ESR Consortium

MicroUI-2.0

Micro User Interface
Profile Specification

ESR002

Reference: ESR-SPE-002-MicroUI
Version: 2.0
Rev: B

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 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 claim any right in this ESR. You are free to use this ESR to make any clean room implementations or derivative work as long as You don't 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 2.0 , MicroUI 2.0.* 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.

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. 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).

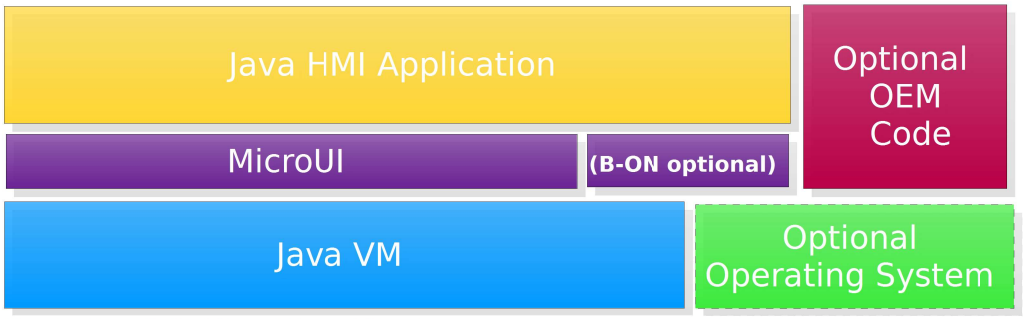


Illustration 2-2: MicroUI architecture

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, with depth is 1-bit or more.
- **Input**: optional
  - Any user-input mechanisms: buttons, rotary switches, keyboards, multi-touch screens, mouse-like-pointers, etc.
- **Output**: optional
  - Any kind of LEDs.

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, an LCD display, a set of buttons, a set of LEDs. 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 object: 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.

2.2.3 Specification

This section sets out the requirements of this specification. Compliant MicroUI 2.0 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 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

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.

specification and are implementation specific. If no such facility is provided then the mappings will be fixed.

3 MICROUI BASIC CONCEPTS

3.1 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, an 8-bit value that forms the most significant byte of the `int`, followed by 8-bit which identify the generator of the event, followed by 16-bit of data, as described in Illustration 3-1.

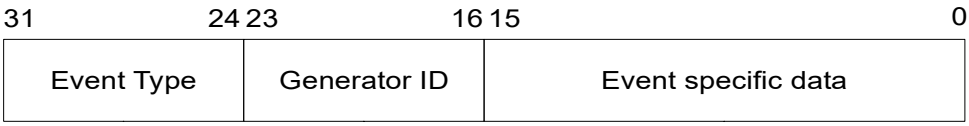


Illustration 3-1: MicroUI int-based event format

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-2.0 defines 5 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	4.10.1
1	BUTTON	2 states buttons events	YES	4.10.2
2	KEYBOARD	Input letter events	NO	4.10.3
3	POINTER	Pointing device events	NO	4.10.4
4	KEYPAD	Standard keypads events	NO	4.10.5
5	STATE	State device events	YES	4.10.6

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 3.2, 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.

3.2 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 3-2.

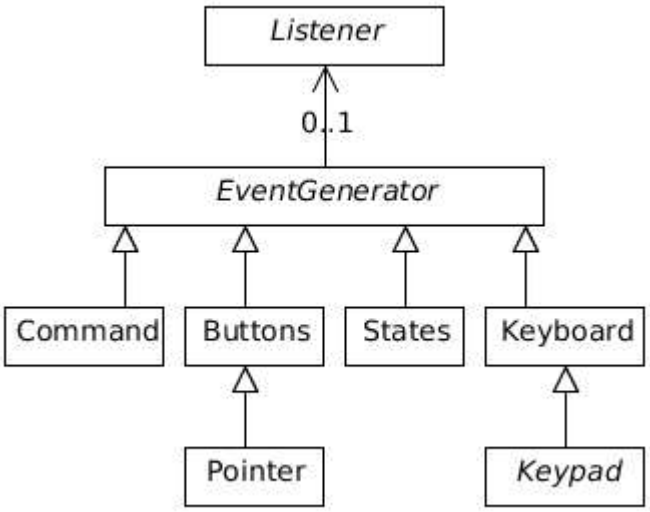


Illustration 3-2: Built-in event generators

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].

3.2.1 Event handling

An event generator is normally associated with an `EventHandler` that handles the events it generates. When an event occurs, the event handler's `handleEvent(int)` method is called. Note

2 Typical devices have less than 10 registered event generators.

that an application must call `EventGenerator.setEventHandler(EventHandler)` to set the event handler.

3.2.2 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].

3.3 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 interface `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. An abstract integer based pump (`IntPump`) manipulates 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.

The IntPump can then use a queue that contains the data to pump. The `FIFOIntQueue` returns the oldest data or blocks until data is available. The queue will expand to hold as many data items as the maximum size of the queue. When the queue is full, the new data is not added and an exception is thrown.

4 MICROUI ARCHITECTURE

4.1 Graphical display

4.1.1 Display class

The class `Display` represents a physical display and defines characteristics such as:

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

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)`)

3 This facility is intensively used by automated test suites

- *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 display are redirected to the currently shown `Displayable`.

4.1.2 Display characteristics

A graphical display is also sometimes called pixelated display. `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.

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 4.1.6).

4.1.3 Displays FIFO events queue

Each `Display` instance manages 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 3.1, 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 event handler (controller), through `handleEvent(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()` 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.

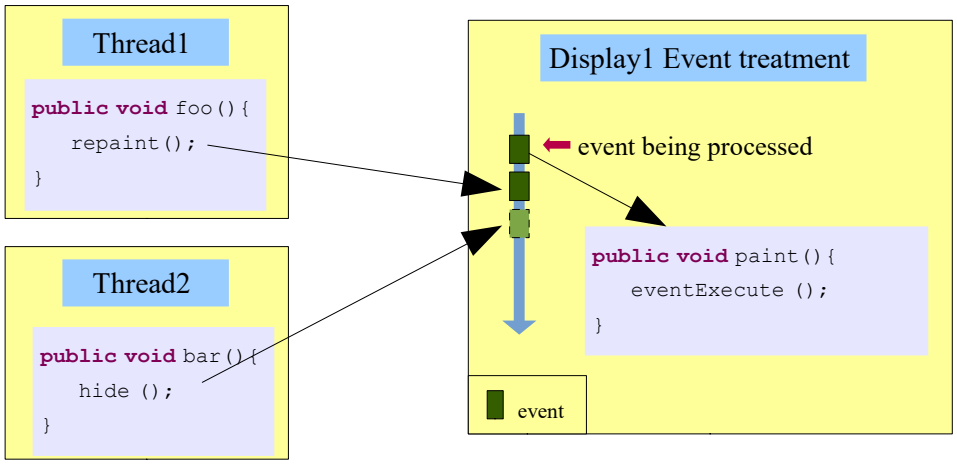
- `Displayable` *show* and *hide events*: these are issued when a `Displayable` object is placed on a display (or removed from a display) using `Displayable.show()` and `Displayable.hide()`.
- `FlyingImage` *show* and *hide events*: these are issued when a `FlyingImage` object is placed on a display (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

4.1.4 Event producers and event consumers

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

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 encourage MicroUI implementations to start giving precedence to the consumers before the pending event buffer of a display is completely full.

4.1.5 Displayable

4.1.5.1 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`.

In order to become visible (or invisible), the application needs to send `show()` (respectively `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).

4.1.5.2 Rendering

When a `Display` issues a repaint for a `Displayable` (`Displayable.repaint()`), the whole `Displayable` gets repainted. `The Displayable subclasses` can implement the rendering of the `Displayable` overriding the `paint(GraphicsContext)` method (section 4.1.6 for more information about drawing).

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 4.1.3).

In order to perform rendering operations, a `GraphicsContext` object holds a color, a font, a translation, and a clip rectangle. According to the MicroUI implementation, the drawings are aliased or antialiased. 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 `Displayable` 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 4.2).
- *a translation*, defines the origin of the `GraphicsContext` relative to the upper left corner of the `Displayable` 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`, `getARGB`, `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 4.1.6.4).

4.1.5.3 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. An event handler can handle this event if registered by the `Displayable getController()` method.

4.1.6 Drawing

4.1.6.1 Coordinates

The origin of the coordinate system is at the upper left corner (Illustration 4-2). 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)`.

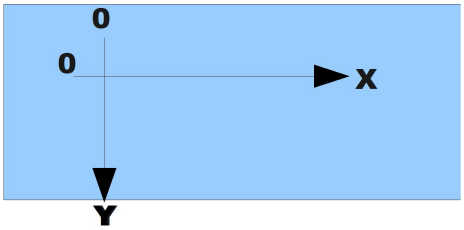


Illustration 4-2: Coordinate system

4.1.6.2 Serialized 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 displayable, 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 displayable 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.

4.1.6.3 Direct Drawing

It is possible to draw on a display without using the `Displayable`'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 4.1.3). If this is done there is no interaction with the current displayables parameters (color, clipping, etc.). This mechanism should be used with caution, for instance, to draw an abnormal message such as a warning.

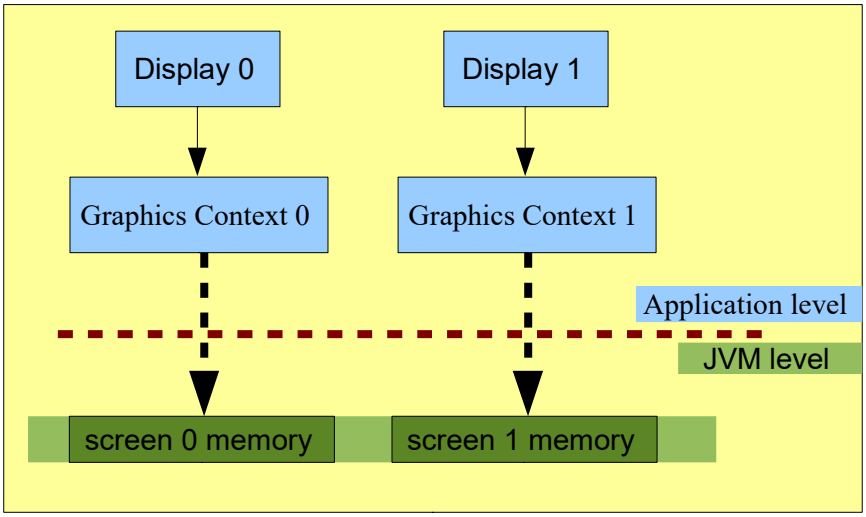


Illustration 4-3: Default display architecture

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 the 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 4-4). 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 4.1.6.4).

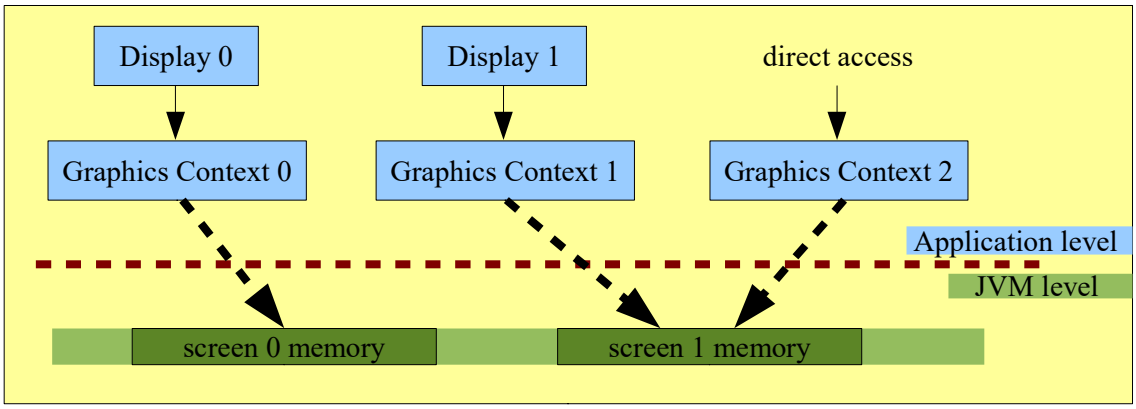


Illustration 4-4: Displays and graphics context direct access

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

4.1.6.4 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 4.1.3), 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.

4.1.6.5 Drawing algorithms

4.1.6.5.1 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()`.

4.1.6.5.2 Background Color

MicroUI offers a primitive to set the current background color. This color is used in these cases:

- Antialiasing: when the MicroUI implementation applies the antialiasing on the drawings, the background color is used to merge the foreground color with the transparency level of the pixel to draw.
- Font: when a font uses more than one bit per pixel, the background color is used to merge the foreground color with the transparency level stored in the font pixels.

When the background color is reseted (not set), the previous algorithms have to read the destination pixel color before merging and drawing the new pixel. This operation can take more time according the MicroUI implementation.

4.1.6.5.3 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 4-5 (P *n* represent the order of the points to give to the draw or fill polygon method).

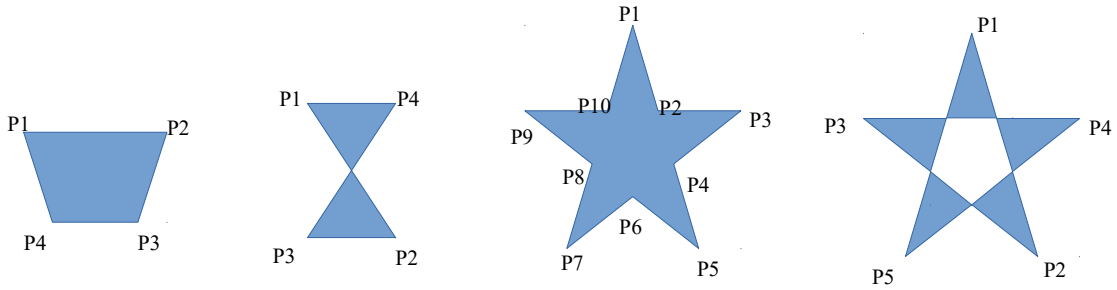


Illustration 4-5: Example of polygons

4.1.6.5.4 Deformed images

The `ImageDeformation` class 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. `ImageDeformation.draw()` method controls the modification through the image's four corners.

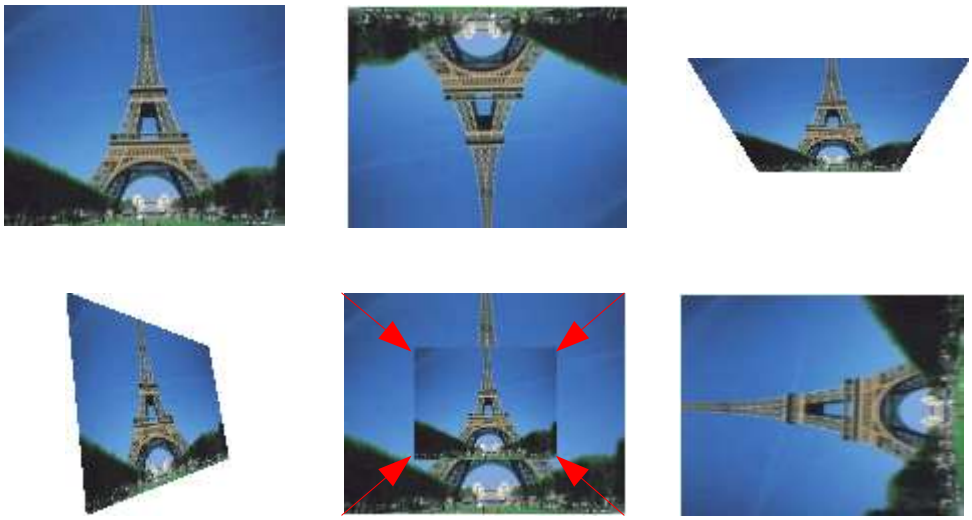


Illustration 4-6: Example of deformed images

4.2 Fonts

4.3 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.

4.3.1 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 5.1).

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 characterize 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[4] 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 4-7).

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()`.
- `getIdentifiers()`, the font hold an array of identifiers (see section 5.1).
- `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 4-7: Rendering example of an unknown character

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

An application can get a specific font using the method `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.

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

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).

A font can be resized at runtime using the method `setRatios()`. This call forces to change the font rendering. All next font drawings will use the new ratios. To reset them, use the method `resetRatios()`.

4.3.2 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 Error: Reference source not found). 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 Error: Reference source not found). 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.

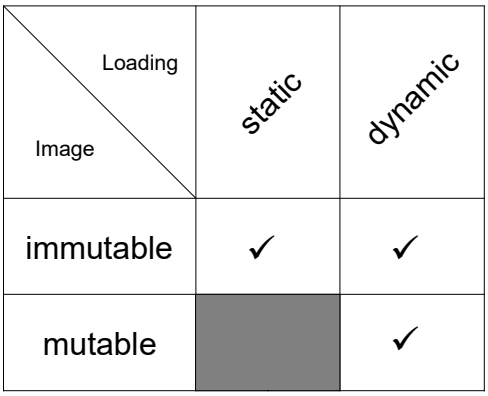


Illustration 4-8: Image creation policies

4.3.3 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`).

4.3.4 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`)

4.3.5 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.

4.3.6 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.

4.4 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.

4.5 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 4.10.4).

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 4.1.6.4)
- 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.

4.6 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.

4.7 Startup and termination

4.7.1 MicroUI startup

The MicroUI implementation is not started automatically, it must be started with `MicroUI.start()` method.

4.7.2 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.

4.8 System Properties

The MicroUI specification defines a set of properties, described in Table 4-1.

Property	Description
`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 4-1: System Properties

4.9 Error management

MicroUI defines a way to notify users of erroneous behavior using a `UncaughtExceptionHandler`. By default MicroUI prints the errors on standard error stream (`System.err()`). When an `UncaughtExceptionHandler` is set by `MicroUI.setUncaughtExceptionHandler()`, all errors are sent to this handler.

4.10 Built-in events and event generators

4.10.1 COMMAND

4.10.1.1 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 4-9.

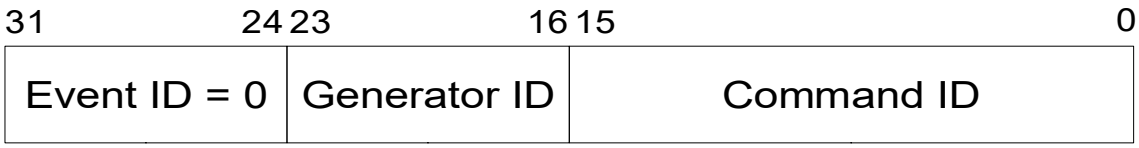


Illustration 4-9: Command event format

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 4-2). 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 4-2: Predefined commands

4.10.1.2 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.

4.10.2 BUTTON

4.10.2.1 Event format

The data of buttons event is composed of an action and a button id (Illustration 4-10), so it allows at most 256 buttons to have at most 256 different actions. MicroUI defines 6 basic actions described in Table 4-3.

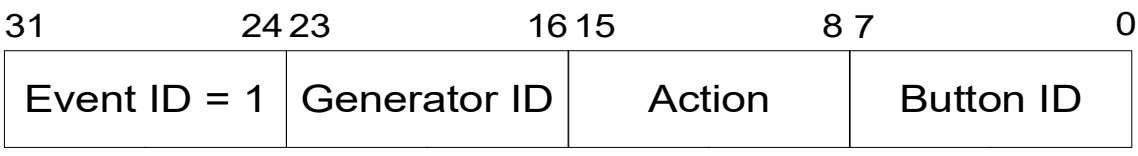


Illustration 4-10: Buttons event format

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 4-3: Basic button actions

4.10.2.2 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)`.

4.10.2.3 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.

4.10.3 KEYBOARD

The format of the data field of a keyboard event is not defined in this specification; it is implementation specific (Illustration 4-11).

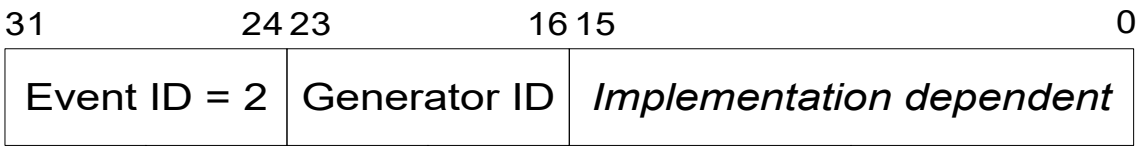


Illustration 4-11: Keyboard event format

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)`.

4.10.4 POINTER

The data of pointer event is composed as a buttons event of an action and a data (Illustration 4-12), so it allows at most 256 different actions. The first 6 actions are defined by buttons. Pointer defines another 4 actions described in Table 4-4.

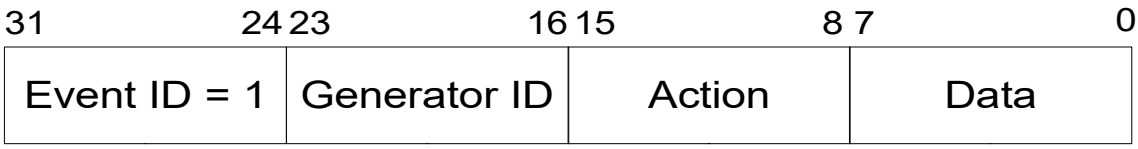


Illustration 4-12: Pointer event format

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 4-4: 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 4-14 shows an example of coordinates system configuration.

