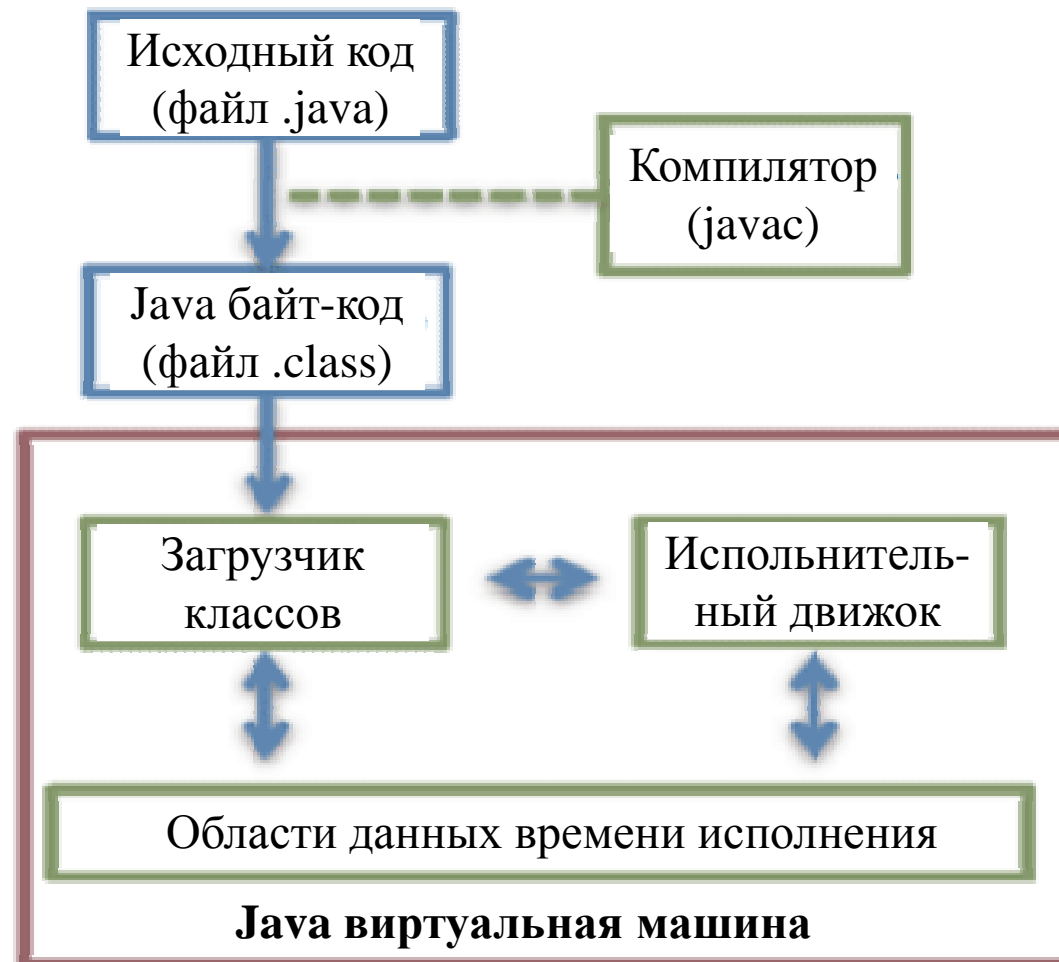


# **Процесс загрузки классов**

## Процесс загрузки классов

# ЗАПУСК НА ИСПОЛНЕНИЕ ПРОГРАММЫ

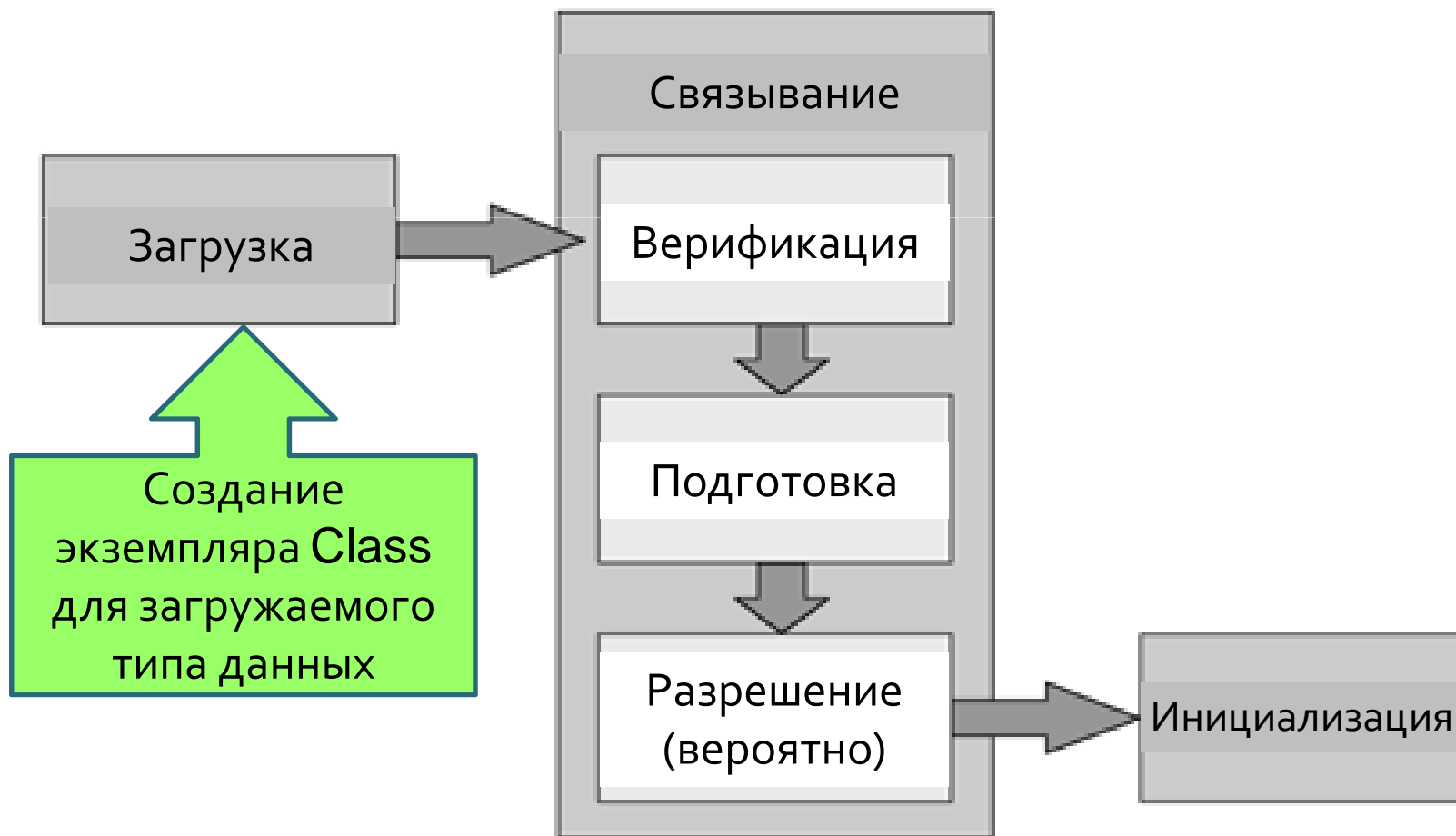
Запуск программы на исполнение проходит следующие шаги:



## Процесс загрузки классов

### ПОДГОТОВКА ТИПОВ ДАННЫХ К ИСПОЛЬЗОВАНИЮ

- Java виртуальная машина выполняет следующие действия для подготовки типов данных к использованию:



## Процесс загрузки классов

- ❑ **ЗАГРУЗКА** – это процесс нахождения файла **.class**, который представляет тип класса или интерфейса с указанным именем, и его чтения в массив байтов (байты обрабатываются, чтобы подтвердить, что они представляют собой экземпляр типа **Class**);
- ❑ **СВЯЗЫВАНИЕ** – это процесс включения двоичных данных типа в состояние выполнения виртуальной машиной:
  - ✓ **Верификация** – это подтверждение, что представление класса/интерфейса является структурно правильным и подчиняется семантическим требованиям языка программирования Java и JVM (позволяет избежать необходимости выполнять многократные проверки при выполнении байт-кода);

