphaNumericDisplayFont`[]` **`getAllFonts`**`()`

Returns an array containing all available fonts in the `AlphaNumericDisplay`.

Returns:
an array of `AlphaNumericDisplayFont[]`

---

getDefaultFont

`public` AlphaNumericDisplayFont **`getDefaultFont`**`()`

Returns the default font of the `AlphaNumericDisplay`.
This method may return `null` if no font is declared in the `AlphaNumericDisplay`.

Returns:
the `AlphaNumericDisplayFont` default font

---

getFont

`public` AlphaNumericDisplayFont **`getFont`**`(int identifier,`
`int style)`

Returns an `AlphaNumericDisplayFont` matching the requested characteristics as close as possible.

Font is requested with an identifier and style. If no available font matches the request, the system will attempt to provide the most appropriate font.

This method returns a valid font object or null if no font is available.

Parameters:
`identifier` - the identifier of the font, as defined in Font.
`style` - combination of style constants, as defined in Font
Returns:
an `AlphaNumericDisplayFont` object

---

setFont

`public boolean` **`setFont`**`(`AlphaNumericDisplayFont `font)`

Set `font` as the current font. All next drawing and scrolling actions will use the new font. `font` is an `AlphaNumericDisplayFont` got from getAllFonts(), getDefaultFont() or getFont(int, int).

Parameters:
`font` - the new current font*
Returns:
`true` if `font` has been set as new current font, `false` otherwise
Throws:
`NullPointerException` - if `font` is null
`IllegalArgumentException` - if `font` is not a platform font.

---

flush

```
public void flush()
```

Forces any buffered characters to be sent to the display.

---

setImplicitFlush

```
public void setImplicitFlush(boolean state)
```

Enable/disable implicit flush after next drawing operations.

Parameters:

`state` - if true enable implicit flush else disable it.

Class AlphaNumericDisplayFont

ej.microui.io

```
java.lang.Object
  └
    ej.microui.io.Font
      └
        ej.microui.io.AlphaNumericDisplayFont
```

```
abstract public class AlphaNumericDisplayFont
extends Font
```

Represents a font that can be displayed on an `AlphaNumericDisplay`.

`AlphaNumericDisplayFonts` are never created by applications, but are rather retrieved from the implementation environment.

See Also:
AlphaNumericDisplay

Fields inherited from class ej.microui.io.Font
ARABIC, ARMENIAN, BALINESE, BAMUM, BATAK, BENGALI, BOPOMOFO, BRAILLE, BUGINESE, BUHID, CANADIAN_ABORIGINAL, CHAM, CHEROKEE, COMMON, COPTIC, CUNEIFORM, CYPRIOT, CYRILLIC, DESERET, DEVANAGARI, ETHIOPIC, GEORGIAN, GLAGOLITIC, GOTHIC, GREEK, GUJARATI, GURMUKHI, HAN, HANGUL, HANUNOO, HEBREW, HIRAGANA, INHERITED, JAVANESE, KANNADA, KATAKANA, KAYAH_LI, KHAROSHTHI, KHMER, LAO, LATIN, LEPCHA, LIMBU, LISU, MALAYALAM, MANDAIC, MEETEI_MAYEK, MONGOLIAN, MYANMAR, NEW_TAI_LUE, NKO, OGHAM, OL_CHIKI, ORIYA, OSMANYA, PHAGS_PA, PHOENICIAN, REJANG, RUNIC, SAMARITAN, SAURASHTRA, SHAVIAN, SINHALA, STYLE_BOLD, STYLE_ITALIC, STYLE_PLAIN, STYLE_UNDERLINED, SUNDANESE, SYLOTI_NAGRI, SYRIAC, TAGALOG, TAGBANWA, TAI_LE, TAI_THAM, TAI_VIET, TAMIL, TELUGU, THAANA, THAI, TIBETAN, TIFINAGH, UGARITIC, VAI, YI

Method Summary		Page
char	**convertChar**(char c) **Deprecated.** *conversion is implicit*	105
char[]	**convertChars**(char[] chars, int offset, int length) **Deprecated.** *conversion is implicit*	105
char[]	**convertString**(String str) **Deprecated.** *conversion is implicit*	104
char[]	**convertSubstring**(String str, int offset, int length) **Deprecated.** *conversion is implicit*	104
AlphaNumericDisplay	**getAlphaNumericDisplay**() Gets the `AlphaNumericDisplay` where this font is renderable	104
boolean	**isMonospaced**() An `AlphaNumericDisplayFont` font is always monospaced.	104

Methods inherited from class ej.microui.io.Font
charsWidth, charWidth, getDescriptor, getIdentifiers, getStyle, isBold, isIdentifierSupported, isItalic, isPlain, isUnderlined, stringWidth, substringWidth, supportIdentifiers

Method Detail

getAlphaNumericDisplay

```
public AlphaNumericDisplay getAlphaNumericDisplay()
```

Gets the `AlphaNumericDisplay` where this font is renderable

Returns:
the `AlphaNumericDisplayFont`'s display

---

isMonospaced

```
public boolean isMonospaced()
```

An `AlphaNumericDisplayFont` font is always monospaced. This method returns always true

Overrides:
isMonospaced in class Font
Returns:
true

---

convertString

```
public char[] convertString(String str)
```

Deprecated. *conversion is implicit*

Converts all `str`'s characters according to same specification as `convertChar()` method. The returned array is a new array whose length is `str`'s length.

Parameters:
`str` - the string to convert
Returns:
a new char array
Throws:
`NullPointerException` - if `str` is `null`

---

convertSubstring

```
public char[] convertSubstring(String str,
                               int offset,
                               int length)
```

Deprecated. *conversion is implicit*

Converts all `str` chars from `offset` to `offset+length` according to same specification as `convertChar()` method. The returned array is a new array whose length is `length`.

Parameters:
`str` - the string to convert
`offset` - index of the first character in the string to convert
`length` - number of characters to convert from `offset`

Returns:
a new char array
Throws:
`ArrayIndexOutOfBoundsException` - if `offset` and `length` do not specify a valid range within `str`
`NullPointerException` - if `str` is `null`

---

convertChars

```
public char[] convertChars(char[] chars,
                           int offset,
                           int length)
```

Deprecated. *conversion is implicit*

Converts all chars from `offset` to `offset+length` according to same specification as `convertChar()` method. The returned array is a new array whose length is `length`.

Parameters:
`chars` - the array of characters to convert
`offset` - offset of the first character to convert in `chars`
`length` - the number of characters to convert from `offset`
Returns:
a new char array
Throws:
`ArrayIndexOutOfBoundsException` - if `offset` and `length` do not specify a valid range within `chars`
`NullPointerException` - if `chars` is `null`

---

convertChar

```
public char convertChar(char c)
```

Deprecated. *conversion is implicit*

Returns the index of the character in the font, by default the character itself. The input character is assumed to follow the ISO/IEC 10646 specification. The result character is the character as close as possible to ISO/IEC 10646 specification rendering.
Example:
The character 0x007B (defined in ISO/IEC 10646 specification as {) is converted into 0x00FD for a DIP204 alpha numeric display.

Parameters:
`c` - the character to convert
Returns:
the character index or a font specific character if the character is unknown

Class AudioOut

ej.microui.io

```
java.lang.Object
  ↳
    ej.microui.io.AudioOut
```

```
final public class AudioOut
extends Object
```

The `AudioOut` class allows sounds to be played.

A tone is an integer between `0` and `127`, which can be retrieved from its frequency with the following formula:

```
SEMITONE_CONST = 1/(ln(2^(1/12))) = 17.31234049066755
 note = ln(freq/8.176)*SEMITONE_CONST
 The musical note A = MIDI note 69 (0x45) = 440 Hz.
```

The available non-silent tone values are in range 0 = tone = 127. Negative tone values are silent tone.

Octave	C	C#	D	D#	E	F	F#	G	G#	A	A#	B
-1	0	1	2	3	4	5	6	7	8	9	10	11
0	12	13	14	15	16	17	18	19	20	21	22	23
1	24	25	26	27	28	29	30	31	32	33	34	35
2	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59
4	60	61	62	63	64	65	66	67	68	69	70	71
5	72	73	74	75	76	77	78	79	80	81	82	83
6	84	85	86	87	88	89	90	91	92	93	94	95
7	96	97	98	99	100	101	102	103	104	105	106	107
8	108	109	110	111	112	113	114	115	116	117	118	119

Field Summary		Page
static byte	**COMMAND_AUTOREPEAT** The "auto repeat" command constant.	107
static byte	**COMMAND_BPM** The "BPM" command constant.	107
static byte	**COMMAND_PLAY** The "play" command constant.	107
static byte	**COMMAND_VOLUME** The "volume" command constant.	107
static byte	**MAX_VOLUME** Maximum volume value	107
static byte	**MIN_VOLUME** Mute volume value.	107

Method Summary		Page
static void	**pause**`(boolean pause)` Pause or play the current multi tones sound.	108
static void	**playTone**`(byte[] data)` Renders a multi tones sound and returns immediately.	109

static void	**playTone**(int note, int duration, int volume) Plays a tone, defined by a note and a duration and returns immediately.	*108*
static void	**setListener**(Listener listener) Set a listener on current multi tones sound.	*108*
static void	**setMasterVolume**(int volume) Set the master volume. 0 means mute and 100 means maximum of volume.	*109*
static void	**stopTone**() Tells the system to stop the basic sound rendering as soon as it can if some sound was currently playing.	*108*

Field Detail

MIN_VOLUME

```
public static final byte MIN_VOLUME
```

Mute volume value.

MAX_VOLUME

```
public static final byte MAX_VOLUME
```

Maximum volume value

COMMAND_PLAY

```
public static final byte COMMAND_PLAY
```

The "play" command constant.
The value `0` is assigned to `COMMAND_PLAY`.

COMMAND_BPM

```
public static final byte COMMAND_BPM
```

The "BPM" command constant.
The value `1` is assigned to `COMMAND_BPM`.

COMMAND_AUTOREPEAT

```
public static final byte COMMAND_AUTOREPEAT
```

The "auto repeat" command constant.
The value `2` is assigned to `COMMAND_AUTOREPEAT`.

COMMAND_VOLUME

```
public static final byte COMMAND_VOLUME
```

The "volume" command constant.
The value `3` is assigned to `COMMAND_VOLUME`.

Method Detail

playTone

```
public static void playTone(int note,
                            int duration,
                            int volume)
```

Plays a tone, defined by a note and a duration and returns immediately. The duration is given in milliseconds and the volume is an integer value between `0` and `100` (`0` stands for the minimum volume whereas `100` is the maximum of the master volume set by `setMasterVolume()` method).

Parameters:

`note` - the tone to play
`duration` - the duration of the tone in milliseconds or 0 to play the note indefinitely.
`volume` - audio volume range from `MIN_VOLUME` to `MAX_VOLUME`.

---

stopTone

```
public static void stopTone()
```

Tells the system to stop the basic sound rendering as soon as it can if some sound was currently playing.

---

pause

```
public static void pause(boolean pause)
```

Pause or play the current multi tones sound. If there is no current multi tones sound nothing is done.

This method stops a sound played thanks the method `playTone(byte[])` without restart it.

Parameters:

`pause` - `true` to pause the multi tones sound, `false` to play again.

---

setListener

```
public static void setListener(Listener listener)
```

Set a listener on current multi tones sound. As soon as note is played, a call to `listener.performAction(int)` is performed. The given `integer` is the index of the current played note in the user byte array.

If there is no current multi tones sound nothing is done.
The new listener replaces the old one.

Parameters:

`listener` - the multi tones sound listener or null.

---

setMasterVolume

```
public static void setMasterVolume(int volume)
```

Set the master volume. 0 means mute and 100 means maximum of volume.
At startup, the master volume is 100.

Parameters:

`volume` - the master volume

---

playTone

```
public static void playTone(byte[] data)
```

Renders a multi tones sound and returns immediately. The array of bytes is a list of commands and tones to be rendered by the sound system.

Following lines explains the array format.
Note: u*n*: *n* is the number of bytes to specify a value, in big endian format (MSB,...,LSB).

Header: [`Version, BPM, Resolution, NbBlocks, Blocks, NbCmds, Commands`]

- `Version` ::= u1 // currently only 1 is available
- `BPM` ::= u2 // beat per minute
- `Resolution` ::= u1 // number of duration per beat
- `NbBlocks` ::= u4
- `Blocks` ::= blocks list (see Block)
- `NbCmds` ::= u4
- `Commands` ::= commands list (see Command)

Block: [`BlockID, Length, Duration, Tones`]

- `BlockID` ::= u4
- `Length` ::= u4
- `Duration` ::= u1
- `Tones` ::= u1 // list of tones

Command: [`COMMAND_PLAY | COMMAND_BPM | COMMAND_AUTOREPEAT | COMMAND_VOLUME, data`]

- `COMMAND_PLAY` // followed by BlockID (u4)
- `COMMAND_BPM` // followed by BPM (u2)
- `COMMAND_AUTOREPEAT` // after the last command, restart
- `COMMAND_VOLUME` // followed by Volume (u1 in range [MIN_VOLUME..MAX_VOLUME])

The `BlockID` identifies the block. If two or several blocks have the same id, the block played is platform dependent but no error is thrown.

The `Duration` of the block defines the semantic of the `Tones` bytes. If `Duration` is 0, the `Tones` bytes are organized as a sequence of two 'u1' representing one tone and its duration. If `Duration` is greater than zero, the sequence of `Tones` bytes are just the tones, all with the same `Duration`. The length of a block includes the `Duration` and the `Tones`. The available non-silent tone values are in range 0 = tone = 127. Negative tone values are silent tone.

The `Volume` is of the current percentage of volume set by `setVolume()` method)

The given byte array is not duplicated by the implementation in order to play it: user should not modify it during the playing.

Example 1:

```
byte C = 60, D = 62, E = 64;
byte[] music = new byte[]{

//header
1, //version
0, 60, //beat per minute
2, //resolution (one duration is (60/60)/2 second, i.e. 1/2 second)
0, 0, 0, 1, //number of blocks

//blocks of tones
0, 0, 0, 0, //block ID 0
0, 0, 0,12, //length of the block
1, //duration of all the tones within the block
C, C, C, D, E, D, C, E, D, D, C,

//commands
0, 0, 0, 2, 3, 100, //set volume to its maximum
AudioOut.COMMAND_PLAY, 0, 0, 0, 0 //play block 0
};
```

Example 2:

```
byte C = 60, D = 62, E = 64;
byte[] music = new byte[]{

//header
1, //version
0, 60, //beat per minute
2, //resolution (one duration is (60/60)/2 second, i.e. 1/2 second)
0, 0, 0, 1, //number of blocks

//blocks of tones
0, 0, 0, 0, //block ID 0
0, 0, 0,23, //length of the block
0, //no common duration
C, 1, C, 1, C, 1, D, 1, E, 1, D, 1, C, 1, E, 1, D, 1, D, 1, C, 1,

//commands
0, 0, 0, 2, 3, 100, //set volume to its maximum
AudioOut.COMMAND_PLAY, 0, 0, 0, 0 //play block 0
};
```

Parameters:

`data` - the list of commands and tones

Throws:

`NullPointerException` - if `array` is null

`IllegalArgumentException` - if `array` bytes does not respect the specified format.

Class Buttons

ej.microui.io

```
java.lang.Object
  └
    ej.microui.EventGenerator
      └
        ej.microui.io.Buttons
```

Direct Known Subclasses:
Pointer, PointerButtons

```
public class Buttons
extends EventGenerator
```

A Buttons event generator is usually associated to a group of physical buttons and allow to generate events relating to them.

For a specified subset of buttons it holds the elapsed time since the last event occurrence for that button and supports the optional generation of click and double click events. Note that Buttons pre-configured by the system normally support these extended features for all their buttons. However, it is implementation dependent whether or not the features are enabled by default.

This class defines generic button actions : `PRESSED`, `RELEASED`, `LONG`, `REPEATED`, `CLICKED` and `DOUBLE_CLICKED`.

Buttons allows a button to have at most 256 kind of actions per button. Each Buttons may be associated with at most 256 buttons.

This class also contains a number of static helper methods that return information extracted from an event.

Field Summary		Page
`static int`	**CLICKED** The "clicked" action.	113
`static int`	**DOUBLE_CLICKED** The "double clicked" action.	113
`static int`	**LONG** The "long" action (button pressed for a "long" time).	113
`static int`	**PRESSED** The "pressed" action.	112
`static int`	**RELEASED** The "released" action.	113
`static int`	**REPEATED** The "repeated" action (button held down).	113

Constructor Summary	Page
Buttons`()` Creates a Buttons event generator that does not support click, doubleClick nor elapsedTime for any of its buttons.	114
Buttons`(int nbButtons)` Creates a Buttons event generator where elapsedTime, click and doubleClick features are supported and enabled for the first `nbButtons` (doubleClick feature is initialized with a 200ms delay).	113

Method Summary	Page

static int	**action**(int event) **Deprecated.** *use getAction(int)*	117
static int	**buttonID**(int event) **Deprecated.** *use getButtonID(int)*	117
boolean	**clickEnabled**(int buttonID) Returns true if the generator should send a click event.	115
boolean	**doubleClickEnabled**(int buttonID) Returns true if the generator should send a double click event.	115
long	**elapsedTime**(int buttonID) Returns the elapsed time in milliseconds between the two previous PRESSED events that occurred on the specified button.	117
void	**enableClick**(boolean enable, int buttonID) For the given button, specify whether the generator should send a click event for each pressed event.	114
void	**enableDoubleClick**(boolean enable, int click, int buttonID) For the given button, specify whether the generator should send a double click event.	114
static int	**getAction**(int event) Returns the button's action held by the button event.	117
static int	**getButtonID**(int event) Returns the button's id held by the button event.	117
int	**getEventType**() Returns the MicroUI event type for this button EventGenerator.	115
static boolean	**isClicked**(int event) Tells if an button event is a click event.	116
static boolean	**isDoubleClicked**(int event) Tells if an button event is a double click event.	116
static boolean	**isLong**(int event) Tells if an button event is a long event.	116
static boolean	**isPressed**(int event) Tells if a button event is a press event.	115
static boolean	**isReleased**(int event) Tells if a button event is a release event.	115
static boolean	**isRepeated**(int event) Tells if a button event is a repeat event.	116
void	**send**(int action, int buttonID) Sends a MicroUI event for the given action on given button to the listener of the Buttons.	117
boolean	**supportsExtendedFeatures**(int buttonID) Returns true if the button supports the extended features elapsedTime, click and doubleClick features.	114

Methods inherited from class ej.microui.EventGenerator
addToSystemPool, eventType, get, get, get, getID, getListener, removeFromSystemPool, setListener

Field Detail

PRESSED

```
public static final int PRESSED
```

The "pressed" action.

The value `0x00` is assigned to `PRESSED`.

RELEASED

`public static final int` **`RELEASED`**

The "released" action.

The value `0x01` is assigned to `RELEASED`.

LONG

`public static final int` **`LONG`**

The "long" action (button pressed for a "long" time).

The value `0x02` is assigned to `LONG`.

REPEATED

`public static final int` **`REPEATED`**

The "repeated" action (button held down).

The value `0x03` is assigned to `REPEATED`.

CLICKED

`public static final int` **`CLICKED`**

The "clicked" action.

The value `0x04` is assigned to `CLICKED`.

DOUBLE_CLICKED

`public static final int` **`DOUBLE_CLICKED`**

The "double clicked" action.

The value `0x05` is assigned to `DOUBLE_CLICKED`.

Constructor Detail

Buttons

`public` **`Buttons`**`(int nbButtons)`

Creates a Buttons event generator where elapsedTime, click and doubleClick features are supported and enabled for

the first `nbButtons` (doubleClick feature is initialized with a 200ms delay).

Parameters:
`nbButtons` - the number of buttons that support the extended features
See Also:
Buttons()

Buttons

```
public Buttons()
```

Creates a Buttons event generator that does not support click, doubleClick nor elapsedTime for any of its buttons. The effect is identical to:
`new Buttons(0).`

Method Detail

enableClick

```
public void enableClick(boolean enable,
                        int buttonID)
```

For the given button, specify whether the generator should send a click event for each pressed event. Note that this method has no effect if `buttonID` refers to a button that does not have support for click events (see Buttons(int)).

Parameters:
`enable` - `true` to enable the click function on the button
`buttonID` - the button

---

enableDoubleClick

```
public void enableDoubleClick(boolean enable,
                              int click,
                              int buttonID)
```

For the given button, specify whether the generator should send a double click event. Note that this method has no effect if `buttonID` refers to a button that does not have support for doubleClick events (see Buttons(int)).

Parameters:
`enable` - `true` to enable the double click function on the button. Click event is also enabled if it was not.
`click` - the maximum time elapsed between two clicks (in milliseconds) to generate a double click event
`buttonID` - the button

---

supportsExtendedFeatures

```
public boolean supportsExtendedFeatures(int buttonID)
```

Returns `true` if the button supports the extended features elapsedTime, click and doubleClick features.

Parameters:
`buttonID` - the button
Returns:
`true` if the button supports the extended features. and double click.

clickEnabled

```
public boolean clickEnabled(int buttonID)
```

Returns `true` if the generator should send a click event. Note that this method has no effect if `buttonID` refers to a button that does not have support for click events (see Buttons(int)).

Parameters:
`buttonID` - the button
Returns:
`true` if the generator should send a click event.

doubleClickEnabled

```
public boolean doubleClickEnabled(int buttonID)
```

Returns `true` if the generator should send a double click event. Note that this method has no effect if `buttonID` refers to a button that does not have support for double click events (see Buttons(int)).

Parameters:
`buttonID` - the button
Returns:
`true` if the generator should send a dooble click event.

getEventType

```
public int getEventType()
```

Returns the MicroUI event type for this button `EventGenerator`. Default value is `Event.BUTTON`.

Overrides:
getEventType in class EventGenerator
Returns:
the event type

isPressed

```
public static boolean isPressed(int event)
```

Tells if a button event is a press event.

Parameters:
`event` - the button event.
Returns:
true if the button event is a press event.

isReleased

```
public static boolean isReleased(int event)
```

Tells if a button event is a release event.

Parameters:
`event` - the button event.
Returns:
true if the button event is a release event.

---

isRepeated

```
public static boolean isRepeated(int event)
```

Tells if a button event is a repeat event.

Parameters:
`event` - the button event.
Returns:
true if the button event is a repeat event.

---

isLong

```
public static boolean isLong(int event)
```

Tells if an button event is a long event.

Parameters:
`event` - the button event.
Returns:
true if the button event is a long event.

---

isClicked

```
public static boolean isClicked(int event)
```

Tells if an button event is a click event.

Parameters:
`event` - the button event.
Returns:
true if the button event is a click event.

---

isDoubleClicked

```
public static boolean isDoubleClicked(int event)
```

Tells if an button event is a double click event.

Parameters:
`event` - the button event.
Returns:
true if the button event is a double click event.

---

buttonID

```
public static int buttonID(int event)
```

Deprecated. *use getButtonID(int)*

---

getButtonID

```
public static int getButtonID(int event)
```

Returns the button's id held by the button event.

Parameters:
`event` - the button event.
Returns:
the button's id held by the button event.

---

action

```
public static int action(int event)
```

Deprecated. *use getAction(int)*

---

getAction

```
public static int getAction(int event)
```

Returns the button's action held by the button event.

Parameters:
`event` - the button event.
Returns:
the button's action held by the button event.

---

elapsedTime

```
public long elapsedTime(int buttonID)
```

Returns the elapsed time in milliseconds between the two previous PRESSED events that occurred on the specified button. The elapsedTime for the very first occurrence has no meaning.

Parameters:
`buttonID` - the button on which to get the elapsed time
Returns:
the elapsed time in milliseconds or -1 when the elapsedTime has no meaning or if `buttonID` refers to a button that does not have support for elapsedTime (see Buttons(int))

---

send

```
public void send(int action,
                 int buttonID)
```

Sends a MicroUI event for the given action on given button to the listener of the Buttons. Buttons will generate a will generate a CLICKED and/or DOUBLE_CLICKED events if the matching button's feature is enabled.

This method is useful when other input mechanisms wish to simulate button actions.

Parameters:

`action` - the button's action: PRESSED, RELEASED, LONG, REPEATED.
`buttonID` - the button on which the action occurred

Class ComponentView

ej.microui.io

```
java.lang.Object
  └
    ej.microui.io.ComponentView
```

Direct Known Subclasses:
CompositeView, View

```
abstract public class ComponentView
extends Object
```

`ComponentView` is an abstract class that represents a visible area. A `ComponentView` is defined by a relative position `(x,y)`, and a size `(width,height)`.
Once this `ComponentView` is set on a `Viewable` and the latter is shown, the paint method of the former is called automatically by the system when a repaint is required. The position of the `ComponentView` in this case is relative to its `Viewable`.

Constructor Summary	Page
ComponentView`(int x, int y, int width, int height)` Creates a `ComponentView` with given attributes: its size is `width * height` and its relative location is `(x,y)`.	120

Method Summary		Page
int	**getAbsoluteX**`()` Returns the absolute x coordinate of the `ComponentView`.	121
int	**getAbsoluteY**`()` Returns the absolute y coordinate of the `ComponentView`.	121
int	**getHeight**`()` Returns the height of the `ComponentView`.	122
int	**getWidth**`()` Returns the width of the `ComponentView`.	122
int	**getX**`()` Returns the relative x coordinate of the `ComponentView`.	121
int	**getY**`()` Returns the relative y coordinate of the `ComponentView`.	121
boolean	**isVisible**`()` Checks whether the `ComponentView` is visible on a display.	123
abstract void	**paint**`(`GraphicsContext `g)` Draws the `ComponentView`.	123
void	**repaint**`()` Requests a repaint for the `ComponentView`.	122
void	**repaint**`(int x, int y, int width, int height)` Requests a repaint for the specified area in the `ComponentView`.	122
void	**update**`(int x, int y, int width, int height)` Updates the `ComponentView` attributes.	120
void	**updateLocation**`(int x, int y)` Updates the `ComponentView` location.	120

void	**updateSize**(int width, int height) Updates the ComponentView size.	121

Constructor Detail

ComponentView

```
public ComponentView(int x,
                     int y,
                     int width,
                     int height)
```

Creates a ComponentView with given attributes: its size is width * height and its relative location is (x,y).

Parameters:

x - the relative x coordinate - a negative value is permitted
y - the relative y coordinate - a negative value is permitted
width - the zone width
height - the zone height

Method Detail

update

```
public void update(int x,
                   int y,
                   int width,
                   int height)
```

Updates the ComponentView attributes. This does not trigger a repaint() of the ComponentView.

