Generics

Классы-оболочки

В языке Java существуют классы-оболочки, которые являются объектным представлением восьми примитивных типов. Все классы-оболочки являются immutable. Автоупаковка и распаковка позволяют легко конвертировать примитивные типы в их соответствующие классы-оболочки и наоборот.

Пример упаковки и распаковки

```
int a = 5;
Integer b = a; // автоупаковка
Integer c = new Integer(a); // упаковка
int d = b; // распаковка
int e = (int)c; // необязательно
System.out.println(c); // 5
```

Примитивные типы и обёртки

Примитивный тип	Класс-обертка	Аргументы
byte	Byte	byte или String
short	Short	short или String
int	Integer	int или String
long	Long	long или String
float	Float	float, double или String
double	Double	double или String
char	Character	char
boolean	Boolean	boolean или String

Зачем нужны оболочки

Разработчиками языка Java было принято решение отделить примитивные типы и классы-оболочки, указав при этом следующее:

- Используйте классы-обёртки, когда работаете со стандартными коллекциями
- Используйте примитивные типы для того, чтобы ваши программы были максимально просты

Ещё одним важным моментом является то, что примитивные типы **не могут** быть **null**, а классы-оболочки — **могут**. Также классы-оболочки могут быть использованы для достижения полиморфизма.

Практика

Создайте объект типа Double, и изучите список методов, предоставляемых этим классом. Создайте объект на основе целого числа, вещественного числа, строки. Попытайтесь изменить состояние объекта.

https://stackoverflow.com/questions/3130311/weird-integer-boxing-in-java

Обобщения (generics)

Нередко, создаваемые разработчиками алгоритмы и коллекции могут быть успешно использованы для разных типов данных без какого-либо изменения. Например, не зависят от типа данных алгоритмы поиска и сортировки, а класс List пригодился бы как для хранения целых чисел, так и для хранения объектов типа Student. Чтобы не создавать однообразные реализации для каждого типа данных, в языке Java начиная с версии SE5.0 были введены **обобщения,** или **обобщённые типы**, которые позволяют создавать более безопасный и при этом универсальный код.

https://urvanov.ru/2016/04/28/java-8-%D0%BE%D0%B1%D0%BE%D0%B1%D1%89%D0%B5%D0%BD%D0%B8%D1%8F/

Безопасность

```
int x = 31;
String s = "hello";
ArrayList array = new ArrayList();
array.add(x); // упаковка (boxing)
array.add(s); // упаковки нет!
int y = (int) array.get(0); // unboxing
int z = (int) array.get(1); // упс!!!
```

Упаковка и распаковка

В примере используется стандартный класс ArrayList из пакета java.util, который представляет коллекцию объектов. Чтобы поместить объект в коллекцию, применяется метод add. И хотя в коллекцию добавляются число и строка, по существу ArrayList содержит коллекцию значений типа Object. Таким образом, в вызове array.add(x); значение переменной x вначале "**упаковывается**" в объект типа Integer и апкастится до типа Object, а потом при получении элементов из коллекции - наоборот, "**распаковывается**" в нужный тип.

Устройство ArrayList

ArrayList устроен как массив ссылок типа Object, что позволяет добавлять в коллекцию переменные любого типа. Такая гибкость в некоторых случаях удобна, однако чаще всего в коллекции хранятся переменные одного и того же типа. Можно легко допустить ошибку приведения при извлечении данных из коллекции, т.е. поместить в коллекцию переменную одного типа, а при извлечении выполнить приведение к другому типу…

Проблемы

Упаковка и распаковка (**boxing** и **unboxing**) ведут к снижению производительности, поскольку система должна выполнить необходимые преобразования. Существует и другая проблема, связанная с упаковкой-распаковкой, - проблема безопасности типов. Например, во время выполнения последней строки возникает ошибка.

Хранение ссылок

Следует отметить, что если хранить в коллекции объекты ссылочных (не примитивных) типов, то снижения производительности происходить не будет, так как выполняется не упаковка-распаковка, а лишь формальное преобразование пользовательского типа в Object или наоборот.

