



Architecture of software systems

Course 11: Data structures, memory management with garbage collector

David Šišlák

david.sislak@fel.cvut.cz



- » primitives: boolean, byte, char, int, long, float, double
 - » without implicit allocation
 - » placed in frame in variables or operand stack
- » objects
 - » every object is descendant of Object by default
 - » methods – clone(), equals, getClass(), hashCode(), wait(...), notify(...), finalize()
 - » objects for primitives: Boolean, Byte, Char, Integer, Long, Float, Double
can be **null**
 - » other objects
- » arrays
 - » special data structure which store a number of items of the same type in linear order; have the defined limit
 - » JAVA automatically check limitations
 - » allocated on the heap
 - » multi-dimensional arrays = arrays of arrays; ragged array



- » automatic conversion from primitive to object representation and vice versa
- » since JAVA 5
- » for example
 - » autoboxing for Integer is based on `valueOf(int)` and `intValue()` methods

```
int myInt = 3;  
myInt.toString();
```



- » automatic conversion from primitive to object representation and vice versa
- » since JAVA 5
- » for example
 - » autoboxing for Integer is based on `valueOf(int)` and `intValue()` methods

```
int myInt = 3;  
myInt.toString();
```

- » works only during assignment or parameter passing

```
String a = myInt+"";
```



- » automatic conversion from primitive to object representation and vice versa
- » since JAVA 5
- » for example
 - » autoboxing for Integer is based on `valueOf(int)` and `intValue()` methods

```
int myInt = 3;  
myInt.toString();
```

- » works only during assignment or parameter passing

```
String a = myInt + ""; Integer.toString(myInt);
```

- » example: count word frequency/histogram

```
public static void main(String[] args) {  
    Map<String, Integer> m = new TreeMap<String, Integer>();  
    for (String word : args) {  
        Integer freq = m.get(word);  
        m.put(word, (freq == null ? 1 : freq + 1));  
    }  
    System.out.println(m);  
}
```

- » boxing and un-boxing brings inefficiencies !

Example



```
int i = 2;
int j = 2;
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(i);
list.add(j);
System.out.printf(Boolean.toString(i==j));
System.out.printf(Boolean.toString(list.get(0)==list.get(1)));
System.out.printf(Boolean.toString(list.get(0).equals(list.get(1))));
```

» what is the output? and what is the output for i=2000 and j=2000 ?

Example



```
int i = 2;
int j = 2;
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(i);
list.add(j);
System.out.printf(Boolean.toString(i==j));
System.out.printf(Boolean.toString(list.get(0)==list.get(1)));
System.out.printf(Boolean.toString(list.get(0).equals(list.get(1))));
```

» what is the output? and what is the output for i=2000 and j=2000 ?

true	true
true	false
true	true

» but not after serialization, there is no readResolve !



- » similar concept as in multiton

```
// set from vm option -XX:AutoBoxCacheMax=<size>
private static String integerCacheHighPropValue;

private static class IntegerCache {
    static final int high;
    static final Integer cache[];

    static {
        final int low = -128;

        int h = 127;
        if (integerCacheHighPropValue != null) {
            int i = Long.decode(integerCacheHighPropValue).intValue();
            i = Math.max(i, 127);
            h = Math.min(i, Integer.MAX_VALUE - -low);
        }
        high = h;

        cache = new Integer[(high - low) + 1];
        int j = low;
        for(int k = 0; k < cache.length; k++)
            cache[k] = new Integer(j++);
    }

    private IntegerCache() {}
}
```




```
public static class Integer {
    private final int value;

    public Integer(int value) {
        this.value = value;
    }

    public int intValue() {
        return value;
    }

    public static Integer valueOf(int i) {
        if(i >= -128 && i <= IntegerCache.high)
            return IntegerCache.cache[i + 128];
        else
            return new Integer(i);
    }
}
```

Widening vs. autoboxing



```
public static void hello(Integer x) {
    System.out.println("Integer");
}

public static void hello(long x) {
    System.out.println("long");
}

public static void main(String[] args) {
    int i = 5;
    hello(i);
}
```

```
public static void hello(Integer x) {
    System.out.println("Integer");
}

public static void hello(Long x) {
    System.out.println("long");
}

public static void main(String[] args) {
    int i = 5;
    hello(i);
}
```