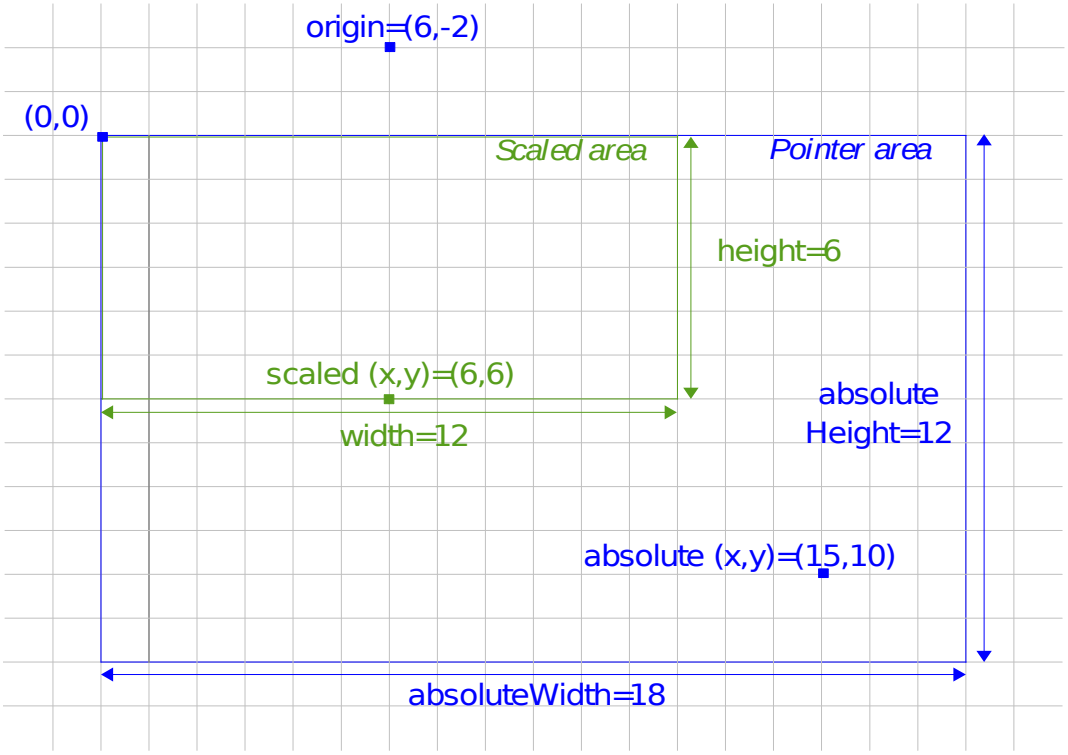


Illustration 4-13: Example of the `Pointer` coordinate system

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.

4.10.5 KEYPAD

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

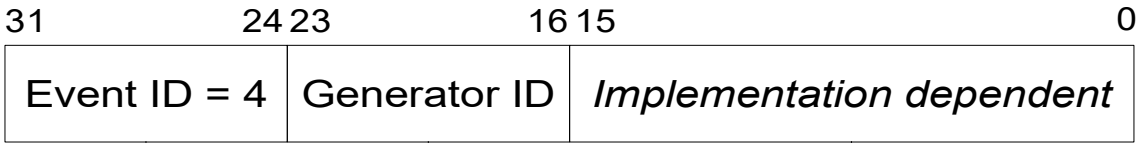


Illustration 4-14: Keypad event format

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 4-5) 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 4.10.3), 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 4-5: Keypad selection modes

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

4.10.6 STATE

4.10.6.1 Event format

The data of `STATE` event is composed of a state value and a state id (Illustration 4-15), so it allows at most 256 states to have at most 256 different state values.

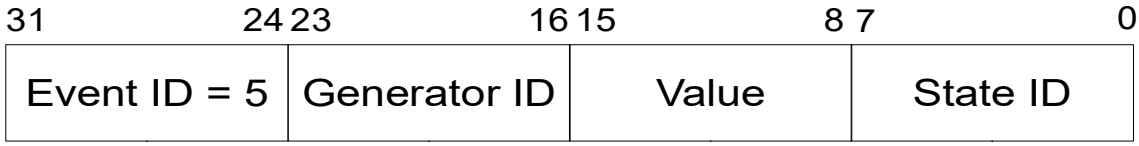


Illustration 4-15: States event format

4.10.6.2 Event generator

A `States` event genera`tor 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.

4.11 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 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

6 JAVA SPECIFICATION

Package Summary		Page
ej.microui	Contains MicroUI framework base classes.	33
ej.microui.display	Contains display management.	39
ej.microui.display.shape	Contains shapes rendering management.	131
ej.microui.display.transform	Contains image transformation management.	140
ej.microui.event	Contains events management.	158
ej.microui.event.controller	Contains helpers to handle events.	169
ej.microui.event.generator	Contains standard event generators.	188
ej.microui.led	Contains LEDs management.	225
ej.microui.util	Contains MicroUI utilities.	228

Package ej.microui

Contains MicroUI framework base classes.

See:
Description

Interface Summary		Page
UncaughtExceptionHandler	Interface for handlers invoked when MicroUI library receives an uncaught exception.	38

Class Summary		Page
MicroUI	The `MicroUI` class offers basic services in the MicroUI implementation. MicroUI is started explicitly by calling `start()`.	34
MicroUIPermission	Permission to start or stop MicroUI.	36

Package ej.microui Description

Contains MicroUI framework base classes.

Class MicroUI

ej.microui

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

```
abstract public class MicroUI
extends Object
```

The `MicroUI` class offers basic services in the MicroUI implementation.
MicroUI is started explicitly by calling `start()`. MicroUI may also be stopped with `stop()`.

Uncaught user errors may be handled defining an `UncaughtExceptionHandler`.

Method Summary		Page
static boolean	**isStarted**() Checks if MicroUI has been started.	35
static void	**setUncaughtExceptionHandler**(UncaughtExceptionHandler exceptionHandler) Sets the exception handler called when an exception occurred in the library.	35
static void	**start**() Starts MicroUI. It implies starting event serialization as well as rendering mechanisms. This method does nothing if MicroUI is already started.	34
static void	**stop**() Stops MicroUI. It implies stopping any potential event serialization as well as rendering mechanisms. This method does nothing if MicroUI is already stopped.	34

Method Detail

start

```
public static void start()
                  throws SecurityException
```

Starts MicroUI.
It implies starting event serialization as well as rendering mechanisms.
This method does nothing if MicroUI is already started.

Throws:

`SecurityException` - if a security manager exists and does not allow the caller to start MicroUI.

---

stop

```
public static void stop()
                  throws SecurityException
```

Stops MicroUI.
It implies stopping any potential event serialization as well as rendering mechanisms.
This method does nothing if MicroUI is already stopped.

Throws:
`SecurityException` - if a security manager exists and does not allow the caller to stop MicroUI.

---

isStarted

```
public static boolean isStarted()
```

Checks if MicroUI has been started.

Returns:
true when MicroUI is running.

---

setUncaughtExceptionHandler

```
public static final void setUncaughtExceptionHandler(UncaughtExceptionHandler exceptionHandler
)
```

Sets the exception handler called when an exception occurred in the library.

When no handler is defined, the exception stack trace is printed on standard error stream.

Parameters:
`exceptionHandler` - the handler to use when an exception occurred in the library.
Since:
2.0
See Also:
`System.err`

Class MicroUIPermission

ej.microui

```
java.lang.Object
  └java.security.Permission
      └ej.microui.MicroUIPermission
```

All Implemented Interfaces:

Guard, Serializable

```
public class MicroUIPermission
extends Permission
```

Permission to start or stop MicroUI. Permissions are checked when calling `MicroUI.start()` and `MicroUI.stop()`

See Also:

`MicroUI.start()`, `MicroUI.stop()`

Field Summary		Page
static String	**ACTION_START** Action to start MicroUI.	36
static String	**ACTION_STOP** Action to stop MicroUI.	36

Constructor Summary	Page
MicroUIPermission(String action) Creates a permission for events generated by the given event generator with `null` as name.	37

Method Summary		Page
boolean	**equals**(Object obj)	37
String	**getActions**()	37
int	**hashCode**()	37
boolean	**implies**(Permission permission)	37

Field Detail

ACTION_START

```
public static final String ACTION_START
```

Action to start MicroUI.

ACTION_STOP

```
public static final String ACTION_STOP
```

Action to stop MicroUI.

Constructor Detail

MicroUIPermission

```
public MicroUIPermission(String action)
```

Creates a permission for events generated by the given event generator with `null` as name.

Parameters:

`action` - the action to check.

Method Detail

equals

```
public boolean equals(Object obj)
```

Overrides:

`equals` in class `Permission`

---

getActions

```
public String getActions()
```

Overrides:

`getActions` in class `Permission`

---

hashCode

```
public int hashCode()
```

Overrides:

`hashCode` in class `Permission`

---

implies

```
public boolean implies(Permission permission)
```

Overrides:

`implies` in class `Permission`

Interface UncaughtExceptionHandler

ej.microui

```
public interface UncaughtExceptionHandler
```

Interface for handlers invoked when MicroUI library receives an uncaught exception.

Since:
2.0
See Also:
`MicroUI.setUncaughtExceptionHandler(UncaughtExceptionHandler)`

Method Summary		Page
void	**uncaughtException**(Throwable e) Method invoked when the given uncaught exception occurs in MicroUI library.	38

Method Detail

uncaughtException

```
void uncaughtException(Throwable e)
```

Method invoked when the given uncaught exception occurs in MicroUI library.

Parameters:
`e` - the uncaught exception.

Package ej.microui.display

Contains display management.

See:
Description

Interface Summary		Page
Colors	The interface `Colors` provides useful constants to handle RGB colors format.	41

Class Summary		Page
Display	A `Display` object represents a pixelated screen in the platform, and there is a display for each such pixelated screen.	45
Displayable	`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.	55
DisplayPermission	Permission to access a Display.	58
ExplicitFlush	An `ExplicitFlush` is a `GraphicsContext` where flushing data to the screen must be done explicitly by the application.	60
FlyingImage	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`.	61
Font	A `DisplayFont` defines how text is rendered on a `Display`.	64
FontPermission	Permission to create a `Font`.	85
GraphicsContext	The `GraphicsContext` class offers basic drawing facilities, to render lines, rectangles, polygons, arcs and text. `GraphicsContext` uses 24-bit RGB color.	87
Image	An `Image` object holds graphical display data.	114
ImagePermission	Permission to create an `Image`.	127
RenderableString	This class associates a string with a font.	129

Enum Summary		Page
Image.OutputFormat	Specify the format to apply when creating an immutable image.	123

Exception Summary		Page
ImageCreationException	Thrown when an image creation fail.	126

Package ej.microui.display Description

Contains display management.

Interface Colors

ej.microui.display

`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. The value `0x000000` is assigned to `BLACK`.	42
int	**BLUE** The blue RGB color constant. The value `0x0000ff` is assigned to `BLUE`.	42
int	**CYAN** The cyan RGB color constant. The value `0x00ffff` is assigned to `CYAN`.	42
int	**GRAY** The gray RGB color constant. The value `0x808080` is assigned to `GRAY`.	43
int	**GREEN** The green RGB color constant. The value `0x008000` is assigned to `GREEN`.	43
int	**LIME** The lime RGB color constant. The value `0x00ff00` is assigned to `LIME`.	43
int	**MAGENTA** The magenta RGB color constant. The value `0xff00ff` is assigned to `MAGENTA`.	43
int	**MAROON** The maroon RGB color constant. The value `0x800000` is assigned to `MAROON`.	43
int	**NAVY** The navy RGB color constant. The value `0x000080` is assigned to `NAVY`.	43

int	**OLIVE** The olive RGB color constant. The value `0x808000` is assigned to `OLIVE`.	43
int	**PURPLE** The purple RGB color constant. The value `0x800080` is assigned to `PURPLE`.	43
int	**RED** The red RGB color constant. The value `0xff0000` is assigned to `RED`.	44
int	**SILVER** The silver RGB color constant. The value `0xc0c0c0` is assigned to `SILVER`.	44
int	**TEAL** The teal RGB color constant. The value `0x008080` is assigned to `TEAL`.	44
int	**WHITE** The white RGB color constant. The value `0xffffff` is assigned to `WHITE`.	44
int	**YELLOW** The yellow RGB color constant. The value `0xffff00` is assigned to `YELLOW`.	44

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 Display

ej.microui.display

```
java.lang.Object
  └ ej.microui.display.Display
```

```
public class Display
extends Object
```

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.

All events on a display are serialized: repaint, callSerially, handleEvent etc. A display uses a `FIFOPump` to manage its serialized event mechanism.

Constructor Summary	Page
Display()	47

Method Summary		Page
void	**callSerially**(Runnable run) Serializes a call event in the system event stream.	52
static Display[]	**getAllDisplays**() Returns all available displays.	50
int	**getBacklight**() Returns the current backlight setting	50
int	**getBacklightColor**() Returns the current backlight color.	50
int	**getBPP**() Returns the number of bits per pixel of the display.	48
int	**getContrast**() Returns the contrast of the display.	49
static Display	**getDefaultDisplay**() Returns the default display of the system.	50

Displayable	**getDisplayable**() Returns the current `Displayable` object in the `Display`. The value returned by `getDisplayable()` may be `null` if no `Displayable` is visible.	51
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.	48
EventHandler	**getEventSerializer**() Returns the display's event serializer or `null`.	47
int	**getHeight**() Returns the height in pixels of the display screen area available to the application.	47
ExplicitFlush	**getNewExplicitFlush**() Returns a new `ExplicitFlush` which works on the same system screen as this display.	51
GraphicsContext	**getNewGraphicsContext**() Returns a new `GraphicsContext` which works on the same system screen as this display.	51
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.	48
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.	48
Image	**getScreenshot**() Creates an image with the full content of the display.	51
Image	**getScreenshot**(int x, int y, int width, int height) Creates an image with the content of the display region specified thanks the given rectangle.	52
int	**getWidth**() Returns the width in pixels of the display screen area available to the application.	47
void	**handleEvent**(int event) Injects a MicroUI event to be handled by the event generator associated with this Display.	53
boolean	**hasBacklight**() Tells whether the display has backlight.	49
boolean	**isColor**() Tells whether the display offers color.	48
boolean	**isDisplayThread**() Gets whether the current thread is the display events thread.	53
boolean	**isDisplayThread**(Thread thread) Gets whether the given thread is the display events thread.	53
boolean	**isDoubleBuffered**() Returns if the display uses an underlying double buffer (either hardware or software).	48
void	**setBacklight**(int backlight) Sets the backlight of the display.	49
void	**setBacklightColor**(int rgbColor) Sets the current backlight color, if it is allowed by implementation.	50

void	**setContrast**(int contrast) Sets the contrast of the display.	49
void	**setPriority**(int priority) Sets the priority of the display events processing.	53
void	**switchBacklight**(boolean on) Switches on or off the backlight of the display.	49
void	**waitForEvent**() Blocks the current thread (with all its locks) until all events outstanding at the time of the call have been processed.	53
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.	52

Constructor Detail

Display

```
public Display()
```

Method Detail

getEventSerializer

```
public EventHandler getEventSerializer()
```

Returns the display's event serializer or `null`.

Returns:
the display's event serializer or `null`.

Since:
2.0

---

getHeight

```
public int getHeight()
```

Returns the height in pixels of the display screen area available to the application.

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.

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.

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.

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

---

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).

---

hasBacklight

```
public boolean hasBacklight()
```

Tells whether the display has backlight.

Returns:
if display has backlight

---

setContrast

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

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

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

---

getContrast

```
public int getContrast()
```

Returns the contrast of the display.

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.

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

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

---

getBacklight

```
public int getBacklight()
```

Returns the current backlight setting

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.

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

getAllDisplays

```
public static Display[] getAllDisplays()
                                throws IllegalStateException
```

Returns all available displays. It is never `null` but the array may be empty.

Returns:
all available displays.
Throws:
`IllegalStateException` - if MicroUI is not started.

getDefaultDisplay

```
public static Display getDefaultDisplay()
                                 throws IllegalStateException
```

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`.
Throws:
`IllegalStateException` - if MicroUI is not started.

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.

getScreenshot

```
public Image getScreenshot()
```

Creates an image with the full content of the display.

This call is identical to: `getScreenshot(0, 0, display.getWidth(), display.getHeight())`.

Returns:
the created image.
Throws:
`ImageCreationException` - if MicroUI implementation cannot create the image.
`OutOfMemoryError` - if there is not enough room to add a new image.

Since:
2.0
See Also:
`GraphicsContext.drawRegion(Display, int, int, int, int, int, int, int, int)`

---

getScreenshot

```
public Image getScreenshot(int x,
                           int y,
                           int width,
                           int height)
```

Creates an image with the content of the display region specified thanks the given rectangle.

Parameters:
`x` - the x coordinate of the upper-left corner of the region to copy.
`y` - 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.
Returns:
the created image.
Throws:
`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.
`ImageCreationException` - if MicroUI implementation cannot create the image.
`OutOfMemoryError` - if there is not enough room to add a new image.
Since:
2.0
See Also:
`GraphicsContext.drawRegion(Display, int, int, int, int, int, int, int, int)`

---

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
Throws:
`SecurityException` - if a security manager exists and does not allow the caller to get the display.

---

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.

---

isDisplayThread

`public boolean **isDisplayThread**(Thread thread)`

Gets whether the given thread is the display events thread.

Parameters:
`thread` - the thread to check
Returns:
`true` if the given thread is the display events thread, `false` otherwise.
Since:
2.0

---

isDisplayThread

`public boolean **isDisplayThread**()`

Gets whether the current thread is the display events thread.

Returns:
`true` if the current thread is the display events thread, `false` otherwise.
Since:
2.0

---

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)`

Injects 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.display

```
java.lang.Object
  └ej.microui.display.Displayable
```

```
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.	56

Method Summary		Page
abstract EventHandler	**getController**() Gets the displayable controller or `null` if none.	57
Display	**getDisplay**() Gets the displayable's display.	56
void	**hide**() Sets the displayable as hidden on its display.	56
protected void	**hideNotify**() This method is called by system as soon as the displayable becomes hidden.	57
boolean	**isShown**() Checks whether the displayable is visible on its display.	56
abstract void	**paint**(GraphicsContext g) Draws the displayable.	57
void	**repaint**() Requests a repaint for the entire displayable.	57
void	**show**() Sets the displayable as visible on its display.	56
protected void	**showNotify**() This method is called by system as soon as the displayable becomes visible.	56

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 the given display is `null`

Method Detail

getDisplay

```
public Display getDisplay()
```

Gets the displayable's display.

The returned display can't be `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()
          throws SecurityException
```

Sets the displayable as visible on its display.

Throws:
`SecurityException` - if a security manager exists and does not allow the caller to get the 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(GraphicsContext)` on the displayable.

If the displayable is not visible, this call has no effect.

The call(s) to `paint(GraphicsContext)` occurs asynchronously to this call. That is, this method will not block waiting for `paint(GraphicsContext)` to finish.

To synchronize with the `paint(GraphicsContext)` routine, applications can use either `Display.callSerially(Runnable)` or `Display.waitForEvent()`, or they can code explicit synchronization into their `paint(GraphicsContext)` routine.

paint

```
public abstract void paint(GraphicsContext g)
```

Draws the displayable. This method must be implemented by subclasses to render graphics on a display.

Parameters:
`g` - the `GraphicsContext` of the displayable's `Display`.

getController

```
public abstract EventHandler getController()
```

Gets the displayable controller or `null` if none.

Returns:
the displayable controller or `null`.
Since:
2.0

Class DisplayPermission

ej.microui.display

```
java.lang.Object
  └java.security.Permission
      └ej.microui.display.DisplayPermission
```

All Implemented Interfaces:

Guard, Serializable

```
public class DisplayPermission
extends Permission
```

Permission to access a Display. Permission is checked when calling `Displayable.show()`.

Since:

2.0

Constructor Summary	Page
DisplayPermission(Display display) Creates a display permission for the given display with `null` as name.	58

Method Summary		Page
boolean	**equals**(Object obj)	59
String	**getActions**()	59
Display	**getDisplay**() Gets the display handled by this permission.	58
int	**hashCode**()	59
boolean	**implies**(Permission permission)	59

Constructor Detail

DisplayPermission

```
public DisplayPermission(Display display)
```

Creates a display permission for the given display with `null` as name.

Parameters:

`display` - the display.

Method Detail

getDisplay

```
public Display getDisplay()
```

Gets the display handled by this permission.

Returns:
the display.

