Zadanie 1/ programowanie obiektowe

Co jest nie tak z poniższym programem? Czy mimo tego skompiluje się poprawnie?

```
public class SomethingIsWrong {
    public static void main(String[] args) {
        Rectangle myRect;
        myRect.width = 40;
        myRect.height = 50;
        System.out.println("myRect's area= " + myRect.area());
    }
}
```

Zadanie 2/ programowanie obiektowe

W jaki sposób poradzić sobie z uciążliwym ustawianiem wartości pól obiektu (poprzez `this.x = x;`) w podanym przykładzie?

```
class Book {
        private String author;
        private double price;
        private Genre genre;

        public Book(String author){
                this.author = author;
        }
        public Book(String author, double price){
                this.author = author;
                this.price = price;
        }
        public Book(String author, double price, Genre, genre){
                this.author = author;
                this.price = price;
                this.genre = genre;
        }
}
```

Podpowiedź: Należy użyć tzw. konstruktora teleskopowego.

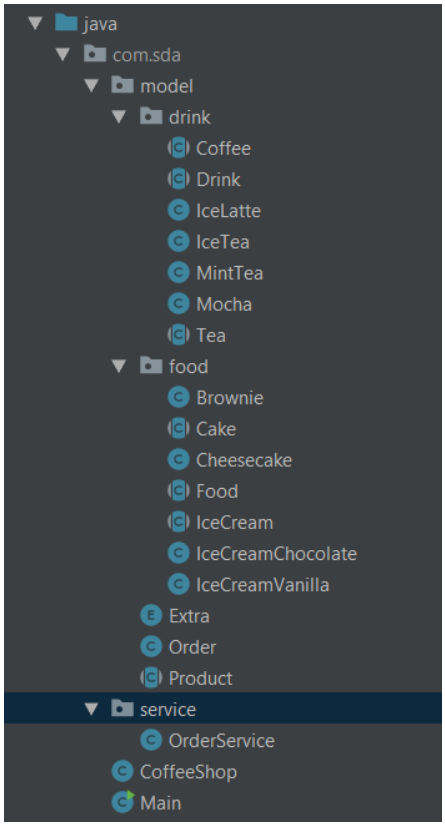
Zadanie 3/ programowanie obiektowe

W jaki sposób poradzić sobie z podobnym problemem jak w zadaniu 2, w przypadku dziedziczenia pól z innych klas? Przykład:

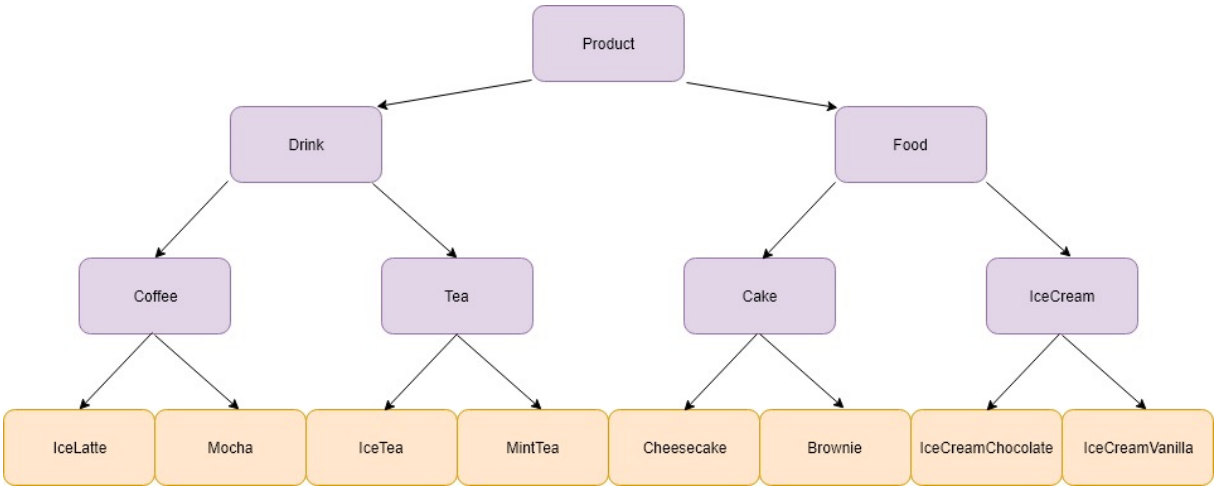
```
public class Drink {
	Size size;
	public Drink(){}
	public Drink(int size){
		this.size = size;
	}
}
public class Tea extends Drink {
	int sugarSpoons;
	public Tea(){}
	public Tea(Size size, int sugarSpoons){
		this.size = size;
		this. sugarSpoons = sugarSpoons;
	}
}
public class IceTea extends Tea {
	int iceCubes;
	public Tea(Size size, int sugarSpoons, int iceCubes){
		this.size = size;
		this.sugarSpoons = sugarSpoons;
		this.iceCubes = iceCubes;
	}
}
```

Zadanie 4/ programowanie obiektowe

Stworzymy model prostego systemu, jakim posługują się np. popularne fastfoody. Utwórz klasy według podanego poniżej schematu pakietów.



