1 Introdução

Neste material, vamos estudar sobre a forma como objetos podem ser construídos utilizando-se boas práticas. Em particular, vamos estudar sobre

- Factory Method

2 Desenvolvimento

Workspace, novo projeto, VS Code, Typescript

Comece criando uma pasta para desempenhar o papel de Workspace. Ou seja, uma pasta que abriga subpastas, cada qual representando um projeto novo. No Windows, você pode usar algo assim:

C:\Users\seuUsuario\Documents\dev\

A seguir, abra um terminal e vincule-o ao diretório que acabou de criar com

cd C:\Users\seuUsuario\Documents\dev

Use

mkdir nome_desejado

para criar uma pasta que abrigará os arquivos desta nova solução computacional. Vincule o VS Code a ela com

code nome_desejado

No VS Code, clique Terminal >> New Terminal para abrir um terminal interno do VS Code, simplificando o trabalho. Use

npm init -y

para criar um projeto. Use

npm i typescript @types/node --save-dev

para instalar

- typescript: esse é o compilador Typescript. Ele é alimentado com código Typescript e produz código Javascript. Há quem chame isso de "**transpilação**". O nome historicamente

usado para esse fenômeno, em que ocorre compilação de uma linguagem a outra e ambas possuem nível semelhante de abstração, é "**compilação fonte a fonte**".

- @types/node: é um pacote que contém definições de tipo do Typescript para o ambiente NodeJS. Viabiliza o funcionamento do "intellisense" (auto complete "inteligente"), verificação de tipos em tempo de compilação, acesso a documentação mais detalhada etc.

Nota. O compilador Typescript e as funcionalidades providas pelo pacote @types/node são necessárias apenas em tempo de compilação. Se um dia o aplicativo for implantado, não será necessário compilar código ou utilizar intellisense em tempo de produção. Por isso usamos a opção --save-dev, instruindo o npm a registrar os pacotes como dependências em tempo de desenvolvimento apenas.

O compilador Typescript pode ter seu funcionamento configurado por meio de um arquivo chamado **tsconfig.json**. Ele pode ser criado utilizando-se o próprio compilador Typescript. Para tal, use

```
npx tsc --init
```

Abra o arquivo tsconfig.json recém criado e ajuste a propriedade **outDir** para que ela fique associada ao valor "./dist". Ou seja, estamos instruindo o compilador Typescript a armazenar o código Javascript compilado na pasta dist. A propriedade provavelmente já existe no arquivo, basta remover o comentário e ajustar o valor.

```
...
// "outFile": "./", /* Specify a file that bundles all outputs into
one JavaScript file. If 'declaration' is true, also designates a file
that bundles all .d.ts output. */
"outDir": "./dist", /* Specify an output folder for all emitted
files. */
// "removeComments": true, /* Disable emitting comments. */
// "noEmit": true,
...
```

Abra o arquivo **package.json** e crie os scripts a seguir.

build: será usado para colocar o compilador Typescript (tsc) em execução, gerando o código Javascript que o NodeJS interpretará.

start: será usado para colocar o NodeJS em execução, alimentado pelo arquivo Javascript compilado.

```
{
  "name": "simplefactory_factorymethod",
  "version": "1.0.0",
  "description": "",
  "main": "app.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "build": "tsc",
    "start": "node dist/app.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "@types/node": "^20.8.10",
    "typescript": "^5.2.2"
  }
}
```

Use

mkdir src

para criar a pasta que armazenará o código-fonte. Crie um arquivo chamado **app.ts** na pasta src. Veja seu conteúdo para o teste inicial.

```
let hello: string = "Hello, Typescript";
console.log(hello)
```

