Varargs, try-finally, коллекции

Валеев Тагир

Переменное число аргументов

varargs

```
static void printAll(Object... objects) {
    for (Object object : objects) {
        System.out.println(object);
    }
}

printAll("a", 1, "b", 2.0);
```

- ✓Последний параметр
- ✓Похож на массив
- ✓Можно превратить массив в vararg, не теряя совместимости

Переменное число аргументов

varargs

```
static void printAll(Object... objects) {
    for (Object object : objects) {
        System.out.println(object);
    }
}

printAll("a", 1, "b", 2.0);

Object[] objects = {"a", 1, "b", 2.0};
printAll(objects);
```

- ✓Последний параметр
- ✓Похож на массив
- ✓Можно превратить массив в vararg, не теряя совместимости

Переменное число аргументов

varargs

```
static void printAll(Object... objects) {
    for (Object object : objects) {
        System.out.println(object);
    }
}

printAll(null, null);
printAll(null);
```

Переменное число аргументов

varargs

```
static void printAll(Object... objects) {
    for (Object object : objects) {
        System.out.println(object);
    }
}

printAll(null, null);
printAll(null); ❌
```

Varargs + generics:

полезно, но осторожно

```
static <T> boolean isOneOf(T value, T... options) {
    for (T option : options) {
        if (Objects.equals(value, option)) return true;
    }
    return false;
}
```

Varargs + generics:

полезно, но осторожно

```
void validateGender(String gender) {
  if(!isOneOf(gender, "Male", "Female")) {
    throw new IllegalArgumentException(
                    "Wrong gender: "+gender);
  }
}
```

Heap pollution

```
static <T> boolean isOneOf(T value, T... options) {
    for (T option : options) {
        if (Objects.equals(value, option)) return true;
    }
    return false;
}

isOneOf(shmoption, new Shmoption<>("foo"), new Shmoption<>("bar"));
```

Heap pollution

```
static <T> boolean isOneOf(T value, T... options) {
    for (T option : options) {
        if (Objects.equals(value, option)) return true;
    }
    return false;
}

isOneOf(shmoption, new Shmoption<>("foo"), new Shmoption<>("bar"));
```

Warning:(70, 35) java: unchecked generic array creation for varargs parameter of type Shmoption<java.lang.String>[]

@SafeVarargs

```
@SafeVarargs
static <T> boolean isOneOf(T value, T... options) {
    for (T option : options) {
        if (Objects.equals(value, option)) return true;
    }
    return false;
}

isOneOf(shmoption, new Shmoption<>("foo"), new Shmoption<>("bar"));
```

try-finally

```
try {
    for (String arg : args) {
        System.out.println(arg);
        if(arg.equals("exit")) return;
    }
} finally {
    System.out.println("Exited!");
}
```

try-finally

```
PrintStream oldOutput = System.out;
ByteArrayOutputStream baos = new ByteArrayOutputStream();
System.setOut(new PrintStream(baos, false, "UTF-8"));
try {
    printSomething();
} finally {
    System.setOut(oldOutput);
}
String output = baos.toString("UTF-8");
```

try-finally: способы выхода

```
for (int i = 0; i < 10; i++) {
    try {
        if (i == 5) continue;
        if (i == 7) break;
        System.out.println("Do iteration");
    } finally {
        System.out.println("I = " + i);
    }
}
```

