

Exercício 04

1. (V) Em **TypeScript**, os **objetos** podem ser vistos como instâncias ou modelos das classes, e isso pode ser exemplificado através da criação de classes e objetos na linguagem. Irei demonstrar isso em código TypeScript disponibilizado no VS Code abaixo:



```
TS carro_molde.ts 3 x JS carro_molde.js
TS carro_molde.ts > Carro
1 class Carro {
2   marca : string;
3   modelo : string;
4   ano : number;
5
6   constructor (marca : string, modelo : string, ano : number) {
7     this.marca = marca;
8     this.modelo = modelo;
9     this.ano = ano;
10  }
11
12   acelerar() {
13     console.log(`O ${this.marca} ${this.modelo} está acelerando.`);
14   }
15
16   frear() {
17     console.log(`O ${this.marca} ${this.modelo} está freando.`);
18   }
19 }
20
21 // Instanciando objetos da classe Carro
22 let carro1 = new Carro('Toyota', 'Japonesa', 2022);
23 let carro2 = new Carro('Hyundai', 'Sul-Coreana', 2024);
24
```

Considere o seguinte exemplo de uma classe “Carro” que define características e comportamentos comuns de um carro:

Aqui, “Carro” é uma classe que define propriedades como “marca”, “modelo” e “ano”, além de métodos como “acelerar()” e “frear()”. Agora, em “carro1” e em “carro2” são objetos que foram criados a partir da classe “Carro”. Eles representam instâncias específicas do conceito de carro definido pela classe.

Agora, a **Justificativa em si**:

- **Encapsulamento**: A classe “Carro” encapsula as propriedades e métodos relacionados a um carro. Os objetos “carro1” e “carro2” encapsulam dados específicos, como marca, modelo e ano, e implementam os comportamentos definidos pelos métodos “acelerar()” e “frear()”. Portanto, os objetos são

modelos para suas classes em termos de encapsulamento de dados e comportamentos.

- **Abstração:** A classe “Carro” fornece uma abstração do conceito de carro, especificando características e comportamentos comuns que os objetos do tipo carro podem ter. Os objetos “carro1” e “carro2” são instâncias dessa abstração, representando casos específicos do conceito de carro definido pela classe.

Portanto, em **TypeScript**, **os objetos** podem ser considerados modelos para as classes, pois encapsulam dados e comportamentos conforme definido pelas classes, e representa instâncias concretas dos conceitos abstratos representados pelas classes.

Com isso, a Programação Orientada a Objetos é um paradigma de programação que se baseia no conceito de objetos. Isto é, um objeto contém informações e ações, que interagem entre si durante a execução do programa. Geralmente, um objeto é uma representação de algo do mundo real. Irei comentar mais um exemplo simples:

Objeto do tipo **celular**

Modelo: Iphone 5S;

Câmera: 8 mp;

Memória Interna: 64 gb;

Tipo: Smartphone;

Marca: Apple;

Objeto do tipo **celular**

Comentário sobre: Acima, temos o objeto do tipo celular. Esse objeto contém características que lhe são particulares, por exemplo, é um Iphone 5S, tem câmera de 8 mp, memória interna de 64 gb, ... agora, iremos ver outro exemplo:

Objeto do tipo **celular**

Modelo: Lumia 930;

Câmera: 21 mp;

Memória Interna: 32 gb;

Tipo: Smartphone;

Marca: Nokia;

Comentário sobre: Assim como no exemplo anterior, nós também temos um objeto do tipo celular. Porém, apesar de ambos compartilharem os mesmos **atributos**, as suas características são diferentes. Podemos dizer que ambos são **instâncias da Classe Celular**. Classe? Mas como assim?

Já vimos que um objeto tem **informações** (que a partir de agora, chamaremos de **atributos**), e **ações** (que chamaremos de **métodos**). Vimos também que podem existir vários objetos que compartilham dos mesmos atributos, porém com características diferentes. E é aqui que entra a famosa **Classe**.

Na OO, uma **Classe** é um **molde** com o qual os objetos são “**modelados**”. É nela que está discriminado que atributos um objeto deve conter e os métodos relacionados ao mesmo. Por isso, quando eu disse que o Iphone 5S e o Lumia 930 são **instâncias de uma Classe**, eu quis dizer que eles são objetos diferentes, mas que compartilham do mesmo “**molde**”. Ele seria mais ou menos assim:

```
Classe celular {  
  modelo: String;  
  câmera : String;  
  memória interna : String;  
  tipo : String;  
  marca : String;  
}
```

Acima, eu defini a **Classe Celular**, já seguindo um **padrão de programação**. E, por fim, como ela se aplica na prática.

