JavaSymphony

User Guide

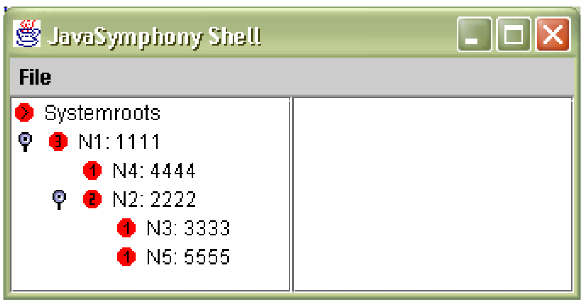
Johannes Testori (e9425269)
Markus Winnisch (e9425268)
Matthias Wohlmann (e9425261)

I. Documentation for the API programmer

1.) The Network Agent

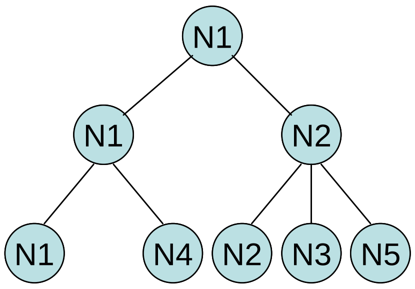
How to find a virtual architecture in a physical architecture?

Look at the following physical architecture like created in the Shell:

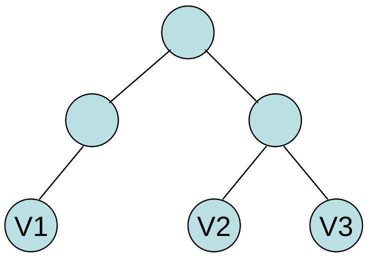


Systemroots
N1: 1111
N4: 4444
N2: 2222
N3: 3333
N5: 5555

To create an easier to understand tree you have to remember that a node with level 3 can have children with level 2 and with level 3. So a node with level 3 is also a node with level 2 and a node with level 1. The following picture will show the example above:



This representation of the tree is better to understand which virtual architectures can be found in the physically. A virtual architecture is a tree structure with working nodes only at the leafs. Look at the following example of a virtual architecture:



This virtual architecture can be found in the physical architecture above, but there are more different solutions and without knowing the constraints of the nodes in the physical architecture the following solutions are possible:

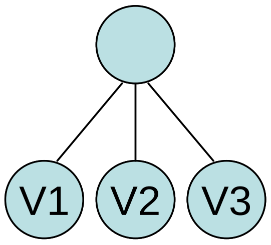
- V1 = N1; V2 = N2; V3 = N3
- V1 = N4; V2 = N5; V3 = N2
- V1 = N5; V2 = N1; V3 = N4
- …

there are many solutions but the following solutions are not possible:

- V1 = N2; V2 = N3; V3 = N5
- …

The order of the nodes of the physical architecture is not important to find a virtual architecture. Only the structure to be found is important to exist in the physical architecture.

The level of the root elements of the two architectures have not to be the same. The level of the physical architecture has to be higher or equal to the level of the virtual architecture. Remember the following virtual architecture can be found in the physical architecture above to:

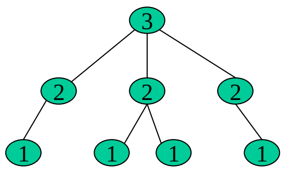


Solutions are:

- V1 = N2; V2 = N3; V3 = N5
- V1 = N3; V2 = N5; V3 = N2
- …

2.) The JavaSymphony Shell

JavaSymphony Shell is part of the JavaSymphony system. JS Shell is a graphical user interface designed to support the creation of a physical architecture consisting of connected JS Network Agents. This physical architecture is used by JS Applications to create virtual architectures and run distributed programs on them.
If a JS application demands a virtual architecture, it will be mapped to a physical architecture that fulfils the needs of the application (see documentation for network agents for information how this is done).
A physical architecture consists of nodes (Network Agents, NA's). Nodes can be grouped under a cluster; clusters can be grouped under a site and so on (where clusters and sites are normal nodes that additionally have a cluster manager or a site manager. For being expansible, we decided to use level numbers instead of names for the hierarchy, so a node has level one, a cluster level two, a site level three and so on. The maximum level is nine.
The following diagram shows a node of level three (a site) with three nodes of level two (clusters), that also have some nodes of level one under them. Remember that there are still eight network agents in this system (every higher level node has still the functionality of a normal node).



Starting the JS Shell

You can start the JS Shell by executing the batch file
run_shell.bat
in the programs main directory. This will start up the GUI and enable you to create a physical architecture. If you have already created a physical architecture and stored it in a file (see "Save Configuration"), you can use the JS Shell program to generate this stored architecture without starting the GUI by appending filename as a program argument:
run_shell [filename]
This will generate the architecture stored in the file, showing all information and errors as console output. The program will stop after generation.

