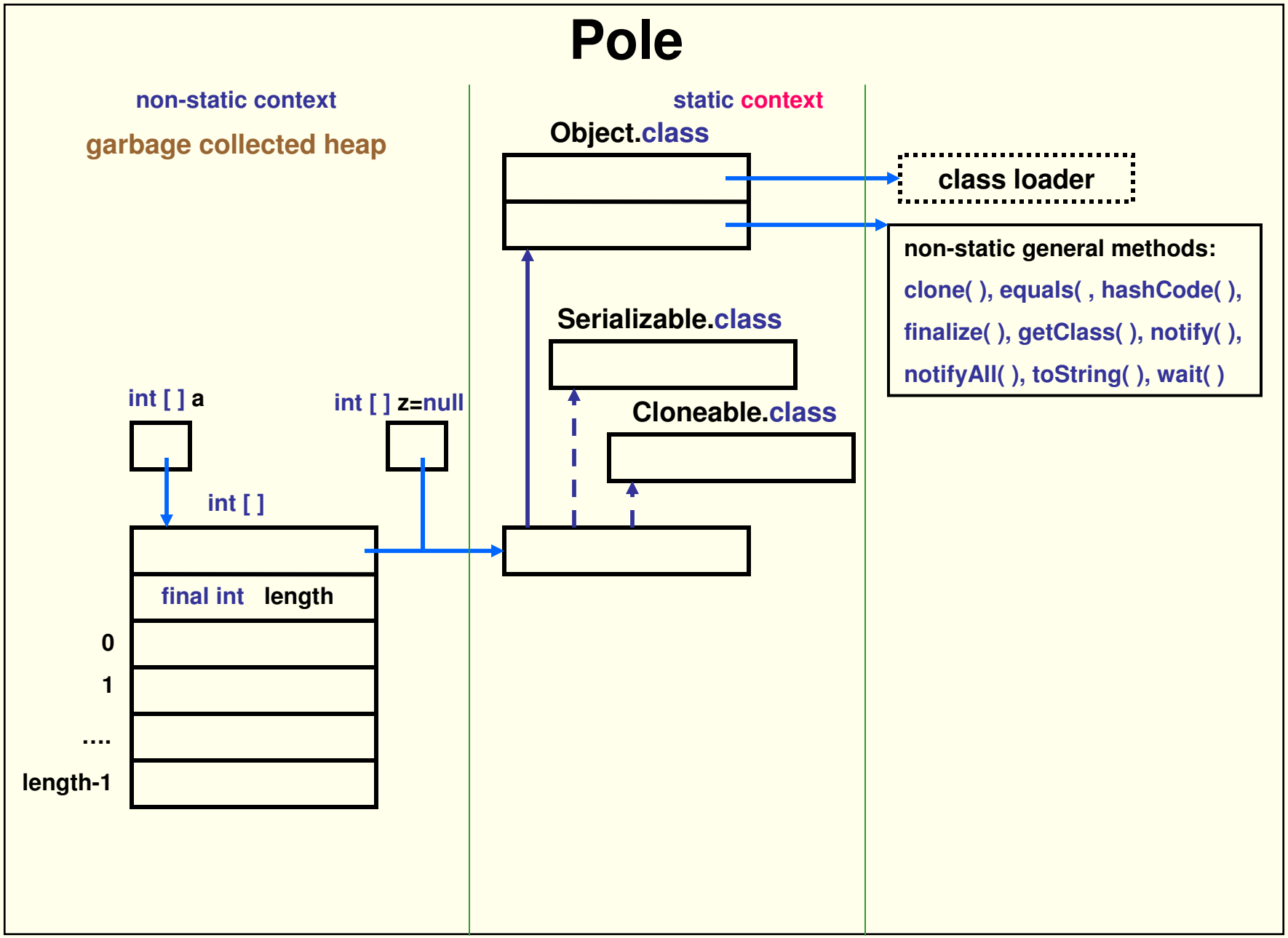
Pole (Array)

