12. Обектно ориентирано програмиране на базата на Java: Основни принципи. Класове и обекти. Конструктори и деструктори. Оператори. Производни класове и наследяване.

Анотация: Изложението по въпроса трябва да включва следните по-съществени елементи:

1. Основни принципи на обектно ориентираното програмиране;
2. Класове. Дефиниране на класове. Област на класове. Обекти;
3. Конструктори. Дефиниране на конструктори. Видове конструктори;
4. Указатели към обекти. Масиви и обекти. Динамични обекти;
5. Деструктори. Създаване и разрушаване на обекти на класове;
6. Оператори. Предефиниране на оператори;
7. Производни класове. Дефиниране. Наследяване и достъп до наследените компоненти.

Обектно-ориентираното програмиране (ООП) е породено от множество причини но една от основните е необходимостта за по-добър контрол над конкурентната модификация на споделени данни. Основната идея е – да няма директен достъп до данните, а те да се достъпват през предназначен за целта слой код. Тъй като данните трябва да се предават и модифицират, се ражда концепцията за обект.

Обект в най-общ смисъл е множество данни, които могат да се предават и достъпват само чрез множество методи, които се подават заедно с данните. Данните съставляват състоянието на обекта, а методите съставляват поведението на обекта. Състоянието на обекта е скрито (енкапсулирано) от директен достъп. Обектът е инстанция (представител) на даден клас. Обекти се създават явно чрез оператора new или неявно, и живеят в heap паметта.

Създаване на обекти:

```
String message = new String("Operation completed."); // explicit
Integer studentsCount = Integer.valueOf(60); // implicit
int[ ] intArr = new int[100]; // explicit
String[ ] stringArr = {"Some", "example"}; // implicit
stringArr = new String[ ]{"Changed", "my", "mind"}; // explicit
```

От лявата страна на равенството на третия ред, например, имаме просто една декларация. Тя декларира една референция, която се казва intArr и е от тип масив от цели числа. На този етап не се заделя никаква памет освен за референцията. А от дясно стои инициализацията, където се случва алокирането на паметта – в случая за масив от 100 цели числа.

На последните два реда имаме имплицитено и експлицитно инициализиране на масив от стрингове. Експлицитното инициализиране върши работа само в комбинация с декларация.

Клас

- Клас – дефиниция на (клас от) обекти
  - описва състояние чрез член-данни (член-променливи)
  - описва поведение чрез методи
- Конструктор(и)
- Метод(и)
- Възможно е даден клас да няма състояние или да няма поведение
- Всеки клас има определен интерфейс – формално описание на начина, по който други обекти могат да взаимодействат с него

Ако нямаме имплицитно дефиниран конструктор за даден клас се създава неявно такъв по подразбиране. Това е конструктор без аргументи, които ще служи за инстанциране на дадения клас. Може да имаме един или повече явни конструктори с различен на брой аргументи.

Метод на клас

- Функция за манипулиране на състоянието на класа
  - име и списък от параметри (сигнатура)

        - броят на параметрите се нарича арност
        - два метода имат еднаква сигнатура, ако имат еднакви имена, еднаква арност и еднаква последователност от типове в списъка от параметри
    - модификатори, тип на връщаната стойност, сигнатура и тяло, оградено с {}
    - тялото може да съдържа декларации на локални променливи и statements
    - може да има странични ефекти

Всичко друго освен връщането на резултат на една функция се счита за страничен ефект. Това може да е модифициране на някакви входни данни или член данни на класа, четене от или писане на конзола, файл, база от данни и други.

Пример за метод на клас:

```
public class AdmissionTestProblems {

    public static int minSwaps(String s1, String s2) {

        int differences = 0;
        for (int i = 0; i < s1.length(); ++i) {
            differences += (s1.charAt(i) == s2.charAt(i)) ? 0 : 1;
        }

        return differences / 2;
    }
}
```

Методи с променлив брой аргументи

- специален вид параметър на метод: редица от нула или повече аргументи от един и същ тип
- Нарича се varargs, от variable arguments
- Синтаксис: име на тип, следван от три точки (многоточие)
- Ако метод има varargs параметър, той трябва да е един и да е последен в списъка
- При достигане до varargs параметър, компилаторът създава масив от останалите аргументи и ги подава на метода като масив
- В тялото на метода се достъпва като масив

```
// A method that takes variable number of integer arguments
static void funWithVarargs(int... a) {
    System.out.println("Number of arguments: " + a.length);

    // using for-each loop to display contents of a
    for (int i : a) {
        System.out.print(i + " ");
    }
}

public static void main(String[ ] args) {
    // standard syntax
}

public static void main(String... args) {
    // varargs syntax - equivalent to the above
}
```

Статични член-променливи и статични методи

- Те са част от класа, а не от конкретна негова инстанция (обект)
- Могат да се достъпват, без да е създаден обект: само с името на класа, точка, името на статичната член-променлива или метод

```
System.out.println(Math.PI);          // 3.141592653589793
System.out.println(Math.pow(3, 2));  // 9.0
```