The Main Window

The main window is divided into two parts. On the right side there is a list of available NA's that can be added to a physical architecture on the left side. A NA is defined by the computer name (or IP) on witch the NA is running and the port on which the NA is listening ([computer_name]:[port]). At start-up, the default list stored in a file (serverlist.ini) will be loaded and displayed in the list window as well as the configuration stored in the file config.txt will be displayed as a tree in the left part. Of course you can load and store other lists and configuration files (see "Load Configuration", "Load NA-List").

The List

The list displays all NA's that are currently available (marked with a green tick) as well as some other computers, where no NA is currently running on the specified port (marked with a red cross). NA's, that are not marked could not be reached yet, they will be marked in a few seconds. You can add NA's to the list, remove NA's from the list and load and store lists of NA's (see "Load NA-List). You can add an NA to the physical architecture on the left side by dragging and dropping it on the desired position in the architecture tree.

The Tree

The Tree window allows you to design a physical architecture consisting of NA's. You can drag and drop NA's in the tree around as well as remove them from the tree by dragging them to the list window. You can also load and store configurations to a file. If the design of your physical architecture is finished, you can create the physical architecture my selecting "Make Configuration" from the menu. Each NA in the tree will be contacted and the physical architecture will be created.
The tree in the picture above displays two independent virtual architectures, one with root NA "agnes.par.univie.ac.at" and one with "daphne.par.univie.ac.at". The node named "Systemroots" is just a dummy tree node and not a real NA.
The first one is a NA of level three (agnes) consisting of two NA's of level two (amanda, claire). The first one (amanda) consists of two NA's of level one (becky, brooke), the second consists of three NA's (darlene, dolly, edwin).
The root of the second system is a NA of level five (daphne) consisting of one NA of level four (cybill) and another one of level 2 (lisa), and so on.

The NA “sonja.par.univie.ac.at” is a backup manager (marked with a small “B”), whereas “silvia.par.univie.ac.at” is a normal NA.
If you want, you can call NA’s of level one “node”, NA’s of level two “Cluster”, NA’s of level three “Site” and NA’s of level four “Domains”, although you can create NA’s up to level nine.

The Main Menu

The Main menu is divided into two parts, the first one is associated with the tree, the second one with the list.

Load Configuration, Save Configuration, Save Configuration as

Load and store created configurations for physical architectures. At start-up, the config.txt configuration file will automatically be loaded and displayed as a tree. You can use a stored configuration file as a command line argument when starting the JS Shell, and the Shell will create the configuration immediately without starting the GUI. The configuration file is written in plain text, so you can change it if you want, but there is no syntax check when loading the file or creating a configuration using the file as a command line argument.

Make Configuration

After you are finished designing a physical architecture, you have to create it using this command. The Shell will contact every NA in the tree, setting it’s parent, children, backup parameters and the rate at which a NA checks it’s children. If the configuration is successful, you will get a notification, otherwise you get an error message and an exception trace at the command window.

Load NA-List, Save NA-List, Save NA-List as

Enables you to load and store lists of NA’s. Of course you can edit these lists, if you want to, but there is no syntax check when loading a list. At start-up serverlist.ini will automatically be loaded and displayed.

Popup-Menu Entries for the Tree

The popup-menu is accessible by right click somewhere in the tree window and consists of the same commands as the first part of the main menu (see “The Main Menu”)

Popup-Menu Entries for a NA in the Tree

Right click on a NA in the tree to get the menu. The menu consists of the entries for the tree (see “Popup-Menu Entries for the Tree”) plus the following ones:

Load Configuration, Save Configuration, Save Configuration as, Make Configuration

See “The Main Menu”

Set Level

Use it to set the Level of a NA directly. A NA that has children of level one has to have level higher than one, and so on, so the program will not allow you to set it’s level to one. If you want to change the level of an NA to a value less than it was, the program tries to lower the values of all children of this NA recursively, checking if no NA will get a level less than zero.

Set Rate

A dialog appears, where you can see and change the rate at which a NA will check if it’s children are alive. The value is in milliseconds and will not be set before you create the configuration (see “Make Configuration”). The default value is 10000.

Toggle Backup Manager

A backup manager is a NA, that checks at a given rate (see “Set Backup Rate”) if its parent is still alive, and if not, will overtake its functionality. You can set a NA to backup manager only if it has a parent. A node can only have one backup manager, so if there already is one, it will be changed. Backup managers are marked with a small “B” at the bottom of the NA icon. This setting will not be set before you create the configuration (see “Make Configuration”).

