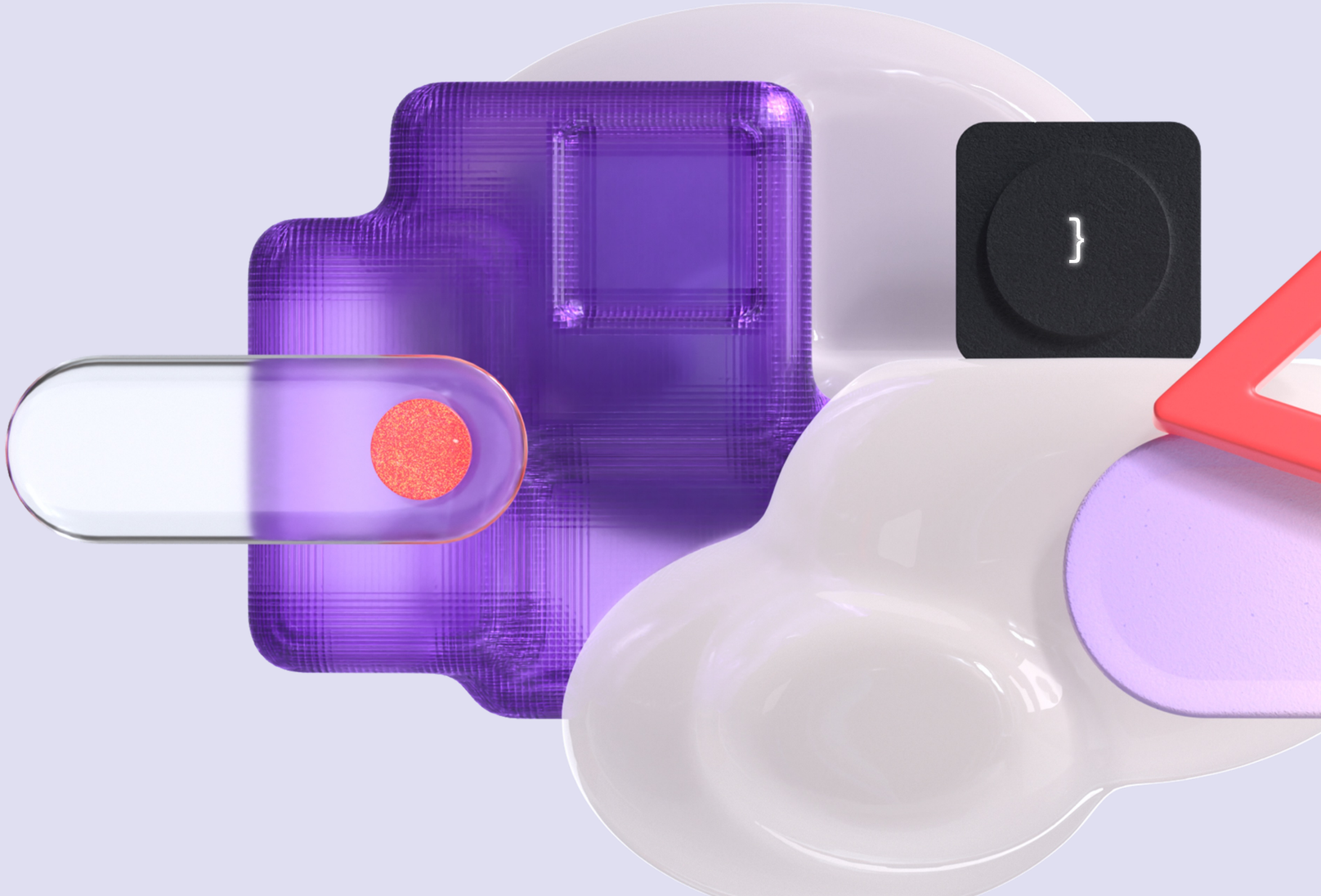
MTC True Tech +

Лекция №6

Java Generics



Бобряков Александр

Tech Lead, Команда Идентификации

Меня хорошо видно и слышно?

