ESR Consortium
B-ON-1.2

Beyond
Profile Specification

ESR001

Reference: ESR-SPE-001-B-ON
Version: 1.2
Rev: E

Contents

Tables

Illustrations

1 PREFACE TO B-ON PROFILE, ESR001

This document defines the B-ON profile, targeting Java virtual machines.

1.1 Who should use this specification?

This specification is targeted at the following audiences:

- Implementors of the B-ON specification.
- Application developers that target embedded Java applications for resource-constrained devices.
- Java virtual machine providers.
- Hard Real Time Java application developers.

1.2 Comments

Your comments about B-ON are welcome. Please send them by email to `comments@e-s-r.net`, with B-ON as subject.

1.3 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.

Although this specification defines minimal requirements, devices with more resources may also benefit from B-ON specification, especially when users are concerned with optimal resource usage.

The B-ON specification makes no hardware requirement for devices that run a Java virtual machine that implements this specification. Typical hardware for B-ON ranges from 8-bit to 64-bit multi-core cpu.

The B-ON profile specification makes minimal assumptions about the system software of the device. Although a Java virtual machine is required, the kernel does not need to support an OS/RTOS while the virtual machine may be baremetal (i.e. the device boots directly in Java).

Compliant B-ON 1.2 implementations MUST include all packages, classes, and interfaces described in this specification, and implement the associated behavior.

1.4 Related Literature

JVM2: Tim Lindholm & Frank Yellin, The Java™ Virtual Machine Specification, Second Edition, 1999
JLS: James Gosling, Guy Steele, Bill Joy, Gilad Bracha, The Java™ Language Specification, Third Edition, 2005

1.5 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.

1.6 Implementation Notes

The B-ON specification does not include any implementation details. B-ON implementors are free to use whatever techniques they deem appropriate to implement the specification, with (or without) collaboration of any Java virtual machine provider. B-ON experts have taken great care not to mention any special Java 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 and simple architecture to enable an open, third-party, application development environment for controlling both its startup sequence and its memory resource in the best possible way.

Although this specification spans a potentially wide set of devices, it focus on devices that have non-volatile memories and volatile ones (eeprom, flash, ram, …). At the application level, it focuses on applications that have some sort of initialization phase before entering into a mission phase that then exists forever until the device gets shutdown or reboots.

2.1 Why B-ON?

Many languages let software engineers define the memory management of their applications. One reason is that most embedded devices have scarce physical memory, while being cost driven. On the other hand, it is well known that memory allocation is one of the most difficult tasks to achieve efficiently as soon as the application uses more than a few objects.

In order to cope with these two contradictory issues, there are two main approaches, each one at the extremity of the possibility spectrum:

- Pre-allocate all of what is needed for the program to run, either statically (at compile time) or dynamically once and for all at system startup. While running, no extra allocation is done. This approach is often used for Hard Real-Time systems when the memory consumption cannot be defined at compile-time through formal analysis.
- Let the runtime system manage the memory, fully freeing the engineers from that task. This is done through the use of garbage collectors. A huge number of different garbage collection policies are available and each have their own benefits and drawbacks.

The Java virtual machine specification [JVM2] defines a heap where Java objects reside. This heap is automatically managed by a garbage collector. Every Java virtual machine is free to implement the memory management that best fits the application domain it is designed for.

[JVM2] defines a semantically immortal set of objects: a pool of `java.lang.String`, which are references in classfile constant pools[1]. [JVM2] also defines the way applications get initialized, even though it is quite a loose process where lazy initialization is permitted. Intuitively, classes must be initialized before any instance creation or access to its static variables (see 4.1.2).

One of the newer trends in software involves designing simple solutions that are easy to understand and most importantly easy to manipulate and control. Developers must be able to minimize development time, often dealing with small memory budgets for their application. On typical microcontrollers, the ratio between read-only memory and volatile memory is four, meaning that there is 4 times more read-only memory (eeprom, flash, …) than volatile memory (ram). For really cost sensitive devices, the ratio may drop to 8 (ram is what costs the most), while for rather more expensive ones it may reach 2.

1 Literal strings are turned into CONSTANT_String_info by Java compilers.

B-ON defines a suitable and flexible way to fully control both memory usage and start-up sequences on devices with limited memory resources. It does so within the boundaries of the Java semantic. More precisely, it allows:

- Controlling the initialization sequence in a deterministic way.
- Defining persistent immutable read-only objects (that may be placed into non-volatile memory areas), and do not require copies to be made in ram to be manipulated.
- Defining immortal read-write objects that are always alive.

B-ON serves as a very robust foundation for implementing Java software, in particular embedded Java Software.

B-ON also add a small set of useful utilities:

- A Timer facilities that allows to schedule small activities repeatedly (or not). Such activities are Runnable objects that are automatically scheduled by the timer.
- Platform time which cannot be changed: the time from the very last boot of the device.
- Read-write `ByteArray` support according to the underlying processor endianness.
- A set of useful math operators.

2.2 Basic Concepts

B-ON experts agreed to limit the set of APIs specified to only those required to achieve a high level of portability and successful deployments. Their main concern was to stay within the boundaries of the Java semantics [JLS].

B-ON defines two phases for the execution stream:

- The initialization phase: the initialization sequence executes all the static initializer methods (known as the `<clinit>` methods).
- The mission phase: the `main(String[])` method of the main class is called. The application runs until the device is switched off.

There is a kind of objects, named immutable objects, that are alive at system startup. They are read-only objects that most probably reside in non-volatile memory. All together they form a pre-existing world that exists on its own, just like the hardware does.

2.3 First Example

The simple next example illustrates the use of big buffers. They are made immortal in order to recycle them manually while they represent the most critical ram consumption. This example also makes use of an immutable object, an array of values that never changes during the lifetime of the device.

```
package example;
public class Filter {
    public static final int BufferSize = 4096;// 16k (an int is 32-bit)
    public static int[][] Buffers;
    public static int[] ValidValues;
    static {
        ValidValues =(int[])ej.bon.Immutables.get("filter");
    }

    static {
        // Allocate the immortal pool of buffers. Only two Filters may
        // be alive at the same time: ==> 32k of ram for two Filters
        Buffers = new int[2][];
        Buffers[0] = (int[])Immortals.setImmortal(new int[BufferSize]);
        Buffers[1] = (int[])Immortals.setImmortal(new int[BufferSize]);
    }

    public int ptr;
    public int[] buffer;

    public Filter(){
        // grab a buffer from the pool, or throw an exception
        synchronized (Buffers){
            for (int i = Buffers.length; --i >= 0; ){
                if (Buffers[i] != null ){
                    buffer = Buffers[i];
                    ptr=-1;
                    Buffers[i] = null ;
                    break ;
                }
            }
        }
        if (buffer == null ){
            throw new OutOfMemoryError();
        }
    }

    public void close(){
        // recycle manually the immortal buffer: store it in
        // global Buffers array pool
        synchronized (Buffers){
            for (int i = Buffers.length; --i >= 0; ){
                if (Buffers[i] == null ){
                    Buffers[i] = buffer;
                    break;
                }
            }
        }
    }

    public synchronized void insert( int value){
        // only insert permitted values

        if (ptr >= BufferSize) return; // full

        for (int j = ValidValues.length; --j >= 0; ){
            if (value == ValidValues[j]){
                buffer[++ptr] = value;
            }
        }
    }
}
```

3 OBJECT NATURES

The B-ON specification defines three natures for objects: persistent immutable objects (3.1), immortal objects (3.2), and reclaimable objects (3.3). Immutable[2] objects are also referred to as read-only objects, whereas reclaimable objects are regular objects.

2 Persistent immutable objects are named immutable objects throughout the specification.

Although objects get a liveness nature, this is fully transparent at the Java semantic level. A semantically correct software assuming B-ON will behave exactly the same on a Java virtual machine that does not implement the three B-ON object natures[3].

3.1 Persistent Immutable Objects

Immutable objects are read-only objects. They are instances of any concrete class. Although they are immutable, they obey all the Java object's semantics. In particular, they hold a hash code, have a class and have a monitor that a thread may enter into.

There is no way for an immutable object to directly refer to a non-immutable object. References from immutable objects always refer to other immutable objects. Writing into an immutable object (field write access) results in an unspecified behavior. The B-ON experts group strongly encourages implementations of the B-ON specification to raise an uncatchable exception when there is an attempt to write into an immutable object, although a no-op operation may be sufficient[4].

Immutable objects may be created in two ways:

- At run-time, if the implementation allows this, as described in section 3.1.5,
- At system/application configuration time by specifying objects in an XML configuration file, as described in the sections immediately below.

In the second case the way the immutable object's descriptions are given to the Java virtual machine at startup is implementation-dependent. Most implementations will assume the immutable objects to be at some particular location in (read-only) memory: the implementation-dependent way to give the immutable objects at startup would therefore be trivial (nothing to do, but know the memory address). Note that immutable objects do not need to be copied in (scarce) ram memory to be manipulated[5].

Software is made up of several parts, often called libraries, that may come with their own immutable object descriptions. Therefore more than one immutable description may be given to the Java virtual machine.

3.1.1 Object ID and Immutable Object Querying

Immutable objects are semantically organized into one global pool, just like the Java interned `java.lang.String` objects.

An immutable object may be attached to a `java.lang.String` key, known as its ID. This ID allows an immutable object to be retrieved out of the global pool of immutable objects, thanks to the method `Immutables.get(String)`. The ID of an object is globally unique: the ID `"ANONYMOUS"` is a reserved ID (see 3.1.5) and cannot be used to qualify an immutable object.

3.1.2 Immutable Objects Descriptions and Creation

Descriptions are based on the structure of objects, that is, they embed structural information such as fully qualified class names and field names. Fields[6] that need to get initialized with some value (base-type or another immutable object) are described using a pair: field-name, value.

3 Writing into an immutable object is considered as a semantic error.
4 The write access to read-only memory is often a no-op operation that has no cost.
5 A specification with persistent storage that would force to copies the data/objects in ram would be impractical for small devices.
6 Only instance fields of objects are involved, i.e. not static fields.

Fields that are not described get initialized with the default Java value (`0` for numeric types, `null` for objects, `false` for booleans, `0.0` for floating-point numbers [JLS]). No visibility rule applies, that is, any kind of field may be listed, even private ones. Final fields must be initialized.

There is no particular order for the creation of the immutable objects. The B-ON experts recommend the use of tools for the creation of large graphs of immutable objects.

3.1.3 XML Grammar

Immutable objects are described according to the following XML syntax (Annex 6 gives the DTD).

- `<immutables>`: the root element of one immutable objects description.
- attributes:
    - `name`: an optional attribute that defines the content of the XML description.
- child elements: `<object>` , `<objectAlias>`, `<array>` , `<string>`, `<class>`, `<null>`, `<importObject>`.
- `<object>`: element that defines a new object.
    - attributes:
        - `id`: the ID string that allows the object to be retrieved through the use of `Immutables.get(String)`
        - `type`: the name of the class of the object. An alias may be used instead of the fully-qualified class name.
        - `private`: a boolean that indicates whether the object will be accessible using the `Immutables.get(String)` method. If false, the objects can only be referenced within the XML immutable objects descriptions.
    - child elements: `<field>` , `<refField>`
- `<objectAlias>`: element that defines a new key for an existing object.
    - attributes:
        - `id`: the ID string that allows the object to be retrieved through the use of `Immutables.get(String).`
        - `object`: the existing object ID or alias ID.
        - `private`: a boolean that indicates whether the object will be accessible using the `Immutables.get(String)` method. If false, the objects can only be referenced within the XML immutable objects descriptions.
    - child elements: none
- `<string>`: element that defines an interned string.
    - attributes:
        - `id`: the ID that allows the object to be retrieved through the use of `Immutables.get(String)`
        - `value`: the string literal
        - `private`: a boolean that indicates whether the object will be accessible using the `Immutables.get(String)` method. If false, the objects can only be referenced within the XML immutable objects descriptions.

