Введение в программирование на Java

Лекция 14. Дженерики (часть 2). Вариантность.

Виталий Олегович Афанасьев

12 мая 2025

Ограничения на типовые параметры

Интерфейс Comparable (1)

В стандартной библиотеке Java определён интерфейс `Comparable`:

```
1 public interface Comparable<T> {
2     int compareTo(T other);
3 }
```

Логика метода `compareTo`:

- Если `this` равен `other` — вернуть 0.
- Если `this` больше `other` — вернуть положительное число.
- Если `this` меньше `other` — вернуть отрицательное число.

Интерфейс Comparable (2)

Этот интерфейс можно определить для своих типов:

```
1  public class MyInteger implements Comparable<MyInteger> {
2      private int value;
3      ...
4      @Override
5      public int compareTo(MyInteger other) {
6          if (this.value > other.value) {
7              return 1;
8          } else if (this.value < other.value) {
9              return -1;
10         } else {
11             return 0;
12         }
13     }
14 }
```

Ограничения на типовые параметры (1)

```
1  public static <T> T max(
2      List<T> list,
3      T defaultValue
4  ) {
5      T max = defaultValue;
6      for (T element : list) {
7          if (element > max) { // ERROR: Оператор '>' не определён для T
8              max = element;
9          }
10     }
11     return max;
12 }
```

Ограничения на типовые параметры (2)

На типовые параметры можно вводить ограничения. Таким образом, на значениях с обобщёнными типами будут доступны методы из типов-ограничений.

```
1  public static <T extends Comparable<T>> T max(
2      List<T> list,
3      T defaultValue
4  ) {
5      T max = defaultValue;
6      for (T element : list) {
7          if (element.compareTo(max) > 0) {
8              max = element;
9          }
10     }
11     return max;
12 }
```

Ограничения на типовые параметры (3)

Можно вводить несколько ограничений на типовой параметр при помощи &. При этом можно указывать не более одного класса, но произвольное количество интерфейсов.

```
public <T extends ClassOrIntfc & Intfc1 & Intfc2> void example() { ... }
```

Также можно ограничивать типовой параметр другим типовым параметром:

```
public <T, U extends T> void example() { ... }
```

Type Erasure

О type erasure

Обобщённые типы существуют и проверяются только в момент компиляции.

После компиляции происходит т.н. ”type erasure” — обобщённые типы стираются. Во время выполнения программы информация об обобщённых типах недоступна.

Механизм type erasure (1)

В обобщённых типах...

```
1 public class Pair<T, U> {
2     T first;
3     U second;
4 }
```

...типовые параметры стираются и заменяются на `Object`:

```
1 public class Pair {
2     Object first;
3     Object second;
4 }
```

Механизм type erasure (2)

В обобщённых типах с ограничениями на типовые параметры...

```
1 public class Pair<T extends Number, U> {
2     T first;
3     U second;
4 }
```

...типовые параметры стираются и заменяются на их границы:

```
1 public class Pair {
2     Number first;
3     Object second;
4 }
```

Механизм type erasure (3)

Использования обобщённых типов...

```
1 List<String> list = new ArrayList<>();
```

...заменяются на так называемые **raw types ("сырые" типы)**:

```
1 List list = new ArrayList();
```

Raw-типы можно использовать и в своём коде, но не рекомендуется (они нужны лишь для совместимости со старыми версиями языка).

Проблемы с type erasure (1)

Следствие type erasure: если перегруженные методы отличаются только типовыми аргументами параметров — возникает ошибка компиляции.

```
public String sum(List<String>  values) { ... } // Для JVM: sum(List)
public int    sum(List<Integer> values) { ... } // Для JVM: sum(List)
```

Проблемы с type erasure (2)

Другое следствие: нельзя одновременно расширять/реализовывать один и тот же интерфейс с разными типовыми аргументами.

```
public class MyInteger
        implements Comparable<MyInteger>, Comparable<Integer> {
        // implements Comparable, Comparable
    ...
}
```

Другие проблемы с обобщёнными типами

- Нельзя вызвать конструктор типового параметра:
  - `new T(...)`
- Нельзя вызвать конструктор массива из типовых параметров:
  - `new T[...]`
- Нельзя создавать массив из параметризованных типов:
  - `new MyStack<String>[...]`
  - Но объявить переменную/параметр/поле с таким типом можно.
- Нельзя сделать исключение generic:
  - `class MyException<T> extends Throwable`

Вариантность

Проблема при использовании массивов и наследования

Пусть есть следующие типы:

```
public class Employee { ... }
public class Manager extends Employee { ... }
```