Проверить, идет ли запись

Ставим «+», если все хорошо
«-», если есть проблемы

Александр Бобряков

Tech Lead

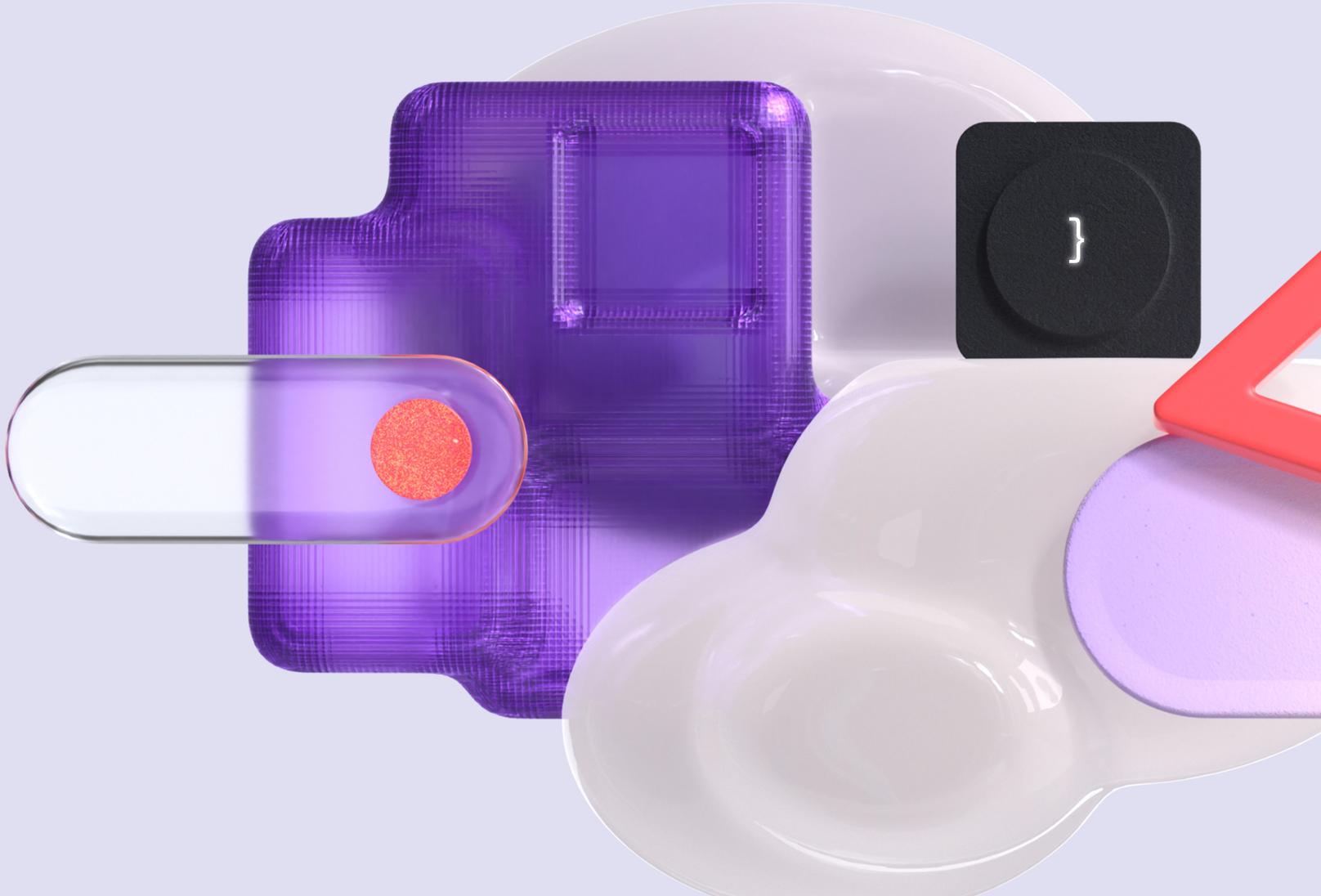
- Tech Lead команды Идентификации в MWS
- Lead Java-направления в МТС Тета, автор образовательных курсов
- Спикер на IT-конференциях, автор статей на Хабр
- Преподаватель ВШПИ
- TechMaster по направлению Java и TechLead в MWS
- Основной профиль – real time обработка данных в higload-среде

Содержание лекции

- Необходимость Generics
- Базовый синтаксис Generics
- Обертки вокруг примитивов
- Обобщенные методы
- Инвариантность Generics
- Wildcards
- Стирание типов
- Ограничения Generics

MTC True Tech

1. Необходимость Generics



Класс Container для Car

```
public class Container {
    private final Car value;
    // constructor...

    public boolean isPresent() {
        return value != null;
    }

    public Car getValue() {
        return value;
    }
}
```

Использование:

```
public static void main(String[] args) {
    final Container container = new Container(new Car(...));
    final Car car = container.getValue();
    car.someMethod();
}
```

Универсальный класс Container для всего

```
public class Container {
    private final Object value;
    // constructor...

    public boolean isPresent() {
        return value != null;
    }

    public Object getValue() {
        return value;
    }
}
```

Использование:

```
public static void main(String[] args) {
    final Container container = new Container(new Car(...));
    final Object car = container.getValue();
    car.someMethod(); ← не компилируется

    /* но мы же знаем, что передали именно
    объект класса Car */
    final Car carcar = (Car) container.getValue();
    carcar.someMethod(); ← работает
}
```

Ошибка кастинга

Если в контейнере (Container) на самом деле не Car, то упадет ошибка ClassCastException

Использование:

```
public static void main(String[] args) {
    final Container container = new Container(new Car(...));
    final Object car = container.getValue();
    car.someMethod(); ← не компилируется

    /* но мы же знаем, что передали именно
    объект класса Car */
    final Car carcar = (Car) container.getValue();
    carcar.someMethod(); ← работает
}
```

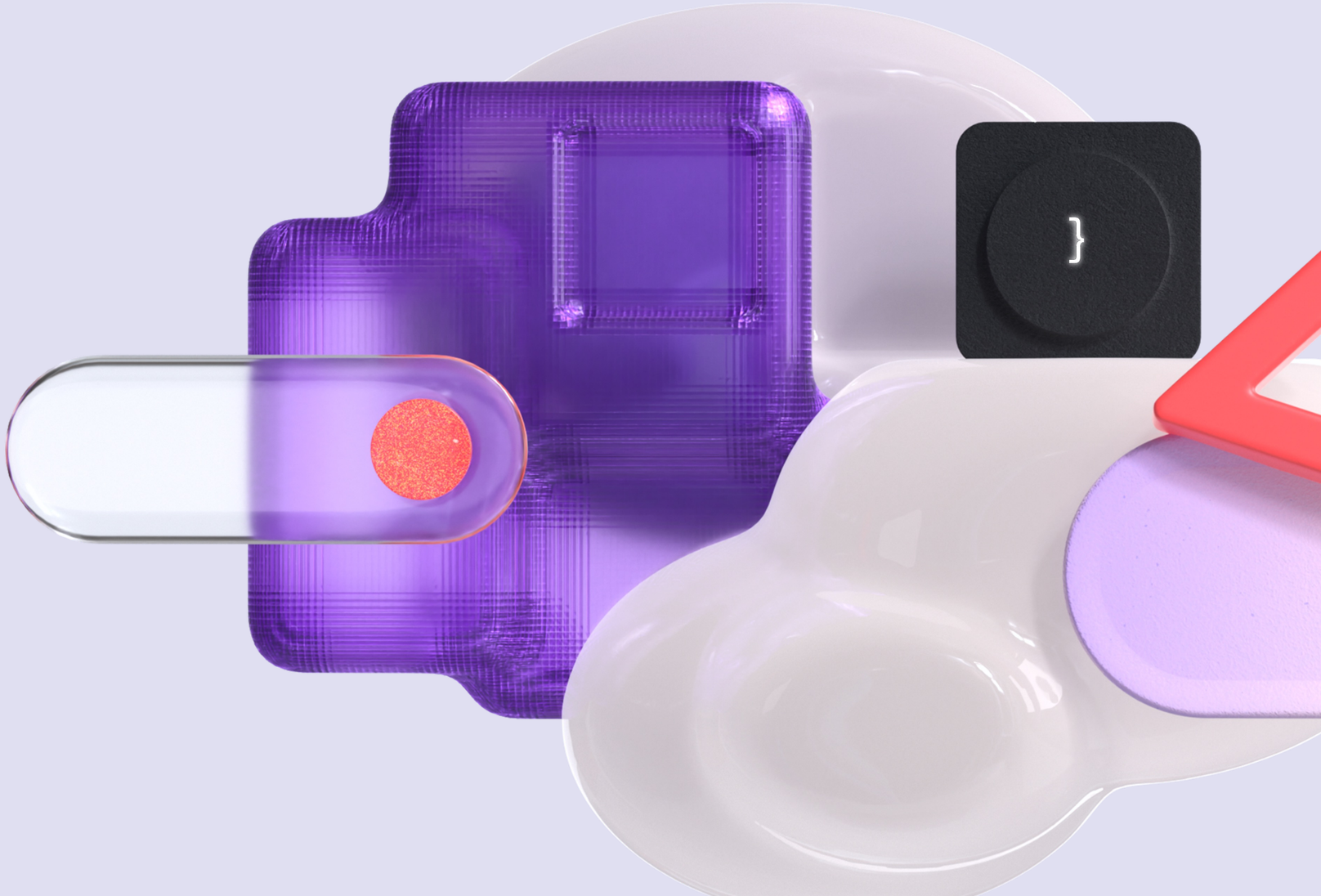
ClassCastException

```
public static void main(String[] args) {
    final String str = "some_text";
    final Object obj = str;
    final LocalDate date = (LocalDate) obj;
}
```