Klasy produktów powinny mieć hierarchię, jak na schemacie poniżej (klasami abstrakcyjnymi nie są tylko klasy konkretnych produktów).



Pakiet com.sda.model wypełnia lista typowych POJO (posiadają jedynie pola, konstruktory i gettery). Wszystkie powinny posiadać gettery dla swoich pól.

- Klasa abstrakcyjna `Product` –nadklasa wszystkich możliwych produktów, które można zamówić w kawiarni. Powinna posiadać pole:

`private BigDecimal price;`
i konstruktor wypełniający te pole.

- Enum `Extra` – enum wyliczający możliwe dodatki do zamówienia (SUGAR, MILK, ESPRESSO, COCOA, CREAM, WAFER) z dowolnie dobranymi cenami ustawianymi przez pole
  `private BigDecimal price;`
  i konstruktor wypełniający te pole.
- Klasa finalna `Order` –klasa zamówienia, powinna posiadać pola:
  ```
  private final long oredrID;
  private final List<Product> products;
  private final List<Extra> extras;
  private final BigDecimal price;
  ```
  i jedyny konstruktor – wypełniający wszystkie pola.

- W podpakiecie com.sda.model.drink:
  - Klasa abstrakcyjna `Drink` – nadklasa napojów, dziedziczy z `Product`. Powinna zawierać pole
    `private boolean syrup;`
    i konstruktor wypełniający te pole i pola nadklasy.
  - Klasa abstrakcyjna `Coffee` – nadklasa kaw, dziedziczy z `Drink`. Powinna zawierać pole:
    `private boolean soyMilk;`
    i konstruktor wypełniający te pole i pola nadklasy.
  - Klasa abstrakcyjna `Tea` – nadklasa herbat, dziedziczy z `Drink`. Powinna zawierać pole:
    `private int lemon;`
    i konstruktor wypełniający te pole i pola nadklasy.
  - Klasa `IceLatte` – klasa produktu, dziedziczy z `Coffee`. Powinna zawierać pole:
    `private int iceCubes;`
    i konstruktor wypełniający te pole i pola nadklasy, a `price` ustawiać na sztywno.
  - Klasa `Mocha` – klasa produktu, dziedziczy z `Coffee`. Powinna zawierać pole:
    `private boolean seasoning;`
    i konstruktor wypełniający te pole i pola nadklasy, a `price` ustawiać na sztywno.
  - Klasa `IceTea` – klasa produktu, dziedziczy z `Tea`. Powinna zawierać pole:
    `private boolean mint;`
    i konstruktor wypełniający te pole i pola nadklasy, a `price` ustawiać na sztywno.
  - Klasa `MintTea` – klasa produktu, dziedziczy z `Tea`. Powinna zawierać pole:
    `private boolean longBrewed;`
    i konstruktor wypełniający te pole i pola nadklasy, a `price` ustawiać na sztywno.
- W podpakiecie com.sda.model.food:
  - Klasa abstrakcyjna `Food` – nadklasa jedzenia, dziedziczy z `Product`. Powinna zawierać pole:
    `private boolean cream;`
    i konstruktor wypełniający te pole i pola nadklasy.
  - Klasa abstrakcyjna `Cake` – nadklasa ciast, dziedziczy z `Food`. Powinna zawierać pole:
    `private boolean chocolateIcing;`
    i konstruktor wypełniający te pole i pola nadklasy.
  - Klasa abstrakcyjna `IceCream` – nadklasa lodów, dziedziczy z `Food`. Powinna zawierać pole:

`private int wafers;`
i konstruktor wypełniający te pole i pola nadklasy.

