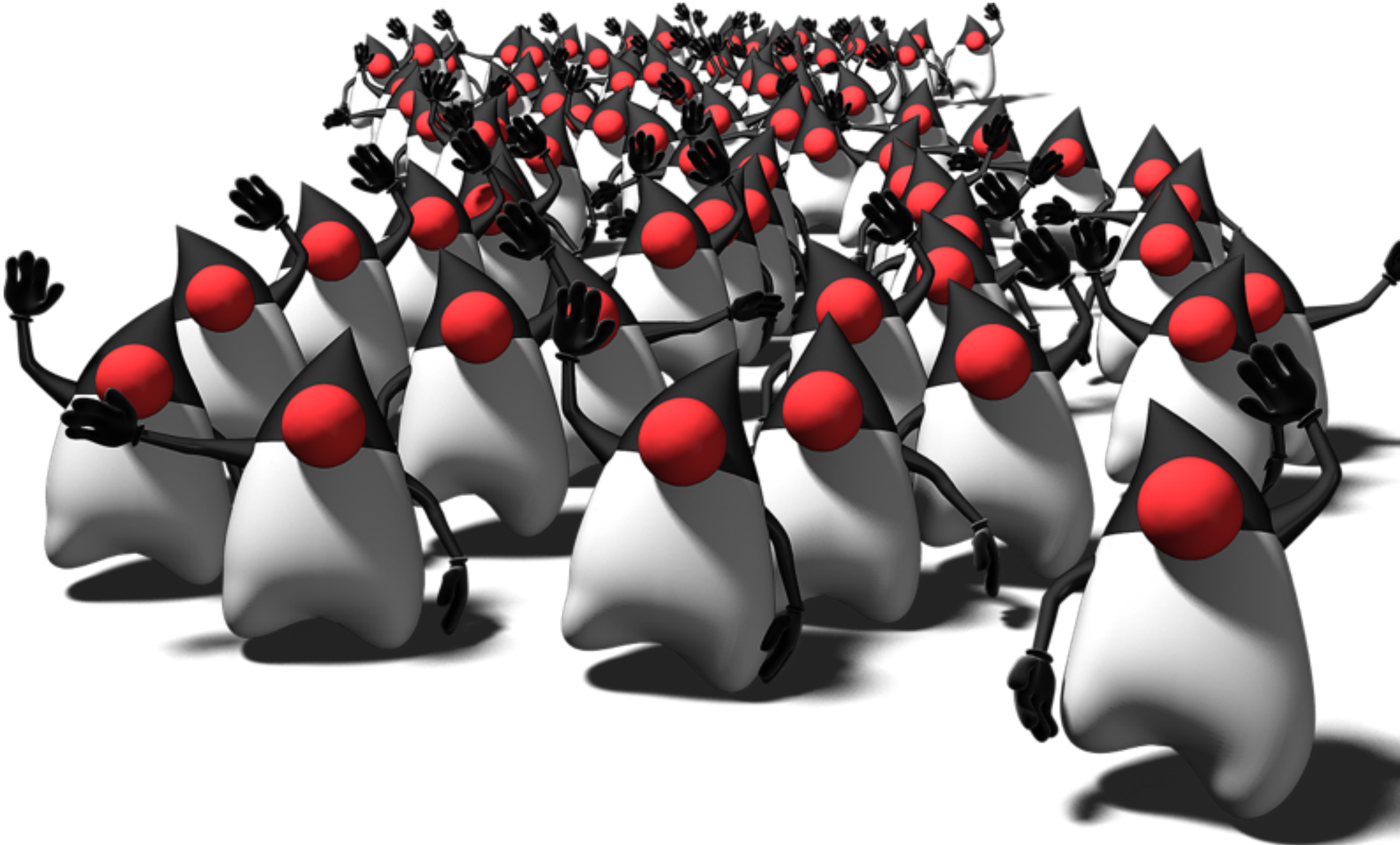
ZEROTURNAROUND

Java Fundamentals 2017

COLLECTIONS & GENERICS



Arnel Pällo
13.02.2017

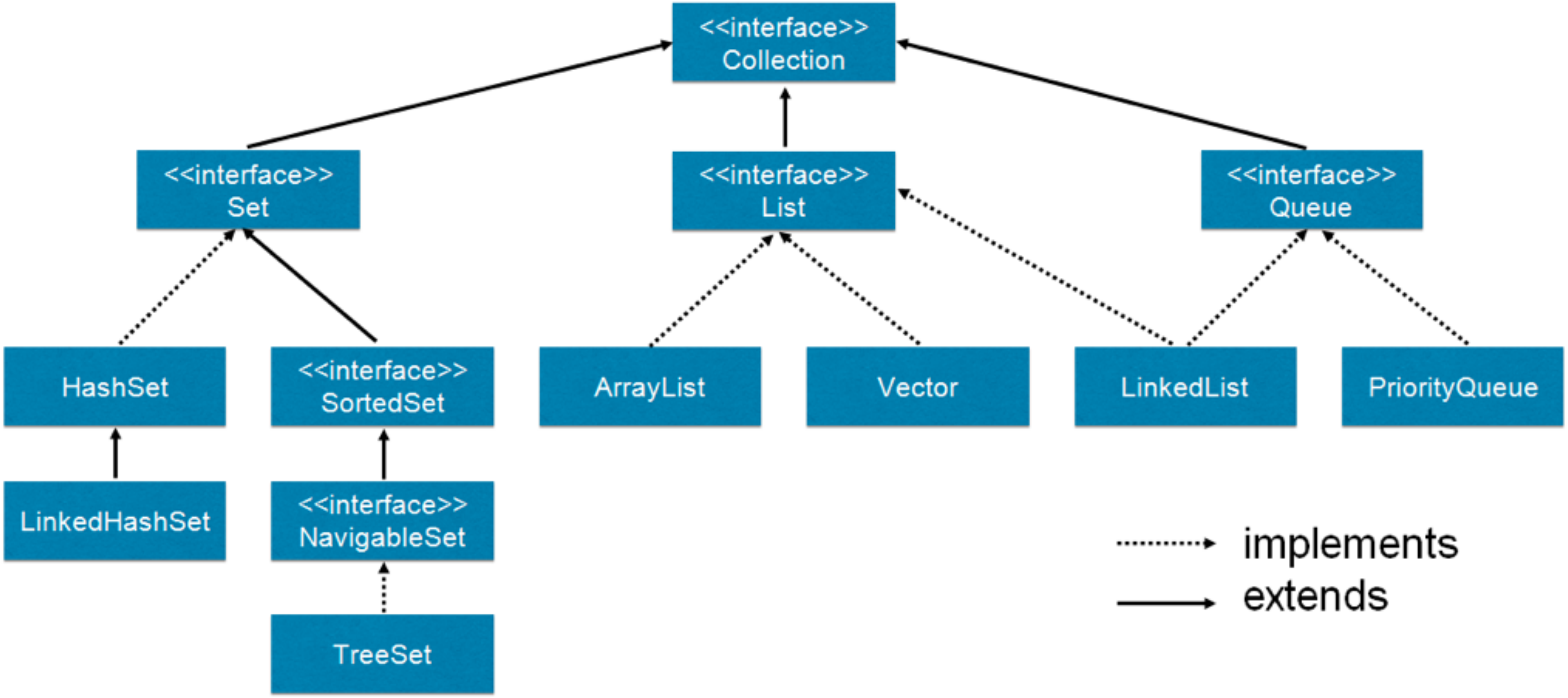
Collections

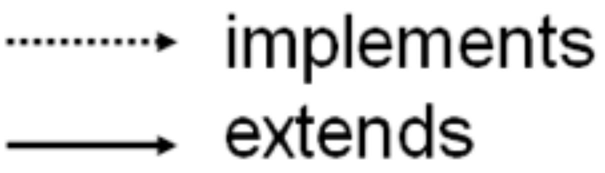
- overview
- common, special & legacy impl's
- linked list
- equals & hashCode
- arrays
- libraries

Collections overview

- **Collection** - iterable, basic operations
    - **List** - ordered, with duplicates
    - **Set** - unordered*, no duplicates
    - **Queue & Deque -** queues, stacks
- **Map** - unordered*, key value pairs
    - keySet(), entrySet(), values()

Collection Interface





Iterable

- Only one method
  - Iterator<T> iterator();
- allows for-of loops (pre java8 style)
- adds forEach via default method

interface Collection

- contains(el) & containsAll(coll)
- add(el) & addAll(coll)
- remove(el) & removeAll(coll)
- retainAll(coll)
- **default Stream<E> stream()**
- default removeIf(predicate)

List

