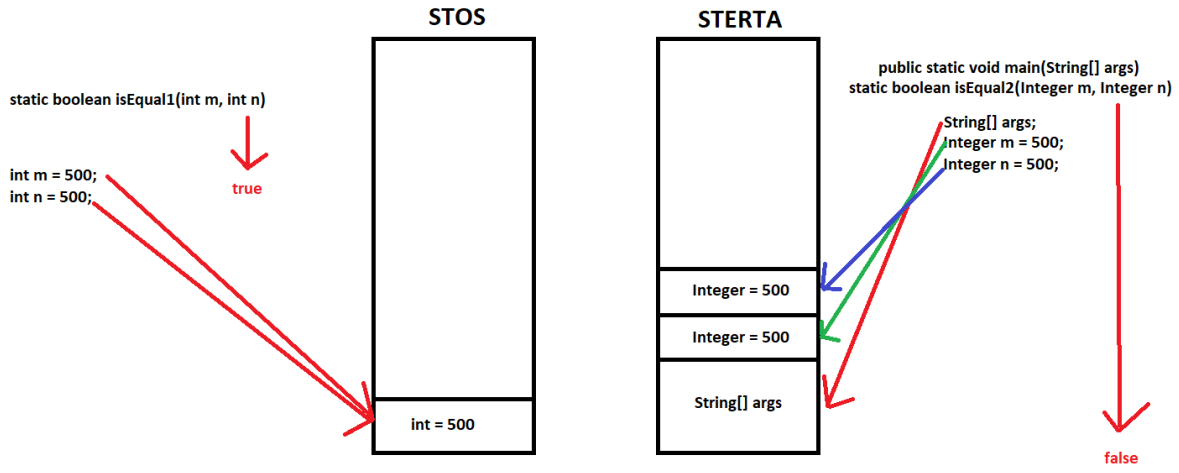
//Jakub Bednarek

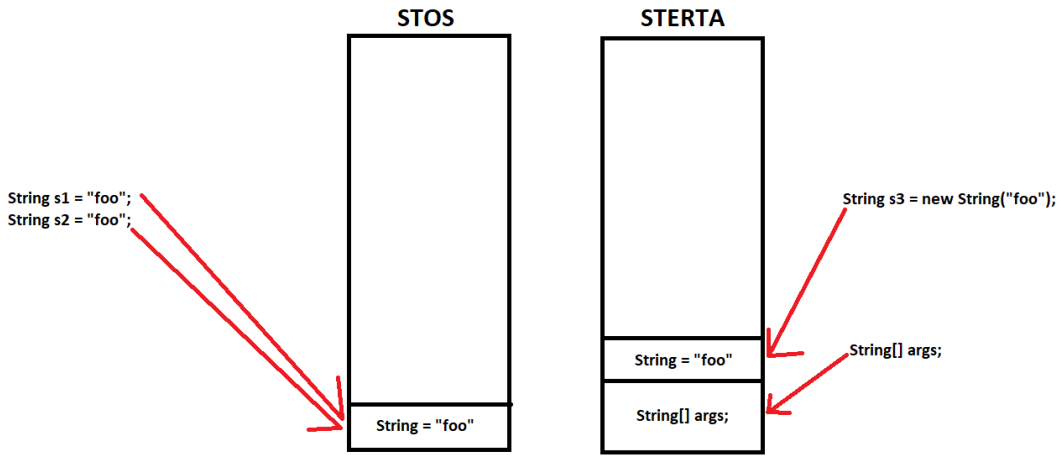
Zadanie 3



Funkcja isEqual1(int m, int n) przyjmuje 2 argumenty typu int. Przy wywołaniu funkcji dla takich samych argumentów tj. 500 JVM utworzy komórkę w pamięci której wartość będzie równa 500, a następnie przypisze do nich „referencje”, tak aby nie tworzyć niepotrzebnie nowej komórki pamięci z tą samą wartością. Oznacza to że wywołanie funkcji isEqual1(500, 500) zwróci true, ponieważ operator „==” zwraca równość tożsamości, a z powyższego rysunku wynika, że zmienne m oraz n będą odnosiły się do tej samej komórki pamięci.