Статични член-променливи

- Статичните член-променливи имат едно-единствено копие, което се споделя от всички инстанции на класа
  - ако са константи, пестим памет (няма смисъл да се мултиплицират във всяка инстанция)
  - ако са променливи, всяка инстанция „вижда“ и променя една и съща стойност, което е механизъм за комуникация между всички инстанции на дадения клас
- Статичните методи имат достъп само до статичните член-променливи и други статични методи на класа
- Нестатичните методи имат достъп както до статичните, така и до нестатичните членове на класа

```
public class Utils {
   public static final double PI = 3.1415; // constant
   private static int radius = 10; // static member
   private String fact5 = "5!"; // non-static member

   // static method
   public static long fact(int n) {
      if (n == 1) { return 1; } else { return n * fact(n - 1); }
   }

   // non-static method
   public String getFact( ) {
      return fact5;
   }

   public static void main(String[ ] args) {
      System.out.println("Perimeter is " + 2 * Utils.PI * radius);
      System.out.println(new Utils( ).getFact( ) + "=" + Utils.fact(5));
      Utils.getFact( ); // won't compile
   }
}
```

Ключовата дума this

- Референция към конкретния обект
- Неявно се подава като параметър на всеки конструктор и нестатичен метод на класа
- Употребява се за:
  - достъпване на член-променливи, „скрити“ от едноименни параметри на метод или локални променливи
  - извикване от конструктор на друг overloaded конструктор в същия клас
  - извикване на произволен метод на класа.

```
public class Human {

   private String name;

   public Human( ) {
      // Извикване на overload-натия конструктор със String параметър
      this("Unknown");
   }

   public Human(String name) {
      // Достъпване на член-променлива, скрита от едноименен параметър
      this.name = name;
   }

   public void whoAmI( ) {
      System.out.println("My name is " + name);
   }
}
```

Обект – конкретна инстанция на даден клас

```
public class MainApp {

    public static void main(String[ ] args) {
        Human pesho = new Human("Pesho");
        pesho.whoAml( );
        Human gosho = new Human("Gosho");
        gosho whoAml( );
    }
}
```

Пакети

- Именувани групи от семантично свързани класове
- Служат за йерархично организиране на кода
- Съответстват на директорно дърво на файловата система
- Конвенция за именуване:
  - само малки букви, точка за разделител
  - компаниите използват обърнат домейн адрес
    - mail.google.com → com.google.mail
- всеки Java клас се намира в някакъв пакет
- пакетът се указва в началото на сорс файла (преди дефиницията на класа) с ключовата дума package
- ако липсва package декларация, класът е в пакета по подразбиране (който няма име) – което е лоша практика

```
package bg.sofia.uni.fmi.mjt.example;

public class Example {
    // class Example is in package bg.sofia.uni.fmi.mjt.example
}
```

Достъп до клас от друг пакет

- Всеки клас има достъп по подразбиране (имплицитно) до:
  - класовете от собствения си пакет
  - класовете в пакета java.lang
- Ако искаме клас да има достъп до клас в някой друг пакет, трябва експлицитно да го заявим с import декларация, която поставяме над декларацията на класа
- import декларацията може да е за конкретен клас с пълното му име: име на пакет + . + име на клас, или на всички класове в даден пакет (т.нар. wildcard import: име на пакет + .*)

```
package bg.sofia.uni.fmi.mjt.example;

import java.util.Arrays;
import java.util.Scanner;

public class StringUtils {

    public static void main(String[ ] args) {
        Scanner scanner = new Scanner(System.in);
        char[ ] tokens = scanner.nextLine( ).toCharArray( );
        Arrays.sort(tokens);
        System.out.println(tokens);
    }
}
```

- Прието е import-ите да се подреждат в сортиран лексикографски ред по <package>.<class>
- По-чисто е да се изброят конкретните класове от пакета, вместо import java.util.*

Модификатори за достъп
за top-level клас, т.е. невложен в друг

Модификатор	Видимост
public	Достъпен за всеки клас във всеки пакет
без модификатор*	Достъпен само за класовете в собствения си пакет

// казваме също "package-private", "default"

Модификатори за достъп
За член-променливи и методи на клас

Модификатор	Клас	Пакет	Подклас	Всички други
public	да	да	да	да
protected	да	да	да	не
no modifier	да	да	не	не
private	да	не	не	не

Енкапсулация
Енкапсулацията адресира основния проблем, мотивирал създаването на ООП: по-добро управление на конкурентния достъп до споделени данни.

- Множеството текущи стойности на член-данните на даден обект се наричат негово състояние
- Само вътрешните методи на даден обект имат достъп до неговото състояние, като правят невъзможни неочакваните промени на състоянието от външния свят
- В Java се постига чрез модификаторите за достъп

Енкапсулация – пример за нарушаване

```
public class Human {

    public String name; // no encapsulation!

    public Human(String name) {
        this.name = name;
    }
}

public class Main {

    public static void main(String[ ] args) {
        Human ivan = new Human("Ivan");
        ivan.name = "Faked Ivan"; // Hm...
    }
}
```

Енкапсулация – пример за спазване

```
public class Human {

    private String name; // stays hidden

    public Human(String name) {
        this.name = name;
    }
}

public class Main {

    public static void main(String[ ] args) {
        Human ivan = new Human("Ivan");
        ivan.name = "Faked Ivan"; // Won't compile...
    }
}
```

