Zagadnienia Programowania Obiektowego

Ownership and Immutability in Generic Java

Yoav Zibin
Alex Potanin
Paley Li
Mahmood Ali
Michael D. Ernst

Krótkie przypomnienie - final

- Final class – nie może być rozszerzona.
- Final method – nie może być przedefiniowana przez podklasy.
- Final variable – może być tylko raz przypisana, ale nie gwarantuje to niezmienności (immutability) obiektu. W szczególności jeżeli jest to referencja, to nie może wskazywać na inny obiekt.
- Blank final – można przypisać tylko raz, ale niekoniecznie w miejscu deklaracji.

Krótkie przypomnienie – const w C++

- Cała seria na ten temat: „const-correctness”.
- Zmienne typu const.
- Metody typu const.
- Const przed gwiazdką i po niej (tak samo dla &).
- Z grubsza gwarantuje niezmienność obiektu, ale można to obejść za pomocą mutable oraz const_cast.

Wstęp

- Java nie dysponuje notacją, która mówiłaby, czy obiekt jest czyjąś własnością (ownership) lub czy jest niezmienny (mutable, immutable, read-only).
- Ownership and Immutability in Generic Java (OIGJ) to rozszerzenie języka, które jest wstecznie zgodne i zapewnia wyżej wymienione cechy.
- OIGJ jest czysto statyczne i może być wyliczone na etapie kompilacji, nie powodując narzutu podczas wykonania.
- Nadaje się do innych języków obiektowych, takich jak C++ i C#.

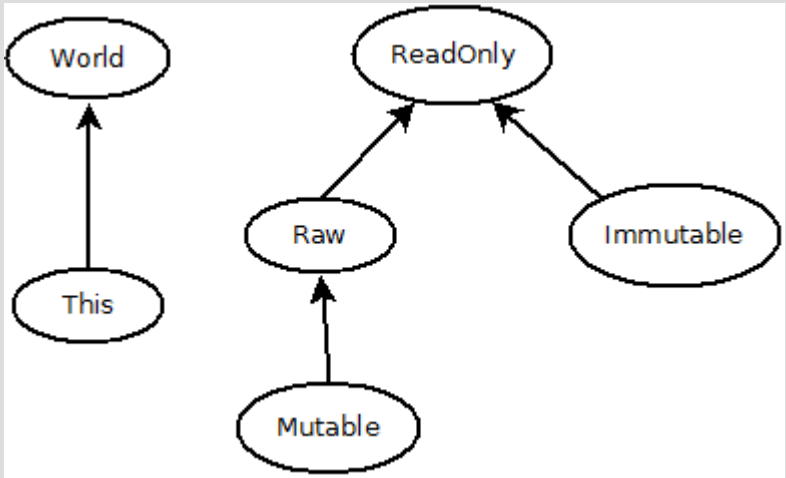
Cele autorów

- Prostota mechanizmów
- Brak konieczności refaktoryzacji istniejącego kodu
- Elastyczność
- Formalizacja

OIGJ korzysta z zasady „owner-as-dominator”, która mówi o tym, że referencja do obiektu nie może wyciec poza jego właściciela (private to za mało). Na przykład lista powinna posiadać na własność swoje obiekty Entry (ale już nie elementy tych Entry).

Składnia

- Dwa dodatkowe typy:
  - owner parameter
  - immutability parameter
- Korzeniem drzewa dziedziczenia jest Object<O, I>, gdzie O opisuje właściciela (owner) a I typ (immutability)



