1S Analise a figura e as proposições a seguir.

	Extra small <576px	Small ≥576px	Medium ≥768px	Large ≥992px	X-Large ≥1200px	XX-Large ≥1400px
.container	100%	540px	720px	960px	1140px	1320px
.container-sm	100%	540px	720px	960px	1140px	1320px
.container-md	100%	100%	720px	960px	1140px	1320px
.container-lg	100%	100%	100%	960px	1140px	1320px
.container-xl	100%	100%	100%	100%	1140px	1320px
.container-xxl	100%	100%	100%	100%	100%	1320px
.container-fluid	100%	100%	100%	100%	100%	100%

I A figura revela que o Grid System do Bootstrap reserva espaços para margens laterais quando a tela do dispositivo tem menos de 576 pixels.

II A figura revela que o Grid System do Bootstrap reserva espaços para margens laterais para todas as telas que tenham, pelo menos, 1400 pixels de largura.

III A figura revela que o Grid System do Bootstrap admite o uso de uma classe que reserva 100% do espaço da tela de maneira incondicional.

É correto apenas o que se afirma em

I
II
III
I e II
II e III

Feedback

A proposição I é falsa. O funcionamento do Grid System para telas com menos de 576 pixels está retratado na primeira coluna da figura. Ela mostra que, independentemente da classe container escolhida, o conteúdo ocupa sempre 100% da tela, não deixando espaço para margens laterais.
A proposição II é falsa. O funcionamento do Grid System para telas com pelo menos 1400 pixels está retratado na última coluna da figura. A existência de margens laterais depende, também, da classe escolhida. A última linha revela que não há margens laterais caso a classe escolhida seja container-fluid, ainda que a tela tenha pelo menos 1400 pixels de largura.
A proposição III é verdadeira. Essa classe se chama container-fluid e seu funcionamento está retratado na última linha da tabela.

2S Analise as figuras e proposições a seguir.

```
10      <div class="container">
11
12        <div class="row">
13          <div class="col-md-4 col-lg-6">
14            <div class="p-4 border">
15              Caixa 1
16            </div>
17          </div>
18          <div class="col-md-4 col-lg-4">
19            <div class="p-4 border">
20              Caixa 2
21            </div>
22          </div>
23        </div>
24
25        <div class="row">
26          <div class="col-md-4">
27            <div class="p-4 border">
28              Caixa 3
29            </div>
30          </div>
31        </div>
32
33      </div>
```

	Extra small <576px	Small ≥576px	Medium ≥768px	Large ≥992px	X-Large ≥1200px	XX-Large ≥1400px
.container	100%	540px	720px	960px	1140px	1320px
.container-sm	100%	540px	720px	960px	1140px	1320px
.container-md	100%	100%	720px	960px	1140px	1320px
.container-lg	100%	100%	100%	960px	1140px	1320px
.container-xl	100%	100%	100%	100%	1140px	1320px
.container-xxl	100%	100%	100%	100%	100%	1320px
.container-fluid	100%	100%	100%	100%	100%	100%

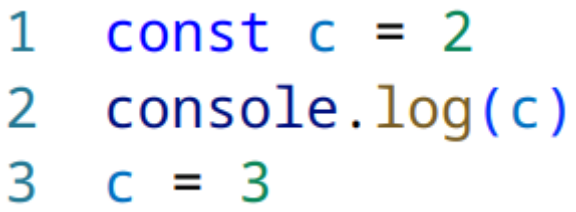
I As três caixas ficam lado a lado caso a tela do dispositivo tenha pelo menos 768 pixels.
II Caixa 1 e Caixa 2 ficam lado a lado mesmo quando a tela é pelo menos “grande” (lg).
III Há casos em que as três caixas ocupam 12 colunas.

É correto apenas o que se afirma em
I
II
III
I e II
II e III

Feedback

A proposição I é falsa. Há duas “rows”. As duas primeiras caixas estão na primeira. A terceira caixa está na segunda. A terceira caixa sempre fica abaixo das duas primeiras, independentemente do tamanho da tela ou número de colunas escolhido.
A proposição II é verdadeira. As caixas 1 e 2 estão na mesma “row” e, quando a tela é pelo menos grande (lg), a primeira ocupa 6 colunas e a segunda ocupa 4 colunas, sobrando ainda 2 colunas do Grid System do Bootstrap.
A proposição III é verdadeira. Quando não especificamos uma quantidade de colunas, fica implícito que desejamos 12. E, de fato, não especificamos o número de colunas desejado para nenhuma das caixas quando a tela for extra pequena, por exemplo. Neste caso, todas elas terão 12 colunas.

