

# Обектно-ориентирано програмиране с **Java** (част II)

# В предната лекция говорихме за...

Класове и обекти

`java.lang.Object`

Абстрактни класове и интерфейси

Stack vs. Heap

Фундаменталните ООП принципи

- Енкапсулация
- Наследяване
- Полиморфизъм
- Абстракция

# Днес продължаваме с...

Wrapper types

Статични член-променливи и статични методи

Enums

Исключения


# Видове памет на JVM-а: stack и heap

Stack	Heap
<ul style="list-style-type: none"><li>• Литерали и променливи от примитивните типове</li><li>• Параметри и локални променливи в методи</li><li>• Референции към обекти</li></ul>	<ul style="list-style-type: none"><li>• Обекти (инстанции на класове)</li><li>• Масиви</li></ul>
<ul style="list-style-type: none"><li>• Заделяне: при извикване на метод</li><li>• Освобождаване: при приключване на изпълнението на метод</li></ul>	<ul style="list-style-type: none"><li>• Заделяне: с оператора new</li><li>• Освобождаване: от JVM Garbage Collector-а</li></ul>
<ul style="list-style-type: none"><li>• с кратък живот (method invocation)</li><li>• бърз достъп</li><li>• не се фрагментира</li></ul>	<ul style="list-style-type: none"><li>• Обикновено по-дълъг живот</li><li>• по-бавен достъп</li><li>• може да се фрагментира</li></ul>

Размерите на stack-а и heap-а се указват като command-line аргументи при стартиране на JVM-а и не могат да се променят без рестарта ѝ.

# Примитивни типове и wrapper типове

Primitive type	Size	Minimum	Maximum	Wrapper type
<b>boolean</b>	—	—	—	<b>Boolean</b>
<b>char</b>	16 bits	Unicode 0	Unicode $2^{16}-1$	<b>Character</b>
<b>byte</b>	8 bits	-128	+127	<b>Byte</b>
<b>short</b>	16 bits	$-2^{15}$	$+2^{15}-1$	<b>Short</b>
<b>int</b>	32 bits	$-2^{31}$	$+2^{31}-1$	<b>Integer</b>
<b>long</b>	64 bits	$-2^{63}$	$+2^{63}-1$	<b>Long</b>
<b>float</b>	32 bits	IEEE754	IEEE754	<b>Float</b>
<b>double</b>	64 bits	IEEE754	IEEE754	<b>Double</b>
<b>void</b>	—	—	—	<b>Void</b>



range at full precision	precision*
$\pm 1.18 \times 10^{-38}$ to $\pm 3.4 \times 10^{38}$	approx. 7 decimal digits
$\pm 2.23 \times 10^{-308}$ to $\pm 1.80 \times 10^{308}$	approx. 15 decimal digits

# Къде се ползват wrapper типовете?

- Където синтаксисът на езика изисква обект, а не примитивен тип.

Например в колекциите, за които предстои да учим: може да имате масиви от `int`, `boolean` и т.н., но не може да имате колекции от тях, а само от обекти: `Integer`, `Boolean` и т.н.

- Когато ви трябват константи или помощни функции, които са имплементирани в съответния wrapper клас, например

```
Integer.MAX_VALUE    // максималната стойност на типа
Integer.MIN_VALUE    // минималната стойност на типа
intValue(String)      // връща int стойността, “опакована” в дадената инстанция
Integer.parseInt(String) // конвертира низ с текстово представяне на цяло число към int
```

# Autoboxing

```
// initialization of a primitive char with a literal
```

```
char c = 'a';
```

```
// initialization of a Character object with an instance with value 'a'
```

```
Character ch = new Character('a');
```

```
// autoboxing: char implicitly converted to Character
```

```
Character ch = 'x';
```

```
// autounboxing: Character instance implicitly converted to char
```

```
char c = ch;
```

# Числа с голяма / произволна\* точност

`java.math.BigInteger` → цели числа

`java.math.BigDecimal` → реални числа с десетична точка

`add()`, `multiply()`, `subtract()`, `divide()`, ...

Приличат на Wrapper класове, но нямат „примитивен“ аналог

\**произволна* точност / брой цифри не значи *неограничена*: ограничава ни на практика наличната памет



static

# 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. Например:

```
Math.PI           // константата л  
Math.pow(double, double) // вдига първия аргумент на степен втория
```

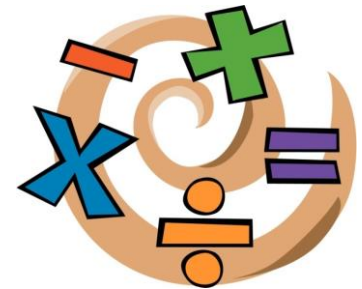
Статичните член-променливи имат едно-единствено копие, което се споделя от всички инстанции на класа.