Conditional Java

- Programista może napisać strażników dla metod. Mają oni postać <X extends Y>? deklaracja metody.
- Po pierwsze oznacza to, że metodę można wykonać tylko wtedy, gdy typ argumentu jest rozszerzeniem typu Y.
- Po drugie, w ciele metody X jest traktowane jako Y.

```
1:class Foo<O extends World,I extends ReadOnly>
2:      Date<O,Immut> imD = new Date<O,Immut>();
3:      Date<O,Mutable> mutD = new Date<O,Mutable>();
4:      Date<O,ReadOnly> roD = ... ? imD : mutD;
5:      Date<O,I> sameD;
6:      Date<This,I> ownedD;
7:      Date<World,I> publicD;
8:      <I extends ReadOnly>? int readonlyMethod() {...}
9:      <I extends Mutable>? void mutatingMethod() {...}
10:     <I extends Raw>? Foo(Date<O,I> d) {
11:             this.sameD = d;
12:             this.ownedD = new Date<This,I>();
13:             // Illegal, because sameD came from the outside.
                this.sameD.setTime(...);
14:             // OK, because Raw is transitive for owned fields.
this.ownedD.setTime(...);
15:
```

```
1:class Entry<O,I,E> {
2:      E element;
3:      Entry<O,I,E> next, prev;
4: }
5: class LinkedList<O,I,E> {
6:      Entry<This,I,E> header;
7:      <I extends Raw>? LinkedList() {
8:          this.header = new Entry<This,I,E>();
9:          header.next = header.prev = header;
10:     }
11:     <I extends Raw>? LinkedList(
12:             Collection<?,ReadOnly,E> c) {
13:         this(); this.addAll(c);
14:     }

25:     int size() {...}
```

```
15:     <I extends Raw>? void addAll(
16:                     Collection<?,ReadOnly,E> c) {
17:         Entry<This,I,E> succ = this.header,
18:         pred = succ.prev;
19:         for (E e : c) {
20:             Entry<This,I,E> en=new Entry<This,I,E>();
21:             en.element=e; en.next=succ; en.prev=pred;
22:             pred.next = en; pred = en; }
23:         succ.prev = pred;
24:     }
27:     <ItrI extends ReadOnly> Iterator<O,ItrI,I,E>
28:     iterator() {
29:         return this.new ListItr<ItrI>();
30:     }
31:     void remove(Entry<This,Mutable,E> e) {
32:         e.prev.next = e.next;
33:         e.next.prev = e.prev;
34:     }
```

```
35: class ListItr<ItrI> implements
36:     Iterator<O,ItrI,I,E> {
37:         Entry<This,I,E> current;
38:         <ItrI extends Raw>? ListItr() {
39:             this.current = LinkedList.this.header;
40:         }
41:         <ItrI extends Mutable>? E next() {
42:             this.current = this.current.next;
43:             return this.current.element;
44:         }
45:         <I extends Mutable>? void remove() {
46:             LinkedList.this.remove(this.current);
47:         }
48: } }
49: interface Iterator<O,ItrI,CollectionI,E> {
50:     boolean hasNext();
51:     <ItrI extends Mutable>? E next();
52:     <CollectionI extends Mutable>? void remove();
53: }
```

Kolekcje i ich iteratory

- Iterator ma „pod spodem” kolekcję, którą obsługuje ale jego niezmienność jest czymś innym niż niezmienność jego kolekcji
- Mutable iterator i mutable collection (next() i remove())
- Mutable iterator i read only collection (next())
- Read only iterator i mutable collection (remove())
- Ale właściciel iteratora i kolekcji musi być ten sam!

Zagnieżdżanie właścicieli

List<This,I,Date<World,I>> l1; // Legal nesting
List<World,I,Date<This,I>> l2; // Illegal!

Parametry opisujące właścicieli muszą być poprawnie zagnieżdżone, to jest pierwszy z nich musi być nie wyżej niż pozostałe parametry w drzewie właścicieli.

