How JS engine optimizes JavaScript code

It starts with the java script you write, and the JavaScript parser the parser tokenizes source code and changes it in too abstract syntax tree (AST) and feed to interrupter to generates bytecodes. And the bytecode feed to optimizing complier and generate highly optimizing code that runs more efficiently.

Example let b= 5. ---→JavaScript AST visualizer

|

Program -→ variable Declaration --→ variable Declarator -→ b

→ 5

-Interpreter (ignition in google chrome)-→ generate bytecode

- Compiler (Turbofans in goggle chrome) -→ optimized machine code, to scan the code ahead of time and to find stab.

Example2 :>
```
function concatArray(arr){
        Var result = “”;
    for (let i=0; i<arr.length;i++){
        Result += arr[i]
      }
    Return result;
    }
```

think every step of the for loop, the interpreter is asking:
Is I an intereger?
Is result a string?
Is arr actually an array?
Is arr[i] a string?

-The baseline compiler will turn the result +=arr[i] line into Stub, but because this instruction is polymorphic (there is nothing to ensuring every time or that arr[i] is going to be string for every position on the array) it’ll create Stub for every possible combination.

- Jit profiler while the code is executed by the interpreter, a profiler will keep track of how many times the different statements get hit. The moment that number start growing, it will mark it as warm and if it grows enough, it will mark as hot. In other words, it will detect which part of your code are being used the most, then it will send them over to be compiled and stored.

Seven tips for improving performance by leveraging optimizations in v8

Tip1: Declare object properties in constructor
Tip2: Keep object property ordering constant
Tip3: Fix function argument types
Tip4: Declare classes in script scope

Tip5: Use for …in
Tip6: Irrelevant characters do not affect performance
Tip7: Try/catch/finally is not ruinous

How to write optimized JavaScript

1. **Order of object properties**: always instantiate your object properties in the same order so that hidden classes, and subsequently optimized code, can be shared.

2. **Dynamic properties**: adding properties to an object after instantiation will force a hidden class change and slow down any methods that were optimized for the previous hidden class. Instead, assign all an object's properties in its constructor.

3. **Methods**: code that executes the same method repeatedly will run faster than code that executes many different methods only once (due to inline caching).

4. **Arrays:** avoid sparse arrays where keys are not incremental numbers. Sparse arrays which don't have every element inside them are a **hash table**. Elements in such arrays are more expensive to access. Also, try to avoid pre-allocating large arrays. It's better to grow as you go. Finally, don't delete elements in arrays. It makes the keys sparse.

5. **Tagged values**: V8 represents objects and numbers with 32 bits. It uses a bit to know if it is an object (flag = 1) or an integer (flag = 0) called SMI (SMall Integer) because of its 31 bits. Then, if a numeric value is bigger than 31 bits, V8 will box the number, turning it into a double and creating a new object to put the number inside. Try to use 31-bit signed numbers whenever possible to avoid the expensive boxing operation into a JS object.