  - child elements: none
- `<class>`: element that defines an instance of a `java.lang.Class`. The ID of this object can be used for type attributes.
  - attributes:
    - `id`: the ID that allows the object to be retrieved through the use of `Immutables.get(String)`
    - `value`: the class fully qualified name like `java.lang.Object`.
    - `private`: a boolean that indicates whether the object will be accessible using the `Immutables.get(String)` method. If false, the objects can only be referenced within the XML immutable objects descriptions.
  - child elements: none
- `<field>`: elements that state a field with its associated literal value.
  - attributes:
    - `name`: the name of the field as defined in the class that defines it.
    - `value`: the value of the field. The value is a primitive type (numeric or boolean) or a literal string (see Table 3-1: Immutables Primitive Type Format).
    - `type`: this attribute is optional. It represents the class where the field is defined. A field without its `type` attribute refers to the first field found while scanning the class hierarchy from the bottom to the top (following the superclass link).
  - child elements: none
- `<refField>`: elements that state a field that references an immutable object.
  - attributes:
    - `name`: the name of the field as defined in the class that defines it.
    - `ref`: the ID of the referenced immutable object.
    - `type`: this attribute is optional. It represents the class where the field is defined. A field without its type attribute refers to the first field found while scanning the class hierarchy from the bottom to the top (following the superclass link).
- child elements: none
- `<array>`: element that defines a new array.
  - attributes:
    - `id`: the ID that allows the object to be retrieved through the use of `Immutables.get(String)`
    - `type`: the array type. An alias may be used instead of the fully qualified class name. Dimensions are given using the Java notation `[]`.
    - `length`: this attribute is optional. It represents the number of elements the array has.
    - `private`: a boolean that indicates whether the object will be accessible using the `Immutables.get(String)` method. If false, the objects can only be referenced within the XML immutable objects descriptions.

  - child elements: `<elem>` , `<refElem>`
- `<elem>`: element that defines an array element with its literal value.
  - attributes:
    - `value`: the value of the element. The value is a primitive type (numeric or boolean) or a literal string (see Table 3-1: Immutables Primitive Type Format).
  - child elements: none
- `<refElem>`: element that defines an array element. Such element references an immutable object.
- attributes:
  - `ref`: the ID of the referenced immutable object.
- child elements: none
- `<null>`: element that defines a null object that can be referenced by an object field or an array element
- attributes:
  - `id`: the ID that allows the null object to be retrieved through the use of `Immutables.get(String)`
  - `private`: a boolean that indicates whether the object will be accessible using the `Immutables.get(String)` method. If false, the objects can only be referenced within the XML immutable objects descriptions.
- child elements: none
- `<importObject>`: element that import an object that is defined in another immutable file. The referenced object may be private or public.
  - attributes:
    - `id`: the ID of the imported object
  - child elements: none

Class names use the Java notation (using a `'.'` as separator): `java.lang.Object` is an example.

String literals are defined as in XML specification. To allow quotes in XML string data use the apostrophe `' ' '` separator as XML separator or the escape character `&quot;`.

To define the next 9 characters String, `my"String`, as string literal value, use one of following syntax:

```
<field name="f1" value='my"String' />
<field name="f1" value="my&quot;String" />
```

Next table lists the format for the primitive values:

Primitive Type	Format	Example
boolean	`true` or `false`	`<…value="true"/>`

byte, short, int, long	Format defined in the Java method `Long.decode(String)`	`<…value="123"/>`
		`<…value="0x2A"/>`
		`<…value="-561"/>`

char	`Format defined in the Java method Long.decode(String)` or a character value between simple quotes	`<…value="'123"/>`
		`<…value="'z'"/>`
		`<…value="'&#xA9;'"/>`

float	Format defined in the Java method Float.parseFloat`(String)`	`<…value="2.3"/>`
		`<…value="4.2e12"/>`
		`<…value="-5.671"/>`

double	Format defined in the Java method Double.parseDouble`(String)`	`<…value="2.3"/>`
		`<…value="4.2e12"/>`
		`<…value="-5.671"/>`

Table 3-1: Immutables Primitive Type Format

IDs define one global name space[7]: an ID only refers to only one object. It is an error to have objects sharing ID.

3.1.4 Immutable XML Description Examples

```
<immutables name="MyCorp objects">

  <array id="corp.immut00" type="boolean[]" length="2">
    <elem value="true"/>
    <elem value="false"/>
  </array>

  <array id="corp.immut01" type="int[]">
    <elem value="3"/>
    <elem value="2"/>
    <elem value="1"/>
  </array>

  <class id="MyClass" type="myCompany.mypackage.MyClass"
   private="true"/>

  <object id="corp.immut02" type="MyClass">
    <field name="a" value="50" />
    <field name="str" value="Hello" />
    <refField name="b" ref="corp.null" />
  </object>

  <object id="corp.immut03" type="myCompany.mypackage.A">
    <refField name="f" ref="corp.immut04" />
    <refField name="s" ref="corp.internalKey"/>
    <refField name="o" ref="corp2.immut"/>
  </object>
```

7 As a good practice, it is recommended to define ID using a qualified name, such as "myCorp.myApp.MyID12".

```
  <string id="corp.immut04" value='Hello World!' />

  <string id="corp.internalKey" value="one" private="true" />

  <string id="key1" value="two" />

  <string id="key2" value='thr"ee' />

  <object id="value1" type="java.lang.Object" />

  <null id="corp.null"/>

  <importObject id="corp2.immut"/>

</immutables>
```

3.1.5 Turning Objects Into Immutable Objects

Some systems may define persistent memory where new immutable objects can be stored. Such objects remain “live” through device reboots. The number of available persistent memory is system dependent and is described within the datasheet of the Java virtual machine that implements the B-ON specification. `Immutables.totalMemory()` returns this persistent immutable memory size, whereas `Immutables.freeMemory()` returns the left remaining persistent memory size.

If an object is persistently added with an ID that was already in use by a previously defined immutable object, the new added object takes precedence over the object that was referred to by that ID: the next call to `Immutables.get(ID)` returns the last added object with that ID[8].

Objects are (runtime) added to the persistent memory as a graph, defined by the values held in an hash table.

- The keys of the hash table represent the IDs of the objects: these keys must be of the `java.lang.String`[9] class. Objects that are not in the hash table do not take part in the immutable storage action.
- The special ID `"ANONYMOUS"` allows objects held by an array (from the type `java.lang.Object[]`) to be added into the hash table. They are considered as being part of the hash table, but anonymously. The key `"ANONYMOUS"` is not added to the set of all available IDs[10].

8 This feature is often used to overwrite default system setting.
9 On highly resource constrained devices, it might be important to define short ID.
10 The ANONYMOUS key holds all the private immutable objects.

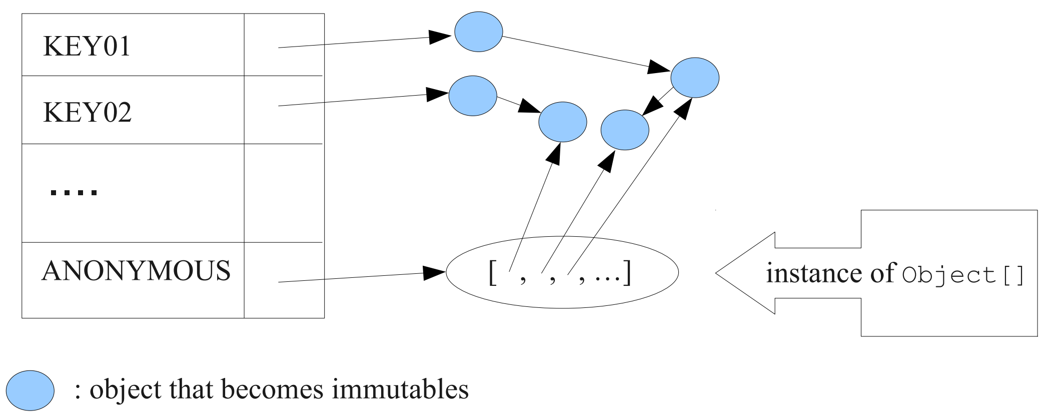

Illustration 3-1: Example of hash table passed to `Immutables.putAll(Hashtable).`

The hash table is added using the `Immutables.putAll(Hashtable)` method. It persistently writes copies of all the values in the hash table that are not already immutable objects. All references within those objects to non-immutable objects or to objects that are outside the hash table are set to `null`. References to immutable objects remain, literal strings[11] are considered as immutable (see 2.1). Keys that refer to `null` are ignored.

The `Immutables.put(String, Object)` method allows to store a single object.

All operations on `Immutables` must be thread safe.

3.2 Immortal Objects

3.2.1 Non Garbageable Objects

Immortal objects are regular objects that are not managed by the Java virtual machine garbage collector. Immortal objects do not move around in memory: they remain physically located in one memory location forever.

3.2.2 Turning Objects Into Immortal Objects

Reclaimable objects may be turned into immortal objects using the `Immortals.setImmortal(Object)` method. Only the object passed as argument is turned into an immortal object, i.e. none of the objects it refers to through its fields become immortal. This is in contrast with `Immortals.deepImmortal(Object)` that turns the object passed as the argument and all objects referred to from the argument into immortal objects. Note that weakly reachable objects are not turned into immortal objects; in other words the `WeakReference` semantic is not affected by this operation.

The total amount of free immortal memory still available is `Immortals.freeMemory()`. It is system dependent.

The system provides the possibility to create objects directly as immortal objects using the method `Immortals.run(Runnable)`: while the `run()` method of the `Runnable` executes, all created objects are allocated as immortal objects.

11 Dynamically created interned String (using `intern()` method) are not considered as immutable objects.

The system may define a property `ej.bon.immortalAfterInit`. If the property exists and if set to `true`, a global memory collection is triggered at the end of the initialization phase, reclaiming all dead objects that were created to get the system initialized. All remaining objects become immortal, and accessible for the mission phase.

3.3 Reclaimable Objects

3.3.1 Death Notification

Most objects are reclaimable objects. Sometimes, they interact with the underlying system using handles. Those handles represent underlying data that needs to be closed/freed/acknowledged/… when the object that holds the handle dies.

The B-ON profile defines a sound and easy way to get notified when an object is dead through the use of `EnqueuedWeakReference` objects: `EnqueuedWeakReference` is a subclass of `WeakReference`. When such objects get their weak reference set to `null` by the system, they are added to a `ReferenceQueue` they were assigned to at their creation.

3.3.2 Death Notification Actions

Once an object has expired, it cannot be brought to life again. It is the responsibility of the application to make provisions for all actions that have to be taken on an object death. Such provisions are materialized by subclasses of the `EnqueueWeakReference` class.

`ReferenceQueue.poll()` and `ReferenceQueue.remove()` allow the execution of a hook at the death of the object referenced by the weak reference. The first one returns `null` when queue is empty whereas the second one blocks while the queue is empty.

The application is responsible of the execution of such hook.

3.3.3 Weak objects association

`java.util.Hashtable` allows to associate a `value` with a `key` within a table (the key indexes the value within the table for fast searches). It prevents both key and value from being discarded by the garbage collector.
B-ON defines the `ej.bon.WeakHashtable` class as a subclass of `java.util.Hashtable`. `WeakHashtable` allows to relax such hard constraint on the key, which becomes a weak reference within the table. If no other regular reference refers the key, the key can be removed automatically by the system, which removes the associated value too.

4 RUNTIME PHASES

B-ON defines two phases of execution:

- *initialization phase*: this is the very first Java code that executes. Its purpose is to let the device "boot", that is, to initialize all necessary resources, like allocating buffers for drivers, performing default sanity checks, scanning hardware, etc.
- *mission phase*: once initialized, the device switches to the endless mission phase. The device and its software application run until they are switched off.

4.1 Initialization Phase

`ej.bon.Util.isInInitialization()` allows the phase to be tested.

4.1.1 Mono-threaded Phase

During the initialization phase, there is only one Java thread running: the main thread which will eventually execute the `main(String[])` method once the system enters the mission phase.

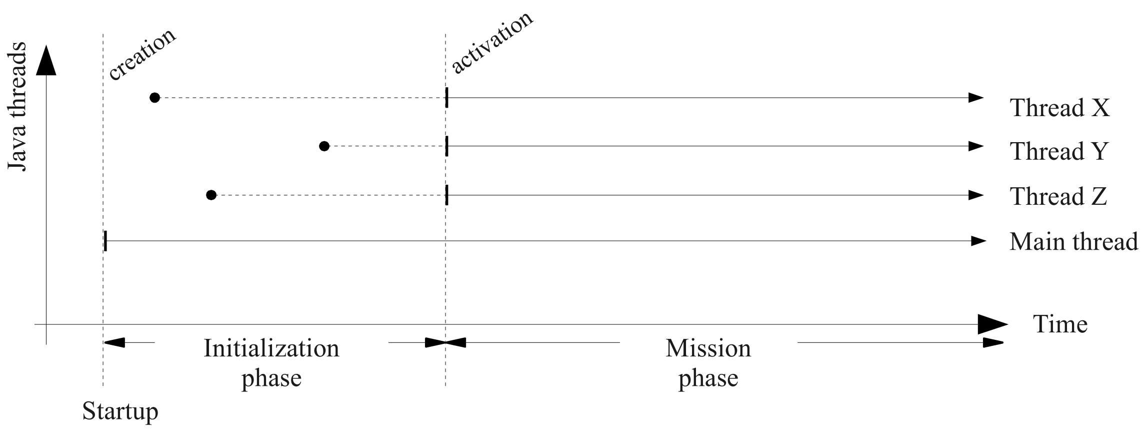

Illustration 4-1: B-ON phases and thread activation.