3S Analise o trecho de código e as proposições a seguir.



I O programa exibe o número 2.
II O programa entra em execução e causa um erro.
III A remoção da linha 3 não altera o comportamento atual do programa

É correto apenas o que se afirma em

I
II
III
I e II
II e III

Feedback

A proposição I é verdadeira. A linha 2 causa a exibição do número 2, ainda que, depois, o programa cause um erro.
A proposição II é verdadeira. O programa entra em execução e exibe o número 2. Na linha 3, há uma tentativa de atribuição a uma constante já inicializada, o que causa o erro mencionado.
A proposição III é falsa. O comportamento deixa de causar um erro caso a linha 3 seja removida, ou seja, seu comportamento passa a ser diferente.

4S Analise o trecho de código e as proposições a seguir.

```
1  var nome
2  var vaiChover = true
3  if(vaiChover){
4    console.log(nome + ', leve guarda-chuva')
5    nome = 'João'
6  }
7  console.log('Até mais, ' + nome)
```

I O programa exibe a palavra undefined pelo menos uma vez.
II Há um erro que impede a execução de todas as linhas.
III O programa apresenta um erro na linha 5.

É correto apenas o que se afirma em

I
II
III
I e II
II e III

Feedback
A proposição I é verdadeira. O programa exibe a palavra undefined na linha 4, já que a variável nome é inicializada apenas na linha 5.
A proposição II é falsa. Embora o programa exiba undefined e esse não seja o comportamento naturalmente esperado, isso não impede a execução de todas as linhas. Não há nenhum outro erro que o faça.
A proposição III é falsa. A linha 5 inicializa uma variável previamente declarada. Tudo certo com ela.

5S Analise o trecho de código e as proposições a seguir.

```
1  console.log(idade)
2  if( idade >= 18){
3    let idade = 18
4    var nome = 'Ana'
5    console.log (`Sim, ${nome}. Você pode dirigir.`)
6  }
7  console.log(`Parabéns pelos seus ${idade} anos.`)
```

I O programa exibe um erro e termina, sem exibir nada além disso.
II O programa ilustra o uso do mecanismo conhecido como “içamento”, do inglês “hoist”.
III O programa exibe um valor, causa um erro e encerra a execução sem executar todas as linhas.

É correto apenas o que se afirma em

I
II
III
I e II
II e III

Feedback

A proposição I é verdadeira. Embora a variável idade tenha sido declarada, ela foi declarada utilizando-se let e isso aconteceu dentro do bloco if. Ou seja, seu escopo está restrito àquele bloco, já que ela não é envolvida no mecanismo de içamento do Javascript. A linha 1 tenta usar uma variável que não existe e causa o encerramento do programa, exibindo uma mensagem de erro e nada além disso.
A proposição II é verdadeira. A variável nome foi declarada com “var”. Por isso, ela é içada para fora da estrutura if.
A proposição III é falsa. Embora a variável idade tenha sido declarada, ela foi declarada utilizando-se let e isso aconteceu dentro do bloco if. Ou seja, seu escopo está restrito àquele bloco, já que ela não é envolvida no mecanismo de içamento do Javascript. A linha 1 tenta usar uma variável que não existe e causa o encerramento do programa, exibindo uma mensagem de erro e nada além disso.

OBS: O mecanismo hoist está descrito no bloco de código 1.1.2 da apostila de Javascript.