- 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 = 3.14; // constant  
    private static int radius = 10; // static member  
    private String fact5 = "5!"; // non-static member  
  
    public static long fact(int n) { // static method  
        if (n == 1) { return 1; } else { return n*fact(n-1); }  
    }  
    public String getFact() { // non-static method  
        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() will not compile  
    }  
}
```

# Enum тип



# Enum тип

Специален тип (клас), представящ фиксирано множество от инстанции-константи

Нарича се *enum(eration)*, защото инстанциите се дефинират чрез *изброяване* в декларацията на enum типа и множеството им е константно, т.е. не могат да се добавят и махат runtime

Декларира се с ключовата дума enum

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.

# Enum тип

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 името ѝ (т.е. идентификатора, с който е декларирана).

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

```
Day day = Day.SATURDAY;
```

```
if (day == Day.SUNDAY) {  
  
}
```

```
public class EnumExample {
    Day day;

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

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

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





```

public enum Weekdays {
    MONDAY(1, "MON"),
    TUESDAY(2, "TUE"),
    WEDNESDAY(3, "WED"),
    THURSDAY(4, "THU"),
    FRIDAY(5, "FRI"),
    SATURDAY(6, "SAT"),
    SUNDAY(7, "SUN");

    private final int dayNumber;
    private final String shortName;

    private Weekdays(int dayNumber, String shortName) {
        this.shortName = shortName;
        this.dayNumber = dayNumber;
    }

    public int getDayNumber() {
        return dayNumber;
    }

    public String getShortName() {
        return shortName;
    }

    public String toString() {
        return String.format("%d %s", this.getDayNumber(), this.getShortName());
    }
}

```

```

public class WeekdaysTest {
    public static void main(String[] args) {
        System.out.println(Weekdays.MONDAY);
        System.out.println(Weekdays.SUNDAY);

        for (Weekdays weekday : Weekdays.values()) {
            System.out.println(weekday);
        }
    }
}

```

```

1 MON
7 SUN
1 MON
2 TUE
3 WED
4 THU
5 FRI
6 SAT
7 SUN

```

# Исключения

# Изключения (Exceptions)

*Изключение* е събитие (проблем), което се случва по време на изпълнение на дадена програма и нарушава нормалната последователност на изпълнение на инструкциите ѝ.

*(Изключение е съкратено за изключително събитие)*

Още един начин за комуникация на метод с извикващите го: връщана стойност при нормално изпълнение и изключение при проблем.

# Например...

- Подали сме невалидни входни данни
- Опитваме се да отворим несъществуващ файл
- Мрежата се е разкачила по време на комуникация
- Свършила е паметта на виртуалната машина
- ...

## Как се генерира („хвърля“) изключение?

```
public Object pop() {  
    if (size == 0) {  
        throw new EmptyStackException();  
    }  
    ...  
}
```

# Как се обработва („лови“) изключение

```
try {  
    // код, който може да хвърли изключение  
} catch (Exception e) {  
    // обработваме изключението (“exception handler”).  
    // Може да има повече от един catch блок  
} finally {  
    // при нужда, някакви заключителни операции  
    // (finally блокът е optional, но ако го има, се изпълнява задължително  
    щом влезем в try-a)  
}
```

