

Java

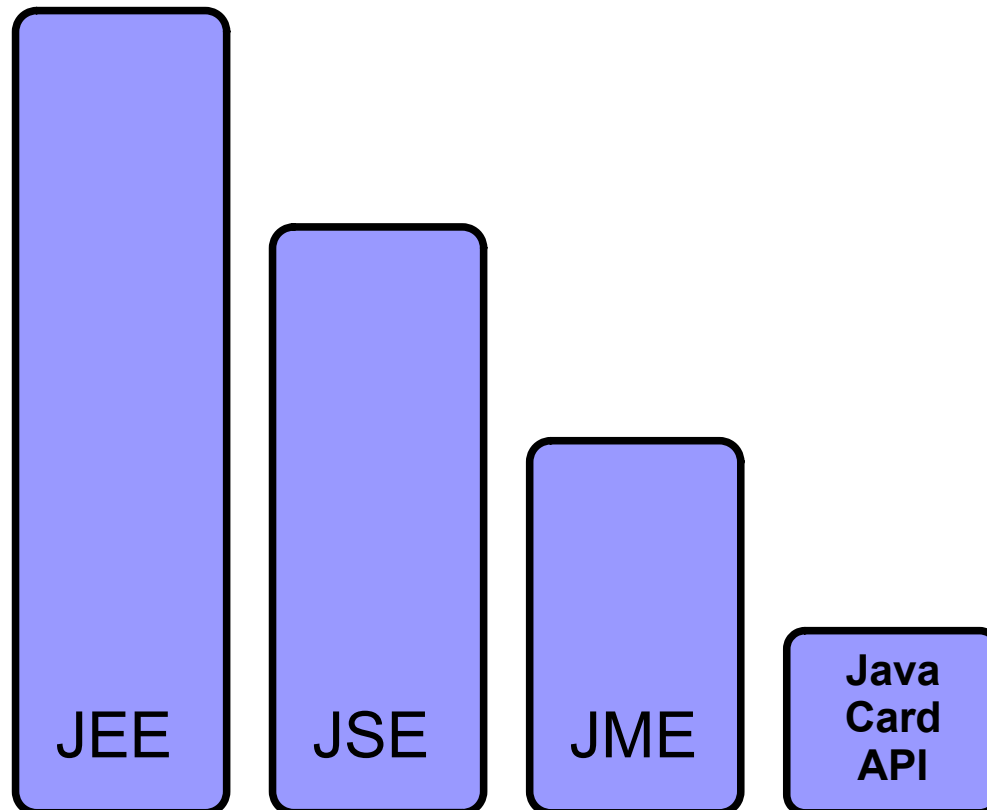
Platformy apod.

Historie

- JDK 1.0 – 1996
- JDK 1.1 – 1997
 - Vnitřní třídy
- Java 2 platform – 2000
 - JDK 1.2, 1.3 – změny pouze v knihovnách
- JDK 1.4 – 2002
 - Assert
- JDK 5.0 – 2004
 - změny v jazyce
 - generické typy
 - anotace
 - ...
- JDK 6 – 2006
- JDK 7 – 2011 – „malé“ změny v jazyce
- JDK 8 – 2014 – lambda, ...
- JDK 9 – 2017 – moduly
- JDK 10 – 2018 – odvození typu lokálních proměnných (var)
- JDK 11 – 2018 – rozšíření použití var
 - redukce std knihovny!
 - long-term support
- JDK 12 – 2019 – rozšířený switch (jen jako „preview“)
- JDK 13 – 2019 – další úprava switch, textové bloky (stále jen „preview“)

Java platform

- JSE – standard edition
- JEE – enterprise edition
- JME – micro edition



"Výkon"

- původně (~ JDK 1.1, 1998)
 - Java programy 6x pomalejší než C
- nyní:
 - Just-In-Time (JIT) compilation
 - v rámci spuštění se program přeloží
 - provádí se nativní kód
 - pomalý start, pak rychlé
- výkon ~ srovnatelný s normálními aplikacemi
- velká "spotřeba" paměti