- get(index): Element
- add(index, element) : void
- set(index, element) : Element
- remove(index) : Element
- default void sort()

Set

- does not define any additional methods

Queue

- first-in-first-out
- may have capacity restrictions
- add(), remove(), element() - exception
- offer(), poll(), peek() - special val

- Often Used via sub-interface Deque

Deque

- Double ended queue and a stack

- addFirst, addLast, offerFirst, offerLast
- removeFirst, removeLast, pollFirst, pollLast
- getFirst, getLast, peekFirst, peekLast
- push(), pop()

Common implementations

- List - ArrayList
- Set - HashSet
- Map - HashMap
- Queue - ArrayDeque

Very often the best choices

Special implementations

- Ordered
  - LinkedHashSet
  - LinkedHashMap
- Sorted
  - TreeSet
  - TreeMap

Legacy Collections

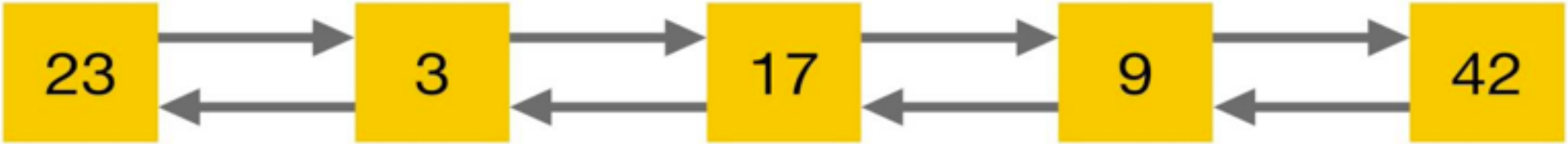
- Synchronized
  - Vector (use ArrayList)
  - Hashtable (use HashMap)
  - Stack (use ArrayDeque)
- Enumeration (succeeded by Iterator)

