Программные и аппаратные средства информатики

Лекция 3. Введение в программирование на Java

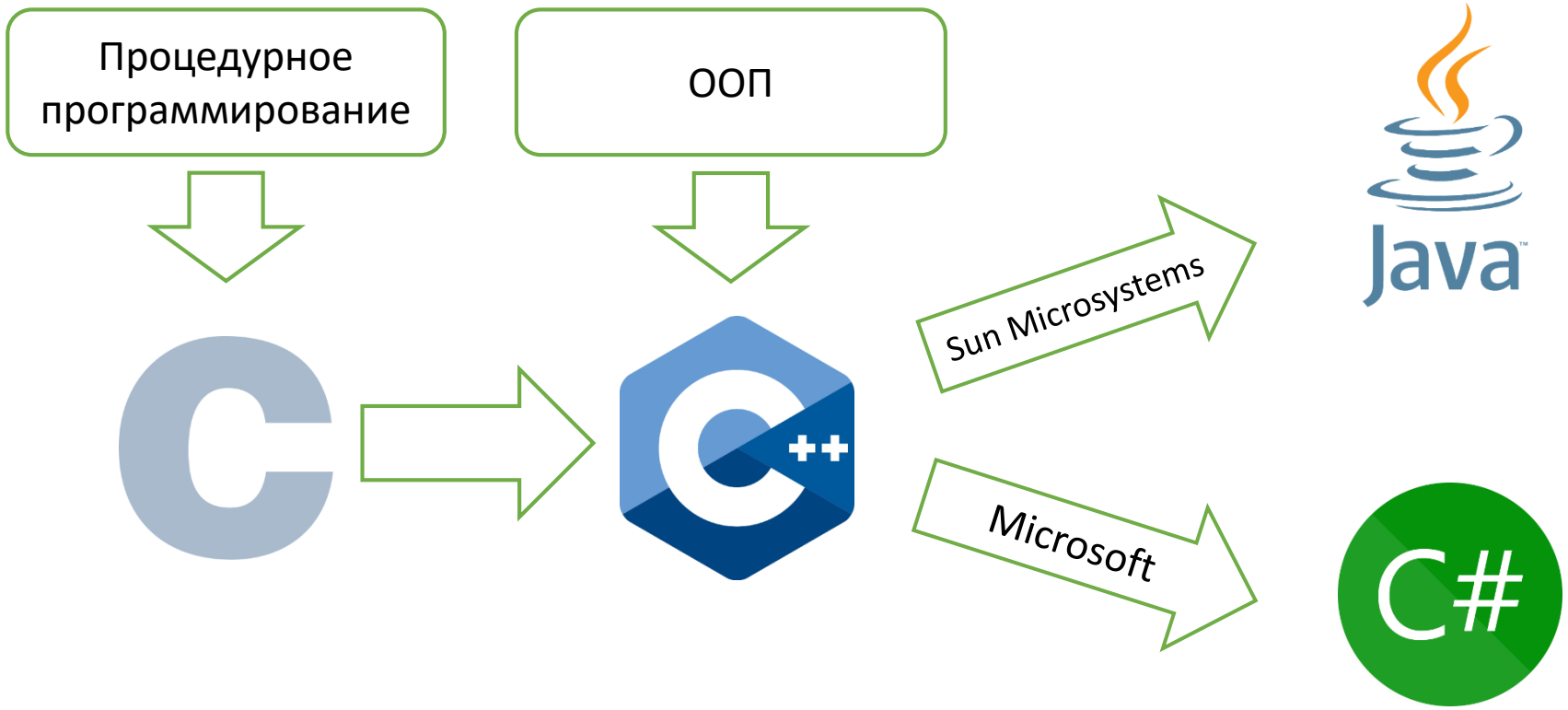
Содержание лекции:

- Применение языка и рабочая среда
- Классы и объекты
- Инкапсуляция
- Наследование
- Абстракция
- Полиморфизм
- Функциональный интерфейс
- Анонимный класс и лямбда-функция