If other threads are created while the class initializations execute (`<clinit>` methods), those threads will be on hold (i.e. waiting) until the system enters the mission phase, even if those threads have received the `start()` message and have a higher priority than the main thread.

If the property `ej.bon.immortalAfterInit` is set, all live objects become immortal (see 3.2.2) at the end of the initialization phase.

4.1.2 Deterministic Initialization Order

If a class needs to be initialized, it defines a `<clinit>` method[12] [JVM2].

During the initialization phase, all classes which are involved within the application are initialized. It implies calling all `<clinit>` methods, in sequence.

Although the precise order of the sequence of calls is not known, it MUST be defined once for all, before any code execution. This order does not rely on runtime behavior, but only on the application code. The constraint is: if the application code does not change, the order remains the same.

The order must be compatible with the Java semantic [JLS]. Intuitively, a class may depend on other classes. Those classes should be initialized first. We list a few of these dependencies: object creation, superclass, methods receiver, arguments and fields types, … Refer to [JVM2] for a complete description of the initialization process and its implications on the order of the `<clinit>` sequence.

Dependencies of classes upon themselves define a graph of dependencies. This graph may depict cycles. The graph is linearized in an order which depends only on the graph itself.

Although B-ON experts encourage implementors of this specification to explain the order of the `<clinit>` sequence to engineers in some useful way, this is not mandatory.

The classes dependencies MUST include all the classes of pre-configured immutable objects.

12 `<clinit>` methods are not visible per se at the Java source level. They are generated by compilers: they capture the semantic of the initialization of both static fields and static initializers of classes.

The `main(String[])` method of the main class [JVM2] is an entry point in the dependencies graph.

4.2 Mission Phase

4.2.1 Thread Activations

At the beginning of the mission phase, all threads that have been started during the initialization phase activated. `ej.bon.Util.isInMission()` allows the phase to be tested.

4.2.2 Thread Control

In mission phase, one thread may send an exception within the context of another thread, using the `ej.bon.Util.throwExceptionInThread(RuntimeException,Thread)` method.

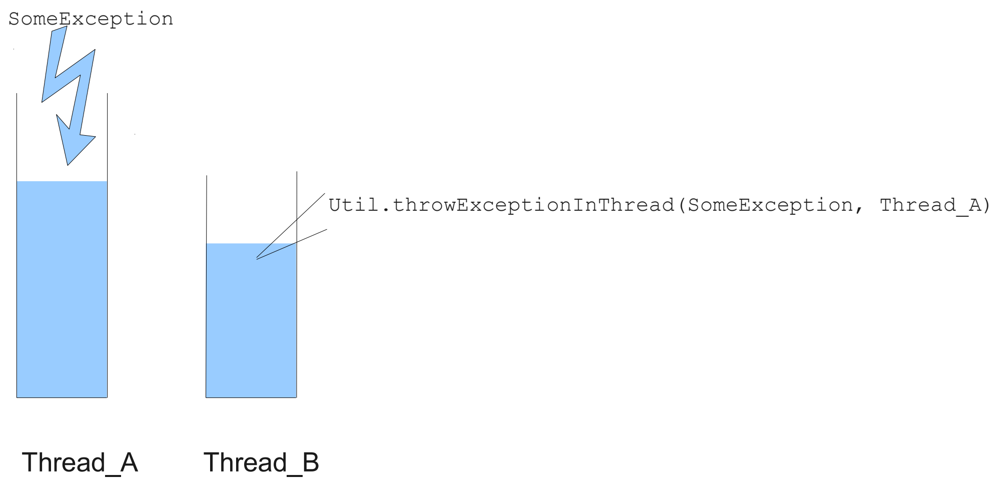

Illustration 4-2: Sending an asynchronous exception Util.throwExceptionInThread.

The exact moment at which the exception is thrown is system dependent. However, if the thread in which the exception is to be thrown has entered one or more critical sections (i.e. it holds some object's monitor) the exception is not thrown until the thread has exited all the critical sections. In such situations, the system should make its best effort to have the thread exit all the critical sections it has entered as fast as possible.

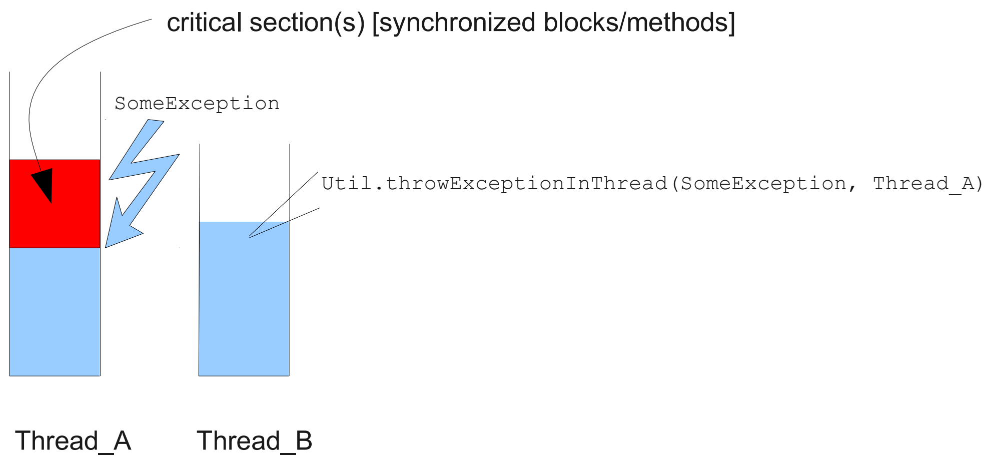
critical section(s) [synchronized blocks/methods]
SomeException
Util.throwExceptionInThread(SomeException, Thread_A)
Thread_A
Thread_B

Illustration 4-3: Exception thrown when thread has exited all critical sections.

The `ej.bon.Util.throwHardExceptionInThread(RuntimeException,Thread)` just throw the exception, as if it was sent from inside the thread. It does not wait for the critical sections to finish.

For both `throwHardExceptionInThread` and `throwExceptionInThread`, if the thread is either sleeping or waiting, the thread is unblocked (i.e. thread is interrupted: a `java.lang.InterruptedException` is thrown) and the exception is thrown as soon as possible.

4.2.3 Class.forName

If the system is capable of dynamic code downloading, `Util.dynamicCodeAllowed()` returns `true`, and this specification defines a consistent and sound way for downloading code that matches the overall semantic of B-ON:

- All referenced classes from the class given in `Class.forName(String)` have to be determined at once. They form the downloaded classes.
- Initialization of all downloaded classes MUST be ordered as specified in 4.1.2. All methods of the downloaded classes that are accessible from outside the downloaded classes scope are considered as entry points for the dependencies graph computation.
- A new thread is created in an initialization phase, which means that `Util.isInInitialization()` return `true` if executed in the context of this new thread. All class initializations of the downloaded classes are executed in that thread. As in 4.1.1, all thread activations (i.e. `Thread.start()`) are disabled until this initialization thread is done. Note that even if the property `ej.bon.immortalAfterInit` is set, objects created during this initialization phase do not become immortal.

4.3 B-ON Properties

The B-ON specification defines a set of optional properties:

- `"ej.bon.version"`: the version holds three positive integers separated by `'.'` (e.g.: `1.2.0`).
- `"ej.bon.vendor"`: the name of the B-ON library provider.

- `"ej.bon.vendor.url"`: the web site of the B-ON library provider.
- `"ej.bon.immortalAfterInit"`: if set to `true`, turn as immortal all remaining live objects at the end of the initialization phase (see 3.2.2).

5 UTILITIES

5.1 Timer & TimerTask

An `ej.bon.Timer` defines a single Java thread in charge of scheduling `Runnable` objects from the `ej.bon.TimerTask` class. All `TimerTask` are executed sequentially, according to their schedule. A `Timer` does its best effort to schedule the `TimerTask` appropriately, which depends on the `TimerTask` durations and schedules (there is no real-time guaranties).

A `TimerTask` may be scheduled repeatedly. In that case, the delay for the next schedule may depends on the end of the previous ending of the `TimerTask`, and not on some absolute time: if the previously execution of the `TimerTask` is delayed for some reason, the next executions are delayed too by the same amount of time. It is also possible to schedule repeatedly a `TimerTask` at fixed rate, which allows executions to be independent .

In case a `TimerTask` execution terminates unexpectedly, the other tasks are not impacted: the `TimerTask` is assumed to have terminated its execution regularly, and is not rescheduled event if it was scheduled repeatedly.

The main APIs are:

- `schedule(TimerTask, long)` and `schedule(TimerTask, Date)` methods allow to schedule one execution after the specified delay.
- `schedule(TimerTask task, long, long)` and `schedule(TimerTask, Date, long)` methods allow to schedule repeatedly executions, the first one after the specified delay. The waiting time between two executions is relative to the end of the previous execution.
- `scheduleAtFixedRate(TimerTask task, long, long)` and `scheduleAtFixedRate(TimerTask, Date, long)` methods allow to schedule repeatedly executions, the first one after the specified delay. The waiting time between two executions is independent of the end of the previous execution.

5.2 Platform time

The application time is the user time: it depends on its localization. `java.lang.System.currentTimeMillis` returns the application time expressed in milliseconds since midnight, January 1, 1970 UTC.

B-ON introduces a platform time that is independent from any user considerations: it materializes the running time since the very last start of the device. This time cannot be changed.

The `ej.bon.Util` class defines several methods to handle both application time and platform time:

- `platformTimeNanos` and `platformTimeMillis` method return the platform time, a `long`, expressed in nanoseconds and in milliseconds.
- `setCurrentTimeMillis(long)` and `setCurrentTimeMillis(Date)` methods allow to change the application time in order to match a user localization. This has no effect on the

platform time. `ej.bon.Util.currentTimeMillis()` method is a synonym of `java.lang.System.currentTimeMillis`.

5.3 Byte Array Accesses

The addresses space is 8-bit oriented even if there are platforms that manipulate quantities that are larger than an 8-bit: 32-bit processors for example do so. The ordering of individual addressable sub-components within the representation of a larger data item is called the endianness. `BigEndian` describes an ordering with the most significant byte first, whereas `LittleEndian` describes an ordering with the least significant byte first.

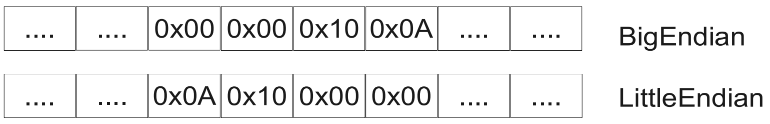
.... 0x00 0x00 0x10 0x0A ....
BigEndian
.... 0x0A 0x10 0x00 0x00 ....
LittleEndian

Illustration 5-1: Representation of the 32-bit quantity 0x0000100A using both BigEndian and in LittleEndian layout.

B-ON introduces methods to read and write into array of byte (byte[]) according to the platform endianness, or according to a specific provided endianness. The `ej.bon.ByteArray` class provides such APIs:

- `getPlatformEndianness()` returns the underlying system-dependent endianness, which mostly depends on the target processor(s).
- `readInt(byte[], int)` and `writeInt(byte[], int, int)` reads and writes an `int` using the platform specific endianness.
- `readInt(byte[], int, int)` and `writeInt(byte[], int, int, int)` reads and writes an `int` using the specified endianness as last argument, which may be either `LITTLE_ENDIAN` or `BIG_ENDIAN`.

Similar methods are provided for `short`, `char`, `long` types.

5.4 Math

The `ej.bon.XMath` complements the math operations provided by `java.lang.Math`. The new operations are: `limit`, `asin`, `acos`, `atan`, `log`, `exp`, `pow`.

