Typy opakowujące typów prostych

Czego się dowiesz

- Czym są typy opakowujące typów prostych,
- czym jest autoboxing i unboxing.

Wstęp

Czasami zdarzają się sytuacja, w których będziemy chcieli zastosować konwersję nie tylko pomiędzy typami zbliżonymi do siebie (np. double i int), ale również pomiędzy typem String i int, albo odwrotnie. Z tego co już wiemy konwersja taka nie jest możliwa w prosty sposób, jednak istnieje sposób na rozwiązanie tego problemu.

W Javie każdy typ prosty posiada swój typ opakowujący. To znaczy, że liczby, czy znaki będą wtedy reprezentowane za pomocą obiektów. Odpowiedniki klas dla typów prostych wymienione są poniżej:

- byte - Byte
- short - Short
- int - Integer
- float - Float
- double - Double
- char - Character
- boolean - Boolean

Jak widzisz najczęściej jest to kwestia zmiany litery z małej na wielką na początku nazwy. Wszystkie z powyższych klas, oprócz Character i Boolean, rozszerzają klasę Number, dzięki czemu możemy także korzystać z polimorfizmu, np. możemy stworzyć tablicę typu Number, która będzie przechowywała obiekty Integer i Double. W celu stworzenia obiektów odpowiednich typów stosujemy konstruktor lub statyczna metodę valueOf().

```
int num = 5;
Integer number1 = new Integer(num);
//alternatywnie
Integer number2 = Integer.valueOf(num);
```

Od Javy 9 używanie konstruktorów jest odradzane i zostały one oznaczone adnotacją @Depracated, co spowoduje, że w środowisku typu IntelliJ, czy eclipse jego wywołanie będzie oznaczone ostrzeżeniem.

Klasy opakowujące pozwalają na wygodne konwertowanie liczb na napisy za pomocą statycznej metody toString() np.:

```
int num = 5;
String number = Integer.toString(num);
```

Analogicznie możemy postępować z każdym innym typem.

Każda z klas opakowujących posiada dodatkowo metody to zamiany napisów na wartości odpowiednich typów, co może być przydatne np. przy wczytywaniu danych z użyciem klasy BufferedReader, która nie posiada metod do wczytywania danych odpowiednich typów, a potrafi czytać dane jedynie wiersz po wierszu jako tekst. Wspomniane metody to:

- **valueOf(String)**,
- **parseXXX(String)**, gdzie XXX to odpowiedni typ, np. Int lub Double.

Przykład:

```
String numberString = "5.5";
double num = Double.parseDouble(numberString);
double num2 = Double.valueOf(numberString);
```

Autoboxing i unboxing

Do typów obiektowych takich jak Integer, czy Double możemy przypisywać bezpośrednio wartości typów prostych bez potrzeby wywoływania konstruktora lub metody valueOf() jak pokazaliśmy wcześniej. W Javie funkcjonuje mechanizm **autoboxingu**, czyli automatycznego opakowania i **unboxingu**, czyli automatycznego rozpakowania wartości z typu obiektowego. Możliwy jest więc zapis:

```
Integer number = 5;
int otherNumber = number;
```

Pierwszą z powyższych operacji nazwiemy **autoboxingiem** (opakowywaniem) natomiast drugą **unboxingiem** (rozpakowywaniem) typu. W podobny sposób działa to dla wszystkich pozostałych typów prostych.

