JSweet Language Specifications
Version 2.x

Renaud Pawlak
renaud.pawlak@jsweet.org
http://www.jsweet.org

Louis Grignon
louis.grignon@gmail.com
http://www.jsweet.org

Contents

Basic concepts

This section presents the JSweet language basic concepts. One must keep in mind that JSweet, as a Java-to-JavaScript transpiler, is an extension of Java at compile-time, and executes as JavaScript at runtime. JSweet aims at being a trade-off between Java and JavaScript, by respecting as much as possible the Java semantics, but without loosing interoperability with JavaScript. So, in a way, JSweet can be seen as a fusion between Java and JavaScript, trying to get the best of both worlds in one unique and consistent language. In some cases, it is hard to get the best of both worlds and JSweet makes convenient and practical choices.

Because JSweet is an open JavaScript transpiler, the user can tune the JavaScript generation without much efforts, thus making other choices than default ones to map Java to JavaScript. For example, if the way JSweet implements Java maps does not suit your context or use case, you can program a JSweet extension to override the default strategy. Programming and activating a JSweet extension is fully explained in Section 6.

Core types and objects

JSweet allows the use of primitive Java types, core Java objects (defined in `java.lang`, many JDK classes (especially `java.util` but not only), and of core JavaScript objects, which are defined in the `def.js` package. Next, we describe the use of such core types and objects.

Primitive Java types

JSweet allows the use of Java primitive types (and associated literals).

- `int`, `byte`, `short`, `double`, `float` are all converted to JavaScript numbers (TypeScript `number` type). Precision usually does not matter in JSweet, however, casting to `int`, `byte`, or `short` forces the number to be rounded to the right-length integer.
- `char` follows the Java typing rules but is converted to a JavaScript `string` by the transpiler.
- `boolean` corresponds to the JavaScript `boolean`.
- `java.lang.String` corresponds to the JavaScript `string`. (not per say a primitive type, but is immutable and used as the class of string literals in Java)

A direct consequence of that conversion is that it is not always possible in JSweet to safely overload methods with numbers or chars/strings. For instance, the methods `pow(int, int)` and `pow(double, double)` may raise overloading issues. With a JSweet context, the transpiler will be able to select the right method, but the JavaScript interoperability may be a problem. In short, since there is no difference between `n instanceof Integer` and `n instanceof Double` (it both means `typeof n === 'number'`) calling `pow(number, number)` from JavaScript will randomly select one implementation or the other. This should not be always a problem, but in some particular cases, it can raise subtle errors. Note that in these cases, the programmers will be able to tune the JavaScript generation, as it is fully explained in Section 6.

Examples of valid statements:

```
// warning '==' behaves like JavaScript '===' at runtime
int i = 2;
assert i == 2;
double d = i + 4;
assert d == 6;
String s = "string" + '0' + i;
assert s == "string02";
boolean b = false;
assert !b;
```

The == operator behaves like the JavaScript strict equals operator === so that it is close to the Java semantics. Similarly, != is mapped to !==. There is an exception to that behavior which is when comparing an object to a null literal. In that case, JSweet translates to the loose equality operators so that the programmers see no distinction between null and undefined (which are different in JavaScript but it may be confusing to Java programmers). To control whether JSweet generates strict or loose operators, you can use the following helper methods: jsweet.util.Lang.$strict and jsweet.util.Lang.$loose. Wrapping a comparison operator in such a macro will force JSweet to generate a strict or loose operator. For example:

```
import static jsweet.util.Lang.$loose;
[...]
int i = 2;
assert i == 2; // generates i === 2
assert !((Object)"2" == i);
assert $loose((Object)"2" == i); // generates "2" == i
```

Allowed Java objects

By default, JSweet maps core Java objects and methods to JavaScript through the use of built-in macros. It means that the Java code is directly substituted with a valid JavaScript code that implements similar behavior. A default mapping is implemented for most useful core Java classes (java.lang, java.util). When possible (and when it makes sense), some partial mapping is implemented for other JDK classes such as input and output streams, locales, calendars, reflection, etc.

With the default behavior, we can point the following limitations:

- Extending a JDK class is in general not possible, except for some particular contexts. If extending a JDK class is required, should should consider to refactor your program, or use a JavaScript runtime (such as J4TS), which would allow it.
- The Java reflection API (java.lang.reflect) is limited to very basic operations. It is possible to access the classes and the members, but it is not possible to access types. A more complete support of Java reflection would be possible, but it would require a JSweet extension.
- Java 8 streams are not supported yet, but it would be simple to support them partially (contributions are welcome).

Examples of valid statements:

```
Integer i = 2;
assert i == 2;
Double d = i + 4d;
assert d.toString() == "6";
```

```
assert !((Object) d == "6");
BiFunction<String, Integer, String> f = (s, i) -> { return s.substring(i); };
assert "bc" == f.apply("abc", 1);
```

Getting more Java APIs

With JSweet, it is possible to add a runtime that implements Java APIs in JavaScript, so that programmers can access more Java APIs and thus share the same code between Java and JavaScript. The core project for implementing Java APIs for JSweet is J4TS (https://github.com/cincheo/j4ts) and contains a quite complete implementation of `java.util.*` classes and other core package. J4TS is based on a fork of the GWT's JRE emulation, but it is adapted to be compiled with JSweet. Programmers can use J4TS as a regular JavaScript library available in our Maven repository.

Although J4TS cannot directly implement the Java core types that conflict with JavaScript ones (`Boolean`, `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Character`, `String`), J4TS contributes to supporting the static part of them by providing helpers for each class (`javaemul.internal.BooleanHelper`, `javaemul.internal.ByteHelper`, ...). When the JSweet transpiler meets a static Java method on a type `java.lang.T` that is not supported as a built-in macro, it delegates to `javaemul.internal.THelper`, which can provide a JavaScript implementation for the given static method. That way, by using J4TS, programmers can use even more of the core JRE API.

Java arrays

Arrays can be used in JSweet and are transpiled to JavaScript arrays. Array initialization, accesses and and iteration are all valid statements.

```
int[] arrayOfInts = { 1, 2, 3, 4};
assert arrayOfInts.length == 4;
assert arrayOfInts[0] == 1;
int i = 0;
for (int intItem : arrayOfInts) {
    assert arrayOfInts[i++] == intItem;
}
```

Core JavaScript API

The core JavaScript API is defined in `def.js` (the full documentation can be found at http://www.jsweet.org/core-api-javadoc/). Main JavaScript classes are:

- `def.js.Object`: JavaScript Object class. Common ancestor for JavaScript objects functions and properties.

- `def.js.Boolean`: JavaScript Boolean class. A wrapper for boolean values.
- `def.js.Number`: JavaScript Number class. A wrapper for numerical values.
- `def.js.String`: JavaScript String class. A wrapper and constructor for strings.
- `def.js.Function`: JavaScript Function class. A constructor for functions.
- `def.js.Date`: JavaScript Date class, which enables basic storage and retrieval of dates and times.
- `def.js.Array<T>`: JavaScript Array class. It is used in the construction of arrays, which are high-level, list-like objects.
- `def.js.Error`: JavaScript Error class. This class implements `java.lang.RuntimeException` and can be thrown and caught with `try` ... `catch` statements.

When using JavaScript frameworks, programmers should use this API most of the time, which is HTML5 compatible and follows the JavaScript latest supported versions. However, for objects that need to be used with Java literals (numbers, booleans, and strings), the use of the `java.lang` package classes is recommended. For instance, the jQuery API declares `$(java.lang.String)` instead of `$(def.js.String)`. This allows the programmer to write expressions using literals, such as `$("a")` (for selecting all links in a document).

With JSweet, programmers can easily switch from the Java to JavaScript API (and conversely) depending on their needs. The `jsweet.util.Lang` class defines convenient static methods to cast back and forth core Java objects to their corresponding JavaScript objects. For instance the `string(...)` method will allow the programmer to switch from the Java to the JavaScript strings and conversely.

```
import static jsweet.util.Lang.string;
// str is a Java string, but is actually a JavaScript string at runtime
String str = "This is a test string";
// str is exactly the same string object, but shown through the JS API
def.js.String str2 = string(str);
// valid: toLowerCase it defined both in Java and JavaScript
str.toLowerCase();
// this method is not JS-compatible, so a macro generates the JS code
str.equalsIgnoreCase("abc");
// direct call to the JS substr method on the JavaScript string
string(str).substr(1);
// or
str2.substr(1);
```

Note: for code sharing between a JavaScript client and a Java server for instance, it is better to use Java APIs only and avoid JavaScript ones. JavaScript API

will compile valid Java bytecode but trying to execute them on a JVM will raise unsatisfied link errors.

Here is another example that shows the use of the `array` method to access the `push` method available on JavaScript arrays.

```
import static jsweet.util.Lang.array;
String[] strings = { "a", "b", "c" };
array(strings).push("d");
assert strings[3] == "d";
```

Classes

Classes in JSweet fully support all types of Java classes declarations. For example:

```
public class BankAccount {
    public double balance = 0;
    public double deposit(double credit) {
        balance += credit;
        return this.balance;
    }
}
```

Which is transpiled to the following JavaScript code:

```
var BankAccount = (function () {
    function BankAccount() {
        this.balance = 0;
    }
    BankAccount.prototype.deposit = function(credit) {
        this.balance += credit;
        return this.balance;
    };
    return BankAccount;
})();
```

Classes can define constructors, have super classes and be instantiated exactly like in Java. Similarly to Java, inner classes and anonymous classes are allowed in JSweet (since version 1.1.0). JSweet supports both static and regular inner/anonymous classes, which can share state with enclosing classes. Still like in Java, anonymous classes can access final variables declared in their scope. For example, the following declarations are valid in JSweet and will mimic the Java semantics at runtime so that Java programmers can benefit all the features of the Java language.

```
abstract class C {
    public abstract int m();
}
```

```
public class ContainerClass {
    // inner class
    public class InnerClass {
        public I aMethod(final int i) {
            // anonymous class
            return new C() {
                @Override
                public int m() {
                    // access to final variable i
                    return i;
                }
            }
        }
    }
}
```

Interfaces

In JSweet, an interface can be use like in Java. However, on contrary to Java, there is no associated class available as runtime. When using interfaces, JSweet generates code to emulate specific Java behaviors (such as `instanceof` on interfaces).

JSweet supports Java 8 static and default methods. However default methods are experimental so far and you should use them at your own risks.

In JSweet, interfaces are more similar to interfaces in TypeScript than in Java. It means that they must be seen as object signatures, which can specify functions, but also properties. In order to allow using fields as properties when defining interfaces, JSweet allows the use of regular classes annotated with `@jsweet.lang.Interface`. For example, the following interface types an object `Point` with 2 properties.

```
@Interface
public class Point {
    public double x;
    public double y;
}
```

To Java programmers, this may look like a very odd way to define an object, but you must remember that it is not a class, but a type for a JavaScript object. As such, it does not go against the OOP principles. We can create a JavaScript object typed after the interface. Note that the following code is not actually creating an instance of the `Point` interface, it is creating an object that conforms to the interface.

```
Point p1 = new Point() {{ x=1; y=1; }};
```

This object creation mechanism is a TypeScript/JavaScript mechanism and shall not be confused with anonymous classes, which is a Java-like construction. Because `Point` is annotated with `@Interface`, the transpiled JavaScript code will be similar to:

```
var p1 = Object.defineProperty({ x:1, y:1 }, "_interfaces", ["Point"]);
```

Note that, for each object, JSweet keeps track of which interface it was created from and of all the potential interfaces implemented by its class. This interface tracking system is implemented as a special object property called `__interfaces`. Using that property, JSweet allows the use of the `instanceof` operator on interfaces like in Java.

Optional fields in interfaces

Interfaces can define *optional fields*, which are used to report errors when the programmer forgets to initialize a mandatory field in an object. Supporting optional fields in JSweet is done through the use of `@jsweet.lang.Optional` annotations. For instance:

```
@Interface
public class Point {
    public double x;
    public double y;
    @Optional
    public double z = 0;
}
```

It is the JSweet compiler that will check that the fields are correctly initialized, when constructing an object from an interface.

```
// no errors (z is optional)
Point p1 = new Point() {{ x=1; y=1; }};
// JSweet reports a compile error since y is not optional
Point p2 = new Point() {{ x=1; z=1; }};
```

Special JavaScript functions in interfaces

In JavaScript, objects can have properties and functions, but can also (not exclusively), be used as constructors and functions themselves. This is not possible in Java, so JSweet defines special functions for handling these cases.

- `$apply` is used to state that the object can be used as a function.
- `$new` is used to state that the object can be used as a constructor.

For instance, if an object `o` is of interface `O` that defines `$apply()`, writing:

```
o.$apply();
```

Will transpile to:

```
o();
```

Similarly, if `O` defines `$new()`:

```
o.$new();
```

Will transpile to:

```
new o();
```

Yes, it does not make sense in Java, but in JavaScript it does!

Untyped objects (maps)

In JavaScript, object can be seen as maps containing key-value pairs (key is often called *index*, especially when it is a number). So, in JSweet, all objects define the special functions (defined on `def.js.Object`):

- `$get(key)` accesses a value with the given key.
- `$set(key,value)` sets or replace a value for the given key.
- `$delete(key)` deletes the value for the given key.

