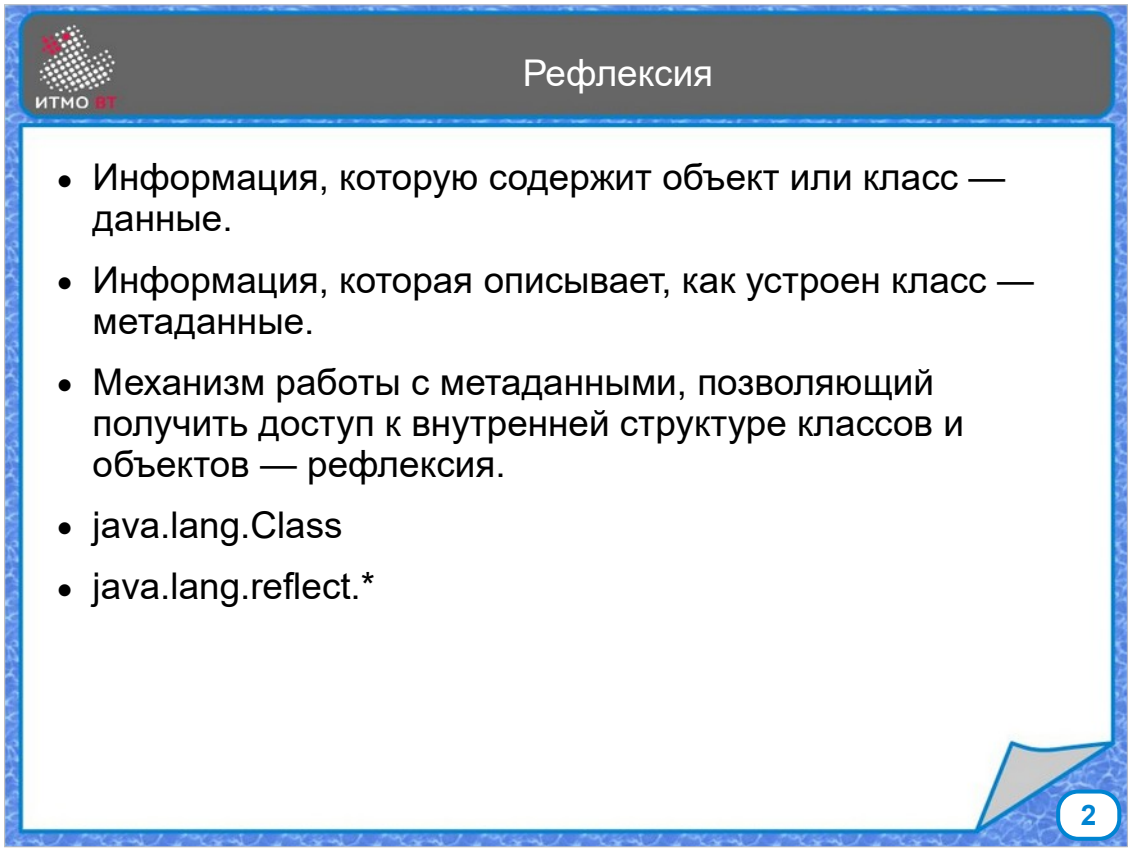
ИТМО ВТ

Рефлексия

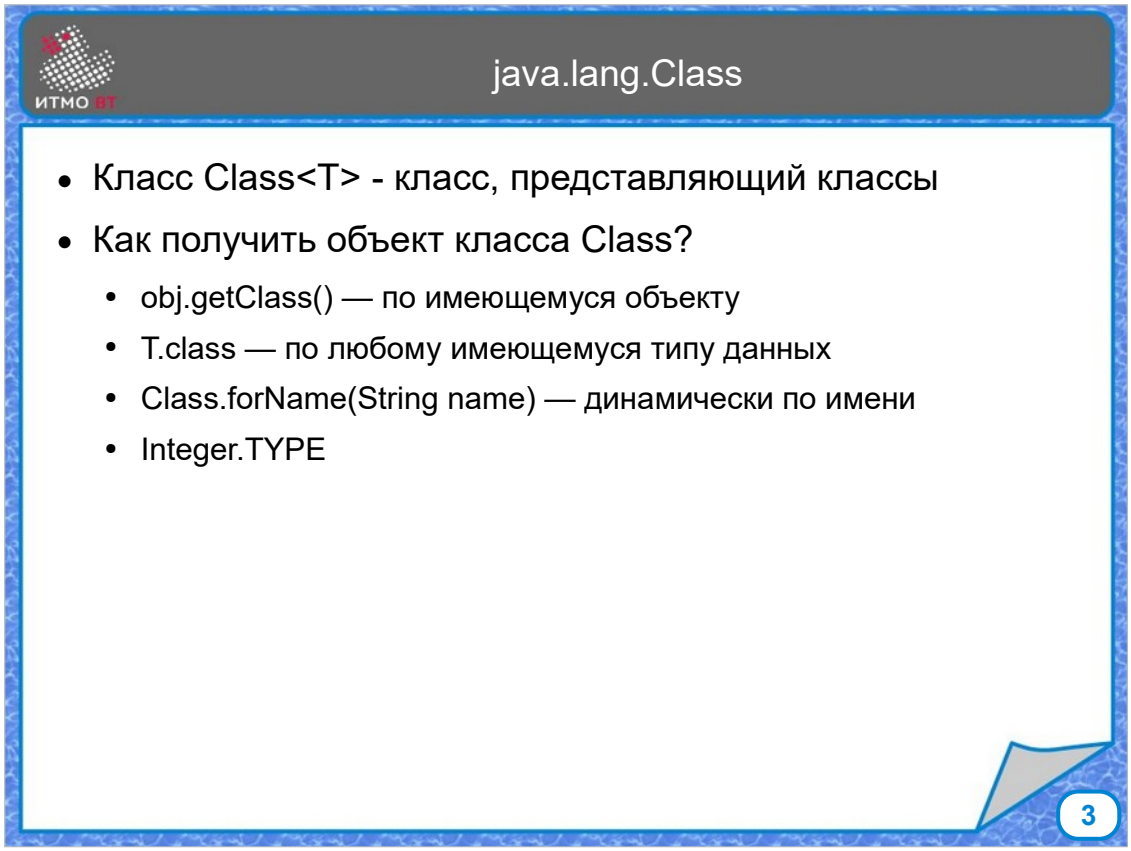


Данные - это информация, с которой работает программа, которую содержат объекты и переменные.

Метаданные - это информация о данных, об их типе, структуре, состоянии.

