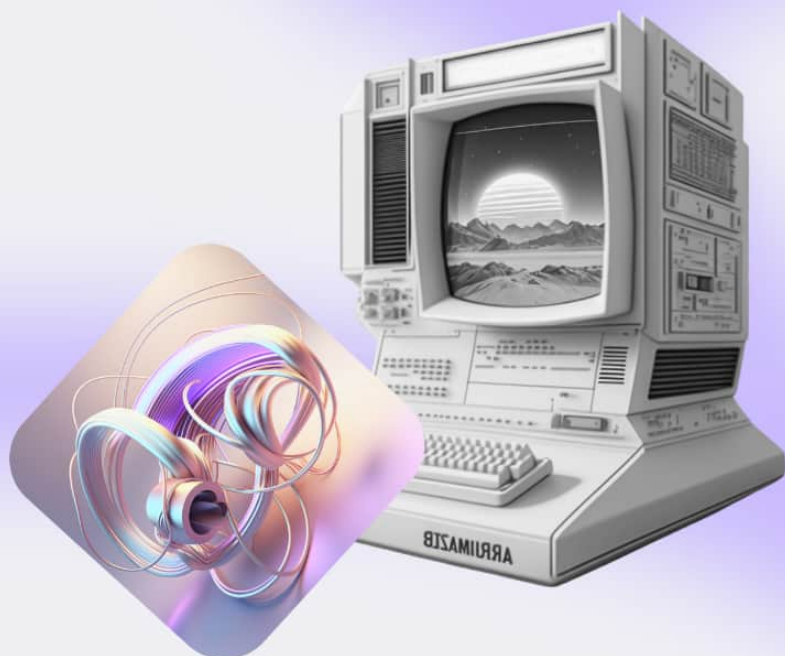


Инструментарий: Обобщения

Лекция 3



Оглавление

8. Инструментарий: Обобщения	3
В предыдущем разделе	3
8.1. Понятие обобщения	4
8.1.1. Работа без обобщений	4
8.1.2. Создание обобщения	6
8.1.3. Ограничения обобщений	8
8.1.4. Работа с обобщёнными объектами	8
8.2. Варианты обобщений	9
8.2.1. Множество параметризованных типов	9
8.2.2. Raw type (сырой тип)	10
8.2.3. Вопросы для самопроверки	12
8.2.4. Обобщённые методы	12
8.3. Ограниченные параметры типа	13
8.3.1. Ограничение «сверху»	13
8.3.2. Ограничение «снизу»	15
8.4. Выведение типов	18
8.6. Вопросы для самопроверки	20
8.7. Подстановочный символ (wildcard)	21
8.7.1. Краткое руководство по использованию подстановочных символов	26
8.7.2. Вопросы для самопроверки	27
8.8. Стирание типа и загрязнение кучи	27
8.8.1. Стирание типа	27
8.8.2. Загрязнение кучи	28
8.9. Ограничения обобщений (резюме)	29
Практическое задание	31
Термины, определения и сокращения	32

8. Инструментарий: Обобщения

В предыдущем разделе

- программные интерфейсы
- понятие и принцип работы;
- ключевое слово `implements`;
- Наследование и множественное наследование интерфейсов;
- реализация, реализации по-умолчанию;
- частичная реализация интерфейсов, адаптеры;
- анонимные классы.

В этом разделе

Смысл обобщений в программировании примерно такой же, как и в обычной разговорной речи – отсутствие деталей реализации. С точки зрения проектирования обобщения – это более сложный полиморфизм. Будет рассмотрены `diamond operator`, обобщённые методы и подстановочные символы (Wildcards). Применение обобщений для ограничений типов сверху и снизу. Выведение и стирание типов, целевые типы и загрязнение кучи.

- Обобщения;
- Wildcard;
- Diamond operation;
- Стирание типа;
- Выведение типа;
- Сырой (raw) тип;
- Целевой тип.

8.1. Понятие обобщения

8.1.1. Работа без обобщений

Обобщения – это механизм создания общего подхода к реализации одинаковых алгоритмов, но с разными данными. Обобщения – это некоторые конструкции, позволяющие работать с данными не задумываясь о том, какие именно данные лежат внутри структуры или метода (строки, числа или коты). Обобщения позволяют единообразно работать с разными типами данных.



Обобщённое программирование в Java позволяет создавать классы, интерфейсы и методы, которые могут работать с различными типами данных. В результате, код становится более универсальным и повторно используемым.

Для приведения примера поведения контейнера, который может хранить данные любого типа необходимо создать некоторый класс, который будет хранить внутри Object (любые данные Java). На примере чисел, в основном методе приложения созданы два экземпляра коробок с числами. Числа, которые сохранены в коробке желательно иметь возможность не только хранить, но и использовать, например, производить операцию сложения.

```

1 private static class Box {
2     private Object obj;
3     public Box(Object obj) {
4         this.obj = obj;
5     }
6
7     public Object getObj() {
8         return obj;
9     }
10    public void setObj(Object obj) {
11        this.obj = obj;
12    }
13    public void printInfo() {
14        System.out.printf("Box (%s): %s\n",
15            obj.getClass().getSimpleName(),
16            obj.toString());
17    }
18 }
19 public static void main(String[] args) {
20     Box b1 = new Box(20);
21     Box b2 = new Box(30);
22     System.out.println(b1.getObj() + b2.getObj());
23 }

```

Листинг 1: Контейнер, хранящий любые данные

Поскольку в коробке, с точки зрения языка, хранятся объекты, а оператор сложения для объектов не определён, следует применить операцию приведения типов. Средства языка не запрещают создать больше коробок, например, со строками, и будет производиться уже не складывание чисел, а конкатенация строк. Проблема такого способа в том, что при каждом получении данных из коробки необходимо делать приведение типов, чтобы указать какие именно в контейнере лежат данные. Данную проблему для программы возможно решить организационно, если, например, называть переменные в венгерской нотации¹.

```

1 public static void main(String[] args) {
2     Box b1 = new Box(20);
3     Box b2 = new Box(30);
4     System.out.println((Integer) b1.getObj() + (Integer) b2.getObj());
5
6     Box b3 = new Box("Hello, ");
7     Box b4 = new Box("World!");
8     System.out.println((String) b3.getObj() + (String) b4.getObj());
9 }

```

Листинг 2: Согласование типов

¹ Венгерская нотация в программировании – соглашение об именовании переменных, констант и прочих идентификаторов в коде программ. Тип переменной указывается строчной буквой перед именем переменной, например для типа `int` переменная может называться `iMyVariable`.

```

1 public static void main(String[] args) {
2     Box b1 = new Box(20);
3     Box b2 = new Box(30);
4     System.out.println((Integer) b1.getObj() + (Integer) b2.getObj());
5
6     Box b3 = new Box("Hello, ");
7     Box b4 = new Box("World!");
8     System.out.println((String) b3.getObj() + (String) b4.getObj());
9 }

