Smalltalk for Java/JavaScript Guys

Introduction

Java and JavaScript inherited a lot of ideas and mechanisms from Smalltalk. Especially JavaScript has a lot in common with Smalltalk (actually, semantically, it is much nearer to Smalltalk, than to Java). Don't be fooled by syntax - it's the semantic which counts. This document shows some of them.

Please try to understand them.
It may even be interesting to read, if you are a smalltalker !

Syntax - So you can read Smalltalk Code

First some info on syntax. It is often stated, that the Smalltalk syntax is strange, weird and hard to read. Actually the opposite is true - it is **you** being used and trained to the C-like syntax. First of all, Smalltalk is older, so never blame Smalltalk for not adapting the syntax of the reset of the world...

Variables

Smalltalk variables are actually names for a binding of a name to a value. The binding consists of an association of a name (a so called symbol) to an object (the so called value). Technically, this is implemented by a pointer: you can also think of every name being "a pointer to some object".

The variable is untyped - it can hold a reference to any object. However, the object itself "knows" what it is. It knows the class of which it is an instance. Smalltalk is a dynamic strong typed language, in contrast to e.g. C, which is a static weak typed language.
This is very similar to how JavaScript deals with variables and types.

Local Variables

Within a method (which corresponds to a "virtual function") or a Smalltalk block (which corresponds to an inner, anonymous function), local variables are declared by placing them inside vertical bars. Thus, the JavaScript code:

```
var foo, bar, baz;
var a, b, c;
```

is equivalent to:

```
|foo bar baz a b c|
```

Variable declarations must be placed at the very beginning of a scope (i.e. method or block). In contrast to JavaScript, where variable declarations can also occur later in a function However, in JavaScript, the scope is also "the whole scope", so some coding guidelines prefer to also force programmers to put those declarations at the beginning, to not confuse readers of the code.

The Current Receiver: "self" vs. "this"

As methods are only executed as the result a message send to some object instance, there is always a "current receiver object". This is the object to which the message was sent, and it is always an instance of some class.

In Smalltalk, the pseudo variable "*self*" refers to this object. In Java/JavaScript, this is called "*this*".

As Smalltalk classes are also instances of some (meta-) class, the above also holds if a class method is called for (i.e. if a message is sent to a class object). There, "self" refers to the receiving class.

There is no corresponance to this in Java. Even if static functions might look like similar to Smalltalk's class methods, they are not: in Smalltalk, class methods can be redefined in subclasses just like other ordinary instance methods, and the class method sees the current receiver as "self".

Inside a block (the "inner function"), "self" refers to the receiver of the method in which the block was defined. Thus blocks are actually closures and behave much like inner functions in JavaScript (or lambdas in new Java).

Statements

In Smalltalk, statements are separated by periods (much like english sentences are). In Java and JavaScript, they are delimited by semicolon. Notice the difference: "separating" vs. "delimiting". In Smalltalk, the last statement within a method or block does not need a period (because there is nothing to separate it from), whereas in Java/JavaScript, every statement needs its semicolon, except for the brace-block construct, which does not. In Smalltalk, really EVERY non-last statement needs its period. It does not hurt, to add a period after the last statement in Smalltalk (just think of the rest as being an "empty statement".

So, the javaScript:

```
<statement1> ;
<statement2> ;
<statement3> ;
```

looks in Smalltalk like:

```
<statement1> .
<statement2> .
<statement3> .
```

with the last period being optional.

The semicolon has a meaning in Smalltalk, which is described below.

Assignments

All three languages are not functional languages (although a functional programming style is definitely possible and often a good choice, the major paradigma is an "imperative OO style"). So, all of them offer the assignment.

In Smalltalk, to avoid confusion with the equality comparison operators, assignment is written as ":=",

whereas in Java/JavaScript, you would simply write "=".
So:

```
    var a, b, c;

    ...
    a = 5;
    b = 4;
    c = a;
    ...
```

becomes in Smalltalk:

```
    |a b c|
    ...
    a := 5.
    b := 4.
    c := a.
    ...
```

This is actually quite nice, if you think of how often you mistyped "=" instead of "==" in C/Java/JavaScript. Because this happens so often, experienced C/Java programmers often write "if (const == expr)" instead of "if (expr == const)" to ensure that this kind of typo is caught by the compiler.

