## EPAM Systems, RD Dep.

## Конспект и раздаточный материал

# JAVA.SE.02 Object-oriented programming in Java

| REVISION HISTORY | | | | | |
|---|---|---|---|---|---|
| **Ver.** | **Description of Change** | **Author** | **Date** | **Approved** | |
| | | | | **Name** | **Effective Date** |
| <1.0> | Первая версия | Игорь Блинов | <14.09.2011> | | |
| <2.0> | Вторая версия. Конспект переделан под обновленное содержание материала модуля. | Ольга Смолякова | <27.03.2014> | | |

# Содержание
# JAVA.SE.02 OBJECT-ORIENTED PROGRAMMING IN JAVA

1. **Принципы ООП**
2. **Простейшие классы и объекты**
3. **Классы и объекты**
4. **Наследование**
5. **Интерфейсы**
6. **Параметризованные классы**
7. **Перечисления**
8. **Классы внутри классов**
9. **Аннотации (основы)**
10. **Введение в Design Patterns**

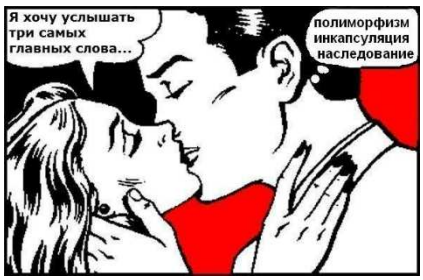
# Принципы ООП

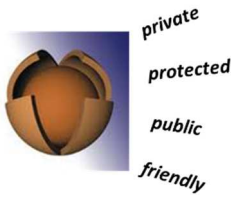
Объектно-ориентированное программирование основано на трех принципах:

- **Инкапсуляции;**
- **Наследовании;**
- **Полиморфизме.**

и одном механизме:

- **Позднее связывание**




**Инкапсуляция** (encapsulation) - это механизм, который объединяет данные и код, манипулирующий этими данными, а также защищает и то, и другое от внешнего вмешательства или неправильного использования.

**Наследование** (inheritance) - это процесс, посредством которого, один объект может наследовать свойства другого объекта и добавлять к ним черты, характерные только для него.

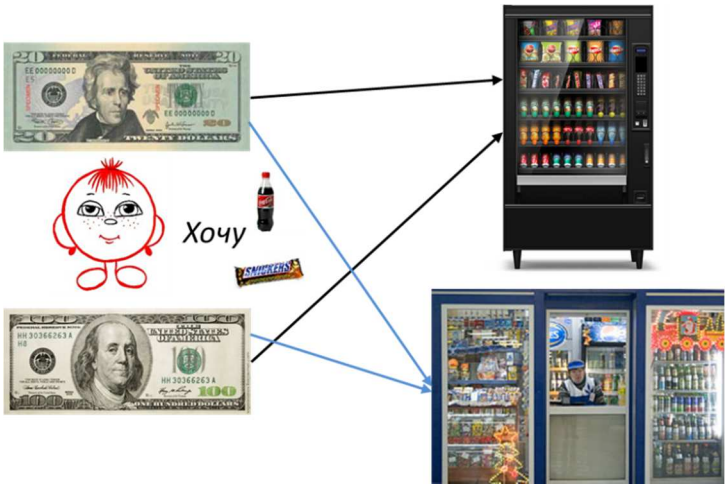
Наследование бывает двух видов:

- *одиночное* - когда каждый класс имеет одного и только одного предка;
- *множественное* - когда каждый класс может иметь любое количество предков.



**Полиморфизм** (polymorphism) - это свойство, которое позволяет одно и то же имя использовать для решения двух или более схожих, но технически разных задач.

**Динамическое связывание (dynamic binding)** – связывание, при котором ассоциация между ссылкой(именем ) и классом не устанавливается, пока объект с заданным именем не будет создан на стадии выполнения программы.




# Простейшие классы и объекты

## Класс

***Классом называется*** *описание совокупности объектов с общими атрибутами, методами, отношениями и семантикой.*

Классы **определяют структуру и поведение** некоторого набора элементов предметной области, для которой разрабатывается программная модель.


Каждый класс имеет свое имя, отличающее его от других классов, и относится к определенному пакету.

Имя класса в пакете должно быть уникальным.

*Объявление класса имеет вид:*

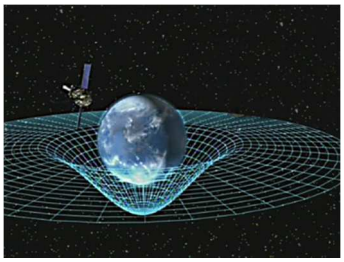
```
[спецификаторы] class имя_класса
                        [extends суперкласс]
                        [implements список_интерфейсов]{
                            /*определение класса*/
}
```

```
package _java._se._02._easyclass;
public class Point2D {
	private int x;
	private int y;

	public void setX(int _x) {
		x = _x;
	}

	public void setY(int _y) {
		y = _y;
	}

	public int getX() {
		return x;
	}

	public int getY() {
		return y;
	}

}
```

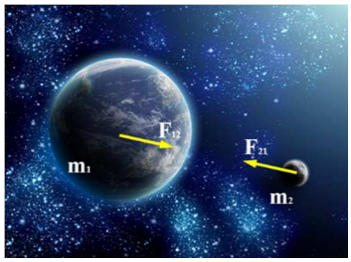
# Объект

**Объект**. Понятие "объект" не имеет в ООП канонического определения.

***Объект*** *- это осязаемая сущность, которая четко проявляет свое поведение.*



***Объект ООП*** *- это совокупность переменных состояния и связанных с ними методов(операций). Эти методы определяют как объект взаимодействует с окружающим миром.*




# Спецификаторы класса

Спецификатор класса может быть:

- **public** (класс доступен объектам данного пакета и вне пакета).
- **final** (класс не может иметь подклассов).
- **abstract** (класс содержит абстрактные методы, объекты такого класса могут создавать только подклассы).

По умолчанию спецификатор доступа устанавливается в ***friendly*** (класс доступен в данном пакете). Данное слово при объявлении вообще не используется и не является ключевым словом языка.

# Методы

Все функции определяются внутри классов и называются **методами**.
*Объявление метода имеет вид*:

[спецификаторы] [static|abstract]
**возвращаемый_тип** *имя_метода*([аргументы]) {
/*тело метода*/
} | ;

*Невозможно создать метод, не являющийся методом класса или объявить метод вне класса.*

***Спецификаторы доступа методов*:**

| | |
|---|---|
| **static** | **public** |
| **final** | **private** |
| **protected** | **abstract** |
| ***friendly*** | **native** |
| **strictfp** | **synchronized** |

# Поля

**Данные** – члены класса, которые называются полями или переменными класса, объявляются в классе следующим образом:

**спецификатор** ***тип*** имя**;**

***Спецификаторы доступа полей класса*:**

| | |
|---|---|
| **static** | **public** |
| **final** | **private** |
| **protected** | ***friendly*** |
| **transient** | **volatile** |

# Конструкторы

**Конструктор** – это метод, который автоматически вызывается при создании объекта класса и выполняет действия ***только*** по *инициализации объекта*;

- Конструктор имеет то же имя, что и класс;
- Вызывается не по имени, а только вместе с ключевым словом **new** при создании экземпляра класса;
- Конструктор не возвращает значение, но может иметь параметры и быть перегружаемым.

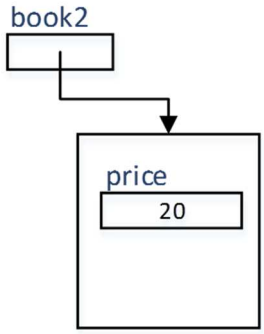
```
package _java._se._02._easyclass;
public class Book {
	private int price;

	public Book() {
```

```
                price = 0;
        }

        public Book(int price){
                setPrice(price);
        }

        public void setPrice(int price){
                this.price = price;
        }

        public int getPrice(){
                return price;
        }
}
```

*Создание объекта имеет вид:*

**ИмяКласса** ссылкаНаОбъект = **new КонструкторКласса(**[аргументы]**);**

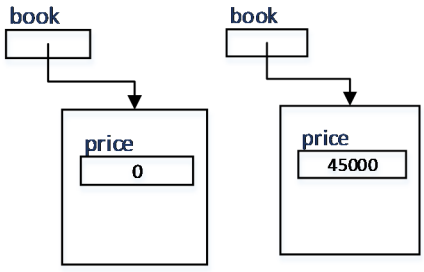
```
package _java._se._02._easyclass;
public class BookInspector {

        public static void main(String[] args) {
                Book book1 = new Book();
                Book book2 = new Book(20);
                java.util.Date date = new java.util.Date();
        }
}
```




## Вызов методов

Для вызова методов в Java используется оператор **. (dot operator)**

```
package _java._se._02._easyclass;
public class BookMethodInspector {

        public static void main(String[] args) {
                Book book = new Book();
                book.setPrice(45_000);
        }

}
```




# Классы и объекты

## Перегрузка методов

**Overload (перегрузка метода)** – определение методов с одинаковым наименованием но различной сигнатурой. Фактически, такие методы – это совершенно разные методы с совпадающим наименованием. Сигнатура метода определяется наименованием метода, а также числом и типом параметров метода.

```
package _java._se._02.classandobject;
```

```
import java.util.Date;

public class DatePrinter {

	public int printDate(String s) {
		System.out.printf("String s=", s);
		return 1;
	}

	public void printDate(int day, int month, int year) {
		System.out.println("int day=" + day);
	}

	public static void printDate(Date d) {
		System.out.printf("Date d=", d);
	}
}
```

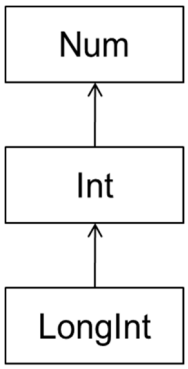
```
package _java._se._02.classandobject;
import java.util.Date;
public class DatePrinterInspector {

	public static void main(String[] args) {
		DatePrinter dp = new DatePrinter();
		int x = dp.printDate("01.01.2015");
		dp.printDate(new Date());
		dp.printDate(1, 1, 2015);
	}
}
```

- Перегрузка реализует «*раннее связывание*».
- Статические методы могут перегружаться нестатическими и наоборот – без ограничений.
- При непосредственной передаче объекта в метод выбор производится в зависимости от типа ссылки на этапе компиляции.

# Перезрузка конструкторов

Перегрузка конструкторов – конструкторы перегружаются аналогично другим методам.

```
package _java._se._02.classandobject;
public class Mathematica {
	public Mathematica(Num obj){}
	public Mathematica(Int obj){}
	public Mathematica(Num obj1, Int obj2){}
	public Mathematica(Int obj1, Int obj2) {}

	public static void main(String[] args){
		Num o1 = new Num();
		Int o2 = new Int();
		LongInt o3 = new LongInt();
		Num o4 = new Int();

		Mathematica m1 = new Mathematica(o1);
		Mathematica m2 = new Mathematica(o2);
		Mathematica m3 = new Mathematica(o3);
		Mathematica m4 = new Mathematica(o4);
		Mathematica m5 = new Mathematica(o1, o2);
		Mathematica m6 = new Mathematica(o3, o2);
```



```
		Mathematica m7 = new Mathematica(o1, o4);//error
		Mathematica m8 = new Mathematica(o3, o4);//error
	}
}
```

При перегрузке всегда следует придерживаться следующих правил:

- не использовать сложных вариантов перегрузки;
- не использовать перегрузку с одинаковым числом параметров;
- заменять при возможности перегруженные методы на несколько разных методов.