Наследяване

- Позволява преизползване и разширяване на състояние и поведение на вече съществуващи класове
- Когато клас наследява друг клас, за него казваме, че е наследник, или дете, или подклас, а втория клас наричаме родител, или базов клас, или суперклас
- В Java се реализира с ключовата дума extends
- Класът-наследник получава достъп до всички public и protected член-променливи и методи на родителския клас
- Java не поддържа множествено наследяване

Наследяване и method overriding

- Класът-наследник може да предостави собствена дефиниция на (т.е. да предефинира) методи на родителския клас (method overriding)
- Сигнатурата на метода в класа-родител и класа-наследник трябва да е идентична;
- Модификаторът за достъп на метод в класа–наследник трябва да съвпада или да разширява модификатора за достъп на метода в родителския клас (но не може да го свива/ограничава)
- Типът на връщаната стойност трябва да е съвместим с този на override-вания метод. Съвместим означава идентичен или наследник (за референтните типове) - return type covariance
- Методът в класа-наследник е препоръчително да се анотира с опционалната анотация @Override. Така компилаторът ще ни помага да не нарушаваме горните правила

Ключовата дума super

- Референция към прекия родител на обекта
- Употребява се за:
  - достъпване на член-променливи на родителя
  - извикване от конструктор в текущия клас на конструктор в родителския клас
  - извикване на произволен метод на родителския клас
- Неявно се подава като параметър на всеки конструктор и нестатичен метод на класа
- Не нарушава енкапсулацията - през super може да достъпим само public и protected членове на родителския клас

```
public class Student extends Human {

    private int facultyNumber;

    public Student(String name, int facultyNumber) {
        super(name); // извикване на родителския конструктор
        this.facultyNumber = facultyNumber;
    }

    public static void main(String[ ] args) {
        Student ivan = new Student("Ivan", 61786);
        ivan.whoAmI( ); // наследен от родителя метод
    }
}
```

Йерархията от класове в Java

- Всички класове в Java са (преки или косвени) наследници на класа java.lang.Object
- Липсата на множествено наследяване означава, че всеки клас има точно един родител (с изключение на един-единствен клас, java.lang.Object, който няма родител)
- Следователно, йерархията от класове е дърво, с java.lang.Object в корена

java.lang.Object

- boolean equals(Object obj)
- int hashCode( )
- String toString( )
- Object clone( )

equals( )

- Трябва да го предефинираме, ако сравняваме два обекта за семантична (т.е. смислова) еднаквост, а не по референциите им (т.е. адреса им в паметта)
- Например, две инстанции на клас Student смислово са еднакви (отговарят на един и същи студент), ако факултетните им номера са еднакви – без значение дали референциите са еднакви или не

hashCode( )

- Трябва да го предефинираме, ако сме предефинирали equals( )
- При предефинирането на hashCode( ), ако equals( ) връща true, hashCode-ът на съответните обекти трябва да е равен
- Ако hashCode-ът на два обекта е равен, не е задължително equals( ) да връща true

Операторът instanceof

- Използва се за type checking на референтните типове – дали даден обект е инстанция на даден клас

```
Student ivan = new Student("Ivan", 61786);
Human petar = new Human("Petar");

System.out.println(ivan instanceof Student);    // true
System.out.println(ivan instanceof Human);      // true
System.out.println(petar instanceof Student);   // false
System.out.println(petar instanceof Human);     // true

// arrays are reference types
int[ ] intArr = new int[2];
System.out.println(intArr instanceof int[ ]);   // true

// null is not an instance of anything: false for any class
System.out.println(null instanceof AnyClass);

// true for any non-null ref, because any class extends java.lang.Object
System.out.println(ref instanceof Object);
```

Pattern matching for instanceof (since Java 16)

```
// until now
if (obj instanceof String) {
    int stringLength = ((String) obj).length( );
}

// since Java 16 (preview feature in Java 14 & 15)
if (obj instanceof String s) {
    int stringLength = s.length( );
}
```

Ключовата дума final

- в декларация на променлива → прави я константа
- в декларация на метод → методът не може да се override-ва
- в декларация на клас → класът не може да се наследява

Полиморфизъм

- От гръцки: poly (много) + morphe (форма)
- Дефиниция от биологията – съществуване на морфологично различни индивиди в границите на един вид
- В контекста на ООП, полиморфизъм е способността на даден обект да се държи като инстанция на друг клас или като имплементация на друг интерфейс
- ООП – наследниците на даден клас споделят поведение от родителския клас, но могат да дефинират и собствено поведение
- Всички Java обекти са полиморфични, понеже всеки обект наследява java.lang.Object класа

Method overriding vs method overloading

- Overriding – класът-наследник предефинира поведението на класа-родител
- Overloading – класът декларира методи с едно и също име и различен брой и/или тип параметри

Runtime полиморфизъм чрез method overriding

```
public class Human {
  private String name;

    public Human(String name) {
        this.name = name;
    }

    public void whoAmI( ) {
        System.out.println("My name is " + name);
    }
}

public class Student extends Human {

    private int facultyNumber;

    public Student(String name, int facultyNumber) {
        super(name);
        this.facultyNumber = facultyNumber;
    }

    @Override
    public void whoAmI( ) {
        super.whoAmI( );
        System.out.println("My faculty number is " + this.facultyNumber);
    }
}
```