Set Backup Rate

A dialog appears where you can see and change the rate at which a NA, that is a backup manager, will check if its parent is alive (see “Toggle Backup Manager” for information about backup managers). The value is in milliseconds and will not be set before you create the configuration (see “Make Configuration”). The default value is 10000.

Show Info

Use this command, if you have already created a physical architecture using “Make Configuration”, and there is an JS application running on your system. You will get information about applications and Objects running on the NA. See “The Information Window / Migrating Objects” for further information

Event Properties

Allows you to change some event mechanism options at the event agent (see “Documentation for Object Agent System”) running on the NA. A window will pop up showing all available options and their current settings. Use the “Set”-button to change the settings immediately, the NA doesn’t necessarily have to be part of a physical architecture to do this. See ????? for information about the available options.

Threadpoolsize (PubOA)

A dialog appears, where you can see and change the thread pool size of the public object agent (PubOA) running on this NA. (see “Documentation for Object Agent System”) The settings will take place immediately, the NA doesn’t necessarily have to be part of a physical architecture to do this.

Popup-Menu Entries for the List

The popup-menu is accessible by right click somewhere in the list window and consists of the same commands as the second part of the main menu (see “The Main Menu) except for the following two:

Remove All

Removes all NA’s in the list

Add NA

Opens a dialog allowing you to enter a new NA for the list. You have to enter the name of the computer where the NA is running and the port at which the NA is listening. Alternatively you can enter the IP-address of the computer instead of a name, although it is not recommended to do this (there could be problems with using a NA twice, once with a name and once with an IP).

Popup-Menu Entries for a NA in the List

Right click on a NA in the list to get the menu. The menu consists of the entries for the list (see “Popup-Menu Entries for the List”) plus the following ones:

Get current Configuration

If the NA is part of a physical architecture that is currently active, the architecture will be displayed as a tree in the left part of the main window.

Ping

The program tries to connect to the NA. If successful, it will be marked with a green tick, else with a red cross. Each entry in the list already has an own thread that pings the NA every 30 seconds.

Show Info

See “Show Info” under “Popup-Menu Entries for a NA in the Tree”.

Event Properties

See “Event Properties” under “Popup-Menu Entries for a NA in the Tree”.

Threadpoolsize (PubOA)

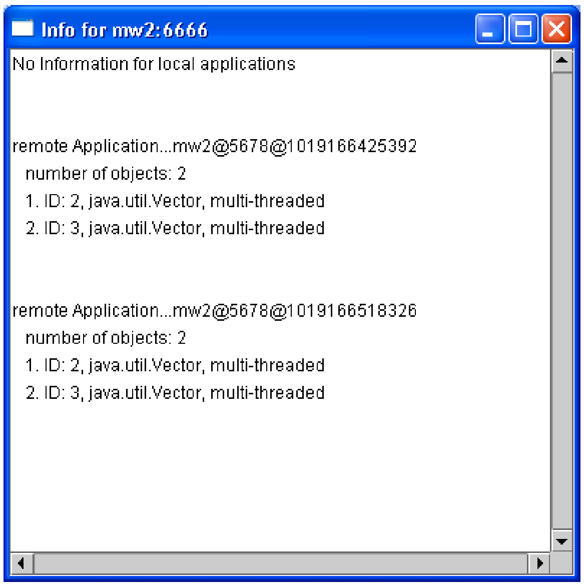
See “Threadpoolsize (PubOA)” under “Popup-Menu Entries for a NA in the Tree”.

Remove

Removes the NA from the list.

The Information Window

You can get Information about applications and objects residing on a NA by selecting “Show Info” from the NA-popup menu. A window will appear, displaying information about the application and objects residing on this NA. The following is an example window, how the information could look like.



```
No Information for local applications


remote Application...mw2@5678@1019166425392
  number of objects: 2
  1. ID: 2, java.util.Vector, multi-threaded
  2. ID: 3, java.util.Vector, multi-threaded


remote Application...mw2@5678@1019166518326
  number of objects: 2
  1. ID: 2, java.util.Vector, multi-threaded
  2. ID: 3, java.util.Vector, multi-threaded
```

This window displays information for an NA “mw2:6666”. There is no local application running on this NA, but there are two remote applications that are using this NA, having each two objects residing on it. Each application is unique in the system using an identifier consisting of the computer name and the port of the NA where the application is running, and a unique number within the NA.

Migrating Objects

You can migrate objects from one NA to another NA within the same application by dragging and dropping them from one NA information window to another one by mouse (point on the text line for the object to do this). You can drop the object anywhere in the destination window, if the same application is running on that NA, to object will be migrated.

Changing Threat Pool Size of the AppOA

You can view and change the thread pool size of the Application Object Agent (AppOA) in the information window by right-clicking on the local application text line (if there is a local application). The settings will take place immediately. (see “Documentation for Object Agent System” for information about thread pools)

How to create a new physical architecture

The following guide tells you how to design and create a physical architecture using the JS Shell.