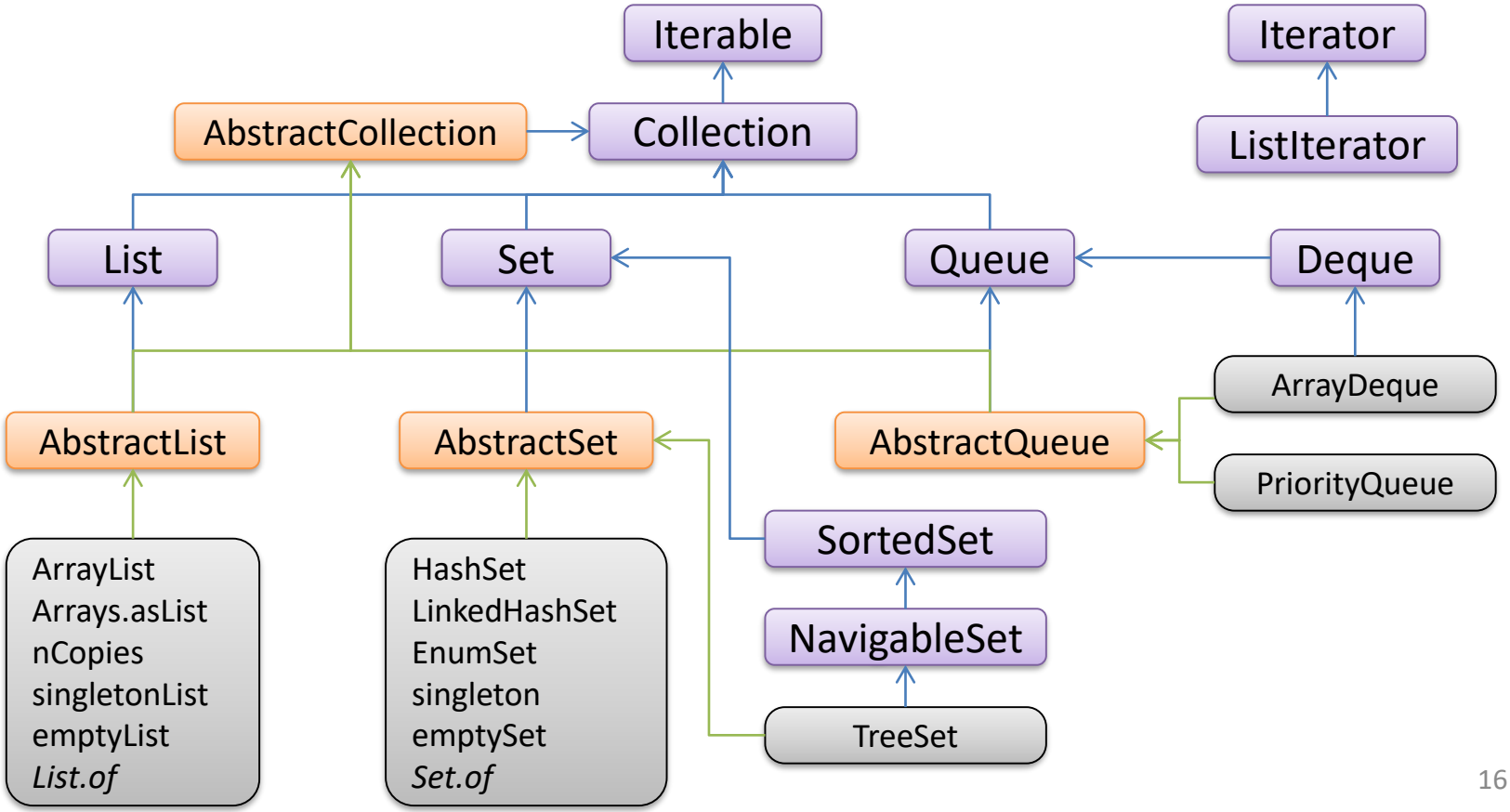
try-finally: как не надо делать

```
public static int test() {
    try {
        return 5;
    }
    finally {
        return 6;
    }
}
```

try-finally: как не надо делать

```
public static int test() {
    int x = 5;
    try {
        return x;
    }
    finally {
        x = 6;
    }
}
```

Стандартные неконкуррентные коллекции



Iterable<E>

```
void printAll(Iterable<?> iterable) {
    Iterator<?> iterator = iterable.iterator();
    while (iterator.hasNext()) {
        Object obj = iterator.next();
        System.out.println(obj);
    }
}
```

Iterable<E>

```
void printAll(Iterable<?> iterable) {
    Iterator<?> iterator = iterable.iterator();
    while (iterator.hasNext()) {
        Object obj = iterator.next();
        System.out.println(obj);
    }
}

void printAll(Iterable<?> iterable) {
    for (Object obj : iterable) {
        System.out.println(obj);
    }
}
```

Iterable<E>

```
void removeEmpty(Iterable<String> iterable) {
    Iterator<String> iterator = iterable.iterator();
    while (iterator.hasNext()) {
        if (iterator.next().isEmpty()) {
            iterator.remove();
        }
    }
}
```

Iterable<E> и Iterator<E>

- ✓ **Iterable** (обычно – `DirectoryStream`!) можно обходить много раз, вызывая iterator() повторно
- ✓ **Iterator** обходится ровно один раз, вперёд:
  - ✓ **hasNext()**: true, если ещё есть элемент, false если нет, идемпотентна
  - ✓ **next()**: следующий элемент или NoSuchElementException()
  - ✓ **remove()**: удалить последний полученный через next() элемент или UnsupportedOperationException()
- ✓ **Iterable** может быть неизменяемым, **Iterator** – нет (разве что если пустой)

```
static <T> Iterable<T> nCopies(T value, int count) {
  if (count < 0)
    throw new IllegalArgumentException("Negative count: " + count);
  return new Iterable<T>() {
    @Override public Iterator<T> iterator() {
      return new Iterator<T>() {
        int rest = count;

        @Override public boolean hasNext() { return rest > 0; }

        @Override public T next() {
          if (rest == 0)
            throw new NoSuchElementException();
          rest--;
          return value;
        }
      };
    }
  };
}
```

```java
static <T> Iterable<T> nCopies(T value, int count) {
  if (count < 0)
    throw new IllegalArgumentException("Negative count: " + count);
  return () -> new Iterator<T>() { // Lambda!
    int rest = count;

    @Override
    public boolean hasNext() {
      return rest > 0;
    }

    @Override
    public T next() {
      if (rest == 0)
        throw new NoSuchElementException();
      rest--;
      return value;
    }
  };
}
```