Compile-time полиморфизъм чрез method overloading

```
public class Human {

    public void move() {
        System.out.println("I am walking using two legs.");
    }

    public void move(String vehicle) {
        System.out.println("I move using a " + vehicle);
    }
}

public class Main {

    public static void main(String[ ] args) {
        Human ivan = new Human( );
        ivan.move( );
        ivan.move("Car");
    }
}
```

Method overriding vs method overloading

	Overloading	Overriding
Кога	Compile-time	Runtime
Къде	В същия клас	В класовете - наследници
Списък от аргументи	Различен	Идентичен
Return type	Може да бъде различен	Съвместим
static, private и final методи	Да	Не
Свързване (binding)	Статично	Динамично
Runtime performance	Better	

Non-polymorphic code

```
Student ivan = new Student("Ivan", 61786);
Human petar = new Student("Petar", 74451);
Object[ ] objs = {ivan, petar};

for (Object obj : objs) { // instanceof and explicit casts are the "red lights"
*   if (obj instanceof Student) {
*       ((Student) obj).whoAmI( );
*   } else if (obj instanceof Human) {
*       ((Human) obj).whoAmI( );
*   }
}
```

Polymorphic code

```
Human[ ] humans = {ivan, petar};

for (Human human : humans) {
    human.whoAmI( );
}
```

Полиморфният код е не само по-кратък и четим. Ако в бъдеще се появят нови класове – наследници на Human, в полиморфния код няма да има промяна.

Абстрактни класове

- Дефинират се с модификатора abstract
- Могат да имат методи без имплементация, които се декларират с модификатора abstract
- Не са напълно дефинирани (оставят на наследниците си да ги конкретизират/ допълнят)
  - не могат да се създават обекти от тях
- Един клас не може да бъде едновременно abstract и final, тъй като ако е abstract, ще означава че няма да могат да се създават обекти от тях, а ако е и final ще означава, че няма да може да се наследяват, което автоматично ги прави неизползваеми

Абстрактни класове – пример

```
public abstract class Cat {
    public void move( ) {
        System.out.println("I am walking on 4 toes.");
    }

    public void communicate( ) {
        System.out.println("I mew.");
    }

    public abstract void eat( );
}

public class DomesticCat extends Cat {
    public void eat( ) {
        System.out.println("I eat Whiskas.");
    }
}

public class Leopard extends Cat {
    public void eat( ) {
        System.out.println("I eat any prey.");
    }
}
```

Един клас може да е с дефинирани всичките си методи, но да е абстрактен.

Интерфейси

- Съвкупност от декларации на методи без имплементация;
- Описват формално поведение, без да го имплементират;
- Може да съдържат static final член-променливи == константи.

```
public interface Animal {
    void move( );
    void communicate( );
}
public class Human implements Animal {
    private String name;
    public Human(String name) {
        this.name = name;
    }
    public void move( ) {
        System.out.println("I am walking using two legs");
    }
    public void communicate( ) {
        System.out.println("I speak");
    }
}
public class Cat implements Animal {
    public void move( ) {
        System.out.println("I am walking using 4 toes");
    }
    public void communicate( ) {
        System.out.println("I meow");
    }
}
```

Методи на интерфейсите

- Методите на интерфейсите са public и abstract по подразбиране
- Модификаторите public и abstract (заедно или поотделно) могат да бъдат указани и експлицитно
  - дали да бъдат експлицитно указани, е въпрос на стил - но е добре да сме консистентни
- Тъй като методите са абстрактни, не могат да бъдат декларирани като final

Интерфейси и наследяване

- Интерфейсите могат да се наследяват
- Един интерфейс може да наследява множество интерфейси

Интерфейси и имплементаците им

- Интерфейсите не могат да се инстанцират
- Можем да инстанцираме (конкретни) класове, които ги имплементират
- Можем да присвояваме инстанция на клас на променлива от тип интерфейс, който класът имплементира
- Можем да проверяваме с instanceof дали клас имплементира даден интерфейс;
- Един клас може да имплементира множество интерфейси
- Ако даден клас декларира, че имплементира интерфейс, той трябва или
  - да даде дефиниции на всичките му методи, или
  - да бъде деклариран като абстрактен
- Като следствие, ако променим сигнатурата на метод(и) на интерфейса, или ако добавим нов(и) метод(и), трябва да променим и всички имплементиращи интерфейса класове
- Това често е проблем, а понякога е и невъзможно (нямаме контрол върху всички имплементации)

Default методи в интерфейсите (от Java 8)

