

# ИДЕНТИФИКАЦИЯ НА ТИПОВЕ ПО ВРЕМЕ НА ИЗПЪЛНЕНИЕ

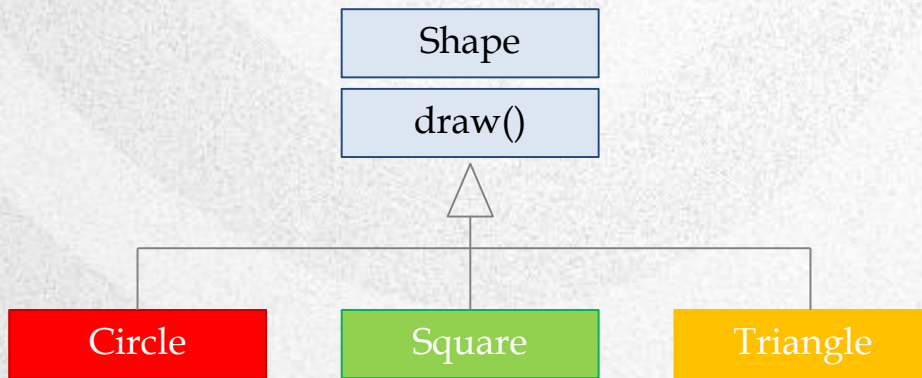
ЛЕКЦИОНЕН КУРС “ПРОГРАМИРАНЕ НА JAVA”



# СТРУКТУРА НА ЛЕКЦИЯТА

- Въведение
- Необходимост от RTTI
- Обект Class
- Отражение
- Примери

# ПОВТОРЕНИЕ



- Обикновено, цел на ООП: по-голямата част на кода да обработва референции към базовия тип
- Разширения на програмата правим като добавяме нови класове
- В примера: целта на програмиста на клиентската програма е да извика `draw()` чрез референция към родителския клас `Shape`
- `draw()` е предефиниран във всички производни класове и понеже е динамично свързан метод, той ще работи коректно и в случай, че се извика чрез референция към родителския клас (това е полиморфизъм)



# ВЪВЕДЕНИЕ

- Run-time Type Identification (RTTI)
- **Идея:** откриване на точния тип на даден обект, когато имаме само референция към базовия тип
- RTTI предполага множество интересни (и често смущаващи) проблеми на ОО развоя и поставя фундаментални въпроси за начина, по който трябва да структурираме програмите
- Две форми на RTTI
  - **Традиционна** – приема, че всички типове са достъпни по време на компилиране и по време на изпълнение
  - **Отражение** – позволява откриване информация за класовете единствено по време на изпълнение

# ТРАДИЦИОНЕН RTTI

- Възможности за реализиране на традиционна RTTI
  - Преобразуване надолу (downcast)
    - Извършва се явно
    - В примера преобразуването от Circle към Shape е преобразуване нагоре, а преобразуването от Shape към Circle е преобразуване **надолу**
  - Използване на обект от тип **Class**, представляващ типа на използвания обект
  - Ключовата дума **instanceof**
    - Указва дали един обект е екземпляр от определен тип
    - Връща като резултат boolean

# ПРИМЕР

```
import java.util.*;
class Shape {
    void draw() {
        System.out.println(this + ".draw()");
    }
}
class Circle extends Shape {
    public String toString() { return "Circle"; }
}
class Square extends Shape {
    public String toString() { return "Square"; }
}
class Triangle extends Shape {
    public String toString() { return "Triangle"; }
}
public class Shapes {
    public static void main(String[] args) {
        ArrayList s = new ArrayList();
        s.add(new Circle());
        s.add(new Square());
        s.add(new Triangle());
        Iterator e = s.iterator();
        while (e.hasNext())
            ((Shape)e.next()).draw();
    }
}
```



# ПРИМЕР

```
import java.util.*;
class Shape {
    void draw() {
        System.out.println(this + ".draw()");
    }
}
class Circle extends Shape {
    public String toString() { return "Circle"; }
}
class Square extends Shape {
    public String toString() { return "Square"; }
}
class Triangle extends Shape {
    public String toString() { return "Triangle"; }
}
public class Shapes {
    public static void main(String[] args) {
        ArrayList s = new ArrayList();
        s.add(new Circle());
        s.add(new Square());
        s.add(new Triangle());
        Iterator e = s.iterator();
        while (e.hasNext())
            ((Shape)e.next()).draw();
    }
}
```

← Всеки производен клас  
← предефинира метода toString(),  
← така че draw() отпечатва  
различен резултат за всеки  
отделен случай

# ПРИМЕР

```
import java.util.*;
class Shape {
    void draw() {
        System.out.println(this + ".draw()");
    }
}
class Circle extends Shape {
    public String toString() { return "Circle"; }
}
class Square extends Shape {
    public String toString() { return "Square"; }
}
class Triangle extends Shape {
    public String toString() { return "Triangle"; }
}
public class Shapes {
    public static void main(String[] args) {
        ArrayList s = new ArrayList();
        s.add(new Circle());
        s.add(new Square());
        s.add(new Triangle());
        Iterator e = s.iterator();
        while (e.hasNext())
            ((Shape)e.next()).draw();
    }
}
```

- В main() се създават специфични типове на Shape, след което се добавят към ArrayList
- Тук се прави преобразуване нагоре – ArrayList съдържа само Object обекти
- Понеже всичко е обект от клас Object, ArrayList може да съдържа също обекти от клас Shape
- При преобразуването нагоре се загубва цялата специфична информация, включително фактът, че обектите са Shape
- За ArrayList са просто обекти от клас Object
- Използваме итератор



# СЪЩЕСТВЕНИ КОНТЕЙНЕРИ ДАННИ

## **ArrayList:**

- Масив, който автоматично се саморазширява (използва индекс)
- Поддържа обекти от тип Object

## **Употреба:**

- Създаваме
- Попълваме елементи с add()
- Извличаме елементи чрез метода get()

## **Iterator:** абстракция на по-високо ниво

- Обект, чиято задача е да се движи през последователност от обекти и да избира всеки обект от тази последователност, без клиент-програмистът да знае или да се интересува от структурата ѝ
- „олекотен“ обект – не изисква много средства за да се създаде

## **Употреба:**

- next() – получаваме следващ елемент
- hasNext() – проверка за наличие на елементи

# ПРИМЕР

```
import java.util.*;
class Shape {
    void draw() {
        System.out.println(this + ".draw()");
    }
}
class Circle extends Shape {
    public String toString() { return "Circle"; }
}
class Square extends Shape {
    public String toString() { return "Square"; }
}
class Triangle extends Shape {
    public String toString() { return "Triangle"; }
}
public class Shapes {
    public static void main(String[] args) {
        ArrayList s = new ArrayList();
        s.add(new Circle());
        s.add(new Square());
        s.add(new Triangle());
        Iterator e = s.iterator();
        while (e.hasNext())
            → ((Shape)e.next()).draw();
    }
}
```

- Когато извличаме елементи от `ArrayList`, използвайки `next()`, нещата стават по-сложни
- Понеже `ArrayList` съдържа само `Object` обекти, `next()` генерира референции към `Object`
- Знаем, че това са `Shape` обекти и искаме да изпратим съобщения към такива обекти
- Така, че е необходимо преобразуване към `Shape`

# ПРИМЕР

```
import java.util.*;
class Shape {
    void draw() {
        System.out.println(this + ".draw()");
    }
}
class Circle extends Shape {
    public String toString() { return "Circle"; }
}
class Square extends Shape {
    public String toString() { return "Square"; }
}
class Triangle extends Shape {
    public String toString() { return "Triangle"; }
}
public class Shapes {
    public static void main(String[] args) {
        ArrayList s = new ArrayList();
        s.add(new Circle());
        s.add(new Square());
        s.add(new Triangle());
        Iterator e = s.iterator();
        while (e.hasNext())
            ((Shape)e.next()).draw();
    }
}
```

- Преобразуването към Shape “(Shape)” е основната форма на RTTI, понеже в Java правилността на всички преобразования се проверява по време на изпълнение на програмата
- Това е точното значение на RTTI – типът на обекта се идентифицира по време на изпълнение