(F) Em **TypeScript**, os atributos e uma classe não precisam ser obrigatoriamente inicializados para que as classes compilem. No entanto, dependendo das configurações do compilador, pode ser necessário garantir que esses atributos sejam inicializados antes de serem usados.

Aqui está um exemplo para ilustrar, disponibilizado no VS Code:



```
TS atributos_classe.ts X  
TS atributos_classe.ts > Pessoa > constructor  
1  class Pessoa {  
2      nome : string;  
3      idade : number;  
4  
5      constructor (nome : string, idade : number) {  
6          this.nome = nome;  
7          this.idade = idade;  
8      }  
9  
10     apresentar() {  
11         console.log(`Olá, meu nome é ${this.nome} e eu tenho ${this.idade} anos de idade.`);  
12     }  
13 }  
14  
15 let pessoa1 = new Pessoa("Vinicius", 28);  
16 pessoa1.apresentar();  
17  
18
```

Neste exemplo acima, a classe “Pessoa” possui os atributos “**nome**” e “**idade**”, mas eles não são inicializados no momento da declaração. Eles são inicializados no construtor da classe.

Caso haja uma configuração mais rigorosa no TypeScript, como a opção “**strict**”, no arquivo “**tsconfig.json**”, o compilador pode exigir que esses atributos sejam inicializados explicitamente no construtor ou em algum outro lugar antes de serem usados para evitar erros de tipo.

Por exemplo, se a opção “**strict**” estiver habilitada, você pode precisar inicializar os atributos diretamente na declaração ou no construtor da seguinte maneira:

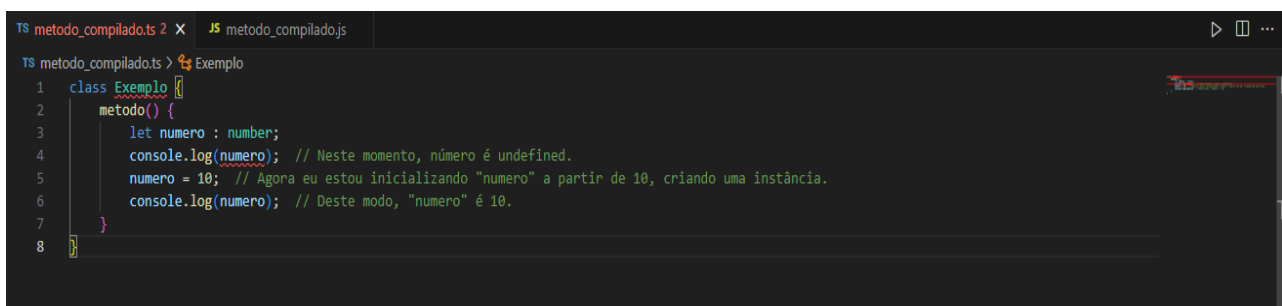


```
TS funcao_strict.ts > Pessoa > constructor
1 class Pessoa {
2     nome : string = "";
3     idade : number = 0;
4
5     constructor (nome : string, idade : number) {
6         this.nome = nome;
7         this.idade = idade;
8     }
9
10    apresentar() {
11        console.log(`Olá, meu nome é ${this.nome} e eu tenho ${this.idade} anos de idade.`);
12    }
13 }
```

Então, em resumo, os atributos de uma classe em TypeScript não precisam ser obrigatoriamente inicializados diretamente na declaração **para que a classe compile**. No entanto, dependendo das configurações do compilador e das regras de estilo de código, pode ser necessário inicializá-los antes de serem usados para evitar erros de tipo.

(F) Em **TypeScript**, variáveis declaradas dentro de um método não precisam ser inicializadas explicitamente para que a classe seja compilável. No entanto, dependendo das configurações do compilador, pode ser necessário garantir que essas variáveis sejam inicializadas antes de serem usadas para evitar erros de tipo ou garantir a conformidade com as regras de estilo do código.