Parameters:

x - the new relative x coordinate - a negative value is permitted.
y - the new relative y coordinate - a negative value is permitted.
width - the new ComponentView width - a negative value is turned to 0.
height - the new ComponentView height - a negative value is turned to 0.

See Also:

updateSize(int, int), updateLocation(int, int)

---

updateLocation

```
public void updateLocation(int x,
                           int y)
```

Updates the ComponentView location. This does not trigger a repaint() of the ComponentView.

Parameters:

x - the new relative x coordinate - a negative value is permitted.
y - the new relative y coordinate - a negative value is permitted.

See Also:

update(int, int, int, int), updateSize(int, int)

---

updateSize

```
public void updateSize(int width,
                       int height)
```

Updates the `ComponentView` size. This does not trigger a repaint() of the ComponentView.

Parameters:

`width` - the new `ComponentView` width - a negative value is turned to 0.

`height` - the new `ComponentView` height - a negative value is turned to 0.

See Also:

update(int, int, int, int), updateLocation(int, int)

---

getX

```
public int getX()
```

Returns the relative x coordinate of the `ComponentView`.

Returns:

the relative x coordinate of the `ComponentView`

---

getY

```
public int getY()
```

Returns the relative y coordinate of the `ComponentView`.

Returns:

the relative y coordinate of the `ComponentView`

---

getAbsoluteX

```
public int getAbsoluteX()
```

Returns the absolute x coordinate of the `ComponentView`. That is, the x coordinate relative to the origin of the display.

Returns:

the absolute x coordinate of the `ComponentView`

Throws:

`IllegalArgumentException` - if the view is not connected to a Viewable

---

getAbsoluteY

```
public int getAbsoluteY()
```

Returns the absolute y coordinate of the `ComponentView`. That is, the y coordinate relative to the origin of the display.

Returns:

the absolute y coordinate of the `ComponentView`

Throws:
`IllegalArgumentException` - if the view is not connected to a Viewable

---

getWidth

```
public int getWidth()
```

Returns the width of the `ComponentView`.

Returns:
the width of the `ComponentView`

---

getHeight

```
public int getHeight()
```

Returns the height of the `ComponentView`.

Returns:
the height of the `ComponentView`

---

repaint

```
public void repaint()
```

Requests a repaint for the `ComponentView`. Calling this method may result in subsequent calls to `ComponentView.paint()` methods.
The call to `ComponentView.paint()` occurs asynchronously to the call to `repaint()`. That is, `repaint()` will not block waiting for `ComponentView.paint()` to finish. The `ComponentView.paint()` method will either be called after the caller of `repaint()` (if the caller is a callback) or in another thread entirely.

To synchronize with their `ComponentView.paint()` routine, applications can use either `Display.callSerially()` or `Display.waitForEvent()`, or they can code explicit synchronization into their `ComponentView.paint()` routine.

---

repaint

```
public void repaint(int x,
                    int y,
                    int width,
                    int height)
```

Requests a repaint for the specified area in the `ComponentView`. Repaint area is a rectangle defined relatively to the `ComponentView` object. Calling this method may result in subsequent call to paint(GraphicsContext) methods. Moreover, no hypothesis can be made on the clipping region of the GraphicsContext instance passed to the next paint(GraphicsContext) call.
The call to `ComponentView.paint()` occurs asynchronously to the call to `repaint()`. That is, `repaint()` will not block waiting for `ComponentView.paint()` to finish. The `ComponentView.paint()` method will either be called after the caller of `repaint()` (if the caller is a callback) or in another thread entirely.

To synchronize with their `ComponentView.paint()` routine, applications can use either `Display.callSerially()` or `Display.waitForEvent()`, or they can code explicit synchronization into their `ComponentView.paint()` routine.

Parameters:

`x` - the relative x coordinate of the area to repaint
`y` - the relative y coordinate of the area to repaint
`width` - the width of the area to repaint
`height` - the height of the area to repaint

---

paint

```
public abstract void paint(GraphicsContext g)
```

Draws the `ComponentView`. This method must be implemented by subclasses to render graphics on a display.

The `ComponentView` can be drawn within the given `GraphicsContext` which defines a clip region where drawings can be performed. The `GraphicsContext` object's clip region will be set to the `ComponentView` region.

No hypothesis can be made on the received region, for instance implementation is not supposed to adjust the clipping zone. As a result an implementation of `paint()` should draw pixels in the entire clipping region to remove all previous drawings.
Drawings done in the `paint()` method must effectively be visible on the display by the return of the method at the latest.

The `GraphicsContext g` must only be used in the scope of this method. It must not be cached since its later state and future drawing operations are undefined.

The `paint()` method should never be called directly by the application, but instead by the system through the use of a serialized repaint event.

On entry the `GraphicsContext`'s color is black, its font is the default font of the system, its stroke style is `SOLID` and the origin of its coordinate system is at the upper-left corner of the `ComponentView`.

Parameters:

`g` - the `GraphicsContext` object to use to draw for this `ComponentView`

---

isVisible

```
public boolean isVisible()
```

Checks whether the `ComponentView` is visible on a display. The `ComponentView` is visible if it is set on a viewable and this viewable is visible on the display.

Returns:

`true` if the `ComponentView` is currently visible, `false` otherwise

Class CompositeView

ej.microui.io

```
java.lang.Object
  └
    ej.microui.io.ComponentView
      └
        ej.microui.io.CompositeView
```

```
public class CompositeView
extends ComponentView
```

`CompositeView` is a container class of `ComponentView`s. Note that `CompositeView` is also a subclass of `ComponentView`.

Once a `ComponentView` is assigned to a `Viewable` it must not be added:
- another time to the hierarchy of `ComponentView`s that are displayed on the `Viewable`
- to the hierarchy of another `Viewable`

The position of a `ComponentView` in a `CompositeView` is relative to the latter.
By default, the `ComponentView`s of a `CompositeView` are rendered from the first one added to the last one.

Field Summary		Page
`static int`	**BRING_FORWARD** The arrange "bring forward" constant.	125
`static int`	**BRING_TO_FRONT** The arrange "bring to front" constant.	125
`static int`	**SEND_BACKWARD** The arrange "send backward" constant.	125
`static int`	**SEND_TO_BACK** The arrange "send to back" constant.	125

Constructor Summary	Page
CompositeView`(int x, int y, int width, int height)` Creates a zone with given attributes.	126

Method Summary		Page
`void`	**add** (ComponentView `view`) Adds a `ComponentView` object to this `CompositeView` object.	127
`void`	**arrange** (ComponentView `view, int arrange`) By default, the `ComponentView`s are rendered from the first one added to the `CompositeView` to the last one.	127
`void`	**fillBackground** (`boolean fill`) Sets whether the `CompositeView`'s background has to be filled when this `CompositeView` gets painted.	126
`void`	**fillBackground** (GraphicsContext `g`) Fills the background of the `CompositeView`.	126
ComponentView`[]`	**getViews** () Returns all `ComponentView` objects contained in this `CompositeView`.	127

void	**paint** (GraphicsContext g) Renders the CompositeView.	*126*
void	**remove** (ComponentView view) Removes a View from this CompositeView.	*127*
void	**removeAllViews** () Removes all ComponentView objects contained in this CompositeView.	*127*

Methods inherited from class ej.microui.io.ComponentView
getAbsoluteX, getAbsoluteY, getHeight, getWidth, getX, getY, isVisible, repaint, repaint, update, updateLocation, updateSize

Field Detail

BRING_TO_FRONT

```
public static final int BRING_TO_FRONT
```

The arrange "bring to front" constant. Use this constant to move the a ComponentView to the top of the stacking order, so that it is in front of other ComponentViews.

Value 0 is assigned to BRING_TO_FRONT.

BRING_FORWARD

```
public static final int BRING_FORWARD
```

The arrange "bring forward" constant. Use this constant to move the a ComponentView up one level, so that it is closer to top of the stacking order.

Value 1 is assigned to BRING_FORWARD.

SEND_BACKWARD

```
public static final int SEND_BACKWARD
```

The arrange "send backward" constant. Use this constant to move a ComponentView down one level, so that it is closer to the bottom of the stacking order.

Value 2 is assigned to SEND_BACKWARD.

SEND_TO_BACK

```
public static final int SEND_TO_BACK
```

The arrange "send to back" constant. Use this constant to move a ComponentView to the bottom of the stacking order, so that it is behind the other objects.

Value 3 is assigned to SEND_TO_BACK.

Constructor Detail

CompositeView

```
public CompositeView(int x,
                     int y,
                     int width,
                     int height)
```

Creates a zone with given attributes.

Parameters:

`x` - the relative x coordinate
`y` - the relative y coordinate
`width` - the zone width
`height` - the zone height

Method Detail

fillBackground

```
public final void fillBackground(boolean fill)
```

Sets whether the `CompositeView`'s background has to be filled when this `CompositeView` gets painted. By default background filling is disabled. If enabled the background is painted white.

Parameters:

`fill` - true if the background has to be filled

---

paint

```
public void paint(GraphicsContext g)
```

Renders the CompositeView.
The application code should never call `paint()` directly. This method is called only by the MicroUI system. When a general (`Viewable.repaint()`) or local `repaint()` is done, the `paint()` method is called. This very method is only in charge of painting the `CompositeView` instance. Its default behavior is to fill the background if `fillBackground(true)` has been invoked on this `CompositeView` object first.

Overrides:

paint in class ComponentView

Parameters:

`g` - the `GraphicsContext` object to use for rendering the `CompositeView`

---

fillBackground

```
public void fillBackground(GraphicsContext g)
```

Fills the background of the `CompositeView`. This method should not be called directly by the application but rather from `paint` method.

Parameters:

`g` - the `GraphicsContext` object to use for filling the background

add

```
public final void add(ComponentView view)
```

Adds a ComponentView object to this CompositeView object. This does not trigger a ComponentView.repaint() of the ComponentView.

Parameters:
view - the ComponentView to be added.
Throws:
NullPointerException - if view is null
IllegalArgumentException - if the specified view or one of its children is already connected to a Viewable.

remove

```
public final void remove(ComponentView view)
```

Removes a View from this CompositeView. This does not trigger a ComponentView.repaint() of the ComponentView. If the given View is not contained in this CompositeView, the method has no effect.

Parameters:
view - the view to remove from the composite.

removeAllViews

```
public final void removeAllViews()
```

Removes all ComponentView objects contained in this CompositeView.

arrange

```
public final void arrange(ComponentView view,
                          int arrange)
```

By default, the ComponentViews are rendered from the first one added to the CompositeView to the last one. This method allows the user to change a ComponentView's position in relation to the other ComponentViews. This method does nothing if the given ComponentView has not been added to the CompositeView.

Parameters:
view - the ComponentView to arrange
arrange - the arrange movement: BRING_TO_FRONT, BRING_FORWARD, SEND_BACKWARD or SEND_TO_BACK.
Throws:
IllegalArgumentException - if the arrange value is invalid

getViews

```
public final ComponentView[] getViews()
```

Returns all ComponentView objects contained in this CompositeView.

Returns:

all `ComponentView` objects contained in this `CompositeView`

Class Display

ej.microui.io

```
java.lang.Object
  └
    ej.microui.io.Screen
      └
        ej.microui.io.Display
```

```
public class Display
extends Screen
```

A `Display` object represents a pixelated screen in the platform, and there is a display for each such pixelated screen. Available displays can be retrieved with the method getAllDisplays(). A default display is defined in every MicroUI implementation and can be fetched with the method getDefaultDisplay().

A display is able to render a `Displayable` on its implementation screen. Only one `Displayable` can be set on a display at a time; it is said to be visible or to be shown. The visible `Displayable` can be retrieved with the method getDisplayable().

Displayable.show() allows the Displayable to be selected for rendering on its display. It can be called at any time by the application, for instance in response to user inputs.

`Display` uses a GraphicsContext to draw on its corresponding screen. All draw actions are serialized. The application should not use a display's graphics context outside the events mechanism `repaint()` and `paint()`. Nevertheless, for exceptional cases a new `GraphicsContext` may be created using `getNewGraphicsContext()`. This new `GraphicsContext` bypasses the standard serialized drawing mechanism and allows drawings to be rendered on the display at any time.

See Viewable and View for more information on serialized draw events.

All events on a display are serialized: repaint, callSerially, handleEvent etc. A display uses a `FIFOPump` to manage its serialized event mechanism.

Method Summary		Page
void	**callSerially**(Runnable run) Serializes a call event in the system event stream.	135
static Display[]	**getAllDisplays**() Returns all available displays.	134
int	**getBacklight**() Returns the current backlight setting	133
int	**getBacklightColor**() Returns the current backlight color.	133
int	**getBPP**() Returns the number of bits per pixel of the display.	131
int	**getContrast**() Returns the contrast of the display.	132
static Display	**getDefaultDisplay**() Returns the default display of the system.	134
Displayable	**getDisplayable**() Returns the current `Displayable` object in the `Display`.	134
int	**getHeight**() Returns the height in pixels of the display screen area available to the application.	130

ExplicitFlush	**getNewExplicitFlush**() Returns a new `ExplicitFlush` which works on the same system screen as this display.	*134*
GraphicsContext	**getNewGraphicsContext**() Returns a new `GraphicsContext` which works on the same system screen as this display.	*134*
int	**getNumberOfAlphaLevels**() Gets the number of alpha transparency levels supported by the implementation.	*131*
int	**getNumberOfColors**() Gets the number of colors that can be represented on the device.	*131*
int	**getWidth**() Returns the width in pixels of the display screen area available to the application.	*131*
void	**handleEvent**(int event) Inject a MicroUI event to be handled by the event generator associated with this Display.	*136*
boolean	**hasBacklight**() Tells whether the display has backlight.	*132*
boolean	**isColor**() Tells whether the display offers color.	*131*
boolean	**isDoubleBuffered**() Returns if the display uses an underlying double buffer (either hardware or software).	*132*
void	**setBacklight**(int backlight) Sets the backlight of the display.	*133*
void	**setBacklightColor**(int rgbColor) Sets the current backlight color, if it is allowed by implementation.	*133*
void	**setContrast**(int contrast) Sets the contrast of the display.	*132*
void	**setPriority**(int priority) Sets the priority of the display events processing.	*135*
void	**switchBacklight**(boolean on) Switches on or off the backlight of the display.	*132*
void	**waitForEvent**() Blocks the current thread (with all its locks) until all events outstanding at the time of the call have been processed.	*135*
void	**waitForEvent**(int event) Sends `event` in the event stream and blocks the current thread (with all its locks) until the `event` processing is finished.	*135*

Methods inherited from class ej.microui.io.Screen
getEventHandler

Method Detail

getHeight

```
public int getHeight()
```

Returns the height in pixels of the display screen area available to the application.

Overrides:
getHeight in class Screen

Returns:
height of the display screen area.

getWidth

```
public int getWidth()
```

Returns the width in pixels of the display screen area available to the application.

Overrides:
getWidth in class Screen
Returns:
width of the display screen area.

getBPP

```
public int getBPP()
```

Returns the number of bits per pixel of the display.

Returns:
the number of bits per pixel

isColor

```
public boolean isColor()
```

Tells whether the display offers color.

Overrides:
isColor in class Screen
Returns:
if display has color

getNumberOfColors

```
public int getNumberOfColors()
```

Gets the number of colors that can be represented on the device.
Note that the number of colors for a black and white display is 2.

Overrides:
getNumberOfColors in class Screen
Returns:
the number of colors

getNumberOfAlphaLevels

```
public int getNumberOfAlphaLevels()
```

Gets the number of alpha transparency levels supported by the implementation.

The minimum possible is 2, which represents full transparency and full opacity with no blending. If the return value is greater than 2, the implementation manages blending.

Returns:
the number of alpha levels

---

isDoubleBuffered

```
public boolean isDoubleBuffered()
```

Returns if the display uses an underlying double buffer (either hardware or software). This technique is useful to avoid flickering while the user is drawing.

Returns:
true if and only if a double buffer is used for the display

---

hasBacklight

```
public boolean hasBacklight()
```

Tells whether the display has backlight.

Overrides:
hasBacklight in class Screen
Returns:
if display has backlight

---

setContrast

```
public void setContrast(int contrast)
```

Sets the contrast of the display. `contrast` value range is 0-100

Overrides:
setContrast in class Screen
Parameters:
`contrast` - the new value of the contrast

---

getContrast

```
public int getContrast()
```

Returns the contrast of the display.

Overrides:
getContrast in class Screen
Returns:
the current contrast of the display (range 0-100)

---

switchBacklight

```
public void switchBacklight(boolean on)
```

Switches on or off the backlight of the display.

Overrides:
switchBacklight in class Screen
Parameters:
`on` - Switch on the backlight if `true`; switch off the backlight if `false`

---

setBacklight

```
public void setBacklight(int backlight)
```

Sets the backlight of the display. `backlight` value range is 0-100

Overrides:
setBacklight in class Screen
Parameters:
`backlight` - the new value of the backlight

---

getBacklight

```
public int getBacklight()
```

Returns the current backlight setting

Overrides:
getBacklight in class Screen
Returns:
the current backlight setting (range 0-100)

---

setBacklightColor

```
public void setBacklightColor(int rgbColor)
```

Sets the current backlight color, if it is allowed by implementation.

Overrides:
setBacklightColor in class Screen
Parameters:
`rgbColor` - the color to set

---

getBacklightColor

```
public int getBacklightColor()
```

Returns the current backlight color. Returned value is interpreted as a 24-bit RGB color, where the eight less significant bits matches the blue component, the next eight bits matches the green component and the next eight bits matches the red component. By default, this method returns 0xFFFFFF (white) and sub-classes should overwrite this default behavior.

Overrides:
getBacklightColor in class Screen
Returns:
the color of the backlight

---

getAllDisplays

`public static Display[] getAllDisplays()`

Returns all available displays. It is never `null` but the array may be empty.

Returns:
all available displays

---

getDefaultDisplay

`public static Display getDefaultDisplay()`

Returns the default display of the system. It can be null if there is no display. The notion of default display is defined by the implementation.

Returns:
the default display or null.

---

getDisplayable

`public Displayable getDisplayable()`

Returns the current `Displayable` object in the `Display`.
The value returned by `getDisplayable()` may be `null` if no `Displayable` is visible.

Returns:
the current `Displayable` object in the `Display`

---

getNewGraphicsContext

`public GraphicsContext getNewGraphicsContext()`

Returns a new `GraphicsContext` which works on the same system screen as this display. With this `GraphicsContext`, it is possible to draw on the system screen at any time without modifying the normal system execution. The new graphics context has its own clip, color, font etc. After each draw action (a drawLine for example), the system screen will show the drawn pixels.
If the normal system execution is repainting at the same time, the last draw action will be visible (the previous one will be hidden by the last one). It is not possible to determine which draw action will be done last.

Returns:
a new graphics context on the display
Throws:
`OutOfMemoryError` - if there is not enough room to add a new graphics context.

---

getNewExplicitFlush

`public ExplicitFlush getNewExplicitFlush()`

Returns a new `ExplicitFlush` which works on the same system screen as this display. With this `ExplicitFlush`, it is possible to draw on the system screen at any time without modifying the normal system execution. The new

graphics context has its own clip, color, font etc. Each draw action will not be automatically flushed. The user has to flush it via the `ExplicitFlush.flush()` method.
If the normal system execution is repainting at the same time, the last unflushed draw actions will be visible (the previous one will be hidden by the last one). It is not possible to determine which draw action will be done last.

Returns:
a new graphics context with explicit flush on the display
Throws:
`OutOfMemoryError` - if there is not enough room to add a new graphics context.

---

callSerially

```
public void callSerially(Runnable run)
```

Serializes a call event in the system event stream. When the event is processed, the `run()` method of the `Runnable` object is called.
Multiple call events may be requested with `callSerially()`: they will occur in the order in which they were requested (first in first out policy).
The call to the `run()` method of the `Runnable` object is performed asynchronously Therefore `callSerially()` will never block waiting for the `run()` method to finish.
The `run()` method should return quickly, as with other callback methods.
The `callSerially()` mechanism may be used by applications as a synchronization tool in the event stream.

Parameters:
`run` - a `Runnable` object to call

---

waitForEvent

```
public void waitForEvent(int event)
```

Sends `event` in the event stream and blocks the current thread (with all its locks) until the `event` processing is finished.

Parameters:
`event` - the event to send and to wait for
Throws:
`RuntimeException` - if the current thread is the `Display`'s events thread.

---

waitForEvent

```
public void waitForEvent()
```

Blocks the current thread (with all its locks) until all events outstanding at the time of the call have been processed.

Throws:
`RuntimeException` - if the current thread is the `Display`'s events thread.

---

setPriority

```
public void setPriority(int priority)
```

Sets the priority of the display events processing.

Parameters:
`priority` - the new priority of display events processing
Throws:
`IllegalArgumentException` - If the priority is not in the range `Thread.MIN_PRIORITY` to `Thread.MAX_PRIORITY`.

---

handleEvent

```
public void handleEvent(int event)
```

Inject a MicroUI event to be handled by the event generator associated with this Display.

Parameters:
`event` - an event in the MicroUI format

Class Displayable

ej.microui.io

```
java.lang.Object
  └
    ej.microui.io.Displayable
```

Direct Known Subclasses:
Viewable

```
abstract public class Displayable
extends Object
```

`Displayable` is an abstract class which defines the very objects that can be shown on a `Display`.
A `Displayable` object is built for a specific `Display` which can not be changed afterwards. A `Displayable` may be shown or hidden, but at most one `Displayable` is shown per `Display`.

Subclasses should define the `Displayable` contents and their possible interactions with the user.

By default, a new `Displayable` object is not visible on its display.

See Also:
Display

Constructor Summary	Page
Displayable(Display display) The newly created displayable is built for the given `Display` and is hidden.	138

Method Summary		Page
Display	**getDisplay**() Returns the displayable's display which is never `null`.	138
void	**hide**() Sets the displayable as hidden on its display.	138
protected void	**hideNotify**() This method is called by system as soon as the displayable becomes hidden.	139
boolean	**isShown**() Checks whether the `Displayable` is visible on its display.	138
abstract void	**paint**(GraphicsContext g) Draws the `Displayable`.	139
abstract void	**performAction**(int event) Handles an event.	139
void	**repaint**() Requests a repaint for the entire `Displayable`.	139
void	**show**() Sets the displayable as visible on its display.	138
protected void	**showNotify**() This method is called by system as soon as the displayable becomes visible.	138

Constructor Detail

Displayable

`public` **`Displayable`**`(`Display `display)`

The newly created displayable is built for the given `Display` and is hidden.

Parameters:
`display` - the display for which the displayable is created
Throws:
`NullPointerException` - if `display` is `null`

Method Detail

getDisplay

`public` Display **`getDisplay`**`()`

Returns the displayable's display which is never `null`.

Returns:
the displayable's display

---

isShown

`public boolean` **`isShown`**`()`

Checks whether the `Displayable` is visible on its display.

Returns:
`true` if the `Displayable` is currently visible, `false` otherwise

---

show

`public void` **`show`**`()`

Sets the displayable as visible on its display.

---

hide

`public void` **`hide`**`()`

Sets the displayable as hidden on its display. If the displayable is not visible, this method has no effect.

---

showNotify

`protected void` **`showNotify`**`()`

This method is called by system as soon as the displayable becomes visible. Application should override this method

to control its own displayables.

---

hideNotify

`protected void` **`hideNotify`**`()`

This method is called by system as soon as the displayable becomes hidden. Application should override this method to control its own displayables.

---

repaint

`public void` **`repaint`**`()`

Requests a repaint for the entire `Displayable`. Calling this method may result in subsequent call(s) to `paint()` on the displayable.

If the viewable is not visible, this call has no effect.

The call(s) to `paint()` occurs asynchronously to the call to `repaint()`. That is, `repaint()` will not block waiting for `paint()` to finish. The `paint()` method will either be called after the caller of `repaint()` (if the caller is a callback) or on another thread entirely.

To synchronize with the `paint()` routine, applications can use either `Display.callSerially()` or `Display.waitForEvent()`, or they can code explicit synchronization into their `paint()` routine.

---

paint

`public abstract void` **`paint`**`(`GraphicsContext `g)`

Draws the `Displayable`. This method must be implemented by subclasses to render graphics on a display.

---

performAction

`public abstract void` **`performAction`**`(int event)`

Handles an event. This method must be implemented by subclasses to handle the given event.

Class DisplayFont

ej.microui.io

```
java.lang.Object
  └
    ej.microui.io.Font
      └
        ej.microui.io.DisplayFont
```

```
final public class DisplayFont
extends Font
```

A `DisplayFont` defines how text is rendered on a Display.
DisplayFonts are never created by applications, but are rather retrieved from the implementation environment.

An application can get all available fonts with getAllFonts(), or query for a particular font: in this case the implementation will return the most appropriate font matching the request.

The height gives the height of a line of text with the font.

See Also:
Font

Fields inherited from class ej.microui.io.Font
ARABIC, ARMENIAN, BALINESE, BAMUM, BATAK, BENGALI, BOPOMOFO, BRAILLE, BUGINESE, BUHID, CANADIAN_ABORIGINAL, CHAM, CHEROKEE, COMMON, COPTIC, CUNEIFORM, CYPRIOT, CYRILLIC, DESERET, DEVANAGARI, ETHIOPIC, GEORGIAN, GLAGOLITIC, GOTHIC, GREEK, GUJARATI, GURMUKHI, HAN, HANGUL, HANUNOO, HEBREW, HIRAGANA, INHERITED, JAVANESE, KANNADA, KATAKANA, KAYAH_LI, KHAROSHTHI, KHMER, LAO, LATIN, LEPCHA, LIMBU, LISU, MALAYALAM, MANDAIC, MEETEI_MAYEK, MONGOLIAN, MYANMAR, NEW_TAI_LUE, NKO, OGHAM, OL_CHIKI, ORIYA, OSMANYA, PHAGS_PA, PHOENICIAN, REJANG, RUNIC, SAMARITAN, SAURASHTRA, SHAVIAN, SINHALA, STYLE_BOLD, STYLE_ITALIC, STYLE_PLAIN, STYLE_UNDERLINED, SUNDANESE, SYLOTI_NAGRI, SYRIAC, TAGALOG, TAGBANWA, TAI_LE, TAI_THAM, TAI_VIET, TAMIL, TELUGU, THAANA, THAI, TIBETAN, TIFINAGH, UGARITIC, VAI, YI

Method Summary		***Page***
`static DisplayFont[]`	**getAllFonts**`()` Returns an array containing all available DisplayFonts in the system.	*142*
`int`	**getBaselinePosition**`()` Returns the distance in pixels from the top of the text to the text's baseline.	*143*
`static DisplayFont`	**getDefaultFont**`()` Returns the default font for all Displays.	*141*
`String`	**getDescriptor**`()` Returns the descriptor of the font or null if no descriptor is available.	*141*
`static DisplayFont`	**getFont**`(int identifier, int height, int style)` Returns a `DisplayFont` matching the requested characteristics as close as possible.	*142*
`int`	**getHeight**`()` Returns the height of a line of text with this font.	*142*
`int[]`	**getIdentifiers**`()` Returns an array of identifiers supported by the font.	*141*
`boolean`	**isMonospaced**`()` Returns `true` if the font is monospaced.	*141*

Methods inherited from class ej.microui.io.Font
charsWidth, charWidth, getStyle, isBold, isIdentifierSupported, isItalic, isPlain, isUnderlined, stringWidth, substringWidth, supportIdentifiers

Method Detail

getIdentifiers

```
public int[] getIdentifiers()
```

Returns an array of identifiers supported by the font. An identifier is an integer specified in this class or a specific integer defined by the MicroUI implementation.