## Применение this в конструкторе

Для вызова тела одного конструктора из другого первым оператором вызывающего конструктора должен быть оператор **this([аргументы])**.

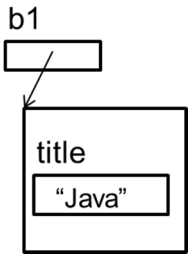
```
package _java._se._02.classandobject;
public class Point2D {
	private int x;
	private int y;

	public Point2D(int x, int y) {
		this.x = x;
		this.y = y;
	}

	public Point2D(int size) {
		this(size, size);
	}
}
```

## Явные и неявные параметры метода

Явные параметры метода определяются списком параметров. Неявный параметр – это this – ссылка на вызвавший метод объект.

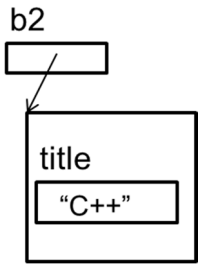
```
package _java._se._02.classandobject;
public class Book {
	private String title;

	public Book(String title) {
		this.title = title;
	}

	public String getTitle() {
		return this.title;
	}
}
```

```
Book b1 = new Book("Java");
Book b2 = new Book("C++");
String s1 = b1.getTitle();
String s2 = b2.getTitle();
```




```
public String getTitle(Book this)
{
      return this.title;
}
```

```
b1.getTitle()
==
getTitle(b1)
```




# Статические методы

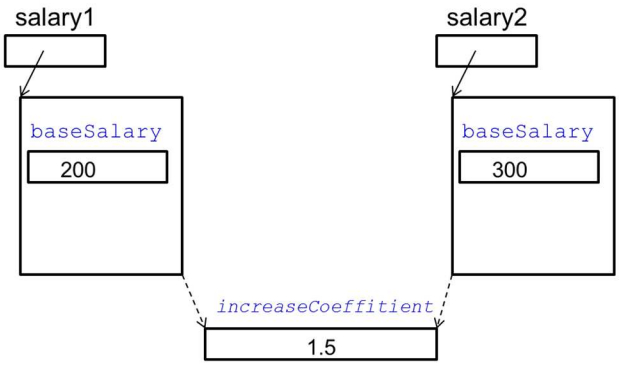
- являются методами класса;
- не привязаны ни к какому объекту;
- не содержат указателя **this** на конкретный объект, вызвавший метод;
- реализуют парадигму «раннего связывания», жестко определяющую версию метода на этапе компиляции;

*Объявление статического метода имеет вид:*

**[спецификаторы] static возвращаемый_тип**

**имя_метода([аргументы]) {/*тело метода*/}**

# Статические поля




```
Salary salary1 = new Salary(200);
Salary salary2 = new Salary(300);
```

Поля данных, объявленные в классе как **static**, являются общими для всех объектов класса и называются **переменными класса**.

Если один объект изменит значение такого поля, то это изменение увидят все объекты.

```
package _java._se._02.classandobject;
public class Salary {
      private double baseSalary;
      public static double increaseCoeffitient = 1.5;

      public Salary(double baseSalary) {
            if (baseSalary <= 0) {
                  this.baseSalary = 100;
            } else {
                  this.baseSalary = baseSalary;
            }
      }
```

```
	public double calcSalary() {
		return baseSalary * increaseCoeffitient;
	}
}
```

Статические поля и методы не могут обращаться к нестатическим полям и методам напрямую (по причине недоступности ссылки **this**), так как для обращения к статическим полям и методам достаточно имени класса, в котором они определены.

```
package _java._se._02.classandobject;
public class SalaryWithWrongStatic {
	private double baseSalary;
	public static double increaseCoeffitient = 1.5;

	public SalaryWithWrongStatic(double baseSalary) {
		if (baseSalary <= 0) {
			this.baseSalary = 100;
		} else {
			this.baseSalary = baseSalary;
		}
	}

	public double calcSalary() {
		return baseSalary * increaseCoeffitient;
	}

	public static void setIncreaseCoeffitient(double newIncreaseCoeffitient) {
		if (newIncreaseCoeffitient <= 0) {
			throw new IllegalArgumentException(
				"Wrong parameter: newIncreaseCoeffitient = "
					+ newIncreaseCoeffitient);
		}
		increaseCoeffitient = newIncreaseCoeffitient;
		// calcSalary(); // ERROR
	}
}
```

# Применение статических методов

Статические методы не работают с объектами, поэтому их использовать следует в двух случаях:

- когда методу *не нужен доступ к состоянию объекта*, а все необходимые параметры задаются явно (например, метод Math.pow(…));
- когда методу нужен *доступ только к статическим полям* класса (статический метод не может получить доступ к нестатическим полям класса, так как они принадлежат объектам, а не классам).

**Статические методы можно вызывать, даже если ни один объект этого класса не создан**. Кроме того, статические методы часто используют в качестве порождающих.

```
package _java._se._02.classandobject;
public class SimpleSingletone {
	private static SimpleSingletone instance;

	private SimpleSingletone(){}

	public static SimpleSingletone getInstance()
	{
```

```
            if (null == instance){
                  instance = new SimpleSingletone();
            }
            return instance;
      }
}
```

Статические поля используются довольно редко, а вод поля **static final** наоборот часто.

Статические константы нет смысла делать закрытыми, а обращаются к ним через имя класса:

**имя_класса.имя_статической_константы**

```
public class System
{
      …
      public static final PrintStream out = …
      …
}
```

# Логический блок инициализации

При описании класса могут быть использованы логические блоки. **Логическим блоком называется код**, заключенный в фигурные скобки и не принадлежащий ни одному методу текущего класса.

**{ /* код */ }**

При создании объекта блоки инициализации класса вызываются последовательно, в порядке размещения, вместе с инициализацией полей как простая последовательность операторов, и только после выполнения последнего блока будет вызван конструктор класса.

```
package _java._se._02.classandobject;
public class LogicBlock {
      private int x = 89;

      {
            x = 20;
      }

      public LogicBlock() {
      }

      public LogicBlock(int x) {
            this.x = x;
      }

      public int getX() {
            return x;
      }

      public static void main(String[] args) {
            LogicBlock logic1 = new LogicBlock();
            LogicBlock logic2 = new LogicBlock(200);
            System.out.println("logic1.x = " + logic1.getX());
            System.out.println("logic2.x = " + logic2.getX());
      }
}
```

Логические блоки чаще всего используются в качестве инициализаторов полей, но могут содержать вызовы методов как текущего класса, так и не принадлежащих ему.

```
package _java._se._02.classandobject;
import java.util.Date;
public class LogicBlock2 {

	{
		System.out.println("logic (1) id=" + this.id);
	}
	private int id = 7;
	{
		System.out.println("logic (2) id=" + id);
		Date d = new Date();
		calc(d);
	}
	public LogicBlock2(int d) {
		id = d;
		System.out.println("конструктор id=" + id);
	}

	{
		id = 10;
		System.out.println("logic (3) id=" + id);
	}

	private void calc(Date d){
		System.out.println(d.getTime());
	}

	public static void main(String[] args) {
		LogicBlock2 logic = new LogicBlock2(3);
	}
}
```

## Статические блоки инициализации

Для инициализации статических переменных существуют **статические блоки инициализации**. В этом случае фигурные скобки предваряются ключевым словом **static**. В этом случае он вызывается только один раз в жизненном цикле приложения:

- при **создании объекта** или
- при **обращении к статическому методу (полю)** данного класса.

```
package _java._se._02.classandobject;
public class StaticBlock {
	private double baseSalary;
	public static double increaseCoeffitient = 2.5;

	static{
		increaseCoeffitient = 1.5;
		//baseSalary = 100; // error
	}

	public StaticBlock(double baseSalary) {
		this.baseSalary = baseSalary;
	}
	public double calcSalary() {
		return baseSalary * increaseCoeffitient;
```

```
	}
	public static void setIncreaseCoeffitient(double newIncreaseCoeffitient) {
		increaseCoeffitient = newIncreaseCoeffitient;
	}
}
```

# Инициализация полей класса

Общий порядок инициализации следующий

1. При создании объекта или при первом обращении к статическому методу (полю) статические поля инициализируются значениями по умолчанию.
2. Инициализаторы всех статических полей и статические блоки инициализации выполняются в порядке их перечисления в объявлении класса.
3. При вызове конструктора класса все поля данных инициализируются своими значениями, предусмотренными по умолчанию.
4. Инициализаторы всех полей и блоки инициализации выполняются в порядке их перечисления в объявлении класса.
5. Если в первой строке конструктора вызывается тело другого конструктора, то выполняется вызванный конструктор.
6. Выполняется тело конструктора.

# final

Модификатор **final** используется для определения констант в качестве члена класса, локальной переменной или параметра метода.

**Константа** может быть объявлена **как поле экземпляра класса, но не проинициализирована**. В этом случае она должна быть проинициализирована в логическом блоке класса или конструкторе, но только в одном из указанных мест. Константные статические поля могут быть проинициализированы или при объявлении, или в статическом блоке инициализации.

Значение по умолчанию константа получить не может в отличие от переменных класса.

```
package _java._se._02.classandobject;
import java.util.Date;

public class FinalVar {
	private final int finalVar;
	public static final int staticFinalVar;
	private final Date date;

	static {
		staticFinalVar = 2;
	}
	{
		finalVar = 1;
	}

	public void method(final int var) {
		final int temp = 12;
```

```
		// var++; // error
		date.setYear(2999 - 1900);
		// date = new Date(); //error
	}

	public FinalVar() {
		// finalVar = 3;//error
		// staticFinalVar = 4;//error
		date = new Date();
	}

}
```

# native

Модификатор native указывает на то, что метод написан не на Java. Методы, помеченные **native**, можно переопределять обычными методами в подклассах.

Тело нативного метода должно заканчиваться на (;) как в абстрактных методах, идентифицируя то, что реализация опущена.

```
public native int loadCripto(int num);
```

```
package java.lang;
public class Object {
…
protected native Object clone() throws CloneNotSupportedException;
…
}
```

# synchronized

При использовании нескольких потоков управления в одном приложении необходимо синхронизировать методы, обращающиеся к общим данным.