1) If there is already a tree on the left side of the main window, remove all NA's from the tree by dragging and dropping them into the list on the right side of the main window. (you can not remove the root node called "Systemroots", because it is not a NA).
2) Add all NA's that you want to use in your physical architecture (see "Add NA") to the list.
3) Create your physical architecture by dragging NA's from the list to left side of the main window, creating a tree representing your architecture. You should only use NA's that are marked with a green tick, because an NA could be contacted there. You can also drag and drop nodes around within the tree.
4) You can set the level of some nodes explicitly, if you want to, using the "Set Level".
5) For each node in the tree that has children, you should define one of them as a backup manager (see "Toggle Backup Manager"). For backup managers you can set a backup rate different from the default value, if you want to (see "Set Backup Rate").
6) You can change the rate at which a NA checks its children to a value different from the default value, if you want to (see "Set Rate").
7) If the design fulfils your expectations, you can create the physical architecture by using "Make Configuration". If this was successful, the architecture has been created, and you can use it for you JavaSymphony applications
8) If you want, you can change some event properties and the thread pool size of some NA's. You can do this at any time (See "Event Properties" and "Threadpoolsize").
9) As soon as some applications are running on your physical architectures, you can watch them and migrate objects between NA's by using "Show Info".
10) If you plan to re-use your architecture, you should store it to a file ("Save Configuration")

3.) **The Object Agent System:** How to write an application for JavaSymphony

JavaSymphony Registry

It is necessary that you register your application at the JavaSymphony Runtime System (JRS). With the following statement you can register your application at the local PubOA (or the PubOA specified in
the jsapp.ini).

```
JSRegistry reg = new JSRegistry()
```

It's necessary that this statement is the first access to JavaSymphony in your application. And it's very important that you do this only once at the beginning of the main program.
If your application has finished, you should unregister to clean up memory and resources.

```
reg.unregister();
```

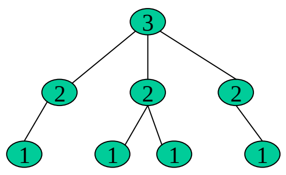
Managing Virtual Architectures

Every component in a Virtual Architecture is represented as an instance of the class **VA**. A Virtual Architecure is a tree built of instances of VA. Each VA has a defined level, that is the level in the tree. A VA with level 1 is also called a node. It is connected to a NetworkAgent of the Physical Architecture.

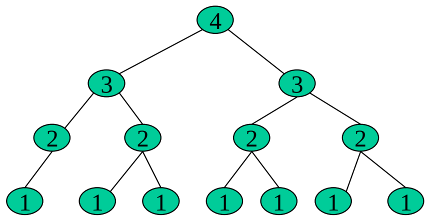
```
// creates a node; the JRS will look after an available NetworkAgent
VA v1 = new VA (1);

// creates a VA of level 2 with 5 nodes, i.e. a tree with a root and
// 5 leaves
VA v2 = new VA(2,5);
```

```
// creates a VA of level 3 with three VAs of level 2, the first with
// 1 node, the second with 2 nodes, and the third with 1 node
VA v3 = new VA(3,new int[] {1,2,1});
```



```
// creates a VA of level 4 with 2 VAs  of level 3, {1,2} and {2,2}
VA v4 = new VA(4,new int[][] {{1,2}{2,2}});
```



```
// creates a VA of level 5 with 2 VAs of level 4, {{2,3},{1},{7,6,1}}
// and {{1,3,5}{6,4}}
VA v5 = new VA(5,new int[][][] {{{2,3},{1},{7,6,1}},{{1,3,5}{6,4}}});
```

You can also create Virtual Architectures by creating “empty” VAs and adding one to another. You can only add a VA to another VA if the level of child is the level of the parent minus one.

```
VA v3 = new VA(3); // VA of level 3
VA v2 = new VA(2);
VA v1 = new VA(1);

// ok
v3.addVA(v2);
v2.addVA(v1);

// not ok
v3.addVA(v1);
v1.addVA(v2);
```

When you create a Virtual Architecture, you can also specify a set of constraints that the nodes have to meet. With the following piece of code you create a node that is connected to a Network Agent that’s running on a computer with idle time > 80% (if such a Network Agent is available).

```
JSConstraints constr = new JSConstraints();
constr.setConstraints(JSConstraints.C_CPU_IDLE, “>”, “80”);
VA node = new VA(1, constr) ;
```

You can also pass a JSConstraints object to the more complex constructors. The following piece of code creates a Virtual Architecture where all nodes meet the given constraints.

```
VA v = new VA(4,{{1,2}{2,2}}, constr);
```