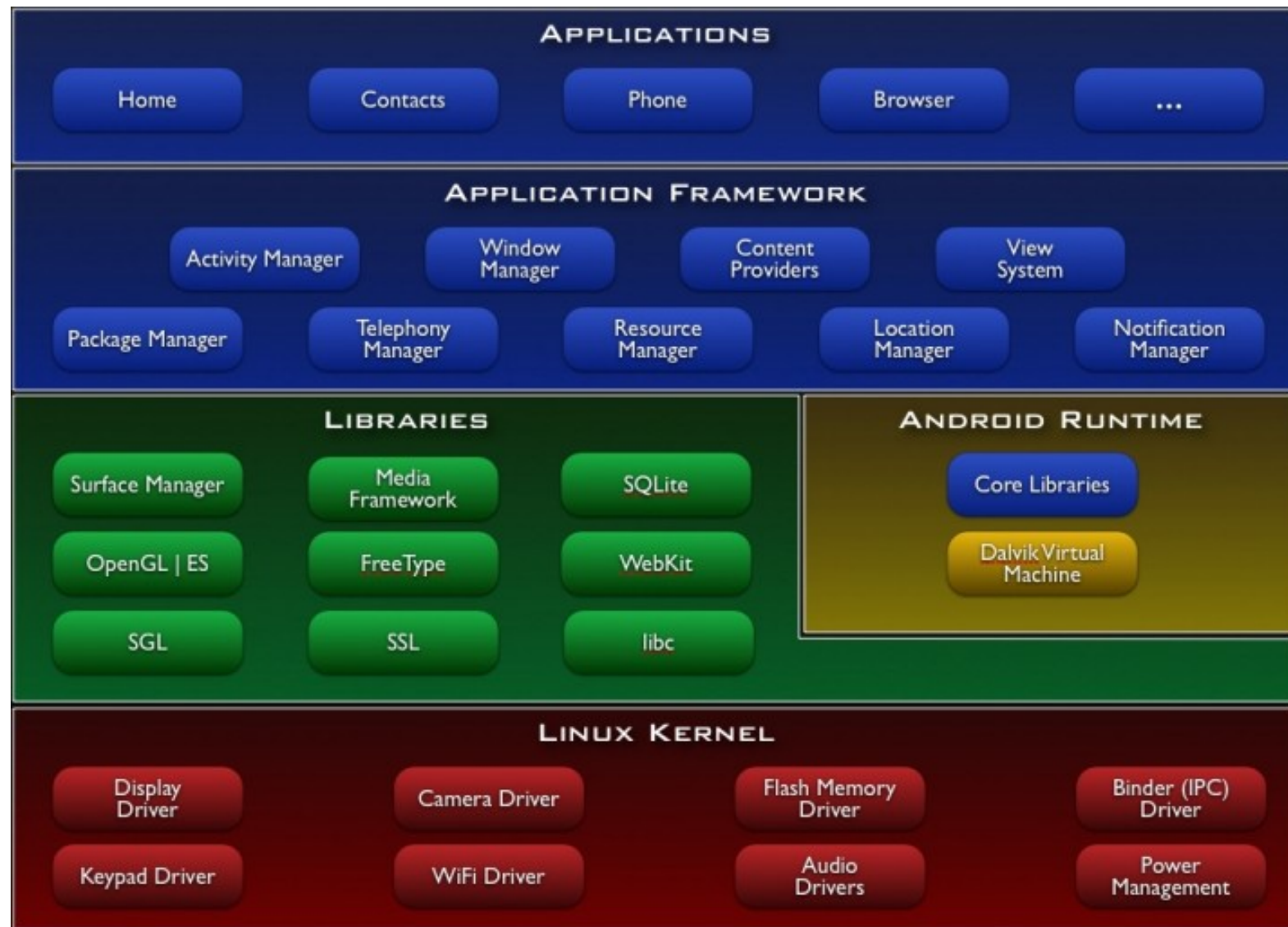
Implementace Javy

- Oracle
 - "oficiální" implementace
 - Windows, Solaris, Linux, macOS
- OpenJDK
 - <http://openjdk.java.net/>
 - open-source
 - podporováno Oracle
 - oficiální implementace vychází z OpenJDK
- IBM
 - IBM JDK
 - 2017 => Eclipse OpenJ9 – open-source