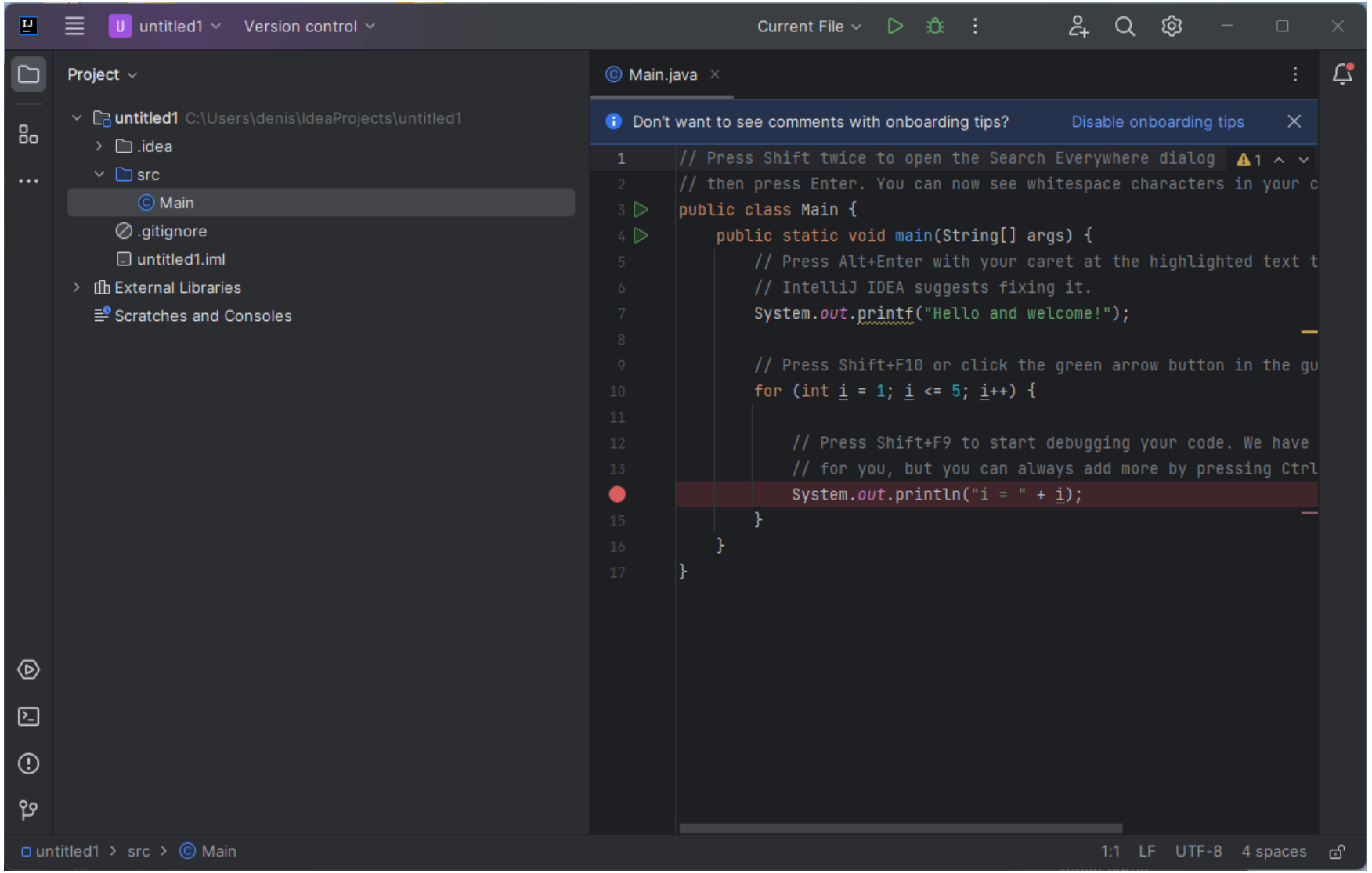
Преподаватель курса: Нефедов Денис Геннадьевич, к.т.н., доцент

Применение языка и рабочая среда

Java - строго типизированный объектно-ориентированный язык программирования, развиваемый Oracle. Приложения транслируются в байт-код, выполняемый Java Virtual Machine. Используется для создания десктопных приложений, веб-сервисов, приложений для мобильных платформ.