LinkedList

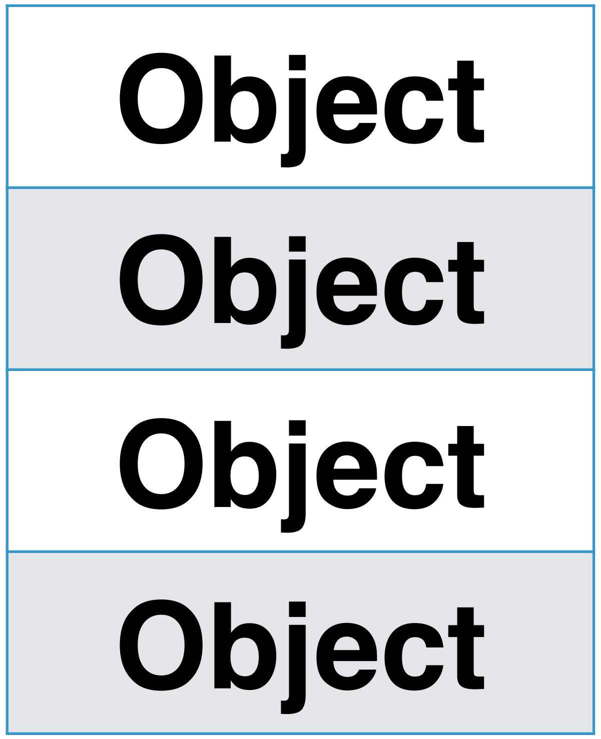
- Often the wrong choice. Except when
    - O(1) is critical for appending
- Random access is horrid - O(n)
- Uses more memory
- Iteration is slower