При использовании этих типов в массивах можно наткнуться на проблему:

```
1 Manager[] managers = { new Manager("first"), new Manager("second") };
2
3
4
```

Проблема при использовании массивов и наследования

Пусть есть следующие типы:

```
public class Employee { ... }
public class Manager extends Employee { ... }
```

При использовании этих типов в массивах можно наткнуться на проблему:

```
1 Manager[] managers = { new Manager("first"), new Manager("second") };
2 Employee[] employees = managers;
3
4
```

Проблема при использовании массивов и наследования

Пусть есть следующие типы:

```
public class Employee { ... }
public class Manager extends Employee { ... }
```

При использовании этих типов в массивах можно наткнуться на проблему:

```
1 Manager[] managers = { new Manager("first"), new Manager("second") };
2 Employee[] employees = managers;
3 employees[1] = new Employee("third");
4
```

Проблема при использовании массивов и наследования

Пусть есть следующие типы:

```
public class Employee { ... }
public class Manager extends Employee { ... }
```

При использовании этих типов в массивах можно наткнуться на проблему:

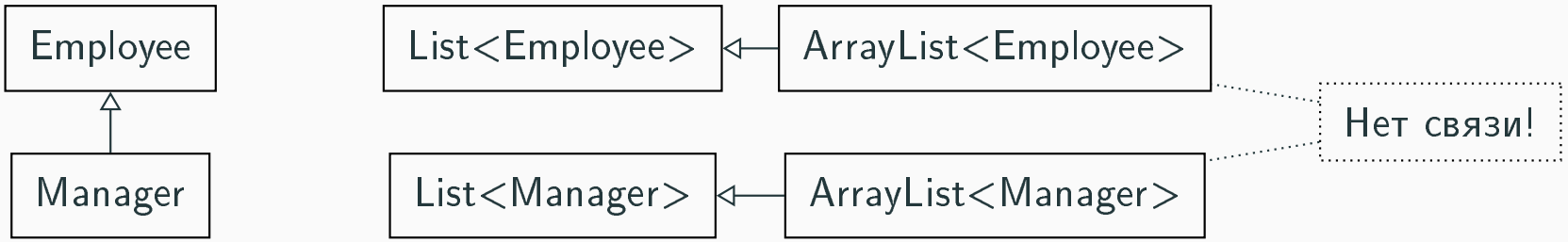
```
1 Manager[] managers = { new Manager("first"), new Manager("second") };
2 Employee[] employees = managers;
3 employees[1] = new Employee("third");
4         // ArrayStoreException
```

Инвариантность (1)

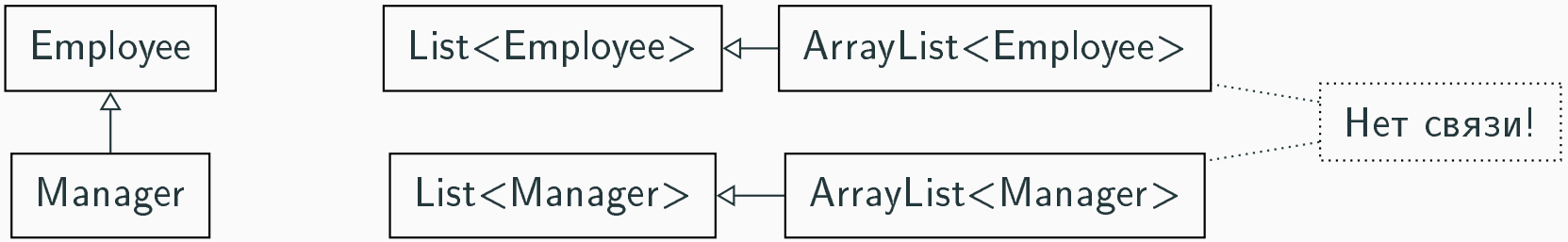
Но со списками (и любыми другими обобщёнными типами) таких проблем не возникает:

```
1 List<Manager> managers = new ArrayList<>();
2 managers.add(new Manager("first"));
3 managers.add(new Manager("second"));
4 List<Employee> employees = managers; // Ошибка компиляции
```

Инвариантность (2)



Инвариантность (2)



Если тип `Child` является подтипом типа `Parent`, но при этом типы `Wrapper<Child>` и `Wrapper<Parent>` никак не связаны, то такая ситуация называется **инвариантностью**.

То есть, `List<Employee>` инвариантный тип.