Решение

Обе проблемы смогут решить обобщённые типы. Они позволяют указать конкретный тип данных, который будет использоваться для коллекции или алгоритма (поддерживаются обобщённые классы, интерфейсы и методы). Например, в Java также существует обобщённая версия класса ArrayList:

Обобщённая версия

```
int x = 32;
String s = "hello";
ArrayList<Integer> ar = new ArrayList<>();
ar.add(x); // упаковка не нужна
ar.add(s); // ошибка компиляции!
int y = ar.get(0); // распаковка не нужна
```

Комментарии к примеру

Так как теперь используется обобщённая версия класса ArrayList, то нужно будет задать определённый тип данных, для которого этот класс будет применяться. Далее добавляется число и строка в коллекцию. Но если число будет добавлено в коллекцию без проблем, так как коллекция типизирована типом int, то на строке ar.add(s); возникнет ошибка времени компиляции, и придётся удалить эту строку. Таким образом, при применении обобщённого варианта класса снижается как количество потенциальных ошибок, так и время на выполнение программы.

Пример generic-класса (Point)

https://git.io/vokjC

Два параметра типа

https://git.io/vot6i

Raw types (сырые типы)

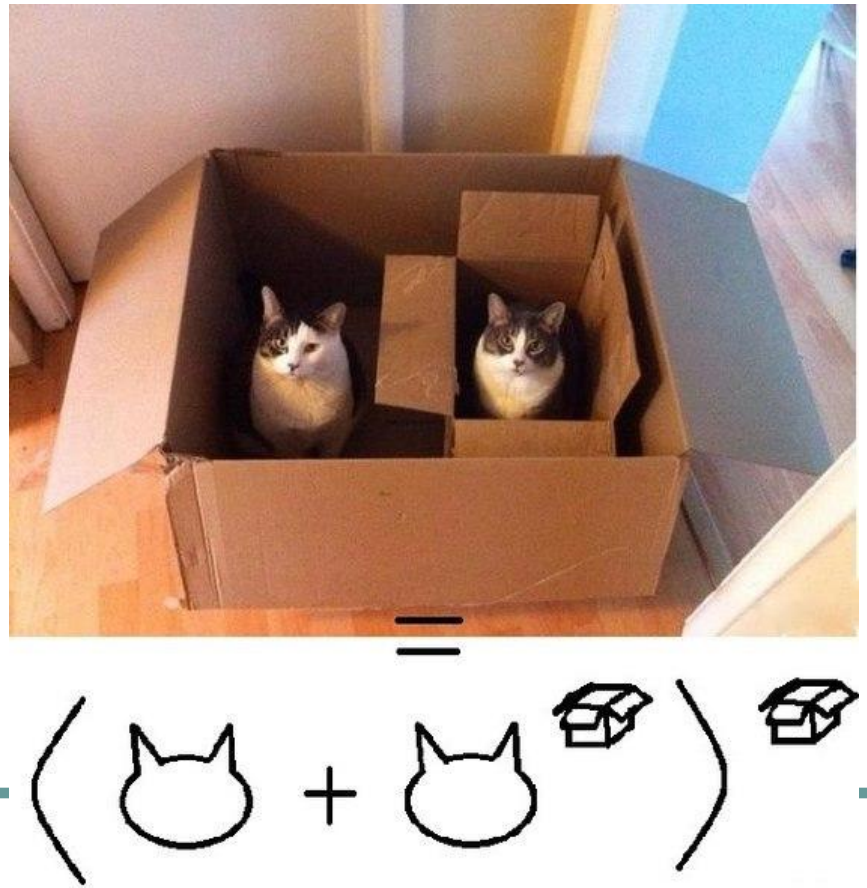
```
Forest f = new Forest();
f.setInhabitant1(new Fairy());
f.setInhabitant2(new Elf());
f.setInhabitant2(new Fairy());

Fairy fairy = (Fairy) f.getInhabitant1();
Elf elf = (Elf) f.getInhabitant2(); // упс!

Forest<Fairy, Elf> f2 = f;
Forest f3 = new Forest<Fairy, Elf>();
```

Определение

Сырой тип — это имя обобщённого класса или интерфейса без аргументов типа. Можно часто увидеть использование сырых типов в старом коде, поскольку многие классы (например, коллекции), до Java 5 были необобщёнными. При использовании сырых типов получается то же самое поведение, которое было до введения обобщений в Java.