# ПРИМЕР

```
import java.util.*;
class Shape {
    void draw() {
        System.out.println(this + ".draw()");
    }
}
class Circle extends Shape {
    public String toString() { return "Circle"; }
}
class Square extends Shape {
    public String toString() { return "Square"; }
}
class Triangle extends Shape {
    public String toString() { return "Triangle"; }
}
public class Shapes {
    public static void main(String[] args) {
        ArrayList s = new ArrayList();
        s.add(new Circle());
        s.add(new Square());
        s.add(new Triangle());
        Iterator e = s.iterator();
        while (e.hasNext())
            ((Shape)e.next()).draw();
    }
}
```

❓ Какво е RTTI преобразуване

- Частично – Object се преобразува до Shape
- Не по-нататък до Circle, Square или Triangle
- Това е така, че единственото, което знаем тук е, че ArrayList съдържа обекти от клас Shape обекти и искаме да изпратим съобщения към такива обекти
- Така, че е необходимо преобразуване към Shape

❓ Защо не преобразуваме след Shape

# ПРИМЕР

```
import java.util.*;
class Shape {
    void draw() {
        System.out.println(this + ".draw()");
    }
}
class Circle extends Shape {
    public String toString() { return "Circle"; }
}
class Square extends Shape {
    public String toString() { return "Square"; }
}
class Triangle extends Shape {
    public String toString() { return "Triangle"; }
}
public class Shapes {
    public static void main(String[] args) {
        ArrayList s = new ArrayList();
        s.add(new Circle());
        s.add(new Square());
        s.add(new Triangle());
        Iterator e = s.iterator();
        while (e.hasNext())
            ((Shape)e.next()).draw();
    }
}
```



Результат

```
Circle.draw()
Square.draw()
Triangle.draw()
```

# ОБЕКТ CLASS

- За да разберем начина на опериране на RTTI трябва да знаем как се представя информацията за типа по време на изпълнение на една програма
- Това се постига посредством използването на специален обект Class
  - Съдържа информация за нашия клас (мета-клас)
  - Един обект от тип Class се използва за създаване на всички обекти на нашите класове
  - За всеки клас от нашата програма съществува кореспондиращ обект Class
- Наследява Object
  - Голям брой методи - повече от 50



All Methods	Static Methods	Instance Methods	Concrete Methods
Modifier and Type	Method and Description		
<U> <b>Class</b> <? extends U>	<b>asSubclass</b> ( <b>Class</b> <U> clazz) Casts this <b>Class</b> object to represent a subclass of the class represented by the specified class object.		
<b>T</b>	<b>cast</b> ( <b>Object</b> obj) Casts an object to the class or interface represented by this <b>Class</b> object.		
<b>boolean</b>	<b>desiredAssertionStatus</b> () Returns the assertion status that would be assigned to this class if it were to be initialized at the time this method is invoked.		
<b>static Class</b> <?>	<b>forName</b> ( <b>String</b> className) Returns the <b>Class</b> object associated with the class or interface with the given string name.		
<b>static Class</b> <?>	<b>forName</b> ( <b>String</b> name, <b>boolean</b> initialize, <b>ClassLoader</b> loader) Returns the <b>Class</b> object associated with the class or interface with the given string name, using the given class loader.		
<b>AnnotatedType</b> []	<b>getAnnotatedInterfaces</b> () Returns an array of <b>AnnotatedType</b> objects that represent the use of types to specify superinterfaces of the entity represented by this <b>Class</b> object.		
<b>AnnotatedType</b>	<b>getAnnotatedSuperclass</b> () Returns an <b>AnnotatedType</b> object that represents the use of a type to specify the superclass of the entity represented by this <b>Class</b> object.		
<A extends <b>Annotation</b> > <b>A</b>	<b>getAnnotation</b> ( <b>Class</b> <A> annotationClass) Returns this element's annotation for the specified type if such an annotation is <i>present</i> , else null.		
<b>Annotation</b> []	<b>getAnnotations</b> () Returns annotations that are <i>present</i> on this element.		
<A extends <b>Annotation</b> > <b>A</b> []	<b>getAnnotationsByType</b> ( <b>Class</b> <A> annotationClass) Returns annotations that are <i>associated</i> with this element.		
<b>String</b>	<b>getCanonicalName</b> () Returns the canonical name of the underlying class as defined by the Java Language Specification.		

<b>Class&lt;?&gt;[]</b>	<b>getClasses()</b> Returns an array containing <b>Class</b> objects representing all the public classes and interfaces that are members of the class represented by this <b>Class</b> object.
<b>ClassLoader</b>	<b>getClassLoader()</b> Returns the class loader for the class.
<b>Class&lt;?&gt;</b>	<b>getComponentType()</b> Returns the <b>Class</b> representing the component type of an array.
<b>Constructor&lt;T&gt;</b>	<b>getConstructor(Class&lt;?&gt;... parameterTypes)</b> Returns a <b>Constructor</b> object that reflects the specified public constructor of the class represented by this <b>Class</b> object.
<b>Constructor&lt;?&gt;[]</b>	<b>getConstructors()</b> Returns an array containing <b>Constructor</b> objects reflecting all the public constructors of the class represented by this <b>Class</b> object.
<b>&lt;A extends Annotation&gt; A</b>	<b>getDeclaredAnnotation(Class&lt;A&gt; annotationClass)</b> Returns this element's annotation for the specified type if such an annotation is <i>directly present</i> , else null.
<b>Annotation[]</b>	<b>getDeclaredAnnotations()</b> Returns annotations that are <i>directly present</i> on this element.
<b>&lt;A extends Annotation&gt; A[]</b>	<b>getDeclaredAnnotationsByType(Class&lt;A&gt; annotationClass)</b> Returns this element's annotation(s) for the specified type if such annotations are either <i>directly present</i> or <i>indirectly present</i> .
<b>Class&lt;?&gt;[]</b>	<b>getDeclaredClasses()</b> Returns an array of <b>Class</b> objects reflecting all the classes and interfaces declared as members of the class represented by this <b>Class</b> object.
<b>Constructor&lt;T&gt;</b>	<b>getDeclaredConstructor(Class&lt;?&gt;... parameterTypes)</b> Returns a <b>Constructor</b> object that reflects the specified constructor of the class or interface represented by this <b>Class</b> object.
<b>Constructor&lt;?&gt;[]</b>	<b>getDeclaredConstructors()</b> Returns an array of <b>Constructor</b> objects reflecting all the constructors declared by the class represented by this <b>Class</b> object.
<b>Field</b>	<b>getDeclaredField(String name)</b> Returns a <b>Field</b> object that reflects the specified declared field of the class or interface represented by this <b>Class</b> object.



Field[]	<b>getDeclaredFields()</b> Returns an array of <b>Field</b> objects reflecting all the fields declared by the class or interface represented by this <b>Class</b> object.
Method	<b>getDeclaredMethod(String name, Class&lt;?&gt;... parameterTypes)</b> Returns a <b>Method</b> object that reflects the specified declared method of the class or interface represented by this <b>Class</b> object.
Method[]	<b>getDeclaredMethods()</b> Returns an array containing <b>Method</b> objects reflecting all the declared methods of the class or interface represented by this <b>Class</b> object, including public, protected, default (package) access, and private methods, but excluding inherited methods.
Class<?>	<b>getDeclaringClass()</b> If the class or interface represented by this <b>Class</b> object is a member of another class, returns the <b>Class</b> object representing the class in which it was declared.
Class<?>	<b>getEnclosingClass()</b> Returns the immediately enclosing class of the underlying class.
Constructor<?>	<b>getEnclosingConstructor()</b> If this <b>Class</b> object represents a local or anonymous class within a constructor, returns a <b>Constructor</b> object representing the immediately enclosing constructor of the underlying class.
Method	<b>getEnclosingMethod()</b> If this <b>Class</b> object represents a local or anonymous class within a method, returns a <b>Method</b> object representing the immediately enclosing method of the underlying class.
T[]	<b>getEnumConstants()</b> Returns the elements of this enum class or null if this <b>Class</b> object does not represent an enum type.
Field	<b>getField(String name)</b> Returns a <b>Field</b> object that reflects the specified public member field of the class or interface represented by this <b>Class</b> object.
Field[]	<b>getFields()</b> Returns an array containing <b>Field</b> objects reflecting all the accessible public fields of the class or interface represented by this <b>Class</b> object.
Type[]	<b>getGenericInterfaces()</b> Returns the <b>Types</b> representing the interfaces directly implemented by the class or interface represented by this object.



