ToPrimitive

In-Depth Understanding of Abstract Operations: ToPrimitive

In JavaScript, abstract operations are internal methods that provide fundamental operations necessary for the language's behavior. One such abstract operation is **ToPrimitive**. This operation is crucial in type conversion, especially when dealing with objects and their primitive values.

ToPrimitive Overview

ToPrimitive is an abstract operation that attempts to convert an object to a primitive value. The process follows these steps:

1. **Preferred Type**: The operation can take a hint about the preferred type (either `Number` or `String`).
2. **Conversion Attempt**:
   - If the preferred type is `String`, the operation first attempts to call the `toString` method of the object.
   - If the preferred type is `Number`, it first tries to call the `valueOf` method.
   - If the first method does not yield a primitive, it attempts the other method.

Steps of ToPrimitive

1. **Check for Primitive**: If the input is already a primitive (e.g., `null`, `undefined`, `boolean`, `number`, `string`, or `symbol`), return it as-is.
2. **Get Methods**:
   - If the preferred type is `String`, first try `toString`, then `valueOf`.
   - If the preferred type is `Number`, first try `valueOf`, then `toString`.
3. **Call Methods**: Invoke the methods in the specified order:
   - If the method returns a primitive, use that value.
   - If the method returns an object, proceed to the next method.
4. **Throw Error**: If neither method returns a primitive, throw a `TypeError`.

Practical Example

Let's consider a practical example to illustrate how **ToPrimitive** works.

```
const obj = {
  toString() {
    return "string value";
  },
```

```
  valueOf() {
    return 42;
  }
};

// Preferred type: Number
console.log(+obj); // Output: 42 (valueOf is called first)

// Preferred type: String
console.log(`${obj}`); // Output: "string value" (toString is
called first)
```

Symbol.toPrimitive

JavaScript also provides a way to customize the **ToPrimitive** operation using the `Symbol.toPrimitive` method. This method allows an object to define its preferred conversion logic explicitly.

Example with Symbol.toPrimitive

```
const customObj = {
  [Symbol.toPrimitive](hint) {
    if (hint === "number") {
      return 42;
    }
    if (hint === "string") {
      return "custom string";
    }
    return null;
  }
};

// Preferred type: Number
console.log(+customObj); // Output: 42

// Preferred type: String
console.log(`${customObj}`); // Output: "custom string"

// Default hint (not string or number)
console.log(customObj + ""); // Output: "null"
```

Common Use Cases

1. **String Concatenation**:
    - When concatenating objects with strings, JavaScript uses the **ToPrimitive** operation to convert objects to strings.

- Example: `"Hello " + obj` triggers `obj.toString()` or `obj[Symbol.toPrimitive]('string')`.

2. **Numeric Operations**:
   - Operations like addition, subtraction, multiplication, etc., will convert objects to their numeric primitive values.
   - Example: `obj * 2` triggers `obj.valueOf()` or `obj[Symbol.toPrimitive]('number')`.
3. **Comparison Operations**:
   - Comparisons involving objects and primitives use **ToPrimitive** to convert objects.
   - Example: `obj > 10` triggers `obj.valueOf()` or `obj[Symbol.toPrimitive]('number')`.

Extended Details for ToPrimitive

1. **Preferred Type Hints**:
   - **Hint "default"**: In most cases, JavaScript uses the hint `"default"`, which typically behaves like `"number"`.
   - **Exceptions**: Certain operations like `+` and template literals treat the hint differently. For example, the `+` operator with an object as one operand uses the `"default"` hint, leading to string concatenation if possible.
2. **Fallback Mechanism**:
   - **Order of Operations**: If `toString` and `valueOf` are both defined, and neither returns a primitive, JavaScript will throw a `TypeError`. Ensuring these methods return appropriate primitive values is crucial to avoid errors.

