

This article is archived because it is not considered relevant for third-party developers creating commercial solutions today. The article is believed to be still valid for the original topic scope.



Contents

- [1 Introduction](#)
- [2 Design principles](#)
- [3 WidSets Scripting Language \(WSL\) concepts](#)
- [4 Syntax](#)
 - ◆ [4.1 What's similar](#)
 - ◆ [4.2 What's missing](#)
 - ◆ [4.3 What's different](#)
 - ◆ [4.4 What's new](#)
 - ◆ [4.5 Keywords](#)
- [5 Supported types](#)
- [6 Casting](#)
- [7 Method resolution](#)
- [8 Tuples](#)
- [9 Structs](#)
- [10 Widget functions](#)
 - ◆ [10.1 Interfaces to handle the widget's lifecycle](#)
 - ◆ [10.2 Interfaces that are associated with the widget's menu control](#)
 - ◆ [10.3 Interfaces responding to user actions](#)
 - ◆ [10.4 View elements construction interface](#)
 - ◆ [10.5 Callback function on timer events](#)
 - ◆ [10.6 Callback functions on server communication events](#)
- [11 Inner functions & closures](#)
- [12 Language rules](#)
- [13 See also](#)

Introduction

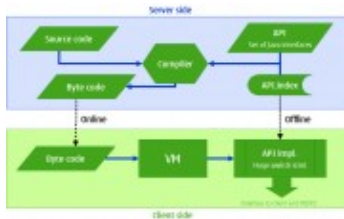
- A strongly typed widget ?scripting? language
- Needed due to the lack of ClassLoaders on mobile information device profile (MIPD) 2.0 platform. Without ClassLoader it is impossible to import new .class files (=functionality) into already installed application.
- Needed to minimize the use of resources
- Enhances the widget programming experience
- It has a look and feel of Java programming language and should be immediately familiar to any Java developer

You can find the online WidSets API documentation [here](#).

Design principles

- Low memory footprint
- Low codesize footprint
- High performance
- Minimum or no use of wrapper objects
- Familiar syntax
- Easily extendable/replaceable API
- Applicable for other domains as well, for instance, server side

WidSets Scripting Language (WSL) concepts



See the image for the idea behind the WSL concept.

Syntax

What's similar

Comments

```
/* comment here*/
// another comment
```

Literals

```
; // line ends
?lorem ipsum? /* string */
?\n? /* character, produces int */
123456 /* integer */
0777777 /* octal */
0xcafebabe /* hexa */
0b101010101 /* binary */
```

Statements

```
if else while do for switch
case default return break continue
foreach [Java5]
```

Operators

```

+   -   *   /   %   &   |   ^   <<   >>
+=  -=  *=  /=  %=  &=  |=  ^=  <<=  >>=
<   <=  >   >=  ==  !=
!   &&  ||
++  --  ~
instanceof  new
?:

```

What's missing

No access qualifiers

```

public protected private
static synchronized
abstract transient volatile
native strictfp

```

- Except ?final?, it can be used for both Widget- and local variables

No user defined classes

- The whole script is like a class
- ?Member? variables live for the duration of client run
- Initializers are executed just before invoking first widget callback

No import

- All API declarations are directly accessible

No exception handling

- When needed error handling can be done with return codes
- Any exception will terminate run and generate error

No floating points

- Not available in MIDP 2.0

No language level support for Java-style array[] syntax

- Operator overloading allows close enough emulation

```

ByteArray init_rc4(ByteArray key)
{
  ByteArray perm = new ByteArray(256);
  for(int i=0; i<256; i++) {
    perm[i] = i;
  }
  int j = 0;
  for(int i=0; i<256; i++) {
    j = (j + perm[i] + key[i % key.length()]) & 0xff;

```

What's similar

```
    perm[i], perm[j] = perm[j], perm[i];
}
return perm;
}
```

What's different

Flow-control statements *break* and *continue* do not support labels

- Rarely needed.

Limited set of primitive types:

```
boolean char int long
```

Local variables must be introduced with an assignment

- Like Java, variables exists on the scope they are declared

```
void test()
{
    int j; /* error, missing assignment */
    {
        int c = 0;
    }
    {
        int d = c; /* error, 'c' is not visible here */
    }
}
```

Widget variables can be left unassigned.

- Object references will default to null, numbers to 0 and boolean to false

```
class
{
    Object foo;

    void test()
    {
        if (foo == null) {
            printf("It's null alright");
        }
    }
}
```

What's new

Shell style comments

```
# crazy think
```

Binary literals

What's missing

```
int i = 0b101010101;
```

XOR operator: ^^

- Oh well, it's actually equal to !=
- But looks l33t

Constants ? with primitive and String types

```
const int MAX = 100;  
const String PASSWORD = "n00Bh2xöR";
```

Typedef - type definition

- Define your own function prototypes
- Like a single-function interface (Java) that can be declared as function parameter or return value
- WSL API has predefined Callback prototypes for event handling, which can be used from WSL code too

```
typedef int Arithmetic(int a, int b);  
  
int add(final int a, final int b)  
{  
    return a+b;  
}  
  
void test()  
{  
    Arithmetic a = add;  
    int c = a(10, 20); // -> not needed  
}
```

Keywords

```
boolean break  
case class const continue  
default do  
else  
false for foreach  
if instanceof int  
long  
new null  
return  
struct switch  
true  
void  
while
```

Supported types

Primitives

```
boolean /* true/false */
```

```
char    /* character      */
int     /* signed 32 bit   */
long    /* signed 64 bit      */
```

Boolean is internally handled as int.