Reflective/untyped accesses

The functions `$get(key)`, `$set(key,value)` and `$delete(key)` can be seen as a simple reflective API to access object fields and state. Note also the static method `def.js.Object.keys(object)`, which returns all the keys defined on a given object.

The following code uses this API to introspect the state of an object `o`.

```
for(String key : def.js.Object.keys(o)) {
  console.log("key=" + key +  " value=" + o.$get(key));
});
```

When not having the typed API of a given object, this API can be useful to manipulate the object in an untyped way (of course it should be avoided as much as possible).

Untyped objects initialization

One can use the `$set(key,value)` function to create new untyped object. For instance:

```
Object point = new def.js.Object() {{ $set("x", 1); $set("y", 1); }};
```

It transpiles also to:

```
var point = { "x": 1, "y": 1};
```

As a shortcut, one can use the `jsweet.util.Lang.$map` function, which transpiles to the exact same JavaScript code:

```
import static jsweet.util.Lang.$map;
[...]
Object point = $map("x", 1, "y", 1);
```

Indexed objects

The type of keys and values can be overloaded for every object. For example, the `Array<T>` class, will define keys as numbers and values as objects conforming to type `T`.

In the case of objects indexed with number keys, it is allowed to implement the `java.lang.Iterable` interface so that it is possible to use they in *foreach* loops. For instance, the `NodeList` type (from the DOM) defines an indexed function:

```
@Interface
class NodeList implements java.lang.Iterable {
    public double length;
    public Node item(double index);
    public Node $get(double index);
}
```

In JSweet, you can access the node list elements with the `$get` function, and you can also iterate with the *foreach* syntax. The following code generates fully valid JavaScript code.

```
NodeList nodes = ...
for (int i = 0; i < nodes.length; i++) {
    HTMLElement element = (HTMLElement) nodes.$get(i);
    [...]
}
// same as:
NodeList nodes = ...
for (Node node : nodes) {
    HTMLElement element = (HTMLElement) node;
    [...]
}
```

Enums

JSweet allows the definition of enums similarly to Java. The following code declares an enum with tree possible values (`A`, `B`, and `C`).

```
enum MyEnum {
    A, B, C
}
```

The following statements are valid statements in JSweet.

```
MyEnum e = MyEnum.A;
assert MyEnum.A == e;
assert e.name() == "A";
assert e.ordinal() == 0;
assert MyEnum.valueOf("A") == e;
assert array(MyEnum.values()).indexOf(MyEnum.valueOf("C")) == 2;
```

Like Java enums, additional methods, constructors and fields can be added to enums.

```
enum ScreenRatio {
    FREE_RATIO(null),
    RATIO_4_3(4f / 3),
    RATIO_3_2(1.5f),
    RATIO_16_9(16f / 9),
    RATIO_2_1(2f / 1f),
    SQUARE_RATIO(1f);

    private final Float value;

    private MyComplexEnum(Float value) {
        this.value = value;
    }

    public Float getValue() {
        return value;
    }
}
```

Enums portability notes

Simple enums are translated to regular TypeScript enums, that is to say numbers. In JavaScript, at runtime, an enum instance is simple encode as its ordinal. So, JSweet enums are easy to share with TypeScript enums and a JSweet program can interoperate with a TypeScript program even when using enums.

Enums with additional members are also mapped to TypeScript enums, but an additional class is generated to store the additional information. When interoperating with TypeScript, the ordinal will remain, but the additional information will be lost. The programmers wanting to share enums with TypeScript should be aware of that behavior.

Globals

In Java, on contrary to JavaScript, there is no such thing as global variables or functions (there are only static members, but even those must belong to a class). Thus, JSweet introduces reserved `Globals` classes and `globals` packages. These have two purposes:

- Generate code that has global variables and functions (this is discouraged in Java)
- Bind to existing JavaScript code that defines global variables and functions (as many JavaScript frameworks do)

In Globals classes, only static fields (global variables) and static methods (global functions) are allowed. Here are the main constraints applying to Globals classes:

- no non-static members
- no super class
- cannot be extended
- cannot be used as types like regular classes
- no public constructor (empty private constructor is OK)
- cannot use $get, $set and $delete within the methods

For instance, the following code snippets will raise transpilation errors.

```
class Globals {
    public int a;
    // error: public constructors are not allowed
    public Globals() {
        this.a = 3;
    }
    public static void test() {
        // error: no instance is available
        $delete("key");
    }
}

// error: Globals classes cannot be used as types
Globals myVariable = null;
```

One must remember that `Globals` classes and `global` packages are erased at runtime so that their members will be directly accessible. For instance `mypackage.Globals.m()` in a JSweet program corresponds to the `mypackage.m()` function in the generated code and in the JavaScript VM at runtime. Also, `mypackage.globals.Globals.m()` corresponds to *m()*.

In order to erase packages in the generated code, programmers can also use the `@Root` annotation, which will be explained in Section 5.

Optional parameters and overloading

In JavaScript, parameters can be optional, in the sense that a parameter value does not need to be provided when calling a function. Except for varargs, which are fully supported in JSweet, the general concept of an optional parameter does not exist in Java. To simulate optional parameters, JSweet programmers can use method overloading, which is supported in Java. Here are some examples of supported overloads in JSweet:

```
String m(String s, double n) { return s + n; }
// simple overloading (JSweet transpiles to optional parameter)
String m(String s) { return m(s, 0); }
// complex overloading (JSweet generates more complex code to mimic the Java behavior)
String m(String s) { return s; }
```

Bridging to external JavaScript elements

It can be the case that programmers need to use existing libraries from JSweet. In most cases, one should look up in the available candies, a.k.a. bridges at http://www.jsweet.org/jsweet-candies/. When the candy does not exist, or does not entirely cover what is needed, one can create new definitions in the program just by placing them in the `def.libname` package. Definitions only specify the types of external libraries, but no implementations. Definitions are similar to TypeScript's `*.d.ts` definition files (actually JSweet generates intermediate TypeScript definition files for compilation purposes). Definitions can also be seen as similar to `*.h` C/C++ header files.

Examples

The following example shows the backbone store class made accessible to the JSweet programmer with a simple definition. This class is only for typing and will be generated as a TypeScript definition, and erased during the JavaScript generation.

```
package def.backbone;
class Store {
    public Store(String dbName) {}
}
```

Note that definition classes constructors must have an empty body. Also, definition classes methods must be `native`. For instance:

```
package def.mylib;
class MyExternalJavaScriptClass {
    public native myExternalJavaScriptMethod();
}
```

It is possible to define properties in definitions, however, these properties cannot be initialized.

Rules for writing definitions (a.k.a. bridges)

By convention, putting the classes in a `def.libname` package defines a set of definitions for the `libname` external JavaScript library called `libname`. Note that this mechanism is similar to the TypeScript `d.ts` definition files.

Candies (bridges to external JavaScript libraries) use definitions. For instance, the jQuery candy defines all the jQuery API in the `def.jquery` package.

Here is a list of rules and constraints that need to be followed when writing definitions.

- Interfaces are preferred over classes, because interfaces can be merged and classes can be instantiated. Classes should be used only if the API defines an explicit constructor (objects can be created with `new`). To define an interface in JSweet, just annotate a class with `@jsweet.lang.Interface`.
- Top-level functions and variables must be defined as `public static` members in a `Globals` class.
- All classes, interfaces and packages, should be documented with comments following the Javadoc standard.
- When several types are possible for a function parameter, method overloading should be preferred over using union types. When method overloading is not possible, it can be more convenient to simply use the `Object` type. It is less strongly typed but it is easier to use.
- One can use string types to provide function overloading depending on a string parameter value.
- In a method signature, optional parameters can be defined with the `@jsweet.lang.Optional` annotation.
- In an interface, optional fields can be defined with the `@jsweet.lang.Optional` annotation.

Definitions can be embedded directly in a JSweet project to access an external library in a typed way.

Definitions can also be packaged in a candy (a Maven artifact), so that they can be shared by other projects. See the *Packaging* section for full details on how to create a candy. Note that you do not need to write definitions when a library is written with JSweet because the Java API is directly accessible and the TypeScript definitions can be automatically generated by JSweet using the `declaration` option.

Untyped accesses

Sometimes, definitions are not available or are not correct, and just a small patch is required to access a functionality. Programmers have to keep in mind that JSweet is just a syntactic layer and that it is always possible to bypass typing in order to access a field or a function that is not explicitly specified in an API.

Although, having a well-typed API is the preferred and advised way, when such an API is not available, the use of `def.js.Object.$get` allows reflective access to methods and properties that can then be cast to the right type. For accessing functions in an untyped way, one can cast to `def.js.Function` and call the generic and untyped method `$apply` on it. For example, here is how to invoke the jQuery `$` method when the jQuery API is not available :

```
import def.dom.Globals.window;
[...]
Function $ = (Function)window.$get("$");
$.$apply("aCssSelector"):
```

The `$get` function is available on instances of `def.js.Object` (or subclasses). For a `def.js.Object`, you can cast it using the `jsweet.util.Lang.object` helper method. For example:

```
import static jsweet.dom.Lang.object;
[...]
object(anyObject).$get("$");
```

In last resort, the `jsweet.util.Lang.$insert` helper method allows the user to insert any TypeScript expression within the program. Invalid expressions will raise a TypeScript compilation error, but it is however not recommended to use this technique.

```
import static jsweet.dom.Lang.$get;
import static jsweet.dom.Lang.$apply;
[...]
// generate anyObject["prop"]("param");
$apply($get(anyObject, "prop"), "param");
```

Finally, note also the use of the `jsweet.util.Lang.any` helper method, which can be extremely useful to erase typing. Since the `any` method generates a cast to the `any` type in TypeScript, it is more radical than a cast to `Object` for instance. The following example shows how to use the `any` method to cast an `Int32Array` to a Java `int[]` (and then allow direct indexed accesses to it.

```
ArrayBuffer arb = new ArrayBuffer(2 * 2 * 4);
int[] array = any(new Int32Array(arb));
int whatever = array[0];
```

Mixins

In JavaScript, it is common practice to enhance an existing class with news elements (field and methods). It is an extension mechanism used when a framework defines plugins for instance. Typically, jQuery plugins add new elements to the `JQuery` class. For example the jQuery timer plugin adds a `timer` field to the `JQuery` class. As a consequence, the `JQuery` class does not have the same prototype if you are using jQuery alone, or jQuery enhanced with its timer plugin.

In Java, this extension mechanism is problematic because the Java language does not support mixins or any extension of that kind by default.

Untyped accesses to mixins

Programmers can access the added element with `$get` accessors and/or with brute-force casting.

Here is an example using `$get` for the timer plugin case:

```
((Timer)$("#myId").$get("timer")).pause();
```

Here is an other way to do it exampled through the use of the jQuery UI plugin (note that this solution forces the use of `def.jqueryui.JQuery` instead of `def.jquery.JQuery` in order to access the `menu()` function, added by the UI plugin):

```
import def.jqueryui.JQuery;
[...]
Object obj = $("#myMenu");
JQuery jq = (JQuery) obj;
jq.menu();
```

However, these solutions are not fully satisfying because clearly unsafe in terms of typing.

Typed accesses with mixins

When cross-candy dynamic extension is needed, JSweet defines the notion of a mixin. A mixin is a class that defines members that will end up being directly accessible within a target class (mixin-ed class). Mixins are defined with a `@Mixin` annotation. Here is the excerpt of the `def.jqueryui.JQuery` mixin:

```
package def.jqueryui;
import def.dom.MouseEvent;
import def.js.Function;
import def.js.Date;
import def.js.Array;
import def.js.RegExp;
```

```
import def.dom.Element;
import def.jquery.JQueryEventObject;
@jsweet.lang.Interface
@jsweet.lang.Mixin(target=def.jquery.JQuery.class)
public abstract class JQuery extends def.jquery.JQuery {
    native public JQuery accordion();
    native public void accordion(jsweet.util.StringTypes.destroy methodName);
    native public void accordion(jsweet.util.StringTypes.disable methodName);
    native public void accordion(jsweet.util.StringTypes.enable methodName);
    native public void accordion(jsweet.util.StringTypes.refresh methodName);
    ...
    native public def.jqueryui.JQuery menu();
    ...
```

One can notice the `@jsweet.lang.Mixin(target=def.jquery.JQuery.class)` that states that this mixin will be merged to the `def.jquery.JQuery` so that users will be able to use all the UI plugin members directly and in a well-typed way.

How to use

TBD.

Generating JSweet candies from existing TypeScript definitions

TBD.

Auxiliary types

JSweet uses most Java typing features (including functional types) but also extends the Java type system with so-called *auxiliary types*. The idea behind auxiliary types is to create classes or interfaces that can hold the typing information through the use of type parameters (a.k.a *generics*), so that the JSweet transpiler can cover more typing scenarios. These types have been mapped from TypeScript type system, which is much richer than the Java one (mostly because JavaScript is a dynamic language and requires more typing scenarios than Java).