Overrides:
getIdentifiers in class Font
Returns:
an array of identifier.
See Also:
Font.getIdentifiers()

---

getDescriptor

```
public String getDescriptor()
```

Returns the descriptor of the font or null if no descriptor is available.

Overrides:
getDescriptor in class Font
Returns:
the descriptor of the font
See Also:
Font.getDescriptor()

---

isMonospaced

```
public boolean isMonospaced()
```

Returns `true` if the font is monospaced. A monospaced font is a font which all characters have the same width.

Overrides:
isMonospaced in class Font
Returns:
`true` if the font is monospaced.
See Also:
Font.isMonospaced()

---

getDefaultFont

```
public static DisplayFont getDefaultFont()
```

Returns the default font for all Displays.
This method may return `null` if no font is declared in the system.

Returns:
the default font

---

getAllFonts

```
public static DisplayFont[] getAllFonts()
```

Returns an array containing all available DisplayFonts in the system.

Returns:
an array of fonts

---

getFont

```
public static DisplayFont getFont(int identifier,
                                  int height,
                                  int style)
```

Returns a `DisplayFont` matching the requested characteristics as close as possible.

Font is requested by specifying the required identifier, height and style. If no available font exactly matches the request, the system will attempt to provide the most appropriate font.

The implementation should use the following rules to determine a suitable font:

- A suitable font must support the specified identifier. If there is no available font with a matching identifier return the default font (null if there is no default font).
- From within the fonts that support the specified identifier, select the font that is the closest in height to the specified height. If there are two or more fonts equally close in height to the specified height select them all.
- From within the fonts selected in the previous rule, pick the font or fonts that match the most style flags.
- If more than one font is identified by the previous rule, the choice of font to return is implementation dependent (perhaps selected on the basis of which font will render at the highest quality).

Parameters:
`identifier` - the required identifier of the font
`height` - the required height of the font
`style` - the required combination of style constants
Returns:
a `DisplayFont` object or `null`

---

getHeight

```
public int getHeight()
```

Returns the height of a line of text with this font.

The height includes the size of the font as well as sufficient spacing below the text to ensure that lines of text drawn at this distance will be spaced appropriately.

Returns:
height of a line of text with this font

---

getBaselinePosition

```
public int getBaselinePosition()
```

Returns the distance in pixels from the top of the text to the text's baseline.

Returns:
the font baseline

Class ExplicitFlush

ej.microui.io

```
java.lang.Object
  └
    ej.microui.io.GraphicsContext
      └
        ej.microui.io.ExplicitFlush
```

```
public class ExplicitFlush
extends GraphicsContext
```

An `ExplicitFlush` is a `GraphicsContext` where flushing data to the screen must be done explicitly by the application. An `ExplicitFlush` is useful if the `Display` on which the `ExplicitFlush` is writing is double buffered. By using an `ExplicitFlush` the user can choose the best moment to flush its output. If the display is not double buffered, all drawing actions on this `GraphicsContext` are rendered immediately and the flush method has no effect.

See Also:

GraphicsContext, Display.getNewExplicitFlush()

Fields inherited from class ej.microui.io.GraphicsContext
AND, BASELINE, BOTTOM, DOTTED, HCENTER, INV_COLOR, INV_RESULT, LEFT, MINUS, OR, PLUS, RESET_FILTER, RIGHT, SOLID, TOP, VCENTER, XOR

Method Summary		Page
void	**flush**() Updates the display with the draw actions since the last flush if and only if the display is double buffered.	144

Methods inherited from class ej.microui.io.GraphicsContext
clipRect, copyArea, drawArc, drawARGB, drawChar, drawChars, drawCircle, drawDeformedImage, drawEllipse, drawHorizontalLine, drawImage, drawLine, drawPixel, drawPolygon, drawPolygon, drawRect, drawRegion, drawRoundRect, drawString, drawSubstring, drawVerticalLine, fillArc, fillCircle, fillEllipse, fillPolygon, fillPolygon, fillRect, fillRoundRect, getARGB, getClipHeight, getClipWidth, getClipX, getClipY, getColor, getDisplay, getDisplayColor, getEllipsis, getFilter, getFont, getStrokeStyle, getTranslateX, getTranslateY, readPixel, setClip, setColor, setEllipsis, setFilter, setFont, setStrokeStyle, translate

Method Detail

flush

```
public void flush()
```

Updates the display with the draw actions since the last flush if and only if the display is double buffered.

See Also:

Display.isDoubleBuffered()

Class FlyingImage

ej.microui.io

```
java.lang.Object
  └
    ej.microui.io.FlyingImage
```

```
public class FlyingImage
extends Object
```

The `FlyingImage` class defines an image to be displayed at the top level in the rendering depth of a display.

A `FlyingImage` contains an `Image`. This image associates the `FlyingImage` with a specific display (since an `Image` is created for a specific display).
Several `FlyingImage` objects may be associated with a `Display`. The flying images of an application are drawn above all drawings coming from the standard `paint()` calls. Flying images are drawn on a display in the order they are shown: The flying image drawn on top of the display is the last shown (it is above all the other drawings)

See Also:
Pointer.setFlyingImage(FlyingImage)

Constructor Summary	Page
FlyingImage(Image `skin`) Creates a new `FlyingImage`.	146

Method Summary		Page
Display	**getDisplay**() Returns the display associated to the `FlyingImage`.	147
Image	**getImage**() Returns the image associated to the `FlyingImage`.	146
int	**getX**() Get the x coordinate of the `FlyingImage` position	147
int	**getY**() Get the y coordinate of the `FlyingImage` position	147
void	**hide**() Sets the `FlyingImage` as hidden on its display.	146
boolean	**isShown**() Checks whether the `FlyingImage` is visible on its display.	146
void	**repaint**() Requests a repaint of the `FlyingImage`.	146
void	**setLocation**(`int x, int y`) Sets the location of the `FlyingImage`.	147
void	**show**() Sets the `FlyingImage` as visible on its display.	146

Constructor Detail

FlyingImage

`public` **`FlyingImage`**`(`Image `skin)`

Creates a new `FlyingImage`. On creation the `FlyingImage` is not shown - call show() to show it.

Parameters:
`skin` - the `Image` the `FlyingImage` is associated with.
Throws:
`NullPointerException` - if `skin` is null.
`OutOfMemoryError` - if there is not enough room to add a new flying image.

Method Detail

show

`public void` **`show`**`()`

Sets the `FlyingImage` as visible on its display.

---

hide

`public void` **`hide`**`()`

Sets the `FlyingImage` as hidden on its display.

---

isShown

`public boolean` **`isShown`**`()`

Checks whether the `FlyingImage` is visible on its display.

Returns:
`true` if the `FlyingImage` is currently visible, `false` otherwise

---

repaint

`public void` **`repaint`**`()`

Requests a repaint of the `FlyingImage`.

---

getImage

`public` Image **`getImage`**`()`

Returns the image associated to the `FlyingImage`.

Returns:
the image associated to this `FlyingImage`

---

setLocation

```
public void setLocation(int x,
                        int y)
```

Sets the location of the `FlyingImage`.

Parameters:
`x` - the x coordinate where to set the `FlyingImage`
`y` - the y coordinate where to set the `FlyingImage`

---

getX

```
public int getX()
```

Get the x coordinate of the `FlyingImage` position

Returns:
the x coordinate of the `FlyingImage` position

---

getY

```
public int getY()
```

Get the y coordinate of the `FlyingImage` position

Returns:
the y coordinate of the `FlyingImage` position

---

getDisplay

```
public Display getDisplay()
```

Returns the display associated to the `FlyingImage`.

Returns:
the display associated to the `FlyingImage`.

Class Font

ej.microui.io

```
java.lang.Object
  └
    ej.microui.io.Font
```

Direct Known Subclasses:
AlphaNumericDisplayFont, DisplayFont

```
abstract public class Font
extends Object
```

A `Font` defines how text is rendered on a screen.

A `Font` is defined by one or more identifiers, a style and a descriptor. It may or may not be monospaced.

An identifier is an integer and it specifies the font's capabilities. For instance, when a font holds the `LATIN` identifier, that means the font is able to render all Latin languages. If the font holds too the `ARABIC` identifier, that means the font is also able to print Arabic words.
There are 80 predefined identifiers (1 to 80). A font can also hold other special identifiers that provide a useful way to recognize a specific font. For instance, a font that contains some special characters as arrows, smileys, can be tagged by the font's creator with a special identifier.

The style may combine several style attributes such as `STYLE_PLAIN`, `STYLE_BOLD`, `STYLE_ITALIC` or `STYLE_UNDERLINED`.

The descriptor is a helpful string that describes the font.

Field Summary		Page
static int	**ARABIC** Constant for arabic font identifier.	153
static int	**ARMENIAN** Constant for armenian font identifier.	153
static int	**BALINESE** Constant for balinese font identifier.	153
static int	**BAMUM** Constant for bamum font identifier.	166
static int	**BATAK** Constant for batak font identifier.	166
static int	**BENGALI** Constant for bengali font identifier.	153
static int	**BOPOMOFO** Constant for bopomofo font identifier.	154
static int	**BRAILLE** Constant for braille font identifier.	154
static int	**BUGINESE** Constant for buginese font identifier.	154
static int	**BUHID** Constant for buhid font identifier.	154
static int	**CANADIAN_ABORIGINAL** Constant for canadian aboriginal font identifier.	154

static int	**CHAM** Constant for cham font identifier.	165
static int	**CHEROKEE** Constant for cherokee font identifier.	154
static int	**COMMON** Constant for common font identifier.	163
static int	**COPTIC** Constant for coptic font identifier.	155
static int	**CUNEIFORM** **Deprecated.**	155
static int	**CYPRIOT** **Deprecated.**	155
static int	**CYRILLIC** Constant for cyrillic font identifier.	155
static int	**DESERET** **Deprecated.**	155
static int	**DEVANAGARI** Constant for devanagari font identifier.	155
static int	**ETHIOPIC** Constant for ethiopic font identifier.	156
static int	**GEORGIAN** Constant for georgian font identifier.	156
static int	**GLAGOLITIC** Constant for glagolitic font identifier.	156
static int	**GOTHIC** **Deprecated.**	156
static int	**GREEK** Constant for greek font identifier.	156
static int	**GUJARATI** Constant for gujarati font identifier.	156
static int	**GURMUKHI** Constant for gurmukhi font identifier.	157
static int	**HAN** Constant for han font identifier.	157
static int	**HANGUL** Constant for hangul font identifier.	157
static int	**HANUNOO** Constant for hanunoo font identifier.	157
static int	**HEBREW** Constant for hebrew font identifier.	157
static int	**HIRAGANA** Constant for hiragana font identifier.	157
static int	**INHERITED** Constant for inherited font identifier.	163
static int	**JAVANESE** Constant for javanese font identifier.	166
static int	**KANNADA** Constant for kannada font identifier.	158
static int	**KATAKANA** Constant for katakana font identifier.	158

static int	**KAYAH_LI** Constant for kayah li font identifier.	*164*
static int	**KHAROSHTHI** **Deprecated.**	*158*
static int	**KHMER** Constant for khmer font identifier.	*158*
static int	**LAO** Constant for lao font identifier.	*158*
static int	**LATIN** Constant for latin font identifier.	*158*
static int	**LEPCHA** Constant for lepcha font identifier.	*164*
static int	**LIMBU** Constant for limbu font identifier.	*159*
static int	**LISU** Constant for lisu font identifier.	*165*
static int	**MALAYALAM** Constant for malayalam font identifier.	*159*
static int	**MANDAIC** Constant for mandaic font identifier.	*166*
static int	**MEETEI_MAYEK** Constant for meetei mayek font identifier.	*166*
static int	**MONGOLIAN** Constant for mongolian font identifier.	*159*
static int	**MYANMAR** Constant for myanmar font identifier.	*159*
static int	**NEW_TAI_LUE** Constant for new tai lue font identifier.	*159*
static int	**NKO** Constant for nko font identifier.	*159*
static int	**OGHAM** Constant for ogham font identifier.	*160*
static int	**OL_CHIKI** Constant for ol chiki font identifier.	*164*
static int	**ORIYA** Constant for oriya font identifier.	*160*
static int	**OSMANYA** **Deprecated.**	*160*
static int	**PHAGS_PA** Constant for phags pa font identifier.	*160*
static int	**PHOENICIAN** **Deprecated.**	*160*
static int	**REJANG** Constant for rejang font identifier.	*165*
static int	**RUNIC** Constant for runic font identifier.	*161*
static int	**SAMARITAN** Constant for samaritan font identifier.	*165*
static int	**SAURASHTRA** Constant for saurashtra font identifier.	*164*

static int	**SHAVIAN** **Deprecated.**	161
static int	**SINHALA** Constant for sinhala font identifier.	161
static int	**STYLE_BOLD** The bold style constant.	152
static int	**STYLE_ITALIC** The italic style constant.	153
static int	**STYLE_PLAIN** The plain style constant.	152
static int	**STYLE_UNDERLINED** The underlined style constant.	153
static int	**SUNDANESE** Constant for sundanese font identifier.	164
static int	**SYLOTI_NAGRI** Constant for syloti nagri font identifier.	161
static int	**SYRIAC** Constant for syriac font identifier.	161
static int	**TAGALOG** Constant for tagalog font identifier.	161
static int	**TAGBANWA** Constant for tagbanwa font identifier.	162
static int	**TAI_LE** Constant for tai le font identifier.	162
static int	**TAI_THAM** Constant for tai tham font identifier.	165
static int	**TAI_VIET** Constant for tai viet font identifier.	165
static int	**TAMIL** Constant for tamil font identifier.	162
static int	**TELUGU** Constant for telugu font identifier.	162
static int	**THAANA** Constant for thaana font identifier.	162
static int	**THAI** Constant for thai font identifier.	162
static int	**TIBETAN** Constant for tibetan font identifier.	163
static int	**TIFINAGH** Constant for tifinagh font identifier.	163
static int	**UGARITIC** **Deprecated.**	163
static int	**VAI** Constant for vai font identifier.	164
static int	**YI** Constant for yi font identifier.	163

Constructor Summary		Page
	Font() Forbidden constructor.	166

Method Summary		Page
int	**charsWidth**(char[] ch, int offset, int length) Returns the width of the characters in ch from offset to offset+length with this font.	169
int	**charWidth**(char ch) Returns the width of the specified character with this font.	168
abstract String	**getDescriptor**() Returns the descriptor of the font or null if no descriptor is available.	167
abstract int[]	**getIdentifiers**() Returns an array of identifiers supported by the font.	167
int	**getStyle**() Returns the style of the font.	167
boolean	**isBold**() Returns true if the font is bold.	168
boolean	**isIdentifierSupported**(int identifier) Returns true if the font supports the given identifier.	167
boolean	**isItalic**() Returns true if the font is italic.	168
abstract boolean	**isMonospaced**() Returns true if the font is monospaced.	168
boolean	**isPlain**() Returns true if the font is plain.	168
boolean	**isUnderlined**() Returns true if the font is underlined.	168
int	**stringWidth**(String str) Returns the width of the string with this font.	169
int	**substringWidth**(String str, int offset, int len) Returns the width of the string from offset to offset+len with this font.	169
boolean	**supportIdentifiers**(int identifier) **Deprecated.** *use isIdentifierSupported(int)*	167

Field Detail

STYLE_PLAIN

```
public static final int STYLE_PLAIN
```

The plain style constant. It may be combined with other style constants.

Value 0 is assigned to STYLE_PLAIN.

STYLE_BOLD

```
public static final int STYLE_BOLD
```

The bold style constant. It may be combined with other style constants.

Value 1 is assigned to STYLE_BOLD.

STYLE_ITALIC

```
public static final int STYLE_ITALIC
```

The italic style constant. It may be combined with other style constants.

Value `2` is assigned to `STYLE_ITALIC`.

STYLE_UNDERLINED

```
public static final int STYLE_UNDERLINED
```

The underlined style constant. It may be combined with other style constants.

Value `4` is assigned to `STYLE_UNDERLINED`.

ARABIC

```
public static final int ARABIC
```

Constant for arabic font identifier.

Value `1` is assigned to `ARABIC`.

ARMENIAN

```
public static final int ARMENIAN
```

Constant for armenian font identifier.

Value `2` is assigned to `ARMENIAN`.

BALINESE

```
public static final int BALINESE
```

Constant for balinese font identifier.

Value `3` is assigned to `BALINESE`.

BENGALI

```
public static final int BENGALI
```

Constant for bengali font identifier.

Value `4` is assigned to `BENGALI`.

BOPOMOFO

```
public static final int BOPOMOFO
```

Constant for bopomofo font identifier.

Value `5` is assigned to `BOPOMOFO`.

BRAILLE

```
public static final int BRAILLE
```

Constant for braille font identifier.

Value `6` is assigned to `BRAILLE`.

BUGINESE

```
public static final int BUGINESE
```

Constant for buginese font identifier.

Value `7` is assigned to `BUGINESE`.

BUHID

```
public static final int BUHID
```

Constant for buhid font identifier.

Value `8` is assigned to `BUHID`.

CANADIAN_ABORIGINAL

```
public static final int CANADIAN_ABORIGINAL
```

Constant for canadian aboriginal font identifier.

Value `9` is assigned to `CANADIAN_ABORIGINAL`.

CHEROKEE

```
public static final int CHEROKEE
```

Constant for cherokee font identifier.

Value `10` is assigned to `CHEROKEE`.

COPTIC

```
public static final int COPTIC
```

Constant for coptic font identifier.

Value `11` is assigned to `COPTIC`.

CUNEIFORM

```
public static final int CUNEIFORM
```

Deprecated.

Constant for cuneiform font identifier.

Value `12` is assigned to `CUNEIFORM`.

CYPRIOT

```
public static final int CYPRIOT
```

Deprecated.

Constant for cypriot font identifier.

Value `13` is assigned to `CYPRIOT`.

CYRILLIC

```
public static final int CYRILLIC
```

Constant for cyrillic font identifier.

Value `14` is assigned to `CYRILLIC`.

DESERET

```
public static final int DESERET
```

Deprecated.

Constant for deseret font identifier.

Value `15` is assigned to `DESERET`.

DEVANAGARI

```
public static final int DEVANAGARI
```

Constant for devanagari font identifier.

Value `16` is assigned to `DEVANAGARI`.

ETHIOPIC

`public static final int` **`ETHIOPIC`**

Constant for ethiopic font identifier.

Value `17` is assigned to `ETHIOPIC`.

GEORGIAN

`public static final int` **`GEORGIAN`**

Constant for georgian font identifier.

Value `18` is assigned to `GEORGIAN`.

GLAGOLITIC

`public static final int` **`GLAGOLITIC`**

Constant for glagolitic font identifier.

Value `19` is assigned to `GLAGOLITIC`.

GOTHIC

`public static final int` **`GOTHIC`**

Deprecated.

Constant for gothic font identifier.

Value `20` is assigned to `GOTHIC`.

GREEK

`public static final int` **`GREEK`**

Constant for greek font identifier.

Value `21` is assigned to `GREEK`.

GUJARATI

`public static final int` **`GUJARATI`**

Constant for gujarati font identifier.

Value `22` is assigned to `GUJARATI`.

GURMUKHI

```
public static final int GURMUKHI
```

Constant for gurmukhi font identifier.

Value `23` is assigned to `GURMUKHI`.

HAN

```
public static final int HAN
```

Constant for han font identifier.

Value `24` is assigned to `HAN`.

HANGUL

```
public static final int HANGUL
```

Constant for hangul font identifier.

Value `25` is assigned to `HANGUL`.

HANUNOO

```
public static final int HANUNOO
```

Constant for hanunoo font identifier.

Value `26` is assigned to `HANUNOO`.

HEBREW

```
public static final int HEBREW
```

Constant for hebrew font identifier.

Value `27` is assigned to `HEBREW`.

HIRAGANA

```
public static final int HIRAGANA
```

Constant for hiragana font identifier.

Value 28 is assigned to HIRAGANA.

KANNADA

public static final int **KANNADA**

Constant for kannada font identifier.

Value 29 is assigned to KANNADA.

KATAKANA

public static final int **KATAKANA**

Constant for katakana font identifier.

Value 30 is assigned to KATAKANA.

KHAROSHTHI

public static final int **KHAROSHTHI**

Deprecated.

Constant for kharoshthi font identifier.

Value 31 is assigned to KHAROSHTHI.

KHMER

public static final int **KHMER**

Constant for khmer font identifier.

Value 32 is assigned to KHMER.

LAO

public static final int **LAO**

Constant for lao font identifier.

Value 33 is assigned to LAO.

LATIN

public static final int **LATIN**

Constant for latin font identifier.

Value `34` is assigned to `LATIN`.

LIMBU

```
public static final int LIMBU
```

Constant for limbu font identifier.

Value `35` is assigned to `LIMBU`.

MALAYALAM

```
public static final int MALAYALAM
```

Constant for malayalam font identifier.

Value `36` is assigned to `MALAYALAM`.

MONGOLIAN

```
public static final int MONGOLIAN
```

Constant for mongolian font identifier.

Value `37` is assigned to `MONGOLIAN`.

MYANMAR

```
public static final int MYANMAR
```

Constant for myanmar font identifier.

Value `38` is assigned to `MYANMAR`.

NEW_TAI_LUE

```
public static final int NEW_TAI_LUE
```

Constant for new tai lue font identifier.

Value `39` is assigned to `NEW_TAI_LUE`.

NKO

```
public static final int NKO
```

Constant for nko font identifier.

Value 40 is assigned to NKO.

OGHAM

```
public static final int OGHAM
```

Constant for ogham font identifier.

Value 41 is assigned to OGHAM.

ORIYA

```
public static final int ORIYA
```

Constant for oriya font identifier.

Value 42 is assigned to ORIYA.

OSMANYA

```
public static final int OSMANYA
```

Deprecated.

Constant for osmanya font identifier.

Value 43 is assigned to OSMANYA.

PHAGS_PA

```
public static final int PHAGS_PA
```

Constant for phags pa font identifier.

Value 44 is assigned to PHAGS_PA.

PHOENICIAN

```
public static final int PHOENICIAN
```

Deprecated.

Constant for phoenician font identifier.

Value 45 is assigned to PHOENICIAN.

RUNIC

```
public static final int RUNIC
```

Constant for runic font identifier.

Value `46` is assigned to `RUNIC`.

SHAVIAN

```
public static final int SHAVIAN
```

Deprecated.

Constant for shavian font identifier.

Value `47` is assigned to `SHAVIAN`.

SINHALA

```
public static final int SINHALA
```

Constant for sinhala font identifier.

Value `48` is assigned to `SINHALA`.

SYLOTI_NAGRI

```
public static final int SYLOTI_NAGRI
```

Constant for syloti nagri font identifier.

Value `49` is assigned to `SYLOTI_NAGRI`.

SYRIAC

```
public static final int SYRIAC
```

Constant for syriac font identifier.

Value `50` is assigned to `SYRIAC`.

TAGALOG

```
public static final int TAGALOG
```

Constant for tagalog font identifier.

Value `51` is assigned to `TAGALOG`.

TAGBANWA

```
public static final int TAGBANWA
```

Constant for tagbanwa font identifier.

Value `52` is assigned to `TAGBANWA`.

TAI_LE

```
public static final int TAI_LE
```

Constant for tai le font identifier.

Value `53` is assigned to `TAI_LE`.

TAMIL

```
public static final int TAMIL
```

Constant for tamil font identifier.

Value `54` is assigned to `TAMIL`.

TELUGU

```
public static final int TELUGU
```

Constant for telugu font identifier.

Value `55` is assigned to `TELUGU`.

THAANA

```
public static final int THAANA
```

Constant for thaana font identifier.

Value `56` is assigned to `THAANA`.

THAI

```
public static final int THAI
```

Constant for thai font identifier.

Value `57` is assigned to `THAI`.

TIBETAN

```
public static final int TIBETAN
```

Constant for tibetan font identifier.

Value `58` is assigned to `TIBETAN`.

TIFINAGH

```
public static final int TIFINAGH
```

Constant for tifinagh font identifier.

Value `59` is assigned to `TIFINAGH`.

UGARITIC

```
public static final int UGARITIC
```

Deprecated.

Constant for ugaritic font identifier.

Value `60` is assigned to `UGARITIC`.

YI

```
public static final int YI
```

Constant for yi font identifier.

Value `61` is assigned to `YI`.

COMMON

```
public static final int COMMON
```

Constant for common font identifier.

Value `62` is assigned to `COMMON`.

INHERITED

```
public static final int INHERITED
```

Constant for inherited font identifier.

Value `63` is assigned to `INHERITED`.

SUNDANESE

```
public static final int SUNDANESE
```

Constant for sundanese font identifier.

Value `64` is assigned to `SUNDANESE`.

LEPCHA

```
public static final int LEPCHA
```

Constant for lepcha font identifier.

Value `65` is assigned to `LEPCHA`.

OL_CHIKI

```
public static final int OL_CHIKI
```

Constant for ol chiki font identifier.

Value `66` is assigned to `OL_CHIKI`.

VAI

```
public static final int VAI
```

Constant for vai font identifier.

Value `67` is assigned to `VAI`.

SAURASHTRA

```
public static final int SAURASHTRA
```

Constant for saurashtra font identifier.

Value `68` is assigned to `SAURASHTRA`.

KAYAH_LI

```
public static final int KAYAH_LI
```

Constant for kayah li font identifier.

Value `69` is assigned to `KAYAH_LI`.

REJANG

```
public static final int REJANG
```

Constant for rejang font identifier.

Value `70` is assigned to `REJANG`.