Type	<b>getGenericSuperclass()</b> Returns the <b>Type</b> representing the direct superclass of the entity (class, interface, primitive type or void) represented by this <b>Class</b> .
<b>Class&lt;?&gt;[]</b>	<b>getInterfaces()</b> Determines the interfaces implemented by the class or interface represented by this object.
Method	<b>getMethod(String name, Class&lt;?&gt;... parameterTypes)</b> Returns a <b>Method</b> object that reflects the specified public member method of the class or interface represented by this <b>Class</b> object.
Method[]	<b>getMethods()</b> Returns an array containing <b>Method</b> objects reflecting all the public methods of the class or interface represented by this <b>Class</b> object, including those declared by the class or interface and those inherited from superclasses and superinterfaces.
int	<b>getModifiers()</b> Returns the Java language modifiers for this class or interface, encoded in an integer.
String	<b>getName()</b> Returns the name of the entity (class, interface, array class, primitive type, or void) represented by this <b>Class</b> object, as a <b>String</b> .
Package	<b>getPackage()</b> Gets the package for this class.
ProtectionDomain	<b>getProtectionDomain()</b> Returns the <b>ProtectionDomain</b> of this class.
URL	<b>getResource(String name)</b> Finds a resource with a given name.
InputStream	<b>getResourceAsStream(String name)</b> Finds a resource with a given name.
Object[]	<b>getSigners()</b> Gets the signers of this class.
String	<b>getSimpleName()</b> Returns the simple name of the underlying class as given in the source code.

<code>Class&lt;? super T&gt;</code>	<b><code>getSuperclass()</code></b> Returns the <b>Class</b> representing the superclass of the entity (class, interface, primitive type or void) represented by this <b>Class</b> .
<code>String</code>	<b><code>getTypeName()</code></b> Return an informative string for the name of this type.
<code>TypeVariable&lt;Class&lt;T&gt;&gt;[]</code>	<b><code>getTypeParameters()</code></b> Returns an array of <b>TypeVariable</b> objects that represent the type variables declared by the generic declaration represented by this <b>GenericDeclaration</b> object, in declaration order.
<code>boolean</code>	<b><code>isAnnotation()</code></b> Returns true if this <b>Class</b> object represents an annotation type.
<code>boolean</code>	<b><code>isAnnotationPresent(Class&lt;? extends Annotation&gt; annotationClass)</code></b> Returns true if an annotation for the specified type is <i>present</i> on this element, else false.
<code>boolean</code>	<b><code>isAnonymousClass()</code></b> Returns <b>true</b> if and only if the underlying class is an anonymous class.
<code>boolean</code>	<b><code>isArray()</code></b> Determines if this <b>Class</b> object represents an array class.
<code>boolean</code>	<b><code>isAssignableFrom(Class&lt;?&gt; cls)</code></b> Determines if the class or interface represented by this <b>Class</b> object is either the same as, or is a superclass or superinterface of, the class or interface represented by the specified <b>Class</b> parameter.
<code>boolean</code>	<b><code>isEnum()</code></b> Returns true if and only if this class was declared as an enum in the source code.
<code>boolean</code>	<b><code>isInstance(Object obj)</code></b> Determines if the specified <b>Object</b> is assignment-compatible with the object represented by this <b>Class</b> .
<code>boolean</code>	<b><code>isInterface()</code></b> Determines if the specified <b>Class</b> object represents an interface type.
<code>boolean</code>	<b><code>isLocalClass()</code></b> Returns <b>true</b> if and only if the underlying class is a local class.

boolean

`isMemberClass()`

Returns **true** if and only if the underlying class is a member class.

boolean

`isPrimitive()`

Determines if the specified **Class** object represents a primitive type.

boolean

`isSynthetic()`

Returns **true** if this class is a synthetic class; returns **false** otherwise.

T

`newInstance()`

Creates a new instance of the class represented by this **Class** object.

String

`toGenericString()`

Returns a string describing this **Class**, including information about modifiers and type parameters.

String

`toString()`

Converts the object to a string.



# ПРИМЕР

```
import java.lang.*;

public class ClassDemo {

    public static void main(String[] args) {

        ClassDemo c = new ClassDemo();
        Class cls = c.getClass(); ←

        // returns the string representation of this class object
        String str = cls.toString(); ←
        System.out.println("Class = " + str);

        // returns the name of the class
        str = cls.getName(); ←
        System.out.println("Class = " + str);
    }
}
```

**getClass():** връща runtime класа на обекта (метод на Object)

**toString():** конвертира обекта към String (препокрит метод на Object)

**getName():** връща като String името на идентичност (class, interface, array class, primitive type, or void), представена от този клас

# ПРИМЕР

```
import java.lang.*;

public class ClassDemo {

    public static void main(String[] args) {

        ClassDemo c = new ClassDemo();
        Class cls = c.getClass();

        // returns the string representation of this class object
        String str = cls.toString();
        System.out.println("Class = " + str);

        // returns the name of the class
        str = cls.getName();
        System.out.println("Class = " + str);
    }
}
```



Результат

```
Class = class ClassDemo
Class = ClassDemo
```

# JVM

- Всеки път когато създаваме и компилираме нов клас, се създава съответен обект на Class
  - Съхранява се в идентично именуван .class файл
- По време на изпълнение, когато искаме да създадем обект от нашия клас JVM първо проверява дали обектът Class за този тип е зареден
  - Ако не е , JVM го намира и го зарежда
  - След като обектът Class за съответния тип (клас) е в паметта, той се използва за създаване на всички обекти от този тип
- Така, една Java програма не е напълно заредена преди да започне нейното изпълнение
  - Този подход се различава от много традиционни езици за програмиране

# ПРИМЕР

```
class Candy {
    static { System.out.println("Loading Candy"); }
}
class Gum {
    static { System.out.println("Loading Gum"); }
}
class Cookie {
    static { System.out.println("Loading Cookie"); }
}

public class SweetShop {
    public static void main(String[] args) {
        System.out.println("inside main");
        new Candy();
        System.out.println("After creating Candy");
        try {
            Class.forName("Gum");
        } catch (ClassNotFoundException e) {
            e.printStackTrace(System.err);
        }
        System.out.println("After Class.forName(\"Gum\")");
        new Cookie();
        System.out.println("After creating Cookie");
    }
}
```



# ПРИМЕР

```
class Candy {  
    static { System.out.println("Loading Candy"); }  
}  
class Gum {  
    static { System.out.println("Loading Gum"); }  
}  
class Cookie {  
    static { System.out.println("Loading Cookie"); }  
}  
  
public class SweetShop {  
    public static void main(String[] args) {  
        System.out.println("inside main");  
        new Candy();  
        System.out.println("After creating Candy");  
        try {  
            Class.forName("Gum");  
        } catch (ClassNotFoundException e) {  
            e.printStackTrace(System.err);  
        }  
        System.out.println("After Class.forName(\"Gum\")");  
        new Cookie();  
        System.out.println("After creating Cookie");  
    }  
}
```

- Всеки един от тези класове има клауза `static`, която се изпълнява когато класът е зареден за първи път
- Тя прави възможно отпечатване на информация когато класът се зарежда за първи път
- `static block`:
  - Използва се за инициализиране на статичните данни
  - Изпълнява се по време на зареждане на класа

# ПРИМЕР

```
class Candy {
    static { System.out.println("Loading Candy"); }
}
class Gum {
    static { System.out.println("Loading Gum"); }
}
class Cookie {
    static { System.out.println("Loading Cookie"); }
}

public class SweetShop {
    public static void main(String[] args) {
        System.out.println("inside main");
        new Candy();
        System.out.println("After creating Candy");
        try {
            Class.forName("Gum");
        } catch (ClassNotFoundException e) {
            e.printStackTrace(System.err);
        }
        System.out.println("After Class.forName(\"Gum\")");
        new Cookie();
        System.out.println("After creating Cookie");
    }
}
```