Returning a Value from a Mehod

To return a value from a method, use "^ expression".
Thus the JavaScript code:

```
    ...
    return ( <someExpression> );
    ...
```

is written in Smalltalk as:

```
    ...
    ^ <someExpression>.
    ...
```

In Smalltalk, every method invocation returns a value. If you don't return anything via the above return statement (i.e. if execution reaches the end of a method's code), the receiver object "self" is returned implicitly.

In Smalltalk, a method return always returns a value from the currently executing method. Even if that return is inside an inner block, which is described below.

Message Sends (Virtual Function Calls)

The term "message send" is called "virtual function call" in many other languages. This may have historic reasons, one being probably, that Java inherits a lot of nomenclature from the C/C++ world, for obvious reasons.

Every such message send invokes a function (called "method" in Smalltalk) of the "receiving" object; it is the receiver's class, which determines which method will actually be executed. That is almost the same behavior for all three languages.

The syntax is slightly different:

Unary Messages (no argument passed)

For message sends (calls) without argument, simply write the name of the message (the function name), after the receiver:

```
    receiver.foo();
```

becomes in Smalltalk:

```
    receiver foo.
```

(without parenthesis).
As a more concrete example, consider:

```
    var a, b, c;

    ...
    a = b.abs();
    c = a.foo();
    b = a.foo().bar().baz();
    ...
```

which becomes in Smalltalk:

```
    |a b c|
    ...
    a := b abs.
    c := a foo.
    b := a foo bar baz.
    ...
```

Such messages (function calls) are called "unary messages", and the corresponding methods, which implement the response to such a call are called "unary methods". "unary" means, that there is no argument passed with the call.

Keyword Messages (arguments passed)

If arguments are to be passed, Smalltalk uses a syntax which combines argument count, names and position information in the name of the method (function). Instead of placing the call-arguments into a parenthized list after the name of the called function, the Smalltalk code slices those right into the name, each separated by a colon. Because these fragments look like calls with named parameters, they are called keyword messages.

The name of that function is actually the concatenation of multiple parts, including the colon(s), to distinguish it from the above unary messages. In Smalltalk, every function which expects an argument must have colons in its name (one for each argument). Thus, the number and position of the arguments is fixed and defined by the function's name. So, a call to a "fooBar" function, with 2 arguments,

```
    var a, b, c;

    ...
    a = b.fooBar(1, 2);
    c = a.fooBarBaz(1, 2, a);
    ...
```

becomes in Smalltalk:

```
    |a b c|
    ...
    a := b foo:1 bar:2.
    c := a foo:1 bar:2 baz:a.
    ...
```

There is no direct 1-to-1 mapping of names possible: the colon is not a valid character in a name in Java/JavaScript.

Notice, that it is possible in JavaScript, to call a function with a wrong number of arguments:

```
    var a, b, c;

    ...
    a = b.fooBar(1);
    c = b.fooBar(1, 2, 3);
    ...
```

in Java, this is cought by the compiler, which will bark at you. In Smalltalk, you cannot use the same name for 1 or 3 arguments; one of the names must be "fooBar:", the other something like "foo:bar:anyThingElse:" because there are three arguments.

Sometimes, especially when code is generated automatically from Java/JavaScript like languages, methods use a single underscore character to separate arguments, as in:

```
    x.someGeneratedName:arg1 _:arg2 _:arg3.
```

But still: do not forget that "foo:" and "foo:_:" and "foo:_:_:" are three different names for three different methods.

Also notice that in Smalltalk, "foo:bar:" is a different message name than "bar:foo:", and that these will invoke different methods. There is no "automatic reordering" of message arguments and no "automatic renaming" of message names.