- Pole je také objekt mající typ a tudíž se konstruuje na haldě.
  Typy prvků pole musí být kompatibilní s typem pole. Mohou to být primitivy anebo reference – dle toho se defaultně inicializují na příslušné nuly či false či null. Každé pole lze referovat typem java.lang.Object.
- Neměnnou délku pole určuje jeho atribut public final int length >=0.
- Pole a prvky pole se poznají právě dle hranatých závorek vyjadřujících dimensionalitu pole. Dimensionalita má rozsah 1 … 255.
- Prvky pole jsou přistupné pomocí indexu v mezích 0 ... pole.length-1, jinak vznikne vyjímka java.lang.ArrayIndexOutOfBoundsException.
- Příklady jednorozměrných polí:

```
int[ ] a = { 1, 2, 3 };
int[ ] b = new int[ 10 ];                                  // initialized by 0
String[ ] s = { "A", "BB", "CCC" };                        // homogen.
Object[ ] p0 = new Point[ 5 ];                             // initialized by null
Object[ ] p1 = { new Long(1), "CCC", new Short(0) };       // heterogen.
Point[ ] p2 =  { new Point(1, 2), new Point(3, 4), };
int len = p2.length;                                       // get array length
p2[1] = new Point(5, 6);                                   // assignment
```

Pole


non-static context
garbage collected heap
static context
Object.class
class loader
non-static general methods:
clone( ), equals(, hashCode( ),
finalize( ), getClass( ), notify( ),
notifyAll( ), toString( ), wait( )
Serializable.class
Cloneable.class
int [ ] a
int [ ] z=null
int [ ]
final int length
0
1
....
length-1

java.util.Arrays

Tato třída poskytuje řadu statických metod usnadňujících práci s jednorozměrnými poli. Metody jsou zpravidla přetížené pro různé typy.

- **asLlst – pro převod do kolekce**
- **binarySearch – hledání v seřazeném poli či jeho části**
- **copyOf, copyOfRange – kopie pole dle zadané délky a i s přetypováním**
- **deepEquals**
- **deepHashCode**
- **deepToString - výpis pole i vícedimezionálního**
- **equals – test ekvivalence**
- **fill – vyplnění všech položek danou hodnotou**
- **hashCode – výpočet pro pole**
- **sort – vzestupné řazení, případně dle zadaného komparátoru**
- **toString – výpis pole**

Vícedimensionální pole

- Tvoří se skládáním jednodimensionálních - a nemusejí být rektangulární.
- Příklady:

```
int[ ][ ] aa = new int[10][5];            // rectangular matrix 10 rows 5 cols
int[ ][ ] bb = { {1}, {2, 3}, {4, 5, 6} }; // triangle matrix with 3 rows
int len = bb[1].length;                   // length of second row is 2
bb[2][1] = 9;                             // assign 9 (old value was 5)
bb[1] = null;                             // second row does not exist
Point[ ][ ] pp = { { new Point(0,0), new Point(1,1), null } ,
                   { new Point(2,2) },
                   null,
                 };
```

- RTTI umožňuje určit typ pole a jeho dimensionalitu dle přiřazeného písmena a počtu otvíracích hranatých závorek takto:
  - pro primitivní pole: B, S, C, I, J, F, D, Z
    pro byte, short, char, int, long, float, double, boolean
  - pro referenční pole: L a úplným jménem třídy.

Pole (Array)

Pomocí metody getClass( ) lze zjistit vlastnosti pole.

Rafinované triky nabízí třída java.lang.reflect.Array, která umožňuje dynamicky vytvářet a modifikovat pole.

Příklad:

```
String[ ] s = { "AAA",  " BBB",  "CCC" };
System.out.println( s.getClass( )+"   "+Array.getLength(s) );
int [ ][ ] a = new int [ ][ ] { { 333, 666 }, { 777 } };
System.out.println( a.getClass( )+"   "+Array.getLength(a) );
```

Vypíše:

```
class  [ L java.lang.String;    3
class  [ [ I    2
```

Velmi užitečná Třída java.util.Arrays umožňuje třídění, vyhledávání, vyplňování a porovnávání polí. Navazuje také na operace s kolekcemi.