CHAM

```
public static final int CHAM
```

Constant for cham font identifier.

Value `71` is assigned to `CHAM`.

TAI_THAM

```
public static final int TAI_THAM
```

Constant for tai tham font identifier.

Value `72` is assigned to `TAI_THAM`.

TAI_VIET

```
public static final int TAI_VIET
```

Constant for tai viet font identifier.

Value `73` is assigned to `TAI_VIET`.

SAMARITAN

```
public static final int SAMARITAN
```

Constant for samaritan font identifier.

Value `74` is assigned to `SAMARITAN`.

LISU

```
public static final int LISU
```

Constant for lisu font identifier.

Value `75` is assigned to `LISU`.

BAMUM

```
public static final int BAMUM
```

Constant for bamum font identifier.

Value `76` is assigned to `BAMUM`.

JAVANESE

```
public static final int JAVANESE
```

Constant for javanese font identifier.

Value `77` is assigned to `JAVANESE`.

MEETEI_MAYEK

```
public static final int MEETEI_MAYEK
```

Constant for meetei mayek font identifier.

Value `78` is assigned to `MEETEI_MAYEK`.

BATAK

```
public static final int BATAK
```

Constant for batak font identifier.

Value `79` is assigned to `BATAK`.

MANDAIC

```
public static final int MANDAIC
```

Constant for mandaic font identifier.

Value `80` is assigned to `MANDAIC`.

Constructor Detail

Font

```
Font()
```

Forbidden constructor.

See Also:

AlphaNumericDisplay.getDefaultFont(), AlphaNumericDisplay.getAllFonts(), DisplayFont.getDefaultFont(), DisplayFont.getAllFonts()

Method Detail

getIdentifiers

```
public abstract int[] getIdentifiers()
```

Returns an array of identifiers supported by the font. An identifier is an integer specified in this class or a specific integer defined by the MicroUI implementation.

Returns:
an array of identifier.

---

supportIdentifiers

```
public boolean supportIdentifiers(int identifier)
```

Deprecated. *use isIdentifierSupported(int)*

---

isIdentifierSupported

```
public boolean isIdentifierSupported(int identifier)
```

Returns true if the font supports the given identifier.

Parameters:
`identifier` - the wanted identifier.
Returns:
true if the font supports the given identifier.

---

getStyle

```
public int getStyle()
```

Returns the style of the font.
The returned value may only be a combination of the following style constants: `STYLE_BOLD`, `SYTLE_ITALIC`, `STYLE_UNDERLINED` or `STYLE_PLAIN`.

Returns:
the style of the font

---

getDescriptor

```
public abstract String getDescriptor()
```

Returns the descriptor of the font or null if no descriptor is available.

Returns:
the descriptor of the font

---

isPlain

```
public boolean isPlain()
```

Returns `true` if the font is plain.

Returns:
`true` if the font is plain

---

isBold

```
public boolean isBold()
```

Returns `true` if the font is bold.

Returns:
`true` if the font is bold

---

isItalic

```
public boolean isItalic()
```

Returns `true` if the font is italic.

Returns:
`true` if the font is italic

---

isUnderlined

```
public boolean isUnderlined()
```

Returns `true` if the font is underlined.

Returns:
`true` if the font is underlined

---

isMonospaced

```
public abstract boolean isMonospaced()
```

Returns `true` if the font is monospaced. A monospaced font is a font which all characters have the same width.

Returns:
`true` if the font is monospaced.

---

charWidth

```
public int charWidth(char ch)
```

Returns the width of the specified character with this font.

The width is the horizontal distance that would be occupied if `ch` was drawn using this font. It also includes the horizontal space that would be added after `ch` to separate it appropriately from the following characters.

Parameters:
`ch` - the character to measure
Returns:
the width of `ch` with this font

---

charsWidth

```
public int charsWidth(char[] ch,
                      int offset,
                      int length)
```

Returns the width of the characters in `ch` from `offset` to `offset+length` with this font.
The width is the horizontal distance that would be occupied if the `length` characters were drawn using this font. It also includes the horizontal spaces between characters to separate them appropriately.

Parameters:
`ch` - an array of characters
`offset` - the index of the first character to measure
`length` - the number of characters to measure
Returns:
the width taken by the specified characters in `ch`
Throws:
`ArrayIndexOutOfBoundsException` - if `offset` and `length` are out of `ch` range
`NullPointerException` - if `ch` is null

---

stringWidth

```
public int stringWidth(String str)
```

Returns the width of the string with this font.
The width is the horizontal distance that would be occupied if the string was drawn using this font. It also includes the horizontal spaces between characters to separate them appropriately.

Parameters:
`str` - the string to measure
Returns:
the width taken by `str`
Throws:
`NullPointerException` - if `str` is null

---

substringWidth

```
public int substringWidth(String str,
                          int offset,
                          int len)
```

Returns the width of the string from `offset` to `offset+len` with this font.
The width is the horizontal distance that would be occupied if the substring was drawn using this font. It also includes the horizontal spaces between characters to separate them appropriately.

Parameters:
`str` - the string to measure

`offset` - index of the first character in the substring
`len` - length of the substring

Returns:

the width taken by the substring of `str`

Throws:

`StringIndexOutOfBoundsException` - if `offset` and `length` are out of `str` range
`NullPointerException` - if `str` is null

Class GraphicsContext

ej.microui.io

```
java.lang.Object
  └
    ej.microui.io.GraphicsContext
```

Direct Known Subclasses:
ExplicitFlush

```
public class GraphicsContext
extends Object
```

The `GraphicsContext` class offers basic drawing facilities, to render lines, rectangles, polygons, arcs and text.

`GraphicsContext` uses 24-bit RGB color. Each color: red, green and blue is defined with an 8-bit value.
Not all displays may support such color depth. Therefore the implementation is in charge of mapping application colors to the most appropriate available colors.

A `GraphicsContext` object may be used either to

- paint on a display in the "normal" rendering procedure (using `paint(GraphicsContext)` methods), or
- draw on a mutable image, or
- directly draw on a display bypassing the "normal" rendering mechanism.

When a visible object (for instance a `Viewable` or a `View`) has to be painted on a `Display`, the `paint` method is given a `GraphicsContext` as argument and should use it to render the visible object.

A `GraphicsContext` may be requested for a mutable image. This graphics context can be used to draw in the image.

Direct drawing on a display can be done from application by retrieving a `GraphicsContext` with Display.getNewGraphicsContext() or Display.getNewExplicitFlush(). Using this mechanism does not ensure drawings will be performed before, during or after the current `paint()`, since it bypasses the serialization system events.

Drawing text relies on available fonts. Text can be drawn using drawChar, drawChars, drawString or drawSubstring. Characters are drawn with the current color of the `GraphicsContext` object.

The coordinate system is as follows:

- origin is at the upper left corner of the destination.
- X-axis is positive towards the right.
- Y-axis is positive downwards.

A coordinate does not map a pixel, but rather the location between pixels. For instance, the first pixel in the upper left corner matches a square of coordinates: `(0,0), (1,0), (1,1) and (0,1)`. The call (where g is a `GraphicsContext`) `g.fillRect(1,0,2,3)` paints six pixels.

Two different stroke styles may be used when drawing lines, arcs or rectangles: either `SOLID` or `DOTTED`. Stroke style has no effect on fill, text and image handling.

The `SOLID` stroke style allows drawing with a one-pixel wide pen. Drawing at a specific coordinate fills the adjacent down-right pixel. For instance, although the next line has a width of 1, `g.drawLine(0,0,1,0)` draws 2 pixels: the upper-left corner of the display and its adjacent right pixel.

The `DOTTED` stroke style allows drawing a subset of the pixels that would have been drawn with the `SOLID` stroke style. Length and frequency of dots is implementation dependent and, as a result, so are the drawn pixels. Note that end of lines or end of arcs, as well as the corner of rectangles may not be drawn with the `DOTTED` stroke style.

One important remark has to be made about rectangle drawing and filling. Drawing a rectangle with the code:
```
drawRect(x,y,w,h);
```
is equivalent to the following code sequence:
```
drawLine(x,y,x+w,y);
drawLine(x+w,y,x+w,y+h);
drawLine(x+w,y+h,x,y+h);
drawLine(x,y+h,x,y);
```

In addition, the following code:
```
fillRect(x,y,w,h);
```
results in filling rectangle area which differs from the rectangle drawn by `drawRect(x,y,w,h)`. Indeed, the filled area counts `w*h` pixels, whereas the area delimited by `drawRect(x,y,w,h)` counts `(w+1)*(h+1)` pixels.
A filled area must overlap exactly or be contiguous to its matching drawn area. That is to say that there must be no blank space between a filled area and its matching drawn area and that the filled area must not be out of the bounds of the drawn area.

Note that the exact number of pixels drawn by `drawLine()` and `drawArc()` are implementation dependent.

A `GraphicsContext` defines a clipping zone which specifies the destination area that can be modified by calls to the `GraphicsContext`. The clipping zone can be set by the application but is more commonly set by the UI framework. The clipping zone may be empty (i.e. its size is zero), in that case, every rendering operation will have no effect. It may also be out of the bounds of the destination, in which case every rendering operation out of the range of the destination is ignored. Modification of the coordinate system (with the method `translate` for instance) has no effect on the clipping zone.

When positioning a visible object (text or image for instance) into a drawable area, a coordinate `(x,y)` location or anchor point is used. In addition it is possible to express how the object is set around the anchor point. Several constants have been thus defined; they can be combined bit-wise to precisely define how the object is set around the anchor point. For instance,

```
g.drawString("test",x,y,TOP|LEFT);
```

draws a string and defines `(x,y)` as the upper left point of the text zone.

```
g.drawString("test",x,y,TOP|HCENTER);
```

will draw string `"test"` above and centered on `(x,y)`.

Note that any anchor constants combination must be limited to one of the horizontal constants (`LEFT`, `HCENTER`, `RIGHT`) and one of the vertical constants (`TOP`, `BASELINE` for text positioning exclusively, `VCENTER`, `BOTTOM`). The default anchor position, obtained with value `0`, matches the `TOP | LEFT` constant combination.

Field Summary		Page
static int	**AND** Constant for the `AND` operator on the filter.	178
static int	**BASELINE** Constant for positioning the baseline of the text at the anchor point.	176
static int	**BOTTOM** Constant for positioning the bottom of the drawing at the anchor point.	176
static int	**DOTTED** Constant for the `DOTTED` stroke style.	177
static int	**HCENTER** Constant for centering drawing horizontally around the anchor point.	175
static int	**INV_COLOR** Constant for the `INV_COLOR` operator mask on the filter.	178
static int	**INV_RESULT** Constant for the `INV_RESULT` operator mask on the filter.	178
static int	**LEFT** Constant for positioning the left side of the drawing at the anchor point.	176

static int	**MINUS** Constant for the MINUS operator on the filter.	*177*
static int	**OR** Constant for the OR operator on the filter.	*178*
static int	**PLUS** Constant for the PLUS operator on the filter.	*177*
static int	**RESET_FILTER** Constant to apply any filter on the color.	*177*
static int	**RIGHT** Constant for positioning the right side of the drawing at the anchor point.	*176*
static int	**SOLID** Constant for the SOLID stroke style.	*177*
static int	**TOP** Constant for positioning the top of the drawing at the anchor point.	*176*
static int	**VCENTER** Constant for centering the drawing vertically around the anchor point.	*176*
static int	**XOR** Constant for the XOR operator on the filter.	*178*

Constructor Summary		Page
	GraphicsContext() Forbidden constructor: use Display.getNewGraphicsContext() to get an instance of GraphicsContext.	*179*

Method Summary		Page
void	**clipRect**(int x, int y, int width, int height) Sets the clipping area to be the intersection of the specified rectangle with the current clipping rectangle.	*182*
void	**copyArea**(int x_src, int y_src, int width, int height, int x_dest, int y_dest, int anchor) Copies an area within a GraphicsContext.	*192*
void	**drawArc**(int x, int y, int width, int height, int startAngle, int arcAngle) Draws the outline of a circular or elliptical arc covering the specified rectangle, using the current color and stroke style.	*187*
void	**drawARGB**(int[] argbData, int offset, int scanlength, int x, int y, int width, int height, int anchor) Gets ARGB pixel data from the provided array of integers and draws it in the specified region of this graphics context.	*193*
void	**drawChar**(char character, int x, int y, int anchor) Draws a character using the current font and color.	*194*
void	**drawChars**(char[] data, int offset, int length, int x, int y, int anchor) Draws some characters using the current font and color.	*195*
void	**drawCircle**(int x, int y, int diameter) Draws the outline of a circle covering the rectangle specified by its diameter, using the current color and stroke style.	*189*
void	**drawDeformedImage**(Image img, int x, int y, int[] xys, int anchor) Draws a deformed image at the given anchor point.	*190*
void	**drawEllipse**(int x, int y, int width, int height) Draws the outline of a ellipse covering the specified rectangle, using the current color and stroke style.	*189*

void	**drawHorizontalLine**(int x, int y, int width) Draws an horizontal line from (x,y) to (x+width,y) using the current color and stroke style.	184
void	**drawImage**(Image img, int x, int y, int anchor) Draws an image at the given anchor point.	190
void	**drawLine**(int x1, int y1, int x2, int y2) Draws a line from (x1,y1) to (x2,y2) using the current color and stroke style.	185
void	**drawPixel**(int x, int y) Draws a pixel at (x,y) using the current color.	183
void	**drawPolygon**(int[] xys) Draws the closed polygon which is defined by the array of integer coordinates, using the current color and stroke style.	186
void	**drawPolygon**(int[] xys, int offset, int length) Draws the closed polygon which is defined by the array of integer coordinates, using the current color and stroke style.	186
void	**drawRect**(int x, int y, int width, int height) Draws the outline of the specified rectangle using the current color and stroke style.	185
void	**drawRegion**(Image src, int x_src, int y_src, int width, int height, int x_dest, int y_dest, int anchor) Draws the specified region of an image.	191
void	**drawRoundRect**(int x, int y, int width, int height, int arcWidth, int arcHeight) Draws the outline of the specified rounded corner rectangle using the current color and stroke style.	185
void	**drawString**(String str, int x, int y, int anchor) Draws the string using the current font and color.	194
void	**drawSubstring**(String str, int offset, int len, int x, int y, int anchor) Draws the string from offset to offset+length using the current font and color.	194
void	**drawVerticalLine**(int x, int y, int height) Draws a vertical line from (x,y) to (x,y+height-1) using the current color and stroke style.	184
void	**fillArc**(int x, int y, int width, int height, int startAngle, int arcAngle) Fills a circular or elliptical arc covering the specified rectangle with the current color.	188
void	**fillCircle**(int x, int y, int diameter) Fills a circle covering the rectangle specified by its diameter with the current color.	189
void	**fillEllipse**(int x, int y, int width, int height) Fills a ellipse covering the specified rectangle with the current color.	190
void	**fillPolygon**(int[] xys) Fills the closed polygon which is defined by the array of integer coordinates, using the current color.	187
void	**fillPolygon**(int[] xys, int offset, int length) Fills the closed polygon which is defined by the array of integer coordinates, using the current color.	187
void	**fillRect**(int x, int y, int width, int height) Fills the specified rectangle with the current color.	185
void	**fillRoundRect**(int x, int y, int width, int height, int arcWidth, int arcHeight) Fills the specified rounded corner rectangle with the current color.	186
void	**getARGB**(int[] argbData, int offset, int scanlength, int x, int y, int width, int height) Obtains ARGB pixel data from the specified region of this graphics context and stores it in the provided array of integers.	192
int	**getClipHeight**() Returns the height of the current clipping zone.	183
int	**getClipWidth**() Returns the width of the current clipping zone.	183

int	**getClipX**() Returns the x offset of the current clipping zone, relative to the graphics context's origin.	*183*
int	**getClipY**() Returns the y offset of the current clipping zone, relative to graphics context's origin.	*183*
int	**getColor**() Returns the current color: a 24-bits value interpreted as: `0xRRGGBB`, that is, the eight least significant bits give the blue color, the next eight bits the green value and the next eight bits the red color.	*181*
Display	**getDisplay**() Returns the display associated with the GraphicsContext.	*195*
int	**getDisplayColor**(int color) Gets the color that will be displayed if the specified color is requested.	*181*
boolean	**getEllipsis**() Returns `true` if the truncation mechanism is enabled.	*196*
int	**getFilter**(int rgbFilter, int factor) Gets a filter from the given filter and the given factor.	*181*
DisplayFont	**getFont**() Returns the current font.	*182*
int	**getStrokeStyle**() Returns the current stroke style.	*182*
int	**getTranslateX**() Returns the x coordinate of the translated origin of the graphics context.	*179*
int	**getTranslateY**() Returns the y coordinate of the translated origin of the graphics context.	*179*
int	**readPixel**(int x, int y) Obtains the RGB color of the pixel at `(x,y)`.	*184*
void	**setClip**(int x, int y, int width, int height) Sets the current clipping zone to the rectangle defined by the given location `(x,y)` and size `(width,height)`.	*182*
void	**setColor**(int rgbColor) Sets the current color.	*180*
void	**setEllipsis**(boolean enable) Enables (disables) truncation when rendering characters.	*195*
void	**setFilter**(int rgbFilter, int operator) Sets the current filter.	*180*
void	**setFont**(DisplayFont font) Sets the font for subsequent text operations.	*182*
void	**setStrokeStyle**(int style) Sets the stroke style of the `GraphicsContext` used for drawing lines, arcs and rectangles.	*181*
void	**translate**(int x, int y) Translates the `GraphicsContext` origin with the given vector `(x,y)`.	*179*

Field Detail

HCENTER

```
public static final int HCENTER
```

Constant for centering drawing horizontally around the anchor point.

Value `1` is assigned to `HCENTER`.

VCENTER

```
public static final int VCENTER
```

Constant for centering the drawing vertically around the anchor point.

Value `2` is assigned to `VCENTER`.

LEFT

```
public static final int LEFT
```

Constant for positioning the left side of the drawing at the anchor point.

Value `4` is assigned to `LEFT`.

RIGHT

```
public static final int RIGHT
```

Constant for positioning the right side of the drawing at the anchor point.

Value `8` is assigned to `RIGHT`.

TOP

```
public static final int TOP
```

Constant for positioning the top of the drawing at the anchor point.

Value `16` is assigned to `TOP`.

BOTTOM

```
public static final int BOTTOM
```

Constant for positioning the bottom of the drawing at the anchor point.

Value `32` is assigned to `BOTTOM`.

BASELINE

```
public static final int BASELINE
```

Constant for positioning the baseline of the text at the anchor point.

Value `64` is assigned to `BASELINE`.

SOLID

```
public static final int SOLID
```

Constant for the `SOLID` stroke style.

Value `0` is assigned to `SOLID`.

DOTTED

```
public static final int DOTTED
```

Constant for the `DOTTED` stroke style.

Value `1` is assigned to `DOTTED`.

RESET_FILTER

```
public static final int RESET_FILTER
```

Constant to apply any filter on the color.

Value `0` is assigned to `RESET_FILTER`.

See Also:
setFilter(int, int)

PLUS

```
public static final int PLUS
```

Constant for the `PLUS` operator on the filter.

Value `1` is assigned to `PLUS`.

See Also:
setFilter(int, int)

MINUS

```
public static final int MINUS
```

Constant for the `MINUS` operator on the filter.

Value `2` is assigned to `MINUS`.

See Also:
setFilter(int, int)

OR

```
public static final int OR
```

Constant for the `OR` operator on the filter.

Value `3` is assigned to `OR`.

See Also:
setFilter(int, int)

AND

```
public static final int AND
```

Constant for the `AND` operator on the filter.

Value `4` is assigned to `AND`.

See Also:
setFilter(int, int)

XOR

```
public static final int XOR
```

Constant for the `XOR` operator on the filter.

Value `5` is assigned to `XOR`.

See Also:
setFilter(int, int)

INV_COLOR

```
public static final int INV_COLOR
```

Constant for the `INV_COLOR` operator mask on the filter.

Value `0x10` is assigned to `INV_COLOR`.

See Also:
setFilter(int, int)

INV_RESULT

```
public static final int INV_RESULT
```

Constant for the `INV_RESULT` operator mask on the filter.

Value `0x20` is assigned to `INV_RESULT`.

See Also:
setFilter(int, int)

Constructor Detail

GraphicsContext

```
GraphicsContext()
```

Forbidden constructor: use Display.getNewGraphicsContext() to get an instance of `GraphicsContext`.

See Also:
Display.getNewGraphicsContext(), ComponentView.paint(GraphicsContext)

Method Detail

translate

```
public final void translate(int x,
                            int y)
```

Translates the `GraphicsContext` origin with the given vector `(x,y)`. Subsequent rendering operations on the graphics context will be relative to the new origin.

This method can be used to set an absolute origin to a `GraphicsContext`. For instance, the following code:
`g.translate(ax-g.getTranslateX(),ay-g.getTranslateY());`
will set the origin of `g` at `(ax,ay)`.

Parameters:
`x` - the translation for the x coordinate
`y` - the translation for the y coordinate

---

getTranslateX

```
public final int getTranslateX()
```

Returns the x coordinate of the translated origin of the graphics context.

Returns:
x coordinate of the translated origin

---

getTranslateY

```
public final int getTranslateY()
```

Returns the y coordinate of the translated origin of the graphics context.

Returns:
y coordinate of the translated origin

---

setColor

```
public final void setColor(int rgbColor)
```

Sets the current color.
Given value `rgbColor` is interpreted as a 24-bit RGB color, where the eight least significant bits matches the blue component, the next eight more significant bits matches the green component and the next eight more significant bits matches the red component.

Parameters:
`rgbColor` - the color to set

---

setFilter

```
public final void setFilter(int rgbFilter,
                            int operator)
```

Sets the current filter.
Given value `rgbFilter` is interpreted as a 24-bit RGB, where the eight least significant bits matches the blue component, the next eight more significant bits matches the green component and the next eight more significant bits matches the red component.
The filter is applied on each pixel drawn.
The operator is a combinaison between a binary operator and two unary operators:

- binary operators are:
  - PLUS: perform a saturated addition between each color component (R, G, and B) and each corresponded filter component. If the result is higher than 0xff, the component is 0xff. The final color becomes so lighter.
  - MINUS: perform a saturated substraction between each color component (R, G, and B) and each corresponded filter component. If the result is lower than 0x00, the component is 0x00. The final color becomes so darker
  - OR: perform an 'OR' logical operation on each color component (R, G, or B) and each corresponded filter component.
  - AND: perform an 'AND' logical operation on each color component (R, G, or B) and each corresponded filter component.
  - XOR: perform an 'XOR' logical operation on each color component (R, G, or B) and each corresponded filter component.

  - unary operators are:
    - INV_COLOR: invert the source color before performing the filter operatorion.
    - INV_RESULT: perform the filter operation and invert the resulted color.

Examples:

- AND | INV_COLOR: invert the source color and perform an 'AND' between the color and the filter.
- OR | INV_COLOR | INV_RESULT: invert the source color, perform an 'OR' between the color and the filter and invert the result.
- INV_COLOR: invert only the source color.
- INV_RESULT: idem as INV_COLOR.
- RESET_FILTER: remove the current filter, in this case, rgbFilter parameter is so useless.

A new filter erase previous one.

Parameters:
`rgbFilter` - the filter to set
`operator` - the operator between the color and the filter

---

getFilter

```
public final int getFilter(int rgbFilter,
                           int factor)
```

Gets a filter from the given filter and the given factor.
Given value `rgbFilter` is interpreted as a 24-bit RGB, where the eight least significant bits matches the blue component, the next eight more significant bits matches the green component and the next eight more significant bits matches the red component.
Each component of this filter is multiplied by the factor and the result is saturate to 0xff.

Parameters:
`rgbFilter` - the source filter
`factor` - the factor to apply to the filter, only the low significant byte is used.
Returns:
the saturated filter

---

getColor

```
public final int getColor()
```

Returns the current color: a 24-bits value interpreted as: `0xRRGGBB`, that is, the eight least significant bits give the blue color, the next eight bits the green value and the next eight bits the red color.

Returns:
current color

---

getDisplayColor

```
public final int getDisplayColor(int color)
```

Gets the color that will be displayed if the specified color is requested.
For example, with a monochrome display, this method will return either 0xFFFFFF (white) or 0x000000 (black) depending on the brightness of the specified color.

Parameters:
`color` - the desired color in 0x00RRGGBB format.
Returns:
the corresponding color that will be displayed on the graphics context (in 0x00RRGGBB format).

---

setStrokeStyle

```
public final void setStrokeStyle(int style)
```

Sets the stroke style of the `GraphicsContext` used for drawing lines, arcs and rectangles.

Parameters:
`style` - either `SOLID` or `DOTTED`
Throws:
`IllegalArgumentException` - if the style is not valid

---

getStrokeStyle

```
public final int getStrokeStyle()
```

Returns the current stroke style.

Returns:
stroke style, `SOLID` or `DOTTED`

---

setFont

```
public final void setFont(DisplayFont font)
```

Sets the font for subsequent text operations. If given font is `null`, the `GraphicsContext`'s font is set to `DisplayFont.getDefaultFont()`.

Parameters:
`font` - the new font to use

---

getFont

```
public final DisplayFont getFont()
```

Returns the current font.

Returns:
current font

---

clipRect

```
public final void clipRect(int x,
                           int y,
                           int width,
                           int height)
```

Sets the clipping area to be the intersection of the specified rectangle with the current clipping rectangle. It is legal to specify a clip rectangle whose width or height is zero or negative. In this case the clip is considered to be empty, that is, no pixels are contained within it. Therefore, if any graphics operations are issued under such a clip, no pixels will be modified.

Parameters:
`x` - the x coordinate of the rectangle
`y` - the y coordinate of the rectangle
`width` - the width of the rectangle
`height` - the height of the rectangle

---

setClip

```
public final void setClip(int x,
                          int y,
                          int width,
                          int height)
```

Sets the current clipping zone to the rectangle defined by the given location `(x,y)` and size `(width,height)`. Given width or height may be zero or negative, in that case the clip is considered to be empty, i.e. it contains no pixels. Nothing is done when drawing in an empty clip. Rendering operations have no effect outside of the clipping area.

Parameters:

`x` - the x coordinate of the new clip rectangle
`y` - the y coordinate of the new clip rectangle
`width` - the width of the new clip rectangle
`height` - the height of the new clip rectangle

---

getClipX

```
public final int getClipX()
```

Returns the x offset of the current clipping zone, relative to the graphics context's origin.

Returns:

x offset of the current clipping zone

---

getClipY

```
public final int getClipY()
```

Returns the y offset of the current clipping zone, relative to graphics context's origin.