Binary Messages

Although Smalltalk is almost minimalistic in its syntax, and the above keyword messages would (semantically) suffice for all needs, the designers of the Smalltalk language thought that arithmetic messages look quite hard to read, if written like:

```
    (x add:1) mul:5.
```

For this, a special syntax for binary messages like +, -, * etc. is provided, which allows for those special characters to be written like infix operators. However, as these are still messages (virtual function calls), the language does not imply any meaning into those.

Especially, the language does not imply any arithmetic meaning and the compiler treats all such messages the same. Therefore, no precedence or grouping as in Java/Javascript is associated to those. They are simply evaluated from left to right.
That may make mathematical expressions hard to read, as you have to use parenthesis for grouping (and you often should, even in case the left-to-right order is the desired one).

So, the Java expression:

```
a + b * 5 + 4
```

MUST be written as:

```
a + (b * 5) + 4
```

in Smalltalk. Otherwise, the expression would simply be evaluated left to right, giving a wrong answer.

This may be a bit confusing or annoying for beginners. Experienced Smalltalkers use parenthesis and don't even think about it. Actually, making the evaluation order explicit using parenthesis is considered good style anyway - even in C, Java or JavaScript: you don't have to think if the shift operator has precedence over a comparison or not.

Notice that not only "+", "-", "*" and the other arithmetic operators are allowed as binary message names. Almost any other non-letter can be used, and even multi-character combinations are allowed (the original Smalltalk-80 version did not explicitly define a limit on the number of characters, but actually limited this to a max. of 2. Modern Smalltalk implementations usually allow up to 3).

Now you understand the meaning of the comma-operator ",": it is a binary message implemented in many collection classes which creates a concatenation of the receiver and the argument. So `'hello','world'` creates a new string containing 'helloworld', and because it groups left to right, you can create longer strings as in `'hello',' ',(OperatingSystem getLoginName).`

Semicolon - Sending Multiple Messages to the Same Receiver

Often, especially when initializing an object, or asking it to perform a sequence of operations, you want to send multiple messages to the same object. For example,

```
var t = new Executor;

t doThis();
t doThat(someArg);
t doSomethingElseWith(arg2, arg3);
...
```

this looks of course similar in Smalltalk:

```
|t|

t := Executor new.
t doThis.
t doThat:someArg.
t doSomethingElse:arg2 with:arg3.
...
```

however, in the above, there is a sequence of messages being sent to the same object, and the variable t is only needed for this. In Smalltalk, the semicolon (;) means: "send the following message to the same receiver". And the above code can also be written as:

```
(Executor new)
    doThis;
    doThat:someArg;
    doSomethingElse:arg2 with:arg3.
...
```

i.e. you don't need the extra temporary variable, and also not to write it multiple times. This is purely syntactic sugar, not adding any new semantic feature. But it is useful in some situations (as seen below, in the exception handler set example).

Such constructs are called "*cascaded messages*", and the value if the cascade is the value returned by the last cascaded message send. So you have to be a little careful, when assigning the value of a cascade, as in:

```
e := (Executor new)
        doThis;
        doThat:someArg;
        doSomethingElse:arg2 with:arg3.
...
```

because afterwards, the returned value from "`doSomethingElse:with:`" is assigned the variable "e". But often we do not want to depend on what this returns - it could be something else but the receiver object. To ensure that the original receiver value gets assigned, add a "`yourself`" as the last message of the cascade:

```
e := (Executor new)
        doThis;
        doThat:someArg;
        doSomethingElse:arg2 with:arg3;
        yourself
...
```

This little trick works because "`yourself`" is implemented in the Object class (i.e. every other object understands it), and is quaranteed to return itself. Actually, the "`yourself`" message is implemented there for this very reason.

