An introduction to the Java Collections Framework
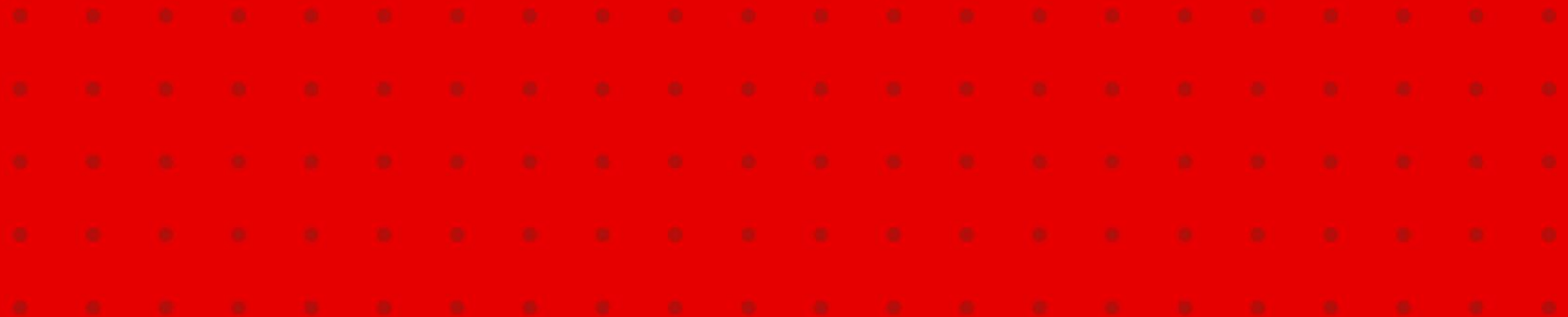
Bogdan ȘTEFAN
Radu HOAGHE
Ciprian PARASCHIVESCU

Outline

1. General concepts
2. Containers in Java
3. Container utility classes

Teamnet

1

General concepts

General concepts

❑ Every programming language makes use of some **base data structures** to assist in developer productivity.

❑ In programming literature these are known as **compound data types** – and are especially useful for dynamicity at run-time.

❑ They are split into three categories, which we'll henceforth call *containers*:
1. Tuples
2. Lists
3. Dictionaries

TEAMNET

2

Containers… the Java way
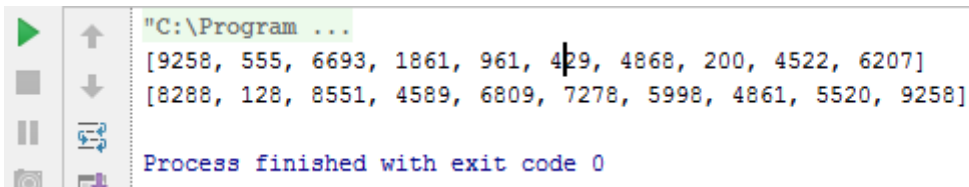
TeαmneT

2

But first, a few words about Tuples! ☺

❑ They represent **ordered** (not sorted!) sequences of elements.

❑ They are **immutable** (i.e. they cannot be changed at element level).

❑ **Not part of Java**, by default.

❑ Their purpose: ???   Let's find out… ☺

TΣAMNΣT

Problem formulation:

```java
public static void main(String[] args) {
    Sample samplesA = generateRandomSample(10);
    Sample samplesB = generateRandomSample(10);

    System.out.println(samplesA);
    System.out.println(samplesB);
}
```

Given multiple batches of experimental data

```
"C:\Program ...
[9258, 555, 6693, 1861, 961, 429, 4868, 200, 4522, 6207]
[8288, 128, 8551, 4589, 6809, 7278, 5998, 4861, 5520, 9258]

Process finished with exit code 0
```