Use

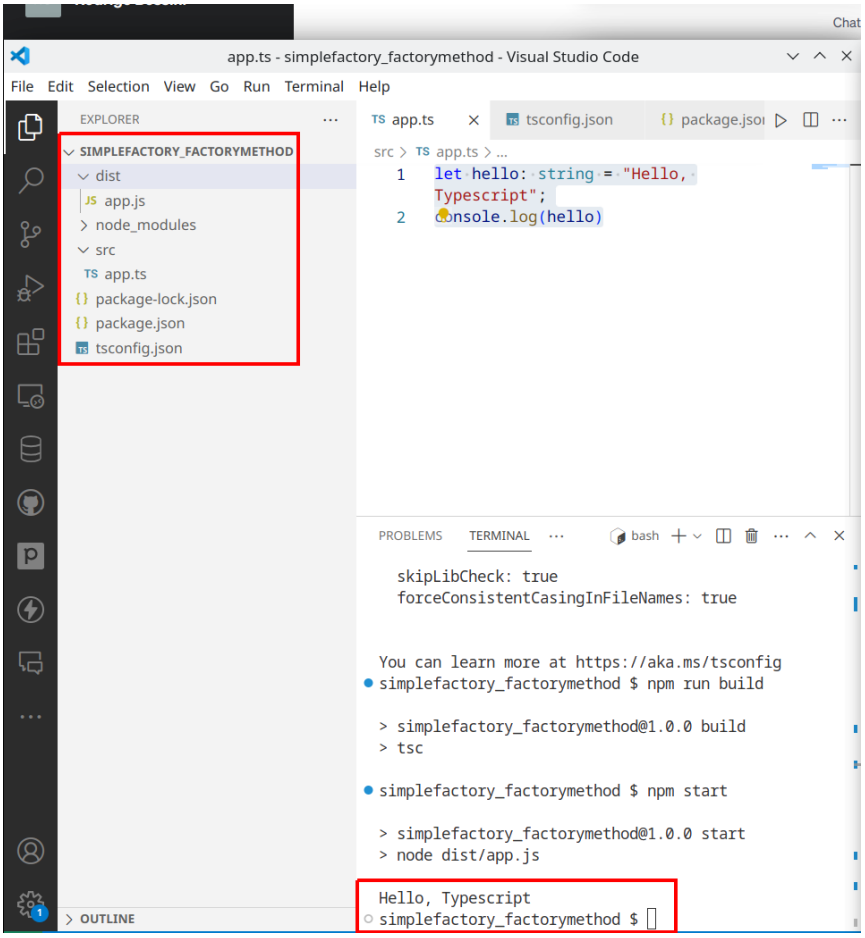
npm run build

para compilar. E

npm start

para executar.

Veja o resultado esperado, incluindo a estrutura de pastas e arquivos.



Para simplificar, você pode fazer uso do pacote **ts-node**. Ele permite que a compilação e execução ocorram de uma única vez. Não é apropriado para ambiente de produção. Mas pode ser muito útil para ambientes de desenvolvimento. Instale-o com

npm i ts-node --save-dev

Depois, ajuste o script start no arquivo **package.json**.

```
...
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1",
  "build": "tsc",
  "start": "ts-node src/app.ts"
},
...
```

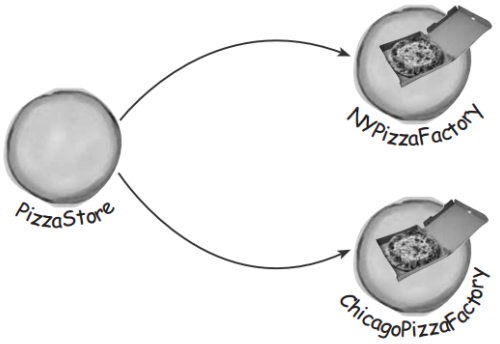
Você pode escolher entre manter como estava antes ou utilizar o ts-node.

Factory Method

A sua pizzaria fez muito sucesso e agora você está se preparando para trabalhar com franquias. Como o dono da franquia, você deseja que seus franqueados ofereçam pizzas de alta qualidade, que mantenham o padrão utilizando seu código que já foi muito testado.