Exception in thread "main" java.lang.ClassCastException: java.lang.String incompatible with java.time.LocalDate
at ...

MTC True Tech

2. Базовый синтаксис Generics



Внедрение дженериков

До

```
public class Container {
    private final Object value;
    // constructor...

    public boolean isPresent() {
        return value != null;
    }

    public Object getValue() {
        return value;
    }
}
```

После

```
public class Container<T> {
    private final T value;
    // constructor...

    public boolean isPresent() {
        return value != null;
    }

    public T getValue() {
        return value;
    }
}
```

Внедрение дженериков

```
public static void main(String[] args) {
    // для Car
    final Container<Car> container1 = new Container<>(new Car(...));
    final Car car = container1.getValue();
    car.someMethod();

    // для String
    final Container<String> container2 = new Container<>("string");
    final String str = container2.getValue();
    str.toLowerCase();
}
```

Соглашение о наименовании типов

Наименование может быть любым (даже не одной буквой)

The most commonly used type parameter names are:

- E - Element (used extensively by the Java Collections Framework)
- K - Key
- N - Number
- T - Type
- V - Value
- S,U,V etc. - 2nd, 3rd, 4th types

https://docs.oracle.com/javase/tutorial/java/generics/types.html

Обобщенные интерфейсы

```
public interface Pair<K, V> {
    public K getKey();
    public V getValue();
}
```

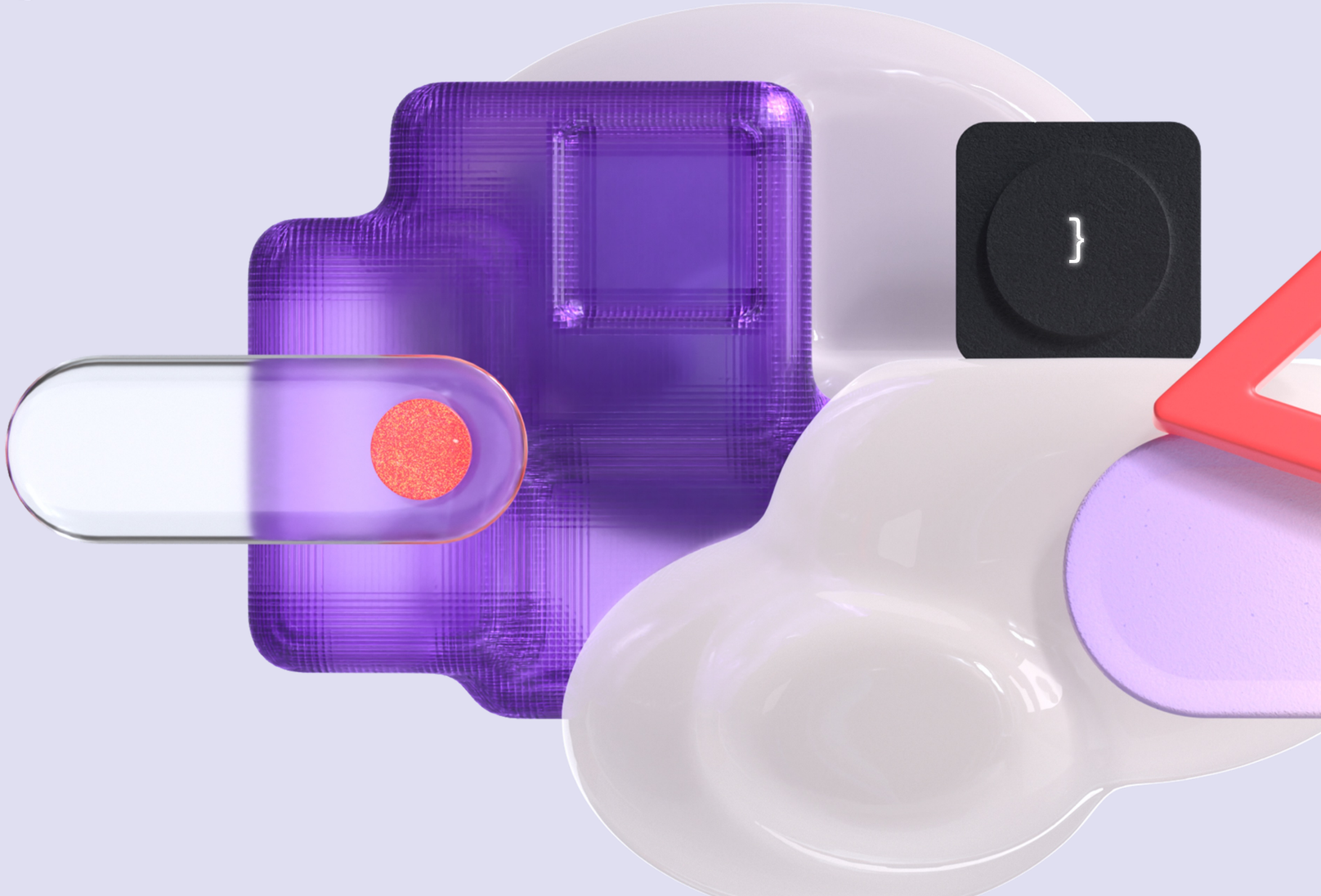
```
public class StringPair<K> implements Pair<K, String> {
    private K key;
    private String value;

    @Override
    public K getKey() {
        return key;
    }

    @Override
    public String getValue() {
        return value;
    }
}
```

