**Обобщения –** это параметризованные типы. Такие типы позволяют объявлять классы, интерфейсы и методы, где тип данных указан в виде параметра.

Классы, интерфесы или методы, оперирующие параметризованными типами, называются обобщенными.

**Синтаксис:** class Clasname<T>{}

T обозначает имя параметра типа. Везде где пишем T будет подставляться конкретный тип, который мы передадим.

**Плюсы обобщений:**

* Безопасность типов (идет проверка передаваемого типа. Нельзя в Gen<Integer> присвоить Gen<String>);
* Не требуется приведение типов, как при использовании Objects;
* Многие ошибки несоответствия типов могут быть обнаружены на этапе компиляции, а не во время выполнения как с Object.

Компилятор Java не создает разные версии обобщенного класса. Вместо этого компилятор удаляет все сведения об обобщенных типах, выполняя необходимые операции приведения типов, чтобы сделать поведение прикладного кода таким, будто создана конкретная версия обобщенного класса.

Удаление сведений об обобщенных типах при компиляции называется **Стиранием типов** (Type Erasure). Это означает, что параметры типа сначала заменяются их ограничивающим типом, которым является тип Object, если явного ограничения не указано. Затем выполняется требуемое приведение типов, определяемое аргументами типов, для обеспечения совместимости с типами, указанными в этих аргументах. Компилятор также обеспечивает эту совместимость типов.

Обобщения действуют только со ссылочными типами. Из-за стирания типов, все преобразуется к Object. Примитивы не наследуют объект.

**Ограниченные типы**

<T extends суперкласс> - указывает, что можно передавать только тип суперкласса или его подклассы. Суперкласс объявляет верхнюю границу типов.

<T super подкласс> - указывает, что можно передавать только тип подкласса или его суперклассы.

Ограничение может включать в себя так же тип одного или нескольких интерфейсов.

class Gen < T extends MyClass & Myinterface >

**Ковариантность, контравариантность и инвариантность**

Допустим *кошка* подтип *животное*

**Ковариантность** – Множество<Кошки> это подтип Множество<Животные> и можно выполнить присваивание

*Множество<Животные>  = Множество<Кошки>*

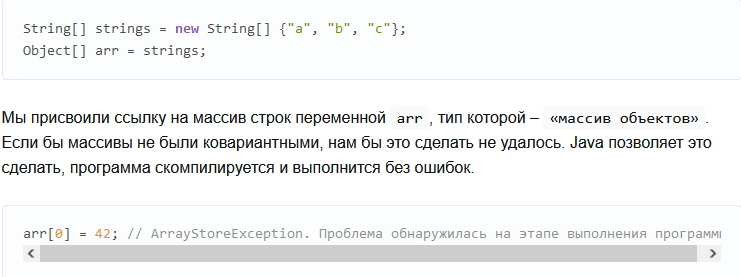
**Контрвариантность**

*Множество<Кошки> = Множество<Животные>*

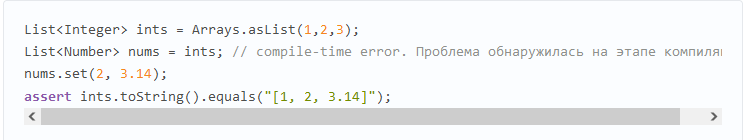
**Инвариантность** — отсутствие наследования между производными типами.

Множество<Кошки> не является подтипом Множество<Животные> и Множество<Животные> не является подтипом Множество<Кошки>.

Массивы в Java ковариантны



**«Дженерики» инвариантны**



Если взять список целых чисел, то он не будет являться ни подтипом типа Number, ни каким-либо другим подтипом. Он является только подтипом самого себя. То есть List <Integer> — это List<Integer> и ничего больше.

**Метасимвольный аргумент (Wildcards)**

Метасимвольный аргумент – это обозначение неизвестного типа, который используется в обобщенных классах.

ClassName<?> instance

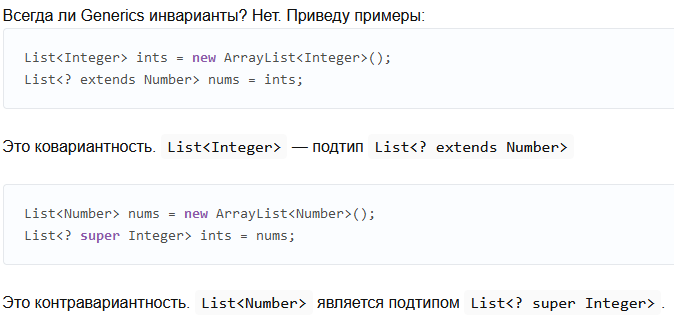
здесь

* ClassName – имя обобщенного класса;
* **?** – обозначение метасимвольного аргумента, который означает любой тип;
* instance – переменная-экземпляр обобщенного класса, который может оперировать любым типом.

