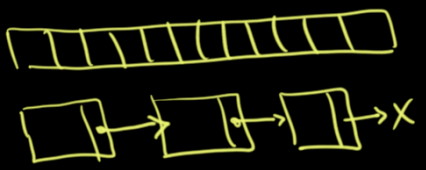
Java's Collections

primitive types — int, long, double, float, char, bool

arrays — int[]

List<A> — sequence of Elements ~ conventional array

List<A> =
- → ArrayList
- → LinkedList



Set<A> - it gets rid of duplicates
it guarantees uniqueness

```java
public static List<Integer> filterNegative(List<Integer> data) {
  ArrayList<Integer> result = new ArrayList<>();
  for (int i = 0; i < data.size(); i++) {
    int x = data.get(i);
    if (x < 0) result.add(x);
  }
  return result;
}
```

a lot of places for a mistake

```java
for (int x: data) {
  if (x < 0) result.add(x);
}
```

1. no initial index declaration
2. no final check size
3. no increment
4. no direct list access

```java
data.forEach(x -> {
  if (x < 0) result.add(x);
});
```

1. we don't copy/paste for...
2. we delegate traversing / iteration to the List

```
default void forEach(Consumer<T> action) {
    Objects.requireNonNull(action);
    for (T t : this) {
        action.accept(t);
    }
}
```

It's generic

```
public interface Consumer<T> {

    void accept(T t);
```

```
class NegativeProcessor implements Consumer<Integer> {
  @Override
  public void accept(Integer x) {
    if (x < 0) result.add(x);
  }
}

NegativeProcessor np = new NegativeProcessor();

data.forEach(x -> np.accept(x));
```

```
data.forEach((Integer x) -> {
  if (x < 0) result.add(x);
});
```

* since Consumer<T> has only one method => there is no sence to name it.

```
ArrayList<Integer> result = new ArrayList<>();

data.forEach((Integer x) -> {
  if (x < 0) result.add(x);
});
```

```
data.forEach(x -> {
  if (x < 0) result.add(x);
});
```

```java
public static List<Integer> filterNegative(List<Integer> data) {
  ArrayList<Integer> result = new ArrayList<>();
  data.forEach(x -> {
    if (x < 0) result.add(x);
  });
  return result;
}

public static List<Integer> filterPositive(List<Integer> data) {
  ArrayList<Integer> result = new ArrayList<>();
  data.forEach(x -> {
    if (x > 0) result.add(x);
  });
  return result;
}
```

90% of duplication

we need to write one function filterBy and pass condition x<0 x>0 as a parameter

x > 0
x < 0
x == 0
x % 2 == 0

f: x → boolean

int → bool

```java
interface FilterFn {
  boolean check(int x);
}
```

```java
public static List<Integer> filter(List<Integer> data, FilterFn fn) {
  ArrayList<Integer> result = new ArrayList<>();
  data.forEach(x -> {
    if (fn.check(x)) result.add(x);
  });
  return result;
}
```

```java
filter(data, x -> x > 0)
filter(data, x -> x < 0)
```

```java
public static List<Integer> filterPositive(List<Integer> data) {
  ArrayList<Integer> result = new ArrayList<>();
  for (int i = 0; i < data.size(); i++) {
    int x = data.get(i);
    if (x > 0) result.add(x);
  }
  return result;
}
```

← this implement requires copy/paste of ALL our code

BAD

```java
public static List<Integer> filterList(List<Integer> arr, boolean isPositive){
  List<Integer> resArr = new ArrayList<>();
  arr.forEach(item -> {
    if(item!=0 && (item>0) == isPositive)
      resArr.add(item);
  });
  return resArr;
}
```

← we moved copy/paste problem inside our fn.
we made it even more complex

VERY BAD

```java
interface FilterFn {
  boolean check(int x);
}

public static List<Integer> filter(List<Integer> data, FilterFn fn) {
  ArrayList<Integer> result = new ArrayList<>();
  data.forEach(x -> {
    if (fn.check(x)) result.add(x);
  });
  return result;
}
```

← 1. we introduced new interface
2. we completely decoupled interface and impl

```java
filter(data, x -> x > 0);
filter(data, x -> x < 0);
```

```
@Override
public int hashCode()
```

we need hashCode only for collections based on Hash Impl.