OIGJ reguły

- Dostęp do pola **o.f** jest legalny wtw
  Owner(f) = This → o = this
- Przypisanie do **o.f** jest legalne wtw
  (i) Immutability(o) <= Raw
  (ii) Immutability(o) = Raw → (o = this or Owner(o) = This)
  (iii) dostęp do tego pola jest legalny
- Wołanie metody T0 m(T1, .. , Tn) jako **o.m** jest legalne wtw
  (i) Owner(Ti) = This → o = this
  (ii) Immutability(m) = Raw → punkt (ii) powyżej
- Klasy wewnętrzne:
  współdzielą właściciela
  mają własny parametr niezmienności

OIGJ reguły

- Strażnik postaci <X extends Y>? m

  (i) **o.m** jest legalne, jeżeli o spełnia wymagania strażnika
  (ii) w ciele metody X jest traktowane jako Y
  (iii) strażnik w metodzie podklasy jest równy lub słabszy od tego z nadklasy
- This nie może być użyte w połączeniu ze słowem static
- Raw może być użyty tylko po słowie extends
- Joker „?” nie może być używany jako właściciel pola klasy
- Konstruktor nie może mieć argumentów, których właścicielem jest this
  new Foo<X, ..>(..) jest legalne wtw strażnik <I extends Y>? spełnia X = Y = Mutable lub Y = Raw

OIGJ reguły

- Fresh owner, czyli
  <O,TmpO> void deserialize(ByteStream<O> bs) {
ObjectStream<TmpO,ByteStream<O>> os = ... }

Wyzwanie: „factory design pattern”

```
b = new LinkedList<T>();
l = Collections.synchronizedList(b);
```

Z oficjalnej dokumentacji Suna:
“In order to guarantee serial access, it is critical that all access to the backing list is accomplished through the returned list.”

```
1: class SafeSyncList<O,I,E> implements List<O,I,E> {
2:       List<This,I,E> l;
3:       <I extends Raw>? SafeSyncList(
4:       Factory<?,ReadOnly,E> f)
5:       { List<This,I,E> b = f.create();
6:       l = Collections.synchronizedList(b); }
7:       ... // delegate methods to l
8: }
9: class Collections<O,I> {
10: // Sun's original implementation, augmented only by O2 and
I2
11: static <O2,I2,E> List<O2,I2,E>
12: synchronizedList(List<O2,I2,E> list) { ... }
13: }
14: interface Factory<O,I,E>
15: { <O2,I2> List<O2,I2,E> create(); }
16: class LinkedListFactory<O,I,E> implements
17: Factory<O,I,E> {
18: <O2,I2> List<O2,I2,E> create() {
19: return new LinkedList<O2,I2,E>();
20: } }
```

```
1: interface Visitor<O,I,NodeO,NodeI> {
2:       <I extends Mutable>? void
3:        visit(Node<NodeO,NodeI> n);
4: }
5: class Node<O,I> {
6:       void accept(Visitor<?,Mutable,O,I> v)
7:       { v.visit(this) }
8: }
10: Node<This,ReadOnly> readonlyNode = ...;
11: readonlyNode.accept( new
12:     Visitor<World,Mutable,This,ReadOnly>() {
13:             <I extends Mutable>? void
14:             visit(Node<This,ReadOnly> n)
15:      { ... // Can mutate the visitor, but not the nodes. }});
17: // Visiting a mutable node hierarchy.
18: Node<This,Mutable> mutableNode = ...;
19: mutableNode.accept( new
20:     Visitor<World,Mutable,This,Mutable>() {
21:         <I extends Mutable>? void
22:             visit(Node<This,Mutable> n)
23:         { ... // Can mutate the visitor and the nodes. }
24: });
```

Jak to sprawdzali?

- Java 7 adnotacje – zamiast Date<O, I> można @O @I Date @This @I Entry[@This @I] table (dla tablicy haszującej)
- Wtyczka do sprawdzania typów
- Rozszerzona Featherweight Java

Porównanie

	OIGJ	OGJ [33]	IGJ [40]	GUT [14]	UTT [25]	IOJ [19]	JOE_3 [30]
Owner-as-dominator	+	+				+	+
Owner-as-modifier				+	+		
Readonly references	+		+	+	+		+
Immutable objects	+		+			+	+
Uniqueness							+
Ownership transfer					+		
Factory method pattern	+	+	+	+	+		+
Visitor pattern	+		+				
Sun's `LinkedList`	+						