Для разработки на Java необходимо установить специальный комплект инструментов JDK или Java Development Kit.
Вариант IDE - IntelliJ IDEA



Классы и объекты

Объявление класса

```
/* модификаторы доступа */ class Example {
/* содержимое класса : поля и методы */
}
```

Поля

```
/* модификаторы доступа */ int number ;
/* модификаторы доступа */ String text = " hello ";
```

Методы

```
/* модификаторы доступа */ int getNumber () {
return number ;
}
```

Статические классы

- используются, когда необходим вложенный класс, который не требует доступа к экземпляру обрамляющего класса.
- могут быть более эффективны в памяти, поскольку они не содержат ссылки на экземпляры обрамляющего класса.

Статические поля, методы, константы и инициализаторы

- могут использоваться без создания объектов класса
- являются общими для всего класса, а не для отдельных объектов

```
public class OuterClass {
    static class StaticNestedClass {
        public void someMethod() { /* ... */ }
        public static void someStaticMethod() { /* ... */ }
    }
}
```

Константы

```
final double PI = 3.14;
```

Структуры

- отсутствуют

Конструкторы

```
class Example {
private int number ;
/* modifiers */ Example (int number ) {
this.number = number ;
}
}
```

Деструкторы

- отсутствуют
- при необходимости освободить ресурсы используют обычный метод void close() или void dispose()
- есть метод finalize (не рекомендуется)

```
protected void finalize() throws Throwable {
  super.finalize();   // Always call parent's finalizer
}
```

Свойства

```
private int mSize;
public int getSize() { return mSize; }
public void setSize(int value) {
  if (value < 0)
    mSize = 0;
  else
    mSize = value;
}
```

Работа со строками

```
// String concatenation
String school = "Harding ";
school = school + "University";   // school is "Harding University"
```

Общая структура программы

JAVA

HelloWorld.java

```java
package hello;

public class HelloWorld {
   public static void main(String[] args) {
      String name = "Java";

      // See if an argument was passed from
the command line
      if (args.length == 1)
         name = args[0];

      System.out.println("Hello, " + name +
"!");
    }
}
```

C#

HelloWorld.cs

```csharp
using System;

namespace Hello {
   public class HelloWorld {
      public static void Main(string[] args) {
         string name = "C#";

         // See if an argument was passed from
the command line
         if (args.Length == 1)
            name = args[0];

         Console.WriteLine("Hello, " + name +
"!");
      }
   }
}
```

JAVA

Импорт классов

```java
// Import single class
import harding.compsci.graphics.Rectangle;

// Import all classes
import harding.compsci.graphics.*;
```

Операции ввода/вывода в консоли

```java
java.io.DataInput in = new
java.io.DataInputStream(System.in);
System.out.print("What is your name? ");
String name = in.readLine();
System.out.print("How old are you? ");
int age = Integer.parseInt(in.readLine());
System.out.println(name + " is " + age + "
years old.");
int c = System.in.read(); // Read single char
```

C#

```c#
// Import single class
using Rectangle =
Harding.CompSci.Graphics.Rectangle;

// Import all class
using Harding.Compsci.Graphics;
```

```c#
Console.Write("What's your name? ");
string name = Console.ReadLine();
Console.Write("How old are you? ");
int age =
Convert.ToInt32(Console.ReadLine());
Console.WriteLine("{0} is {1} years old.",
name, age);
// or
Console.WriteLine(name + " is " + age + "
years old.");
int c = Console.Read();  // Read single
char
```

JAVA

Запись файлов

```java
import java.io.*;

// Character stream writing
FileWriter writer = new
FileWriter("c:\\myfile.txt");
writer.write("Out to file.\n");
writer.close();
```

Чтение файлов

```java
// Character stream reading
FileReader reader = new
FileReader("c:\\myfile.txt");
BufferedReader br = new BufferedReader(reader);
String line = br.readLine();
while (line != null) {
  System.out.println(line);
  line = br.readLine();
}
reader.close();
```

