innopolis university

Using collections in Java

Stanislav I. Protasov

28/07/2015

Questions

- Copy constructor / clone / …
- Enum constructor
- Inheritance
- Initializing variables

Copy constructor (recommended)

- **Allows you to access private fields!**

```java
public class Class1 {
    private int value;
    private String string;

    public void print() {
        System.out.println(string + " = " + value);
    }

    public Class1(String string, int value) {
        this.value = value;
        this.string = string;
    }

    public Class1(Class1 source) {
        this.value = source.value;  // !!!
        this.string = source.string;// !!!
    }
}
```

Java approach

- Implement Cloneable

```java
public class Class2 implements Cloneable {

    private int value;
    private String string;

    public void print() {
        System.out.println(string + " = " + value);
    }

    @Override
    public Object clone() {
        try {
            return super.clone();
        } catch (CloneNotSupportedException ex) {
            return null;
        }
    }

    public Class2(String string, int value) {
        this.value = value;
        this.string = string;
    }

}
```

Static factory

```
public class Class3 {

    int value;

    private Class3() { }

    public static Class3 create() {
        Class3 c = new Class3();
        c.value = 123;
        return c;
    }
}
```

Enum is a Class

- All enums extend java.lang.Enum
- Special methods

```
for (MyEnum e: MyEnum.values()) {
    System.out.println(e);
}

public enum MyEnum {

    ONE(1.0), TWO(2.0), THREE(3.0);

    private double valueInsideInstance;

    public void print() {
        System.out.println(valueInsideInstance);
    }

    MyEnum(double value) {
        valueInsideInstance = value;
    }
}
```

- Constructor: private or package-private

Initialization and GC

- **Initialization blocks and initialization order**

```
public class Class4 {
    int a1 = 1; // before constructor call
    int a2;
    {
        a2 = 2; // before constructor call
    }

    static int a3 = 3; // when class access 1st time
    static int a4;
    static {
        a4 = 4;        // when class access 1st time
    }
}
```

- **Garbage collector**

Agenda

- Equality and comparison
- About storing the data
  - Pools and Sets
- Java collections
  - Collection and Iterator, Set, Map
- Common operations
- Generics
- hashCode()

Equality and comparison

```
int a = 4;
int b = 4;
System.out.println("int == int: " + (a == b));
Integer ai = (Integer)a;
Integer bi = (Integer)b;
Integer ci = new Integer(b);
System.out.println("Int == Int: " + (ai == bi));
System.out.println("Int == Int: " + (ai == ci));
Foo af = new Foo();
Foo bf = new Foo();
System.out.println("Foo == Foo: " + (af == bf));
```

Equality and comparison

```
int a = 4;
int b = 4;
System.
Integer
Integer
Integer
System.
System.
Foo af =
Foo bf = new Foo();
System.out.println("Foo == Foo: " + (af == bf));
```

```
int == int: true
Int == Int: true
Int == Int: false
Foo == Foo: false
```

Equality and comparison

```
int a = 4;
int b = 4;
// System.out.println("int == int: " + (a == b));
Integer ai = (Integer)a;
Integer bi = (Integer)b;
Integer ci = new Integer(b);
System.out.println("Int eq Int: " + ai.equals(bi));
System.out.println("Int eq Int: " + ai.equals(ci));
Foo af = new Foo();
Foo bf = new Foo();
System.out.println("Foo eq Foo: " + (af.equals(bf)));
```

Equality and comparison

```
int a = 4;
int b = 4;
// System.out.println("int == int: " + (a == b));
Integer
Integer
Integer
System.o                                  als(bi));
System.o                                  als(ci));
Foo af =
Foo bf = new Foo();
System.out.println("Foo eq Foo: " + (af.equals(bf)));
```

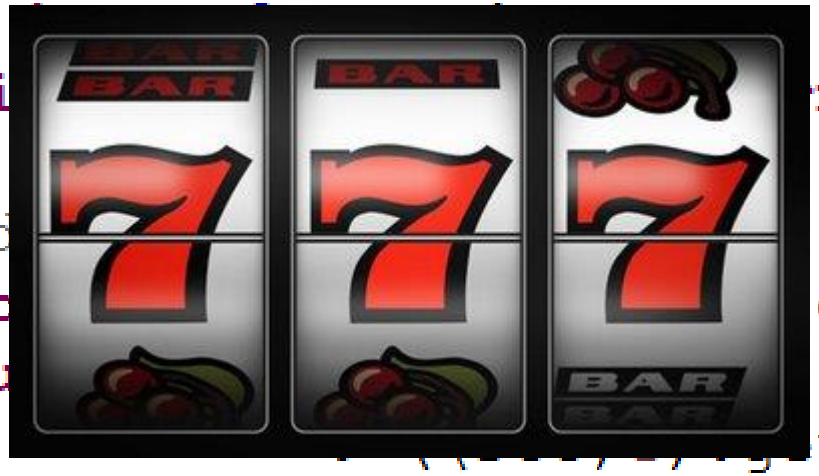
```
Int eq Int: true
Int eq Int: true
Foo eq Foo: false
```

