

GOLD RUSH



Iteracja 1, tydzień 2: poszukiwacz złota na planszy

Jaki jest plan?

Czyścimy kod z poprzedniego tygodnia: drobna refaktoryzacja.

Zawsze można coś poprawić :) Dobrym zwyczajem jest zapisywanie pomysłów, które przychodzą nam do głowy w trakcie pisania kodu (TODO). Niezapisane pomysły z reguły odlatują w niepamięć.

Bierzemy na warsztat żetony.

Co mają wspólnego, a co je różni? Jak to obsłużyć w kodzie?

Zmuszamy w końcu pionek gracza do ruchu.

I tak dugo z tym zwlekaliśmy, szacunek za cierpliwość.

Musimy podjąć decyzję, gdzie zaimplementować ten ruch...

Planujemy zadania na przyszłą iterację.

Musimy realistycznie spojrzeć na tempo naszej pracy i zadania, które przed nami stoją.

Refaktoryzacja

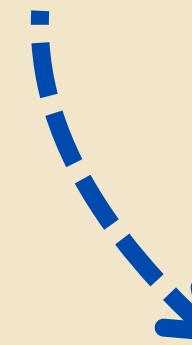
czyli co można poprawić?

Enkapsulacja pól z reguły jest lepsza niż brak enkapsulacji:

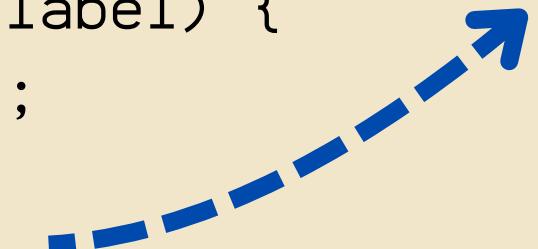
```
class Token {  
    public String label;
```



```
class Token {  
    private final String label;
```



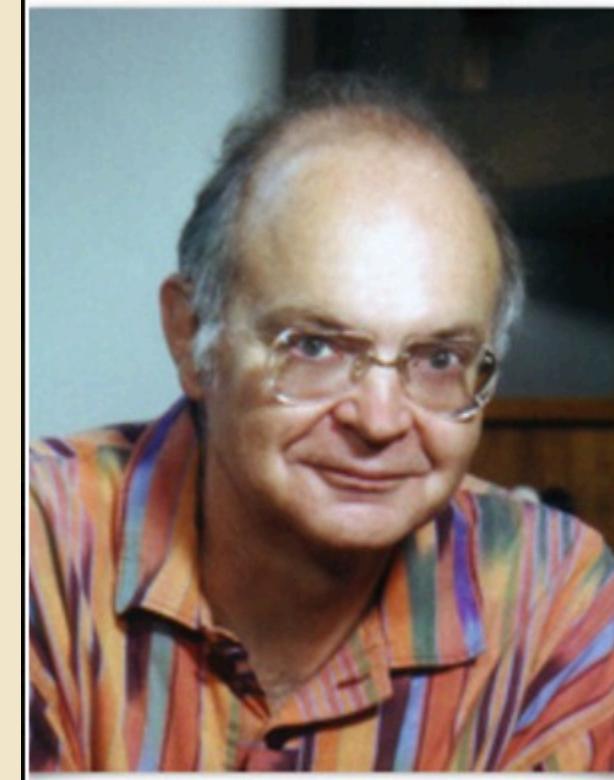
```
class Token {  
    public final String label;  
  
    public Token(String label) {  
        this.label = label;  
    }  
}
```



```
public Token(String label) {  
    this.label = label;  
}  
  
public String label() {  
    return label;  
}
```

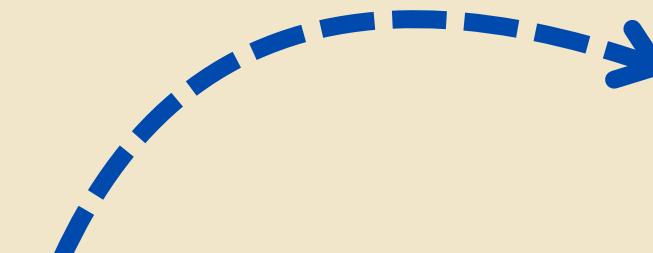


Na pewno nie powinniśmy **optymalizować** kodu pod względem wydajności **zbyt wcześnie**.