C#

```c#
using System.IO;

// Character stream writing
StreamWriter writer =
File.CreateText("c:\\myfile.txt");
writer.WriteLine("Out to file.");
writer.Close();
```

```c#
// Character stream reading
StreamReader reader =
File.OpenText("c:\\myfile.txt");
string line = reader.ReadLine();
while (line != null) {
  Console.WriteLine(line);
  line = reader.ReadLine();
}
reader.Close();
```

JAVA

Работа с массивами

```java
String names[] = new String[5];
names[0] = "David";

float twoD[][] = new
float[rows][cols];
twoD[2][0] = 4.5;
int[][] jagged = new int[5][];
jagged[0] = new int[5];
jagged[1] = new int[2];
jagged[2] = new int[3];
jagged[0][4] = 5;
```

C#

```csharp
string[] names = new string[5];
names[0] = "David";

float[,] twoD = new float[rows,
cols];
twoD[2,0] = 4.5f;
int[][] jagged = new int[3][] {
    new int[5], new int[2], new
int[3] };
jagged[0][4] = 5;
```

JAVA

C#

Работа со списками

```java
import java.util.ArrayList;

ArrayList<Object> list = new
ArrayList<Object>();
list.add(10);    // boxing
converts to instance of Integer
list.add("Bisons");
list.add(2.3);    // boxing
converts to instance of Double

for (Object o : list)
  System.out.println(o);
```

```csharp
using System.Collections;

ArrayList list = new
ArrayList();
list.Add(10);

list.Add("Bisons");
list.Add(2.3);


foreach (Object o in list)
  Console.WriteLine(o);
```

Инкапсуляция

Типы модификаторов доступа в Java:

- публичный (public) - доступ открыт из любого места в текущем пакете и из внешних пакетов;
- защищенный (protected) - доступ открыт внутри класса и классам-наследникам;
- приватный (private) - доступ открыт только внутри класса;
- по умолчанию или пакетный (default или package visible) – доступ во всех классах внутри текущего пакета

```
public class SomePhone {

    private int year;
    private String company;
    public SomePhone(int year, String company) {
        this.year = year;
        this.company = company;
    }
}
```

Наследование

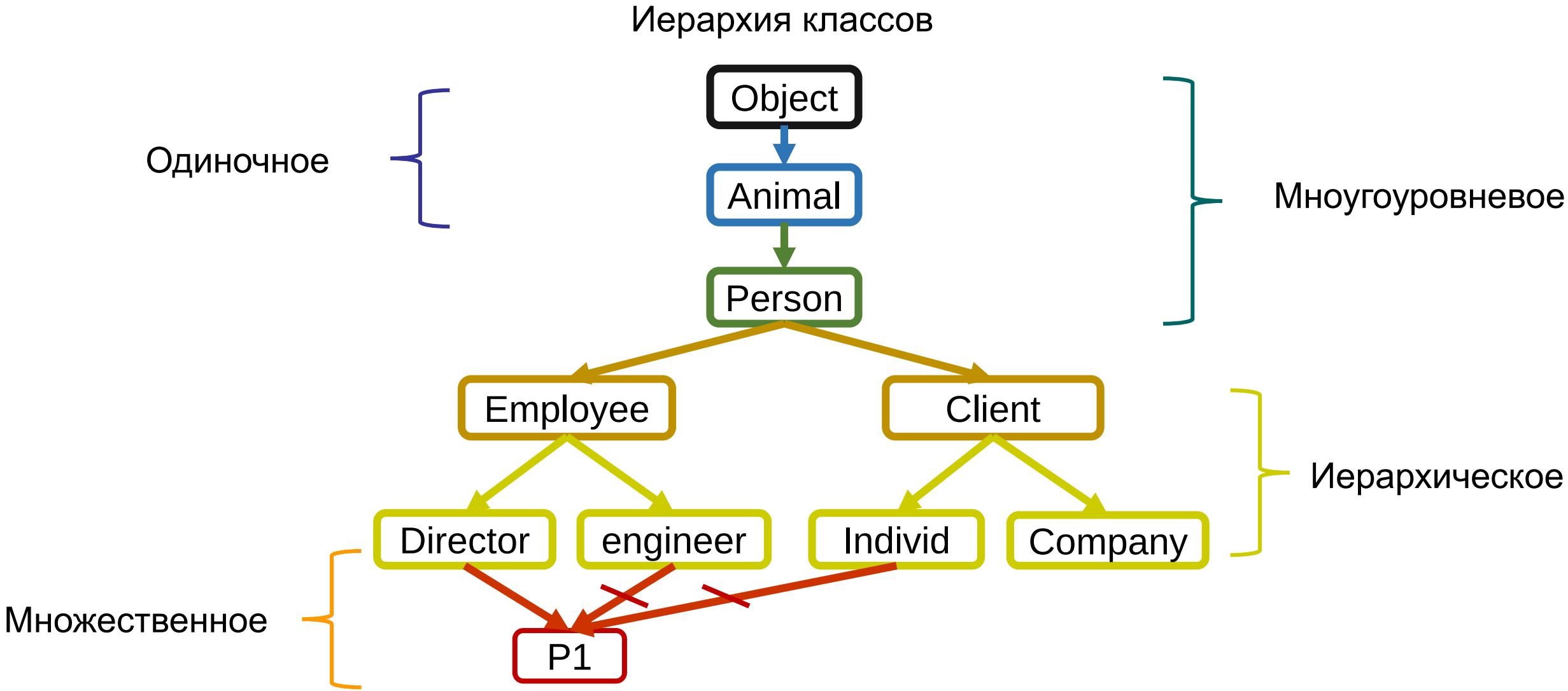
В Java родительский класс называется суперклассом , а класс-наследник называется подклассом.