```

Листинг 2: Согласование типов

Вторая проблема в том, что данные в контейнере ничем не защищены. Java не запрещает менять данные внутри контейнера, поскольку для платформы это объект. Проблему возможно решить сделав проверку instanceof перед приведением типов. Третья проблема в том, что все проблемы подобного рода проявляют себя только во время исполнения приложения, то есть у конечного пользователя перед глазами, когда разработчик ничего исправить не может.

```

1 public static void main(String[] args) {
2     Box iBox1 = new Box(20);
3     Box iBox2 = new Box(30);
4     if (iBox1.getObj() instanceof Integer && iBox2.getObj() instanceof Integer) {
5         int sum = (Integer) iBox1.getObj() + (Integer) iBox2.getObj();
6         System.out.println("sum = " + sum);
7     } else {
8         System.out.println("The contents of the boxes differ by type");
9     }
10    iBox1.setObj("sdf"); // Java: "What can go wrong here? You can do it!"
11 }

```

Листинг 3: Изменение типа хранения во время исполнения

Таким образом, в языке Java возможно создавать классы, которые могут работать с любыми типами данных, но при любом обращении к таким классам и данным необходимо делать достаточно сложные проверки.

8.1.2. Создание обобщения

— Java generics – это механизм языка, который позволяет создавать обобщенные (шаблонизированные) типы и методы в Java (особый подход к описанию данных и алгоритмов, позволяющий работать с различными типами данных без изменения внешнего описания);

— Java generics были добавлены в Java 1.5 и стали одной из наиболее важных новых функций в языке; — Java generics работают только со ссылочными типами данных;

— Java generics предоставляют безопасность типов во время компиляции, что означает, что ошибки связанные с типами данных могут быть обнаружены на этапе компиляции, а не во время выполнения программы.

Для описания обобщения в треугольных скобках пишется буква *T*, чтобы обозначить Type, Тип. На этапе описания класса невозможно сказать, какого типа данные будут лежать в переменной во время исполнения (число, строка или кот).



Если написать *T* не в треугольных скобках при описании класса, то Java будет искать реально существующий класс, который она не видит.

Таким образом указывается, что это обобщение и тип будет задаваться при создании объекта. Естественно поменять его будет нельзя, потому что Java – это язык сильной статической типизации.

```

1 private static class GBox<T> {
2     private T value;
3
4     public GBox(T value) {
5         this.value = value;
6     }
7
8     public T getValue() {
9         return value;
10    }
11    public void setValue(T value) {
12        this.value = value;
13    }
14    public void showType() {
15        System.out.printf("Type is %s, with value %s\n",
16            value.getClass().getName(), getValue());
17    }
18 }
19
20 public static void main(String[] args) {
21     GBox<String> stringBox = new GBox<>("Hello!");
22     stringBox.showType();
23     GBox<Integer> integerBox = new GBox<>(12);
24     integerBox.showType();
25 }

```

Листинг 4: Создание обобщённого контейнера

При вызове конструктора такого объекта, будет указано, что конструктор ожидает указанный ранее тип, а не *T*. При указании типа в левой части, этот тип подставляется во все места класса, где компилятор обнаружит *T*. При получении значений из `integerBox` и `stringBox` не требуется преобразование типов, `integerBox.getValue()` сразу возвращает `Integer`, а `stringBox.getValue()` – `String`. Если объект создан как `Integer`, то становится невозможно записать в него строку. При попытке написать такую строку кода, получится ошибка на этапе компиляции, то

есть обобщения отслеживают корректность используемых типов данных. По соглашению, переменные типа именуются одной буквой в верхнем регистре. Если обобщённых переменных более одной – они пишутся через запятую.

- 💡 E – элемент (Element, Entity обширно используется Java Collections);
- 💡 K – Ключ;
- 💡 N – Число;
- 💡 T – Тип;
- 💡 V – Значение;
- 💡 S, U, и т. п. — 2-й, 3-й, 4-й типы.


8.1.3. Ограничения обобщений

Обобщения накладывают на работу с собой некоторые ограничения.

1. Невозможно внутри метода обобщённого класса создать экземпляр параметризующего класса T, потому что на этапе компиляции об этом классе ничего не известно. Это ограничение возможно обойти, используя паттерн проектирования абстрактная фабрика.
2. Нельзя создавать внутри обобщения массив из обобщённого типа данных. Но всегда можно подать такой массив снаружи.
3. По причине отсутствия информации о параметризующем классе, невозможно создать статическое поле типа. Конкретный тип для параметра T становится известен только при создании объекта обобщённого класса.
4. Нельзя создать исключение обобщённого типа.

8.1.4. Работа с обобщёнными объектами

При обращении к обобщённому классу необходимо заменить параметры типа на конкретные классы или интерфейсы, например строку, целое число или кота.

 «Параметр типа» и «аргумент типа» – это два разных понятия. Когда объявляется обобщённый тип `GBox`, то `T` является параметром типа, а когда происходит обращение к обобщённому типу, передается аргумент типа, например `Integer`.

Как и любое другое объявление переменной запись вида `GBox integerBox` сам по себе не создаёт экземпляра класса `GBox`. Такой код объявляет идентификатор типа `GBox`, но сразу уточняет, что это будет коробка с целыми числами. Такой идентификатор обычно называется **параметризованным типом**.

```
1 GBox<Integer> integerBox0;
2 GBox<Integer> integerBox1 = new GBox<Integer>(1);
3 GBox<Integer> integerBox2 = new GBox<>(1);
```

Листинг 5: Способы создания обобщённых идентификаторов и объектов

Чтобы создать экземпляр класса, используется ключевое слово `new` и, в дополнение, указывается, что создаётся не просто `GBox`, а обобщённый, поэтому пишется `.` Компиляторы, начиная с Java 1.7, научились самостоятельно подставлять в треугольные скобки нужный тип (выведение типа из контекста). Если тип совпадает с аргументом типа в идентификаторе, в скобках экземпляра его можно не писать. Это называется бриллиантовый оператор.

8.2. Варианты обобщений

8.2.1. Множество параметризованных типов

Ограничений на количество параметризованных типов не накладывается. Часто можно встретить обобщения с двумя типами, например, в коллекциях, хранящих

пары ключ-значение. Также, нет ограничений на использование типов внутри угловых скобок.

```

1 private static class KVBox<K, V> {
2     private K key;
3     private V value;
4
5     public KVBox(K key, V value) {
6         this.key = key;
7         this.value = value;
8     }
9
10    public V getValue() {
11        return value;
12    }
13    public K getKey() {
14        return key;
15    }
16    public void showType() {
17        System.out.printf("Type of key is %s, key = %s, " +
18            "type of value is %s, value = %s\\n",
19            key.getClass().getName(), getKey(),
20            value.getClass().getName(), getValue());
21    }
22 }
23
24 public static void main(String[] args) {
25     KVBox<Integer, String> kvb0 = new KVBox<>(1, "Hello");
26     KVBox<String, GBox<String>> kvb1 = new KVBox<>("World", new GBox<>("Java"));
27 }

```

Листинг 6: Множество параметризованных типов

8.2.2. Raw type (сырой тип)

Сырой тип – это имя обобщённого класса или интерфейса без аргументов типа, то есть это, фактически, написание идентификатора и вызов конструктора обобщённого класса как обычного, без треугольных скобок. При использовании сырых типов, программируется поведение, которое существовало до введения обобщений в Java. Геттеры сырых типов возвращают объекты. Это логично, потому что ни на одном из этапов не указан аргумент типа.