Пример с котиками

https://git.io/volfe



Ограниченный тип

В некоторых случаях имеет смысл ограничить типы, которые можно использовать в качестве аргументов в параметризованных типах. Например, в Термос можно будет наливать только ГорячиеНапитки. Подобное ограничение можно сделать с помощью **ограниченного параметра типа (bounded type parameters)**.

Чтобы объявить ограниченный параметр типа, нужно после имени параметра указать ключевое слово extends, а затем указать верхнюю границу (upper bound). В этом контексте extends означает как extends, так и implements.

Ограничение параметра типа

https://git.io/votrb

```
class AverageCalculator<T extends
Number & Comparable & Serializable> {
```

Соглашение об именовании

Переменные типа именуются одной буквой в верхнем регистре. Это позволяет легко отличить переменную типа от класса или интерфейса. Наиболее часто используемые имена для параметров типа:

- E — элемент (Element, широко используется в Java Collections Framework)
- K — Ключ
- N — Число
- T — Тип
- V — Значение
- S, U, V и т. п. — 2-й, 3-й, 4-й типы

Generic method

https://git.io/votdx

Generic constructor

https://git.io/votFV

Конструкторы могут быть обобщёнными как в обобщённых, так и в необобщённых классах.

Generic interface

- Iterable<T>
- Comparable<T>

https://git.io/votbd

Обобщения и наследование

Можно присвоить объекту одного типа объект другого типа, если эти типы совместимы. Например, можно присвоить объект типа Integer переменной типа Object, так как Object является одним из супертипов Integer:

```
Object someObject = new Object();
Integer someInteger = new Integer(10);
someObject = someInteger;   // OK
```

Обобщения и наследование

В объектно-ориентированной терминологии это называется связью «является» (“is a”). Так как Integer является Object -ом, то такое присвоение разрешено. Но Integer также является и Number-ом, поэтому следующий код тоже корректен:

```
public void someMethod(Number n) { /* ... */ }
someMethod(new Integer(10));   // OK
someMethod(new Double(10.1));   // OK
```

Обобщения и наследование

Это также верно для обобщений. Можно осуществить вызов обобщённого типа, передав Number в качестве аргумента типа, и любой дальнейший вызов будет разрешён, если аргумент совместим с Number:

```
Box<Number> box = new Box<Number>();
box.add(new Integer(10));   // OK
box.add(new Double(10.1));  // OK
```

Обобщения и наследование

```
void boxTest(Box<Number> n) { /* ... */ }
```

Можно ли будет передать в этот метод объект типа Box<Integer> или Box<Double>?

Нет, так как Box<Integer> и Box<Double> не являются потомками Box<Number>!

Важно запомнить!

Для двух типов A и B (например, Number и Integer), MyClass<A> не имеет никакой связи или родства с MyClass<B> , независимо от того, как A и B связаны между собой. Общий родитель MyClass<A> и MyClass<B> — это Object.

Неизвестный тип (wildcard)

В обобщённом коде иногда встречается знак вопроса (?), называемый подстановочным символом, и означает это «неизвестный тип». Подстановочный символ может использоваться в разных ситуациях: как параметр типа, поля, локальной переменной, иногда в качестве возвращаемого типа.

Unbounded wildcard

https://git.io/vothz

Если просто использовать подстановочный символ <?>, то получится подстановочный символ без ограничений. Например, List<?> означает список неизвестного (т.е., почти любого) типа.

Зачем нужен wildcard

```java
public static void printList(List<Object> list) {
    for (Object elem : list)
        System.out.println(elem + " ");
    System.out.println();
}
```

Цель метода `printList` — вывод в консоль списка любого типа, но сейчас он её не выполняет, так как он может вывести в консоль только список объектов типа `Object`. Он не может принимать в качестве параметра `List<Integer>`, `List<String>`, `List<Double>` и какие-либо ещё, так как они не являются дочерними типами для `List<Object>`.

