

Java

Řetězce

Řetězec

- instance třídy `java.lang.String`
kompilátor s nimi zachází *téměř* jako s primit. typy
 - řetězcové konstanty = instance třídy `String`
- **nezměnitelné!!!**
 - pro změny – třídy `StringBuffer`, `StringBuilder`
- operátor `+`
 - spojování řetězců
 - pokud je ve výrazu aspoň jeden řetězec -> vše se zkonvertuje na řetězec
 - metoda `toString()`
 - definována na třídě `Object`
 - obvyklý kandidát na předefinování
 - vznikne nový řetězec

java.lang.String

- konstruktory

```
String();  
String(char[] value);  
String(byte[] bytes);  
String(byte[] bytes, String charsetName);  
String(String value);  
String(StringBuffer value);  
String(StringBuilder value);
```

java.lang.String

- metody

- `int length();`
- `char charAt(int index);`
 - `IndexOutOfBoundsException`
- `boolean equals(Object o);`
 - porovnává řetězce
 - `==` porovnává reference

```
String a = new String("ahoj");  
String b = new String("ahoj");  
System.out.println(a==b); // false  
System.out.println(a.equals(b)); //true
```

java.lang.String

- metody

- `int compareTo(String s);`

- porovnává lexikograficky

- `int compareToIgnoreCase(String s);`

- `int indexOf(char c);`

- `int indexOf(String s);`

- vracejí -1, pokud se nevyskytuje

- `String substring(int beginIndex);`

- `String substring(int beginIndex, int endIndex);`

- `String replaceFirst(String regexp, String repl);`

- `String replaceAll(String regexp, String repl);`

Řetězce

- metody (pokrač.)
 - `String join(CharSequence delimiter, CharSequence... elements);`
 - od Java 8
- metody lze volat i na řetězcových konstantách

```
String s;  
...  
if ("ahoj".equals(s)) {  
    ...  
}
```

Java

Wrapper typy

Wrapper typy

- neměnitelné
- Integer
 - konstruktory – od Java 9 jsou deprecated
 - ~~Integer(int value)~~
 - ~~Integer(String s)~~
 - metody
 - `int intValue()`
 - `static Integer valueOf(int i)`
 - může kešovat hodnoty
 - `static int parseInt(String s)`
 - ...
- ostatní wrapper typy obdobně

Java

Ještě k metodám

Lokální proměnné

- definice kdekoliv v těle
- viditelnost v bloku
 - viz první přednášku
- neinicializované
- lze definovat jako **final**
 - konstanta
 - jiný modifikátor nelze
- *effectively final*
 - není definována jako **final**, ale po inicializaci se hodnota nemění

Inference typů lok. prom.

- od Java 10
- pouze u lokálních proměnných

```
var s = "hello";  
var list = new ArrayList<String>();
```

- var – rezervované jméno typu
 - není to klíčové slovo
- musí být inicializace
- ne vždy lze použít
 - nelze
 - null
 - inicializace pole
 - lambda

Přetížení metod

- více metod se stejným jménem a různými parametry
 - různý počet a/nebo typ

```
public void draw(String s) {  
    ...  
}  
public void draw(int i) {  
    ...  
}  
public void draw(int i, double f) {  
    ...  
}
```

- nelze přetížit jen pomocí změny návratového typu

Rekurzivní volání

- rekurze – metoda volá sebe sama

```
public static long factorial(int n) {  
    if (n == 1) return 1;  
    return n * factorial(n-1);  
}
```

- pozor na ukončení
- neukončení -> přetečení zásobníku
 - velikost lze nastavit

Java

Výjimky

Výjimky (exceptions)

- hlášení a ošetření chyb
 - výjimka signalizuje *nějaký* chybový stav
- výjimka = instance třídy `java.lang.Throwable`
- dvě podtřídy – `java.lang.Error` a `java.lang.Exception`
 - konkrétní výjimky – jejich potomci
- `java.lang.Error`
 - "nezotavitelné" chyby
 - neměly by se odchyťovat
 - př. `OutOfMemoryError`
- `java.lang.Exception`
 - indikují zotavitelné chyby
 - lze je odchyťovat
 - př. `ArrayIndexOutOfBoundsException`