- Създаването на обекти се документира
- Цел: идентификация на момента на зареждане



# ПРИМЕР

```
class Candy {
    static { System.out.println("Loading Candy"); }
}
class Gum {
    static { System.out.println("Loading Gum"); }
}
class Cookie {
    static { System.out.println("Loading Cookie"); }
}

public class SweetShop {
    public static void main(String[] args) {
        System.out.println("inside main");
        new Candy();
        System.out.println("After creating Candy");
        try {
            Class.forName("Gum"); ←
        } catch (ClassNotFoundException e) {
            e.printStackTrace(System.err);
        }
        System.out.println("After Class.forName(\"Gum\")");
        new Cookie();
        System.out.println("After creating Cookie");
    }
}
```

- Методът е член на класа Class
- Обектът клас е като всички други – можем да получаваме и използваме референция към него (извършва се от зареждащата програма)
- Един от начините за получаване на референция към обекта Class е методът `forName`
  - Аргумент: низ, който е текстовото име на съответния клас
  - Резултат: връща референция към Class

# ПРИМЕР

```
class Candy {
    static { System.out.println("Loading Candy"); }
}
class Gum {
    static { System.out.println("Loading Gum"); }
}
class Cookie {
    static { System.out.println("Loading Cookie"); }
}

public class SweetShop {
    public static void main(String[] args) {
        System.out.println("inside main");
        new Candy();
        System.out.println("After creating Candy");
        try {
            Class.forName("Gum");
        } catch (ClassNotFoundException e) {
            e.printStackTrace(System.err);
        }
        System.out.println("After Class.forName(\"Gum\")");
        new Cookie();
        System.out.println("After creating Cookie");
    }
}
```



Резултат

inside main  
Loading Candy  
After creating Candy  
Loading Gum  
After Class.forName("Gum")  
Loading Cookie  
After creating Cookie

- Всеки Class обект се зарежда само когато е необходимо
- static инициализацията се извършва по време на зареждане на класа

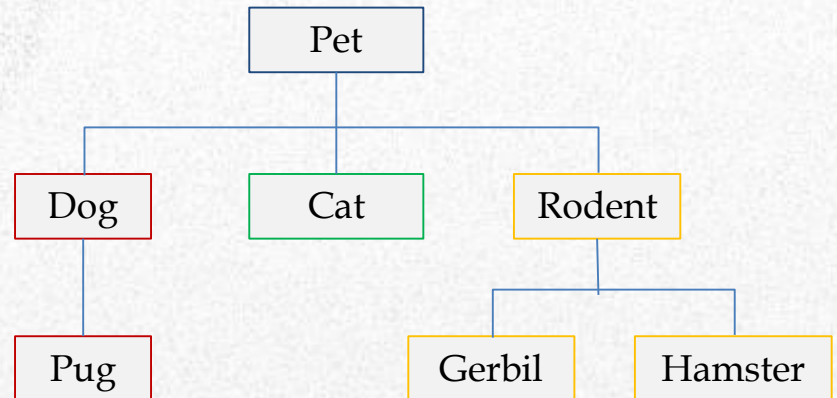


# ЛИТЕРАЛИ НА КЛАСОВЕ

- Java предоставя втори начин за генериране на референция към обекта Class
  - Използва литерал на клас
  - Напр., `Gum.class`
- По-проста и по-безопасна възможност
  - Проверява се по време на компилиране
- По-ефективна
  - Елиминира извикване на метод

# ПРИМЕР

```
class Pet { }  
class Dog extends Pet { }  
class Pug extends Dog { }  
class Cat extends Pet { }  
class Rodent extends Pet { }  
class Gerbil extends Rodent { }  
class Hamster extends Rodent { }  
  
class Counter { int i; }
```



# ПРИМЕР

```
import java.util.*;
public class PetCount {
    static String[] typenames = { "Pet", "Dog", "Pug", "Cat",
                                   "Rodent", "Gerbil", "Hamster", };
    public static void main(String[] args) throws Exception {
        ArrayList pets = new ArrayList();
        try {
            Class[] petTypes = {
                Class.forName("Dog"),
                Class.forName("Pug"),
                Class.forName("Cat"),
                Class.forName("Rodent"),
                Class.forName("Gerbil"),
                Class.forName("Hamster"), };
            for(int i = 0; i < 15; i++)
                pets.add(
                    petTypes[(int)(Math.random()*petTypes.length)].newInstance());
        } catch(InstantiationException e) {
            System.err.println("Cannot instantiate");
            throw e;
        } catch(IllegalAccessException e) {
            System.err.println("Cannot access");
            throw e;
        } catch(ClassNotFoundException e) {
            System.err.println("Cannot find class");
            throw e;
        }
    }
}
```

```
HashMap h = new HashMap();
for(int i = 0; i < typenames.length; i++)
    h.put(typenames[i], new Counter());
for(int i = 0; i < pets.size(); i++) {
    Object o = pets.get(i);
    if(o instanceof Pet)
        ((Counter)h.get("Pet")).i++;
    if(o instanceof Dog)
        ((Counter)h.get("Dog")).i++;
    if(o instanceof Pug)
        ((Counter)h.get("Pug")).i++;
    if(o instanceof Cat)
        ((Counter)h.get("Cat")).i++;
    if(o instanceof Rodent)
        ((Counter)h.get("Rodent")).i++;
    if(o instanceof Gerbil)
        ((Counter)h.get("Gerbil")).i++;
    if(o instanceof Hamster)
        ((Counter)h.get("Hamster")).i++;
}
for(int i = 0; i < pets.size(); i++)
    System.out.println(pets.get(i).getClass());
for(int i = 0; i < typenames.length; i++)
    System.out.println(typenames[i] + " quantity: " +
        ((Counter)h.get(typenames[i])).i);
}
```

# ПРИМЕР

```
import java.util.*;
public class PetCount {
    static String[] typenames = { "Pet", "Dog", "Pug", "Cat",
                                   "Rodent", "Gerbil", "Hamster", };
    public static void main(String[] args) throws Exception {
        ArrayList pets = new ArrayList();
        try {
            Class[] petTypes = {
                Class.forName("Dog"),
                Class.forName("Pug"),
                Class.forName("Cat"),
                Class.forName("Rodent"),
                Class.forName("Gerbil"),
                Class.forName("Hamster"), };
            for(int i = 0; i < 15; i++)
                pets.add(
                    petTypes[(int)(Math.random()*petTypes.length)].newInstance());
        } catch(InstantiationException e) {
            System.err.println("Cannot instantiate");
            throw e;
        } catch(IllegalAccessException e) {
            System.err.println("Cannot access");
            throw e;
        } catch(ClassNotFoundException e) {
            System.err.println("Cannot find class");
            throw e;
        }
    }
}
```

```
HashMap h = new HashMap();
```

Дефинираме имена на класове  
„домашни любимци“

```
Object o = pets.get(i);
```

Дефинираме масив съдържащ  
референции към обекти от тип  
Class за различните „домашни  
любимци“

```
((Counter)n.get("Pug")).i++;
if(o instanceof Pet)
    ((Counter)h.get("Cat")).i++;
if(o instanceof Pet)
    ((Counter)h.get("Rodent")).i++;
if(o instanceof Pet)
```

newInstance(): генерира нов обект  
от тип Class за дадения клас  
„домашни любимци“

```
for(int i = 0; i < pets.size(); i++)
    System.out.println(pets.get(i).getClass());
for(int i = 0; i < typenames.length; i++)
    System.out.println(typenames[i] + " quantity: " +
        ((Counter)h.get(typenames[i])).i);
}
```



# ПРИМЕР

?