Functional types

For functional types, JSweet reuses the `java.Runnable` and `java.util.function` functional interfaces of Java 8. These interfaces are generic but only support up to 2-parameter functions. Thus, JSweet adds some support for more parameters in `jsweet.util.function`, since it is a common case in JavaScript APIs.

Here is an example using the `Function` generic functional type:

```
import java.util.function.Function;

public class C {

    String test(Function<String, String> f) {
        f.apply("a");
    }

    public static void main(String[] args) {
        String s = new C().test(p -> p);
        assert s == "a";
    }
}
```

We encourage programmers to use the generic functional interfaces defined in the `jsweet.util.function` and `java.util.function` (besides `java.lang.Runnable`). When requiring functions with more parameters, programmers can define their own generic functional types in `jsweet.util.function` by following the same template as the existing ones.

In some cases, programmers will prefer defining their own specific functional interfaces. This is supported by JSweet. For example:

```
@FunctionalInterface
interface MyFunction {
    void run(int i, String s);
}

public class C {
    void m(MyFunction f) {
        f.run(1, "test");
    }
    public static void main(String[] args) {
        new C().m((i, s) -> {
            // do something with i and s
        });
    }
}
```

Important warning: it is to be noted here that, on contrary to Java, the use of the `@FunctionInterface` annotation is mandatory.

Note also the possible use of the `apply` function, which is by convention always a functional definition on the target object (unless if `apply` is annotated with the `@Name` annotation). Defining/invoking `apply` can done on any class/object (because in JavaScript any object can become a functional object).

Object types

Object types are similar to interfaces: they define a set of fields and methods that are applicable to an object (but remember that it is a compile-time contract). In TypeScript, object types are inlined and anonymous. For instance, in TypeScript, the following method `m` takes a parameter, which is an object containing an `index` field:

```
// TypeScript:
public class C {
    public m(param : { index : number }) { ... }
}
```

Object types are a convenient way to write shorter code. One can pass an object that is correctly typed by constructing an object on the fly:

```
// TypeScript:
var c : C = ...;
c.m({ index : 2 });
```

Obviously, object types are a way to make the typing of JavaScript programs very easy to programmers, which is one of the main goals of TypeScript. It makes the typing concise, intuitive and straightforward to JavaScript programmers. In Java/JSweet, no similar inlined types exist and Java programmers are used to defining classes or interfaces for such cases. So, in JSweet, programmers have to define auxiliary classes annotated with `@ObjectType` for object types. This may seem more complicated, but it has the advantage to force the programmers to name all the types, which, in the end, can lead to more readable and maintenable code depending on the context. Note that similarily to interfaces, object types are erased at runtime. Also `@ObjectType` annotated classes can be inner classes so that they are used locally.

Here is the JSweet version of the previous TypeScript program.

```
public class C {
    @ObjectType
    public static class Indexed {
        int index;
    }
    public void m(Indexed param) { ... }
}
```

Using an object type is similar to using an interface:

```
C c = ...;
c.m(new Indexed() {{ index = 2; }});
```

When object types are shared objects and represent a typing entity that can be used in several contexts, it is recommended to use the `@Interface` annotation instead of `@ObjectType`. Here is the interface-based version.

```
@Interface
public class Indexed {
    int index;
}

public class C {
    public m(Indexed param) { ... }
}

C c = ...;
c.m(new Indexed {{ index = 2; }});
```

String types

In TypeScript, string types are a way to simulate function overloading depending on the value of a string parameter. For instance, here is a simplified excerpt of the DOM TypeScript definition file:

```
// TypeScript:
interface Document {
    [...]
    getElementsByTagName(tagname: "a"): NodeListOf<HTMLAnchorElement>;
    getElementsByTagName(tagname: "b"): NodeListOf<HTMLPhraseElement>;
    getElementsByTagName(tagname: "body"): NodeListOf<HTMLBodyElement>;
    getElementsByTagName(tagname: "button"): NodeListOf<HTMLButtonElement>;
    [...]
}
```

In this code, the `getElementsByTagName` functions are all overloads that depend on the strings passed to the `tagname` parameter. Not only string types allow function overloading (which is in general not allowed in TypeScript/JavaScript), but they also constrain the string values (similarly to an enumeration), so that the compiler can automatically detect typos in string values and raise errors.

This feature being useful for code quality, JSweet provides a mechanism to simulate string types with the same level of type safety. A string type is a public static field annotated with `@StringType`. It must be typed with an interface of the same name declared in the same container type.

For JSweet translated libraries (candies), all string types are declared in a the `jsweet.util.StringTypes` class, so that it is easy for the programmers to find them. For instance, if a `"body"` string type needs to be defined, a Java interface called `body` and a static final field called `body` are defined in a `jsweet.util.StringTypes`.

Note that each candy may have its own string types defined in the `jsweet.util.StringTypes` class. The JSweet transpiler merges all these classes at the bytecode level so that all the string types of all candies are

available in the same `jsweet.util.StringTypes` utility class. As a result, the JSweet DOM API will look like:

```
@Interface
public class Document {
    [...]
    public native NodeListOf<HTMLAnchorElement> getElementsByTagName(a tagname);
    public native NodeListOf<HTMLPhraseElement> getElementsByTagName(b tagname);
    public native NodeListOf<HTMLBodyElement> getElementsByTagName(body tagname);
    public native NodeListOf<HTMLButtonElement> getElementsByTagName(button tagname);
    [...]
}
```

In this API, `a`, `b`, `body` and `button` are interfaces defined in the `jsweet.util.StringTypes` class. When using one the method of `Document`, the programmer just need to use the corresponding type instance (of the same name). For instance:

```
Document doc = ...;
NodeListOf<HTMLAnchorElement> elts = doc.getElementsByTagName(StringTypes.a);
```

Note: if the string value is not a valid Java identifier (for instance `"2d"` or `"string-with-dashes"`), it is then translated to a valid one and annotated with `@Name("originalName")`, so that the JSweet transpiler knows what actual string value must be used in the generated code. For instance, by default, `"2d"` and `"string-with-dashes"` will correspond to the interfaces `StringTypes._2d` and `StringTypes.string_with_dashes` with `@Name` annotations.

Programmers can define string types for their own needs, as shown below:

```
import jsweet.lang.Erased;
import jsweet.lang.StringType;

public class CustomStringTypes {
    @Erased
    public interface abc {}

    @StringType
    public static final abc abc = null;

    // This method takes a string type parameter
    void m2(abc arg) {
    }

    public static void main(String[] args) {
        new CustomStringTypes().m2(abc);
    }
}
```

Note the use of the `@Erased` annotation, which allows the declaration of the

`abc` inner interface. This interface is used to type the string type field `abc`. In general, we advise the programmer to group all the string types of a program in the same utility class so that it is easy to find them.

Tuple types

Tuple types represent JavaScript arrays with individually tracked element types. For tuple types, JSweet defines parameterized auxiliary classes `TupleN<T0, ... TN-1>`, which define `$0`, `$1`, ... `$N-1` public fields to simulate typed array accessed (field `$i` is typed with `Ti`).

For instance, given the following tuple of size 2:

```
Tuple2<String, Integer> tuple = new Tuple2<String, Integer>("test", 10);
```

We can expect the following (well-typed) behavior:

```
assert tuple.$0 == "test";
assert tuple.$1 == 10;
tuple.$0 = "ok";
tuple.$1--;
assert tuple.$0 == "ok";
assert tuple.$1 == 9;
```

Tuple types are all defined (and must be defined) in the `jsweet.util.tuple` package. By default classes `Tuple[2..6]` are defined. Other tuples (> 6) are automatically generated when encountered in the candy APIs. Of course, when requiring larger tuples that cannot be found in the `jsweet.util.tuple` package, programmers can add their own tuples in that package depending on their needs, just by following the same template as existing tuples.

Union types

Union types represent values that may have one of several distinct representations. When such a case happens within a method signature (for instance a method allowing several types for a given parameter), JSweet takes advantage of the *method overloading* mechanism available in Java. For instance, the following `m` method accept a parameter `p`, which can be either a `String` or a `Integer`.

```
public void m(String p) {...}
public void m(Integer p) {...}
```

In the previous case, the use of explicit union types is not required. For more general cases, JSweet defines an auxiliary interface `Union<T1, T2>` (and `UnionN<T1, ... TN>`) in the `jsweet.util.union` package. By using this auxiliary type and a `union` utility method, programmers can cast back and forth between union types and union-ed type, so that JSweet can ensure similar properties as TypeScript union types.

The following code shows a typical use of union types in JSweet. It simply declares a variable as a union between a string and a number, which means that the variable can actually be of one of that types (but of no other types). The switch from a union type to a regular type is done through the `jsweet.util.Lang.union` helper method. This helper method is completely untyped, allowing from a Java perspective any union to be transformed to another type. It is actually the JSweet transpiler that checks that the union type is consistently used.

```
import static jsweet.util.Lang.union;
import jsweet.util.union.Union;
[...]
Union<String, Number> u = ...;
// u can be used as a String
String s = union(u);
// or a number
Number n = union(u);
// but nothing else
Date d = union(u); // JSweet error
```

The `union` helper can also be used the other way, to switch from a regular type back to a union type, when expected.

```
import static jsweet.util.Lang.union;
import jsweet.util.union.Union3;
[...]
public void m(Union3<String, Number, Date>> u) { ... }
[...]
// u can be a String, a Number or a Date
m(union("a string"));
// but nothing else
m(union(new RegExp(".*"))); // compile error
```

Note: the use of Java function overloading is preferred over union types when typing function parameters. For example:

```
// with union types (discouraged)
native public void m(Union3<String, Number, Date>> u);
// with overloading (preferred way)
native public void m(String s);
native public void m(Number n);
native public void m(Date d);
```

Intersection types

TypeScript defines the notion of type intersection. When types are intersected, it means that the resulting type is a larger type, which is the sum of all the

intersected types. For instance, in TypeScript, `A & B` corresponds to a type that defines both `A` and `B` members.

Intersection types in Java cannot be implemented easily for many reasons. So, the practical choice being made here is to use union types in place of intersection types. In JSweet, `A & B` is thus defined as `Union<A, B>`, which means that the programmer can access both `A` and `B` members by using the `jsweet.util.Lang.union` helper method. It is of course less convenient than the TypeScript version, but it is still type safe.

Semantics

Semantics designate how a given program behaves when executed. Although JSweet relies on the Java syntax, programs are transpiled to JavaScript and do not run in a JRE. As a consequence, the JavaScript semantics will impact the final semantics of a JSweet program compared to a Java program. In this section, we discuss the semantics by focusing on differences or commonalities between Java/JavaSript and JSweet.

Main methods

Main methods are the program execution entry points and will be invoked globally when a class containing a `main` method is evaluated. For instance:

```
public class C {
    private int n;
    public static C instance;
    public static void main(String[] args) {
        instance = new C();
        instance.n = 4;
    }
    public int getN() {
        return n;
    }
}
// when the source file containing C has been evaluated:
assert C.instance != null;
assert C.instance.getN() == 4;
```

The way main methods are globally invoked depends on how the program is packaged. See the appendixes for more details.

Initializers

Initializers behave like in Java.

For example:

```
public class C1 {
    int n;
    {
        n = 4;
    }
}
assert new C1().n == 4;
```

And similarly with static initializers:

```
public class C2 {
    static int n;
    static {
        n = 4;
    }
}
assert C2.n == 4;
```

While regular initializers are evaluated when the class is instantiated, static initializers are lazily evaluated in order to avoid forward-dependency issues, and mimic the Java behavior for initializers. With JSweet, it is possible for a programmer to define a static field or a static intializer that relies on a static field that has not yet been initialized.

More details on this behavior can be found in the appendixes.

Arrays initialization and allocation

Arrays can be used like in Java.

```
String[] strings = { "a", "b", "c" };
assert strings[1] == "b";
```

When specifying dimensions, arrays are pre-allocated (like in Java), so that they are initialized with the right length, and with the right sub-arrays in case of multiple-dimensions arrays.

```
String[][] strings = new String[2][2];
assert strings.length == 2;
assert strings[0].length == 2;
strings[0][0] = "a";
assert strings[0][0] == "a";
```

The JavaScript API can be used on an array by casting to a `def.js.Array` with `jsweet.util.Lang.array`.

```
import static jsweet.util.Lang.array;
[...]
String[] strings = { "a", "b", "c" };
assert strings.length == 3;
```

```
array(strings).push("d");
assert strings.length == 4;
assert strings[3] == "d";
```

In some cases it is preferable to use the `def.js.Array` class directly.

```
Array<String> strings = new Array<String>("a", "b", "c");
// same as: Array<String> strings = array(new String[] { "a", "b", "c" });
// same as: Array<String> strings = new Array<String>(); strings.push("a", "b", "c");
assert strings.length == 3;
strings.push("d");
assert strings.length == 4;
assert strings.$get(3) == "d";
```

Asynchronous programming

JSweet supports advanced asynchronous programming beyond the basic callback concepts with the help of the ES2015+ Promise API.