Когда интерпретатор обнаруживает **synchronized**, он включает код, блокирующий доступ к данным при запуске потока и снимающий блок при его завершении.

```
package _java._se._02.classandobject;
public class SinchronizedSingleton {
	private static SinchronizedSingleton instance;

	private SinchronizedSingleton(){}

	public static synchronized SinchronizedSingleton getInstance()
	{
		if (null == instance){
			instance = new SinchronizedSingleton();
		}
		return instance;
	}
}
```

# Класс Object

**Класс java.lang.Object** - родительский для всех классов.

Содержит следующие методы:

- **protected Object clone()** – создает и возвращает копию вызывающего объекта;

- **boolean equals(Object ob)** – предназначен для переопределения в подклассах с выполнением общих соглашений о сравнении содержимого двух объектов;
- **Class<? extends Object> getClass()** – возвращает объект типа **Class;**
- **protected void finalize()** – вызывается перед уничтожением объекта автоматическим сборщиком мусора (garbage collection);
- **int hashCode()** – возвращает хэш-код объекта;
- **String toString()** – возвращает представление объекта в виде строки.

## Переопределение метода equals()

Метод **equals()** при сравнении двух объектов возвращает истину, если содержимое объектов эквивалентно, и ложь – в противном случае.
При переопределении должны выполняться соглашения:

- **рефлексивность** – объект равен самому себе;
- **симметричность** – если **x.equals(y)** возвращает значение **true**, то и **y.equals(x)** всегда возвращает значение **true**;
- **транзитивность** – если метод **equals()** возвращает значение true при сравнении объектов **x** и **y,** а также **y** и **z**, то и при сравнении **x** и **z** будет возвращено значение **true**;
- **непротиворечивость** – при многократном вызове метода для двух не подвергшихся изменению за это время объектов возвращаемое значение всегда должно быть одинаковым;
- **ненулевая ссылка** при сравнении с литералом **null** всегда возвращает значение **false**.

## Переопределение метода hashCode()

Метод **int hashCode()** возвращает хэш-код объекта, вычисление которого управляется следующими соглашениями:

- во время работы приложения значение хэш-кода объекта не изменяется, если объект не был изменен;
- все одинаковые по содержанию объекты одного типа должны иметь одинаковые хэш-коды;
- различные по содержанию объекты одного типа могут иметь различные хэш-коды.

*Следует переопределять всегда, когда переопределен метод* **equals().**

## toString()

Метод **toString()** следует переопределять таким образом, чтобы кроме стандартной информации о пакете (опционально), в котором находится класс, и самого имени класса (опционально), он возвращал значения полей объекта, вызвавшего этот метод (то есть всю полезную информацию объекта), вместо хэш-кода, как это делается в классе **Object**.
В классе **Object** возвращает строку с описанием объекта в виде:

**getClass().getName() + '@' + Integer.toHexString(hashCode())**

```
package _java._se._02.classandobject;
class Pen {
	private int price;
```

```
	private String producerName;

	public Pen(int price, String producerName) {
		this.price = price;
		this.producerName = producerName;
	}

	public boolean equals(Object obj) {
		if (this == obj) {
			return true;
		}
		if (null == obj) {
			return false;
		}
		if (getClass() != obj.getClass()) {
			return false;
		}

		Pen pen = (Pen) obj;
		if (price != pen.price) {
			return false;
		}
		if (null == producerName) {
			return (producerName == pen.producerName);
		} else {
			if (!producerName.equals(pen.producerName)) {
				return false;
			}
		}

		return true;
	}

	public int hashCode() {
		return (int) (31 * price + ((null == producerName) ? 0 : producerName
				.hashCode()));
	}

	public String toString() {
		return getClass().getName() + "@" + "price: " + price
				+ ", producerName: " + producerName;
	}

}
```

# finalize

Иногда при уничтожении объект должен выполнять какое-либо действие. Используя метод finalize(), можно определить конкретные действия, которые будут выполняться непосредственно перед удалением объекта сборщиком мусора.

```
protected void finalize() throws Throwable{
	try{
		// освобождение ресурсов
	}finally{
		super.finalize();
	}
}
```

Метод finalize() вызывается, когда сборщик мусора решит уничтожить объект.

"Сборка мусора" происходит нерегулярно во время выполнения программы. Можно ее выполнить вызовом метода **System.gc()** или **Runtime.getRuntime().gc()**.

Вызов метода **System.runFinalization()** приведет к запуску метода **finalize()** для объектов утративших все ссылки.

По возможности следует избегать использование метода finalize (из-за невозможности предсказать последствия его работы). Лучше освобождать ресурсы программно.

## Методы с переменным числом параметров

Возможность передачи в метод нефиксированного числа параметров позволяет отказаться от предварительного создания массива объектов для его последующей передачи в метод.

```
void methodName(Тип ... args){}
```

Чтобы передать несколько массивов в метод по ссылке, следует использовать следующее объявление:

```
void methodName(Тип[]... args){}
```

В списке аргументов аргумент с переменным числом параметров должен быть самым последним.

```
    void methodName(char s, int ... args){}
void methodName(int ... x, char s){}//error
```

Методы с переменным числом аргументов могут быть перегружены:

```
void methodName(Integer...args) {}
void methodName(int x1, int x2) {}
 void methodName(String...args) {}
```

```
package _java._se._02.classandobject;
public class VarArgs {
	public static int getArgCount(Integer... args) {
		if (args.length == 0) {
			System.out.print("No arg");
		}
		for (int i : args) {
			System.out.print("arg:" + i + "  ");
		}
		return args.length;
	}

	public static void getArgCount(Integer[]... args) {
		if (args.length == 0) {
			System.out.print("No arg2");
		}
		for (Integer[] mas : args) {
			for (int x : mas) {
				System.out.print("arg2:" + x + "  ");
			}
		}

	}

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

```
                System.out.println("N=" + getArgCount(7, 71, 555));
                Integer[] i = { 1, 2, 3, 4, 5, 6, 7 };
                System.out.println("N=" + getArgCount(i));
                getArgCount(i, i);
                // getArgCount(); //error
        }
}

package _java._se._02.classandobject;
public class DemoOverload {

        public static void printArgCount(Object... args) { // 1
                System.out.println("Object args: " + args.length);
        }

        public static void printArgCount(Integer[]... args) { // 2
                System.out.println("Integer[] args: " + args.length);
        }

        public static void printArgCount(int... args) { // 3
                System.out.print("int args: " + +args.length);
        }
        public static void main(String[] args) {
                Integer[] i = { 1, 2, 3, 4, 5 };
                printArgCount(7, "No", true, null);
                printArgCount(i, i, i);
                printArgCount(i, 4, 71);
                printArgCount(i); // будет вызван метод 1
                printArgCount(5, 7); //неопределенность!
        }
}
```

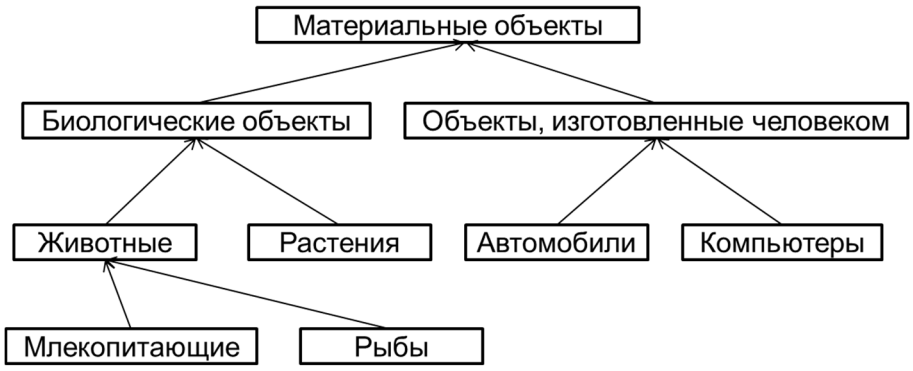
# Наследование

## Понятие наследования

Один класс может наследовать или расширять поля и методы другого класса с помощью ключевого слова **extends**.

Класс, который выступает базой для расширения, называют *суперклассом*, класс, который непосредственно проводит расширение, - *подклассом*.

Иерархии классов



# Синтаксис наследования

Подкласс имеет доступ **ко всем открытым и защищенным полям** и методам суперкласса, так, словно они описаны в подклассе: производный класс не имеет доступа к закрытым полям и методам класса. Также подкласс может добавлять методы и переопределять методы.

*Объявление производного класса имеет вид:*

**[спецификаторы] class имя_класса**
**extends суперкласс [implements список_интерфейсов]{**
**/*определение класса*/**
**}**

# Вызов конструкторов при наследовании

При создании объектов производного класса, конструктор производного класса вызывает соответствующий конструктор базового класса с помощью ключевого слова super(параметры).

Вызов конструктора базового класса из конструктора производного должен быть произведен в первой строке конструктора производного класса.

Если конструктор производного класса явно не вызывает конструктор базового, то происходит вызов конструктора по умолчанию базового класса, в этом случае в базовом классе должен быть определен конструктор по умолчанию.

```
package _java._se._02.inheritance;
public class Book {
    private String title;
    private int price;

    public Book() {}

    public Book(String title, int price)    {
        this.title = title;
        this.price = price;
    }

    public String getInfo()    {
        return "Название: "+title+", цена: "+price;
    }
}

package _java._se._02.inheritance;
public class ProgrammerBook extends Book {
       private String language;

       public ProgrammerBook() {
       }

       public ProgrammerBook(String title, int price, String language) {
              super(title, price);
              this.language = language;
       }

       public String getInfo() {
```

```
                return super.getInfo() + ", язык: " + language;
        }
}
```

Если метод базового класса переопределен (имеет ту же сигнатуру) в производном классе, то такой метод базового класса можно вызвать из производного с помощью конструкции.

Следует помнить, что при вызове **super.метод()** обращение производится к ближайшему суперклассу.

**super.имя_метода(параметры);**

## Инициализация статических полей

При создании объекта производного класса вызываются сначала статические блоки базового класса, а затем производного, при условии, что до этого они вызваны не были.

Если какой-либо статический блок отработал к моменту создания объекта – его вызов пропускается.

Инициализаторы всех статических полей и статические блоки инициализации выполняются в порядке их перечисления в объявлении класса.

```
package _java._se._02.inheritance;
class Man {
        public static String form = "man";
        static {
                System.out.println("static block in Man.");
        }
        public static void stMan() {
                System.out.println("static method in Man.");
        }
}


package _java._se._02.inheritance;
class Doctor extends Man {
        static {
                System.out.println("static block in Doctor");
        }
        public static void stDoctor() {
                System.out.println("static method in Doctor.");
        }
}


package _java._se._02.inheritance;
public class InitialBlockInheritance {
        public static void main(String[] args) {
                Doctor.stMan();
                System.out.println("Run.");
                Doctor doctor = new Doctor();
                System.out.println(doctor.form);
                Doctor.stDoctor();
        }
}
```

## Инициализация полей экземпляра класса

Инициализаторы всех полей и блоки инициализации выполняются в порядке их перечисления в объявлении класса, перед выполнением конструктора сначала для базового класса, затем для производного.