6 ANNEX A: IMMUTABLES DTD

```
<!ELEMENT immutables ( object*, objectAlias*, array*, string*, class*,
null*, importObject* ) >
<!ATTLIST immutables
      name     CDATA #IMPLIED
>

<!ELEMENT object ( field*, refField* ) >
<!ATTLIST object
      id          ID    #REQUIRED
      private     (true | false) "false"
      type        CDATA #REQUIRED
>

<!ELEMENT objectAlias EMPTY >
<!ATTLIST objectAlias
      id          ID    #REQUIRED
      private     (true | false) "false"
      object      IDREF #REQUIRED
>


<!ELEMENT array ( elem*, refElem* ) >
<!ATTLIST array
      id          ID    #REQUIRED
      private     (true | false) "false"
      type        CDATA #REQUIRED
      length      CDATA #IMPLIED
>

<!ELEMENT elem EMPTY >
<!ATTLIST elem
      value CDATA #REQUIRED
>

<!ELEMENT refElem EMPTY >
<!ATTLIST refElem
      ref IDREF #REQUIRED
>

<!ELEMENT class EMPTY >
<!ATTLIST class
      id          ID    #REQUIRED
      private     (true | false) "false"
      type        CDATA #REQUIRED
>

<!ELEMENT string EMPTY >
<!ATTLIST string
      id          ID    #REQUIRED
      private     (true | false) "false"
      value       CDATA #REQUIRED
>

<!ELEMENT field EMPTY >
<!ATTLIST field
```

```
      name  CDATA #REQUIRED
      value CDATA #REQUIRED
      type  CDATA #IMPLIED
>

<!ELEMENT refField EMPTY >
<!ATTLIST refField
      name  CDATA #REQUIRED
      ref   IDREF #REQUIRED
      type  CDATA #IMPLIED
>

<!ELEMENT null EMPTY >
<!ATTLIST null
      id          ID    #REQUIRED
      private     (true | false) "false"
>

<!ELEMENT importObject EMPTY >
<!ATTLIST importObject
      id          ID    #REQUIRED
>
```

7 JAVA SPECIFICATION

Package ej.bon

Class Summary		*Page*
ByteArray	This class provides some utilities to manage I/O on a byte array.	*21*
EnqueuedWeakReference	EnqueuedWeakReference are objects that are queued in an ReferenceQueue by the system when the object they point at (see `Reference.get()`) is set to `null` by the system.	*30*
Immortals	This class gives access to the global immortal objects pool.	*32*
Immutables	This class gives access to the global immutable objects pool.	*35*
ReferenceQueue	ReferenceQueue represents a queue of EnqueuedWeakReference.	*39*
Timer	A facility for threads to schedule tasks for future execution in a background thread.	*41*
TimerTask	A task that can be scheduled for one-time or repeated execution by a Timer.	*46*
Util	This class offers basic services for B-ON implementation.	*48*
WeakHashtable	A `Hashtable` implementation with *weak keys*.	*52*
XMath		*53*

Exception Summary		*Page*
IllegalStateException	Signals that a method has been invoked at an illegal or inappropriate time.	*31*

Error Summary		*Page*
ImmutablesError	Indicates an error accessing immutables data.	*38*

Class ByteArray

ej.bon

```
java.lang.Object
  └
    ej.bon.ByteArray
```

```
public class ByteArray
extends Object
```

This class provides some utilities to manage I/O on a byte array.

Field Summary		Page
`static int`	**BIG_ENDIAN** Access mode big endian.	*22*
`static int`	**BYTE_SIZE** The size of a byte.	*22*
`static int`	**CHAR_SIZE** The size of a char.	*22*
`static int`	**INT_SIZE** The size of an int.	*23*
`static int`	**LITTLE_ENDIAN** Access mode little endian.	*22*
`static int`	**LONG_SIZE** The size of a long.	*23*
`static int`	**SHORT_SIZE** The size of a short.	*23*

Constructor Summary	Page
ByteArray`()`	*23*

Method Summary		Page
`static void`	**clear**`(byte[] array, int offset, int length)` Fills a zone of a byte array with `0`.	*28*
`static int`	**getPlatformEndianness**`()` Gets whether the platform is in big endian or little endian.	*23*
`static char`	**readChar**`(byte[] array, int offset)` Reads a char in the given byte array at the given offset respecting the endianness of the platform.	*24*
`static char`	**readChar**`(byte[] array, int offset, int endianness)` Reads a char in the given byte array at the given offset respecting the endianness of the array.	*25*
`static int`	**readInt**`(byte[] array, int offset)` Reads an int in the given byte array at the given offset respecting the endianness of the platform.	*24*
`static int`	**readInt**`(byte[] array, int offset, int endianness)` Reads an int in the given byte array at the given offset respecting the endianness of the array.	*26*
`static long`	**readLong**`(byte[] array, int offset)` Reads a long in the given byte array at the given offset respecting the endianness of the platform.	*25*
`static long`	**readLong**`(byte[] array, int offset, int endianness)` Reads a long in the given byte array at the given offset respecting the endianness of the array.	*26*

static short	**readShort**(byte[] array, int offset) Reads a short in the given byte array at the given offset respecting the endianness of the platform.	*24*
static short	**readShort**(byte[] array, int offset, int endianness) Reads a short in the given byte array at the given offset respecting the endianness of the array.	*25*
static int	**readUnsignedByte**(byte[] array, int offset) Reads an unsigned-byte in the given byte array at the given offset respecting the endianness of the platform.	*23*
static void	**set**(byte[] array, byte value, int offset, int length) Fills a zone of a byte array with the given value.	*29*
static void	**writeInt**(byte[] array, int offset, int value) Writes an int in the given byte array at the given offset respecting the endianness of the platform.	*27*
static void	**writeInt**(byte[] array, int offset, int value, int endianness) Writes an int in the given byte array at the given offset respecting the endianness of the array.	*28*
static void	**writeLong**(byte[] array, int offset, long value) Writes a long in the given byte array at the given offset respecting the endianness of the platform.	*27*
static void	**writeLong**(byte[] array, int offset, long value, int endianness) Writes a long in the given byte array at the given offset respecting the endianness of the array.	*28*
static void	**writeShort**(byte[] array, int offset, int value) Writes a short in the given byte array at the given offset respecting the endianness of the platform.	*26*
static void	**writeShort**(byte[] array, int offset, int value, int endianness) Writes a short in the given byte array at the given offset respecting the endianness of the array.	*27*

Field Detail

LITTLE_ENDIAN

`public static final int` **`LITTLE_ENDIAN`**

Access mode little endian.

BIG_ENDIAN

`public static final int` **`BIG_ENDIAN`**

Access mode big endian.

BYTE_SIZE

`public static final int` **`BYTE_SIZE`**

The size of a byte.

CHAR_SIZE

`public static final int` **`CHAR_SIZE`**

The size of a char.

SHORT_SIZE

```
public static final int SHORT_SIZE
```

The size of a short.

INT_SIZE

```
public static final int INT_SIZE
```

The size of an int.

LONG_SIZE

```
public static final int LONG_SIZE
```

The size of a long.

Constructor Detail

ByteArray

```
public ByteArray()
```

Method Detail

getPlatformEndianness

```
public static int getPlatformEndianness()
```

Gets whether the platform is in big endian or little endian.

Returns:
BIG_ENDIAN if the platform is in big endian, LITTLE_ENDIAN if in little endian

---

readUnsignedByte

```
public static int readUnsignedByte(byte[] array,
                                   int offset)
```

Reads an unsigned-byte in the given byte array at the given offset respecting the endianness of the platform.

Parameters:
`array` - the byte array to read in
`offset` - the offset of the value to read
Returns:
the read value
Throws:
`NullPointerException` - if the given array is null
`ArrayIndexOutOfBoundsException` - if read outside the bounds of the given array
See Also:
BYTE_SIZE

---

readShort

```
public static short readShort(byte[] array,
                              int offset)
```

Reads a short in the given byte array at the given offset respecting the endianness of the platform.

Parameters:
`array` - the byte array to read in
`offset` - the offset of the value to read
Returns:
the read value
Throws:
`NullPointerException` - if the given array is null
`ArrayIndexOutOfBoundsException` - if read outside the bounds of the given array
See Also:
SHORT_SIZE

---

readChar

```
public static char readChar(byte[] array,
                            int offset)
```

Reads a char in the given byte array at the given offset respecting the endianness of the platform.

Parameters:
`array` - the byte array to read in
`offset` - the offset of the value to read
Returns:
the read value
Throws:
`NullPointerException` - if the given array is null
`ArrayIndexOutOfBoundsException` - if read outside the bounds of the given array
See Also:
CHAR_SIZE

---

readInt

```
public static int readInt(byte[] array,
                          int offset)
```

Reads an int in the given byte array at the given offset respecting the endianness of the platform.

Parameters:
`array` - the byte array to read in
`offset` - the offset of the value to read
Returns:
the read value
Throws:
`NullPointerException` - if the given array is null
`ArrayIndexOutOfBoundsException` - if read outside the bounds of the given array
See Also:
INT_SIZE

---

readLong

```
public static long readLong(byte[] array,
                            int offset)
```

Reads a long in the given byte array at the given offset respecting the endianness of the platform.

Parameters:
`array` - the byte array to read in
`offset` - the offset of the value to read
Returns:
the read value
Throws:
`NullPointerException` - if the given array is null
`ArrayIndexOutOfBoundsException` - if read outside the bounds of the given array
See Also:
LONG_SIZE

---

readShort

```
public static short readShort(byte[] array,
                              int offset,
                              int endianness)
```

Reads a short in the given byte array at the given offset respecting the endianness of the array.

Parameters:
`array` - the byte array to read in
`offset` - the offset of the value to read
`endianness` - the access mode (BIG_ENDIAN or LITTLE_ENDIAN)
Returns:
the read value
Throws:
`NullPointerException` - if the given array is null
`ArrayIndexOutOfBoundsException` - if read outside the bounds of the given array
See Also:
SHORT_SIZE

---

readChar

```
public static char readChar(byte[] array,
                            int offset,
                            int endianness)
```

Reads a char in the given byte array at the given offset respecting the endianness of the array.

Parameters:
`array` - the byte array to read in
`offset` - the offset of the value to read
`endianness` - the access mode (BIG_ENDIAN or LITTLE_ENDIAN)
Returns:
the read value
Throws:
`NullPointerException` - if the given array is null
`ArrayIndexOutOfBoundsException` - if read outside the bounds of the given array
See Also:
CHAR_SIZE

---

readInt

```
public static int readInt(byte[] array,
                          int offset,
                          int endianness)
```

Reads an int in the given byte array at the given offset respecting the endianness of the array.

Parameters:
`array` - the byte array to read in
`offset` - the offset of the value to read
`endianness` - the access mode (BIG_ENDIAN or LITTLE_ENDIAN)

Returns:
the read value

Throws:
`NullPointerException` - if the given array is null
`ArrayIndexOutOfBoundsException` - if read outside the bounds of the given array

See Also:
INT_SIZE

---

readLong

```
public static long readLong(byte[] array,
                            int offset,
                            int endianness)
```

Reads a long in the given byte array at the given offset respecting the endianness of the array.

Parameters:
`array` - the byte array to read in
`offset` - the offset of the value to read
`endianness` - the access mode (BIG_ENDIAN or LITTLE_ENDIAN)

Returns:
the read value

Throws:
`NullPointerException` - if the given array is null
`ArrayIndexOutOfBoundsException` - if read outside the bounds of the given array

See Also:
LONG_SIZE

---

writeShort

```
public static void writeShort(byte[] array,
                              int offset,
                              int value)
```

Writes a short in the given byte array at the given offset respecting the endianness of the platform.

Parameters:
`array` - the byte array to write in
`offset` - the offset of the value to write
`value` - the value to write

Throws:
`NullPointerException` - if the given array is null
`ArrayIndexOutOfBoundsException` - if write outside the bounds of the given array

See Also:
SHORT_SIZE

---

writeInt

```
public static void writeInt(byte[] array,
                            int offset,
                            int value)
```

Writes an int in the given byte array at the given offset respecting the endianness of the platform.

Parameters:
`array` - the byte array to write in
`offset` - the offset of the value to write
`value` - the value to write
Throws:
`ArrayIndexOutOfBoundsException` - if write outside the bounds of the given array
See Also:
INT_SIZE

---

writeLong

```
public static void writeLong(byte[] array,
                             int offset,
                             long value)
```

Writes a long in the given byte array at the given offset respecting the endianness of the platform.

Parameters:
`array` - the byte array to write in
`offset` - the offset of the value to write
`value` - the value to write
Throws:
`ArrayIndexOutOfBoundsException` - if write outside the bounds of the given array
See Also:
LONG_SIZE

---

writeShort

```
public static void writeShort(byte[] array,
                              int offset,
                              int value,
                              int endianness)
```

Writes a short in the given byte array at the given offset respecting the endianness of the array.

