

ИНТЕРФЕЙСИ

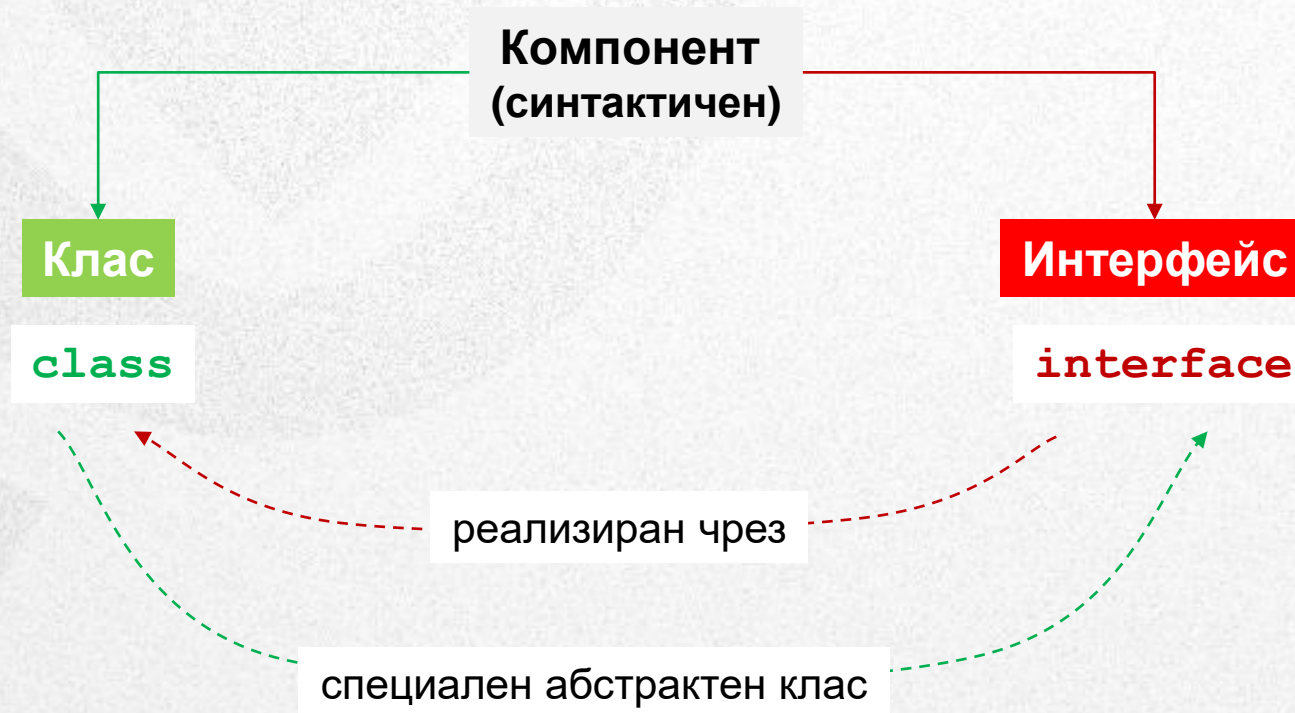
ЛЕКЦИОНЕН КУРС “ПРОГРАМИРАНЕ НА JAVA”



СТРУКТУРА НА ЛЕКЦИЯТА

- Обща характеристика
- Реализация
- Използване
- Примери

КОМПОНЕНТИ В JAVA



JAVA-СИНТАКСИС: КОМПОНЕНТИ

```
Source_text_file ::=  
    [Packet_declaration]  
    {Import}  
    {Typ_declaration}
```