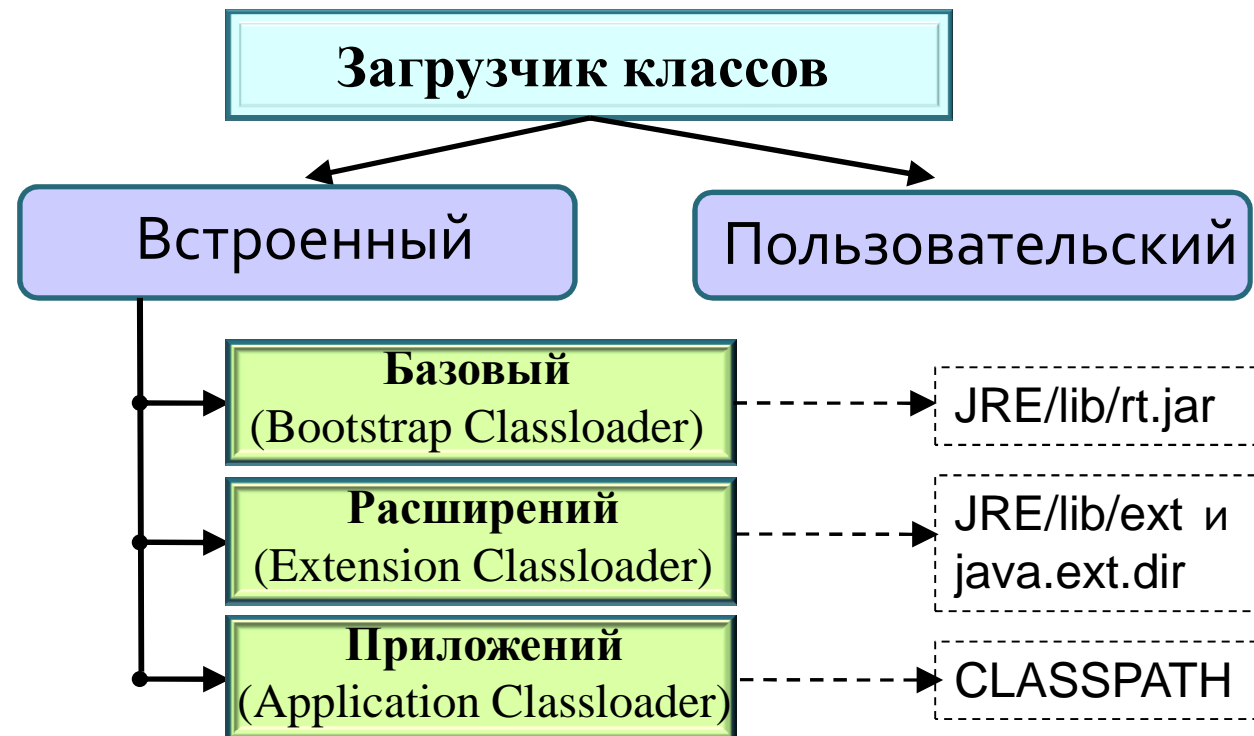
## Процесс загрузки классов

- **Подготовка** – это выделение памяти для статического хранилища класса (статические данные и таблица методов);
- ***Разрешение*** – это проверка символьных ссылок на ссылаемые классы или интерфейсы в пуле констант на корректность и их преобразование в прямые ссылки (исполнение этого шага может быть отложено до времени реального использования ссылки, т.е. после этапа инициализации);
- ❑ ***ИНИЦИАЛИЗАЦИЯ*** – это процесс установки переменных класса в их начальные значения (т.е. выполнение метода `<clinit>` ).


## Процесс загрузки классов

### ЗАГРУЗЧИКИ КЛАССОВ

- ❑ *Загрузчик классов* – это компонент java виртуальной машины, задача которого найти и загрузить класс для исполнения.



## Процесс загрузки классов

- ❑ Загрузчики классов связаны иерархической структурой;
- ❑ Базовый загрузчик классов **Bootstrap** является родителем для всех загрузчиков классов;
- ❑ Каждый загрузчик классов ведет список (свое пространство имен - кэш) загруженных им классов;
-  ❑ Для загрузки классов используется модель делегирования загрузки, которая гарантирует, что:
  - каждый класс будет загружен только один раз;
  - класс будет загружен тем загрузчиком классов, который ближе всего расположен в иерархии к базовому загрузчику;
  - поиск классов будет производиться в источниках в порядке их доверия (начиная с наиболее надежного).

## Процесс загрузки классов

### МОДЕЛЬ ДЕЛЕГИРОВАНИЯ ЗАГРУЗКИ КЛАССОВ

- ❑ Запрос на загрузку всегда приходит к младшему загрузчику (нижнему подклассу, *например*, **Application ClassLoader**), который сначала проверяет свое пространство загруженных классов:
  - если класс не найден, то запрос передается (делегируется) его родителю;
  - если класс был уже загружен, то происходит возврат этого класса;
- ❑ Каждый загрузчик повторяет проверку и в случае неудачи передает запрос своему родителю. Так происходит до тех пор, пока самый старший загрузчик классов (суперкласс **Bootstrap**) тоже не обнаружит у себя в кэше требуемый класс;