# Catch block chain

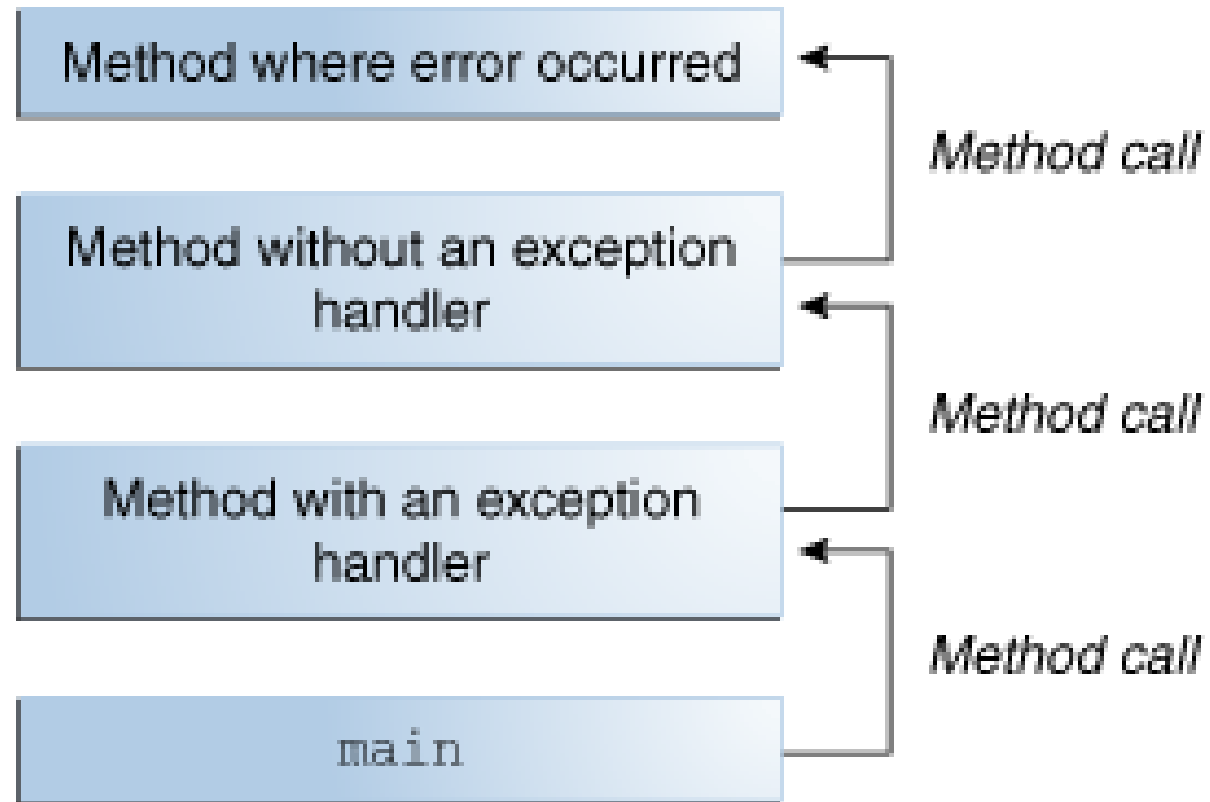
```
try {  
    ...  
} catch (MostSpecificException mse) {  
} catch (MoreGeneralException mge) {  
} ... {  
} catch (LeastSpecificException lse) {  
}
```

# Multi catch block

```
catch (IOException | SQLException ex) {  
    logger.log(ex);  
    throw ex;  
}
```



# Стек на извикванията (call stack)



# Finally – не само за обработка на изключения

```
try {  
    // тук може да се хвърлят изключения  
    // или да има return/continue/break  
} finally {  
    // някакъв важен cleanup code -  
    // ще се изпълни винаги*, независимо какво се случи в try блока  
}
```