HashMap
HashSet

```
Set<Pizza> ps = new HashSet<>();
ps.add(new Pizza( size: 30));
ps.add(new Pizza( size: 60));
ps.add(new Pizza( size: 60));
```

pizza.equals(pizza_given)
it can take a lot of time
=> pizza can contain a lot of fields

hashCode → for faster comparing

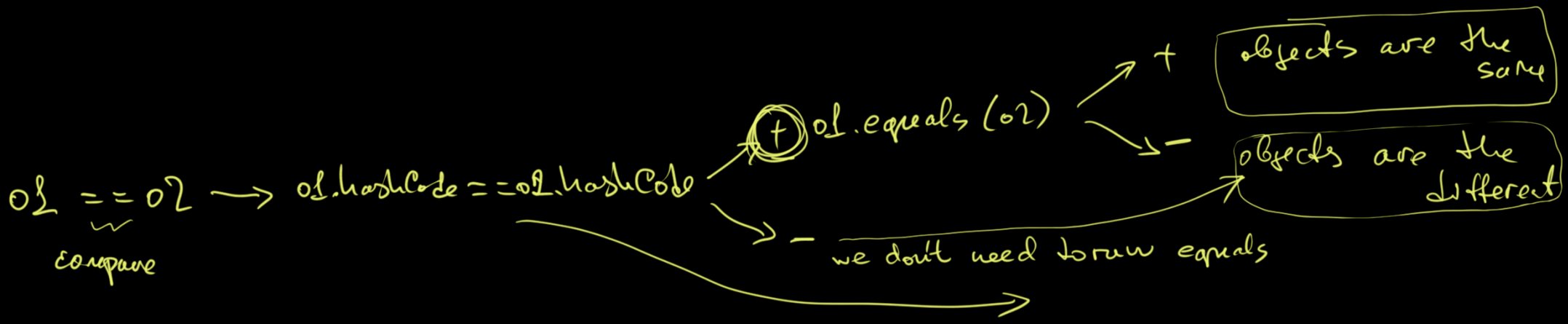
o1 o2

1. o1.hashCode == o2.hashCode
   - \+ → o1.equals(o2)
     - \+ → objects are the same
     - \- → objects are different
   - \- → objects are different

```
@Override
public int hashCode() {
  return (size * 7 + name.hashCode()) * 7 + Arrays.hashCode(extras);
}
```

← we need to somehow mix our data

* equals can take a lot of time (due to many fields)
* we implement hashcode to make comparing faster



* we need to mix all the fields in the implementation

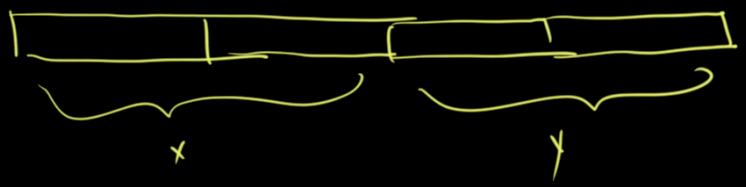
```
public int hashCode() {
  Random random = new Random();
  return random.nextInt();
```

→ gives us <u>false</u> knowledge that obj. are DIFF

```
public int hashCode() {
  return 0;
```

→ this impl. forces using equals each time

what to write in hashCode




```
class Point {
    int x          0<x<1000
    int y
}
hashcode {
    return x + y<<16;
}
```

```
class Person
    a1
    a2
    a3
    a4
    ⋮
```

15, 51, 97

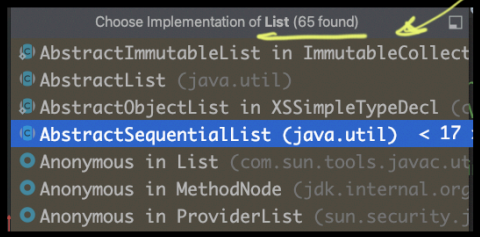
(((a1.hash *7) + a2.hash) *7 +

a3.hash) *7 + a4.hash .