Zpracování výjimek

- příkaz `try/catch/finally`

```
try {  
    ... // zde je blok kodu, kde muze nastat  
        // chyba a chceme ji osetrit  
} catch (Exception1 e) {  
    // osetreni vyjimky typu Exception1  
    // pripadne jejich podtypu  
} catch (Exception2 e) {  
    // osetreni vyjimky typu Exception2  
    // pripadne jejich podtypu  
} finally {  
    // provede se vzdy  
}
```


Zpracování výjimek

- pokud výjimku neodchytí blok, kde nastala, šíří se do následujícího vyššího bloku
- pokud není odchycena v metodě, šíří se do volající metody
- pokud se dostane až do `main()` a není odchycena, způsobí ukončení interpretu Javy
 - vypíší se informace o výjimce + kde nastala a jak se šířila

try/catch/finally

- lze vynechat catch nebo finally
 - nelze vynechat oboje zároveň

Rozšířený try (od Java 7)

- interface **AutoClosable** a rozšířený **try**

– př:

```
class Foo implements AutoClosable {  
    ...  
    public void close() { ... }  
}
```

```
try ( Foo f1 = new Foo(); Foo f2 = new Foo() ) {  
    ...  
} catch (...) {  
    ...  
} finally {  
    ...  
}
```

- při ukončení try (normálně nebo výjimkou) se vždy zavolá `close()` na objekty z deklarace v try
 - volá se v opačném pořadí než deklarováno

Rozšířený try

- lze vynechat catch i finally zároveň

```
try (Resource r = new Resource()) {  
    ...  
}
```

- od Java 9 lze v hlavičce try použít (effectively) final proměnné

```
final Resource resource1 = new Resource("res1");  
Resource resource2 = new Resource("res2");  
  
try (resource1; resource2) {  
    ...  
}
```

„multi“ catch (od Java 7)

```
class Exception1 extends Exception {}  
class Exception2 extends Exception {}
```

```
try {  
    boolean test = true;  
    if (test) {  
        throw new Exception1();  
    } else {  
        throw new Exception2();  
    }  
} catch (Exception1 | Exception2 e) {  
    ...  
}
```

Deklarace výjimek

- metoda, která může způsobit výjimku
 - musí výjimku odchytit
 - nebo specifikovat typ výjimky pomocí `throws`

```
public void openFile() throws IOException {  
    ...  
}
```

- pomocí `throws` **nemusejí být deklarovány**
 - potomci `java.lang.Error`
 - potomci `java.lang.RuntimeException`
 - je potomek `java.lang.Exception`
 - př. `NullPointerException`,
`ArrayIndexOutOfBoundsException`

Generování výjimek

- příkaz `throw`
 - vyhodí výjimku
 - "parametr" – reference na objekt typu `Throwable`

```
throw new MojeVyjimka();
```

- vyhazovat lze existující výjimky, častěji však vlastní výjimky
- výjimky lze „znovu-vyhazovat“

```
try {  
    ...  
} catch (Exception e) {  
    ...  
    throw e;  
}
```

Znovu-vyhození výjimky (Java 7)

```
class Exception1 extends Exception {}  
class Exception2 extends Exception {}
```

```
public static void main(String[] args) throws  
Exception1, Exception2 {  
    try {  
        boolean test = true;  
        if (test) {  
            throw new Exception1();  
        } else {  
            throw new Exception2();  
        }  
    } catch (Exception e) {  
        throw e;  
    }  
}
```

- od Java 7 si výjimka “pamatuje” svůj typ
- na Java 6 nelze přeložit
- vyžadovalo by `throws Exception`

java.lang.Throwable

- má atribut (private) typu `String`
 - obsahuje podrobnější popis výjimky
 - metoda `String getMessage()`
- konstruktory
 - `Throwable()`
 - `Throwable(String msg)`
 - `Throwable(String msg, Throwable cause)`
// od 1.4
 - `Throwable(Throwable cause)` // od 1.4
- metody
 - `void printStackTrace()`

Vlastní výjimky

```
public class MyException extends Exception {
    public MyException() {
        super();
    }
    public MyException(String s) {
        super(s);
    }
    public MyException(String s, Throwable t) {
        super(s, t);
    }
    public MyException(Throwable t) {
        super(t);
    }
}
```