To access to the components of a Virtual Architecture, you can use the following statements:

```
va.getVA(1) // returns the first child of va
va.getVA(1).getVA(3).getVA(2) // returns the second child of the
                                 third child of the first child of va
va.getVA(new int[] {1,3,2}); // equal to the example above
```

To get the number of VAs of a specified level of a Virtual architecture, use the nrVA-method:

```
int x = va.nrVA(2); // returns the number of VAs of level to of the
                       Virtual Architecture va
```

It’s also possible to get an enumeration of all VAs of a specified level.

```
VAEnum enum = va.enumerateVA(2);
while (enum.hasMoreVA()) {
      VA v = enum.nextVA();   // get the next VA of the enumeration
      int x = enum.getOriginalSize();  // get the original number of
                                        elements of the enumeration
                                         (the same like va.nrVA(2)
      int y = enum.getRemainingSize(); // get the remaining number
                                         of VAs of the enumeration
      // do what you want
}
```

Important: The methods nrVA and enumerateVA only work top down, so you can only use levels equal or lower to the level of the VA object you use.

To get the level of a VA, you can use this method:

```
int level = va.getLevel();
```

If you need the parent of a VA object, use the following statement:

```
VA parentVA = childVA.getPred(int level);
```

This will return the parent VA with the specified level (or null, if it doesn't exist or va.getLevel() >= level).

Reading or manipulating Virtual Architectures is potentially unsafe, because another thread could manipulate the Virtual Architecture, too. So if you want exclusive access to a Virtual Architecure, you can lock it. This also only works top down.

```
va.lock();   // locks the Virtual Architecture
// do something
va.unlock();  // unlocks it
```

It is very important to unlock the Virtual Architecure after you’ve finished you work. I would recommend to use the locking mechanism this way:

```
va.lock();
try {
      // do something
} finally {
      va.unlock();
}
```

You can check, if a Virtual Architecture meets its constraints (or a set of constraints you specify):

```
boolean  ok = va.constrHold() // true if all nodes in the Virtual
                                 Architecture meet their constraints
                                (if specified)
boolean ok = va.constrHold(constr); // true if all nodes in the
                                       Virtual Architecture meet the
                                       given  constraints
```

It’s also possible to get the values for a couple of system parameters of a VA. If the VA is a node, the actual value is returned, otherwise the average value of all nodes in the VA is returned (the possible parameters are specified as constants in the JSConstraints class.

```
public String getSysParamAsString(int param) // can only be
                                             // used for nodes
public int getSysParamAsInt(int param)
public float getSysParamAsFloat(int param)
```

To free a VA, use the following methods:

```
va.free();  // frees this VA and all children
va.free(2); // frees child with index 2
va.free(new int {1,3,2}); // short for
                          // va.getVA(1).getVA(3).getVA(2).free()
```

Loading the codebase

Before you can create objects on you nodes and invoke methods, you have to transfer the class you need. This is done using the JSCodebase class.

```
JSCodebase cb = new JSCodebase();
cb.add(“./classes/TestClass.class”); // add a class
cb.add("c:\\temp\\js\\examples.jar"); // add a package
cb.load(va); // transfer to all nodes of the Virtual Architecture va
JSCodebase.free(va); // deletes ALL codebases from the Virtual
                        Architecture va
cb.destroy(); // deletes locally created temporary files
```

Create JSObjects

Now you can start creating your objects on the nodes of your virtual architecture.
There are two kinds of objects in JavaSymphony, single-threaded and multi-threaded obejcts. Single-threaded objects have their own thread on their node, multi-threaded objects get a thread from a pool whenever they have to invoke a method. By using single-threaded objects, you prevent them from concurrent access, because all method invocations are processed sequentially. Multi-threaded objects can be used concurrently, because every method invocation just gets a thread from the pool. Be careful when you use single-threaded objects, because some operating systems limit the number of threads. To create an object on a node, you can use one of the following constructors.

```
// create a new object; the JRS will search all Virtual
// Architectures for the best node available (the JRS is not very
// smart, it just takes the node with the maximum idle time
public JSObject([boolean singleThreaded],
                String className,
                [Object[] conArgs])

// create a new object in the given VA if it meets the given
// constraints (if this is no node, the JRS looks for the best node
// in the VA that meets the given constraints)
public JSObject([boolean singleThreaded],
                String className,
                [Object[] conArgs],
                VA va,
                [JSConstraints constr])

// create an object; the JRS will search all Virtual Architectures
// for a node that meets the given constraints
public JSObject([boolean singleThreaded],
                String className,
                [Object[] conArgs],
                JSConstraints constr)
```

The parameters mean:

boolean singleThreaded – true for single-, false for multi-threaded objects
String classsName – class of the object you want to create

VA node – VA where the object should be created
JSConstraints constr – create the object on a node with the given constraints
Object[] conArgs – the parameters for the constructor of the object

[boolean singleThreaded] means that this parameter is optional; if not specified, a multi-threaded object will be created. [Object[] conArgs] is also optional; if not specified, the no-arg constructor will be used. [JSConstraints constr] is optional, if not specified, no constraints are used.