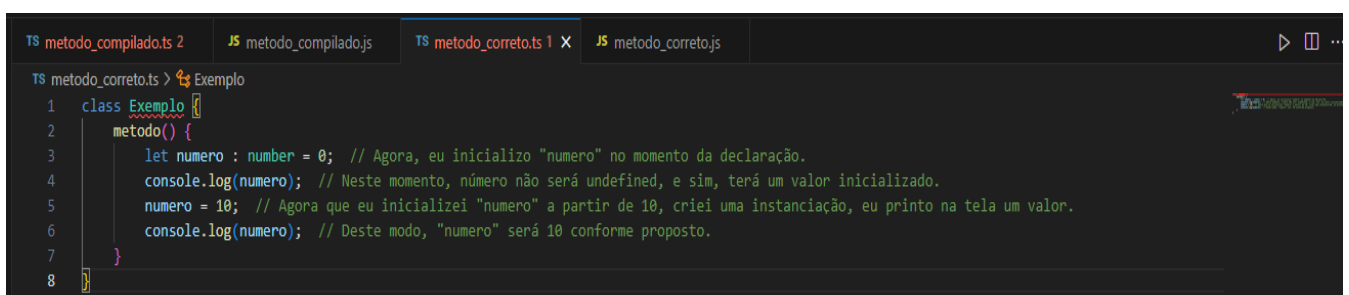
Embaixo, eis um exemplo para demonstrar, disponível também no VS Code:



```
TS metodo_compilado.ts > Exemplo
1 class Exemplo {
2     metodo() {
3         let numero : number;
4         console.log(numero); // Neste momento, número é undefined.
5         numero = 10; // Agora eu estou inicializando "numero" a partir de 10, criando uma instância.
6         console.log(numero); // Deste modo, "numero" é 10.
7     }
8 }
```

Neste exemplo, a variável “**número**” é declarada dentro do método “**método()**” sem ser inicializada explicitamente. Isso é aceitável em **TypeScript**. No entanto, se você estiver utilizando uma configuração mais rigorosa no **TypeScript**, como a opção “**strict**” no arquivo “**tsconfig.json**”, o compilador pode emitir um aviso ou erro indicando que a variável “**numero**” está sendo usada antes de ser inicializada.

Para garantir a inicialização antes do uso, você pode inicializar a variável no momento da declaração ou antes de usar a variável:

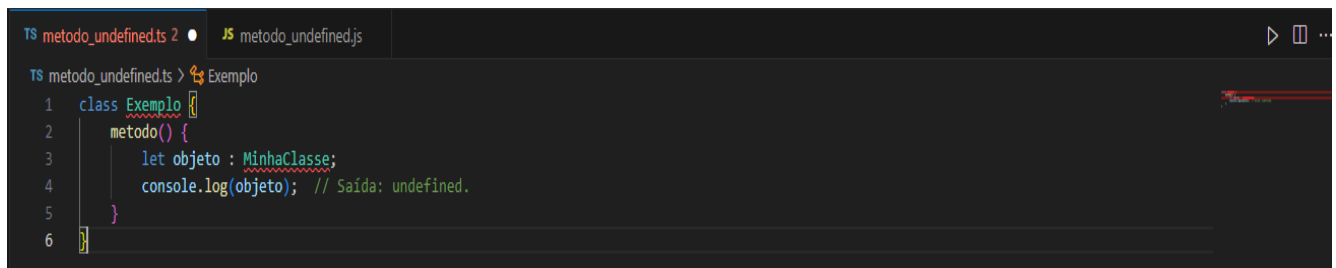


```
TS metodo_correto.ts > Exemplo
1 class Exemplo {
2     metodo() {
3         let numero : number = 0; // Agora, eu inicializo "numero" no momento da declaração.
4         console.log(numero); // Neste momento, número não será undefined, e sim, terá um valor inicializado.
5         numero = 10; // Agora que eu inicializei "numero" a partir de 10, criei uma instância, eu printo na tela um valor.
6         console.log(numero); // Deste modo, "numero" será 10 conforme proposto.
7     }
8 }
```

Em resumo, em **TypeScript**, variáveis declaradas dentro de métodos não precisam ser obrigatoriamente inicializadas no momento da declaração para que a classe seja compilável. No entanto, é uma boa prática inicializar as variáveis antes de usá-las para evitar problemas de tipo e garantir a clareza e a segurança do código.

(V) Sim, em **TypeScript**, se uma variável que é uma classe é declarada dentro de um método, ela é automaticamente inicializada com “**undefined**”. Isso é parte do comportamento padrão de inicialização de variáveis em **TypeScript** e em muitas outras linguagens de programação.