Any classes declared on API

- Handling of non-primitive types is similar to Java
- == and != operators compare the identity of objects
- Method boolean Object.equals(Object) compares the actual object contents

Casting

Implicit casting occurs if possible.

Any primitive converts to any other primitive

- Even if it would be narrowing (e.g. long ? int)

Explicit casting is different

```
Object f = ...;
Gizmo g = Gizmo(f); /* whoa, looks like an function call */
boolean b = ...;
int zeroOrOne = int(b);
```

Benefit of this syntax is reduced noise and chaining:

```
Object f = ...;
int length = String(f).length();
/* ugly java style: ((String)f).length(); */
```

On assignments, super type is promoted to sub type automatically

```
Object a = ...;
String b = a; /* yep, that?s right ! */
```

Contrary to Java, the casting is purely a compile time feature

- For performance reasons, it is not reflected on byte code.
- All casts always succeed but the problems may raise later:

```
List a = new List();
Object b = a;
String c = String(b);
int i = c.length(); /* Croak! ClassCastException */
```

NullPointerException has similar semantics as well.

On the previous example, this is what happens

```
/* byte code */
aload $1 /* b */
astore $2 /* c */
pop
aload $2 /* c */
invoke_native1 #40 /* int String.length() */

/* jump to Java */
public int invoke(int sp, int[] istack, Object[] astack, long[] lstack, int method) {
    switch(method) {
        case 40:
            /* int String.length() */
            istack[sp] = ((String)astack[sp]).length(); /* Croak! ClassCastException */
            return 0;
        }
    }
}
```

Method resolution

When selecting target method for invocation, all arguments are checked.

There are four match categories:

- Exact match
 - ◆ Source and target are equal.
- Close match
 - ◆ Source and target are primitives.
 - ◆ Target is super type of source.
- Loose match
 - ◆ Source is null and target is class.
 - ◆ Source is any and target is a String.
 - ◆ Source is string and target is primitive.
 - ◆ There exists cast operator from source to target.
- No match

Result of argument list is equal to the ?worst? match. If there are multiple matches on same category, ambiguity is presented as to user. However, the resolution is quite natural - only some rare situations require explicit casts

Tuples

- Functions may return multiple values - tuples.
- Returned tuples may be embedded to function call arguments and return values.
- Opportunity to use stack instead of heap for temporary containers.
- Tuples can be used on both sides of assignment, for example, to swap values.

```
String, int getNameAndAge() {
    return "John", 27;
}

int, int sort(int a, int b) {
    if (a<b) {
```

```
    return a, b;
} else {
    return b, a;
}
}

void test() {
    int age, String name = getNameAndAge();
    printf("name=%s, name=%i", getNameAndAge());
    int x, int y = sort(random(100), random(100));
    printf("x=%i, y=%i", x, y);
    x, y = y, x;
}
```

Structs

- Behave like the Java-classes with public fields and no methods.
- **instanceof** does not work due to lack of RTTI

```
struct Celestial
{
    String name;
    int diameter;
    List satellites;
}

Celestial newBody(String name, int diameter)
{
    return newBody(name, diameter, new List());
}

Celestial newBody(String name, int diameter, List satellites)
{
    Celestial c = new Celestial()
    c.name = name;
    c.diameter = diameter;
    c.satellites = satellites;
    return c;
}

void test()
{
    Celestial solarSystem = newBody("Sun", 696000,
        new List().add(newBody("Mercury", 2439))
            .add(newBody("Venus", 6051))
            .add(newBody("Earth", 6371)));
}
```

Widget functions

Widget functions can be divided into several categories based on their functionality.

Interfaces to handle the widget's lifecycle

```
void startWidget();
void stopWidget();
```

When the WidSets mobile client is opened, the `startWidget()` function is called for all widgets installed on the client's dashboard. In most cases, you should implement this function to create the widget's minimized view to be displayed on the WidSets dashboard.

The `stopWidget()` function is called whenever a widget is reloaded or removed from the dashboard or when the WidSets client is terminated. You should implement this function to release system resources (if any) before the client is terminated.

```
Shell openWidget();
void closeWidget();
```

The `openWidget()` function is called when the user selects a widget from dashboard. This is when the widget is also opened to the maximized mode. Most widgets implement the maximized mode. Within this function a Shell for using the views is typically created. The views can be either defined in the `widget.xml` file or created dynamically.

The `closeWidget()` function is launched when the maximized mode is exited by either with `popShell()` or `slideOut()`, the last widget-created shell, or when the widget is stopped for any reason. Here you might usually want to, for example, cancel a timer or set it to "tick" less frequently.