Returns:

y offset of the current clipping zone

---

getClipWidth

```
public final int getClipWidth()
```

Returns the width of the current clipping zone.

Returns:

width of the current clipping zone

---

getClipHeight

```
public final int getClipHeight()
```

Returns the height of the current clipping zone.

Returns:

height of the current clipping zone.

---

drawPixel

```
public final void drawPixel(int x,
                            int y)
```

Draws a pixel at `(x,y)` using the current color.

Parameters:
`x` - the x coordinate of the pixel
`y` - the y coordinate of the pixel

---

readPixel

```
public final int readPixel(int x,
                           int y)
```

Obtains the RGB color of the pixel at `(x,y)`. The read color may be different than the drawing color. It is screen dependent, according to the number of bits per pixels (see Display.getBPP()).

Parameters:
`x` - the x coordinate of the pixel
`y` - the y coordinate of the pixel
Returns:
the rgb color of the pixel

---

drawHorizontalLine

```
public final void drawHorizontalLine(int x,
                                     int y,
                                     int width)
```

Draws an horizontal line from `(x,y)` to `(x+width,y)` using the current color and stroke style. The drawn line counts `(width+1)` pixels.
If `width` is negative, nothing is drawn.

Parameters:
`x` - the x coordinate of the start of the line
`y` - the y coordinate of the start of the line
`width` - the width of the horizontal line to draw

---

drawVerticalLine

```
public final void drawVerticalLine(int x,
                                   int y,
                                   int height)
```

Draws a vertical line from `(x,y)` to `(x,y+height-1)` using the current color and stroke style. The drawn line counts `(height+1)` pixels.
If `height` is negative, nothing is drawn.

Parameters:
`x` - the x coordinate of the start of the line
`y` - the y coordinate of the start of the line
`height` - the width of the vertical line to draw

---

drawLine

```
public final void drawLine(int x1,
                           int y1,
                           int x2,
                           int y2)
```

Draws a line from `(x1,y1)` to `(x2,y2)` using the current color and stroke style.

Parameters:

`x1` - the x coordinate of the start of the line
`y1` - the y coordinate of the start of the line
`x2` - the x coordinate of the end of the line
`y2` - the y coordinate of the end of the line

---

drawRect

```
public final void drawRect(int x,
                           int y,
                           int width,
                           int height)
```

Draws the outline of the specified rectangle using the current color and stroke style. The drawn rectangle includes `(width+1)*(height+1)` pixels.
If either `width` or `height` is negative, nothing is drawn.

Parameters:

`x` - the x coordinate of the rectangle to draw
`y` - the y coordinate of the rectangle to draw
`width` - the width of the rectangle to draw
`height` - the height of the rectangle to draw

---

fillRect

```
public final void fillRect(int x,
                           int y,
                           int width,
                           int height)
```

Fills the specified rectangle with the current color. If either `width` or `height` is negative or zero, nothing is drawn.

Parameters:

`x` - the x coordinate of the rectangle to be filled
`y` - the y coordinate of the rectangle to be filled
`width` - the width of the rectangle to be filled
`height` - the height of the rectangle to be filled

---

drawRoundRect

```
public final void drawRoundRect(int x,
                                int y,
                                int width,
                                int height,
                                int arcWidth,
                                int arcHeight)
```

Draws the outline of the specified rounded corner rectangle using the current color and stroke style. Drawn rectangle is `width+1`-pixel wide and `height+1`-pixel high. If either `width` or `height` is negative, nothing is drawn.

Parameters:

`x` - the x coordinate of the rectangle to draw
`y` - the y coordinate of the rectangle to draw
`width` - the width of the rectangle to draw
`height` - the height of the rectangle to draw
`arcWidth` - the horizontal diameter of the arc at the corners
`arcHeight` - the vertical diameter of the arc at the corners

---

fillRoundRect

```
public final void fillRoundRect(int x,
                                int y,
                                int width,
                                int height,
                                int arcWidth,
                                int arcHeight)
```

Fills the specified rounded corner rectangle with the current color. If either `width` or `height` is negative or zero, nothing is drawn.

Parameters:

`x` - the x coordinate of the rectangle to fill
`y` - the y coordinate of the rectangle to fill
`width` - the width of the rectangle to fill
`height` - the height of the rectangle to fill
`arcWidth` - the horizontal diameter of the arc at the corners
`arcHeight` - the vertical diameter of the arc at the corners

---

drawPolygon

```
public final void drawPolygon(int[] xys)
```

Draws the closed polygon which is defined by the array of integer coordinates, using the current color and stroke style. Lines are drawn between each consecutive pair, and between the first pair and last pair in the array. The effect is identical to
`drawPolygon(xys,0,xys.length);`

Parameters:

`xys` - the array of coordinates : x1,y1,.....xn,yn.

Throws:

`NullPointerException` - if the `xys` array is null.
`IllegalArgumentException` - if the `xys` length is odd.

---

drawPolygon

```
public final void drawPolygon(int[] xys,
                              int offset,
                              int length)
```

Draws the closed polygon which is defined by the array of integer coordinates, using the current color and stroke style. Lines are drawn between each consecutive pair, and between the first pair and last pair in the array.

Parameters:

`xys` - the array of coordinates : x1,y1,.....xn,yn.
`offset` - the `x1` index in `xys`.
`length` - the number of coordinates, must be even.

Throws:

`NullPointerException` - if the `xys` array is null.
`IllegalArgumentException` - if the `xys` length is odd.
`ArrayIndexOutOfBoundsException` - the wanted data is outside the array bounds.

---

fillPolygon

```
public final void fillPolygon(int[] xys)
```

Fills the closed polygon which is defined by the array of integer coordinates, using the current color. Lines are drawn between each consecutive pair, and between the first pair and last pair in the array. The lines connecting each pair of points are included in the filled polygon. The effect is identical to
`fillPolygon(xys,0,xys.length);`

Parameters:

`xys` - the array of coordinates : x1,y1,.....xn,yn.

Throws:

`NullPointerException` - if the `xys` array is null.
`IllegalArgumentException` - if the `xys` length is odd.

---

fillPolygon

```
public final void fillPolygon(int[] xys,
                              int offset,
                              int length)
```

Fills the closed polygon which is defined by the array of integer coordinates, using the current color. Lines are drawn between each consecutive pair, and between the first pair and last pair in the array. The lines connecting each pair of points are included in the filled polygon. The effect is identical to
`fillPolygon(xys,0,xys.length);`

Parameters:

`xys` - the array of coordinates : x1,y1,.....xn,yn.
`offset` - the `x1` index in `xys`.
`length` - the number of coordinates, must be even.

Throws:

`ArrayIndexOutOfBoundsException` - if `offset` and `length` do not specify a valid range within `xys`
`NullPointerException` - if the `xys` array is null.
`IllegalArgumentException` - if the `xys` length is odd.

---

drawArc

```
public final void drawArc(int x,
                          int y,
                          int width,
                          int height,
                          int startAngle,
                          int arcAngle)
```

Draws the outline of a circular or elliptical arc covering the specified rectangle, using the current color and stroke style.

The arc is drawn from `startAngle` up to `arcAngle` degrees. The center of the arc is defined as the center of the rectangle whose origin is at `(x,y)` (upper-left corner) and whose dimension is given by `width` and `height`.

Angles are interpreted such that 0 degrees is at the 3 o'clock position. A positive value indicates a counter-clockwise rotation while a negative value indicates a clockwise rotation.

If either `width` or `height` is negative, nothing is drawn.

The angles are given relative to the rectangle. For instance an angle of 45 degrees is always defined by the line from the center of the rectangle to the upper right corner of the rectangle. Thus for a non squarred rectangle angles are skewed along either height or width.

Parameters:

`x` - the x coordinate of the upper-left corner of the rectangle where the arc is drawn
`y` - the y coordinate of the upper-left corner of the rectangle where the arc is drawn
`width` - the width of the arc to draw
`height` - the height of the arc to draw
`startAngle` - the beginning angle of the arc to draw
`arcAngle` - the angular extent of the arc from `startAngle`

---

fillArc

```
public final void fillArc(int x,
                          int y,
                          int width,
                          int height,
                          int startAngle,
                          int arcAngle)
```

Fills a circular or elliptical arc covering the specified rectangle with the current color.

The arc is drawn from `startAngle` up to `arcAngle` degrees. The center of the arc is defined as the center of the rectangle whose origin is at `(x,y)` (upper-left corner) and whose dimension is given by `width` and `height`.

Angles are interpreted such that 0 degrees is at the 3 o'clock position. A positive value indicates a counter-clockwise rotation while a negative value indicates a clockwise rotation.

This method fills the area bounded from the center of the arc to the arc itself.

If either `width` or `height` is negative, nothing is drawn.

The angles are given relatively to the rectangle. That is to say that the angle of 45 degrees is always defined by the line from the center of the rectangle to the upper-right corner of the rectangle. Thus for a non squarred rectangle angles are skewed along either height or width.

Parameters:

`x` - the x coordinate of the upper-left corner of the rectangle where the arc is filled.
`y` - the y coordinate of the upper-left corner of the rectangle where the arc is filled.
`width` - the width of the arc to fill
`height` - the height of the arc to fill
`startAngle` - the beginning angle of the arc to draw
`arcAngle` - the angular extent of the arc from `startAngle`

---

drawCircle

```
public final void drawCircle(int x,
                             int y,
                             int diameter)
```

Draws the outline of a circle covering the rectangle specified by its diameter, using the current color and stroke style.

The center of the circle is defined as the center of the rectangle whose origin is at `(x,y)` (upper-left corner) and whose dimension is given by `diameter`.

If `diameter` is negative, nothing is drawn.

Parameters:

`x` - the x coordinate of the upper-left corner of the rectangle where the circle is drawn
`y` - the y coordinate of the upper-left corner of the rectangle where the circle is drawn
`diameter` - the diameter of the circle to draw

---

fillCircle

```
public final void fillCircle(int x,
                             int y,
                             int diameter)
```

Fills a circle covering the rectangle specified by its diameter with the current color.

The center of the circle is defined as the center of the rectangle whose origin is at `(x,y)` (upper-left corner) and whose dimension is given by `diameter`.

If `diameter` is negative, nothing is drawn.

Parameters:

`x` - the x coordinate of the upper-left corner of the rectangle where the circle is filled.
`y` - the y coordinate of the upper-left corner of the rectangle where the circle is filled.
`diameter` - the diameter of the circle to fill

---

drawEllipse

```
public final void drawEllipse(int x,
                              int y,
                              int width,
                              int height)
```

Draws the outline of a ellipse covering the specified rectangle, using the current color and stroke style.

The center of the ellipse is defined as the center of the rectangle whose origin is at `(x,y)` (upper-left corner) and whose dimension is given by `width` and `height`.

If either `width` or `height` is negative, nothing is drawn.

Parameters:

`x` - the x coordinate of the upper-left corner of the rectangle where the ellipse is drawn
`y` - the y coordinate of the upper-left corner of the rectangle where the ellipse is drawn
`width` - the width of the ellipse to draw

`height` - the height of the ellipse to draw

---

fillEllipse

```
public final void fillEllipse(int x,
                              int y,
                              int width,
                              int height)
```

Fills a ellipse covering the specified rectangle with the current color.

The center of the ellipse is defined as the center of the rectangle whose origin is at `(x,y)` (upper-left corner) and whose dimension is given by `width` and `height`.

If either `width` or `height` is negative, nothing is drawn.

Parameters:

`x` - the x coordinate of the upper-left corner of the rectangle where the ellipse is filled.
`y` - the y coordinate of the upper-left corner of the rectangle where the ellipse is filled.
`width` - the width of the ellipse to fill
`height` - the height of the ellipse to fill

---

drawImage

```
public final void drawImage(Image img,
                            int x,
                            int y,
                            int anchor)
```

Draws an image at the given anchor point.
The image anchor point is at position `(x,y)`. Position constants may be given to specify the precise location of the image around the anchor point.

Parameters:

`img` - the image to draw
`x` - the x coordinate of the anchor point
`y` - the y coordinate of the anchor point
`anchor` - position of the image around the anchor point

Throws:

`IllegalArgumentException` - if `anchor` is not a valid value (`BASELINE` is illegal)., if `img` and this `GraphicsContext` target different displays
`NullPointerException` - if `img` is `null`

---

drawDeformedImage

```
public final void drawDeformedImage(Image img,
                                    int x,
                                    int y,
                                    int[] xys,
                                    int anchor)
```

Draws a deformed image at the given anchor point.
The image anchor point is at position `(x,y)`. Position constants may be given to specify the precise location of the image around the anchor point.

The deformed image is identified by its four corner points. These points are defined by the array of integer coordinates and they must respect the following order: first is the top-left corner, second is the top-right, third is the bottom-right and fourth is the bottom-left.

Examples with img an image and imgWidth and imgHeight its size.

- To draw normal img, the array should be : {0,0, imgWidth-1,0, imgWidth-1,imgHeight-1, 0,imgHeight-1}.
- To draw img with a rotation clockwise by 90 degrees, the array should be : {imgHeight-1,0, imgHeight-1,imgWidth-1, imgWidth-1,0, 0,0}.
- To draw img mirrored about the vertical axis, the array should be : {0,0, -(imgWidth-1),0, -(imgWidth-1),-(imgHeight-1), 0,-(imgHeight-1)}.
- To draw img with a double scale, the array should be : {0,0, (imgWidth-1)*2,0, (imgWidth-1)*2, (imgHeight-1)*2, 0,(imgHeight-1)*2}.

Parameters:

`img` - the image to draw
`x` - the x coordinate of the anchor point
`y` - the y coordinate of the anchor point
`xys` - the array of coordinates : x1,y1,x2,y2,x3,y3,x4,y4.
`anchor` - position of the image around the anchor point

Throws:

`NullPointerException` - if `img` is `null`,
if the `xys` array is null.
`IllegalArgumentException` - if the `xys` length is different than 2*4.,
if `img` and this `GraphicsContext` target different displays

---

drawRegion

```
public final void drawRegion(Image src,
                             int x_src,
                             int y_src,
                             int width,
                             int height,
                             int x_dest,
                             int y_dest,
                             int anchor)
```

Draws the specified region of an image. The region in `src` is given relative to the image (origin at the upper-left corner) as a rectangle whose origin is at `(x_src,y_src)` and whose dimension is given by `width` and `height`.

The image region anchor point in destination is at the relative position `(x_dest,y_dest)`. Position constants may be given to specify the precise location of the image around the anchor point.

If the specified source region exceeds the image bounds, the copied region is limited to the image boundary. If the copied region goes out of the bounds of the `GraphicsContext` area, pixels out of the range will not be drawn.

Parameters:

`src` - the image to copy from
`x_src` - the x coordinate of the upper-left corner of the region to copy
`y_src` - the y coordinate of the upper-left corner of the region to copy
`width` - the width of the region to copy
`height` - the height of the region to copy
`x_dest` - the x coordinate of the anchor point in the destination
`y_dest` - the y coordinate of the anchor point in the destination
`anchor` - position of the region around the anchor point

Throws:

`NullPointerException` - if `src` is `null`

`IllegalArgumentException` - if anchor is not valid,
if `src` and this `GraphicsContext` target different displays

---

copyArea

```
public final void copyArea(int x_src,
                           int y_src,
                           int width,
                           int height,
                           int x_dest,
                           int y_dest,
                           int anchor)
```

Copies an area within a `GraphicsContext`. The region to copy is specified as rectangular area whose origin is at `(x_src,y_src)` and whose dimension is given by `width` and `height`.
The destination is defined by an anchor point at `(x_dest,y_dest)`. Position constants may be given to specify the precise location of the area around the anchor point.

If the specified source region exceeds the clipping rectangle, the copied region is limited to the clipping rectangle. If the copied region goes out of the bounds of the `GraphicsContext` area, pixels out of the range will not be drawn.

Parameters:

`x_src` - the x coordinate of upper-left corner of source area
`y_src` - the y coordinate of upper-left corner of source area
`width` - the width of the source area
`height` - the height of the source area
`x_dest` - the x coordinate of the destination anchor point
`y_dest` - the y coordinate of the destination anchor point
`anchor` - position of the region around the anchor point within the destination image

Throws:

`IllegalArgumentException` - if anchor is not valid, or if destination (aka 'this' object) is a GraphicsContext of a Screen.

---

getARGB

```
public final void getARGB(int[] argbData,
                          int offset,
                          int scanlength,
                          int x,
                          int y,
                          int width,
                          int height)
```

Obtains ARGB pixel data from the specified region of this graphics context and stores it in the provided array of integers. Each pixel value is stored in `0xAARRGGBB` format, where the high-order byte contains the alpha channel and the remaining bytes contain color components for red, green and blue, respectively. The alpha channel specifies the opacity of the pixel, where a value of `0x00` represents a pixel that is fully transparent and a value of `0xFF` represents a fully opaque pixel.

Color values may be resampled to reflect the display capabilities of the device (for example, red, green or blue pixels may all be represented by the same gray value on a grayscale device).

The `scanlength` specifies the relative offset within the array between the corresponding pixels of consecutive rows. In order to prevent rows of stored pixels from overlapping, the absolute value of `scanlength` must be greater than or equal to `width`. Negative values of `scanlength` are allowed. In all cases, this must result in every reference being within the bounds of the `rgbData` array.

Parameters:

`argbData` - an array of integers in which the ARGB pixel data is stored
`offset` - the index into the array where the first ARGB value is stored
`scanlength` - the relative offset in the array between corresponding pixels in consecutive rows of the region
`x` - the x-coordinate of the upper left corner of the region
`y` - the y-coordinate of the upper left corner of the region
`width` - the width of the region
`height` - the height of the region

Throws:

`ArrayIndexOutOfBoundsException` - if the requested operation would attempt to access an element in the `rgbData` array whose index is either negative or beyond its length (the contents of the array are unchanged)
`IllegalArgumentException` - if the area being retrieved exceeds the bounds of the source graphics context,
if the absolute value of `scanlength` is less than `width`
`NullPointerException` - if `rgbData` is null

---

drawARGB

```
public void drawARGB(int[] argbData,
                     int offset,
                     int scanlength,
                     int x,
                     int y,
                     int width,
                     int height,
                     int anchor)
```

Gets ARGB pixel data from the provided array of integers and draws it in the specified region of this graphics context. Each pixel value is stored in `0xAARRGGBB` format, where the high-order byte contains the alpha channel and the remaining bytes contain color components for red, green and blue, respectively. The alpha channel specifies the opacity of the pixel, where a value of `0x00` represents a pixel that is fully transparent and a value of `0xFF` represents a fully opaque pixel.

Color values may be resampled to reflect the display capabilities of the device (for example, red, green or blue pixels may all be represented by the same gray value on a grayscale device).

The `scanlength` specifies the relative offset within the array between the corresponding pixels of consecutive rows. In order to prevent rows of stored pixels from overlapping, the absolute value of `scanlength` must be greater than or equal to `width`. Negative values of `scanlength` are allowed. In all cases, this must result in every reference being within the bounds of the `rgbData` array.

Parameters:

`argbData` - an array of integers in which the ARGB pixel data is stored
`offset` - the index into the array where the first ARGB value is stored
`scanlength` - the relative offset in the array between corresponding pixels in consecutive rows of the region
`x` - the x coordinate of the anchor point in the destination
`y` - the x coordinate of the anchor point in the destination
`width` - the width of the region
`height` - the height of the region
`anchor` - position of the region around the anchor point

Throws:

`NullPointerException` - if `argbData` is `null`
`IllegalArgumentException` - if anchor is not valid,
if the absolute value of `scanlength` is less than `width`
`ArrayIndexOutOfBoundsException` - if the requested operation would attempt to access an element in the `argbData` array whose index is either negative or beyond its length.

---

drawString

```
public final void drawString(String str,
                             int x,
                             int y,
                             int anchor)
```

Draws the string using the current font and color.
The text anchor point is at position `(x,y)`. Position constants may be given to specify the precise location of the text around the anchor point.

See GraphicsContext for details of anchors.

Parameters:

`str` - the string to draw
`x` - the x coordinate of the anchor point
`y` - the y coordinate of the anchor point
`anchor` - position of the text around the anchor point

Throws:

`NullPointerException` - if str is null
`IllegalArgumentException` - if anchor is not a valid value

---

drawSubstring

```
public final void drawSubstring(String str,
                                int offset,
                                int len,
                                int x,
                                int y,
                                int anchor)
```

Draws the string from `offset` to `offset+length` using the current font and color.
The text anchor point is at position `(x,y)`. Position constants may be given to specify the precise location of the text around the anchor point.

See GraphicsContext for details of anchors.

Parameters:

`str` - the string to draw
`offset` - index of the first character in the string to draw
`len` - number of characters to draw from `offset`
`x` - the x coordinate of the anchor point
`y` - the y coordinate of the anchor point
`anchor` - position of the string text around the anchor point

Throws:

`StringIndexOutOfBoundsException` - if `offset` and `length` do not specify a valid range within `str`
`IllegalArgumentException` - if anchor is not a valid value
`NullPointerException` - if `str` is `null`

---

drawChar

```
public final void drawChar(char character,
                           int x,
                           int y,
                           int anchor)
```

Draws a character using the current font and color.
The text anchor point is at position `(x,y)`. Position constants may be given to specify the precise location of the

character around the anchor point.

Parameters:
`character` - the character to draw
`x` - the x coordinate of the anchor point
`y` - the y coordinate of the anchor point
`anchor` - position of the character around the anchor point
Throws:
`IllegalArgumentException` - if anchor is not a valid value

---

drawChars

```
public final void drawChars(char[] data,
                            int offset,
                            int length,
                            int x,
                            int y,
                            int anchor)
```

Draws some characters using the current font and color.
The text anchor point is at position `(x,y)`. Position constants may be given to specify the precise location of the text around the anchor point.

Parameters:
`data` - the array of characters to draw
`offset` - offset of the first character to draw in `data`
`length` - the number of characters to draw from `offset`
`x` - the x coordinate of the anchor point
`y` - the y coordinate of the anchor point
`anchor` - position of the text around the anchor point
Throws:
`IndexOutOfBoundsException` - if `offset` and `length` do not specify a valid range within `data`
`IllegalArgumentException` - if anchor is not a valid value
`NullPointerException` - if `data` is `null`

---

setEllipsis

```
public void setEllipsis(boolean enable)
```

Enables (disables) truncation when rendering characters. When enabled, a text that would be outside the current clip will have its last visible character replaced by ellipsis (three dots).

Parameters:
`enable` - true to enable the ellipsis mode

---

getDisplay

```
public Display getDisplay()
```

Returns the display associated with the GraphicsContext.

Returns:
the display associated with the GraphicsContext

---

getEllipsis

```
public boolean getEllipsis()
```

Returns `true` if the truncation mechanism is enabled.

Returns:

`true` if the truncation mechanism is enabled.

Class Image

ej.microui.io

```
java.lang.Object
  └
    ej.microui.io.Image
```

```
public class Image
extends Object
```

An `Image` object holds graphical display data. An `Image` is created for a specific `Display`, i.e. it may only be displayed on the display it has been created for.

An `Image` is either mutable or immutable depending on the way it has been created.

Immutable images are created/loaded from data resources, and they may not be modified. They can be either loaded from an internal non-standard image format or dynamically created from supported standard image formats, depending on the MicroUI implementation.

Creating images may not be possible within a particular MicroUI implementation. If the MicroUI implementation doesn't allow the allocation of memory to store the image's buffer, a `MicroUIException` is thrown.

Mutable images are created as blank images containing only white pixels. The application may render on a mutable image by calling `getGraphicsContext()` on the image to obtain a `GraphicsContext` object expressly for this purpose.

