ES5.1 11.6.2 The Subtraction Operator

1. Let *lref* be the result of evaluating AdditiveExpression.
2. Let *lval* be GetValue(*lref*).
3. Let *rref* be the result of evaluating MultiplicativeExpression.
4. Let *rval* be GetValue(*rref*).
5. Let *lnum* be ToNumber(*lval*).
6. Let *rnum* be ToNumber(*rval*).
7. Return the result of applying the subtraction operation to *lnum* and *rnum*. See the note below 11.6.3.

Modified 11.6.2 The Overloadable Subtraction Operator

1. Let *lref* be the result of evaluating AdditiveExpression.
2. Let *lval* be GetValue(*lref*).
3. Let *rref* be the result of evaluating MultiplicativeExpression.
4. Let *rval* be GetValue(*rref*).
5. If Type(*lval*) is Number and Type(*rval*) is Number, then
   a. Return the result of applying the subtraction operation to *lval* and *rval*. See the note below 11.6.3.
6. Let *dispatchable* be the result of calling the [[Get]] internal methods of *lval* with the private name @operatorMinus as argument.
7. If IsCallable(*dispatchable*) is **true**, then
   a. Return the result of calling the [[Call]] internal method of *dispatchable* with providing *lval* as the **this** value and *rval* as the argument.
8. Let *lnum* be ToNumber(*lval*).
9. Let *rnum* be ToNumber(*rval*).
10. Return the result of applying the subtraction operation to *lnum* and *rnum*. See the note below 11.6.3.

Note that @operatorMinus is a new convention for expressing a private name value that is both known to the implementation and made publically available to ECMAScrpt code. In a primordial ECMAScript environment no ES5.1 built-in objects would have own or inherited such private named operator methods. So, the above algorithm changes have no semantic impact on existing code that does not explicitly add such operator methods to built-in or user defined objects.

Note that an implementation could treat lines 8-10 as the body of a default fallback @operatorMinus method. This enables the generic – operator to be code generated as a guarded double subtract with a fall back to a PIC-able method call.

Similar modification would need to be made to the definition of all the chapter 11 operators. These are essentially the only changes that need to be made to the core language semantics to support over-loadable operators. All support for any new data types that over-load the operators is provided library methods and functions (either standard, implementation, or user provided libraries). In particular, all (beyond what exits in ES5.1) type promotion and coercion semantics is implemented in the library code and is not part of the base language operator semantics. If the libraries are implemented in ES then any inter-procedural optimization techniques available in an implementation are fully applicable to them.

An operator method can be implemented to do anything that is desired and appropriate for its operands. For example, such a method might do explicit type analysis on the this value and the argument value to determine what operation to perform based upon a closed ended set of types. Note that the type of the first lval is typically implicitly known by the operator method based upon the property lookup that was performed to retrieve the method. The method can then apply the "double dispatch" technique with the argument to invoke a "minus" function that is appropriate to that specific operator combination. Use of this technique allows operator over-loading an open ended set of extensible numeric types.

For example:

```
// overload – to perform scalar subtraction from array elements and to subtract
// and to subtract corresponding

Array.prototype.@operatorMinus = function (rval) {return rval.@subFromArray(this)};

Number.prototype.@subFromArray = function(minuend) {"use strict";
   // array-number subtracts number from each minuend element and returns a new array
   var self = this;
   return anArray.map(function(v) {return v-self});
}

Array.prototype.@subFromArray(minuend)
   // array-array subtracts corresponding elements and collects in a new array
   var result = new Array;
   for (var i = Math.max(this.length, minuend.length)-1; i>= 0; i--)
      result[i] = minuend[i]-this[i];
   return result;
}
```

Note that while monkey patching is an easy way to extend existing objects to support operator double dispatch for new types, it is not the only extensibility mechanisms that can be used. Side-tables can also be used to accomplish the same result.

Finally, there are no restrictive requirements imposed upon the objects that implement operator methods. In particular, they need not be immutable. However, if a standard library defines certain classes of immutable objects that include support for operator methods then an implementation might choose to recognize such objects in the generic or specialized code emitted for operators.