## Защо такива структури

За броене на отделните „домашни любимци“ използваме оператора instanceof

- instanceof: оператор за проверка дали един обект е инстанция на специфицирания тип
- Можем да сравняваме само с именуван тип, но не и с Class обекти
- Създаваме масив от Class обекти, но не можем да го използваме директно за сравняване с instanceof

```
Class.forName("Hamster"), };
```

### Map:

- Тези контейнерни структури данни съхраняват двойки „(ключ, стойност)“

### HashMap:

- За съхраняване на данни се използват хеш-таблици
- Позволяват ефективно изпълнение на обичайните операции, като напр. get(Object), put(K,V), size()

```
HashMap h = new HashMap();
for(int i = 0; i < typenames.length; i++)
    h.put(typenames[i], new Counter());
for(int i = 0; i < pets.size(); i++) {
    Object o = pets.get(i);
    if(o instanceof Pet)
        ((Counter)h.get("Pet")).i++;
    if(o instanceof Dog)
        ((Counter)h.get("Dog")).i++;
    if(o instanceof Pug)
        ((Counter)h.get("Pug")).i++;
    if(o instanceof Cat)
        ((Counter)h.get("Cat")).i++;
    if(o instanceof Rodent)
        ((Counter)h.get("Rodent")).i++;
    if(o instanceof Gerbil)
        ((Counter)h.get("Gerbil")).i++;
    if(o instanceof Hamster)
        ((Counter)h.get("Hamster")).i++;
}
for(int i = 0; i < pets.size(); i++)
    System.out.println(pets.get(i).getClass());
for(int i = 0; i < typenames.length; i++)
    System.out.println(typenames[i] + " quantity: " +
        ((Counter)h.get(typenames[i])).i);
}
```

Броим  
отделните  
ТИПОВЕ

# ПРИМЕР

?

## Результат

```
import java.util.*;
public class PetCount {
    static String[] typenames = { "Pet", "Dog", "Pug", "Cat",
                                   "Rodent", "Gerbil", "Hamster" };

    public static void main(String[] args)
        throws Exception {
        ArrayList pets = new ArrayList();
        try {
            Class[] petTypes = {
                Class.forName("Dog"),
                Class.forName("Pug"),
                Class.forName("Cat"),
                Class.forName("Rodent"),
                Class.forName("Gerbil"),
                Class.forName("Hamster"), };
            for(int i = 0; i < 15; i++)
                pets.add(
                    petTypes[(int)(Math.random()*petTypes.length)]);
        } catch(InstantiationException e) {
            System.err.println("Cannot instantiate");
            throw e;
        } catch(IllegalAccessException e) {
            System.err.println("Cannot access");
            throw e;
        } catch(ClassNotFoundException e) {
            System.err.println("Cannot find class");
            throw e;
        }
    }
}
```

```
class Pug
class Gerbil
class Pug
class Cat
class Dog
class Cat
class Gerbil
class Hamster
class Hamster
class Hamster
class Gerbil
class Rodent
class Cat
class Gerbil
class Gerbil
Pet quantity: 15
Dog quantity: 3
Pug quantity: 2
Cat quantity: 3
Rodent quantity: 9
Gerbil quantity: 5
Hamster quantity: 3
```

```
HashMap h = new HashMap();
for(int i = 0; i < typenames.length; i++)
    h.put(typenames[i], new Counter());
for(int i = 0; i < pets.size(); i++) {
    Object o = pets.get(i);
    if(o instanceof Pet)
        ((Counter)h.get("Pet")).i++;
    if(o instanceof Dog)
        ((Counter)h.get("Dog")).i++;
    if(o instanceof Pug)
        ((Counter)h.get("Pug")).i++;
    if(o instanceof Cat)
        ((Counter)h.get("Cat")).i++;
    if(o instanceof Rodent)
        ((Counter)h.get("Rodent")).i++;
    if(o instanceof Gerbil)
        ((Counter)h.get("Gerbil")).i++;
    if(o instanceof Hamster)
        ((Counter)h.get("Hamster")).i++;
}
for(int i = 0; i < pets.size(); i++)
    System.out.println(pets.get(i).getClass());
for(int i = 0; i < typenames.length; i++)
    System.out.println(typenames[i] + " quantity: " +
        ((Counter)h.get(typenames[i])).i);
}
```

# ПРИМЕР

```
import java.util.*;
public class PetCount2 {
    public static void main(String[] args) throws Exception {
        ArrayList pets = new ArrayList();
        Class[] petTypes = {
            Pet.class,
            Dog.class,
            Pug.class,
            Cat.class,
            Rodent.class,
            Gerbil.class,
            Hamster.class, };

        try {
            for(int i = 0; i < 15; i++) {
                int rnd = 1 + (int) (Math.random() * (petTypes.length-1));
                pets.add(petTypes[rnd].newInstance());
            }
        } catch(InstantiationException e) {
            System.err.println("Cannot instantiate");
            throw e;
        } catch(IllegalAccessException e) {
            System.err.println("Cannot access");
            throw e;
        }
        HashMap h = new HashMap();
        for(int i = 0; i < petTypes.length; i++)
            h.put(petTypes[i].toString(), new Counter());
    }
}
```

- Премахнат е масивът `typenames`, поради получаване на низове с имена на типове от обекта `Class`
- Системата може да разграничи класовете от интерфейсите
- Използват се литерали на класове

```
for(int i = 0; i < pets.size(); i++) {
    Object o = pets.get(i);
    if(o instanceof Pet)
        ((Counter)h.get("class Pet")).i++;
    else if(o instanceof Dog)
        ((Counter)h.get("class Dog")).i++;
    else if(o instanceof Pug)
        ((Counter)h.get("class Pug")).i++;
    else if(o instanceof Cat)
        ((Counter)h.get("class Cat")).i++;
    else if(o instanceof Rodent)
        ((Counter)h.get("class Rodent")).i++;
    else if(o instanceof Gerbil)
        ((Counter)h.get("class Gerbil")).i++;
    else if(o instanceof Hamster)
        ((Counter)h.get("class Hamster")).i++;
    else
        ((Counter)h.get("class Unknown")).i++;
}
for(int i = 0; i < pets.size(); i++)
    System.out.println(pets.get(i).getClass());
Iterator keys = h.keySet().iterator();
while(keys.hasNext()) {
    String nm = (String)keys.next();
    Counter cnt = (Counter)h.get(nm);
    System.out.println(nm.substring(nm.lastIndexOf('.') + 1) + "
        quantity: " + cnt.i);
}
}
```

# ПРИМЕР

```
import java.util.*;
```

❓ **Защо petTypes не е в try блок, за разлика от първата версия**

```
ArrayList pets = new ArrayList();
```

- Не е необходимо, понеже се изчислява по време на компилиране
- Така, няма да доведе до никакви изключения по време на изпълнение на програмата, за разлика от Class.forName()

```
Cat.class,  
Rodent.class,  
Gerbil.class,  
Hamster.class, ;
```

```
try {  
    for(int i = 0; i < 15; i++) {  
        int rnd = 1 + (int) (Math.random() * (petTypes.length-1));  
        pets.add(petTypes[rnd].newInstance());  
    }  
} catch(InstantiationException e) {  
    System.err.println("Cannot instantiate");  
    throw e;  
} catch(IllegalAccessException e) {  
    System.err.println("Cannot access");  
    throw e;  
}  
HashMap h = new HashMap();  
for(int i = 0; i < petTypes.length; i++)  
    h.put(petTypes[i].toString(), new Counter());
```

```
for(int i = 0; i < pets.size(); i++) {
```

```
((Counter)h.get("class Pet")).i++;
```

```
if(o instanceof Pet)  
    ((Counter)h.get("class Cat")).i++;  
if(o instanceof Pet)  
    ((Counter)h.get("class Rodent")).i++;  
if(o instanceof Pet)  
    ((Counter)h.get("class Gerbil")).i++;  
if(o instanceof Pet)  
    ((Counter)h.get("class Hamster")).i++;
```

```
}  
for(int i = 0; i < pets.size(); i++)  
    System.out.println(pets.get(i).getClass());  
Iterator keys = h.keySet().iterator();  
while(keys.hasNext()) {  
    String nm = (String)keys.next();  
    Counter cnt = (Counter)h.get(nm);  
    System.out.println(nm.substring(nm.lastIndexOf('.') + 1) + "  
        quantity: " + cnt.i);  
}  
}
```



# ПРИМЕР

?