Implementace Javy

- Jikes RVM
 - Research Virtual Machine
 - open-source
 - pro testování různých rozšíření
 - napsáno v Javě
 - "self-hosting"
 - nepotřebuje k běhu jinou JVM
 - boot-image writer
 - Java program, spustí se pod existující JVM
 - boot-image loader
 - program v C++
 - nepodporuje celé Java API

Android



zdroj: <http://developer.android.com/>

Bck2brwsr

- <http://wiki.apidesign.org/wiki/Bck2Brwsr>
- Java běžící v prohlížeči
- Cíle projektu
 - „Create small Java capable to boot fast and run in 100% of modern browsers including those that have no special support for Java.
 - Demonstrate that Java has benefits over JavaScript when creating larger HTML5 applications“
 - ...

JAVA

Historie a budoucnost

Změny v jazyce od Java 5

- static import
- auto-boxing a auto-unboxing
- nový **for** cyklus
- generické typy
- výčtový typ **enum**
- metody s proměnným počtem parametrů (printf)
- anotace (metadata)

Java 7

- změny
 - změny v syntaxi
 - podpora dynamických jazyků (nová instrukce v bytekódu)
 - změny v NIO
 - Nimbus (Swing LaF)
 - nová verze JDBC
 - ...

Java 7 – změny v syntaxi

- zápis konstant
 - binární konstanty
 - `0b010101`
 - oddělení řádu v konstantách pomocí podtržítek
 - `1_000_000`
- String v switch příkazu

```
String month;
```

```
...
```

```
switch (month) {  
    case "January":  
    case "February":  
        ...  
}
```

Java 7 – změny v syntaxi

- operátor `<>`
 - zjednodušená vytváření generických typů
 - typ u `new` se automaticky odvodí
 - př.

```
List<String> list = new ArrayList<>();  
List<List<String>> list = new ArrayList<>();  
List<List<List<String>>> list =  
                                new ArrayList<>();  
Map<String, Collection<String>> map =  
                                new LinkedHashMap<>();
```

- otázka
Proč není `new` bez `<>` ? Tj. např.
`List<String> list = new ArrayList();`

Java 7 – změny v syntaxi

- 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

Java 7 – změny v syntaxi

- rozšíření **catch** na více výjimek


– př:

```
try {  
    ...  
} catch (Exception1 | Exception2 ex) {  
    ...  
}
```

- lepší typová kontrola výjimek při **re-throw**

Je tento kód správně

```
private void foo(int i) throws Ex1, Ex2 {  
    try {  
        if (i < 0) {  
            throw new Ex1();  
        } else {  
            throw new Ex2();  
        }  
    } catch (Exception ex) {  
        throw ex;  
    }  
}
```



- v Java 7 ano
- v Java 6 ne
 - kompilátor vypíše chybu zde

Java 8

- type annotations
 - lze anotovat použití typů
 - opakování stejné anotace
- default a static metody v interfacech
- lambda výrazy
- odvození typu u generických typů
- profily
 - „podmnožina“ std knihovny
 - javac -profile ...