If you let the JRS search a suitable node, be aware that the JRS won't return nodes that are locked by another thread.

Once you've created an object, you can change its type (single- or multi-threaded) with the following statements:

```
JSObject obj = new JSObject(…);
obj.singleThreaded(); // make object single-threaded
obj.multiThreaded(); // make object multi-threaded
```

There is one more possibility to create a JSObject.

```
JSObject obj = JSObject.convertToJSObject(Object o,
                                          [boolean singleThreaded)
```

This creates a JSObject that points to the given object.

Invoking methods

After you've created your objects, you can invoke methods on them. There are three possibilities for method invocation in JavaSymphony, synchron, asynchron, and one-sided.

```
// synchron
public Object sinvoke(String methodName,
                      Object[] params,
                      [Class[] paramTypes]);

// asynchron
public ResultHandle ainvoke(String methodName,
                            Object[] params,
                            [Class[] paramTypes]);

// one-sided
public void sinvoke(String methodName,
                    Object[] params,
                    [Class[] paramTypes]);
```

The parameters mean:
String methodName – the name of the method that should be invoked
Object[] params – the parameters for the method
Class[] paramTypes – the types of the parameters (this argument is optional)

The methods are invoked using the reflection mechanism of Java. Because of this it's possible that the Java Virtual Machine will find more than one method for the given parameters. The JRS will invoke a method only if it finds exactly one method (by using reflection) that matches the desired method signature (you can help the JRS by specifying the types of the parameters in the Class array paramTypes), otherwise an exception is thrown.

Synchronous method invocation behaves like normal method invocation in Java. The thread is blocked until the method has finished. You have to cast the returning object to the Class you expect.

Asynchronous method invocation doesn't block the current thread, but returns a handle. You can ask the handle if the result it has already received the result. If you want to get the result from the handle, the current thread is blocked until the result has received.

```
JSObject obj = new JSObject(…);
ResultHandle handle = obj.ainvoke(…);
boolean finished = handle.isReady(); // true if the handle has
                                        already received the result
Object result =  handle.getResult(); // returns the result of the
                                        methods; blocks the current
                                        thread until the handle has
                                        received the result
```

One-sided method invocation behaves like the asynchronous one but doesn't return a result.

If you invoke a method on an object that is on a local node, i.e. on the same computer than the main application, for reasons of performance the method will be invoked in the same JVM as the application.
If you want a local node, you get get it with a static method of the class VA. The JRS will search all Virtual Architectures for a local node, if it doesn't find one, null will be returned (there will not be created a new node).

```
VA local = VA.getLocalNode();
```

You can pass JSObjects and VAs as parameters to method invocations (and also when creating JSObjects). You can do the same with Virtual Architectures and JSObjects in your methods of the remote objects as in your main application.

Migrating objects

It's possible to transfer an object from one node to another by using the migrate-methods of the JSObject class.

```
// migrate the object to a node determined by the JRS (if specified,
// depending on the given constraints
// if transfercodebase is not specified or specified and true, the
// codebase of the source node will be transferred to the
// target node
public boolean migrate([JSConstraints constr],
                       [boolean transferCodebase)

// same as above, but loads the given codebase to the target node
public boolean migrate([JSConstraints constr], JSCodebase cb)

// migrates the object to the given VA. If this is not a node,
// the JRS will look for the best node in the VA. It will also
// load the codebase of the source node to the target node
public boolean migrate(VA va)

// same as above ; if constraints are specified, the JRS will look
// for a VA that meets the given constraints; if a JSCodebase is
// specified, it will be loaded to the target node
public boolean migrate(VA va,
                       [JSConstraints constr], [JSCodebase cb])

// migrate the object to a node in the VA determined by the JRS
// (if specified, depending on the given constraints
// if transfercodebase is not specified or specified and true, the
// codebase of the source node will be transferred to the
// target node

public boolean migrate(VA va,
                       [JSConstraints constr],
                        boolean transferCodebase)
```

The default for transferCodebase is always true (if node JSCodebase is specified).

If you let the JRS search a suitable node, be aware that the JRS won't return nodes that are locked by another thread.