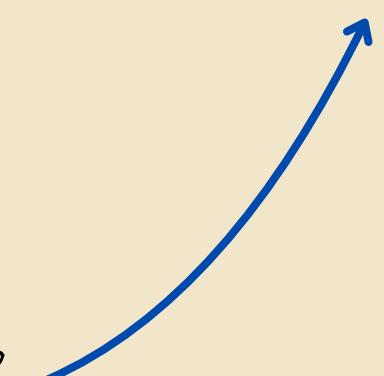
Nie oznacza to jednak,
że możemy pisać *niemądry* kod:

```
void clean() {  
    for (row...)  
        for (col...)  
            grid[col][row] = new Token("•")  
}
```



```
void clean() {  
    Token t = new Token("•")  
    for (row...)  
        for (col...)  
            grid[col][row] = t  
}
```

ile obiektów powstanie?



I tak w przyszłości lepiej będzie postawić jakąś fabryczkę żetonów.
(wagi muszej...)



We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.

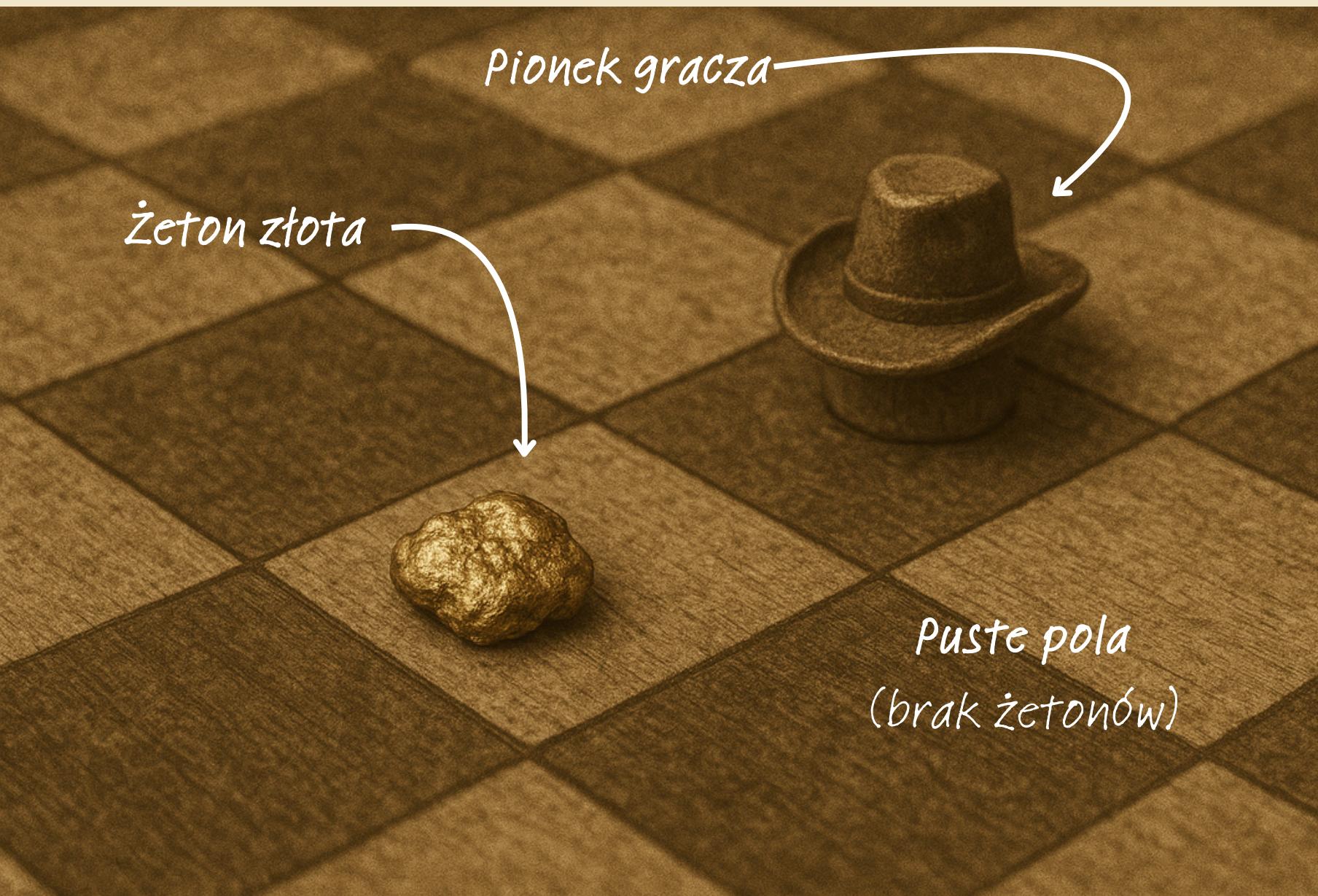
— Donald Knuth —

AZ QUOTES

Zagadka dla wnikliwych

Żetony

Ustaliliśmy, że pojęciem **żeton** (*token*) określamy wszystko, co może się znaleźć na polu planszy.