Anonymous (inner) Functions

One of the biggest differences lies in the use of anonymous functions (called "*Block*" in Smalltalk). These play a minor role in Java and are occasionally used in JavaScript (as callbacks), but are a major program building element in Smalltalk. A Smalltalk block consists of a number of statements (separated by period), written in brackets, as in:

```
myBlock := [ a foo. b bar ].
```

which corresponds to the JavaScript code:

```
myFunc = function () { a.foo(); return b.bar(); };
```

as you can see, Smalltalk is a bit shorter and less confusing in its syntax. By the way, there is nothing comparable to this in Java (*), because blocks are not only wrapping the contained statements, but also allow full access to the visible variables, can be assigned to variables, be stored in other objects or be returned from a method call or block evaluation:

```
|m1 m2 outerBlock|

m1 := 1233.
outerBlock :=
    [
        |o1 o2 innerBlock|

        o1 := 1.
        o2 := m1 + 1.
```

```
            innerBlock :=
                [
                    |i1 i2|

                    i1 := o2 + m1 + 4.
                    [ i1 + 1 ]
                ].
            innerBlock
        ].
```

corresponds to:

```
    var m1, m2, outerFunc;

    m1 = 1233;
    outerFunc = function() {
            var o1, o2, innerFunc;

            o1 = 1;
            o2 = m1 + 1;
            innerFunc = function() {
                    var i1, i2;

                    i1 = o2 + m1 + 4;
                    return function() { i1 + 1; };
                };
            return innerFunc;
    };
```

(*): newer versions of Java added a lambda feature, which provides a subset of the block semantics.

If a block gets evaluated, its inner statemens are evaluated, and the block's return value is the last inner statement's expression-value. So, inside a Smalltalk block, there is no return-from-this-block-statement, only expressions which are evaluated in sequence, for the last one to provide the return value of the block. You can (and often will) put a return inside a block, but it has a completely different behavior and is described below in more detail.

Blocks can have arguments, these are listed at the beginning, each with a trailing colon, before a vertical bar, as in:

```
    |block1 block2 block3|

    block1 := [ self helloWorld ].
    block2 := [:arg1 | self helloWorld ].
    block3 := [:arg1 :arg2 :arg3 | self helloWorld ].
```

As already mentioned, blocks can be passed to other code as argument. The collection classes in Smalltalk make heavy use of that, in that they provide an extensive set of enumeration functions, which iterate on the collection arguments, using a block argument which performs the operation. For example, here is a general iterator (which is implemented by every collection class), which enumerates the elements of an array. The array is written as an array-literal (that is a compile-time generated object):

```
    |myCollection|

    myCollection := #(1 2 3 4 5).
    ...
    myCollection do:[:el | Stdout show:el].
    ...
```

ignore the code inside the block for a moment - the "`show:`" message, if sent to a stream-like object will print its argument (el in the above example) on the stream. Look at how the block is passed as an argument of the "`do:`" message, which is sent to the array-instance (myCollection).
The above is also possible in JavaScript, but is relatively unreadable there, because it requires a full-featured function definition:

```
var myCollection;

myCollection = [1, 2, 3, 4, 5];.
...
myCollection.forEach( function(el) { Stdout.show(el); };
...
```

Notice that the Java community saw the power and usability of such constructs and added syntactic sugar to newer Java/JavaScript versions. Now, these also support a comprehensive "arg => expr" form. However, this only allows for a single expression as lambda body, without control structures (which of course is a consequence of having control structures as syntax, whereas in Smalltalk, all control is via message sends).

Returning through a Block

One very special feature of Smalltalk blocks (which is not even available in JavaScript) is their ability to return from their owning method. If a return statement ("`^ expression`") is evaluated inside a block, the containing method is returned from (not the block). Thus, the following code searches a collection for the first element for which a filter returns a true value:

```
myMethod
    ...
    someCollection do:[:el |
        (aMatchBlock value:el) ifTrue:[^ el]
    ].
    ...
```

this corresponds to:

```
function myMethod() {
    ...
    someCollection.forEach( function(el) {
        if (aMatchBlock(el)) return el from myMethod;
    }
    ...
```

or with some syntactic sugar:

```
function myMethod() {
    ...
    forEach( el in someCollection) {
        if (aMatchBlock(el)) return el from myMethod;
    }
    ...
```

in Java, you have to simulate that behavior using exceptions, which might look quite complicated. Passing a returning-block down to some other call corresponds technically to a *non-local return*, from however deep down that block was passed.