---

equals

```
public boolean equals(Object obj)
```

Overrides:
`equals` in class `Permission`

---

getActions

```
public String getActions()
```

Overrides:
`getActions` in class `Permission`

---

hashCode

```
public int hashCode()
```

Overrides:
`hashCode` in class `Permission`

---

implies

```
public boolean implies(Permission permission)
```

Overrides:
`implies` in class `Permission`

Class ExplicitFlush

ej.microui.display

```
java.lang.Object
  └ej.microui.display.GraphicsContext
      └ej.microui.display.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.display.GraphicsContext
BASELINE, BOTTOM, DOTTED, HCENTER, HCENTER_BOTTOM, HCENTER_TOP, HCENTER_VCENTER, LEFT, LEFT_BOTTOM, LEFT_TOP, LEFT_VCENTER, OPAQUE, RIGHT, RIGHT_BOTTOM, RIGHT_TOP, RIGHT_VCENTER, SOLID, TOP, TRANSPARENT, VCENTER

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.	60

Methods inherited from class ej.microui.display.GraphicsContext
clipRect, drawChar, drawChars, drawCircle, drawCircleArc, drawEllipse, drawEllipseArc, drawHorizontalLine, drawImage, drawImage, drawLine, drawPixel, drawPolygon, drawPolygon, drawRect, drawRegion, drawRegion, drawRegion, drawRegion, drawRoundRect, drawString, drawSubstring, drawVerticalLine, fillCircle, fillCircleArc, fillEllipse, fillEllipseArc, fillPolygon, fillPolygon, fillRect, fillRoundRect, getAlpha, getARGB, getBackgroundColor, getClipHeight, getClipWidth, getClipX, getClipY, getColor, getDisplay, getDisplayColor, getEllipsis, getFont, getStrokeStyle, getTranslateX, getTranslateY, hasBackgroundColor, readPixel, removeBackgroundColor, setBackgroundColor, setClip, setColor, setEllipsis, 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.display

```
java.lang.Object
  └ ej.microui.display.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(Display display, Image skin) Creates a new `FlyingImage`.	62
FlyingImage(Image skin) Creates a new `FlyingImage`.	62

	Method Summary	Page
Display	**getDisplay**() Returns the display associated to the `FlyingImage`.	63
Image	**getImage**() Returns the image associated to the `FlyingImage`.	63
int	**getX**() Get the x coordinate of the `FlyingImage` position	63
int	**getY**() Get the y coordinate of the `FlyingImage` position	63
void	**hide**() Sets the `FlyingImage` as hidden on its display.	62
boolean	**isShown**() Checks whether the `FlyingImage` is visible on its display.	62
void	**repaint**() Requests a repaint of the `FlyingImage`.	62
void	**setLocation**(int x, int y) Sets the location of the `FlyingImage`.	63
void	**show**() Sets the `FlyingImage` as visible on its display.	62

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.

FlyingImage

```
public FlyingImage(Display display,
                   Image skin)
```

Creates a new `FlyingImage`. On creation the `FlyingImage` is not shown - call `show()` to show it.

Parameters:
`display` - the `Display` where the `FlyingImage` will appears.
`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.display

```
java.lang.Object
  └ ej.microui.display.Font
```

```
public class Font
extends Object
```

A `DisplayFont` defines how text is rendered on a `Display`.

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.

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.

Field Summary		Page
static int	**ARABIC** Constant for arabic font identifier.	70
static int	**ARMENIAN** Constant for armenian font identifier.	70
static int	**BALINESE** Constant for balinese font identifier.	70
static int	**BAMUM** Constant for bamum font identifier.	78
static int	**BATAK** Constant for batak font identifier.	79
static int	**BENGALI** Constant for bengali font identifier.	70
static int	**BOPOMOFO** Constant for bopomofo font identifier.	70
static int	**BRAILLE** Constant for braille font identifier.	70

static int	**BUGINESE** Constant for buginese font identifier.	70
static int	**BUHID** Constant for buhid font identifier.	70
static int	**CANADIAN_ABORIGINAL** Constant for canadian aboriginal font identifier.	71
static int	**CHAM** Constant for cham font identifier.	77
static int	**CHEROKEE** Constant for cherokee font identifier.	71
static int	**COMMON** Constant for common font identifier.	76
static int	**COPTIC** Constant for coptic font identifier.	71
static int	**CYRILLIC** Constant for cyrillic font identifier.	71
static int	**DEVANAGARI** Constant for devanagari font identifier.	71
static int	**ETHIOPIC** Constant for ethiopic font identifier.	71
static int	**GEORGIAN** Constant for georgian font identifier.	71
static int	**GLAGOLITIC** Constant for glagolitic font identifier.	71
static int	**GREEK** Constant for greek font identifier.	72
static int	**GUJARATI** Constant for gujarati font identifier.	72
static int	**GURMUKHI** Constant for gurmukhi font identifier.	72
static int	**HAN** Constant for han font identifier.	72
static int	**HANGUL** Constant for hangul font identifier.	72
static int	**HANUNOO** Constant for hanunoo font identifier.	72
static int	**HEBREW** Constant for hebrew font identifier.	72
static int	**HIRAGANA** Constant for hiragana font identifier.	73
static int	**INHERITED** Constant for inherited font identifier.	76
static int	**JAVANESE** Constant for javanese font identifier.	78
static int	**KANNADA** Constant for kannada font identifier.	73

static int	**KATAKANA** Constant for katakana font identifier.	73
static int	**KAYAH_LI** Constant for kayah li font identifier.	77
static int	**KHMER** Constant for khmer font identifier.	73
static int	**LAO** Constant for lao font identifier.	73
static int	**LATIN** Constant for latin font identifier.	73
static int	**LEPCHA** Constant for lepcha font identifier.	77
static int	**LIMBU** Constant for limbu font identifier.	73
static int	**LISU** Constant for lisu font identifier.	78
static int	**MALAYALAM** Constant for malayalam font identifier.	73
static int	**MANDAIC** Constant for mandaic font identifier.	79
static int	**MEETEI_MAYEK** Constant for meetei mayek font identifier.	78
static int	**MONGOLIAN** Constant for mongolian font identifier.	74
static int	**MYANMAR** Constant for myanmar font identifier.	74
static int	**NEW_TAI_LUE** Constant for new tai lue font identifier.	74
static int	**NKO** Constant for nko font identifier.	74
static int	**OGHAM** Constant for ogham font identifier.	74
static int	**OL_CHIKI** Constant for ol chiki font identifier.	77
static int	**ORIYA** Constant for oriya font identifier.	74
static int	**PHAGS_PA** Constant for phags pa font identifier.	74
static int	**REJANG** Constant for rejang font identifier.	77
static int	**RUNIC** Constant for runic font identifier.	74
static int	**SAMARITAN** Constant for samaritan font identifier.	78
static int	**SAURASHTRA** Constant for saurashtra font identifier.	77

static int	**SINHALA** Constant for sinhala font identifier.	75
static int	**STYLE_BOLD** The bold style constant.	69
static int	**STYLE_ITALIC** The italic style constant.	69
static int	**STYLE_PLAIN** The plain style constant.	69
static int	**STYLE_RESIZED** The underlined style constant.	69
static int	**STYLE_UNDERLINED** The underlined style constant.	69
static int	**SUNDANESE** Constant for sundanese font identifier.	77
static int	**SYLOTI_NAGRI** Constant for syloti nagri font identifier.	75
static int	**SYRIAC** Constant for syriac font identifier.	75
static int	**TAGALOG** Constant for tagalog font identifier.	75
static int	**TAGBANWA** Constant for tagbanwa font identifier.	75
static int	**TAI_LE** Constant for tai le font identifier.	75
static int	**TAI_THAM** Constant for tai tham font identifier.	78
static int	**TAI_VIET** Constant for tai viet font identifier.	78
static int	**TAMIL** Constant for tamil font identifier.	75
static int	**TELUGU** Constant for telugu font identifier.	76
static int	**THAANA** Constant for thaana font identifier.	76
static int	**THAI** Constant for thai font identifier.	76
static int	**TIBETAN** Constant for tibetan font identifier.	76
static int	**TIFINAGH** Constant for tifinagh font identifier.	76
static int	**VAI** Constant for vai font identifier.	77
static int	**YI** Constant for yi font identifier.	76

Constructor Summary	***Page***

	Font() Forbidden constructor: use getAllFonts() or getDefaultFont() to get the available fonts.	79

Method Summary		Page
int	**charsWidth**(char[] ch, int offset, int length) Gets the width of the characters in ch from offset to offset+length with this font and its x ratio. The width is the horizontal distance that would be occupied if the length characters were drawn using this font.	81
int	**charWidth**(char ch) Gets the width of the specified character with this font and its x ratio. The width is the horizontal distance that would be occupied if ch was drawn using this font.	81
boolean	**equals**(Object obj)	79
static Font[]	**getAllFonts**() Gets an array containing all available DisplayFonts in the system.	82
int	**getBaselinePosition**() Gets the distance in pixels from the top of the text to the text's baseline.	84
static Font	**getDefaultFont**() Gets the default font for all Displays. This method may return null if no font is declared in the system.	82
String	**getDescriptor**() Returns the descriptor of the font or null if no descriptor is available.	80
static Font	**getFont**(int identifier, int height, int style) Gets a DisplayFont matching the requested characteristics as close as possible.	83
int	**getHeight**() Gets the height of a line of text with this font and its y ratio.	84
int[]	**getIdentifiers**() Gets an array of identifiers supported by the font.	79
int	**getStyle**() Gets 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.	80
float	**getXRatio**() Gets x ratio.	83
float	**getYRatio**() Gets y ratio.	84
int	**hashCode**()	79
boolean	**isBold**() Gets whether the font is bold or not.	80
boolean	**isIdentifierSupported**(int identifier) Checks whether the font supports the given identifier or not.	79
boolean	**isItalic**() Gets whether the font is italic or not.	80
boolean	**isMonospaced**() Gets whether the font is monospaced or not.	81

boolean	**isPlain**() Gets whether the font is plain or not.	80
boolean	**isUnderlined**() Gets whether the font is underlined or not.	81
void	**resetRatios**() Resets x and y ratios at their original value (construction value).	84
void	**setRatios**(float xRatio, float yRatio) Sets x and y ratios.	83
int	**stringWidth**(String str) Gets the width of the given string with this font and its x ratio. The width is the horizontal distance that would be occupied if the string was drawn using this font.	82
int	**substringWidth**(String str, int offset, int len) Gets the width of the string from offset to offset+len with this font and its x ratio.	82

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.

~~STYLE_RESIZED~~

```
public static final int STYLE_RESIZED
```

The underlined style constant. It may be combined with other style constants.

Value 8 is assigned to STYLE_RESIZED.

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`.

CYRILLIC

```
public static final int CYRILLIC
```

Constant for cyrillic font identifier.

Value `14` is assigned to `CYRILLIC`.

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`.

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`.

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`.

PHAGS_PA

```
public static final int PHAGS_PA
```

Constant for phags pa font identifier.

Value `44` is assigned to `PHAGS_PA`.

RUNIC

```
public static final int RUNIC
```

Constant for runic font identifier.

Value `46` is assigned to `RUNIC`.

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`.

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: use `getAllFonts()` or `getDefaultFont()` to get the available fonts.

See Also:

`getAllFonts()`, `getDefaultFont()`

Method Detail

equals

```
public boolean equals(Object obj)
```

Overrides:

`equals` in class `Object`

hashCode

```
public int hashCode()
```

Overrides:

`hashCode` in class `Object`

getIdentifiers

```
public int[] getIdentifiers()
```

Gets 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.

isIdentifierSupported

```
public boolean isIdentifierSupported(int identifier)
```

Checks whether the font supports the given identifier or not.

Parameters:
`identifier` - the identifier to check.
Returns:
`true` if the font supports the given identifier, `false` otherwise.

---

getStyle

```
public int getStyle()
```

Gets 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 String getDescriptor()
```

Returns the descriptor of the font or `null` if no descriptor is available.

Returns:
the descriptor of the font or `null`.

---

isPlain

```
public boolean isPlain()
```

Gets whether the font is plain or not.

Returns:
`true` if the font is plain, `false` otherwise.

---

isBold

```
public boolean isBold()
```

Gets whether the font is bold or not.

Returns:
`true` if the font is bold, `false` otherwise.

---

isItalic

```
public boolean isItalic()
```

Gets whether the font is italic or not.

Returns:
`true` if the font is italic, `false` otherwise.

---

isUnderlined

```
public boolean isUnderlined()
```

Gets whether the font is underlined or not.

Returns:
`true` if the font is underlined, `false` otherwise.

isMonospaced

```
public boolean isMonospaced()
```

Gets whether the font is monospaced or not.

A monospaced font is a font which all characters have the same width.

Returns:
`true` if the font is monospaced, `false` otherwise.

charWidth

```
public int charWidth(char ch)
```

Gets the width of the specified character with this font and its x ratio.
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
See Also:
`getXRatio()`

charsWidth

```
public int charsWidth(char[] ch,
                      int offset,
                      int length)
```

Gets the width of the characters in `ch` from `offset` to `offset+length` with this font and its x ratio.
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
See Also:
`getXRatio()`

stringWidth

```
public int stringWidth(String str)
```

Gets the width of the given string with this font and its x ratio.
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 the given string.
Throws:
`NullPointerException` - if the given string is `null`
See Also:
`getXRatio()`

substringWidth

```
public int substringWidth(String str,
                          int offset,
                          int len)
```

Gets the width of the string from `offset` to `offset+len` with this font and its x ratio.
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
See Also:
`getXRatio()`

getDefaultFont

```
public static Font getDefaultFont()
                           throws IllegalStateException
```

Gets the default font for all Displays.
This method may return `null` if no font is declared in the system.

Returns:
the default font
Throws:
`IllegalStateException` - if MicroUI is not started

getAllFonts

```
public static Font[] getAllFonts()
                           throws IllegalStateException
```

Gets an array containing all available DisplayFonts in the system.

Returns:
an array of fonts
Throws:
`IllegalStateException` - if MicroUI is not started

---

getFont

```
public static Font getFont(int identifier,
                           int height,
                           int style)
                    throws IllegalStateException
```

Gets 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`.
Throws:
`IllegalStateException` - if MicroUI is not started.

---

setRatios

```
public void setRatios(float xRatio,
                      float yRatio)
```

Sets x and y ratios.

Parameters:
`xRatio` - the x ratio to set.
`yRatio` - the y ratio to set.
Since:
1.5

---

getXRatio

```
public float getXRatio()
```

Gets x ratio.

Returns:
the x ratio.
Since:
1.5

---

getYRatio

```
public float getYRatio()
```

Gets y ratio.

Returns:
the y ratio.
Since:
1.5

---

resetRatios

```
public void resetRatios()
```

Resets x and y ratios at their original value (construction value).

Since:
1.5

---

getHeight

```
public int getHeight()
```

Gets the height of a line of text with this font and its y ratio.

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.
See Also:
`getYRatio()`

---

getBaselinePosition

```
public int getBaselinePosition()
```

Gets the distance in pixels from the top of the text to the text's baseline. The distance includes the y ratio.

Returns:
the font baseline.

Class FontPermission

ej.microui.display

```
java.lang.Object
  └java.security.Permission
      └ej.microui.display.FontPermission
```

All Implemented Interfaces:

Guard, Serializable

```
public class FontPermission
extends Permission
```

Permission to create a `Font`. Permission is checked when fonts are added to the system during the initialization process.

Since:

2.0

Constructor Summary	Page
FontPermission() Creates a font permission with `null` as name.	85

Method Summary		Page
boolean	**equals**(Object obj)	85
String	**getActions**()	85
int	**hashCode**()	86
boolean	**implies**(Permission permission)	86

Constructor Detail

FontPermission

```
public FontPermission()
```

Creates a font permission with `null` as name.

Method Detail

equals

```
public boolean equals(Object obj)
```

Overrides:

`equals` in class `Permission`

---

getActions

```
public String getActions()
```

Overrides:

`getActions` in class `Permission`

---

hashCode

```
public int hashCode()
```

Overrides:

`hashCode` in class `Permission`

---

implies

```
public boolean implies(Permission permission)
```

Overrides:

`implies` in class `Permission`

Class GraphicsContext

ej.microui.display

```
java.lang.Object
  └ ej.microui.display.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 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	**BASELINE** Constant for positioning the baseline of the text at the anchor point.	93
static int	**BOTTOM** Constant for positioning the bottom of the drawing at the anchor point.	93
static int	**DOTTED** Constant for the `DOTTED` stroke style.	94
static int	**HCENTER** Constant for centering drawing horizontally around the anchor point.	92
static int	**HCENTER_BOTTOM** Equivalent to `HCENTER	BOTTOM`.	94

static int	**HCENTER_TOP** Equivalent to HCENTER | TOP.	94
static int	**HCENTER_VCENTER** Equivalent to HCENTER | VCENTER.	94
static int	**LEFT** Constant for positioning the left side of the drawing at the anchor point.	93
static int	**LEFT_BOTTOM** Equivalent to LEFT | BOTTOM.	94
static int	**LEFT_TOP** Equivalent to LEFT | TOP.	93
static int	**LEFT_VCENTER** Equivalent to LEFT | VCENTER.	93
static int	**OPAQUE** Maximal opacity.	95
static int	**RIGHT** Constant for positioning the right side of the drawing at the anchor point.	93
static int	**RIGHT_BOTTOM** Equivalent to RIGHT | BOTTOM.	94
static int	**RIGHT_TOP** Equivalent to RIGHT | TOP.	94
static int	**RIGHT_VCENTER** Equivalent to RIGHT | VCENTER.	94
static int	**SOLID** Constant for the SOLID stroke style.	94
static int	**TOP** Constant for positioning the top of the drawing at the anchor point.	93
static int	**TRANSPARENT** Minimal opacity.	95
static int	**VCENTER** Constant for centering the drawing vertically around the anchor point.	93

Constructor Summary		Page
	GraphicsContext() Forbidden constructor: use Display.getNewGraphicsContext() to get an instance of GraphicsContext.	95

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.	98
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).	111
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).	112

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.	105
void	**drawCircleArc**(int x, int y, int diameter, int startAngle, int arcAngle) Draws the outline of a circular arc covering the specified square, using the current color and stroke style.	103
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.	106
void	**drawEllipseArc**(int x, int y, int width, int height, int startAngle, int arcAngle) Draws the outline of a elliptical arc covering the specified rectangle, using the current color and stroke style.	104
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.	100
void	**drawImage**(Image img, int x, int y, int anchor) Draws an image at the given anchor point.	107
void	**drawImage**(Image img, int x, int y, int anchor, int alpha) Draws an image at the given anchor point.	107
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.	101
void	**drawPixel**(int x, int y) Draws a pixel at (x,y) using the current color.	100
void	**drawPolygon**(int[] xys) Draws the closed polygon which is defined by the array of integer coordinates, using the current color and stroke style.	102
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.	102
void	**drawRect**(int x, int y, int width, int height) Draws the outline of the specified rectangle using the current color and stroke style.	101
void	**drawRegion**(Display display, int xSrc, int ySrc, int width, int height, int xDest, int yDest, int anchor) Draws a region of a display.	109
void	**drawRegion**(Display display, int xSrc, int ySrc, int width, int height, int xDest, int yDest, int anchor, int alpha) Draws a region of a display.	109
void	**drawRegion**(Image src, int xSrc, int ySrc, int width, int height, int xDest, int yDest, int anchor) Draws a region of an image.	108
void	**drawRegion**(Image src, int xSrc, int ySrc, int width, int height, int xDest, int yDest, int anchor, int alpha) Draws a region of an image.	108
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.	101
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).	111

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.	111
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.	100
void	**fillCircle**(int x, int y, int diameter) Fills a circle covering the rectangle specified by its diameter with the current color.	106
void	**fillCircleArc**(int x, int y, int diameter, int startAngle, int arcAngle) Fills a circular arc covering the specified square with the current color.	104
void	**fillEllipse**(int x, int y, int width, int height) Fills a ellipse covering the specified rectangle with the current color.	106
void	**fillEllipseArc**(int x, int y, int width, int height, int startAngle, int arcAngle) Fills an elliptical arc covering the specified rectangle with the current color.	105
void	**fillPolygon**(int[] xys) Fills the closed polygon which is defined by the array of integer coordinates, using the current color.	103
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.	103
void	**fillRect**(int x, int y, int width, int height) Fills the specified rectangle with the current color.	101
void	**fillRoundRect**(int x, int y, int width, int height, int arcWidth, int arcHeight) Fills the specified rounded corner rectangle with the current color.	102
static int	**getAlpha**(int opacityPercent) Gets the alpha level for the given opacity.	95
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.	110
int	**getBackgroundColor**() Gets the current background color.	97
int	**getClipHeight**() Gets the height of the current clipping zone.	99
int	**getClipWidth**() Gets the width of the current clipping zone.	99
int	**getClipX**() Gets the x offset of the current clipping zone, relative to the graphics context's origin.	99
int	**getClipY**() Gets the y offset of the current clipping zone, relative to graphics context's origin.	99
int	**getColor**() Gets 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.	97
Display	**getDisplay**() Gets the display associated with the GraphicsContext.	112

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.	97
boolean	**getEllipsis**() Gets whether the truncation mechanism is enabled or not.	112
Font	**getFont**() Gets the current font.	98
int	**getStrokeStyle**() Gets the current stroke style: SOLID or DOTTED.	98
int	**getTranslateX**() Gets the x coordinate of the translated origin of the graphics context.	96
int	**getTranslateY**() Gets the y coordinate of the translated origin of the graphics context.	96
boolean	**hasBackgroundColor**() Gets whether there us a background color.	97
int	**readPixel**(int x, int y) Obtains the ARGB color of the pixel at (x,y).	100
void	**removeBackgroundColor**() Removes the current background color.	97
void	**setBackgroundColor**(int rgbColor) Sets the current background 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. The background color is used by several drawings: draw text, draw anti-aliased line etc.	96
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).	99
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.	96
void	**setEllipsis**(boolean enable) Enables (disables) truncation when rendering characters.	112
void	**setFont**(Font font) Sets the font for subsequent text operations.	98
void	**setStrokeStyle**(int style) Sets the stroke style of this graphics context.	97
void	**translate**(int x, int y) Translates the GraphicsContext origin with the given vector (x,y).	95

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`.