## Результат

```
import java.util.*;
public class PetCount2 {
    public static void main(String[] args) throws Exception {
        ArrayList pets = new ArrayList();
        Class[] petTypes = {
            Pet.class,
            Dog.class,
            Pug.class,
            Cat.class,
            Rodent.class,
            Gerbil.class,
            Hamster.class, };
        try {
            for(int i = 0; i < 15; i++) {
                int rnd = 1 + (int) (Math.random() * (petT
                pets.add(petTypes[rnd].newInstance());
            }
        } catch(InstantiationException e) {
            System.err.println("Cannot instantiate");
            throw e;
        } catch(IllegalAccessException e) {
            System.err.println("Cannot access");
            throw e;
        }
        HashMap h = new HashMap();
        for(int i = 0; i < petTypes.length; i++)
            h.put(petTypes[i].toString(), new Counter());
```

```
class Cat
class Gerbil
class Dog
class Gerbil
class Cat
class Cat
class Hamster
class Gerbil
class Rodent
class Hamster
class Pug
class Hamster
class Gerbil
class Pug
class Dog
class Gerbil quantity: 4
class Pet quantity: 15
class Dog quantity: 4
class Pug quantity: 2
class Hamster quantity: 3
class Rodent quantity: 8
class Cat quantity: 3
```

```
for(int i = 0; i < pets.size(); i++) {
    Object o = pets.get(i);
    if(o instanceof Pet)
        ((Counter)h.get("class Pet")).i++;
    if(o instanceof Dog)
        (Counter)h.get("class Dog")).i++;
    instanceof Pug)
        (Counter)h.get("class Pug")).i++;
    instanceof Cat)
        (Counter)h.get("class Cat")).i++;
    instanceof Rodent)
        (Counter)h.get("class Rodent")).i++;
    instanceof Gerbil)
        (Counter)h.get("class Gerbil")).i++;
    instanceof Hamster)
        (Counter)h.get("class Hamster")).i++;

    for(int i = 0; i < pets.size(); i++)
        System.out.println(pets.get(i).getClass());
    for keys = h.keySet().iterator();
        (keys.hasNext()) {
            nm = (String)keys.next();
            cnt = (Counter)h.get(nm);
            System.out.println(nm.substring(nm.lastIndexOf('.') + 1) + "
            quantity: " + cnt.i);
        }
    }
}
```

# ПРИМЕР

```
import java.util.*;

public class PetCount3 {
    public static void main(String[] args)
        throws Exception {
        ArrayList pets = new ArrayList();
        Class[] petTypes = {
            Pet.class,
            Dog.class,
            Pug.class,
```

```
HashMap h = new HashMap();
for(int i = 0; i < petTypes.length; i++)
    h.put(petTypes[i].toString(), new Counter());
for(int i = 0; i < pets.size(); i++) {
    Object o = pets.get(i);
    for(int j = 0; j < petTypes.length; ++j)
        if(petTypes[j].isInstance(o)) {
            String key = petTypes[j].toString();
            ((Counter)h.get(key)).i++;
        }
    }
```



- Методът `Class.isInstance` предоставя начин за динамично извикване на оператора `instanceof`
- Премахва необходимостта от `instanceof` изразите
- Можем да добавяме нови типове домашни любимци като просто променяме масива `petTypes`
- Не е необходимо да се променя останалата част на програмата (както е при използване на `instanceof` изразите)

```
try {
    for(int i = 0; i < petTypes.length; i++)
        pets.add(petTypes[i].newInstance());
} catch(InstantiationException e) {
    System.err.println("Cannot instantiate");
    throw e;
} catch(IllegalAccessException e) {
    System.err.println("Cannot access");
    throw e;
}
```

```
}
}
```

# ПРИМЕР

?

## Результат

```
import java.util.*;

public class PetCount3 {
    public static void main(String[] args)
        throws Exception {
        ArrayList pets = new ArrayList();
        Class[] petTypes = {
            Pet.class,
            Dog.class,
            Pug.class,
            Cat.class,
            Rodent.class,
            Gerbil.class,
            Hamster.class, };
        try {
            for(int i = 0; i < 15; i++) {
                int rnd = 1 + (int) (Math.random() *
                    - 1));
                pets.add(petTypes[rnd].newInstance());
            }
        } catch(InstantiationException e) {
            System.err.println("Cannot instantiate");
            throw e;
        } catch(IllegalAccessException e) {
            System.err.println("Cannot access");
            throw e;
        }
    }
}
```

```
class Dog
class Gerbil
class Pug
class Dog
class Dog
class Dog
class Hamster
class Pug
class Dog
class Dog
class Rodent
class Dog
class Cat
class Pug
class Hamster
class Gerbil quantity: 1
class Pet quantity: 15
class Dog quantity: 10
class Pug quantity: 3
class Hamster quantity: 2
class Rodent quantity: 4
class Cat quantity: 1
```

```
HashMap h = new HashMap();
for(int i = 0; i < petTypes.length; i++)
    h.put(petTypes[i].toString(), new Counter());
for(int i = 0; i < pets.size(); i++) {
    Object o = pets.get(i);
    for(int j = 0; j < petTypes.length; ++j)
        if(petTypes[j].isInstance(o)) {
            String key = petTypes[j].toString();
            ((Counter)h.get(key)).i++;
        }
    }
    for(int i = 0; i < pets.size(); i++)
        System.out.println(pets.get(i).getClass());
    Iterator keys = h.keySet().iterator();
    while(keys.hasNext()) {
        String nm = (String)keys.next();
        Counter cnt = (Counter)h.get(nm);
        System.out.println(nm.substring(nm.lastIndexOf('.') + 1)
            + " quantity: " + cnt.i);
    }
}
```

# ПРИМЕР

```
class Base { }
class Derived extends Base { }
public class FamilyVsExactType { ←
    static void test(Object x) {
        System.out.println("Testing x of type " +
            x.getClass());
        System.out.println("Testing x of Base " +
            (x instanceof Base));
        System.out.println("Testing x of Derived " +
            (x instanceof Derived));
        System.out.println("Base.isInstance(x) " +
            Base.class.isInstance(x));
        System.out.println("Derived.isInstance(x) " +
            Derived.class.isInstance(x));
        System.out.println("x.getClass() == Base.class " +
            (x.getClass() == Base.class));
        System.out.println("x.getClass() == Derived.class " +
            (x.getClass() == Derived.class));
        System.out.println("x.getClass().equals(Base.class) " +
            (x.getClass().equals(Base.class)));
        System.out.println("x.getClass().equals(Derived.class) " +
            (x.getClass().equals(Derived.class)));
    }
    public static void main(String[] args) {
        test(new Base());
        test(new Derived());
    }
}
```

Когато правим заявка за информация за типа, съществува съществена разлика между двете форми (`instanceof` и `isInstance()`), които дават едни и същи резултати) и директното сравняване на Class обектите



# ПРИМЕР

```
class Base { }
class Derived extends Base { }
public class FamilyVsExactType {
    static void test(Object x) {
        System.out.println("Testing x of type " +
            x.getClass());
        System.out.println("Testing x of Base " +
            (x instanceof Base));
        System.out.println("Testing x of Derived " + ←
            (x instanceof Derived));
        System.out.println("Base.isInstance(x) " +
            Base.class.isInstance(x));
        System.out.println("Derived.isInstance(x) " +
            Derived.class.isInstance(x));
        System.out.println("x.getClass() == Base.class " +
            (x.getClass() == Base.class));
        System.out.println("x.getClass() == Derived.class " +
            (x.getClass() == Derived.class)); ←
        System.out.println("x.getClass().equals(Base.class) " +
            (x.getClass().equals(Base.class)));
        System.out.println("x.getClass().equals(Derived.class) " +
            (x.getClass().equals(Derived.class)));
    }
    public static void main(String[] args) {
        test(new Base());
        test(new Derived());
    }
}
```

- Използването на `instanceof` и `isInstance()` дават еднакъв резултат, както и `equals` и `"=="`
- Но самите тестове водят до различни заключения
  - `instanceof`: „Ти този клас ли си или си негов производен клас?“
  - `==`: „Това ли е точният тип или не е?“ (не се интересуваме за наследяването)

# ПРИМЕР

?

