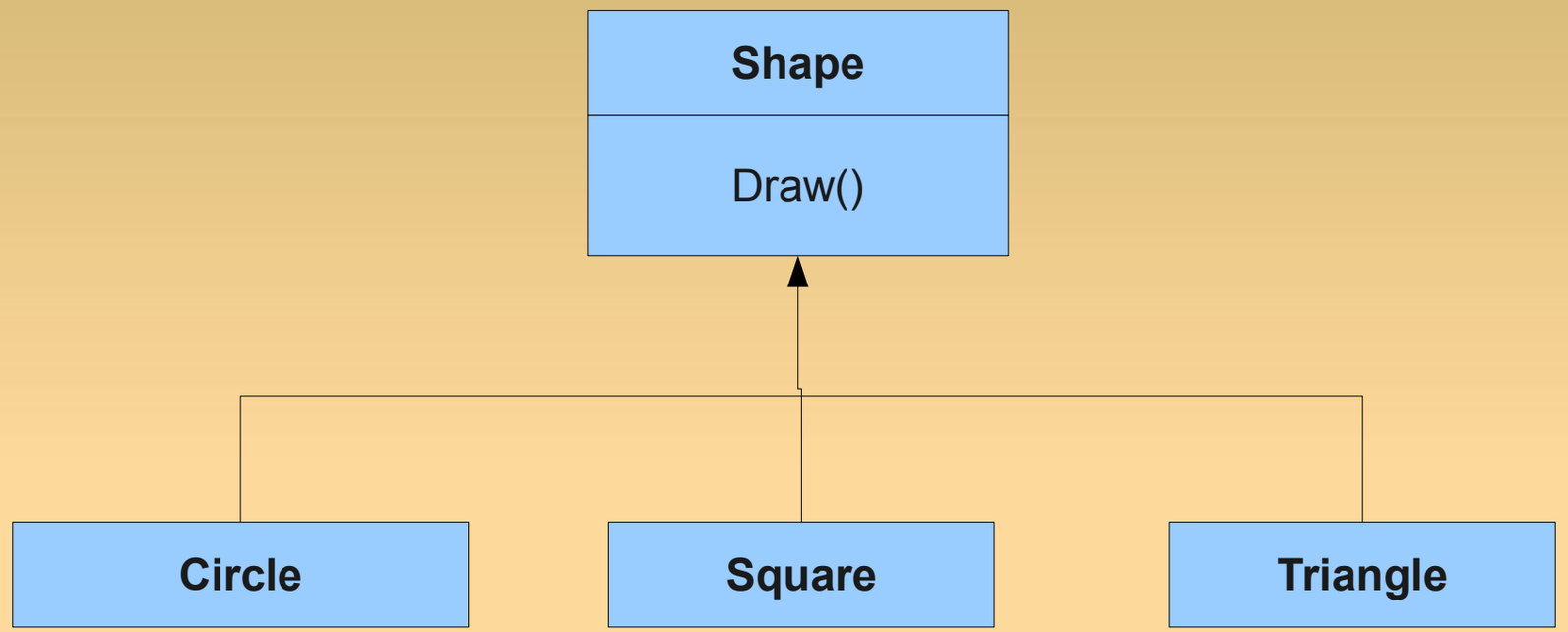
RTTI.

Практика 23 - Айрат Хасьянов

Зачем нужен RTTI?




```
Shape shapesArr[] = {new Triangle(), new Circle(),
new Shape()};
```

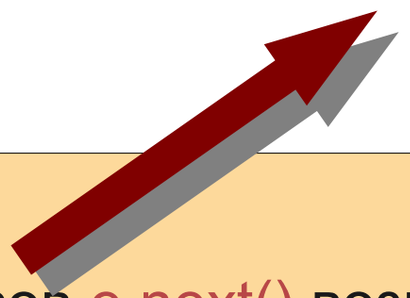
Как в таком массиве различать экземпляры разных классов?

Рассорим иерархию классов

```
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"; }
}
```

Приведение типов

```
public class Shapes {
  public static void main(String[] args) {
    ArrayList<Shape> s = new ArrayList<Shape>();
    s.add(new Circle());
    s.add(new Square());
    s.add(new Triangle());
    Iterator<Shape> e = s.iterator();
    while(e.hasNext())
      ((Shape)e.next()).draw();
  }
}
```



Вызов e.next() возвращает Object. При помощи RTTI Java проверяет, допустимо ли приведение типов!

Это Класс!

RTTI работает, потому что информация о типе любого объекта хранится в его свойстве типа Class!

- Объект Class существует для каждого класса, который является частью программы
- Во время компиляции класса создается объект типа Class и хранится в виде одноименного файла с расширением .class
- При создании объекта JVM загружает соответствующий файл .class.
- Программа загружается частями, по мере необходимости!