- Klasa `Brownie` – klasa produktu, dziedziczy z `Cake`. Powinna zawierać pole:
  `private boolean chutney;`
  i konstruktor wypełniający te pole i pola nadklasy, a `price` ustawiać na sztywno.
- Klasa `Cheesecake` – klasa produktu, dziedziczy z `Cake`. Powinna zawierać pole:
  `private boolean freshFruits;`
  i konstruktor wypełniający te pole i pola nadklasy, a `price` ustawiać na sztywno.
- Klasy `IceCreamChocolate` i `IceCreamVanilla` – klasy produktu, dziedziczą z `IceCream`.
  Powinny zawierać konstruktor wypełniający pola nadklasy, a `price` ustawiać na sztywno.

Pakiet com.sda.service zawiera klasę `OrderService`, której zadaniem powinno być wyliczenie ceny, numeru zamówienia i stworzenie obiektu zamówienia. Klasa `OrderService` powinna zawierać:

- pole `private long currentOrderNumber;`
- bezparametrowy konstruktor ustawiający powyższe pole na 0,
- metodę `public Order createOrder(List<Product products, List<Extra> extras)`, która:
  - sprawdzi czy przekazywana lista produktów jest nullem i jest pusta i w takim wypadku rzuci odpowiedni wyjątek,
  - obliczy sumę cen produktów i dodatków z podanych list,
  - stworzy nowy obiekt zamówienia (na podstawie list, wyliczonej ceny i jako `orderID` podając `currentOrderNumber`),
  - zinkrementuje `currentOrderNumber`,
  - zwróci obiekt zamówienia.

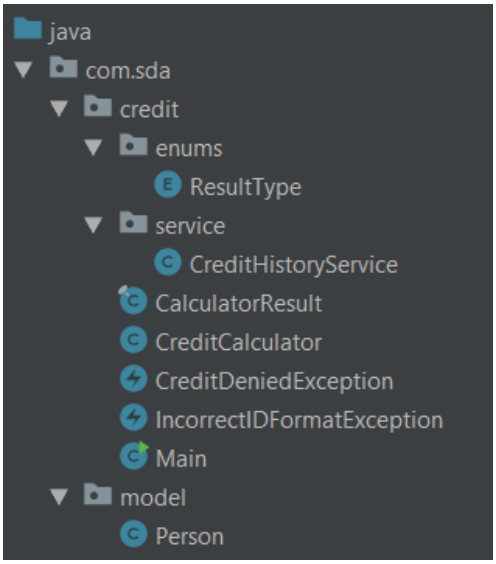
Pakiet com.sda zawiera klasę `CoffeeShop`, klasę kawiarni (główną), która powinna zarządzać zamówieniami i tworzyć je przy użyciu `OrderService`. Klasa powinna zawierać:

- pole `private OrderService orderService;`
- pole `private Map<Long, Order> ordersPending;` (lista zamówień oczekujących),
- pole `private Map<Long, Order> ordersDone;` (lista zamówień wykonanych),
- konstruktor bezparametrowy, który zainicjalizuje listy i `OrderService`,
- metodę `public Order createOrder(List<Product> products, List<Extra> extras)`, która wywoła metodę tworzenia zamówienia z `OrderService` i doda utworzone zamówienie do listy oczekujących,
- metodę `public void orderDone(long orderID)`, która przeniesie zamówienie z listy oczekujących na listę wykonanych).

Podpowiedź: Zastosuj konstruktory jak w zadaniu 3/.

Zadanie 5/* programowanie obiektowe i TDD

Stworzymy model kalkulatora zdolności kredytowej.



Korzystając z TDD, stwórz test klasy CreaditCalculator.

1. Stwórz klasę `Person` z następującymi polami:

`id (String, PESEL),`

   - `name` (String, imię),
   - `familyName` (String, nazwisko),
   - `birthDate` (LocalDate, data urodzenia),
   - `income` (BigDecimal, miesięczny przychód).

2. Stwórz klasę `CalculatorResult` z następującymi polami:
   - `type` (enum `ResultType` zawierający opcje: DENIAL, LOW_RISK, HIGH_RISK, MEDIUM_RISK),
   - `amount` (BigDecimal, przyznana kwota kredytu),
   - `period` (Integer, maksymalny okres spłaty kredytu w latach).

3. Stwórz klasę `CreditHistoryService` z metodami:
   - `Integer findDebtor(String id)`
     odnajdywanie w pliku (po numerze PESEL) kwoty istniejącego zadłużenia danej osoby (w pliku powinny się znajdować tylko numery PESEL i przypisana im kwota):
     funkcja powinna zwracać kwotę w Optional (Optional<BigDecimal>), jeśli dana osoba ma już zadłużenie lub pusty Optional (Optional.empty()) w przypadku, gdy go nie ma.

Klasa nie powinna pozwolić na wyszukanie użytkownika z niepoprawnym PESELem (musi mieć 11 cyfr), powinna w takim wypadku rzucić odpowiedni wyjątek.