TEAMNET

Problem formulation:

```java
public static void main(String[] args) {
    Sample samplesA = generateRandomSample(10);
    Sample samplesB = generateRandomSample(10);

    System.out.println(samplesA);
    System.out.println(samplesB);
}
```

```
"C:\Program ...
[9258, 555, 6693, 1861, 961, 429, 4868, 200, 4522, 6207]
[8288, 128, 8551, 4589, 6809, 7278, 5998, 4861, 5520, 9258]
samplesA(min, max): (200, 9258)
samplesB(min, max): (128, 9258)

Process finished with exit code 0
```
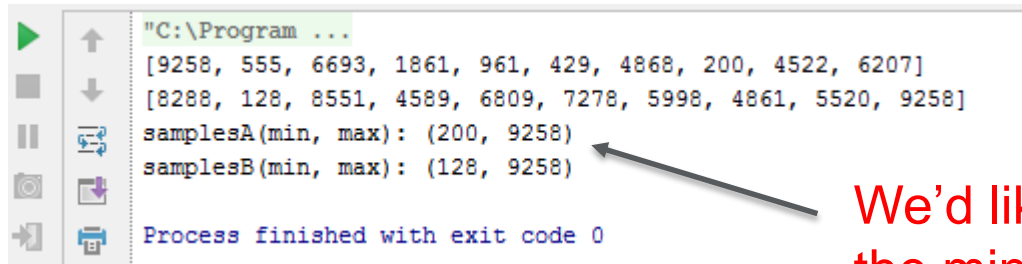
We'd like *to* compute both the minimum
and the maximum…

using a single method!

TEAMNET

We define a ***2-Tuple*** to hold our data:

```java
public class TwoTuple<A, B> {
    public final A first;
    public final B second;

    public TwoTuple(A first, B second) {
        this.first = first;
        this.second = second;
    }

    public String toString() { return "(" + first + ", " + second + ")"; }
}
```

public access, immutable holders for
1st and 2nd element

In the future, we might need 3 items? No problem!

```java
public class ThreeTuple<A, B, C>
        extends TwoTuple<A, B> {

    public final C third;

    public ThreeTuple(A first, B second, C third) {
        super(first, second);
        this.third = third;
    }

    public String toString() {
        return "(" + first + ", "
                + second + ", "
                + third + ")";
    }
}
```

adding a 3rd *holder* item using inheritance;

We now have a 3-Tuple

TΣΩMΠΣT

We define a ***2-Tuple*** to hold our data:

```java
public class TwoTuple<A, B> {
    public final A first;
    public final B second;

    public TwoTuple(A first, B second) {
        this.first = first;
        this.second = second;
    }

    public String toString() { return "(" + first + ", " + second + ")"; }
}
```

public access, immutable holders for
1st and 2nd element

Build our algorithm*, using above type

```java
public static TwoTuple<Integer, Integer>
computeBatchCharacteristics(Sample sampleBatch) {
    // Compute minimum and maximum
    return tuple(min(sampleBatch),  // and return
                max(sampleBatch)); // as a 2-tuple
}
```

return a 2-tuple

TEAMNET

❑ They represent ordered (not sorted!) sequences of elements.

❑ They are immutable (they cannot be changed at element level).

❑ **Not part of Java JDK**, by default.

❑ Their purpose: they allow **multi-return** in methods/functions.

TEAMNET

2.1

Arrays in Java

Array(s)

❑ The most basic (primitive) "containers" of any statically typed programming language.

Declaration (two alternatives):

```
// Declaration through initializer
int[] arrayOfIntegers = new int[] { 1, 3, 5, 7, 9, };
```

type
variable name
"new" operator
type[desired_size]
"array initializer"

Setting values explicitly:

```
// Explicitly setting values
arrayOfIntegers[0] = 1; // Notice: the first entry always starts at position '0' !!!
arrayOfIntegers[1] = 3;
```

explicit position a.k.a. "the array **Index**"
explicit value

Teαmnet

- **Array**(s) – Adding and retrieving values

Adding values (most often done way):

```
// Automate addition by iterating over array
for (int i = 2; i < arrayOfIntegers.length; i++) {
    // Double the value and set on explicit position
    arrayOfIntegers[i] = i * 2;

    // Other processing steps
    // could follow here
    // ...
}
```

"length" (the only) **property** is always available

Retrieval/accessing (explicit):

```
// Retrieving values explicitly
int firstValue = arrayOfIntegers[0]; // Access first value, save its reference value
int secondValue = arrayOfIntegers[1]; // Access second value, save its reference value
```

❏ They offer the best **random access performance***
compared with any other containers (for both *addition*
and *retrieval* of data).