## Результат

```
class Base { }
class Derived extends Base { }
public class FamilyVsExactType {
    static void test(Object x) {
        System.out.println("Testing x of type " +
            x.getClass());
        System.out.println("Testing x of Base " +
            (x instanceof Base));
        System.out.println("Testing x of Derived " +
            (x instanceof Derived));
        System.out.println("Base.isInstance(x) " +
            Base.class.isInstance(x));
        System.out.println("Derived.isInstance(x) " +
            Derived.class.isInstance(x));
        System.out.println("x.getClass() == Base.class " +
            (x.getClass() == Base.class));
        System.out.println("x.getClass() == Derived.class " +
            (x.getClass() == Derived.class));
        System.out.println("x.getClass().equals(Base.class) " +
            (x.getClass().equals(Base.class)));
        System.out.println("x.getClass().equals(Derived.class) " +
            (x.getClass().equals(Derived.class)));
    }
    public static void main(String[] args) {
        test(new Base());
        test(new Derived());
    }
}
```

```
Testing x of type class Base
Testing x of Base true
Testing x of Derived false
Base.isInstance(x) true
Derived.isInstance(x) false
x.getClass() == Base.class true
x.getClass() == Derived.class false
x.getClass().equals(Base.class) true
x.getClass().equals(Derived.class) false
Testing x of type class Derived
Testing x of Base true
Testing x of Derived true
Base.isInstance(x) true
Derived.isInstance(x) true
x.getClass() == Base.class false
x.getClass() == Derived.class true
x.getClass().equals(Base.class) false
x.getClass().equals(Derived.class) true
```

# ВЪЗМОЖНОСТИ НА CLASS

- Java изпълнява RTTI, използвайки обекта Class
- Този клас доставя различни възможности за използване на RTTI – първо трябва да получим референция към подходящия Class обект
  - Метод `Class.forName()` – не е необходимо наличието на обект за да получим референция към Class
  - Метод `getClass` – ако имаме обект от интересуващия ни тип, можем да извлечем референция към Class (част от `Object`)

# ПРИМЕР

```
interface HasBatteries { }
interface Waterproof { }
interface ShootsThings { }
class Toy {
    Toy() { }
    Toy(int i) { }
}
class FancyToy extends Toy ←
implements HasBatteries, Waterproof, ShootsThings {
    FancyToy() { super(1); }
}
public class ToyTest {
    public static void main(String[] args)
        throws Exception {
        Class c = null;
        try {
            c = Class.forName("FancyToy"); ←
        } catch (ClassNotFoundException e) {
            System.err.println("Can't find FancyToy");
            throw e;
        }
        printInfo(c);
    }
}
```

```
Class[] faces = c.getInterfaces();
for(int i = 0; i < faces.length; i++)
    printInfo(faces[i]);
Class cy = c.getSuperclass();
Object o = null;
try {
    o = cy.newInstance();
} catch (InstantiationException e) {
    System.err.println("Cannot instantiate");
    throw e;
}
```

- Сравнително сложен клас
- Референция към Class се инициализира към класа FancyToy като се използва forName

```
static void printInfo(Class cc) {
    System.out.println(
        "Class name: " + cc.getName() +
        " is interface? [" +
        cc.isInterface() + "]" );
}
}
```



# ПРИМЕР

```
interface HasBatteries { }
```

Методът `Class.getInterface()` връща масив от `Class` обекти, представлящи интерфейсите, които се съдържат в интересувания ни `Class` обект

```
class FancyToy extends Toy
```

- Питаме за директен базов клас като използваме `getSuperclass()`
- Така, по време на изпълнение можем да открием цялата класова йерархия на даден обект

```
try {  
    c = Class.forName("FancyToy");  
} catch (ClassNotFoundException e) {  
    System.err.println("Can't find FancyToy");  
    throw e;  
}  
printInfo(c);
```

```
Class[] faces = c.getInterfaces();  
for(int i = 0; i < faces.length; i++)  
    printInfo(faces[i]);  
Class cy = c.getSuperclass();  
Object o = null;  
try {  
    o = cy.newInstance();  
} catch (InstantiationException e) {  
    System.err.println("Cannot instantiate");  
    throw e;  
} catch (IllegalAccessException e) {  
    System.err.println("Cannot access");  
    throw e;  
}  
printInfo(o.getClass());  
}  
static void printInfo(Class cc) {  
    System.out.println(  
        "Class name: " + cc.getName() +  
        " is interface? [" +  
        cc.isInterface() + "]);  
}  
}
```

# ПРИМЕР

```
interface HasBatteries { }
interface Waterproof { }
interface ShootsThings { }
class Toy {
    Toy() { }
    Toy(int i) { }
}
```

```
class FancyToy extends Toy
```

```
import java.lang.reflect.*;
public class Main {
    public static void main(String[] args) {
        // ...
        Class c = Toy.class;
        // ...
        printInfo(c);
    }
}
```

- Можем да създаваме нов обект с `newInstance()` без да съществува обект, а само референция към `Class` обекта
- Един от начините за реализиране на „виртуален конструктор“
- Класът, от който се създава обектът трябва да има конструктор по подразбиране

```
Class[] faces = c.getInterfaces();
for(int i = 0; i < faces.length; i++)
    printInfo(faces[i]);
Class cy = c.getSuperclass();
Object o = null;
try {
    o = cy.newInstance();
} catch(InstantiationException e) {
    System.err.println("Cannot instantiate");
    throw e;
} catch(IllegalAccessException e) {
    System.err.println("Cannot access");
    throw e;
}
printInfo(o.getClass());
}
static void printInfo(Class cc) {
    System.out.println(
        "Class name: " + cc.getName() +
        " is interface? [" +
        cc.isInterface() + "]");
}
}
```

# ПРИМЕР

?

## Результат

```
interface HasBatteries { }
interface Waterproof { }
interface ShootsThings { }
class Toy {
    Toy() { }
    Toy(int i) { }
}
class FancyToy extends Toy
implements HasBatteries, Waterproof {
    FancyToy() { super(1); }
}
public class ToyTest {
    public static void main(String[] args)
        throws Exception {
        Class c = null;
        try {
            c = Class.forName("FancyToy");
        } catch (ClassNotFoundException e) {
            System.err.println("Can't find FancyToy");
            throw e;
        }
        printInfo(c);
    }
}
```

```
Class name: FancyToy is interface? [false]
Class name: HasBatteries is interface? [true]
Class name: Waterproof is interface? [true]
Class name: ShootsThings is interface? [true]
Class name: Toy is interface? [false]
```

```
Class[] faces = c.getInterfaces();
for(int i = 0; i < faces.length; i++)
    printInfo(faces[i]);
Class cy = c.getSuperclass();
Object o = null;
try {
    o = cy.newInstance();
} catch (InstantiationException e) {
    println("Cannot instantiate");
} catch (IllegalAccessException e) {
    println("Cannot access");
}
println(o.getClass());
}
static void printInfo(Class cc) {
    System.out.println(
        "Class name: " + cc.getName() +
        " is interface? [" +
        cc.isInterface() + "]);
}
}
```

# ПРИМЕР

## Результат

```
import java.lang.*;
class superClass {
    // super class
}
class subClass extends superClass {
    // sub class
}
public class ClassDemo1 {
    public static void main(String args[]) {
        superClass val1 = new superClass();
        subClass val2 = new subClass();
        Class cls;
        cls = val1.getClass();
        System.out.println("val1 is object of type = " + cls.getName());

        cls = cls.getSuperclass();
        System.out.println("super class of val1 = " + cls.getName());

        cls = val2.getClass();
        System.out.println("val2 is object of type = " + cls.getName());

        cls = cls.getSuperclass();
        System.out.println("super class of val2 = " + cls.getName());
    }
}
```

val1 is object of type = superClass  
super class of val1 = java.lang.Object  
val2 is object of type = subClass  
super class of val2 = superClass