Java 9 – modules

- Modul
 - pojmenovaná a „sebe-popisující“ kolekce balíčků (packages) s typy (třídy,...) a daty
 - deklaruje
 - závislosti (vyžadované moduly)
 - poskytované balíčky

module-info.class

```
module com.foo.bar {  
  requires com.foo.baz;  
  exports com.foo.bar.alpha;  
  exports com.foo.bar.beta;  
}
```

Java 9 – moduly

- JSE platforma – rozdělena na moduly

```
module java.base {  
    exports java.io;  
    exports java.lang;  
    exports java.lang.annotation;  
    exports java.lang.invoke;  
    exports java.lang.module;  
    exports java.lang.ref;  
    exports java.lang.reflect;  
    exports java.math;  
    exports java.net;  
    ...  
}
```

Java 10

- odvození typu u lokálních proměnných

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

- var – rezervované jméno typu
- není to klíčové slovo
- musí být inicializace

Java 11

- použití var pro parametry lambda výrazů

Java 13

- upravený switch
 - šipka místo dvojtečky
 - bez break
 - více hodnot
 - lze použít jako výraz
- textové bloky
 - `"""multiline
string"""`

Java

Generické typy

Úvod

- *obdoba* šablon z C#/C++
 - na první pohled
- parametry pro typy
- cíl
 - přehlednější kód
 - typová bezpečnost

Motivační příklad

- bez gen. typů (<=JDK 1.4)

```
List myIntList = new LinkedList();  
myIntList.add(new Integer(0));  
Integer x = (Integer)myIntList.iterator().next();
```

- JDK 5

```
List<Integer> myIntList = new LinkedList<Integer>();  
myIntList.add(new Integer(0));  
Integer x = myIntList.iterator().next();
```

- bez explicitního přetypování
- kontrola typů během překlada

Definice generických typů

```
public interface List<E> {
    void add(E x);
    Iterator<E> iterator();
    E get(int i);
}

public interface Iterator<E> {
    E next();
    boolean hasNext();
}
```

- `List<Integer>` si lze představit jako

```
public interface IntegerList {
    void add(Integer x);
    Iterator<Integer> iterator();
}
```

- ve skutečnosti ale takový kód nikde neexistuje
 - negeneruje se kód jako C++

Vztahy mezi typy

- nejsou povoleny žádné změny v typových parametrech

```
List<String> ls = new ArrayList<String> ();  
List<Object> lo = ls;
```

```
lo.add(new Object());  
String s = ls.get(0);
```

chyba – přiřazení Object do String

- druhý řádek způsobí chybu při překladu
- List<String> **není** podtyp List<Object>

Vztahy mezi typy

- příklad – tisk všech prvků kolekci
≤ JDK 1.4

```
void printCollection(Collection c) {  
    Iterator i = c.iterator();  
    for (k = 0; k < c.size(); k++) {  
        System.out.println(i.next());  
    }  
}
```

naivní pokus v JDK 5

```
void printCollection(Collection<Object> c) {  
    for (Object e : c) {  
        System.out.println(e);  
    }  
}
```

- nefunguje (viz předchozí příklad)

Vztahy mezi typy

- `Collection<Object>` není nadtyp všech kolekcí
- **správně**

```
void printCollection(Collection<?> c) {
    for (Object e : c) {
        System.out.println(e);
    }
}
```
- `Collection<?>` je nadtyp všech kolekcí
 - kolekce neznámého typu (collection of unknown)
 - lze přiřadit kolekci jakéhokoliv typu
- **pozor** - do `Collection<?>` nelze přidávat
 - `Collection<?> c = new ArrayList<String>();`
 - `c.add(new Object());` **<= chyba při překladu**
- **volat** `get()` lze - výsledek do typu `Object`

Vztahy mezi typy

- ? - wildcard
- „omezený ?“ (bounded wildcard)

```
public abstract class Shape {
    public abstract void draw(Canvas c);
}
public class Circle extends Shape { ... }
public class Canvas {
    public void drawAll(List<Shape> shapes) {
        for (Shape s:shapes) {
            s.draw(this)
        }
    }
}
```

- umožní vykreslit pouze seznamy přesně typu `List<Shape>`, ale už ne `List<Circle>`

Vztahy mezi typy

covariantní

- řešení - omezený ?

```
public void drawAll(List<? extends Shape> shapes) {  
    for (Shape s:shapes) {  
        s.draw(this)  
    }  
}
```

