**Generics** – механизм, позволяющий типобезопасно работать с различными типами данных без изменения описания работы с этими данными путем добавления типа-параметра. То есть позволяют определять тип данных, которые будут использоваться в структурах во время компиляции, а не во время написания кода. Появились в Java 5. Поддерживают только ссылочные типы (примитивы типов не имеют).

Решают проблему дублирования кода при работе с разными типами данных, при этом избавляют от необходимости проверки типа на его принадлежность.

**Использование/Объявление:**

Класс – параметр может использоваться в любом члене класса (переменная, non-static метод, конструктор).  
Конструктор/non-static метод – могут быть объявлены локально, но ограничены использованием в области объявления.  
Static метод – параметр для использования в static методе объявляется только в нем.  
Переменные – non-static переменные только классом, static не могут быть параметризованы.

Ограничение на параметризацию static переменных связано с постоянством статического контекста (параметр задается для каждого объекта в отдельности и могут отличаться, а статический объект подразумевает неизменный контекст).

Ограничение на static методы (отдельная параметризация и невозможность использования нестатических параметров) нестатические параметры – также из-за общего статического контекста и «плавающего» контекста объектов + статический метод может быть вызван раньше создания экземпляра класса и определения типа параметра.  
Но при этом возможно независимая параметризация методов, не зависящая от нестатического контекста.

Объявленные параметры по умолчанию **extends Object** (могут использоваться Object и его наследники)**,** работают правила наследования – тип-параметр может наследовать 1 класс и несколько интерфейсов.

Доступно только extends, можно объявлять сразу несколько типов при этом действуют правила наследования (1 класс, неограничено интерфейсов), если среди них присутствует класс, то он должен быть объявлен первым.

public class A <T extends Number & Serializable > {  
}

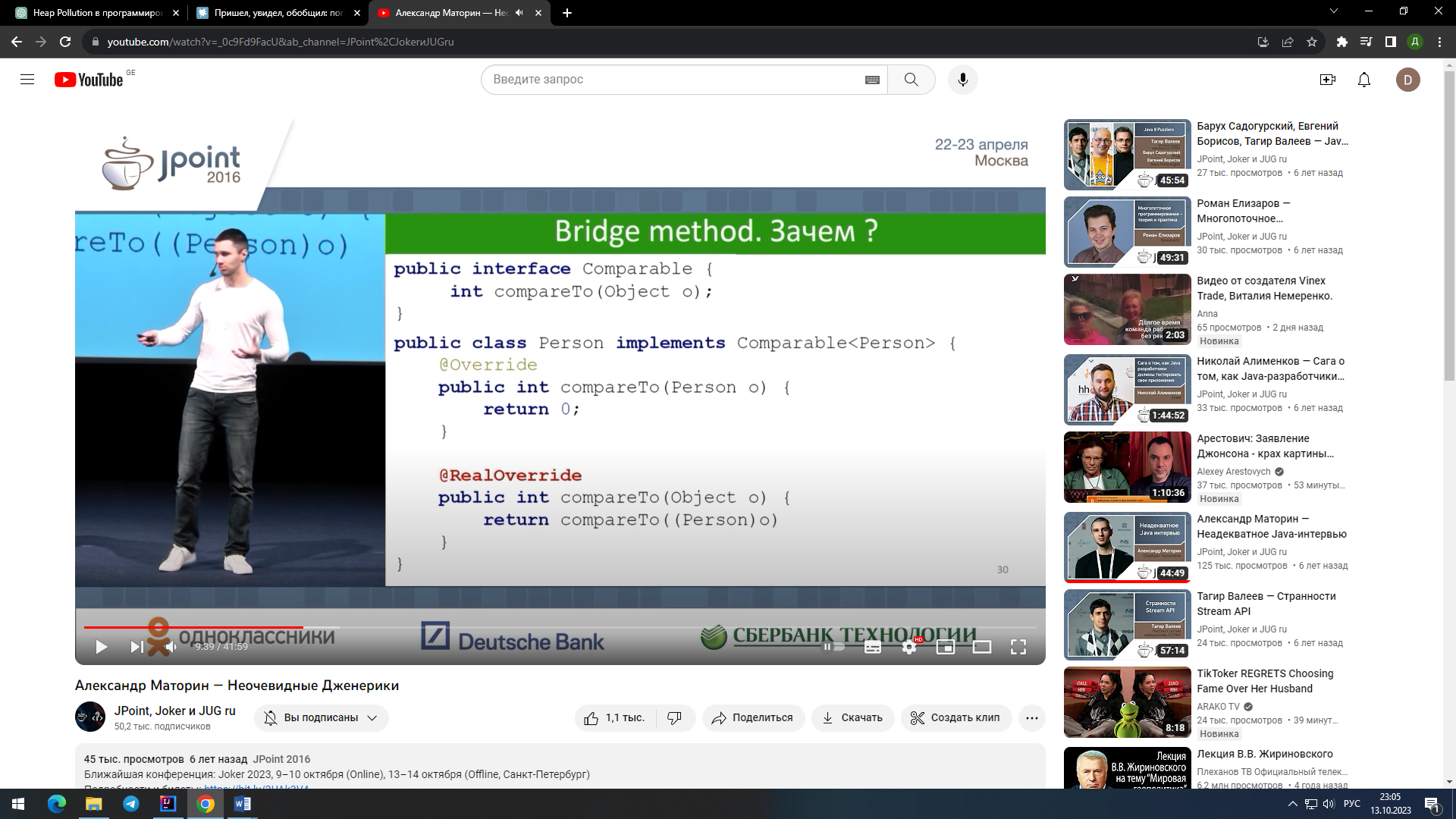
**Реализация и стирание типов (type erasure):**