GBox – это сырой тип обобщённого типа GBox. Однако необобщённый класс или интерфейс не являются сырыми типами.

Для совместимости со старым кодом допустимо присваивать параметризованный тип своему собственному сырому типу.

```

1 GBox<Integer> intBox = new GBox<>(1);
2 GBox box = intBox;
3
4 GBox box = new GBox(1);
5 GBox<Integer> intBox = box;
6
7 GBox<Integer> intBox0 = new GBox<>(1);
8 GBox box0 = intBox0;
9 box.setValue(4);

```

Листинг 7: Использование сырых типов

Также, если присвоить параметризованному типу сырой тип, или если попытаться вызвать обобщённый метод в сыром типе, то буквально каждое слово в программе будет с предупреждением среды разработки. Предупреждения показывают, что сырой тип обходит проверку обобщённого типа, что откладывает обнаружение потенциальной ошибки на время выполнения программы. Предупреждения среды, а на самом деле, предупреждения компилятора, обычно имеют вид, представленный на рис 1.

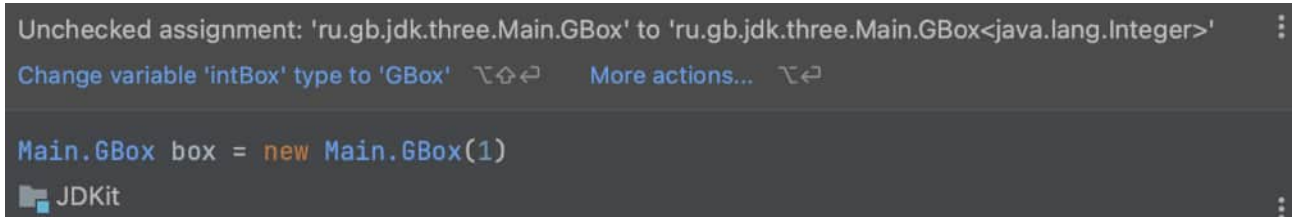


Рис. 1: Предупреждение среды о непроверенном типе

Термин «unchecked» означает непроверенные, то есть компилятор не имеет достаточного количества информации для обеспечения безопасности типов. По умолчанию этот вид предупреждений выключен, поэтому компилятор в терминале на самом деле даёт подсказку. Note: .java uses unchecked or unsafe operations. Note: Recompile with -Xlint:unchecked for details. Чтобы увидеть все «unchecked» предупреждения нужно перекомпилировать код с опцией -Xlint:unchecked.

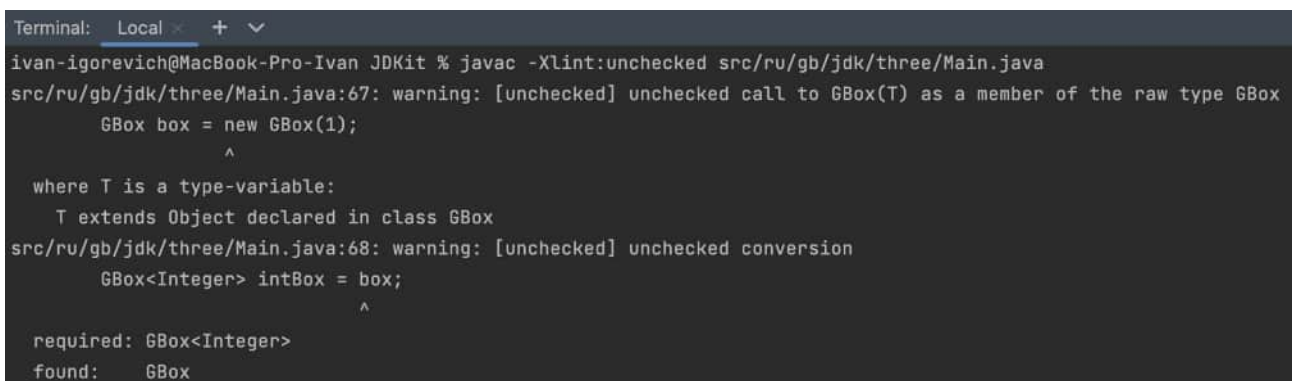


Рис. 2: Предупреждение компилятора о непроверенном типе

К предупреждениям среды в написанном коде желательно относиться внимательно, и придирчиво перепроверять код на наличие ненадёжных конструкций.

8.2.3. Вопросы для самопроверки

1. Обобщения – это способ создания общих
 - (a) классов для разных пакетов;
 - (b) алгоритмов для разных типов данных;
 - (c) библиотек для разных приложений.
2. Что именно значит буква в угловых скобках?
 - (a) Название обобщённого класса;
 - (b) имя класса, с которым будет работать обобщение;
 - (c) название параметра, используемого в обобщении.
3. Возможно ли передать обобщённым аргументом примитивный тип?
 - (a) Да;
 - (b) нет;
 - (c) только строку.

8.2.4. Обобщённые методы

Обобщённые методы синтаксически схожи с обобщёнными классами, но параметры типа относятся к методу, а не к классу. Допустимо делать обобщёнными статические и не статические методы, а также конструкторы. Синтаксис обобщённого метода включает параметры типа внутри угловых скобок, которые указываются перед возвращаемым типом.

```

1 private static <T> void setIfNull(GBox<T> box, T t) {
2     if (box.getValue() == null) {
3         box.setValue(t);
4     }
5 }
6
7 public static void main(String[] args) {
8     GBox<Integer> box = new GBox<>(null);
9     setIfNull(box, 13);
10    System.out.println(box.getValue());
11    GBox<Integer> box0 = new GBox<>(1);
12    setIfNull(box0, 13);
13    System.out.println(box0.getValue());

```

Листинг 8: Обобщённый метод

8.3. Ограниченные параметры типа

Bounded type parameters позволяют ограничить типы данных, которые могут быть использованы в качестве параметров. Использование bounded type parameters в Java является хорошей практикой, которая позволяет более точно определить используемые типы данных и обеспечивает более безопасный и читаемый код.