LEFT_TOP

```
public static final int LEFT_TOP
```

Equivalent to `LEFT | TOP`.

LEFT_VCENTER

```
public static final int LEFT_VCENTER
```

Equivalent to `LEFT | VCENTER`.

LEFT_BOTTOM

```
public static final int LEFT_BOTTOM
```

Equivalent to `LEFT | BOTTOM`.

HCENTER_TOP

```
public static final int HCENTER_TOP
```

Equivalent to `HCENTER | TOP`.

HCENTER_VCENTER

```
public static final int HCENTER_VCENTER
```

Equivalent to `HCENTER | VCENTER`.

HCENTER_BOTTOM

```
public static final int HCENTER_BOTTOM
```

Equivalent to `HCENTER | BOTTOM`.

RIGHT_TOP

```
public static final int RIGHT_TOP
```

Equivalent to `RIGHT | TOP`.

RIGHT_VCENTER

```
public static final int RIGHT_VCENTER
```

Equivalent to `RIGHT | VCENTER`.

RIGHT_BOTTOM

```
public static final int RIGHT_BOTTOM
```

Equivalent to `RIGHT | BOTTOM`.

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`.

OPAQUE

```
public static final int OPAQUE
```

Maximal opacity.

Since:
2.0
See Also:
`drawImage(Image, int, int, int, int)`

TRANSPARENT

```
public static final int TRANSPARENT
```

Minimal opacity.

Since:
2.0
See Also:
`drawImage(Image, int, int, int, int)`

Constructor Detail

GraphicsContext

```
GraphicsContext()
```

Forbidden constructor: use `Display.getNewGraphicsContext()` to get an instance of `GraphicsContext`.

See Also:
`Display.getNewGraphicsContext()`

Method Detail

getAlpha

```
public static int getAlpha(int opacityPercent)
```

Gets the alpha level for the given opacity.

It can be used to draw transparent images.

Parameters:
`opacityPercent` - the expected opacity in percentage.
Returns:
the alpha level.
Since:
2.0
See Also:
`drawImage(Image, int, int, int, int)`

---

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`**`()`

Gets the x coordinate of the translated origin of the graphics context.

Returns:

x coordinate of the translated origin.

---

getTranslateY

`public final int` **`getTranslateY`**`()`

Gets 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.

---

setBackgroundColor

`public final void` **`setBackgroundColor`**`(int rgbColor)`

Sets the current background 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.
The background color is used by several drawings: draw text, draw anti-aliased line etc. When not set, these algorithms have to read the pixel destination color before blending it with the current foreground color.
Note: this background color is useless for black and white displays.

Parameters:
`rgbColor` - the color to set.

---

removeBackgroundColor

```
public final void removeBackgroundColor()
```

Removes the current background color.

---

getColor

```
public final int getColor()
```

Gets 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:
the current color.

---

getBackgroundColor

```
public final int getBackgroundColor()
```

Gets the current background color.

Returns:
the current background color.

---

hasBackgroundColor

```
public final boolean hasBackgroundColor()
```

Gets whether there us a background color.

Returns:
`true` if there is a background color, `false` otherwise.

---

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 this graphics context. It is used to draw lines, arcs, circle, rectangles and polygons.

Parameters:
`style` - either `SOLID` or `DOTTED`.
Throws:
`IllegalArgumentException` - if the style is not valid.

---

getStrokeStyle

```
public final int getStrokeStyle()
```

Gets the current stroke style: `SOLID` or `DOTTED`.

Returns:
the stroke style.

---

setFont

```
public void setFont(Font 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 Font getFont()
```

Gets the current font.

Returns:
the 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()
```

Gets the x offset of the current clipping zone, relative to the graphics context's origin.

Returns:

the x offset of the current clipping zone.

getClipY

```
public final int getClipY()
```

Gets the y offset of the current clipping zone, relative to graphics context's origin.

Returns:

the y offset of the current clipping zone.

getClipWidth

```
public final int getClipWidth()
```

Gets the width of the current clipping zone.

Returns:

the width of the current clipping zone.

getClipHeight

```
public final int getClipHeight()
```

Gets the height of the current clipping zone.

Returns:

the 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 ARGB 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 ARGB color of the pixel.

Throws:

`IllegalArgumentException` - if the pixel coordinates are out of bounds of the source graphics context.

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.

Parameters:

`xys` - the array of coordinates : x1,y1,.....xn,yn.

Throws:

`NullPointerException` - if the given array is `null`.
`IllegalArgumentException` - if the given array 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 given array is `null`.
`IllegalArgumentException` - if the given array 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 given array is `null`.

`IllegalArgumentException` - if the given array 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.

---

drawCircleArc

```
public final void drawCircleArc(int x,
                                 int y,
                                 int diameter,
                                 int startAngle,
                                 int arcAngle)
```

Draws the outline of a circular arc covering the specified square, 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 `diameter`.

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 the given diameter 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
`diameter` - the diameter of the arc to draw
`startAngle` - the beginning angle of the arc to draw
`arcAngle` - the angular extent of the arc from `startAngle`

---

drawEllipseArc

```
public final void drawEllipseArc(int x,
                                 int y,
                                 int width,
                                 int height,
                                 int startAngle,
                                 int arcAngle)
```

Draws the outline of a 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 the given 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`

---

fillCircleArc

```
public final void fillCircleArc(int x,
                                int y,
                                int diameter,
                                int startAngle,
                                int arcAngle)
```

Fills a circular arc covering the specified square 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 `diameter`.

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 the given diameter 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.
`diameter` - the diameter of the arc to fill
`startAngle` - the beginning angle of the arc to draw
`arcAngle` - the angular extent of the arc from `startAngle`

---

fillEllipseArc

```
public final void fillEllipseArc(int x,
                                 int y,
                                 int width,
                                 int height,
                                 int startAngle,
                                 int arcAngle)
```

Fills an 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 the given 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 squared 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 the given start angle.

---

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 the given coordinates (upper-left corner) and whose dimension is given by the diameter parameter.

If the given 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 the given coordinates (upper-left corner) and whose dimension is given by the diameter parameter.

If the given 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 the given coordinates (upper-left corner) and whose dimension is given by the width and height parameters.

If either the given 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 the given coordinates (upper-left corner) and whose dimension is given by the width and height parameters.

If either the given 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 the given position. Position constants may be given to specify the precise location of the image around the anchor point.

Equivalent to calling `drawImage(Image, int, int, int, int)` with `OPAQUE` as alpha.

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 the given anchor is not a valid value (`BASELINE` is illegal).
`NullPointerException` - if the given image is `null`.

---

drawImage

```
public final void drawImage(Image img,
                            int x,
                            int y,
                            int anchor,
                            int alpha)
```

Draws an image at the given anchor point. In addition with `drawImage(Image, int, int, int)`, this method allows to specify the global opacity value to apply during the image rendering.

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.
`alpha` - the global opacity rendering value.

Throws:

`IllegalArgumentException` - if the given anchor is not a valid value (`BASELINE` is illegal)., if the given alpha is not a value between `TRANSPARENT` and `OPAQUE`.
`NullPointerException` - if the given image is `null`.

Since:

2.0

---

drawRegion

```
public final void drawRegion(Image src,
                             int xSrc,
                             int ySrc,
                             int width,
                             int height,
                             int xDest,
                             int yDest,
                             int anchor)
```

Draws a region of an image.

Equivalent to calling `drawRegion(Image, int, int, int, int, int, int, int, int)` with `OPAQUE` as alpha.

Parameters:

`src` - the image to copy from.
`xSrc` - the x coordinate of the upper-left corner of the region to copy.
`ySrc` - 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.
`xDest` - the x coordinate of the anchor point in the destination.
`yDest` - the y coordinate of the anchor point in the destination.
`anchor` - the position of the region around the anchor point.

Throws:

`NullPointerException` - if the given image is `null`.
`IllegalArgumentException` - if the given anchor is not a valid value (`BASELINE` is illegal).

drawRegion

```
public final void drawRegion(Image src,
                             int xSrc,
                             int ySrc,
                             int width,
                             int height,
                             int xDest,
                             int yDest,
                             int anchor,
                             int alpha)
```

Draws a region of an image.

The region of the image to draw is given relative to the image (origin at the upper-left corner) as a rectangle.

The image region anchor point in destination is at the given relative position. 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 graphics context area, pixels out of the range will not be drawn.

This method allows to specify the opacity value to apply during the image rendering.

Parameters:

`src` - the image to copy from.
`xSrc` - the x coordinate of the upper-left corner of the region to copy.
`ySrc` - 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.
`xDest` - the x coordinate of the anchor point in the destination.
`yDest` - the y coordinate of the anchor point in the destination.

`anchor` - the position of the region around the anchor point.
`alpha` - the alpha to apply to the region.

Throws:

`NullPointerException` - if the given image is `null`.
`IllegalArgumentException` - if the given anchor is not a valid value (`BASELINE` is illegal)., if the given alpha is not a value between `TRANSPARENT` and `OPAQUE`.

Since:

2.0

---

drawRegion

```
public void drawRegion(Display display,
                       int xSrc,
                       int ySrc,
                       int width,
                       int height,
                       int xDest,
                       int yDest,
                       int anchor)
```

Draws a region of a display.

Equivalent to calling `drawRegion(Display, int, int, int, int, int, int, int, int)` with `OPAQUE` as alpha.

Parameters:

`display` - the display to copy from.
`xSrc` - the x coordinate of the upper-left corner of the region to copy.
`ySrc` - 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.
`xDest` - the x coordinate of the anchor point in the destination.
`yDest` - the y coordinate of the anchor point in the destination.
`anchor` - the position of the region around the anchor point.

Throws:

`NullPointerException` - if the given display is `null`.
`IllegalArgumentException` - if the given anchor is not a valid value (`BASELINE` is illegal).

See Also:

`Display.getScreenshot()`, `Display.getScreenshot(int, int, int, int)`

---

drawRegion

```
public void drawRegion(Display display,
                       int xSrc,
                       int ySrc,
                       int width,
                       int height,
                       int xDest,
                       int yDest,
                       int anchor,
                       int alpha)
```

Draws a region of a display.

The region of the display to draw is given relative to the display (origin at the upper-left corner) as a rectangle.

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 graphics context area, pixels out of the range will not be drawn.

This method allows to specify the opacity value to apply during the display content rendering.

Parameters:

`display` - the display to copy from.
`xSrc` - the x coordinate of the upper-left corner of the region to copy.
`ySrc` - 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.
`xDest` - the x coordinate of the anchor point in the destination.
`yDest` - the y coordinate of the anchor point in the destination.
`anchor` - the position of the region around the anchor point.
`alpha` - the alpha to apply to the region.

Throws:

`NullPointerException` - if the given display is `null`.
`IllegalArgumentException` - if the given anchor is not a valid value (`BASELINE` is illegal).,
if the given alpha is not a value between `TRANSPARENT` and `OPAQUE`.

Since:

2.0

See Also:

`Display.getScreenshot()`, `Display.getScreenshot(int, int, int, int)`

---

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 @{code 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 @{code 0x00} represents a pixel that is fully transparent and a value of @{code 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 scan length 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 scan length must be greater than or equal to the given width. Negative values of scan length are allowed. In all cases, this must result in every reference being within the bounds of the ARGB pixel data 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 given ARGB pixel data 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 the given scan length is less than the given width.
`NullPointerException` - if the given ARGB pixel data is @{code null}.

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 the given string 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 the given 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 the given offset and length do not specify a valid range within the given string.
`IllegalArgumentException` - if anchor is not a valid value.
`NullPointerException` - if the given string 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 the given 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 the char array.
`length` - the number of characters to draw from the 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 the given offset and length do not specify a valid range within the given char array.
`IllegalArgumentException` - if the given anchor is not a valid value.
`NullPointerException` - if the given char array 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, `false` otherwise.

---

getDisplay

```
public Display getDisplay()
```

Gets the display associated with the GraphicsContext.

Returns:

the display associated with the GraphicsContext

---

getEllipsis

```
public boolean getEllipsis()
```

Gets whether the truncation mechanism is enabled or not.

Returns:

`true` if the truncation mechanism is enabled, `false` otherwise.

Class Image

ej.microui.display

```
java.lang.Object
  └ ej.microui.display.Image
```

```
public class Image
extends Object
```

An `Image` object holds graphical display data.
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, an `OutOfMemoryError` 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.

Nested Class Summary		Page
static enum	**Image.OutputFormat** Specify the format to apply when creating an immutable image.	123

Method Summary		Page
static Image	**createImage**(Display d, int width, int height) Creates a new mutable image which respects the pixel organization (layout, bpp etc.) of the given display and with the given size.	116
static Image	**createImage**(Image image, int x, int y, int width, int height) Creates an immutable image from another image area.	117
static Image	**createImage**(Image image, int x, int y, int width, int height, Image.OutputFormat format) Creates an immutable image from another image area.	118
static Image	**createImage**(int width, int height) Creates a new mutable image which respects the pixel organization (layout, bpp etc.) of the default display and with the given size.	115
static Image	**createImage**(InputStream stream, int size) Creates an immutable image from an `InputStream`. The resource image format (input format) is a standard image format (`PNG` etc.) According the MicroUI implementation the image can be loaded or not.	118
static Image	**createImage**(InputStream stream, int size, Image.OutputFormat format) Creates an immutable image from an `InputStream`. The resource image format (input format) is a standard image format (`PNG` etc.) According the MicroUI implementation the image can be loaded or not.	119

static Image	**createImage**(String name) Creates an immutable image from a resource. The resource image format (input format) is a standard image format (PNG etc.) or an internal MicroUI implementation specific format.	116
static Image	**createImage**(String name, Image.OutputFormat format) Creates an immutable image from a resource. The resource image format (input format) is a standard image format (PNG etc.) or an internal MicroUI implementation specific format.	117
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.	121
GraphicsContext	**getGraphicsContext**() Returns a new GraphicsContext object to draw on the image.	120
int	**getHeight**() Returns the height of the image in pixels.	120
int	**getWidth**() Returns the width of the image in pixels.	120
boolean	**isMutable**() Tells whether this image is mutable.	120
boolean	**isTransparent**() Tells whether this image contains at least a transparent pixel.	120
int	**readPixel**(int x, int y) Obtains the ARGB color of the pixel at (x,y).	121

Method Detail

createImage

```
public static Image createImage(int width,
                                int height)
```

Creates a new mutable image which respects the pixel organization (layout, bpp etc.) of the default display and with the given size. Every pixel within the newly created image is white. Given `width` and `height` must be greater than zero.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.
`ImageCreationException` - if MicroUI implementation cannot create the image.
`OutOfMemoryError` - if there is not enough room to add a new image.
`IllegalStateException` - if MicroUI is not started.
`SecurityException` - if a security manager exists and does not allow the caller to create an image.

See Also:

`createImage(Display, int, int)`

createImage

```
public static Image createImage(Display d,
                                 int width,
                                 int height)
```

Creates a new mutable image which respects the pixel organization (layout, bpp etc.) of 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
`ImageCreationException` - if MicroUI implementation cannot create the image.
`OutOfMemoryError` - if there is not enough room to add a new image.
`SecurityException` - if a security manager exists and does not allow the caller to create an image.

---

createImage

```
public static Image createImage(String name)
                         throws IOException
```

Creates an immutable image from a resource.

The resource image format (input format) is a standard image format (`PNG` etc.) or an internal MicroUI implementation specific format. According the MicroUI implementation the image can be loaded or not. For instance, when MicroUI implementation does not hold a dynamic `PNG` decoder, load a `PNG` is impossible. The MicroUI implementation is responsible to retrieve the right image decoder and/or loader to decode and/or load the image.
When a resource cannot be loaded, an `IOException` is thrown.

The output format is the best format for the input format. For instance the `PNG` image output format is `Image.OutputFormat.ARGB8888` (32 bits per pixels with 256 levels of transparency).
The output format for the MicroUI implementation specific images formats is often the encoded format.

The effect is identical to
`createImage(name, null);`

Parameters:

`name` - a resource name matching image data in a supported image 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`.
`ImageCreationException` - if MicroUI implementation cannot create the image.
`OutOfMemoryError` - if there is not enough room to add a new image.
`SecurityException` - if a security manager exists and does not allow the caller to create an image.
`IllegalStateException` - if MicroUI is not started.

Since:

2.0

See Also:

`createImage(String, OutputFormat)`

---

createImage

```
public static Image createImage(String name,
                                 Image.OutputFormat format)
                          throws IOException
```

Creates an immutable image from a resource.

The resource image format (input format) is a standard image format (`PNG` etc.) or an internal MicroUI implementation specific format. According the MicroUI implementation the image can be loaded or not. For instance, when MicroUI implementation does not hold a dynamic `PNG` decoder, load a `PNG` is impossible. The MicroUI implementation is responsible to retrieve the right image decoder and/or loader to decode and/or load the image.
When a resource cannot be loaded, an `IOException` is thrown.

The output format is the format specified as paremeter. This format may be different than the best format for the input format (@see `createImage(String)`.

Parameters:

`name` - a resource name matching image data in a supported image format.
`format` - the expected output format or `null` to use the best output format according the input image 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`.
`ImageCreationException` - if MicroUI implementation cannot create the image.
`OutOfMemoryError` - if there is not enough room to add a new image.
`SecurityException` - if a security manager exists and does not allow the caller to create an image.
`IllegalStateException` - if MicroUI is not started.

Since:

2.0

---

createImage

```
public static Image createImage(Image image,
                                int x,
                                int y,
                                int width,
                                int height)
```

Creates an immutable image from another image area. The image is created with the same format as the original image.
The area 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.

The effect is identical to
`createImage(image, x, y, width, height, null);`

Parameters:

`image` - the source image.
`x` - the x coordinate of the area to copy.
`y` - the y coordinate of the area to copy.
`width` - the width of the area to copy.
`height` - the height of the area to copy.

Returns:
an immutable image.
Throws:
`NullPointerException` - if image is `null`.
`IllegalArgumentException` - if the area to copy is out of the bounds of the source image or if either `width` or `height` is zero or negative.
`ImageCreationException` - if MicroUI implementation cannot create the image.
`OutOfMemoryError` - if there is not enough room to add a new image.
`SecurityException` - if a security manager exists and does not allow the caller to create an image.
See Also:
`createImage(Image, int, int, int, int, OutputFormat)`

---

createImage

```
public static Image createImage(Image image,
                                int x,
                                int y,
                                int width,
                                int height,
                                Image.OutputFormat format)
```

Creates an immutable image from another image area. The image is created with the specified format. The area 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 area to copy.
`y` - the y coordinate of the area to copy.
`width` - the width of the area to copy.
`height` - the height of the area to copy.
`format` - the expected output format or `null` to use the source image format.
Returns:
an immutable image
Throws:
`NullPointerException` - if image is `null`.
`IllegalArgumentException` - if the area to copy is out of the bounds of the source image or if either `width` or `height` is zero or negative.
`ImageCreationException` - if MicroUI implementation cannot create the image
`OutOfMemoryError` - if there is not enough room to add a new image.
`SecurityException` - if a security manager exists and does not allow the caller to create an image.
Since:
2.0

---

createImage

```
public static Image createImage(InputStream stream,
                                int size)
                         throws IOException
```

Creates an immutable image from an `InputStream`.

The resource image format (input format) is a standard image format (`PNG` etc.) According the MicroUI implementation the image can be loaded or not. For instance, when MicroUI implementation does not hold a dynamic `PNG` decoder, load a `PNG` is impossible.
The MicroUI implementation is responsible to retrieve the right image decoder and/or loader to decode and/or load the image.

When a resource cannot be loaded, an `IOException` is thrown.

The output format is the best format for the input format. For instance the `PNG` image output format is `Image.OutputFormat.ARGB8888` (32 bits per pixels with 256 levels of transparency).

The effect is identical to
`createImage(stream, size, format);`

Parameters:
`stream` - a stream providing image data.
`size` - the number of bytes to read.
Returns:
the created image
Throws:
`IOException` - if the resource is not found, if the data can not be decoded or if the stream cannot be read.
`NullPointerException` - if stream is `null`.
`ImageCreationException` - if MicroUI implementation cannot create the image.
`OutOfMemoryError` - if there is not enough room to add a new image.
`SecurityException` - if a security manager exists and does not allow the caller to create an image.
`IllegalStateException` - if MicroUI is not started.
Since:
2.0
See Also:
`createImage(InputStream, int, OutputFormat)`

---

createImage