Promises

It is very simple to define an asynchronous method by declaring a `Promise` return type. The following method's `Promise` will be *fulfilled* when millis milliseconds elapsed.

```
Promise<Void> delay(int millis) {
  return new Promise<Void>((Consumer<Void> resolve, Consumer<Object> reject) -> {
    setTimeout(resolve, millis);
  });
}
```

You can then chain synchronous and asynchronous actions to be executed once the promise is fulfilled.

```
delay(1000)
  // chain with a synchronous action with "then". Here we just return a constant.
  .then(() -> {
    System.out.println("wait complete");
    return 42;
  })
  // chain with an asynchronous action with "thenAsync". Here it is implied that anotherAsyn
  .thenAsync((Integer result) -> {
    System.out.println("previous task result: " + result); // will print "previous task resu

    return anotherAsyncAction("param");
  })
  // this chained action will be executed once anotherAsyncAction finishes its execution.
  .then((String result) -> {
```

```
    System.out.println("anotherAsyncAction returned " + result);
  })
  // catch errors during process using this method
  .Catch(error -> {
    System.out.println("error is " + error);
  });
```

This allows a totally type-safe and fluent asynchronous programming model.

async/await

`Promise`s are really interesting to avoid callback but writing it still requires a lot of boilerplate code. It is better than pure callbacks but less readable and straightforward than linear programming. That's where `async/await` comes to help.

With the `await` keyword, you can tell the runtime to wait for a `Promise` to be fulfilled without having to write a then method. The code after the `await` "is" the `then` part. The result is that you can write your asynchronous code with linear programming.

```
import static jsweet.util.Lang.await;

// wait for the Promise returned by the delay method to be fulfilled
await(delay(1000));

System.out.println("wait complete");
```

It goes the same for error handling. You can just use the plain old `try / catch` idiom to handle your exceptions.

```
import static jsweet.util.Lang.await;
import def.js.Error;

try {
  Integer promiseResult = await(getANumber());
  assert promiseResult == 42;
} catch(Error e) {
  System.err.println("something unexpected happened: " + e);
}
```

You have to declare as `async` every asynchronous method / lambda (i.e. every method which would await something).

```
import static jsweet.util.Lang.await;
import static jsweet.util.Lang.async;
import static jsweet.util.Lang.function;
```

```
import jsweet.lang.Async;
import def.js.Function;

@Async
Promise<Integer> findAnswer() {
  await(delay(1000)); // won't compile if the enclosing method isn't @Async
  return asyncReturn(42); // converts to Promise
}

@Async
void askAnswerThenVerifyAndPrintIt() {

  try {

    Integer answer = await(findAnswer());

    // lambda expressions can be async
    Function verifyAnswerAsync = async(function(() -> {
      return await(answerService.verifyAnswer(answer));
    }))

    Boolean verified = await(verifyAnswerAsync.$apply());

    if (!verified) {
      throw new Error("cannot verify this answer");
    }

    console.log("answer found: " + answer);

  } catch (Error e) {
    console.error(e, "asynchronous process failed");
  }
}
```

Sweet, isn't it? ;)

Name clashes

On contrary to TypeScript/JavaScript, Java makes a fundamental difference between methods, fields, and packages. Java also support method overloading (methods having different signatures with the same name). In JavaScript, object variables and functions are stored within the same object map, which basically means that you cannot have the same key for several object members (this also explains that method overloading in the Java sense is not possible in JavaScript). Because of this, some Java code may contain name clashes when generated as is in TypeScript. JSweet will avoid name clashes automatically when possible,

and will report sound errors in the other cases.

Methods and fields names clashes

JSweet performs a transformation to automatically allow methods and private fields to have the same name. On the other hand, methods and public fields of the same name are not allowed within the same class or within classes having a subclassing link.

To avoid programming mistakes due to this JavaScript behavior, JSweet adds a semantics check to detect duplicate names in classes (this also takes into account members defined in parent classes). As an example:

```
public class NameClashes {

    // error: field name clashes with existing method name
    public String a;

    // error: method name clashes with existing field name
    public void a() {
        return a;
    }

}
```

Method overloading

On contrary to TypeScript and JavaScript (but similarly to Java), it is possible in JSweet to have several methods with the same name but with different parameters (so-called overloads). We make a distinction between simple overloads and complex overloads. Simple overloading is the use of method overloading for defining optional parameters. JSweet allows this idiom under the condition that it corresponds to the following template:

```
String m(String s, double n) { return s + n; }
// valid overloading (JSweet transpiles to optional parameter)
String m(String s) { return m(s, 0); }
```

In that case, JSweet will generate JavaScript code with only one method having default values for the optional parameters, so that the behavior of the generated program corresponds to the original one. In this case:

```
function m(s, n = 0) { return s + n; }
```

If the programmer tries to use overloading differently, for example by defining two different implementations for the same method name, JSweet will fallback on a complex overload, which consists of generating a root implementation (the method that hold the more parameters) and one subsidiary implementation per

overloading method (named with a suffix representing the method signature). The root implementation is generic and dispatches to other implementations by testing the values and types of the given parameters. For example:

```
String m(String s, double n) { return s + n; }
String m(String s) { return s; }
```

Generates the following (slightly simplified) JavaScript code:

```
function m(s, n) {
    if(typeof s === 'string' && typeof n === 'number') {
        return s + n;
    } else if(typeof s === 'string' && n === undefined) {
        return this.m$java_lang_String(s);
    } else {
        throw new Error("invalid overload");
    }
}

function m$java_lang_String(s) { return s; }
```

Local variable names

In TypeScript/JavaScript, local variables can clash with the use of a global method. For instance, using the `alert` global method from the DOM (`jsweet.dom.Globals.alert`) requires that no local variable hides it:

```
import static jsweet.dom.Globals.alert;

[...]

public void m1(boolean alert) {
    // JSweet compile error: name clash between parameter and method call
    alert("test");
}

public void m2() {
    // JSweet compile error: name clash between local variable and method call
    String alert = "test";
    alert(alert);
}
```

Note that this problem also happens when using fully qualified names when calling the global methods (that is because the qualification gets erased in TypeScript/JavaScript). In any case, JSweet will report sound errors when such problems happen so that programmers can adjust local variable names to avoid clashes with globals.

Testing the type of an object

To test the type of a given object at runtime, one can use the `instanceof` Java operator, but also the `Object.getClass()` function.

`instanceof`

The `instanceof` is the advised and preferred way to test types at runtime. JSweet will transpile to a regular `instanceof` or to a `typeof` operator depending on the tested type (it will fallback on `typeof` for `number`, `string`, and `boolean` core types).

Although not necessary, it is also possible to directly use the `typeof` operator from JSweet with the `jsweet.util.Lang.typeof` utility method. Here are some examples of valid type tests:

```
import static jsweet.util.Lang.typeof;
import static jsweet.util.Lang.equalsStrict;
[...]
Number n1 = 2;
Object n2 = 2;
int n3 = 2;
Object s = "test";
MyClass c = new MyClass();

assert n1 instanceof Number; // transpiles to a typeof
assert n2 instanceof Number; // transpiles to a typeof
assert n2 instanceof Integer; // transpiles to a typeof
assert !(n2 instanceof String); // transpiles to a typeof
assert s instanceof String; // transpiles to a typeof
assert !(s instanceof Integer); // transpiles to a typeof
assert c instanceof MyClass;
assert typeof(n3) == "number";
```

From JSweet version 1.1.0, the `instanceof` operator is also allowed on interfaces, because JSweet keeps track of all the implemented interfaces for all objects. This interface tracking is ensured through an additional hidden property in the objects called `__interfaces` and containing the names of all the interfaces implemented by the objects (either directly or through its class inheritance tree determined at compile time). So, in case the type argument of the `instanceof` operator is an interface, JSweet simply checks out if the object's `__interfaces` field exists and contains the given interface. For example, this code is fully valid in JSweet when `Point` is an interface:

```
Point p1 = new Point() {{ x=1; y=1; }};
[...]
assert p1 instanceof Point
```

`Object.getClass()` and `X.class`

In JSweet, using the `Object.getClass()` on any instance is possible. It will actually return the constructor function of the class. Using `X.class` will also return the constructor if `X` is a class. So the following assertion will hold in JSweet:

```
String s = "abc";
assert String.class == s.getClass()
```

On a class, you can call the `getSimpleName()` or `getName()` functions.

```
String s = "abc";
assert "String" == s.getClass().getSimpleName()
assert String.class.getSimpleName() == s.getClass().getSimpleName()
```

Note that `getSimpleName()` or `getName()` functions will also work on an interface. However, you have to be aware that `X.class` will be encoded in a string (holding the interface's name) if `X` is is an interface.

Limitations and constraints

Since all numbers are mapped to JavaScript numbers, JSweet make no distinction between integers and floats for example. So, `n instanceof Integer` and `n instanceof Float` will always give the same result whatever the actual type of `n` is. The same limitation exists for strings and chars, which are not distinguishable at runtime, but also for functions that have the same number of parameters. For example, an instance of `IntFunction<R>` will not be distinguishable at runtime from a `Function<String,R>`.

These limitations have a direct impact on function overloading, since overloading uses the `instanceof` operator to decide which overload to be called.

Like it is usually the case when working in JavaScript, serialized objects must be properly "revived" with their actual classes so that the `instanceof` operator can work again. For example a point object created through `Point p = (Point)JSON.parse("{x:1,y:1}")` will not work with regard to the `instanceof` operator. In case you meet such a use case, you can contact us to get some useful JSweet code to properly revive object types.

Variable scoping in lambda expressions

JavaScript variable scoping is known to pose some problems to the programmers, because it is possible to change the reference to a variable from outside of a lambda that would use this variable. As a consequence, a JavaScript programmer cannot rely on a variable declared outside of a lambda scope, because when the lambda is executed, the variable may have been modified somewhere else in the program. For instance, the following program shows a typical case:

```
NodeList nodes = document.querySelectorAll(".control");
for (int i = 0; i < nodes.length; i++) {
    HTMLElement element = (HTMLElement) nodes.$get(i); // final
    element.addEventListener("keyup", (evt) -> {
        // this element variable will not change here
        element.classList.add("hit");
    });
}
```

In JavaScript (note that EcmaScript 6 fixes this issue), such a program would fail its purpose because the `element` variable used in the event listener is modified by the for loop and does not hold the expected value. In JSweet, such problems are dealt with similarly to final Java variables. In our example, the `element` variable is re-scoped in the lambda expression so that the enclosing loop does not change its value and so that the program behaves like in Java (as expected by most programmers).

Scope of *this*

On contrary to JavaScript and similarly to Java, using a method as a lambda will prevent loosing the reference to `this`. For instance, in the `action` method of the following program, `this` holds the right value, even when `action` was called as a lambda in the `main` method. Although this seem logical to Java programmers, it is not a given that the JavaScript semantics ensures this behavior.

```
package example;
import static jsweet.dom.Globals.console;

public class Example {
    private int i = 8;
    public Runnable getAction() {
        return this::action;
    }
    public void action() {
        console.log(this.i); // this.i is 8
    }
    public static void main(String[] args) {
        Example instance = new Example();
        instance.getAction().run();
    }
}
```

It is important to stress that the `this` correct value is ensured thanks to a similar mechanism as the ES5 `bind` function. A consequence is that function references are wrapped in functions, which means that function pointers (such as `this::action`) create wrapping functions on the fly. It has side effects when

manipulating function pointers, which are well described in this issue https://github.com/cincheo/jsweet/issues/65.

Packaging

Packaging is one of the complex point of JavaScript, especially when coming from Java. Complexity with JavaScript packaging boils down to the fact that JavaScript did not define any packaging natively. As a consequence, many *de facto* solutions and guidelines came up along the years, making the understanding of packaging uneasy for regular Java programmers. JSweet provides useful options and generates code in order to simplify the life of Java programmers by making the packaging issues much more transparent and as "easy" as in Java for most cases. In this section, we will describe and explain typical packaging scenarios.

Use your files without any packaging

The most common and simple case for running a program is just to include each generated file in an HTML page. This is the default mode when not precising any packaging options. For example, when your program defines two classes `x.y.z.A` and `x.y.z.B` in two separated files, you can use them as following:

```
<script type="text/javascript" src="target/js/x/y/z/A.js"></script>
<script type="text/javascript" src="target/js/x/y/z/B.js"></script>
[...]
<!-- access a method later in the file -->
<script type="text/javascript">x.y.z.B.myMethod()</script>
```

When doing so, programmers need to be extremely cautious to avoid forward static dependencies between the files. In other words, the `A` class cannot use anything from `B` in static fields, static initializers, or static imports, otherwise leading to runtime errors when trying to load the page. Additionally, the `A` class cannot extend the `B` class. These constraints come from JavaScript/TypeScript and have nothing to do with JSweet.

As you can imagine, running simple programs with this manual technique is fine, but can become really uncomfortable for developing complex applications. Complex applications most of the time bundle and/or package the program with appropriate tools in order to avoid having to manually handle dependencies between JavaScript files.

Creating a bundle for a browser

To avoid having to take care of the dependencies manually, programmers use bundling tools to bundle up their classes into a single file. Such a bundle is included in any web page using something like this:

```
<script type="text/javascript" src="target/js/bundle.js"></script>
[...]
<!-- access a method later in the file -->
<script type="text/javascript">x.y.z.B.myMethod()</script>
```

JSweet comes with such a bundling facility. To create a bundle file, just set to `true` the `bundle` option of JSweet. Note that you can also set to `true` the `declaration` option that will ask JSweet to generate the TypeScript definition file (`bundle.d.ts`). This file allows you to use/compile your JSweet program from TypeScript in a well-typed way.