See Also:

GraphicsContext

Field Summary		Page
static int	**APP1** Constant for app1 image.	204
static int	**ART** Constant for art image.	204
static int	**AVI** Constant for avi image.	204
static int	**AVS** Constant for avs image.	205
static int	**B** Constant for b image.	205
static int	**BIE** Constant for bie image.	205
static int	**BIM8** Constant for bim8 image.	205
static int	**BMP** Constant for bmp image.	205
static int	**BMP_MONOCHROM** Constant for bmp_monochrom image.	205
static int	**C** Constant for c image.	206
static int	**CAPTION** Constant for caption image.	206

static int	**CMYK** Constant for cmyk image.	206
static int	**CMYKA** Constant for cmyka image.	206
static int	**CUT** Constant for cut image.	206
static int	**DCM** Constant for dcm image.	206
static int	**DCX** Constant for dcx image.	207
static int	**DIB** Constant for dib image.	207
static int	**DPS** Constant for dps image.	207
static int	**DPX** Constant for dpx image.	207
static int	**EPDF** Constant for epdf image.	207
static int	**EPI** Constant for epi image.	207
static int	**EPS** Constant for eps image.	208
static int	**EPS2** Constant for eps2 image.	208
static int	**EPS3** Constant for eps3 image.	208
static int	**EPSF** Constant for epsf image.	208
static int	**EPSI** Constant for epsi image.	208
static int	**EPT** Constant for ept image.	208
static int	**FAX** Constant for fax image.	209
static int	**FILE** Constant for file image.	209
static int	**FITS** Constant for fits image.	209
static int	**FPX** Constant for fpx image.	209
static int	**FRACTAL** Constant for fractal image.	209
static int	**FTP** Constant for ftp image.	209
static int	**G** Constant for g image.	210
static int	**G3** Constant for g3 image.	210
static int	**GIF** Constant for gif image.	210

static int	**GIF87** Constant for gif87 image.	*210*
static int	**GRADIENT** Constant for gradient image.	*210*
static int	**GRANITE** Constant for granite image.	*210*
static int	**GRAY** Constant for gray image.	*211*
static int	**H** Constant for h image.	*211*
static int	**HDF** Constant for hdf image.	*211*
static int	**HISTOGRAM** Constant for histogram image.	*211*
static int	**HTM** Constant for htm image.	*211*
static int	**HTML** Constant for html image.	*211*
static int	**HTTP** Constant for http image.	*212*
static int	**ICB** Constant for icb image.	*212*
static int	**ICM** Constant for icm image.	*212*
static int	**ICO** Constant for ico image.	*212*
static int	**ICON** Constant for icon image.	*212*
static int	**IPTC** Constant for iptc image.	*212*
static int	**JBG** Constant for jbg image.	*213*
static int	**JBIG** Constant for jbig image.	*213*
static int	**JP2** Constant for jp2 image.	*213*
static int	**JPC** Constant for jpc image.	*213*
static int	**JPEG** Constant for jpeg image.	*213*
static int	**JPG** Constant for jpg image.	*213*
static int	**K** Constant for k image.	*214*
static int	**LABEL** Constant for label image.	*214*
static int	**LOGO** Constant for logo image.	*214*
static int	**M** Constant for m image.	*214*

static int	**M2V** Constant for m2v image.	214
static int	**MAP** Constant for map image.	214
static int	**MAT** Constant for mat image.	215
static int	**MATTE** Constant for matte image.	215
static int	**MIFF** Constant for miff image.	215
static int	**MNG** Constant for mng image.	215
static int	**MONO** Constant for mono image.	215
static int	**MPC** Constant for mpc image.	215
static int	**MPEG** Constant for mpeg image.	216
static int	**MPG** Constant for mpg image.	216
static int	**MPR** Constant for mpr image.	216
static int	**MPRI** Constant for mpri image.	216
static int	**MSL** Constant for msl image.	216
static int	**MTV** Constant for mtv image.	216
static int	**MVG** Constant for mvg image.	217
static int	**NETSCAPE** Constant for netscape image.	217
static int	**NULL** Constant for null image.	217
static int	**O** Constant for o image.	217
static int	**OTB** Constant for otb image.	217
static int	**P7** Constant for p7 image.	217
static int	**PAL** Constant for pal image.	218
static int	**PALM** Constant for palm image.	218
static int	**PBM** Constant for pbm image.	218
static int	**PCD** Constant for pcd image.	218
static int	**PCDS** Constant for pcds image.	218

static int	**PCL** Constant for pcl image.	218
static int	**PCT** Constant for pct image.	219
static int	**PCX** Constant for pcx image.	219
static int	**PDB** Constant for pdb image.	219
static int	**PDF** Constant for pdf image.	219
static int	**PFA** Constant for pfa image.	219
static int	**PFB** Constant for pfb image.	219
static int	**PGM** Constant for pgm image.	220
static int	**PICON** Constant for picon image.	220
static int	**PICT** Constant for pict image.	220
static int	**PIX** Constant for pix image.	220
static int	**PLASMA** Constant for plasma image.	220
static int	**PM** Constant for pm image.	220
static int	**PNG** Constant for png image.	221
static int	**PNM** Constant for pnm image.	221
static int	**PPM** Constant for ppm image.	221
static int	**PREVIEW** Constant for preview image.	221
static int	**PS** Constant for ps image.	221
static int	**PS2** Constant for ps2 image.	221
static int	**PS3** Constant for ps3 image.	222
static int	**PSD** Constant for psd image.	222
static int	**PTIF** Constant for ptif image.	222
static int	**PWP** Constant for pwp image.	222
static int	**R** Constant for r image.	222
static int	**RAS** Constant for ras image.	222

static int	**RGB** Constant for rgb image.	223
static int	**RGBA** Constant for rgba image.	223
static int	**RLA** Constant for rla image.	223
static int	**RLE** Constant for rle image.	223
static int	**ROSE** Constant for rose image.	223
static int	**SCT** Constant for sct image.	223
static int	**SFW** Constant for sfw image.	224
static int	**SGI** Constant for sgi image.	224
static int	**SHTML** Constant for shtml image.	224
static int	**STEGANO** Constant for stegano image.	224
static int	**SUN** Constant for sun image.	224
static int	**SVG** Constant for svg image.	224
static int	**TEXT** Constant for text image.	225
static int	**TGA** Constant for tga image.	225
static int	**TIF** Constant for tif image.	225
static int	**TIFF** Constant for tiff image.	225
static int	**TILE** Constant for tile image.	225
static int	**TIM** Constant for tim image.	225
static int	**TTF** Constant for ttf image.	226
static int	**TXT** Constant for txt image.	226
static int	**UIL** Constant for uil image.	226
static int	**UYVY** Constant for uyvy image.	226
static int	**VDA** Constant for vda image.	226
static int	**VICAR** Constant for vicar image.	226
static int	**VID** Constant for vid image.	227

static int	**VIFF** Constant for viff image.	227
static int	**VST** Constant for vst image.	227
static int	**WBMP** Constant for wbmp image.	227
static int	**WPG** Constant for wpg image.	227
static int	**X** Constant for x image.	227
static int	**XBM** Constant for xbm image.	228
static int	**XC** Constant for xc image.	228
static int	**XCF** Constant for xcf image.	228
static int	**XPM** Constant for xpm image.	228
static int	**XV** Constant for xv image.	228
static int	**XWD** Constant for xwd image.	228
static int	**Y** Constant for y image.	229
static int	**YUV** Constant for yuv image.	229

Method Summary		Page
boolean	**collidesWith**(int x0, int y0, int w, int h, Image image, int posX, int posY, int ix0, int iy0, int iw, int ih, boolean checkTransparency) Checks for a collision between this Image and the specified Image.	236
boolean	**collidesWith**(int x0, int y0, int w, int h, Image image, int posX, int posY, int ix0, int iy0, int iw, int ih, int checkColor) Checks for a collision between this Image and the specified Image.	237
static Image	**createImage**(byte[] imageData, int imageOffset, int imageLength, int imageFormat) Creates an immmutable image from a byte array for the default display.	231
static Image	**createImage**(Display d, byte[] imageData, int imageOffset, int imageLength, int imageFormat) Creates an immmutable image from a byte array for the given display.	231
static Image	**createImage**(Display d, int width, int height) Creates a new mutable image for the given display and with the given size.	229
static Image	**createImage**(Display d, int[] argbData, int offset, int scanlength, int width, int height, boolean processAlpha) Creates an immutable image from an int array of pixels for the specified display.	234
static Image	**createImage**(Display d, InputStream stream, int imageFormat) Creates an immutable image from an InputStream.	233
static Image	**createImage**(Display d, String name, int imageFormat) Creates an immutable image from a resource for the given display.	230
static Image	**createImage**(Image source) Returns an immutable image from another image.	230

static Image	**createImage**(Image image, int x, int y, int width, int height) Creates an immutable image from another image zone.	*232*
static Image	**createImage**(int width, int height) Creates a new mutable image for the default display and with the given size.	*229*
static Image	**createImage**(int[] argbData, int offset, int scanlength, int width, int height, boolean processAlpha) Creates an immutable image from an int array for the default display.	*233*
static Image	**createImage**(InputStream stream, int imageFormat) Creates an immutable image from an InputStream.	*232*
static Image	**createImage**(String name, int imageFormat) Creates an immutable image from a resource for the default display.	*230*
void	**getARGB**(int[] argbData, int offset, int scanlength, int x, int y, int width, int height) Obtains ARGB pixel data from the specified region of this image and stores it in the provided array of integers.	*235*
Display	**getDisplay**() Returns the display associated with the image.	*235*
GraphicsContext	**getGraphicsContext**() Returns a new GraphicsContext object to draw on the image.	*234*
int	**getHeight**() Returns the height of the image in pixels.	*235*
int	**getWidth**() Returns the width of the image in pixels.	*235*
boolean	**isMutable**() Tells whether this image is mutable.	*235*

Field Detail

APP1

```
public static final int APP1
```

Constant for app1 image.

Value 1 is assigned to APP1.

ART

```
public static final int ART
```

Constant for art image.

Value 2 is assigned to ART.

AVI

```
public static final int AVI
```

Constant for avi image.

Value 3 is assigned to AVI.

AVS

```
public static final int AVS
```

Constant for avs image.

Value 4 is assigned to AVS.

B

```
public static final int B
```

Constant for b image.

Value 5 is assigned to B.

BIE

```
public static final int BIE
```

Constant for bie image.

Value 6 is assigned to BIE.

BIM8

```
public static final int BIM8
```

Constant for bim8 image.

Value 7 is assigned to BIM8.

BMP

```
public static final int BMP
```

Constant for bmp image.

Value 8 is assigned to BMP.

BMP_MONOCHROM

```
public static final int BMP_MONOCHROM
```

Constant for bmp_monochrom image.

Value 9 is assigned to BMP_MONOCHROM.

C

```
public static final int C
```

Constant for c image.

Value `10` is assigned to `C`.

CAPTION

```
public static final int CAPTION
```

Constant for caption image.

Value `11` is assigned to `CAPTION`.

CMYK

```
public static final int CMYK
```

Constant for cmyk image.

Value `12` is assigned to `CMYK`.

CMYKA

```
public static final int CMYKA
```

Constant for cmyka image.

Value `13` is assigned to `CMYKA`.

CUT

```
public static final int CUT
```

Constant for cut image.

Value `14` is assigned to `CUT`.

DCM

```
public static final int DCM
```

Constant for dcm image.

Value `15` is assigned to `DCM`.

DCX

```
public static final int DCX
```

Constant for dcx image.

Value `16` is assigned to `DCX`.

DIB

```
public static final int DIB
```

Constant for dib image.

Value `17` is assigned to `DIB`.

DPS

```
public static final int DPS
```

Constant for dps image.

Value `18` is assigned to `DPS`.

DPX

```
public static final int DPX
```

Constant for dpx image.

Value `19` is assigned to `DPX`.

EPDF

```
public static final int EPDF
```

Constant for epdf image.

Value `20` is assigned to `EPDF`.

EPI

```
public static final int EPI
```

Constant for epi image.

Value `21` is assigned to `EPI`.

EPS

```
public static final int EPS
```

Constant for eps image.

Value `22` is assigned to `EPS`.

EPS2

```
public static final int EPS2
```

Constant for eps2 image.

Value `23` is assigned to `EPS2`.

EPS3

```
public static final int EPS3
```

Constant for eps3 image.

Value `24` is assigned to `EPS3`.

EPSF

```
public static final int EPSF
```

Constant for epsf image.

Value `25` is assigned to `EPSF`.

EPSI

```
public static final int EPSI
```

Constant for epsi image.

Value `26` is assigned to `EPSI`.

EPT

```
public static final int EPT
```

Constant for ept image.

Value `27` is assigned to `EPT`.

FAX

```
public static final int FAX
```

Constant for fax image.

Value `28` is assigned to `FAX`.

FILE

```
public static final int FILE
```

Constant for file image.

Value `29` is assigned to `FILE`.

FITS

```
public static final int FITS
```

Constant for fits image.

Value `30` is assigned to `FITS`.

FPX

```
public static final int FPX
```

Constant for fpx image.

Value `31` is assigned to `FPX`.

FRACTAL

```
public static final int FRACTAL
```

Constant for fractal image.

Value `32` is assigned to `FRACTAL`.

FTP

```
public static final int FTP
```

Constant for ftp image.

Value `33` is assigned to `FTP`.

G

```
public static final int G
```

Constant for g image.

Value 34 is assigned to G.

G3

```
public static final int G3
```

Constant for g3 image.

Value 35 is assigned to G3.

GIF

```
public static final int GIF
```

Constant for gif image.

Value 36 is assigned to GIF.

GIF87

```
public static final int GIF87
```

Constant for gif87 image.

Value 37 is assigned to GIF87.

GRADIENT

```
public static final int GRADIENT
```

Constant for gradient image.

Value 38 is assigned to GRADIENT.

GRANITE

```
public static final int GRANITE
```

Constant for granite image.

Value 39 is assigned to GRANITE.

GRAY

```
public static final int GRAY
```

Constant for gray image.

Value `40` is assigned to `GRAY`.

H

```
public static final int H
```

Constant for h image.

Value `41` is assigned to `H`.

HDF

```
public static final int HDF
```

Constant for hdf image.

Value `42` is assigned to `HDF`.

HISTOGRAM

```
public static final int HISTOGRAM
```

Constant for histogram image.

Value `43` is assigned to `HISTOGRAM`.

HTM

```
public static final int HTM
```

Constant for htm image.

Value `44` is assigned to `HTM`.

HTML

```
public static final int HTML
```

Constant for html image.

Value `45` is assigned to `HTML`.

HTTP

```
public static final int HTTP
```

Constant for http image.

Value `46` is assigned to `HTTP`.

ICB

```
public static final int ICB
```

Constant for icb image.

Value `47` is assigned to `ICB`.

ICM

```
public static final int ICM
```

Constant for icm image.

Value `48` is assigned to `ICM`.

ICO

```
public static final int ICO
```

Constant for ico image.

Value `49` is assigned to `ICO`.

ICON

```
public static final int ICON
```

Constant for icon image.

Value `50` is assigned to `ICON`.

IPTC

```
public static final int IPTC
```

Constant for iptc image.

Value `51` is assigned to `IPTC`.

JBG

```
public static final int JBG
```

Constant for jbg image.

Value `52` is assigned to `JBG`.

JBIG

```
public static final int JBIG
```

Constant for jbig image.

Value `53` is assigned to `JBIG`.

JP2

```
public static final int JP2
```

Constant for jp2 image.

Value `54` is assigned to `JP2`.

JPC

```
public static final int JPC
```

Constant for jpc image.

Value `55` is assigned to `JPC`.

JPEG

```
public static final int JPEG
```

Constant for jpeg image.

Value `56` is assigned to `JPEG`.

JPG

```
public static final int JPG
```

Constant for jpg image.

Value `57` is assigned to `JPG`.

K

```
public static final int K
```

Constant for k image.

Value 58 is assigned to K.

LABEL

```
public static final int LABEL
```

Constant for label image.

Value 59 is assigned to LABEL.

LOGO

```
public static final int LOGO
```

Constant for logo image.

Value 60 is assigned to LOGO.

M

```
public static final int M
```

Constant for m image.

Value 61 is assigned to M.

M2V

```
public static final int M2V
```

Constant for m2v image.

Value 62 is assigned to M2V.

MAP

```
public static final int MAP
```

Constant for map image.

Value 63 is assigned to MAP.

MAT

```
public static final int MAT
```

Constant for mat image.

Value `64` is assigned to `MAT`.

MATTE

```
public static final int MATTE
```

Constant for matte image.

Value `65` is assigned to `MATTE`.

MIFF

```
public static final int MIFF
```

Constant for miff image.

Value `66` is assigned to `MIFF`.

MNG

```
public static final int MNG
```

Constant for mng image.

Value `67` is assigned to `MNG`.

MONO

```
public static final int MONO
```

Constant for mono image.

Value `68` is assigned to `MONO`.

MPC

```
public static final int MPC
```

Constant for mpc image.

Value `69` is assigned to `MPC`.

MPEG

```
public static final int MPEG
```

Constant for mpeg image.

Value `70` is assigned to `MPEG`.

MPG

```
public static final int MPG
```

Constant for mpg image.

Value `71` is assigned to `MPG`.

MPR

```
public static final int MPR
```

Constant for mpr image.

Value `72` is assigned to `MPR`.

MPRI

```
public static final int MPRI
```

Constant for mpri image.

Value `73` is assigned to `MPRI`.

MSL

```
public static final int MSL
```

Constant for msl image.

Value `74` is assigned to `MSL`.

MTV

```
public static final int MTV
```

Constant for mtv image.

Value `75` is assigned to `MTV`.

MVG

```
public static final int MVG
```

Constant for mvg image.

Value `76` is assigned to `MVG`.

NETSCAPE

```
public static final int NETSCAPE
```

Constant for netscape image.

Value `77` is assigned to `NETSCAPE`.

NULL

```
public static final int NULL
```

Constant for null image.

Value `78` is assigned to `NULL`.

O

```
public static final int O
```

Constant for o image.

Value `79` is assigned to `O`.

OTB

```
public static final int OTB
```

Constant for otb image.

Value `80` is assigned to `OTB`.

P7

```
public static final int P7
```

Constant for p7 image.

Value `81` is assigned to `P7`.

PAL

```
public static final int PAL
```

Constant for pal image.

Value `82` is assigned to `PAL`.

PALM

```
public static final int PALM
```

Constant for palm image.

Value `83` is assigned to `PALM`.

PBM

```
public static final int PBM
```

Constant for pbm image.

Value `84` is assigned to `PBM`.

PCD

```
public static final int PCD
```

Constant for pcd image.

Value `85` is assigned to `PCD`.

PCDS

```
public static final int PCDS
```

Constant for pcds image.

Value `86` is assigned to `PCDS`.

PCL

```
public static final int PCL
```

Constant for pcl image.

Value `87` is assigned to `PCL`.

PCT

```
public static final int PCT
```

Constant for pct image.

Value `88` is assigned to `PCT`.

PCX

```
public static final int PCX
```

Constant for pcx image.

Value `89` is assigned to `PCX`.

PDB

```
public static final int PDB
```

Constant for pdb image.

Value `90` is assigned to `PDB`.

PDF

```
public static final int PDF
```

Constant for pdf image.

Value `91` is assigned to `PDF`.

PFA

```
public static final int PFA
```

Constant for pfa image.

Value `92` is assigned to `PFA`.

PFB

```
public static final int PFB
```

Constant for pfb image.

Value `93` is assigned to `PFB`.

PGM

```
public static final int PGM
```

Constant for pgm image.

Value `94` is assigned to `PGM`.

PICON

```
public static final int PICON
```

Constant for picon image.

Value `95` is assigned to `PICON`.

PICT

```
public static final int PICT
```

Constant for pict image.

Value `96` is assigned to `PICT`.

PIX

```
public static final int PIX
```

Constant for pix image.

Value `97` is assigned to `PIX`.

PLASMA

```
public static final int PLASMA
```

Constant for plasma image.

Value `98` is assigned to `PLASMA`.

PM

```
public static final int PM
```

Constant for pm image.

Value `99` is assigned to `PM`.

PNG

```
public static final int PNG
```

Constant for png image.

Value `100` is assigned to `PNG`.

PNM

```
public static final int PNM
```

Constant for pnm image.

Value `101` is assigned to `PNM`.

PPM

```
public static final int PPM
```

Constant for ppm image.

Value `102` is assigned to `PPM`.

PREVIEW

```
public static final int PREVIEW
```

Constant for preview image.

Value `103` is assigned to `PREVIEW`.

PS

```
public static final int PS
```

Constant for ps image.

Value `104` is assigned to `PS`.

PS2

```
public static final int PS2
```

Constant for ps2 image.

Value `105` is assigned to `PS2`.

PS3

```
public static final int PS3
```

Constant for ps3 image.

Value `106` is assigned to `PS3`.

PSD

```
public static final int PSD
```

Constant for psd image.

Value `107` is assigned to `PSD`.

PTIF

```
public static final int PTIF
```

Constant for ptif image.

Value `108` is assigned to `PTIF`.

PWP

```
public static final int PWP
```

Constant for pwp image.

Value `109` is assigned to `PWP`.

R

```
public static final int R
```

Constant for r image.

Value `110` is assigned to `R`.

RAS

```
public static final int RAS
```

Constant for ras image.

Value `111` is assigned to `RAS`.

RGB

```
public static final int RGB
```

Constant for rgb image.

Value `112` is assigned to `RGB`.

RGBA

```
public static final int RGBA
```

Constant for rgba image.

Value `113` is assigned to `RGBA`.

RLA

```
public static final int RLA
```

Constant for rla image.

Value `114` is assigned to `RLA`.

RLE

```
public static final int RLE
```

Constant for rle image.

Value `115` is assigned to `RLE`.

ROSE

```
public static final int ROSE
```

Constant for rose image.

Value `116` is assigned to `ROSE`.

SCT

```
public static final int SCT
```

Constant for sct image.

Value `117` is assigned to `SCT`.

SFW

```
public static final int SFW
```

Constant for sfw image.

Value `118` is assigned to `SFW`.

SGI

```
public static final int SGI
```

Constant for sgi image.

Value `119` is assigned to `SGI`.

SHTML

```
public static final int SHTML
```

Constant for shtml image.

Value `120` is assigned to `SHTML`.

STEGANO

```
public static final int STEGANO
```

Constant for stegano image.

Value `121` is assigned to `STEGANO`.

SUN

```
public static final int SUN
```

Constant for sun image.

Value `122` is assigned to `SUN`.

SVG

```
public static final int SVG
```

Constant for svg image.

Value `123` is assigned to `SVG`.

TEXT

```
public static final int TEXT
```

Constant for text image.

Value `124` is assigned to `TEXT`.

TGA

```
public static final int TGA
```

Constant for tga image.

Value `125` is assigned to `TGA`.

TIF

```
public static final int TIF
```

Constant for tif image.

Value `126` is assigned to `TIF`.

TIFF

```
public static final int TIFF
```

Constant for tiff image.

Value `127` is assigned to `TIFF`.

TILE

```
public static final int TILE
```

Constant for tile image.

Value `128` is assigned to `TILE`.

TIM

```
public static final int TIM
```

Constant for tim image.

Value `129` is assigned to `TIM`.

TTF

```
public static final int TTF
```

Constant for ttf image.

Value `130` is assigned to `TTF`.

TXT

```
public static final int TXT
```

Constant for txt image.

Value `131` is assigned to `TXT`.

UIL

```
public static final int UIL
```

Constant for uil image.

Value `132` is assigned to `UIL`.

UYVY

```
public static final int UYVY
```

Constant for uyvy image.

Value `133` is assigned to `UYVY`.

VDA

```
public static final int VDA
```

Constant for vda image.

Value `134` is assigned to `VDA`.

VICAR

```
public static final int VICAR
```

Constant for vicar image.

Value `135` is assigned to `VICAR`.

VID

```
public static final int VID
```

Constant for vid image.

Value `136` is assigned to `VID`.

VIFF

```
public static final int VIFF
```

Constant for viff image.

Value `137` is assigned to `VIFF`.

VST

```
public static final int VST
```

Constant for vst image.

Value `138` is assigned to `VST`.

WBMP

```
public static final int WBMP
```

Constant for wbmp image.

Value `139` is assigned to `WBMP`.

WPG

```
public static final int WPG
```

Constant for wpg image.

Value `140` is assigned to `WPG`.

X

```
public static final int X
```

Constant for x image.

Value `141` is assigned to `X`.

XBM

```
public static final int XBM
```

Constant for xbm image.

Value `142` is assigned to `XBM`.

XC

```
public static final int XC
```

Constant for xc image.

Value `143` is assigned to `XC`.

XCF

```
public static final int XCF
```

Constant for xcf image.

Value `144` is assigned to `XCF`.

XPM

```
public static final int XPM
```

Constant for xpm image.

Value `145` is assigned to `XPM`.

XV

```
public static final int XV
```

Constant for xv image.

Value `146` is assigned to `XV`.

XWD

```
public static final int XWD
```

Constant for xwd image.

Value `147` is assigned to `XWD`.

Y

```
public static final int Y
```

Constant for y image.

Value `148` is assigned to `Y`.

YUV

```
public static final int YUV
```

Constant for yuv image.

Value `149` is assigned to `YUV`.

Method Detail

createImage

```
public static Image createImage(int width,
                                int height)
```

Creates a new mutable image for the default display and with the given size.
The effect is identical to
`createImage(Display.getDefaultDisplay(), width, height);`

Parameters:
`width` - the width of the new image, in pixels
`height` - the height of the new image, in pixels
Returns:
the created image
Throws:
`IllegalArgumentException` - if either `width` or `height` is zero or less
`MicroUIException` - if MicroUI implementation cannot create the image
`OutOfMemoryError` - if there is not enough room to add a new image.
See Also:
`createImage(Display, int, int)`

---

createImage

```
public static Image createImage(Display d,
                                int width,
                                int height)
```

Creates a new mutable image for the given display and with the given size. Every pixel within the newly created image is white. Given `width` and `height` must be greater than zero.

Parameters:
`d` - the display for which the image is created
`width` - the width of the new image, in pixels
`height` - the height of the new image, in pixels
Returns:
the created image

Throws:

`IllegalArgumentException` - if either `width` or `height` is zero or less
`MicroUIException` - if MicroUI implementation cannot create the image
`OutOfMemoryError` - if there is not enough room to add a new image.

---

createImage

```
public static Image createImage(Image source)
```

Returns an immutable image from another image. If `source` is an immutable image, it may simply be returned. If `source` is a mutable image, a new immutable image, copy of `source`, is returned.

Parameters:

`source` - the source image to be copied

Returns:

an immutable image copy of/or `source`

Throws:

`NullPointerException` - if source is null
`MicroUIException` - if MicroUI implementation cannot create the image
`OutOfMemoryError` - if there is not enough room to add a new image.

---

createImage

```
public static Image createImage(String name,
                                int imageFormat)
                         throws IOException
```

Creates an immutable image from a resource for the default display.
The effect is identical to
`createImage(Display.getDefaultDisplay(), name, imageFormat);`

Parameters:

`name` - a resource name matching image data in a supported standard image format
`imageFormat` - the image data format

Returns:

the created image

Throws:

`IOException` - if the resource is not found, if the data can not be decoded or if the name is not an absolute path.
`NullPointerException` - if name is null
`MicroUIException` - if MicroUI implementation cannot create the image
`OutOfMemoryError` - if there is not enough room to add a new image.

See Also:

`createImage(Display, String, int)`

---

createImage

```
public static Image createImage(Display d,
                                String name,
                                int imageFormat)
                         throws IOException
```

Creates an immutable image from a resource for the given display. `name` allows to retrieve a resource which must support standard image data in a supported standard image format matching the given `imageFormat`. Image data is decoded to create the new `Image`. `name` is an absolute resource name (start with'/').

Parameters:
`d` - the display for which the image is created
`name` - a resource name matching image data in a supported standard image format
`imageFormat` - the image data format
Returns:
the created image
Throws:
`IOException` - if the resource is not found, if the data can not be decoded or if the name is not an absolute path.
`NullPointerException` - if name is null
`MicroUIException` - if MicroUI implementation cannot create the image
`OutOfMemoryError` - if there is not enough room to add a new image.

---

createImage

```
public static Image createImage(byte[] imageData,
                                int imageOffset,
                                int imageLength,
                                int imageFormat)
```

Creates an immmutable image from a byte array for the default display.
The effect is identical to
`createImage(Display.getDefaultDisplay(), imageOffset, imageLength, imageFormat);`

Parameters:
`imageData` - the data in a supported standard image format
`imageOffset` - the offset of the start of the data in `imageData`
`imageLength` - the length of the data
`imageFormat` - the image data format
Returns:
the created image
Throws:
`ArrayIndexOutOfBoundsException` - if `imageOffset` and `imageLength` specify an invalid range
`NullPointerException` - if `imageData` is null
`IllegalArgumentException` - if `imageData` can not be decoded or if `imageFormat` is not coherent with `imageData`
`MicroUIException` - if MicroUI implementation cannot create the image
`OutOfMemoryError` - if there is not enough room to add a new image.
See Also:
`createImage(Display, byte[], int, int, int)`

---

createImage

```
public static Image createImage(Display d,
                                byte[] imageData,
                                int imageOffset,
                                int imageLength,
                                int imageFormat)
```

Creates an immmutable image from a byte array for the given display. Data in `imageData` from `imageOffset` to `imageOffset+imageLength` must be of a supported standard image format, matching the given `imageFormat`. Data is decoded to create the new `Image`.