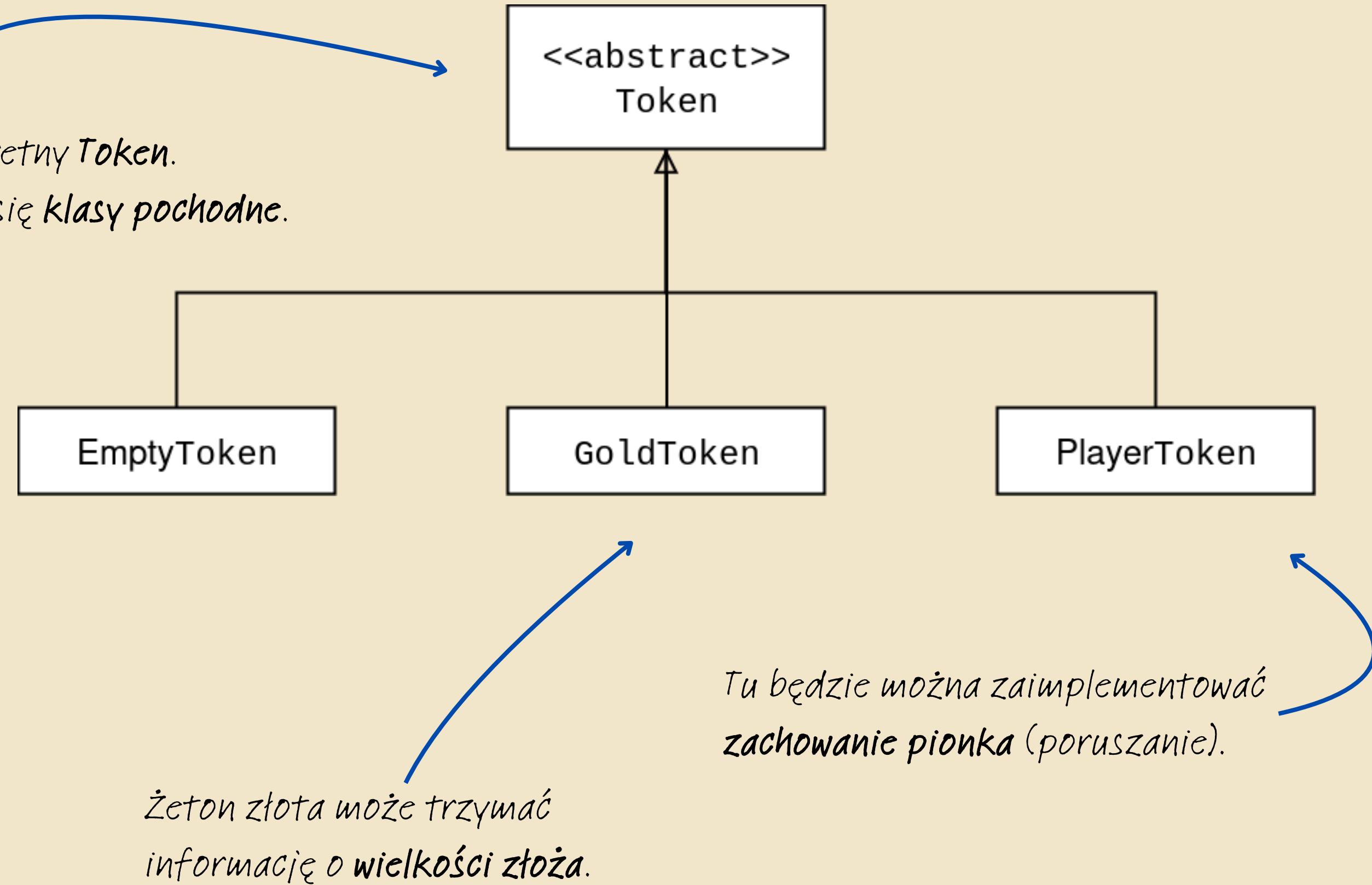


Jeżeli chcemy odwzorować w kodzie rzeczywiste zależności między żetonami,
możemy zastosować dziedziczenie.

Zakładamy rodzinę klas żetonów.

Ojcem założycielem jest mało konkretny *Token*.

Ukonkretnianiem żetonów zajmują się klasy pochodne.

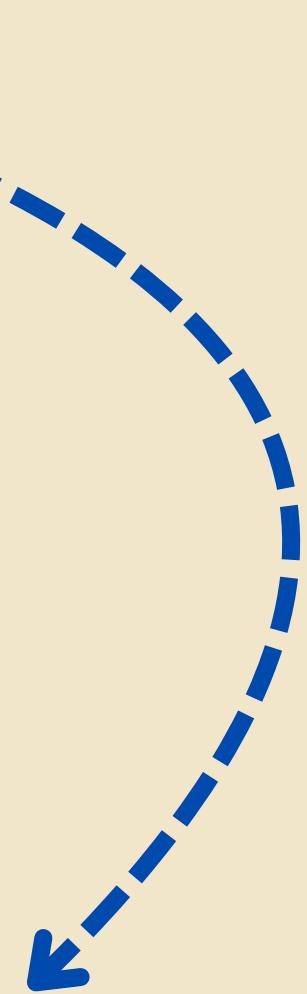


Do roboty!

Tworzymy klasy żetonów, np.:

```
public class GoldToken extends Token {  
    public GoldToken() {  
        super(" $" );  
    }  
}
```

```
public GoldToken() {  
    super(Label.GOLD_TOKEN_LABEL);  
}
```