Дженерики – синтаксический сахар. Типы-параметры существуют только на этапе компиляции (недоступны в рантайме). Во время компиляции программы информация о типах объектов стирается до верхней границы типа параметра, указанного в классе, а тип объекта при получении параметризированного типа будет выведен исходя из границ, определенных при создании объекта и границы типа-параметра класса.   
Информация о параметризации класса остается на уровне метаданных и может быть получена с помощью рефлексии (хранится в иерархии классов Type), при этом информация о значении этих параметров в каждом отдельном объекте стирается и определить ее невозможно. Т.е. для класса ArrayList<E> можно получить информацию о том, что он в целом параметризован <E>, но информация при создании экземпляра класса с конкретным типом будет утеряна.

Пример:

public class Product <T>{  
 void foo(T number){}  
 void foo(Object number){}  
} // не скомпилируется, т.к. это 2 одинаковых метода (T скастится до Object)

Стирание типов обеспечивает обратную совместимость с кодом, написанным до Java 5.

**Bridge methods** – синтетический метод, созданный компилятором в процессе стирания типов для обеспечения полиморфного поведения.

Интерфейс Comparable<T> параметризирован, то есть затирается до Object. А когда мы имплементим этот интерфейс мы можем засунуть в параметр какой угодно ссылочный тип, компилятор покажет, будто мы перегрузили этот метод, но родительский метод при стирании типов должен принимать Object, чтобы обеспечить предсказуемое поведение, в процессе стирания создается некий адаптер к написанному методу, который реализует нужный метод и в себе вызывает пользовательский, кастуя аргумент до нужного.

**Raw type** – использование параметризированных классов без указания типов-параметров при создании объекта параметризированного класса. Использование Raw type приводит к тому, что компилятор не будет автоматически проверять добавляемые типы и осуществлять их автоматическое приведение (все типы будут записаны как Object, достаются тоже Object).

\*Создавая параметры, мы ограничиваем кол-во методов, которые можно вызывать на данным объекте до методов, соответствующих правилам типобезопасности.

**Ограничения Дженериков:**

1) Статик переменные - является общей переменной для всех объектов класса, ее изменение в 1 объекте меняет ее значение для всех объектов. Также по причине связи дженериков с объектами, а не с классом static методы параметризуются отдельно (если бы параметр задавался объектом, то создание другого объекта с иным типом вызывало бы изменение всех предыдущих объектов, т.е. static методы отвязали от объекта).

2) Throwable – нельзя параметризовывать классы-наследники Throwable, т.к. информация о типе объекта в рантайме будет стерта до общего и не будут различимы в рантайме.

try {  
 run();  
 } catch (GenegicException<Integer> exception) {  
 //...  
 } catch(GenegicException<String> exception) {  
 //...  
}

Но можно параметризовывать сами исключения.

public class Product<T extends Throwable> {  
 void foo() throws T {  
 throw (T) new Throwable();  
 }  
}

3) Анонимные классы – бессмысленно, т.к. они подразумевают создание только одного инстанса и в этом случае нет вариативности.

4) Enum

5) Массивы – нельзя объявлять параметризованные массивы из-за потери информации о их типах в рантайме и ковариантности массивов. Сочетание этих свойств приводит к невозможности обеспечить типобезопасность и может привести к heap pollution (см. пункт 2 heap pollution)

**Основное предназначение дженериков – обеспечивать типобезопасность или избегать heap pollution.**

**Heap pollution** — это ситуация, при которой фактически получаемые типы не соответствуют ожидаемым типам данных. Из-за ошибок, связанных с типобезопасностью и это обнаруживается в рантайме.

Причины: 1) Использование Raw-types

static List<String> makeHeapPollution() {  
 List numbers = new ArrayList<Number>(); //Используем лист без параметров.  
 numbers.add(1);  
 List<String> strings = numbers; //у numbers не кастятся типы из-за отсутствия <>  
 //Если поставить <>, то компилятор не даст сделать присвоение, так как знает параметр numbers  
   
 strings.add(""); //в листе стрингов оказались и инты и стринги, при этом ожидаем возврат листа стрингов.   
 return strings;  
}

2)varargs

Использование varargs это по сути создание массива параметризированного типа, что запрещено в Java, но разрешено с varargs, но компилятор выкинет предупреждение, если уверен, что проблем не будет, то можно поставить аннотацию @SafeVarargs

static void makeHeapPollution(List<String>... stringsLists) {  
 Object[] array = stringsLists; //List<String>[] stringsLists стал List[] с Raw типом.  
 List<Integer> numbersList = Arrays.*asList*(66, 22, 44, 12);  
  
 array[0] = numbersList; //так как мы потеряли параметр листа, мы можем писать туда что угодно  
 String str = stringsLists[0].get(0);  
}

**Wildcard** – неизвестный тип. Позволяет работать с параметризированными типами, не привязываясь к конкретному типу при создании экземпляра класса параметризованного типа.