Collection<E> extends Iterable<E>

- ✓ `int size();`
- ✓ `boolean isEmpty();`
- ✓ `boolean contains(Object o);`
- ✓ `Object[] toArray();`
- ✓ `<T> T[] toArray(T[] a);`

- ✓ `boolean add(E e);`
- ✓ `boolean remove(Object o);`
- ✓ `boolean containsAll(Collection<?> c);`
- ✓ `boolean addAll(Collection<? extends E> c);`
- ✓ `boolean removeAll(Collection<?> c);`
- ✓ `boolean retainAll(Collection<?> c);`
- ✓ `void clear();`

Set<E> extends Collection<E>

- ✓ Не содержит новых методов
- ✓ Содержит уточнённый контракт
- ✓ Не может содержать сам себя
- ✓ Сравним с любым другим Set по содержимому

```
static Set<Integer> rangeSet(int fromInclusive, int toExclusive) {
  return new AbstractSet<Integer>() {
    public Iterator<Integer> iterator() {
      return new Iterator<Integer>() {
        int next = fromInclusive;

        public boolean hasNext() { return next != toExclusive; }

        public Integer next() {
          if (next == toExclusive) {
            throw new NoSuchElementException();
          }
          return next++;
        }
      };
    }

    public int size() { return toExclusive - fromInclusive; }
  };
}
```

```
System.out.println(rangeSet(10, 20).contains(10));

System.out.println(rangeSet(10, 20).contains(20));

System.out.println(rangeSet(10, 20));
```

```
System.out.println(rangeSet(10, 20).contains(10));
true
System.out.println(rangeSet(10, 20).contains(20));
false
System.out.println(rangeSet(10, 20));
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

```
System.out.println(rangeSet(0, Integer.MAX_VALUE).contains(-1));
```

```
public boolean contains(Object o) {
    Iterator<E> it = iterator();
    if (o==null) {
        while (it.hasNext())
            if (it.next()==null)
                return true;
    } else {
        while (it.hasNext())
            if (o.equals(it.next()))
                return true;
    }
    return false;
}
```

```
@Override
public boolean contains(Object o) {
  return o instanceof Integer &&
          ((Integer) o) >= fromInclusive &&
          ((Integer) o) < toExclusive;
}
```

List<E> extends Collection<E>

- ✓ `boolean addAll(int index, Collection<? extends E> c);`
- ✓ `void sort(Comparator<? super E> c);`
- ✓ `E get(int index);`
- ✓ `E set(int index, E element);`
- ✓ `void add(int index, E element);`
- ✓ `E remove(int index); // boolean remove(Object o);`
- ✓ `int indexOf(Object o);`
- ✓ `int lastIndexOf(Object o);`
- ✓ `ListIterator<E> listIterator();`
- ✓ `ListIterator<E> listIterator(int index);`
- ✓ `List<E> subList(int fromIndex, int toIndex);`

ListIterator<E> extends Iterator<E>

- ✓ `boolean hasPrevious();`
- ✓ `E previous();`
- ✓ `int nextIndex();`
- ✓ `int previousIndex();`
- ✓ `void set(E e);`
- ✓ `void add(E e);`

```
static List<Integer> rangeList(int fromInclusive, int toExclusive) {
  return new AbstractList<Integer>() {
    @Override
    public Integer get(int index) {
      if (index < 0 || index >= size())
        throw new IndexOutOfBoundsException(index);
      return fromInclusive + index;
    }

    @Override
    public int size() { return toExclusive - fromInclusive; }
  };
}
```

```
System.out.println(rangeList(10, 20).get(0));

System.out.println(rangeList(10, 20));

System.out.println(rangeList(10, 20).subList(3, 7));
```

```
System.out.println(rangeList(10, 20).get(0));
10
System.out.println(rangeList(10, 20));
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
System.out.println(rangeList(10, 20).subList(3, 7));
[13, 14, 15, 16]
```