8.3.1. Ограничение «сверху»

В качестве примера необходимости ограничений будет использоваться уже написанная коробка. В коробке описывается операция сложения. Если сложение чисел и конкатенация строк – это понятные операции, то сложение котиков или потоков ввода-вывода не определены. Применение **bounded type parameters** позволяет более точно задавать ограничения на используемые типы данных, что помогает писать более безопасный и читаемый код. То есть, в обобщениях существует возможность ограничиться только теми типами, которые соответствуют определенным требованиям. Например, коробку предлагается сделать так, чтобы в ней хранились только числа – наследники класса `Number`. Подобное ограничение делается с помощью ограниченного параметра типа (bounded type parameter). Чтобы объявить «ограниченный сверху» параметр типа необходимо после имени параметра указать ключевое слово `extends`, а затем указать верхнюю границу (upper bound), которой в данном примере является класс `Number`.

```

1 public class Box<V extends Number> {
2     public V getValue() {
3         return value;
4     }
5     public void setValue(V value) {
6         this.value = value;
7     }
8     private V value;
9 }
10
11 private static <T extends Number> void setIfNull(BBox<T> box, T t) {
12     if (box.getValue() == null) {
13         box.setValue(t);
14     }
15 }
16
17 public static void main(String[] args) {
18     BBox<Integer> integerBBox = new BBox<>();
19     BBox<String> stringBBox = new BBox<>();
20
21     setIfNull(integerBBox, 4);
22     setIfNull(stringBBox, "hello");
23 }

```

Листинг 9: Ограничение параметра типа «сверху»

В рассматриваемом примере типы, которые можно использовать в параметризованных классах BBox, ограничены наследниками класса Number. Если попытаться создать переменную с типом, например, Box, то возникнет ошибка компиляции. Аналогичным образом создаются обобщённые методы с ограничением.

```

1 class Bird{}
2 interface Animal{}
3 interface Man{}
4 class CBox<T extends Bird & Animal & Man> {
5     // ...
6 }

```

Листинг 10: Множественные ограничения

Также возможно задать несколько границ. При этом, важно помнить, что также, как и в объявлении классов, возможен только один класс-наследник и он указывается первым. Все остальные границы могут быть только интерфейсами и указываются через знак амперсанда.



В языке Java, несмотря на строгость типизации, возможно присвоить идентификатору 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. В

объектно-ориентированной терминологии это называется связью «является» («is a»).

Предположим, что существует метод, описанный с generic параметром.


```

1 private static void boxTest(GBox<Number> n) { /* ... */ }
2
3 public static void main(String[] args) {
4     boxTest(new GBox<Number>(10));
5     boxTest(new GBox<Integer>(1)); // compile error
6     boxTest(new GBox<Float>(1.0f)); // compile error

```

Листинг 11: Generic параметр метода

На первый взгляд кажется, что в метод возможно передать Box или Box, но нельзя, так как Box и Box не являются потомками Box.

 Это частое недопонимание принципов работы обобщений, и это важно знать. Наследование не работает в Java generics так, как оно работает в обычной Java.

Идентификатор коробки с Number не может в себе хранить коробку с Integer. Обобщение защищает от попыток положить в коробку, например, строк – не строку. То есть, предположим, в коробку кладётся Integer, как наследник Number, а затем, например Float, получится путаница. Из приведённого примера возможно сделать вывод о том, что если методу с таким параметром передать коробку с Integer – он не будет работать. Чтобы допустить передачу таких контейнеров, в аргументе следует указать что в параметре возможен любой тип, являющийся наследником Number, то есть использовать маску *. Ограничивать такую маску возможно как сверху так и снизу. Таким образом, обобщения защищают самих себя от путаницы и не дают складывать в одни и те же контейнеры разные типы данных. Поскольку коробку с чем то ещё, кроме Number и его наследников создавать нельзя, маскирование при вызове метода будет избыточно. На самом деле обобщения – это так называемый синтаксический сахар. То есть, когда в коде используется обобщение, во время компиляции произойдёт так называемое «стирание», и все обобщённые типы данных преобразуются в Object, соответствующие проверки и приведения типов.

8.3.2. Ограничение «снизу»

Ограничение типов возможно вводить как сверху, так и снизу. На примере обобщённых методов. Такие методы необходимы, когда требуется объединить несколько похожих, но всё же разных типов данных. Если подать на вход обобщённого метода два типа – Integer и Float в итоге для работы будет выбран ближайший старший для них обоих – Number.

```

1 private static <T extends Number> boolean compare(T src, T dst) {
2     return src.equals(dst);
3 }
4
5 public static void main(String[] args) {
6     System.out.println(compare(1, 1.0f));
7     System.out.println(compare(1.0f, 1.0f));
8     System.out.println(compare(1, 1));

```

Листинг 12: Пример обобщённого метода

Например, даны два списка – один с Integer другой с Number, и требуется написать метод, который будет перекидывать числа из одного списка в другой. Для совершения этого действия описан метод, при работе которого возможны два сценария – копировать элементы Number в список Integer или элементы Integer в список Number.

```

1 public static void copyTo(ArrayList src, ArrayList dst) {
2     for (Object o : src) dst.add(o);
3 }
4
5 public static void main(String[] args) {
6     ArrayList<Integer> ial = new ArrayList<>(Arrays.asList(1, 2, 3));
7     ArrayList<Number> nal = new ArrayList<>(Arrays.asList(1f, 2, 3.0));
8     System.out.println(ial);
9     System.out.println(nal);
10    copyTo(ial, nal);
11    System.out.println(nal);
12    copyTo(nal, ial);
13    System.out.println(ial);

```

Листинг 13: Копирование чисел в списки

Правильным рабочим сценарием будет только второй – копирование более точного типа в более общий список. Java при компиляции пропустит в работу оба сценария, но действительно работающим всё равно остаётся только один. Для примера, описаны два класса: «животное» и наследник животного – «кот». На их основе создаются обобщённые списки и вызывается обобщённый метод копирования списков.


```

1 public static void copyTo(ArrayList src, ArrayList dst) {
2     for (Object o : src) dst.add(o);
3 }
4
5 private static class Animal {
6     protected String name;
7     protected Animal() { this.name = "Animal"; }
8     @Override public String toString() { return name; }
9 }
10 private static class Cat extends Animal {
11     protected Cat() { this.name = "Cat"; }
12 }
13
14 public static void main(String[] args) {
15     ArrayList<Cat> cats = new ArrayList<>(Arrays.asList(new Cat()));
16     ArrayList<Animal> animals = new ArrayList<>(Arrays.asList(new Animal()));
17     copyTo(animals, cats);
18     System.out.println(cats);

```

Листинг 14: Копирование списков с «животными» и «котами»

При компиляции и исполнении ошибок не возникло. К классу кота добавляется метод голос и совершается попытка вызвать голос у объекта из списка.