The "magic" with JSweet bundling option is that it analyzes the dependencies in the source code and takes care of solving forward references when building the bundle. In particular, JSweet implements a lazy initialization mechanism for static fields and initializers in order to break down static forward references across the classes. There are no specific additional declarations to be made by the programmers to make it work (on contrary to TypeScript).

Note that there are still some minor limitations to it (when using inner and anonymous classes for instance), but these limitations will be rarely encountered and will be removed in future releases.

Note also that JSweet will raise an error if you specify the `module` option along with the `bundle` option.

Packaging with modules

First, let us start by explaining modules and focus on the difference between Java *packages* (or TypeScript *namespaces*) and *modules*. If you feel comfortable with the difference, just skip this section.

Packages and modules are two similar concepts but for different contexts. Java packages must be understood as compile-time *namespaces*. They allow a compile-time structuration of the programs through name paths, with implicit or explicit visibility rules. Packages have usually not much impact on how the program is actually bundled and deployed.

Modules must be understood as deployment / runtime "bundles", which can be `required` by other modules. The closest concept to a module in the Java world would probably be an OSGi bundle. A module defines imported and exported elements so that they create a strong runtime structure that can be used for deploying software components independently and thus avoiding name clashes. For instance, with modules, two different libraries may define a `util.List` class and be actually running and used on the same VM with no naming issues (as long as the libraries are bundled in different modules).

Nowadays, a lot of libraries are packaged and accessible through modules. The standard way to use modules in a browser is the AMD, but in Node.js it is the commonjs module system.

Modules in JSweet

JSweet supports AMD, commonjs, and UMD module systems for packaging. JSweet defines a `module` option (value: `amd`, `commonjs` or `umd`). When specifying this option, JSweet automatically creates a default module organization following the simple rule: one file = one module.

For example, when packaged with the `module` option set to `commonjs`, one can write:

```
> node target/js/x/y/z/MyMainClass.js
```

Where `MyMainClass` contains a `main` method.

The module system will automatically take care of the references and require other modules when needed. Under the hood, JSweet analysis the Java import statements and transform them to `require` instructions.

Note: once the program has been compiled with the `module` option, it is easy to package it as a bundle using appropriate tools such as Browserify, which would give similar output as using the `bundle` option of JSweet. Note also that JSweet will raise an error when specifying both `module` and `bundle`, which are exclusive options.

External modules

When compiling JSweet programs with the `module` options, all external libraries and components must be required as external modules. JSweet can automatically require modules, simply by using the `@Module(name)` annotation. In JSweet, importing or using a class or a member annotated with `@Module(name)` will automatically require the corresponding module at runtime. Please not that it is true only when the code is generated with the `module` option. If the `module` option is off, the `@Module` annotations are ignored.

```
package def.jquery;
public final class Globals extends def.js.Object {
    ...
    @jsweet.lang.Module("jquery")
    native public static def.jquery.JQuery $(java.lang.String selector);
    ...
}
```

The above code shows an excerpt of the JSweet jQuery API. As we can notice, the `$` function is annotated with `@Module("jquery")`. As a consequence, any call to this function will trigger the require of the `jquery` module.

Note: the notion of manual require of a module may be available in future releases. However, automatic require is sufficient for most programmers and hides the complexity of having to require modules explicitly. It also brings the advantage of having the same code whether modules are used or not.

Troubleshooting: when a candy does not define properly the `@Module` annotation, it is possible to force the declaration within the comment of a special file called `module_defs.java`. For example, to force the `BABYLON` namespace of the Babylonjs candy to be exported as a `babylonjs` module, you can write the following file:

```
package myprogram;
// declare module "babylonjs" {
//    export = BABYLON;
// }
```

Note that a JSweet project can only define one `module_defs.java` file, which shall contain all the module declarations in a comment. Note also that it is a hack and the preferred method would be to contribute to the candy to fix the problem.

Root packages

Root packages are a way to tune the generated code so that JSweet packages are erased in the generated code and thus at runtime. To set a root package, just define a `package-info.java` file and use the `@Root` annotation on the package, as follows:

```
@Root
package a.b.c;
```

The above declaration means that the `c` package is a root package, i.e. it will be erased in the generated code, as well as all its parent packages. Thus, if `c` contains a package `d`, and a class `C`, these will be top-level objects at runtime. In other words, `a.b.c.d` becomes `d`, and `a.b.c.C` becomes `C`.

Note that since that packaged placed before the `@Root` package are erased, there cannot be any type defined before a `@Root` package. In the previous example, the *a* and *b* packages are necessarily empty packages.

Behavior when not using modules (default)

By default, root packages do not change the folder hierarchy of the generated files. For instance, the `a.b.c.C` class will still be generated in the `<jsout>/a/b/c/C.js` file (relatively to the `<jsout>` output directory). However, switching on the `noRootDirectories` option will remove the root directories so that the `a.b.c.C` class gets generated to the `<jsout>/C.js` file.

When not using modules (default), it is possible to have several `@Root` packages (but a `@Root` package can never contain another `@Root` package).

Behavior when using modules

When using modules (see the *module* option), only one `@Root` package is allowed, and when having one `@Root` package, no other package or type can be outside of the scope of that `@Root` package. The generated folder/file hierarchy then starts at the root package so that all the folders before it are actually erased.

Packaging a JSweet jar (candy)

A candy is a Maven artifact that contains everything required to easily access a JavaScript library from a JSweet client program. This library can be an external JavaScript library, a TypeScript program, or another JSweet program.

Anatomy of a candy

Like any Maven artifact, a candy has a group id, a artifact id (name), and a version. Besides, a typical candy should contain the following elements:

1. The compiled Java files (*.class), so that your client program that uses the candy can compile.
2. A `META-INF/candy-metadata.json` file that contains the expected target version of the transpiler (to be adapted to your target transpiler version).
3. The program's declarations in `d.ts` files, to be placed in the `src/typings` directory of the jar. Note that these definitions are not mandatory if you intend to use JSweet for generating TypeScript source code (`tsOnly` option). In that case, you may delegate the JavaScript generation to an external `tsc` compiler and access the TypeScript definitions from another source.
4. Optionally, the JavaScript bundle of the library, which can in turn be automatically extracted and used by the JSweet client programs. JSweet expects the JavaScript to be packaged following the Webjars conventions: http://www.webjars.org/. When packaged this way, a JSweet transpiler using your candy will automatically extract the bundled JavaScript in a directory given by the `candiesJsOut` option (default: `js/candies`).

Here is an example of the `META-INF/candy-metadata.json` file:

```
{
   "transpilerVersion": "2.0.0"
}
```

How to create a candy from a JSweet program

A typical use case when building applications with JSweet, is to share a common library or module between several other JSweet modules/applications. Note that since a JSweet candy is a regular Maven artifact, it can also be used by a regular Java program as long as it does not use any JavaScript APIs.

So, a typical example in a project is to have a *commons* library containing DTOs and common utility functions, which can be shared between a Web client written in JSweet (for example using the angular or knockout libraries) and a mobile client written also in JSweet (for example using the ionic library). The great news is that this *commons* library can also be used by the Java server (JEE, Spring, ...) as is, because the DTOs do not use any JavaScript, and that the compiled Java code packaged in the candy can run on a Java VM. This this extremely helpful, because it means that when you develop this project in your favorite IDE, you will be able to refactor some DTOs and common APIs, and it will directly impact your Java server code, your Web client code, and your mobile client code!

We provide a quick start project to help you starting with such a use case: https://github.com/cincheo/jsweet-candy-quickstart

How to create a candy for an existing JavaScript or TypeScript library

We provide a quick start project to help you starting with such a use case: https://github.com/cincheo/jsweet-candy-js-quickstart

Extending the transpiler

JSweet is an Open Transpiler from Java to TypeScript. It means that it provides ways for programmers to tune/extend how JSweet generates the intermediate TypeScript code. Tuning the transpiler is a solution to avoid repetitive tasks and automatize them.

For instance, say you have a legacy Java code that uses the Java API for serializing objects (`writeObject`/`readObject`). With JSweet, you can easily erase these methods from your program, so that the generated JavaScript code is free from any Java-specific serialization idioms.

As another example, say you have a Java legacy code base that uses a Java API, which is close to (but not exactly) a JavaScript API you want to use in your final JavaScript code. With JSweet, you can write an adapter that will automatically map all the Java calls to the corresponding JavaScript calls.

Last but not least, you can tune JSweet to take advantage of some specific APIs depending on the context. For instance, you may use ES6 maps if you know that your targeted browser supports them, or just use an emulation or a simpler implementation in other cases. You may adapt the code to avoid using some canvas or WebGL primitives when knowing they are not well supported for a given mobile browser. An so on... Using an Open Transpiler such as JSweet has many practical applications, which you may or may not have to use, but in any case it is good to be aware of what is possible.

Tuning can be done declaratively (as opposed to programmatically) using annotations. Annotations can be added to the Java program (hard-coded), or they can be centralized in a unique configuration file so that they don't even appear in the Java code (we call them soft annotations). Using annotations is quite simple and intuitive. However, when complex customization of the transpiler is required, it is most likely that annotations are not sufficient anymore. In that case, programmers shall use the JSweet extension API, which entails all the tuning that can be done with annotations, and much more. The extension API gives access to a Java AST (Abstract Syntax Tree) within the context of so-called *printer adapters*. Printer adapters follow a decorator pattern so that they can be chained to extend and/or override the way JSweet will print out the intermediate TypeScript code.

Core annotations

The package `jsweet.lang` defines various annotations that can be used to tune the way JSweet generates the intermediate TypeScript code. Here we explain these annotations and give examples on how to use them.

- `@Erased`: This annotation type is used on elements that should be erased at generation time. It can be applied to any program element. If applied to a type, casts and constructor invocations will automatically be removed, potentially leading to program inconsistencies. If applied to a method, invocations will be removed, potentially leading to program inconsistency, especially when the invocation's result is used in an expression. Because of potential inconsistencies, programmers should use this annotation carefully, and applied to unused elements (also erasing using elements).
- `@Root`: This package annotation is used to specify a root package for the transpiled TypeScript/JavaScript, which means that all transpiled references in this package and subpackages will be relative to this root package. As an example, given the `org.mycompany.mylibrary` root package (annotated with `@Root`), the class `org.mycompany.mylibrary.MyClass` will actually correspond to `MyClass` in the JavaScript runtime. Similarly, the `org.mycompany.mylibrary.mypackage.MyClass` will transpile to `mypackage.MyClass`.
- `@Name(String value)`: This annotation allows the definition of a name that will be used for the final generated code (rather than the Java name). It can be used when the name of an element is not a valid Java identifier. By convention, JSweet implements a built-in convention to save the use of @Name annotations: `Keyword` in Java transpiles to `keyword`, when `keyword` is a Java keyword (such as `catch`, `finally`, `int`, `long`, and so forth).
- `@Replace(String value)`: This annotation allows the programmer to substitute a method body implementation by a TypeScript implementa-

tion. The annotation's value contains TypeScript which is generated as is by the JSweet transpiler. The code will be checked by the TypeScript transpiler. The replacing code can contain variables substituted using a mustache-like convention (`{{variableName}}`). Here is the list of supported variables:

- `{{className}}`: the current class.
- `{{methodName}}`: the current method name.
- `{{body}}`: the body of the current method. A typical use of this variable is to wrap the original behavior in a lambda. For instance: `/* before code */ let _result = () => { {{body}} }(); /* after code */ return _result;`.
- `{{baseIndent}}`: the indentation of the replaced method. Can be used to generate well-formatted code.
- `{{indent}}`: substituted with an indentation. Can be used to generate well-formatted code.

Example

The following example illustrates the use of the `@Erased` and `@Replace` annotations. Here, the `@Erased` annotation is used to remove the `readObject` method from the generated code, because it does not make sense in JavaScript (it is a Java-serialization specific method). The `@Replace` annotation allows defining a direct TypeScript/JavaScript implementation for the `searchAddress` method.

```
class Person {

  List<String> addresses = new ArrayList<String>();

  @Erased
  private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException {
    [...]
  }

  @Replace("return this.addresses.filter(address => address.match(regex))[0]")
  public String searchAddress(String regex) {
          Optional<String> match = addresses.stream().filter(address -> address.matches(rege
          return match.isPresent()?match.get():null;
  }
}
```

Using JSweet annotations makes it possible to share classes between Java and JavaScript in a flexible way. Useless methods in JavaScript are erased, and some methods can have different implementations for Java and JavaScript.

Centralizing annotations in `jsweetconfig.json`

JSweet supports the definition of annotations within a unique configuration file (`jsweetconfig.json`). There are many reasons why programmers would want to define annotations within that file instead of defining annotations in the Java source code.