Equality and comparison

```
public class Foo {

    private int value = 4;
    public int getValue() { return value; }

    @Override
    public boolean equals(Object e) {
        return e instanceof Foo
                    ? ((Foo)e).getValue() == value
                    : false;
    }
}
```

Equality and comparison

```
public class Foo {

    private
    public i                                  n value; }

    @Overrid
    public b                                  e) {
        retu
                                              tValue() == value
                                  : false;
    }
}
```



Very generic approach to store you data

- **Pools** – allow you to store **duplicate** data, add and remove elements. *E.g. Queue, Stack*

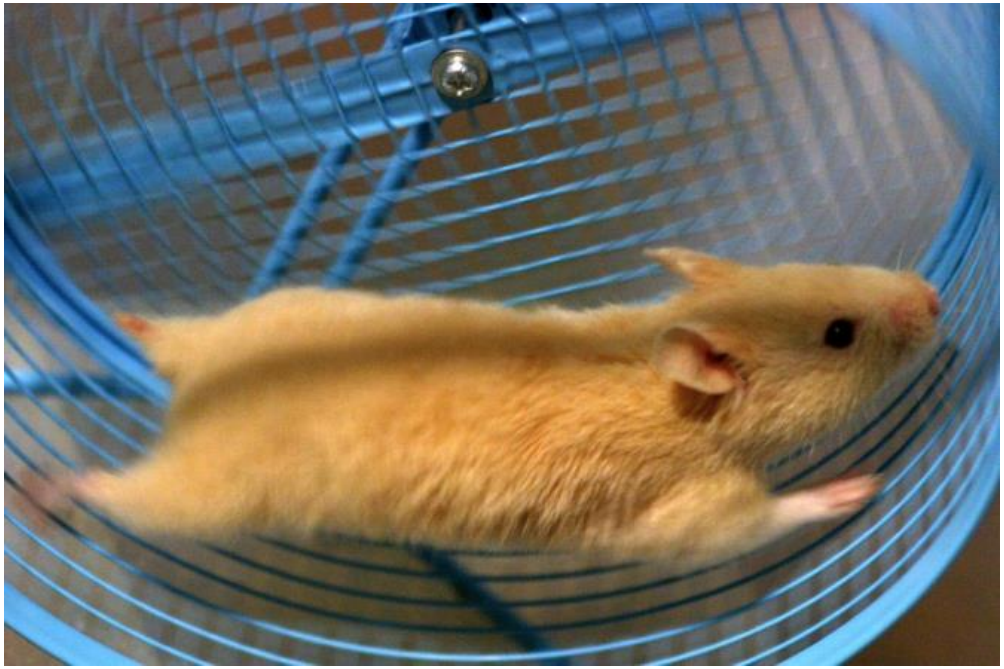
- **Sets** – allow you to store only **unique** data; add, remove and search for elements. *E.g. Map, Set*

Java thinks the same way: Pools

- **Pools**
  - **Collection interface**
    - `add(Object)`
    - `remove(Object)` / **`removeAll(Object)`**
    - `contains(Object)`
      - **`(o==null ? e==null : o.equals(e))`**
    - `size()`, `toArray()`, `clear()`
    - **`iterator()`**

Iterator

```
Object[] array = new Object[100];

for (int i = 0; i < array.length; i++) {
    array[i].toString();
}
```



Iterator

- **Iterator is an interface, that allows you do the following things**

```
Collection c1 = new ArrayList();
Iterator iterator = c1.iterator();
while (iterator.hasNext())
    iterator.next().toString();

Iterable c2 = new ArrayList();
for (Object o : c2)
    o.toString();
```

Collection – who's there?

- Some implementations
  - ArrayList
  - Stack
  - ArrayBlockingQueue
  - LinkedList
  - PriorityQueue

Java thinks the same way: Sets

- **Sets**
  - **Map interface –** stores data accessed by **unique** *key*
    - `put(Object key, Object value)`
    - `get(Object key)`
    - `containsKey(Object key)`
    - `containsValue(Object value)`

Map interface

```
Map mapMusic = new HashMap();
mapMusic.put("Ivanov", "AC/DC");
mapMusic.put("Petrov", "Ivan Dorn");
mapMusic.put("Ivanov", "Behemoth");
for (Object o : mapMusic.keySet())
    System.out.println(o + " likes " + mapMusic.get(o));
```

```
Petrov likes Ivan Dorn
Ivanov likes Behemoth
```

Map – who's there?

