

GOLD RUSH

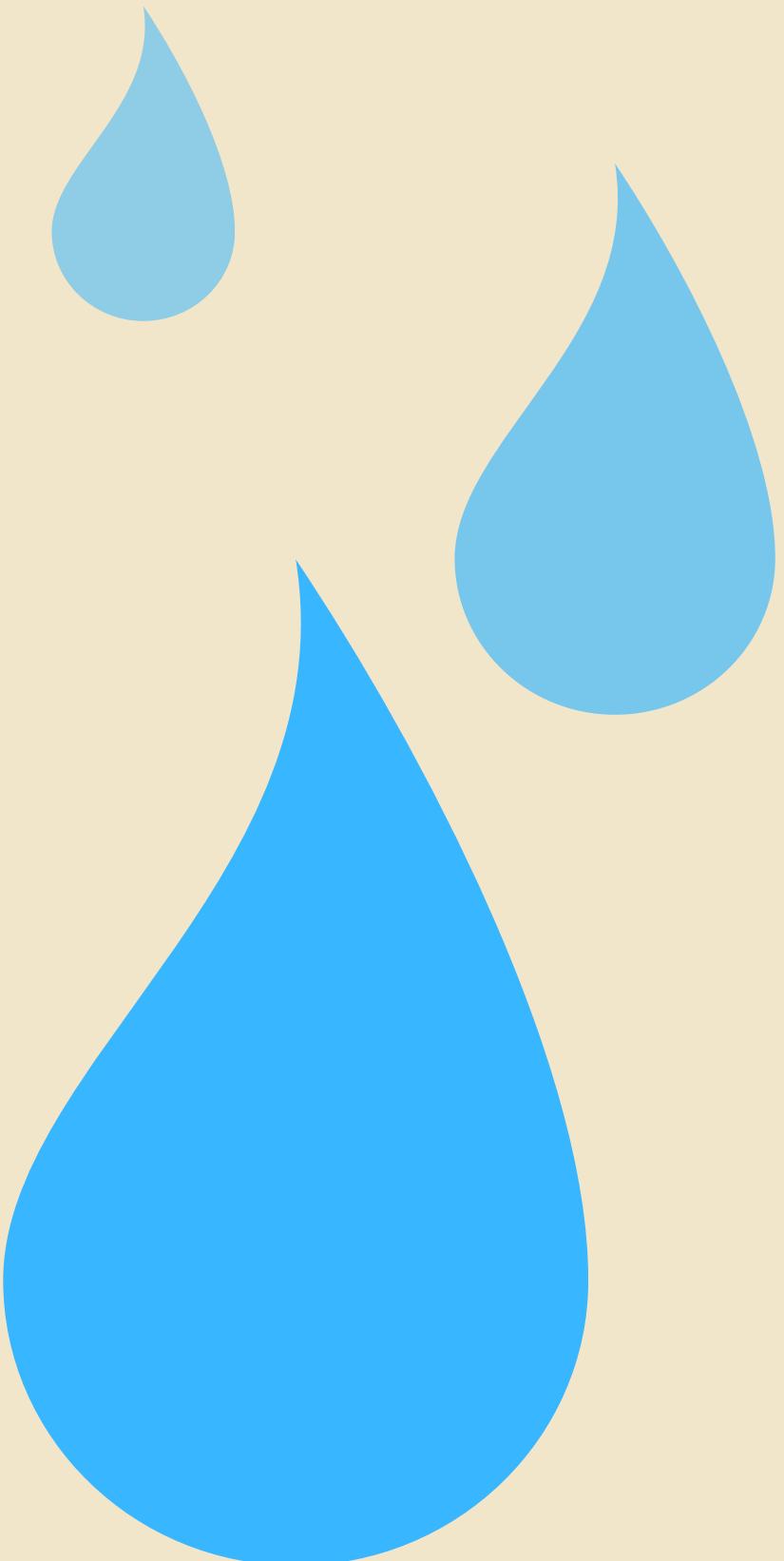


Iteracja 3, tydzień 1: nie ma wody na pustyni

Jaki jest plan?

Parametry życiowe gracza

Na początek wprowadzamy jeden parametr: nawodnienie.



Działasz – zużywasz wodę

Każde działanie gracza zmniejsza poziom nawodnienia.

Lepiej się nie odwodnić

Jeśli nawodnienie zejdzie do zera, gracz „umiera”

(symbolicznie – zostaje na planszy, ale nie może wykonać ruchu).