1. Annotations or annotation contents may differ depending on the context. It may be convenient to have different configuration files depending on the context. It is easier to switch a configuration file with another than having to change all the annotations in the program.
2. Adding annotations to the Java program is convenient to tune the program locally (on a given element). However, in some cases, similar annotations should apply on a set of program elements, in order to automatize global tuning of the program. In that case, it is more convenient to install annotations by using an expression, that will match a set of program elements at once. This mechanism is similar to the *pointcut* mechanism that can be found in Aspect Oriented Software Design. It allows capturing a global modification in a declarative manner.
3. Using annotations in the Java source code entails a reference to the JSweet API (the `jsweet.lang` package) that may be seen as an unwanted dependency for some programmers who want their Java code to remain as "pure" as possible.

The JSweet configuration file (`jsweetconf.json`) is a JSON file containing a list of configuration entries. Among the configuration entries, so-called global filters can be defined using the following structure:

```
<annotation>: {
      "include": <match_expressions>,
      "exclude": <match_expressions>
}
```

Where `<annotation>` is the annotation to be added, with potential parameters, and `include` and `exclude` are lists of match expressions. An element in the program will be annotated with the annotation, if its signature matches any of the expressions in the `include` list and does not match any the expressions in the `exclude` list.

A match expression is a sort of simplified regular expression, supporting the following wildcards:

1. `*` matches any token or token sub-part in the signature of the program element (a token is an identifier part of a signature, for instance `A.m(java.lang.String)` contains the tokens `A`, `m`, and `java.lang.String`).
2. `**` matches any list of tokens in signature of the program element.

3. `..` matches any list of tokens in signature of the program element. (same as `**`)
4. `!` negates the expression (first character only).

For example:

```
// all the elements and subelements (fields, methods, ...) in the x.y.z package
x.y.z.**

// all the methods in the x.y.z.A class
x.y.z.A.*(..)

// all the methods taking 2 arguments in the \texttt{x.y.z.A} class
x.y.z.A.*(*,*)

// all fields called aField in all the classes of the program
**.aField
```

Here is a more complete example with a full `jsweetconfig.json` configuration file.

```
{
  // all classes and packages in x.y.z will become top level
  "@Root": {
    "include": [ "x.y.z" ]
  },
  // do not generate any TypeScript code for Java-specific methods
  "@Erased": {
    "include": [ "**.writeObject(..)", "**.readObject(..)", "**.hashCode(..)" ]
  },
  // inject logging in all setters and getters of the x.y.z.A class
  "@Replace('console.info('entering {{methodName}}'); let _result = () => { {{body}} }(); co
    "include": [ "x.y.z.A.set*(*)", "x.y.z.A.get*()", "x.y.z.A.is*()" ]
  }
}
```

Note that annotations are defined with simple names only. That's because they are core JSweet annotations (defined in `jsweet.lang`). Non-core annotations can be added the same way, but the programmer must use fully qualified names.

Programmatic tuning with adapters

Declarative tuning through annotation rapidly hits limitations when tuning the generation for specific purposes (typically when supporting additional Java libraries). Hence, JSweet provides an API so that programmers can extend the way JSweet generates the intermediate TypeScript code. Writing such an adaptation program is similar to writing a regular Java program, except that it will

apply to your programs to transform them. As such, it falls into the category of so-called meta-programs (i.e. programs use other programs as data). Since programmers may write extensions that leads to invalid code, that is where it becomes really handy to have an intermediate compilation layer. If the generated code is invalid, the TypeScript to JavaScript compilation will raise errors, thus allowing the programmer to fix the extension code.

Introducing the extension API

The extension API is available in the `org.jsweet.transpiler.extension` package. It is based on a factory pattern (`org.jsweet.transpiler.JSweetFactory`) that allows the programmer to adapt all the main components of the transpiler by subclassing them. In practice, most adaptations can be done by creating new printer adapters, as subclasses of `org.jsweet.transpiler.extension.PrinterAdapter`. Adapters are the core extension mechanism because they are chainable and can be composed (it is a sort of decorator pattern). JSweet uses default adapters in a default adaptation chain and tuning JSweet will then consist in adding new adapters to the chain.

An adapter will typically perform three kinds of operations to tune the generated code:

1. Map Java types to TypeScript ones.
2. Add annotations to the program either in a declarative way (with global filters) or in a programmatic way (with annotation managers).
3. Override printing methods defined in `PrinterAdapter` in order to override the TypeScript core that is generated by default. Printing methods take program elements, which are based on the standard `javax.lang.model.element` API. It provides an extension of that API for program elements that are expressions and statements (`org.jsweet.transpiler.extension.model`).

The following template shows the typical sections when programming an adapter. First, an adapter must extend `PrinterAdapter` or any other adapter. It must define a constructor taking the parent adapter, which will be set by JSweet when inserting the adapter in the chain.

```
public class MyAdapter extends PrinterAdapter {

    public MyAdapter(PrinterAdapter parent) {
        super(parent);
        ...
```

In the constructor, an adapter typically maps Java types to TypeScript types.

```
        // will change the type in variable/parameters declarations
        addTypeMapping("AJavaType", "ATypeScriptType");
```

```
        // you may want to erase type checking by mapping to 'any'
        addTypeMapping("AJavaType2", "any");
        [...]
```

In the constructor, an adapter can also add annotations in a more flexible way than when using the `jsweetconfig.json` syntax.

```
        // add annotations dynamically to the AST, with global filters
        addAnnotation("jsweet.lang.Erased", //
                "**.readObject(..)", //
                "**.writeObject(..)", //
                "**.hashCode(..)");
        // or with annotation managers (see the Javadoc and the example below)
        addAnnotationManager(new AnnotationManager() { ... });
    }
```

Most importantly, an adapter can override substitution methods for most important AST elements. By overriding these methods, an adapter will change the way JSweet generates the intermediate TypeScript code. To print out code, you can use the `print` method, which is defined in the root `PrinterAdapter` class. For example, the following code will replace all `new AJavaType(...)` with `new ATypeScriptType(...)`.

```
    @Override
    public boolean substituteNewClass(NewClassElement newClass) {
        // check if the 'new' applies to the right class
        if ("AJavaType".equals(newClass.getTypeAsElement().toString())) {
            // the 'print' method will generate intermediate TypeScript code
            print("new ATypeScriptType(")
                    .printArgList(newClass.getArguments()).print(")");
            // once some code has been printed, you should return true to break
            // the adapter chain, so your code will replace the default one
            return true;
        }
        // if not substituted, delegate to the adapter chain
        return super.substituteNewClass(newClass);
    }
```

Most useful substitution method remains invocation substitution, which is typically used to map a Java API to a similar JavaScript API.

```
    @Override
    public boolean substituteMethodInvocation(MethodInvocationElement invocation) {
        // substitute potential method invocation here
        [...]
        // delegate to the adapter chain
        return super.substituteMethodInvocation(invocation);
    }
}
```

Note also a special method to insert code after a Java type has been printed out:

```
@Override
public void afterType(TypeElement type) {
    super.afterType(type);
    // insert whatever TypeScript you need here
    [...]
}
```

There are many applications to adapters (see the examples below). Besides tuning the code generation and supporting Java APIs at compile-time, adapters can also be used to raise errors when the compiled code does not conform to expected standards depending on the target context. Another very useful use case it to allow the generation of proxies. For instance one can write an adapter that will generate JavaScript stubs to invoke Java services deployed with JAX-RS.

Installing and activating adapters

Once you have written an adapter, you need to compile it and add it to the adapter chain. The simplest way to do it with JSweet is to put it in the `jsweet_extension` directory that you need to create at the root of your project JSweet. In that directory, you can directly add Java source files for adapters, that will be compiled by JSweet on the fly. For instance, you may add two custom adapters `CustomAdapter1.java` and `CustomAdapter2.java` in `jsweet_extension/com/mycompany/`.

Then, in order to activate that adapter, you just need to add the `jsweetconfig.json` file at the root of the project and define the `adapters` configuration option, like this:

```
{
  // JSweet will add the declared adapters at the beginning of the default
  // chain... you can add as many adapters as you need
  adapters: [ "com.mycompany.CustomAdapter1", "com.mycompany.CustomAdapter2" ]
}
```

Hello world adapter

Here, we will step through how to tune the JSweet generation to generate strings in place of dates when finding `java.util.Date` types in the Java program.

First, create the `HelloWorldAdapter.java` file in the `jsweet_extension` directory at the root of your project. Copy and paste the following code in that file:

```
import org.jsweet.transpiler.extension.PrinterAdapter;
public class HelloWorldAdapter extends PrinterAdapter {
    public HelloWorldAdapter(PrinterAdapter parent) {
        super(parent);
        addTypeMapping(java.util.Date.class.getName(), "string");
    }
}
```

Second, in the project's root directory, create the `jsweetconfig.json` file with the following configuration:

```
{
  adapters: [ "HelloWorldAdapter" ]
}
```

Done. Now you can just try this extension on the following simple Java DTO:

```
package source.extension;
import java.util.Date;
/**
 * A Hello World DTO.
 *
 * @author Renaud Pawlak
 */
public class HelloWorldDto {
    private Date date;
    /**
     * Gets the date.
     */
    public Date getDate() {
        return date;
    }
    /**
     * Sets the date.
     */
    public void setDate(Date date) {
        this.date = date;
    }
}
```

The generated code should look like:

```
/* Generated from Java with JSweet 2.XXX - http://www.jsweet.org */
namespace source.extension {
    /**
     * A Hello World DTO.
     *
     * @author Renaud Pawlak
     * @class
```

```
     */
    export class HelloWorldDto {
        /*private*/ date : string;
        public constructor() {
            this.date = null;
        }
        /**
         * Gets the date.
         * @return {string}
         */
        public getDate() : string {
            return this.date;
        }
        /**
         * Sets the date.
         * @param {string} date
         */
        public setDate(date : string) {
            this.date = date;
        }
    }
    HelloWorldDto["__class"] = "source.extension.HelloWorldDto";
}
```

Note that all the date types have been translated to strings as expected. By the way, note also the JSDoc support, which makes JSweet a powerful tool to create well-documented JavaScript APIs from Java (doc comments are also tunable in adapters!).

Extension examples

The following sections illustrate the use of JSweet adapters with 5 real-life examples. Most of these adapters are built-in with JSweet (in the `org.jsweet.transpiler.extension` package) and can just be activated by adding them to the adapter chain as explained above. If you want to modify the adapters, just copy-paste the code in the `jsweet_extension` directory and change the names.

Example 1: an adapter to rename private fields

This simple adapter renames non-public members by adding two underscores as a prefix. Note that this could be dangerous to use for protected fields if wanting to access them from subclasses declared in other JSweet projects. So you may want to use carefully or to modify the code for your own needs.

This adapter is a good example for demonstrating how to use annotation man-

agers. Annotation managers are used to add (soft) annotations to program elements driven by some Java code (programmatically). Annotation managers are added to the context and will be chained to other existing annotation managers (potentially added by other adapters). An annotation manager must implement the `manageAnnotation` method, that will tell if a given annotation should be added, removed, or left unchanged on a given element. If the annotation has parameters, an annotation manager shall implement the `getAnnotationValue` in order to specify the values.

In this example, the annotation manager adds the `@jsweet.lang.Name` annotation to all non-public elements in order to rename them and add the underscores to the initial name.

```
package org.jsweet.transpiler.extension;

import javax.lang.model.element.Element;
import javax.lang.model.element.ExecutableElement;
import javax.lang.model.element.Modifier;
import javax.lang.model.element.TypeElement;
import javax.lang.model.element.VariableElement;
import org.jsweet.transpiler.util.Util;

public class AddPrefixToNonPublicMembersAdapter extends PrinterAdapter {

    public AddPrefixToNonPublicMembersAdapter(PrinterAdapter parentAdapter) {
        super(parentAdapter);
        // add a custom annotation manager to the chain
        addAnnotationManager(new AnnotationManager() {

            @Override
            public Action manageAnnotation(Element element, String annotationType) {
                // add the @Name annotation to non-public elements
                return "jsweet.lang.Name".equals(annotationType)
                        && isNonPublicMember(element) ? Action.ADD : Action.VOID;
            }

            @Override
            public <T> T getAnnotationValue(Element element,
                    String annotationType, String propertyName,
                    Class<T> propertyClass, T defaultValue) {
                // set the name of the added @Name annotation (value)
                if ("jsweet.lang.Name".equals(annotationType) && isNonPublicMember(element))
                    return propertyClass.cast("__" + element.getSimpleName());
                } else {
                    return null;
                }
            }
```

```
            private boolean isNonPublicMember(Element element) {
                return (element instanceof VariableElement || element instanceof ExecutableE
                        && element.getEnclosingElement() instanceof TypeElement
                        && !element.getModifiers().contains(Modifier.PUBLIC)
                        && Util.isSourceElement(element);
            }
        });
    }
}
```

Example 2: an adapter to use ES6 Maps

JSweet default implementation of maps behaves as follows:

- If the key type is a string, the map is transpiled to a regular JavaScript object, where property names will be the keys.
- If the key type is an object (any other than a string), the map is transpiled as a list of entries. The implementation is quite inefficient because finding a key requires iterating over the entries to find the right entry key.