К ним также можно применять ограничения extends и super.

В List<?> нельзя добавить никаких элементов (кроме null), и прочитать из него можно только Object. По сути <?> это <? extends Object>. Тип неизвестен на этапе компиляции, поэтому мы не знаем, какие объекты находятся в списке, и не можем туда ничего добавить. Так же и при чтении, мы не знаем что там внутри, но точно знаем что это Object.

**Wildcards неинвариантны.**



**PECS (Producer Extends Consumer Super)**

* Если мы объявили wildcard с extends, то это producer. Из него мы можем только получать элементы. Мы не можем записать ничего кроме null.

List<? extends T> prod, будет означать что мы сможем получать из списка элементы типа T.

* Если же мы объявили wildcard с super — то это consumer. Он только принимает, а предоставить ничего не может (кроме Object).

List<? super T> cons, будет означать, что мы сможем записать в список все, что является T.

**Обобщенные методы**

Параметр типа объявляется до типа, возвращаемого методом

<список\_параметров\_типа> возвращаемый\_тип имя\_метода (список\_параметров) {}

Static <T extends Comparable<T>, V extends T> Boolean isIn(T x, V[] y) {}

Конструкторы также могут быть обобщенными , даже если их классы таковыми не являются.

**Обобщенный интерфейс**

Обобщенный интерфейс объявляется таким же образом, как

и обобщенный класс:

*interface имя\_интерфейса<список\_параметров\_типа>*

interface MinMax<T extends Comporable<T>>{}

Синтаксис класса, реализующего обобщенный интерфейс

*Class имя\_класса<список\_параметров\_типа> implements имя\_интерфейса<список\_аргументов\_типа>{}*

class MyClass<T extends Comparable<T>> implements MinMax<T>{}

Как правило, класс, реализующий обобщенный интерфейс должен быть так же обобщенным, если он принимает параметр типа, передаваемый далее интерфейсу.

Если класс реализует конкретный тип обобщенного интерфейса, то реализующий класс не должен быть обобщенным:

Class MyClass implements MinMax<Integer> {}

**Преимущества обобщенного интерфейса**:

* может быть реализован для разных типов данных;
* позволяет наложить ограничения на типы данных, для которых он может быть реализован.

**Сырые типы**

Поддержка обобщений отсутствовала до версии JDK 5. Чтобы сохранить совместимость со старым кодом, в Java допускается применять обобщенные классы без указания типа параметра. Этот тип называется базовым или сырым (raw types). При этом утрачивается типовая безопасность.

Можно будет передавать объект любого класса, теряется проверка типов на этапе компиляции.

Базовые типы не обеспечивают нужной безопасности. Это означает, что пере­менной базового типа можно присвоить ссылку на любой тип объектов параметризованного класса и наоборот.

*Gen<Integer> iOb = new Gen<Integer>(88);*

*Gen raw = new Gen(new Double(98.6));*

Следующий код скомпилируется, но нарушает типовую безопасность (не выполняется проверка типов):

*Raw = iOb;*

*Double d = (Double) raw.getob(); // ОШИБКА во время выполнения!*

Применение базовых типов следует ограничить теми случаями, где нужна совместимость со старым кодом.

**Разница между List и List<Object>**

В List можно присвоить ссылку на любой тип объектов параметризованного класса

*List<String> strings = new ArrayList<>();*

*List list = strings;*

Теперь в list можно добавить объект любого типа.

Если объявить list следующим образом *List<Object> list* , то в него нельзя будет присвоить список строк.

**Разница между List и List<?>**

List подразумевает что внутри храниться Object, или его подтипы. По сути туда можно засунуть что угодно, но это чревато ошибками. В List<?> лежат идентичные объекты, но мы точно не знаем какие. Поэтому компилятор запрещает добавлять туда что-то помимо null.

Не используйте Raw типы! Если аргумент типа не определен, то используйте wildcard <?>.

**Разница между <T> и <?>**

Параметризованный тип <T> инвариантен, <?> может быть ковариантным или контравариантным. <?> представляет собой любой тип, что запрещает вносить туда элементы, и читать помимо object. <T> конкретный тип, и нет таких ограничений (Можем положить T, и прочитать T). Мы не можем объявлять класс или метод, параметризованный <?>. Используем <?> там, где должен приниматься любой тип (чаще всего в параметрах метода).