Проблема инвариантности

Представим метод, который вычисляет суммарную зарплату всех сотрудников:

```
1 public static int totalSalary(List<Employee> employees) {
2     int total = 0;
3     for (Employee employee : employees) {
4         total += employee.getSalary();
5     }
6     return total;
7 }
```

Как в такой метод передать список менеджеров?

```
1 List<Employee> employees = ...;
2 totalSalary(employees); // OK
3
4 List<Manager> managers = ...;
5 totalSalary(managers); // Ошибка компиляции
```

Upper bounded wildcard (1)

```
1  List<Manager> managers = new ArrayList<>();
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
```

Upper bounded wildcard (1)

```
1  List<Manager> managers = new ArrayList<>();
2  List<Employee> employees = new ArrayList<>();
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
```

Upper bounded wildcard (1)

```
1  List<Manager> managers = new ArrayList<>();
2  List<Employee> employees = new ArrayList<>();
3  List<Object> objects = new ArrayList<>();
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
```

Upper bounded wildcard (1)

```
1  List<Manager> managers = new ArrayList<>();
2  List<Employee> employees = new ArrayList<>();
3  List<Object> objects = new ArrayList<>();
4
5  List<? extends Employee> extEmployees;
6
7
8
9
10
11
12
13
14
15
16
17
18
```

Upper bounded wildcard (1)

```
1  List<Manager> managers = new ArrayList<>();
2  List<Employee> employees = new ArrayList<>();
3  List<Object> objects = new ArrayList<>();
4
5  List<? extends Employee> extEmployees;
6
7  extEmployees = managers;  // OK
8
9
10
11
12
13
14
15
16
17
18
```

Upper bounded wildcard (1)

```
1  List<Manager> managers = new ArrayList<>();
2  List<Employee> employees = new ArrayList<>();
3  List<Object> objects = new ArrayList<>();
4
5  List<? extends Employee> extEmployees;
6
7  extEmployees = managers;  // OK
8  extEmployees = employees; // OK
9
10
11
12
13
14
15
16
17
18
```

Upper bounded wildcard (1)

```
1  List<Manager> managers = new ArrayList<>();
2  List<Employee> employees = new ArrayList<>();
3  List<Object> objects = new ArrayList<>();
4
5  List<? extends Employee> extEmployees;
6
7  extEmployees = managers;  // OK
8  extEmployees = employees; // OK
9  extEmployees = objects;   // Ошибка компиляции
10
11
12
13
14
15
16
17
18
```

Upper bounded wildcard (1)

```
1  List<Manager> managers = new ArrayList<>();
2  List<Employee> employees = new ArrayList<>();
3  List<Object> objects = new ArrayList<>();
4
5  List<? extends Employee> extEmployees;
6
7  extEmployees = managers;  // OK
8  extEmployees = employees; // OK
9  extEmployees = objects;   // Ошибка компиляции
10
11 Employee employee = extEmployees.get(0); // OK
12
13
14
15
16
17
18
```

Upper bounded wildcard (1)

```
1  List<Manager> managers = new ArrayList<>();
2  List<Employee> employees = new ArrayList<>();
3  List<Object> objects = new ArrayList<>();
4
5  List<? extends Employee> extEmployees;
6
7  extEmployees = managers;  // OK
8  extEmployees = employees; // OK
9  extEmployees = objects;   // Ошибка компиляции
10
11 Employee employee = extEmployees.get(0); // OK
12 Manager manager = extEmployees.get(0);   // Ошибка компиляции
13
14
15
16
17
18
```

Upper bounded wildcard (1)

```
1  List<Manager> managers = new ArrayList<>();
2  List<Employee> employees = new ArrayList<>();
3  List<Object> objects = new ArrayList<>();
4
5  List<? extends Employee> extEmployees;
6
7  extEmployees = managers;  // OK
8  extEmployees = employees; // OK
9  extEmployees = objects;   // Ошибка компиляции
10
11 Employee employee = extEmployees.get(0); // OK
12 Manager manager = extEmployees.get(0);   // Ошибка компиляции
13 Object object = extEmployees.get(0);     // OK
14
15
16
17
18
```

Upper bounded wildcard (1)

```
1  List<Manager> managers = new ArrayList<>();
2  List<Employee> employees = new ArrayList<>();
3  List<Object> objects = new ArrayList<>();
4
5  List<? extends Employee> extEmployees;
6
7  extEmployees = managers;  // OK
8  extEmployees = employees; // OK
9  extEmployees = objects;   // Ошибка компиляции
10
11 Employee employee = extEmployees.get(0); // OK
12 Manager manager = extEmployees.get(0);   // Ошибка компиляции
13 Object object = extEmployees.get(0);     // OK
14
15 extEmployees.add(new Manager("..."));  // Ошибка компиляции
16
17
18
```