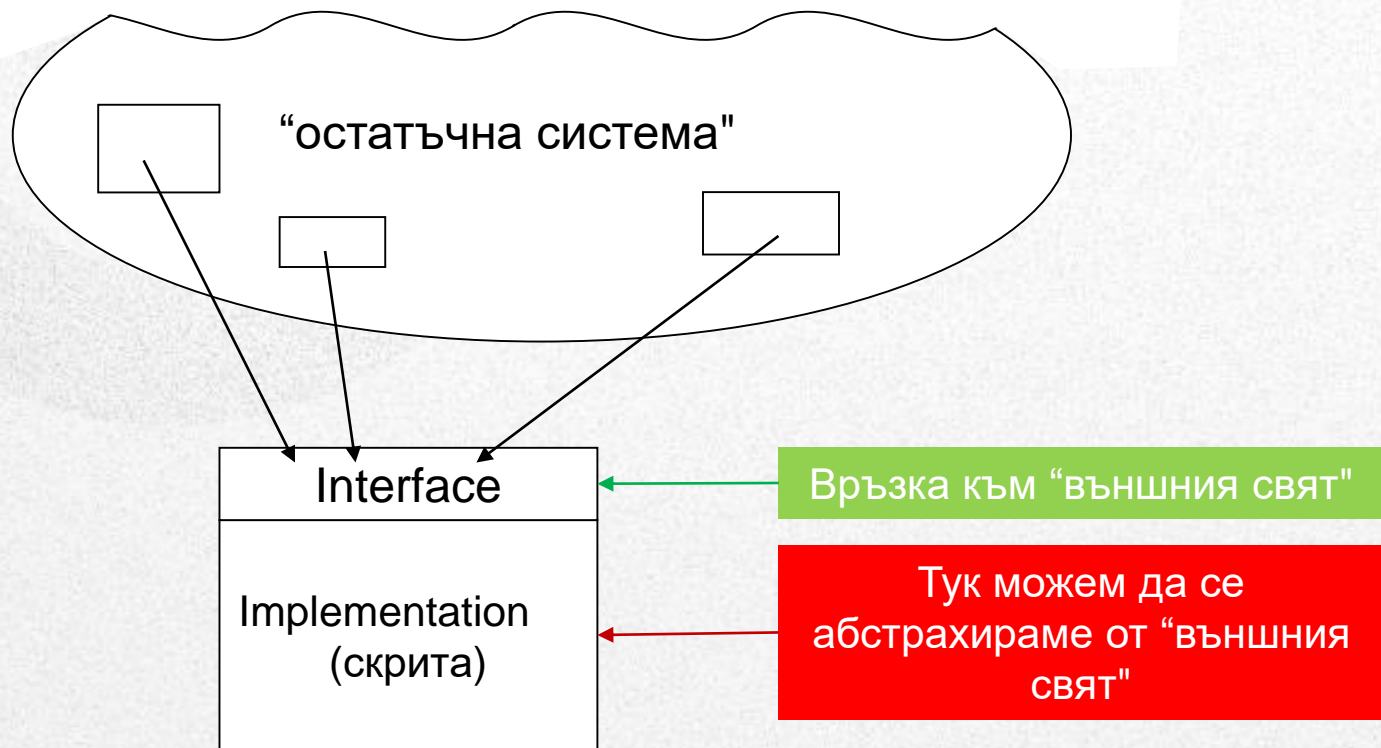
```
Typ_declaration ::=  
    Class_declaration |  
    Interface_declaration .
```