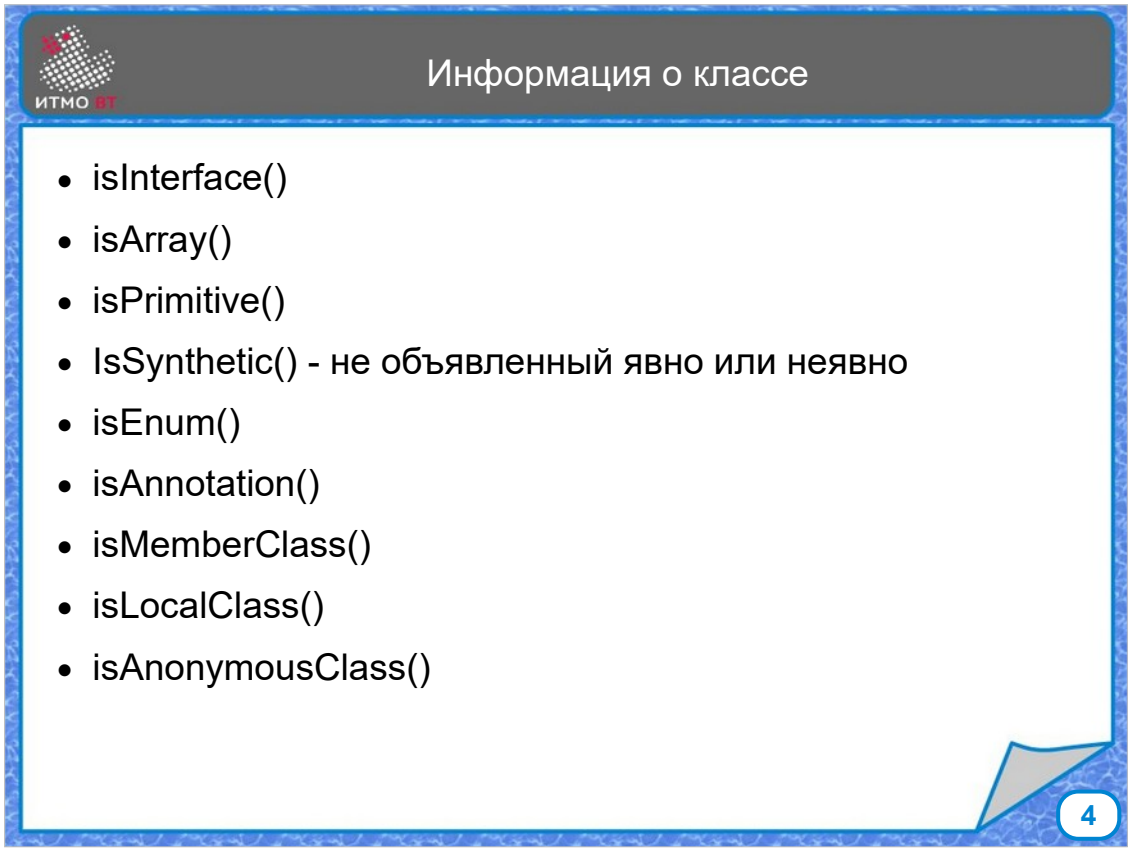
Рефлексия - механизм работы с метаданными, позволяющий динамически получать доступ к объектам, управлять ими во время исполнения программы.

Для реализации рефлексии в Java есть класс Class и пакет java.lang.reflect, содержащий классы для представления других элементов программы.

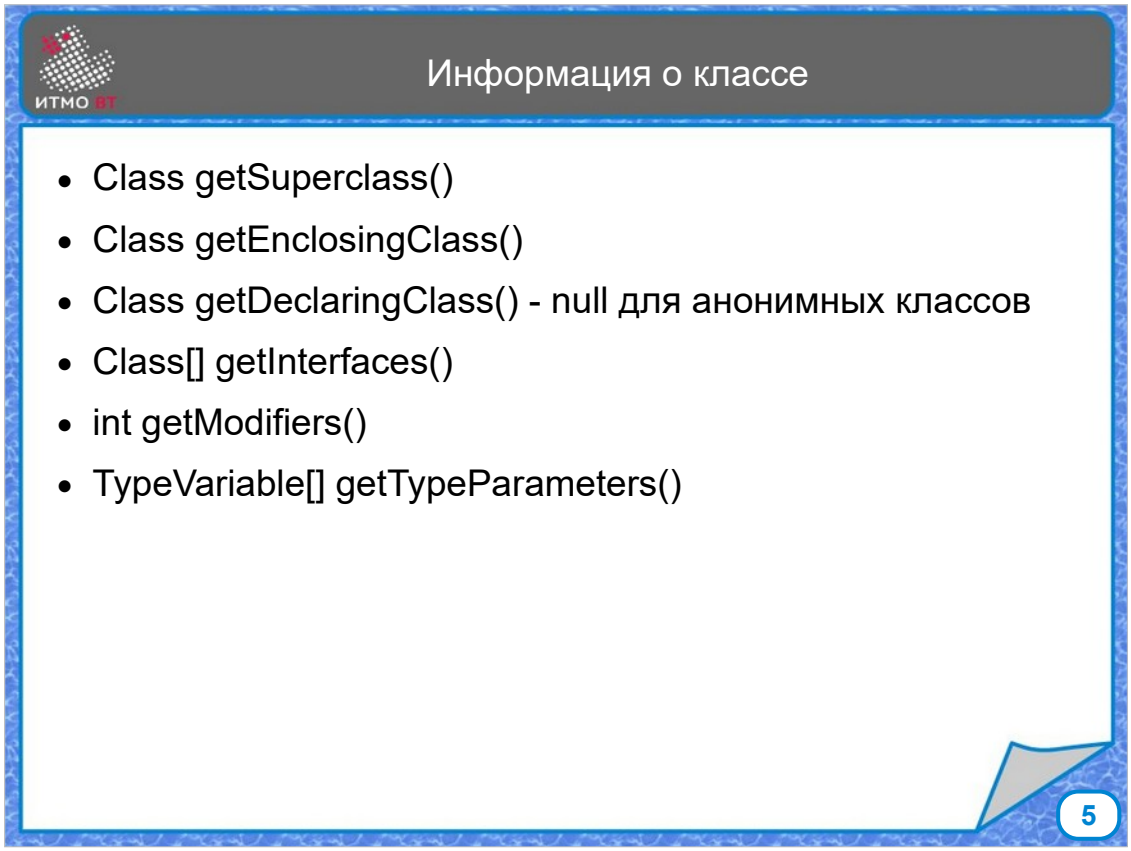


Для работы с типами данных объектов предназначен класс Class. Его экземпляр можно получить несколькими способами:

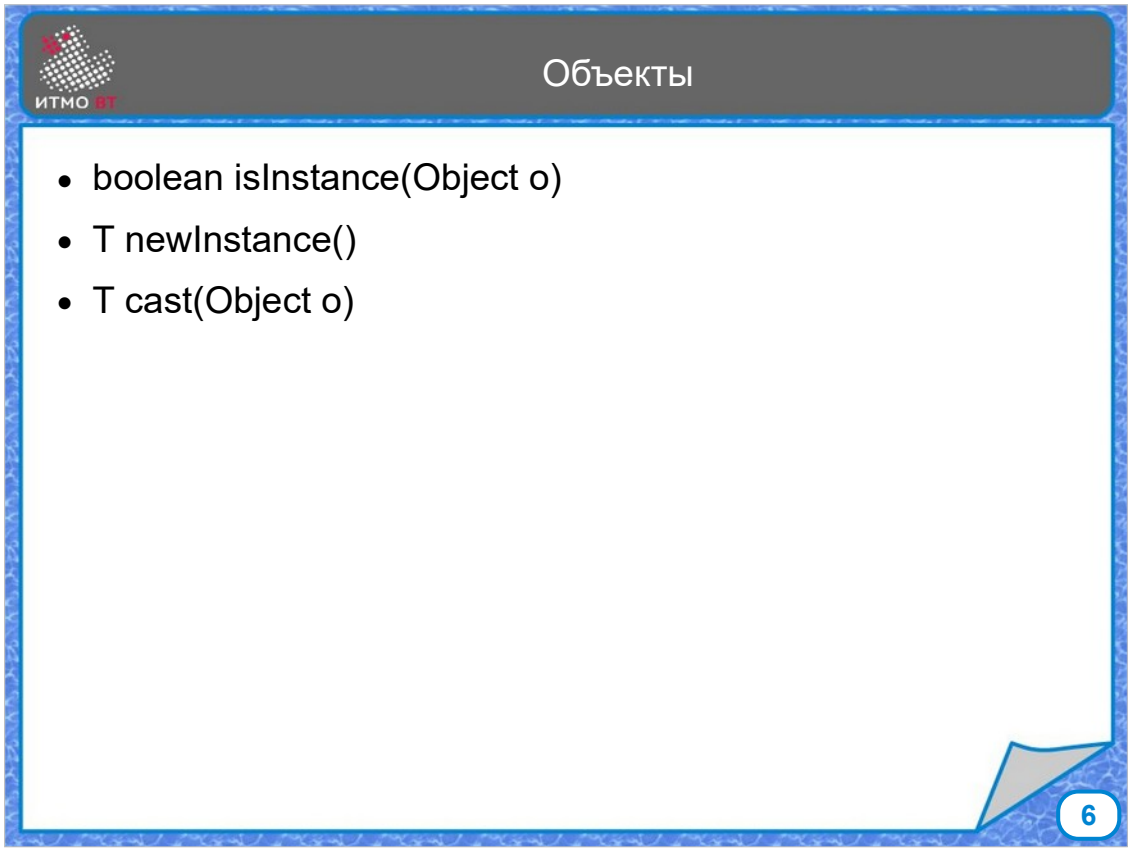
1. Если есть объект - методом getClass()
2. Если есть имя класса - методом Class.forName()
3. С помощью литерала .class для любого типа
4. С помощью константы TYPE для оберток