```
package _java._se._02.inheritance;
public class ManBlock {
	private int age;
	{
		age = 0;
		System.out.println("logic block in Man.");
	}

	public ManBlock() {
		System.out.println("Constructor in Man.");
	}

	public int getAge() {
		return age;
	}

}

package _java._se._02.inheritance;
public class DoctorBlock extends ManBlock {
	private String speciality;
	{
		System.out.println("logic block in Doctor");
		speciality = "surgeon";
	}

	public DoctorBlock() {
		System.out.println("Constructor in Doctor.");
	}

	public String getSpeciality() {
		return speciality;
	}
}

package _java._se._02.inheritance;
public class InitialBlockInheritance {
	public static void main(String[] args) {
		DoctorBlock doctor = new DoctorBlock();
		System.out.println("Age: " + doctor.getAge());
		System.out.println("Speciality: " + doctor.getSpeciality());

	}
}
```

# Переопределение методов

Переопределенным методом называют метод, описанный в производном классе, сигнатура этого метода совпадает с сигнатурой метода, описанного в суперклассе.

```
package _java._se._02.inheritance._override;
class MedicalStaff {
	public void info() {
		System.out.println("MedicalStaff");
	}
}
```

```
package _java._se._02.inheritance._override;
class Doctor extends MedicalStaff {
	public void info() {
		System.out.println("Doctor");
	}
}

package _java._se._02.inheritance._override;
public class Hospital {
	public static void main(String[] args) {
		Doctor doctor = new Doctor();
		doctor.info();
		MedicalStaff med = new Doctor();
		med.info();
	}
}
```

## Вызов переопределенных методов

Объектная переменная базового класса может ссылаться на объекты как базового, так и производного классов. Такая возможность называется полиморфизмом.

Автоматический выбор нужного метода во время выполнения программы называется динамическим связыванием (dynamic binding).

Для статических методов в Java полиморфизм неприменим.

```
package _java._se._02.inheritance._override2;
class MedicalStaff {

	public static void staticMedical() {
		System.out.println("staticMedicalStaff");
	}

	public void prescriptionMedicine() {
		System.out.println("prescriptionMedicine");
	}

	public void info() {
		System.out.println("MedicalStaff");
	}
}


package _java._se._02.inheritance._override2;
class Doctor extends MedicalStaff {

	public static void staticMedical() {
		System.out.println("staticDoctor");
	}

	public void createMedicine() {
		System.out.println("createMedicine");
	}

	public void info() {
		System.out.println("Doctor");
	}
```

```
}

public class Hospital {
	public static void main(String[] args) {
		MedicalStaff med = new Doctor();
		Doctor doctor = new Doctor();

		med.info();
		med.prescriptionMedicine();
		// med.createMedicine();
		med.staticMedical();

		doctor.info();
		doctor.prescriptionMedicine();
		doctor.createMedicine();
		doctor.staticMedical();
	}
}
```

Статические методы не переопределяются нестатическими, нестатические методы не переопределяются статическими.

```
class MedicalStaff {
	public void info1() {
		System.out.println("MedicalStaff1");
	}
	public static void info2() {
		System.out.println("MedicalStaff2");
	}
}
class Doctor extends MedicalStaff {
	public static void info1() {//Error
		System.out.println("Doctor1");
	}
	public void info2() {//Error
		System.out.println("Doctor2");
	}
}
```

# Методы подставки

С пятой версии языка появилась возможность при переопределении методов указывать другой тип возвращаемого значения, в качестве которого можно использовать только типы, находящиеся ниже в иерархии наследования, чем исходный тип.

```
package _java._se._02.inheritance;

class Course {
}

class BaseCourse extends Course {
}

class CourseHelper {
	public Course getCourse() {
		System.out.println("Course");
		return new Course();
	}
}

class BaseCourseHelper extends CourseHelper {
```

```
        public BaseCourse getCourse() {
                System.out.println("BaseCourse");
                return new BaseCourse();
        }
}

public class CourseInspector {
        public static void main(String[] args) {
                CourseHelper bch = new BaseCourseHelper();
                Course course = bch.getCourse();
                // BaseCourse course = bch.getCourse();//ошибка компиляции
                bch.getCourse();
        }
}
```

В данном случае при компиляции в подклассе **BaseCourseHelper** создаются два метода. При обращении к методу **getCourse()** версия метода определяется «ранним связыванием» без использования полиморфизма, но при выполнении вызывается метод-подставка.

## Перегрузка методов

Методы с одинаковыми именами, но с различающимися списком параметров и возвращаемыми значениями могут находиться в разных классах одной цепочки наследования и также будут являться перегруженными.

Статические методы перегружаются нестатическими, нестатические методы перегружаются статическими.

## Предотвращение переопределения методов

Чтобы предотвратить переопределение методов их необходимо объявить терминальными с помощью ключевого слова final.

```
public class Book {
…
        public final int getPrice(){
                return price;
    }…
}
public class ProgrammerBook extends Book{
…
    public final int getPrice(){ // error
                return price;
    } …
}
```

## Предотвращение наследования

Классы, объявленные как терминальными, нельзя расширить. Объявить терминальный класс можно следующим образом.

```
public final class Book {}
public class ProgrammerBook extends Book{} // error
```

Если класс объявлен терминальным, то это не значит, что его поля стали константными.

## Приведение типов при наследовании

На основе описания классов компилятор проверяет, сужает или расширяет возможности класса программист, объявляющий переменную.

Если переменной суперкласса присваивается объект подкласса, возможности класса сужаются, и компилятор без проблем позволяет программисту сделать это.

Если, наоборот, объект суперкласса присваивается переменной подкласса, возможности класса расширяются, поэтому программист должен подтвердить это с помощью обозначения, предназначенного для приведения типов, указав в скобках имя подкласса (subclass).

```
package _java._se._02.inheritance.type;

class Book {
}
class ProgrammerBook extends Book {
}

public class BookInspector {

	public static void main(String[] args) {
		Book book = new ProgrammerBook();
		ProgrammerBook progrBook = new ProgrammerBook();

		Book goodBook = progrBook;
		ProgrammerBook goodProgrBook = (ProgrammerBook) book;

		Book simpleBook = new Book();
		ProgrammerBook simpleProgrBook = (ProgrammerBook) simpleBook;// error
	}
}
```

При недопустимом преобразовании типов при выполнении программы система обнаружит несоответствие и возбудит исключительную ситуацию. Если её не перехватить, то работа программы будет остановлена.

Перед приведением типов следует проверить его на корректность. Делается это с помощью оператора **instanceof**.

```
Book simpleBook = ...;
ProgrammerBook simpleProgrBook = ...;

if (simpleBook instanceof ProgrammerBook){
	simpleProgrBook = (ProgrammerBook)simpleBook;
}
```

Компилятор не позволить выполнить некорректное приведение типов. Например, приведение типов.

```
Date dt = (Date)SimpleBook;
```

приведет к ошибке на стадии компиляции, поскольку класс **Date** не является подклассом класса **Book**.

# Переопределение метода equals

При переопределении метода equalsв производных классах, для сравнения его базовой составляющей следует вызывать метод equals базового класса.

```
package _java._se._02.inheritance;
class VipPen extends Pen {
	private int preciousMetalCost;

	public VipPen(int price, String producerName, int preciousMetalCost) {
		super(price, producerName);
		this.preciousMetalCost = preciousMetalCost;
	}

	public boolean equals(Object obj) {
		if (this == obj) {
			return true;
		}
		if (null == obj) {
			return false;
		}
		if (getClass() != obj.getClass()) {
			return false;
		}

		VipPen vipPen = (VipPen) obj;
		if (!super.equals(vipPen)) {
			return false;
		}
		if (preciousMetalCost != vipPen.preciousMetalCost) {
			return false;
		}
		return true;
	}
}
```

# Абстрактные методы и классы

Часто при проектировании иерархии классов верхние классы иерархии становятся все более и более абстрактными, так что реализовывать некоторые методы в них не имеет никакого смысла.

Однако удалить их из класса нельзя, так как при дальнейшем использовании базовых объектных ссылок на объекты производных классов необходим доступ к переопределенным методам, а он возможен только при наличии в них метода с такой же сигнатурой как в базовом классе. В таком случае метод следует объявлять абстрактным.

В классе, где метод объявляется абстрактным, его реализация не требуется.

Если в классе есть абстрактные методы, то класс следует объявить абстрактным.

Абстрактные классы и методы объявляются с ключевым словом abstract.

При расширении абстрактного класса все его абстрактные методы необходимо определить или подкласс также объявить абстрактным.

Нельзя создавать объекты абстрактных классов, однако можно объявлять объектные переменные.

```
package _java._se._02.inheritance._abstract;
```

```
abstract class Course {
	public abstract String getInformation();
}

class BaseCourse extends Course {
	public String getInformation() {
		return "Base course";
	}
}

class OptionalCourse extends Course {
	public String getInformation() {
		return "Optional course";
	}
}

public class CourseInspector {

	public static void main(String[] args) {
		Course course1 = new BaseCourse();
		Course course2 = new OptionalCourse();
		System.out.println(course1.getInformation());
		System.out.println(course2.getInformation());
	}
}
```

## Наследование от стандартных классов

Кроме собственных Java позволяет расширять и стандартные классы.

```
import java.sql.Time;
public class MyTime extends Time {
    public MyTime(long i) {
        super(i);
    }
    public String current(){
        long hours = getHours();
        if(hours >= 4 && hours < 12) return "утро";
        else if ((hours >12 && hours < 17)) return "день";
        else if (hours >= 17 && hours < 23) return "вечер";
        else return "ночь";
    }
    public static void main(String[] args){
        MyTime mytime = new MyTime(300000000);
        System.out.println(mytime.current());
    }
}
```

# Интерфейсы

## Определение интерфейса

**Интерфейсы** в Java применяются для добавления к классам новых возможностей, которых нет и не может быть в базовых классах.

Интерфейсы говорят о том, что класс может делать, но не говорят, как он должен это делать.

Интерфейс только гарантирует (определяет контракт), какие методы должен выполнять класс, но как класс выполняет контракт интерфейс контролировать не может.

## Определение интерфейса

*Объявление интерфейса имеет вид:*

**[спецификаторы] interface имя_интерфейса**
**extends имя_базового_интерфейса {**
**/*объявление интерфейса*/**
**}**

Поля интерфейса по умолчанию являются **final static**. Все методы по умолчанию открыты (**public**).

```
public interface Square {
    double PI = 3.1415926;
       double square();
}
```

**Реализация интерфейса** происходит в классе с помощью ключевого слова implements.

Если реализуемых интерфейсов несколько, то они перечисляются через запятую.

Интерфейс считается реализованным, когда в классе и/или в его суперклассе реализованы все методы интерфейса.

```
package _java._se._02._interface;
public class Quadrate implements Square {
       private int a;

       public Quadrate(int a) {
              this.a = a;
       }

       public double square() {
              return a * a;
       }

       public void print() {
              System.out.println("Square box: " + square());
       }
}


package _java._se._02._interface;
public class Circle implements Square {
       private int r;

       public Circle(int r) {
              this.r = r;
       }

       public double square() {
              return r * r * Square.PI;
       }

       public void print() {
```

```
                System.out.println("Square circle: " + square());
        }
}


package _java._se._02._interface;
public class Rectangle implements Square {
        private int a, b;

        public Rectangle(int a, int b) {
                this.a = a;
                this.b = b;
        }

        public double square() {
                return a * b;
        }

        public void print() {
                System.out.println("Square rectangle: " + square());
        }
}

package _java._se._02._interface;
public class SquareInspector {

        public static void main(String[] args) {
                Quadrate box = new Quadrate(4);
                Rectangle rectangle = new Rectangle(2, 3);
                Circle circle = new Circle(3);
                box.print();
                rectangle.print();
                circle.print();
                System.out.println("Box: " + box.square());
                System.out.println("Rectangle: " + rectangle.square());
                System.out.println("Circle: " + circle.square());

        }

}
```

# Свойства интерфейсов

- С помощью оператора **new** нельзя создать экземпляр интерфейса.
- Можно объявлять интерфейсные ссылки.
- Интерфейсные ссылки должны ссылать на объекты классов, реализующих данный интерфейс.
- Через интерфейсную ссылку можно вызвать только методы определенные с интерфейсе.