Example with Custom toString and valueOf

```
const complexObj = {
  toString() {
    return "[object Complex]";
  },
  valueOf() {
    return 100;
  }
};

console.log(String(complexObj)); // Output: "[object Comple
x]"
console.log(Number(complexObj)); // Output: 100
console.log(complexObj + "");    // Output: "100" (uses value
Of due to default hint)
console.log(+complexObj);        // Output: 100 (uses valueOf
for numeric conversion)
```

Interview Preparation Tips

1. **Understand Edge Cases**:
    - Be prepared to discuss what happens if `toString` and `valueOf` methods both fail to return a primitive.
    - Know the default behavior when neither method is present or both methods return objects.
2. **Explain Symbol.toPrimitive**:
    - Be ready to explain how `Symbol.toPrimitive` provides finer control over type conversion.
    - Discuss scenarios where using `Symbol.toPrimitive` is preferable to defining `toString` and `valueOf`.
3. **Discuss Performance Implications**:
    - Understand the performance implications of custom `toString` and `valueOf` methods, especially in frequently called operations.
    - Be able to explain how improper implementations can lead to performance bottlenecks or errors.

Development Best Practices

1. **Implement Meaningful Methods**:
    - Ensure that `toString` and `valueOf` return meaningful and useful primitive values that make sense in the context of the object.
2. **Use Symbol.toPrimitive**:
    - Use `Symbol.toPrimitive` for objects that require precise control over their conversion to different primitive types.
3. **Testing**:
    - Write comprehensive tests to ensure that objects convert correctly in various contexts (e.g., arithmetic, string concatenation, comparisons).