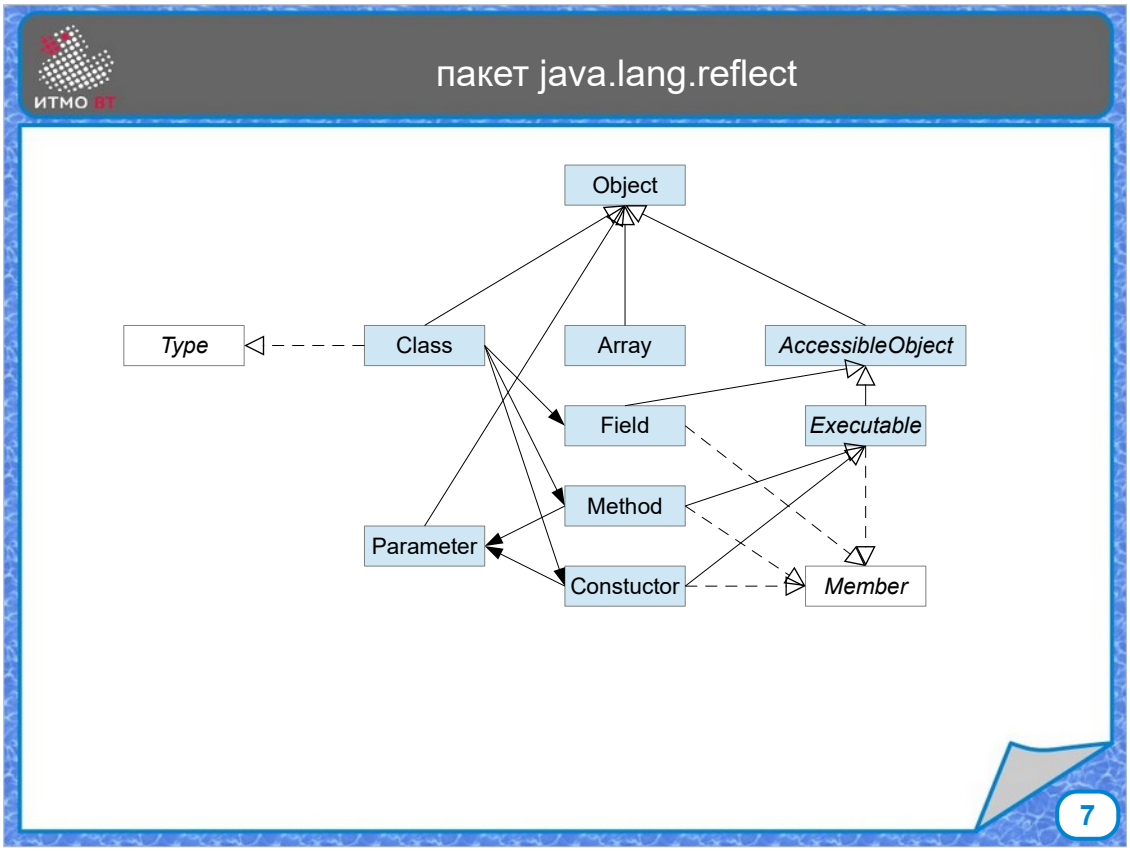
В классе Class определены методы, позволяющие получить информацию о данном классе. Synthetic - это синтетический класс, созданный компилятором, который явно или неявно не объявлен в коде (класс лямбда-выражения). MemberClass - это внутренний класс.



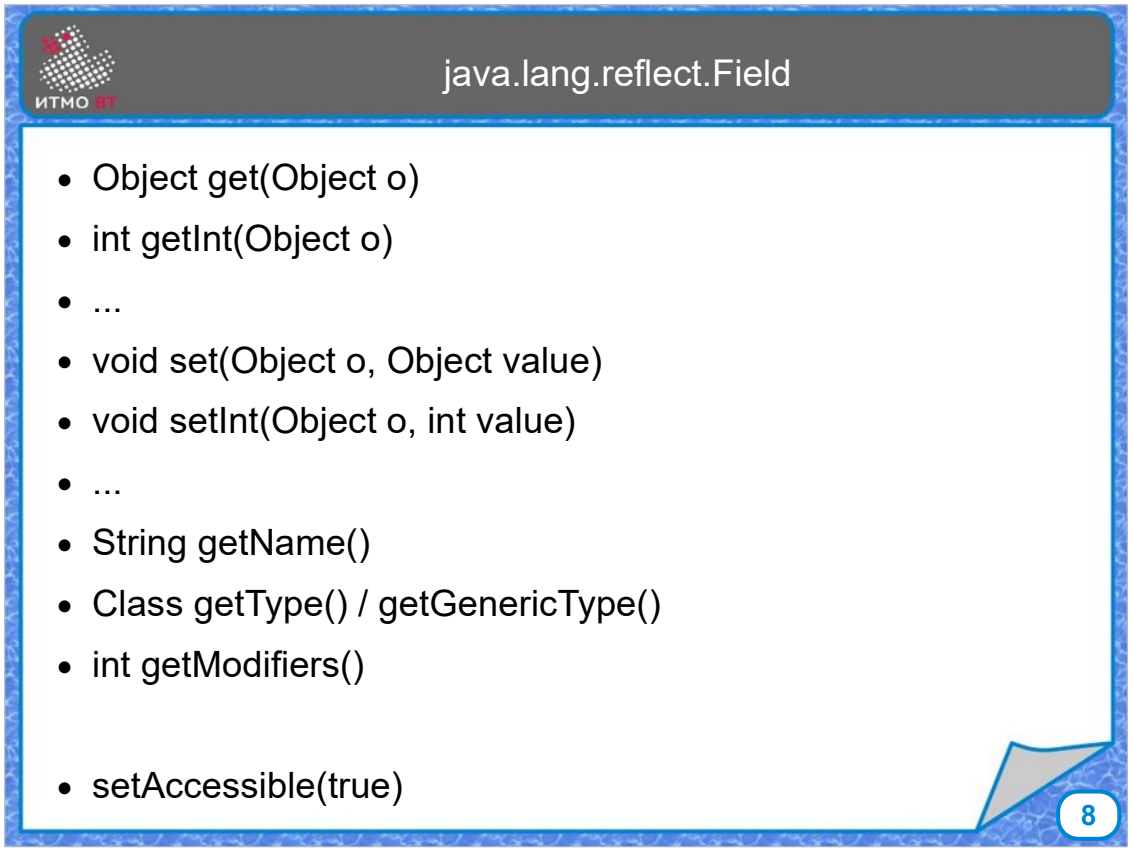
У класса можно получить его суперкласс, окружающий класс (для вложенных), класс, в котором объявлен данный класс (null для анонимных, так как они не объявлены), а также реализованные интерфейсы, модификаторы и параметры типа (обычно только для параметров метода или локальных переменных).



Также для объекта можно проверить, является ли он экземпляром класса, создать новый экземпляр, и привести его к данному типу.

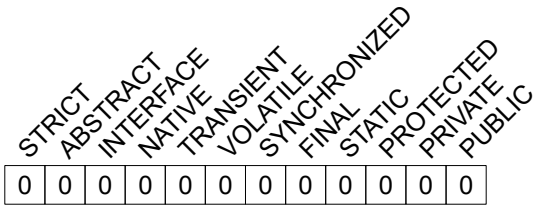


Кроме класса Class есть еще пакет java.lang.reflect, где находятся интерфейсы Type и Member, а также классы, представляющие другие элементы рефлексии: Field, Method, Constructor, Parameter, Array и другие.



С помощью класса Field можно получить или установить значения поля, получить имя, тип и множество модификаторов, а также управлять доступом с помощью метода setAccessible.