- Default-ен метод в интерфейс е метод, който
  - има имплементация
  - има модификатора default в декларацията си
- Имплементиращите класове имплицитно ползват default-ната имплементация на методите, но могат и да я предефинират
- Клас може да имплементира произволен брой интерфейси;
- Ако два или повече от тях съдържат default метод с еднаква сигнатура, класът трябва задължително да предефинира този метод;
- В предефинирания метод може експлицитно да се укаже, default-ната имплементация от кой родителски интерфейс да се ползва. В този случай, синтаксисът е, <имеНаИнтерфейс>.super.<имеНаDefaultМетод>( )

```
public interface OptimisticLockable {
    default boolean isLocked( ) {
        return false;
    }
}
public interface PessimisticLockable {
    default boolean isLocked( ) {
        return true;
    }
}
public class Door implements OptimisticLockable, PessimisticLockable {

    // We will get a compile-time error, if we don't override the isLocked( ) method here:
    // - "Door inherits unrelated defaults for isLocked( ) from types
    //       OptimisticLockable and PessimisticOldLockable"
    @Override
    public boolean isLocked( ) {
        return OptimisticLockable.super.isLocked( );
    }
}
```

- От Java 8, интерфейсите могат да съдържат и статични методи с имплементация;
- Една класическа употреба е, за factory методи (ще говорим за тях в лекцията за design patterns)

Private методи в интерфейсите (от Java 9)

- От Java 9, интерфейсите могат да съдържат и private методи с имплементация
- Използват се, когато в интерфейса има два ли повече default-ни или статични метода, чиято имплементация частично се дублира
  - тогава изнасяме повтарящия се код в private метод, за да предотвратим code duplication

Интерфейсите – обобщение

- Интерфейсите могат да съдържат
  - Публични, абстрактни методи без имплементация
  - static final член-променливи == константи
  - default и static методи с имплементация (от Java 8)
  - private методи (от Java 9)

Интересни частни случаи на интерфейси

- Интерфейс, който
  - не съдържа нито един метод, се нарича маркерен
  - има точно един публичен абстрактен метод, се нарича функционален

Абстракция

- Абстракция означава, моделирайки в обектно-ориентиран език за програмиране обекти от реалния или виртуалния свят, да се ограничим само до съществените им за конкретната задача характеристики и да се абстрахираме (пропуснем) в модела несъществените или нерелевантни за задачата
  - Пример: моделирайки студент, да го характеризираме само с име и факултетен номер, абстрахирайки се от всички други характеристики на студента в реалния свят (напр. цвят на очите)
- Абстракция също означава да работим с нещо, което знаем как да използваме, без да знаем как работи вътрешно. Всяка конкретна имплементация на поведение е скрита в своя обект, за външния свят е видимо само поведението (т.е. интерфейсът)
- Принципът за абстракция се постига в Java чрез интерфейси и абстрактни класове.

Предаване на аргументи в Java

- конструкторите и методите имат списък от (нула или повече) параметри (или формални параметри)
- параметрите са списък (наредена n-торка) от тип и име на параметъра
- при извикване на конструктора или метода, се подава списък от аргументи (или фактически параметри), които трябва да са съвместими като брой, подредба и тип със списъка параметри

Предаване на примитивни типове като аргументи

Примитивните типове се предават по стойност, т.е. създават се техни копия и промяната на стойностите на параметрите вътре в конструктора или метода не се отразява на стойностите на променливите, подадени като аргументи.

```
public class PassPrimitiveByValue {

   public static void main(String[ ] args) {
      int x = 3;
      // invoke passMethod( ) with x as argument
      passMethod(x);
      // print x to see if its value has changed
      System.out.println("After invoking passMethod( ), x = " + x);
   }

   // change parameter in passMethod( )
   public static void passMethod(int p) {
      p = 10;
   }
}
```

Предаване на референтни типове като аргументи

Референтните типове (обекти, масиви) също се предават по стойност: референциите, подадени като аргументи, имат същата стойност (т.е. сочат към същото място в паметта), както и преди извикването. Стойностите на член-данните на реферираните обекти обаче могат да се променят в метода, ако е налице необходимият достъп (в смисъл на енкапсулация) до тях.