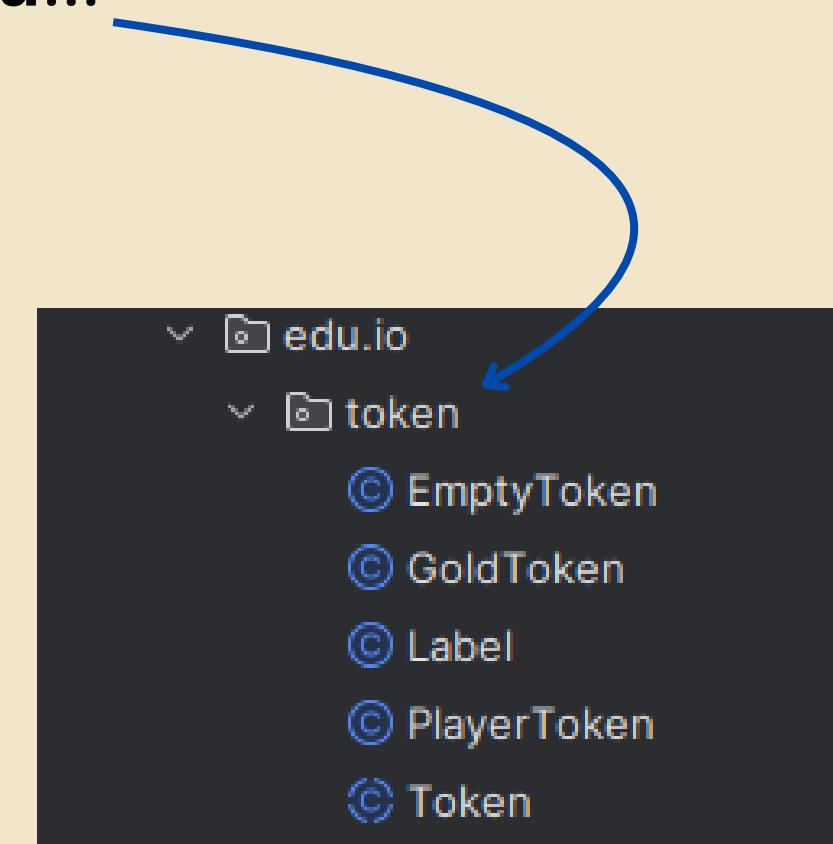
Dla ułatwienia możemy na tym etapie wprowadzić „stałe” z etykietami żetonów:

```
public class Label {  
    public static final String EMPTY_TOKEN_LABEL = " • ";  
    public static final String PLAYER_TOKEN_LABEL = " 웃";  
    public static final String GOLD_TOKEN_LABEL = " $" ;  
}
```

Dzięki temu konkretne znaki mamy w jednym miejscu. W przyszłości moglibyśmy np. wczytywać je z pliku konfiguracyjnego.

Dodaliśmy w sumie cztery klasy związane z żetonami. Robi się bałagan w plikach.

Czas na **wprowadzenie pakietu...**



...i przeniesienie do niego wszystkich klas powiązanych z żetonami.

Pionek gracza

Pionkiem gracza powinniśmy umieć **poruszać**.

Wstępny plan był taki, że zaimplementujemy to w klasie Board.

Główne dla tego, że poruszanie nie pasowało do ogólnego żetonu.
Teraz mamy wyspecjalizowany żeton: *PlayerToken*.

Zmieniamy zdanie: lepszym miejscem będzie klasa reprezentująca **pionek gracza**.

Pionek zna planszę, na której się znajduje...
...i wie, na jakim polu aktualnie stoi.
W momencie tworzenia wskazujemy planszę
oraz współrzędne początkowe.
Możemy zmieniać położenie pionka, wskazując
kierunek ruchu (góra, dół, lewo, prawo).
Możemy się dowiedzieć, gdzie pionek aktualnie jest.

PlayerToken
- board: Board
- col: int
- row: int
+ PlayerToken(board, col, row)
+ move(dir)
+ pos(): Coords

Dwa drobiazgi

W metodzie **move()** pojawił się parametr **dir** oznaczający kierunek ruchu.
Jakiego powinien być typu?

Z kolei metoda **pos()** zwraca typ **Coords**. Chodzi tu o „opakowanie” na współrzędne pola, czyli parę (*kolumna, wiersz*). Co zastosować w tym przypadku?



Kierunek ruchu

Do wskazywania kierunku przemieszczania pionka moglibyśmy zastosować „stałe”, np.:

```
public class PlayerToken {  
    public static final int MOVE_NONE = 0;  
    public static final int MOVE_LEFT = 1;  
    public static final int MOVE_RIGHT = 2;  
    ...  
  
    public void move(int dir) { ... }  
}
```

Jednak w tym przypadku lepiej sprawdzi się **enumeracja**:

```
public class PlayerToken {  
    public enum Move {  
        NONE,  
        LEFT,  
        RIGHT, ...  
    }  
  
    public void move(Move dir) { ... }  
}
```

Używamy zdefiniowanego na te potrzeby typu.