If you think carefully, you'll see that it does not really make sense to return from the block alone in the above enumeration code, as this would simply leave the inner block, and continue enumeration with the next element.

If at all, it sometimes makes sense to break out the whole enumeration loop. For this, Smalltalk collection classes offer an additional enumerator called "`doWithExit:`". This passes an additional "exit"-argument to the block:

```
    ...
    someCollection doWithExit:[:element :exit |
        ...
        someCondition ifTrue:[ exit value ]
        ...
    ].
    ...
```

which corresponds to:

```
    ...
    forEach (el in someCollection) {
        ...
        if (someCondition) break;
        ...
    }
    ...
```

Try / Catch / Finally

Smalltalk has more functionality in its exception handling mechanism, of which the Java exception handling is a subset. To handle an exception, the Java code:

```
    try {
        ...
        some computation
    } catch(<someErrorClass> e) {
        ...
        handler action
        ...
    } finally {
        ...
        cleanup
        ...
    }
```

is written in Smalltalk as:

```
    [
        ...
        some computation
        ...
    ] on:<someErrorClass> do:[:e |
        ...
        handler action
        ...
    ] ensure:[
        ...
        cleanup
        ...
    ]
```

there are also variations without ensure block (`on:do:`) and a version which corresponds to a "`finally`", without exception handler (`ensure:`). In addition, Smalltalk offers a variant of the finally, which is ONLY invoked in the non-normal-return situation (called "`ifCurtailed:`"), but not in a regular return (so the programmer can differentiate between exceptional and normal situations, which may be useful in some situations.

In both cases, "*someErrorClass*" is the class of the error to handle, and "*e*" is object providing exception information details. Also, in both languages, the Exception class forms a hierarchy, and a handler for an exception class E will also catch exceptions for any of its subclasses.

In addition to this hierarchical organisation, Smalltalk also offers handler sets, which catch a bunch of possibly unrelated error types. For example, to catch both arithmetic and file-operation errors in a single handler. And, another useful feature are ad-hoc exceptions (called "*Signal*"), which do not need a class, but are created "on-the-fly" and allow for purely private, completely anonymous exceptions, queries and notifications.

Semantic Differences

The biggest difference between Smalltalk and Java is not the syntax, but the semantics: when any handler is executed, in Java, the callstack has already been unwound and removed. The information object "*e*" only has a copy of the stack at hand, which can only be displayed or ignored. The actual execution context and state at the time of the exception raise is lost already. In Smalltalk, the execution is suspended when the exception is raised, and the stack frames and caller chain is still active and alive when the handler is executed. The handler can provide a value, which is to be returned from the raise.

This subtle difference has many implications:

- Smalltalk exceptions can be proceeded
  a handler may cleanup the situation and proceed. For example, a file-not-found exception could be cought, a dialog opened in the handler asking for a replacement file, and the handler could proceed execution passing an alternative filename or stream as return value from the raise.

- a debugger can show life values
  in Smalltalk, there is always a default fallback handler active (i.e. even if your code does not contain a handler, there is always another one around in an outer scope). This default handler opens a symbolic debugger on the suspended execution. As all local variables, arguments and other execution state is still alive, the debugger allows for very easy inspection of the program's state. There is no need to prime exceptions (i.e. tell the systems, which exceptions are to be debugged) for being debuggable in advance.

- a debugger can proceed after fixing
  the proceedability allows for even more fancy things to be done in the debugger: you can change/fix any invalid code right in the suspended state and proceed the suspended execution. Or restart any of the active methods in the suspended calling chain from the beginning, or simulate a return from any of them.