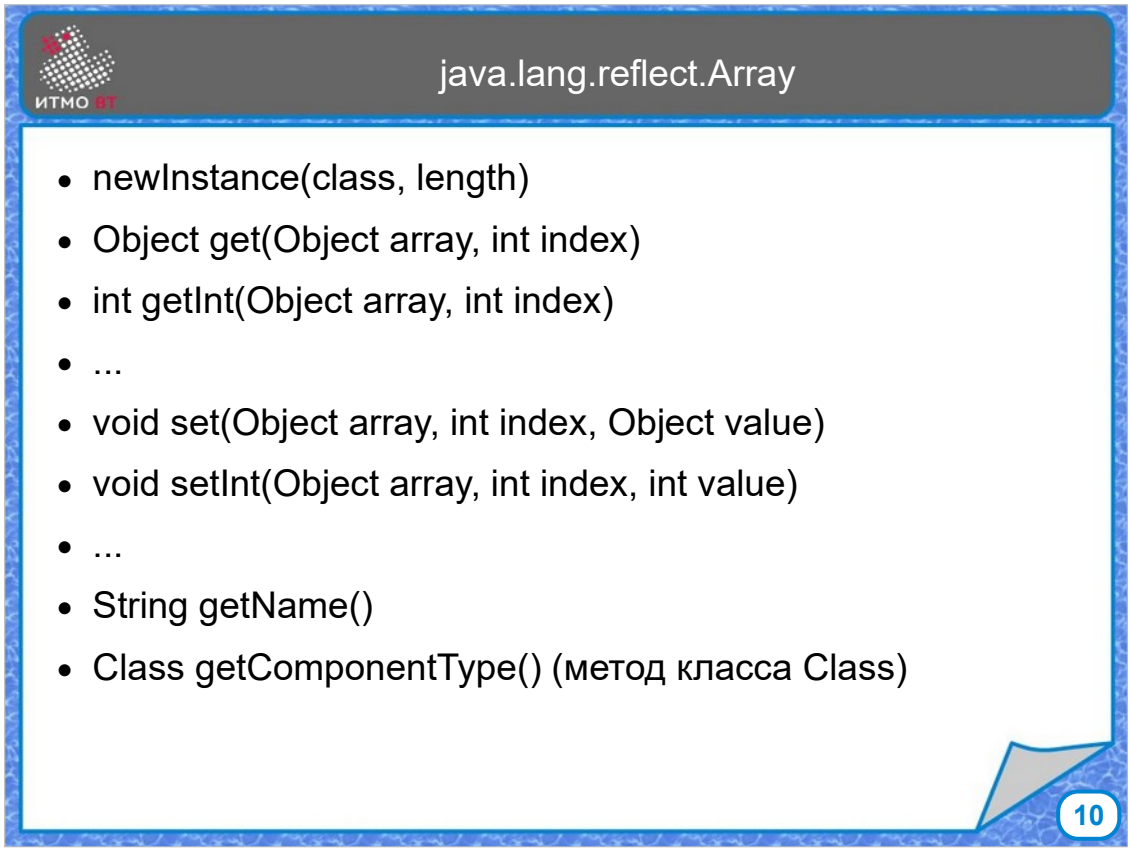
java.lang.reflect.Modifier



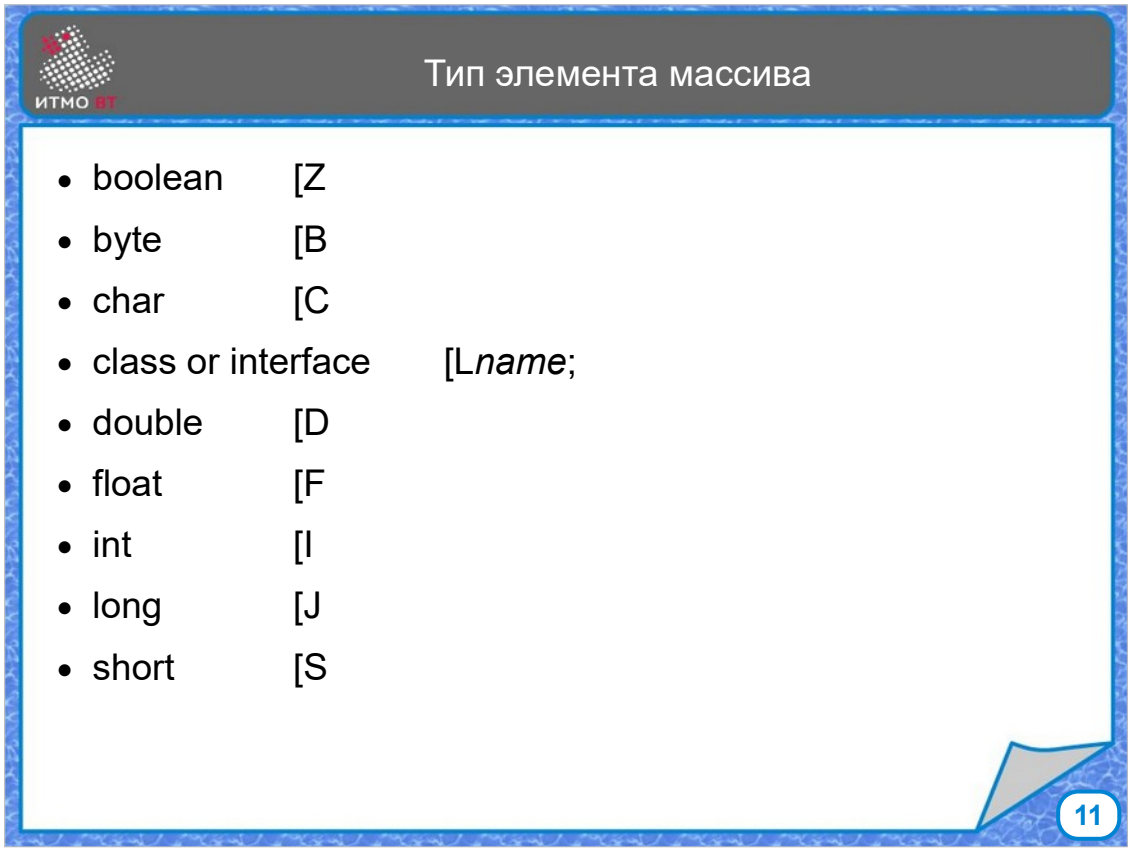
- Modifier.PUBLIC ...
- boolean Modifier.isPublic() ...
- int Modifier.fieldModifiers() ...

```
if (field.getModifiers() &
(Modifier.PUBLIC | Modifier.STATIC | Modifier.FINAL) != 0)
```