Вложенные типы

```
public static void main(String[] args) {
  List<List<List<List<List<String>>>>> listsOfLists = new ArrayList<>();
}
```

MTC True Tech

3. Обертки вокруг примитивов



Будут ли работать примитивы в дженериках

```
public static void main(String[] args) {
    final Container<int> = ...← не компилируется
    final Container<Integer> ← работает
}
```

Классы-обертки

Wrapper Classes for Primitive Data Types	
Primitive Data Types	**Wrapper Classes**
int	Integer
short	Short
long	Long
byte	Byte
float	Float
double	Double
char	Character
boolean	Boolean

Классы-обертки

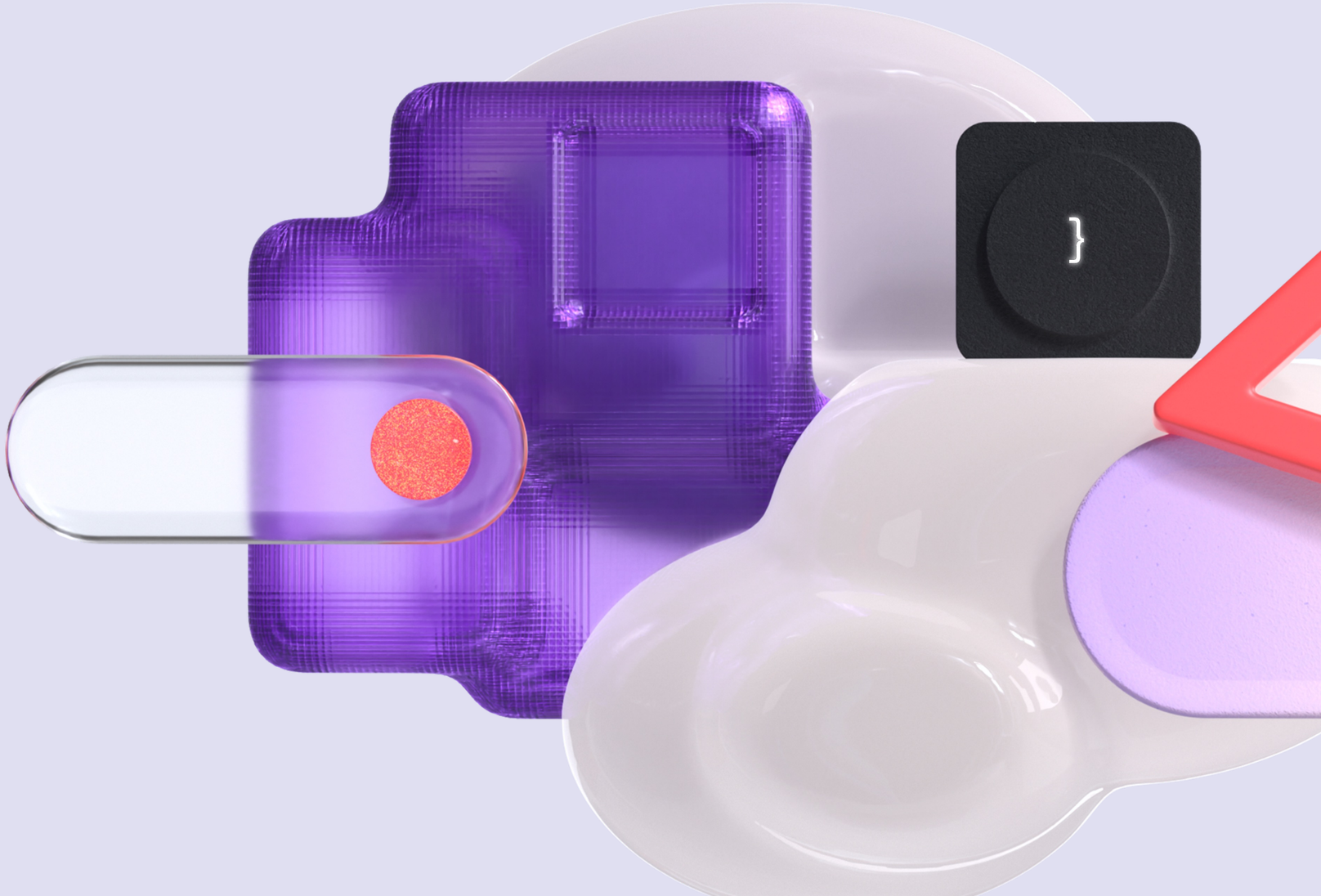
```
final Integer int_1 = Integer.valueOf( i: 345);
final Integer int_2 = 123;
int_2.
```

byteValue() byte
intValue() int
compareTo(Integer anotherInteger) int
fori for (int i = 0; i < expr.length;…
describeConstable() Optional<Integer>
doubleValue() double
equals(Object obj) boolean
floatValue() float
hashCode() int
longValue() long
resolveConstantDesc(Lookup lo… Integer
shortValue() short