ФУНКЦИИ НА ИНТЕРФЕЙСИТЕ

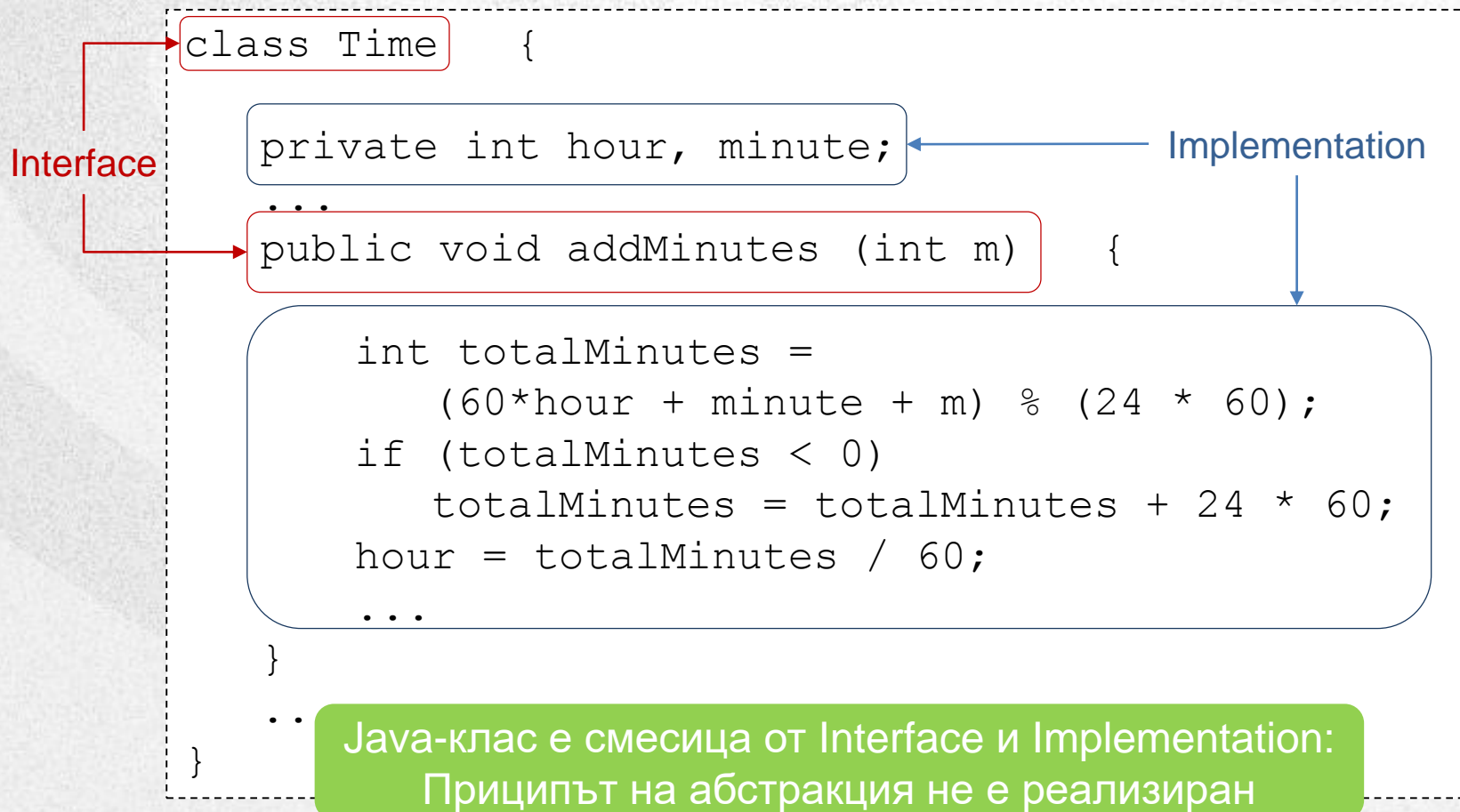
- Съдържателно-логическа функция:
 - Разделяне Interface – Implementation
- Техническа функция:
 - Ограничено многократно наследяване