6S Analise o trecho de código e as proposições a seguir.

```
1  const nomes = ['Ana Maria', 'João']
2  const r1 = nomes.every(n => n.startsWith('A'))
3  console.log(r1)
4  const r2 = nomes.reduce((ac, v) => ac + v)
5  console.log(r2)
```

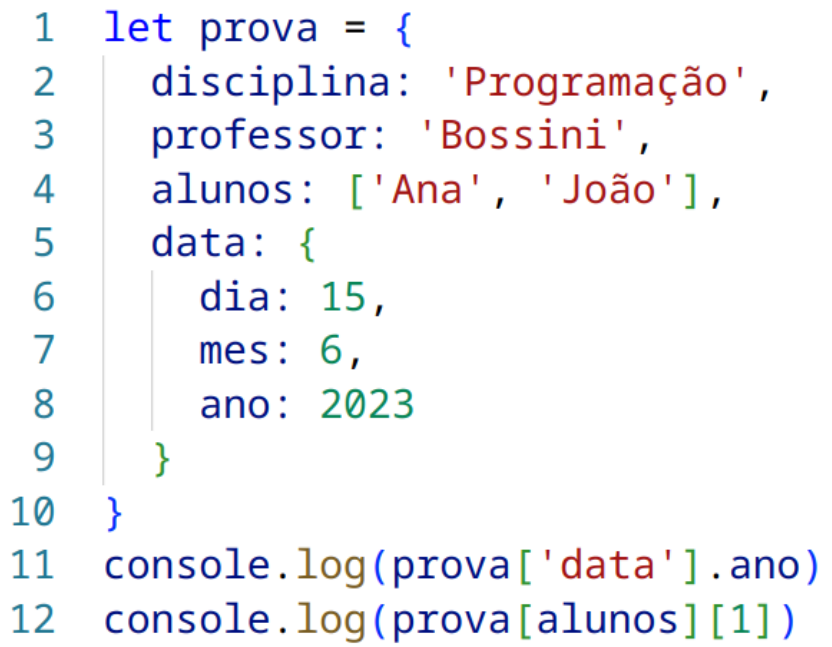
I A linha 2 causa um erro já que a variável n está sendo utilizada sem ter sido declarada.
II A linha 3 exibe false, ainda que exista pelo menos uma pessoa cujo nome começa com A.
III A linha 5 exibe undefined, já que a operação + utilizada pela função reduce na linha 4 não é definida para o tipo dos objetos armazenados no vetor declarado na linha 1.

É correto apenas o que se afirma em

I
II
III
I e II
II e III

Feedback
A proposição I é falsa. n é um parâmetro da arrow function entregue à função every.
A proposição II é verdadeira. A função “every” responde se **todos** os elementos da coleção possuem a característica descrita pela função que recebe como parâmetro. Não basta que apenas um possua. Como “João” não começa com A, ela devolve false.
A proposição III é falsa. A operação + é, sim, definida para os objetos do vetor da linha 1. Eles são strings. Portanto, + realiza a concatenação entre eles. O resultado é “Ana MariaJoão”.

7S Analise o trecho de código e as proposições a seguir.



```
1  let prova = {
2    disciplina: 'Programação',
3    professor: 'Bossini',
4    alunos: ['Ana', 'João'],
5    data: {
6      dia: 15,
7      mes: 6,
8      ano: 2023
9    }
10 }
11 console.log(prova['data'].ano)
12 console.log(prova[alunos][1])
```

I. A linha 11 mostra 2023.
II. As linhas de 5 a 9 definem um objeto aninhado.
III. O objeto referenciado por prova é inválido, já que ele possui uma coleção como valor associado a uma de suas chaves.

É correto apenas o que se afirma em

I
II
III
I e II
II e III

Feedback
A proposição I é verdadeira. A linha 11 acessa a o valor associado à chave ano corretamente, utilizando colchetes para encontrar o objeto associado à chave data e o operador . para encontrar o valor associado à chave ano. Ambos os operadores são válidos e podem ser combinados.
A proposição II é verdadeira. prova referencia um objeto e o valor associado à chave data também é um objeto. Este cenário caracteriza o que chamamos de objeto aninhado.
A proposição III é falsa. Coleções Javascript são objetos válidos e nada impede que sejam valores associados a chaves de outros objetos.

8S Analise o trecho de código e as proposições a seguir.

```
1   const fs = require("fs");
2   const abrirArquivo = function (nomeArquivo) {
3     const exibirConteudo = function (erro, conteudo) {
4       if (erro) {
5         console.log(`Deu erro: ${erro}`);
6       } else {
7         console.log(conteudo.toString());
8         const dobro = +conteudo.toString() * 2;
9         const finalizar = function (erro){
10            if(erro){
11                console.log('Deu erro tentando salvar o dobro')
12            }
13            else{
14                console.log("Salvou o dobro com sucesso");
15            }
16        }
17        fs.writeFile('dobro.txt', dobro.toString(), finalizar);
18      }
19    };
20
21    fs.readFile(nomeArquivo, exibirConteudo);
22  };
23  abrirArquivo("arquivo.txt");
```

I O código faz uso de promises e, por isso, caracteriza aquilo que chamamos de inferno de callbacks.
II O código ilustra o que conhecemos como processamento assíncrono.
III O código utiliza pelo menos uma função callback.