Parameters:
`array` - the byte array to write in
`offset` - the offset of the value to write
`value` - the value to write
`endianness` - the access mode (BIG_ENDIAN or LITTLE_ENDIAN)
Throws:
`ArrayIndexOutOfBoundsException` - if write outside the bounds of the given array
See Also:
SHORT_SIZE

---

writeInt

```
public static void writeInt(byte[] array,
                            int offset,
                            int value,
                            int endianness)
```

Writes an int in the given byte array at the given offset respecting the endianness of the array.

Parameters:
`array` - the byte array to write in
`offset` - the offset of the value to write
`value` - the value to write
`endianness` - the access mode (BIG_ENDIAN or LITTLE_ENDIAN)

Throws:
`ArrayIndexOutOfBoundsException` - if write outside the bounds of the given array

See Also:
INT_SIZE

---

writeLong

```
public static void writeLong(byte[] array,
                             int offset,
                             long value,
                             int endianness)
```

Writes a long in the given byte array at the given offset respecting the endianness of the array.

Parameters:
`array` - the byte array to write in
`offset` - the offset of the value to write
`value` - the value to write
`endianness` - the access mode (BIG_ENDIAN or LITTLE_ENDIAN)

Throws:
`ArrayIndexOutOfBoundsException` - if write outside the bounds of the given array

See Also:
LONG_SIZE

---

clear

```
public static void clear(byte[] array,
                         int offset,
                         int length)
```

Fills a zone of a byte array with `0`.

Parameters:
`array` - the byte array to clear
`offset` - the offset of the zone to clear
`length` - the length of the zone to clear

Throws:
`ArrayIndexOutOfBoundsException` - if write outside the bounds of the given array

---

set

```
public static void set(byte[] array,
                       byte value,
                       int offset,
                       int length)
```

Fills a zone of a byte array with the given value.

Parameters:

`array` - the byte array to set
`value` - the value to fill the zone with
`offset` - the offset of the zone to set
`length` - the length of the zone to set

Throws:

`ArrayIndexOutOfBoundsException` - if write outside the bounds of the given array

Class EnqueuedWeakReference

ej.bon

```
java.lang.Object
  └
    java.lang.ref.Reference
      └
        java.lang.ref.WeakReference
          └
            ej.bon.EnqueuedWeakReference
```

```
public class EnqueuedWeakReference
extends WeakReference
```

EnqueuedWeakReference are objects that are queued in an ReferenceQueue by the system when the object they point at (see `Reference.get()`) is set to `null` by the system. A typical use is to subclass `EnqueuedWeakReference` with classes that hold native handles that need to be freed at the native level.

Constructor Summary	*Page*
EnqueuedWeakReference(Object ref, ReferenceQueue queue) Creates a new `EnqueuedWeakReference`.	*30*

Constructor Detail

EnqueuedWeakReference

```
public EnqueuedWeakReference(Object ref,
                             ReferenceQueue queue)
```

Creates a new `EnqueuedWeakReference`.
The given reference can be retrieved using `Reference.get()` until the object is garbage collected. Then the method will return `null`.

Parameters:

`ref` - object the new weak reference will refer to
`queue` - the queue with which the reference is to be registered

Class IllegalStateException

ej.bon

```
java.lang.Object
  └
    java.lang.Throwable
      └
        java.lang.Exception
          └
            java.lang.RuntimeException
              └
                ej.bon.IllegalStateException
```

```
public class IllegalStateException
extends RuntimeException
```

Signals that a method has been invoked at an illegal or inappropriate time. In other words, the Java environment or Java application is not in an Appropriate state for the requested operation.

Constructor Summary	*Page*
IllegalStateException`()` Constructs an IllegalStateException with no detail message.	*31*
IllegalStateException`(String s)` Constructs an IllegalStateException with the specified detail message.	*31*

Constructor Detail

IllegalStateException

`public IllegalStateException()`

Constructs an IllegalStateException with no detail message.

IllegalStateException

`public IllegalStateException(String s)`

Constructs an IllegalStateException with the specified detail message.

Parameters:
`s` - the String that contains a detailed message

Class Immortals

ej.bon

```
java.lang.Object
  └─
    ej.bon.Immortals
```

```
public class Immortals
extends Object
```

This class gives access to the global immortal objects pool.

An immortal object has two major properties:

- it is not managed by the garbage collector,
- it does not move around in memory, i.e. its physical memory location remains the same forever.

Constructor Summary	*Page*
Immortals()	*32*

Method Summary		*Page*
static Object	**deepImmortal**(Object root) Turns the given object and all objects referred from it into immortal objects.	*33*
static long	**freeMemory**() Returns the amount of free immortal memory still available.	*34*
static boolean	**isImmortal**(Object o) Gets whether an object is immortal or not.	*32*
static void	**run**(Runnable runnable) Calls the method `Runnable.run()` of the given runnable.	*33*
static Object	**setImmortal**(Object o) Turns the given object into an immortal object.	*33*
static long	**totalMemory**() Returns the total amount of immortal memory.	*34*

Constructor Detail

Immortals

```
public Immortals()
```

Method Detail

isImmortal

```
public static boolean isImmortal(Object o)
```

Gets whether an object is immortal or not.

An object is immortal:

- if it has been declared as immortal calling setImmortal(Object),
- if it is immutable.

Parameters:
`o` - the object to check
Returns:
`true` if the queried object is immortal or immutable, `false` otherwise
Throws:
`NullPointerException` - if given object is null
See Also:
Immutables.isImmutable(Object)

---

setImmortal

`public static Object **setImmortal**(Object o)`

Turns the given object into an immortal object.

If the received object was immutable, it remains immutable.
If the object was already immortal, it remains immortal.
If the object was a reclaimable object it turns into an immortal object. Upon success, the returned object is immortal, otherwise an `OutOfMemoryError` is thrown.

Returns:
the given object turned into immortal
Throws:
`OutOfMemoryError` - if the immortal memory is full

---

deepImmortal

`public static Object **deepImmortal**(Object root)`

Turns the given object and all objects referred from it into immortal objects.

Weakly reachable objects are not turned into immortal objects.

Parameters:
`root` - the root of the objects graph to turn into immortal
Returns:
the given object turned into immortal

---

run

`public static void **run**(Runnable runnable)`

Calls the method `Runnable.run()` of the given runnable.

All the objects allocated in the context of this method are directly allocated as immortals. While the `Runnable.run()` method of the runnable executes all created objects are allocated as immortal objects.

Parameters:
`runnable` - the runnable to execute
Throws:
`NullPointerException` - if the given runnable is `null`

freeMemory

```
public static long freeMemory()
```

Returns the amount of free immortal memory still available.

Returns:
the amount of free immortal memory

totalMemory

```
public static long totalMemory()
```

Returns the total amount of immortal memory.

Note that the amount of memory required to hold an object of any given type may be implementation-dependent.

Returns:
the total amount of immortal memory

Class Immutables

ej.bon

```
java.lang.Object
  └
    ej.bon.Immutables
```

```
public class Immutables
extends Object
```

This class gives access to the global immutable objects pool.

Immutable objects are persistent and normally reside in read-only-memory, such as flash memory.

Immutable objects are created in two ways:

- at runtime by calling put(String, Object) or putAll(Hashtable),
- at system/application configuration time by specifying objects in an XML configuration file.

The method get(String) allows to retrieved an object from the pool using its ID.

Constructor Summary	Page
Immutables()	35

Method Summary		Page
static String[]	**allIDs**() Returns an array with the IDs of all the objects in the pool.	37
static long	**freeMemory**() Returns the amount of free immutable memory still available.	37
static Object	**get**(String ID) Retrieves the object that match the the given ID in the immutable objects pool.	36
static boolean	**isImmutable**(Object object) Gets whether or not the given object is in the immutable objects pool or not.	36
static void	**put**(String ID, Object object) Maps the given ID to the given object in the immutable objects pool.	36
static void	**putAll**(Hashtable table) Maps all the mappings in the given table in the immutable objects pool.	36
static long	**totalMemory**() Returns the total amount of immutable memory.	37

Constructor Detail

Immutables

```
public Immutables()
```

Method Detail

get

```
public static Object get(String ID)
```

Retrieves the object that match the the given ID in the immutable objects pool.

If no object can be found with such ID, a `NoSuchElementException` is thrown.

Parameters:
`ID` - the ID of the immutable object to get
Returns:
the immutable object matching the ID
Throws:
`NoSuchElementException` - if the ID is not found
ImmutablesError - if an internal error occurred during immutable access

---

put

```
public static void put(String ID,
                       Object object)
```

Maps the given ID to the given object in the immutable objects pool.

The object can be retrieved by calling the get method with an ID that is equal to the original ID.

Parameters:
`ID` - the ID of the immutable object to set
`object` - the object to set immutable
Throws:
`NullPointerException` - if given object is `null`
`OutOfMemoryError` - if the immutable memory is full
See Also:
freeMemory()

---

putAll

```
public static void putAll(Hashtable table)
```

Maps all the mappings in the given table in the immutable objects pool.

Parameters:
`table` - the table that contains the objects to set immutable
Throws:
`NullPointerException` - if given table is `null`
`ClassCastException` - if an ID is not a `String`
`OutOfMemoryError` - if the immutable memory is full
See Also:
freeMemory(), put(String, Object)

---

isImmutable

```
public static boolean isImmutable(Object object)
```

Gets whether or not the given object is in the immutable objects pool or not.

Parameters:
`object` - the object to check

Returns:
`true` if the given object is immutable, `false` otherwise

Throws:
`NullPointerException` - if given object is `null`

---

allIDs

`public static String[] allIDs()`

Returns an array with the IDs of all the objects in the pool.

Returns:
all the immutable objects ID.

See Also:
get(String)

---

freeMemory

`public static long freeMemory()`

Returns the amount of free immutable memory still available.

Returns:
the amount of free immutable memory

---

totalMemory

`public static long totalMemory()`

Returns the total amount of immutable memory.

Note that the amount of memory required to hold an object of any given type may be implementation-dependent.

Returns:
the total amount of immutable memory

Class ImmutablesError

ej.bon

```
java.lang.Object
  └─
    java.lang.Throwable
      └─
        java.lang.Error
          └─
            java.lang.VirtualMachineError
              └─
                ej.bon.ImmutablesError
```

```
public class ImmutablesError
extends VirtualMachineError
```

Indicates an error accessing immutables data.

Constructor Summary	*Page*
ImmutablesError()	*38*
ImmutablesError(String msg)	*38*

Constructor Detail

ImmutablesError

```
public ImmutablesError()
```

ImmutablesError

```
public ImmutablesError(String msg)
```

Class ReferenceQueue

ej.bon

```
java.lang.Object
  └─
    ej.bon.ReferenceQueue
```

```
final public class ReferenceQueue
extends Object
```

ReferenceQueue represents a queue of EnqueuedWeakReference. The system is responsible for adding such EnqueuedWeakReference into the ReferenceQueue.
There are two way to query the queue in order to check and remove if a Reference has been added to the queue.
- poll() : returns null if the queue is empty, otherwise returns and removes the first element of the FIFO queue.
- remove() : blocks the current thread until the queue becomes not empty. returns and removes the first element of the FIFO queue.

See Also:
EnqueuedWeakReference

Constructor Summary	*Page*
ReferenceQueue()	*39*

Method Summary		*Page*
EnqueuedWeakReference	**poll**() Queries the queue and returns and removes the first element of the queue.	*39*
EnqueuedWeakReference	**remove**() Queries the queue, returns and removes the first element of the queue.	*39*

Constructor Detail

ReferenceQueue

```
public ReferenceQueue()
```

Method Detail

poll

```
public EnqueuedWeakReference poll()
```

Queries the queue and returns and removes the first element of the queue. If the queue is empty, returns `null`.

Returns:
EnqueuedWeakReference or `null`

---

remove

```
public EnqueuedWeakReference remove()
                             throws InterruptedException
```

Queries the queue, returns and removes the first element of the queue. If the queue is empty, blocks the current thread until the queue gets at least one element (automatically added by the system).

Returns:

EnqueuedWeakReference

Throws:

`InterruptedException` - if the thread is interrupted

Class Timer

ej.bon

```
java.lang.Object
  └
    ej.bon.Timer
```

```
public class Timer
extends Object
```

A facility for threads to schedule tasks for future execution in a background thread. Tasks may be scheduled for one-time execution, or for repeated execution at regular intervals.

Corresponding to each Timer object is a single background thread that is used to execute all of the timer's tasks, sequentially. Timer tasks should complete quickly. If a timer task takes excessive time to complete, it "hogs" the timer's task execution thread. This can, in turn, delay the execution of subsequent tasks, which may "bunch up" and execute in rapid succession when (and if) the offending task finally completes.