СОФТУЕРНИТЕ КОМПОНЕНТИ СА АБСТРАКЦИИ

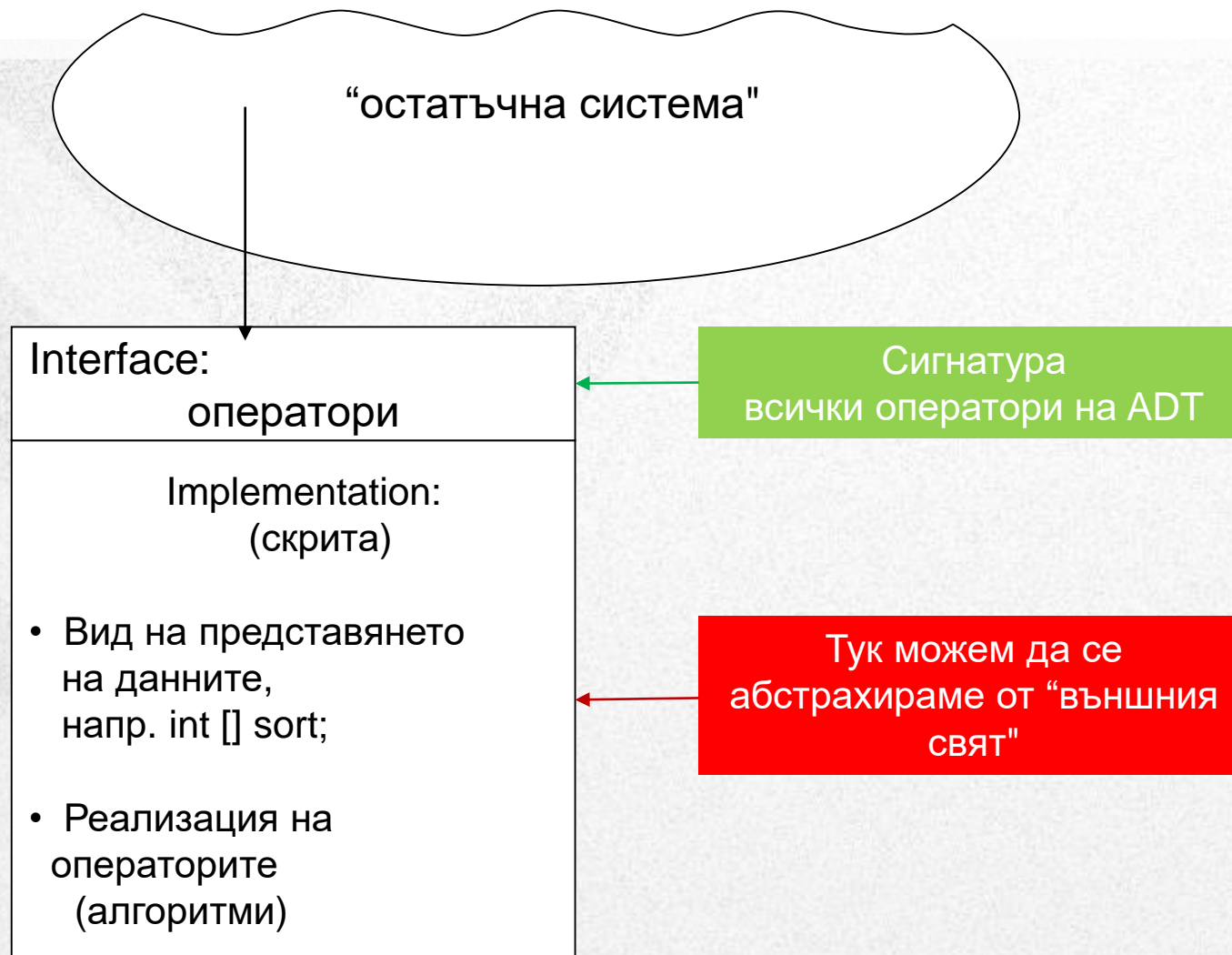
Компонент: Interface + Implementation



JAVA-КЛАС: ЕДНА АБСТРАКЦИЯ?



ИНТЕРФЕЙС НА ADT



C++ КЛАС: ЕДНА АБСТРАКЦИЯ ?

```
class Time {  
  
    public:  
        Time ();  
        Time (int h, int m);  
        void addMinutes (int m);  
        void printTime ();  
        ...  
  
    private:  
        int hour, minute;  
  
}
```

Извод: класовете в C++ са по-добри абстракции от класовете в Java

Но: Java-Interface преодолява този недостатък

JAVA-ИНТЕРФЕЙС: ЕДНА (ПОЧТИ) “ЧИСТА” АБСТРАКЦИЯ

```
interface TimeI {  
    public void addMinutes (int m);  
    public void subtractMinutes (int m);  
    public void printTime ();  
    public void printTimeInMinutes ();  
}
```

- Само заглавните части на методите
- Няма данни + няма алгоритми
- Какво липсва?