Ratunek dla spragnionych

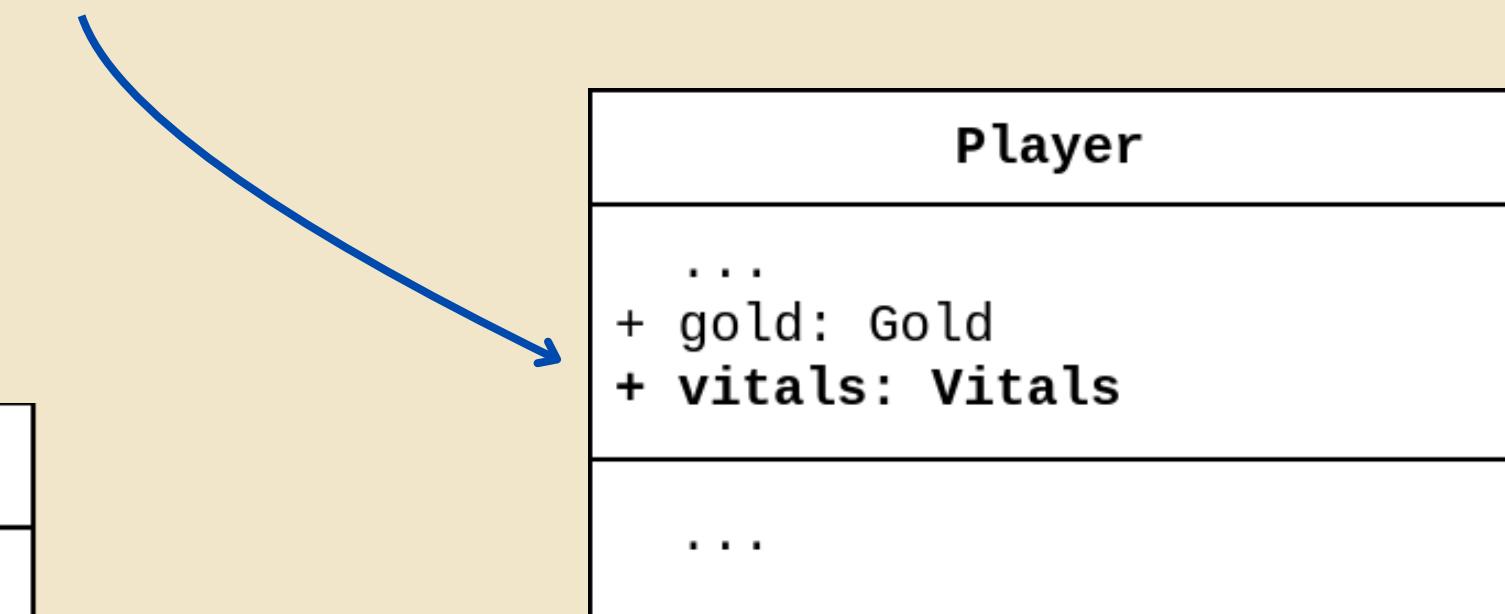
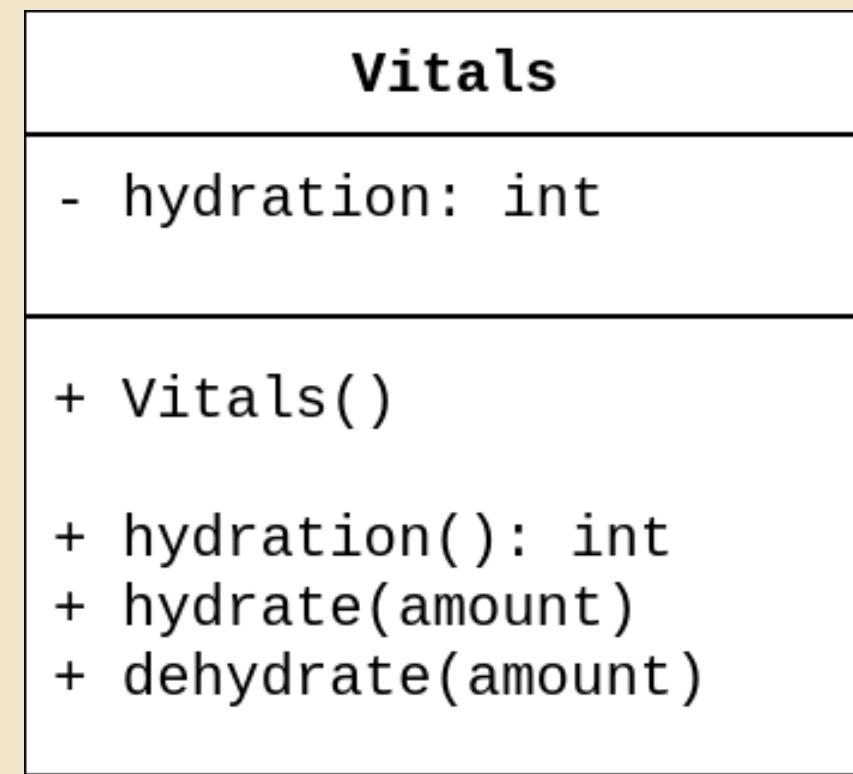
Na planszy może się pojawić żeton wody uzupełniający nawodnienie gracza.

