2.

```
 let nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
//
let odds = nums.filter(val => val % 2);
//
let even = nums.filter(item => odds.indexOf(item) < 0);
console.log(even);
```

3.

```
Using the navigator object
```

4.

```
 The above code will output the following to the console:
outer func:  this.foo = bar
outer func:  self.foo = bar
inner func:  this.foo = undefined
inner func:  self.foo = bar
```

> In the outer function, both this and self refer to myObject and therefore both can properly reference and access foo. In the inner function, though, this no longer refers to myObject. As a result, this.foo is undefined in the inner function, whereas the reference to the local variable self remains in scope and is accessible there.

5.

> The reason for this has to do with the fact that semicolons are technically optional in JavaScript (*although omitting them is generally really bad form*). As a result, when the line containing the `return` statement (*with nothing else on the line*) is encountered in `foo2()`, a semicolon is automatically inserted immediately after the `return` statement.

6.

```
console.log(1 +  +"2" + "2");
//  Outputs: "32"
```

> Based on order of operations, the first operation to be performed is +"2" (the extra + before the first "2" is treated as a unary operator). Thus, JavaScript converts the type of "2" to numeric and then applies the unary + sign to it (i.e., treats it as a positive number). As a result, the next operation is now 1 + 2 which of course yields 3. But then, we have an operation between a number and a string (i.e., 3 and "2"), so once again JavaScript converts the type of the numeric value to a string and performs string concatenation, yielding "32".

7.

```
The output of this code will be 456 (not 123).
```

> The reason for this is as follows:
>
> When setting an object property, JavaScript will implicitly stringify the parameter value. In this case, since b and c are both objects, they will both be converted to "[object Object]".
>
> As a result, a[b] anda[c] are both equivalent to a["[object Object]"] and can be used interchangeably.
>
> Therefore, setting or referencing a[c] is precisely the same as setting or referencing a[b].

8.

```
 // the code will output:

console.log(stoleSecretIdentity());
console.log(hero.getSecretIdentity());
//
>>>   undefined
>>>   John Doe
```

> The first console.log prints undefined because we are extracting the method from the hero object, so `stoleSecretIdentity()` is being invoked in the global context (i.e., the window object) where the `_name` property does not exist.

9.

> The reason is that when the function is executed, it checks that there’s a local x variable present but doesn’t yet declare it, so it won’t look for global one.)

10.

> The first statement returns true which is as expected.
>
> The second returns false because of how the engine works regarding operator associativity for < and >. It compares left to right, so 3 > 2 > 1 JavaScript translates to true > 1. true has value 1, so it then compares 1 > 1, which is false.