Конструктори: в Interface не са разрешени

```
TimeI(int h, int m);  
TimeI();
```

РЕАЛИЗАЦИЯ НА ИНТЕРФЕЙСИ

Потребител

```
interface TimeI {
    public void addMinutes (int m);
    public void subtractMinutes (int m);
    public void printTime ();
    public void printTimeInMinutes ();
}
```

```
class Time implements TimeI {
    private int hour, minute;

    public Time (int h, int m) {...}
    public void addMinutes (int m) {
        int totalMinutes =
            (60*hour + minute + m) % (24 * 60);
        if (totalMinutes < 0)
            totalMinutes = totalMinutes + 24 * 60;
        hour = totalMinutes / 60;
        ...
    }
    ...
}
```

Потребителят
трябва да знае от
реализация клас:
име, конструктори

Добавени:

- локални променливи
- конструктор
- тела на методите

ПРИЛОЖЕНИЕ НА ИНТЕРФЕЙСИТЕ

```
interface TimeI {  
    public void addMinute (int m);  
    public void subtractMinutes (int m);  
    public void printTime ();  
    public void printTimeInMinutes ();  
}
```

```
class Time implements TimeI {  
    private int hour, minute;  
    public Time (int h, int m)  
    public void addMinutes (int m)  
    ...  
}
```

Interface като нов (дефиниран от потребителя) тип

Променливи от
Interface-тип

Разрешени стойности:
обекти от ВСИЧКИ
имплементиращи класове

```
public static void main (...) {  
    TimeI t1;  
  
    t1 = new Time(8, 30);  
    t1.addMinutes(30);  
}
```

Конструкторът идентифицира
възнамеряван имплементиращ клас

INTERFACE: СПЕЦИАЛЕН СЛУЧАЙ НА АБСТРАКТНИ КЛАСОВЕ

```
abstract class TimeI {
    public abstract void addMinutes (int m);
    public abstract void subtractMinutes (int m);
    public abstract void printTime ();
    public abstract void printTimeInMinutes();
}

class Time extends TimeI {
    private int hour, minute;
    public void addMinutes (int m); {
        ...
    }
    ...
}
```

Interface и абстрактен клас: две различия

ИНТЕРФЕЙСИ И АБСТРАКТНИ КЛАСОВЕ

- **Интерфейси:** по-нататъшно развитие на абстракцията и известно опростяване на записа
 - Всички методи се приемат за абстрактни и публични
 - Съответните ключови думи могат да се пропуснат
 - Всички символни константи са `public`, `static`, `final`
 - Съответните ключови думи могат също така да бъдат пропуснати
- **Един клас** може да имплементира повече от един интерфейс
 - Множествено наследяване
 - Може да бъде подклас само на един клас

ОГРАНИЧЕНО МНОЖЕСТВЕНО НАСЛЕДЯВАНЕ

1 Как в Java?

Множествено наследяване:

- Забранено за класове
- Разрешено за интерфейси

1 Кой вариант?

```
(abstract) class K1 ...  
(abstract) class K2 ...  
class K3 extends K1, K2 ...
```