Upper bounded wildcard (1)

```
1  List<Manager> managers = new ArrayList<>();
2  List<Employee> employees = new ArrayList<>();
3  List<Object> objects = new ArrayList<>();
4
5  List<? extends Employee> extEmployees;
6
7  extEmployees = managers;  // OK
8  extEmployees = employees; // OK
9  extEmployees = objects;   // Ошибка компиляции
10
11 Employee employee = extEmployees.get(0); // OK
12 Manager manager = extEmployees.get(0);   // Ошибка компиляции
13 Object object = extEmployees.get(0);     // OK
14
15 extEmployees.add(new Manager("..."));  // Ошибка компиляции
16 extEmployees.add(new Employee("..."));// Ошибка компиляции
17
18
```

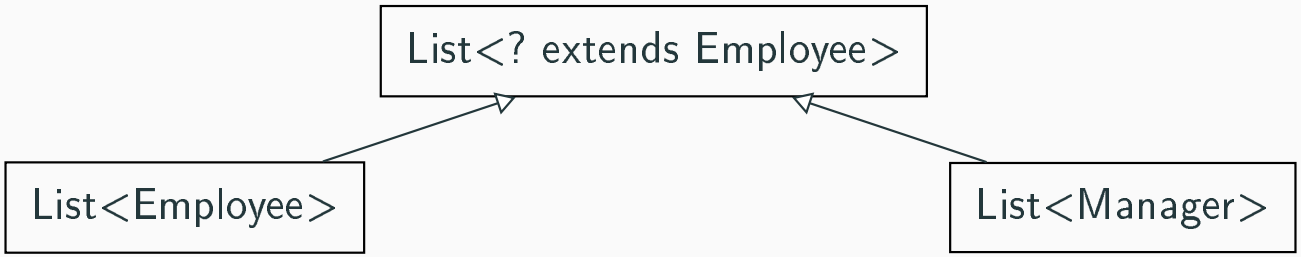
Upper bounded wildcard (1)

```
1  List<Manager> managers = new ArrayList<>();
2  List<Employee> employees = new ArrayList<>();
3  List<Object> objects = new ArrayList<>();
4
5  List<? extends Employee> extEmployees;
6
7  extEmployees = managers;  // OK
8  extEmployees = employees; // OK
9  extEmployees = objects;   // Ошибка компиляции
10
11 Employee employee = extEmployees.get(0); // OK
12 Manager manager = extEmployees.get(0);   // Ошибка компиляции
13 Object object = extEmployees.get(0);     // OK
14
15 extEmployees.add(new Manager("..."));  // Ошибка компиляции
16 extEmployees.add(new Employee("..."));// Ошибка компиляции
17 extEmployees.add(new Object());        // Ошибка компиляции
18
```

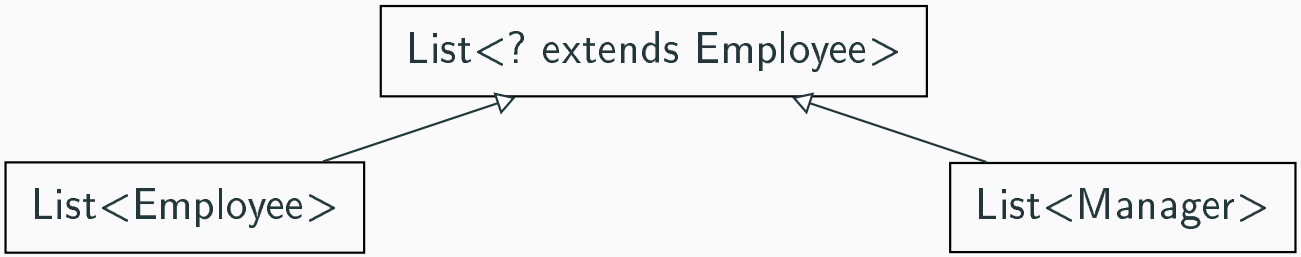
Upper bounded wildcard (1)

```
1  List<Manager> managers = new ArrayList<>();
2  List<Employee> employees = new ArrayList<>();
3  List<Object> objects = new ArrayList<>();
4
5  List<? extends Employee> extEmployees;
6
7  extEmployees = managers;  // OK
8  extEmployees = employees; // OK
9  extEmployees = objects;   // Ошибка компиляции
10
11 Employee employee = extEmployees.get(0); // OK
12 Manager manager = extEmployees.get(0);   // Ошибка компиляции
13 Object object = extEmployees.get(0);     // OK
14
15 extEmployees.add(new Manager("..."));  // Ошибка компиляции
16 extEmployees.add(new Employee("..."));// Ошибка компиляции
17 extEmployees.add(new Object());        // Ошибка компиляции
18 extEmployees.add(null);               // OK
```

Upper bounded wildcard (2)



Upper bounded wildcard (2)



Если тип `Child` является подтипом типа `Parent`, и при этом тип `Wrapper<Child>` является подтипом `Wrapper<Parent>`, то такая ситуация называется **ковариантностью**.