Press ⏎ to insert, ⇥ to replace

MTC True Tech

4. Обобщенные методы



Обобщенные методы

Обобщенные методы позволяют параметризовать отдельные методы, независимо от того, является ли класс обобщенным

```
public class Utils {
    // Обобщенный метод - <T> перед возвращаемым типом
    public static <T> T getMiddleValue(T[] array) {
        if (array == null || array.length == 0) return null;
        return array[array.length / 2];
    }

    // Метод с несколькими параметрами типа
    public static <K, V> String formatPair(K key, V value) {
        return key.toString() + " -> " + value.toString();
    }
}
```

Ограничение типа

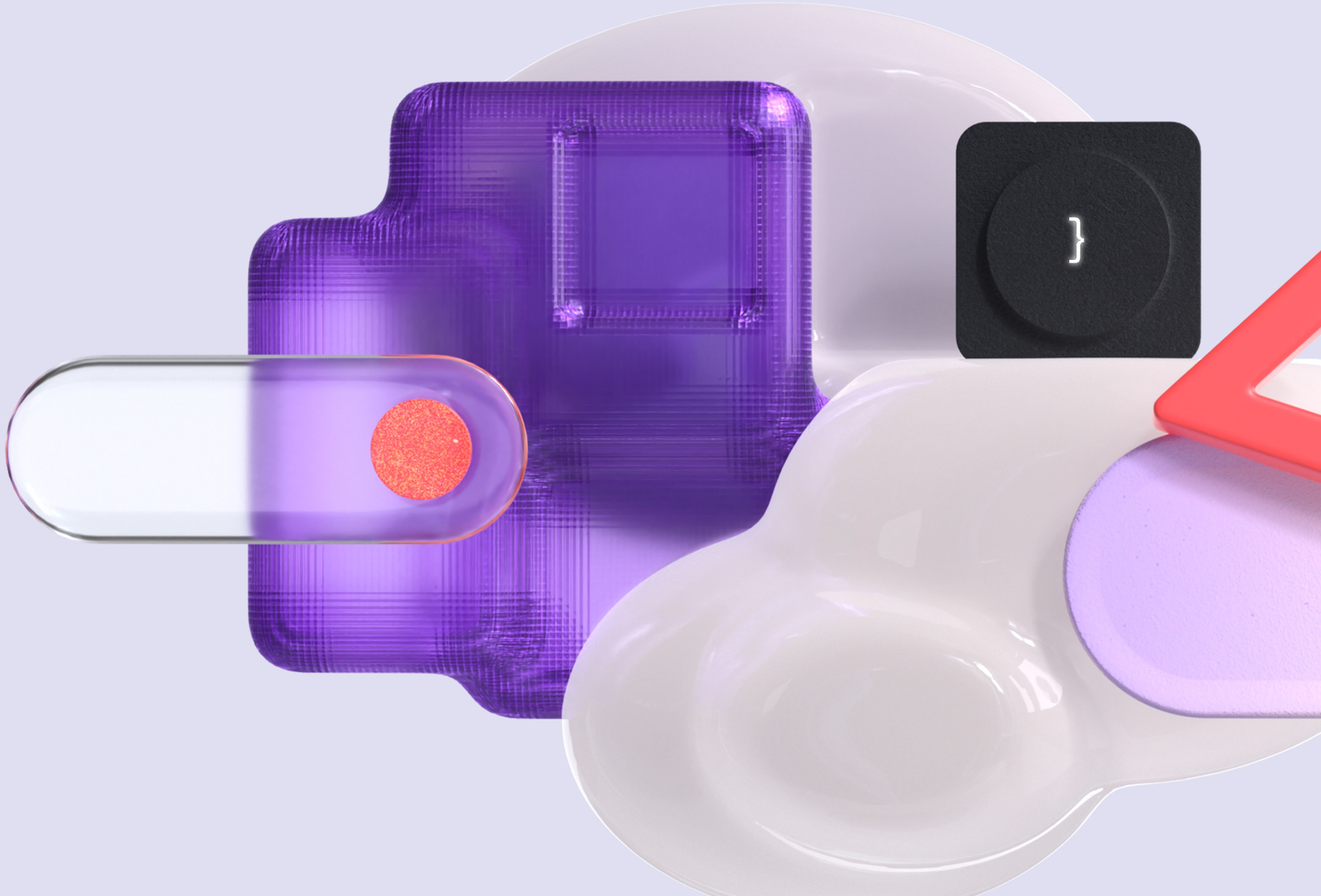
Обобщенные методы позволяют параметризовать отдельные методы, независимо от того, является ли класс обобщенным

```
public class Utils {
    // Метод с ограниченным типом
    public static <T extends Comparable<T>> T max(T a, T b) {
        return a.compareTo(b) > 0 ? a : b;
    }
}
```

```
final Integer maxNumber = Utils.max(10, 20); // T выводится как Integer
```