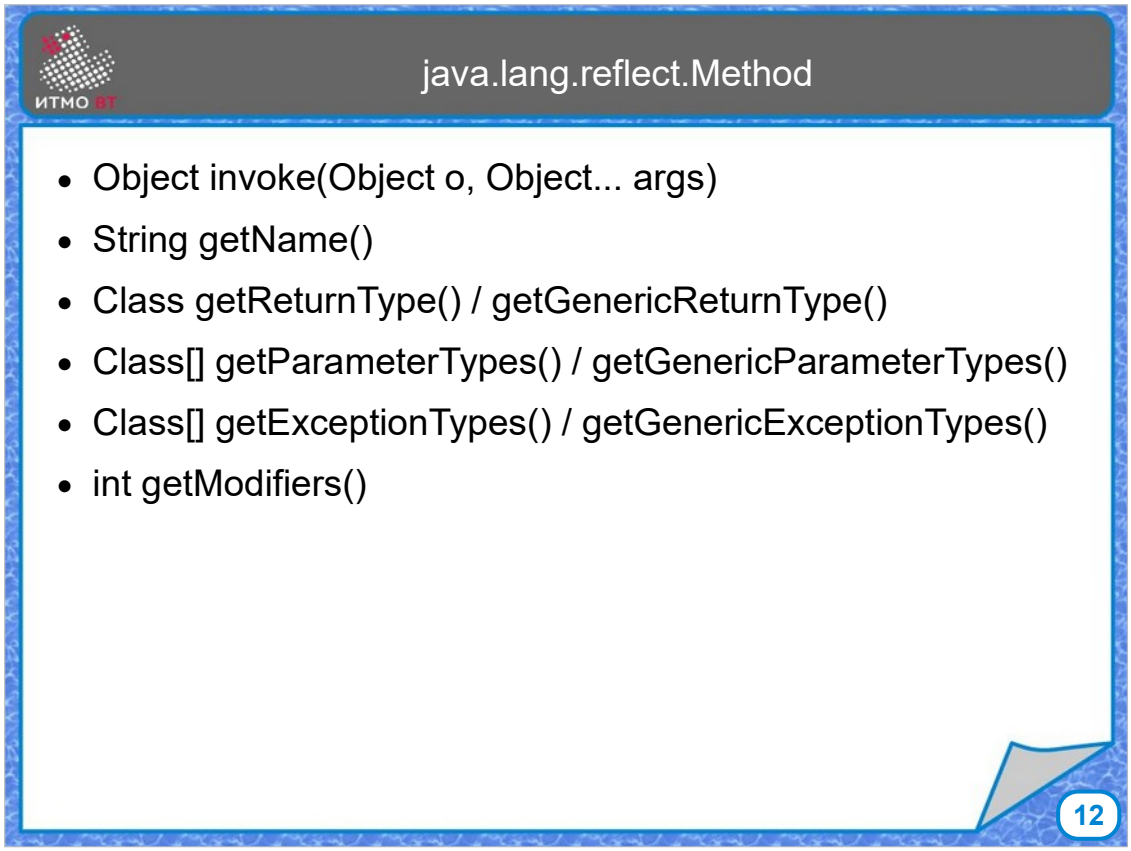

Класс Modifier содержит битовое поле, представляющее множество возможных модификаторов полей, методов и классов.



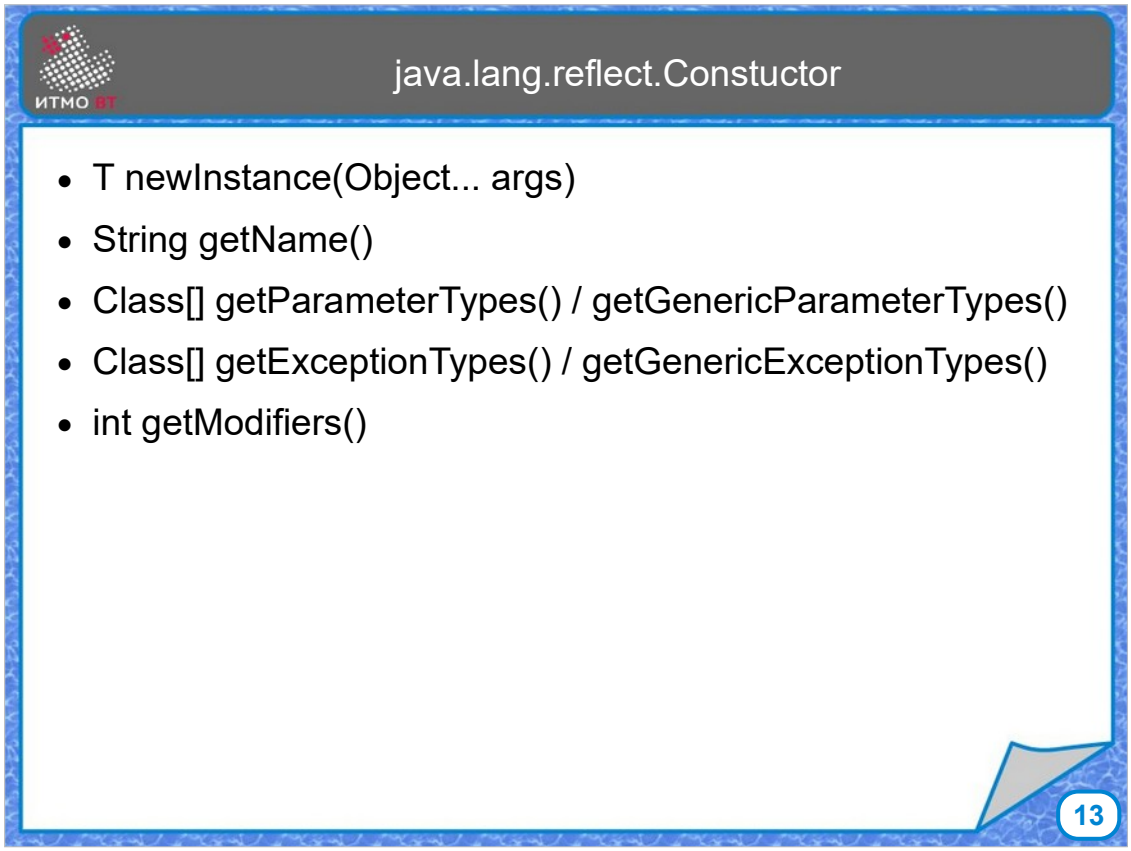
Класс Array позволяет делать почти то же самое, что и класс Field, но для массивов. По индексу можно получать значения элементов массива, либо устанавливать их С помощью метода getComponentType можно узнать тип элемента массива.



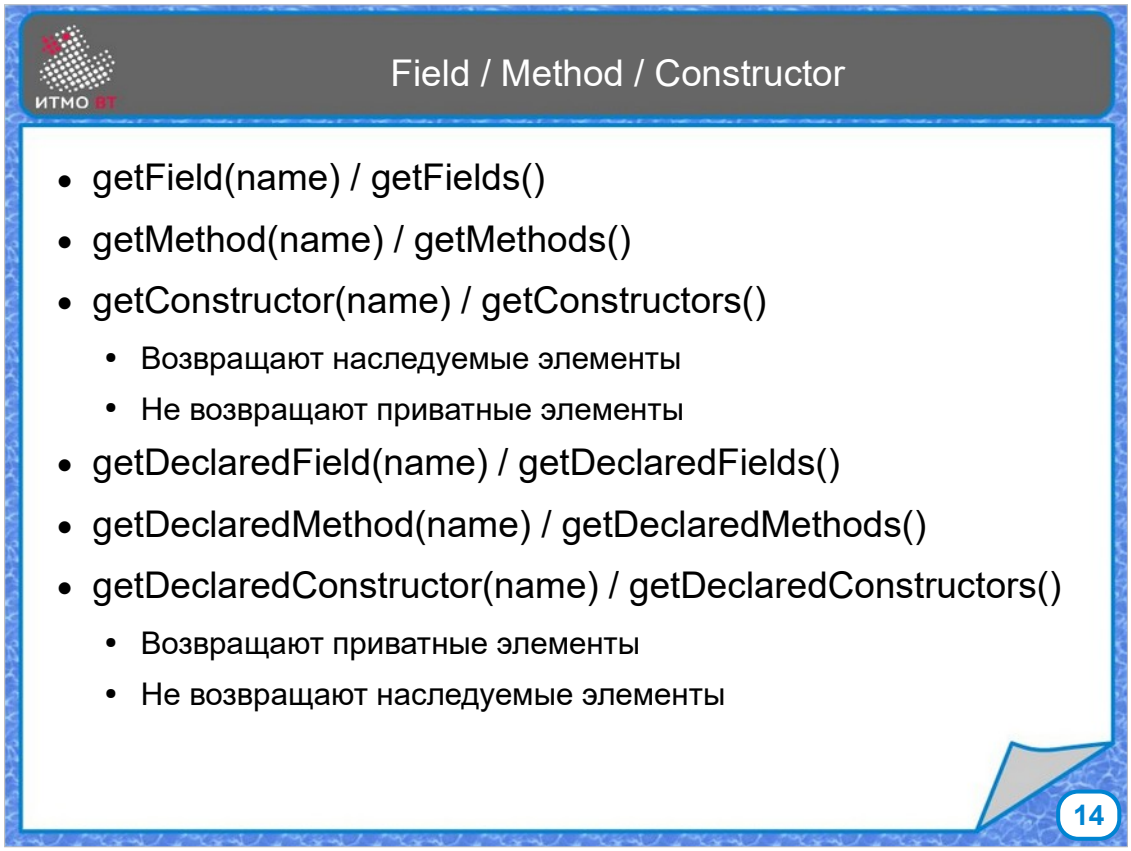
Тип элемента массива обозначается двумя символами - открывающей квадратной скобкой и буквой, которая обозначает тип. L - это ссылочный тип, используется, если в массиве находятся объекты, после буквы L указывается имя типа.