То есть, `List<? extends Employee>` ковариантный тип.

Массивы ковариантны!

Пример использования upper bounded wildcard

```java
public static int totalSalary(List<? extends Employee> employees) {
    int total = 0;
    for (Employee employee : employees) {
        total += employee.getSalary();
    }
    return total;
}
```

```java
List<Employee> employees = ...;
totalSalary(employees); // OK

List<Manager> managers = ...;
totalSalary(managers); // OK
```

Lower bounded wildcard (1)

```
1  List<Manager> managers = new ArrayList<>();
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
```

Lower bounded wildcard (1)

```
1  List<Manager> managers = new ArrayList<>();
2  List<Employee> employees = new ArrayList<>();
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
```

Lower bounded wildcard (1)

```
1  List<Manager> managers = new ArrayList<>();
2  List<Employee> employees = new ArrayList<>();
3  List<Object> objects = new ArrayList<>();
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
```

Lower bounded wildcard (1)

```
1  List<Manager> managers = new ArrayList<>();
2  List<Employee> employees = new ArrayList<>();
3  List<Object> objects = new ArrayList<>();
4
5  List<? super Employee> supEmployees;
6
7
8
9
10
11
12
13
14
15
16
17
18
```

Lower bounded wildcard (1)

```
1  List<Manager> managers = new ArrayList<>();
2  List<Employee> employees = new ArrayList<>();
3  List<Object> objects = new ArrayList<>();
4
5  List<? super Employee> supEmployees;
6
7  supEmployees = objects;   // OK
8
9
10
11
12
13
14
15
16
17
18
```

Lower bounded wildcard (1)

```
1  List<Manager> managers = new ArrayList<>();
2  List<Employee> employees = new ArrayList<>();
3  List<Object> objects = new ArrayList<>();
4
5  List<? super Employee> supEmployees;
6
7  supEmployees = objects;   // OK
8  supEmployees = employees; // OK
9
10
11
12
13
14
15
16
17
18
```

Lower bounded wildcard (1)

```
1  List<Manager> managers = new ArrayList<>();
2  List<Employee> employees = new ArrayList<>();
3  List<Object> objects = new ArrayList<>();
4
5  List<? super Employee> supEmployees;
6
7  supEmployees = objects;   // OK
8  supEmployees = employees; // OK
9  supEmployees = managers;  // Ошибка компиляции
10
11
12
13
14
15
16
17
18
```

Lower bounded wildcard (1)

```
1  List<Manager> managers = new ArrayList<>();
2  List<Employee> employees = new ArrayList<>();
3  List<Object> objects = new ArrayList<>();
4
5  List<? super Employee> supEmployees;
6
7  supEmployees = objects;   // OK
8  supEmployees = employees; // OK
9  supEmployees = managers;  // Ошибка компиляции
10
11 Employee employee = supEmployees.get(0); // Ошибка компиляции
12
13
14
15
16
17
18
```

Lower bounded wildcard (1)

```
1  List<Manager> managers = new ArrayList<>();
2  List<Employee> employees = new ArrayList<>();
3  List<Object> objects = new ArrayList<>();
4
5  List<? super Employee> supEmployees;
6
7  supEmployees = objects;   // OK
8  supEmployees = employees; // OK
9  supEmployees = managers;  // Ошибка компиляции
10
11 Employee employee = supEmployees.get(0); // Ошибка компиляции
12 Manager manager = supEmployees.get(0);   // Ошибка компиляции
13
14
15
16
17
18
```

Lower bounded wildcard (1)

```
1  List<Manager> managers = new ArrayList<>();
2  List<Employee> employees = new ArrayList<>();
3  List<Object> objects = new ArrayList<>();
4
5  List<? super Employee> supEmployees;
6
7  supEmployees = objects;   // OK
8  supEmployees = employees; // OK
9  supEmployees = managers;  // Ошибка компиляции
10
11 Employee employee = supEmployees.get(0); // Ошибка компиляции
12 Manager manager = supEmployees.get(0);   // Ошибка компиляции
13 Object object = supEmployees.get(0);     // OK
14
15
16
17
18
```

