**Background**

JSX is a statically-typed, object-oriented programming language compiling to standalone JavaScript. The reason why JSX was developed is our need for a more robust programming language than JavaScript. JSX is, however, fairly close to JavaScript especially in its statements and expressions.

Statically-typed programming language is robust because certain sorts of problems, for example typos in variable names or missing function definitions, are detected at compile-time. This is important especially in middle- to large-scale software development in which a number of engineers may be engaged.

Therefore, JSX is designed as a statically-typed language. All the values and variables have a static type and you can only assign a correctly-typed value to a variable. In addition, all the functions including closures have types which are determined by the types of parameters and the return values, where you cannot call a function with incorrectly typed arguments.

Also, another important reason why JSX was developed is to boost JavaScript performance. JavaScript itself is not so slow but large-scale development tends to have many abstraction layers, e.g. proxy classes and accessor methods, which often have negative impact on performance. JSX boosts performance by *inline expansion*: function bodies are expanded to where they are being called, if the functions being called could be determined at compile-time. This is the power of the statically-typed language in terms of performance.

**Run "Hello, World!"**

Let's start by running our first JSX program: hello.jsx. We use the jsx command, which is the JSX compiler in the JSX distribution, to compile JSX source code to JavaScript.

First, install [jsx](https://npmjs.org/package/jsx) with npm:

$ npm install -g jsx

Then, make the code below as hello.jsx:

class \_Main {

static function main(args : string[]) : void {

log "Hello, world!";

}

}

Now you can run a JSX program with the following command and you will get Hello, world on the console.

$ jsx --run hello.jsx

We will look into the detail of hello.jsx in the next section.

**Program Structure**

Here is hello.jsx, the source code of the "Hello world!" example. You can see several features of JSX in this program, namely, static types and class structure within the source code.

class \_Main {

static function main(args : string[]) : void {

log "Hello, world!";

}

}

Class \_Main has a static member function (a.k.a. a class method) named main, that takes an array of strings and returns nothing. \_Main.main(:string[]):void is the entry point of JSX applications that is called when a user invokes an application from command line. JSX, like Java, does not allow top-level statements or functions.

The log statement is mapped to console.log() in JavaScript, which displays the arguments to stdout with a newline.

Next, we look into another typical library class, Point:

class Point {

var x = 0;

var y = 0;

function constructor() {

}

function constructor(x : number, y : number) {

this.set(x, y);

}

function constructor(other : Point) {

this.set(other);

}

function set(x : number, y : number) : void {

this.x = x;

this.y = y;

}

function set(other : Point) : void {

this.set(other.x, other.y);

}

}

As you can see, member variables of Point, var x and var y, are declared without types, but their types are deducted from their initial values to be number.

You might be surprised at multiple definition of constructors: one takes no parameters and the others take parameters. They are overloaded by their types of parameters. When you construct the class with new Point(), the first constructor, which takes no parameters, is called. The second with two parameters will be called on new Point(2, 3) and the third with one parameter will be called as a copy constructor. Other forms of construction, e.g. new Point(42) or new Point("foo", "bar") will cause compilation errors of mismatching signatures. The Point#set() functions are also overloaded and the compiler know how to call the correct one.

**Static Typing**

Basic type concept will be described in this section. Primitive types, object types, variant type, and Nullable types exist in JSX.

**Pritimive Types**

There are three pritimive types in JSX: string, number, and boolean. The three are non-nullable, immutable types. The code snippet below declares three variables s, n, b with their repective types, annocated to the right of the name of the variables using the : mark.

var s : string;

var n : number;

var b : boolean;

Type annotations can be omitted when a variable is initialized at the same moment of declaration.

var s = "hello"; // s is string, initialized as "hello"

var n = 42; // n is number, initialized as 42

var b = true; // b is boolean, initialized as true

**Object Types**

Object types are types of values to hold reference to objects - which are instances of classes. For example, functions, string[] (array of strings), Date are all object types. Whether they are mutable or not depends on the definition of each class.