```
public static Image createImage(InputStream stream,
                                int size,
                                Image.OutputFormat format)
                         throws IOException
```

Creates an immutable image from an `InputStream`.

The resource image format (input format) is a standard image format (`PNG` etc.) According the MicroUI implementation the image can be loaded or not. For instance, when MicroUI implementation does not hold a dynamic `PNG` decoder, load a `PNG` is impossible.
The MicroUI implementation is responsible to retrieve the right image decoder and/or loader to decode and/or load the image.
When a resource cannot be loaded, an `IOException` is thrown.

The output format is the format specified as paremeter. This format may be different than the best format for the input format (@see `createImage(InputStream, int)`.

Parameters:
`stream` - a stream providing image data
`size` - the number of bytes to read
`format` - the expected output format or null to use the best output format according the input image format
Returns:
the created image
Throws:
`IOException` - if the resource is not found, if the data can not be decoded or if the stream cannot be read.
`NullPointerException` - if stream is null
`ImageCreationException` - if MicroUI implementation cannot create the image
`SecurityException` - if a security manager exists and does not allow the caller to create an image.
`OutOfMemoryError` - if there is not enough room to add a new image.

`IllegalStateException` - if MicroUI is not started.

Since:

2.0

---

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

---

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.

---

isTransparent

```
public boolean isTransparent()
```

Tells whether this image contains at least a transparent pixel.

Returns:

whether the image contains at least a transparent pixel.

---

readPixel

```
public int readPixel(int x,
                     int y)
```

Obtains the ARGB 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 ARGB color of the pixel

Throws:

`IllegalArgumentException` - if the pixel coordinates are out of bounds of the source graphics context

---

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

Enum Image.OutputFormat

ej.microui.display

```
java.lang.Object
  └java.lang.Enum<Image.OutputFormat>
      └ej.microui.display.Image.OutputFormat
```

All Implemented Interfaces:

Comparable<Image.OutputFormat>, Serializable

Enclosing class:

Image

```
public static enum Image.OutputFormat
extends Enum<Image.OutputFormat>
```

Specify the format to apply when creating an immutable image.

Since:

2.0

See Also:

`Image.createImage(String, OutputFormat)`, `Image.createImage(InputStream, int, OutputFormat)`, `Image.createImage(Image, int, int, int, int, OutputFormat)`

Enum Constant Summary	Page
A8 Each pixel value is stored on 8 bits, in `0xA` format 8 bits for `alpha` (0 to 255) 0 bits for `red` 0 bits for `green` 0 bits for `blue`	125
ARGB1555 Each pixel value is stored on 16 bits, in `0xARGB` format 1 bits for `alpha` (0 to 1) 5 bits for `red` (0 to 31) 5 bits for `green` (0 to 31) 5 bits for `blue` (0 to 31)	125
ARGB4444 Each pixel value is stored on 16 bits, in `0xARGB` format 4 bits for `alpha` (0 to 15) 4 bits for `red` (0 to 15) 4 bits for `green` (0 to 15) 4 bits for `blue` (0 to 15)	125
ARGB8888 Each pixel value is stored on 32 bits, in `0xARGB` format.	124

RGB565 Each pixel value is stored on 16 bits, in 0xRGB format 0 bits for alpha (0 to 255) 5 bits for red (0 to 31) 6 bits for green (0 to 63) 5 bits for blue (0 to 31)	124
RGB888 Each pixel value is stored on 24 bits, in 0xRGB format 0 bits for alpha 8 bits for red (0 to 255) 8 bits for green (0 to 255) 8 bits for blue (0 to 255)	124

Method Summary		Page
static Image.OutputFormat	**valueOf**(String name)	125
static Image.OutputFormat[]	**values**()	125

Enum Constant Detail

ARGB8888

```
public static final Image.OutputFormat ARGB8888
```

Each pixel value is stored on 32 bits, in 0xARGB format.
8 bits for alpha (0 to 255)
8 bits for red (0 to 255)
8 bits for green (0 to 255)
8 bits for blue (0 to 255)

~~RGB888~~

```
public static final Image.OutputFormat RGB888
```

Each pixel value is stored on 24 bits, in 0xRGB format
0 bits for alpha
8 bits for red (0 to 255)
8 bits for green (0 to 255)
8 bits for blue (0 to 255)

~~RGB565~~

```
public static final Image.OutputFormat RGB565
```

Each pixel value is stored on 16 bits, in 0xRGB format
0 bits for alpha (0 to 255)
5 bits for red (0 to 31)
6 bits for green (0 to 63)
5 bits for blue (0 to 31)

ARGB1555

```
public static final Image.OutputFormat ARGB1555
```

Each pixel value is stored on 16 bits, in `0xARGB` format
1 bits for `alpha` (0 to 1)
5 bits for `red` (0 to 31)
5 bits for `green` (0 to 31)
5 bits for `blue` (0 to 31)

ARGB4444

```
public static final Image.OutputFormat ARGB4444
```

Each pixel value is stored on 16 bits, in `0xARGB` format
4 bits for `alpha` (0 to 15)
4 bits for `red` (0 to 15)
4 bits for `green` (0 to 15)
4 bits for `blue` (0 to 15)

A8

```
public static final Image.OutputFormat A8
```

Each pixel value is stored on 8 bits, in `0xA` format
8 bits for `alpha` (0 to 255)
0 bits for `red`
0 bits for `green`
0 bits for `blue`

Method Detail

values

```
public static Image.OutputFormat[] values()
```

valueOf

```
public static Image.OutputFormat valueOf(String name)
```

Class ImageCreationException

ej.microui.display

```
java.lang.Object
  └java.lang.Throwable
      └java.lang.Exception
          └java.lang.RuntimeException
              └ej.microui.display.ImageCreationException
```

All Implemented Interfaces:

Serializable

```
public class ImageCreationException
extends RuntimeException
```

Thrown when an image creation fail.

Since:

2.0

Constructor Summary	Page
ImageCreationException() Constructs an ImageCreationException with no detail message.	126
ImageCreationException(String message) Constructs an ImageCreationException with the specified detail message.	126

Constructor Detail

ImageCreationException

```
public ImageCreationException()
```

Constructs an ImageCreationException with no detail message.

ImageCreationException

```
public ImageCreationException(String message)
```

Constructs an ImageCreationException with the specified detail message.

Parameters:

`message` - the exception message

Class ImagePermission

ej.microui.display

```
java.lang.Object
  └java.security.Permission
      └ej.microui.display.ImagePermission
```

All Implemented Interfaces:

Guard, Serializable

```
public class ImagePermission
extends Permission
```

Permission to create an `Image`. Permission is checked when calling `Image.createImage()` methods.

Since:

2.0

Constructor Summary	Page
ImagePermission() Creates an image permission with `null` as name.	127

Method Summary		Page
boolean	**equals**(Object obj)	127
String	**getActions**()	127
int	**hashCode**()	128
boolean	**implies**(Permission permission)	128

Constructor Detail

ImagePermission

```
public ImagePermission()
```

Creates an image permission with `null` as name.

Method Detail

equals

```
public boolean equals(Object obj)
```

Overrides:

`equals` in class `Permission`

---

getActions

```
public String getActions()
```

Overrides:

`getActions` in class `Permission`

---

hashCode

```
public int hashCode()
```

Overrides:

`hashCode` in class `Permission`

---

implies

```
public boolean implies(Permission permission)
```

Overrides:

`implies` in class `Permission`

Class RenderableString

ej.microui.display

```
java.lang.Object
  └ej.microui.display.RenderableString
```

```
public class RenderableString
extends Object
```

This class associates a string with a font. The string rendering is accelerated because the couple string/font is fixed.

Constructor Summary	Page
RenderableString(String string, Font font) Creates a renderable string for given string and font.	129

Method Summary		Page
void	**draw**(GraphicsContext gc, int x, int y, int anchor) Draws the string using the font and given graphics context color.	130
Font	**getFont**() Gets the font used to draw the string.	130
int	**getHeight**() Gets the height of a line of text with this font and its y ratio.	130
String	**getString**() Gets the string to draw.	129
int	**getWidth**() Gets the width of the string with the font.	130

Constructor Detail

RenderableString

```
public RenderableString(String string,
                        Font font)
```

Creates a renderable string for given string and font.

Parameters:

`string` - the string to render
`font` - the font used to render the string

Throws:

`NullPointerException` - when font or string is null
`IllegalArgumentException` - when string is empty

Method Detail

getString

```
public String getString()
```

Gets the string to draw.

Returns:
the string.

---

getFont

```
public Font getFont()
```

Gets the font used to draw the string.

Returns:
the font.

---

getWidth

```
public int getWidth()
```

Gets the width of the string with the font. Same specification than `Font.stringWidth(String)`.

Returns:
the width taken by the string.
See Also:
`Font.stringWidth(String)`

---

getHeight

```
public int getHeight()
```

Gets the height of a line of text with this font and its y ratio. Same specification than `Font.getHeight()`.

Returns:
height of a line of text with this font.

---

draw

```
public void draw(GraphicsContext gc,
                 int x,
                 int y,
                 int anchor)
```

Draws the string using the font and given graphics context color. Same specification than `GraphicsContext.drawString(String, int, int, int)`

Parameters:
`gc` - the graphics context where draw the string
`x` - the x coordinate of the anchor point.
`y` - the y coordinate of the anchor point.
`anchor` - position of the string around the anchor point.
See Also:
`GraphicsContext.drawString(String, int, int, int)`

Package ej.microui.display.shape

Contains shapes rendering management.

See:
Description

Class Summary		Page
AntiAliasedShapes	The `AntiAliasedShapes` class offers advanced drawing facilities, to render anti aliased lines, circles etc.	132

Enum Summary		Page
AntiAliasedShapes.Cap	Define the cap representation when drawing a circle arc	138

Package ej.microui.display.shape Description

Contains shapes rendering management.

Since:
2.0

Class AntiAliasedShapes

ej.microui.display.shape

```
java.lang.Object
  └ ej.microui.display.shape.AntiAliasedShapes
```

```
public class AntiAliasedShapes
extends Object
```

The `AntiAliasedShapes` class offers advanced drawing facilities, to render anti aliased lines, circles etc.

An `AntiAliasedShapes` instance holds a global state (thickness and fade) for all drawings. Several `AntiAliasedShapes` instances can be created at same time. However a default instance is created on MicroUI framework startup and it always available.

Since:

2.0

Nested Class Summary		Page
static enum	**AntiAliasedShapes.Cap** Define the cap representation when drawing a circle arc	138

Field Summary		Page
static AntiAliasedShapes	**Singleton** Default `AntiAliasedShapes` instance created on MicroUI framework startup.	133

Constructor Summary	Page
AntiAliasedShapes() Creates a new `AntiAliasedShapes` instance.	133

Method Summary		Page
void	**drawCircle**(`GraphicsContext gc, int x, int y, int diameter`) Draws the outline of a circle covering the rectangle specified by its diameter, using the `GraphicsContext` 's 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.	137
void	**drawCircleArc**(`GraphicsContext gc, int x, int y, int diameter, int startAngle, int arcAngle`) Draws the outline of a circular arc covering the specified square, using the current color, stroke style and caps The arc is drawn from `startAngle` up to `arcAngle` degrees.	136

void	**drawEllipse**(GraphicsContext gc, int x, int y, int width, int height) Draws the outline of a ellipse covering the specified rectangle, using the GraphicsContext's 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.	137
void	**drawLine**(GraphicsContext gc, int x1, int y1, int x2, int y2) Draws a line from (x1,y1) to (x2,y2) using the GraphicsContext's current color.	136
void	**drawPoint**(GraphicsContext gc, int x, int y) Draws a point at (x,y) using the GraphicsContext's current color.	136
AntiAliasedShapes.Cap	**getEndCap**() Returns the current shape end cap.	135
int	**getFade**() Returns the current fade.	134
AntiAliasedShapes.Cap	**getStartCap**() Returns the current shape start cap.	135
int	**getThickness**() Returns the current thickness.	134
void	**reset**() Resets the AntiAliasedShapes global state to its initial values.	134
void	**setCaps**(AntiAliasedShapes.Cap start, AntiAliasedShapes.Cap end) Configures the caps representation.	135
void	**setEndCap**(AntiAliasedShapes.Cap cap) Configures the cap representation of end of shape.	135
void	**setFade**(int fade) Apply a new fade.	134
void	**setStartCap**(AntiAliasedShapes.Cap cap) Configures the cap representation of start of shape.	135
void	**setThickness**(int thickness) Apply a new thickness.	134

Field Detail

Singleton

```
public static AntiAliasedShapes Singleton
```

Default AntiAliasedShapes instance created on MicroUI framework startup.

Constructor Detail

AntiAliasedShapes

```
public AntiAliasedShapes()
```

Creates a new AntiAliasedShapes instance. The global state is set to its initial value.

See Also:
`reset()`

Method Detail

reset

```
public void reset()
```

Resets the `AntiAliasedShapes` global state to its initial values. Default `fade` is 1 and default `thickness` is 0.

getFade

```
public int getFade()
```

Returns the current `fade`.

Returns:
the current `fade`.

setFade

```
public void setFade(int fade)
```

Apply a new `fade`.

Parameters:
`fade` - the new fade to apply.
Throws:
`IllegalArgumentException` - when given fade is negative

getThickness

```
public int getThickness()
```

Returns the current `thickness`.

Returns:
the current `thickness`.

setThickness

```
public void setThickness(int thickness)
```

Apply a new `thickness`.

Parameters:
`thickness` - the new thickness to apply.
Throws:
`IllegalArgumentException` - when given thickness is negative

getStartCap

```
public AntiAliasedShapes.Cap getStartCap()
```

Returns the current shape start cap.

Returns:
the current shape start cap.

setStartCap

```
public void setStartCap(AntiAliasedShapes.Cap cap)
```

Configures the cap representation of start of shape. This configuration is used by `drawCircleArc(GraphicsContext, int, int, int, int, int)`.

Parameters:
`cap` - the shape cap configuration

getEndCap

```
public AntiAliasedShapes.Cap getEndCap()
```

Returns the current shape end cap.

Returns:
the current shape end cap.

setEndCap

```
public void setEndCap(AntiAliasedShapes.Cap cap)
```

Configures the cap representation of end of shape. This configuration is used by `drawCircleArc(GraphicsContext, int, int, int, int, int)`.

Parameters:
`cap` - the shape cap configuration

setCaps

```
public void setCaps(AntiAliasedShapes.Cap start,
                    AntiAliasedShapes.Cap end)
```

Configures the caps representation. This configuration is used by `drawCircleArc(GraphicsContext, int, int, int, int, int)`. It is equivalent to the following code sequence:

```
setStartCap(start);
setEndCap(end);
```

Parameters:
`start` - the shape cap configuration
`end` - the shape cap configuration

drawPoint

```
public void drawPoint(GraphicsContext gc,
                      int x,
                      int y)
```

Draws a point at `(x,y)` using the `GraphicsContext`'s current color.

Parameters:

`gc` - the `GraphicsContext` where render the drawing.
`x` - the x coordinate of the point
`y` - the y coordinate of the point

---

drawLine

```
public void drawLine(GraphicsContext gc,
                     int x1,
                     int y1,
                     int x2,
                     int y2)
```

Draws a line from `(x1,y1)` to `(x2,y2)` using the `GraphicsContext`'s current color.

Parameters:

`gc` - the `GraphicsContext` where render the drawing.
`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

---

drawCircleArc

```
public final void drawCircleArc(GraphicsContext gc,
                                int x,
                                int y,
                                int diameter,
                                int startAngle,
                                int arcAngle)
```

Draws the outline of a circular arc covering the specified square, using the current color, stroke style and caps

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 `diameter`.

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 `diameter` 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:

`gc` - the `GraphicsContext` where render the drawing.
`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
`diameter` - the diameter of the arc to draw
`startAngle` - the beginning angle of the arc to draw
`arcAngle` - the angular extent of the arc from `startAngle`

See Also:

`setStartCap(Cap)`, `setEndCap(Cap)`, `setCaps(Cap, Cap)`

---

drawCircle

```
public void drawCircle(GraphicsContext gc,
                       int x,
                       int y,
                       int diameter)
```

Draws the outline of a circle covering the rectangle specified by its diameter, using the `GraphicsContext` 's 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:

`gc` - the `GraphicsContext` where render the drawing.
`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

---

drawEllipse

```
public void drawEllipse(GraphicsContext gc,
                        int x,
                        int y,
                        int width,
                        int height)
```

Draws the outline of a ellipse covering the specified rectangle, using the `GraphicsContext`'s 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:

`gc` - the `GraphicsContext` where render the drawing.
`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

Enum AntiAliasedShapes.Cap

ej.microui.display.shape

```
java.lang.Object
  └java.lang.Enum<AntiAliasedShapes.Cap>
      └ej.microui.display.shape.AntiAliasedShapes.Cap
```

All Implemented Interfaces:

Comparable<AntiAliasedShapes.Cap>, Serializable

Enclosing class:

AntiAliasedShapes

```
public static enum AntiAliasedShapes.Cap
extends Enum<AntiAliasedShapes.Cap>
```

Define the cap representation when drawing a circle arc

See Also:

`AntiAliasedShapes.setStartCap(Cap)`, `AntiAliasedShapes.setEndCap(Cap)`, `AntiAliasedShapes.setCaps(Cap, Cap)`, `AntiAliasedShapes.drawCircleArc(GraphicsContext, int, int, int, int, int)`

Enum Constant Summary	Page
NONE No specific cap is drawn	138
PERPENDICULAR Cap is perpendicular to the line	139
ROUNDED Cap is represented by a semi circle (default configuration)	138

Method Summary		Page
static AntiAliasedShapes.Cap	**valueOf**(String name)	139
static AntiAliasedShapes.Cap[]	**values**()	139

Enum Constant Detail

NONE

```
public static final AntiAliasedShapes.Cap NONE
```

No specific cap is drawn

ROUNDED

```
public static final AntiAliasedShapes.Cap ROUNDED
```

Cap is represented by a semi circle (default configuration)

PERPENDICULAR

```
public static final AntiAliasedShapes.Cap PERPENDICULAR
```

Cap is perpendicular to the line

Method Detail

values

```
public static AntiAliasedShapes.Cap[] values()
```

valueOf

```
public static AntiAliasedShapes.Cap valueOf(String name)
```

Package ej.microui.display.transform

Contains image transformation management.

See:

Description

Class Summary		Page
AbstractTransform	The `AbstractTransform` class holds a generic context for all kinds of transformations (see sub classes).	141
ImageDeformation	This class holds a context in order to perform a free deformation on images.	144
ImageFlip	This class holds a context in order to perform a flip on images (0, 90, 180 or 270 degrees).	146
ImageRotation	This class holds a context in order to perform a free rotation (0 to 360 degrees) on images.	150
ImageScale	This class holds a context in order to draw images with scaling.	154

Enum Summary		Page
ImageFlip.Action	Specify the flip to apply when drawing an image.	148

Package ej.microui.display.transform Description

Contains image transformation management.

Since:

2.0

Class AbstractTransform

ej.microui.display.transform

```
java.lang.Object
  └ej.microui.display.transform.AbstractTransform
```

Direct Known Subclasses:

ImageDeformation, ImageFlip, ImageRotation, ImageScale

```
abstract public class AbstractTransform
extends Object
```

The `AbstractTransform` class holds a generic context for all kinds of transformations (see sub classes).

Since:

2.0

Constructor Summary	Page
AbstractTransform()	141

Method Summary		Page
int	**getAlpha**() Returns the current opacity.	142
int	**getTranslateX**() Returns the current x translation.	143
int	**getTranslateY**() Returns the current y translation.	143
boolean	**isMirrored**() Returns `true` if vertical mirror is enabled.	142
void	**resetAlpha**() Restore the opacity to its default value (`GraphicsContext.OPAQUE`).	142
void	**resetTranslate**() Resets translation (`(0,0)`).	143
void	**setAlpha**(int alpha) Specifiy the global opacity to apply on drawing.	142
void	**setMirror**(boolean applyMirror) Apply a vertical mirror on source image before drawing it.	142
void	**translate**(int x, int y) Translates the transformation origin with the given vector `(x,y)`.	142

Constructor Detail

AbstractTransform

```
public AbstractTransform()
```

Method Detail

setAlpha

```
public void setAlpha(int alpha)
```

Specifiy the global opacity to apply on drawing. This opacity is in addition with the opacity encoded in the pixels of the `Image` to draw.
The default opacity is `GraphicsContext.OPAQUE`.

Parameters:
`alpha` - the opacity

---

resetAlpha

```
public void resetAlpha()
```

Restore the opacity to its default value (`GraphicsContext.OPAQUE`).

---

getAlpha

```
public int getAlpha()
```

Returns the current opacity.

Returns:
the current opacity.

---

setMirror

```
public void setMirror(boolean applyMirror)
```

Apply a vertical mirror on source image before drawing it.

Parameters:
`applyMirror` - enable or not vertical mirror.

---

isMirrored

```
public boolean isMirrored()
```

Returns `true` if vertical mirror is enabled.

Returns:
true if vertical mirror is enabled.

---

translate

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

Translates the transformation origin with the given vector `(x,y)`. Subsequent rendering operations on a `GraphicsContext` will be relative to the new origin (in addition of `GraphicsContext`'s translation.

Parameters:
`x` - the translation for the x coordinate

`y` - the translation for the y coordinate

---

resetTranslate

```
public void resetTranslate()
```

Resets translation (`(0,0)`).

---

getTranslateX