- proceedable exceptions are a synonym for notifications
  you can also use proceedable exceptions to tell an outer caller about ongoing activity. For example, info messages, logging, progress information and other such UI-related stuff can be sent out from

an inner processing function by raising a notification (which is a subclass of exception, but always proceedable, and - auto-proceeds if no handler is present). Thus, if some method is interested in such notifications, it can simply provide a handler. If no handler is present, the notification is ignored. This is a great way to pass compiler, parser, file-reader and other warnings to whomever it may concern.

Handler Sets

In addition to handling exceptions for a single error class (and its subclasses), you can create a handlerSet, which catches multiple, hierarchically unrelated error classes. For this, the concatenation operator (",") is used as in:

```
[
    ...
    some computation
    ...
] on:OpenError, ArithmeticError do:[:e |
    ...
    handler action
    ...
]
```

and of course, you can provide different handlers for each class individually:

```
(ExceptionHandlerSet new)
    on:ZeroDivide    do:[:ex | 'division by zero' printCR. ex proceed];
    on:HaltInterrupt do:[:ex | 'halt encountered ' printCR. ex proceed];
    on:DomainError   do:[:ex | 'domain error  ' printCR. ex proceed];
    handleDo:[
        ...
        your code here
        ...
    ]
```

Classes are Objects, too!

In Smalltalk, classes are also objects. Real objects with inheritance and local state (private variables). This is one of the major differences between Smalltalk and Java, and is also one of the major drawbacks of Java.

In Smalltalk, classes can be passed as argument, returned as result, stored in variables, etc. Everything that can be done with regular objects can also be done with classes.

One major effect of this, is that makes many design patterns which deal with factories, facades etc. obsolete in Smalltalk. For example, to instanciate either a FileLogger or a DummyLogger, depending on some condition, you'd simply write:

```
logger := (debugging ifTrue:[FileLogger] ifFalse:[DummyLogger]) new.
```

and start logging.

Or you can pass a class as parameter:

```
makeLogger:whichLoggerClass
    ^ whichLoggerClass new
```

being real objects, classes respond to class methods, just like ordinary instances respond to instance methods. Classes also inherit methods from their superclass(es).

The instance creation method "new" is therefore a regular class method, which can be redefined in a subclass. And the subclass may do something completely different in its own new method. For example, a singleton class may want to redefine new, to ensure that only one single instance of it is ever instantiated, by redefining new as:

```
new
    TheOneAndOnlyInstance isNil ifTrue:[
        TheOneAndOnlyInstance := super new
    ].
    ^ TheOneAndOnlyInstance
```

Be aware, that super new still returns an instance of the receiver's class - in contrast to java, where "super new" is not even possible, and if it was, it would return an instance of the superclass!

Another example is a class which redefines "new" to count the number of instantiated instances:

```
new
    Count := Count + 1.
    ^ super new
```

Having redefinable class protocol makes another big part of the design patterns obsolete. After all, some (cynic) tongues say that after all, "*most design patterns are workarounds for bad language design*".

Class Instance Variables vs. Class Variables

In Smalltalk, classes may define two type of class-related variables: so called "*Class Variables*", which correspond to static variables in Java, and "*Class Instance Variables*", for which no similar construct exists in Java.

Class Variables have a global scope, but are only visible in the class and its subclasses. A class variable exists exactly once, and all references to it refer to the same binding.

Class Instance Variables are additional private slots in the class object, and inherited by subclasses. However, each subclass gets its own binding slot. Thus, Class Instance Variables are for the class, what "private variables" (Instance Variables) are for regular objects. The code to work on them is shared by a class and its subclasses, but each class may have a different value for it.

In the above Count-example, if "Count" was a class variable, it would exist exactly once, and count the number of instantiations of the class and all of its subclasses.

In contrast, if "Count" was a class instance variable, the class and each subclass would have a private counter, which counted the number of instantiations for each class separately.

Class instance variables are very useful to provide per class caches, configurations etc. which have a similar functionality, but require different state as per subclass.

There is no corresponding mechanism in Java, and actually it would be harder to simulate. An implementation might use a HashTable, using a combination of (sub-)class and variable name as index.