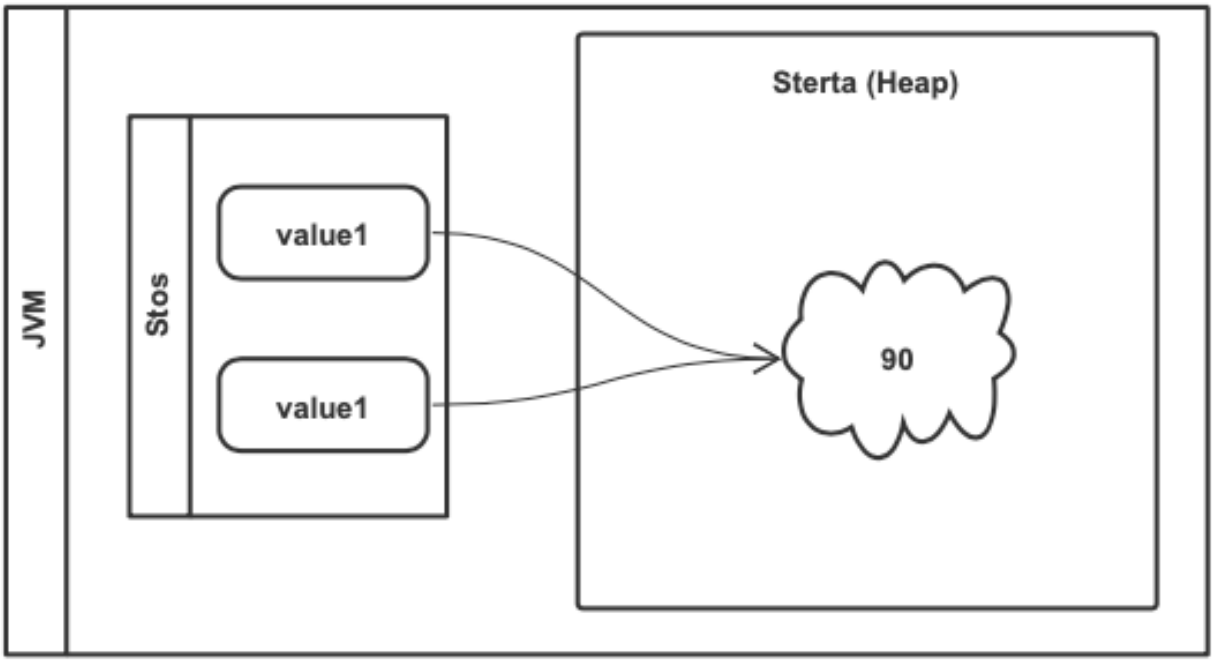
Cache dla liczb całkowitych

Typy opakowujące mają jedną poważną wadę w porównaniu do swoich odpowiedników w postaci typów prostych. Rozmiar w pamięci. W przypadku typów prostych wartości przechowywane są po prostu jako liczby, wartość true/false albo liczbo liczba odpowiadająca znakowi w tabeli Unicodu. Dla typów opakowujących powstają obiekty, więc oprócz samej przechowywanej wartości przechowywany jest nagłówek obiektu. Dla przykładu wartość typu int zajmuje w pamięci 4 bajty, a ta sama wartość opakowana w obiekt Integer będzie zajmowała już 16 bajtów.

Z tego względu wprowadzone zostały podstawowe optymalizacje dla typów całkowitoliczbowych polegające na cacheowaniu wartości z przedziału od -127 do 128. Polega to na tym, że jeżeli tworzymy obiekt `Integer` (lub innego typu całkowitoliczbowego) na podstawie wartości z podanego przedziału używając metody `valueOf()`, to nie jest tworzony nowy obiekt, a jedynie zwracana jest referencja na już istniejący obiekt. W przypadku wartości spoza tego zakresu, lub tworzeniu obiektów przy użyciu konstruktora a nie metody `valueOf()`, zawsze tworzony będzie nowy obiekt. M.in. z tego powodu konstruktory zostały oznaczone adnotacją `@Deprecated` i ich używanie od Javy 9 jest odradzane.