Рис. 3: Проблема объединения списков

При попытке вызвать у более общего «животного» метод более частного «кота» выбрасывается исключение о невозможности приведения типов. Для того, чтобы избежать подобных проблем, необходимо описать метод таким образом, чтобы он работал только с каким-то одним типом и списки будут и в цикле тоже будут перебираться элементы типа T. Если более не уточнять, получится, что в список котов возможно класть только котов, а в список животных класть только животных. Но кот – это наследник животного, его присутствие в списке животных уместно. Получается, что источником может быть список из заданного типа или его наследников, а приёмником – тип или его родители, и далее метод, виртуальная машина и другие механизмы сами разбираются, кто подходит под эти параметры. При таком описании метода, неверные варианты будут отсекаются на этапе компиляции.

```

1 public static <T> void copyTo (
2     ArrayList<? extends T> src, ArrayList<? super T> dst) {
3     for (T o : src) {
4         dst.add(o);
5     }
6 }

```

8.4. Выведение типов

Алгоритм вывода типов определяет типы аргументов, а также, если это применимо, тип, в который присваивается результат или в котором возвращается результат.



Выведение типов – это возможность компилятора автоматически определять аргументы типа на основе контекста.

Алгоритм работает от наиболее общего типа (Object) к наиболее точному, подходящему к данной ситуации. Например, добавив к коту реализацию интерфейса Serializable и написав метод, работающий с одним и только одним типом данных T будет создана ситуация, в которой никакие аргументы не связаны наследованием.

```

1 private static class Cat extends Animal implements Serializable {
2     protected Cat() { this.name = "Cat"; }
3     public void voice(){ System.out.println("meow"); }
4 }
5 private static <T> T pick(T first, T second) { return second; }
6
7 public static void main(String[] args) {
8     Serializable se1 = pick("d", new Cat());
9     Serializable se2 = pick("d", new ArrayList<String>());

```

Листинг 15: Ситуация, в которой работает вывод типов

В таком методе вывод типов определяет, что вторые аргументы метода pick, а именно Cat и ArrayList, передаваемые в метод имеют тип Serializable, но этого недостаточно, потому что первый аргумент тоже должен быть того же типа. Удачно, что строка – это тоже Serializable. В описании обобщённых методов, вывод типа делает возможным вызов обобщённого метода так, будто это обычный метод, без указания типа в угловых скобках.

```

1 public class App {
2     public static <U> void addBox(U u, List<Box<U>> boxes) {
3         Box<U> box = new Box<>();
4         box.setValue(u);
5         boxes.add(box);
6     }
7
8     public static void main( String[] args ) {
9         ArrayList<Box<Cat>> catsInBoxes = new ArrayList<>();
10        App.<Cat>addBox(new Cat("Kusya"), catsInBoxes);
11        addBox(new Cat("Kusya"), catsInBoxes);
12        addBox(new Cat("Murka"), catsInBoxes);
13        printBoxes(catsInBoxes);
14    }

```

Листинг 16: Выведение типов в обобщённых методах

Очевидно, что в листинге 16 обобщённый метод `addBox()` объявляет один параметр типа `U`. В большинстве случаев компилятор Java может вывести параметры типа вызова обобщённого метода, в результате чаще всего вовсе не обязательно их указывать. Чтобы вызвать обобщённый метод `addBox()`, возможно указать параметры типа (строка 10) либо опустить их, тогда компилятор языка автоматически выведет тип `Cat` из аргументов метода при вызове (строка 11).

Выведение типа при создании экземпляра обобщённого класса позволяет заменить аргументы типа, необходимые для вызова конструктора обобщённого класса пустым множеством параметров типа (пустые треугольные скобки, бриллиантовая операция), так как компилятор может вывести аргументы типа из контекста.

Очевидно, что конструкторы могут быть обобщёнными как в обобщённых, так и в необобщённых классах.

```

1 public class Box<T> {
2     <U> Box(U u){
3         // ...
4     }
5     public T getValue() {
6         return value;
7     }
8     public void setValue(T value) {
9         this.value = value;
10    }
11    private T value;
12 }
13
14 public static void main(String[] args) {
15     Box<Cat> box = new Box<Cat>("Some message");

```

Листинг 17: Обобщённый конструктор

Например, возможно явно указать, что у коробки будет обобщённый аргумент «кот», а у конструктора – какой-то другой аргумент, например, строка или число. Компилятор выведет тип `String` для формального параметра `U`, так как фактически переданный аргумент является экземпляром класса `String`.



Целевой тип выражения – это тип данных, который компилятор Java ожидает в зависимости от того, в каком месте находится выражение.

Например, как в методе `emptyBox()` (листинг 18, строка 12). В основном методе инициализация ожидает экземпляр `Box`. Этот тип данных является целевым типом. Поскольку метод `emptyBox()` возвращает значение обобщённого типа `Box`.

```

1 public class TBox<T> {
2     public static final TBox EMPTY_BOX = new TBox<>();
3
4     public T getValue() { return value; }
5
6     public void setValue(T value) {
7         this.value = value;
8     }
9
10    private T value;
11
12    static <T> TBox<T> emptyBox(){
13        return (TBox<T>) EMPTY_BOX;
14    }
15 }
16
17 public static void main( String[] args ) {
18     TBox<String> box = TBox.emptyBox();
19 }

```

Листинг 18: Целевые типы

8.6. Вопросы для самопроверки

1. Что из следующего является недопустимым?

- (a) `ArrayList al1 = new ArrayList();`
- (b) `ArrayList al2 = new ArrayList();`
- (c) `ArrayList al3 = new ArrayList();`
- (d) Всё допустимо.

2. параметры метода `ArrayList src`, `ArrayList dst` вызов метода `copyTo(cats, animals);`
Какой тип данных будет взят в качестве `T`?

- (a) `Animal`;
- (b) `Cat`;
- (c) `Object`.

8.7. Подстановочный символ (wildcard)

В обобщённом коде знак вопроса, называемый подстановочным символом, означает неизвестный тип. Подстановочный символ может использоваться в разных ситуациях: как параметр типа, поля, локальной переменной, иногда в качестве возвращаемого типа. Подстановочный символ никогда не используется (рис 4) в качестве аргумента типа для вызова обобщённого метода, создания экземпляра обобщённого класса или супертипа.