In some contexts, you may want to get more efficient map implementations. If you target modern browsers (or expect to have the appropriate polyfill available), you can simply use the `Map` object, which was standardized with ES6. Doing so requires an adapter that performs the following actions:

- Erase the Java `Map` type and replace it with the JavaScript `Map` type, or actually the `any` type, since you may want to keep the `Object` implementation when keys are strings.
- Substitute the construction of a map with the corresponding JavaScript construction.
- Substitute the invocations on Java maps with the corresponding JavaScript invocations.

Note that the following adapter is a partial implementation that shall be extended to support more cases and adapted to your own requirements. Additionally, this implementation generates untyped JavaScript in order to avoid having to have the ES6 API in the compilation path.

```
package org.jsweet.transpiler.extension;

import java.util.Arrays;
import java.util.HashMap;
import java.util.Hashtable;
import java.util.Map;
import java.util.TreeMap;
```

```
import javax.lang.model.element.Element;
import org.jsweet.transpiler.model.MethodInvocationElement;
import org.jsweet.transpiler.model.NewClassElement;

public class MapAdapter extends PrinterAdapter {

    // all the Java types that will be translated to ES6 maps
    static String[] mapTypes = {
        Map.class.getName(), HashMap.class.getName(),
        TreeMap.class.getName(), Hashtable.class.getName() };

    public MapAdapter(PrinterAdapter parent) {
        super(parent);
        // rewrite all Java map and compatible map implementations types
        // note that we rewrite to 'any' because we don't want to require the
        // ES6 API to compile (all subsequent accesses will be untyped)
        for (String mapType : mapTypes) {
            addTypeMapping(mapType, "any");
        }
    }

    @Override
    public boolean substituteNewClass(NewClassElement newClass) {
        String className = newClass.getTypeAsElement().toString();
        // map the map constructor to the global 'Map' variable (untyped access)
        if (Arrays.binarySearch(mapTypes, className) >= 0) {
            // this access is browser/node-compatible
            print("new (typeof window == 'undefined'?global:window)['Map'](")
                    .printArgList(newClass.getArguments()).print(")");
            return true;
        }
        // delegate to the adapter chain
        return super.substituteNewClass(newClass);
    }

    @Override
    public boolean substituteMethodInvocation(MethodInvocationElement invocation) {
        if (invocation.getTargetExpression() != null) {
            Element targetType = invocation.getTargetExpression().getTypeAsElement();
            if (Arrays.binarySearch(mapTypes, targetType.toString()) >= 0) {
                // Java Map methods are mapped to their JavaScript equivalent
                switch (invocation.getMethodName()) {
                case "put":
                    printMacroName(invocation.getMethodName());
                    print(invocation.getTargetExpression()).print(".set(")
                            .printArgList(invocation.getArguments())
```

```
                        .print(")");
                    return true;
                // although 'get' has the same name, we still rewrite it in case
                // another extension would provide it's own implementation
                case "get":
                    printMacroName(invocation.getMethodName());
                    print(invocation.getTargetExpression()).print(".get(")
                            .printArgList(invocation.getArguments())
                            .print(")");
                    return true;
                case "containsKey":
                    printMacroName(invocation.getMethodName());
                    print(invocation.getTargetExpression()).print(".has(")
                            .printArgList(invocation.getArguments())
                            .print(")");
                    return true;
                // we use the ES6 'Array.from' method in an untyped way to
                // transform the iterator in an array
                case "keySet":
                    printMacroName(invocation.getMethodName());
                    print("(<any>Array).from(")
                            .print(invocation.getTargetExpression()).print(".keys())");
                    return true;
                case "values":
                    printMacroName(invocation.getMethodName());
                    print("(<any>Array).from(")
                            .print(invocation.getTargetExpression()).print(".values())");
                    return true;
                // in ES6 maps, 'size' is a property, not a method
                case "size":
                    printMacroName(invocation.getMethodName());
                    print(invocation.getTargetExpression()).print(".size");
                    return true;
                }
            }

        }
        // delegate to the adapter chain
        return super.substituteMethodInvocation(invocation);
    }
}
```

Example 3: an adapter to support Java BigDecimal

Java's BigDecimal API is a really good API to avoid typical floating point

precision issues, especially when working on currencies. This API is not available by default in JavaScript and would be quite difficult to emulate. GWT provides an emulation of the BigDecimal API, which is implemented with Java, but JSweet proposes another way to do it, which consists of mapping the BigDecimal API to an existing JavaScript API called Big.js. Mapping to an existing JS library has several advantages compared to emulating an API:

1. The implementation is already available in JavaScript, so there is less work emulating the Java library.
2. The implementation is pure JavaScript and is made specifically for JavaScript. So we can assume that will be more efficient that an emulation, and even more portable.
3. The generated code is free from any Java APIs, which makes it more JavaScript friendly and more inter-operable with existing JavaScript programs (legacy JavaScript clearly uses Big.js objects, and if not, we can decide to tune the adapter).

The following code shows the adapter that tunes the JavaScript generation to map the Java's BigDecimal API to the Big JavaScript library. This extension requires the big.js candy to be available in the JSweet classpath: https://github.com/jsweet-candies/candy-bigjs.

```
package org.jsweet.transpiler.extension;

import java.math.BigDecimal;
import javax.lang.model.element.Element;
import org.jsweet.transpiler.extension.PrinterAdapter;
import org.jsweet.transpiler.model.MethodInvocationElement;
import org.jsweet.transpiler.model.NewClassElement;

public class BigDecimalAdapter extends PrinterAdapter {

    public BigDecimalAdapter(PrinterAdapter parent) {
        super(parent);
        // all BigDecimal types are mapped to Big
        addTypeMapping(BigDecimal.class.getName(), "Big");
    }

    @Override
    public boolean substituteNewClass(NewClassElement newClass) {
        String className = newClass.getTypeAsElement().toString();
        // map the BigDecimal constructors
        if (BigDecimal.class.getName().equals(className)) {
            print("new Big(").printArgList(newClass.getArguments()).print(")");
            return true;
        }
```

```
        // delegate to the adapter chain
        return super.substituteNewClass(newClass);
    }

    @Override
    public boolean substituteMethodInvocation(MethodInvocationElement invocation) {
        if (invocation.getTargetExpression() != null) {
            Element targetType = invocation.getTargetExpression().getTypeAsElement();
            if (BigDecimal.class.getName().equals(targetType.toString())) {
                // BigDecimal methods are mapped to their Big.js equivalent
                switch (invocation.getMethodName()) {
                case "multiply":
                    printMacroName(invocation.getMethodName());
                    print(invocation.getTargetExpression())
                            .print(".times(").printArgList(invocation.getArguments())
                            .print(")");
                    return true;
                case "add":
                    printMacroName(invocation.getMethodName());
                    print(invocation.getTargetExpression())
                            .print(".plus(").printArgList(invocation.getArguments())
                            .print(")");
                    return true;
                case "scale":
                    printMacroName(invocation.getMethodName());
                    // we assume that we always have a scale of 2, which is a
                    // good default if we deal with currencies...
                    // to be changed/implemented further
                    print("2");
                    return true;
                case "setScale":
                    printMacroName(invocation.getMethodName());
                    print(invocation.getTargetExpression())
                            .print(".round(").print(invocation.getArguments().get(0))
                            .print(")");
                    return true;
                case "compareTo":
                    printMacroName(invocation.getMethodName());
                    print(invocation.getTargetExpression()).print(".cmp(")
                            .print(invocation.getArguments().get(0))
                            .print(")");
                    return true;
                case "equals":
                    printMacroName(invocation.getMethodName());
                    print(invocation.getTargetExpression()).print(".eq(")
                            .print(invocation.getArguments().get(0))
```

```
                        .print(")");
                return true;
            }
        }

    }
    // delegate to the adapter chain
    return super.substituteMethodInvocation(invocation);
  }
}
```

Example 4: an adapter to map enums to strings

This example tunes the JavaScript generation to remove enums and replace them with strings. It only applies to enums that are annotated with `@jsweet.lang.StringType`.

For instance: `@StringType enum MyEnum { A, B, C }` will be erased and all subsequent accesses to the enum constants will be mapped to simple strings (`MyEnum.A => "A", MyEnum.B => "B", MyEnum.C => "C"`). Typically, a method declaration such as `void m(MyEnum e) {...}` will be printed as `void m(e : string) {...}`. And of course, the invocation `xxx.m(MyEnum.A)` will be printed as `xxx.m("A")`.

```
package org.jsweet.transpiler.extension;

import javax.lang.model.element.Element;
import javax.lang.model.element.ElementKind;
import org.jsweet.JSweetConfig;
import org.jsweet.transpiler.model.CaseElement;
import org.jsweet.transpiler.model.ExtendedElement;
import org.jsweet.transpiler.model.MethodInvocationElement;
import org.jsweet.transpiler.model.VariableAccessElement;

public class StringEnumAdapter extends PrinterAdapter {

    private boolean isStringEnum(Element element) {
        // note: this function could be improved to exclude enums that have
        // fields or methods other than the enum constants
        return element.getKind() == ElementKind.ENUM
                && hasAnnotationType(element, JSweetConfig.ANNOTATION_STRING_TYPE);
    }

    public StringEnumAdapter(PrinterAdapter parent) {
        super(parent);
        // eligible enums will be translated to string in JS
```

```
        addTypeMapping((typeTree, name) ->
                isStringEnum(typeTree.getTypeAsElement()) ? "string" : null);

        // ignore enum declarations with a programmatic annotation manager
        addAnnotationManager(new AnnotationManager() {
            @Override
            public Action manageAnnotation(Element element, String annotationType) {
                // add the @Erased annotation to string enums
                return JSweetConfig.ANNOTATION_ERASED.equals(annotationType)
                        && isStringEnum(element) ? Action.ADD : Action.VOID;
            }
        });

    }

    @Override
    public boolean substituteMethodInvocation(MethodInvocationElement invocation) {
        if (invocation.getTargetExpression() != null) {
            Element targetType = invocation.getTargetExpression().getTypeAsElement();
            // enum API must be erased and use plain strings instead
            if (isStringEnum(targetType)) {
                switch (invocation.getMethodName()) {
                case "name":
                    printMacroName(invocation.getMethodName());
                    print(invocation.getTargetExpression());
                    return true;
                case "valueOf":
                    printMacroName(invocation.getMethodName());
                    print(invocation.getArgument(0));
                    return true;
                case "equals":
                    printMacroName(invocation.getMethodName());
                    print("(").print(invocation.getTargetExpression()).print(" == ")
                            .print(invocation.getArguments().get(0)).print(")");
                    return true;
                }
            }
        }
        return super.substituteMethodInvocation(invocation);
    }

    @Override
    public boolean substituteVariableAccess(VariableAccessElement variableAccess) {
        // accessing an enum field is replaced by a simple string value
        // (MyEnum.A => "A")
        if (isStringEnum(variableAccess.getTargetElement())) {
```

```
            print("\"" + variableAccess.getVariableName() + "\"");
            return true;
        }
        return super.substituteVariableAccess(variableAccess);
    }

    @Override
    public boolean substituteCaseStatementPattern(CaseElement caseStatement,
            ExtendedElement pattern) {
        // map enums to strings in case statements
        if (isStringEnum(pattern.getTypeAsElement())) {
            print("\"" + pattern + "\"");
            return true;
        }
        return super.substituteCaseStatementPattern(caseStatement, pattern);
    }
}
```

Example 5: an adapter to generate JavaScript JAX-RS proxies/stubs

It is a common use case to implement a WEB or mobile application with Java on the server and JavaScript on the client. Typically, a JEE/Jackson server will expose a REST API through the JAX-RS specifications, and the HTML5 client will have to invoke this API using `XMLHttpRequest` or higher-level libraries such as jQuery. However, manually coding the HTTP invocations comes with many drawbacks:

- It requires the use of specific APIs (XHR, jQuery), which is not easy for all programmers and may imply different programming styles that would make the code more difficult to read and maintain.
- It requires the programmers to handle manually the serialization/deserialization, while it can be done automatically trough the use of annotation-driven generative programming.
- It leads to unchecked invocations, which means that it is easy for the programmer to make an error in the name of the service/path/parameters, and in the expected DTOs. No refactoring and content-assist is available.

With a JSweet adapter, using the `afterType` method it is easy to automatically generate a TypeScript stub that is well-typed and performs the required operations for invoking the REST service, simply by using the service API and the JAX-RS annotations. This type of tooling falls in the category of so-called Generative Programming.

The following code is only a partial implementation of an adapter that would introspect the program's model and generate the appropriate stubs in TypeScript.