```
package _java._se._02._interface;
public class InterfaceProperties {

        public static void main(String[] args) {
                Quadrate box = new Quadrate(4);
         //box = new Square();  // ERROR
         Square square;
         square = box;
         box.print();
```

```
            System.out.println("Box: "+square.square());
            // square.print() // ERROR
            if (box instanceof Square)    {
                System.out.println("box implements square");
            }
        }
}
```

- С помощью оператора **instanseof** можно проверять, реализует ли объект определенный интерфейс.
- Если класс не полностью реализует интерфейс, то он должен быть объявлен как **abstract**.

```
package _java._se._02._interface;
public abstract class Ellipse implements Square {

}
```

- Интерфейс может быть расширен при помощи наследования от другого интерфейса, синтаксис в этом случае аналогичен синтаксисом наследования классов .

```
interface Collection<E>{

}

public interface List<E> extends Collection<E> {

}
```

## Вложенные интерфейсы

Интерфейсы можно вложить (объявить членом) другого класса или интерфейса.

Когда вложенный интерфейс использует вне области вложения, то он используется вместе с именем класса или интерфейса.

```
public interface Map<K, V> {

        interface Entry<K, V>{

        }
}
```

## Клонирование объектов. Интерфейс Cloneable

Для создания нового объекта с таким же состоянием используется клонирование объекта.

Метод **clone()** класса **Object** объявлен с атрибутом доступа **protected**.

Клонирование объекта можно реализовать, имплементировав интерфейс **Cloneable** и реализовав копирование состояний полей и агрегированных объектов.

Интерфейс **Cloneable** не содержит методов относится к помеченным (**tagged**) интерфейсам, а его реализация гарантирует, что метод **clone**() класса **Object** возвратит точную копию вызвавшего его объекта с воспроизведением значений всех его полей.

В противном случае метод генерирует исключение **CloneNotSupportedException**.

```
package java.lang;
public interface Cloneable {

}
```

```
package _java._se._02._interface;
import java.util.Date;
public class Department implements Cloneable {
	private Integer name;
	private Date date = new Date();

	public Object clone() throws CloneNotSupportedException {
		Department obj = null;

		obj = (Department) super.clone();
		if (null != this.date) {
			obj.date = (Date) this.date.clone();
		}

		return obj;
	}
}
```

```
package _java._se._02._interface;
import java.util.ArrayList;
import java.util.List;

class Faculty implements Cloneable {
	private String name;
	private int numberDepartments;
	private List<Department> departmentList;

	public Object clone() throws CloneNotSupportedException {
		Faculty obj = null;

		obj = (Faculty) super.clone();
		if (null != this.departmentList) {
			ArrayList<Department> tempList = new
ArrayList<Department>(this.departmentList.size());
			for (Department listElem : this.departmentList) {
				tempList.add((Department) listElem.clone());
			}
			obj.departmentList = tempList;
		}

		return obj;
	}
}
```

# Сравнение объектов. Интерфейс Comparable

Метод sort(…) класса Arrays позволяет упорядочивать массив, переданный ему в качестве параметра. Для элементарных типов правила определения больше/меньше известны.

```
package _java._se._02._interface;
import java.util.Arrays;
public class SortArray {
```

```
	public static void main(String[] args) {
		int[] mas = { 3, 6, 5, 1, 2, 9, 8 };
		printArray(mas);
		Arrays.sort(mas);
		printArray(mas);
	}

	public static void printArray(int[] ar) {
		for (int i : ar)
			System.out.print(i + " ");
		System.out.println();
	}
}
```

**Естественный порядок сортировки (natural sort order)** — естественный и реализованный по умолчанию (реализацией метода **compareTo** интерфейса **java.lang.Comparable**) способ сравнения двух экземпляров одного класса.

- **int compareTo(E other)** — сравнивает **this** объект с **other** и возвращает отрицательное значение если this<other, 0 — если они равны и положительное значение если this>other.

```
public interface Comparable<T> {
	int  compareTo (T other);
}
```

Метод **compareTo** должен выполнять следующие условия:

- **sgn(x.compareTo(y)) == -sgn(y.compareTo(x))**
- если **x.compareTo(y)** выбрасывает исключение, то и **y.compareTo(x)** должен выбрасывать то же исключение
- если **x.compareTo(y)>0** и **y.compareTo(z)>0**, тогда **x.compareTo(z)>0**
- если **x.compareTo(y)==0**, и **x.compareTo(z)==0**, то и **y.compareTo(z)==0**
- **x.compareTo(y)==0**, тогда и только тогда, когда **x.equals(y) ;** (правило рекомендуемо но не обязательно)

```
package _java._se._02._interface;
public class Person implements Comparable<Person> {
	private String firstName;
	private String lastName;
	private int age;

	public Person(String firstName, String lastName, int age) {
		this.firstName = firstName;
		this.lastName = lastName;
		this.age = age;
	}

	public int getAge() {
		return age;
	}

	public int compareTo(Person anotherPerson) {
		int anotherPersonAge = anotherPerson.getAge();
		return this.age - anotherPersonAge;
	}
}
```

# Сравнение объектов. Интерфейс Comparator

При реализации интерфейса **Comparator<T>** существует возможность сортировки списка объектов конкретного типа по правилам, определенным для этого типа.

Для этого необходимо реализовать метод **int compare(T ob1, T ob2)**, принимающий в качестве параметров два объекта для которых должно быть определено возвращаемое целое значение, знак которого и определяет правило сортировки.

**java.util.Comparator** — содержит два метода:

- **int compare(T o1, T o2)** — сравнение, аналогичное **compareTo**
- **boolean equals(Object obj)** — **true** если **obj** это **Comparator** и у него такой же принцип сравнения.

```
package _java._se._02._interface;
public abstract class GeometricObject {
	public abstract double getArea();
}

package _java._se._02._interface;
public class RectangleGO extends GeometricObject {
	private double sideA;
	private double sideB;

	public RectangleGO(double a, double b) {
		sideA = a;
		sideB = b;
	}

	@Override
	public double getArea() {
		return sideA * sideB;
	}
}

package _java._se._02._interface;
public class CircleGO extends GeometricObject {

	private double radius;

	public CircleGO(double r) {
		radius = r;
	}

	@Override
	public double getArea() {
		// TODO Auto-generated method stub
		return 2 * 3.14 * radius * radius;
	}
}

package _java._se._02._interface;

import java.util.Comparator;

public class GeometricObjectComparator implements Comparator<GeometricObject> {
	public int compare(GeometricObject o1, GeometricObject o2) {
		double area1 = o1.getArea();
		double area2 = o2.getArea();
		if (area1 < area2) {
			return -1;
```

```
			} else if (area1 == area2) {
				return 0;
			} else {
				return 1;
			}
		}
}
```

```
package _java._se._02._interface;

import java.util.Comparator;
import java.util.Set;
import java.util.TreeSet;

public class TreeSetWithComparator {
	public static void main(String[] args) {
		Comparator<GeometricObject> comparator = new GeometricObjectComparator();
		Set<GeometricObject> set = new TreeSet<GeometricObject>(comparator);
		set.add(new RectangleGO(4, 5));
		set.add(new CircleGO(40));
		set.add(new CircleGO(40));
		set.add(new RectangleGO(4, 1));
		System.out.println("A sorted set of geometric objects");
		for (GeometricObject elements : set) {
			System.out.println("area = " + elements.getArea());
		}
	}

}
```

# Параметризированные классы.

## Определение параметризованного класса

С помощью шаблонов можно создавать **параметризованные** (родовые, generic) **классы и методы**, что позволяет использовать более строгую типизацию.

Пример класса-шаблона с двумя параметрами:

```
package _java._se._02._generics;

public class Message<T1, T2> {
	T1 id;
	T2 name;
}
```

T1, T2 – фиктивные типы, которые используются при объявлении атрибутов класса. Компилятор заменит все фиктивные типы на реальные и создаст соответствующий им объект.

Объект класса Message можно создать, например, следующим образом:

```
		Message <Integer, String> ob =
				new Message <Integer, String> ();
```

```
public class Optional <T> {
	private T value;
	public Optional() {
	}
```

```
	public Optional(T value) {
		this.value = value;
	}
	public T getValue() {
		return value;
	}
	public void setValue(T val) {
		value = val;
	}
	public String toString() {
	if (value == null) return null;
	    return value.getClass().getName() + " " + value;
	}
}

package _java._se._02._generics;

public class OptionalDemo {
	public static void main(String[] args) {

		Optional<Integer> ob1 = new Optional<Integer>();

		ob1.setValue(1);

		// ob1.setValue("2");// ERROR
		int v1 = ob1.getValue();
		System.out.println(v1);

		// параметризация типом String
		Optional<String> ob2 = new Optional<String>("Java");
		String v2 = ob2.getValue();
		System.out.println(v2);

		// ob1 = ob2; //ERROR

		Optional ob3 = new Optional();

		System.out.println(ob3.getValue());
		ob3.setValue("Java SE 6");

		System.out.println(ob3.toString());

		ob3.setValue(71);
		System.out.println(ob3.toString());

		ob3.setValue(null);
	}

}
```

# Применение extends при параметризации

*Объявление* generic-типа в виде **<T>**, несмотря на возможность использовать любой тип в качестве параметра, *ограничивает область применения* разрабатываемого класса.

Переменные такого типа *могут вызывать только методы класса Object*.

Доступ к другим методам ограничивает компилятор, предупреждая возможные варианты возникновения ошибок.