```
public int getTranslateX()
```

Returns the current x translation.

Returns:
x the current x translation.

---

getTranslateY

```
public int getTranslateY()
```

Returns the current y translation.

Returns:
y the current y translation.

Class ImageDeformation

ej.microui.display.transform

```
java.lang.Object
  └ej.microui.display.transform.AbstractTransform
      └ej.microui.display.transform.ImageDeformation
```

```
public class ImageDeformation
extends AbstractTransform
```

This class holds a context in order to perform a free deformation on images. The deformation is specified by a coordinates array .

An image deformation instance holds a global state for all drawings. Several instances can be created at the same time. However a default instance is created on MicroUI framework startup and is always available.

Since:

2.0

Field Summary		Page
static ImageDeformation	**Singleton** Default instance created on MicroUI framework startup.	144

Constructor Summary	Page
ImageDeformation()	145

Method Summary		Page
void	**draw**(GraphicsContext gc, Image image, int[] xys, int x, int y, int anchor) Draws a deformed image at the given anchor point. The image anchor point is at position @{code (x,y)}.	145

Methods inherited from class ej.microui.display.transform.AbstractTransform
getAlpha, getTranslateX, getTranslateY, isMirrored, resetAlpha, resetTranslate, setAlpha, setMirror, translate

Field Detail

Singleton

```
public static ImageDeformation Singleton
```

Default instance created on MicroUI framework startup.

Constructor Detail

ImageDeformation

```
public ImageDeformation()
```

Method Detail

draw

```
public void draw(GraphicsContext gc,
                 Image image,
                 int[] xys,
                 int x,
                 int y,
                 int anchor)
```

Draws a deformed image at the given anchor point.
The image anchor point is at position @{code (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, 0, imgWidth-1, 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:

`gc` - the `GraphicsContext` where to render the drawing.
`image` - the `Image` to draw
`xys` - the array of coordinates : x1,y1,x2,y2,x3,y3,x4,y4.
`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:

`NullPointerException` - if @{code image}, @{code gc} or @{code xys} array is @{code null}
`IllegalArgumentException` - if the @{code xys} length is different than 2*4.

Class ImageFlip

ej.microui.display.transform

```
java.lang.Object
  └ej.microui.display.transform.AbstractTransform
      └ej.microui.display.transform.ImageFlip
```

```
public class ImageFlip
extends AbstractTransform
```

This class holds a context in order to perform a flip on images (0, 90, 180 or 270 degrees).

An image flip instance holds a global state for all drawings. Several instances can be created at the same time. However a default instance is created on MicroUI framework startup and is always available.

Since:
2.0

Nested Class Summary		Page
static enum	**ImageFlip.Action** Specify the flip to apply when drawing an image.	148

Field Summary		Page
static ImageFlip	**Singleton** Default instance created on MicroUI framework startup.	147

Constructor Summary	Page
ImageFlip()	147

Method Summary		Page
void	**draw**(GraphicsContext gc, Image image, int x, int y, int anchor) Draw the Image in the GraphicsContext at given anchor position and using the current ImageFlip.Action.	147
ImageFlip.Action	**getAction**() Returns the current ImageFlip.Action.	147
void	**setAction**(ImageFlip.Action action) Set the new ImageFlip.Action.	147

Methods inherited from class ej.microui.display.transform.AbstractTransform
getAlpha, getTranslateX, getTranslateY, isMirrored, resetAlpha, resetTranslate, setAlpha, setMirror, translate

Field Detail

Singleton

```
public static final ImageFlip Singleton
```

Default instance created on MicroUI framework startup.

Constructor Detail

ImageFlip

```
public ImageFlip()
```

Method Detail

getAction

```
public ImageFlip.Action getAction()
```

Returns the current `ImageFlip.Action`.

Returns:

the current `ImageFlip.Action`.

---

setAction

```
public void setAction(ImageFlip.Action action)
```

Set the new `ImageFlip.Action`.

Parameters:

`action` - the action to set.

---

draw

```
public void draw(GraphicsContext gc,
                 Image image,
                 int x,
                 int y,
                 int anchor)
```

Draw the `Image` in the `GraphicsContext` at given anchor position and using the current `ImageFlip.Action`.

Parameters:

`gc` - the `GraphicsContext` where to render the drawing.
`image` - the `Image` to draw
`x` - the x coordinate of the image reference anchor point
`y` - the y coordinate of the image reference anchor point
`anchor` - position of the image reference point around the anchor point

Throws:

`NullPointerException` - if @{code image} or @{code gc} is @{code null}
`IllegalArgumentException` - if `anchor` is not a valid value (`BASELINE` is illegal).

Enum ImageFlip.Action

ej.microui.display.transform

```
java.lang.Object
  └java.lang.Enum<ImageFlip.Action>
      └ej.microui.display.transform.ImageFlip.Action
```

All Implemented Interfaces:

Comparable<ImageFlip.Action>, Serializable

Enclosing class:

ImageFlip

```
public static enum ImageFlip.Action
extends Enum<ImageFlip.Action>
```

Specify the flip to apply when drawing an image.

See Also:

`ImageFlip.draw(GraphicsContext, Image, int, int, int)`

Enum Constant Summary	Page
FLIP_180 Flip at 180 degrees.	149
FLIP_270 Flip anticlockwise at 270 degrees.	149
FLIP_90 Flip anticlockwise at 90 degrees.	148
FLIP_NONE Reset the flip action (0 degrees). This is the default action.	148

Method Summary		Page
static ImageFlip.Action	**valueOf**(String name)	149
static ImageFlip.Action[]	**values**()	149

Enum Constant Detail

FLIP_NONE

```
public static final ImageFlip.Action FLIP_NONE
```

Reset the flip action (0 degrees).
This is the default action.

FLIP_90

```
public static final ImageFlip.Action FLIP_90
```

Flip anticlockwise at 90 degrees.

FLIP_180

```
public static final ImageFlip.Action FLIP_180
```

Flip at 180 degrees.

FLIP_270

```
public static final ImageFlip.Action FLIP_270
```

Flip anticlockwise at 270 degrees.

Method Detail

values

```
public static ImageFlip.Action[] values()
```

valueOf

```
public static ImageFlip.Action valueOf(String name)
```

Class ImageRotation

ej.microui.display.transform

```
java.lang.Object
  └ej.microui.display.transform.AbstractTransform
      └ej.microui.display.transform.ImageRotation
```

```
public class ImageRotation
extends AbstractTransform
```

This class holds a context in order to perform a free rotation (0 to 360 degrees) on images. The rotation is specified by the center and the angle. The reference point is the image top-left corner. The rotation point is relative to the current translation of the graphics context where the image will be drawn. The image anchor point is relative to the same translation and to the given alignment.

To rotate an image on itself, use the following lines:

```
ImageRotation rotation = new ImageRotation();
int imageWidth = image.getWidth();
int imageHeight = image.getHeight();
int rx = x + imageWidth / 2;
int ry = y + imageHeight / 2;
rotation.setRotationCenter(rx, ry);
rotation.setAngle(78);
rotation.drawNearestNeighbor(gc, image, rx, ry, GraphicsContext.HCENTER |
GraphicsContext.VCENTER);
```

To rotate an image around a circle, use the following lines:

```
ImageRotation rotation = new ImageRotation();
rotation.setRotationCenter(rx, ry);
for (int i = 0; i < 360; i += 45) {
rotation.setAngle(i);
rotation.drawBilinear(gc, image, rx - diameter / 2, ry - diameter / 2,
GraphicsContext.TOP | GraphicsContext.LEFT);
}
```

An image rotation instance holds a global state for all drawings. Several instances can be created at the same time. However a default instance is created on MicroUI framework startup and is always available.

Since:

2.0

Field Summary		Page
static ImageRotation	**Singleton** Default instance created on MicroUI framework startup.	151

Constructor Summary	Page
ImageRotation()	151

Method Summary	Page

void	**drawBilinear**(GraphicsContext gc, Image image, int x, int y, int anchor) Draws the given Image applying the current rotation.	152
void	**drawNearestNeighbor**(GraphicsContext gc, Image image, int x, int y, int anchor) Draws the given Image applying the current rotation.	153
int	**getAngle**() Returns the current rotation angle.	151
int	**getRotationX**() Returns the current X coordinate.	152
int	**getRotationY**() Returns the current Y coordinate.	152
void	**setAngle**(int angle) Set the new rotation angle.	151
void	**setRotationCenter**(int x, int y) Set the new rotation center coordinates.	152

Methods inherited from class ej.microui.display.transform.AbstractTransform
getAlpha, getTranslateX, getTranslateY, isMirrored, resetAlpha, resetTranslate, setAlpha, setMirror, translate

Field Detail

Singleton

```
public static final ImageRotation Singleton
```

Default instance created on MicroUI framework startup.

Constructor Detail

ImageRotation

```
public ImageRotation()
```

Method Detail

getAngle

```
public int getAngle()
```

Returns the current rotation angle.

Returns:
the current rotation angle.

---

setAngle

```
public void setAngle(int angle)
```

Set the new rotation angle.

Parameters:
`angle` - the new rotation angle

---

setRotationCenter

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

Set the new rotation center coordinates.

Parameters:
`x` - the x coordinate
`y` - the y coordinate

---

getRotationX

```
public int getRotationX()
```

Returns the current X coordinate.

Returns:
the current X coordinate.

---

getRotationY

```
public int getRotationY()
```

Returns the current Y coordinate.

Returns:
the current Y coordinate.

---

drawBilinear

```
public void drawBilinear(GraphicsContext gc,
                         Image image,
                         int x,
                         int y,
                         int anchor)
```

Draws the given `Image` applying the current rotation. This method uses the `bilinear` algorithm to render the image. This algorithm performs better rendering than `nearest neighbor` algorithm but it is slower to apply.

Parameters:
`gc` - the `GraphicsContext` where to render the drawing.
`image` - the `Image` to draw
`x` - the x coordinate of the image reference anchor point
`y` - the y coordinate of the image reference anchor point
`anchor` - position of the image reference point around the anchor point
Throws:
`NullPointerException` - if @{code image} or @{code gc} is @{code null}
`IllegalArgumentException` - if `anchor` is not a valid value (`BASELINE` is illegal).
See Also:
`drawNearestNeighbor(GraphicsContext, Image, int, int, int)`

---

drawNearestNeighbor

```
public void drawNearestNeighbor(GraphicsContext gc,
                                Image image,
                                int x,
                                int y,
                                int anchor)
```

Draws the given `Image` applying the current rotation. This method uses the `nearest neighbor` algorithm to render the image. This algorithm is faster than `bilinear` algorithm but its rendering is more simple.

Parameters:

`gc` - the `GraphicsContext` where render the drawing.
`image` - the `Image` to draw
`x` - the x coordinate of the image reference anchor point
`y` - the y coordinate of the image reference anchor point
`anchor` - position of the image reference point around the anchor point

Throws:

`NullPointerException` - if @{code image} or @{code gc} is @{code null}
`IllegalArgumentException` - if `anchor` is not a valid value (`BASELINE` is illegal).

See Also:

`drawBilinear(GraphicsContext, Image, int, int, int)`

Class ImageScale

ej.microui.display.transform

```
java.lang.Object
  └ej.microui.display.transform.AbstractTransform
      └ej.microui.display.transform.ImageScale
```

```
public class ImageScale
extends AbstractTransform
```

This class holds a context in order to draw images with scaling.

An image scale instance holds a global state for all drawings. Several instances can be created at the same time. However a default instance is created on MicroUI framework startup and is always available.

Since:
2.0

Field Summary		Page
static ImageScale	**Singleton** Default instance created on MicroUI framework startup.	155

Constructor Summary	Page
ImageScale()	155

Method Summary		Page
void	**draw**(GraphicsContext gc, Image image, int x, int y, int anchor) **Deprecated.** Use drawNearestNeighbor(GraphicsContext, Image, int, int, int) or drawBilinear(GraphicsContext, Image, int, int, int) instead.	156
void	**drawBilinear**(GraphicsContext gc, Image image, int x, int y, int anchor) Draw the Image in the GraphicsContext at given anchor position and using the current scaling factor.	157
void	**drawNearestNeighbor**(GraphicsContext gc, Image image, int x, int y, int anchor) Draw the Image in the GraphicsContext at given anchor position and using the current scaling factor.	157
float	**getFactor**() **Deprecated.** Use getFactorX() or getFactorY() instead.	155
float	**getFactorX**() Returns the current scaling X factor.	155
float	**getFactorY**() Returns the current scaling Y factor.	155
void	**setFactor**(float factor) Set a new scaling factor.	156

void	**setFactorX**(float factor) Set a new scaling X factor.	156
void	**setFactorY**(float factor) Set a new scaling Y factor.	156

Methods inherited from class ej.microui.display.transform.AbstractTransform
getAlpha, getTranslateX, getTranslateY, isMirrored, resetAlpha, resetTranslate, setAlpha, setMirror, translate

Field Detail

Singleton

```
public static ImageScale Singleton
```

Default instance created on MicroUI framework startup.

Constructor Detail

ImageScale

```
public ImageScale()
```

Method Detail

getFactor

```
public float getFactor()
```

Deprecated. *Use* `getFactorX()` *or* `getFactorY()` *instead.*

Returns:
the current scaling X factor.

---

getFactorX

```
public float getFactorX()
```

Returns the current scaling X factor.

Returns:
the current scaling X factor.

---

getFactorY

```
public float getFactorY()
```

Returns the current scaling Y factor.

Returns:
the current scaling Y factor.

---

setFactor

```
public void setFactor(float factor)
```

Set a new scaling factor. Equivalent to
`setFactorX(factor);`
`setFactorY(factor);`

Parameters:
`factor` - the new scaling factor
Throws:
`IllegalArgumentException` - if `factor` is lower than or equal to zero

setFactorX

```
public void setFactorX(float factor)
```

Set a new scaling X factor.

Parameters:
`factor` - the new scaling X factor
Throws:
`IllegalArgumentException` - if `factor` is lower than or equal to zero

setFactorY

```
public void setFactorY(float factor)
```

Set a new scaling Y factor.

Parameters:
`factor` - the new scaling Y factor
Throws:
`IllegalArgumentException` - if `factor` is lower than or equal to zero

draw

```
public void draw(GraphicsContext gc,
                 Image image,
                 int x,
                 int y,
                 int anchor)
```

Deprecated. *Use `drawNearestNeighbor(GraphicsContext, Image, int, int, int)` or `drawBilinear(GraphicsContext, Image, int, int, int)` instead.*

Parameters:
`gc` - the `GraphicsContext` where to render the drawing.
`image` - the `Image` to draw
`x` - the x coordinate of the image reference anchor point
`y` - the y coordinate of the image reference anchor point
`anchor` - position of the image reference point around the anchor point

drawBilinear

```
public void drawBilinear(GraphicsContext gc,
                         Image image,
                         int x,
                         int y,
                         int anchor)
```

Draw the `Image` in the `GraphicsContext` at given anchor position and using the current scaling factor. This method uses the `bilinear` algorithm to render the image. This algorithm performs better rendering than `nearest neighbor` algorithm but it is slower to apply.

Parameters:

`gc` - the `GraphicsContext` where to render the drawing.
`image` - the `Image` to draw
`x` - the x coordinate of the image reference anchor point
`y` - the y coordinate of the image reference anchor point
`anchor` - position of the image reference point around the anchor point

drawNearestNeighbor

```
public void drawNearestNeighbor(GraphicsContext gc,
                                Image image,
                                int x,
                                int y,
                                int anchor)
```

Draw the `Image` in the `GraphicsContext` at given anchor position and using the current scaling factor. This method uses the `nearest neighbor` algorithm to render the image. This algorithm is faster than `bilinear` algorithm but its rendering is more simple.

Parameters:

`gc` - the `GraphicsContext` where to render the drawing.
`image` - the `Image` to draw
`x` - the x coordinate of the image reference anchor point
`y` - the y coordinate of the image reference anchor point
`anchor` - position of the image reference point around the anchor point

Package ej.microui.event

Contains events management.

See:
Description

Class Summary		Page
Event	MicroUI features int-based events, allowing for a rich event mechanism compatible with scare resources.	159
EventGenerator	Event generators generate int-based events (see `Event`).	163
EventPermission	Permission to handle events generated by an `EventGenerator`.	167

Package ej.microui.event Description

Contains events management.

Class Event

ej.microui.event

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

```
public class Event
extends Object
```

MicroUI features int-based events, allowing for a rich event mechanism compatible with scare resources. This 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.	160
static int	**COMMAND** The COMMAND event type.	160
static int	**KEYBOARD** The KEYBOARD event type.	160
static int	**KEYPAD** The KEYPAD event type.	160
static int	**POINTER** The POINTER event type.	160
static int	**STATE** The STATE event type.	160

Method Summary		Page
static int	**buildEvent**(int type, EventGenerator gen, int data) Builds an event from a given type, an eventGenerator and data.	161

static int	**getData**(int event) Returns the event's data issued by a generator.	161
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.	161
static int	**getGeneratorID**(int event) Returns the event's generator id.	161
static int	**getType**(int event) Returns the type of an event.	161

Field Detail

COMMAND

```
public static final int COMMAND
```

The COMMAND event type.

~~BUTTON~~

```
public static final int BUTTON
```

The BUTTON event type.

~~KEYBOARD~~

```
public static final int KEYBOARD
```

The KEYBOARD event type.

~~POINTER~~

```
public static final int POINTER
```

The POINTER event type.

~~KEYPAD~~

```
public static final int KEYPAD
```

The KEYPAD event type.

~~STATE~~

```
public static final int STATE
```

The STATE event type.

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.event

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

Direct Known Subclasses:

Buttons, Command, GenericEventGenerator, Keyboard, States

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

Event generators generate int-based events (see `Event`). They are responsible for holding data that cannot be sent within the event. Event generators 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 event generator.	164

Method Summary		Page
int	**addToSystemPool**() Adds the generator in the system pool.	164
static EventGenerator	**get**(int id) Gets a generator from its id.	164
static EventGenerator[]	**get**(Class<E> clazz) Gets all generators of the given type from the system pool in an array.	165
static E	**get**(Class<E> clazz, int fromIndex) Gets a generator whose class is `clazz` from the system pool starting the search from `fromIndex`.	165
EventHandler	**getEventHandler**() Gets the generator's event handler.	166
abstract int	**getEventType**() Gets the event type associated with the event generator	166
int	**getID**() Gets the generator's unique id.	165
static List<E>	**getList**(Class<E> clazz) Gets all generators of the given type from the system pool in a list.	165
void	**removeFromSystemPool**() Removes the generator from the system generators pool.	164

protected void	**sendEvent**(int event) Sends the given event to the current EventHandler.	166
void	**setEventHandler**(EventHandler eventHandler) Sets an event handler to this event generator.	164

Constructor Detail

EventGenerator

```
public EventGenerator()
```

Creates a new event generator. As soon as a new event generator 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 event generator's id in the system pool.

Throws:
`RuntimeException` - if the maximum number of event generators 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.

---

setEventHandler

```
public void setEventHandler(EventHandler eventHandler)
                    throws SecurityException
```

Sets an event handler to this event generator. It will receive the generated events.

It replaces the old one and can be `null`.

Parameters:
`eventHandler` - the event handler to add.

Throws:
`SecurityException` - if a security manager exists and does not allow the caller to handle events from this event generator.

Since:
2.0

---

get

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

Gets a generator from its id.

Parameters:
`id` - the generator's id.
Returns:
the associated event generator.
Throws:
`IndexOutOfBoundsException` - if the generator does not exist.

---

get

```
public static E get(Class<E> 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`.
Throws:
`NullPointerException` - if the given class is `null`.

---

get

```
public static EventGenerator[] get(Class<E> clazz)
```

Gets all generators of the given type from the system pool in an array.

Parameters:
`clazz` - the type of the event generators to return.
Returns:
the arrays of event generators.

---

getList

```
public static List<E> getList(Class<E> clazz)
```

Gets all generators of the given type from the system pool in a list.

Parameters:
`clazz` - the type of the event generators to return.
Returns:
the list of event generators.
Since:
2.0

---

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`.

---

getEventHandler

```
public EventHandler getEventHandler()
```

Gets the generator's event handler.

Returns:
the generator's event handler.
Since:
2.0

---

getEventType

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

Gets the event type associated with the event generator

Returns:
the event type

---

sendEvent

```
protected void sendEvent(int event)
```

Sends the given event to the current `EventHandler`. Does nothing if there is no event handler set to this event generator.

Parameters:
`event` - the event to send

Class EventPermission

ej.microui.event

```
java.lang.Object
  └java.security.Permission
      └ej.microui.event.EventPermission
```

All Implemented Interfaces:

Guard, Serializable

```
public class EventPermission
extends Permission
```

Permission to handle events generated by an `EventGenerator`. Permission is checked when calling `EventGenerator.setEventHandler(EventHandler)`.

Constructor Summary	Page
EventPermission(EventGenerator gen) Creates a permission for events generated by the given event generator with `null` as name.	167

Method Summary		Page
boolean	**equals**(Object obj)	168
String	**getActions**()	168
EventGenerator	**getEventGenerator**() Gets the event generator handled by this permission.	167
int	**hashCode**()	168
boolean	**implies**(Permission permission)	168

Constructor Detail

EventPermission

```
public EventPermission(EventGenerator gen)
```

Creates a permission for events generated by the given event generator with `null` as name.