Řetězení výjimek

```
...  
try {  
    ...  
    ...  
} catch (Exception1 e) {  
    ...  
    throw new Exception2 (e);  
}  
...
```

- reakce na výjimku jinou výjimkou
 - běžná praxe
 - reakce na „systémovou“ výjimku „vlastní“ výjimkou

Potlačení výjimek

- v některých případech jedna výjimka může potlačit jinou výjimku
 - není to řetězení výjimek!
 - typicky se může stát
 - při výjimce ve **finally** bloku
 - v rozšířeném **try** bloku (Java 7)
- **Throwable[] getSuppressed()**
 - metoda na **Throwable**
 - vrátí pole potlačených výjimek

JAVA

Vnitřní třídy

Vnitřní třídy

- inner classes
- definice třídy v těle třídy

```
public class MyClass {
    class InnerClass {
        int i = 0;
        public int value() { return i; }
    }
    public void add() {
        InnerClass a = new InnerClass();
    }
}
```

Vnitřní třídy

- vnější třída může vrátet reference na vnitřní

```
public class MyClass {
    class InnerClass {
        int i = 0;
        public int value() { return i; }
    }
    public InnerClass add() {
        return new InnerClass();
    }
    public static void main(String[] args) {
        MyClass p = new MyClass();
        MyClass.InnerClass a = p.add();
    }
}
```

Skrytí vnitřní třídy

- vn. třída může být `private` i `protected`
- přístup k ní přes interface

```
public interface MyIface {
    int value();
}
public class MyClass {
    private class InnerClass implements MyIface {
        private i = 0;
        public int value() {return i;}
    }
    public MyIface add() {return new InnerClass();}
}
...
public static void main(String[] args) {
    MyClass p = new MyClass();
    MyIface a = p.add();
    // nelze - MyClass.InnerClass a = p.add();
}
```


Vn. třídy v metodách

- vn. třídy lze definovat i v metodách nebo jen v bloku
- platnost jen v dané metodě (bloku)

```
public class MyClass {
    public MyIface add() {
        class InnerClass implements MyIface {
            private i = 0;
            public int value() {return i;}
        }
        return new InnerClass();
    }
    public static void main(String[] args) {
        MyClass p = new MyClass();
        MyIface a = p.add();
        // nelze - MyClass.InnerClass a = p.add();
    }
}
```

Anonymní vn. třídy

```
public class MyClass {
    public MyIface add() {
        return new MyIface() {
            private i = 0;
            public int value() {return i;}
        };
    }

    public static void main(String[] args) {
        MyClass p = new MyClass();
        MyIface a = p.add();
    }
}
```

Anonymní vn. třídy

```
public class Wrap {
    private int v;
    public Wrap(int value) { v = value; }
    public int value() { return v; }
}

public class MyClass {
    public Wrap wrap(int v) {
        return new Wrap(v) {
            public int value() {
                return super.value() * 10;
            }
        };
    }

    public static void main(String[] args) {
        MyClass p = new MyClass();
        Wrap a = p.wrap(5);
    }
}
```

Anon. vn. třídy: inicializace

- objekty mimo anon. vn. třídu potřebné v anon. vn. třídě – **final**
- bez `final` – chyba při překladu
- od Java 8 – stačí „**effectively**“ `final`
 - tj. lze i bez modifikátoru `final`, ale obsah se nesmí měnit

- do Java 7 je zde **final** nutné
- od Java 8 lze i bez **final**
 - obsah `val` se nemění

```
public class MyClass {  
    public MyIface add(final int val) {  
        return new MyIface() {  
            private int i = val;  
            public int value() {return i;}  
        };  
    }  
}
```

Anon. vn. třídy: inicializace

- anon. vn. třídy nemůžou mít konstruktor
 - protože jsou anonymní
- inicializátor objektu

```
public class MyClass {  
    public MyIface add(final int val) {  
        return new MyIface() {  
            private i;  
            {  
                if (val < 0)  
                    i = 0;  
                else  
                    i = val;  
            }  
            public int value() {return i;}  
        };  
    }  
}
```