Kolekce

Je sbírka objektů v operační paměti, organizovaná dle JCF (Java Collection Framework). JCF sídlí v balíčku java.util a zahrnuje zejména:

- **Interfejsy:**
  - **Collection<E> - sbírka objektů**
  - **Set<E>, SortedSet <E> - sbírka unikátních objektů – příp. uspořádaná**
  - **Map<K,V>, SortedMap <K,V> - sbírka dvojic (zobrazení) objektů**
    key → value - příp. uspořádaná
  - **List<E> - indexovaná sbírka objektů**
  - **Queue<E> - fronta (FIFO)**
  - **Comparator<T>, java.lang.Comparable - pro porovnávání objektů**
  - **Iterator<E>, ListIterator<E>, RandomAccess - pro přístup k prvkům**

- **Abstraktní třídy:**
  - **AbstractCollection**
  - **AbstractSet, AbstractMap, AbstractList, AbstractSequentialList**

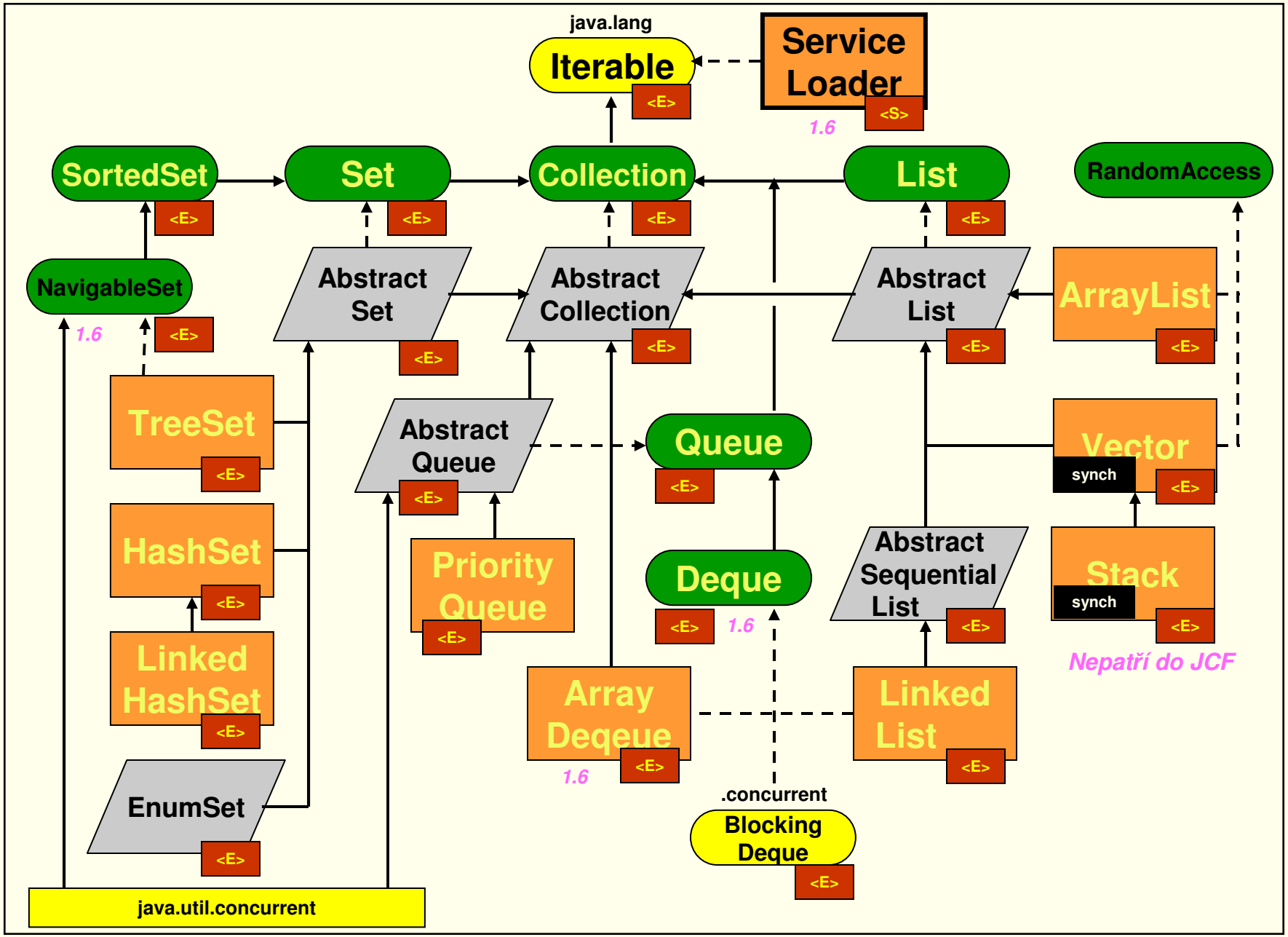
Kolekce

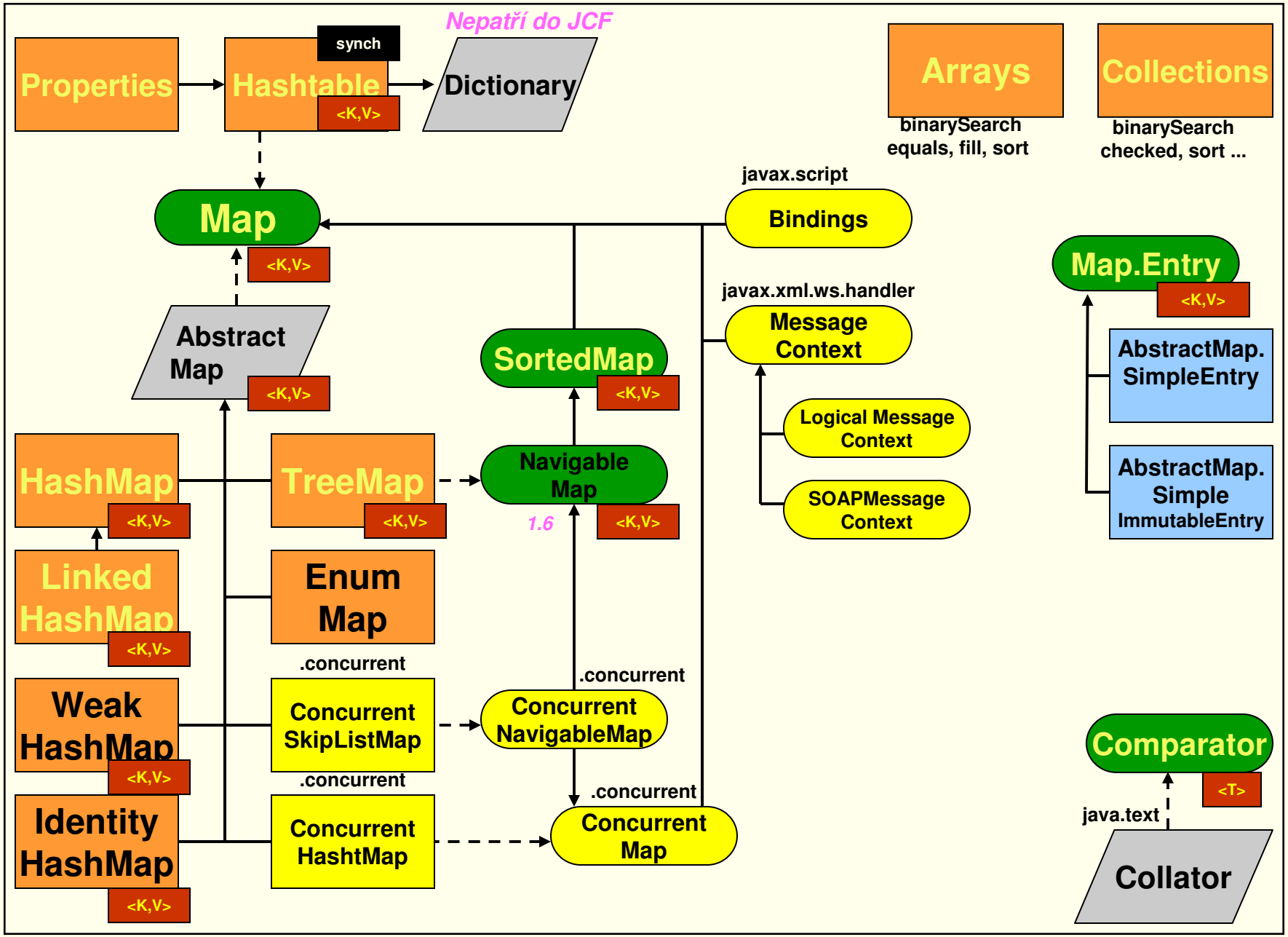
- **Třídy:**

  ArrayList, LinkedList
  HashSet, LinkedHashSet,
  HashMap, LinkedHashMap
  TreeSet, TreeMap
  Vector, Stack, Hashtable, Properties
  PriorityQueue, ArrayDeque
  Arrays, Collections