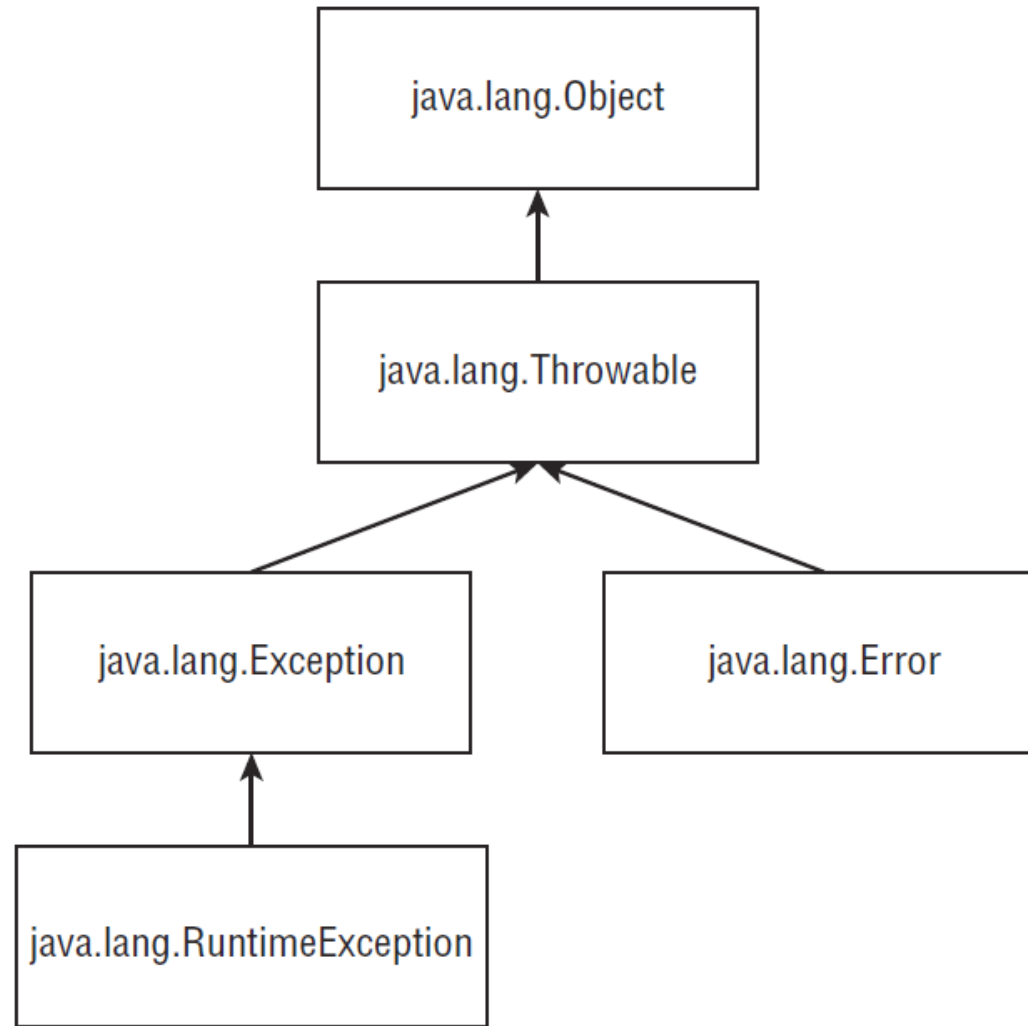
# Видове изключения

Изключителните събития могат да се дължат на грешка на потребителя, бърк в кода или физически ресурс, който не е достъпен.

Делят се на три вида:

- Checked exceptions
- Unchecked (Runtime) exceptions
- Errors

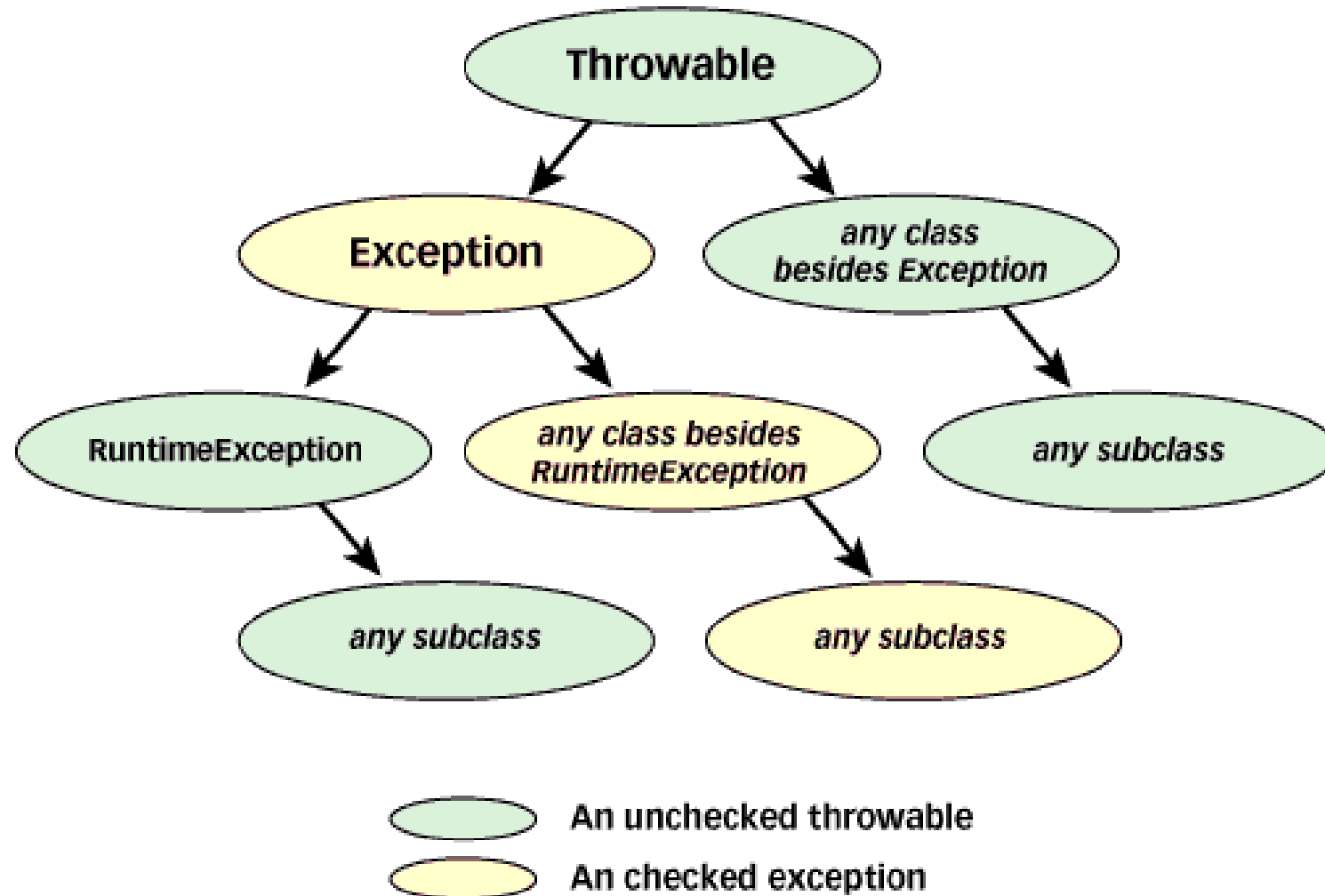
# Видове изключения



# Видове изключения

Вид	Как да го разпознаем	ОК е да бъде catch-нат / обработен	Задължително е да бъде обработен или деклариран
Runtime Exception	Наследник на <code>java.lang.RuntimeException</code>	Да*	Не
Checked Exception	Наследник на <code>java.lang.Exception</code> без да е наследник на <code>java.lang.RuntimeException</code>	Да	Да
Error	Наследник на <code>java.lang.Error</code>	Не*	Не

# Checked vs. unchecked exceptions



# Checked Exceptions

Наричат се още *compile-time exceptions*, защото компилаторът ни задължава да ги обработим в кода. Едно добре написано приложение би трябвало да ги очаква и да се възстановява от тях.

Пример:

- `FileNotFoundException` при опит да отворим файл по име, какъвто не съществува
- `IOException` при проблем с четене или писане във файл

# Unchecked (Runtime) Exceptions

Възникват по време на изпълнение на приложението, затова се наричат още *runtime exceptions*. Приложението обикновено не може да ги очаква или да се възстанови от тях. Най-често са резултат от бъгове (логически грешки) в кода, неправилно извикване на API-та и т.н. Може да се „ловят“ и обработват, но обикновено е по-правилно да се дебъгне и елиминира бъгът, който ги причинява.

Примери:

- `ArithmeticException` при опит за деление на нула
- `ArrayIndexOutOfBoundsException` при опит да достъпим елемент на масив по индекс извън размера на масива
- `NullPointerException` при подаване на `null reference`, където се очаква обект
- `NumberFormatException` при опит да се коинвертира низ в неподходящ формат към числов тип
- `ClassCastException` при опит да се `cast`-не обект към клас, на който обектът не е инстанция



# Errors

Проблеми, които възникват извън приложението, и приложението обикновено не може да ги очаква или да се възстанови от тях. Обикновено се генерират от самата виртуална машина.

Примери:

- `OutOfMemoryError` при опит да заделим памет, когато свободната памет не е достатъчна (и не може да освободи с `garbage collection`)
- `StackOverflowError` когато метод извиква свое копие твърде много пъти (напр. при безкрайна рекурсия)

# Деклариране на хвърляни изключения

Ако метод не прехваща/обработва даден **checked** exception, който може да се хвърли в тялото му, той трябва да го декларира в прототипа си, за да „предупреди“ тези, които го викат:

```
public void writeList() throws IOException, FileNotFoundException {  
  
    ...  
}
```

# Chained exceptions

```
try {  
    ...  
} catch (IOException e) {  
    // прехващаме изключение, обработваме го и хвърляме ново, към което го  
    „закачаме“  
    throw new SampleException("Other IOException", e);  
}
```

# Защо да ползваме изключения?

- Отделяме кода за обработка на грешки от останалия → става по-четим
- „Препредаване“ на грешки по стека на извикванията
- Групиране и диференциране на различните типове грешки

**Въпроси?**