It is not meant to be operational, so you need to modify to fit your own use case.

```
import javax.lang.model.element.Element;
import javax.lang.model.element.ExecutableElement;
import javax.lang.model.element.TypeElement;
import javax.lang.model.element.VariableElement;
import javax.lang.model.type.TypeKind;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;

class JaxRSStubAdapter extends PrinterAdapter {

    public JaxRSStubAdapter(PrinterAdapter parent) {
        super(parent);
        // erase service classes (server-side only)
        addAnnotationManager(new AnnotationManager() {
            @Override
            public Action manageAnnotation(Element element, String annotationType) {
                return JSweetConfig.ANNOTATION_ERASED.equals(annotationType)
                        && hasAnnotationType(element, Path.class.getName()) ?
                        Action.ADD : Action.VOID;
            }
        });
    }

    @Override
    public void afterType(TypeElement type) {
        super.afterType(type);
        if (hasAnnotationType(type, Path.class.getName())) {
            // actually generates the JAX-RS stub
            println().printIndent();
            print("class ").print(type.getSimpleName()).print(" {");
            startIndent();
            String typePathAnnotationValue = getAnnotationValue(type,
                    Path.class.getName(), String.class, null);
            String typePath = typePathAnnotationValue != null ? typePathAnnotationValue : "'
            for (Element e : type.getEnclosedElements()) {
                if (e instanceof ExecutableElement
                        && hasAnnotationType(e, GET.class.getName(),
                                PUT.class.getName(), Path.class.getName())) {
                    ExecutableElement method = (ExecutableElement) e;
                    println().printIndent().print(method.getSimpleName().toString())
                            .print("(");
```

```
                    for (VariableElement parameter : method.getParameters()) {
                        print(parameter.getSimpleName())
                                .print(" : ").print(getMappedType(parameter.asType()))
                                .print(", ");
                    }
                    print("successHandler : (");
                    if (method.getReturnType().getKind() != TypeKind.VOID) {
                        print("result : ").print(getMappedType(method.getReturnType()));
                    }
                    print(") => void, errorHandler?: () => void").print(") : void");
                    print(" {").println().startIndent().printIndent();
                    String pathAnnotationValue = getAnnotationValue(e, Path.class.getName(),
                            String.class, null);
                    String path = pathAnnotationValue != null ? pathAnnotationValue : "";
                    String httpMethod = "POST";
                    if(hasAnnotationType(e, GET.class.getName())) {
                        httpMethod = "GET";
                    }
                    if(hasAnnotationType(e, POST.class.getName())) {
                        httpMethod = "POST";
                    }
                    String[] consumes = getAnnotationValue(e, "javax.ws.rs.Consumes",
                            String[].class, null);
                    if (consumes == null) {
                        consumes = new String[] { "application/json" };
                    }
                    // actual code to be done
                    print("// modify JaxRSStubAdapter to generate an HTTP invocation here")
                            .println().printIndent();
                    print("//   - httpMethod: " + httpMethod).println().printIndent();
                    print("//   - path: " + typePath + path).println().printIndent();
                    print("//   - consumes: " + consumes[0]);
                    println().endIndent().printIndent().print("}");
                }
            }
            println().endIndent().printIndent().print("}");
        }
    }
}
```

NOTE: for compilation, you need the JAX-RS API in your classpath.

```
<dependency>
    <groupId>javax.ws.rs</groupId>
    <artifactId>javax.ws.rs-api</artifactId>
    <version>2.1-m07</version>
</dependency>
```

As an example, let us consider the following JAX-RS service.

```
@Path("/hello")
public class HelloWorldService {
    @GET
    @Path("/{param}")
    @Produces(MediaType.APPLICATION_JSON)
    public HelloWorldDto getMsg(@PathParam("param") String msg) {
        String output = "service says : " + msg;
        return new HelloWorldDto(output);
    }
}
```

Using the following DTO:

```
public class HelloWorldDto {
    private String msg;
    public HelloWorldDto(String msg) {
        super();
        this.msg = msg;
    }
    public String getMsg() {
        return msg;
    }
    public void setMsg(String msg) {
        this.msg = msg;
    }
}
```

If you apply JSweet enhanced with `JaxRSStubAdapter`, you will get the following TypeScript code (and corresponding JavaScript):

```
export class HelloWorldDto {
    /*private*/ msg : string;
    public constructor(msg : string) {
        this.msg = msg;
    }
    public getMsg() : string {
        return this.msg;
    }
    public setMsg(msg : string) {
        this.msg = msg;
    }
}
HelloWorldDto["__class"] = "HelloWorldDto";

class HelloWorldService {
    getMsg(msg : string,
```

```
            successHandler : (result : HelloWorldDto) => void,
            errorHandler?: () => void) : void {
        // modify JaxRSStubAdapter to generate an HTTP invocation here
        //   - httpMethod: GET
        //   - path: /hello/{param}
        //   - consumes: application/json
    }
}
```

So, all you need to do is to modify the code of the adapter to generate the actual invocation code in place of the comment. Once it is done, you can use the generated JavaScript code as a bundle to access your service in a well-typed way. Moreover, you can use JSweet to generate the TypeScript definitions of your services and DTOs, so that your TypeScript client are well-typed (see the JSweet's `declaration` option).

Example 6: an adapter to disallow global variables

This is a quite special adapter since it does not really generate any code, but it reports errors when the source code does not conform to certain coding standards. Here, we implement a simple constraint that reports errors when the user tries to declare global variables (i.e. in JSweet non-final static field declared in a `Globals` class).

```
package org.jsweet.transpiler.extension;

import javax.lang.model.element.Element;
import javax.lang.model.element.ElementKind;
import javax.lang.model.element.Modifier;
import javax.lang.model.element.TypeElement;
import javax.lang.model.element.VariableElement;

import org.jsweet.transpiler.JSweetProblem;

public class DisallowGlobalVariablesAdapter extends PrinterAdapter {

    public DisallowGlobalVariablesAdapter(PrinterAdapter parentAdapter) {
        super(parentAdapter);
    }

    @Override
    public void afterType(TypeElement type) {
        // we only check for static variables that are in a Globals class but
        // this could be generalized to any static variable
        if (!type.getQualifiedName().toString().startsWith("def.")
                && type.getSimpleName().toString().equals("Globals")) {
```

```
            for (Element member : type.getEnclosedElements()) {
                if (member.getKind() == ElementKind.FIELD) {
                    VariableElement field = (VariableElement) member;
                    // only non-final static variable have side effect
                    if (field.getModifiers().contains(Modifier.STATIC)
                            && !field.getModifiers().contains(Modifier.FINAL)) {
                        report(field, JSweetProblem.USER_ERROR, "global variables are not al
                    }
                }
            }
        }
        super.afterType(type);
    }

}
```

This adapter falls into the category of static analysis, which can be useful (along with code generation) to check that the input complies to the expected coding guidelines. It would be easy to enhance this adapter to add check on any static fields. A nice feature would be to disable the check when the field is annotated with a specific annotation (for instance `@AllowSideEffect`).

Appendix 1: JSweet transpiler options

```
  [-h|--help]

  [-w|--watch]
        Start a process that watches the input directories for changes and
        re-run transpilation on-the-fly.

  [-v|--verbose]
        Turn on all levels of logging.

  [--encoding <encoding>]
        Force the Java compiler to use a specific encoding (UTF-8, UTF-16, ...).
        (default: UTF-8)

  [--jdkHome <jdkHome>]
        Set the JDK home directory to be used to find the Java compiler. If not
        set, the transpiler will try to use the JAVA_HOME environment variable.
        Note that the expected JDK version is greater or equals to version 8.

  (-i|--input) input1:input2:...:inputN
        An input directory (or column-separated input directories) containing
        Java files to be transpiled. Java files will be recursively looked up in
```

```
      sub-directories. Inclusion and exclusion patterns can be defined with
      the 'includes' and 'excludes' options.

[--includes includes1:includes2:...:includesN ]
      A column-separated list of expressions matching files to be included
      (relatively to the input directory).

[--excludes excludes1:excludes2:...:excludesN ]
      A column-separated list of expressions matching files to be excluded
      (relatively to the input directory).

[(-d|--defInput) defInput1:defInput2:...:defInputN ]
      An input directory (or column-separated input directories) containing
      TypeScript definition files (*.d.ts) to be used for transpilation.
      Definition files will be recursively looked up in sub-diredctories.

[--noRootDirectories]
      Skip the root directories (i.e. packages annotated with
      @jsweet.lang.Root) so that the generated file hierarchy starts at the
      root directories rather than including the entire directory structure.

[--tsout <tsout>]
      Specify where to place generated TypeScript files. (default: .ts)

[(-o|--jsout) <jsout>]
      Specify where to place generated JavaScript files (ignored if jsFile is
      specified). (default: js)

[--disableSinglePrecisionFloats]
      By default, for a target version >=ES5, JSweet will force Java floats to
      be mapped to JavaScript numbers that will be constrained with ES5
      Math.fround function. If this option is true, then the calls to
      Math.fround are erased and the generated program will use the JavaScript
      default precision (double precision).

[--tsOnly]
      Do not compile the TypeScript output (let an external TypeScript
      compiler do so).

[--ignoreDefinitions]
      Ignore definitions from def.* packages, so that they are not generated
      in d.ts definition files. If this option is not set, the transpiler
      generates d.ts definition files in the directory given by the tsout
      option.

[--declaration]
```

```
      Generate the d.ts files along with the js files, so that other programs
      can use them to compile.

[--dtsout <dtsout>]
      Specify where to place generated d.ts files when the declaration option
      is set (by default, d.ts files are generated in the JavaScript output
      directory - next to the corresponding js files).

[--candiesJsOut <candiesJsOut>]
      Specify where to place extracted JavaScript files from candies.
      (default: js/candies)

[--sourceRoot <sourceRoot>]
      Specify the location where debugger should locate Java files instead of
      source locations. Use this flag if the sources will be located at
      run-time in a different location than that at design-time. The location
      specified will be embedded in the sourceMap to direct the debugger where
      the source files will be located.

[--classpath <classpath>]
      The JSweet transpilation classpath (candy jars). This classpath should
      at least contain the core candy.

[(-m|--module) <module>]
      The module kind (none, commonjs, amd, system or umd). (default: none)

[-b|--bundle]
      Bundle up all the generated code in a single file, which can be used in
      the browser. The bundle files are called 'bundle.ts', 'bundle.d.ts', or
      'bundle.js' depending on the kind of generated code. NOTE: bundles are
      not compatible with any module kind other than 'none'.

[(-f|--factoryClassName) <factoryClassName>]
      Use the given factory to tune the default transpiler behavior.

[--sourceMap]
      Generate source map files for the Java files, so that it is possible to
      debug Java files directly with a debugger that supports source maps
      (most JavaScript debuggers).

[--enableAssertions]
      Java 'assert' statements are transpiled as runtime JavaScript checks.

[--header <header>]
      A file that contains a header to be written at the beginning of each
      generated file. If left unspecified, JSweet will generate a default
```

```
      header.

[--workingDir <workingDir>]
      The directory JSweet uses to store temporary files such as extracted
      candies. JSweet uses '.jsweet' if left unspecified.

[--targetVersion <targetVersion>]
      The EcmaScript target (JavaScript) version. Possible values: [ES3, ES5,
      ES6] (default: ES3)
```

Appendix 2: packaging and static behavior

This appendix explains some static behavior with regards to packaging.

When main methods are invoked

When main methods are invoked depends on the way the program is packaged.

- `module`: off, `bundle`: off. With default packaging, one Java source file corresponds to one generated JavaScript file. In that case, when loading a file in the browser, all the main methods will be invoked right at the end of the file.
- `module`: off, `bundle`: on. When the `bundle` option is on and the `module` option is off, main methods are called at the end of the bundle.
- `module`: on, `bundle`: off. With module packaging (`module` option), one Java package corresponds to one module. With modules, it is mandatory to have only one main method in the program, which will be the global entry point from which the module dependency graph will be calculated. The main module (the one with the main method) will use directly or transitively all the other modules. The main method will be invoked at the end of the main module evaluation.

Because of modules, it is good practice to have only one main method in an application.

Static and inheritance dependencies

In TypeScript, programmers need to take care of the ordering of classes with regards to static fields and initializers. Typically, a static member cannot be initialized with a static member of a class that has not yet been defined. Also, a class cannot extend a class that has not been defined yet. This forward-dependency issue triggers runtime errors when evaluating the generated JavaScript code, which can be quite annoying for the programmers and may requires the use of external JavaScript bundling tools, such as Browserify.

JSweet's statics lazy initialization allows static forward references within a given file, and within an entire bundle when the `bundle` option is set. Also, when bundling a set of files, JSweet analyses the inheritance tree and performs a partial order permutation to eliminate forward references in the inheritance tree. Note that TypeScript bundle provide a similar feature, but the references need to be manually declared, which is not convenient for programmers.

To wrap it up, here are the guidelines to be followed by the programmers depending on the packaging method:

- `module`: off, `bundle`: off. One JavaScript file is generated per Java file. The programmer must take care of including the files in the right order in the HTML page, so that there are no forward references with regard to inheritance and statics. Within a given file, static forward references are allowed, but inheritance forward reference are not supported yet (this will be supported in coming releases).
- `module`: off, `bundle`: on. This configuration produces a unique browser-compatible bundle file that can be included in an HTML page. Here, the programmer does not have to take care at all of the forward references across files. Exactly like in Java, the order does not matter. Within a single file, the programmer still have to take care of the inheritance forward references (in other words, a subclass must be declared after its parent class) (this will be supported in coming releases).
- `module`: commonjs, amd or umd, `bundle`: off. This configuration produces one module file per Java package so that they can be used within a module system. For instance, using the `commonjs` module kind will allow the program to run on Node.js. In that configuration, the program should contain one main method and only the module file containing the main method should be loaded (because it will take care loading all the other modules). This configuration imposes the same constraint within a single file (no forward-references in inheritance).