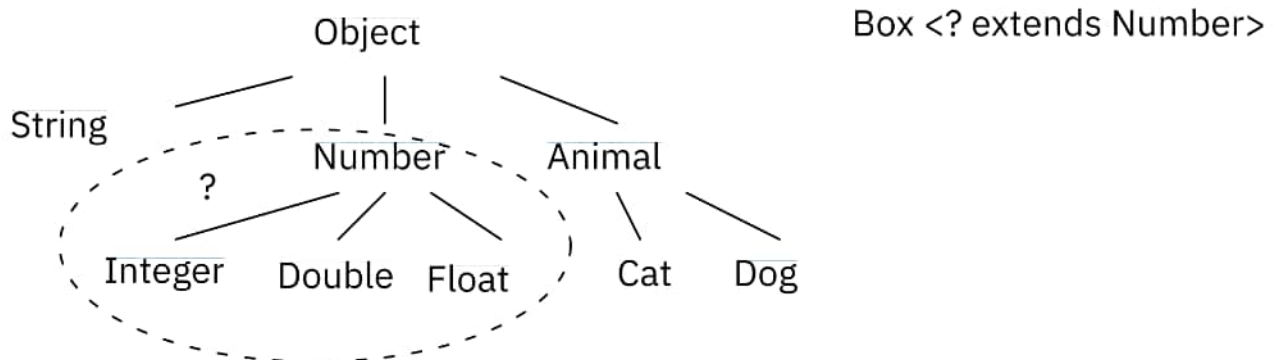
```

119 |
120 |     private static <?> T method(T first, T second) { return second; }
121 |     public static void main(String[] args) {
122 |         TBox<String> b = new TBox<?>();
123 |

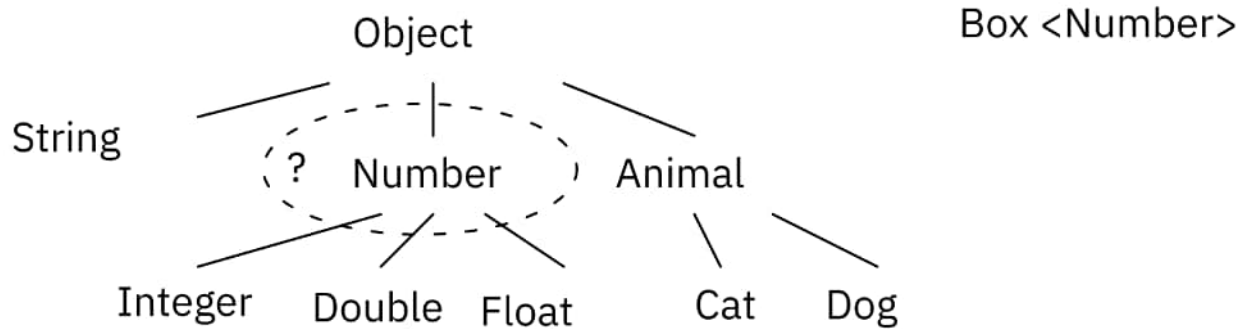
```

Рис. 4: Ошибки при работе с подстановочным символом

Передаваемые типы для уточнения, возможно **ограничивать** сверху и снизу. Ограниченный сверху подстановочный символ применяется, чтобы **ослабить ограничения** переменной.



Чтобы написать метод, который работает с коробками, в которых содержится `Number` и дочерние от `Number` типы, например `Integer`, `Double` и другие, необходимо указать `Box`.



Пример кода будет приводиться с классами животного и наследников.

```

1 private static class Animal {
2     protected String name;
3     protected Animal(String name) { this.name = name; }
4     @Override public String toString() {
5         return this.getClass().getSimpleName() + " with name " + name;
6     }
7 }
8 private static class Cat extends Animal {
9     protected Cat(String name) { super(name); }
10 }
11 private static class Dog extends Animal {
12     protected Dog(String name) { super(name); }
13 }

```

Листинг 19: Подготовка классов для демонстрации

Внутри метода, выводящего информацию о содержимом коробки присутствует ограниченный сверху подстановочный символ , где вместо Animal может быть любой тип.

```

1 public static class TBox<T> {
2     public static final TBox EMPTY_BOX = new TBox<>();
3     private T value;
4
5     public T getValue() { return value; }
6     public void setValue(T value) { this.value = value; }
7     static <T> TBox<T> emptyBox() {
8         return (TBox<T>) EMPTY_BOX;
9     }
10    @Override public String toString() {
11        return value.toString();
12    }
13 }
14 static void printInfo(TBox<? extends Animal> animalInBox) {
15     System.out.println("Information about animal: " + animalInBox);
16 }

```

Листинг 20: Использование ограниченного подстановочного символа

Создав соответствующие объекты, положив их в коробки и запустив код возможно наблюдать, что коробка вмещает как животных, так и наследников животного, чего не произошло бы при использовании в параметре типа животного без подстановочного символа.

💡 Если просто использовать подстановочный символ `TBox`, то получится подстановочный символ без ограничений. `Box` означает коробку с неизвестным содержимым (неизвестным типом).

Такой синтаксис существует для того, чтобы продолжать использовать обобщённый тип без уточнения типа, как следствие, без использования обобщённой функциональности. Неограниченный подстановочный символ полезен, если нужен метод, который может быть реализован с помощью функциональности класса `Object`. Когда код использует методы обобщённого класса, которые не зависят от параметра типа.

💡 В программах, использующих `Reflection API` конструкция `Class` используется чаще других конструкций, потому что большинство методов объекта `Class` не зависят от расположенного внутри типа.

Например, метод `printInfo()` (строка 14) из листинга 20, не использует никаких методов животного, цель метода – вывод в консоль информации об объекте в коробке любого типа, поэтому в параметре метода можно заменить «коробку с наследниками животного» на «коробку с чем угодно», ведь в методе будет использоваться только метод коробки `toString()`.

```
1 static void printInfo(TBox<?> animalInBox) {
2     System.out.println("Information about animal: " + animalInBox);
3 }
```

Листинг 21: Исправление метода вывода информации на экран

🔥 Важно запомнить, что `Box` и `Box` – это не одно и то же.


```

127
128 static void printInfo(TBox<Object> animalInBox){
129     System.out.println("Information about animal: " + animalInBox);
130 }
131
132 public static void main(String[] args) {
133     TBox<Cat> catInBox = TBox.emptyBox();
134     catInBox.setValue(new Cat("Vasya"));
135     printInfo(catInBox);
136
137     TBox<Dog> dogInBox = TBox.emptyBox();
138     dogInBox.setValue(new Dog("Boby"));
139     printInfo(dogInBox);

```

Required type: TBox <Object>
 Provided: TBox <Cat>
 Change 1st parameter of method 'printInfo' from 'TBox<Object>' to 'TBox<Cat>' More actions...
 Main.TBox<Main.Cat> catInBox = TBox.emptyBox()
 JDKit

Рис. 5: Ошибка компиляции при использовании параметра типа Object

Ограниченный снизу подстановочный символ ограничивает неизвестный тип так, чтобы он был либо указанным типом, либо одним из его предков. В обобщённых конструкциях возможно указать либо только верхнюю границу для подстановочного символа, либо только нижнюю.