**Иерархия обобщенных классов**

Обобщенный класс может действовать в качестве суперкласса или подкласса. В обобщенной иерархии любые аргументы типа, требующиеся обобщенному суперклассу, должны передаваться всеми подклассами вверх по иерархии.

*class Gen<T> {}*

*class Gen2<T, V> extends Gen<T>*

Суперклассом для обобщенного класса может служить и необобщенный класс.

*class NonGen {}*

*class Gen<T> extends NonGen {}*

Тип одного экземпляра класса можно привести к другому только в том случае, если они совместимы и их аргументы типа одинаковы.

**ВЫВЕДЕНИЕ ТИПОВ**

В JDK 7 было добавлено выведение типов. Тип аргументов типа может быть выведен из левой части, в правой части указываются пустые угловые скобки <>. Он называется ромбовидный оператор или Dimond.

*Имя\_класса<список\_аргументов\_типа> имя\_переменной = new имя\_класса<>(список\_аргументов\_конструктора);*

Выведение ттипов можно выполнять и при передаче параметров в метод

boolean isSame(MyClass<T, V> o) {  
 if(ob1 == o.ob1 && ob2 == o.ob2) return true;  
 else return false;  
}

**Мостовые методы** создаются компилятором в тех случаях, когда результат стирания типов в перегружаемом методе из подкласса не совпадает с тем, что получается при стирании в аналогичном методе из суперкласса.

class Gen<T> {  
 T ob;  
 Gen(T o) {  
 ob = o;  
 }  
   
 T getOb() {  
 return ob;  
 }  
}  
  
class Gen2 extends Gen<String> {  
 Gen2(String o) {  
 super(o);  
 }  
 String getOb() {  
 System.*out*.println("Вызван метод String getOb(): ");  
 return ob;  
 }  
}

Из-за стирания типов ожидаемая в подклассе форма метода getOb() будет *Object getOb() {}*, а нужно *String getOb();*

Поэтому в классе Gen2 компилятором создается метод *Object getOb() {}* который называется **мостовой метод,** который вызывает строковый вариант метода getOb();

**Ограничения на использование дженериков**

* **Создать экземпляр по параметру типа нельзя**

class Gen<T> {  
 T ob;  
 Gen() {  
 ob = new T(); // недопустимо  
 }  
}

Компилятору неизвестен тип создаваемого объекта из-за стирания типов.

* **В статических членах нельзя использовать параметр типа, объявленный в его классе** (создавать параметризованные статические поля, использовать параметризованные типы в статических методах).

Объявить статические методы со своими параметрами типа можно.

class Gen<T> {  
 static T *ob*; // нельзя  
 static T getOb() { // нельзя  
 return *ob*;  
 }  
   
 static <V> void doSmth(){} // можно  
}

* **Нельзя создать экземпляр массива, тип элемента которого определяется параметром типа.**

Объявить ссылку на массив типа T можно

Т vals[]; // Верно!

Но нельзя создать экземпляр массива с помощью new

vals = new T[10]; // нельзя

Создать массив типа T нельзя, потому что тип Т не существует во время выполнения из-за стирания типов.

Но можно присвоить такому массиву ссылку на совместимый по типу массив, поскольку передаваемый массив имеет известный тип, совпадающий с типом T на момент создания объекта

void method(T[] nums) {  
 T valse[] = nums; // можно  
}

* **Нельзя создать массив спецефических для типа обобщенных ссылок**.

Gen<Integer> gens[] = new Gen<Integer>[10]; // нельзя

Массив на обобщенный тип можно создать с помощью wildcards

Gen<?> gens[] = new Gen<?>[10]; // можно

* Обобщенный класс не может расширять Throwable. Это означает, что создавать обобщенные классы исключений нельзя. Каждое выражение try-catch проверяет тип полученного исключения во время выполнения, но тип обобщений затирается, и такая проверка невозможна.

Дженерики реализуют **параметрический полиморфизм**. Существует возможность описывать функции, которые будут работать с любыми типами данных. Такие функции называют параметрически полиморфными.

**Практическое задание:** нужно передать аргумент типа Class<T> в метод и вернуть из него объект типа T.

public static <T> T someMethod(Class<T> cls) {  
 try {  
 return cls.newInstance();  
 } catch (InstantiationException | IllegalAccessException e) {  
 e.printStackTrace();  
 }  
 return null;  
}