Lower bounded wildcard (1)

```
1  List<Manager> managers = new ArrayList<>();
2  List<Employee> employees = new ArrayList<>();
3  List<Object> objects = new ArrayList<>();
4
5  List<? super Employee> supEmployees;
6
7  supEmployees = objects;   // OK
8  supEmployees = employees; // OK
9  supEmployees = managers;  // Ошибка компиляции
10
11 Employee employee = supEmployees.get(0); // Ошибка компиляции
12 Manager manager = supEmployees.get(0);   // Ошибка компиляции
13 Object object = supEmployees.get(0);     // OK
14
15 supEmployees.add(new Manager("..."));  // OK
16
17
18
```

Lower bounded wildcard (1)

```
1  List<Manager> managers = new ArrayList<>();
2  List<Employee> employees = new ArrayList<>();
3  List<Object> objects = new ArrayList<>();
4
5  List<? super Employee> supEmployees;
6
7  supEmployees = objects;   // OK
8  supEmployees = employees; // OK
9  supEmployees = managers;  // Ошибка компиляции
10
11 Employee employee = supEmployees.get(0); // Ошибка компиляции
12 Manager manager = supEmployees.get(0);   // Ошибка компиляции
13 Object object = supEmployees.get(0);     // OK
14
15 supEmployees.add(new Manager("..."));  // OK
16 supEmployees.add(new Employee("..."); // OK
17
18
```

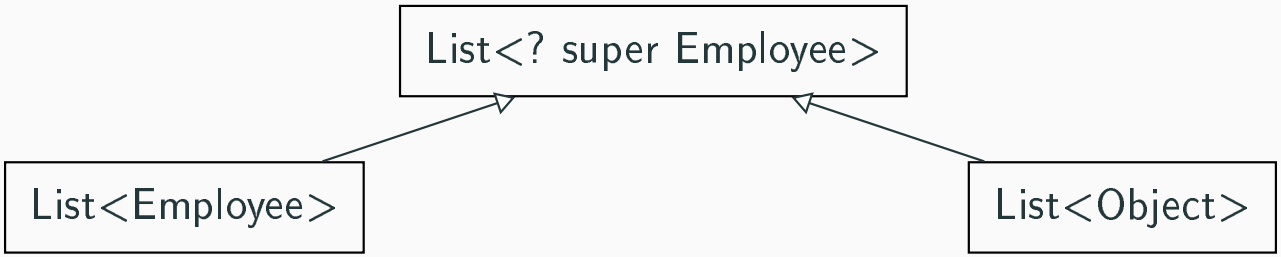
Lower bounded wildcard (1)

```
1  List<Manager> managers = new ArrayList<>();
2  List<Employee> employees = new ArrayList<>();
3  List<Object> objects = new ArrayList<>();
4
5  List<? super Employee> supEmployees;
6
7  supEmployees = objects;   // OK
8  supEmployees = employees; // OK
9  supEmployees = managers;  // Ошибка компиляции
10
11 Employee employee = supEmployees.get(0); // Ошибка компиляции
12 Manager manager = supEmployees.get(0);   // Ошибка компиляции
13 Object object = supEmployees.get(0);     // OK
14
15 supEmployees.add(new Manager("..."));  // OK
16 supEmployees.add(new Employee("..."));// OK
17 supEmployees.add(new Object());        // Ошибка компиляции
18
```

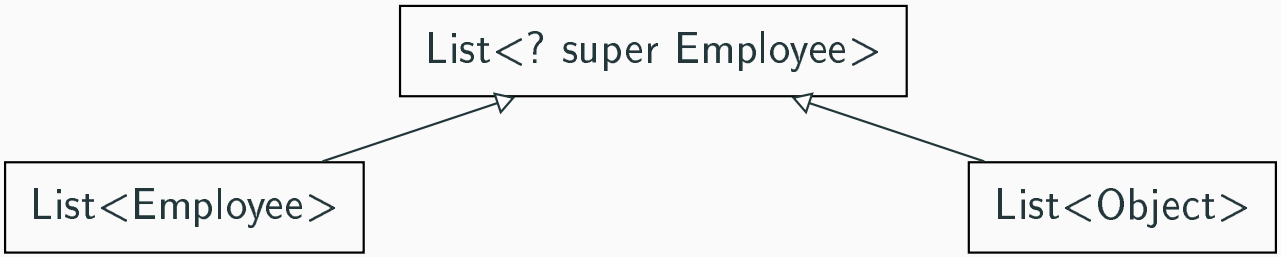
Lower bounded wildcard (1)

```
1  List<Manager> managers = new ArrayList<>();
2  List<Employee> employees = new ArrayList<>();
3  List<Object> objects = new ArrayList<>();
4
5  List<? super Employee> supEmployees;
6
7  supEmployees = objects;   // OK
8  supEmployees = employees; // OK
9  supEmployees = managers;  // Ошибка компиляции
10
11 Employee employee = supEmployees.get(0); // Ошибка компиляции
12 Manager manager = supEmployees.get(0);   // Ошибка компиляции
13 Object object = supEmployees.get(0);     // OK
14
15 supEmployees.add(new Manager("..."));  // OK
16 supEmployees.add(new Employee("..."));// OK
17 supEmployees.add(new Object());        // Ошибка компиляции
18 supEmployees.add(null);               // OK
```

