

3. Klasy

Zadanie 3.1 Pracownik

Stwórz klasę **Pracownik**, której obiekt służy do zliczania godzin pracy jednego pracownika i wyliczania wypłaty zgodnie z modelem „timesheetowym” (płatność za godzinę zgodnie z ustaloną stawką), przy czym uwzględniana jest podwójna stawka za nadgodziny.

Jeśli na kursie pojawiły się już testy JUnit, to staraj się tworzyć tę klasę stopniowo dodając odpowiednie testy („Test-Driven Development”). Jeśli testów jeszcze nie znasz, to przetestuj w tradycyjny sposób (program z mainem), a testy JUnit dodasz w przyszłości.

Konstruktor **Pracownik(int stawka)** przyjmuje jedną wartość - stawkę godzinową pracownika.

```
Pracownik p1 = new Pracownik(100);
```

```
Pracownik p2 = new Pracownik(150);
```

Metody:

void praca(int godziny) – rejestruje jedną „dniówkę” pracy. Jeśli wartość podana jako parametr jest większa niż 8, to za godziny powyżej 8 naliczana jest podwójna stawka. Przepracowane godziny są w jakiś sposób zapamiętywane w obiekcie aż do momentu wypłaty.

int wypłata() – zwraca kwotę do wypłaty za przepracowane do tej pory godziny (jako iloczyn godzin i stawki godzinowej, z uwzględnieniem nadgodzin) i zeruje licznik - od tej pory godziny naliczane są od nowa. Jeśli nie ma aktualnie naliczonych godzin, zwraca zero.

Nadgodziny dotyczą tylko przekroczenia 8 godzin jednego dnia:

```
p1.praca(10);
```

```
int x = p1.wypłata(); // wynikiem jest 1200 : 8 godzin * 100 + 2 godziny * 200
```

```
p1.praca(5);
```

```
p1.praca(5);
```

```
int y = p1.wypłata(); // wynikiem jest 1000 : 10 godzin * 100
```

Zadanie 3.2 Rozszerzenie: pracownik z premią

Jako podklasę **Pracownika**, dodaj klasę **PracownikZPremia**, która dodatkowo posiada metodę **premia(int kwota)** – do normalnie zarobionych pieniędzy dodaje podaną kwotę premii. Wiele premii jest sumowanych. Po wypłacie naliczane premie są zerowane.

Zadanie 3.3 Klasa dla ułamków

Stwórz klasę, której obiekty reprezentują liczby wymierne w postaci ułamkowej. Klasa powinna implementować operacje na ułamkach.

Z kilku możliwych podejść najbardziej rekomenduję utworzenie klasy niemutowalnej, zgodnie z wzorcem „value object”: pola final, brak setterów, operacje matematyczne zawsze zwracają nowy obiekt w wyniku, a nie modyfikują bieżącego; metody porównujące biorą pod uwagę wartość, a nie tożsamość obiektu. Przykładami takich klas są BigDecimal oraz LocalDate.

Nie dopuszczaj do sytuacji, aby w mianowniku znalazło się zero – wyrzucaj wyjątek IllegalArgumentException. Zalecam taką „normalizację” tworzonych ułamków, aby mianownik był zawsze dodatni, a tylko licznik mógł być ujemny. Operacje arytmetyczne powinny także zwracać wynik w postaci maksymalnie skróconej.

Proponowane publiczne API klasy (tym razem piszę po angielsku):

- static Fraction of(long nom, long denom)
- static Fraction of(long number) // na podstawie liczby całkowitej, mianownik = 1
- static final Fraction ZERO, ONE, HALF, ... – kilka stałych, podobnie jak w BigInteger
- long getNominator(), long getDenominator()
- String toString()
- equals + hashCode – w oparciu o wartości licznika i mianownika,
imo $3/4$ i $6/8$ powinny być uznane za różne, podobnie jak jest to w klasie BigDecimal, gdy wartość jest równa, ale skala (precyzja) się różni...
- int compareTo(Fraction other) – klasa powinna implementować interfejs Comparable
 - Fraction shortened() – zwraca ułamek o tej samej wartości, skrócony w miarę możliwości (o największy wspólny dzielnik licznika i mianownika)
- Fraction reciprocal() – zwraca odwrotność ułamka
- Fraction neg() – zwraca liczbę ze zmienionym znakiem (plus/minus)
- Fraction add(Fraction) – suma
- Fraction sub(Fraction) – różnica
- Fraction mul(Fraction) – iloczyn
- Fraction div(Fraction) – iloraz
- double getAsDouble() – zwraca przybliżoną wartość tego ułamka jako double