Mas como lidar com diferenças regionais? Cada franquia pode desejar oferecer pizzas em estilos diferentes, dependendo da região (por exemplo, pizzas de New York, Chicago, Califórnia etc).

A figura a seguir exibe o que você deseja. Você quer impor seu código para que as pizzas sejam preparadas sempre do mesmo jeito e ao mesmo tempo quer permitir que cada franquia tenha seu próprio estilo. Cada região poderia utilizar um Factory diferente, não é mesmo?



Exercício. Escreva classes para representar pizzas

- queijo de Chicago e NY
- Pepperoni de Chicago e NY

```
class PizzaDeQueijoDeNY extends PizzaDeQueijo{
 preparar(): void{
   console.log('Preparando pizza de queijo de NY...');
 }

 assar(): void{
   console.log('Assando pizza de queijo de NY...');
 }
```

```
 cortar(): void{
   console.log('Cortando pizza de queijo de NY...');
 }

 empacotar(): void{
   console.log('Empacotando pizza de queijo de NY...');
 }
}

class PizzaDePepperoniDeNY extends PizzaDePepperoni{
 preparar(): void{
   console.log('Preparando pizza de pepperoni de NY...');
 }

 assar(): void{
   console.log('Assando pizza de pepperoni de NY...');
 }

 cortar(): void{
   console.log('Cortando pizza de pepperoni de NY...');
 }

 empacotar(): void{
   console.log('Empacotando pizza de pepperoni de NY...');
 }
}

class PizzaDeQueijoDeChicago extends PizzaDeQueijo{
 preparar(): void{
   console.log('Preparando pizza de queijo de Chicago...');
 }

 assar(): void{
   console.log('Assando pizza de queijo de Chicago...');
 }

 cortar(): void{
   console.log('Cortando pizza de queijo de Chicago...');
 }

 empacotar(): void{
```

```
    console.log('Empacotando pizza de queijo de Chicago...');
  }
}

class PizzaDePepperoniDeChicago extends PizzaDePepperoni{
 preparar(): void{
    console.log('Preparando pizza de pepperoni de Chicago...');
  }

 assar(): void{
    console.log('Assando pizza de pepperoni de Chicago...');
  }

 cortar(): void{
    console.log('Cortando pizza de pepperoni de Chicago...');
  }

 empacotar(): void{
    console.log('Empacotando pizza de pepperoni de Chicago...');
  }
}
```

Exercício. Escreva as duas factories sugeridas. Lembre-se de que cada uma tem que passar no teste É-UM SimplePizzaFactory.

```
//simplefactory ny
class SimplePizzaFactoryNY extends SimplePizzaFactory{
 criarPizza(tipo: string): Pizza | null{
    let pizza: Pizza | null = null;
    if (tipo === 'Queijo')
      pizza = new PizzaDeQueijoDeNY();
    else if (tipo === 'Pepperoni')
      pizza = new PizzaDePepperoniDeNY();
    return pizza;
  }
}

//simplefactory chicago
```

```
class SimplePizzaFactoryChicago extends SimplePizzaFactory{
 criarPizza(tipo: string): Pizza | null{
   let pizza: Pizza | null = null;
   if (tipo === 'Queijo')
     pizza = new PizzaDeQueijoDeChicago();
   else if (tipo === 'Pepperoni')
     pizza = new PizzaDePepperoniDeChicago();
   return pizza;
 }
}
```

Exercício. Escreva uma lista de duas franquias: uma de NY e outra de Chicago.