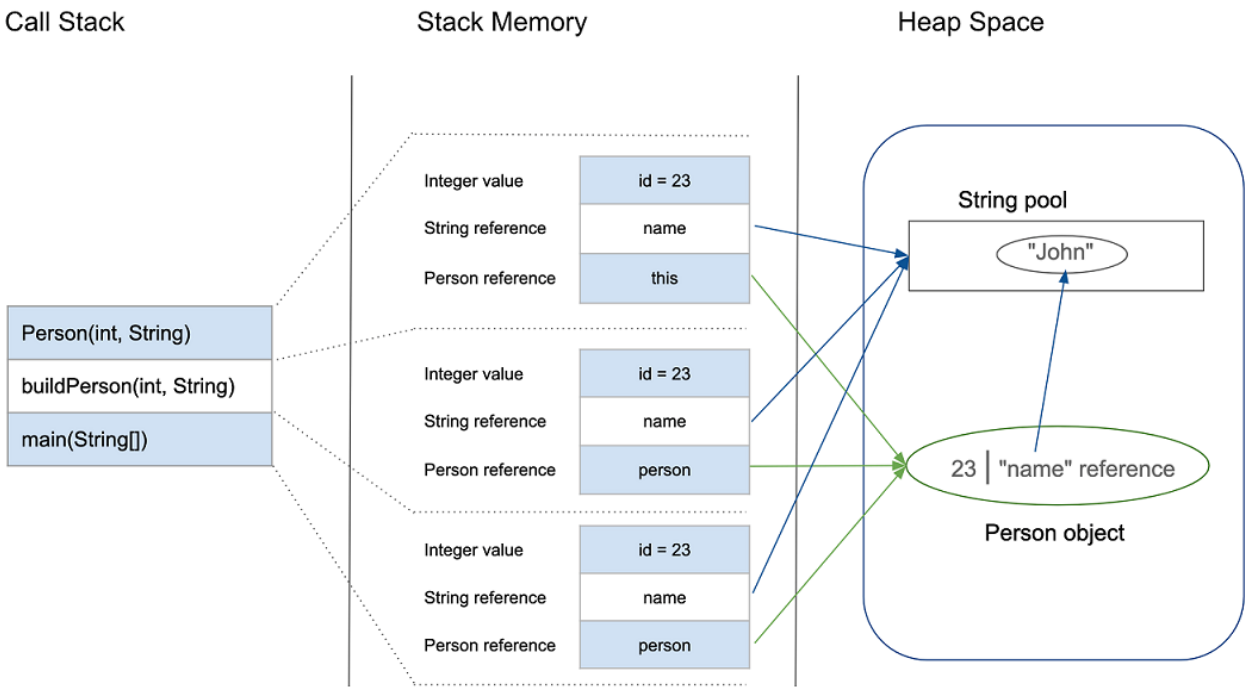
```
public void moveCircle(Circle circle, int deltaX, int deltaY) {
   // code to move origin of circle to x+deltaX, y+deltaY
   circle.setX(circle.getX( ) + deltaX);
   circle.setY(circle.getY( ) + deltaY);

   // code to assign a new reference to circle
   circle = new Circle(0, 0);
}
// [...]
moveCircle(myCircle, 23, 56); // myCircle reference remains unchanged,
// but coordinates of the referred circle are changed after the call
```

```
public class Person {
    int id;
    String name;

    public Person(int id, String name) {
        this.id = id;
        this.name = name;
    }
}
public class PersonBuilder {
    private static Person buildPerson(int id, String name) {
        return new Person(id, name);
    }

    public static void main(String[ ] args) {
        int id = 23;
        String name = "John";
        Person person = null;
        person = buildPerson(id, name);
    }
}
```



Характеристики на stack паметта

- автоматично се заделя при извикване на метод и освобождава при приключването му
- променливите в стека съществуват, докато трае изпълнението на метода, който ги е създал (като параметри и локални променливи)
- ако стек паметта се запълни, се хвърля java.lang.StackOverFlowError
- обикновенно е с по-малък размер от heap паметта
- достъпът до стек паметта е бърз в сравнение с достъпа до heap паметта

Характеристики на heap паметта

- динамична памет за обекти и масиви
- заделя се явно с оператора new или неявно, и се освобождава от garbage collector-a
- новосъздаваните обекти стоят в heap-a, а референциите към тях - в stack-a
- ако heap паметта се запълни, се хвърля java.lang.OutOfMemoryError

- достъпът до heap паметта е по-бавен в сравнение с достъпа до stack паметта

Създаване и инициализация на обектите

- член-променливите на даден клас се инициализират със стойността по подразбиране на дадения тип
- може да ги инициализираме явно, с присвояване, като част от декларацията им
  - това е подходящо, ако е с литерал или прост израз, но не и ако инициализацията има по-сложна логика, например включва обработка на грешки, завъртане на цикъл за масиви и т.н.
- нестатичните член-променливи могат да се инициализират и в конструкторите;
- за статичните член-променливи обаче, няма такава възможност. Затова са измислени т.нар. инициализационни блокове (initializers)

Initializer

- представлява блок код, който
  - не е свързан с име или тип данни
  - намира се в тяло на дефиниция на клас, но извън конструктор или метод
- в даден клас може да има нула, един или няколко инициализационни блока;
- могат да се ползват за инициализация на член-данните на класа (статични и нестатични)
- алтернативата е, инициализационната логика да е отделена в методи

```
public class InitializeMe {

    private int a;
    private static int b;

    static {
        // static initializer block
        b = 100;
    }

    {
        // initializer block
        a = 5;
    }
}
```

Ред на инициализация на обектите

1. инициализират се статичните член-променливи и се изпълняват статичните инициализатори на базовия клас, в реда на срещането им в дефиницията на класа
2. инициализират се статичните член-променливи и се изпълняват статичните инициализатори на класа, в реда на срещането им в дефиницията му
3. изпълняват се инициализаторите на базовия клас, в реда на срещането им
4. изпълнява се конструкторът на базовия клас
5. изпълняват се инициализаторите на класа, в реда на срещането им
6. изпълнява се конструкторът на класа

Enum

- Специален референтен тип (клас), представящ фиксирано множество от инстанции-константи
- Нарича се enum(eration), защото инстанциите се дефинират чрез изброяване

```
public enum Day {
    SUNDAY,
    MONDAY,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY
}
```