Vztah vnitřní a vnější třídy

- instance vnitřní třídy může přistupovat ke **všem** členům nadřazené třídy

```
interface Iterator {
    boolean hasNext();
    Object next();
}

public class Array {
    private Object[] o;
    private int next = 0;
    public Array(int size) {
        o = new Object [size];
    }
    public void add(Object x) {
        if (next < o.length) {
            o[next] = x;
            next++;
        }
    }
} // pokračovani....
```

Vztah vnitřní a vnější třídy

```
// pokračovani....
private class AIterator implements Iterator {
    int i = 0;
    public boolean hasNext() {
        return i < o.length;
    }
    public Object next() {
        if (i < o.length)
            return o[i++];
        else
            throw new NoNextElement();
    }
}

public Iterator getIterator() {
    return new AIterator();
}
}
```

Vztah vnitřní a vnější třídy

- reference na objekt vnější třídy
 - `JmenoTridy.this`
 - předchozí příklad – třídy `Array` a `AIterator`
 - reference na objekt `Array` z `Array.AIterator` – `Array.this`

Vztah vnitřní a vnější třídy

- vytvoření objektu vnitřní třídy vně třídy s definicí vnitřní třídy

```
public class MyClass {  
    class InnerClass {  
    }  
    public static void main(String[] args) {  
        MyClass p = new MyClass();  
        MyClass.InnerClass i = p.new InnerClass();  
    }  
}
```

- nelze vytvořit objekt vnitřní třídy bez objektu vnější třídy
 - objekt vnitřní třídy má vždy (skrytou) referenci na objekt vnější třídy

Vícenásobné vnoření tříd

- při vícenásobném vnoření lze z vnitřních přistupovat na vnější třídy na libovolné úrovni

```
class A {
    private void f() {}
    class B {
        private void g() {}
        class C {
            void h() {
                g();
                f();
            }
        }
    }
}
public class X {
    public static void main(String[] args) {
        A a = new A();
        A.B b = a.new B();
        A.B.C c = b.new C();
        c.h();
    }
}
```

Dědění od vnitřní třídy

- **explicitně** předat referenci na objekt vnější třídy

```
class WithInner {
    class Inner {}
}
class InheritInner extends WithInner.Inner {
    InheritInner(WithInner wi) {
        wi.super();
    }
    // InheritInner() {} // špatne, chyba pri
    // prekladu

    public static void main(String[] argv) {
        WithInner wi = new WithInner();
        InheritInner ii = new InheritInner(wi);
    }
}
```

Vnořené (nested) třídy

- definovány s `static`
- nemají referenci na objekt vnější třídy
- může mít statické metody a atributy
 - vnitřní třída nemůže mít statické členy
- pro vytváření instancí nepotřebují objekt vnější třídy
 - nemají na něj referenci
- v podstatě normální třídy, pouze umístěné do jmenového prostoru vnější třídy

```
public class MyClass {  
    public static class NestedClass {  
    }  
  
    public static void main(String[] args) {  
        MyClass.NestedClass nc =  
            new MyClass.NestedClass();  
    }  
}
```

Vnořené třídy

- lze je definovat uvnitř interfacu
 - vnitřní třídy nelze

```
interface MyInterface {  
    static class Nested {  
        int a, b;  
        public Nested() {}  
        void m();  
    }  
}
```

Vnitřní třídy a .class soubory

- vnitřní (i vnořené) třída – vlastní .class soubor
- `JmenoVnejsi$JmenoVnitрни.class`
 - `MyClass$InnerClass.class`
- **anonymní vnitřní třídy**
 - `JmenoVnejsi$PoradoveCislo.class`
 - `MyClass$1.class`
- **vnořená třída může obsahovat metodu `main`**
 - spuštění programu: `java`
`JmenoVnejsi$JmenoVnořené`

Důvody použití vnitřních tříd

- ukrytí implementace
- přístup ke všem prvkům vnější třídy
- "callbacks"
- ...



Verze prezentace J03.cz.2018.01

Tato prezentace podléhá licenci [Creative Commons Uveďte autora-Neužívejte komerčně 4.0 Mezinárodní License](https://creativecommons.org/licenses/by-nc/4.0/).