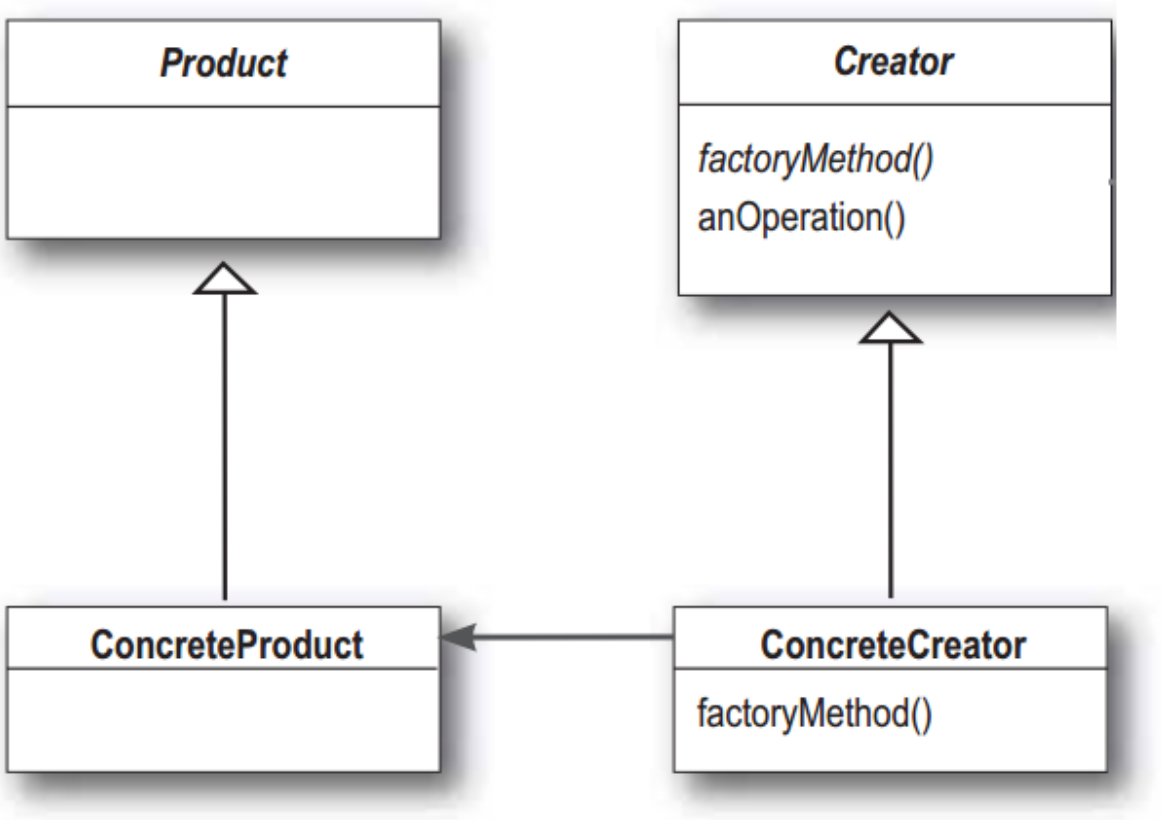
```
//lista com uma pizzastore de ny e outra de chicago
let franquias = [
 new PizzaStore(new SimplePizzaFactoryNY()),
 new PizzaStore(new SimplePizzaFactoryChicago())
];
```

Exercício. Pesquise um pacote que possa ser utilizado para obter dados que o usuário digita no teclado. Permita que o usuário escolha um sabor de pizza e um local de onde deseja que ela seja obtida.

Veja uma definição formal para o Factory Method.

O Design Pattern Factory Method define uma interface para criação de objetos mas permite que subclasses decidam qual classe instanciar. O Factory Method permite que uma classe delegue a criação de objetos para subclasses.

E o seu diagrama de classes.


Product
Creator
factoryMethod()
anOperation()
ConcreteProduct
ConcreteCreator
factoryMethod()

Referências

FREEMAN, Eric; FREEMAN, Elisabeth; SIERRA, Kathy; BATES, Bert. **Head First Design Patterns**. 1. ed. São Francisco: O'Reilly Media, 2004.

GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John. **Design Patterns: Elements of Reusable Object-Oriented Software**. 1. ed. Reading, Massachusetts: Addison-Wesley, 1995.

TypeScript: JavaScript that scales. Disponível em <https://www.typescriptlang.org/>. Acesso em: 03 de novembro de 2023.