Por exemplo:

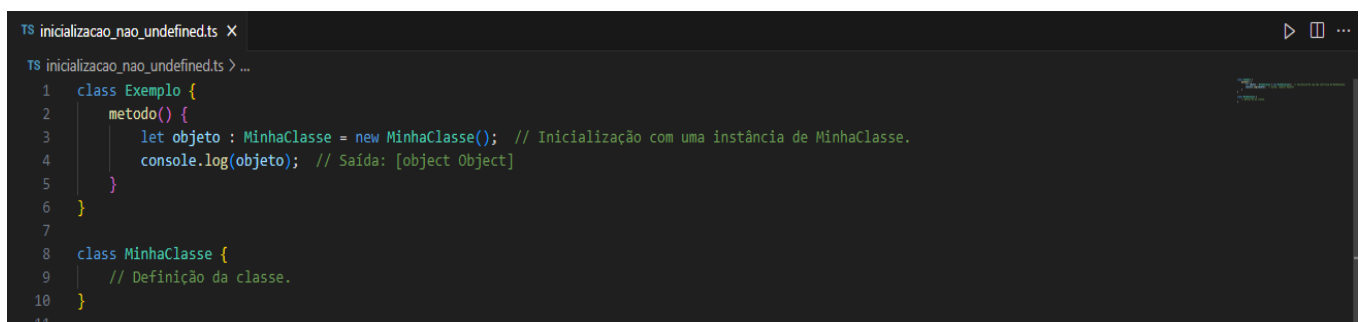


```
TS metodo_undefined.ts 2 JS metodo_undefined.js
TS metodo_undefined.ts > Exemplo
1 class Exemplo {
2   metodo() {
3     let objeto : MinhaClasse;
4     console.log(objeto); // Saída: undefined.
5   }
6 }
```

Neste exemplo, a variável do tipo “**objeto**”, que é do tipo “**MinhaClasse**”, é declarada dentro do método “**metodo()**” da classe “**Exemplo**”, mas não é inicializada com nenhum valor. Portanto, seu valor padrão é “**undefined**”.

É importante observar que embora a variável seja inicializada com “**undefined**”, a referência à classe em si ainda não está definida. Se você tentar acessar qualquer propriedade ou método dessa variável antes de atribuir um valor a ela, resultará em **um erro de tempo de execução**, indicando que a propriedade ou método não pode ser acessada em “**undefined**”.

Para evitar erros, é uma boa prática inicializar explicitamente a variável com um valor apropriado antes de usá-la:



```
TS inicializacao_nao_undefined.ts X
TS inicializacao_nao_undefined.ts > ...
1 class Exemplo {
2   metodo() {
3     let objeto : MinhaClasse = new MinhaClasse(); // Inicialização com uma instância de MinhaClasse.
4     console.log(objeto); // Saída: [object Object]
5   }
6 }
7
8 class MinhaClasse {
9   // Definição da classe.
10 }
11
```

Neste caso, “**objeto**” é inicializado como uma instância de “**MinhaClasse**”, garantindo que ele não seja mais “**undefined**” quando acessado dentro do método “**metodo()**”.

(V) Sim, em **TypeScript**, assim como em outras linguagens de programação orientadas a objetos, os construtores são rotinas especiais dentro de uma classe que servem para inicializar e configurar os objetos no momento da instanciação.

Um **construtor** é um **método especial** dentro de uma classe que é chamado automaticamente quando um objeto da classe é criado usando um operador “**new**”. Ele tem o mesmo nome da classe e pode aceitar parâmetros para configurar o objeto dentro da instanciação.

Aqui está um exemplo simples em **TypeScript**:



```
TS construtores_rotina_especial.ts X
TS construtores_rotina_especial.ts > Pessoa > constructor
1 class Pessoa {
2   nome : string;
3   idade : number;
4
5   constructor(nome : string, idade : number) {
6     this.nome = nome;
7     this.idade = idade;
8   }
9
10  apresentar() {
11    console.log(`Olá, meu nome é ${this.nome} e eu tenho ${this.idade} anos de idade.`);
12  }
13 }
14 // Criando um objeto a partir da classe Pessoa
15 let pessoa1 = new Pessoa("Vinicius", 28);
16
17 // Chamando o método apresentar() do Objeto
18 pessoa1.apresentar();
19
```

No exemplo acima, a classe “**Pessoa**” tem um construtor que aceita dois parâmetros, “**nome**” e “**idade**”, e inicializa as propriedades correspondentes do objeto (“**this.nome**” e “**this.idade**”) com esses valores.

Quando um objeto da classe Pessoa é criado usando “**new Pessoa(“Vinicius”,28)**”, o construtor é automaticamente invocado com os argumentos fornecidos, configurando assim o objeto com os valores desejados.