Lower bounded wildcard (2)



Lower bounded wildcard (2)



Если тип `Child` является **под**типом типа `Parent`, но при этом тип `Wrapper<Child>` является **над**типом `Wrapper<Parent>`, то такая ситуация называется **контравариантностью**.

То есть, `List<? super Employee>` контравариантный тип.

Пример использования lower bounded wildcard

```
1 public static void addAll(
2     List<? extends Employee> source,
3     List<? super Employee> destination
4 ) {
5     for (Employee employee : source) {
6         destination.add(employee);
7     }
8 }
```

```
1 List<Object> objects = ...;
2 List<Employee> employees = ...;
3 List<Manager> managers = ...;
4
5 addAll(managers, employees); // OK
6 addAll(managers, objects); // OK
```

Unbounded wildcard

```
1  List<Manager> managers = new ArrayList<>();
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
```

Unbounded wildcard

```
1  List<Manager> managers = new ArrayList<>();
2  List<Employee> employees = new ArrayList<>();
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
```

Unbounded wildcard

```
1  List<Manager> managers = new ArrayList<>();
2  List<Employee> employees = new ArrayList<>();
3  List<Object> objects = new ArrayList<>();
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
```

Unbounded wildcard

```
1  List<Manager> managers = new ArrayList<>();
2  List<Employee> employees = new ArrayList<>();
3  List<Object> objects = new ArrayList<>();
4
5  List<?> unbounded;
6
7
8
9
10
11
12
13
14
15
16
17
18
```

Unbounded wildcard

```
1  List<Manager> managers = new ArrayList<>();
2  List<Employee> employees = new ArrayList<>();
3  List<Object> objects = new ArrayList<>();
4
5  List<?> unbounded;
6
7  unbounded = objects;   // OK
8
9
10
11
12
13
14
15
16
17
18
```

Unbounded wildcard

```
1  List<Manager> managers = new ArrayList<>();
2  List<Employee> employees = new ArrayList<>();
3  List<Object> objects = new ArrayList<>();
4
5  List<?> unbounded;
6
7  unbounded = objects;   // OK
8  unbounded = employees; // OK
9
10
11
12
13
14
15
16
17
18
```

Unbounded wildcard

```
1  List<Manager> managers = new ArrayList<>();
2  List<Employee> employees = new ArrayList<>();
3  List<Object> objects = new ArrayList<>();
4
5  List<?> unbounded;
6
7  unbounded = objects;   // OK
8  unbounded = employees; // OK
9  unbounded = managers;  // OK
10
11
12
13
14
15
16
17
18
```

Unbounded wildcard

```
1  List<Manager> managers = new ArrayList<>();
2  List<Employee> employees = new ArrayList<>();
3  List<Object> objects = new ArrayList<>();
4
5  List<?> unbounded;
6
7  unbounded = objects;   // OK
8  unbounded = employees; // OK
9  unbounded = managers;  // OK
10
11 Employee employee = unbounded.get(0); // Ошибка компиляции
12
13
14
15
16
17
18
```

Unbounded wildcard

```
1  List<Manager> managers = new ArrayList<>();
2  List<Employee> employees = new ArrayList<>();
3  List<Object> objects = new ArrayList<>();
4
5  List<?> unbounded;
6
7  unbounded = objects;   // OK
8  unbounded = employees; // OK
9  unbounded = managers;  // OK
10
11 Employee employee = unbounded.get(0); // Ошибка компиляции
12 Manager manager = unbounded.get(0);   // Ошибка компиляции
13
14
15
16
17
18
```