```
public class OptionalExt<T extends Тип> {
	private T value;
```

```
}
```

Чтобы расширить возможности параметризованных членов класса, можно ввести ограничения на используемые типы при помощи следующего объявления класса:

```
public class OptionalExt<T extends Тип> {
	private T value;
}
```

Такая запись говорит о том, что в качестве типа T разрешено применять только классы, являющиеся наследниками (производными) класса Тип, и соответственно появляется возможность вызова методов ограничивающих (bound) типов.

# Метасимвол ?

Если возникает необходимость в метод параметризованного класса одного допустимого типа передать объект этого же класса, но параметризованного другим типом, то при определении метода следует применить метасимвол “**?**”.

**<?>**

Метасимвол также может использоваться с ограничением **extends** для передаваемого типа.

**<? extends Number>**

```
package _java._se._02._generics;
public class Mark<T extends Number> {
	public T mark;

	public Mark(T value) {
		mark = value;
	}
	public T getMark() {
		return mark;
	}
	public int roundMark() {
		return Math.round(mark.floatValue());
	}
	/* вместо *// public boolean sameAny (Mark<T> ob) {
	public boolean sameAny(Mark<?> ob) {
		return roundMark() == ob.roundMark();
	}
	public boolean same(Mark<T> ob) {
		return getMark() == ob.getMark();
	}
}
```

```
package _java._se._02._generics;
public class MarkInspector {
	public static void main(String[] args) {
		// Mark<String> ms = new Mark<String>(“7”); //ошибка компиляции
		Mark<Double> md = new Mark<Double>(71.4D);// 71.5d
		System.out.println(md.sameAny(md));
		Mark<Integer> mi = new Mark<Integer>(71);
		System.out.println(md.sameAny(mi));
		// md.same (mi); //ошибка компиляции
```

```
                System.out.println(md.roundMark());
        }

}
```

Метод **sameAny(Mark<?> ob)** может принимать объекты типа **Mark**, инициализированные любым из допустимых для этого класса типов, в то время как метод с параметром **Mark<T>** мог бы принимать объекты с инициализацией того же типа, что и вызывающий метод объект.

## Использование extends с метасимволом ?

С помощью ссылки **List<? extends T>** невозможно добавлять элементы в коллекцию, так как невозможно гарантировать, что в список добавятся объекты допустимого типа.

Гарантируется только чтение объектов типа **T** или его подклассов

## Использование super с метасимволом ?

При чтении из коллекции с помощью ссылки типа **List<? super T>** нельзя гарантировать тип возвращаемого объекта иным, кроме как тип **Object**, так как ссылка, параметризированная данным образом может ссылаться на коллекции, параметризированные типом **T** и его базовыми типами.

Добавление элементов в коллекцию элементов возможно, элементы должны иметь тип **T** или тип его подклассов.

```
package _java._se._02._generics;
import java.util.ArrayList;
import java.util.List;

class MedicalStaff {
}
class Doctor extends MedicalStaff {
}
class HeadDoctor extends Doctor {
}
class Nurse extends MedicalStaff {
}

public class MetaGenericInspector {
        public static void main(String[] args) {
                List<? extends Doctor> list1 = new ArrayList<MedicalStaff>(); // error
                List<? extends Doctor> list2 = new ArrayList<Doctor>();
                List<? extends Doctor> list3 = new ArrayList<HeadDoctor>();

                List<? super Doctor> list7 = new ArrayList<HeadDoctor>(); // error
                List<? super Doctor> list6 = new ArrayList<Doctor>();
                List<? super Doctor> list5 = new ArrayList<MedicalStaff>();
                List<? super Doctor> list4 = new ArrayList<Object>();

                list5.add(new Object()); // error
                list5.add(new MedicalStaff()); // error
                list5.add(new Doctor());
                list5.add(new HeadDoctor());

                Object object = list5.get(0);
```

```
		MedicalStaff medicalDtaff = list5.get(0); // error
		Doctor doctor = list5.get(0); // error
		HeadDoctor headDoctor = list5.get(0); // error

	}
}
```

# Параметризованные методы

Параметризованный (generic) метод определяет базовый набор операций, которые будут применяться к разным типам данных, получаемых методом в качестве параметра.

```
<T extends Тип> Тип method(T arg) {}
     <T> Тип method(T arg) {}
```

Описание типа должно находиться перед возвращаемым типом. Запись первого вида означает, что в метод можно передавать объекты, типы которых являются подклассами класса, указанного после **extends**. Второй способ объявления метода никаких ограничений на передаваемый тип не ставит.

```
package _java._se._02._generics;

public class GenericMethod {

	public static <T extends Number> byte asByte(T num) {
		long n = num.longValue();
		if (n >= -128 && n <= 127) return (byte)n;
		else return 0;
	}
	public static void main(String [] args) {
		System.out.println(asByte(7));
		System.out.println(asByte(new Float("7.f")));
		// System.out.println(asByte(new Character('7')));
		// ошибка компиляции
	}
}
```

Параметризованные методы применяются когда необходимо разработать базовый набор операций, который будет работать с различными типами данных.

Описание типа всегда находится перед возвращаемым типом. Параметризованные методы могут размещаться как в обычных, так и в параметризованных классах.

Параметр метода может не иметь никакого отношения к параметру класса.

Метасимволы применимы и к generic-методам.

```
public static <T> T
	copy(List<? super T> dest, List<? extends T> src)
	{...}
```

Параметризованные методы можно перегружать.

```
package _java._se._02._generics;
import java.util.Date;
public class GenericMethodOverload {
	public static <Type> void method(Type obj) {
		System.out.println("<Type> void method(Type obj)");
	}
```

```
	public static void method(Number obj) {
		System.out.println("void method(Number obj)");
	}

	public static void method(Integer obj) {
		System.out.println("void method(Integer obj)");
	}

	public static void method(String obj) {
		System.out.println("void method(String obj)");
	}

	public static void main(String[] args) {
		Number number = new Integer(1);
		Integer integer = new Integer(2);
		method(number);
		method(integer);
		method(23);
		method("string");
		method(new Date());
	}
}
```

## Ограничения при использовании параметризации

Параметризованные поля *не могут быть статическими, статические методы не могут иметь параметризованные классом поля.*

```
package _java._se._02._generics;
public class GenericRestriction<T> {
	private static T x; // error
	private T y;
	public static <Type> void method(Type obj) {
		T var; // error
		Type typeVar;
	}
}
```

# Перечисления (enums)

## Синтаксис

**Examples:**

- dayOfWeek: SUNDAY, MONDAY, TUESDAY, …
- month: JAN, FEB, MAR, APR, …
- gender: MALE, FEMALE
- title: MR, MRS, MS, DR
- appletState: READY, RUNNING, BLOCKED, DEAD

```
public enum Season {
	WINTER,
	SPRING,
	SUMMER,
	FALL
}
```

В отличие от статических констант, предоставляют типизированный, безопасный способ задания фиксированных наборов значений

Являются классами специального вида, не могут иметь наследников, сами в свою очередь наследуются от **java.lang.Enum** и реализуют *java.lang.Comparable* (следовательно, могут быть сортированы) и *java.io.Serializable.*

Перечисления не могут быть абстрактными и содержать абстрактные методы (кроме случая, когда каждый объект перечисления реализовывает абстрактный метод), но могут реализовывать интерфейсы.

```
package _java._se._02._enum;
public class SeasonInspector {

	public static void main(String[] args) {
		Season season = Season.WINTER;
		System.out.println(season);
		// prints WINTER
		season = Season.valueOf("SPRING");
		// sets season to Season.SPRING
	}
}
```

## Создание объектов перечисления.

Экземпляры объектов **перечисления нельзя создать с помощью new**, каждый объект перечисления уникален, создается при загрузке перечисления в виртуальную машину, поэтому допустимо сравнение ссылок для объектов перечислений, **можно использовать оператор switch.**

Как и обычные классы могут реализовывать поведение, содержать вложенные классы.

Элементы перечисления по умолчанию **public**, **static** и **final.**

```
package _java._se._02._enum;
public enum Day {
	SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY;

	public boolean isWeekend() {
		switch (this) {
		case SUNDAY:
		case SATURDAY:
			return true;
		default:
			return false;
		}
	}

}
```

```
System.out.println( Day.MONDAY + " isWeekEnd(): " + Day.MONDAY.isWeekend() );
```

## Методы перечисления.

Каждый класс перечисления неявно содержит следующие методы:

- **static enumType[] values()** – возвращает массив, содержащий все элементы перечисления в порядке их объявления;

- **static T valueOf(Class<T> enumType, String arg)** – возвращает элемент перечисления, соответствующий передаваемому типу и значению передаваемой строки;
- **static enumType valueOf(String arg)** – возвращает элемент перечисления, соответствующий значению передаваемой строки;

(статические методы, выбрасывает **IllegalArgumentException** если нет элемента с указанным именем)

Каждый класс перечисления неявно содержит следующие методы:

- **int ordinal()** – возвращает позицию элемента перечисления.
- **String toString()**
- **boolean equals(Object other)**

```
package _java._se._02._enum;
public enum Shape {
	RECTANGLE("red"), TRIANGLE("green"), CIRCLE("blue");

	String color;

	Shape(String color){
		this.color = color;
	}

	public String getColor(){
		return color;
	}
}

package _java._se._02._enum;
public class ShapeInspector {

	public static void main(String[] args) {
		double x = 2, y = 3;
		Shape[] arr = Shape.values();
		for (Shape sh : arr)
			System.out.println(sh + " " + sh.getColor());
	}
}
```

## Анонимные классы перечисления

Отдельные элементы перечисления **могут реализовывать** свое **собственное поведение**.

```
package _java._se._02._enum;
public enum Direction {
	FORWARD(1.0) {
		public Direction opposite() {
			return BACKWARD;
		}
	},
	BACKWARD(2.0) {
		public Direction opposite() {
			return FORWARD;
		}
	};
	private double ratio;
```

```
	Direction(double r) {
		ratio = r;
	}

	public double getRatio() {
		return ratio;
	}

	public static Direction byRatio(double r) {
		if (r == 1.0)
			return FORWARD;
		else if (r == 2.0)
			return BACKWARD;
		else
			throw new IllegalArgumentException();
	}
}
```

## Сравнение переменных перечисления

На равенство переменные перечислимого типа можно сравнить с помощью операции == в операторе if, или с помощью оператора switch.

```
package _java._se._02._enum;
enum Faculty {
	MMF, FPMI, GEO
}

public class SwitchWithEnum {
	public static void main(String[] args) {
		Faculty current;
		current = Faculty.GEO;
		switch (current) {
		case GEO:
			System.out.print(current);
			break;
		case MMF:
			System.out.print(current);
			break;
		// case LAW : System.out.print(current);//ошибка компиляции!
		default:
			System.out.print("вне case: " + current);
		}
	}
}
```

# Классы внутри классов

В Java можно объявлять классы внутри других классов и даже внутри методов.

Они делятся на внутренние нестатические, вложенные статические и анонимные классы.

Такая возможность используется, если класс более нигде не используется, кроме как в том, в который он вложен.

Более того, использование внутренних классов позволяет создавать простые и понятные программы, управляющие событиями.

Методы внутреннего класса имеют прямой доступ ко всем полям и методам внешнего класса.

```
package _java._se._02._innerclass;
import java.util.Date;
public class Outer1 {
	private String str;
	Date date;

	Outer1() {
		str = "string in outer";
		date = new Date();
	}
	class Inner {
		public void method() {
			System.out.println(str);
			System.out.println(date.getTime());
		}
	}
}
```

Доступ к элементам внутреннего класса возможен только из внешнего класса через объект внутреннего класса.

```
package _java._se._02._innerclass;
import java.util.Date;

public class Outer2 {
	private Inner inner;
	private String str;
	private Date date;

	Outer2() {
		str = "string in outer";
		date = new Date();
		inner = new Inner();
	}

	class Inner {
		public void method() {
			System.out.println(str);
			System.out.println(date.getTime());
		}
	}

	public void callMethodInInner() {
		inner.method();
	}
}
```

Внутренние классы не могут содержать **static**-полей, кроме **final static**.