Parameters:
`d` - the display for which the image is created
`imageData` - the data in a supported standard image format
`imageOffset` - the offset of the start of the data in `imageData`
`imageLength` - the length of the data

`imageFormat` - the image data format

Returns:

the created image

Throws:

`ArrayIndexOutOfBoundsException` - if `imageOffset` and `imageLength` specify an invalid range
`NullPointerException` - if `imageData` is null
`IllegalArgumentException` - if `imageData` can not be decoded or if `imageFormat` is not coherent with `imageData`
`MicroUIException` - if MicroUI implementation cannot create the image
`OutOfMemoryError` - if there is not enough room to add a new image.

---

createImage

```
public static Image createImage(Image image,
                                int x,
                                int y,
                                int width,
                                int height)
```

Creates an immutable image from another image zone. The image is created for the same display as the original image.
The zone of the source image to copy is defined by an upper-left location `(x,y)` relative to the image and a size `(width,height)`.

If the defined zone matches the entire source image and if the source image is immutable, then the source image `image` may be returned.

Parameters:

`image` - the source image
`x` - the x coordinate of the zone to copy
`y` - the y coordinate of the zone to copy
`width` - the width of the zone to copy
`height` - the height of the zone to copy

Returns:

an immutable image

Throws:

`NullPointerException` - if image is null
`IllegalArgumentException` - if the zone to copy is out of the bounds of the source image or if either `width` or `height` is zero or negative
`MicroUIException` - if MicroUI implementation cannot create the image
`OutOfMemoryError` - if there is not enough room to add a new image.

---

createImage

```
public static Image createImage(InputStream stream,
                                int imageFormat)
                         throws IOException
```

Creates an immutable image from an `InputStream`.
The effect is identical to
`createImage(Display.getDefaultDisplay(), stream, imageFormat);`

Parameters:

`stream` - a stream providing image data
`imageFormat` - the image data format

Returns:

the created image

Throws:

`IOException` - if an I/O error occurs or if the image data cannot be decoded
`NullPointerException` - if stream is null
`MicroUIException` - if MicroUI implementation cannot create the image
`OutOfMemoryError` - if there is not enough room to add a new image.

See Also:

`createImage(Display, InputStream, int)`

---

createImage

```
public static Image createImage(Display d,
                                InputStream stream,
                                int imageFormat)
                         throws IOException
```

Creates an immutable image from an `InputStream`. The input stream must provide data bytes of a supported standard image format, matching the given `imageFormat`. Data is decoded to create the new `Image` When this method returns, the stream is not closed.

Parameters:

`d` - the display for which the image is created
`stream` - a stream providing image data
`imageFormat` - the image data format

Returns:

the created image

Throws:

`IOException` - if an I/O error occurs or if the image data cannot be decoded
`NullPointerException` - if stream is null
`MicroUIException` - if MicroUI implementation cannot create the image
`OutOfMemoryError` - if there is not enough room to add a new image.

---

createImage

```
public static Image createImage(int[] argbData,
                                int offset,
                                int scanlength,
                                int width,
                                int height,
                                boolean processAlpha)
```

Creates an immutable image from an int array for the default display.
The effect is identical to
`createImage(Display.getDefaultDisplay(), argbData, offset, scanlength, width, height, processAlpha);`

Parameters:

`argbData` - an array of integers in which the ARGB pixel data is stored
`offset` - the index into the array where the first ARGB value is stored
`scanlength` - the relative offset in the array between corresponding pixels in consecutive rows of the region
`width` - the width of the image
`height` - the height of the image
`processAlpha` - `true` if `argbData` has an alpha channel, `false` if all pixels are fully opaque

Returns:

the created image

Throws:

`ArrayIndexOutOfBoundsException` - if the requested operation would attempt to access an element in the `argbData` array whose index is either negative or beyond its length.
`IllegalArgumentException` - if the absolute value of `scanlength` is less than `width`

`MicroUIException` - if MicroUI implementation cannot create the image

---

createImage

```
public static Image createImage(Display d,
                                int[] argbData,
                                int offset,
                                int scanlength,
                                int width,
                                int height,
                                boolean processAlpha)
```

Creates an immutable image from an int array of pixels for the specified display.
Each pixel value is stored in `0xAARRGGBB` format, where the high-order byte contains the alpha channel and the remaining bytes contain color components for red, green and blue, respectively. The alpha channel specifies the opacity of the pixel, where a value of `0x00` represents a pixel that is fully transparent and a value of `0xFF` represents a fully opaque pixel.

Color values may be resampled to reflect the display capabilities of the device (for example, red, green or blue pixels may all be represented by the same gray value on a grayscale device).

The `scanlength` specifies the relative offset within the array between the corresponding pixels of consecutive rows. In order to prevent rows of stored pixels from overlapping, the absolute value of `scanlength` must be greater than or equal to `width`. Negative values of `scanlength` are allowed. In all cases, this must result in every reference being within the bounds of the `rgbData` array.

Parameters:

`d` - the display for which the image is created
`argbData` - an array of integers in which the ARGB pixel data is stored
`offset` - the index into the array where the first ARGB value is stored
`scanlength` - the relative offset in the array between corresponding pixels in consecutive rows of the region
`width` - the width of the image
`height` - the height of the image
`processAlpha` - `true` if `argbData` has an alpha channel, `false` if all pixels are fully opaque

Returns:

the created image

Throws:

`ArrayIndexOutOfBoundsException` - if the requested operation would attempt to access an element in the `argbData` array whose index is either negative or beyond its length.
`IllegalArgumentException` - if the absolute value of `scanlength` is less than `width`
`MicroUIException` - if MicroUI implementation cannot create the image
`OutOfMemoryError` - if there is not enough room to add a new image.

---

getGraphicsContext

```
public GraphicsContext getGraphicsContext()
```

Returns a new `GraphicsContext` object to draw on the image. The image must be mutable, otherwise an `IllegalArgumentException` is thrown. The returned `GraphicsContext` object has the default `GraphicsContext` behavior and allows the entire image to be drawn.

Returns:

a `GraphicsContext` object which maps this image

Throws:

`IllegalArgumentException` - if the image is immutable

---

getDisplay

```
public Display getDisplay()
```

Returns the display associated with the image.

Returns:
the display associated with the image

---

getWidth

```
public int getWidth()
```

Returns the width of the image in pixels.

Returns:
width of the image

---

getHeight

```
public int getHeight()
```

Returns the height of the image in pixels.

Returns:
height of the image

---

isMutable

```
public boolean isMutable()
```

Tells whether this image is mutable.

Returns:
whether the image is mutable.

---

getARGB

```
public void getARGB(int[] argbData,
                    int offset,
                    int scanlength,
                    int x,
                    int y,
                    int width,
                    int height)
```

Obtains ARGB pixel data from the specified region of this image and stores it in the provided array of integers. Each pixel value is stored in 0xAARRGGBB format, where the high-order byte contains the alpha channel and the remaining bytes contain color components for red, green and blue, respectively. The alpha channel specifies the opacity of the pixel, where a value of 0x00 represents a pixel that is fully transparent and a value of 0xFF represents a fully opaque pixel.

The returned values are not guaranteed to be identical to values from the original source, such as from

createRGBImage or from a PNG image. Color values may be resampled to reflect the display capabilities of the device (for example, red, green or blue pixels may all be represented by the same gray value on a grayscale device). On devices that do not support alpha blending, the alpha value will be 0xFF for opaque pixels and 0x00 for all other pixels. On devices that support alpha blending, alpha channel values may be resampled to reflect the number of levels of semitransparency supported.

The scanlength specifies the relative offset within the array between the corresponding pixels of consecutive rows. In order to prevent rows of stored pixels from overlapping, the absolute value of scanlength must be greater than or equal to width. Negative values of scanlength are allowed. In all cases, this must result in every reference being within the bounds of the rgbData array.

Parameters:

`argbData` - an array of integers in which the ARGB pixel data is stored
`offset` - the index into the array where the first ARGB value is stored
`scanlength` - the relative offset in the array between corresponding pixels in consecutive rows of the region
`x` - the x-coordinate of the upper left corner of the region
`y` - the y-coordinate of the upper left corner of the region
`width` - the width of the region
`height` - the height of the region

Throws:

`ArrayIndexOutOfBoundsException` - if the requested operation would attempt to access an element in the rgbData array whose index is either negative or beyond its length (the contents of the array are unchanged)
`IllegalArgumentException` - if the area being retrieved exceeds the bounds of the source image, if the absolute value of scanlength is less than width
`NullPointerException` - - if argbData is null

---

collidesWith

```
public boolean collidesWith(int x0,
                            int y0,
                            int w,
                            int h,
                            Image image,
                            int posX,
                            int posY,
                            int ix0,
                            int iy0,
                            int iw,
                            int ih,
                            boolean checkTransparency)
```

Checks for a collision between this `Image` and the specified `Image`. (`x0`,`y0`) and (`w`*`h`) represents this `Image`'s collision rectangle. (`ix0`,`iy0`) and (`iw`*`ih`) represents the given `image`'s collision rectangle. (`posX`, `posY`) represents the upper-left corner of the given `image` relative to this `Image`.

If pixel-level detection is used, a collision is detected only if opaque pixels collide. That is, an opaque pixel in the `Image` would have to collide with an opaque pixel in given `Image` for a collision to be detected. Only those pixels within the `Image`s' collision rectangles are checked.

If pixel-level detection is not used, this method simply checks if this `Image`'s collision rectangle intersect with the given `Image`'s collision rectangle.

Parameters:

`x0` - this `Image`'s left side coordinate.
`y0` - this `Image`'s upper side coordinate.
`w` - this `Image`'s collision rectangle width
`h` - this `Image`'s collision rectangle height
`image` - the `Image` to test for collision

posX - the image's left position relative to this Image left
posY - the image's up position relative to this Image top
ix0 - the image's left side coordinate.
iy0 - the image's upper side coordinate.
iw - the image's collision rectangle width
ih - the image's collision rectangle height
checkTransparency - true to test for collision on a pixel-by-pixel basis, false to test using simple bounds checking

Returns:

true if this Image has collided with the given Image, otherwise false

Throws:

NullPointerException - if image is null

---

collidesWith

```
public boolean collidesWith(int x0,
                            int y0,
                            int w,
                            int h,
                            Image image,
                            int posX,
                            int posY,
                            int ix0,
                            int iy0,
                            int iw,
                            int ih,
                            int checkColor)
```

Checks for a collision between this Image and the specified Image. (x0,y0) and (w*h) represents this Image's collision rectangle. (ix0,iy0) and (iw*ih) represents the given image's collision rectangle. (posX, posY) represents the upper-left corner of the given image relative to this Image.

If pixel-level detection is used, a collision is detected only if neither of the colliding pixels are the same color as checkColor. Only those pixels within the Images' collision rectangles are checked.

If pixel-level detection is not used, this method simply checks if this Image's collision rectangle intersects with the given Image's collision rectangle.

Parameters:

x0 - this Image's left side coordinate.
y0 - this Image's upper side coordinate.
w - this Image's collision rectangle width
h - this Image's collision rectangle height
image - the Image to test for collision
posX - the image's left position relative to this Image left
posY - the image's up position relative to this Image top
ix0 - the image's left side coordinate.
iy0 - the image's upper side coordinate.
iw - the image's collision rectangle width
ih - the image's collision rectangle height
checkColor - a RGB color to test for collision on a pixel-by-pixel basis, -1 to test using simple bounds checking

Returns:

true if this Image has collided with the given Image, otherwise false

Throws:

NullPointerException - if image is null

Class Keyboard

ej.microui.io

```
java.lang.Object
  └
    ej.microui.EventGenerator
      └
        ej.microui.io.Keyboard
```

Direct Known Subclasses:

Keypad

```
public class Keyboard
extends EventGenerator
```

A Keyboard event generator allows key combinations to generate a key code. A `Keyboard` generates the low-level events KEY_DOWN and KEY_UP and the high-level event TEXT_INPUT. The low-level events may be turned on, as they are off by default.

Pressing the Q key on a PC/AT US keyboard using a US keyboard layout mapping will produce:

- KEY_DOWN with Q as letter
- TEXT_INPUT with q as letter
- KEY_UP with Q as letter

If a shift key is pressed while the same Q key is pressed, the following keyboard events will be produced:

- KEY_DOWN with SHIFT as letter
- KEY_DOWN with Q as letter
- TEXT_INPUT with Q as letter
- KEY_UP with SHIFT as letter
- KEY_UP with Q as letter

Field Summary		Page
static int	**KEY_DOWN** The KEY_DOWN event action.	239
static int	**KEY_UP** The KEY_UP event action.	239
static int	**TEXT_INPUT** The TEXT_INPUT event action.	239

Constructor Summary	Page
Keyboard(int bufferSize) Keyboards hold a buffer (potentially of size one) that stores the keys before they are used by the application (key associated to KEY_UP, KEY_DOWN and TEXT_INPUT event).	240

Method Summary		Page
int	**action**(int event) **Deprecated.** *use getAction(int)*	241
boolean	**dropOnFull**() Subclasses should override this method to specify their policy.	240

int	**getAction**(int event) Returns the keyboard action held by the keyboard event.	*241*
int	**getEventType**() Gets the event type associated with the event generator	*240*
char	**getNextChar**(int event) Gets the next character associated with the specified event.	*241*
char	**nextChar**(int event) **Deprecated.** *use getNextChar(int)*	*241*
void	**onlyTextInput**(boolean onlyText) Specifies whether the KEY_UP and KEY_DOWN events should be generated.	*240*
void	**reset**() Reset the keyboard by flushing all pending characters.	*241*
void	**send**(int type, char c) Send an keyboard event to the MicroUI application.	*241*

Methods inherited from class ej.microui.EventGenerator
addToSystemPool, eventType, get, get, get, getID, getListener, removeFromSystemPool, setListener

Field Detail

TEXT_INPUT

```
public static final int TEXT_INPUT
```

The TEXT_INPUT event action.

The value `0` is assigned to `TEXT_INPUT`.

See Also:
action(int)

KEY_DOWN

```
public static final int KEY_DOWN
```

The KEY_DOWN event action.

The value `1` is assigned to `KEY_DOWN`.

See Also:
action(int)

KEY_UP

```
public static final int KEY_UP
```

The KEY_UP event action.

The value `2` is assigned to `KEY_UP`.

See Also:
action(int)

Constructor Detail

Keyboard

```
public Keyboard(int bufferSize)
```

Keyboards hold a buffer (potentially of size one) that stores the keys before they are used by the application (key associated to KEY_UP, KEY_DOWN and TEXT_INPUT event). By default, a keyboard will only send TEXT_INPUT events.

Parameters:
`bufferSize` - the size of the buffer.
See Also:
dropOnFull(), onlyTextInput(boolean)

Method Detail

dropOnFull

```
public boolean dropOnFull()
```

Subclasses should override this method to specify their policy. By default this method returns `false`, which means the oldest data are overwritten by new data. If it returns `true` new data are dropped when the pump is full.

Returns:
`true` to drop the new data or `false` to overwrite the oldest data.

---

getEventType

```
public int getEventType()
```

Gets the event type associated with the event generator

Overrides:
getEventType in class EventGenerator
Returns:
Event.KEYBOARD

---

onlyTextInput

```
public void onlyTextInput(boolean onlyText)
```

Specifies whether the KEY_UP and KEY_DOWN events should be generated. By default they are not generated.

Parameters:
`onlyText` - When true, the low level KEY_UP and KEY_DOWN are not issued to listener, only TEXT_INPUT events are sent.

---

action

```
public int action(int event)
```

Deprecated. *use getAction(int)*

---

getAction

```
public int getAction(int event)
```

Returns the keyboard action held by the keyboard event.

Parameters:
`event` - the keyboard event.
Returns:
the keyboard action held by the keyboard event.

---

nextChar

```
public char nextChar(int event)
```

Deprecated. *use getNextChar(int)*

---

getNextChar

```
public char getNextChar(int event)
```

Gets the next character associated with the specified event.

Parameters:
`event` - an event in the standard MicroUI format
Returns:
the next available char associated with the event's type, if none is available 0x0000 is returned.

---

reset

```
public void reset()
```

Reset the keyboard by flushing all pending characters.

See Also:
nextChar(int)

---

send

```
public void send(int type,
                 char c)
```

Send an keyboard event to the MicroUI application. Default event type are KEY_UP, KEY_DOWN and TEXT_INPUT

Class Keypad

ej.microui.io

```
java.lang.Object
  └
    ej.microui.EventGenerator
      └
        ej.microui.io.Keyboard
          └
            ej.microui.io.Keypad
```

```
abstract public class Keypad
extends Keyboard
```

Keypad is a Keyboard that defines an event generator for 12-key keypads. It follows the ETSI ES 202 130 mapping, which takes into account ETSI, ITU-T, CEN and ISO/IEC specifications and recommendations. Also see ISO/IEC 10646.

The key mapping is defined in Table 33 and in Table 63 of ETSI ES 202 130 (v1.1.1). In addition the next three keys have extended mapping defined as:
key10: `'*'` : this key is only used to switch from one mode to another
key11: `' '`, `'+'`, `'0'` in order
key12: `'\n'`, `'#'` in order

Keypad sends low-level Keyboard events with basic code of the key (from '0' to '9', '#' or '*') and high level TEXT_INPUT events with next key code mapping until key is validated (Key codes are scrolled in order, circularly). A key is validated when no new key has been pressed before validation delay or if an other physical key of the keypad is pressed. The delay starts when the key is pressed, so a key may be validated even if it is not yet released. When a key is validated Keypad sends KEY_VALIDATED event. The delay for key validation can be modified at any time using setDelay(int) Keypad uses 4 different modes to filter the letters that are scrolled.

- `NUM`: only digits are selected
- `ALPHA`: digits and letters are selected
- `CAP`: only capital letters and digits are selected
- `CAP1`: same as `CAP`, but must switch to `ALPHA` mode after the first character is validated

For example, assuming that low-level events are enabled (see Keyboard) pressing the '2' key twice rapidly and then waiting a little amount of time after activation delay will generate: KEY_DOWN '2'TEXT_INPUT 'a'KEY_UP '2'KEY_DOWN '2'TEXT_INPUT 'b'KEY_UP '2'KEY_VALIDATED (after a delay, see setDelay(int))

See Also:
Keyboard

Field Summary		Page
static int	**ALPHA** The ALPHA mode.	244
static int	**CAP** The CAP mode.	244
static int	**CAP1** The CAP1 mode.	244
static int	**KEY_VALIDATED** The KEY_VALIDATED event action.	243
static int	**NUM** The NUM mode.	244

Fields inherited from class ej.microui.io.Keyboard
KEY_DOWN, KEY_UP, TEXT_INPUT

Constructor Summary	Page
Keypad(int size) Keypads hold a buffer (potentially of size one) that stores the keys before they are used by the application (key associated to KEY_UP, KEY_DOWN and TEXT_INPUT event).	244

Method Summary		Page
abstract char[]	**getAssignment**(char key) Gets the complete array of chars associated with the specified key.	246
int	**getDelay**() Gets the delay.	245
int	**getEventType**() Gets the event type associated with the event generator	245
abstract String	**getLanguage**() Gets the currently selected language.	246
int	**getMode**() Gets the mode.	246
abstract String[]	**getSupportedLanguages**() Gets the supported languages for this keypad.	246
void	**setDelay**(int delay) Sets the delay The delay is the value between two press that allows to select a letter out of several.	245
abstract void	**setLanguage**(String language) Sets the language.	246
void	**setMode**(int mode) Sets the mode.	245
String[]	**supportedLanguages**() **Deprecated.** *use getSupportedLanguages()*	246

Methods inherited from class ej.microui.io.Keyboard
action, dropOnFull, getAction, getNextChar, nextChar, onlyTextInput, reset, send

Methods inherited from class ej.microui.EventGenerator
addToSystemPool, eventType, get, get, get, getID, getListener, removeFromSystemPool, setListener

Field Detail

KEY_VALIDATED

```
public static final int KEY_VALIDATED
```

The KEY_VALIDATED event action.
This event action is sent when last key is validated, meaning that no new key has been pressed during delay.
The value `8` is assigned to `KEY_VALIDATED`.

See Also:
Keyboard.action(int)

ALPHA

```
public static final int ALPHA
```

The ALPHA mode.
In ALPHA mode the keypad may return several letters and digits according to the number of consequent key press.
The value `0` is assigned to `ALPHA`.

See Also:
setMode(int)

NUM

```
public static final int NUM
```

The NUM mode.
In NUM mode the keypad may return several digits according to the number of consequent key press.
The value `1` is assigned to `NUM`.

See Also:
setMode(int)

CAP

```
public static final int CAP
```

The CAP mode.
In CAP mode only capital letters and digits are returned
The value `2` is assigned to `CAP`.

See Also:
setMode(int)

CAP1

```
public static final int CAP1
```

The CAP1 mode.
In CAP1 mode is the same has CAP mode, but automatically switch to ALPHA mode after the KEY_UP event
The value `3` is assigned to `CAP1`.

See Also:
CAP, setMode(int)

Constructor Detail

Keypad

```
public Keypad(int size)
```

Keypads hold a buffer (potentially of size one) that stores the keys before they are used by the application (key associated to KEY_UP, KEY_DOWN and TEXT_INPUT event).

Parameters:
`size` - of the buffer.
See Also:
Keyboard.dropOnFull()

Method Detail

getEventType

```
public int getEventType()
```

Gets the event type associated with the event generator

Overrides:
getEventType in class Keyboard
Returns:
Event.KEYPAD

---

setDelay

```
public void setDelay(int delay)
```

Sets the delay The delay is the value between two press that allows to select a letter out of several. The default value is 750ms.

Parameters:
`delay` - the delay to set
Throws:
`IllegalArgumentException` - if delay is negative or zero

---

getDelay

```
public int getDelay()
```

Gets the delay. The delay is the value between two press that allows to select a letter out of several. The default value is 750ms.

Returns:
the delay

---

setMode

```
public void setMode(int mode)
```

Sets the mode.

Parameters:
`mode` - one of `ALPHA`, `NUM`, `CAP`, `CAP1`
Throws:
`IllegalArgumentException` - if mode is not one of `ALPHA`, `NUM`, `CAP`, `CAP1`

---

getMode

```
public int getMode()
```

Gets the mode.

Returns:
one of ALPHA, NUM, CAP, CAP1

---

setLanguage

```
public abstract void setLanguage(String language)
```

Sets the language. Select the nearest mapping available on the platform.
The language must be a valid ISO language code. These codes are the lower-case, two-letter codes as defined by ISO-639.

Parameters:
`language` - the ISO language code

---

getLanguage

```
public abstract String getLanguage()
```

Gets the currently selected language.

Returns:
the ISO language code
See Also:
setLanguage(String)

---

supportedLanguages

```
public String[] supportedLanguages()
```

Deprecated. *use getSupportedLanguages()*

---

getSupportedLanguages

```
public abstract String[] getSupportedLanguages()
```

Gets the supported languages for this keypad.

Returns:
an array of supported ISO language codes

---

getAssignment

```
public abstract char[] getAssignment(char key)
```

Gets the complete array of chars associated with the specified key. The `key` must be one of `'0'`, `'1'`, `'2'`, `'3'`, `'4'`,

`'5'`, `'6'`, `'7'`, `'8'`, `'9'`, `'*'`, `'0'`, `'#'`.

Parameters:

`key` - the key

Returns:

the array of char associated with the `key`

Class Leds

ej.microui.io

```
java.lang.Object
  └
    ej.microui.io.Leds
```

```
public class Leds
extends Object
```

This class is used to manage all LEDs available on the platform. The available number of LEDs is known thanks to the method getNumberOfLeds(). A LED is identified by to its id. The range of the ids is from 0 to `getNumberOfLeds()-1`.

Field Summary		Page
static int	**MAX_INTENSITY** Maximum intensity that a LED can handle.	248
static int	**MIN_INTENSITY** Intensity value to turn off a LED.	248

Method Summary		Page
static int	**getLedIntensity**`(int ledId)` Gets the intensity of the specified LED.	249
static int	**getNumberOfLeds**`()` Returns the available number of LEDs.	249
static void	**ledOff**`(int ledId)` **Deprecated.** *use setLedOff(int)*	250
static void	**ledOn**`(int ledId)` **Deprecated.** *use setLedOn(int)*	249
static void	**setLedIntensity**`(int ledId, int intensity)` Controls the intensity of the specified LED.	249
static void	**setLedOff**`(int ledId)` Turns off the given LED.	250
static void	**setLedOn**`(int ledId)` Turns on the given LED.	249

Field Detail

MIN_INTENSITY

```
public static final int MIN_INTENSITY
```

Intensity value to turn off a LED.

MAX_INTENSITY

```
public static final int MAX_INTENSITY
```

Maximum intensity that a LED can handle. If a LED does not handle intensity, any valid intensity different from MIN_INTENSITY turns the LED on.

Method Detail

getNumberOfLeds

```
public static int getNumberOfLeds()
```

Returns the available number of LEDs. The range of valid led ids is [0..Leds.getNumberOfLeds()-1].

Returns:
the number of leds

---

setLedIntensity

```
public static void setLedIntensity(int ledId,
                                   int intensity)
```

Controls the intensity of the specified LED. If the id is invalid (out of range) the method has no effect.

Parameters:
`ledId` - the led identifier
`intensity` - the intensity to set on the led

---

getLedIntensity

```
public static int getLedIntensity(int ledId)
```

Gets the intensity of the specified LED. If the id is invalid (out of range) the method returns 0.

Parameters:
`ledId` - the led identifier
Returns:
the led intensity

---

ledOn

```
public static void ledOn(int ledId)
```

Deprecated. *use setLedOn(int)*

---

setLedOn

```
public static void setLedOn(int ledId)
```

Turns on the given LED. The effect is identical to `Leds.setLedIntensity(ledId, MAX_INTENSITY)`.

Parameters:
`ledId` - the led identifier

---

ledOff

```
public static void ledOff(int ledId)
```

Deprecated. *use setLedOff(int)*

---

setLedOff

```
public static void setLedOff(int ledId)
```

Turns off the given LED. The effect is identical to `Leds.setLedIntensity(ledId, MIN_INTENSITY)`.

Parameters:

`ledId` - the led identifier

Class Pointer

ej.microui.io

```
java.lang.Object
  └
    ej.microui.EventGenerator
      └
        ej.microui.io.Buttons
          └
            ej.microui.io.Pointer
```

```
public class Pointer
extends Buttons
```

A `Pointer` event generator represents a pointing device that is usually associated to a group of physical buttons. It reports the position of a pointing device as an x, y position within an area called pointer area. The size of the pointer area is set when the Pointer is constructed and cannot be modified. When a Pointer is pre-configured within a system its area is normally set to be the area of the Display with which it is associated. The associated MicroUI event type is `Event.POINTER`.