Memory Layout

LinkedList



ArrayList



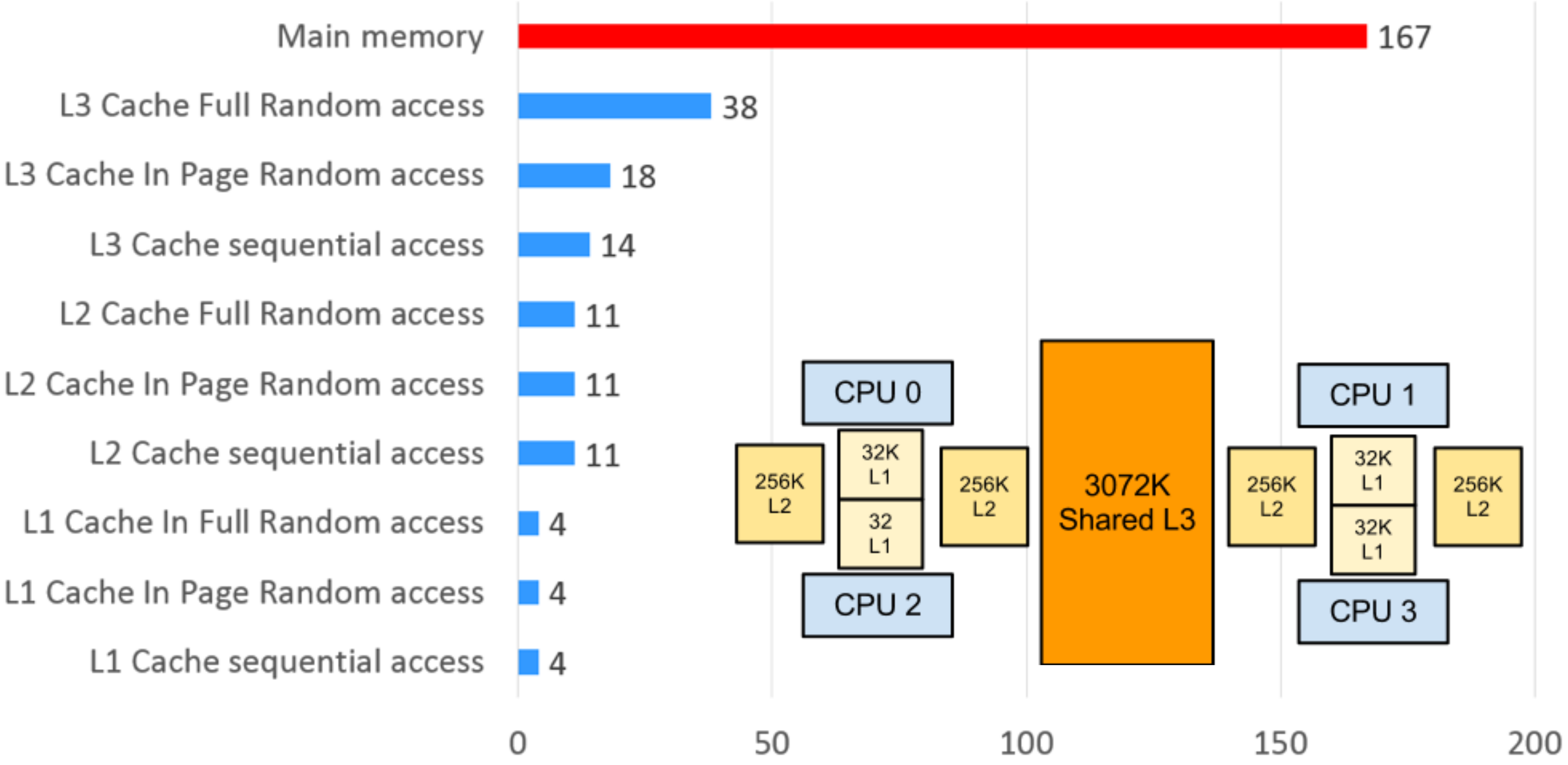
Object

Object

Object

Object

CPU Cache Access Latencies in Clock Cycles



equals & hashCode

- Used by collections and maps
  - contains(el)
  - remove(el)
  - adding to sets and maps
  - performance

equals & hashCode

- if (a.equals(b)) then
    - a.hashCode() == b.hashCode()
    - but not the other way around
    - several non-equal objects might have the same hashCode

Equals

- a.equals(a) **==** true
- a.equals(b) **=>** b.equals(a)
- a.equals(b) **&&** b.equals(c) **=>** a.equals(c)
- a.equals(null) **==** false

```java
public class Circle {
  public int radius;

  public boolean equals(Object obj) {
    if (obj instanceof Circle) {
      Circle c = (Circle) obj;
      return c.radius == radius;
    }
    return false;
  }

}
```

```
class BlingCircle extends Circle {
  public int color;
  public boolean equals(Object obj) {..}
}

circle.equals(blingCircle); //true!
blingCircle.equals(circle); //false
```

Equals & Inheritance

- Mind the subclass
- Either use final class, or

```
obj.getClass().equals(Circle.class)
```

HashCode

- Used by HashMap & HashSet
- Good hashing -> O(1) complexity
- Bad hashing -> bad performance
  - Elements end up in same bucket
  - Slow within the same bucket
- Don't use for equality checking

java.util.Collections

- many great utilities
  - sort, shuffle, min, max, reverse, swap, ...
  - emptyList(), emptyMap(), emptySet()

Collections vs arrays

- Virtually no performance difference
- Arrays hard to use
  - need to do bookkeeping
  - can not instantiate generic arrays
  - need 9 versions of each function

Good uses for arrays

- big arrays of primitives
  - no autoboxing
  - much less memory needed
  - cache locality
- varargs

Varargs

```
static int sum(int... numbers) {
  // numbers is instanceof int[]
  ...
}

public static void main(String[] args) {
  int number = sum(2, 4, 6, 8);
}
```

Collection libraries

- Guava
- Commons collections
- Multiset / MultiSet
- Multimap / MultiValuedMap
- BiMap / BiDiMap
- Table

Multiset / MultiSet

- Set with count

```
Multiset<String> s = HashMultiset.create();
s.add("foo"); //[foo]
s.add("foo"); //[foo x 2]
s.add("bar", 3); //[bar x 3, foo x 2]
s.remove("foo"); //[bar x 3, foo]
s.remove("bar", 3);//[foo]
```

Multimap/MultiValuedMap

- Map with many values per key
- ListMultimap & SetMultimap

```
Multimap<String, Integer> m = HashMultimap.create();
m.put("foo", 2);
m.put("foo", 1);
Collection<Integer>foos=m.get("foo");//[1, 2]
```

BiMap / BidiMap

- Bi-directional map
- Both keys and values are unique

```
BiMap<String, Integer> m = HashBiMap.create();
m.put("foo", 42);      //{foo=42}
m.put("bar", 17);      //{foo=42, bar=17}
m.put("baz", 17);      //ERR
m.forcePut("baz", 17); //{foo=42, baz=17}
m.inverse().get(17);   //baz
```

Table

- Map with two keys (row and column)

```
Table<String, String, Integer> tbl =
HashBasedTable.create();
tbl.put("row1", "col1", 42);
tbl.put("row1", "col2", 255);
System.out.println(tbl);
//{row1={col2=255, col1=42}}

int magic = t.get("row1", "col1");
System.out.println(magic); //42
```

Conclusion

- Java has powerful collections
- LinkedList is overused
- Look into java.util.Collections
- Use arrays only for primitives and in extreme cases
- Use libraries, don't reinvent the wheel
- Remember .stream()