Algoritmy pro řazení, přesouvání, doplňování, kopírování a vyhledávání

Třídy *Hash* vyžadují vhodnou konstrukci funkce hashCode( ).
Ta původní z třídy Objekt dává vždy různé kódy – proto je nevhodná.


java.lang
Iterable
<E>
Service Loader
<S>
1.6
SortedSet
<E>
Set
<E>
Collection
<E>
List
<E>
RandomAccess
NavigableSet
1.6
<E>
Abstract Set
<E>
Abstract Collection
<E>
Abstract List
<E>
ArrayList
<E>
TreeSet
<E>
Abstract Queue
<E>
Queue
<E>
Vector
synch
<E>
HashSet
<E>
Priority Queue
<E>
Deque
<E>
1.6
Abstract Sequential List
<E>
Stack
synch
<E>
Nepatří do JCF
Linked HashSet
<E>
Array Deqeue
1.6
<E>
Linked List
<E>
EnumSet
<E>
.concurrent
Blocking Deque
<E>
java.util.concurrent


Nepatří do JCF
Properties
Hashtable
synch
<K,V>
Dictionary
Arrays
binarySearch
equals, fill, sort
Collections
binarySearch
checked, sort ...
javax.script
Bindings
Map
<K,V>
Abstract Map
<K,V>
SortedMap
<K,V>
javax.xml.ws.handler
Message Context
Logical Message Context
SOAPMessage Context
Map.Entry
<K,V>
AbstractMap. SimpleEntry
AbstractMap. Simple ImmutableEntry
HashMap
<K,V>
TreeMap
<K,V>
Navigable Map
1.6
<K,V>
Linked HashMap
<K,V>
Enum Map
.concurrent
Concurrent SkipListMap
.concurrent
Concurrent NavigableMap
Weak HashMap
<K,V>
.concurrent
Concurrent HashtMap
.concurrent
Concurrent Map
Identity HashMap
<K,V>
Comparator
<T>
java.text
Collator

	Užité techniky				
interface	***HT***	***RA***	***BT***	***LL***	***HT+LL***
List	**-**	**ArrayList**	**-**	**LinkedList**	**-**
Set	**HashSet**	**-**	**-**	**-**	**LinkedHashSet**
SortedSet	**-**	**-**	**TreeSet**	**-**	**-**
Map	**HashMap**	**-**	**-**	**-**	**LinkedHashMap**
SortedMap	**-**	**-**	**TreeMap**	**-**	**-**

- ***HT - hashtable - rozmítaná tabulka,***
- ***RA - resizable array - pole s proměnnou velikostí,***
- ***BT - balanced tree - vyvážený strom,***
- ***LL - linked list - spojový seznam.***

Interfejs Collection<E> extends Iterable<E>

Interfejs Collection<E> je základem seznamů (list) a množin (set).

Umožňuje jednotlivé i hromadné operace pro dotazy, převody a modifikace.

Metody pro plnění kolekce:

boolean add(E o) – vložení jednoho prvku

boolean addAll(Collection< ? extends E > c) – vložení všech prvků nacházejících se v jiné kolekci

Metody pro ubírání z kolekce:

void clear( ) – vyloučení všech prvků z kolekce

boolean remove(Object o) – odstranění jednoho prvku z kolekce

boolean removeAll(Collection <?> c) - odstraní všechny patřící do c

boolean retainAll(Collection <?> c) - odstraní všechny mimo patřící do c

Dynamické vlastnosti kolekcí:

int size( ) – vrátí aktuální počet prvků kolekce

boolean isEmpty( ) – test na prázdnou kolekci

boolean contains(Object o) – test, zda je obsažen prvek o

boolean containsAll(Collection <?> c) - true, když obsahuje všechny prvky c

Interfejs Collection<E> extends Iterable<E>

Získání přístupového objektu:

Iterator<E> iterator( ) – vrátí objekt pro probírku sbírky

Převod kolekce na běžné pole:

Object[ ] toArray( ) – převod na pole typu Object

<T> T [ ] toArray(T [ ] a) - vytvoří pole konkrétního typu

Collection

- **Třída AbstractCollection implements Collection, abstraktními zůstávají pouze metody iterator( ) a size( ).**
- **Interfejs Iterator slouží k postupnému jednosměrnému výpisu sbírky v pořadí v závislosti na jejím typu.**
- **Interfejs ListIterator extends Iterator, navíc má metody pro prohlížení oběma směry, získávání indexů a vkládání prvků.**
- **Interfejs RandomAccess je pouhý marker.**

Interfejs List <E> extends Collection<E>

Vyžaduje indexovaný přístup k prvkům sbírky. Přidává metody, které zavádějí možnost práce s prvky kolekce pomocí indexů.

Změny v kolekci:

- `void` add (int index, E element) – přidání prvku (prvky se stejným a vyššími indexy budou posunuty o jeden výše)
- `boolean` addAll (Collection < ? extends E > c) – přidá prvky z c
- `boolean` addAll (int index, Collection < ? extends E > c)
- `E` set (int index, E element) – změna prvku na daném indexu
- `E` remove (int index) – odstranění prvku; prvky s vyšším indexem budou posunuty o jeden níže

Získání obsahu kolekce:

- `E` get (int index) – vrátí prvek s daným indexem
- `int` indexOf (Object o) - hledání zepředu
- `int` lastIndexOf (Object o) - hledání zezadu
- `List<E>` subList (int fromIndex, int toIndex) - vrátí podseznam

Interfejs List <E> extends Collection<E>

Získání přístupového objektu:

ListIterator<E> listIterator ( )

ListIterator<E> listIterator (int index)

Ostatní:

boolean equals (Object o) – porovnání seznamů

Třída AbstractList

extends AbstractCollection a implements List.

Přidává jen metodu protected void removeRange(int, int) a abstraktní

má jen metody get(int) a size( ).

AbstractList sleduje také modifikace a komodifikace.

Má tři přímé potomky:

- **AbstractSequentialList - polotovar pro sekvenční přístup s abstraktními metodami listIterator(int) a size( ).**

- **ArrayList - nesynchronizovaná obdoba Vectoru.**

- **Vector je stará, synchronizovaná třída - od verze 1.2 redefinovaná. Některé původní metody mají nově, systematičtěji nazvané ekvivalenty, např.: elementAt(int) má nový ekvivalent get(int).**

  Radno používat ty, které jsou zavedené v příslušných interfejsech.

  Při probírce lépe použít Iterator iterator( ) než Enumeration elements( ).

ArrayList<E> a LinkedList<E>

- Třída ArrayList uchovává reference objektů v poli.

  Operace size, isEmpty, get, set, add, iterator, a listIterator probíhají v konstantním čase, ostatní v lineárním.
  Konstruktor a dvě metody nastavují kapacitu tj. velikost pole a tím ovlivňují nárok na paměť i četnost a dobu relokací.

  ArrayList(int pocatecniKapacita) přednastaví seznam na pož. kapacitu

  ensureCapacity( ) realokuje pole o polovinu větší.

  trimToSize( ) redukuje rozsah na stávající velikost.

- Třída LinkedList implements Queue, Deque

  Kolekce objektů je realizována technikou obousměrného spojového seznamu ve kterém se postupně hledá pomocí ListIteratoru.
  Příslušné operace mají tedy lineární časovou složitost.

Pro synchronizaci je třeba zapouzdřit takto:

```
List list = Collections.synchronizedList( new ArrayList( ) );
List list = Collections.synchronizedList( new LinkedList( ) );
```

Třída Stack<E> extends Vector<E>

Zásobník přidává metody:

- E push (E item) - vloží prvek na vrchol.
- E pop ( ) - odebere prvek z vrcholu.
- E peek ( ) - vrátí prvek z vrcholu, ale neodebere.
- boolean empty ( )
- int search (Object x) - pozice objektu od vrcholu počítaná od 1

Tato stará, synchronizovaná třída je užitečná - ač nepatří do JCF a tedy její vlastní metody nelze využít interfejsem List.