The Pointer can be asked for the absolute position, expressed in terms of the pointer area with which it was constructed. It can also be asked for scaled co-ordinates (getX(), getY()). The scaled area is set using the setScale(int, int) method. By default there is no scaling (scaled area is the pointer area).

It is also possible to specify, using setOrigin(int, int), an offset to be applied to the co-ordinates returned by `getX()` and `getX()`. For example, if the origin is set to be (20, 30) then the x position returned will be the absolute x position - 20, and the y position will be the absolute y position - 30. By default there is no offset.

If both scaling and origin adjustment are specified then the origin offset is first applied to the absolute position then the scaling is applied.

If a flying image is associated with the pointer, the generator manages it automatically its location within the scale area when the pointer has moved.

Field Summary		Page
static int	**DRAGGED** The "dragged" action.	253
static int	**ENTERED** The "entered" action.	253
static int	**EXITED** The "exited" action.	253
static int	**MOVED** The "move" action.	253

Fields inherited from class ej.microui.io.Buttons
CLICKED, DOUBLE_CLICKED, LONG, PRESSED, RELEASED, REPEATED

Constructor Summary	Page
Pointer`(int width, int height)` Constructor with a specified area range (`width` and `height`) that does not support click, doubleClick nor elapsedTime for any of its buttons.	254
Pointer`(int nbButtons, int width, int height)` Constructor with a specified area range (`width` and `height`) where elapsedTime, click and doubleClick features are supported and enabled for the first `nbButtons` (doubleClick feature is initialized with a 200ms delay).	253

Method Summary		Page
int	**getAbsoluteHeight** ()	254
int	**getAbsoluteWidth** ()	254
int	**getAbsoluteX** () Returns the last available absolute x coordinate in pointer area	255
int	**getAbsoluteY** () Returns the last available absolute y coordinate in pointer area	255
int	**getEventType** () Gets the event type associated with the event generator	254
FlyingImage	**getFlyingImage** () Gets the FlyingImage associated to this Pointer.	256
int	**getHeight** ()	256
int	**getWidth** ()	256
int	**getX** () Returns the last available x coordinate in scaled area (after applying any origin offset and the scale factor).	254
int	**getY** () Returns the last available y coordinate in scaled area (after applying any origin offset and the scale factor).	255
static boolean	**isDragged** (int event) Tells if a pointer event is a drag event.	258
static boolean	**isEntered** (int event) Tells if a pointer event is an enter event.	258
static boolean	**isExited** (int event) Tells if a pointer event is an exit event.	258
static boolean	**isMoved** (int event) Tells if a pointer event is a move event.	257
void	**move** (int x, int y) Stores the given position and sends a MicroUI Event to the Pointer's listener.	257
void	**reset** (int x, int y) Stores the given position.	257
void	**send** (int action, int buttonID) Sends a MicroUI event for the given action on given button to the listener of the Pointer.	257
void	**setFlyingImage** (FlyingImage flyingImage) A Pointer generator manages positioning an image automatically if such image is set.	256
void	**setFlyingImage** (FlyingImage flyingImage, int anchorX, int anchorY) A Pointer generator manages positioning an image automatically if such image is set.	256
void	**setOrigin** (int x0, int y0) Sets an origin offset.	255
void	**setScale** (Display display) Sets a display area for scaled area.	255
void	**setScale** (int areaWidth, int areaHeight) Sets a scaled area.	255

Methods inherited from class ej.microui.io.Buttons
action, buttonID, clickEnabled, doubleClickEnabled, elapsedTime, enableClick, enableDoubleClick, getAction, getButtonID, isClicked, isDoubleClicked, isLong, isPressed, isReleased, isRepeated, supportsExtendedFeatures

Methods inherited from class ej.microui.EventGenerator
addToSystemPool, eventType, get, get, get, getID, getListener, removeFromSystemPool, setListener

Field Detail

MOVED

```
public static final int MOVED
```

The "move" action.

The value `0x06` is assigned to `MOVE`.

DRAGGED

```
public static final int DRAGGED
```

The "dragged" action.

The value `0x07` is assigned to `DRAGGED`.

ENTERED

```
public static final int ENTERED
```

The "entered" action.

The value `0x08` is assigned to `ENTERED`.

EXITED

```
public static final int EXITED
```

The "exited" action.

The value `0x09` is assigned to `EXITED`.

Constructor Detail

Pointer

```
public Pointer(int nbButtons,
               int width,
               int height)
```

Constructor with a specified area range (`width` and `height`) where elapsedTime, click and doubleClick features are supported and enabled for the first `nbButtons` (doubleClick feature is initialized with a 200ms delay).

Parameters:

`width` - area width
`height` - area height

Pointer

```
public Pointer(int width,
               int height)
```

Constructor with a specified area range (`width` and `height`) that does not support click, doubleClick nor elapsedTime for any of its buttons. The effect is identical to:
`new Pointer(0, width, height).`

Parameters:
`width` - area width
`height` - area height

Method Detail

getAbsoluteWidth

```
public int getAbsoluteWidth()
```

Returns:
pointer area width

---

getAbsoluteHeight

```
public int getAbsoluteHeight()
```

Returns:
pointer area height

---

getEventType

```
public int getEventType()
```

Gets the event type associated with the event generator

Overrides:
getEventType in class Buttons
Returns:
the event type

---

getX

```
public int getX()
```

Returns the last available x coordinate in scaled area (after applying any origin offset and the scale factor).

Returns:
last available x coordinate
See Also:
`getAbsoluteX`

---

getY

```
public int getY()
```

Returns the last available y coordinate in scaled area (after applying any origin offset and the scale factor).

Returns:
last available y coordinate
See Also:
`getAbsoluteY`

---

getAbsoluteX

```
public int getAbsoluteX()
```

Returns the last available absolute x coordinate in pointer area

Returns:
last available absolute x coordinate

---

getAbsoluteY

```
public int getAbsoluteY()
```

Returns the last available absolute y coordinate in pointer area

Returns:
last available absolute y coordinate

---

setScale

```
public void setScale(Display display)
```

Sets a display area for scaled area.
same as `setScale(display.getWidth(), display.getHeight())`

See Also:
setScale(int, int)

---

setScale

```
public void setScale(int areaWidth,
                     int areaHeight)
```

Sets a scaled area. The x position returned by getX() is scaled so that it returns a value between 0 and areaWidth-1. The x position returned by getY() is scaled so that it returns a value between 0 and areaHeight-1.

---

setOrigin

```
public void setOrigin(int x0,
                      int y0)
```

Sets an origin offset. This offset is subtracted from the absolute position (before applying any scaling) when reporting x and y positions.

---

getFlyingImage

```
public FlyingImage getFlyingImage()
```

Gets the FlyingImage associated to this Pointer.

Returns:
null if currently no associated FlyingImage

---

setFlyingImage

```
public void setFlyingImage(FlyingImage flyingImage)
```

A Pointer generator manages positioning an image automatically if such image is set. Same as `setFlyingImage(flyingImage, 0, 0)`

See Also:
setFlyingImage(FlyingImage, int, int)

---

setFlyingImage

```
public void setFlyingImage(FlyingImage flyingImage,
                           int anchorX,
                           int anchorY)
```

A Pointer generator manages positioning an image automatically if such image is set. The flying image is moved as the pointer position moves. Note that associating a FlyingImage with a Pointer automatically causes it to be shown - there is no need to call FlyingImage.show().

Set the anchor of the image to `anchorX` and `anchorY`. If the pointer already has a flying image set the old image is hidden and the new image replaces it.

---

getWidth

```
public int getWidth()
```

Returns:
the width of the scaled area
See Also:
`getX`

---

getHeight

```
public int getHeight()
```

Returns:
the height of the scaled area
See Also:
`getY`

move

```
public void move(int x,
                 int y)
```

Stores the given position and sends a MicroUI Event to the Pointer's listener. Coordinates are clipped to the pointer area.

Parameters:
`x` - the x coordinate
`y` - the y coordinate

---

reset

```
public void reset(int x,
                  int y)
```

Stores the given position. The Pointer's listener is not notified. Coordinates are clipped to the pointer area.

Parameters:
`x` - the x coordinate
`y` - the y coordinate

---

send

```
public void send(int action,
                 int buttonID)
```

Sends a MicroUI event for the given action on given button to the listener of the Pointer. Pointer will generate a Buttons.CLICKED and/or Buttons.DOUBLE_CLICKED events if the matching button's feature is enabled.

This method is useful when other input mechanisms wish to simulate button actions.

Overrides:
send in class Buttons
Parameters:
`action` - the button's action: Buttons.PRESSED, Buttons.RELEASED, Buttons.LONG, Buttons.REPEATED.
`buttonID` - the button on which the action occurred

---

isMoved

```
public static boolean isMoved(int event)
```

Tells if a pointer event is a move event.

Parameters:
`event` - the pointer event.
Returns:
true if the pointer event is a move event.

---

isDragged

```
public static boolean isDragged(int event)
```

Tells if a pointer event is a drag event.

Parameters:
`event` - the pointer event.
Returns:
true if the pointer event is a drag event.

---

isEntered

```
public static boolean isEntered(int event)
```

Tells if a pointer event is an enter event.

Parameters:
`event` - the pointer event.
Returns:
true if the pointer event is an enter event.

---

isExited

```
public static boolean isExited(int event)
```

Tells if a pointer event is an exit event.

Parameters:
`event` - the pointer event.
Returns:
true if the pointer event is an exit event.

Class PointerButtons

ej.microui.io

```
java.lang.Object
  └
    ej.microui.EventGenerator
      └
        ej.microui.io.Buttons
          └
            ej.microui.io.PointerButtons
```

```
public class PointerButtons
extends Buttons
```

Deprecated.

A PointerButtons event generator is usually associated to a group of physical buttons that are linked to a Pointer. It generates Event.POINTER_BUTTON events. When an event is sent by this generator, the related Pointer instance can be retrieved in order to get the `(x,y)` coordinates of the button action.Examples:

- buttons of a mouse
- press/release on a touch device

Since:

1.3.2

Fields inherited from class ej.microui.io.Buttons
CLICKED, DOUBLE_CLICKED, LONG, PRESSED, RELEASED, REPEATED

Constructor Summary	Page
PointerButtons(Pointer p) Creates a PointerButtons instance with the associated Pointer instance on which this generator is associated.	260
PointerButtons(int nbButtons, Pointer p) Creates a PointerButtons instance where the first nbButtons support the elapsedTime, click and doubleClick features.	260

Method Summary		Page
int	**getEventType**() Gets the event type associated with the event generator	260
Pointer	**getPointer**()	260

Methods inherited from class ej.microui.io.Buttons
action, buttonID, clickEnabled, doubleClickEnabled, elapsedTime, enableClick, enableDoubleClick, getAction, getButtonID, isClicked, isDoubleClicked, isLong, isPressed, isReleased, isRepeated, send, supportsExtendedFeatures

Methods inherited from class ej.microui.EventGenerator
addToSystemPool, eventType, get, get, get, getID, getListener, removeFromSystemPool, setListener

Constructor Detail

PointerButtons

```
public PointerButtons(Pointer p)
```

Creates a PointerButtons instance with the associated Pointer instance on which this generator is associated.

See Also:
Buttons.Buttons()

PointerButtons

```
public PointerButtons(int nbButtons,
                      Pointer p)
```

Creates a PointerButtons instance where the first `nbButtons` support the elapsedTime, click and doubleClick features.

See Also:
Buttons.Buttons(int)

Method Detail

getPointer

```
public Pointer getPointer()
```

Returns:
the Pointer instance on which this group of buttons is associated.

---

getEventType

```
public int getEventType()
```

Gets the event type associated with the event generator

Overrides:
getEventType in class Buttons
Returns:
Event.POINTER_BUTTON

Class Screen

ej.microui.io

```
java.lang.Object
  └
    ej.microui.io.Screen
```

Direct Known Subclasses:
AlphaNumericDisplay, Display

```
abstract public class Screen
extends Object
```

A `Screen` represents an hardware screen with its characteristics: width, height, number of supported colors, backlight etc. According to the kind of screen, these characteristics have a different interpretation. For instance, for a QVGA screen the width is represented by a number of pixels whereas for an alpha numeric screen, the width is the number of characters per line.

Constructor Summary	Page
Screen()	262

Method Summary		Page
int	**getBacklight**() Returns the backlight intensity of the screen.	264
int	**getBacklightColor**() Returns the current backlight color.	264
int	**getContrast**() Returns the contrast of the screen.	263
Listener	**getEventHandler**() Returns the `Screen`'s `Listener` or null.	262
abstract int	**getHeight**() Each screen has an area where to render.	262
abstract int	**getNumberOfColors**() Returns the number of available colors for this screen.	262
abstract int	**getWidth**() Each screen has an area where to render.	262
boolean	**hasBacklight**() Tells whether the screen has a backlight.	263
boolean	**isColor**() Tells whether the screen offers color.	262
void	**setBacklight**(int backlight) Sets the intensity of the backlight of the screen.	263
void	**setBacklightColor**(int rgbColor) Sets the current backlight color, if it is allowed by implementation.	264
void	**setContrast**(int contrast) Sets the contrast of the screen.	263
void	**switchBacklight**(boolean on) Switches on or off the backlight of the screen.	263

Constructor Detail

Screen

```
public Screen()
```

Method Detail

getEventHandler

```
public Listener getEventHandler()
```

Returns the `Screen`'s `Listener` or null.

Returns:
the `Screen`'s `Listener` or null.

---

getWidth

```
public abstract int getWidth()
```

Each screen has an area where to render. The width has got a specific unit according the type of screen.

Returns:
the width of the screen

---

getHeight

```
public abstract int getHeight()
```

Each screen has an area where to render. The height has got a specific unit according the type of screen.

Returns:
the height of the screen

---

isColor

```
public boolean isColor()
```

Tells whether the screen offers color. By default, this method returns false; sub-classes should overwrite this default behavior.

Returns:
true if screen has color

---

getNumberOfColors

```
public abstract int getNumberOfColors()
```

Returns the number of available colors for this screen.

The number of colors is 2 for monochrome [screens] LCD displays.
The number of colors is 1 for monochrome AlphaNumericDisplays

Returns:
the number of colors.

---

setContrast

```
public void setContrast(int contrast)
```

Sets the contrast of the screen. Subclasses should overwrite the default behavior which does nothing. `contrast` value range is 0-100.

Parameters:
`contrast` - the new value of the contrast.

---

getContrast

```
public int getContrast()
```

Returns the contrast of the screen. By default, this method returns 0; sub-classes should overwrite this default behavior.

Returns:
the current contrast of the screen (range 0-100)

---

hasBacklight

```
public boolean hasBacklight()
```

Tells whether the screen has a backlight. By default, this method returns false; sub-classes should overwrite this default behavior.

Returns:
if screen has backlight

---

switchBacklight

```
public void switchBacklight(boolean on)
```

Switches on or off the backlight of the screen. Sub-classes should overwrite the default behavior which does nothing.

Parameters:
`on` - switch on the backlight if `true`; switch off the backlight if `false`

---

setBacklight

```
public void setBacklight(int backlight)
```

Sets the intensity of the backlight of the screen. Sub-classes should overwrite the default behavior which does

nothing. `backlight` value range is 0-100

Parameters:
`backlight` - the new value of the backlight

---

getBacklight

```
public int getBacklight()
```

Returns the backlight intensity of the screen. By default, this method returns 0; sub-classes should overwrite this default behavior.

Returns:
the backlight of the screen

---

setBacklightColor

```
public void setBacklightColor(int rgbColor)
```

Sets the current backlight color, if it is allowed by implementation. By default, this method does nothing and sub-classes should overwrite this default behavior.

Parameters:
`rgbColor` - the color to set

---

getBacklightColor

```
public int getBacklightColor()
```

Returns the current backlight color. Returned value is interpreted as a 24-bit RGB color, where the eight less significant bits matches the blue component, the next eight bits matches the green component and the next eight bits matches the red component. By default, this method returns 0xFFFFFF (white) and sub-classes should overwrite this default behavior.

Returns:
the color of the backlight

Class States

ej.microui.io

```
java.lang.Object
  └
    ej.microui.EventGenerator
      └
        ej.microui.io.States
```

```
public class States
extends EventGenerator
```

A States event generator is usually associated to a group of physical devices holding a position (switch, rotary wheel encoder, ...) and allow to generate events relating to them. This class generates Event.STATE events and allows to retrieve for each state its current value. Each instance can manage at most `256` states and each state can have a value between `0` and `255`. A state has a unique ID between `0` and nbStates()`-1`

Constructor Summary	Page
States`(int[] nbValues, int[] initialValues)` Creates a states generator.	266

Method Summary		Page
int	**currentValue**`(int stateID)` **Deprecated.** *use getCurrentValue(int)*	267
int	**getCurrentValue**`(int stateID)` Gets the current value of the given state	267
int	**getEventType**`()` Gets the event type associated with the event generator	267
int	**getNumberOfStates**`()` Gets the number of states managed by this instance.	268
int	**getNumberOfValues**`(int stateID)` Gets the total number of values for the given state	267
static int	**getStateID**`(int event)` Gets the state's id held by the state event.	266
static int	**getStateValue**`(int event)` Gets the state's value held by the state event.	266
int	**nbStates**`()` **Deprecated.** *use getNumberOfStates()*	267
int	**nbValues**`(int stateID)` **Deprecated.** *use getNumberOfValues(int)*	267
void	**send**`(int stateID, int value)` Stores the given state value and sends a MicroUI Event.STATE to the States's listener.	268
static int	**stateID**`(int event)` **Deprecated.** *use getStateID(int)*	266
static int	**stateValue**`(int event)` **Deprecated.** *use getStateValue(int)*	266

Methods inherited from class ej.microui.EventGenerator
addToSystemPool, eventType, get, get, get, getID, getListener, removeFromSystemPool, setListener

Constructor Detail

States

```
public States(int[] nbValues,
              int[] initialValues)
```

Creates a states generator.

Parameters:

`nbValues` - number of values for each state
`initialValues` - initial value for each state

Throws:

`NullPointerException` - if one of the parameters is null.
`IllegalArgumentException` - if both arrays don't have the same length
`IndexOutOfBoundsException` - if arrays length is greater than 255,
if `nbValues[i] 0` or `nbValues[i] > 255`,
if `initialValues[i] 0` or `initialValues[i] >= nbValues[i]`

Method Detail

stateID

```
public static int stateID(int event)
```

Deprecated. *use getStateID(int)*

---

getStateID

```
public static int getStateID(int event)
```

Gets the state's id held by the state event.

Returns:

id between `0` and `255`

---

stateValue

```
public static int stateValue(int event)
```

Deprecated. *use getStateValue(int)*

---

getStateValue

```
public static int getStateValue(int event)
```

Gets the state's value held by the state event.

Returns:

value between `0` and `255`

---

getEventType

```
public int getEventType()
```

Gets the event type associated with the event generator

Overrides:
getEventType in class EventGenerator
Returns:
the event type

---

nbValues

```
public int nbValues(int stateID)
```

Deprecated. *use getNumberOfValues(int)*

---

getNumberOfValues

```
public int getNumberOfValues(int stateID)
```

Gets the total number of values for the given state

Throws:
`IndexOutOfBoundsException` - when stateID is out of `[0,nbStates()-1]`

---

currentValue

```
public int currentValue(int stateID)
```

Deprecated. *use getCurrentValue(int)*

---

getCurrentValue

```
public int getCurrentValue(int stateID)
```

Gets the current value of the given state

Returns:
a number between `0` and nbValues(int)`-1`
Throws:
`IndexOutOfBoundsException` - when stateID is out of `[0,nbStates()-1]`

---

nbStates

```
public int nbStates()
```

Deprecated. *use getNumberOfStates()*

---

getNumberOfStates

```
public int getNumberOfStates()
```

Gets the number of states managed by this instance.

---

send

```
public void send(int stateID,
                 int value)
```

Stores the given state value and sends a MicroUI Event.STATE to the States's listener.

Throws:

`IndexOutOfBoundsException` - when `stateID` is out of `[0,`nbStates()`-1]`, when `value` is out of `[0,`nbValues(int)`-1]`

Class View

ej.microui.io

```
java.lang.Object
  └
    ej.microui.io.ComponentView
      └
        ej.microui.io.View
```

All Implemented Interfaces:

Listener

```
abstract public class View
extends ComponentView
implements Listener
```

`View` is an `abstract` class which extends `ComponentView`. It represents a visible zone on a `Viewable`. The application must create its own views by extending the class `View`.

To be visible a `View` must be connected, directly or indirectly via one or more `CompositeView`'s, to a `Viewable`.

The `View` position is relative to the `CompositeView` or `Viewable` it is in.

`Views` are designed to be used in conjunction with `Models`. A `View` is a `Listener` for changes to its model. That means a `View` can be notified by its model when the model changes. A `View` has no default model: use the method setModel(Model) to specify one. A view can have only one model.

The default behavior of the `performAction()` methods is to request a repaint. Subclasses should override these methods if they want to extend the behavior.

See Also:

Model, Listener, CompositeView

Constructor Summary	Page
View`(int x, int y, int width, int height)` Creates a view with given attributes.	270

Method Summary		Page
Model	**getModel**`()` Get the `View`'s model.	270
void	**performAction**`()` Something has changed in the model.	270
void	**performAction**`(int value)` Something has changed in the model.	270
void	**performAction**`(int value, Object object)` Something has changed in the model.	271
void	**setModel**`(Model model)` Sets the view's model.	270

Methods inherited from class ej.microui.io.ComponentView
getAbsoluteX, getAbsoluteY, getHeight, getWidth, getX, getY, isVisible, paint, repaint, repaint, update, updateLocation, updateSize

Constructor Detail

View

```
public View(int x,
            int y,
            int width,
            int height)
```

Creates a view with given attributes. Note that the system does not check if these attributes are correct or not.

Parameters:

`x` - the relative x coordinate
`y` - the relative y coordinate
`width` - the zone width
`height` - the zone height

Method Detail

setModel

```
public void setModel(Model model)
```

Sets the view's model. If the view is already registered as a model listener, it is first removed from the model's listener's list. If the new model is not `null`, this view is added as a listener to the model.

Parameters:

`model` - the new view's model or `null` to remove the listener from the current model

---

getModel

```
public Model getModel()
```

Get the `View`'s model.

Returns:

the `View`'s model.

---

performAction

```
public void performAction()
```

Something has changed in the model. Triggers a `repaint` event.

Specified by:

performAction in interface Listener

---

performAction

```
public void performAction(int value)
```

Something has changed in the model. Triggers a `repaint` event.

Specified by:
performAction in interface Listener
Parameters:
`value` - given by the model.

---

performAction

```
public void performAction(int value,
                          Object object)
```

Something has changed in the model. Triggers a `repaint` event.

Specified by:
performAction in interface Listener
Parameters:
`value` - given by the model.
`object` - given by the model

Class Viewable

ej.microui.io

```
java.lang.Object
  └
    ej.microui.io.Displayable
      └
        ej.microui.io.Viewable
```

```
public class Viewable
extends Displayable
```

`Viewable` is a `Displayable` subclass that contains a hierarchy of `ComponentView` for its rendering. A `Viewable` should have a `ComponentView` to draw something. A `ComponentView` defines a screen area that can be rendered.

By default, a new `Viewable` object has no listener.

See Also:

Displayable, ComponentView

Constructor Summary	Page
Viewable (Display `display`) The newly created viewable is built for the given `Display` and is hidden.	272

Method Summary		Page
ComponentView	**getComponentView** () Returns the current `ComponentView` of the viewable.	274
Listener	**getEventListener** () Returns the `Viewable`'s `Listener` (may be `null`).	273
CompositeView	**newCompositeView** () Creates a new `CompositeView` for the viewable, set it as the viewable's `ComponentView` and returns it.	274
void	**paint** (GraphicsContext `g`) Paint the viewable's `ComponentView` and all its children.	273
void	**performAction** (`int event`) Sends the given event to the `Viewable`'s `Listener`.	273
void	**setComponentView** (ComponentView `view`) Defines the viewable's `ComponentView`.	274
void	**setEventListener** (Listener `l`) Sets a `Listener` for the `Viewable`.	273

Methods inherited from class ej.microui.io.Displayable
getDisplay, hide, hideNotify, isShown, repaint, show, showNotify

Constructor Detail

Viewable

```
public Viewable(Display display)
```

The newly created viewable is built for the given `Display` and is hidden.

Parameters:
`display` - the display for which the viewable is created
Throws:
`NullPointerException` - if `display` is `null`

Method Detail

setEventListener

`public void setEventListener(Listener l)`

Sets a `Listener` for the `Viewable`. Any previous `Listener` is replaced. The listener may be set to `null`, resulting in no event listener defined for the `Viewable`.

Parameters:
`l` - the new listener, or `null`.

---

getEventListener

`public Listener getEventListener()`

Returns the `Viewable`'s `Listener` (may be `null`).

Returns:
the `Viewable`'s `Listener`.

---

performAction

`public void performAction(int event)`

Sends the given event to the `Viewable`'s `Listener`.

Overrides:
performAction in class Displayable
Parameters:
`event` - the event to handle

---

paint

`public void paint(GraphicsContext g)`

Paint the viewable's `ComponentView` and all its children.

If there is no `ComponentView` on the viewable or if the viewable is not visible, this call has no effect.

Overrides:
paint in class Displayable

---

getComponentView

`public` ComponentView **`getComponentView`**`()`

Returns the current `ComponentView` of the viewable.

Returns:
a `ComponentView`

---

newCompositeView

`public` CompositeView **`newCompositeView`**`()`

Creates a new `CompositeView` for the viewable, set it as the viewable's `ComponentView` and returns it. If another `ComponentView` is set on this viewable, it is detached from it. The newly created `CompositeView` covers the full viewable's area and is set as the viewable's `ComponentView`.

Returns:
the newly created `CompositeView`

---

setComponentView

`public void` **`setComponentView`**`(`ComponentView `view)`

Defines the viewable's `ComponentView`. If the specified `view` is already set on another viewable, it throws an `IllegalArgumentException`. If another `ComponentView` is set on this viewable, it is detached from it. The argument `view` may be `null`, in that case the viewable will no longer have a `ComponentView`. This does not trigger a Displayable.repaint() of the Viewable.

Parameters:
`view` - the new `ComponentView` for the viewable
Throws:
`IllegalArgumentException` - if the specified `view` or one of its children is already connected to a Viewable.