```
private static class RangeList extends AbstractList<Integer>
        implements Set<Integer> {
  private final int fromInclusive, toExclusive;

  public RangeList(int fromInclusive, int toExclusive) { … }

  public boolean contains(Object o) {
    return o instanceof Integer && ((Integer) o) >= fromInclusive
            && ((Integer) o) < toExclusive;
  }

  public Integer get(int index) {
    if (index < 0 || index >= size()) throw new IndexOutOfBoundsException(index);
    return fromInclusive + index;
  }

  public int size() { return toExclusive - fromInclusive; }

  public Spliterator<Integer> spliterator() { return super.spliterator(); }
}
```

hashCode()

List#hashCode

Returns the hash code value for this list. The hash code of a list is defined to be the result of the following calculation:

```
int hashCode = 1;
for (E e : list)
   hashCode = 31*hashCode + (e==null ? 0 : e.hashCode());
```

Set#hashCode

Returns the hash code value for this set. The hash code of a set is defined to be the **sum** of the hash codes of the elements in the set, where the hash code of a null element is defined to be zero.

Стандартные списки

- ✓ **ArrayList** – изменяемый список общего назначения
- ✓ **Arrays.asList** – изменяемая обёртка над массивом
- ✓ **Collections.emptyList()** – неизменяемый пустой список
- ✓ **Collections.singletonList(x)** – неизменяемый список из одного элемента
- ✓ **Collections.nCopies(n, x)** – неизменяемый список из n одинаковых элементов
- ✓ **List.of(…)** – (Java 9) неизменяемый список из указанных элементов (или массива), null не приемлет
- ✓ **Collections.unmodifiableList(list)** – неизменяемая обёртка над списком
- ✓ **Collections.synchronizedList(list)** – синхронизированная обёртка над списком
- ✓ **Collections.checkedList(list, type)** – проверяемая обёртка

Collections.checkedList

```
List<Integer> list = new ArrayList<>();
((List<String>)(List<?>)list).add("foo");
System.out.println(list);

List<Integer> list = Collections.checkedList(new ArrayList<>(), Integer.class);
((List<String>)(List<?>)list).add("foo");
System.out.println(list);
```

```
Exception in thread "main" java.lang.ClassCastException: Attempt to insert
class java.lang.String element into collection with element type class
java.lang.Integer
  at java.base/java.util.Collections$CheckedCollection.typeCheck
  at java.base/java.util.Collections$CheckedCollection.add
  at test.Test.main
```

Стандартные множества

- ✓ **HashSet** – изменяемое неупорядоченное множество общего назначения
- ✓ **LinkedHashSet** – изменяемое упорядоченное (ordered) множество общего назначения
- ✓ **TreeSet** – изменяемое сортированное (sorted) множество
- ✓ **EnumSet** – изменяемое множество элементов enum
- ✓ **Collections.emptySet()** – неизменяемое пустое множество
- ✓ **Collections.singleton(x)** – неизменяемое множество из одного элемента
- ✓ **Set.of(…)** – неупорядоченное неизменяемое множество заданных элементов (без null и повторов)
- ✓ **Collections.unmodifiableSet(set)** – неизменяемая обёртка над множеством
- ✓ **Collections.synchronizedSet(set)** – синхронизированная обёртка над множеством
- ✓ **Collections.checkedSet(set, type)** – проверяемая обёртка

HashSet

- ✓ Использует hashCode и расширяемую hash-таблицу (`table[obj.hashCode() % table.length]`)
- ✓ При коллизии формирует связный список
- ✓ Если коллизий много, а элементы сравнимы (Comparable), то растит дерево

TreeSet<E> implements NavigableSet<E>
Comparable/Comparator

- ✓ first(), last()
- ✓ headSet, tailSet, subSet
- ✓ floor(E), ceiling(E)
- ✓ higher(E), lower(E)
- ✓ descendingSet()
- ✓ descendingIterator()

Queue<E> extends Collection<E>

- ✓ offer/**add**
- ✓ poll/**remove**
- ✓ peek/**element**

Deque<E> extends Queue<E>

- ✓ offerFirst/**addFirst**
- ✓ offerLast/**addLast**
- ✓ pollFirst/**removeFirst**
- ✓ pollLast/**removeLast**
- ✓ peekFirst/**getFirst**
- ✓ peekLast/**getLast**

Стандартные очереди

- ✓ ArrayDeque – изменяемая очередь общего назначения
- ✓ PriorityQueue – изменяемая очередь с приоритетом (heap-based)

Не надо использовать

- ✓ Enumeration -> Iterator
- ✓ Vector -> ArrayList
- ✓ Stack -> ArrayDeque
- ✓ Dictionary -> Map
- ✓ Hashtable -> HashMap
- ✓ LinkedList -> ArrayList/ArrayDeque