С помощью класса Method можно вызывать методы, получать имя, тип возвращаемого значения, параметры и исключения, которые объявлены в заголовке метода.



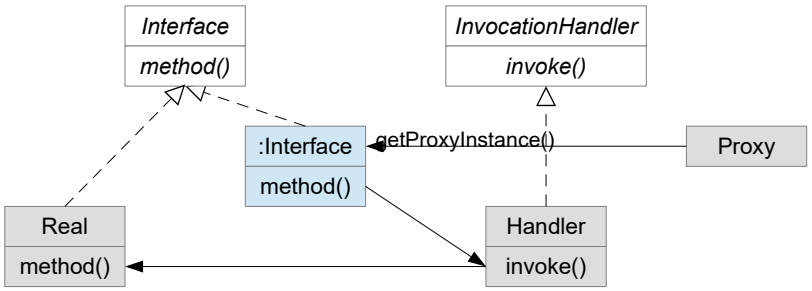
Класс Constructor очень похож на класс Method, только вместо метода invoke у него есть метод newInstance для создания нового объекта.



У класса Class есть целый набор методов, чтобы получить список полей, методов, конструкторов, либо целиком, либо по имени. Отличие методов со словом Declared в названии в том, что например, getMethods() возвращает список доступных методов класса, не включая приватные, так как они не доступны, но включая наследуемые от суперклассов. А метод getDeclaredMethods() вернет список методов, объявленных в этом классе, включая приватные, но не включая наследуемые, так как они в этом классе не объявлены.

Динамические прокси

- класс java.lang.reflect.Proxy
- интерфейс java.lang.reflect.InvocationHandler
- Создает динамический прокси-объект, реализующий заданные интерфейсы, при этом вызовы методов интерфейсов передаются обработчику, реализующему InvocationHandler

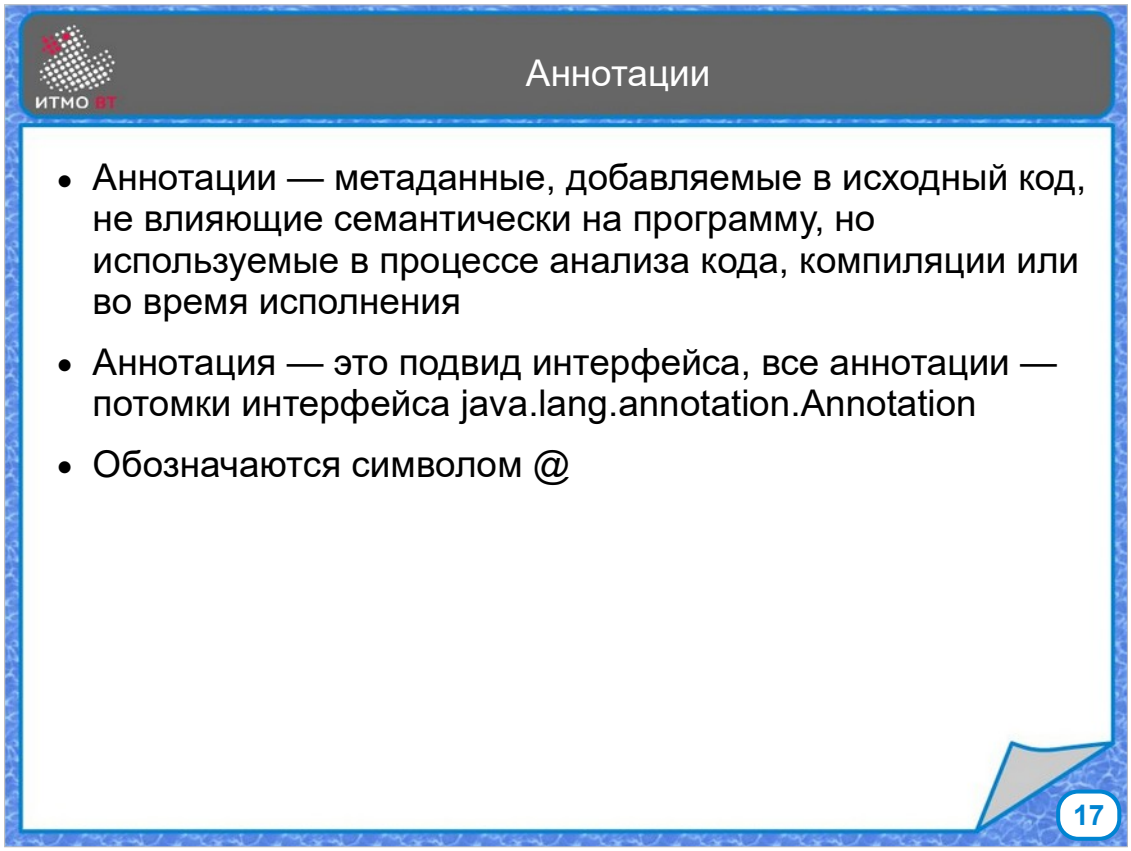


Класс java.lang.reflect.Proxy позволяет динамически создать прокси-объект. для какого-то реального объекта Прокси будет преобразовывать вызовы своих методов в вызовы методов реального объекта.

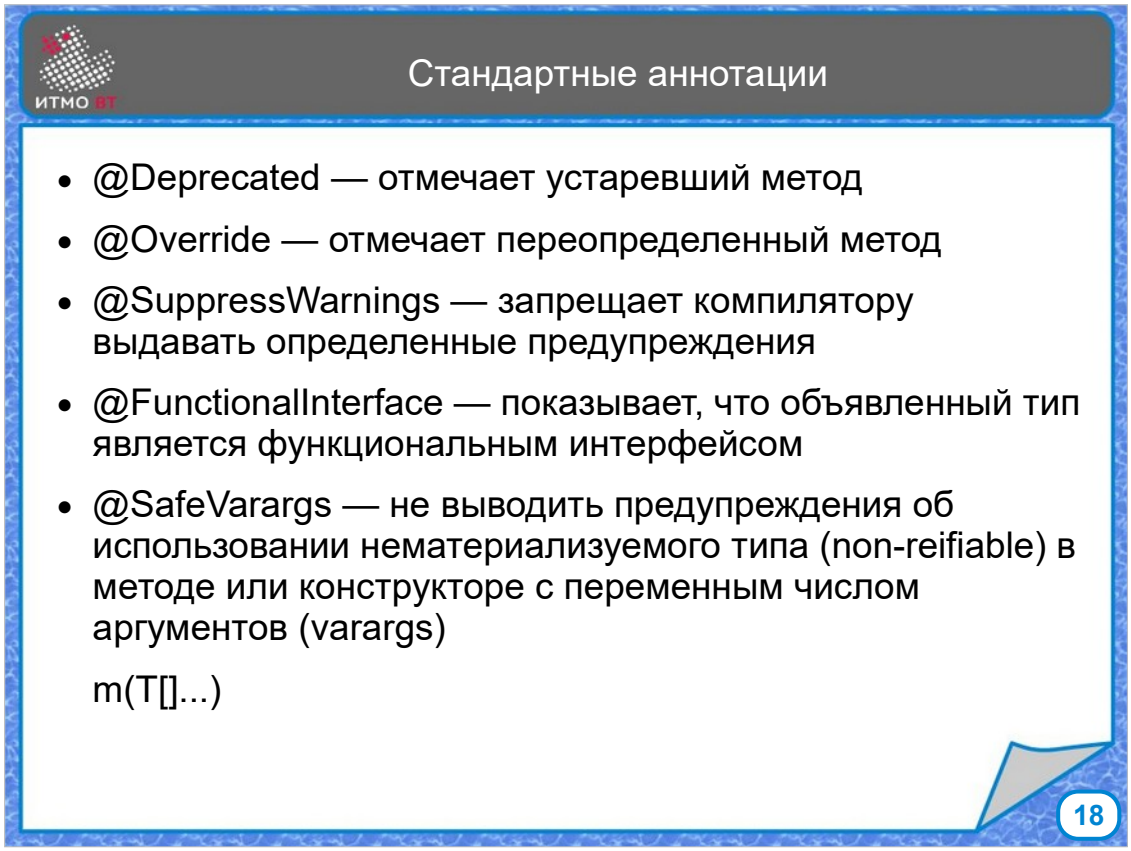
Пример кода

```
// интерфейс
interface Able { public void doIt(); }

// класс, реализующий интерфейс
public class Real implements Able { public doIt() { ... }

// создание прокси
Real real = new Real();
InvokationHandler handler = (proxy, method, args) -> {
// еще действия (логирование)
// можем преобразовать результат .toUpperCase()
  return method.invoke(real, args);
};

Able proxy = (Able)Proxy.newProxyInstance(
  Able.class.getClassLoader(),
  new Class[] { Able.class },
  handler);

proxy.doIt();
```