Portanto, em **TypeScript**, os construtores desempenham um papel fundamental na inicialização e configuração de objetos no momento da instânciação, permitindo que o desenvolvedor forneça valores iniciais para as propriedades do objeto.

(V) Em **TypeScript**, os construtores não possuem tipo de retorno e podem ou não ter parâmetros. Isso acontece por algumas razões.

- **Ausência de Tipo de Retorno:** os construtores não possuem um tipo de retorno explicitamente declarado porque seu objetivo principal é inicializar e configurar objetos. Eles não retornam valores como funções comuns. Em vez disso, eles são responsáveis por configurar o estado inicial de um objeto.
- **Flexibilidade nos Parâmetros:** Os construtores podem ou não ter parâmetros, dependendo das necessidades da classe e da lógica de inicialização do objeto. Se uma classe exigir informações específicas para inicializar seus objetos, é comum ter parâmetros no construtor para receber essas informações. No entanto, se a classe puder ser inicializada com valores padrão ou se não precisar de informações externas para ser inicializada, o construtor pode não ter parâmetros.

Aqui está um exemplo para ilustrar:

```

TS construtores_tipo_retomados X
TS construtores_tipo_retomados > Produto
1 class Pessoa {
2     nome : string;
3     idade : number;
4
5     // Construtor sem parâmetros
6     constructor() {
7         this.nome = "";
8         this.idade = 0;
9     }
10
11     apresentar() {
12         console.log("Olá, meu nome é ${this.nome} e eu tenho ${this.idade} anos de idade.");
13     }
14 }
15
16 // Construindo um objeto da classe Pessoa sem parâmetros
17 let pessoa1 = new Pessoa();
18 pessoa1.apresentar(); // Saída, meu nome é Sem nome e eu tenho 0 anos de idade.
19
20 // Construtor com parâmetros
21 class Produto {
22     constructor(public nome : string, public preco: number) {
23         // Os parâmetros public nome e public preco já inicializam automaticamente as propriedades correspondentes
24     }
25
26     exibirDetalhes() {
27         console.log(`Nome : ${this.nome}, Preço : R${this.preco}`);
28     }
29 }
30
31 // Construindo um objeto da classe Produto com parâmetros
32 let produto1 = new Produto("Notebook", 4100);
33 produto1.exibirDetalhes(); // Saída: Nome : Notebook, Preço: R$4100.

```

Neste exemplo acima, a classe “**Pessoa**” possui um construtor sem parâmetros, enquanto a classe “**Produto**” possui um construtor com parâmetros. Isso demonstra a flexibilidade nos parâmetros dos construtores em **TypeScript**.

(V) Sim, em **TypeScript**, assim como em outras linguagens de programação orientadas a objetos, uma classe pode ter várias instâncias. Isso significa que você pode criar múltiplos objetos (ou instâncias) a partir da mesma classe.

A justificativa para isso é a natureza da programação orientada a objetos, onde as classes são como **moldes** ou **blueprints** que definem a estrutura e o comportamento de um tipo de objeto. Cada instância da classe representa um objeto separado e independente, com seu próprio conjunto de propriedades e métodos.

Aqui está um exemplo em **TypeScript**:

```

TS instancias_da_classe.ts X
TS instancias_da_classe.ts > Carro > modelo
1 class Carro {
2     marca : string;
3     modelo : string;
4
5     constructor (marca : string, modelo : string) {
6         this.marca = marca;
7         this.modelo = modelo;
8     }
9
10    acelerar() {
11        console.log(`O ${this.marca} da ${this.modelo} está acelerando.`);
12    }
13 }
14
15 // Criando instâncias da classe Carro
16 let carro1 = new Carro("Ferrari", "Chevrolet");
17 let carro2 = new Carro("Honda", "HSUV");
18
19 // Chamando métodos a partir das instâncias
20 carro1.acelerar(); // Saída: O Ferrari da Chevrolet está acelerando.
21 carro2.acelerar(); // Saída: O Honda da HSUV está acelerando.
22
23 // Fim.

```

Neste exemplo, a classe “**Carro**” é usada para criar duas instâncias diferentes (“**carro1**” e “**carro2**”), cada uma representando um carro específico com sua própria marca e modelo. Ambas as instâncias compartilham a mesma estrutura e

comportamentos definidos pela classe, mas têm dados distintos para as propriedades “**marca**” e “**modelo**”.

Portanto, uma classe em **TypeScript** pode ter várias instâncias, permitindo a criação de múltiplos objetos que seguem a mesma estrutura e comportamento definidos pela classe. Isso é fundamental para a reutilização de código e a criação de sistemas flexíveis e modulares.