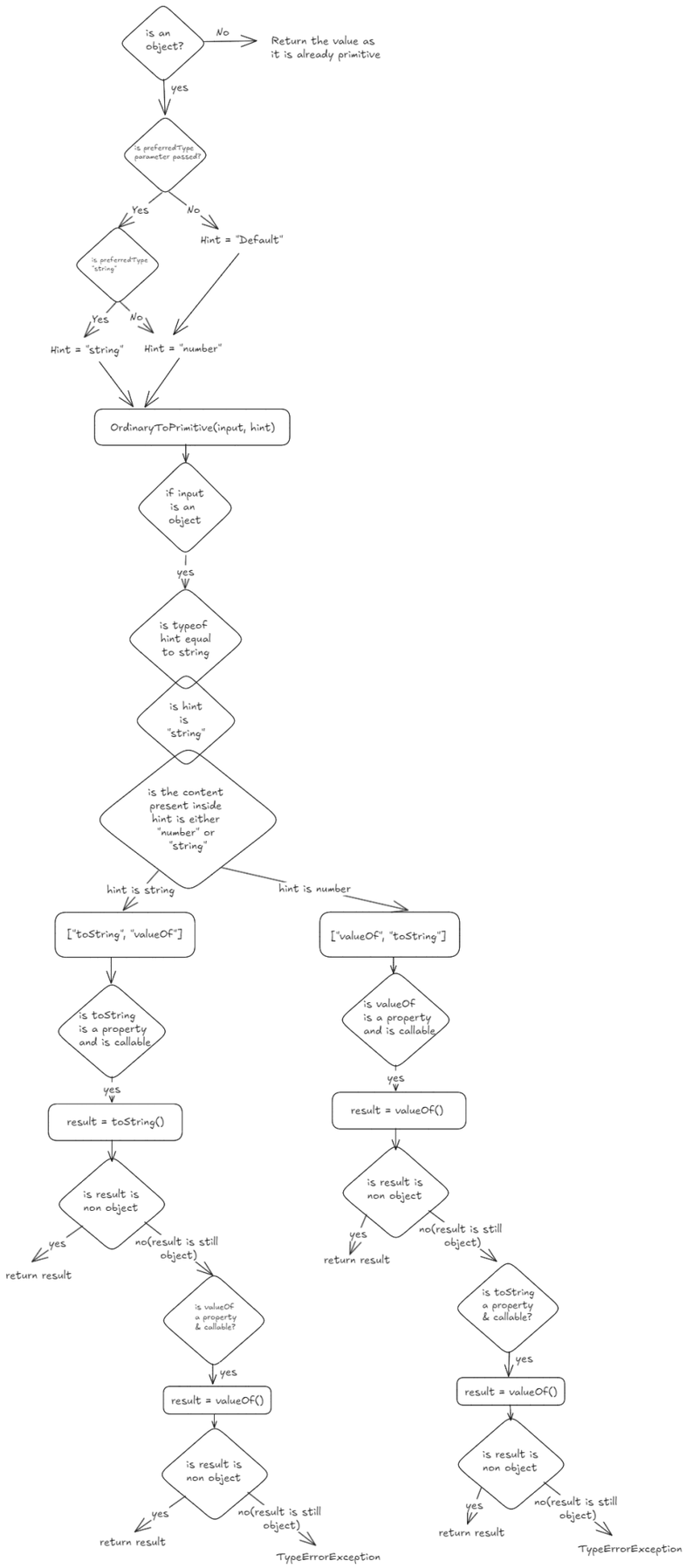
Conclusion

The **ToPrimitive** operation is a foundational concept in JavaScript that plays a crucial role in type conversion. By understanding its mechanics, preferred type hints, and practical applications, you'll be well-equipped for both interviews and development tasks. Moreover, demonstrating knowledge of edge cases, best practices, and performance considerations will help showcase your expertise in JavaScript.

7.1.1 ToPrimitive (*input* [, *preferredType*])

The abstract operation ToPrimitive takes argument *input* (an ECMAScript language value) and optional argument *preferredType* (STRING or NUMBER) and returns either a normal completion containing an ECMAScript language value or a throw completion. It converts its *input* argument to a non-Object type. If an object is capable of converting to more than one primitive type, it may use the optional hint *preferredType* to favour that type. It performs the following steps when called:

1. If *input* is an Object, then
    a. Let *exoticToPrim* be ? GetMethod(*input*, %Symbol.toPrimitive%).
    b. If *exoticToPrim* is not **undefined**, then
        i. If *preferredType* is not present, then
            1. Let *hint* be **"default"**.
        ii. Else if *preferredType* is STRING, then
            1. Let *hint* be **"string"**.
        iii. Else,
            1. Assert: *preferredType* is NUMBER.
            2. Let *hint* be **"number"**.
        iv. Let *result* be ? Call(*exoticToPrim*, *input*, « *hint* »).
        v. If *result* is not an Object, return *result*.
        vi. Throw a **TypeError** exception.
    c. If *preferredType* is not present, let *preferredType* be NUMBER.
    d. Return ? OrdinaryToPrimitive(*input*, *preferredType*).
2. Return *input*.




```
if(PreferredType is not Present){
    hint = "default"
} else if (PreferredType is String){
    hint = "string"
} else {
    hint = "number"
}

// after some operation
if(hint is "default"){
    hint = "number"
}
```

```
for(Name in methodNames){
    if(object has a property with function name as Name){
        if(the property is a callable function){
            result = call the function of the object

            if(result is non object return result;

        }
    }
}
```

```
>> x = {}
<- ▸ Object {  }
>> x.toString();
<- "[object Object]"
>> x.valueOf()
<- ▸ Object {  }
>> y = {a: 10}
<- ▸ Object { a: 10 }
>> y.toString();
<- "[object Object]"
>> y.valueOf()
<- ▸ Object { a: 10 }
>> y = {a: 10, b: 20, toString: function() {return "my tostring";}}
<- ▸ Object { a: 10, b: 20, toString: toString() }
>> y.toString()
<- "my tostring"
>> y.valueOf();
<- ▸ Object { a: 10, b: 20, toString: toString() }
>> y.valueOf = function() {return 100;}
<- ▸ function valueOf()
>> y.valueOf();
<- 100
```

```
>> x =10;
<- 10
>> y = {a:10}
<- ▸ Object { a: 10 }
>> x - y // y.valueOf() => y->y.toString() => '[object Object]' => 10 - "[object Object]" => 10-NaN
<- NaN
>> y.toString = function(){return "99";}
<- ▸ function toString()
>> x-y; // y.valueOf() => y->y.toString() => '99' => 10 - "99" => 10-99
<- -89
>> y.valueOf = function(){return 3000;}
<- ▸ function valueOf()
>> x-y; // y.valueOf() -> 3000 => 10-3000
<- -2990
```

```
class Product{
        constructor(n,p){
            this.price =p;
            this.name =n;
      }
      valueOf(){
        return this.p;
      }
    }
← undefined
p = new Product("iphone", 2000);
← ▸ Object { price: 2000, name: "iphone" }
10 - p
← NaN
p.valueOf();
← undefined
10 - p // p.valueOf() -> undefined // 10 - undefined(NaN) => NaN
← NaN
class Product1{
        constructor(n,p){
            this.price =p;
            this.name =n;
      }
      valueOf(){
        return this.price;
      }
    }
← undefined
p1 = new Product1("iphone", 2000);
← ▸ Object { price: 2000, name: "iphone" }
10 - p1
← -1990
```

```
class Product1{
        constructor(n,p){
            this.price =p;
            this.name =n;
      }
      valueOf(){
        return null;
      }
    }
← undefined
p1 = new Product1("iphone", 2000);
← ▸ Object { price: 2000, name: "iphone" }
10 - p1
← 10
```

```
class Product{
        constructor(n,p){
            this.price =p;
            this.name =n;
      }
      valueOf(){
        return this.price;
      }
    }
undefined
p1 = new Product("iphone", 2000);
▸ Object { price: 2000, name: "iphone" }
p2 = new Product("iphone 1", 2000);
▸ Object { price: 2000, name: "iphone 1" }
p1 < p2
false
p1 <= p2 // compare on price
true
```

```
function fun() {}
undefined
"My function name is" + fun
"My function name isfunction fun() {}"
fun.valueOf()
▸ function fun()
typeof(fun.valueOf())
"function"
fun.toString()
"function fun() {}"
typeof(fun.toString())
"string"
typeof(fun)
"function"
fun.toString = function() {return "fun"}
▸ function toString()
"My function name is " + fun
"My function name is fun"
```

```
>> 1 < 2 < 3
<- true
>> 3 > 2 > 1
<- false
>> 1 < 2 < 3 // 1 < 2 < 3 => (1 < 2) < 3 => (true) < 3 => 1 < 3 -> true
<- true
>> 3 > 2 > 1 // (3 > 2) > 1 => (true) > 1 => 1 > 1 -> false
<- false
>> let x = 10;
<- undefined
>> str = `Hello the value of x is ${x}`
<- "Hello the value of x is 10"
>> y = {

   }
<- ▸ Object {  }
>> str = `Hello the value of x is ${y}`;
<- "Hello the value of x is [object Object]"
>> s = "123"
<- "123"
>> +s
<- 123
>> s = "Hello"
<- "Hello"
>> +s
<- NaN
>> !!s
<- true
```

```
>> Number("123")
<- 123
>> +"123"
<- 123
>> Number(0)
<- 0
>> Number("-0")
<- -0
>> String(10)
<- "10"
>> String(-0)
<- "0"
>> String(0)
<- "0"
>> NaN === NaN
<- false
>> Infinity == -Infinity
<- false
>> isNaN(NaN)
<- true
>> isNaN("sanket")
<- true
>> isNaN(1)
<- false
>> Number.isNaN(NaN)
<- true
```

```
>> x = 0
<- 0
>> x === -0
<- true
>> x == -0
<- true
>> y = -0
<- -0
>> y.toString()
<- "0"
>> x == y
<- true
>> Object.is(y, 0)
<- false
>> Object.is(y, -0)
<- true
>> Math.sign(0)
<- 0
>> Math.sign(-0)
<- -0
```

```
>> x = 10
<- 10
>> x.toString()
<- "10"
>> typeof(x)
<- "number"
>> 10.toString()
Uncaught SyntaxError: identifier starts immediately after numeric literal [Learn More]      debugger eval code:1:4
>> (10).toString()
<- "10"
>> "sanket".toString()
<- "sanket"
>> "sanket".valueOf()
<- "sanket"
```