Зачем нужен wildcard

```java
public static void printList(List<?> list) {
    for (Object elem: list)
        System.out.print(elem + " ");
    System.out.println();
}
```

`List<A>` является дочерним типом для `List<?>` для любого конкретного типа `A` , поэтому вы можете использовать `printList` для вывода в консоль списков любого типа:

```java
List<Integer> li = Arrays.asList(1, 2, 3);
List<String>  ls = Arrays.asList("one", "two", "three");
printList(li);
printList(ls);
```

Заметка: Метод `Arrays.asList` конвертирует массив и возвращает список фиксированного размера.

Upper bounded wildcard

Можно использовать подстановочный символ, ограниченный сверху, чтобы ослабить ограничения для переменной класса. Например, если хочется написать метод, который работает только с List<Integer>, List<Double> и List<Number>, этого можно достичь с помощью ограниченного сверху подстановочного символа.

List<? extends Number>

Пример на UBW

https://git.io/voqe2

Lower bounded wildcard

Ограниченный снизу подстановочный символ ограничивает неизвестный тип так, чтобы он был либо указанным типом, либо одним из его предков. Допустим, хочется написать метод, который добавляет объекты Mops в список. Чтобы максимизировать гибкость, в список можно будет добавлять ещё и Dog с Animal-ом — всё, что может хранить экземпляры класса Mops.

List<? super Mops>

Почему обобщения не работают с примитивными типами?

http://stackoverflow.com/questions/2721546/why-dont-java-generics-support-primitive-types

Generics in Java are an entirely compile-time construct - the compiler turns all generic uses into casts to the right type. This is to maintain backwards compatibility with previous JVM runtimes.

Стирание типа

Обобщения были введены в язык программирования Java для обеспечения более жёсткого контроля типов во время компиляции и для поддержки обобщённого программирования. Для реализации обобщения компилятор:

- Заменяет все параметры типа в обобщённых типах их границами или Object-ами, если параметры типа не ограничены. Сгенерированный байт-код содержит только обычные классы, интерфейсы и методы!
- Вставляет приведение типов где необходимо, чтобы сохранить безопасность типа.

Стирание типа

```
public class Node<T> {

    private T data;
    private Node<T> next;

    public Node(T data, Node<T> next) }
        this.data = data;
        this.next = next;
    }

    public T getData() { return data; }
    // ...
}
```

```
1  public class Node {
2
3      private Object data;
4      private Node next;
5
6      public Node(Object data, Node next) {
7          this.data = data;
8          this.next = next;
9      }
10
11     public Object getData() { return data; }
12     // ...
13 }
```

Стирание типа

```
public class Node<T extends Comparable<T>> {

    private T data;
    private Node<T> next;

    public Node(T data, Node<T> next) {
        this.data = data;
        this.next = next;
    }

    public T getData() { return data; }
    // ...
}
```

```
1  public class Node {
2
3      private Comparable data;
4      private Node next;
5
6      public Node(Comparable data, Node n
7          this.data = data;
8          this.next = next;
9      }
10
11     public Comparable getData() { retur
12     // ...
13 }
```

```java
// Подсчитывает количество вхождений элемента в массив.
//
public static <T> int count(T[] anArray, T elem) {
    int cnt = 0;
    for (T e : anArray)
        if (e.equals(elem))
            ++cnt;
        return cnt;
}
```

```java
public static int count(Object[] anArray, Object elem) {
    int cnt = 0;
    for (Object e : anArray)
        if (e.equals(elem))
            ++cnt;
        return cnt;
}
```

На тему стирания типов

- http://www.journaldev.com/1663/java-generics-example-method-class-interface#type-erasure
- http://www.angelikalanger.com/GenericsFAQ/FAQSections/TechnicalDetails.html#FAQ101

Чего делать нельзя

```java
class Pair<K, V> {

    private K key;
    private V value;

    public Pair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    // ...
}
```

При создании объекта `Pair` вы не можете заменять примитивным типом формальные параметры `K` и `V` :

```java
Pair<int, char> p = new Pair<>(8, 'a');  // compile-time error
```

Вы можете заменить их только непримитивными типами:

```java
Pair<Integer, Character> p = new Pair<>(8, 'a');
```