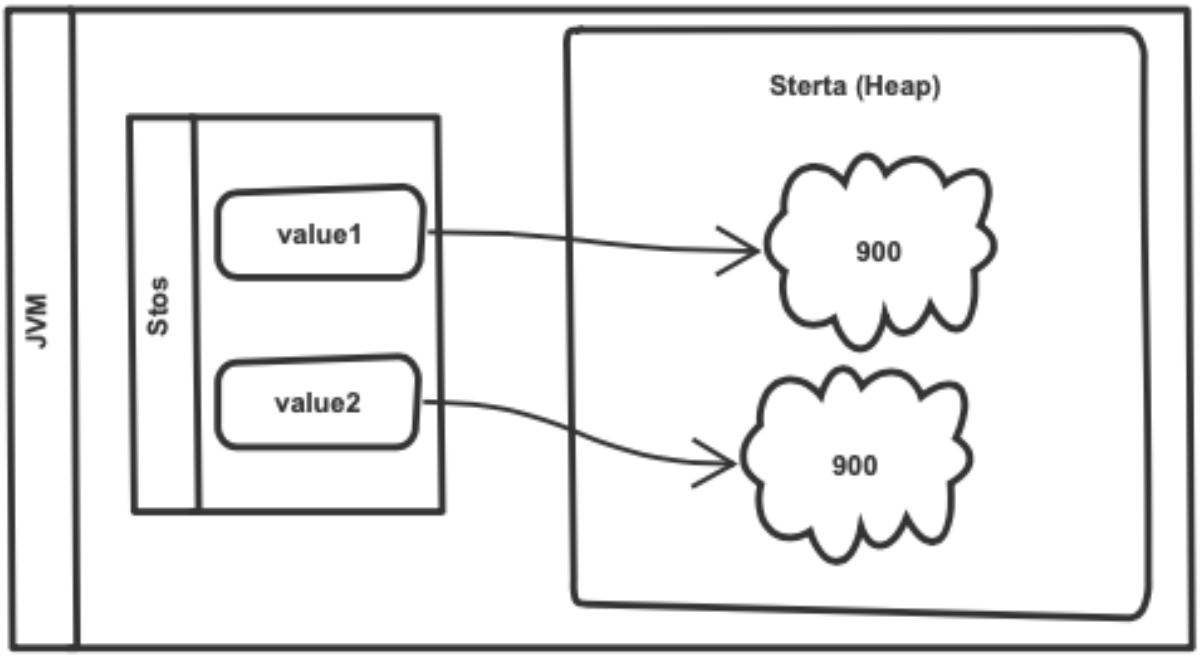
Przykład 1

```
Integer value1 = Integer.valueOf(90);
Integer value2 = Integer.valueOf(90);

//porównanie referencji, a nie wartości!
System.out.println(value1 == value2); //true
```



Przykład 2

```
Integer value1 = Integer.valueOf(900);
Integer value2 = Integer.valueOf(900);

//porównanie referencji, a nie wartości!
System.out.println(value1 == value2); //false
```



Górną granicę dla cache można zmienić ze 127 na dowolną liczbę. Należy w tym celu dodać przy uruchamianiu programu flagę -XX:AutoBoxCacheMax. Warto zapamiętać ten szczegół, jest to jedno z popularnych podchwytliwych pytań na rozmowach kwalifikacyjnych.

Po co używać klas opakowujących?

Wykorzystanie klas opakowujących typów prostych może wydawać się bezsensowne, szczególnie, gdy istnieje mechanizm autoboxingu i unboxingu. Niebawem się jednak przekonasz, że w niektórych sytuacjach niemożliwe jest przekazanie do konstruktora, czy metody wartości typu prostego. W takim przypadku musimy posiadać mechanizm, który pozwoli nam w wygodny sposób konwertować typy proste na obiekty. Oprócz tego typy opakowujące odblokowują dla nas jednak możliwości związane z polimorfizmem. Dzięki nim możemy tworzyć struktury i zmienne, do których może być np. przypisana albo liczba, albo znak. W przypadku samych typów prostych jest to niemożliwe ze względu na statyczną kontrolę typów w Javie.

Wielkie liczby

Czego się dowiesz

- Dlaczego liczby typu double są niedokładne
- Czemu typ double nie nadaje się do przechowywania danych dotyczących walut i do precyzyjnych obliczeń
- Jak działają klasy BigDecimal i BigInteger

Wstęp

Liczby (jak i wszystkie inne dane) w komputerze w rzeczywistości reprezentowane są za pomocą dwóch możliwych stanów, które możemy sobie wyobrazić jako wartości 0 lub 1, czyli tzw. systemu binarnego.

Jeżeli chcemy zapisać jakąś liczbę w systemie dziesiętnym powinniśmy zatem korzystać z kolejnych potęg liczby 2:

Liczba 8 to w systemie binarnym to 1000, co na system dziesiętny przeliczamy w taki sposób $1*2^3 + 0*2^2 + 0*2^1 + 0*2^0$.

W przypadku liczb całkowitych sprawa jest jak widać dosyć prosta, ponieważ poprzez sumę różnych potęg liczby 2 można uzyskać

dowolną wartość. Problem pojawia się w przypadku liczb zmiennoprzecinkowych, ponieważ o ile liczbę typu 0,5 można zapisać jako 2^{-1} (2 do potęgi -1), tak liczbę 0.7, czy 0.3 zaprezentować jest już bardzo ciężko, bo możemy to zrobić co najwyżej za pomocą skończenie dokładnego przybliżenia.