MTC True Tech

5. Инвариантность Generics



Инвариантность

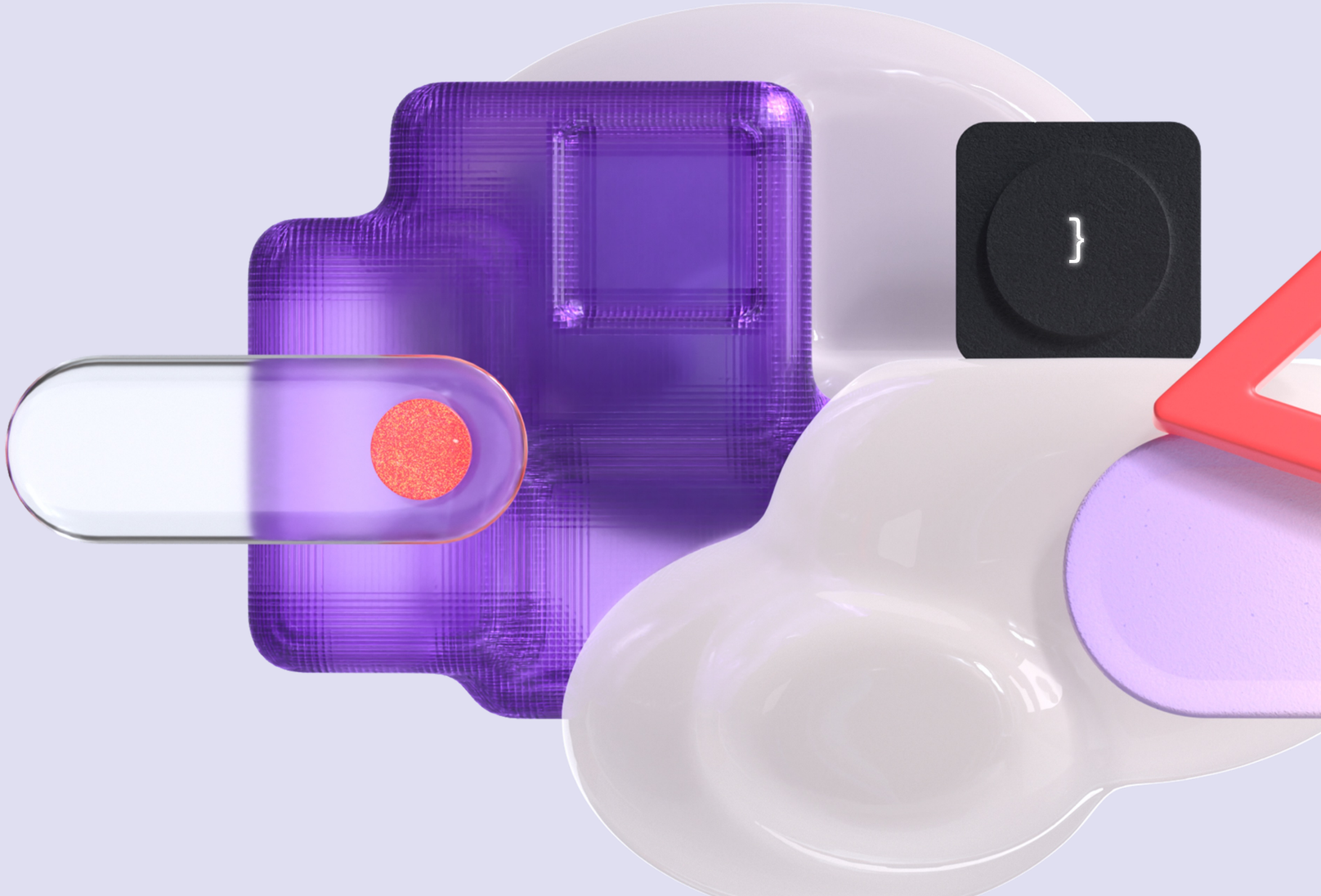
Generics в Java **инвариантны**. Это означает, что если String является подтипом Object, то List<String> не является подтипом List<Object>.

Массивы в Java ковариантны.

```
public static void main(String[] args) {
    // Допустимые присваивания
    final Object obj = "string"; // String наследуется от Object
    final String[] strArray = new String[10];
    final Object[] objArray = strArray; ← работает (Ковариантность массивов)

    // НЕдопустимые присваивания с Generics
    final List<String> strList = new ArrayList<>();
    final List<Object> objList = strList; ← не компилируется
}
```

MTC True Tech

6. Wildcards



Wildcards

Wildcards (подстановки) обеспечивают гибкость при работе с обобщенными типами, позволяя создавать методы, которые могут работать с различными “инстанциациями” generic-типов.

Виды:
1. Unbounded Wildcard **(?)**
2. Upper Bounded Wildcard **(? extends T)**
3. Lower Bounded Wildcard **(? super T)**

1. Unbounded Wildcard **(?)**

```
public static void main(String[] args) {
    final List<String> strList = List.of("3", "et", "4trdbf");
    final List<?> list = strList;

    list.add("new str"); ← не компилируется
    final Object o = list.get(0); ← работает (но достаем только как Object)
}
```

Ограничения:

- Можно читать как Object
- Нельзя добавлять элементы (кроме null)
- Компилятор не знает конкретный тип

Когда использовать:

- Когда метод работает с элементами любого типа
- Когда используются только методы из Object
- Для методов, которые не зависят от типа элементов (size(), clear())

1. Unbounded Wildcard **(?)**

```
public static void printList(List<?> list) {
    for (Object item : list) {
        System.out.println(item);
    }
}
public static boolean isEmpty(Collection<?> collection) {
    return collection.size() == 0;
}
```

```
public static void main(String[] args) {
    final List<String> strings = List.of("A", "B", "C");
    final List<Integer> numbers = List.of(1, 2, 3);

    printList(strings);  // Работает
    printList(numbers);  // Работает
    isEmpty(strings);    // Работает
    isEmpty(numbers);    // Работает
}
```

2. Upper Bounded Wildcard **(? extends T)**

Представляет собой верхнее ограничение типа

```
public static double sumNumbers(List<? extends Number> numbers) {
    ...
}
```

Ограничения:
- Можно читать как тип T (или его предка)
- Нельзя добавлять элементы (кроме null)

Когда использовать:
- Для методов, которые только читают из коллекции
- Когда нужен доступ к методам определенного класса/интерфейса

2. Upper Bounded Wildcard **(? extends T)**

```
public static double sumNumbers(List<? extends Number> numbers) {
    double sum = 0;
    // Можно читать как Number
    for (Number number : numbers) {
        sum += number.doubleValue();
    }
    return sum;
}
```

```
public static void main(String[] args) {
    final List<Integer> integers = List.of(1, 2, 3);
    final List<Long> longs = List.of(1L, 2L, 3L);
    sumNumbers(integers);  // 6.0
    sumNumbers(longs);     // 6.0

    final List<String> strings = List.of("A", "B");
    sumNumbers(strings); ← не компилируется
}
```