Od verze 1.6 se doporučuje používat třídy splňující interfejs Deque, tj.: ArrayDeque, LinkedList, LinkedBlockingDeque.

Interfejsy Set<E>, SortedSet<E>, NavigableSet<E>

- Set extends Collection

  V množině nesmějí být duplikované prvky (podle equals()) a nanejvýš jedna reference null. Zvláštní pozornost zasluhuje případ jsou-li prvky obsahově proměnné. Tento interfejs nepřidává další metody.

- SortedSet extends Set a přidává metody:

  Comparator<? super E> comparator( )
  E first( ) - vrací první (nejmenší) prvek
  SortedSet<E> headSet(E toElement) - vrací pohled
  E last( ) - vrací poslední (největší) prvek
  SortedSet<E> subSet(E fromElement, E toElement)
  SortedSet<E> tailSet(E fromElement) - vrací pohled

- NavigableSet extends SortedSet a přidává metody:

  ceiling( ), floor( ), higher( ), lower( )
  descendingIterator( ), descendingSet( )
  pollFirst( ), pollLast( )

Třída AbstractSet a její potomci

- **AbstractSet přepisuje pouze metody equals( ), hashCode( ), removeAll( ). Abstraktními zůstávají iterator( ) a size ( ).**

  Všichni následující konkrétní potomci jsou nesynchronizovaní.

- **Třída HashSet umožňuje prvek null. Iterátor nezaručuje pořadí.**

  Operace jsou zprostředkovány třídou HashMap v O(1) čase.

- **Třída LinkedHashSet extends HashSet a také umožňuje prvek null. Iterátor zaručuje výstup v pořadí vkládání i v případě reinserce téhož prvku. Operace jsou zprostředkovány třídou LinkedHashMap v O(1) čase.**

- **Třída TreeSet navíc implementuje interfejs SortedSet.**

  Iterátor zaručuje výstup v pořadí objektů buď dle Comparable anebo dodaného Comparatoru.

  Operace jsou zprostředkovány třídou TreeMap v čase O(log n) .

Interfejsy pro fronty Queue<E> a Deque<E>

- Interfejs Queue extends Collection.
  Přidává obdobné metody s mírně jiným chováním:

	vyhazují vyjímky	vracejí spec. hodnotu
vložení	boolean add(E e)	boolean offer(E e)
odstranění	E remove( )	E poll( )
přístoupení	E element( )	E peek( )

- Interfejs Deqeue extends Queue.
  Přidává "zásobníkové" metody:

E pop( )
void push(E e)

E getFirst( )	void addFirst(E e)	E peekFirst( )
E getLast( )	void addLast(E e)	E peekLast( )

Třída PriorityQueue<E>

- PriorityQueue extends AbstractQueue.

  Elementy jsou ve frontě dynamicky řazeny buď podle přirozeného pořadí anebo podle dodaného komparátoru.

```
boolean  add( E e )
boolean  addAll( Collection<? extends E> e )
boolean  contains( Object o)
boolean  offer( E e )
E  remove( )
boolean  remove( Object o )
E  poll( )
E  peek( )
E  element( )
void  clear( E e )
```

Interfejs Map < K, V >

Map znamená zobrazení keys -> values, tedy klíčů (množinu objektů) na hodnoty (multimnožinu objektů) a definuje metody:

```
void          clear ( )
boolean       containsKey ( Object key )
boolean       containsValue ( Object value )
Set <Map.Entry<K,V>>   entrySet ( )
V             get ( Object key )
boolean       isEmpty ( )
Set<K>        keySet ( )
V             put ( K key, V value )
void          putAll ( Map< ? extends K, ? extends V >  m )
Object        remove ( Object key )
int           size ( )
Collection values ( )
```

Interfejsy SortedMap<K,V> a NavigableMap<K,V>

- SortedMap extends Map - přidává metody pro uspořádané zobrazení:
  Comparator<? super K> comparator( )
  K firstKey( ) - vrací první (nejmenší) klíč
  SortedMap<K,V> headMap(Object toKey) - vrací pohled
  K lastKey( ) - vrací poslední (největší) prvek
  SortedMap<K,V> subMap(K fromKey, K toKey) - vrací pohled
  SortedMap<K,V> tailMap(K fromKey) - vrací pohled

- NavigableMap extends SortedMap - přidává metody pro lepší navigaci:
  ceilingEntry( ), floorEntry( ), higherEntry( ), lowerEntry( )
  ceilingKey( ), floorKey( ), higherKey( ), lowerKey( )
  firstEntry( ), lastEntry( )
  descendingKeySet( ), descendingMap( ), navigableKeySet( )
  pollFirstEntry( ), pollLastEntry( )

Třída AbstractMap

implements interfejs Map, ponechává abstraktní jen metodu keySet ( ).