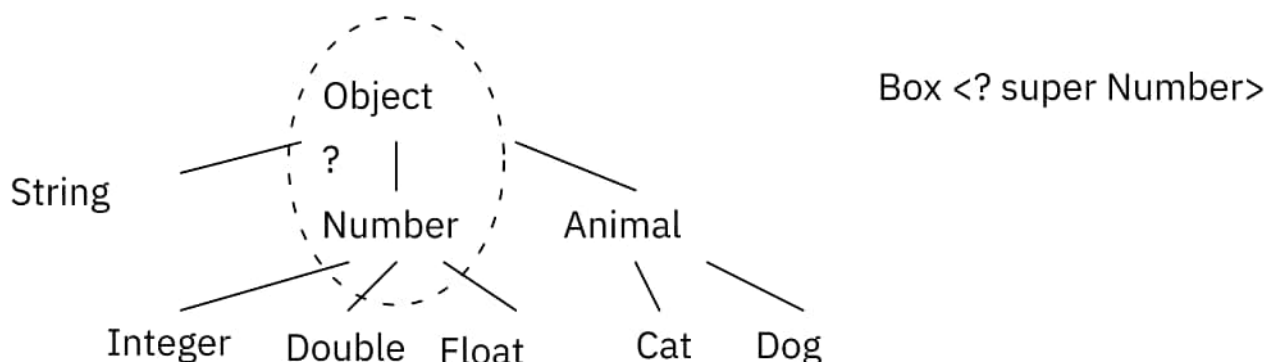
```

127
128 static void printInfo(TBox<? extends Number & ? super Integer> animalInBox){
129     System.out.println("Information about animal: " + animalInBox);
130 }
131

```

'>' or ';' expected
 Identifier expected
 java.lang

То есть, если написать метод printInfo(), с параметром коробки и обобщённым аргументом не `Number`, а `Integer`, то код также не будет работать, потому что метод будет ожидать не «животное и наследников», а «животное и родителей», то есть `Object`.



Обобщённые классы или интерфейсы связаны не только из-за связи между их типами. С обычными, необобщёнными классами, наследование работает по правилу подчинённых типов: класс `Cat` является подклассом класса `Animal`, и расширяет его.

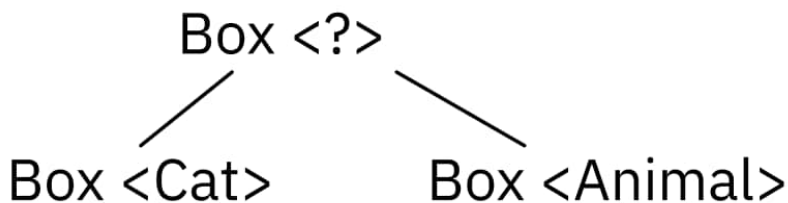

```

1 private static class Animal {
2     protected String name;
3     protected Animal(String name) { this.name = name; }
4     @Override public String toString() {
5         return this.getClass().getSimpleName() + " with name " + name;
6     }
7 }
8 private static class Cat extends Animal {
9     protected Cat(String name) { super(name); }
10 }
11
12 public static void main(String[] args) {
13     Cat cat = new Cat("Vasya");
14
15     Animal animal = cat;
16
17     TBox<Cat> catInBox = new TBox<>();
18     TBox<Animal> animalInBox = catInBox; // Incompatible types

```

Листинг 22: Наследование необобщённых классов

Данное правило не работает для обобщённых типов. Несмотря на то, что Cat является подтипом Animal, Box не является подтипом Box. Общим предком для Box и Box является Box.



```

1 TBox<? extends Cat> catInBox = new TBox<>();
2 TBox<? extends Animal> animalInBox = catInBox; // OK

```

Листинг 23: Наследование в параметрах типа через подстановочный символ

В некоторых случаях компилятор может вывести тип подстановочного символа. Коробка может быть определена как Box, но при вычислении выражения компилятор выведет конкретный тип из кода, такой сценарий называется **захватом подстановочного символа**. В большинстве случаев нет нужды беспокоиться о захвате подстановочного символа, кроме случаев, когда в сообщении об ошибке появляется фраза *capture of*. В примере, приведённом в листинге 24 компилятор обрабатывает параметр коробки как тип Object. Когда вызывается метод внутри `testError()`, компилятор не может подтвердить тип объекта, который будет присутствовать внутри коробки и генерирует ошибку. Обобщения были добавлены в Java именно для этого – чтобы усилить безопасность типов на этапе компиляции.

```

1 static void testError(Box<?> box){
2     box.setValue(box.getValue()); // capture of ?
3 }

```

Листинг 24: Ошибка захвата подстановочного символа

Когда есть уверенность в том, что операция безопасна, ошибку возможно исправить написав приватный вспомогательный метод (в англоязычной литературе private helper method), захватывающий подстановочный символ.

```

1 private static <T> void testErrorHelper(TBox<T> box) {
2     box.setValue(box.getValue());
3 }
4
5 static void testError(TBox<?> box) {
6     testErrorHelper(box);
7 }

```

Листинг 25: Использование вспомогательного метода

8.7.1. Краткое руководство по использованию подстановочных символов

Входная переменная. Предоставляет данные для кода. Для метода copy(src, dst) параметр src предоставляет данные для копирования, поэтому он считается входной переменной. Выходная переменная. Содержит данные для использования в другом месте. В примере copy(src, dst) параметр dst принимает данные и будет считаться выходной переменной.

1. Входная переменная определяется с ограничением сверху.
2. Выходная переменная определяется с ограничением снизу.
3. Если ко входной переменной можно обращаться только как к Object – неограниченный подстановочный символ.
4. Если переменная должна использоваться как входная и как выходная одновременно, НЕ использовать подстановочный символ.
5. Не использовать подстановочные символы в возвращаемых типах.

8.7.2. Вопросы для самопроверки

1. Ограниченный снизу подстановочный символ позволяет передать в качестве аргумента типа

- (a) только родителей типа;
- (b) только наследников типа;
- (c) сам тип и его родителей;
- (d) сам тип и его наследников.

2. Конструкция

- (a) не скомпилируется;
- (b) не имеет смысла;
- (c) не позволит ничего передать в аргумент типа;
- (d) не является чем-то необычным.

8.8. Стирание типа и загрязнение кучи

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 Java применяет стирание типа.

8.8.1. Стирание типа

- Механизм стирания типа фактически заменяет все параметры типа в обобщённых типах их границами или `Object`, если параметры типа не ограничены.
- Сгенерированный байткод содержит только обычные классы, интерфейсы и методы.
- Вставляет явное приведение типов где необходимо.

— Генерирует связующие методы, чтобы сохранить полиморфизм в расширенных обобщённых типах.

— Гарантирует, что никакие новые классы не будут созданы для параметризованных типов, следовательно обобщения не приводят к накладным расходам во время исполнения.

Компилятор Java также стирает параметры типа обобщённых методов. Обобщённый метод в котором используется неограниченный тип будет заменён компилятором на Object.

Аналогично классам для методов происходит стирание типа при расширении ключевым словом `extends` – параметр в угловых скобках заменяется на максимально возможного родителя.

```

1 private static <T> void setIfNull(TBox<T> box, T t) {
2     if (box.getValue() == null) {
3         box.setValue(t);
4     }
5 }
6 // ... both methods have same erasure
7 private static void setIfNull(TBox<Object> box, Object t) {
8     if (box.getValue() == null) {
9         box.setValue(t);
10    }
11 }

```

Стирание типа имеет последствия, связанные с произвольным количеством параметров (varargs).