- Всеки enum неявно наследява абстрактния клас java.lang.Enum
  - не може да наследява явно друг клас, защото би било множествено наследяване;
  - може да имплементира интерфейси
- Тялото на enum класа може да съдържа член-променливи и методи
- Ако има конструктор, той трябва да е package-private или private
- Той автоматично създава константите в дефиницията на enum-а. Не може да се извиква явно

```
public class EnumExample {
    private Day day;

    public EnumExample(Day day) { this.day = day; }

    public void tellItLikeItIs( ) {
        String message = switch (day) {
            case MONDAY -> "Mondays are bad.";
            case FRIDAY -> "Fridays are better.";
            case SATURDAY, SUNDAY -> "Weekends are best.";
            default -> "Midweek days are so-so.";
        };
        System.out.println(message);
    }

    public static void main(String[ ] args) {
        EnumExample example = new EnumExample(Day.TUESDAY);
        example.tellItLikeItIs( );
    }
}
```

- Компилаторът добавя автоматично и няколко специални статични методи:
  - values( ) – връща масив, съдържащ всички стойности в enum, в реда, в който са изброени в декларацията
  - valueOf(String name) – връща enum константата по даденото име

Records (от Java 16)

- една от най-честите критики към Java е, че се пише доста boilerplate код
- например, за почти всеки клас, се налага да дефинираме публичен конструктор, getter методи, toString( ), equals( ) и hashCode( )
- за много класове, те имат стандартни, тривиални имплементации и могат да бъдат генерирани автоматично (IDE-тата имат такава функционалност)
- дори като автори на класа да ги генерираме автоматично обаче, проблемът остава за четящите кода
- както enums, records са нов референтен тип и са (ограничен) вид клас
- описват компактно т.нар value objects - класове, които имат само състояние: състоят се от "полетата, само полетата и нищо освен полетата"

- може да мислим за тях като за именувана наредена n-торка стойности;
- те са (shallowly) immutable агрегации на фиксирано множество от стойности, известни като компоненти на record-а

```
public record Point(int x, int y) {}

// [...]

Point p = new Point(-1, 2);
System.out.println(p.x( ) + ", " + p.y( ));
```

- компилаторът автоматично генерира boilerplate кода за
  - private final член-данна за всяко поле, със същото име и тип
  - каноничен конструктор: със списък с параметри, отговарящ на декларираното описание на състоянието
  - getter методи за полетата (имената им съвпадат с имената на съответните полета);
  - equals( ), hashCode( ) и toString( )
    - ако r е record от тип R с компоненти c1, c2, ..., cN, и създадем копие на инстанцията като R copy = new R(r.c1( ), r.c2( ), ..., r.cN( )), то r.equals(copy) == true
  - компонентите на record-ите могат да имат анотации, например @NotNull
- инстанции се създават с оператора new
- всички records имплицитно наследяват абстрактния клас java.lang.Record, самите те са final и не могат да са abstract
- java.lang.Record не може да бъде явно наследяван
- той дефинира equals( ), hashCode( ) и toString( ) като абстрактни методи

Пример: без records

```
public final class FXOrderClassic {

  private final int units;
  private final CurrencyPair pair;
  private final Side side;
  private final double price;
  private final LocalDateTime sentAt;

  public FXOrderClassic(int units, CurrencyPair pair, Side side, double price, LocalDateTime sentAt) {
     this.units = units;
     this.pair = pair; // CurrencyPair is an enum, e.g. CurrencyPair.BGN_TO_EUR
     this.side = side; // Side is an enum, e.g. Side.BID or Side.ASK
     this.price = price;
     this.sentAt = sentAt;
  }

  public int units( ) {
     return units;
  }
  public CurrencyPair pair( ) {
     return pair;
  }
  public Side side() {
     return side;
  }
  public double price( ) {
     return price;
  }
  public LocalDateTime sentAt( ) {
     return sentAt;
  }

  // continues on next page...
```

```
    // ...continues from previous page

    @Override
    public boolean equals(Object o) {
        if (this == o) {
            return true;
        }
        if (!(o instanceof FXOrderClassic)) {
            return false;
        }

        FXOrderClassic that = (FXOrderClassic) o;
        return units == that.units &&
                Double.compare(that.price, price) == 0 &&
                pair == that.pair &&
                side == that.side &&
                sentAt.equals(that.sentAt);
    }

    @Override
    public int hashCode( ) {
        return Objects.hash(units, pair, side, price, sentAt);
    }

    @Override
    public String toString( ) {
        return "FXOrderClassic{" +
                "units=" + units +
                ", pair=" + pair +
                ", side=" + side +
                ", price=" + price +
                ", sentAt=" + sentAt +
                '}';
    }
}
```

Пример: с records

```
public record FXOrder(int units,
                      CurrencyPair pair,
                      Side side,
                      double price,
                      LocalDateTime sentAt) {}
```

- опционално, освен едноредова декларация, record-ите могат да съдържат допълнителни конструктори, методи, статични член-данни и статични factory методи
- ако обаче се изкушаваме да дефинираме такива, да имплентираме допълнителни интерфейси и т.н., вероятно по-добро дизайн решение би било да създадем обикновен клас
- изключение са т.нар. компактни конструктори: в тялото на каноничния конструктор, добавяме само валидация и/или нормализация на параметрите, а останалата част от кода (присвояванията на полетата) се осигурява от компилатора
- компактните конструктори нямат параметри, пропускат се дори празните скоби ( )
- кодът в компактния конструктор се добавя като допълнителен код в началото на дефиницията на каноничния конструктор (а не е отделен конструктор)