Přidává metody:

```
boolean   containsKey  ( Object x )
boolean   containsValue ( Object x )
```

- **Konkrétní třída HashMap extends AbstractMap, umožňuje null reference, není synchronizovaná.**
- **Třída Hashtable je starší obdobou HashMap, splňuje interfejs Map, je však potomkem zastaralé abstraktní třídy Dictionary, neumožňuje null reference, je synchronizovaná.**
- **Konkrétní třída Properties je potomkem Hashtable, která umožňuje vstup i výstup pomocí proudů (streams).**
- **Konkrétní třída TreeMap extends AbstractMap implements SortedMap. Udržuje unikátní klíče v pořadí objektů buď dle Comparable anebo dodaného Comparatoru.**

Interfejs Comparator < T >

definuje požadavky na objekt rozhodující, který ze dvou objektů je větší.
Jsou to metody:

int compare(T x, T y) - vrací <0, 0, >0 pro x<y, x==y, x>y

boolean equals (Object obj) - srovnává objekty typu Comparator

Příklad:

```
public class PriceComparator implements Comparator {
    int compare( Object x, Object y ) {
        Priceable a = (Priceable) x, b = (Priceable) y;
        float diff = a.getPrice( ) - b.getPrice( );
        if ( diff > 0 ) return  1;
        if ( diff < 0 ) return -1;
        return 0;
    }
}
```

Interfejsy Iterator<E> a ListIterator<E>

- Iterator požaduje metody pro objekt, kterým lze probírat sbírku:

boolean	hasNext ( ) - testuje zda jsou ještě další prvky
E	next( ) - podá další prvek
void	remove( ) – odstraní prvek ze sbírky

- ListIterator extends Iterator s dodatečnými požadavky pro sekvenční přístup, přidávání, změny a indexování prvků metodami:

boolean	hasNext ( ) - testuje zda jsou ještě další prvky
E	previous( ) - podá předchozí prvek
int	nextIndex( ) – vrátí index dalšího prvku
int	previousIndex( ) – vrátí index předchozího prvku
void	set(E e)
void	add(E e)

Modifikace sbírky při užívání iteratorů vede k ConcurrentModificationExc.

Příklad: frekvenční statistika (1/3)

Definice dosti obecné položky s čítačem výskytů:

```
public class Item {
    private  String key;
    private  int count = 1;
    public   Item( String key ) { this.key = key; }

    public   String  getKey( ) { return key; }
    public   int  getCount( ) { return count; }
    public   int  add( ) { return ++count; }
    public   int  add( int any ) { return count += any; }

    public String toString( ) {  return "Item  " + key + ":  " + count;  }
}
```

Příklad: frekvenční statistika (2/3)

```
static Map <String, Item> map = new TreeMap <String, Item> ( );
...
    enter("A");  enter("A");  enter("BB");

    Set ks = map.keySet( );
    System.out.println( ks );
    System.out.println( map );
    Item item = (Item) map.get("A");
...

public static void enter( String key ) {
    if ( map.containsKey( key ) ) {  ( (Item) map.get( key ) ).add( ); }
    else map.put( key, new Item( key ) );
}
```

Zadáním parametrických typů není třeba přetypovávat.

Příklad: frekvenční statistika (3/3)

```
static Map <String, Item> map = new TreeMap <String, Item> ( );

                    //    Výpis relativních četností

int total = 0;

for ( Item it : map.values( ) ) total += it.getCount( );

for ( Map.Entry< String, Item> e : map.entrySet( ) )
    System.out.println(
        e.getKey( )+"  "+ (double) e.getValue( ).getCount( ) / total
);
```

Zadáním parametrických typů není třeba přetypovávat.