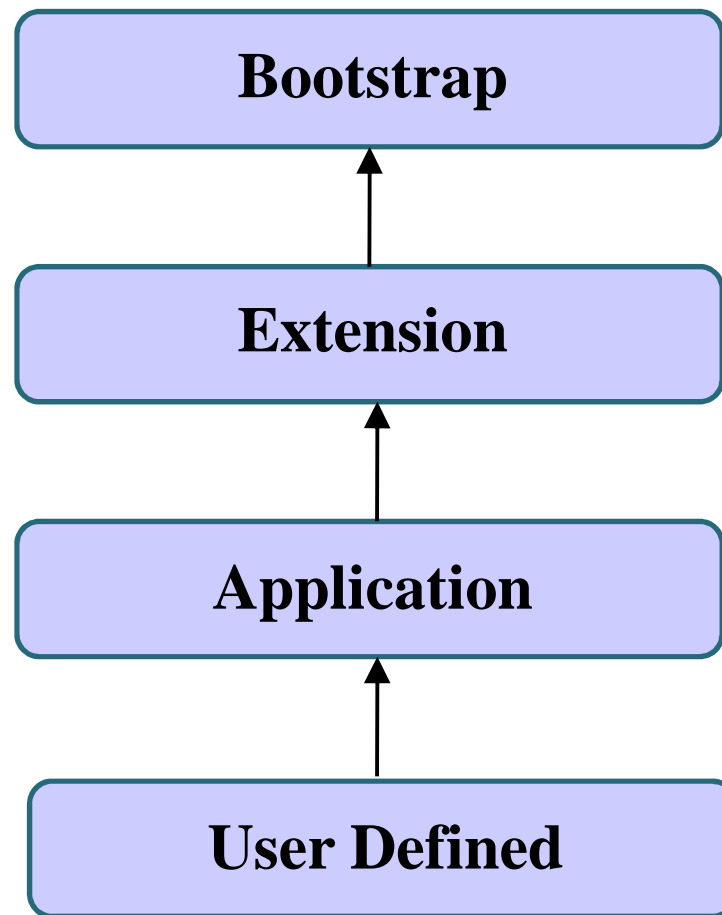


## Процесс загрузки классов

- ❑ Начальный загрузчик (**Bootstrap**) пытается найти в своей локации требуемый класс:
  - если класс обнаружен, то он загружается и возвращается;
  - если класс не найден, то поиск передается (делегировается) его потомку;
- ❑ Каждый загрузчик повторяет эти действия;
- ❑ Если запрос вернулся самому младшему загрузчику классов (например, **Application ClassLoader**), то при неуспешном поиске в своей локации требуемого класса он возвращает ошибку типа **ClassNotFoundException**.