TEAMNET

Array(s) – Further notes on retrieval

Retrieval (*classic* vs. *foreach* iteration):

```java
// --- Does it contain number '5'?
// A flag to denote discovery
boolean containsFive = false;
// Automate retrieval by iterating over array
for (int arrayOfInteger : arrayOfIntegers) {
    // Validate each retrieved value against '5'
    if (arrayOfInteger == 5) {
        // Set flag to true
        containsFive = true;
        break; // No need to proceed any further
    }
}
// Print conclusion
System.out.println(containsFive ? "Five's in here!" : "Sorry buddy, no five for you!");
```

elem

Using "foreach"

Simpler access to elements, instead of "*i* (ndex)"-based values
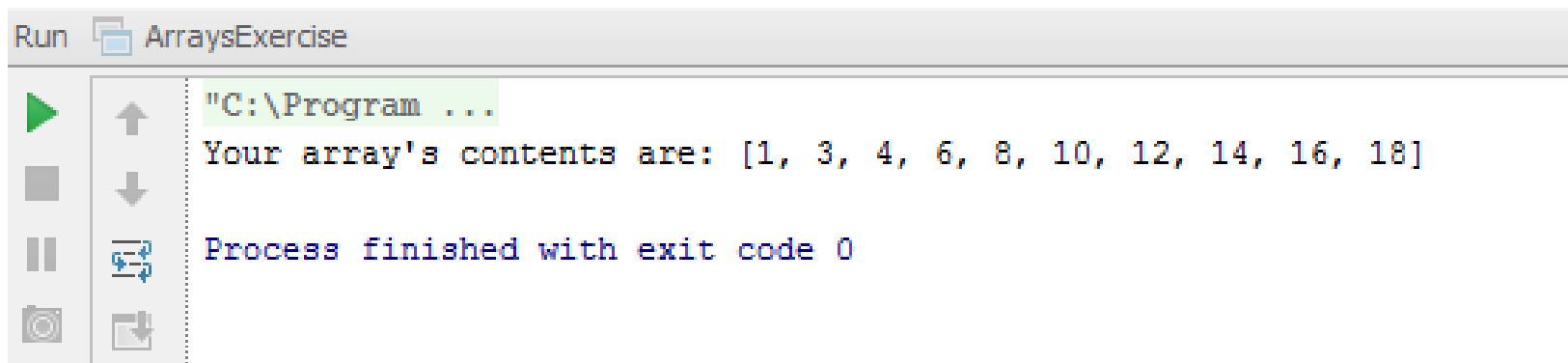
TEAMNET

Array(s) – Printing

Printing (user friendly way):

```java
// Finally, let's print it out
System.out.println("Your array's contents are: " +
        java.util.Arrays.toString(arrayOfIntegers));
```

A-ha, what's this?!

First contact with container utilities! ☺

Print results:

```
Run    ArraysExercise

▶      ⬆      "C:\Program ...
              Your array's contents are: [1, 3, 4, 6, 8, 10, 12, 14, 16, 18]
■      ⬇

II     ⇥      Process finished with exit code 0

◎      ⬒
```

TEAMNET

Array(s) – *"99 problemz"* ☺

❑ They are really really fast; specifically, they provide an *efficient* performance, however at a *low-level*.

❑ They do not play well with *generics*; actually, it is more accurate to state that "*generics are fairly hostile against arrays*."[1]

❑ Their **main issues**?

They (must) have a (pre-known) fixed `size`.

Generally *very expensive to expand*, to hold *other items* (think at BIG scale things!).

TEAMNET

[1]: Arrays p.p. 535, *Thinking in Java (4th Edition)*, Bruce Eckel

❑ What if we want to deal with any known number of "items", dynamically at run-time?

Think in terms of

"*dynamic memory management*"…

TEAMNET

OMG!!! Omagawd!!!

What do we do?!

What… do we… doo?!!

❑ What if we want to deal with any known number of "items", dynamically at run-time?

❑ What if we had some kind of utility that could hold elements and expand in *a natural sort of way*, if needed?