Пример: record с компактен конструктор

```
public record FXOrder(int units,
                      CurrencyPair pair,
                      Side side,
                      double price,
                      LocalDateTime sentAt,
                      int ttl) {

   public FXOrder {
      if (units < 1) {
         throw new IllegalArgumentException("FXOrder units must be positive");
      }

      if (price <= 0.0) {
         throw new IllegalArgumentException("FXOrder price must be positive");
      }
   }
}
```

Пример: record със статичен factory метод

```
public static FXOrder of(CurrencyPair pair, Side side, double price) {
   return new FXOrder(1, pair, side, price, LocalDateTime.now( ));
}
```

Локални интерфейси, enums и records (от Java 16)

- Интерфейси, enums и records могат да бъдат дефинирани в тялото на метод, което ги прави локални за него
- Това подобрява енкапсулацията, ако въпросните типове са нужни само в имплементацията на метода

Sealed класове и интерфейси (от Java 17)

- sealed типовете ни позволяват да ограничим класовете или интерфейсите, които могат да ги наследяват или имплементират
- така могат да се дефинират възможните йерархии в дадена проблемна област по декларативен начин

```
// sealed класовете декларират, кои класове могат да ги наследяват
public abstract sealed class Shape
   permits Circle, Rectangle {...}

// дефиниране на класове - наследници на sealed клас
public class Circle extends Shape {...} // OK
public class Rectangle extends Shape {...} // OK
public class Triangle extends Shape {...} // Compile error

// sealed класове като селектор на switch
// No need for default case if all permitted types are covered
double area = switch (shape) {
   case Circle c    -> Math.pow(c.radius( ), 2) * Math.PI;
   case Rectangle r -> r.a() * r.b( );
};
```

- Като разрешаваме само на предварително дефинирано множество от класове да наследяват даден клас, може да разделим достъпността (accessibility) от разширяемостта (extensibility) на класовете
- С други думи, може да направим sealed клас достъпен за други пакети, и въпреки това да контролираме, кой може да го наследява
- Преди Java 17, за постигнем подобен ефект, имаше две опции:

    - да направим класа package-private (което обаче ограничава достъпа до този клас)
    - да направим класа public, но с private или package-private конструктори (това прави класа видим, но дава много ограничен контрол върху конкретните типове, които могат да го наследят)
- При sealed класовете и интерфейсите, съществува изчерпателен списък от наследниците, който е известен на компилатора, IDE-то и JVM-а, което позволява по-мощен анализ на кода
    - например, instanceof и casts могат да проверяват цялата йерархия статично
- наследниците на sealed клас могат да бъдат final, non-sealed или sealed
- sealed клас може да бъде и абстрактен
- sealed класът и неговите permitted класове трябва да са в един пакет
- ако sealed класовете са кратки и малък брой, те могат да се дефинират в същия сорс файл като родителския клас
    - в този случай, permits клаузата може да се пропусне - компилаторът приема за имплицитно permitted класовете в дадения сорс файл

```
public sealed class Plant permits Herb, Shrub, Climber {}

final class Herb extends Plant {}

non-sealed class Shrub extends Plant {}

sealed class Climber extends Plant permits Cucumber{}

final class Cucumber extends Climber {}
```

- sealed интерфейсите могат да специфицират, кои интерфейси могат да ги наследяват и кои класове могат да ги имплементират
    - интерфейс – наследник на sealed интерфейс може да е деклариран като non-sealed или sealed. Интерфейсите не могат да са final
    - клас, имплементиращ sealed интерфейс, може да бъде final, non-sealed или sealed
    - record също може да имплементира sealed интерфейс
        - Тъй като всеки record е имплицитно final, той няма нужда от явен модификатор

```
sealed public interface Move permits Athlete, Person, Jump, Kick {}

final class Athlete implements Move {}

record Person(String name, int age) implements Move {}

non-sealed interface Jump extends Move {}

sealed interface Kick extends Move permits Karate {}

final class Karate implements Kick {}
```

Сорс файлове на абстрактни класове, интерфейси, enums и records

- Важат същите правила като при конкретните класове:
    - абстрактните класове, интерфейсите, enum-ите и record-ите се записват във файл
        - с име, съвпадащо с името на абстрактния клас/интерфейс/enum/record
        - с разширение .java

Деструктори

Деструкторът е специален метод, който се извиква автоматично веднага щом жизненият цикъл на даден обект приключи. Извиква се деструктор, за да освободи паметта. Следните задачи се изпълняват, когато се извика деструктор:

- Освобождаване на ключалките за освобождаване
- Затваряне на всички връзки към базата данни или файлове
- Освобождаване на всички мрежови ресурси
- Други заключителни задачи

- Възстановяване на хийп пространството, разпределено по време на живота на обект

Деструкторите в Java, известни също като финализатори, не са детерминирани. Разпределението и освобождаването на паметта се управляват имплицитно от събирача на отпадъци в Java (Garbage collector).

Събирач на отпадъци

Събирачът на отпадъци е програма, която работи на виртуалната машина на Java, за да възстанови паметта чрез изтриване на обекти, които вече не се използват или са завършили своя жизнен цикъл. Твърди се, че един обект отговаря на неизползваем (отпадък), тогава и само тогава, когато обектът е недостъпен.