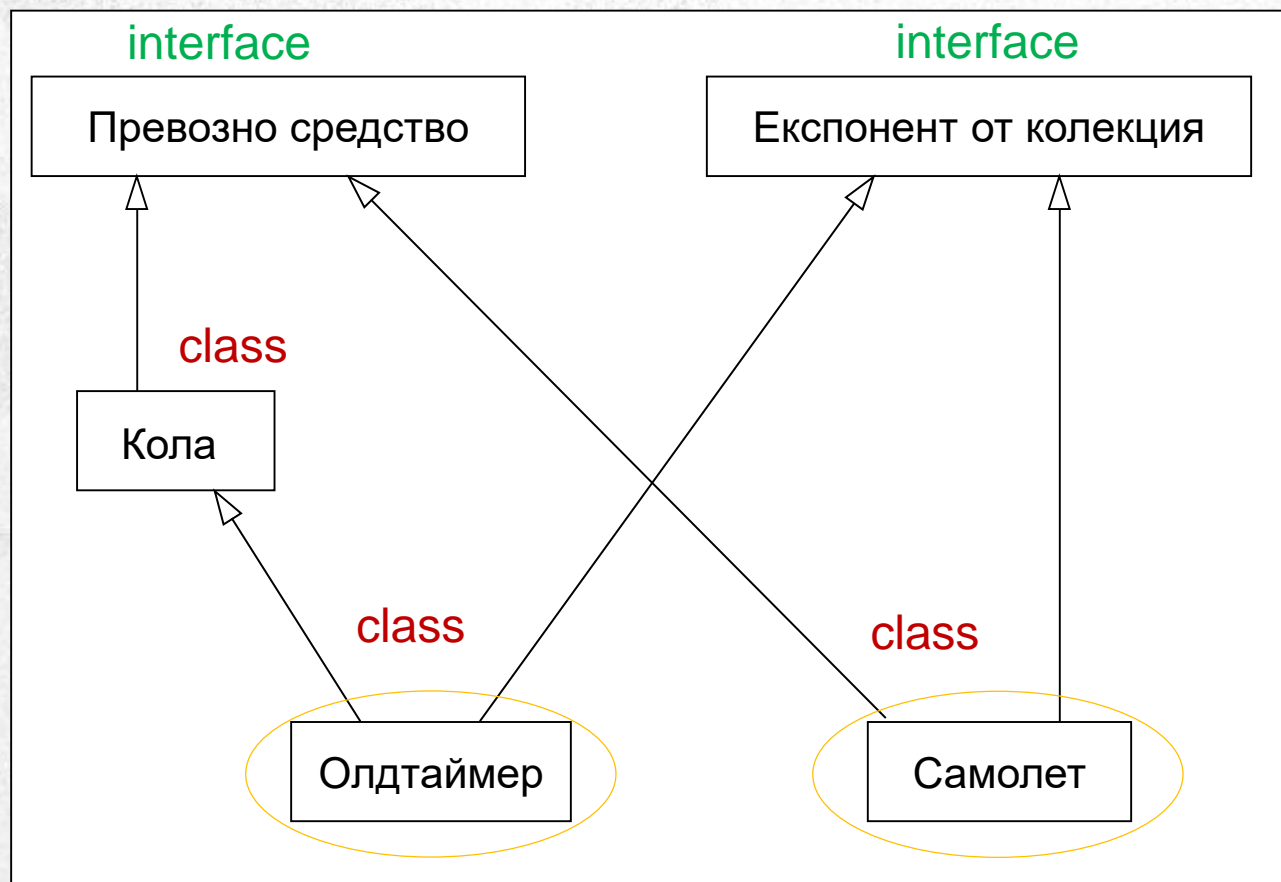
не

```
interface I1 ...  
interface I2 ...  
class K3 implements I1, I2 ...
```

да



ПРИМЕР: ЙЕРАРХИЯ НА НАСЛЕДЯВАНЕ



ПРИМЕР

```
interface Vehicle ...  
  
interface Exponent ...  
  
class Car implements Vehicle ...
```

```
class Oldtimer extends Car  
    implements Exponent
```

```
class Aircraft implements Vehicle,  
    Exponent
```

Всички елементи
(променливи, методи) на
'Car' и 'Exponent'

Всички елементи
(променливи, методи) на
'Vehicle' и 'Exponent'

ПРИМЕР: ПЪЛНА ФОРМА

```
interface Vehicle {  
    public int capacity();  
    public double spend();  
}
```

```
interface Exponent {  
    public double value();  
    public boolean exhibit();  
}
```

```
class Car implements Vehicle {  
    public String name;  
    private int numberCits;  
    private double fuelSpend;  
    public int capacity()  
        { return numberCits; }  
    public double spend()  
        { return fuelSpend; }  
}
```

```
class Oldtimer extends Car  
    implements Exponent {  
    private double exhValue;  
    private boolean exhibition;  
    public double value ()  
        { return exhValue; }  
    public exhibit ()  
        { return exhibition; }  
}
```

```
class Aircraft  
    implements Vehicle, Exponent ...
```