É correto apenas o que se afirma em

I
II
III
I e II
II e III

A proposição I é falsa. O código faz uso exclusivo de funções callback. Não há promise alguma, o que pode ser verificado, também, pela inexistência das construções then/catch e/ou async/await.
A proposição II é verdadeira. Na linha 17, por exemplo, entregamos a função “finalizar” como parâmetro à função writeFile. Ela fica registrada para execução futura quando a writeFile terminar, caracterizando o processamento assíncrono.

A proposição III é verdadeira. Funções callback são aqueles que definimos mas nunca chamamos explicitamente. Elas são chamadas pelo ambiente quando um evento de interesse acontece. A função finalizar é um exemplo de função callback. Ela foi definida nas linhas de 9 a 16 mas nunca chamada explicitamente. Ela é chamada implicitamente pela função writeFile, quando ela termina de fazer o que promete.

9S Analise o trecho de código e as proposições a seguir.

```
1   function calculoRapidinho(numero) {
2     return numero >= 0
3       ? Promise.resolve((numero * (numero + 1)) / 2)
4       : Promise.reject("Somente valores positivos, por favor");
5   }
6
7   calculoRapidinho(10)
8     .then((resultado) => {
9       console.log(resultado);
10    })
11    .catch((err) => {
12      console.log(err);
13    });
14  calculoRapidinho(-1)
15    .then((resultado) => {
16      console.log(resultado);
17    })
18    .catch((err) => {
19      console.log(err);
20    });
21  console.log("esperando...");
```

I A função definida na linha I devolve uma Promise.
II A função definida na linha I opera de maneira assíncrona.
III O exemplo causa a execução de ambas as linhas 9 e 16.

É correto apenas o que se afirma em

I
II
III
I e II
II e III

Feedback
A proposição I é verdadeira. Dependendo do valor da variável numero, a função devolve uma Promise no estado fullfilled ou rejected. Essa decisão é tomada por um operador ternário.
A proposição II é falsa. Embora devolva uma Promise, a função não opera de maneira assíncrona. Por exemplo:

const resultado = calculoRapidinho(numero)

A função que executa esse bloquinho fica bloqueada esperando o término da função calculoRapidinho. Ou seja, processamento bloqueante, síncrono. Não é o caso de ela prosseguir com a sua execução sem esperar a calculoRapidinho terminar, o que caracterizaria o processamento assíncrono.

A proposição III é falsa. A linha 4 faz com que a função calculoRapidinho devolva uma Promise no estado rejected, já que o valor passado como parâmetro é negativo. Neste caso, a linha 16 não executa. A linha 19 é que entra em cena, exibindo o erro.

10S Analise o trecho de código e as proposições a seguir.

```
1  const eAgora = async() => {
2    console.log(1)
3    console.log(2)
4    console.log(3)
5    return '123'
6  }
7
8  async function teste(){
9    console.log(eAgora())
10 }
11 teste()
```

I A função referenciada por eAgora devolve uma Promise.
II A linha 9 exibe apenas o texto “123”.
III Dado que a função teste foi marcada como async, a linha 9 pode ser reescrita da seguinte forma, sem que isso cause qualquer erro ou altere aquilo que o programa exibe.

```
console.log(await eAgora())
```

É correto apenas o que se afirma em

I
II
III
I e II
II e III

Feedback
A proposição I é verdadeira. Embora a função referenciada por eAgora devolva uma String explicitamente, ela foi marcada como async. Implicitamente, ela passa a devolver uma Promise.

A proposição II é falsa. Dado que a função referenciada por eAgora devolve uma Promise, a linha 9 exibe a representação textual dela: Promise { ‘123’ }
A proposição III é falsa. O uso de await não causa erro algum. Porém, a função passa a exibir o resultado produzido pela computação a que a Promise ficou associada, ou seja, apenas a string “123”.