Generics

Generics

- The problem they're solving
- Syntax and where to put them
- Bounded generics
- Inheritance and wildcards
- Type erasure
- Puzzlers

Generics

- type and function parameters
- adds flexibility to strong static typing
- allows to keep Strings and Puppies apart

The Problem

```
class Cup {
  private Object content;

  void fill(Object content) {
    this.content = content;
  }
  Object get() {
    return content;
  }
```

The Problem

```
Cup cup = new Cup();
cup.fill(new Coffee());
cup.fill(new ArrayList());

Object drink = cup.get();
Coffee coffee = (Coffee) drink;
```

Subclassing Solution

```
class TeaCup extends Cup {
  public void fill(Tea content) {
    super.fill(content);
  }
  public Tea get() {
    return (Tea) super.get();
  }
}
```

Generics

```
class SmartCup<T> {

  private T content;

  void fill(T content) {
    this.content = content;
  }

  T get() {
    return this.content;
  }
}
```

Generics

```
SmartCup<Coffee> cup = new SmartCup<>();
cup.fill(new Coffee());
cup.fill(new Tea()); //ERROR

Coffee coffee = cup.get();

SmartCup<Tea> cup2 = new SmartCup<>();
cup2.fill(new Tea());
```

Collections & Generics

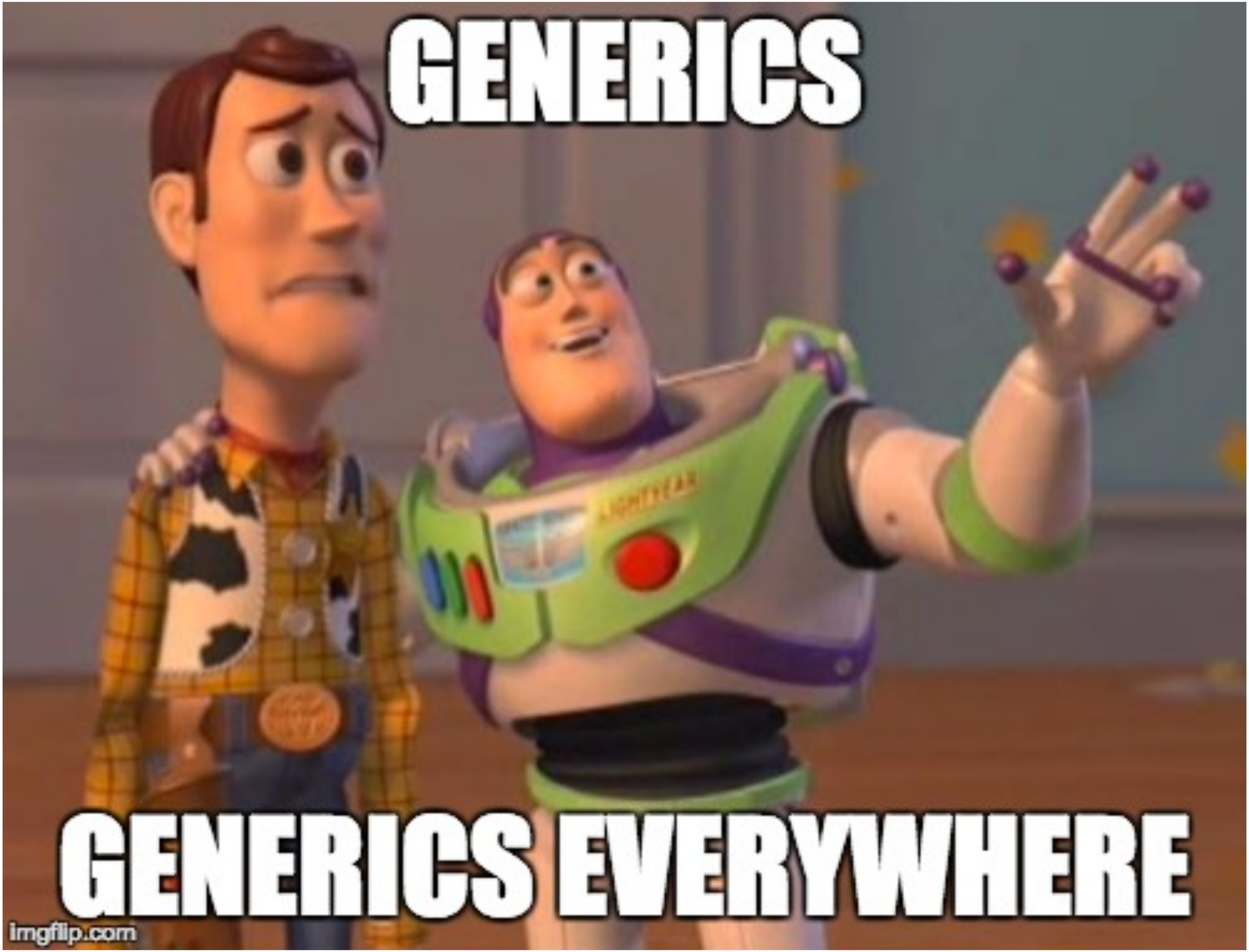
```
interface Collection<E> extends Iterable<E>
    interface List<E> extends Collection<E>
    interface Set<E> extends Collection<E>


interface Map<K,V>

interface Stream<T>
```