Na zewnątrz klasy PlayerToken posługiujemy się tym w taki sposób:
token.move(PlayerToken.Move.UP)

Współrzędne pionka

Z metody **pos()** chcemy zwracać współrzędne pola, czyli parę złożoną z kolumny i wiersza.

Te dwie wartości trzeba opakować. Można by do tego użyć klasy, ale trzeba by się sporo „napisać”, żeby wszystko właściwie obsłużyć.

Od Javy 16 mamy na szczęście drogę na skróty:

```
record Coords(int col, int row) {}
```

Definicję typu **Coords** umieścimy w klasie **Board** (bo dotyczy współrzędnych pól na planszy):

```
public class Board {  
    ...  
    public record Coords(int col, int row) {}  
    ...  
}
```

W gratisie dostajemy:

- konstruktor przyjmujący składniki
- new tworzy *niemutowalne* obiekty
- gettery do składników
- metodę `equals()`
- metodę `hashCode()`
- metodę `toString()`

Implementacja ruchu

Wyjaśniliśmy już szczegóły dotyczące typów. Czas przejść do działania.

W metodzie **move()** w zależności od wartości parametru **dir** trzeba odpowiednio zmodyfikować współrzędne **col** lub **row** pionka, np.:

```
token.move(PlayerToken.Move.LEFT)
```

Trzeba jeszcze pamiętać o „czyszczeniu” pola, które pionek opuszcza.

Zakładamy, że – opuszczając pole – pionek zostawia je puste, czyli z obiektem **EmptyToken**.

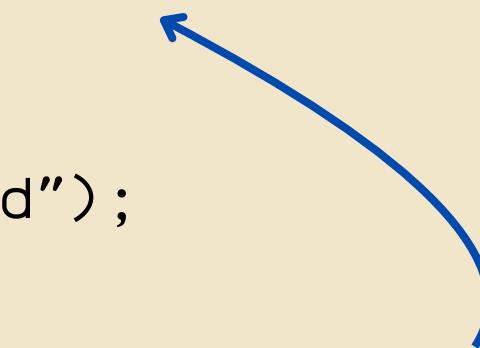
```
public void move(Move dir) {  
    ...  
    switch (dir) {  
        ...  
        case Move.LEFT: col -= 1; break;  
        ...  
    }  
    board.placeToken(col, row, this);  
}
```

Jeszcze jedno: trzeba zadbać sytuacje, w których **pionek chce opuścić planszę**. Musimy to obsłużyć w odpowiedni sposób.

Skorzystamy z **wyjątku**.

W metodzie **move()** musimy sprawdzić, czy po wykonaniu ruchu pionek nadal znajduje się na planszy. Jeżeli nie, ma zostać rzucony wyjątek **IllegalArgumentException**.

```
if (? col < row ?) {  
    throw new IllegalArgumentException("Cannot move outside the board");  
}
```



Czyli sygnalizujemy:
polecenie wykonania ruchu
w tym kierunku jest niepoprawne!

W miejscu wywołania metody **move()** trzeba będzie zastosować instrukcję **try-catch** do obsługi tego wyjątku.

Trzeba oczywiście jeszcze zadbać o to, by w przypadku przekroczenia granicy planszy **ruch nie został ostatecznie wykonany**!