# ОБОБЩЕНИЕ НА ТРАДИЦИОННА RTTI

- Ако не знаем точния обект RTTI ще ни го покаже
  - Съществува едно ограничение
    - Типът трябва да е известен по време на компилиране
    - Т.е., компилаторът трябва да знае за всички класове, с които работим
- Съществуват случаи, където компилаторът не знае за класа, докато компилира кода
  - Как е възможно да се използва един такъв клас?
- Примери:
  - Подадена референция за обект, който не е в нашето програмно пространство
  - RMI (Remote Method Invocation) – създаване и изпълнение на обекти на отдалечени платформи

# ОТРАЖЕНИЕ

- Класът Class поддържа концепцията за отражение
  - Също допълнителна библиотека `java.lang.reflect`
- Обектите от тези типове се създават от JVM по време на изпълнение, за да представят съответния член в неизвестния клас
  - Информацията за класовете за анонимните обекти може да бъде напълно определена по време на изпълнение и не е необходимо нищо да бъде известно по време на компилиране

# ИЗПОЛЗВАНЕ НА ОТРАЖЕНИЕТО

- Когато използване отражение за взаимодействие с обект от неизвестен тип, JVM ще търси обекта и ще се опита да определи класа му
  - Class обектът трябва да бъде зареден
  - Т.е., .class обектът за този тип трябва да бъде достъпен за JVM (локалната машина или по мрежата)
- Разликата между традиционната RTTI и отражението
  - При традиционната компилаторът отваря и проверява .class файловете по време на компилиране
  - При отражението .class файловете не са достъпни по време на компилация – те се отварят и проверяват от средата за изпълнение
- Рядко ще ни се налага да използваме средствата за отражение
- Те са в езика предимно за поддръжка на други възможности на Java
  - Сериализация на обекти
  - JavaBeans
  - RMI




# ПРИМЕР

```
import java.lang.reflect.*;

public class ShowMethods {
    static final String usage =
        "usage: \n" +
        "ShowMethods qualified.class.name\n" +
        "To show all methods in class or: \n" +
        "ShowMethods qualified.class.name word\n" +
        "To search for methods involving 'word'";
    public static void main(String[] args) {
        if(args.length < 1) {
            System.out.println(usage);
            System.exit(0);
        }
    }
}
```

- Методите `getMethods()` и `getConstructors()` връщат масив съответно от `Method` и `Constructor`
- Могат да се анализират също имена, аргументи, връщани стойности, ...
- Можем да използваме `toString` за да получим като `String` сигнатурите на методите



```
try {
    Class c = Class.forName(args[0]);
    Method[] m = c.getMethods();
    Constructor[] ctor = c.getConstructors();
    if(args.length == 1) {
        for(int i = 0; i < m.length; i++)
            System.out.println(m[i]);
        for(int i = 0; i < ctor.length; i++)
            System.out.println(ctor[i]);
    } else {
        for(int i = 0; i < m.length; i++)
            if(m[i].toString()
                .indexOf(args[1]) != -1)
                System.out.println(m[i]);
        for(int i = 0; i < ctor.length; i++)
            if(ctor[i].toString()
                .indexOf(args[1]) != -1)
                System.out.println(ctor[i]);
    }
} catch(ClassNotFoundException e) {
    System.err.println("No such class: " + e);
}
}
```



# ПРИМЕР

```
import java.lang.reflect.*;

public class ShowMethods {
    static final String usage =
        "usage: \n" +
        "ShowMethods qualified.class.name\n" +
        "To show all methods in class or: \n" +
        "ShowMethods qualified.class.name word\n" +
        "To search for methods involving 'word'";
    public static void main(String[] args) {
        if(args.length < 1) {
            System.out.println(usage);
            System.exit(0);
        }
    }
}
```

- Показва отражението в действие
- Полученият от `Class.forName()` метода резултат не може да бъде известен по време на компилиране
- Следователно цялата информация за сигнатурата на метода е била извлечена по време на изпълнение

```
try {
    Class c = Class.forName(args[0]);
    Method[] m = c.getMethods();
    Constructor[] ctor = c.getConstructors();
    if(args.length == 1) {
        for(int i = 0; i < m.length; i++)
            System.out.println(m[i]);
        for(int i = 0; i < ctor.length; i++)
            System.out.println(ctor[i]);
    } else {
        for(int i = 0; i < m.length; i++)
            if(m[i].toString()
                .indexOf(args[1]) != -1)
                System.out.println(m[i]);
        for(int i = 0; i < ctor.length; i++)
            if(ctor[i].toString()
                .indexOf(args[1]) != -1)
                System.out.println(ctor[i]);
    }
} catch(ClassNotFoundException e) {
    System.err.println("No such class: " + e);
}
}
```

# ПРИМЕР

## java ShowMethods

```
import java.lang.reflect.*;

public class ShowMethods {
    static final String usage =
        "usage: \n" +
        "ShowMethods qualified.class.name\n" +
        "To show all methods in class or: \n" +
        "ShowMethods qualified.class.name word\n" +
        "To search for methods involving 'word'";
    public static void main(String[] args) {
        if(args.length < 1) {
            System.out.println(usage);
            System.exit(0);
        }
    }
}
```

usage:  
ShowMethods qualified.class.name  
To show all methods in class or:  
ShowMethods qualified.class.name word  
To search for methods involving 'word'

```
try {
    Class c = Class.forName(args[0]);
    Method[] m = c.getMethods();
    Constructor[] ctor = c.getConstructors();
    if(args.length == 1) {
        for(int i = 0; i < m.length; i++)
            System.out.println(m[i]);
        for(int i = 0; i < ctor.length; i++)
            System.out.println(ctor[i]);
    } else {
        for(int i = 0; i < m.length; i++)
            if(m[i].toString()
                .indexOf(args[1]) != -1)
                System.out.println(m[i]);
        for(int i = 0; i < ctor.length; i++)
            if(ctor[i].toString()
                .indexOf(args[1]) != -1)
                System.out.println(ctor[i]);
    }
} catch(ClassNotFoundException e) {
    System.err.println("No such class: " + e);
}
}
```



Резултат



Коментар

# ПРИМЕР

## java ShowMethods ShowMethods

```
import java.lang.reflect.*;
```

```
public class ShowMethods {
```

```
    static final String usage =
```

```
        "usage: \n" +
```

```
        "ShowMethods qualified.class.name\n" +
```

```
        "To show all methods in class or: \n" +
```

```
        "ShowMethods qualified.class.name word\n" +
```

```
        "To search for methods involving 'word'";
```

```
    public static void main(String[] args) {
```

```
        if (args.length < 1)
```

```
            System.out.println(usage);
```

```
            System.exit(0);
```

```
        }
```

```
    try {
```

```
        2 Class c = Class.forName(args[0]);
```

```
        Method[] m = c.getMethods();
```

```
        1 Constructor[] ctor = c.getConstructors();
```

```
        if (args.length == 1) {
```

```
            for (int i = 0; i < m.length; i++)
```

```
                System.out.println(m[i]);
```

```
            for (int i = 0; i < ctor.length; i++)
```

```
                System.out.println(ctor[i]);
```

```
    public static void ShowMethods.main(java.lang.String[])
```

```
    public final void java.lang.Object.wait() throws java.lang.InterruptedExcep
```

```
    public final void java.lang.Object.wait(long,int) throws java.lang.InterruptedExcep
```

```
    public final native void java.lang.Object.wait(long) throws java.lang.InterruptedExcep
```

```
    public boolean java.lang.Object.equals(java.lang.Object)
```

```
    public java.lang.String java.lang.Object.toString()
```

```
    public native int java.lang.Object.hashCode()
```

```
    public final native java.lang.Class java.lang.Object.getClass()
```

```
    public final native void java.lang.Object.notify()
```

```
    public final native void java.lang.Object.notifyAll()
```

```
    public ShowMethods()
```

```
        System.out.println("No such class: " + c);
```

```
    }
```

```
    }
```

```
    }
```

?

Резултат

?

Коментар

БЛАГОДАРЯ ЗА ВНИМАНИЕТО!

КРАЙ “RTTI”