## Процесс загрузки классов

### ИЕРАРХИЯ ЗАГРУЗЧИКОВ КЛАССОВ





# Рефлексия

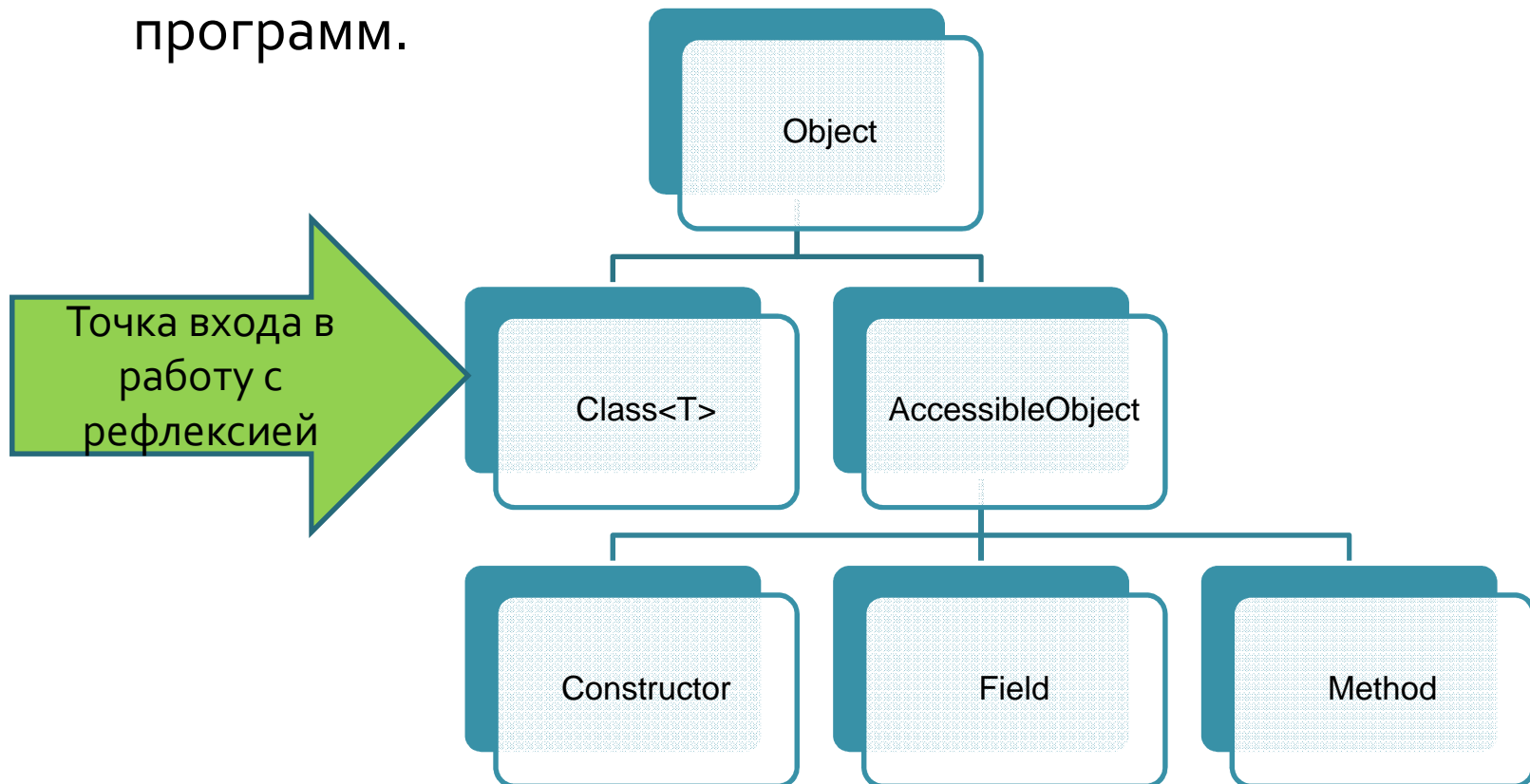


## Рефлексия

- ❑ **Рефлексия** (отражение) - это процесс, во время которого программа может отслеживать и модифицировать собственную структуру и поведение во время выполнения.
- ❑ С помощью рефлексии можно делать следующее:
  - Определить класс объекта;
  - Получить информацию о модификаторах класса, поля, метода, конструктора и суперкласса;
  - Выяснить, какие константы и методы принадлежат интерфейсу;
  - Создать экземпляр класса, имя которого неизвестно до момента выполнения программы;
  - Получить и установить значение свойства объекта;
  - Вызвать метод объекта;
  - Создать новый массив, размер и тип компонентов которого неизвестны до момента выполнения программ

## Рефлексия

- В java API рефлексии находится в пакете **java.lang.reflect** и используется для просмотра информации о классах, интерфейсах, методах, полях, конструкторах, аннотациях во время выполнения java программ.



## Рефлексия

- ❑ Тип **Class** – это тип, который представляет метаданные, т.е. с помощью этого типа описывается структура другого типа данных;
- ❑ Все классы java имеют связанный с ними объект типа **Class**;
- ❑ Класс **Class** не имеет открытых конструкторов и строится автоматически загрузчиком классов;
- ❑ Существует несколько способов получить объект типа **Class**, в зависимости от того, имеется ли доступ к:
  - ✓ объекту;
  - ✓ имени класса;
  - ✓ типу;
  - ✓ другому существующему объекту **Class**.

## Рефлексия

### СПОСОБЫ ПОЛУЧЕНИЯ ЭКЗЕМПЛЯРА Class

#### I) Через существующий объект класса

- Нужно вызвать метод `getClass()` на объекте

*Например,*

```
Student myStud = new Student("Ivan", 213, 19);
```

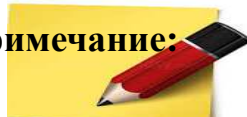
```
Class<?> clazz1 = myStud.getClass();
```

```
double[] arrayDouble = new double[10];
```

```
Class<?> clazz2 = arrayDouble.getClass();
```

```
Class<?> clazz3 = "foo".getClass();
```

Примечание:



Этот способ работает только для ссылочных  
ТИПОВ!

## Рефлексия

### II) Через доступное имя типа (нет доступа к объекту)

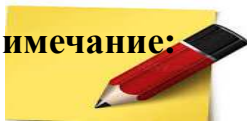
- Нужно к имени типа добавить нотацию **.class**

*Например,*

```
Class<?> clazz1 = Student.class;
```

```
Class<?> clazz2 = int[].class;
```

Примечание:



Этот способ работает и для примитивных типов данных:

*Например,*

```
Class<?> clazz3 = int.class;
```



Экземпляр типа **Class** для примитивного типа можно получить и через поле **TYPE** соответствующего класса-обертки:

```
Class<?> clazz4 = Integer.TYPE;
```



## Рефлексия

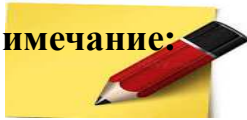
### III) Через вызов статического метода *Class.forName()*

- Нужно этому методу передать полное квалификационное имя типа как строку символов:

Например,

```
Class<?> clazz1 = Class.forName("java.lang.Thread");
```

Примечание:



Этот способ не работает для примитивных типов данных!