Persistent objects

JavaSymphony provides facilities to make objects persistent by saving and loading them from/to external storage, An object can only be stored when none of its methods are currently executed (which is checked by the JRS). The class js.oa.JSObject provides the following methods:

```
// store the object under a unique filename created by the JRS
// the filename is returned from the method
public String store()

// store the object under the specified filename
public void store(String fname);

// load the object and transfer it to the local node
public static JSObject load(String filename, boolean singleThreaded)

// load the object and transfer it to the specified node
public static JSObject load(String filename, VA va,
                              boolean singleThreaded)
```

JavaSymphony events

JavaSymphony provides a mechansim to send events across a virttual architecture and physical architecture of JavaSymphony applications.
To use the event mechanism, you always need two parts: an event consumer and an event producer.
An event consumer can register for specific events and can specifiy teh sources of the events. It also specifies a JavaSymphony object and a method that will be called to notify about events.
An event producer can produce specific events and can restrict the receivers of the event.

Creating an event producer is quite easy, you just have to create an instance of the class JSEventProducer:

```
public JSEventProducer(Object source, int eventType)

public JSEventProducer(Object source, int eventType, int destType)

public JSEventProducer(Object source, int eventType, int destType,
                       Object[] destEntity)

public JSEventProducer(Object source, int eventType, int destType,
                       Object destEntity)
```

The parameters have the following meanings:

Object source – this is the source of event. You can use an instance of JSObject or of VA.
If you pass an instance of another object it will be converted to a JSObject using JSObject.convertToJSObject.
IMPORTANT: don’t use any other object of the JavaSymphony API like JSRegistry, AppOA, etc. This can lead to unexpected and unpredictable behaviour of the JRS and your application.

int eventType – the type of the event (explained later)

int destType – type of the objects that may receive the event (explained later)

Object[] destEntitiy – list of objects that may receive the event, corresponding to destType (explained later)

Object destEntitiy – object that may receive the event, corresponding to destType (explained later)

Default values for parameters that must not be specified are (the meaning of the constants is explained later):

destType: C_ALL_EVENTS
destEntity: null (must not be specified for C_ALL_EVENTS)

To produce an event, you just invoke one of these methods of JSEventProducer:

```
public void produceEvent()
public void produceEvent(Object[] args)
public void produceEvent(Object args)

Object[] args - arguments for the method that will be called to
                handle the event
Object arg - argument for the method that will be called to handle
             the event
```

Example:

```
JSObject obj = new JSObject("TestClass");
JSEventProducer prod = new JSEventProducer(obj, C_USER_TYPE);
pod.produceEvent();
```

Creating an event consumer is very similar to the creation of an event producer. Just create an instance of the class JSEventConsumer:

```
public JSEventConsumer(Object destination, int eventType,
                       int sourceType, Object[] sourceEntity,
                       String methodName)

public JSEventConsumer(Object destination, int eventType,
                       int sourceType, Object sourceEntity,
                       String methodName)

public JSEventConsumer(Object destination, int eventType,
                       int sourceType, String methodName)

public JSEventConsumer(Object destination, int eventType,
                       String methodName)
```

Object destination – destination of the event, i.e. the object that will be notified when an event that matches the conditions occured. You can use an instance of JSObject. If you pass an instance of another object it will be converted to a JSObject using JSObject.convertToJSObject.
IMPORTANT: don't use any other object of the JavaSymphony API like JSRegistry, AppOA, etc. This can lead to unexpected and unpredictable behaviour of the JRS and your application.
int eventType – the type of the event (explained later). A consumer will only be notified if an has the same eventType as the consumer
int sourceType – type of the objects that can be the source of an event for the consumer (explained later).

Object[] sourceEntitiy – list of objects that may be the source of anevent, corresponding to sourceType (explained later)
Object sourceEntitiy – object that may be the source of an event, corresponding to sourceType (explained later)

String methodName – name of the method that will be called if a matching event occurs. Be sure that the destination object provides this method with the expected signature

Default values for parameters that must not be specified are (the meaning of the constants is explained later):
sourceType: C_ALL_EVENTS
sourceEntity: null (must not be specified for C_ALL_EVENTS)

To register a consumer, just call teh following method of JSEventConsumer:

```
public void register()
```

If you are not more interested in an event, you can unregister your consumer:

```
public void unregister()
```

Example:

```
JSObject obj = new JSObject(“TestClass”);
JSEventConsumer cons = new JSEventConsumer(obj, C_ALL_EVENTS,
                                            “handleEvent”);
cons.register();
```

In this example, the class TestClass must specifiy a method “handleEvent”, because this will be called when an event occurs.

There is a number of constants defined in JSCcnstants that can be used for eventType. You can only use the first one for your event producer, events with the other event types are produced by the JRS.

- C_USER_TYPE - your producers and the corresponding consumers should use C_USER_TYPE+1, C_USER_TYPE+2, C_USER_TYPE+3, etc.