ПРИМЕР ЗА МНОЖЕСТВЕНО НАСЛЕДЯВАНЕ

```
import java.util.*;
interface CanFight { void fight(); }
interface CanSwim { void swim(); }
interface CanFly { void fly(); }
class ActionCharacter { public void fight() {System.out.println("Fight");} }
class Hero extends ActionCharacter implements CanFight, CanSwim, CanFly {
    public void swim() {System.out.println("Swim");}
    public void fly() {System.out.println("Fly");}
}
public class Adventure {
    static void t(CanFight x) { x.fight(); }
    static void u(CanSwim x) { x.swim(); }
    static void v(CanFly x) { x.fly(); }
    static void w(ActionCharacter x) { x.fight(); }
    public static void main(String[] args) {
        Hero h = new Hero();
        t(h); // Третира се като CanFight
        u(h); // Третира се като CanSwim
        v(h); // Третира се като CanFly
        w(h); // Третира се като ActionCharacter
    }
}
```



Какъв резултат?

fight
swim
fly
fight

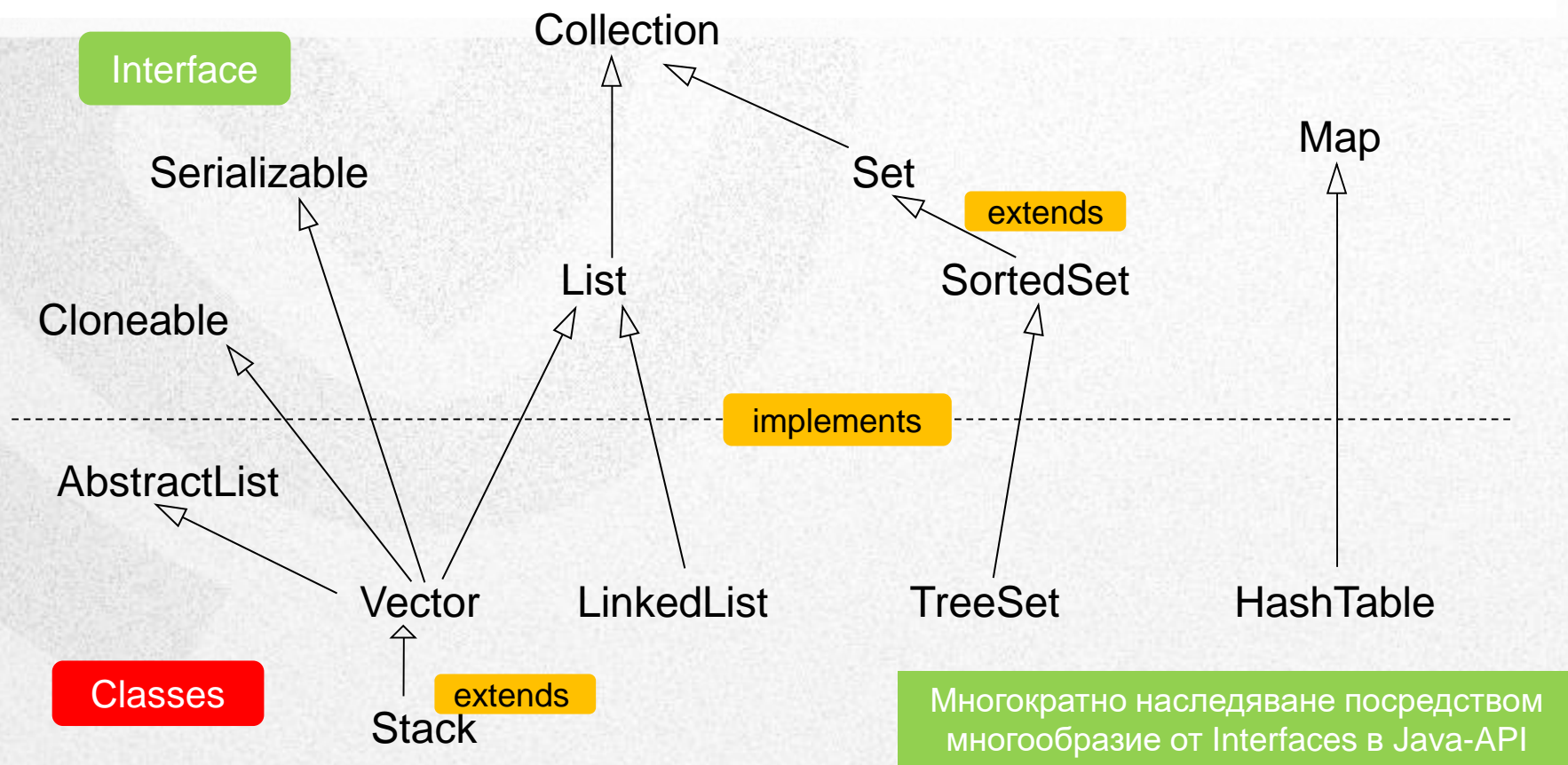
ОСНОВНА ЗАДАЧА НА ИНТЕРФЕЙСИТЕ

- Има 4 метода, които приемат като аргументи различни интерфейси и един конкретен клас
- Когато се създаде обект Hero, той може да се предаде на всеки от тези методи
 - Това означава, че се извършва преобразуване нагоре към всеки интерфейс
- Основна задача на интерфейсите:
 - Можем да преобразуваме нагоре към повече от един базов тип

СЪЩЕСТВЕНИ ПРИЛОЖЕНИЯ

- **Interface на един ADT**
 - две (повече) имплементации:
- **Interface за абстракции данни**
- **Import на общи константи на класове**
- **Алгоритми с 'Function parameters,**
- **Java-API: List – Set – Collection ...**

JAVA-API: МНОГООБРАЗИЕ НА INTERFACES



```
public class Vector extends AbstractList  
    implements List, Cloneable, Serializable
```

ДЕФИНИРАНЕ СВОЙСТВА НА ОБЕКТИ

- Cloneable
 - Обектите поддържат клониране