Для массивов в качестве квалификационного типа можно использовать следующий синтаксис:

```
Class<?> cDoubleArray = Class.forName("[D");
```

Это тоже, что и **double[].class**

## Рефлексия

IV) Через уже существующий экземпляр типа **Class** (т.е. используя средства отражения)

- Нужно вызвать метод `getSuperclass()` на объекте подкласса для получения описания суперкласса

*Например,*

```
Class<?> sclazz = Double.class.getSuperclass();
```

Результат будет типом **java.lang.Number**.

- Нужно вызвать метод `getEnclosingClass()` для получения ограждающего класса

*Например,*

```
Class<?> clazz = Map.Entry.class.getEnclosingClass();
```

Результат будет типом **java.util.Map**

## Рефлексия

- Можно использовать метод *getEnclosingClass()* для получения ограждающего класса для анонимного класса:

Например,

```
public class MyClass {  
    public static void main(String[] args) {  
        Comparator<Integer> comparator = new Comparator<>() {  
            public int compare(Integer o1, Integer o2) {  
                return o2 - o1;  
            }  
        };  
        Class clazz = comparator.getClass().getEnclosingClass();  
    }  
}
```

Результат будет типом **MyClass**

## Рефлексия

- Нужно вызвать метод *getClasses()* для получения *public* типов, которые объявлены как члены класса, включая и наследников

*Например,*

```
Class<?>[] clazz1 = Character.class.getClasses();
```

Результат будут типы **Character.Subset**,  
**Character.UnicodeBlock** и **Character.UnicodeScript**.

- Нужно вызвать метод *getDeclaredClasses()* для получения всех типов, которые объявлены в этом классе

*Например,*

```
Class<?>[] clazz2 = Map.class.getDeclaredClasses();
```

Результат будет типом **java.util.Map\$Entry**

## Рефлексия

Рассмотрим класс, который будет исследоваться через рефлексияю:

```
public final class MyTestReflect {  
    public String str = "data";  
    protected double pr = 100.5;  
    private int index = 2016001;  
    public MyTestReflect() { }  
    public MyTestReflect(int index) {  
        this.index = index;  
    }  
    public MyTestReflect(double pr, int index) {  
        this.pr = pr;  
        this.index = index;  
    }  
    public String toString() {  
        return "str = " + str + ", pr = " + pr + ", index = " + index;  
    }  
}
```

// ....

## Рефлексия

// ...

```
public double getPr() {  
    return pr;  
}  
public int getIndex() {  
    return index;  
}  
public void setIndex(int index) {  
    this.index = index;  
}  
public void calcPr(int coef) {  
    this.pr = mult(coef);  
}  
private double mult(int coef) {  
    return this.index / coef * 2.0;  
}  
}
```

## Рефлексия

### ПОЛУЧЕНИЕ ИНФОРМАЦИИ О МОДИФИКАТОРАХ

- ❑ Необходимо вызвать метод *getModifiers()* на экземпляре **Class**:

- набор модификаторов возвращается в виде целого числа с разными битовыми позициями, отражающими различные модификаторы;

*Например,* `int mods = clazz.getModifiers();`

- ❑ Использовать класс **Modifier** из пакета **java.lang.reflect**, который содержит статические методы и константы для декодирования модификаторов доступ к классу и членам класса.

### Класс Modifier

<i>Имя константы</i>	<i>Название метода</i>	<i>Модификатор</i>
ABSTRACT	isAbstract	abstract
FINAL	isFinal	final
INTERFACE	isInterface	interface
NATIVE	isNative	native
PRIVATE	isPrivate	private
PROTECTED	isProtected	protected
PUBLIC	isPublic	public
STATIC	isStatic	static
STRICT	isStrict	strictfp
SYNCHRONIZED	isSynchronized	synchronized
TRANSIENT	isTransient	transient
VOLATILE	isVolatile	volatile



## Рефлексия

Пример 1:

```
public static void main(String[] args) {  
    Class<?> clazz = MyTestReflect.class;  
    int modifiers = clazz.getModifiers();  
  
    if (Modifier.isPublic(modifiers)) {  
        System.out.println("public ");  
    }  
  
    if (Modifier.isAbstract(modifiers)) {  
        System.out.println("abstract ");  
    }  
  
    if (Modifier.isFinal(modifiers)) {  
        System.out.println("final ");  
    }  
}
```

**Вывод в консоли:**  
public  
final

### ПОЛУЧЕНИЕ ИНФОРМАЦИИ ОБ ЭЛЕМЕНТЕ ТИПА

- ❑ В пакете **java.lang.reflect** находится интерфейс **Member**, который отражает идентификационную информацию об одном элементе (поле или методе) или конструкторе;
- ❑ Это интерфейс реализуется классами **java.lang.reflect.Field**, **java.lang.reflect.Constructor**, **java.lang.reflect.Method**;
- ❑ Эти классы используются для предоставления информации о самом элементе и о динамическом доступе к нему.