The following eventTypes you can only use for your consumers. They are produced by the JRS and can be enabled/disabled with the JavaSymphony Shell. There are also specific parameters that will be passed to the specified method of the consumer’s destination object.

- C_SYSTEM_EVENT – system event (explained later); parameter: VA source, Integer constrCondition
- C_APP_REGISTERED – produced when an application is registered; parameter: String appId (id of the application)
- C_APP_UNREGISTERED – produced when an exception is unregistered; parameter: String appId (id of the application)
- C_NEW_VA_AVAILABLE – produced when a new level-1-VA is available for the application; parameter: VA (the new VA)
- C_VA_UNAVAILABLE – produced when a new VA becomes unavailable for the application; parameter: VA
- C_VA_LOCKED – produced when a VA becomes locked; parameter: VA
- C_VA_UNLOCKED – produced when a VA becomes unlocked; parameter: VA
- C_OBJECT_LOCKED – produced when a JSObject becomes locked; parameter: JSObject
- C_OBJECT_UNLOCKED – produced when a JSObject becomes locked; parameter: JSObject
- C_OBJECT_MIGRATED – produced when a JSObject has been migrated; parameters: JSObject, VA source, VA destination
- C_EVENT_REGISTERED – produced when an EventConsumer is registered: parameter: JSEventConsumer
- C_EVENT_UNREGISTERED – produced when an EventConsumer is unregistered: parameter: JSEventConsumer
- C_VA_USED – produced when a level-1-VA is attached to an application; parameters: VA, String appId
- C_VA_RELEASED – produced when a level-1-VA is released from an application; parameters: VA, String appId
- C_NEW_OBJECT – produced when a new JSObject is created; paramater: JSObject, VA location
- C_OBJECT_RELEASED – produced when a new JSObject is created; paramater: JSObject, VA location

The possible values for sourceType and destType are also defined in JSConstants (it’s also explained which values have to be used for destEntitiy and sourceEntitiy).

for JSEventConsumer:
- C_JSOBJECT_EVENT – accepted events will be produced only by the specific JSObject (sourceEntitiy has to be an instance of JSObject)
- C_VA_EVENT – accepted events will be produced only by the specific VA (sourceEntitiy has to be an instance of VA)
- C_LIST_JSOBJECT_EVENT – accepted events will be produced only by a list of JSObjects (sourceEntitiy has to be an array of JSObjects)
- C_LIST_VA_EVENT – accepted events will be produced only by a list of Vas (sourceEntitiy has to be an array of Vas)
- C_ONLY_APP – accepted events will be produced only by the objects within the same application (sourceEntitiy can be null or omitted)
- C_ALL_EVENTS – accepted events will be produced by any party (sourceEntitiy can be null or omitted)

for JSEventProducer:

- C_JSOBJECT_EVENT – produced events will be sent only to the specific JSObject (destEntity das to be an instance of JSObject)
- C_VA_EVENT – produced events will be sent only to the specific VA (destEntity das to be an instance of VA)
- C_LIST_JSOBJECT_EVENT – produced events will be sent only to a list of JSObjects (destEntitiy has to be an array of JSObjects)
- C_LIST_VA_EVENT – produced events will be sent only to a list of VAs (destEntitiy has to be an array of VAs)
- C_ONLY_APP – produced events will only be sent to the objects within the same application
- C_ALL_EVENTS – produced events will be sent to any party

System events are a special type of events. They are generated in a specific interval by the NetworkAgent and can be consumed using the JSSystemEventConsumer class. It is very similar to JSEventConsumer, bu there is only one constructor.

```
public JSSystemEventConsumer(Object destination,
                             int sourceType,
                             String methodName,
                             JSConstraints constr,
                             int constrCondition)
```

The parameters have the same meaning as for JSEventConsumer. You have to specifiy a condition that will be evaluated by comparing the specified constraints to the current (and previous) system properties. Only if this condition evaluates to true, the event will bve received.
These are the two additional parameters:

JSConstraints constr – constraints that will be compared to the system properties when the event is Produced
int constrCondition – condition that has to be met to receive the system event

In JSConstants are three defined values for constrCondition:

- JS_CONSTRAINTS_CHANGE – event will be received only when the boolen value of constrHold is changed
- JS_CONSTRAINTS_HOLD – event will be received only when the boolean value of constrHold is changed to true
- JS_CONSTRAINTS_NOT_HOLD - event will be received only when the boolean value of constrHold is changed to false

ini-files

Configuration of the application: resources/jsapp.ini (relative to the directory where the application is started)

```
# host where the registry will look after a PubOA
host = localhost

# port on which this PubOA is registered
rmi_port = 5678

# number of local JobHandler (Threads that process the method
                invocation for multi-threaded objects)
number_of_job_handler = 3

# debug = 0: no debug messages
debug = 1
```