Parametry życiowe

Analogicznie do „sakiewki na złoto” (czyli klasy **Gold**) wprowadzamy klasę reprezentującą parametry życiowe gracza. Obiekt tej klasy umieścimy w **Player**.

Pole *nawodnienia* (*hydration*)

wraz z getterem i dwiema metodami modyfikującymi stan:



Przyjmujemy procentową reprezentację nawodnienia: wartości całkowite 0..100.

Działania podejmowane przez gracza, takie jak chodzenie, zbieranie złota itp., powodują zużycie wody (**zmniejszenie nawodnienia**).

Wszystkie te operacje przechodzą przez metodę **Player.interactWithToken()**:

```
switch (token) {  
    case GoldToken...  
    case PickaxeToken...  
    case AnvilToken...  
    default...  
}  
  
...  
  
case AnvilToken... -> {  
    ...  
    vitals.dehydrate(VitalsValues.DEHYDRATION_MOVE);  
}
```

W niektórych gałęziach musimy zmniejszać nawodnienie.

Załóżmy, że różne działania wiążą się z różnym zużyciem wody.
Żeby uniknąć „magic numbers” wprowadźmy stałe:

```
public class VitalsValues {  
    public static final int DEHYDRATION_MOVE = 1;  
    public static final int DEHYDRATION_GOLD = 2;  
    public static final int DEHYDRATION_ANVIL = 3;  
}
```

Sępy już krażą...

Jeśli poziom wody zejdzie do zera, gracz symbolicznie umiera.

Przyjmijmy, że nie znika z planszy. Po prostu nie może wykonać żadnego ruchu. Może np. czekać na deszcz (prawdopodobne, że w grze pojawi się taki „ficzny”). Poza tym jego złoto może zainteresować innego gracza...

Opakowujemy sprawdzanie warunku
 $hydration > 0$



Vitals
...
+ isAlive() : boolean



Należałyby z tym **isAlive()** coś zrobić, żeby powiadamiać inne poziomy aplikacji o tym, że gracz jest „niedysponowany”.

Kluczowym miejscem jest **Player.interactWithToken()**.

Zastanówmy się, co zrobić w tej metodzie, jeśli gracz nie żyje?