## Рефлексия

### I) Получение информации о поле

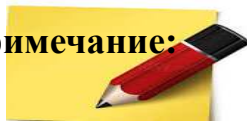
- ❑ Необходимо вызвать метод *getFields()* на экземпляре **Class**:
  - возвращает массив объектов типа **Field**, отражающих все **public** поля, объявленные в классе или интерфейсе;

*Например,* `Field[] fields = clazz.getFields();`

- ❑ Необходимо вызвать метод *getDeclaredFields()* на экземпляре **Class**:
  - возвращает массив объектов типа **Field**, отражающих все поля, объявленные в классе или интерфейсе;

*Например,* `Field[] fields = clazz.getDeclaredFields();`

Примечание:



**Отраженное поле может быть полем класса (статическим) или полем экземпляра!**

## Рефлексия

### Пример 2:

```
System.out.println("Public fields:");
Field[] fields = clazz.getFields();
for (Field field : fields) {
    Class<?> fieldType = field.getType();
    System.out.println("\tName: " + field.getName());
    System.out.println("\tType: " + fieldType.getName());
}
```

#### Вывод в консоли:

Public fields:

    Name: str

    Type: java.lang.String

## Рефлексия

### Пример 3:

```
System.out.println("All fields:");  
Field[] fields = clazz.getDeclaredFields();  
for (Field field : fields) {  
    Class<?> fieldType = field.getType();  
    System.out.println("\tName: " + field.getName());  
    System.out.println("\tType: " + fieldType.getName());  
}
```

#### Вывод в консоли:

All fields:

```
Name: str  
Type: java.lang.String  
Name: pr  
Type: double  
Name: index  
Type: int
```

## Рефлексия

### II) Получение информации о конструкторе

- ❑ Необходимо вызвать метод *getConstructors()* на экземпляре **Class**:
  - возвращает массив объектов типа **Constructor**, отражающих все **public** конструкторы, объявленные в классе;

*Например,* `Constructor[] constrs = clazz.getConstructors();`

- ❑ Необходимо вызвать метод *getDeclaredConstructors()* на экземпляре **Class**:
  - возвращает массив объектов типа **Constructor**, отражающих все конструкторы, объявленные в классе;


*Например,* `Constructor[] constrs = clazz.getDeclaredConstructors();`

## Рефлексия

### Пример 4:

```
Constructor<?>[] constrs = clazz.getConstructors();
int i = 0;
for (Constructor<?> constructor : constrs) {
    System.out.print("Constructor " + (++i) + " : ");
    Class<?>[] paramTypes = constructor.getParameterTypes();
    for (Class<?> paramType : paramTypes) {
        System.out.print(paramType.getName() + " ");
    }
    System.out.println();
}
```

Получение  
информации  
о типах  
параметров



#### Вывод в консоли:

```
Constructor 1 :
Constructor 2 : double int
Constructor 3 : int
```

## Рефлексия

### III) Создание объекта класса через рефлексию

- ❑ Необходимо получить объект типа **Constructor**, который отражает требуемый конструктор для создания объекта:
  - применение метода *getConstructor()* на экземпляре типа **Class**, передав ему массив типов параметров в виде объектов **Class**.
- ❑ Вызвать метод *newInstance()* на экземпляре класса **Constructor**, передав ему список соответствующих аргументов;
  - Класс **Constructor** позволяет расширяющие преобразования в случае сопоставления аргументов в методе *newInstance()* с формальными параметрами базового конструктора, но бросает исключение типа **IllegalArgumentException** при сужающем преобразовании.




## Рефлексия

Пример 5:

```
try {  
    Class<?>[] paramsType =  
        new Class<?>[] {double.class, int.class };  
  
    Constructor<?> constr = clazz.getConstructor(paramsType);  
  
    MyTestReflect obj =  
        (MyTestReflect) constr.newInstance(-10.1, 5);  
  
    System.out.println(obj);  
} catch (Exception ex) {  
    ex.printStackTrace();  
}
```

Создание массива объектов **Class**,  
которые описывают типы параметров  
для конструктора



**Вывод в консоли:**

str = data, pr = -10.1, index = 5

## Рефлексия

### IV) Получение информации о методе

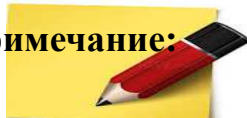
- ❑ Необходимо вызвать метод *getMethods()* на экземпляре **Class**:
  - возвращает массив объектов типа **Method**, отражающих все **public** методы;

*Например,* `Method[] methods = clazz.getMethods();`

- ❑ Необходимо вызвать метод *getDeclaredMethods()* на экземпляре **Class**:
  - возвращает массив объектов типа **Method**, отражающих все методы;

*Например,* `Method[] methods = clazz.getDeclaredMethods();`

**Примечание:**



Возвращаются методы, объявленные в классе/интерфейсе, а также унаследованные от суперкласса и интерфейсов;  
Если открытых методов нет или **Class** описывает примитивный тип, то возвращается массив длиной ноль.