4. Stwórz klasę `CreditCalculator`, która wyliczy zdolność kredytową dla konkretnej osoby.

Klasa powinna zawierać pola obiektów `CalculatorResult`, `Person` i `CreditHistoryService` oraz następujące metody:

- `Integer checkDebt()`
  sprawdzanie zadłużenia danej osoby (poprzez odwołanie się do `CreditHistoryService`) funkcja zwraca kwotę lub 0 - gdy nie ma zadłużenia,
- `BigDecimal calculateInstallment()`
  wyliczenie miesięcznej raty kredytu na podstawie danych z `CalculatorResult`(kwoty i lat) `ze` wzoru:
  `amount/period/12 *1.2`,
  - gdy klient ma status DENIAL, powinien być rzucany odpowiedni wyjątek,
  - gdy pole `CalculatorResult` jest puste, funkcja najpierw wywołuje wyliczanie go (wyliczanie zdolności kredytowej)
- `CalculatorResult calculateCredit()`
  wyliczenie zdolności kredytowej, w tym:
  - wybranie `ResultType`:
    - DENIAL,
      - gdy income jest < 100,
      - gdy wiek przekracza 80 lat,
      - gdy wiek <18 lat,
      - gdy zadłużenie jest większe niż 0.4*income*12*15,
      - wtedy nie są wyliczane pozostałe pola CalculatorResult
    - HIGH_RISK,
      - gdy klient ma powyżej 75 lat,
      - gdy income < 500,
    - MEDIUM_RISK,
      - gdy klient ma powyżej 50 lat,
      - gdy income < 1500,
    - LOW_RISK w pozostałych przypadkach,
  - wyliczenie period:
    - domyślna wartość 30 lat, jeśli klient ma poniżej 50 lat,
    - wartość wyliczana ze wzoru 80-wiek_klienta, jeśli ma 50 lat lub więcej,
  - wyliczenie amount ze wzoru:
    `0.5*income*12*period`

5. * W testach stwórz mocka klasy `CreditHistoryService`.

6. * Klasa `CalculatorResult` powinna być immutable (lub trudno modyfikowalna).

Dla ułatwienia zamiast BigDecimal możesz użyć innego typu.

Podpowiedź: BigDecimal porównuje się przy pomocy metod, a nie operatorów.

Podpowiedź: Warto w pakiecie testowym utworzyć sobie klasę ze stałymi wieku, dochodów i loginów dla każdego scenariusza, żeby zmniejszyć ilość kodu w klasie testowej.

Podpowiedź: W klasie kalkulatora stwórz osobne metody na sprawdzanie warunków dla ResultType DENIAL, HIGH_RISK, MEDIUM_RISK i LOW_RISK.

Podpowiedź: Do odczytu pliku użyj:

```
Files.readAllLines(Paths.get(DEBTORS_DATA))
        .stream()
```

Rozwiązanie do zadań 1/ – 3/

1/

Program nie zadziała i się nie skompiluje, ponieważ zmienna myRect nie została zainicjalizowana, np. poprzez `new Rectangle()`.

2/

Należy użyć tzw. konstruktora teleskopowego, który w konstruktorze dla n parametrów przy pomocy `this()` wywoła konstruktor tej klasy dla liczby parametrów n-1, a nadmiarowe pole ustawi poprzez `this.x = x;`. Rozwiązanie przydatne w przypadku dużej liczby tego typu konstruktorów i pól. Poniżej kod:

```
class Book {
	private String author;
	private double price;
	private Genre genre;

	public Book(String author){
		this.author = author;
	}
	public Book(String author, double price){
		this(author);
		this.price = price;
	}
	public Book(String author, double price, Genre, genre){
		this(author, price);
		this.genre = genre;
	}
}
```

3/

Radzimy sobie podobnie jak w poprzednim zadaniu, tym razem jednak stosując `super()`, które wywoła konstruktor nadklasy. Rozwiązanie w kodzie:

```
public class Drink {
	Size size;
	public Drink(){}
```

```
        public Drink(int size){
                this.size = size;
        }
}
public class Tea extends Drink {
        int sugarSpoons;
        public Tea(){}
        public Tea(Size size, int sugarSpoons){
                super(size);
                this. sugarSpoons = sugarSpoons;
        }
}
public class IceTea extends Tea {
        int iceCubes;
        public Tea(Size size, int sugarSpoons, int iceCubes){
                super(size, sugarSpoons);
                this.iceCubes = iceCubes;
        }
}
```