- do tohoto Listu stále nelze přidávat

```
shapes.add(0, new Rectangle()); chyba při překladu
```

Generické metody

```
static void fromArrayToCollection(Object[] a,  
    Collection<?> c) {  
    for (Object o : a) {  
        c.add(o);    ← chyba při překladu  
    }  
}
```

```
static <T> void fromArrayToCollection(T[] a,  
    Collection<T> c) {  
    for (T o : a) {  
        c.add(o);    ← OK  
    }  
}
```


Generické metody

- použití
 - překladač sám určí typy

```
Object[] oa = new Object[100];
Collection<Object> co = new ArrayList<Object>();
toArray(oa, co); // T → Object
String[] sa = new String[100];
Collection<String> cs = new ArrayList<String>();
toArray(sa, cs); // T → String
toArray(sa, co); // T → Object
```

- i u metod lze použít omezený typ

```
class Collections {
    public static <T> void copy(List<T> dest, List<?
    extends T> src) {...}
}
```

Odvození typu

- ne vždy je kompilátor schopen typ určit

- příklad

```
class Collections {  
    static <T> List<T> emptyList();  
    ...  
}
```

- `List<String> listOne = Collections.emptyList();`

- OK

- `void processStringList(List<String> s) {`

```
    ..  
}
```

- `processStringList(Collections.emptyList());`

- nelze přeložit (v Java 7)

Odvození typu

- překladači lze „napovědět“
 - `processStringList(Collections.<String>emptyList());`
- od Java 8 příklad lze přeložit i bez „nápořvědy“
 - zlepšeno odvozování typů

Generické metody a ?

- kdy použít generické metody a kdy "wildcards"

```
interface Collection<E> {  
    public boolean containsAll(Collection<?> c);  
    public boolean addAll(Collection<? extends E> c);  
}
```

```
interface Collection<E> {  
    public <T> boolean containsAll(Collection<T> c);  
    public <T extends E> boolean addAll(Collection<T>  
    c);  
}
```

- Co je vhodnější?

Generické metody a ?

- kdy použít generické metody a kdy "wildcards"

```
interface Collection<E> {  
    public boolean containsAll(Collection<?> c);  
    public boolean addAll(Collection<? extends E> c);  
}
```

```
interface Collection<E> {  
    public <T> boolean containsAll(Collection<T> c);  
    public <T extends E> boolean addAll(Collection<T>  
    c);  
}
```

- generické metody – vztahy mezi více typy

Generické metody a ?

- lze použít generické metody i "wildcards" najednou

```
class Collections {  
    public static <T> void copy(List<T> dest,  
                               List<? extends T> src) {.....}  
}
```

- lze napsat i jako

```
class Collections {  
    public static <T, S extends T>  
        void copy(List<T> dest, List<S> src) {.....}  
}
```

- "správně" je první zápis

Pole a generické typy

- pole gen. typů
 - lze deklarovat
 - nelze naalokovat

```
List<String>[] lsa = new List<String>[10]; nelze!  
List<?>[] lsa = new List<?>[10]; OK + varování
```

- proč - pole lze přetypovat na Object

```
List<String>[] lsa = new List<String>[10];  
Object[] oa = lsa;  
List<Integer> li = new ArrayList<Integer>();  
li.add(new Integer(3));  
oa[1] = li;  
String s = lsa[1].get(0); ClassCastException
```

„Starý“ a „nový“ kód

- „starý“ kód bez generických typů

```
public class Foo {  
    public void add(List lst) { ... }  
    public List get() { ... }  
}
```

- „nový“ kód používající „starý“

```
List<String> lst1 = new ArrayList<String>();  
Foo o = new Foo();  
o.add(lst1); ← OK - List odpovídá List<?>  
List<String> lst2 = o.get(); ← varování překladače
```


„Starý“ a „nový“ kód

- „nový“ kód s generickými typy

```
public class Foo {  
    public void add(List<String> lst) { ... }  
    public List<String> get() { ... }  
}
```