Generics on a class

```
@FunctionalInterface
public interface Function<T, R> {

    R apply(T t);

}
```

Generics on a Method

```
public class Stream {

 public static <T> Stream<T> of(T... values){
     return Arrays.stream(values);
 }
```


GENERICS
GENERICS EVERYWHERE
imgflip.com

<> on a class and a method

```
interface Stream<T>

  <R> Stream<R> map(Function<T, R> mapper);
```

Referencing itself

```
class String implements Comparable<String>

interface Comparable<N> {
    int compareTo(N o);
}
```

Bounded Generics

```
<R extends Runnable> R startRunnable(R run) {
  new Thread(run).start();
  return run;
}
```

Bounded generics

```
<N extends Comparable<N>> N max(N a, N b) {
  if (a.compareTo(b) >= 0) {
    return a;
  }
  return b;
}
interface Comparable<N> {
     int compareTo(N o);
}
```

Bounded Generics

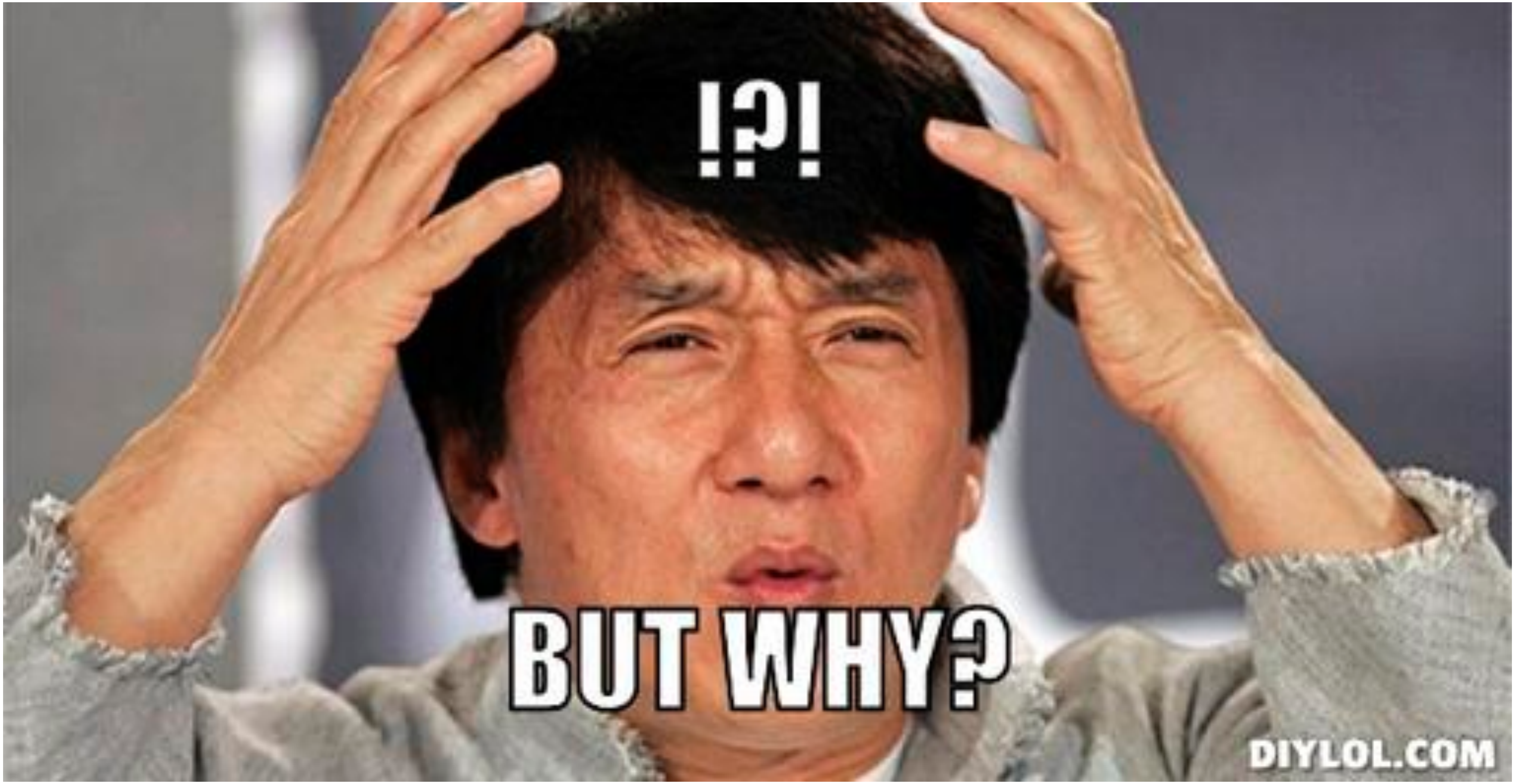
```
System.out.println(max(2.0, 3.0)); //OK
System.out.println(max(2, 3));  //OK
System.out.println(max(2, 3.0)); //ERR
```

