# Object keys type check

nbeloglazov@google.com
EXTERNAL DOCUMENT



## Objective

Introduce a check into js compiler that verifies that only certain types can be used as keys in Object type. The types used as keys have to be either native or "stringifiable" meaning that they have custom toString methods. This check can prevent certain type of errors on compile time.

## Background

Javascript language has a concept of objects. An object is a collection of properties, and a property is an association between a name and a value. Often objects used as maps:

```
var map = {};
map['one'] = 1;
map['two'] = 2;
```

But objects are not true maps: all keys are implicitly converted to strings which may cause errors if not careful about what is used as keys. It's not unreasonable to expect objects to compare keys using identity operator (===) and not convert them to strings. However, that is not what in fact happens. Example of such behaviour:

```
var map = {};

var key1 = {name: 'key1'};
var key2 = {name: 'key2'};
map[key1] = 1;
map[key2] = 2;

console.log(map[key1]); // 2
console.log(key1 === key2); // false
console.log(key1.toString() === key2.toString()); // true
```

## Design

The new check is going to verify all user defined types and if they contain templatized Object type with at least 2 types e.g. "{Object.<number, number>}" then it will check whether first type (key type) is stringifiable. In case it's not - warning or error will be reported.

### Stringifiable type

Stringifiable type is a type that can be used as key in object. Types which are considered stringifiable:

- primitive types: string, number, boolean;
- null and undefined;
- unknown type, current js compiler parses Object.<string> as Object.<?, string> and new check should support this case;
- enums with stringifiable types;
- built-in types: RegEx, Date;
- untyped array or array typed with stringifiable type;
- user-defined class with toString method. toString method can be defined on class itself or one of its parent. Note that native implementation of toString is defined on native Object type which is parent of all other classes so native toString must be ignored;
- union type only if all its alternates are stringifiable;
- records and interfaces, though interfaces/records might not have toString() method it is hard to verify at compile time so they're considered stringifiable to avoid false positives;

All other types are non-stringifiable. Few examples:

- all type (star *);
- Object type;
- function;
- user-defined classes without toString methods;

### What jsdocs tags are checked

The pass will check only types declared by user in jsdocs. Following tags will be checked: @param, @return, @type and @typedef as only they can contain type information.

The new check will be added as part of lintChecks group and will be disabled by default to avoid breaking existing code.

## Testing

- Unit tests.

## Examples of types

Will be used in unit tests

### Bad types

```
// Different tags
/** @type {!Object.<Object, number>} */ var k;
/** @param {!Object.<Object, number>} a */ var f = function(a) {};
/** @return {!Object.<Object, number>} */ var f = function() {return {}};
/** @typedef {!Object.<Object, number>} */ var MyType;

// Non stringifiable built-in types
/** @type {!Object.<!Object, number>} */ var k;
/** @type {!Object.<function(), number>} */ var k;

// Union and templatized type
/** @type {(string|Object.<!Object, number>)} */ var k;
/** @type {!Object.<number, !Object.<!Object, number>>} */ var k;

// Test using custom class or interface without toString method as key.
/** @constructor */
var MyClass = function() {};
/** @type {!Object.<MyClass, number>} */
var k;
```

### Good types

```
// Built-in types
/** @type {!Object.<number, number>} */ var k;
/** @type {!Object.<string, number>} */ var k;
/** @type {!Object.<boolean, number>} */ var k;
/** @type {!Object.<!Date, number>} */ var k;
/** @type {!Object.<!RegExp, number>} */ var k;
/** @type {!Object.<null, number>} */ var k;
/** @type {!Object.<undefined, number>} */ var k;


/** @interface */
var MyInterface = function() {};
/** @type {!Object.<!MyInterface, number>} */
var k;

/** @typedef {{a: number}} */
var MyRecord;
/** @type {!Object.<MyRecord, number>} */
```

```
var k;

// Class with toString
/** @constructor */
var MyClass = function() {};
/** @return {string} */
MyClass.prototype.toString = function() { return ''; };
/** @type {!Object.<!MyClass, number>} */
var k;

// Class which inherits toString from parent.
/** @constructor */
var Parent = function() {};
/** @return {string} */
Parent.prototype.toString = function() { return ''; };
/** @constructor @extends {Parent} */
var Child = function() {};
/** @type {!Object.<!Child, number>} */
var k;
```