Most of the objects (values of object types) are constructed using the new operator.

var d = new Date(); // instantiate an object of class Date

var a = new Array.<string>(); // instantiate an array of string

var m = new Map.<number>(); // instantiate an associative map of strings to numbers

Array and Map types can also be instatiated by using their initializer expressions.

var a1 = [] : Array.<string>; // a1 is Array.<string>, and is empty

var a2 = [ 1, 2, 3 ]; // a2 is Array.<number> with three elements

var m1 : {} : Map.<number>; // m1 is Map.<number>

var m2 = { // m2 is Map.<string>

en: "Good morning",

de: "Guten Morgen",

ja: "ãã¯ãããããã¾ã"

};

Variables of the Function class can only be instantiated as a static function or by using function expression or function statement (the details are described laterwards).

**The Variant Type**

Variant type, which means "no static type information," is useful for interacting with existing JavaScript APIs. Some JavaScript libraries may return a variant value, which type cannot be determined at compile time. All you can do on variant values is to check equality of a variant value to another variant value. You have to cast it to another type before doing anything else on the value.

function guessTheType(v : variant) : void {

if (typeof v == "string") {

log "v is string and the value is:" + v as string;

} else {

log "v is not string";

}

}

**Nullable Types**

Nullable type is a meta type which indicates a value may be null. It is prohibited to assign null to the primitive types (note: Object types are nullable by default). Nullable types should instead be used for such purposes.

var s1 : string;

s1 = "abc"; // ok

s1 = null; // compile error! cannot assign null to string

var s2 : Nullable.<string>;

s2 = "abc"; // ok

s2 = null; // ok

The most prominent use case of Nullable types is when interacting with an array. For example, an out-of-bounds access to an array returns null.

var a = [ 1, 2, 3 ]; // creates Array.<number> with three elements

a[3]; // out-of-bounds access, returns null

There are APIs that return Nullable types also exists. For example, the return type of Array.<string>#shift() is Nullable.<string>. When you use a Nullable value, you have to make sure of the value is not null.

function shiftOrReturnEmptyString(args : string[]) : string {

if (args.length > 0)

return args.shift();

else

return "";

}

When the source code is compiled in debug mode (which is the default), the compiler will insert run-time type-checking code. An exception will be raised (or the debugger will be activated) when misuse of a null value as actual value is detected. Run-time type checks can be omitted by compiling the source code with the --release option.

Please refer to the [Types](https://jsx.github.io/doc/typeref.html) section of the language reference for more information.

**Expressions**

*The definitions of operators in JSX are almost equivalent to JavaScript*, however there are few exceptions.

* arithmetic operators (+, -, \*, /...) only accept numbers as the operands
* var a = 3;
* a + 1; // OK, returns 4
* a \* a; // OK, returns 9

a + "abc"; // compile error

Note: concatenation operator + exists for concatenation of strings

* the dot property accessor can only access the defined properties
* class Point {
* var x : number;
* var y : number;
* function print() : void {
* log this.z; // compile error! no property named z
* }

}

* the [] property accessor can only be applied to values of type Map or variant
* var m = { // m is Map.<string>
* hello: "world!"
* };
* log m["hello"]; // OK

log m.hello; // compile error!

* introduction of the as operator, used for type conversion between primitive types / casting object types
* var n = 123;
* var s = "value of n is " + (n as string);

log s; // print "value of n is 123"

* logical operators (&&, ||) returns boolean, and the introduction of binary ?: operator as the equivalent to the || operator of JavaScript

A complete list of operators can be found in the [Operator Reference](https://jsx.github.io/doc/operatorref.html).

**Statements**

*JSX supports most of the statement types provided by JavaScript.* The exceptions are:

* log statement

log "hello, world"; // log strings to console, can turned off with compile option: --release

* assert statement
* var n = 123;

assert n != 0; // assertions. also can be turned off with --release

* try-catch-finally statement
* try {
* ...
* } catch (e : TypeError) {
* // got TypeError
* } catch (e : Error) {
* // got Error, which is not TypeError
* } catch (e : variant) {
* // applications may throw any kind of value
* } finally {
* ...