2. Sim. Haverá um problema de compilação, pois a variável “**quantReservas**” não foi inicializada previamente e, no **TypeScript**, todas as variáveis devem ser inicializadas antes de serem usadas.

Aqui está uma amostra do código corrigido abaixo:

```
TS quant_reservas_hotel.ts X
C: > OneDrive > Documentos > Semestre 2024.1_Matérias_Pagantes > Programação_Orientada_Objeto_2024.1 > Atividade 04 de POO_Turma Especial_2024.1 > TS quant_reservas_hotel.ts > Hotel
1 class Hotel {
2     quantReservas : number = 0; // Inicializando a variável quantReservas
3
4     adicionarReserva() : void {
5         this.quantReservas++; // Incrementando o número de reservas.
6     }
7 }
```

Neste código acima, inicializamos “**quantReservas**” com o valor zero. Isso resolve problema de compilação, garantindo que a variável seja inicializada antes de ser utilizada.

3. Autoexplicativa, demonstrada no VS Code.
4. O erro de compilação ocorre porque a classe “**Radio**” possui um construtor que espera um parâmetro do tipo “**number**”, mas quando você instancia um objeto da classe “**Radio**” usando “**new Radio()**”, você não está passando nenhum argumento para o construtor, o que resulta em um erro.

Uma solução seria fornecer **um valor padrão para o parâmetro do construtor** ou tornar o parâmetro opcional. No código demonstrado no VS Code eu mostro que, nesta solução, o parâmetro “**volume**” do construtor agora tem um valor padrão de “**0**”, o que significa que, se nenhum argumento for passado **ao criar uma instância de “Radio”**, o volume será inicializado como “**0**”. Isso evitará o erro de compilação.

5. De acordo com os slides trabalhados em sala de aula e com os conceitos definidos de Orientação a Objetos, percebe-se que para responder **a pergunta da 5ª questão, letra a**: é preciso entender o que acontece em cada etapa do código fornecido. Vamos analisar:
 1. Criamos duas instâncias da classe “**Conta**”: “**c1**” e “**c2**”;
 2. Igualamos “**c1**” a “**c2**”. Isso faz com que “**c1**” aponte para o mesmo objeto que “**c2**”;

3. Igualamos “c3” a “c1”. Agora “c3” também aponta para o mesmo objeto que “c1” e “c2”;
4. Realizamos uma operação de saque em “c1” e uma transferência de “c1” para “c2”.

Analisando o comportamento em código, percebe-se então o resultado o qual demonstrei no VS Code.

De acordo com essa análise, eis a explicação do resultado:

- Após “c1” ser igualado a “c2”, ambos apontam para o mesmo objeto. Portanto, quando realizamos operações em “c1”, elas afetam “c2” também;
- “c1” saca 10 unidades, então o saldo será $100 - 10 = 90$;
- “c1” transfere 50 unidades para “c2”, então o saldo da conta “c2” será $100 + 50 = 150$;
- Como “c3” também está apontando para o mesmo objeto que “c1” e “c2”, seu saldo também será afetado pelas operações realizadas em “c1” e em “c2”. Portanto, “c3” terá o mesmo saldo que “c2”, que é 150;

Portanto, **o resultado dos prints serão 40, 140 e 140.**

Letra b: Justificativa – Quando “c1” é igualado a “c2”, a referência anterior de “c1” é substituída pela referência de “c2”. Isso significa que o objeto para o qual a referência “c1” apontava não é mais referenciado por “c1”. No entanto, como “c3” foi igualado a “c1” antes dessa mudança, “c3” continua a apontar para o mesmo objeto.

Em termos de coleta de lixo, se não houver outras referências para o objeto originalmente apontado por “c1”, ele se tornará elegível para coleta de lixo assim que não houver mais referências para ele. Isso significa que, se não houver outras variáveis ou referências apontando para esse objeto, ele será coletado pelo mecanismo de coleta de lixo do **TypeScript** ou **JavaScript** para liberar a memória ocupada por esse objeto não utilizado.

Portanto, o objeto originalmente referenciado por “c1” será coletado pelo coletor de lixo se não houver outras referências a ele no código.

6. Demonstrada no VS Code.
7. Demonstrada no VS Code.
8. Demonstrada no VS Code.
9. Demonstrada no VS Code.
10. Demonstrada no VS Code.

11. Não sei respondê-la de forma correta. Consultar o professor Ely.