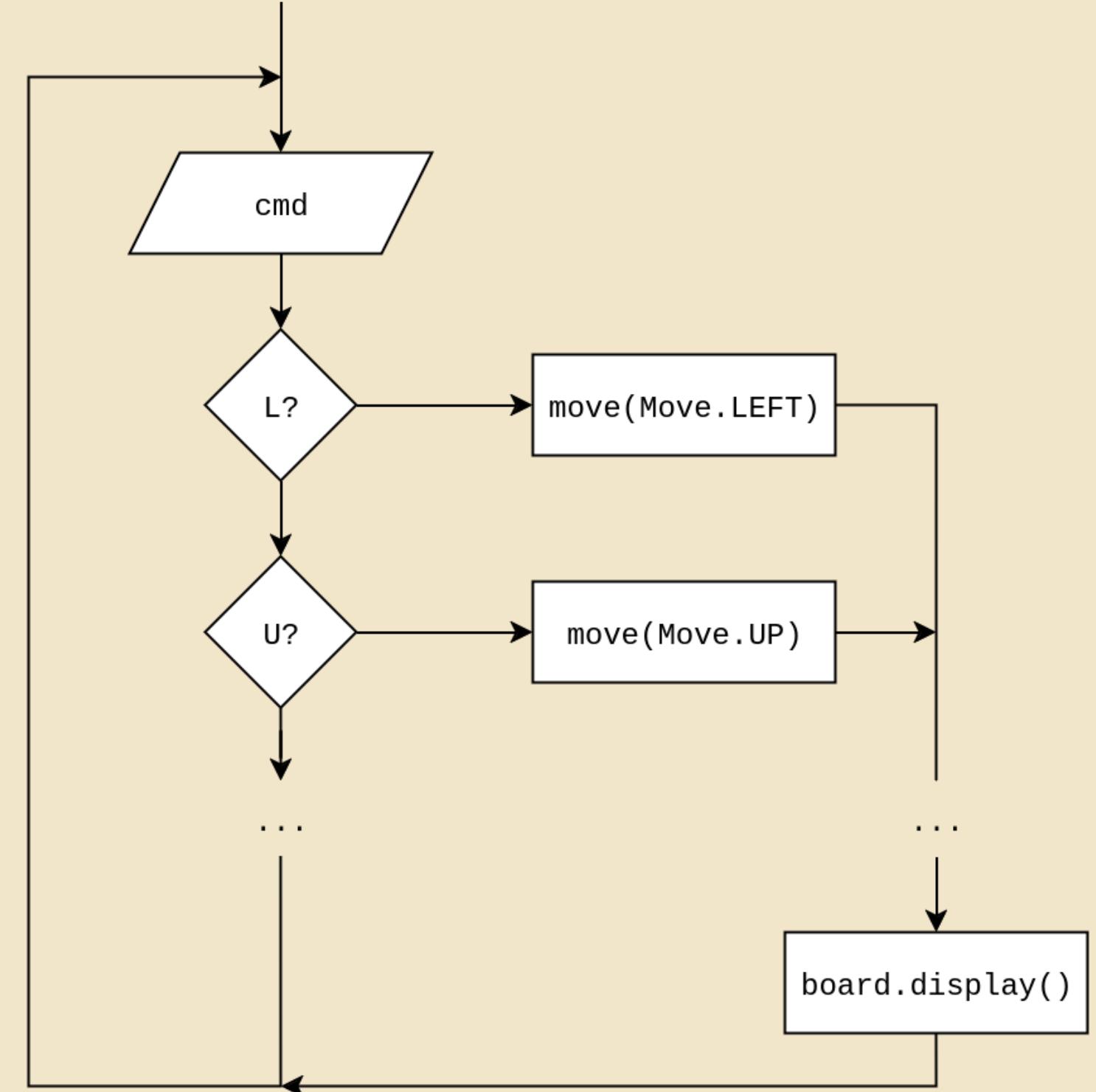
Akcja!

Pionkiem można już poruszać (miejmy nadzieję). Teraz wystarczy obsłużyć sterowanie nim przez użytkownika.

Ponieważ ma obowiązywać system turowy, potrzebujemy **pętli**, która będzie odpytywała użytkownika. Implementujemy to w **main()**.

Na razie nie przejmujemy się zakończeniem tej pętli – niech się kręci w kółko.

Na planszy możecie postawić żetony złota. Jeszcze nie da się ich zbierać – od tego zaczniemy następną iterację.



Plan kolejnej iteracji

- Mechanizm **zbierania złota**
- **Klasa gracza** przechowująca m.in. stan posiadania (ilość złota)
- **Klasa zarządzająca** mechanizmami w grze
- Zróżnicowanie ilości złota w żetonach
- Przemyślenie **interakcji z użytkownikiem**; próba oddzielenia UI od logiki.