Компилятор использует автоупаковку `Integer.valueOf(8)` и `Character('a')`

Чего делать нельзя

Вы не можете создать экземпляр параметра типа. Например, следующий код приведёт к ошибке компиляции:

```java
public static <E> void append(List<E> list) {
    E elem = new E();  // compile-time error
    list.add(elem);
}
```

В качестве обходного пути вы можете создать объект параметра типа с помощью отражения (reflection):

```java
public static <E> void append(List<E> list, Class<E> cls) throws Exception {
    E elem = cls.newInstance();   // OK
    list.add(elem);
}
```

Вы можете вызвать метод `append` вот так:

```java
List<String> ls = new ArrayList<>();
append(ls, String.class);
```

Чего делать нельзя

Статические поля класса являются общими для всех объектов этого класса, поэтому статические поля с типом параметра типа запрещены. Рассмотрите следующий класс:

```java
public class MobileDevice<T> {
    private static T os;

    // ...
}
```

Если бы статические поля с типом параметра типа были бы разрешены, то следующий код сбивал бы с толку:

```java
MobileDevice<Smartphone> phone = new MobileDevice<>();
MobileDevice<Pager> pager = new MobileDevice<>();
MobileDevice<TabletPC> pc = new MobileDevice<>();
```

Так как статическое поле `os` является общим для `phone` , `pager` и `pc` , то какого типа `os` ? Оно не может быть `Smartphone` , `Pager` и `TabletPC` в одно и то же время, поэтому вы не можете создавать статические поля с типом параметра типа.

Чего делать нельзя

Так как компилятор Java стирает все параметры типа из обобщённого кода, то вы не можете проверить во время выполнения, какой параметризованный тип используется для обобщённого типа:

```java
public static <E> void rtti(List<E> list) {
    if (list instanceof ArrayList<Integer>) {  // compile-time error
        // ...
    }
}
```

Множество параметризованных типов, которые можно передать в метод `rtti` :

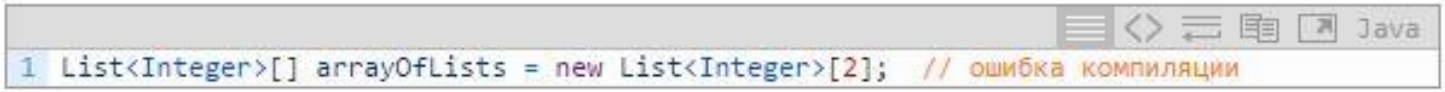
```
S = {ArrayList<Integer>, ArrayList<String>, LinkedList<Character>, ...}
```

Во время выполнения нет параметров типа, поэтому мы не можем различить `ArrayList<Integer>` и `ArrayList<String>`.

Чего делать нельзя

Невозможно создавать массивы параметризованных типов

Вы не можете создавать массивы параметризованных типов. Например, следующий код не будет компилироваться:



It's because Java's arrays (unlike generics) contain, at runtime, information about its component type. So you must know the component type when you create the array. Since you don't know what T is at runtime, you can't create the array.

http://docs.oracle.com/javase/tutorial/java/generics/restrictions.html

Что почитать про обобщения

- https://urvanov.ru/2016/04/28/java-8-%D0%BE%D0%B1%D0%BE%D0%B1%D1%89%D0%B5%D0%BD%D0%B8%D1%8F/
- http://rsdn.ru/article/java/genericsinjava.xml
- http://developer.alexanderklimov.ru/android/java/generic.php
- http://www.k-press.ru/cs/2008/3/generic/generic.asp
- http://www.quizful.net/post/java-generics-tutorial
- http://javarevisited.blogspot.com/2011/09/generics-java-example-tutorial.html
- https://uk.wikipedia.org/wiki/%D0%A3%D0%B7%D0%B0%D0%B3%D0%B0%D0%BB%D1%8C%D0%BD%D0%B5%D0%BD%D0%BD%D1%8F_%D0%B2_Java
- http://docs.oracle.com/javase/tutorial/extra/generics/morefun.html

Практика

- Переделать классы-коллекции ArrayList, SLL, DLL, BinaryTree таким образом, чтобы они стали обобщёнными.
- Реализовать интерфейс Iterable<T> для ваших реализаций типов ArrayList<T> и BinaryTree<T>.