## Рефлексия

### Пример 6:

```
Method[] methods = clazz.getDeclaredMethods();
for (Method method : methods) {
    System.out.println("Name: " + method.getName());
    System.out.println("\tReturn type: "
        + method.getReturnType().getName());
    Class<?>[] paramTypes = method.getParameterTypes();
    System.out.print("\tParam types:");
    for (Class<?> paramType : paramTypes) {
        System.out.print(paramType.getName() + " ");
    }
    System.out.println();
}
```

## Рефлексия

### Вывод в консоли:

Name: calcPr

Return type: void

Parameter types: int

Name: getPr

Return type: double

Parameter types:

Name: toString

Return type: java.lang.String

Parameter types:

Name: getIndex

Return type: int

Parameter types:

Name: setIndex

Return type: void

Parameter types: int

Name: mult

Return type: double

Parameter types: int

## Рефлексия

### V) Динамический вызов метода

- ❑ Класс **Method** содержит метод

*Object **invoke**(Object target, Object ... parameters)*

который используется для динамического вызова метода.

- он принимает первый параметр – ссылку на объект, для которого вызывается;
- второй параметр произвольной длины – перечень аргументов типа **Object** .



### ОСОБЕННОСТИ

- ✓ аргументы автоматически распаковываются, если формальный параметр примитивного типа;
- ✓ при необходимости могут вызываться методы преобразования;



### ОСОБЕННОСТИ

- ✓ если вызываемый метод является статическим, то ссылка на объект игнорируется;
- ✓ при вызове статического метода вместо ссылки на объект можно указывать **null**;
- ✓ если вызываемый метод не имеет параметров, то указывается только ссылка на объект или в качестве второго аргумента значение **null**;
- ✓ если вызываемый метод имеет возвращаемое значение и это значение примитивного типа, то оно упаковывается в объект соответствующего класса-обертки;
- ✓ если вызываемый метод не имеет возвращаемого значения, то из метода *invoke()* возвращается значение **null**.

## Рефлексия



### Ошибки использования метода *invoke()*


- ❑ **IllegalAccessException** - если вызываемый метод недоступен;
- ❑ **IllegalArgumentException**:
  - если вызываемый метод не является методом указанного объекта;
  - если количество аргументов и параметров различно;
  - если приведение для примитивных аргументов не удастся;
- ❑ **InvocationTargetException** - если вызываемый метод бросает исключение;
- ❑ **NullPointerException** - если вызываемый метод является методом экземпляра, а указанный объект является **null**.

## Рефлексия

### Пример 7:

```
MyTestReflect obj = new MyTestReflect();
try {
    Class<?>[] paramsType = new Class<?>[] { int.class };
    Method method = clazz.getMethod("calcPr", paramsType);
    method.invoke(obj, 1000);
    System.out.println(obj);
} catch (NoSuchMethodException | IllegalAccessException |
        InvocationTargetException ee) {
    ee.printStackTrace();
}
```

Создание массива  
объектов **Class**, которые  
описывают типы  
параметров метода



### Вывод в консоли:


```
str = data, pr = 4032.0, index = 2016001
```



## Рефлексия

Пример 8, ошибочный вызов:

```
MyTestReflect obj = new MyTestReflect();
try {
    Class<?>[] paramsType = new Class<?>[] { int.class };
    Method method = clazz.getMethod("mult", paramsType);
    method.invoke(obj, 1000);
    System.out.println(obj);
} catch (NoSuchMethodException | IllegalAccessException |
        InvocationTargetException ee) {
    ee.printStackTrace();
}
```



### Вывод в консоли:

```
java.lang.NoSuchMethodException:
com.reflect.MyTestReflect.mult(int)
```

## Рефлексия


### VI) Доступ к закрытым элементам класса

- Вызвать метод `setAccessible(true)` на объектах классов **Field**, **Constructor**, **Method**, которые отражают представление закрытых элементов класса.

Например,

```
MyTestReflect obj = new MyTestReflect();  
Field field = clazz.getDeclaredField("index");  
field.setAccessible(true);  
System.out.println("Private field value: " + field.getInt(obj));  
field.setInt(obj, 100);  
System.out.println("New private field value: " + field.getInt(obj));
```

В зависимости от типа поля вызывается соответствующий метод



#### Вывод в консоли:

```
Private field value: 2016001  
New private field value: 100
```