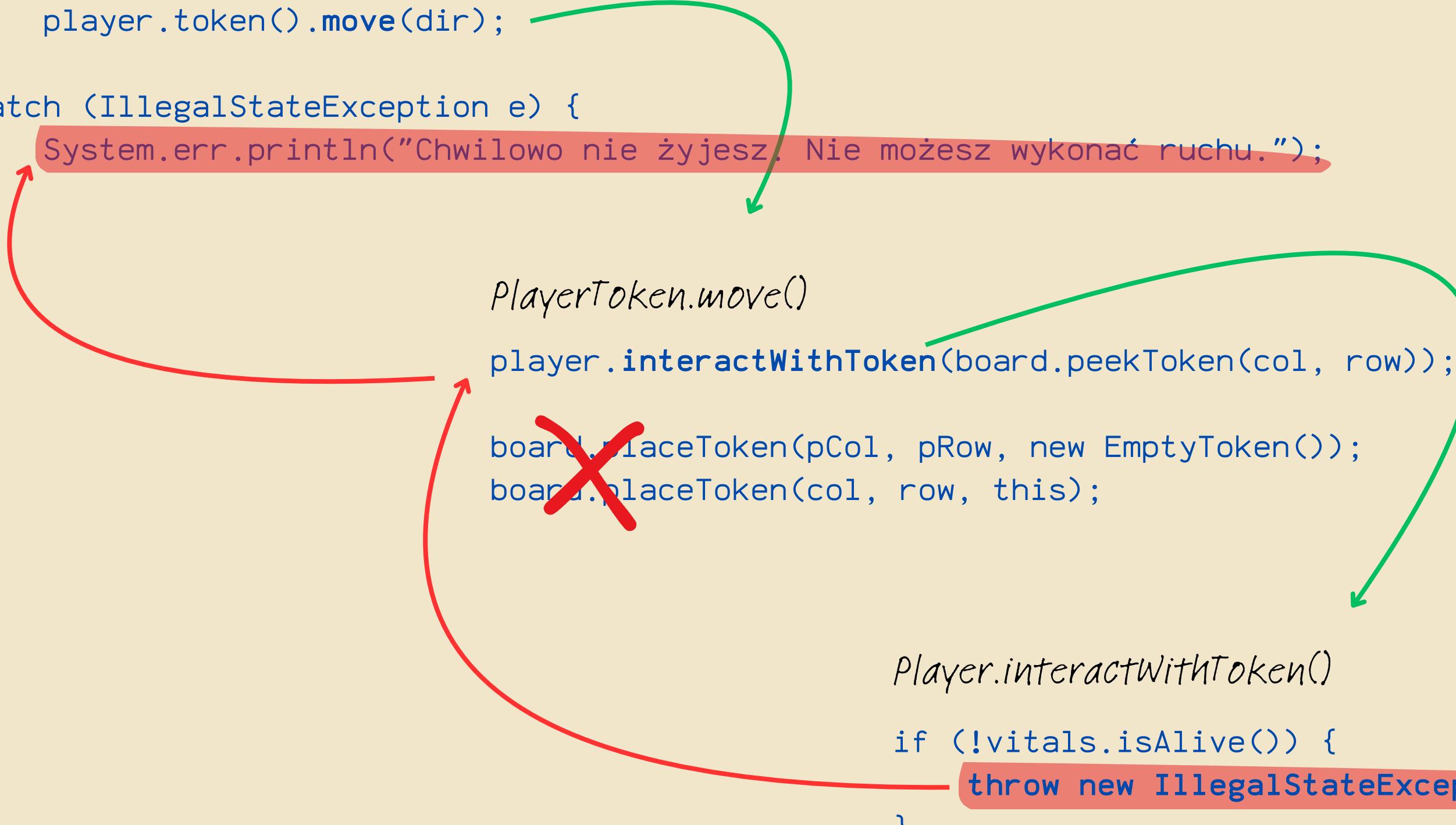
(1) Można na przykład **zwrócić wartość false** (przy założeniu, że *true* oznacza udaną interakcję). Można. Ale blokujemy sobie możliwość zwrócenia przez tę metodę żetonu, który ma się pojawić na opuszczanym przez gracza polu (np. porzucanie narzędzi, zastawianie pułapek itp.).

(2) Można **rzucić wyjątek**. Dzięki temu możemy obsłużyć tę sytuację na każdym „zainteresowanym” poziomie wywołań.

Przeanalizujmy rozwiązanie nr 2.

Przykładowa ilustracja działania wyjątków:

```
Game.start()  
  
try {  
    player.token().move(dir);  
}  
catch (IllegalStateException e) {  
    System.err.println("Chwilowo nie żyjesz. Nie możesz wykonać ruchu.");  
}  
  
PlayerToken.move()  
player.interactWithToken(board.peekToken(col, row));  
board.placeToken(pCol, pRow, new EmptyToken());  
board.placeToken(col, row, this);  
  
Player.interactWithToken()  
if (!vitals.isAlive()) {  
    throw new IllegalStateException("player is dead");  
}
```



Zastosowanie wyjątków ma swoje zalety (i wady).

Zastanówmy się nad jeszcze innymi możliwościami informowania o całkowitym odwodnieniu gracza:

- **funkcja zwrotna (callback)** – najprostszy sposób na „pasywne” powiadamianie,
- **zdarzenie (event)** – system bazujący na mechanizmie pub/sub, wzorcu projektowym obserwatora itp.