Generics on variables

```
List objects = new ArrayList();
List<Integer> ints = new ArrayList<>();

List<?> something = new ArrayList<>();
List<? extends Number> numbers = ...
List<? super Number> anything = ...
```

Inheritance

```
Number num = Integer.valueOf(42); //OK


List<Integer> ints = new ArrayList<Integer>();

List<Number> nums = ints; // ERROR

// in java, generics inheritance does not work
```


!?!
BUT WHY?
DIYLOL.COM

What if..

```
List<Integer> ints = new ArrayList<Integer>();
List<Number> nums = ints; //IF IT WERE LEGAL..

nums.add(1.0); //LEGAL

int num = ints.get(0); //LEGAL, but
```

will throw ClassCastException in runtime.

Inheritance

```
List<Number> n = new ArrayList<Integer>(); //E

List<? extends Number> n =
                                    new ArrayList<Integer>(); //OK

List<? super Integer> n =
                                    new ArrayList<Number>(); //OK
```

Extends

```
List<? extends Number> list;
```

may hold any of the following:

```
List<Number>
List<Integer>
List<Double>
```

```
list.add(1);  //how about List<Double>?
list.add(1.0); //how about List<Integer>
```

Extends

```
List<? extends Number> list;
   may hold any of the following:
      List<Number>
      List<Integer>
      List<Double>

list.get(1);  //always returns a Number!
// Integer and Double both extend Number
```

Super Inheritance

```
List<? super Integer> n =
                    new ArrayList<Number>(); //OK

List<? super Integer> n =
                      new ArrayList<Object>(); //OK

List<? super Number> n =
                   new ArrayList<Integer>(); //ERR
```

Super inheritance

```
List<? super Number> list;
```

may hold any of the following:

```
    List<Number>
    List<Object>
```

```
list.add(1);  //OK
list.add(1.0); //OK
list.add("Bunny"); //ERR (not a Number)
```

Super inheritance

```
List<? super Number> list;
```

may hold any of the following:

```
    List<Number>
    List<Object>
```

```
Number num = lst.get(0);  //ERR
Object obj = lst.get(0);  //OK
```

Implications

PECS - Producers Extend, Consumers Super

producer - get(0) // produces a value

consumer - add(0) // consumes a value

<> on a class and a method

```
interface Stream<T>

  <R> Stream<R> map(Function<T, R> mapper);
```

not entirely accurate...

<> on a class and a method

```
interface Stream<T>
```

~~<R> Stream<R> map(Function<T, R> mapper);~~

```
 <R> Stream<R> map(
   Function<? super T, ? extends R> mapper
 );
```

Type erasure

- Type information lost in bytecode?
  - Not really
  - It's all there!

Type erasure

```
public class ParaClass<T> {
  public List<Number> nums;
  public ParaClass() {
    List<String> myList = new ArrayList<>();
    System.out.println(myList);
  }
  public <R> R wrap(R r) {
    return r;
  }
```

Type erasure

```
> javap ParaClass.class
Compiled from "ParaClass.java"
public class ee.ut.jf2016.ParaClass<T> {
  public java.util.List<java.lang.Number> nums;
  public ee.ut.jf2016.ParaClass();
  public <R> R wrap(R);
}
```

Type erasure

```
public class ParaClass<T> {
  public List<Number> nummers;

  public ParaClass() {
    List<String> myList = new ArrayList();
    System.out.println(myList);
  }

  public <R> R wrap(R r){
    return r;
  }
```

Type erasure

- JVM knows when
  - class is parametrized
  - method is parametrized
- Also knows concrete generic type of
  - fields
  - local variables

Type erasure

- JVM does NOT know about objects

Type erasure

```
public class ParaClass<T> {
  public List<Number> nummers;

  public ParaClass() {
    List<String> myList = new ArrayList();
    System.out.println(myList);
  }

  public <R> R wrap(R r){
    return r;
  }
```

Why Type Erasure?