*Ограниченный сверху Wildcard –*

**<? extends Type>** - неизвестный тип, ограниченный указанным типом (можно присваивать класс и подклассы указанного типа) в рамках ограничений параметра-типа.

public class Main <T extends Number> {  
   
 Main<? extends Integer> main = new Main<>();  
   
}

Тип Main сначала ограничен T extends Number и в этом диапазоне снова ограничен до extends Integer (то есть Integer и его наследники, если бы они были)

**<?> -** неизвестный тип, неявно ограниченный типом типа-параметра. То есть если <T extends Number>, то <?> неявно <? extends Number>

*Ограниченный снизу Wildcard* -

**<? super Type>** - может принимать класс и суперклассы указанного типа.

**Ковариантность, контрвариантность, инвариантность типов.**

**Ковариантное поведение** – это сохранение иерархии наследования исходных типов в произвольных типах.

Выражено **<? extends Type>**, то есть в переменную ArrayList<? extends Number> можно положить лист Number и всех его потомков.

**Контрвариантное поведение** - обращение иерархии на противоположную в произвольных типах.

Выражено **<? super Type>,** то есть в переменную ArrayList<? super Number> можно присвоить лист Number и его предков.

**Инвариантное поведение** – отсутствие ковариантного или контравариантного поведения.  
Выражено <Type>, то есть ArrayList<Number> можно присвоить только лист Number. То есть дженерики, выраженные определенным типом инвариантны   
ArrayList<Number> != ArrayList<Integer>

**PECS**

Из такого поведения присваивания и требований соблюдения типобезопасности дженерики ограничиваются в операциях получения и добавления элементов в параметризированные объекты по правилу PECS (producer extends consumer super):

**<? extends Type>** - из-за того, что такой переменной можно присвоить ссылку на Type и его наследников, то безопасно достать из такого объекта можно Type и его предков, писать нельзя ничего, кроме null.

Пример:

Получение: Есть ArrayList<? extends Number> - в такой переменной могут лежать только ссылки на ArrayList<Number> или его наследников исходя из ковариантного поведения, если там окажется ArrayList<Integer>, то безопасно (без ClassCastException) его можно будет привести только Number и его предкам.

Помещение: Объекты, которые туда можно поместить ограничены с двух сторон – предков Number положить не удастся, потому что это противоречит ковариантному поведению (Number и потомки), а Number и его потомков поместить нельзя поскольку конкретный тип ссылки, которая будет присвоена в переменную недоступен: нельзя положить Number, если окажется что ArrayList<Integer>, то он не может хранить Number, также невозможно угадать и с наследниками Number, поэтому безопасно можно поместить только null.

**<? Super Type>** - из-за того, что такой переменной можно присвоить ссылку на Type и ее предков, то безопасно положить в него можно Type и его наследников, читать допустимо только Object.

Пример:

Получение (те же условия): потомков Number мы получить не можем, т.к. исходя из контравариантного поведения там могут быть только Number и его предки, Number тоже нельзя получить, т.к. там может лежать ArrayList<Object>, а Number не кастится к Object, то есть остается только супер-тип Object.

Помещение: супертипы Number положить нельзя поскольку в переменной может оказаться ссылка на Number, а Number и его потомков положить можно, т.к. все они приводятся и к Object и к Number.

**<Type>** - для такой переменной доступна только ссылка на ArrayList<Number>, положить туда можно Type и его потомков, получить Type и его предков.

Пример:

Получение: Потомков Number получить не можем, т.к. невозможно определить тип приведения, а получение Type и его предков соответствует типобезопасности.

Помещение: Предков добавить нельзя, поскольку ссылка на Number не может содержать Object.

\*\*\*В рассуждениях нужно полагаться на то, какой тип переменной типобезопасно кастится для каждого возможного значения ссылки исходя из вариантности. Примерный алгоритм рассуждений:

Пример получения:

List<? extends Number> - может принимать значения Number и предки, нужно выбрать такие типы, чтобы в перспективе безопасно можно скастить каждый вариант этого диапазона. Integer и др потомки нельзя, потому что предки не кастятся к потомкам, Number и предки безопасны, т.к. все варианты значений ссылок можно привести к ним.

Пример присвоения:

List<? extends Number> - Какие типы безопасно положить в такой диапазон? Integer – нет, поскольку там может лежать любой другой потомок, Number или его предок – нет, поскольку там может лежать потомок, а предка к потомку приводить нельзя   
(Integer I = new Object() – так не работает).