```
package _java._se._02._innerclass;
import java.util.Date;
public class Outer3 {
	private Inner inner;
	private String str;
	private Date date;

	class Inner {
		private int i;
		public static int static_pole; // ERROR
		public final static int pubfsi_pole = 22;
```

```
		private final static int prfsi_polr = 33;

		public void method() {
			System.out.println(str);
			System.out.println(date.getTime());
		}
	}

	public void callMethodInInner() {
		inner.method();
	}
}
```

Доступ к таким полям можно получить извне класса, используя конструкцию:

**имя_внешнего_класса.имя_внутреннего класса.имя_статической_переменной**

```
Outer outer = new Outer();
System.out.println(Outer.Inner.pubfsi_pole);
```

Доступ к переменной типа **final static** возможен во внешнем классе через имя внутреннего класса.

```
package _java._se._02._innerclass;

public class Outer5 {
	Inner inner;
	Outer5() {
		inner = new Inner();
	}
	class Inner {
		public final static int pubfsi_pole = 22;
		private final static int prfsi_polr = 33;
	}
	public void callMethodInInner() {
		System.out.println(Inner.prfsi_polr);
		System.out.println(Inner.pubfsi_pole);
	}
}
```

Внутренние классы могут быть производными от других классов.
Внутренние классы могут быть базовыми (в пределах внешнего класса).
Внутренние классы могут реализовывать интерфейсы.

Если необходимо создать объект внутреннего класса где-нибудь, кроме метода внешнего класса, то нужно определить тип объекта как:

**имя_внешнего_класса.имя_внутреннего_класса**

Объект в этом случае создается по правилу:

**ссылка_на_внешний_объект.new конструктор_внутренего_класса([параметры]);**

```
Outer.Inner1 obj1 = new Outer().new Inner1();
Outer.Inner2 obj2 = new Outer().new Inner2();
obj1.print();
obj2.print();
```

Внутренний класс может быть объявлен внутри метода или логического блока внешнего класса; видимость класса регулируется видимостью того блока, в котором он объявлен; однако класс сохраняет доступ ко всем полям и методам внешнего класса, а также константам, объявленным в текущем блоке кода.

```
package _java._se._02._innerclass;
public class Outer6 {
public void method() {
		final int x = 3;
		class Inner1 {
			void print() {
				System.out.println("Inner1");
				System.out.println("x=" + x);
			}
		}
		Inner1 obj = new Inner1();
		obj.print();
	}
	public static void main(String[] args) {
		Outer6 out = new Outer6();
		out.method();
	}
}
```

Локальные внутренние классы не объявляются с помощью модификаторов доступа.

```
package _java._se._02._innerclass;
public class Outer7 {
	public void method() {
		public class Inner1 {
		} // ОШИБКА
	}
}
```

Ссылка на внешний класс имеет вид:

**имя_внешнего_класса.this**

```
package _java._se._02._innerclass;
public class Outer8 {
	int count = 0;

	class InnerClass {
		int count = 10000;

		public void display() {
			System.out.println("Outer: " + Outer8.this.count);
			System.out.println("Inner: " + count);
		}
	}
}
```

# Статические (nested) классы

Статический вложенный класс для доступа к нестатическим членам и методам внешнего класса должен создавать объект внешнего класса.

```
package _java._se._02._innerclass;
public class Outer9 {
	private int x = 3;

	static class Inner1 {
		public void method() {
			Outer9 out = new Outer9();
			System.out.println("out.x=" + out.x);
		}
	}
}
```

Вложенный класс имеет доступ к статическим полям и методам внешнего класса.

```
package _java._se._02._innerclass;
public class Outer10 {
	private int x = 3;
	private static int y = 4;
	public static void main(String[] args) {
		Inner1 in = new Inner1();
		in.method();
	}
	public void meth() {
		Inner1 in = new Inner1();
		in.method();
	}
	static class Inner1 {
		public void method() {
			System.out.println("y=" + y);
			// System.out.println("x="+x); // ERROR
			Outer10 out = new Outer10();
			System.out.println("out.x=" + out.x);
		}
	}
}
```

Статический метод вложенного класса вызывается при указании полного относительного пути к нему.

```
package _java._se._02._innerclass;
public class Outer11 {
	public static void main(String[] args) {
		Inner1.method();
	}

	public void meth() {
		Inner1.method();
	}

	static class Inner1 {
		public static void method() {
			System.out.println("inner static method");
		}
	}
}
```

```
package _java._se._02._innerclass;
public class Outer12 {
	public static void main(String[] args) {
		Outer11.Inner1.method();
	}
}
```

Подкласс вложенного класса не наследует возможность доступа к членам внешнего класса, которым наделен его суперкласс.

```
package _java._se._02._innerclass;
public class Outer13 {
	private static int x = 10;

	public static void main(String[] args) {
		Inner1.method();
	}

	public void meth() {
		Inner1.method();
	}

	static class Inner1 {
		public static void method() {
			System.out.println("inner1 outer.x=" + x);
		}
	}
}

package _java._se._02._innerclass;
public class Outer14 extends Outer13.Inner1{
	public static void main(String[] args) {
	}

	public void outer2Method() {
		// System.out.println("x="+x); // ERROR
	}
}
```

Класс, вложенный в интерфейс, статический по умолчанию.

```
package _java._se._02._innerclass;
public interface InterfaceWithClass {
	int x = 10;

	class InnerInInterface {
		public void meth() {
			System.out.println("x=" + x);
		}
	}
}
```

Вложенный класс может быть базовым, производным, реализующим интерфейсы.

```
package _java._se._02._innerclass;
public class ExtendNested extends Outer15.Inner {
}

class Outer15 {
```

```
	static class Inner {
	}
	static class Inner2 extends Inner {
	}
	class Inner3 extends Inner {
	}
}
```

# Anonymous (анонимные классы)

Анонимный класс расширяет другой класс или реализует внешний интерфейс при объявлении одного единственного объекта; остальным будет соответствовать реализация, определенная в самом классе.

```
package _java._se._02._innerclass;
import java.util.Date;
public class AnonymEx {
	public void ex(){
		Date d=new Date(){
			@Override
			public String toString(){
				return "new version toString method";
			}
		};
		System.out.println(d);
	}
}
```

Объявление анонимного класса выполняется одновременно с созданием его объекта с помощью операции **new**.

Анонимные классы допускают вложенность друг в друга.

Конструкторы анонимных классов ни определить, ни переопределить нельзя.

```
package _java._se._02._innerclass;
public class SimpleClass {
	public void print() {
		System.out.println("This is Print() in SimpleClass");
	}
}
```

```
package _java._se._02._innerclass;
public class AnonymInspector {
	public void printSecond() {
		SimpleClass myCl = new SimpleClass() {
			public void print() {
				System.out.println("!!!!!!!");
				newMeth();
			}
			public void newMeth() {
				System.out.println("New method");
			}
		};
		SimpleClass myCl2 = new SimpleClass();
		myCl.print();// myCl.newMeth(); // Error
		myCl2.print();
	}
	public static void main(String[] args) {
		AnonymInspector myS = new AnonymInspector();
		myS.printSecond();
	}
}
```

Объявление анонимного класса в перечислении отличается от простого анонимного класса, поскольку инициализация всех элементов происходит при первом обращении к типу.

```
package _java._se._02._innerclass;
enum Color {
	Red(1),
	Green(2),
	Blue(3) {
		int getNumColor() {
			return 222;
		}
	};
	Color(int _num) {
		num_color = _num;
	}

	int getNumColor() {
		return num_color;
	}

	private int num_color;
}

public class EmunWithAnonym {
	public static void main(String[] args) {
		Color color;
		color = Color.Red;
		System.out.println(color.getNumColor());
		color = Color.Blue;
		System.out.println(color.getNumColor());
		color = Color.Green;
		System.out.println(color.getNumColor());
	}
}
```

# Аннотации (основы)

## Использование аннотаций

Аннотации используются для различных целей, среди них:

- **Information for the compiler** — аннотации могут использоваться компилятором, чтобы обнаружить ошибку или подавить предупреждение.
- **Compiler-time and deployment-time processing** — программные инструменты могут обрабатывать информацию из аннотаций, чтобы сгенерировать код, XML-файл или что-то другое.
- **Runtime processing** — некоторые аннотации доступны во время выполнения программы.

Ограничения, накладываемые на аннотации:

- объявляемый метод-аннотация не должен иметь параметров;
- объявление метода не должно содержать ключевое слово **throws**;
- метод должен возвращать одно из следующих типов: любой примитивный тип**, String, Class, enum** или массив указанных типов.

Аннотации могут использоваться со следующими элементами программы:

- класс, интерфейс или перечисления (enum);
- свойства (поля) классов;
- методы, конструкторы и параметры методов;
- локальная переменная;
- блок catch;
- пакет (java package);
- другая аннотация.

Аннотация применяется перед определением самого аннотируемого элемента, и может содержать именованные и неименованные значения.

```
@Author(
   name = "Benjamin Franklin",
   date = "3/27/2003"
)
class MyClass() { }

@SuppressWarnings(value = "unchecked")
void myMethod() { }


@SuppressWarnings("unchecked")
void myMethod() { }
```

Если аннотация не содержит элементов, круглые скобки могут быть опущены.

```
@Override
void mySuperMethod() { }
```

## Определение аннотаций

Определение аннотации выглядит следующим образом:

Определение аннотации подобно определению интерфейса. Тип аннотации фактически является интерфейсом.

Элементы аннотации объявляются подобно абстрактным методам и могут иметь значения по умолчанию.

## Применение аннотаций

```
package _java._se._02._annotation;
@ClassPreamble(
             author = "John Doe",
             date = "3/17/2002",
             currentRevision = 6,
             lastModified = "4/12/2004",
             lastModifiedBy = "Jane Doe",
             reviewers = {"Alice", "Bob", "Cindy"}
)
public class AnnotationClass {
}
```

Чтобы информация из аннотации была доступна для javadoc, необходимо саму аннотацию аннотировать как @Documented

```
import java.lang.annotation.*;
// import this to use @Documented
@Documented
@interface ClassPreamble {
   // Annotation element definitions
}
```

Аннотацией **@Target** указывается, какой элемент программы будет использоваться аннотацией. Объявление **@Target** в любых других местах программы будет воспринято компилятором как ошибка.

```
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import java.lang.annotation.ElementType;
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface Test {
	String sound();
	int color();
}
```

Возможные типы аннотации **@Target**:

- **PACKAGE** - назначением является целый пакет (package);
- **TYPE** - класс, интерфейс, enum или другая аннотация:
- **METHOD** - метод класса, но не конструктор (для конструкторов есть отдельный тип CONSTRUCTOR);
- **PARAMETER** - параметр метода;
- **CONSTRUCTOR** - конструктор;
- **FIELD** - поля-свойства класса;
- **LOCAL_VARIABLE** - локальная переменная (данный тип аннотации может использоваться только на уровне компиляции как, например, аннотация @SuppressWarnings);
- **ANNOTATION_TYPE** - другая аннотация.

## Аннотации, используемые компилятором

Есть три типа предопределенных аннотаций:

- @Deprecated,
- @Override, и
- @SuppressWarnings.

**@Deprecated**—аннотация @Deprecated означает, что помеченный элемент не рекомендуется к дальнейшему использованию. Компилятор генерирует предупреждение каждый раз, когда используется класс, метод или поле с аннотацией @Deprecated. Когда элемент определяется как не рекомендованный, необходимо также его документировать для тег Javadoc @deprecated.