Parameters:

`gen` - the event generator.

Method Detail

getEventGenerator

```
public EventGenerator getEventGenerator()
```

Gets the event generator handled by this permission.

Returns:
the event generator.

---

equals

```
public boolean equals(Object obj)
```

Overrides:
`equals` in class `Permission`

---

getActions

```
public String getActions()
```

Overrides:
`getActions` in class `Permission`

---

hashCode

```
public int hashCode()
```

Overrides:
`hashCode` in class `Permission`

---

implies

```
public boolean implies(Permission permission)
```

Overrides:
`implies` in class `Permission`

Package ej.microui.event.controller

Contains helpers to handle events.

See:
Description

Interface Summary		Page
ButtonEventHandler	Event handler that manages `Buttons` events.	170
CommandEventHandler	Event handler that manages `Command` events.	172
EventGeneratorsHandler	Event handler that manages MicroUI events type.	183
PointerEventHandler	Event handler that manages `Pointer` events.	185

Class Summary		Page
DispatchEventHandler	Dispatches events by event type (defined `Event`) and data (event type specific).	173
DispatchHelper	Helps dispatch some sorts of events.	181

Package ej.microui.event.controller Description

Contains helpers to handle events.

Since:
2.0

Interface ButtonEventHandler

ej.microui.event.controller

All Known Implementing Classes:

DispatchEventHandler

`public interface` **`ButtonEventHandler`**

Event handler that manages `Buttons` events.

Since:

2.0

Method Summary		Page
`boolean`	**onButtonClicked**`(int event)` Handles button clicked event.	171
`boolean`	**onButtonDoubleClicked**`(int event)` Handles button double-clicked event.	171
`boolean`	**onButtonPressed**`(int event)` Handles button pressed event.	170
`boolean`	**onButtonReleased**`(int event)` Handles button released event.	170
`boolean`	**onButtonRepeated**`(int event)` Handles button repeated event.	171

Method Detail

onButtonPressed

`boolean` **`onButtonPressed`**`(int event)`

Handles button pressed event.

Parameters:

`event` - the button event to handle.

Returns:

`true` if the event is consumed, `false` otherwise.

---

onButtonReleased

`boolean` **`onButtonReleased`**`(int event)`

Handles button released event.

Parameters:

`event` - the button event to handle.

Returns:

`true` if the event is consumed, `false` otherwise.

---

onButtonClicked

```
boolean onButtonClicked(int event)
```

Handles button clicked event.

Parameters:
`event` - the button event to handle.
Returns:
`true` if the event is consumed, `false` otherwise.

onButtonDoubleClicked

```
boolean onButtonDoubleClicked(int event)
```

Handles button double-clicked event.

Parameters:
`event` - the button event to handle.
Returns:
`true` if the event is consumed, `false` otherwise.

onButtonRepeated

```
boolean onButtonRepeated(int event)
```

Handles button repeated event.

Parameters:
`event` - the button event to handle.
Returns:
`true` if the event is consumed, `false` otherwise.

Interface CommandEventHandler

ej.microui.event.controller

All Known Implementing Classes:

DispatchEventHandler

`public interface` **`CommandEventHandler`**

Event handler that manages `Command` events.

Since:

2.0

Method Summary		Page
`boolean`	**`onCommand`**`(int command)` Handles command event.	172

Method Detail

onCommand

`boolean` **`onCommand`**`(int command)`

Handles command event.

Parameters:

`command` - the command to handle.

Returns:

`true` if the event is consumed, `false` otherwise.

Class DispatchEventHandler

ej.microui.event.controller

```
java.lang.Object
  └ ej.microui.event.controller.DispatchEventHandler
```

All Implemented Interfaces:

ButtonEventHandler, CommandEventHandler, EventGeneratorsHandler, EventHandler, PointerEventHandler

---

```
public class DispatchEventHandler
extends Object
implements EventHandler, EventGeneratorsHandler, ButtonEventHandler, CommandEventHandler,
PointerEventHandler
```

Dispatches events by event type (defined `Event`) and data (event type specific).

Since:

2.0

Constructor Summary	Page
DispatchEventHandler() Creates a dispatch event handler.	174

Method Summary		Page
boolean	**handleButton**(int event) Handles button events.	174
boolean	**handleCommand**(int event) Handles command events.	175
boolean	**handleEvent**(int event) Handles an event.	174
boolean	**handleKeyboard**(int event) Handles keyboard events.	175
boolean	**handleKeypad**(int event) Handles keypad events.	175
boolean	**handlePointer**(int event) Handles pointer events.	175
boolean	**onButtonClicked**(int event) Handles button clicked event.	176
boolean	**onButtonDoubleClicked**(int event) Handles button double-clicked event.	176
boolean	**onButtonPressed**(int event) Handles button pressed event.	176
boolean	**onButtonReleased**(int event) Handles button released event.	176
boolean	**onButtonRepeated**(int event) Handles button repeated event.	177

boolean	**onCommand**(int command) Handles command event.	177
boolean	**onPointerClicked**(Pointer pointer, int pointerX, int pointerY, int event) Handles pointer clicked event.	179
boolean	**onPointerDoubleClicked**(Pointer pointer, int pointerX, int pointerY, int event) Handles pointer double-clicked event.	180
boolean	**onPointerDragged**(Pointer pointer, int pointerX, int pointerY, int event) Handles pointer dragged event.	178
boolean	**onPointerEntered**(Pointer pointer, int pointerX, int pointerY, int event) Handles pointer entered event.	179
boolean	**onPointerExited**(Pointer pointer, int pointerX, int pointerY, int event) Handles pointer exited event.	179
boolean	**onPointerMoved**(Pointer pointer, int pointerX, int pointerY, int event) Handles pointer moved event.	178
boolean	**onPointerPressed**(Pointer pointer, int pointerX, int pointerY, int event) Handles pointer pressed event.	177
boolean	**onPointerReleased**(Pointer pointer, int pointerX, int pointerY, int event) Handles pointer released event.	178

Constructor Detail

DispatchEventHandler

```
public DispatchEventHandler()
```

Creates a dispatch event handler.

Method Detail

handleEvent

```
public boolean handleEvent(int event)
```

Description copied from interface: `EventHandler`
Handles an event.

Specified by:
handleEvent in interface `EventHandler`

Parameters:
`event` - the event to handle.

Returns:
`true` if the event is consumed, `false` otherwise.

---

handleButton

```
public boolean handleButton(int event)
```

Handles button events.

Specified by:
handleButton in interface `EventGeneratorsHandler`

Parameters:
`event` - the button event to handle.
Returns:
`true` if the event is consumed, `false` otherwise.
See Also:
`DispatchHelper.dispatchButton(int, ButtonEventHandler)`

---

handleCommand

```
public boolean handleCommand(int event)
```

Handles command events.

Calls `onCommand(int)` method passing the event data.

Specified by:
handleCommand in interface `EventGeneratorsHandler`
Parameters:
`event` - the command event to handle.
Returns:
`true` if the event is consumed, `false` otherwise.

---

handleKeyboard

```
public boolean handleKeyboard(int event)
```

Description copied from interface: `EventGeneratorsHandler`
Handles keyboard events.

Specified by:
handleKeyboard in interface `EventGeneratorsHandler`
Parameters:
`event` - the keyboard event to handle.
Returns:
`true` if the event is consumed, `false` otherwise.

---

handleKeypad

```
public boolean handleKeypad(int event)
```

Description copied from interface: `EventGeneratorsHandler`
Handles keypad events.

Specified by:
handleKeypad in interface `EventGeneratorsHandler`
Parameters:
`event` - the keypad event to handle.
Returns:
`true` if the event is consumed, `false` otherwise.

---

handlePointer

```
public boolean handlePointer(int event)
```

Handles pointer events.

Specified by:
handlePointer in interface `EventGeneratorsHandler`
Parameters:
`event` - the pointer event to handle.
Returns:
`true` if the event is consumed, `false` otherwise.
See Also:
`DispatchHelper.dispatchPointer(int, PointerEventHandler)`

---

onButtonPressed

```
public boolean onButtonPressed(int event)
```

Description copied from interface: `ButtonEventHandler`
Handles button pressed event.

Specified by:
onButtonPressed in interface `ButtonEventHandler`
Parameters:
`event` - the button event to handle.
Returns:
`true` if the event is consumed, `false` otherwise.

---

onButtonReleased

```
public boolean onButtonReleased(int event)
```

Description copied from interface: `ButtonEventHandler`
Handles button released event.

Specified by:
onButtonReleased in interface `ButtonEventHandler`
Parameters:
`event` - the button event to handle.
Returns:
`true` if the event is consumed, `false` otherwise.

---

onButtonClicked

```
public boolean onButtonClicked(int event)
```

Description copied from interface: `ButtonEventHandler`
Handles button clicked event.

Specified by:
onButtonClicked in interface `ButtonEventHandler`
Parameters:
`event` - the button event to handle.
Returns:
`true` if the event is consumed, `false` otherwise.

---

onButtonDoubleClicked

```
public boolean onButtonDoubleClicked(int event)
```

Description copied from interface: `ButtonEventHandler`
Handles button double-clicked event.

Specified by:
onButtonDoubleClicked in interface `ButtonEventHandler`
Parameters:
`event` - the button event to handle.
Returns:
`true` if the event is consumed, `false` otherwise.

---

onButtonRepeated

```
public boolean onButtonRepeated(int event)
```

Description copied from interface: `ButtonEventHandler`
Handles button repeated event.

Specified by:
onButtonRepeated in interface `ButtonEventHandler`
Parameters:
`event` - the button event to handle.
Returns:
`true` if the event is consumed, `false` otherwise.

---

onCommand

```
public boolean onCommand(int command)
```

Description copied from interface: `CommandEventHandler`
Handles command event.

Specified by:
onCommand in interface `CommandEventHandler`
Parameters:
`command` - the command to handle.
Returns:
`true` if the event is consumed, `false` otherwise.

---

onPointerPressed

```
public boolean onPointerPressed(Pointer pointer,
                                int pointerX,
                                int pointerY,
                                int event)
```

Description copied from interface: `PointerEventHandler`
Handles pointer pressed event.

Specified by:
onPointerPressed in interface `PointerEventHandler`
Parameters:
`pointer` - the pointer that generates the event.
`pointerX` - the current x coordinate of the pointer.
`pointerY` - the current y coordinate of the pointer.
`event` - the pointer event to handle.
Returns:
`true` if the event is consumed, `false` otherwise.

---

onPointerReleased

```
public boolean onPointerReleased(Pointer pointer,
                                 int pointerX,
                                 int pointerY,
                                 int event)
```

Description copied from interface: `PointerEventHandler`
Handles pointer released event.

Specified by:
onPointerReleased in interface `PointerEventHandler`

Parameters:
`pointer` - the pointer that generates the event.
`pointerX` - the current x coordinate of the pointer.
`pointerY` - the current y coordinate of the pointer.
`event` - the pointer event to handle.

Returns:
`true` if the event is consumed, `false` otherwise.

onPointerMoved

```
public boolean onPointerMoved(Pointer pointer,
                              int pointerX,
                              int pointerY,
                              int event)
```

Description copied from interface: `PointerEventHandler`
Handles pointer moved event.

Specified by:
onPointerMoved in interface `PointerEventHandler`

Parameters:
`pointer` - the pointer that generates the event.
`pointerX` - the current x coordinate of the pointer.
`pointerY` - the current y coordinate of the pointer.
`event` - the pointer event to handle.

Returns:
`true` if the event is consumed, `false` otherwise.

onPointerDragged

```
public boolean onPointerDragged(Pointer pointer,
                                int pointerX,
                                int pointerY,
                                int event)
```

Description copied from interface: `PointerEventHandler`
Handles pointer dragged event.

Specified by:
onPointerDragged in interface `PointerEventHandler`

Parameters:
`pointer` - the pointer that generates the event.
`pointerX` - the current x coordinate of the pointer.
`pointerY` - the current y coordinate of the pointer.
`event` - the pointer event to handle.

Returns:
`true` if the event is consumed, `false` otherwise.

onPointerEntered

```
public boolean onPointerEntered(Pointer pointer,
                                int pointerX,
                                int pointerY,
                                int event)
```

Description copied from interface: `PointerEventHandler`
Handles pointer entered event.

Specified by:
onPointerEntered in interface `PointerEventHandler`

Parameters:
`pointer` - the pointer that generates the event.
`pointerX` - the current x coordinate of the pointer.
`pointerY` - the current y coordinate of the pointer.
`event` - the pointer event to handle.

Returns:
`true` if the event is consumed, `false` otherwise.

onPointerExited

```
public boolean onPointerExited(Pointer pointer,
                               int pointerX,
                               int pointerY,
                               int event)
```

Description copied from interface: `PointerEventHandler`
Handles pointer exited event.

Specified by:
onPointerExited in interface `PointerEventHandler`

Parameters:
`pointer` - the pointer that generates the event.
`pointerX` - the current x coordinate of the pointer.
`pointerY` - the current y coordinate of the pointer.
`event` - the pointer event to handle.

Returns:
`true` if the event is consumed, `false` otherwise.

onPointerClicked

```
public boolean onPointerClicked(Pointer pointer,
                                int pointerX,
                                int pointerY,
                                int event)
```

Description copied from interface: `PointerEventHandler`
Handles pointer clicked event.

Specified by:
onPointerClicked in interface `PointerEventHandler`

Parameters:
`pointer` - the pointer that generates the event.
`pointerX` - the current x coordinate of the pointer.
`pointerY` - the current y coordinate of the pointer.
`event` - the pointer event to handle.

Returns:
`true` if the event is consumed, `false` otherwise.

onPointerDoubleClicked

```
public boolean onPointerDoubleClicked(Pointer pointer,
                                      int pointerX,
                                      int pointerY,
                                      int event)
```

Description copied from interface: **`PointerEventHandler`**
Handles pointer double-clicked event.

Specified by:
onPointerDoubleClicked in interface `PointerEventHandler`

Parameters:
`pointer` - the pointer that generates the event.
`pointerX` - the current x coordinate of the pointer.
`pointerY` - the current y coordinate of the pointer.
`event` - the pointer event to handle.

Returns:
`true` if the event is consumed, `false` otherwise.

Class DispatchHelper

ej.microui.event.controller

```
java.lang.Object
  └ej.microui.event.controller.DispatchHelper
```

```
public class DispatchHelper
extends Object
```

Helps dispatch some sorts of events.

Since:

2.0

Method Summary		Page
static boolean	**dispatchButton**(int event, ButtonEventHandler buttonEventHandler) Dispatch button events.	181
static boolean	**dispatchEvent**(int event, EventGeneratorsHandler eventGeneratorsHandler) Dispatch MicroUI events.	181
static boolean	**dispatchPointer**(int event, PointerEventHandler pointerEventHandler) Dispatch pointer events.	182

Method Detail

dispatchEvent

```
public static boolean dispatchEvent(int event,
                                    EventGeneratorsHandler eventGeneratorsHandler)
```

Dispatch MicroUI events.

Parameters:

`event` - the event to dispatch.
`eventGeneratorsHandler` - the handler to dispatch to.

Returns:

`true` if the event is consumed, `false` otherwise.

---

dispatchButton

```
public static boolean dispatchButton(int event,
                                     ButtonEventHandler buttonEventHandler)
```

Dispatch button events.

Parameters:

`event` - the event to dispatch.
`buttonEventHandler` - the handler to dispatch to.

Returns:

`true` if the event is consumed, `false` otherwise.

---

dispatchPointer

```
public static boolean dispatchPointer(int event,
                                       PointerEventHandler pointerEventHandler)
```

Dispatch pointer events.

Parameters:

`event` - the event to dispatch.
`pointerEventHandler` - the handler to dispatch to.

Returns:

`true` if the event is consumed, `false` otherwise.

Interface EventGeneratorsHandler

ej.microui.event.controller

All Known Implementing Classes:

DispatchEventHandler

```
public interface EventGeneratorsHandler
```

Event handler that manages MicroUI events type.

Since:

2.0

See Also:

`Event`

Method Summary		Page
boolean	**handleButton**(int event) Handles button events.	183
boolean	**handleCommand**(int event) Handles command events.	183
boolean	**handleKeyboard**(int event) Handles keyboard events.	184
boolean	**handleKeypad**(int event) Handles keypad events.	184
boolean	**handlePointer**(int event) Handles pointer events.	184

Method Detail

handleButton

```
boolean handleButton(int event)
```

Handles button events.

Parameters:

`event` - the button event to handle.

Returns:

`true` if the event is consumed, `false` otherwise.

---

handleCommand

```
boolean handleCommand(int event)
```

Handles command events.

Parameters:

`event` - the command event to handle.

Returns:

`true` if the event is consumed, `false` otherwise.

---

handleKeyboard

`boolean` **`handleKeyboard`**`(int event)`

Handles keyboard events.

Parameters:
`event` - the keyboard event to handle.
Returns:
`true` if the event is consumed, `false` otherwise.

handleKeypad

`boolean` **`handleKeypad`**`(int event)`

Handles keypad events.

Parameters:
`event` - the keypad event to handle.
Returns:
`true` if the event is consumed, `false` otherwise.

handlePointer

`boolean` **`handlePointer`**`(int event)`

Handles pointer events.

Parameters:
`event` - the pointer event to handle.
Returns:
`true` if the event is consumed, `false` otherwise.

Interface PointerEventHandler

ej.microui.event.controller

All Known Implementing Classes:

DispatchEventHandler

`public interface` **`PointerEventHandler`**

Event handler that manages `Pointer` events.

Since:

2.0

Method Summary		Page
`boolean`	**onPointerClicked**(Pointer pointer, int pointerX, int pointerY, int event) Handles pointer clicked event.	187
`boolean`	**onPointerDoubleClicked**(Pointer pointer, int pointerX, int pointerY, int event) Handles pointer double-clicked event.	187
`boolean`	**onPointerDragged**(Pointer pointer, int pointerX, int pointerY, int event) Handles pointer dragged event.	186
`boolean`	**onPointerEntered**(Pointer pointer, int pointerX, int pointerY, int event) Handles pointer entered event.	186
`boolean`	**onPointerExited**(Pointer pointer, int pointerX, int pointerY, int event) Handles pointer exited event.	187
`boolean`	**onPointerMoved**(Pointer pointer, int pointerX, int pointerY, int event) Handles pointer moved event.	186
`boolean`	**onPointerPressed**(Pointer pointer, int pointerX, int pointerY, int event) Handles pointer pressed event.	185
`boolean`	**onPointerReleased**(Pointer pointer, int pointerX, int pointerY, int event) Handles pointer released event.	186

Method Detail

onPointerPressed

```
boolean onPointerPressed(Pointer pointer,
                         int pointerX,
                         int pointerY,
                         int event)
```

Handles pointer pressed event.

Parameters:

`pointer` - the pointer that generates the event.
`pointerX` - the current x coordinate of the pointer.
`pointerY` - the current y coordinate of the pointer.
`event` - the pointer event to handle.

Returns:

`true` if the event is consumed, `false` otherwise.

onPointerReleased

```
boolean onPointerReleased(Pointer pointer,
                          int pointerX,
                          int pointerY,
                          int event)
```

Handles pointer released event.

Parameters:

`pointer` - the pointer that generates the event.
`pointerX` - the current x coordinate of the pointer.
`pointerY` - the current y coordinate of the pointer.
`event` - the pointer event to handle.

Returns:

`true` if the event is consumed, `false` otherwise.

onPointerMoved

```
boolean onPointerMoved(Pointer pointer,
                       int pointerX,
                       int pointerY,
                       int event)
```

Handles pointer moved event.

Parameters:

`pointer` - the pointer that generates the event.
`pointerX` - the current x coordinate of the pointer.
`pointerY` - the current y coordinate of the pointer.
`event` - the pointer event to handle.

Returns:

`true` if the event is consumed, `false` otherwise.

onPointerDragged

```
boolean onPointerDragged(Pointer pointer,
                         int pointerX,
                         int pointerY,
                         int event)
```

Handles pointer dragged event.

Parameters:

`pointer` - the pointer that generates the event.
`pointerX` - the current x coordinate of the pointer.
`pointerY` - the current y coordinate of the pointer.
`event` - the pointer event to handle.

Returns:

`true` if the event is consumed, `false` otherwise.

onPointerEntered

```
boolean onPointerEntered(Pointer pointer,
                         int pointerX,
                         int pointerY,
                         int event)
```

Handles pointer entered event.

Parameters:

`pointer` - the pointer that generates the event.
`pointerX` - the current x coordinate of the pointer.
`pointerY` - the current y coordinate of the pointer.
`event` - the pointer event to handle.

Returns:

`true` if the event is consumed, `false` otherwise.

---

onPointerExited

```
boolean onPointerExited(Pointer pointer,
                        int pointerX,
                        int pointerY,
                        int event)
```

Handles pointer exited event.

Parameters:

`pointer` - the pointer that generates the event.
`pointerX` - the current x coordinate of the pointer.
`pointerY` - the current y coordinate of the pointer.
`event` - the pointer event to handle.

Returns:

`true` if the event is consumed, `false` otherwise.