W omawianym przypadku nie ma co przesadzać: zdarzenia byłyby za duże (choć w przyszłości może odpowiednie). Sprawdźmy **callback**.

Zasada działania tego mechanizmu jest prosta:

Za pomocą tej metody przekazujemy "funkcję", która będzie wywoływana w chwili, gdy nawodnienie gracza osiągnie wartość zero.



Vitals
...
- onDeathCallback : Runnable
...
+ setOnDeathHandler(Runnable callback)

W metodzie `dehydrate()`, w chwili wykrycia zejścia do zera, uruchamiamy callback:

`onDeathCallback.run()`

Callback najwygodniej przekazać jako lambdę:

```
player.vitals.setOnDeathHandler(() -> {
    System.out.println("To koniec: pełne odwodnienie.");
});
```

I w tym momencie wraca do nas widmo **null**-a.

Co, jeśli nie ustawimy funkcji obsługującej tę sytuację?

W polu **onDeathCallback** będzie **null**, więc w chwili wywołania **onDeathCallback.run()** dostajemy NPE (czyli *NullPointerException*).

Prosta sprawa: inicjujemy wartość **onDeathCallback** na pustą lambdę (np. w konstruktorze klasy **Vitals**):

```
public Vitals() {  
    hydration = 100;  
    onDeathCallback = () -> {};  
}
```

Pozostaje jeszcze jedna „nieszczelność”: na razie metodzie **setOnDeathHandler()** możemy przekazać **null**:

```
player.vitals.setOnDeathHandler(null);
```

Najprostsze (wydawałoby się) rozwiązańe:

```
if (callback == null) {  
    throw new NullPointerException("callback cannot be null");  
}  
onDeathCallback = callback;
```

Dzięki **Objects.requireNonNull()** można to zrobić ładniej:

```
Objects.requireNonNull(callback, "callback cannot be null");  
onDeathCallback = callback;
```

A nawet jeszcze ładniej:

```
onDeathCallback = Objects.requireNonNull(callback, "callback cannot be null");
```



I na koniec wisienka na torcie:

```
public void setOnDeathHandler(@NotNull Runnable callback)
```

Dzięki tej adnotacji w kodzie pojawi się ostrzeżenie w sytuacji,
gdy do metody będziemy chcieli przekazać null.

```
player.vitals.setOnDeathHandler(null);
```

Passing 'null' argument to parameter annotated as @NotNull :

```
@edu.io.player.Vitals
```

```
public void setOnDeathHandler(
    @NotNull Runnable callback
)
```

```
gold_rush.main
```

Jeżeli chcemy jej użyć, musimy ją zaimportować:
`import org.jetbrains.annotations.NotNull;`

A żeby móc ją zaimportować, w `build.gradle.kts` musimy dodać zależność:

```
dependencies {
    implementation("org.jetbrains:annotations:26.0.2")
    ...
```

Zawsze powinniśmy dbać o wykrywanie null-a przekazywanego do metod. I to **jak najszybsze**.
Trzymajmy się zasad

FAIL FAST

Chodzi o to, żeby problemy wykrywać jak najszybciej i nie pozwolić im propagować.

```
Exception in thread "main" java.lang.NullPointerException Create breakpoint
  at edu.io.player.Vitals.dehydrate(Vitals.java:35)
  at edu.io.player.Player.interactWithToken(Player.java:48)
  at edu.io.token.PlayerToken.move(PlayerToken.java:51)
  at edu.io.Game.start(Game.java:40)
  at edu.io.Main.main(Main.java:12)
```

W obu przypadkach przyczyna jest ta sama:
przekazanie null-a do setOnDeathHandler().

VS

Gdzie łatwiej do tego dojść?

```
Exception in thread "main" java.lang.NullPointerException Create breakpoint : callback cannot be null
  at java.base/java.util.Objects.requireNonNull(Objects.java:246)
  at edu.io.player.Vitals.setOnDeathHandler(Vitals.java:51)
  at edu.io.Main.main(Main.java:9)
```

Żeton wody

Ostatnie, co zaplanowaliśmy, to żeton wody.

Wiadomo: jeżeli coś zniką, trzeba wprowadzić przeciwwagę: coś możemy zyskać.

WaterToken
- amount: int
+ WaterToken()
+ WaterToken(amount)
+ amount(): int

Pozostaje jeszcze uwzględnienie tego żetonu w **interactWithToken()**.
I tyle.