```
// Javadoc comment follows
/**
 * @deprecated
 * explanation of why it was deprecated
 */
@Deprecated
static void deprecatedMethod() { }
}
```

**@Override** — аннотация @Override сообщает компилятору, что элемент переопределяет элемент, объявленный в базовом классе.

```
//mark method as a superclass method
// that has been overridden
@Override
int overriddenMethod() { }
```

При переопределении использование этой аннотации не является обязательным, однако, если метод аннотирован как @Override компилятор сгенерирует ошибку при некорректном переопределении.

**@SuppressWarnings**— аннотация @SuppressWarnings сообщает компилятору о необходимости подавить какое-либо предупреждение, которое иначе было бы сгенерировано.

```
//use a deprecated method and tell
// compiler not to generate a warning
@SuppressWarnings("deprecation")
void useDeprecatedMethod() {
     objectOne.deprecatedMethod(); //deprecation warning - suppressed
}
```

Каждое предупреждение компилятора принадлежит какой-либо категории. The Java Language Specification определяет две категории предупреждений: "deprecation" и "unchecked."

```
package _java._se._02._annotation;
public class SWAnnotationEx<T1> {
       private T1 var;

       @SuppressWarnings("unchecked")
       public <T2> SWAnnotationEx(T2 v) {
          var = (T1) v; // Type safety: Unchecked cast from T2 to T1
       }

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

Чтобы подавить оба предупреждения необходимо использовать следующий синтаксис:

```
@SuppressWarnings({"unchecked", "deprecation"})
```

# Annotation Processing

Аннотации времени выполнения могут быть обработаны java-программой , которая предпримет действия, основанные на этой аннотации (например сгенерировать какой-либо вспомогательный код, освобождая программиста от необходимости писать код по образцу).

JDK в версии 5.0. включает инструмент обработки аннотаций, названный apt.

В JDK версии 6.0 функциональность apt стала стандартной часть java-компилятора.

Чтобы сделать аннотацию доступной во время выполнения, необходимо ее предварить аннотацийТ @Retention(RetentionPolicy.RUNTIME):

```
import java.lang.annotation.*;
@Retention(RetentionPolicy.RUNTIME)
@interface AnnotationForRuntime {
   // Elements that give information
   // for runtime processing
}
```

Возможные типы аннотации:

- **SOURCE** - аннотация доступна только в исходном коде и сбрасывается во время создания .class файла;
- **CLASS** - аннотация хранится в .class файле, но недоступна во время выполнения программы;
- **RUNTIME** - аннотация хранится в .class файле и доступна во время выполнения программы.

Все эти аннотации являются элементами перечисления java.lang.annotation.RetentionPolicy.

**java.lang.annotation.RetentionPolicy:**

- **SOURCE** - этим типом стоит пользоваться если необходимо расширить исходный код, описанный аннотацией;
- **CLASS** – необходимо использовать, если хотите добавить какие-то характеристики к классам (например, создать список классов, которые используют аннотацию) до выполнения программы;
- **RUNTIME** - является наиболее используемым типам, так как видна во время выполнения кода и можно воспользоваться возможностями рефлексии.
- По умолчанию используется тип **CLASS**.

**@Inherited** - аннотация означает, что она автоматически наследуется в дочерних классах описанного аннотацией класса.

```
package _java._se._02._annotation;

import java.lang.annotation.Documented;
import java.lang.annotation.Inherited;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import java.lang.annotation.ElementType;

@Target(ElementType.TYPE)
@Documented
```

```
@Retention(RetentionPolicy.RUNTIME)
@Inherited
public @interface FeatureGiraffe {
	String sound();
	int color();
}

package _java._se._02._annotation;

@FeatureGiraffe(color = 0xFFA844, sound = "uuuu")
public class Giraffe {
}

package _java._se._02._annotation;

public class HomeGiraffe extends Giraffe{
}
```

Теперь класс HomeGiraffe содержит аннотацию @FeatureGiraffe класса Giraffe.

## Получение информации об аннотациях

- Для того чтобы получить информацию об аннотации SOURCE необходимо разобрать исходный код программы.
- При использовании аннотации @Retention с типом CLASS необходимо разобрать не исходный код программы, а скомпилированный .class файл.
- Аннотация с последним типом RUNTIME аннотации @Retention - чтобы получить информацию об аннотациях этого класса необходимо воспользоваться рефлексией.

# Введение в Design Patterns

## Шаблон

- это идея, метод решения, общий подход к целому классу задач, постоянно встречающихся на практике
- систематизация приемов программирования и принципов организации классов

Основные принципы объектно-ориентированного проектирования, применяемого при создании диаграммы классов и распределения обязанностей между ними, систематизированы в шаблонах **GRASP**

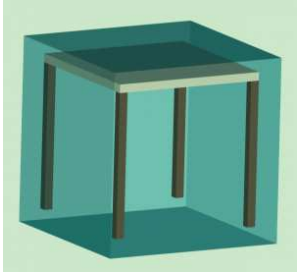
(**General Responsibility Assignment Software Patterns**)

## GRASP. CREATOR

Наиболее частой проблемой ОО-дизайна является проблема “Кто должен создавать объект X?” Объект A должен создавать объект B если:

- A содержит или агрегирует B
- A записывает B
- A широко использует B

- A содержит данные инициализации, которые должны быть переданы при создании объекту B



Пусть есть стол. Самый обычный стол состоит из столешницы и четырех ножек.

Используя объектную декомпозицию мы получим три класса:

- Table
- Desk
- Leg

Есть два пути запрограммировать такое решение. Сначала попробуем не использовать шаблон Creator.

```
package _java._se._01._pattern._creator._nocreator;

class Leg {
	private int width;
	private int length;
	private int height;

	public Leg(int w, int l, int h) {
		width = w;
		length = l;
		height = h;
	}
}

class Desk {
    private int width;
    private int length;
    private int height;

    public Desk(int w, int l, int h){
            width = w;
            length = l;
            height = h;
    }
}

public class Table {
	private Desk desk;// desk object
	private Leg[] legs;// array of legs
	public Table(Desk d, Leg[] l) {
		desk = d;
		legs = l;
	}
}


package _java._se._01._pattern._creator._nocreator;
public class TableInspector {

	public static void main(String[] args) {
		Desk desk = new Desk(900, 900, 20);
		Leg[] legs = { new Leg(40, 40, 880), new Leg(40, 40, 880),
				new Leg(40, 40, 880), new Leg(40, 40, 880) };
		Table table = new Table(desk, legs);
	}
```

```
}
```

Объект создается, однако значительная часть логики создания объекта остается за его пределами объекта. Программист, пользователь такого класса, должен знать внутреннюю структуру объекта, чтобы его создать.

Конструктор класса **Table** должен содержать следующие параметры

- Width – ширина стола
- Length – длина стола
- Height – высота стола
- DeskHeight – высота столешницы
- LegSection – число ножек

```
package _java._se._01._pattern._creator;
public class Table {
	private Desk desk;
	private Leg[] legs;

	public Table(int width, int length, int height, int deskHeight,
			int legSection) {
		desk = new Desk(width, length, deskHeight);
		for (int i = 0; i < 4; i++) {
			legs[i] = new Leg(legSection, legSection, height - deskHeight);
		}
	}
}
```

# GRASP. LOW COUPLING

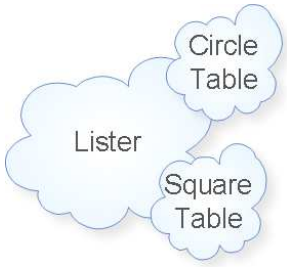
Пусть нам надо напечатать следующую таблицу параметров. В таблице существует два типа строк: square и circle. Square имеет параметр “side”, circle– “radius”.

```
+----------+-----------+---------+
|  Type    | parameter |  value  |
+----------+-----------+---------+
|  Circle  |  radius   |   5.00  |
+----------+-----------+---------+
|  Square  |  side     |   2.00  |
+----------+-----------+---------+
|  Square  |  side     |   8.00  |
+----------+-----------+---------+
```

```
package

_java._se._02._pattern._lowcoupling._nolowcoupling;
class Table {
}

class CircleTable extends Table {
	private double radius;

	public CircleTable(int r) {
		radius = r;
	}

	public double getRadius() {
		return radius;
	}
}

class SquareTable extends Table {
	private double side;

	public SquareTable(int s) {
		side = s;
```

```
	}

	public double getSide() {
		return side;
	}
}
public class TableNoLC {
	private Table[] tables;

	public void out() {
		System.out
				.println("+———-+———-+———+\n|  Type    | parameter |  value
|\n+———-+———-+———+\n");
		for (int i = 0; i < tables.length; i++) {
			if (tables[i] instanceof CircleTable) {
				System.out.printf("|  Circle  |  radius   |  %5.2f  |\n",
						((CircleTable) tables[i]).getRadius());
			} else {
				System.out.printf("|  Square  |  side     |  %5.2f  |\n",
						((SquareTable) tables[i]).getSide());
			}
			System.out.println("+———-+———-+———+\n");
		}
	}
}
```

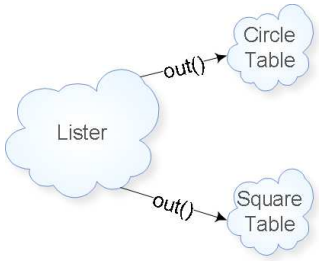
## Classes coupling




```
package _java._se._02._pattern._lowcoupling;

abstract class Table {
	public abstract void out();
}

class CircleTable extends Table {
	private int radius;

	public CircleTable(int r, int h) {
		radius = r;// setting the radius
	}

	public double getSquare() {
		return radius * radius * Math.PI;
	}

	public void out() {
		System.out.printf("|  Circle  |  radius   |  %5.2f  |\n", radius);
	}
}

class SquareTable extends Table {
	private int side;

	public SquareTable(int s, int h) {
		side = s;// setting the side
	}

	public double getSquare() {
		return side * side;
	}

	public void out() {
		System.out.printf("|  Square  |  side     |  %5.2f  |\n", side);
	}
```

```
}

public class TableWithLC {
	private Table[] tables;

	public void out() {
		System.out
				.println("+———-+———-+———+\n|  Type  | parameter |  value
|\n+———-+———-+———+\n");
		for (int i = 0; i < tables.length; i++) {
			tables[i].out();
			System.out.println("+———-+———-+———+\n");
		}
	}
}
```

Low coupling illustration




## Преимущества использования шаблонов:

- Нет необходимости решать каждую задачу с нуля.
- Использование проверенных решений.
- Можно заранее представить последствия выбора того или иного варианта.
- Проектирование с учетом будущих изменений.
- Компактный код, который легко можно будет использовать повторно.