Nic nie równa się wizytówkom!

Często zdarza się, że musimy porównać zmienne przechowujące dwa obiekty i stwierdzić czy są równoważne, czy też różne od siebie. W przypadku zmiennych typów prymitywnych nie ma z tym problemu:

```
int a = 3;
int b = 3;

if (a == b) {
  System.out.println("Wartości takie same!");
} else {
  System.out.println("Wartości różne!");
}
```

Program oczywiście poinforumuje nas, że wartości są takie same. Czy jednak podobna operacja zadziała dla typów obiektowych? Przekonajmy się o tym:

1. Napisz program, który utworzy dwa obiekty identycznych wizytówek - w konstruktorach podaj dokładnie takie same dane.
2. Porównaj utworzone wizytówki operatorem `==`, tak jak w przykładzie powyżej. Jaki jest efekt?
3. Zapoznaj się z informacjami poniżej i zmodyfikuj odpowiednie klasy tak, by dało się stwierdzić, czy dwie wizytówki są tożsame (mają te same wartości atrybutów).
4. Po napisaniu własnych implementacji `equals` i `hashCode` spróbuj wygenerować "gotowce" przy pomocy IntelliJ (menu kontekstowe —> `Generate...`). Porównaj rozwiązanie zaproponowane przez Ciebie i wygenerowane przez IntelliJ. Poszukaj w internecie informacji na temat tego jaką relację algebraiczną musi spełniać dobrze napisana metoda `equals`. Podaj nazwę i 3 cechy takiej relacji. Odpowiedzi umieść jako komentarze w kodzie w odpowiednich miejscach.
5. Przerób klasy wizytówek tak by zumożliwić ich drukowanie bez metody `print()`, tak aby zadziałała instrukcja: `System.out.println(wizytowka)`.

Porównywanie obiektów

Tak jak wspominaliśmy na poprzednich zajęciach, zmienne typów złożonych przechowują jedynie **referencje** (wskazania) na obiekty zaalokowane na stercie. Oznacza to, że jeśli porównujemy dwie takie zmienne, tak naprawdę sprawdzamy jedynie czy wskazują one na ten sam obiekt.
Używając operatora `new` za każdym razem tworzymy zupełnie nowy obiekt, niezależnie od tego jakie wartości atrybutów będzie on zawierał. Oznacza to, że takie dwa obiekty

nigdy nie będą "sobie równe" w znaczeniu "nigdy nie będą posiadały tego samego adresu w pamięci".

```
Point p1 = new Point(1.0, 2.0);
Point p2 = new Point(1.0, 2.0);

Point p3 = p1; // Nie tworzymy nowego obiektu, jedynie nowy wskaźnik!

System.out.println(p1 == p2); // false, adresy są różne!
System.out.println(p1 == p3); // true, adresy wskazują na ten sam obiekt!
```

W jaki więc sposób porównywać wartości obiektów? Trzeba podejść do tego obiektowo i stworzyć odpowiednią metodę porównującą. Na szczęście nie musimy wymyślać całego mechanizmu sami - wszystkie klasy w Javie niejawnie dziedziczą po klasie `Object` zawierającą kilka podstawowych metod, które można nadpisać.

Jedną z nich jest metoda `equals`, która jako parametr przyjmuje obiekt takiego samego typu jak nasza klasa, a zwraca wartość `boolean`. Domyślnie `equals` porównuje jedynie adresy (a więc po prostu używa ==), ale możemy zmienić to zachowanie nadpisując tę metodę.

```
System.out.println(p1 == p2); // false, adresy są różne!
System.out.println(p1.equals(p2)); // true, jeśli zdefiniowaliśmy w klasie Point metodę e
```

Najczęściej w `equals` porównujemy po prostu wartości wszystkich atrybutów obu obiektów. W parze z `equals` powinniśmy również dostarczyć implementację metody `hashCode()`, używanej np. przez niektóre kolekcje (jest to szybka "funkcja skrótu", która powinna współpracować z `equals` na zasadzie: "jeśli `hashCode()` się różni to obiekty na pewno są różne, jeśli jest taki sam to mogą być równe, ale nie muszą - ostateczny wyrok wydaje wówczas `equals`).

Metoda toString()

Inną specjalną metodą, zdefiniowaną w klasie `Object`, którą możemy nadpisać w naszej klasie jest `toString()`. Jak sama nazwa wskazuje, zamienia ona nasz obiekt na jego reprezentację tekstową. `toString()` wołane jest automatycznie jeśli tylko zajdzie potrzeba konwersji naszego typu na `String` (jest to swego rodzaju niejawne rzutowanie). Właśnie dlatego możemy np. łączyć napisy z obiektami dowolnych typów operatorem `+`. Możemy

również wypisywać nasze obiekty podając je bezpośrednio jako argumenty `System.out.println()`.

Domyślnie `toString()` zwraca nazwę typu i identyfikator - nie jest to szczególnie przydatne i czytelne. Jeśli chcemy by nasze obiekty były wypisywane zgodnie z naszymi wymogami, wystarczy nadpisać `toString()` w swojej klasie.

W ramach ćwiczenia spróbuj zastąpić istniejącą metodę `print()` w klasie `BusinessCard` metodą `toString()`, której deklaracja powinna wyglądać następująco:

```
public String toString() {
    ...
}
```

Zauważ, że metoda ta **zwraca** tekstową reprezentację obiektu, a nie wypisuje go! Dlatego konieczna będzie lekka modyfikacja kodu pierwotnej metody `print()` tak by składała ona jeden duży napis reprezentujący wizytówkę (pamiętaj o znakach nowej lini, reprezentowanych jako: `"\n"`).

Jeśli wszystko się uda, poniższy kod powinien wypisać ładną wizytówkę:

```
BusinessCard bc = new BusinessCard("Piotr", "Budynek");
System.out.println(bc);
```