- „starý“ kód používající „nový“

```
List lst1 = new ArrayList();  
Foo o = new Foo();  
o.add(lst1); ← varování překladače  
List lst2 = o.get(); ← OK - List odpovídá List<?>
```

"Erasure"

```
public String loophole(Integer x) {  
    List<String> ys = new LinkedList<String>();  
    List xs = ys;  
    xs.add(x);           ← varování  
    return ys.iterator().next();  
}
```

- při běhu se tento kód chová jako

```
public String loophole(Integer x) {  
    List ys = new LinkedList();  
    List xs = ys;  
    xs.add(x);  
    return (String)ys.iterator().next(); ← běhová chyba  
}
```

"Erasure"

- při překladu se vymažou všechny informace o generických typech
 - "erasure"
 - typové parametry se zruší (`List<Integer>` → `List`)
 - typové proměnné jsou nahrazeny nejobecnějším typem
 - přidána přetypování

Kód generických tříd

```
List<String> l1 = new ArrayList<String>();  
List<Integer> l2 = new ArrayList<Integer>();  
System.out.println(l1.getClass() == l2.getClass());
```

- Co vypíše?
 - a) true
 - b) false

Kód generických tříd

```
List<String> l1 = new ArrayList<String>();  
List<Integer> l2 = new ArrayList<Integer>();  
System.out.println(l1.getClass() == l2.getClass());
```

- Co vypíše?
 - a) true
 - b) false

Přetypování, instanceof

```
Collection cs = new ArrayList<String>();  
if (cs instanceof Collection<String>)....
```

- nelze

```
Collection<String> cstr = (Collection<String>) cs;
```

- varování

- za běhu nelze zajistit

```
<T> T badCast(T t, Object o) {return (T) o;}
```

- varování

```
<T> T[] makeArray(T t) {  
    return new T[100];    ← nelze  
}
```

Další vztahy mezi typy

```
class Collections {  
    public static <T> void copy(List<T> dest, List<?  
        extends T> src) {...}  
}
```

- ve skutečnosti

```
class Collections {  
    public static <T> void copy(List<? super T> dest,  
        List<? extends T> src) {...}  
}
```



contravariantní

- do kolekce `<? super T>` lze přidávat!

Další vztahy mezi typy

- super lze použít jen u gen. metod
- nelze u gen. typů

- nic by to nepřinášelo

```
class Foo<T super Number > {  
    private T v;  
    public Foo(T t) { v = t; }  
}
```

- po *erasure*

```
class Foo {  
    private Object v;  
    public Foo(Object t) { v = t; }  
}
```

- zajistilo by to pouze, že jako parametr lze použít nadtyp `Number`
 - nezajistilo by to, že v prom. `v` bude vždy jen instance nadtypu `Number`

Převod "starého" kódu na nový

```
interface Comparator<T>
    int compare(T fst, T snd);
}
```

```
class TreeSet<E> {
    TreeSet(Comparator<E> c)
    ...
}
```

- `TreeSet<String>`
 - lze použít `Comparator<String>` i `Comparator<Object>`

```
→ class TreeSet<E> {
    TreeSet(Comparator<? super E> c)
    ...
}
```

Převod "starého" kódu na nový

```
Collections {  
    public static <T extends Comparable<T>>  
        T max(Collection<T> coll);  
}
```

```
class Foo implements Comparable<Object> {...}  
Collection<Foo> cf = ...  
Collections.max(cf) nefunguje
```

- lépe

```
public static <T extends Comparable<? super T>>  
    T max(Collection<T> coll);
```

Převod "starého" kódu na nový

```
public static <T extends Comparable<? super T>>  
    T max(Collection<T> coll);
```

- **erasure**

- public static Comparable max(Collection coll)
- není kompatibilní se "starou" metodou max
 - public static Object max(Collection coll)

- **správně**

```
public static <T extends Object & Comparable<? super  
T>> T max(Collection<T> coll);
```