- Comparable
 - Обектите могат да бъдат сравнявани
- Serializable
 - Обектите могат да бъдат записани в обектен байтов поток за пренасяне в нова виртуална машина
- Runnable
 - Обектите имат поведение, което може да се изпълни в самостоятелен процес (нишка)

```
public interface Comparable {  
    int compareTo(Object o);  
}
```

СЪЩЕСТВЕНИ ПРИЛОЖЕНИЯ

- **Interface на ADT**
 - две (повече) имплементации
- Interface за абстракции данни:
- Import на общи константи на класове
- Алгоритми с 'Function parameters'
- Java-API: List – Set – Collection ...

INTERFACE НА ЕДИН ADT – ДВЕ (ПОВЕЧЕ) РЕАЛИЗАЦИИ

Предимство: Добра модифицируемост, ако приложението работи само с информация от Interface

Ограничен размер

Неограничен размер

```
interface Stack {  
    public boolean isempty();  
    public void push(char x);  
    public char top();  
    public void pop();  
}
```

```
class StackN implements Stack {  
    private char[] stackElements;  
    private int top;  
    ...  
}
```

```
class StackU implements Stack {  
    private class Cell {...}  
    private Cell top;  
    ...  
}
```

ИЗПОЛЗВАНЕ: ПОЗНАТ САМО INTERFACE + КОНСТРУКТОР

```
interface Stack {  
    public boolean isempty();  
    public void push(char x);  
    public char top();  
    public void pop();  
}
```

Особеност на TurnNU.java: Първо динамично се избира имплементиращият клас → динамично свързване на методите по време на обработка

Приложение:

Алгоритъмът е коректен за произволни имплементации:
- ограничени стекове
- неограничени стекове

```
Stack s;  
s = new StackN(n);
```

или

```
s = new StackU();
```

```
while (!s.isempty()) {  
    System.out.print(s.top());  
    s.pop();  
}
```

Смяна на реализацията:
само конструкторът се сменя

БЛАГОДАРЯ ЗА ВНИМАНИЕТО!

КРАЙ “ИНТЕРФЕЙСИ”