Shared Pool Variables

Shared Pool Variables are another semantic feature, which is not present in Java. A Shared Pool defines a set of variables, which can be seen and referenced among a set of collaborating classes.

Thus, they can be seen as static variables, with a restricted visibility among multiple classes which need not be subclasses from a common superclass. (in Java, a static variable can never by seen by an outside class).

The Class Library

The class libraries of Smalltalk and Java share a lot in structure and implementation. Actually, many classes in Java were influenced, if not copied from corresponding classes in Smalltalk (which as you remember is much older than Java, and its class libraries have been around and matured for quite some time when Java came up). Also, many of the original Java developers had a Smalltalk background, so they brought some of those ideas into Java.

Number Classes

Smalltalk provide a very rich Number hierarchy, which includes (among others) Integer, Float, Fraction, Decimal and Complex numbers.

Mixed mode arithmetic is provided, and operations return a result as appropriate for the operands.

Smalltalk automatically cares for out-of-range values, and returns exact results for integer and fractional arithmetic (i.e. when adding two integers, it checks the result and automatically converts the result to a LargeInteger (which is Smalltalk's equivalent of BigInteger).

Integer overflows are not possible in Smalltalk.

In Smalltalk, all numeric objects are real objects (with a class, which can be extended to provide additional methods). As a Java programmer, think of all numbers being always boxed. In Smalltalk, there is no such concept as a *native type*, and every number is a full-blown object which can be put into containers, passed as argument or returned as value without special declarations.

Collections

Both Smalltalk and Java provide similar rich Collection hierarchies, which are used in a similar way:

Smalltalk	Java	Notes
Array	Object[]	
IntegerArray	int[]	primitive type in Java
ByteArray	byte[]	primitive type in Java
SignedWordArray	short[]	primitive type in Java
FloatArray	float[]	primitive type in Java
OrderedCollection	ArrayList<Object>	
Dictionary	Hashtable<Object,Object>	
Set	HashSet<Object>	
SortedCollection	??	

String	String	

A little annoying in Java is the different syntax and protocols of primitive type arrays vs. object collections: *length* vs. *length()* vs. *size()* and "*[]*" vs "*.get()*" and "*.put()*".
In Smalltalk, all getters are named "*at:*", all setters are named "*at:put:*" and to get the size of a container, always use "*size*".

With the exception of a few special collections (ByteArray, IntegerArray and FloatArray), all collections can hold any object in Smalltalk. There is no need/support for generics or templates. However, as all collections can be subclassed or wrapped, it is possible to define typed collection classes, which may restrict the set of accepted element types.

The above mentioned exceptions (ByteArray,...) are space-efficient variations.
For example, ByteArray stores byte-valued integer objects and needs 1byte of storage per element. In contrast, a full blown Array instance would require one pointer (4 or 8bytes, depending on the CPU) per element.
A similar "space" efficient container is "String" (which stores 1-byte characters), "Unicode16String" (which stores 2-byte characters), "FloatArray" and a bunch of other classes (IntegerArray, WordArray, DoubleArray etc.)

Streams

Smalltalk	Java
ReadStream	...
WriteStream	...
FileStream	...

Open vs. Closed Class Librariy

Another major difference between Smalltalk and Java is that in Java, all of the above mentioned classes are closed and cannot be extended. Some cannot even be subclassed (final classes).
In contrast, all classes in Smalltalk are extendable by providing additional methods as extensions of an application.

In practice, this has resulted in a lot of duplicate code in typical Java projects. You will often find utility container classes which duplicate existing functionality only because the original class cannot be extended.

For bigger projects, this shows up in the number of classes and methods (and lines of code); Java projects usually require many more lines of code than corresponding Smalltalk projects (we have seen Java projects which take up 5 times as many classes as the corresponding Smalltalk code).

Be prepared for more to come on reflection, the class library, extensions etc.

<*info@exept.de*>

Doc $Revision: 1.7 $
Last modification: $Date: 2016/09/14 09:41:12 $