Jeżeli więc będziemy próbowali odjąć od siebie liczby, które reprezentowane są za pomocą skończonych przybliżeń, to możemy otrzymać nieoczekiwany wynik.

BigNumbers.java
```
public class BigNumbers {
	public static void main(String[] args) {
		double a = 0.7;
		double b = 0.3;
		System.out.println(a - b);
	}
}
```

Widzimy tutaj banalny wynik odejmowania dwóch liczb typu double, jednak w wyniku nie otrzymamy tak jak moglibyśmy się spodziewać 0.4, ale 0.3999999999999997.

Drugim problemem jest to, że zmienne typu int, czy nawet double i long mogą przechowywać pewne ograniczone wartości, ale istnieją nieliczne przypadki, gdy chcielibyśmy móc obliczyć wynik działań na liczbach, które mają 30 zer znaczących.

Klasy BigInteger i BigDecimal

Klasy BigInteger i BigDecimal pozwalają w Javie rozwiązać dwa powyższe problemy:

- Klasa BigDecimal służy do reprezentowania dokładnych liczb zmiennoprzecinkowych
- Klasa BigInteger pozwala przechowywać informacje o ogromnych liczbach całkowitoliczbowych

Obie klasy są typami obiektowymi, nie możemy więc na nich wykonywać bezpośrednio obliczeń za pomocą prostych operatorów (+, -, *, /), nie

działa w ich przypadku mechanizm autoboxingu i unboxingu - zawsze musimy posługiwać się metodami.

Obiekty powyższych klas można stworzyć albo na podstawie liczby reprezentowanej jako String, albo na podstawie istniejących wartości liczbowych typów prostych.

```
// utworzenie obiektu na podstawie wartości typu String
BigInteger big1 = new BigInteger("109876543210987654321");
// utworzenie obiektu za pomocą statycznej metody valueOf(), na
// podstawie wartości typu prostego
BigDecimal big2 = BigDecimal.valueOf(0.7);
```

W celu wykonania obliczeń na liczbach, należy posługiwać się metodami:

- add() - dodawanie
- subtract() - odejmowanie
- multiply() - mnożenie
- divide() - dzielenie

każda z nich przyjmuje wartość typu BigInteger lub BigDecimal.

Dla podsumowania spójrz na przykład w różnicy działania na liczbach typu prostego oraz obiektach typu precyzyjnego:

BigNumbers.java

```
import java.math.BigDecimal;

public class BigNumbers {
	public static void main(String[] args) {
		double a = 0.7;
		double b = 0.3;
		System.out.println("a - b = " +  (a-b));

		BigDecimal aBig = BigDecimal.valueOf(a);
		BigDecimal bBig = BigDecimal.valueOf(b);
		System.out.println("aBig - bBig = " + aBig.subtract(bBig));
	}
}
```

Zwróć jednak uwagę, że każda z metod służąca do obliczeń na typach BigInteger, czy BigDecimal nie modyfikuje oryginalnej wartości obiektu - podobnie jak obiekty klasy String, są to obiekty niemodyfikowalne.

W celu zmiany wartości liczby należy więc posługiwać się zapisem w takim stylu:

```
//tworzymy dwa obiekty typu BigDecimal
BigDecimal aBig = BigDecimal.valueOf(a);
BigDecimal bBig = BigDecimal.valueOf(b);
//do obiektu aBig przypisujemy wynik odejmowania aBig-bBig
aBig = aBig.subtract(bBig);
```

Kiedy stosować klasy BigDecimal/BigInteger

Używaj ich przede wszystkim wtedy, kiedy jest to niezbędne, czyli np. w przypadku obliczeń finansowych, gdzie precyzja jest niezbędna. Pamiętaj jednak, że pomimo tego, że są to klasy o dużych możliwościach to są to obiekty i każde działanie na nich sprawia, że tworzenie nowych obiektów, czy obliczenia są dużo bardziej kosztowne obliczeniowo, a tym samym mogą odbijać się na wydajności Twojej aplikacji.

W większości zastosowań zdecydowanie powinny CI wystarczyć liczby typów podstawowych.