- Some implementations
  - HashMap
  - EnumMap
  - Hashtable
  - TreeMap
  - …

Sets without a key

- What if you don’t want to think about keys?
- Then **Set interface**!
    - `put(Object value)`
    - `remove(Object value)`
    - `contains(Object value)`

Operations with collections

- **Comparator and Comparable**
  - `Comparator: int compare(Object o1, Object o2)`
  - `Comparable: int compareTo(Object o1)`
- **Collections class**
  - `sort(List l) / sort(List l, Comparator c)` | *LinkedList, ArrayList*
  - `binarySearch`
  - `min / max`
  - `replaceAll / frequency`
  - *`fill / nCopies`*
  - `...`
- **Iterable class**
  - `forEach(Consumer c)`
- **java.util.SortedSet ; java.util.SortedMap**

Generics

- Generic is a way to write create classes (collections) that process only **specified types** of arguments with **compile-time check**

```
List v = new ArrayList();
v.add("test");
Integer i = (Integer)v.get(0); // Run time error
```

Generics collections

```
ArrayList<String> al1 = new ArrayList<String>();
ArrayList<Foo> al2 = new ArrayList<>();

al1.add("12312313");
al2.add(new Foo());
al1.add(new Integer(1)); // Compilation error
String string = al1.get(0);
```

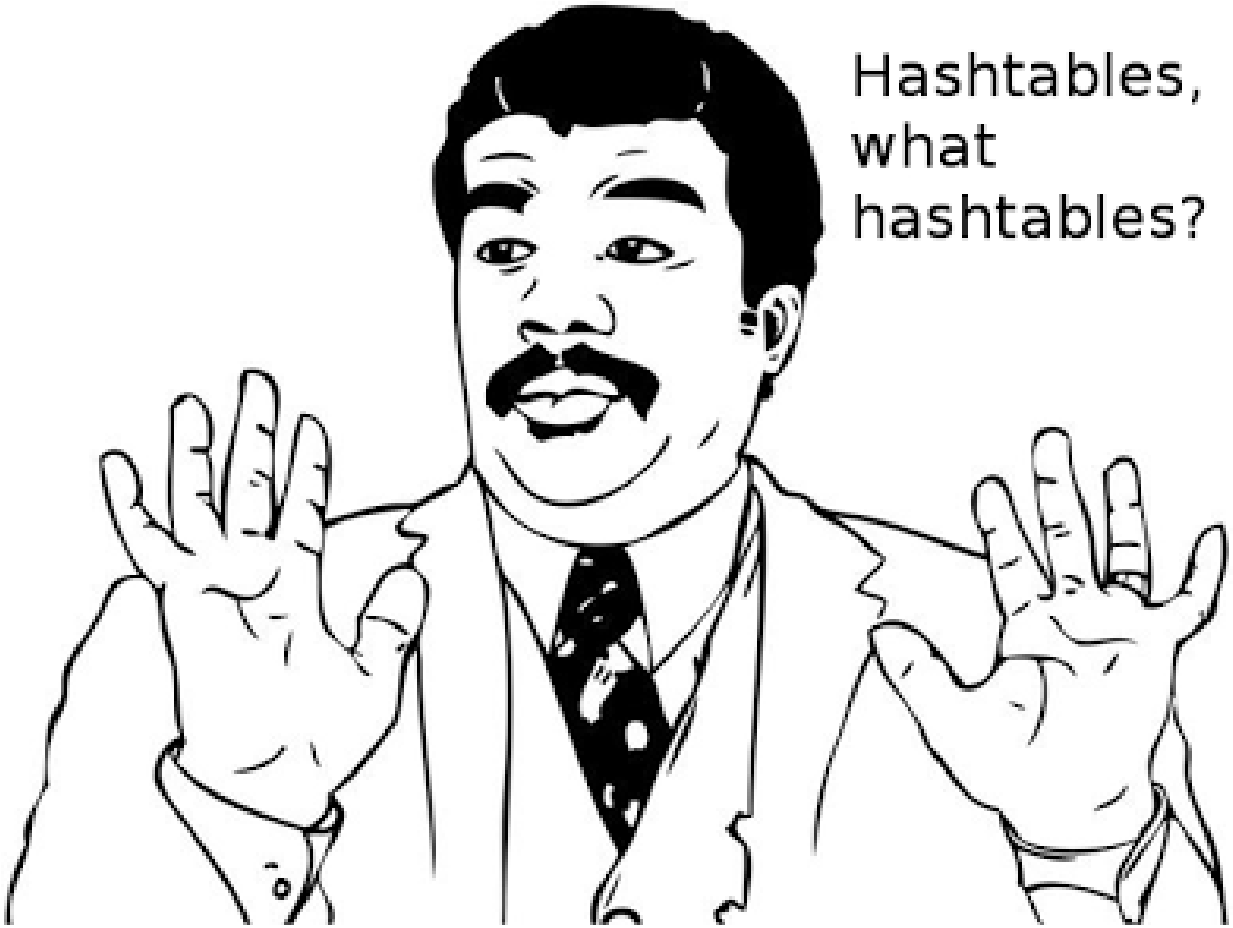
Your own generic

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

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

Let’s talk about hashCode()

Any thift generated
Java class:

```
@Override
public int hashCode(){
        return 0;
}
```



s.protasov@innopolis.ru