Подклассы связаны с суперклассами с помощью ключевого слова extends при их определении.

Подклассы могут определять новые локальные методы или поля для использования или могут использовать ключевое слово super для вызова унаследованных методов или суперконструктора.

```java
class Parent {
    void display() {
        System.out.println("This is the parent class method.");
    }
}
class Child extends Parent {
    void display() {
        System.out.println("This is the child class method.");
    }
    void callParentMethod() {
        super.display(); // вызываем метод родительского класса с помощью super
    }
}
public class Main {
    public static void main(String[] args) {
        Child obj = new Child();
        obj.display(); // вызывается метод дочернего класса
        obj.callParentMethod(); // вызывается метод родительского класса через
метод дочернего класса
    }
}
```

Виды наследования



Все классы неявно наследуются от типа Object.

Восходящее преобразование – от подкласса к суперклассу – осуществляется автоматически

```
Object tom = new Person("Tom");
Object sam = new Employee("Sam", "Oracle");
Object kate = new Client("Kate", "DeutscheBank", 2000);
Person bob = new Client("Bob", "DeutscheBank", 3000);
Person alice = new Employee("Alice", "Google");
```

Нисходящее преобразование – требует преобразование типов

```
Object kate = new Client("Kate", "DeutscheBank", 2000);
((Person)kate).display();

Object sam = new Employee("Sam", "Oracle");
Employee emp = (Employee)sam;
emp.display();
```

Для определения класса объекта используется оператор instanceof, возвращающий true или false

```
if(Jhon instanceof Employee){
      …
}
```

Выражение

```
kate instanceof Client clientKate
```

проверяет объект на принадлежность к указанному классу, и если это так, то создает новый объект этого класса

Для запрета наследования класса или его метода используется ключевое слово final

```
final class Nameclass {}
```

Поддерживаемые типы наследования
Одиночное – для нескольких классов
Множественное – для 1 класса и нескольких интерфейсов.

Интерфейсы определяют некоторый функционал, не имеющий конкретной реализации, который затем реализуют классы, применяющие эти интерфейсы.

Особенности интерфейсов.