Демонстрация загрузки классов

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

public class SweetShop {
public static void main(String[] args) {
 System.out.println("inside main");
 new Candy();
 System.out.println("After creating Candy");
}
```

```
inside main
Loading Candy
After creating Candy
```

Зная имя класса можно получить его объект типа Class

Зная имя класса, можно получить экземпляр объекта типа Class:

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

```
try {
    Class.forName("ru.ksu.javatraining.rtti.Gum");
    System.out.println(gum.getCanonicalName());
 } catch(ClassNotFoundException e) {
    System.out.println("Couldn't find Gum");
 }
 System.out.println("Done Class.forName(\"Gum\")");
}
```

```
Loading Gum
ru.ksu.javatraining.rtti.Gum
```

Как можно получить объект типа Class?

1. Использовать метод getClass(), определенный для объектов всех классов, унаследованных от Object. Пригоден, если объект нужного класса существует
2. Вызвать Class.forName(). Не работает для примитивов!
3. Использовать литерал class, определенный для всех классов, включая примитивные типы. При обращении, например, к Gum.class загрузка Gum не происходит!
4. Объект типа Class для примитивных типов можно получить через поле TYPE «класса обертки», например: Boolean.TYPE возвращает ссылку на boolean.class.

Получим объекты типа Class!

```
Class cl1 = int.class;
Class cl2 = Integer.TYPE;
Class cl3;
try {
   cl3 = Class.forName("java.lang.Integer");
   Integer x = new Integer(7);
   Class cl4 = x.getClass();
   System.out.println(cl1.toString()+'\n'+cl2+'\n'+cl3
+'\n'+cl4);
} catch (ClassNotFoundException e) {
   System.out.println("Class Integer was not found :(");
}
```

```
int
int
class java.lang.Integer
class java.lang.Integer
```

Используем ***параметризацию,*** чтобы ограничить возможный тип объекта Class

```
Class intClass = int.class;
Class<Integer> genericIntClass = int.class;
genericIntClass = Integer.class; // все верно
intClass = double.class; // допустимо!
// genericIntClass = double.class; // недопустимо!
```

Преобразование типов

В Java 5 появился новый синтаксис преобразования типов. Теперь для этой цели можно использовать объекты типа Class!

```
Building b = new House();
Class<House> houseType = House.class;
House h = houseType.cast(b);
h = (House)b; // ... а можно и так.
```

Такой подход полезен при написании параметризованного кода.

Рассмотрим три класса фигур

Опишем три класса:

```
static class Shape{
void draw(){ System.out.print("Shape ");}
}

static class Circle extends Shape{
void draw() {System.out.print("Circle ");}
int getRadius(){ return 1862;}
}

static class Triangle extends Shape{
void draw(){System.out.print("Triangle ");}
}
```

Вот так надо обходиться с приведением типов

```
Shape shapesArr[] = {new Triangle(), new Circle(),
new Shape()};
for(Shape shape: shapesArr){
   shape.draw();
   if((shape instanceof Circle)){
      System.out.print("of radius "+
((Circle)shape).getRadius());
   }
   System.out.println();
}
```

```
Triangle
Circle of radius 1862
Shape
```

Еще один вариант проверки типов

Для определения типа объекта вместо оператора instanceof можно использовать метод isInstance(). Например:

```
Shape shapesArr[] = {new Triangle(), new Circle(),
new Shape()};
for(Shape shape: shapesArr){
   shape.draw();
   if((Circle.class.isInstance(shape))){
      System.out.print("of radius "+
((Circle)shape).getRadius());
   }
   System.out.println();
}
```

Class.isInstance удобен при динамической проверке типа.

Class vs. instanceof

```
class Base {} class Derived extends Base {}

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

instanceof учитывает наследование

```
Testing x of type class typeinfo.Base

Base.isInstance(x) true
Derived.isInstance(x) false
x.getClass() == Base.class true
x.getClass() == Derived.class false
```

```
Testing x of type class typeinfo.Derived

Base.isInstance(x) true
Derived.isInstance(x) true
x.getClass() == Base.class false
x.getClass() == Derived.class true
```

```
class Base {} class Derived extends Base {}

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

Некторые методы Class

- getInterfaces() – возвращает массив объектов типа Class, реализованных данным классом
- getSuperclass() – возвращает класс предка
- newInstance() – создание экземпляра класса
- getName() – имя класса
- isInterface() – является ли данный класс интерфейсом
- cast() - преобразование типа
- forName() - статический метод, позволяет получить объект типа Class по имени класса

Еще раз старый пример: паттерн «Генератор»

Интерфейсы также поддерживают параметризацию!

```
public interface Generator <T> {
   T next();
}
```

```
public class GenericGenerator <T>
   implements Generator<T> {
private Class<T> type;
public GenericGenerator(Class<T>
type){this.type = type;}
@Override
public T next() {...}
```

```
abstract class Alien {}
public class AlienGenerator implements Generator<Alien>{
private int size;
@Override
public Alien next() { ...}}
```

Самостоятельная

- Реализуйте паттерн параметризованный генератор для фигур описанных ранее.
- Используйте вашу коллекцию стек, чтобы заполнить его созданными генератором или генераторам (?) фигурами.
- Напечатайте типы попавших в стек фигур двумя способами: используя RTTI и метод draw().

Домашняя работа

- Подготовить отчеты команд в виде презентации из 4-5 слайдов о ходе месячного проекта в формате

    1. что сделано,
    2. что из запланированного не сделано в срок и почему,
    3. что запланировано далее

- Придумать по 2 задачи на паттерн «Генератор»
- Реализовать коллекцию «Упорядоченное множество» OrderedSet implements ISet так, чтобы добавление элемента и проверка элемента на вхождение в множество выполнялись как можно быстрее. Какова сложность операций с такой коллекцией?