» what are the outputs?



```
public static void hello(Integer x) {
    System.out.println("Integer");
}

public static void hello(long x) {
    System.out.println("long");
}

public static void main(String[] args) {
    int i = 5;
    hello(i);
}
```

```
public static void hello(Integer x) {
    System.out.println("Integer");
}

public static void hello(Long x) {
    System.out.println("long");
}

public static void main(String[] args) {
    int i = 5;
    hello(i);
}
```

» what are the outputs?

long

» why?

prefer widening
before autoboxing

Integer

cannot use autoboxing
to widen primitives
-> error if no
hello(Integer) method



```
public static class ShortSet {
    public static void main(String args[]) {
        Set<Short> s = new HashSet<Short>();
        for (short i = 0; i < 100; i++) {
            s.add(i);
            s.remove(i - 1);
        }
        System.out.println(s.size());
    }
}
```

» what is the outputs?



```
public static class ShortSet {
    public static void main(String args[]) {
        Set<Short> s = new HashSet<Short>();
        for (short i = 0; i < 100; i++) {
            s.add(i);
            s.remove(i - 1);
        }
        System.out.println(s.size());
    }
}
```

» what is the outputs?

100 - because we are removing Integers instead of Short !!

» correct:

```
public static class ShortSet {
    public static void main(String args[]) {
        Set<Short> s = new HashSet<Short>();
        for (short i = 0; i < 100; i++) {
            s.add(i);
            s.remove((short) (i - 1));
        }
        System.out.println(s.size());
    }
}
```



- » method is identified by its signature

```
public static void method(Object obj) {
    System.out.println("method with param type - Object");
}

public static void method(String obj) {
    System.out.println("method with param type - String");
}

public static void main(String[] args) {
    method(null);
}
```

- » can be compiled and what is the output?



- » method is identified by its signature

```
public static void method(Object obj) {
    System.out.println("method with param type - Object");
}

public static void method(String obj) {
    System.out.println("method with param type - String");
}

public static void main(String[] args) {
    method(null);
}
```

- » can be compiled and what is the output?
 - **YES** – no ambiguity

method with parameter type – String

- » due to JLS specification:

“The Java programming language uses the rule that the *most specific* method is chosen.”

Method overloading



```
public static void method(Object obj){
    System.out.println("method with param type - Object");
}

public static void method(String str){
    System.out.println("method with param type - String");
}

public static void method(StringBuffer strBuf){
    System.out.println("method with param type - StringBuffer");
}

public static void main(String[] args) {
    method(null);
}
```

» can be compiled and what is the output?

Method overloading



```
public static void method(Object obj){
    System.out.println("method with param type - Object");
}

public static void method(String str){
    System.out.println("method with param type - String");
}

public static void method(StringBuffer strBuf){
    System.out.println("method with param type - StringBuffer");
}

public static void main(String[] args) {
    method(null);
}
```

- » can be compiled and what is the output?
 - » **NO** – cannot find “most specific”, both are sub-classes of Object but not in the same inheritance hierarchy

Method overloading



```
public static void method(Object obj, Object obj1) {
    System.out.println("method with param types - Object, Object");
}

public static void method(String str, Object obj) {
    System.out.println("method with param types - String, Object");
}

public static void main(String[] args) {
    method(null, null);
}
```

» can be compiled and what is the output?



```
public static void method(Object obj, Object obj1) {
    System.out.println("method with param types - Object, Object");
}

public static void method(String str, Object obj) {
    System.out.println("method with param types - String, Object");
}

public static void main(String[] args) {
    method(null, null);
}
```

» can be compiled and what is the output?

- **YES**

method with param types – String, Object



» BUT

```
public static void method(Object obj, String obj1) {  
    System.out.println("method with param types - Object, String");  
}  
  
public static void method(String str, Object obj) {  
    System.out.println("method with param types - String, Object");  
}
```

» this cannot be compiled – cannot identify “most specific”

Method overloading



```
public static void hello(Collection x) {
    System.out.println("Collection");
}

public static void hello(List x) {
    System.out.println("List");
}

public static void main(String[] args) {
    Collection col = new ArrayList();
    hello(col);
}
```

» can be compiled and what is the output?



```
public static void hello(Collection x) {
    System.out.println("Collection");
}

public static void hello(List x) {
    System.out.println("List");
}

public static void main(String[] args) {
    Collection col = new ArrayList();
    hello(col);
}
```

» can be compiled and what is the output?