Interfaces that are associated with the widget's menu control

```
MenuItem getSoftKey(Shell shell, Component focused, int key);
Menu getMenu(Shell shell, Component focused);
```

When the user presses a softkey of a mobile device (normally a softkey is associated to the widget's menu), WidSets will call the `getSoftKey()` to inform the widget which softkey was just clicked. You must implement this function to detect the softkey and tell WidSets what to do with the event. The `getMenu()` function is called when a softkey is designed to associate with an open menu.

Interfaces responding to user actions

```
void actionPerformed(Shell shell, Component source, int action);
boolean keyAction(Component source, int op, int code);
```

The `actionPerformed()` function is called when a key-pressed action on a softkey is detected, and the `keyAction()` function is called when the user clicks on an alphanumeric key (including the navigation key). These functions can be implemented to detect any actions done by the user and to process them accordingly. Remember that both `actionPerformed()` and `keyAction()` are system callback functions, and they should never be called directly.

View elements construction interface

```
Flow createView(String name, Object context)
Component createElement(String viewId, String elementId,
Style style, Object context);
```

The `createView()` function is used for creating a view for the widget. A view can be a minimized view or a maximized view depending on the widget's status.

Widget's UI components can be constructed within the `createElement()` function. This is a callback function and it gets called as a result every time the `createView()` function is called.

Callback function on timer events

```
void timerEvent(Timer timer);
```

You can set up timers that notify your widget after a specified time or continuously. Timers are created using `schedule(...)` functions. These timers will then call the `timerEvent(i.e. Timer timer)` function of your script, or a separate `TimerCallback` function, if defined.

To release the timer resource, you need to call the `cancel()` function for any `Timer` objects created. This should be done when the timer is not needed anymore, or when a widget is terminated via the `stopWidget()` function.

Callback functions on server communication events

```
void onSuccess(Object state, Value returnValue);
void onFailure(Object state, String errorMessage);
```

Server-side services can be invoked by using the `call(Object state, String service, Value argument)` function. Responses from a server (as the result of the `call()` function) are dispatched via `onSuccess()` or `onFailure()` callback function indicating that the server call was successful or failed, respectively.

In case of multiple server-side service calls, the state parameter in these callback functions can be used to identify which call the response belongs to.

If you are familiar with Java programming language or any other programming language, working with the WidSets Scripting Language is not a difficult task at all. Consider how to implement a simple function which calculates and return the sum of two numbers in WSL:

```
int calculateSum(int number1, int number2) {
    return (number1 + number2);
}
```

Inner functions & closures

Inner functions are useful when implementing callbacks (like anonymous classes in java).

- You have access to enclosing function's variables, even to null references (you can set them from inner function).
- Inner functions **cannot** contain inner functions. However one function can contain multiple inner functions.

Typical example is calling server-side service and provide success- and failure-callbacks.

```
void getPhoneNumber(String name)
{
    Value arg = [
        "url" => "http://www.foo.com/phone",
        "params" => ["name" => name]
    ];

    call(null, "myService", "get", arg, ok, nok);

    void ok(Object state, Value ret)
    {
        setBubble(null, name+"'s number is "+String(ret));
    }

    void nok(Object state, String error)
    {
        setBubble(null, "Could not find number for "+name);
    }
}
```

Language rules

This section describes the fundamental rules of WidSets Scripting Language.

All script functions and variables MUST be implemented within the ONLY class of a widget:

```
class
{
    int globalVar = 100;

    void firstFunction()
    {
        int localVar = 0;
    }
}
```

However, there can be inner functions which can make things easy to do while keeping code elegant:

```
Flow createAddressView(String name, String addr, String city)
{
    Flow flow = new Flow(getStyle("address.view"));

    add("Name", name);
    add("Address", addr);
}
```

WidSets_Scripting_Language

```
add("City", city);

void add(String name, String value)
{
    String text = format("%s: %s?", name, value);
    Label label = new Label(?address.field?), text);
    label.setPreferredWidth(-100);
    label.setFlags(VISIBLE|LINEFEED);
    flow.add(label);
}

return flow;
}
```

Inner functions can access variables defined in the enclosing function and also invoke other inner functions. Unlike in Java, you don't have to mark variables as final in order to access them from inner functions.

All variables must be initialized and they exist within the scope where they are declared.

```
void firstFunction()
{
    int j; /* error, missing assignment */
    {
        int c = 0;
    }
    {
        int d = c; /* error, ?c? is not visible here */
    }
}
```

Explicit typecasting

```
boolean flag = false;
Text displayText;

void someFunction()
{
    int zeroOrOne = int(flag); /* casting boolean to int type */
    ...
    String str = String(zeroOrOne); /* casting int to string type */
    displayText.setText(str);
}
```

See also

- [WidSets SDK](#)
- [Configuring an editor for syntax highlighting](#)
- [WidSets Client](#)
- [Widget examples](#)
 - ◆ [WidClock](#)
 - ◆ [Memory Game](#)
 - ◆ [Filter test](#)
 - ◆ [Hello World](#)

◆ UITest