- lze několik typů: T1 & T2 & ... & Tn
- pro "erasure" se vezme první z nich

- **zcela správně**

```
public static <T extends Object & Comparable<? super  
T>> T max(Collection<? extends T> coll);
```

Java

Anotace

Přehled

- anotace ~ metadata
 - „data o datech“
 - přídavná informace o části programu, která nemá (přímo) vliv na funkčnost programu
- od JDK 5
- příklady
 - @Deprecated
 - @SuppressWarnings
 - @Override

Motivace pro zavedení anotací

- anotace v podstatě existovaly už před JDK 5
 - ale byly definovány nesystematicky a
 - nešly přidávat (snadno)
 - př:
 - modifikátor **transient**
 - `@deprecated` element v javadoc komentáři
 - ...
- XDoclet
 - <http://xdoclet.sourceforge.net/>
 - přidávání anotací do „staré“ Javy
 - jako definovatelné tagy v javadoc komentářích
 - lze z nich generovat cokoliv
 - obsahuje mnoho předdefinovaných tagů a transformací
 - původně nástroj pro podporu vývoje EJB komponent

Používání

- anotace lze použít v podstatě na každý element programu
 - třídy
 - interfacy
 - atributy
 - metody
 - konstruktory
 - balíky
 - použití typů (od Java 8)
- obecné pravidlo
anotaci lze použít tam, kde lze použít nějaký modifikátor
 - výjimka u anotací pro balíky (zapisují se do spec. souboru `package-info.java`) a
 - použití typů
- při definici anotace lze omezit, na co půjde použít

Používání

- př:

```
class A {  
    @Override public boolean equals(A a) { ... }  
    @Deprecated public void myDeprecatedMethod() {  
        ...  
    }  
}
```

- anotace můžou mít parametry

```
@SuppressWarnings("unchecked") public void foo() {
```


Používání

- anotace použití typů (Java 8)
 - `new @Interned MyObject();`
 - `myString = (@NonNull String) str;`
 - `class UnmodifiableList<T> implements @ReadOnly List<@ReadOnly T> { ... }`
 - `void monitorTemperature() throws @Critical TemperatureException { ... }`

Používání

- lze použít mezi modifikátory v jakémkoliv pořadí
 - konvence – použít nejdřív anotace a pak modifikátory
- lze aplikovat libovolné množství anotací na jeden element
- Java 5-7 – nelze aplikovat jednu anotaci vícekrát
 - ani při použití různých parametrů
- Java 8+ lze aplikovat jednu anotaci vícekrát

Definice

- podobně jako interface
 - @interface
 - metody bez implementace
- př.

```
public @interface RequestForEnhancement {  
    int id();  
    String synopsis();  
    String engineer() default "[unassigned]";  
    String date() default "[unimplemented]";  
}
```

Definice

- „speciální“ anotace
- marker
 - bez těla
 - anotace při použití nemá žádné parametry
 - `public @interface Preliminary { }`
- single-member
 - jeden element **value**
 - typ může být různý
 - při použití se zapíše jen anotace a hodnota parametru
 - `public @interface Copyright { String value(); }`

Definice

- použití našich anotací

```
@RequestForEnhancement(  
    id          = 2868724,  
    synopsis   = "Enable time-travel",  
    engineer    = "Mr. Peabody",  
    date       = "4/1/3007"  
)  
public static void travelThroughTime(Date destination)  
{ ... }  
  
@Preliminary public class TimeTravel { ... }  
  
@Copyright("2002 Yoyodyne Propulsion Systems")  
public class OscillationOverthruster { ... }
```

Definice

- stejné jako interface
 - místo deklarace
 - rozsah platnosti
 - rozsah viditelnosti
- nesmějí být generickým typem
- nesmějí obsahovat extends
 - implicitně dědí od **java.lang.annotation.Annotation**
- libovolný počet elementů
- anotace T nesmí obsahovat element typu T
 - přímo i nepřímo

```
@interface SelfRef { SelfRef value(); }  
@interface Ping { Pong value(); }  
@interface Pong { Ping value(); }
```