- **YES**

Collection

- compile time resolution not run-time type



RUNTIME DATA AREA



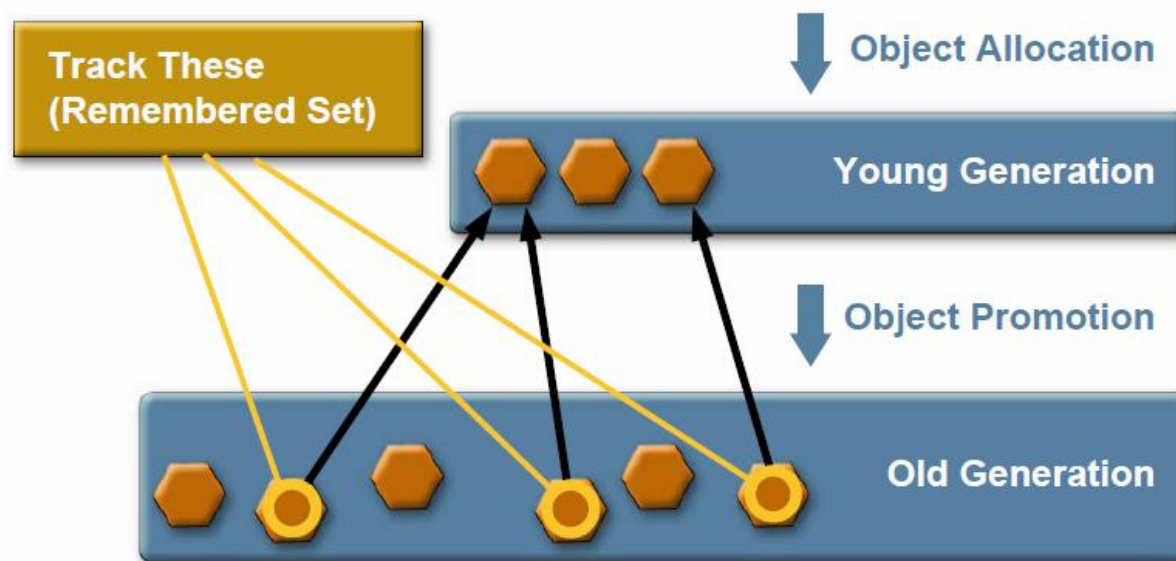


- » explicit *vs. automatic*
 - no crashes due to errors – e.g. usage of de-allocated objects
 - no space leaks
- » garbage collection managed by *garbage collector*
 - live objects (transiently reachable from **roots** – thread frames, static fields) remain in memory
 - dead are reclaimed
- » desired characteristics:
 - allocation performance - find a block of unused memory with certain size
 - avoid fragmentation (e.g. by compaction)
 - efficiency without long pauses in application run
 - no bottleneck for multi-threaded (multi-CPU) systems
- » design architectures:
 - serial *vs.* parallel
 - concurrent *vs.* stop-the-world
 - compacting *vs.* non-compacting *vs.* copying