1. Для их создания используется ключевое слово interface.
2. Для их реализации используется ключевое слово implements.
3. Создать экземпляр интерфейса в Java нельзя.
4. Интерфейс представляет собой абстракцию, но в отличие от абстрактного класса, не имеет реализацию метода.
5. Интерфейс не имеет конструктора.
6. Интерфейс не может быть подклассом у класса, но может реализовать другой интерфейс с использованием ключевого слова extends.
7. Класс, реализующий интерфейс, должен обеспечить реализацию всех его методов (кроме абстрактного класса).
8. По умолчанию любой атрибут интерфейса является public, static и final.
9. По умолчанию методы интерфейса неявно abstract и public.

```
interface Account
{
  int CurrentSum = 10; // public static final
  void Put(); //
}
```

В JDK 8 была добавлена такая функциональность как методы по умолчанию.

```
interface Printable {
    default void print(){
        System.out.println("Undefined printable");
    }
}
```

Различие синтаксиса наследования классов и интерфейсов

1

```
class Super {
.....
}
class Sub extends Super {
.....
}
```

2

```
interface Printable {

    // методы интерфейса
}
interface Searchable {

    // методы интерфейса
}
class Book implements Printable, Searchable{

    // реализация класса
}
```

3

```
interface BookPrintable extends Printable{

    void paint();
}
```

При применении этого интерфейса класс Book должен будет реализовать как методы интерфейса BookPrintable, так и методы базового интерфейса Printable.

Абстракция

При определении абстрактных классов и методов используется ключевое слово abstract.

Производный класс обязан переопределить и реализовать все абстрактные методы, которые имеются в базовом абстрактном классе.

```
// абстрактный класс фигуры
abstract class Figure{
    float x; // x-координата точки
    float y; // y-координата точки
    Figure(float x, float y){
        this.x=x;
        this.y=y;
    }
    // абстрактный метод для получения площади
    public abstract float getArea();
}
```

```
// производный класс прямоугольника
class Rectangle extends Figure
{
    private float width;
    private float height;

    // конструктор с обращением к конструктору класса Figure
    Rectangle(float x, float y, float width, float height){
        super(x,y);
        this.width = width;
        this.height = height;
    }

    public float getArea(){
        return width * height;
    }
}
```

Полиморфизм

Два варианта реализации полиморфизма в Java:

- Overloading – перегрузка метода
- Overriding – переопределение метода

При перегрузке метода в одном классе создаются несколько методов с одинаковым названием, но разным функционалом. Перегруженный метод определяет статический полиморфизм – вызываемый при компиляции метод определяется его сигнатурой – типом возвращаемых данных, свойств и т.д.

При переопределении название метода дочернего класса мы ставим такое же, как и уже объявленного метода родительского класса. Переопределенный метод определяет динамический полиморфизм – выбранный метод определяется уже при выполнении программы по объекту, на который ссылается переменная родительского класса.

Особенности переопределения метода

1. Перед ним ставится (но не обязательно) аннотация @Override.
2. Сигнатура (название и аргументы) должны быть такими же, как у метода родителя
3. Тип возвращаемого значения должен быть таким же, как у метода родителя
4. Завершенные, т.е. final, методы не переопределяются.

Перегрузка

```
class Forms{
  public void shapearea()) {
    System.out.println("Площади фигур:");
  }
public void shapearea(int r) {
    System.out.println("Sкруга = "+3.14*r*r);
  }
public void shapearea(int l, int b) {
    System.out.println("Sпрямоугольника ="+l*b);
  }
}

class Main {
  public static void main(String[] args) {
   Forms xForms = new Forms();
    xForms.shapearea(); // статический полиморфизм
    xForms.shapearea(3);  //он же
    xForms.shapearea(4,7);  //он же
  }
}
```

Переопределение

```
class Beast {
  void eat() { System.out.println("Животные питаются:");
    }
  }
  class herbivorous extends Beast {
    void eat() { System.out.println("Травоядные едят растения");
    }
  }
  class carnivorous extends Beast {
    void eat() { System.out.println("Хищники едят только мясо");
    }
  }
  class main {
    public static void main(String args[]) {
      Beast X = new Beast();
      Beast herb = new herbivorous();
      Beast carn = new carnivorous();
      X.eat(); // динамический полиморфизм
      herb.eat(); // он же
      carn.eat(); // он же
    }
  }
```

Функциональный интерфейс

Делегаты в языке Java отсутствуют.

Их функционал реализуется механизмом обратного вызова с использованием интерфейсов.

Суть обратного вызова состоит в том, что мы создаем действия, которые вызываются при других действиях.