Definice

- metody nesmějí mít žádné parametry
- metody nesmějí být generické
- deklarace metod nesmí obsahovat throws
- návratová hodnota musí být:
 - primitivní typ
 - String
 - Class
 - Enum
 - anotace
 - pole předchozích typů

Definice

- při použití musí anotace obsahovat dvojici element-hodnota pro každý element (metodu) z definice
 - neplatí pro elementy s defaultní hodnotou
- hodnota nesmí být **null**

Předdefinované anotace

- anotace pro použití na anotacích
 - ovlivňují možnosti použití anotace
 - v balíku `java.lang.annotation`
- **@Target**
 - single-member
 - specifikuje, na jakém elementu lze anotaci použít
 - možné hodnoty parametru (enum `ElementType`)
 - `ANNOTATION_TYPE`
 - `CONSTRUCTOR`
 - `FIELD`
 - `LOCAL_VARIABLE`
 - `PACKAGE`
 - `METHOD`
 - `PARAMETER`
 - `TYPE` – použití na třídu, interface, enum, anotaci
 - `TYPE_PARAMETER` – od Java 8
 - `TYPE_USE` – od Java 8
 - `MODULE` – od Java 9

Předdefinované anotace

- **@Retention**
 - single-member
 - specifikuje, kde je anotace použitelná
 - možné hodnoty parametru (enum RetentionPolicy)
 - SOURCE – pouze ve zdrojovém kódu
 - CLASS – v době kompilace
 - RUNTIME – při běhu programu

```
@Retention(RetentionPolicy.RUNTIME)  
public @interface Foo { }
```

Opakované použití anotace

- od Java 8

```
@Schedule (dayOfMonth="last")
@Schedule (dayOfWeek="Fri", hour="23")
public void foo() { ... }
```

- z důvodů kompatibility anotace vloženy automaticky do „kontejneru“
 - kontejner se musí připravit

```
@Repeatable (Schedules.class)
public @interface Schedule { ... }

public @interface Schedules {
    Schedule[] value;
}
```

Získání anotací za běhu

- pomocí Reflection API
- **interface** `AnnotatedElement`
 - `isAnnotationPresent` – zda je anotace přítomna
 - `getAnnotation` – vrátí anotaci daného typu, je-li přítomna
 - `getAnnotations` – vrátí všechny anotace
 - `getDeclaredAnnotations` – vrátí deklarované anotace (bez zděděných)

Zpracování SOURCE anotací

- anotační procesory
 - specifikují se překladači
 - parametr `-processor`
 - `javax.annotation.processing.Processor`
 - od Java 6
- Annotation Processing Tool (APT)
 - externí nástroj pro zpracování anotací
 - Java 5
 - od JDK 8 – APT a související API označeno za „deprecated“

Příklad – Unit Testing

```
import java.lang.annotation.*;  
@Retention(RetentionPolicy.RUNTIME)  
@Target(ElementType.METHOD)  
public @interface Test { }
```

```
public class Foo {  
    @Test public static void m1() {  
        ...  
    }  
    public static void m2() {  
        ...  
    }  
    @Test public static void m3() {  
        ...  
    }  
}
```

Příklad – Unit Testing

```
import java.lang.reflect.*;
public class RunTests {
    public static void main(String[] args)
        throws Exception {
        int passed = 0, failed = 0;
        for (Method m :
            Class.forName(args[0]).getMethods()) {
            if (m.isAnnotationPresent(Test.class)) {
                try {
                    m.invoke(null);
                    passed++;
                } catch (Throwable ex) {
                    System.out.printf("Test %s
                        failed: %s %n", m, ex.getCause());
                    failed++;
                }
            }
        }
        System.out.printf("Passed: %d,
            Failed %d%n", passed, failed);
    }
}
```



Verze prezentace AJ02.cz.2020.01

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