Для создания динамического прокси нужен некий интерфейс, например Able, в котором есть нужный нам метод, например doIt. Создаем класс, например Real, реализующий этот интерфейс и пишем код метода doIt. Дальше создаем объект реального класса, и создаем обработчик, объект типа InvokationHandler, это можно сделать с помощью лямбда-выражения, параметрами которого будут прокси-объект, метод, который нужно будет вызывать, и аргументы этого метода. Можно добавить какие-то дополнительные действия или преобразования. Лямбда-выражение возвращает результат вызова метода реального объекта. Дальше создаем прокси-объект методом newProxyInstance, которому передаем загрузчик классов, массив с литералом класса Able, и обработчик. После чего можем пользоваться прокси-объектом.



Аннотации - метаданные, которые добавляются в исходный код. На работу программы они не влияют, но позволяют добавить дополнительную информацию, доступную на разных этапах, во время компиляции, во время исполнения. В Java аннотации реализованы как подвид интерфейса, задаются ключевым словом @interface. Они все являются потомками интерфейса Annotation. Имя аннотации начинается с символа @



Определено несколько стандартных аннотаций.

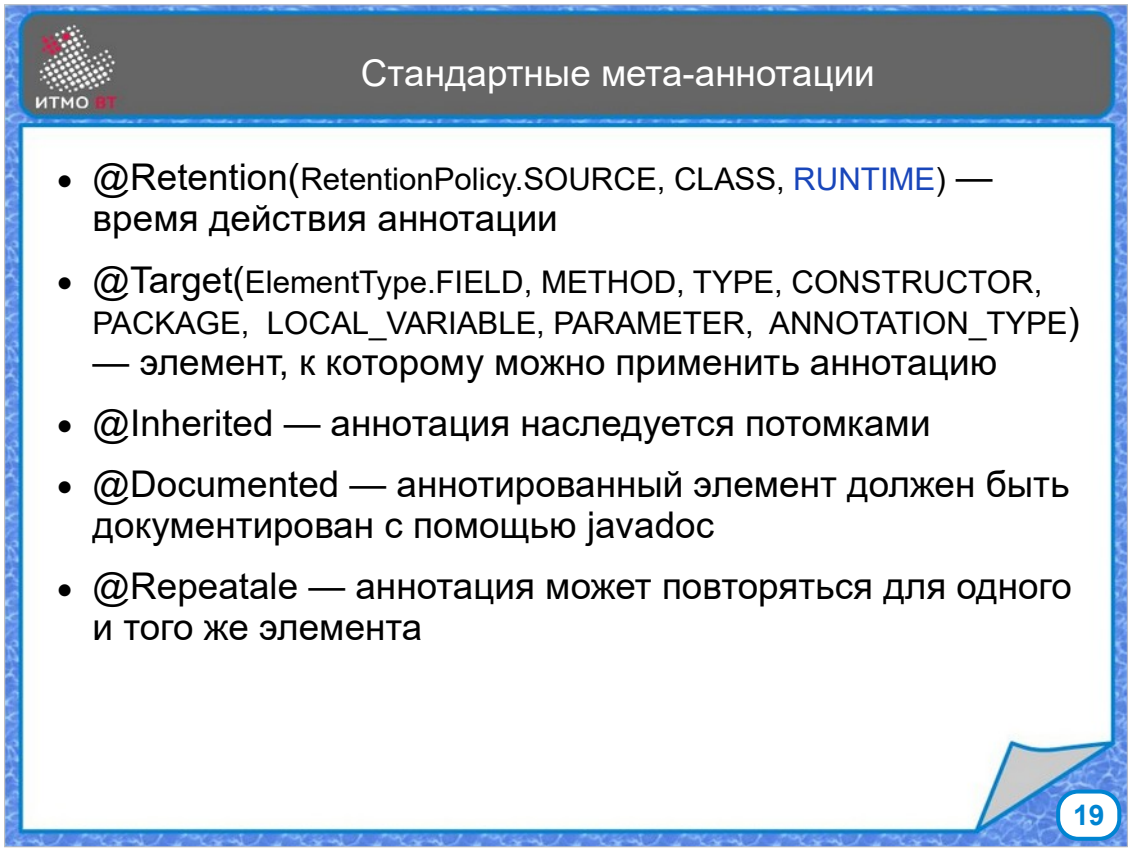
@Deprecated отмечает устаревший элемент.

@Override показывает, что метод переопределенный.

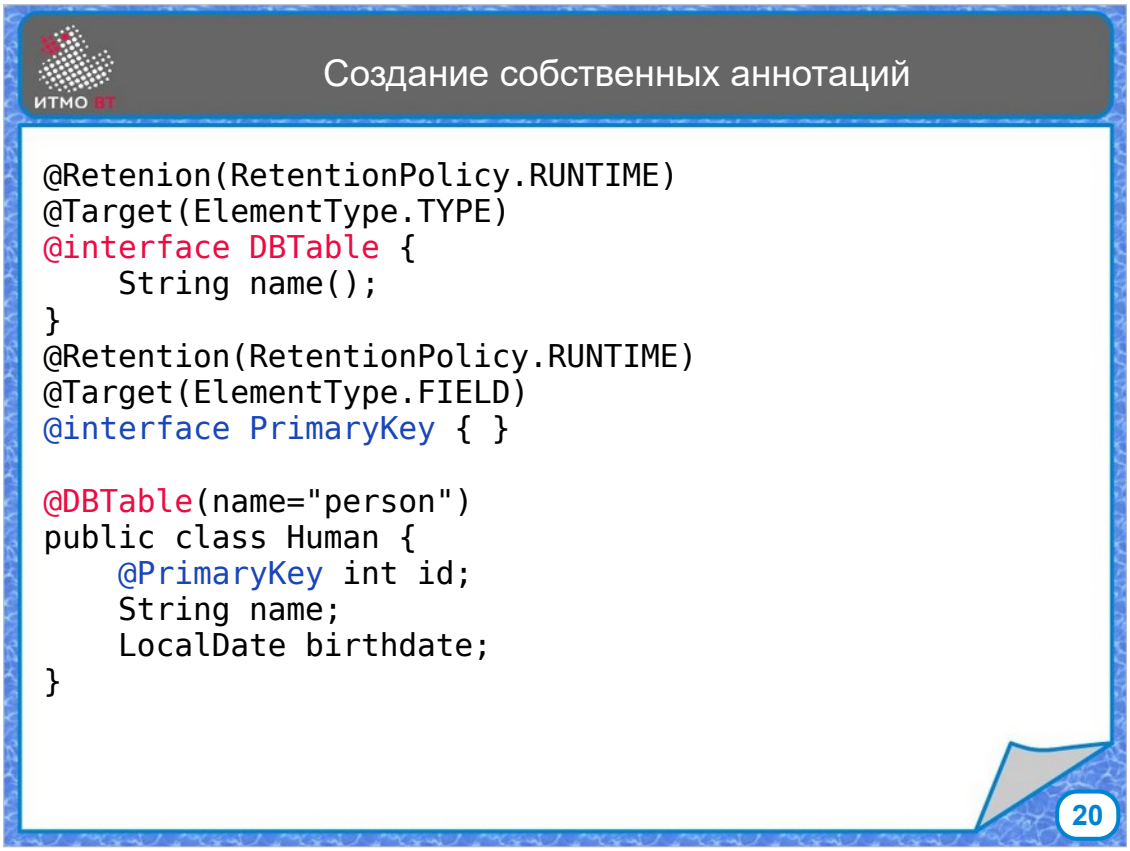
@SuppressWarnings запрещает выдавать предупреждения.