By default, the task execution thread does not run as a daemon thread, so it is capable of keeping an application from terminating. If a caller wants to terminate a timer's task execution thread, the caller should invoke the timer's cancel method.

If a task execution terminates unexpectedly, the uncaughtException method is invoked on this task.

This class is thread-safe: multiple threads can share a single Timer object without the need for external synchronization.

This class does not offer real-time guarantees: it schedules tasks using the Object.wait(long) method. The resolution of the Timer is implementation and device dependent.

Timers function only within a single VM and are canceled when the VM exits. When the VM is started no timers exist, they are created only by application request.

Constructor Summary	*Page*
Timer`()` Creates a new timer.	*42*
Timer`(boolean automatic)` Creates a new timer.	*42*

Method Summary		*Page*
`void`	**cancel**`()` Terminates this timer, discarding any currently scheduled tasks.	*45*
`void`	**run**`()` Executes TimerTask scheduling.	*45*
`void`	**schedule**`(TimerTask task, Date time)` Schedules the specified task for execution at the specified time.	*42*
`void`	**schedule**`(TimerTask task, Date firstTime, long period)` Schedules the specified task for repeated fixed-delay execution, beginning at the specified time.	*43*
`void`	**schedule**`(TimerTask task, long delay)` Schedules the specified task for execution after the specified delay.	*42*
`void`	**schedule**`(TimerTask task, long delay, long period)` Schedules the specified task for repeated fixed-delay execution, beginning after the specified delay.	*43*
`void`	**scheduleAtFixedRate**`(TimerTask task, Date firstTime, long period)` Schedules the specified task for repeated fixed-rate execution, beginning at the specified time.	*44*
`void`	**scheduleAtFixedRate**`(TimerTask task, long delay, long period)` Schedules the specified task for repeated fixed-rate execution, beginning after the specified delay.	*44*

Constructor Detail

Timer

`public Timer()`

Creates a new timer. The associated thread does not run as a daemon thread, which may prevent an application from terminating.

Timer

`public Timer(boolean automatic)`

Creates a new timer.

If the given boolean is `true`, a thread is created to execute this timer. The associated thread does not run as a daemon thread, which may prevent an application from terminating. All the TimerTask scheduled by the timer will be executed in the context of the associated thread.

Otherwise an applicative thread must call the run() method in order to execute tasks scheduling.

Parameters:
`automatic` - If `true` a thread is created to run this timer, otherwise the application must manage this timer.
See Also:
run()

Method Detail

schedule

```
public void schedule(TimerTask task,
                     long delay)
```

Schedules the specified task for execution after the specified delay.

Parameters:
`task` - task to be scheduled.
`delay` - delay in milliseconds before task is to be executed. Note that the actual delay may be different than the amount requested since the resolution of the Timer is implementation and device dependent.
Throws:
`IllegalArgumentException` - if delay is negative, or delay + CurrentTime.get() is negative.
IllegalStateException - if task was already scheduled or canceled, or timer was canceled.

---

schedule

```
public void schedule(TimerTask task,
                     Date time)
```

Schedules the specified task for execution at the specified time. If the time is in the past, the task is scheduled for immediate execution.

Parameters:
`task` - task to be scheduled.
`time` - time at which task is to be executed.
Throws:
`IllegalArgumentException` - if time.getTime() is negative.

IllegalStateException - if task was already scheduled or canceled, timer was canceled, or timer thread terminated.

---

schedule

```
public void schedule(TimerTask task,
                     long delay,
                     long period)
```

Schedules the specified task for repeated fixed-delay execution, beginning after the specified delay. Subsequent executions take place at approximately regular intervals separated by the specified period. Note that the actual delay may be different than the amount requested since the resolution of the Timer is implementation and device dependent.

In fixed-delay execution, each execution is scheduled relative to the actual execution time of the previous execution. If an execution is delayed for any reason (such as garbage collection or other background activity), subsequent executions will be delayed as well. In the long run, the frequency of execution will generally be slightly lower than the reciprocal of the specified period (assuming the system clock underlying Object.wait(long) is accurate).

Fixed-delay execution is appropriate for recurring activities that require "smoothness." In other words, it is appropriate for activities where it is more important to keep the frequency accurate in the short run than in the long run. This includes most animation tasks, such as blinking a cursor at regular intervals. It also includes tasks wherein regular activity is performed in response to human input, such as automatically repeating a character as long as a key is held down.

Parameters:
`task` - task to be scheduled.
`delay` - delay in milliseconds before task is to be executed. Note that the actual delay may be different than the amount requested since the resolution of the Timer is implementation and device dependent.
`period` - time in milliseconds between successive task executions.

Throws:
`IllegalArgumentException` - if delay is negative, or delay + CurrentTime.get() is negative.
IllegalStateException - if task was already scheduled or canceled, timer was canceled, or timer thread terminated.

---

schedule

```
public void schedule(TimerTask task,
                     Date firstTime,
                     long period)
```

Schedules the specified task for repeated fixed-delay execution, beginning at the specified time. Subsequent executions take place at approximately regular intervals, separated by the specified period.

In fixed-delay execution, each execution is scheduled relative to the actual execution time of the previous execution. If an execution is delayed for any reason (such as garbage collection or other background activity), subsequent executions will be delayed as well. In the long run, the frequency of execution will generally be slightly lower than the reciprocal of the specified period (assuming the system clock underlying Object.wait(long) is accurate).

Fixed-delay execution is appropriate for recurring activities that require "smoothness." In other words, it is appropriate for activities where it is more important to keep the frequency accurate in the short run than in the long run. This includes most animation tasks, such as blinking a cursor at regular intervals. It also includes tasks wherein regular activity is performed in response to human input, such as automatically repeating a character as long as a key is held down.

Parameters:
`task` - task to be scheduled.
`firstTime` - First time at which task is to be executed.
`period` - time in milliseconds between successive task executions.

Throws:

`IllegalArgumentException` - if time.getTime() is negative.
IllegalStateException - if task was already scheduled or canceled, timer was canceled, or timer thread terminated.

---

scheduleAtFixedRate

```
public void scheduleAtFixedRate(TimerTask task,
                                long delay,
                                long period)
```

Schedules the specified task for repeated fixed-rate execution, beginning after the specified delay. Subsequent executions take place at approximately regular intervals, separated by the specified period.

In fixed-rate execution, each execution is scheduled relative to the scheduled execution time of the initial execution. If an execution is delayed for any reason (such as garbage collection or other background activity), two or more executions will occur in rapid succession to "catch up." In the long run, the frequency of execution will be exactly the reciprocal of the specified period (assuming the system clock underlying Object.wait(long) is accurate).

Fixed-rate execution is appropriate for recurring activities that are sensitive to absolute time, such as ringing a chime every hour on the hour, or running scheduled maintenance every day at a particular time. It is also appropriate for for recurring activities where the total time to perform a fixed number of executions is important, such as a countdown timer that ticks once every second for ten seconds. Finally, fixed-rate execution is appropriate for scheduling multiple repeating timer tasks that must remain synchronized with respect to one another.

Parameters:

`task` - task to be scheduled.
`delay` - delay in milliseconds before task is to be executed. Note that the actual delay may be different than the amount requested since the resolution of the Timer is implementation and device dependent.
`period` - time in milliseconds between successive task executions.

Throws:

`IllegalArgumentException` - if delay is negative, or delay + CurrentTime.get() is negative.
IllegalStateException - if task was already scheduled or canceled, timer was canceled, or timer thread terminated.

---

scheduleAtFixedRate

```
public void scheduleAtFixedRate(TimerTask task,
                                Date firstTime,
                                long period)
```

Schedules the specified task for repeated fixed-rate execution, beginning at the specified time. Subsequent executions take place at approximately regular intervals, separated by the specified period.

In fixed-rate execution, each execution is scheduled relative to the scheduled execution time of the initial execution. If an execution is delayed for any reason (such as garbage collection or other background activity), two or more executions will occur in rapid succession to "catch up." In the long run, the frequency of execution will be exactly the reciprocal of the specified period (assuming the system clock underlying Object.wait(long) is accurate).

Fixed-rate execution is appropriate for recurring activities that are sensitive to absolute time, such as ringing a chime every hour on the hour, or running scheduled maintenance every day at a particular time. It is also appropriate for for recurring activities where the total time to perform a fixed number of executions is important, such as a countdown timer that ticks once every second for ten seconds. Finally, fixed-rate execution is appropriate for scheduling multiple repeating timer tasks that must remain synchronized with respect to one another.

Parameters:

`task` - task to be scheduled.
`firstTime` - first time at which task is to be executed.

`period` - time in milliseconds between successive task executions.

Throws:

`IllegalArgumentException` - if time.getTime() is negative.
IllegalStateException - if task was already scheduled or canceled, timer was canceled, or timer thread terminated.

---

cancel

```
public void cancel()
```

Terminates this timer, discarding any currently scheduled tasks. Does not interfere with a currently executing task (if it exists). Once a timer has been terminated, its execution thread terminates gracefully, and no more tasks may be scheduled on it. Note that calling this method from within the run method of a timer task that was invoked by this timer absolutely guarantees that the ongoing task execution is the last task execution that will ever be performed by this timer.

---

run

```
public void run()
```

Executes TimerTask scheduling. This method must be called only for Timer that are not automatic (i.e. no thread is automatically started at Timer creation). This is the current thread that executes the Timer scheduling loop.

This method can be called once on this Timer and returns only if cancel() is called.

Throws:

IllegalStateException - if this timer is already running or canceled.

Class TimerTask

ej.bon

```
java.lang.Object
  └
    ej.bon.TimerTask
```

All Implemented Interfaces:

Runnable

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

A task that can be scheduled for one-time or repeated execution by a Timer.

Constructor Summary		Page
protected	TimerTask() Creates a new timer task.	46

Method Summary		Page
boolean	cancel() Cancels this timer task.	47
abstract void	run() The action to be performed by this timer task.	46
long	scheduledExecutionTime() Returns the scheduled execution time of the most recent actual execution of this task.	47
void	uncaughtException(Timer timer, Throwable e) Method invoked when this TimerTask terminates due to the given uncaught exception.	47

Constructor Detail

TimerTask

```
protected TimerTask()
```

Creates a new timer task.

Method Detail

run

```
public abstract void run()
```

The action to be performed by this timer task.

Specified by:

`run` in interface `Runnable`

See Also:

`Thread.run()`

cancel

```
public boolean cancel()
```

Cancels this timer task. If the task has been scheduled for one-time execution and has not yet run, or has not yet been scheduled, it will never run. If the task has been scheduled for repeated execution, it will never run again. (If the task is running when this call occurs, the task will run to completion, but will never run again.) Note that calling this method from within the run method of a repeating timer task absolutely guarantees that the timer task will not run again. This method may be called repeatedly; the second and subsequent calls have no effect.

Returns:

`true` if this task is scheduled for one-time execution and has not yet run, or this task is scheduled for repeated execution. Returns `false` if the task was scheduled for one-time execution and has already run, or if the task was never scheduled, or if the task was already canceled (Loosely speaking, this method returns `true` if it prevents one or more scheduled executions from taking place.)

scheduledExecutionTime

```
public long scheduledExecutionTime()
```

Returns the scheduled execution time of the most recent actual execution of this task. (If this method is invoked while task execution is in progress, the return value is the scheduled execution time of the ongoing task execution.)

This method is typically invoked from within a task's run method, to determine whether the current execution of the task is sufficiently timely to warrant performing the scheduled activity:

```
public void run() {
if (CurrentTime.get() - scheduledExecutionTime() >= MAX_TARDINESS)
return; // Too late; skip this execution.
// Perform the task
```

This method is typically not used in conjunction with fixed-delay execution repeating tasks, as their scheduled execution times are allowed to drift over time, and so are not terribly significant.

Returns:

the time at which the most recent execution of this task was scheduled to occur, in the format returned by Date.getTime(). The return value is undefined if the task has yet to commence its first execution.

See Also:

`Date.getTime()`

uncaughtException

```
public void uncaughtException(Timer timer,
                              Throwable e)
```

Method invoked when this TimerTask terminates due to the given uncaught exception.

Default behavior of this method is to cancel the task and to invoke `Throwable.printStackTrace()` method on the given exception.

Any exception thrown by this method will be ignored.

Parameters:

`timer` - The Timer on which this TimerTask is scheduled.
`e` - The uncaught exception.

Class Util

ej.bon

```
java.lang.Object
  └
    ej.bon.Util
```

```
public class Util
extends Object
```

This class offers basic services for B-ON implementation.