---

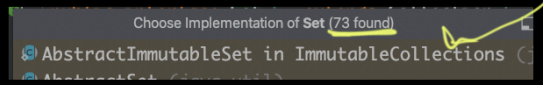
things don't fit into 4 bytes
and we need to mix them somehow

List → ArrayList, LinkedList, ...



having List interface
we can have the same code
to work with any of 65 implementations

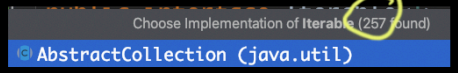
Set → HashSet, TreeSet, ...



having Set interface
same code to work with 73 impl.

```
public interface List<E> extends Collection<E> {
public interface Set<E> extends Collection<E>
public interface Collection<E> extends Iterable<E> {
```

forEach



the ultimate Idea of having interfaces

is writing code w/o any knowledge about particular impl

Map<K,V>

Key	Value
1	Pizza(size: 30, name: "Margarita");
2	Pizza(size: 40, name: "Margarita");
3	Pizza(size: 40, name: "QuattroFormaggio");
4	Pizza(size: 60, name: "QuattroFormaggio");

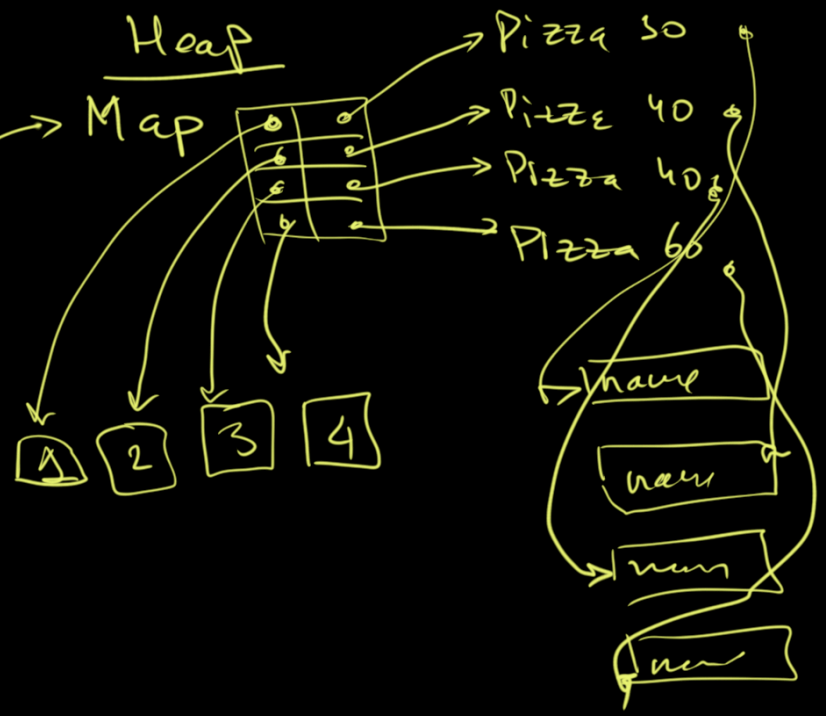
```
map = new HashMap<>();
```

Stack

Key	Value
A	→ 3
W	→ 2
J	→ 1

Character

Integer.



"Hello, World! Java is awesome!"

Char	List<Int>
H	→ 1
e	→ 2, 21, 25
l	→ 3, 4

Map<Character, List<Integer>>

Letter

position

List of positions