---

onPointerClicked

```
boolean onPointerClicked(Pointer pointer,
                         int pointerX,
                         int pointerY,
                         int event)
```

Handles pointer clicked event.

Parameters:

`pointer` - the pointer that generates the event.
`pointerX` - the current x coordinate of the pointer.
`pointerY` - the current y coordinate of the pointer.
`event` - the pointer event to handle.

Returns:

`true` if the event is consumed, `false` otherwise.

---

onPointerDoubleClicked

```
boolean onPointerDoubleClicked(Pointer pointer,
                               int pointerX,
                               int pointerY,
                               int event)
```

Handles pointer double-clicked event.

Parameters:

`pointer` - the pointer that generates the event.
`pointerX` - the current x coordinate of the pointer.
`pointerY` - the current y coordinate of the pointer.
`event` - the pointer event to handle.

Returns:

`true` if the event is consumed, `false` otherwise.

Package ej.microui.event.generator

Contains standard event generators.

See:

Description

Class Summary		Page
Buttons	A Buttons event generator is usually associated to a group of physical buttons and allow to generate events relating to them.	189
Command	Command is an event generator that generates application-level events.	196
GenericEventGenerator	Generic event generator.	202
Keyboard	A Keyboard event generator allows key combinations to generate a key code.	204
Keypad	Keypad is a Keyboard that defines an event generator for 12-key keypads.	208
Pointer	A pointer event generator represents a pointing device that is usually associated to a group of physical buttons.	214
States	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.	222

Package ej.microui.event.generator Description

Contains standard event generators.

Class Buttons

ej.microui.event.generator

```
java.lang.Object
  └ej.microui.event.EventGenerator
      └ej.microui.event.generator.Buttons
```

Direct Known Subclasses:

Pointer

```
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.	191
static int	**DOUBLE_CLICKED** The "double clicked" action.	191
static int	**LONG** The "long" action (button pressed for a "long" time).	191
static int	**PRESSED** The "pressed" action.	191
static int	**RELEASED** The "released" action.	191
static int	**REPEATED** The "repeated" action (button held down).	191

Constructor Summary	Page
Buttons() Creates a buttons event generator that does not support click, doubleClick nor elapsedTime for any of its buttons.	192
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).	191

Method Summary		Page
boolean	**clickEnabled**(int buttonID) Returns true if the generator should send a click event.	193
boolean	**doubleClickEnabled**(int buttonID) Returns true if the generator should send a double click event.	193
long	**elapsedTime**(int buttonID) Returns the elapsed time in milliseconds between the two previous PRESSED events that occurred on the specified button.	195
void	**enableClick**(boolean enable, int buttonID) For the given button, specify whether the generator should send a click event for each pressed event.	192
void	**enableDoubleClick**(boolean enable, int click, int buttonID) For the given button, specify whether the generator should send a double click event.	192
static int	**getAction**(int event) Returns the button's action held by the button event.	195
static int	**getButtonID**(int event) Returns the button's id held by the button event.	194
int	**getEventType**() Returns the MicroUI event type for this button event generator.	193
static boolean	**isClicked**(int event) Tells if an button event is a click event.	194
static boolean	**isDoubleClicked**(int event) Tells if an button event is a double click event.	194
static boolean	**isLong**(int event) Tells if an button event is a long event.	194
static boolean	**isPressed**(int event) Tells if a button event is a press event.	193
static boolean	**isReleased**(int event) Tells if a button event is a release event.	193
static boolean	**isRepeated**(int event) Tells if a button event is a repeat event.	194
void	**send**(int action, int buttonID) Sends a MicroUI event for the given action on given button to the listener of the Buttons.	195
boolean	**supportsExtendedFeatures**(int buttonID) Returns true if the button supports the extended features elapsedTime, click and doubleClick features.	192

Methods inherited from class ej.microui.event.EventGenerator
addToSystemPool, get, get, get, getEventHandler, getID, getList, removeFromSystemPool, sendEvent, setEventHandler

Field Detail

PRESSED

```
public static final int PRESSED
```

The "pressed" action.

~~RELEASED~~

```
public static final int RELEASED
```

The "released" action.

~~LONG~~

```
public static final int LONG
```

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

~~REPEATED~~

```
public static final int REPEATED
```

The "repeated" action (button held down).

~~CLICKED~~

```
public static final int CLICKED
```

The "clicked" action.

~~DOUBLE_CLICKED~~

```
public static final int DOUBLE_CLICKED
```

The "double clicked" action.

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 event generator. 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.

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.

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 Command

ej.microui.event.generator

```
java.lang.Object
  └ej.microui.event.EventGenerator
      └ej.microui.event.generator.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.	200
static int	**BACK** The "back" command constant.	198
static int	**CANCEL** The "cancel" command constant.	199
static int	**CLOCKWISE** The "clockwise" command constant.	200
static int	**COPY** The "copy" command constant.	199
static int	**CUT** The "cut" command constant.	200
static int	**DISPLAY** The "display" command constant.	200
static int	**DOWN** The "down" command constant.	198
static int	**ESC** The "escape" command constant.	198

static int	**EXIT** The "exit" command constant.	199
static int	**HELP** The "help" command constant.	199
static int	**LEFT** The "left" command constant.	198
static int	**MENU** The "menu" command constant.	199
static int	**NEXT** The "next" command constant.	200
static int	**PASTE** The "paste" command constant.	200
static int	**PAUSE** The "pause" command constant.	199
static int	**PREVIOUS** The "previous" command constant.	200
static int	**RESUME** The "resume" command constant.	199
static int	**RIGHT** The "right" command constant.	198
static int	**SELECT** The "select" command constant.	198
static int	**START** The "start" command constant.	199
static int	**STOP** The "stop" command constant.	199
static int	**UP** The "up" command constant.	198

Constructor Summary	Page
Command() Creates a new command event generator.	200

Method Summary	Page

Methods inherited from class ej.microui.event.EventGenerator
addToSystemPool, get, get, get, getEventHandler, getID, getList, removeFromSystemPool, sendEvent, setEventHandler

Field Detail

ESC

```
public static final int ESC
```

The "escape" command constant.

~~BACK~~

```
public static final int BACK
```

The "back" command constant.

~~UP~~

```
public static final int UP
```

The "up" command constant.

~~DOWN~~

```
public static final int DOWN
```

The "down" command constant.

~~LEFT~~

```
public static final int LEFT
```

The "left" command constant.

~~RIGHT~~

```
public static final int RIGHT
```

The "right" command constant.

~~SELECT~~

```
public static final int SELECT
```

The "select" command constant.

CANCEL

```
public static final int CANCEL
```

The "cancel" command constant.

HELP

```
public static final int HELP
```

The "help" command constant.

MENU

```
public static final int MENU
```

The "menu" command constant.

EXIT

```
public static final int EXIT
```

The "exit" command constant.

START

```
public static final int START
```

The "start" command constant.

STOP

```
public static final int STOP
```

The "stop" command constant.

PAUSE

```
public static final int PAUSE
```

The "pause" command constant.

RESUME

```
public static final int RESUME
```

The "resume" command constant.

COPY

```
public static final int COPY
```

The "copy" command constant.

CUT

```
public static final int CUT
```

The "cut" command constant.

PASTE

```
public static final int PASTE
```

The "paste" command constant.

CLOCKWISE

```
public static final int CLOCKWISE
```

The "clockwise" command constant.

ANTICLOCKWISE

```
public static final int ANTICLOCKWISE
```

The "anti-clockwise" command constant.

PREVIOUS

```
public static final int PREVIOUS
```

The "previous" command constant.

NEXT

```
public static final int NEXT
```

The "next" command constant.

DISPLAY

```
public static final int DISPLAY
```

The "display" command constant.

Constructor Detail

Command

```
public Command()
```

Creates a new command event generator.

Method Detail

getEventType

```
public int getEventType()
```

Gets the 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 GenericEventGenerator

ej.microui.event.generator

```
java.lang.Object
  └ej.microui.event.EventGenerator
      └ej.microui.event.generator.GenericEventGenerator
```

```
abstract public class GenericEventGenerator
extends EventGenerator
```

Generic event generator.

Generic communication to receive events.

Since:
2.0

Constructor Summary	Page
GenericEventGenerator()	202

Method Summary		Page
protected abstract void	eventReceived(int event) Called by MicroUI framework when a custom event has been received from native world.	203
protected abstract void	eventsReceived(int[] events) Called by MicroUI framework when a custom event has been received from native world.	203
abstract void	setProperty(String name, String value) Called at startup to configure the event generator with the specific properties set in the .microui file.	202

Methods inherited from class ej.microui.event.EventGenerator
addToSystemPool, get, get, get, getEventHandler, getEventType, getID, getList, removeFromSystemPool, sendEvent, setEventHandler

Constructor Detail

GenericEventGenerator

```
public GenericEventGenerator()
```

Method Detail

setProperty

```
public abstract void setProperty(String name,
                                 String value)
```

Called at startup to configure the event generator with the specific properties set in the .microui file.

Parameters:

`name` - the property name
`value` - the property value

---

eventReceived

```
protected abstract void eventReceived(int event)
```

Called by MicroUI framework when a custom event has been received from native world. The current custom event contains only one 32-bit value.

Parameters:

`event` - the 32-bit custom event value.

---

eventsReceived

```
protected abstract void eventsReceived(int[] events)
```

Called by MicroUI framework when a custom event has been received from native world. The current custom event contains several 32-bit values.

Parameters:

`events` - the 32-bit custom event values.

Class Keyboard

ej.microui.event.generator

```
java.lang.Object
  └ej.microui.event.EventGenerator
      └ej.microui.event.generator.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.	205
static int	**KEY_UP** The KEY_UP event action.	205
static int	**TEXT_INPUT** The TEXT_INPUT event action.	205

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).	206

Method Summary	Page

---

boolean	**dropOnFull**() Subclasses should override this method to specify their policy.	206
int	**getAction**(int event) Returns the keyboard action held by the keyboard event.	206
int	**getEventType**() Gets the event type associated with the event generator	206
char	**getNextChar**(int event) Gets the next character associated with the specified event.	207
void	**onlyTextInput**(boolean onlyText) Specifies whether the KEY_UP and KEY_DOWN events should be generated.	206
void	**reset**() Reset the keyboard by flushing all pending characters.	207
void	**send**(int type, char c) Send an keyboard event to the MicroUI application.	207

Methods inherited from class ej.microui.event.EventGenerator
addToSystemPool, get, get, get, getEventHandler, getID, getList, removeFromSystemPool, sendEvent, setEventHandler

Field Detail

TEXT_INPUT

```
public static final int TEXT_INPUT
```

The TEXT_INPUT event action.

See Also:
getAction(int)

KEY_DOWN

```
public static final int KEY_DOWN
```

The KEY_DOWN event action.

See Also:
getAction(int)

KEY_UP

```
public static final int KEY_UP
```

The KEY_UP event action.

See Also:
getAction(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.

---

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.

---

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:
`getNextChar(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`.

Parameters:
`type` - a type between `KEY_UP`, `KEY_DOWN` and `TEXT_INPUT`.
`c` - the character to send.

Class Keypad

ej.microui.event.generator

```
java.lang.Object
  └ej.microui.event.EventGenerator
      └ej.microui.event.generator.Keyboard
          └ej.microui.event.generator.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 `Keyboard.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:

- `Keyboard.KEY_DOWN` '2'
- `Keyboard.TEXT_INPUT` 'a'
- `Keyboard.KEY_UP` '2'
- `Keyboard.KEY_DOWN` '2'
- `Keyboard.TEXT_INPUT` 'b'
- `Keyboard.KEY_UP` '2'
- `KEY_VALIDATED` (after a delay, see `setDelay(int)`)

See Also:
`Keyboard`

Field Summary		Page
`static int`	**ALPHA** The ALPHA mode. In ALPHA mode the keypad may return several letters and digits according to the number of consequent key press.	210

static int	**CAP** The CAP mode. In CAP mode only capital letters and digits are returned	210
static int	**CAP1** The CAP1 mode. In CAP1 mode is the same has CAP mode, but automatically switch to ALPHA mode after the Keyboard.KEY_UP event	211
static 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.	210
static int	**NUM** The NUM mode. In NUM mode the keypad may return several digits according to the number of consequent key press.	210

Fields inherited from class ej.microui.event.generator.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 Keyboard.KEY_UP, Keyboard.KEY_DOWN and Keyboard.TEXT_INPUT event).	211

	Method Summary	Page
abstract char[]	**getAssignment**(char key) Gets the complete array of chars associated with the specified key.	213
int	**getDelay**() Gets the delay.	212
int	**getEventType**() Gets the event type associated with the event generator	211
abstract String	**getLanguage**() Gets the currently selected language.	212
int	**getMode**() Gets the mode.	212
abstract String[]	**getSupportedLanguages**() Gets the supported languages for this keypad.	212
void	**setDelay**(int delay) Sets the delay The delay is the value between two press that allows to select a letter out of several.	211
abstract void	**setLanguage**(String language) Sets the language.	212
void	**setMode**(int mode) Sets the mode.	212

Methods inherited from class ej.microui.event.generator.Keyboard
dropOnFull, getAction, getNextChar, onlyTextInput, reset, send

Methods inherited from class ej.microui.event.EventGenerator
addToSystemPool, get, get, get, getEventHandler, getID, getList, removeFromSystemPool, sendEvent, setEventHandler

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.

See Also:
`Keyboard.getAction(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.

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.

See Also:
`setMode(int)`

CAP

```
public static final int CAP
```

The CAP mode.
In CAP mode only capital letters and digits are returned

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 `Keyboard.KEY_UP` event

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 `Keyboard.KEY_UP`, `Keyboard.KEY_DOWN` and `Keyboard.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)`

---

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 Pointer

ej.microui.event.generator

```
java.lang.Object
  └ej.microui.event.EventGenerator
      └ej.microui.event.generator.Buttons
          └ej.microui.event.generator.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.	216
static int	**ENTERED** The "entered" action.	216
static int	**EXITED** The "exited" action.	216
static int	**MOVED** The "move" action.	216

Fields inherited from class ej.microui.event.generator.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.	217
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).	217

Method Summary		Page
int	**getAbsoluteHeight**()	217
int	**getAbsoluteWidth**()	217
int	**getAbsoluteX**() Returns the last available absolute x coordinate in pointer area	218
int	**getAbsoluteY**() Returns the last available absolute y coordinate in pointer area	218
int	**getEventType**() Gets the event type associated with the event generator.	217
FlyingImage	**getFlyingImage**() Gets the FlyingImage associated to this Pointer.	219
int	**getHeight**()	220
int	**getWidth**()	220
int	**getX**() Returns the last available x coordinate in scaled area (after applying any origin offset and the scale factor).	218
int	**getY**() Returns the last available y coordinate in scaled area (after applying any origin offset and the scale factor).	218
static boolean	**isDragged**(int event) Tells if a pointer event is a drag event.	221
static boolean	**isEntered**(int event) Tells if a pointer event is an enter event.	221
static boolean	**isExited**(int event) Tells if a pointer event is an exit event.	221
static boolean	**isMoved**(int event) Tells if a pointer event is a move event.	221
void	**move**(int x, int y) Stores the given position and sends a MicroUI Event to the Pointer's listener.	220
void	**reset**(int x, int y) Stores the given position.	220
void	**send**(int action, int buttonID) Sends a MicroUI event for the given action on given button to the listener of the Pointer.	220
void	**setFlyingImage**(FlyingImage flyingImage) A Pointer generator manages positioning an image automatically if such image is set.	219

void	**setFlyingImage**(FlyingImage flyingImage, int anchorX, int anchorY) A Pointer generator manages positioning an image automatically if such image is set.	219
void	**setOrigin**(int x0, int y0) Sets an origin offset.	219
void	**setScale**(Display display) Sets a display area for scaled area. same as `setScale(display.getWidth(), display.getHeight())`	218
void	**setScale**(int areaWidth, int areaHeight) Sets a scaled area.	219

Methods inherited from class ej.microui.event.generator.Buttons
clickEnabled, doubleClickEnabled, elapsedTime, enableClick, enableDoubleClick, getAction, getButtonID, isClicked, isDoubleClicked, isLong, isPressed, isReleased, isRepeated, supportsExtendedFeatures

Methods inherited from class ej.microui.event.EventGenerator
addToSystemPool, get, get, get, getEventHandler, getID, getList, removeFromSystemPool, sendEvent, setEventHandler

Field Detail

MOVED

```
public static final int MOVED
```

The "move" action.

~~DRAGGED~~

```
public static final int DRAGGED
```

The "dragged" action.

~~ENTERED~~

```
public static final int ENTERED
```

The "entered" action.

~~EXITED~~

```
public static final int EXITED
```

The "exited" action.

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:

`nbButtons` - the number of buttons that support the extended features
`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())`

Parameters:
`display` - the display size to take as reference
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.

Parameters:

`areaWidth` - the area width
`areaHeight` - the area height

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.

Parameters:

`x0` - the X coordinate of the new origin
`y0` - the Y coordinate of the new origin

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)`

Parameters:

`flyingImage` - the flying image to update when the pointer changed or `null`

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.

Parameters:

`flyingImage` - the flying image to update when the pointer changed or `null`
`anchorX` - the image anchor X coordinate
`anchorY` - the image anchor Y coordinate

---

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 States

ej.microui.event.generator

```
java.lang.Object
  └ej.microui.event.EventGenerator
      └ej.microui.event.generator.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 allows 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 `getNumberOfStates()-1`

Constructor Summary	Page
States`(int[] nbValues, int[] initialValues)` Creates a states generator.	222

Method Summary		Page
int	**getCurrentValue**`(int stateID)` Gets the current value of the given state.	224
int	**getEventType**`()` Gets the event type associated with the event generator.	223
int	**getNumberOfStates**`()` Gets the number of states managed by this instance.	224
int	**getNumberOfValues**`(int stateID)` Gets the total number of values for the given state.	223
static int	**getStateID**`(int event)` Gets the state's ID held by the state event.	223
static int	**getStateValue**`(int event)` Gets the state's value held by the state event.	223
void	**send**`(int stateID, int value)` Stores the given state value and sends a MicroUI `Event.STATE` to the States's listener.	224

Methods inherited from class ej.microui.event.EventGenerator
addToSystemPool, get, get, get, getEventHandler, getID, getList, removeFromSystemPool, sendEvent, setEventHandler

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

getStateID

```
public static int getStateID(int event)
```

Gets the state's ID held by the state event.

Parameters:
`event` - the state event to decode.
Returns:
id between `0` and `255`.

---

getStateValue

```
public static int getStateValue(int event)
```

Gets the state's value held by the state event.

Parameters:
`event` - the state event to decode.
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.

---

getNumberOfValues

```
public int getNumberOfValues(int stateID)
```

Gets the total number of values for the given state.

Parameters:
`stateID` - the state identifier value.

Returns:
the total number of values for the given state.
Throws:
`IndexOutOfBoundsException` - when stateID is out of `[0,getNumberOfStates()-1]`.

---

getCurrentValue

```
public int getCurrentValue(int stateID)
```

Gets the current value of the given state.

Parameters:
`stateID` - the state identifier value.
Returns:
a number between `0` and `getNumberOfValues(int)-1`.
Throws:
`IndexOutOfBoundsException` - when stateID is out of `[0,getNumberOfStates()-1]`.

---

getNumberOfStates

```
public int getNumberOfStates()
```

Gets the number of states managed by this instance.

Returns:
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.

Parameters:
`stateID` - the state identifier value.
`value` - the new state value
Throws:
`IndexOutOfBoundsException` - when `stateID` is out of `[0,getNumberOfStates()-1]`., when `value` is out of `[0,getNumberOfValues(int)-1]`.

Package ej.microui.led

Contains LEDs management.

See:
Description

Class Summary		Page
Leds	This class is used to manage all LEDs available on the platform.	226

Package ej.microui.led Description

Contains LEDs management.

Since:
2.0

Class Leds

ej.microui.led

```
java.lang.Object
  └ ej.microui.led.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.	226
static int	**MIN_INTENSITY** Intensity value to turn off a LED.	226

Method Summary		Page
static int	**getLedIntensity**(int ledId) Gets the intensity of the specified LED.	227
static int	**getNumberOfLeds**() Returns the available number of LEDs.	227
static void	**setLedIntensity**(int ledId, int intensity) Controls the intensity of the specified LED.	227
static void	**setLedOff**(int ledId) Turns off the given LED.	227
static void	**setLedOn**(int ledId) Turns on the given LED.	227

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

---

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

---

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

Package ej.microui.util

Contains MicroUI utilities.

See:
Description

Interface Summary		Page
EventHandler	An event handler is intended to receive and handle events.	229

Package ej.microui.util Description

Contains MicroUI utilities.

Since:
2.0

Interface EventHandler

ej.microui.util

All Known Implementing Classes:

DispatchEventHandler

```
public interface EventHandler
```

An event handler is intended to receive and handle events.

In the MVC pattern it is the controller.

Since:

2.0

Method Summary		Page
boolean	**handleEvent**(int event) Handles an event.	229

Method Detail

handleEvent

```
boolean handleEvent(int event)
```

Handles an event.

Parameters:

`event` - the event to handle.

Returns:

`true` if the event is consumed, `false` otherwise.

See Also:

`Event`, `DispatchHelper`