3. Lower Bounded Wildcard **(? super T)**

Представляет собой нижнее ограничение типа

```
public void addNumbers(List<? super Integer> list) {
    ...
}
```

Ограничения:

- Можно добавлять элементы типа T (и его подтипов)
- Читать можно только как Object

Когда использовать:

- Для методов, которые только добавляют в коллекцию
- В шаблоне "Producer-Consumer" (PECS)
- Когда нужно записывать элементы определенного типа

3. Lower Bounded Wildcard **(? super T)**

```
public static void addNumbers(List<? super Integer> list) {
    for (int i = 1; i <= 3; i++) {
        list.add(i);  // Можно добавлять Integer
    }
}
```

```
public static void main(String[] args) {
    final List<Object> objects = new ArrayList<>();
    final List<Integer> integers = new ArrayList<>();
    addNumbers(objects);  // OK - Object super Integer
    addNumbers(integers); // OK - Integer super Integer

    final List<Double> doubles = new ArrayList<>();
    addNumbers(doubles); ← не компилируется  - Double не super Integer
}
```

PECS принцип

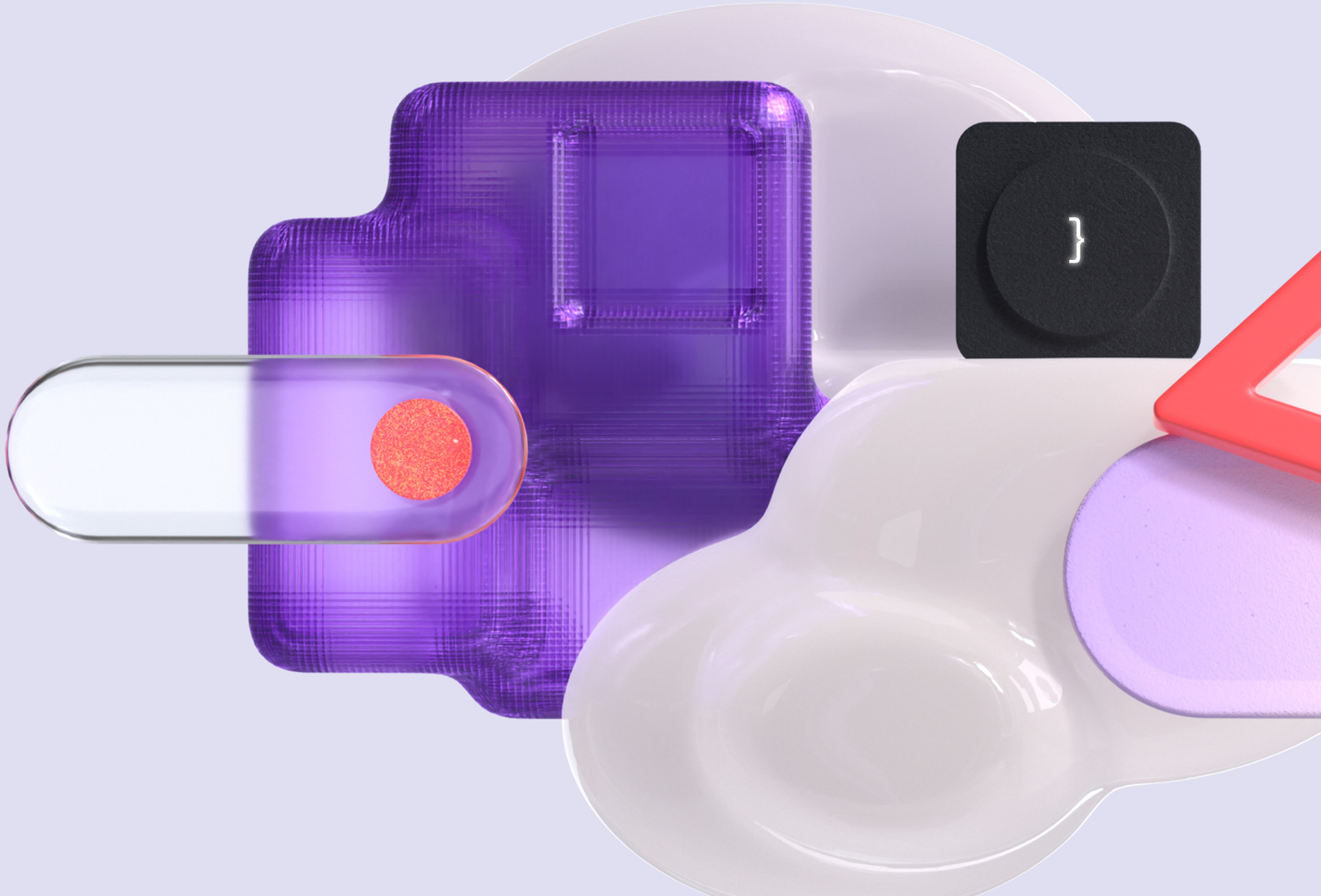
PECS -Producer-Extends, Consumer-Super:

1. Producer (источник) → используй ? extends T
2. Consumer (потребитель) → используй ? super T

```
// Producer - только отдает данные
public void processProducer(List<? extends Number> producer) {
    for (Number number : producer) {
        System.out.println(number);
    }
}


// Consumer - только принимает данные
public void fillConsumer(List<? super Integer> consumer) {
    consumer.add(1);
    consumer.add(2);
}
```