Constructor Summary	Page
Util()	48

Method Summary		Page
static long	currentTimeMillis() Gets the application time in milliseconds.	50
static boolean	dynamicCodeAllowed() Tests the ability of the system to download code through `Class.forName(String)`	49
static boolean	isInInitialization() Indicates whether the current code is part of the initialization phase.	49
static boolean	isInMission() Indicates whether the system has entered the mission phase, i.e. it is initialized.	48
static long	platformTimeMillis() Gets an arbitrary time in milliseconds.	50
static long	platformTimeNanos() Gets an arbitrary time in nanoseconds.	50
static void	setCurrentTimeMillis(Date d) Sets the application time.	51
static void	setCurrentTimeMillis(long t) Sets the application time.	51
static void	throwExceptionInThread(RuntimeException e, Thread t) Throws an exception in a specified thread.	49
static void	throwHardExceptionInThread(RuntimeException e, Thread t) Throws an exception in a specified thread.	50

Constructor Detail

Util

```
public Util()
```

Method Detail

isInMission

```
public static boolean isInMission()
```

Indicates whether the system has entered the mission phase, i.e. it is initialized.

Returns:
`true` if the initialization is done

See Also:
Immortals

---

isInInitialization

```
public static boolean isInInitialization()
```

Indicates whether the current code is part of the initialization phase.
When `Class.forName(String)` triggers classes to be loaded at runtime dynamically, class initializations are done in a context where isInInitialization() is `true` and isInMission() is `true`.

Returns:
`true` if the initialization is ongoing

See Also:
Immortals

---

dynamicCodeAllowed

```
public static boolean dynamicCodeAllowed()
```

Tests the ability of the system to download code through `Class.forName(String)`

Returns:
`true` if the system allows dynamic code to be loaded, `false` otherwise

---

throwExceptionInThread

```
public static void throwExceptionInThread(RuntimeException e,
                                          Thread t)
```

Throws an exception in a specified thread.

- If the thread is either sleeping or waiting, the thread is unblocked and the exception is thrown as soon as possible.
- If the thread is running, the exception is thrown just as if a throw statement was the next instruction to execute.
- If the thread is not started yet or is terminated, nothing is done.
- If the thread has entered one or more critical sections (i.e. it holds some object's monitor) the exception is not thrown until the thread has exited all the critical sections.
- If an exception thrown via throwExceptionInThread(RuntimeException, Thread) or throwHardExceptionInThread(RuntimeException, Thread) is already pending for the thread, nothing is done.

If any of the arguments is `null`, an `IllegalArgumentException` is thrown.

Parameters:
`e` - the exception to throw
`t` - the thread in which the exception is thrown

Throws:
`IllegalArgumentException` - if any of the arguments is `null`.

---

throwHardExceptionInThread

```
public static void throwHardExceptionInThread(RuntimeException e,
                                              Thread t)
```

Throws an exception in a specified thread.

- If the thread is either sleeping or waiting, the thread is unblocked and the exception is thrown as soon as possible.
- If the thread is running, the exception is thrown just as if a throw statement was the next instruction to execute.
- If the thread is not started yet or is terminated, nothing is done.
- If the thread has entered one or more critical sections, it does not wait the critical sections to finish and the exception is thrown as soon as possible.
- If an exception thrown via throwHardExceptionInThread(RuntimeException, Thread) is already pending for the thread, nothing is done.
- If an exception thrown via throwExceptionInThread(RuntimeException, Thread) is already pending for the thread, this exception is replaced by the given exception.

Parameters:
`e` - the exception to throw
`t` - the thread in which the exception is thrown
Throws:
`IllegalArgumentException` - if any of the arguments is `null`.

---

currentTimeMillis

```
public static long currentTimeMillis()
```

Gets the application time in milliseconds.

The result of this method is the same as the `System.currentTimeMillis()` method one.

Returns:
the application time in milliseconds

---

platformTimeMillis

```
public static long platformTimeMillis()
```

Gets an arbitrary time in milliseconds.

Only elapsed time between two calls is meaningful.

Returns:
the platform time in milliseconds

---

platformTimeNanos

```
public static long platformTimeNanos()
```

Gets an arbitrary time in nanoseconds.

Only elapsed time between two calls is meaningful.

Returns:
the platform time in nanoseconds

---

setCurrentTimeMillis

```
public static void setCurrentTimeMillis(long t)
```

Sets the application time.

This time does not change the platform time.

Parameters:
`t` - the application time to set in milliseconds
Throws:
`IllegalArgumentException` - if `t` is negative

---

setCurrentTimeMillis

```
public static void setCurrentTimeMillis(Date d)
```

Sets the application time.

This time does not change the platform time. The `Util.setCurrentTimeMillis(d)` method has the same effect as `Util.setCurrentTimeMillis(d.getTime())`.

Parameters:
`d` - the application time to set

Class WeakHashtable

ej.bon

```
java.lang.Object
  └─
    java.util.Hashtable
      └─
        ej.bon.WeakHashtable
```

```
public class WeakHashtable
extends Hashtable
```

A `Hashtable` implementation with *weak keys*. An entry in a WeakHashtable will automatically be removed when its key is no longer in ordinary use. More precisely, the presence of a mapping for a given key will not prevent the key from being discarded by the garbage collector and then reclaimed. When a key has been discarded its entry is effectively removed from the hashtable, so this class behaves somewhat differently from `Hashtable` implementation.

Each key object in a WeakHashtable is stored indirectly as the referent of a weak reference. Therefore a key will automatically be removed only after the weak references to it, both inside and outside of the map, have been cleared by the garbage collector.

Implementation note: The value objects in a WeakHashtable are held by ordinary strong references. Thus care should be taken to ensure that value objects do not strongly refer to their own keys, either directly or indirectly, since that will prevent the keys from being discarded.

Constructor Summary	*Page*
WeakHashtable`()` Constructs a new, empty weak hashtable with a default capacity and load factor.	*52*
WeakHashtable`(int initialCapacity)` Constructs a new, empty weak hashtable with the specified initial capacity.	*52*

Constructor Detail

WeakHashtable

`public WeakHashtable()`

Constructs a new, empty weak hashtable with a default capacity and load factor.

WeakHashtable

`public WeakHashtable(int initialCapacity)`

Constructs a new, empty weak hashtable with the specified initial capacity.

Class XMath

ej.bon

```
java.lang.Object
  └
    ej.bon.XMath
```

```
final public class XMath
extends Object
```

Field Summary		Page
static double	**E** The `double` value that is closer than any other to *e*, the base of the natural logarithms.	*54*
static double	**PI** The `double` value that is closer than any other to *pi*, the ratio of the circumference of a circle to its diameter.	*55*

Method Summary		Page
static double	**abs**`(double a)` Returns the absolute value of a `double` value.	*55*
static float	**abs**`(float a)` Returns the absolute value of a `float` value.	*55*
static int	**abs**`(int a)` Returns the absolute value of an `int` value.	*55*
static long	**abs**`(long a)` Returns the absolute value of a `long` value.	*56*
static double	**acos**`(double a)` Returns the arc cosine of a value; the returned angle is in the range 0.0 through *pi*.	*61*
static double	**asin**`(double a)` Returns the arc sine of a value; the returned angle is in the range -*pi*/2 through *pi*/2.	*61*
static double	**atan**`(double a)` Returns the arc tangent of a value; the returned angle is in the range -*pi*/2 through *pi*/2.	*61*
static double	**ceil**`(double a)` Returns the smallest (closest to negative infinity) `double` value that is greater than or equal to the argument and is equal to a mathematical integer.	*56*
static double	**cos**`(double a)` Returns the trigonometric cosine of an angle.	*56*
static double	**exp**`(double a)` Returns Euler's number e raised to the power of a double value.	*62*
static double	**floor**`(double a)` Returns the largest (closest to positive infinity) `double` value that is less than or equal to the argument and is equal to a mathematical integer.	*57*
static double	**limit**`(double value, double min, double max)` Limits a value between two others: • If `value` is lower than `min`, returns `min`.	*65*

static float	**limit**(float value, float min, float max) Limits a value between two others: • If value is lower than min, returns min.	64
static int	**limit**(int value, int min, int max) Limits a value between two others: • If value is lower than min, returns min.	63
static long	**limit**(long value, long min, long max) Limits a value between two others: • If value is lower than min, returns min.	64
static double	**log**(double a) Returns the natural logarithm (base e) of a double value.	62
static double	**max**(double a, double b) Returns the greater of two double values.	57
static float	**max**(float a, float b) Returns the greater of two float values.	57
static int	**max**(int a, int b) Returns the greater of two int values.	58
static long	**max**(long a, long b) Returns the greater of two long values.	58
static double	**min**(double a, double b) Returns the smaller of two double values.	58
static float	**min**(float a, float b) Returns the smaller of two float values.	58
static int	**min**(int a, int b) Returns the smaller of two int values.	59
static long	**min**(long a, long b) Returns the smaller of two long values.	59
static double	**pow**(double a, double b) Returns the value of the first argument raised to the power of the second argument.	62
static double	**sin**(double a) Returns the trigonometric sine of an angle.	59
static double	**sqrt**(double a) Returns the correctly rounded positive square root of a double value.	60
static double	**tan**(double a) Returns the trigonometric tangent of an angle.	60
static double	**toDegrees**(double angrad) Converts an angle measured in radians to an approximately equivalent angle measured in degrees.	60
static double	**toRadians**(double angdeg) Converts an angle measured in degrees to an approximately equivalent angle measured in radians.	60

Field Detail

E

`public static final double E`

The `double` value that is closer than any other to *e*, the base of the natural logarithms.

PI

```
public static final double PI
```

The `double` value that is closer than any other to *pi*, the ratio of the circumference of a circle to its diameter.

Method Detail

abs

```
public static double abs(double a)
```

Returns the absolute value of a `double` value. If the argument is not negative, the argument is returned. If the argument is negative, the negation of the argument is returned. Special cases:

- If the argument is positive zero or negative zero, the result is positive zero.
- If the argument is infinite, the result is positive infinity.
- If the argument is NaN, the result is NaN.

In other words, the result is the same as the value of the expression:

```
Double.longBitsToDouble((Double.doubleToLongBits(a)<<1)>>>1)
```

Parameters:
`a` - the argument whose absolute value is to be determined
Returns:
the absolute value of the argument.

---

abs

```
public static float abs(float a)
```

Returns the absolute value of a `float` value. If the argument is not negative, the argument is returned. If the argument is negative, the negation of the argument is returned. Special cases:

- If the argument is positive zero or negative zero, the result is positive zero.
- If the argument is infinite, the result is positive infinity.
- If the argument is NaN, the result is NaN.

In other words, the result is the same as the value of the expression:

```
Float.intBitsToFloat(0x7fffffff & Float.floatToIntBits(a))
```

Parameters:
`a` - the argument whose absolute value is to be determined
Returns:
the absolute value of the argument.

---

abs

```
public static int abs(int a)
```

Returns the absolute value of an `int` value. If the argument is not negative, the argument is returned. If the argument is negative, the negation of the argument is returned.

Note that if the argument is equal to the value of `Integer.MIN_VALUE`, the most negative representable `int` value, the result is that same value, which is negative.

Parameters:
`a` - the argument whose absolute value is to be determined
Returns:
the absolute value of the argument.
See Also:
`Integer.MIN_VALUE`

---

abs

`public static long **abs**(long a)`

Returns the absolute value of a `long` value. If the argument is not negative, the argument is returned. If the argument is negative, the negation of the argument is returned.

Note that if the argument is equal to the value of `Long.MIN_VALUE`, the most negative representable `long` value, the result is that same value, which is negative.

Parameters:
`a` - the argument whose absolute value is to be determined
Returns:
the absolute value of the argument.
See Also:
`Long.MIN_VALUE`

---

ceil

`public static double **ceil**(double a)`

Returns the smallest (closest to negative infinity) `double` value that is greater than or equal to the argument and is equal to a mathematical integer. Special cases:

- If the argument value is already equal to a mathematical integer, then the result is the same as the argument.
- If the argument is NaN or an infinity or positive zero or negative zero, then the result is the same as the argument.
- If the argument value is less than zero but greater than -1.0, then the result is negative zero.

Note that the value of `Math.ceil(x)` is exactly the value of `-Math.floor(-x)`.

Parameters:
`a` - a value.
Returns:
the smallest (closest to negative infinity) floating-point value that is greater than or equal to the argument and is equal to a mathematical integer.

---

cos

`public static double **cos**(double a)`

Returns the trigonometric cosine of an angle. Special cases:

- If the argument is NaN or an infinity, then the result is NaN.