Natomiast funkcja isEqual2(Integer m, Integer n) przyjmuje 2 argumenty typu Integer. Są to „opakowane” typy in. Przy wywołaniu konstruktora Integer zostanie zaalokowana nowa komórka pamięci (na stercie!) do której zostanie wpisana jej wartość. Oznacza to że użycie operatora „==” w funkcji isEqual2(Integer m, Integer n) zwróci false, ponieważ zmienne m oraz n będą wskazywały na inne komórki pamięci.

Zadanie 4



Mamy dane dwie łańcuchy o wartościach „foo”. Java dla optymalizacji utworzy tzw. String pool, w którym zostanie zaalokowana wartość „foo” dla nowej komórki pamięci tak aby zmienne s1 oraz s2 przechowywały referencje do tej komórki. Natomiast dla zmiennej s3 przy wywoałniu operatora „new” zostanie specjalnie zaalokowana nowa komórka pamięci mimo tego, że wartość „foo” jest taka sama dla zmiennych s1 oraz s2. Warto też zauważyć, że operator new zaalokuje pamięć na stercie, a nie na stosie jak było to w przypadku s1 i s2.

Stąd wywołanie kolejno funkcji:

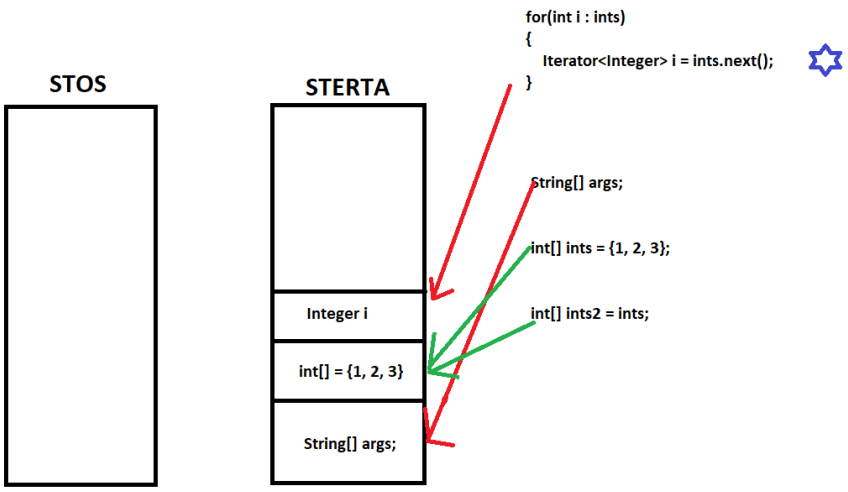
System.out.println(s1 == s2) zwróci true. -> sprawdzamy równość tożsamości.

System.out.println(s1.equals(s2)) zwróci true. -> sprawdzamy równość strukturalną.

System.out.println(s1 == s3) zwróci false -> adres pamięci s1 nie jest równy s3.

System.out.println(s1.equals(s3)) zwróci true -> zawartość zmiennych jest taka sama.

Zadanie 5.



W Javie tablice zawsze są alokowane na stercie, oznacza to że tablica ints = {1, 2, 3} wyląduje w tym miejscu. Przy wywołaniu „range based for loop” w Javie wykorzystujemy „lukier syntaktyczny”, ponieważ ten rodzaj pętli sprowadza się do takiego kodu:

```
For(Iterator<Integer> it = ints.iterator(); it.hasNext(); )
{
        Integer i = it.next();
        System.out.printl(i);
}
```

Oznacza to, że przy każdym przejściu pętli otrzymujemy opakowany obiekt typu Integer utworzony na podstawie typu int. Sprowadza się to do tego, że fragment kodu „i = 0;” nie będzie miał wpływu na wartości elementów, ponieważ wartość ta zostanie przypisana do obiektu stworzonego w pętli w innym miejscu pamięci. Uproszczenie na rysunku zostało zaznaczone gwiazdką.

Jeżeli wykonamy operację int[] ints2 = ints; to do tablicy ints2 zostanie przypisany adres pamięci tablicy ints co oznacza, że w przypadku modyfikacji elementu w tablicy ints modyfikacja także wystąpi w tablicy ints2, taka sama sytuacja wystąpi jeśli zmodyfikujemy element w tablicy ints2 -> wartość elementu w tablicy ints także zostanie zmieniona. Stąd w przypadku przedostatniej pętli for, gdzie przechodzimy po całej tablicy ints2 zmieniamy wartość elementu na -1 wszystkie wartości w tablicy ints oraz ints2 będą miały wartość -1.