Материализуемые типы – это типы, информация о которых полностью доступна во время выполнения, такие как примитивы, необобщённые типы, сырые типы, обращения к неограниченным подстановочным символам. Нематериализуемые типы – это типы, информация о которых удаляется во время компиляции стиранием типов, например, обращения к обобщённым типам, которые не объявлены с помощью неограниченных подстановочных символов. Во время выполнения о нематериализуемых типах нет всей информации. Виртуальная машина Java не может узнать разницу между нематериализуемыми типами во время выполнения.

8.8.2. Загрязнение кучи

Загрязнение кучи (heap pollution) возникает, когда переменная параметризованного типа ссылается на объект, который не является параметризованным типом. Такая ситуация возникает, если программа выполнила операцию, генерирующую

предупреждение `unchecked warning` во время компиляции. Предупреждение `unchecked warning` генерируется, если правильность операции, в которую вовлечён параметризованный тип (например, приведение типа или вызов метода) не может быть проверена. Если компилируются различные части кода отдельно, то становится трудно определить потенциальную угрозу загрязнения кучи.

8.9. Ограничения обобщений (резюме)

В этом разделе приводится некоторое резюме вышесказанного в части наиболее частых ошибок при работе с обобщениями. Нельзя использовать при создании экземпляра примитивы. Выход из ситуации – использование классов-обёрток.

```
155
156     TBox<int> box = new TBox<>();
157
```

Нельзя создавать экземпляры параметров типа. В качестве выхода из ситуации – передавать вновь созданный объект в обобщённые методы в качестве параметра.

```
149
150  @   static <T> void add(Box<T> box) {
151      T t = new T();
152      // ...
153      box.setValue(t);
154  }
155
```

Статические поля класса являются общими для всех объектов этого класса, поэтому статические поля с типом параметра типа запрещены. Так как статическое поле является общим для коробки с животным, коробки с котом и коробки с собакой, то какого типа это поле? Также запрещено использовать приведение типа к параметризованному типу, если он не использует неограниченный подстановочный символ.

```

149 |
150 | public class SBox<T> {
151 |     private static T value;
152 |
153 |     // ...
154 | }
155 |
156 |

```

Static declarations in inner classes are not supported at language level '11' :
 Upgrade JDK to 16+ ↗ ↗ More actions... ↗
 ru.gb.jdk.three.Main.SBox<T>
 private static T value
 JDKit

Запрещено использовать приведения типов для обобщённых объектов. Так как компилятор стирает все параметры типа из обобщённого кода, то нельзя проверить во время выполнения, какой параметризованный тип используется для обобщённого типа.

```

163 |
164 | TBox<Integer> box1 = new TBox<>();
165 | TBox<Number> box2 = (TBox<Number>) box1;
166 |

```

Запрещено создавать массивы параметризованных типов.

```

159 |
160 | Object[] stringLists = new TBox<String>[];
161 | // compilation error, but let's say it's possible
162 | stringLists[0] = new TBox<String>(); // OK
163 | stringLists[1] = new TBox<Integer>(); // here should be
164 | // an ArrayStoreException exception,
165 | // but the runtime cannot notice it.
166 |

```

Обобщённый класс не может расширять класс Throwable напрямую или опосредовано. Метод не может ловить (catch) экземпляр параметра типа. Однако можно использовать параметр типа в throws.

```

157 // Extends Throwable non-direct
158 class MathException<T> extends Exception { /* ... */ } // compilation error
159
160 // Extends Throwable directly
161 class QueueFullException<T> extends Throwable { /* ... */ } // compilation error
162
163 @
164 public static <T extends Exception, J> void execute(TBox<J> box) {
165     try {
166         J j = box.getValue();
167         // ...
168     } catch (T e) { // compilation error
169     }
170 }
171
172 class Parser<T> extends Exception {
173     public void parse(File file) throws T { // OK
174         // ...
175     }
176 }

```

Класс не может иметь два перегруженных метода, которые будут иметь одинаковую сигнатуру после стирания типов. Такой код не скомпилируется.

```

138 @
139 private static <T> void setIfNull(TBox<T> box, T t) {
140     if (box.getValue() == null) {
141         box.setValue(t);
142     }
143 }
144
145 @
146 private static void setIfNull(TBox<Object> box, Object t) {
147     if (box.getValue() == null) {
148         box.setValue(t);
149     }
150 }

```

Fix method 'setIfNull' parameters with bounded wildcard: `<? extends T>` More actions...

```

ru.gb.jdk.three.Main
private static <T> void setIfNull(
    @NotNull Main.TBox<T> box,
    T t
)
JDK11

```

Практическое задание

1. Написать метод, который меняет два элемента массива местами (массив может быть любого ссылочного типа);

2. Большая задача:

- Есть классы Fruit -> Apple, Orange; (больше не надо);
- Класс Box в который можно складывать фрукты, коробки условно сортируются по типу фрукта, поэтому в одну коробку нельзя сложить и яблоки, и апельсины; Для хранения фруктов внутри коробки можете использовать ArrayList;
- Сделать метод `getWeight()` который высчитывает вес коробки, зная количество фруктов и вес одного фрукта (вес яблока – 1.0f, апельсина – 1.5f, не важно в каких единицах);
- Внутри класса коробки сделать метод `compare()`, который позволяет сравнить текущую коробку с той, которую подадут в `compare()` в качестве параметра, `true` – если их веса равны, `false` в противном случае (коробки с яблоками возможно сравнивать с коробками с апельсинами);
- Написать метод, который позволяет пересыпать фрукты из текущей коробки в другую коробку (при этом, нельзя яблоки высыпать в коробку с апельсинами), соответственно, в текущей коробке фруктов не остается, а в другую перекидываются объекты, которые были в этой коробке.

Термины, определения и сокращения

Diamond operation автоматизация подстановки типа, пустые угловые скобки, вынуждающие компилятор вывести тип из контекста.

Wildcard в обобщённом коде знак вопроса, называемый подстановочным символом, означает неизвестный тип. Подстановочный символ может использоваться в разных ситуациях: как параметр типа, поля, локальной переменной, иногда в качестве возвращаемого типа.

Выведение типа это возможность компилятора автоматически определять аргументы типа на основе контекста

Обобщения особый подход к описанию данных и алгоритмов, позволяющий работать с различными типами данных без изменения внешнего описания.

Стирание типа подготовка написанного обобщённого кода к компиляции таким образом, чтобы в байт-коде не было обобщённых конструкций.

Сырой (raw) тип это имя обобщённого класса или интерфейса без аргументов типа, то есть это, фактически, написание идентификатора и вызов конструктора обобщённого класса как обычного, без треугольных скобок.

Целевой тип это тип данных, который компилятор Java ожидает в зависимости от того, в каком месте находится выражение.