Unbounded wildcard

```
1  List<Manager> managers = new ArrayList<>();
2  List<Employee> employees = new ArrayList<>();
3  List<Object> objects = new ArrayList<>();
4
5  List<?> unbounded;
6
7  unbounded = objects;   // OK
8  unbounded = employees; // OK
9  unbounded = managers;  // OK
10
11 Employee employee = unbounded.get(0); // Ошибка компиляции
12 Manager manager = unbounded.get(0);   // Ошибка компиляции
13 Object object = unbounded.get(0);     // OK
14
15
16
17
18
```

Unbounded wildcard

```
1  List<Manager> managers = new ArrayList<>();
2  List<Employee> employees = new ArrayList<>();
3  List<Object> objects = new ArrayList<>();
4
5  List<?> unbounded;
6
7  unbounded = objects;   // OK
8  unbounded = employees; // OK
9  unbounded = managers;  // OK
10
11 Employee employee = unbounded.get(0); // Ошибка компиляции
12 Manager manager = unbounded.get(0);   // Ошибка компиляции
13 Object object = unbounded.get(0);     // OK
14
15 unbounded.add(new Manager("..."));  // Ошибка компиляции
16
17
18
```

Unbounded wildcard

```
1  List<Manager> managers = new ArrayList<>();
2  List<Employee> employees = new ArrayList<>();
3  List<Object> objects = new ArrayList<>();
4
5  List<?> unbounded;
6
7  unbounded = objects;   // OK
8  unbounded = employees; // OK
9  unbounded = managers;  // OK
10
11 Employee employee = unbounded.get(0); // Ошибка компиляции
12 Manager manager = unbounded.get(0);   // Ошибка компиляции
13 Object object = unbounded.get(0);     // OK
14
15 unbounded.add(new Manager("..."));  // Ошибка компиляции
16 unbounded.add(new Employee("..."));// Ошибка компиляции
17
18
```

Unbounded wildcard

```
1  List<Manager> managers = new ArrayList<>();
2  List<Employee> employees = new ArrayList<>();
3  List<Object> objects = new ArrayList<>();
4
5  List<?> unbounded;
6
7  unbounded = objects;   // OK
8  unbounded = employees; // OK
9  unbounded = managers;  // OK
10
11 Employee employee = unbounded.get(0); // Ошибка компиляции
12 Manager manager = unbounded.get(0);   // Ошибка компиляции
13 Object object = unbounded.get(0);     // OK
14
15 unbounded.add(new Manager("..."));  // Ошибка компиляции
16 unbounded.add(new Employee("..."));// Ошибка компиляции
17 unbounded.add(new Object());        // Ошибка компиляции
18
```

Unbounded wildcard

```
1  List<Manager> managers = new ArrayList<>();
2  List<Employee> employees = new ArrayList<>();
3  List<Object> objects = new ArrayList<>();
4
5  List<?> unbounded;
6
7  unbounded = objects;   // OK
8  unbounded = employees; // OK
9  unbounded = managers;  // OK
10
11 Employee employee = unbounded.get(0); // Ошибка компиляции
12 Manager manager = unbounded.get(0);   // Ошибка компиляции
13 Object object = unbounded.get(0);     // OK
14
15 unbounded.add(new Manager("..."));  // Ошибка компиляции
16 unbounded.add(new Employee("..."));// Ошибка компиляции
17 unbounded.add(new Object());        // Ошибка компиляции
18 unbounded.add(null);               // OK
```

PECS

При использовании wildcards полезно пользоваться мнемоникой PECS.

PECS — Producer Extends Consumer Super.

Если объект используется для чтения из него (т.е. как producer) — стоит использовать `? extends ...`

Если объект используется для записи в него (т.е. как consumer) — стоит использовать `? super ...`

Рекомендуется к прочтению: **J. Bloch. Effective Java.**
Use bounded wildcards to increase API flexibility.

PECS

При использовании wildcards полезно пользоваться мнемоникой PECS.

PECS — Producer Extends Consumer Super.

Если объект используется для чтения из него (т.е. как producer) — стоит использовать `? extends ...`

Если объект используется для записи в него (т.е. как consumer) — стоит использовать `? super ...`

```
public static int totalSalary(List<? extends Employee> employees)

public static void addAll(
    List<? extends Employee> source,
    List<? super Employee> destination
)
```

Рекомендуется к прочтению: **J. Bloch. Effective Java. Use bounded wildcards to increase API flexibility**.

Вариантность: Резюме

Резюмируя:

- `List<? extends Type>` — **ковариантный** тип.
- `List<? super Type>` — **контравариантный** тип.
- `List<Type>` — **инвариантный** тип.

Массивы ковариантны.