@FunctionalInterface показывает, что интерфейс должен быть функциональным.

@SafeVarargs показывает, что переменное число аргументов обобщенного типа используется безопасно. При использовании переменного числа аргументов в метод передается массив аргументов. Если это дженерик тип, то использовать элементы по отдельности безопасно, а как массив - нет. Компилятору сложно разобраться с безопасностью использования, поэтому он будет выдавать предупреждение при отсутствии аннотации.

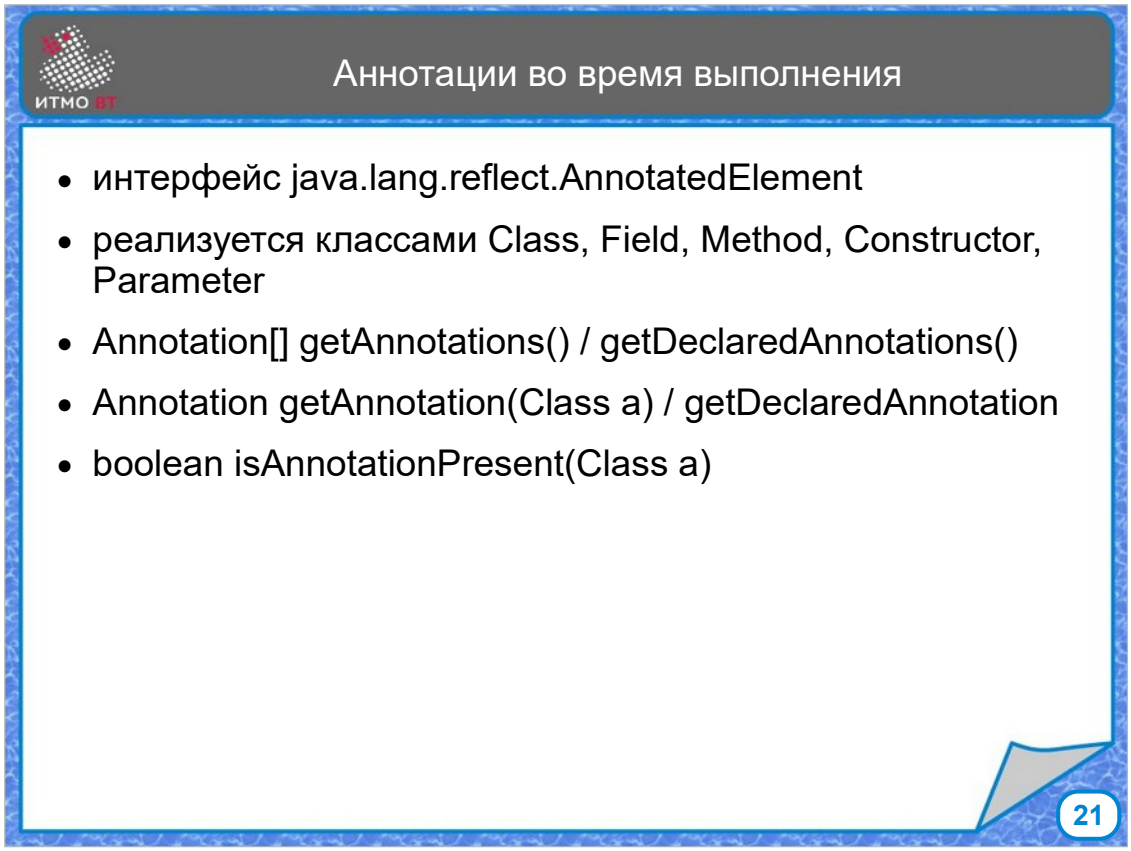


Кроме простых аннотаций, есть мета-аннотации, которые используются при написании аннотаций. Они показывают время действия аннотации, элемент, к которому она может применяться, наследуется ли она, нужна ли документация, а также можно ли повторить аннотацию для того же элемента, но с другими параметрами.

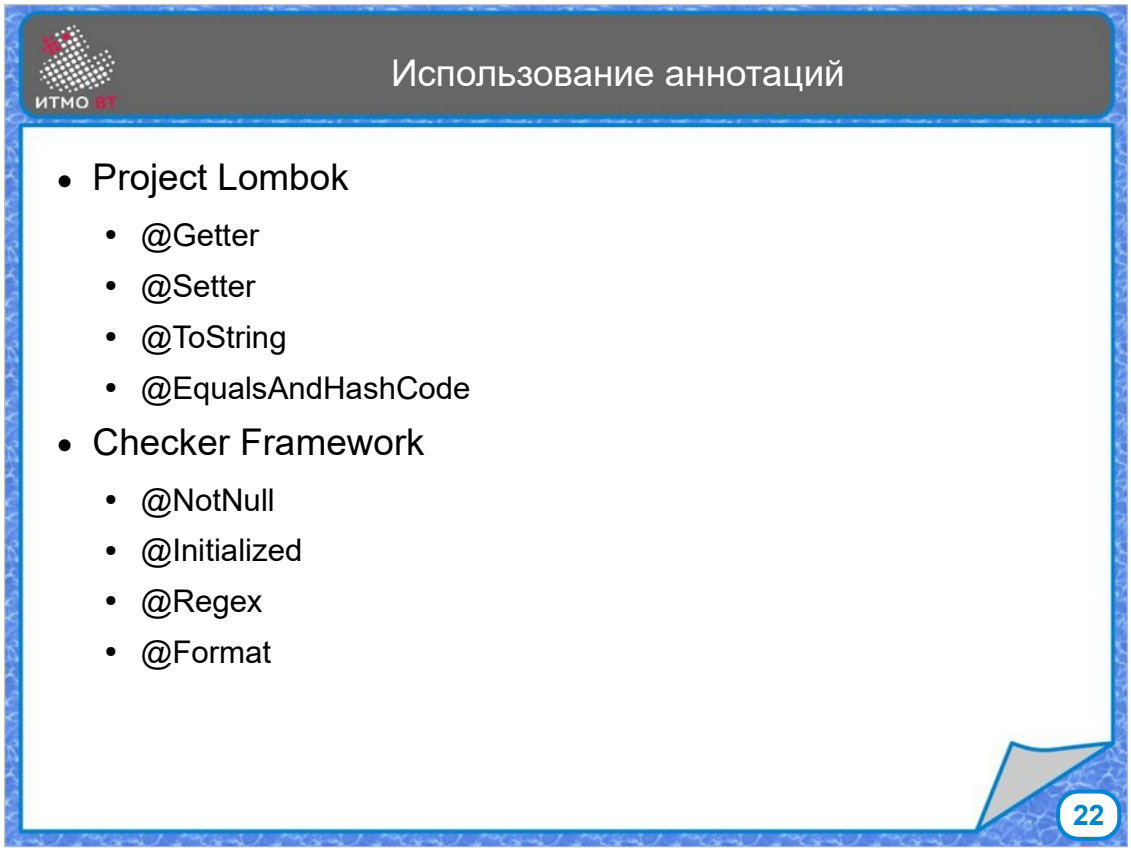


В данном примере создаются 2 аннотации - DBTable, которая будет показывать, что объекты класса, помеченного аннотацией могут храниться в таблице базы данных с указанным именем, и аннотация PrimaryKey, показывающая, что поле, помеченной этой аннотацией, является первичным ключом таблицы. При создании аннотаций указывается, что информация об аннотации должна быть доступна во время исполнения программы, указываются, какие элементы можно помечать аннотациями, и задаются параметры аннотации.

При создании нового класса ему и его элементам можно указать только что созданные аннотации.



Во время выполнения с помощью рефлексии можно получать список аннотаций классов, методов, полей.



Имеются дополнительные библиотеки, использующие аннотации. Например, проект Lombok содержит аннотации для автоматической генерации геттеров, сеттеров, методов toString, equals и hashCode. Фреймворк Checker позволяет аннотациями задавать допустимые значения полей.