The computed result must be within 1 ulp of the exact result. Results must be semi-monotonic.

Parameters:
`a` - an angle, in radians.
Returns:
the cosine of the argument.

---

floor

```
public static double floor(double a)
```

Returns the largest (closest to positive infinity) `double` value that is less than or equal to the argument and is equal to a mathematical integer. Special cases:

- If the argument value is already equal to a mathematical integer, then the result is the same as the argument.
- If the argument is NaN or an infinity or positive zero or negative zero, then the result is the same as the argument.

Parameters:
`a` - a value.
Returns:
the largest (closest to positive infinity) floating-point value that less than or equal to the argument and is equal to a mathematical integer.

---

max

```
public static double max(double a,
                         double b)
```

Returns the greater of two `double` values. That is, the result is the argument closer to positive infinity. If the arguments have the same value, the result is that same value. If either value is NaN, then the result is NaN. Unlike the numerical comparison operators, this method considers negative zero to be strictly smaller than positive zero. If one argument is positive zero and the other negative zero, the result is positive zero.

Parameters:
`a` - an argument.
`b` - another argument.
Returns:
the larger of `a` and `b`.

---

max

```
public static float max(float a,
                        float b)
```

Returns the greater of two `float` values. That is, the result is the argument closer to positive infinity. If the arguments have the same value, the result is that same value. If either value is NaN, then the result is NaN. Unlike the numerical comparison operators, this method considers negative zero to be strictly smaller than positive zero. If one argument is positive zero and the other negative zero, the result is positive zero.

Parameters:
`a` - an argument.
`b` - another argument.

Returns:
the larger of `a` and `b`.

---

max

```
public static long max(long a,
                       long b)
```

Returns the greater of two `long` values. That is, the result is the argument closer to the value of `Long.MAX_VALUE`. If the arguments have the same value, the result is that same value.

Parameters:
`a` - an argument.
`b` - another argument.
Returns:
the larger of `a` and `b`.
See Also:
`Long.MAX_VALUE`

---

max

```
public static int max(int a,
                      int b)
```

Returns the greater of two `int` values. That is, the result is the argument closer to the value of `Integer.MAX_VALUE`. If the arguments have the same value, the result is that same value.

Parameters:
`a` - an argument.
`b` - another argument.
Returns:
the larger of `a` and `b`.
See Also:
`Long.MAX_VALUE`

---

min

```
public static double min(double a,
                         double b)
```

Returns the smaller of two `double` values. That is, the result is the value closer to negative infinity. If the arguments have the same value, the result is that same value. If either value is NaN, then the result is NaN. Unlike the numerical comparison operators, this method considers negative zero to be strictly smaller than positive zero. If one argument is positive zero and the other is negative zero, the result is negative zero.

Parameters:
`a` - an argument.
`b` - another argument.
Returns:
the smaller of `a` and `b`.

---

min

```
public static float min(float a,
                        float b)
```

Returns the smaller of two `float` values. That is, the result is the value closer to negative infinity. If the arguments have the same value, the result is that same value. If either value is NaN, then the result is NaN. Unlike the numerical comparison operators, this method considers negative zero to be strictly smaller than positive zero. If one argument is positive zero and the other is negative zero, the result is negative zero.

Parameters:

`a` - an argument.

`b` - another argument.

Returns:

the smaller of `a` and `b`.

---

min

```
public static int min(int a,
                      int b)
```

Returns the smaller of two `int` values. That is, the result the argument closer to the value of `Integer.MIN_VALUE`. If the arguments have the same value, the result is that same value.

Parameters:

`a` - an argument.

`b` - another argument.

Returns:

the smaller of `a` and `b`.

See Also:

`Long.MIN_VALUE`

---

min

```
public static long min(long a,
                       long b)
```

Returns the smaller of two `long` values. That is, the result is the argument closer to the value of `Long.MIN_VALUE`. If the arguments have the same value, the result is that same value.

Parameters:

`a` - an argument.

`b` - another argument.

Returns:

the smaller of `a` and `b`.

See Also:

`Long.MIN_VALUE`

---

sin

```
public static double sin(double a)
```

Returns the trigonometric sine of an angle. Special cases:

- If the argument is NaN or an infinity, then the result is NaN.
- If the argument is zero, then the result is a zero with the same sign as the argument.

The computed result must be within 1 ulp of the exact result. Results must be semi-monotonic.

Parameters:

`a` - an angle, in radians.

Returns:
the sine of the argument.

---

sqrt

`public static double sqrt(double a)`

Returns the correctly rounded positive square root of a `double` value. Special cases:

- If the argument is NaN or less than zero, then the result is NaN.
- If the argument is positive infinity, then the result is positive infinity.
- If the argument is positive zero or negative zero, then the result is the same as the argument.

Otherwise, the result is the `double` value closest to the true mathematical square root of the argument value.

Parameters:
`a` - a value.
Returns:
the positive square root of `a`. If the argument is NaN or less than zero, the result is NaN.

---

tan

`public static double tan(double a)`

Returns the trigonometric tangent of an angle. Special cases:

- If the argument is NaN or an infinity, then the result is NaN.
- If the argument is zero, then the result is a zero with the same sign as the argument.

The computed result must be within 1 ulp of the exact result. Results must be semi-monotonic.

Parameters:
`a` - an angle, in radians.
Returns:
the tangent of the argument.

---

toDegrees

`public static double toDegrees(double angrad)`

Converts an angle measured in radians to an approximately equivalent angle measured in degrees. The conversion from radians to degrees is generally inexact; users should *not* expect `cos(toRadians(90.0))` to exactly equal `0.0`.

Parameters:
`angrad` - an angle, in radians
Returns:
the measurement of the angle `angrad` in degrees.

---

toRadians

`public static double toRadians(double angdeg)`

Converts an angle measured in degrees to an approximately equivalent angle measured in radians. The conversion from degrees to radians is generally inexact.

Parameters:
`angdeg` - an angle, in degrees

Returns:
the measurement of the angle `angdeg` in radians.

---

asin

```
public static double asin(double a)
```

Returns the arc sine of a value; the returned angle is in the range -*pi*/2 through *pi*/2. Special cases:

- If the argument is NaN or its absolute value is greater than 1, then the result is NaN.
- If the argument is zero, then the result is a zero with the same sign as the argument.

The computed result must be within 1 ulp of the exact result. Results must be semi-monotonic.

Parameters:
`a` - the value whose arc sine is to be returned.

Returns:
the arc sine of the argument.

---

acos

```
public static double acos(double a)
```

Returns the arc cosine of a value; the returned angle is in the range 0.0 through *pi*. Special case:

- If the argument is NaN or its absolute value is greater than 1, then the result is NaN.

The computed result must be within 1 ulp of the exact result. Results must be semi-monotonic.

Parameters:
`a` - the value whose arc cosine is to be returned.

Returns:
the arc cosine of the argument.

---

atan

```
public static double atan(double a)
```

Returns the arc tangent of a value; the returned angle is in the range -*pi*/2 through *pi*/2. Special cases:

- If the argument is NaN, then the result is NaN.
- If the argument is zero, then the result is a zero with the same sign as the argument.

The computed result must be within 1 ulp of the exact result. Results must be semi-monotonic.

Parameters:
`a` - the value whose arc tangent is to be returned.

Returns:
the arc tangent of the argument.

---

log

```
public static double log(double a)
```

Returns the natural logarithm (base e) of a double value. Special cases:

- If the argument is NaN or less than zero, then the result is NaN.
- If the argument is positive infinity, then the result is positive infinity.
- If the argument is positive zero or negative zero, then the result is negative infinity.

The computed result must be within 1 ulp of the exact result. Results must be semi-monotonic.

Parameters:
`a` - the value whose the natural logarithm is to be returned
Returns:
the value ln a

---

exp

```
public static double exp(double a)
```

Returns Euler's number e raised to the power of a double value. Special cases:

- If the argument is NaN, the result is NaN.
- If the argument is positive infinity, then the result is positive infinity.
- If the argument is negative infinity, then the result is positive zero.

The computed result must be within 1 ulp of the exact result. Results must be semi-monotonic.

Parameters:
`a` - the exponent to raise e to.
Returns:
the value ea, where e is the base of the natural logarithms.

---

pow

```
public static double pow(double a,
                         double b)
```

Returns the value of the first argument raised to the power of the second argument. Special cases:

- If the second argument is positive or negative zero, then the result is 1.0.
- If the second argument is 1.0, then the result is the same as the first argument.
- If the second argument is NaN, then the result is NaN.
- If the first argument is NaN and the second argument is nonzero, then the result is NaN.
- If
- the absolute value of the first argument is greater than 1 and the second argument is positive infinity, or
- the absolute value of the first argument is less than 1 and the second argument is negative infinity,

then the result is positive infinity.
- If
- the absolute value of the first argument is greater than 1 and the second argument is negative infinity, or
- the absolute value of the first argument is less than 1 and the second argument is positive infinity,

then the result is positive zero.
- If the absolute value of the first argument equals 1 and the second argument is infinite, then the

result is NaN.

- If
- the first argument is positive zero and the second argument is greater than zero, or
- the first argument is positive infinity and the second argument is less than zero,

then the result is positive zero.

- If
- the first argument is positive zero and the second argument is less than zero, or
- the first argument is positive infinity and the second argument is greater than zero,

then the result is positive infinity.

- If
- the first argument is negative zero and the second argument is greater than zero but not a finite odd integer, or
- the first argument is negative infinity and the second argument is less than zero but not a finite odd integer,

then the result is positive zero.

- If
- the first argument is negative zero and the second argument is a positive finite odd integer, or
- the first argument is negative infinity and the second argument is a negative finite odd integer,

then the result is negative zero.

- If
- the first argument is negative zero and the second argument is less than zero but not a finite odd integer, or
- the first argument is negative infinity and the second argument is greater than zero but not a finite odd integer,

then the result is positive infinity.

- If
- the first argument is negative zero and the second argument is a negative finite odd integer, or
- the first argument is negative infinity and the second argument is a positive finite odd integer,

then the result is negative infinity.

- If the first argument is finite and less than zero
- if the second argument is a finite even integer, the result is equal to the result of raising the absolute value of the first argument to the power of the second argument
- if the second argument is a finite odd integer, the result is equal to the negative of the result of raising the absolute value of the first argument to the power of the second argument
- if the second argument is finite and not an integer, then the result is NaN.
- If both arguments are integers, then the result is exactly equal to the mathematical result of raising the first argument to the power of the second argument if that result can in fact be represented exactly as a double value.

(In the foregoing descriptions, a floating-point value is considered to be an integer if and only if it is finite and a fixed point of the method `ceil` or, equivalently, a fixed point of the method `floor`. A value is a fixed point of a one-argument method if and only if the result of applying the method to the value is equal to the value.) The computed result must be within 1 ulp of the exact result. Results must be semi-monotonic.

Parameters:
`a` - the base.
`b` - the exponent.
Returns:
the value ab.

---

limit

```
public static int limit(int value,
                        int min,
                        int max)
```

Limits a value between two others:

- If `value` is lower than `min`, returns `min`.
- If `value` is greater than `max`, returns `max`.
- Otherwise, returns `value`.

Parameters:
`value` - the value to limit
`min` - the lower bound
`max` - the upper bound
Returns:
the limited value
Throws:
`IllegalArgumentException` - if `min` is greater than `max`

---

limit

```
public static float limit(float value,
                          float min,
                          float max)
```

Limits a value between two others:

- If `value` is lower than `min`, returns `min`.
- If `value` is greater than `max`, returns `max`.
- Otherwise, returns `value`.

Parameters:
`value` - the value to limit
`min` - the lower bound
`max` - the upper bound
Returns:
the limited value
Throws:
`IllegalArgumentException` - if `min` is greater than `max`

---

limit

```
public static long limit(long value,
                         long min,
                         long max)
```

Limits a value between two others:

1. If `value` is lower than `min`, returns `min`.
2. If `value` is greater than `max`, returns `max`.
3. Otherwise, returns `value`.

Parameters:
`value` - the value to limit
`min` - the lower bound
`max` - the upper bound
Returns:
the limited value
Throws:
`IllegalArgumentException` - if `min` is greater than `max`

---

limit

```
public static double limit(double value,
                           double min,
                           double max)
```

Limits a value between two others:

1. If `value` is lower than `min`, returns `min`.
2. If `value` is greater than `max`, returns `max`.
3. Otherwise, returns `value`.

Parameters:

`value` - the value to limit
`min` - the lower bound
`max` - the upper bound

Returns:

the limited value

Throws:

`IllegalArgumentException` - if `min` is greater than `max`