MTC True Tech

7. Стирание типов



Стирание типов

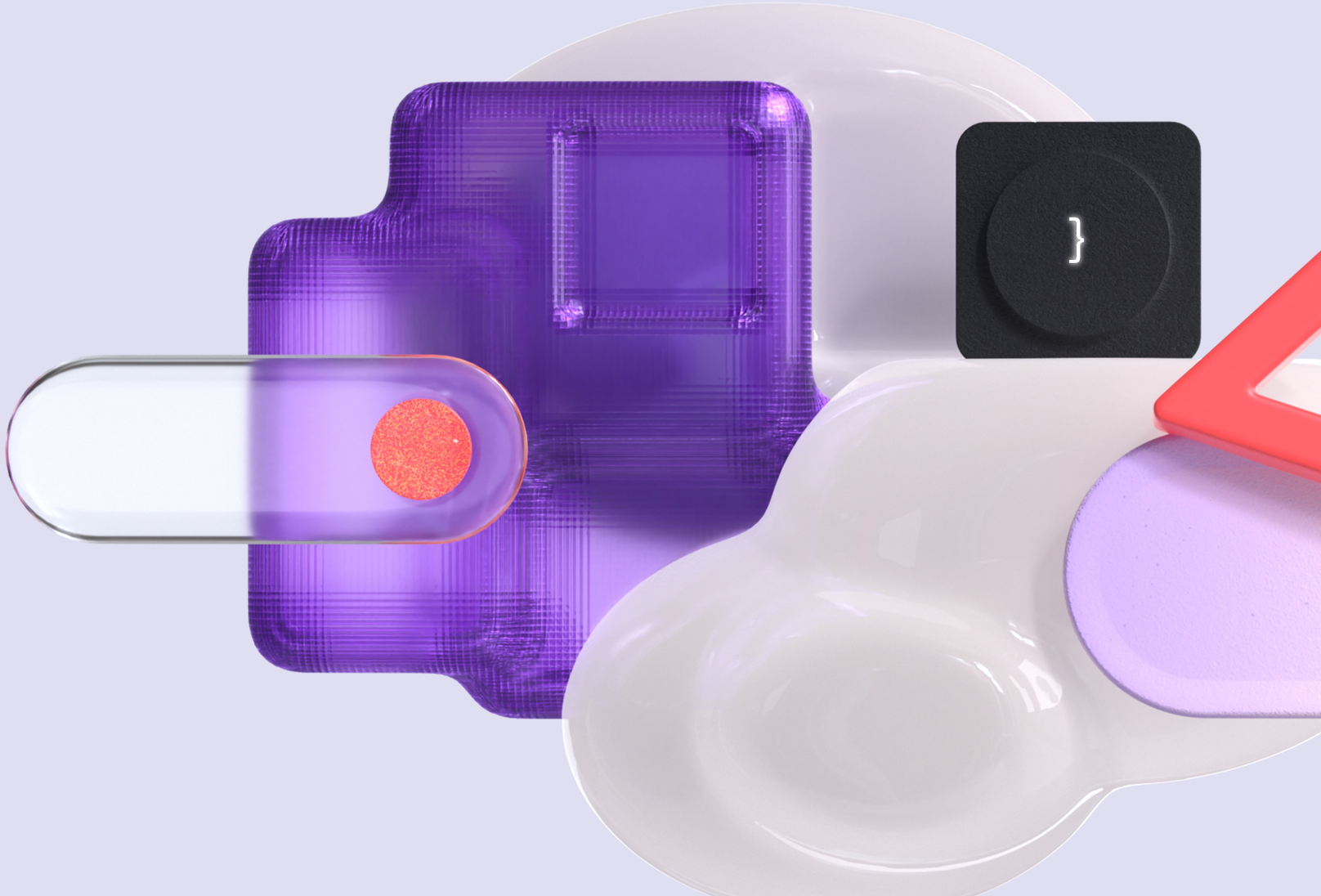
Type Erasure - это процесс, при котором компилятор Java удаляет всю информацию о generic-типах во время компиляции. Это сделано для обратной совместимости с кодом, написанным до Java 5.

```java
// Исходный код
public class ErasureExample<T> {
    private T value;

    public void setValue(T value) {
        this.value = value;
    }

    public T getValue() {
        return value;
    }
}
```

```java
// После компиляции (декомпилированный вид)
public class ErasureExample {
    private Object value; // T заменен на Object

    public void setValue(Object value) {
        this.value = value;
    }

    public Object getValue() {
        return value;
    }
}
```

MTC True Tech

8. Ограничения Generics



1. Нельзя создавать экземпляры generic-типов

```
public class MyClass<T> {
    private T value;

    public MyClass() {
        this.value = new T(); ← не компилируется - неизвестен конструктор
        this.value = T.class.newInstance(); ← не компилируется
    }

    // Обходное решение через Class<T>
    public MyClass(Class<T> clazz) throws Exception {
        this.value = clazz.newInstance(); ← работает
    }
}
```

Причина: Во время выполнения информация о типе T стирается, поэтому JVM не знает, какой конструктор вызывать.

2. Нельзя использовать примитивы

```
public static void main(String[] args) {
    final List<int> primitiveList = null; ← не компилируется
    final List<Integer> wrapperList = new ArrayList<>(); ← работает
}
```

Причина: Generics работают только с ссылочными типами. Используйте классы-обертки для примитивов.

3. Нельзя создавать generic-массивы

```
public class MyClass<T> {
    private T[] array = new T[10]; ← не компилируется

    // Обходное решение
    public T[] createArray(Class<T> clazz, int size) {
        return (T[]) java.lang.reflect.Array.newInstance(clazz, size); ← работает
    }
}
```

Причина: Из-за стирания типов JVM не знает точный тип для создания массива.

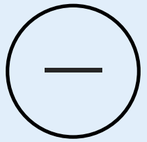
4. Generic в статических полях

```
public class StaticMember<T> {
    private static T staticField;   ← не компилируется
    private static int count = 0; ← работает

    private static void staticMethod(T param) { ← не компилируется
    }

    public static <U> void genericStaticMethod(U param) {← работает
        System.out.println(param);
    }
}
```

Причина: Статические члены принадлежат классу, а не его экземплярам. Поскольку параметры типа связаны с экземплярами, они не могут использоваться в статическом контексте.

Вопросы?

Ставим «+»,
если есть вопросы



Ставим «-»,
если нет вопросов