}

* no with statement

A complete list of statements can be found in the [Statement Reference](https://jsx.github.io/doc/statementref.html).

**Classes and Interfaces**

JSX is a class-based object-oriented language, and its class model is similar to Java.

* a class may extend another class (single inheritance)
* a class may implement multiple interfaces and mixins
* all classes share a single root class: the Object class

interface Flyable {

abstract function fly() : void;

}

abstract class Animal {

function eat() : void {

log "An animal is eating!";

}

}

class Bat extends Animal implements Flyable {

override function fly() : void {

log "A bat is flying!";

}

}

abstract class Insect {

}

class Bee extends Insect implements Flyable {

override function fly() : void {

log "A bee is flying!";

}

}

class \_Main {

static function main(args : string[]) : void {

// fo bar

var bat = new Bat();

var animal : Animal = bat; // OK. A bat is an animal.

animal.eat();

var flyable : Flyable = bat; // OK. A bat can fly

flyable.fly();

// for Bee

var bee = new Bee();

flyable = bee; // A bee is also flyable

flyable.fly();

}

}

In the example, the Bat class extends the Animal class, so it inherits the Animal#eat() member function, and it can be assigned to a variable typed to Animal. The class also implements theFlyable interface overriding the Flyable#fly() member function, so it can be assigned to a variable typed Flyable. There's also another flyable class, Bee. By using the Flyable interface, it is possible to deal with both classes as a flyable being, even if the organ of a bee is completely different from that of a bat.

When overriding a member function, the use the override keyword is mandatory. Otherwise the compiler will report an error. In other words, you are saved from unexpected interface changes in the base classes which cause compilation errors in derived classes instead of undesirable runtime errors.

**Functions and Closures**

In JSX, functions are first-class objects and they have static types. You can declare a variable of a function type like var f : function(arg : number) : number, a function that takes a number as an argument and returns another number (or, just returns the same value as the argument; but it's not important here). The variable f can be called as f(42) from which you will get a number value.

It is possible to define closures using the function expression or the function statement. They are typically used to implement callbacks ore event listeners which are popular in GUI programming. Closures are similar to JavaScript except for what this points at: when a closure is defined within a member function, it refers to the receiver of the member function. See the following example.

class \_Main {

var foo = 42;

function constructor() {

var f = function() : void {

log this.foo;

};

f(); // says 42

}

static function main(args : string[]) : void {

var o = new \_Main();

}

}

Type annocations of function expressions / statements may be omitted if they can be inferred by the compiler. In the exmaple below, both the type of the argument n and the return type of the function expression is inferred from the definition of Array#map to be number.

var doubled = [ 1, 2, 3 ].map(function (n) {

return n \* 2;

});

**Modules**

JSX has a module system. You can use JSX class libraries by the import statement. For example, the following program uses timer.jsx module, which exports the Timer class.

import "timer.jsx";

class \_Main {

static function main(args : string[]) : void {

Timer.setTimeout(function() : void {

log "Hello, world!";

}, 1000);

}

}

A module may export multiple classes, but you can specify what modules you import or name a namespace which the module is imported into.

**Interface to Web Browsers**

The js/web.jsx module provides the interface to web browser APIs, e.g. the window object and DOM APIs. The example below shows how to insert a text node into an HTML.

// hello.jsx

import "js/web.jsx";

class \_Main {

static function main(args : string[]) : void {

var document = dom.window.document;

var text = document.createTextNode("Hello, world!");

document.getElementById("hello").appendChild(text);

}

}

<!DOCTYPE html>

<html>

<head>

<title>Hello, world!</title>

<script src="hello.jsx.js"></script>

</head>

<body>

<p id="hello"></p>

</body>

</html>

Once you compile hello.jsx by the following command, then you can access the HTML and you will see it saying "Hello, world!."

$ bin/jsx --executable web --output hello.jsx.js hello.jsx