- Guiding principle of java always:
    - All existing code keeps working

Old code

```
public class OldClass {
  public List nummers;
  public OldClass() {
    List myList = new ArrayList();
    System.out.println(myList);
  }
}
```

Compiles or Not?

```
interface Drawable {}
   class Shape implements Drawable {}
      class Square extends Shape {}
      class Circle extends Shape {}
          class GreenCircle extends Circle {
```

Compiles or Not?

```
List<Shape> shapes = new ArrayList<>();
List<Circle> circles = new ArrayList<>();

shapes.add(new Circle());
circles.add(new Shape());
Shape val = circles.get(0);
shapes = circles;
```

Compiles or Not?

```
List<Shape> shapes = new ArrayList<>();
List<Circle> circles = new ArrayList<>();

shapes.add(new Circle());   // OK
circles.add(new Shape());   // ERR
Shape val = circles.get(0); // OK
shapes = circles;           // ERR
```

Compiles or Not?

```
List<? extends Shape> shapes =
                         new ArrayList<>();
List<Circle> circles =new ArrayList<>();

shapes = circles;
Circle c = shapes.get(0);
shapes.add(new Circle());
shapes.add(new Shape());
```

Compiles or Not?

```
List<? extends Shape> shapes =
                        new ArrayList<>();
List<Circle> circles =new ArrayList<>();

shapes = circles;          // OK
Circle c = shapes.get(0); // ERR
shapes.add(new Circle()); // ERR
shapes.add(new Shape());  // ERR
```

Compiles or not?

```
List<? super Circle> list = ...

list.add(new Circle());
list.add(new Shape());
list.add(new GreenCircle());
Circle circle = list.get(0);
Shape shape = list.get(0);
```

Compiles or not?

```
List<? super Circle> list = ...

list.add(new Circle()); // OK
list.add(new Shape());  // ERR
list.add(new GreenCircle()); // OK
Circle circle = list.get(0); // ERR
Shape shape = list.get(0);   // ERR
Object obj = list.get(0);   // OK
```

Compiles or Not?

```
class ShapeBox<S extends Shape> {}

ShapeBox<Circle> box;
ShapeBox<Shape> box;
ShapeBox<Drawable> box;

ShapeBox<? extends Circle> box;
ShapeBox<? extends Drawable> box;
```

Compiles or Not?

```
class ShapeBox<S extends Shape> {}

ShapeBox<Circle> box;   // OK
ShapeBox<Shape> box;    // OK
ShapeBox<Drawable> box; // ERR

ShapeBox<? extends Circle> box; //OK
ShapeBox<? extends Drawable> box; //OK
```

Compiles or Not?

```
class ShapeBox<S extends Shape> {}

ShapeBox<? extends Runnable> box;
```

?

Compiles or Not?

```
class ShapeBox<S extends Shape> {}

class Nike
            extends Shape implements Runnable

ShapeBox<? extends Runnable> box =
                         new ShapeBox<Nike>();
```

Compiles or Not?

```
class DrawBox<D extends Drawable> {}

DrawBox<? extends Thread> box;
```

?

Compiles or Not?

```
class DrawBox<D extends Drawable> {}

class DrawableThread
   extends Thread implements Drawable {}

DrawBox<? extends Thread> box =
         new DrawBox<DrawableThread>();
```

Compiles or Not?

```
class DrawBox<D extends Drawable> {}

DrawBox<? extends String> box;
```

?

NO - String is a final class!

Compiles or Not?

```
class DrawBox<D extends Shape> {}

DrawBox<? extends Thread> box;
```

?

NO - cannot extend two classes

Compiles or Not?

```
class DrawBox<D extends Shape> {}

DrawBox<? super Circle> box;
```

?

YES - Circle is a subclass of Shape

Compiles or Not?

```
class DrawBox<D extends Shape> {}

DrawBox<? super Collection> box;
```

?

NO - Collection is not a subclass of Shape

Conclusion

- Generics are an essential feature of a strongly typed language.
- Allows a type to be parametrized
- Allows a function to be parametrized
- Type erasure in objects at runtime
- Good luck with inheritance

Homeworks

https://github.com/JavaFundamentalsZT/jf-hw-3-generics

Homework task 1

- Implement CardDeck
    - CardDeck skeleton provided
    - Keep state and implement methods
    - Classes for Card, Suit and Rank are provided.
        - add methods to Card if needed

CardDeck

- CardDeck() //constructor
- shuffle()
- take()
- add()
- size()

Homework task 1

- Adding an existing card to a deck is an exception.
- Otherwise adding/removing cards must not throw exceptions
- Implement missing test and add more
- Format your code!

Homework task 2

- create function unique
- receives a varg of List-s
- returns a List of values that exist only once across all collections
- unique([1,2,3], [2, 5, 7], [3, 15, 7])
  - [1, 5, 15]

Homework task 2

- Very easy to solve with stream()
- Also very similar to last week

- **Stream API is not allowed this time!**
- Must use collections' api
  - (and feel the pain)

Homework task 2

```
List<Integer> a = Arrays.asList(1, 2, 3);
List<Integer> b = Arrays.asList(2, 5, 7);
List<Integer> c = Arrays.asList(3, 15, 7);
List<Double> d = Arrays.asList(3.0, 5.0, 7.1);
List<String> s = Arrays.asList("a", "b");

List<Number>compiles = unique(a,b,c,d);
List<Number>wontcompile=unique(a,b,c,d,s);
List<Object>compilesAgain=unique(a,b,c,d,s)
```

Homework task 2

- No pre-made tests available this time
  - challenge is in the API design
- Unit tests are mandatory!
- Format your code!

Deadline

19.02.2017 at 23:59