Generational concept



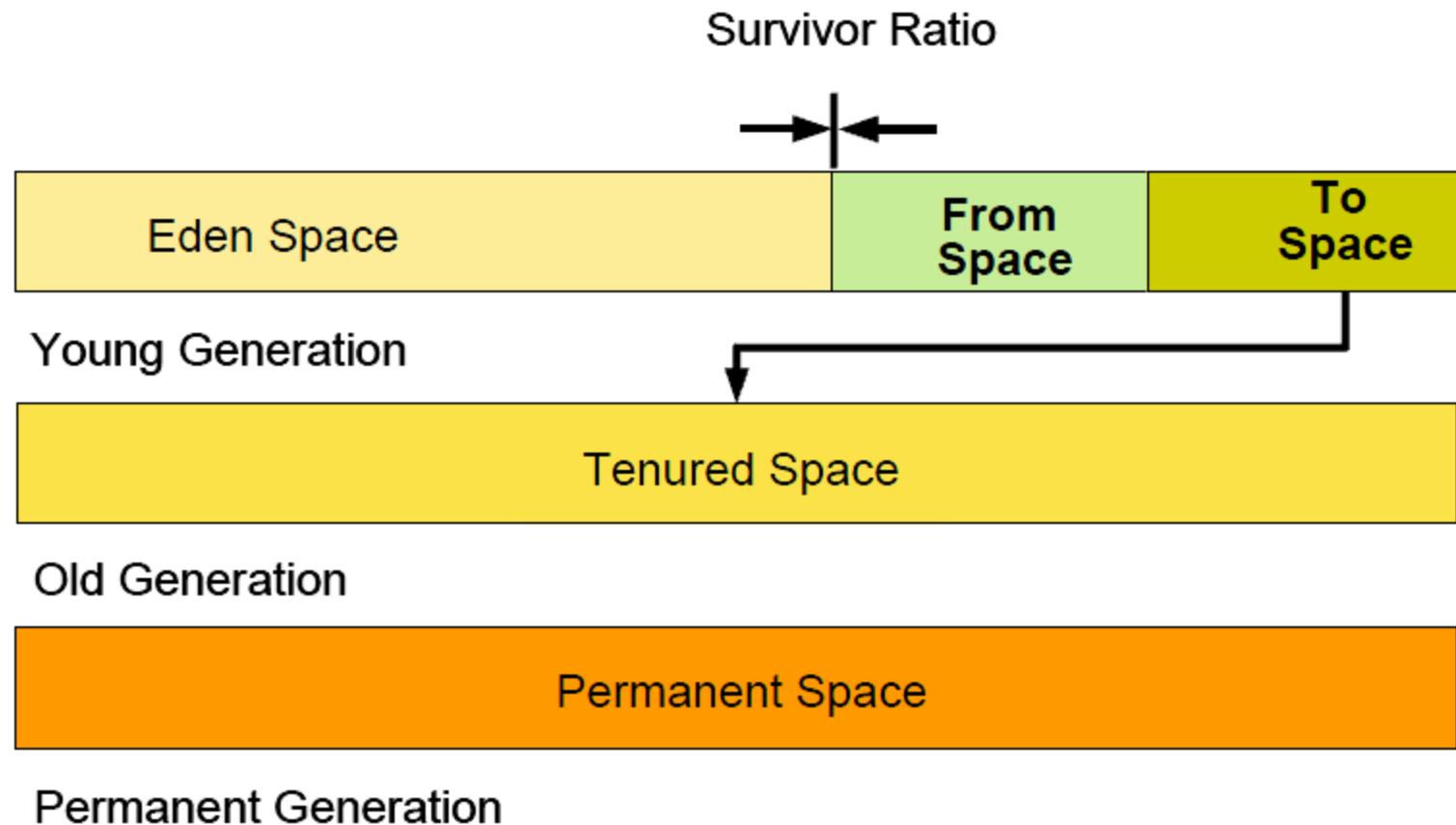
- » heap divided into generations based on object ages:
 - young – frequent GC, small size -> fast GC
 - old – rare GC, large size -> slow GC
- » promotion (tenuring) objects based on survival of objects during GC
- » based on weak generational hypothesis:
 - most allocated objects are not referenced for long – they die young
 - few references from older to younger object exist
- » need track old-to-young references



JAVA heap layout



- » *minor (young)* vs. *major (old)* GC – different algorithms
- » major GC can be invoked by young GC if there is no space in tenured space





- » based on *bump-the-pointer* technique
 - track previously allocated object
 - fit new object into remainder of generation end
- » thread-local allocation buffers (TLABs)
 - remove concurrency bottleneck
 - each thread has very small exclusive area (about 1% of Eden in total)
 - infrequent full TLABs implies synchronization (based on CAS)
 - exclusive allocation takes about *10 native instructions*



- » young collection -> old generations collection serially in *stop-the-world* fashion



- » young generation:

- » age of object (incremented every minor GC)
- » efficiency is proportional to number of copied objects !



Young Generation