Стандартный пример - нажатие на кнопку. Когда мы нажимаем на кнопку, мы производим действие, но в ответ на это нажатие запускаются другие действия.

```
class ButtonClickHandler implements EventHandler{
    public void execute(){
        System.out.println("Кнопка нажата!");
    }
}

interface EventHandler{
    void execute();
}
```

Реализуем метод click (имитация нажатия кнопки), вызывающий метод execute объекта интерфейса EventHandler.

```
public class EventsApp {
    public static void main(String[] args) {
        Button button = new Button(new ButtonClickHandler());
        button.click();
    }
}

class Button{
    EventHandler handler;
    Button(EventHandler action){
        this.handler=action;
    }
    public void click(){
        handler.execute();
    }
}
```

Функциональный интерфейс – интерфейс, имеющий лишь один абстрактный метод, и любое количество статических, приватных и default- методов.

Такие интерфейсы помечаются аннотацией @FunctionalInterface (не обязательно).

Основное назначение – использование в лямбда выражениях и ссылках на методы.

```
public interface MyFunctionalInterface2{

    public void execute();

    public default void print(String text) {
        System.out.println(text);
    }

    public static void print(String text, PrintWriter writer) throws
IOException {
        writer.write(text);
    }
}
```

Некоторые встроенные функциональные интерфейсы.

1. Функциональный интерфейс Function<T, R> содержит метод apply(). Представляет функцию перехода от объекта типа T к объекту типа R.

```
@FunctionalInterface
public interface Function<T, R> {
    R apply(T t);
}
```

Пример использования

```
import java.util.function.Function;

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

Function<Integer, String> convert = x-> String.valueOf(x) + " долларов";
System.out.println(convert.apply(5)); // 5 долларов

}
}
```

2. Предикат Predicate<T>. Проверяет соблюдение некоторого условия. Если оно соблюдается, то возвращается значение true.

```
import java.util.function.Predicate;

Predicate<Integer> isPositive = x -> x > 0;
System.out.println(isPositive.test(5)); // true
System.out.println(isPositive.test(-7)); // false
```

3. Потребитель Consumer<T>. Выполняет некоторое действие над объектом типа T, при этом ничего не возвращая

```
import java.util.function.Consumer;

Consumer<Integer> printer = x-> System.out.printf("%d долларов \n",
x);
printer.accept(600); // 600 долларов
```

Анонимный класс и лямбда-функция

Анонимный класс – это локальный класс без имени. Объявление такого класса выполняется одновременно с созданием его объекта посредством оператора new.

```
public class Potato {
    public void peel() {
        System.out.println("Чистим картошку.");
    }
}
public class Food {
    public static void main(String[] args) {
        Potato potato = new Potato() {
            @Override
            public void peel() {
                System.out.println("Чистим картошку в анонимном классе.");
            }
        };
        potato.peel();
    }
}
```

Анонимный класс также может расширить интерфейс.

```
public interface Moveable {
    void moveRight();
    void moveLeft();
}

public class MoveableDemo {
    public static void main(String[] args) {
        Moveable moveable = new Moveable() {
            @Override
            public void moveRight() {
                System.out.println("MOVING RIGHT!!!");
            }

            @Override
            public void moveLeft() {
                System.out.println("MOVING LEFT!!!");
            }
        };
        moveable.moveRight();
        moveable.moveLeft();
    }
}
```

Лямбда-выражение осуществляет реализацию метода, определенного в функциональном интерфейсе.

```
public class LambdaApp {

    public static void main(String[] args) {

        Operationable operation;
        operation = (x,y)->x+y;

        int result = operation.calculate(10, 20);
        System.out.println(result); //30
    }
}
interface Operationable{
    int calculate(int x, int y);
}
```

Резюме по языку Java.

1. Является компилируемым языком со статической типизацией.
2. Имеет ссылочный синтаксис: объекты создаются таким образом, чтобы невозможно было напрямую (через адреса ячеек памяти, т.н. указатели) повлиять на состояние этих объектов.
3. Использует автоматическую сборку мусора: программист избавлен от необходимости вручную очищать память от переменных, которые больше не понадобятся.
4. Обладает кроссплатформенностью – способностью запускаться на различных платформах.