How about we take a look at what's inside the `java.util` **package**?

TEAMNET

The java.util "toolbox"

Here's an overview of the most often used Java containers:
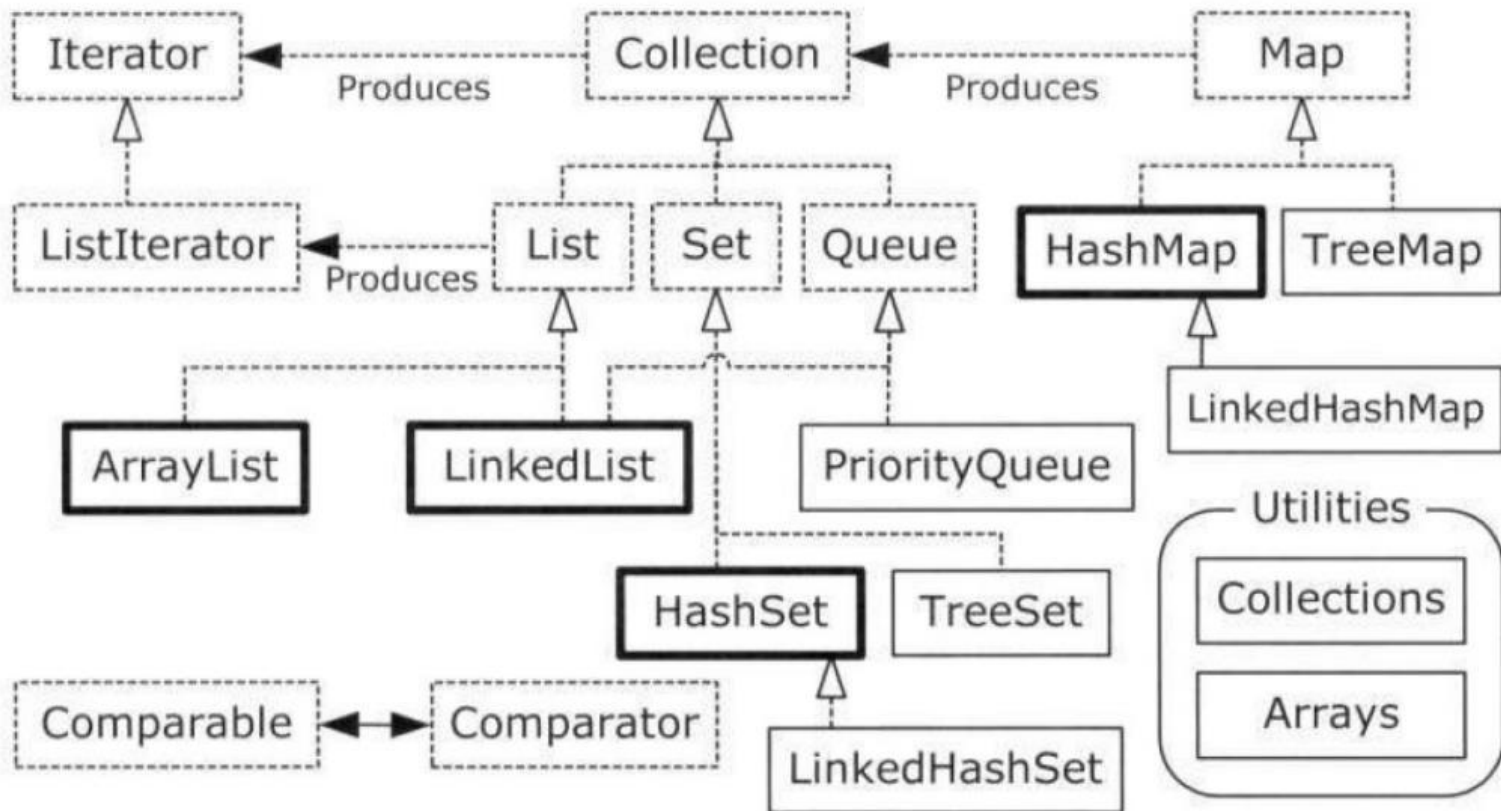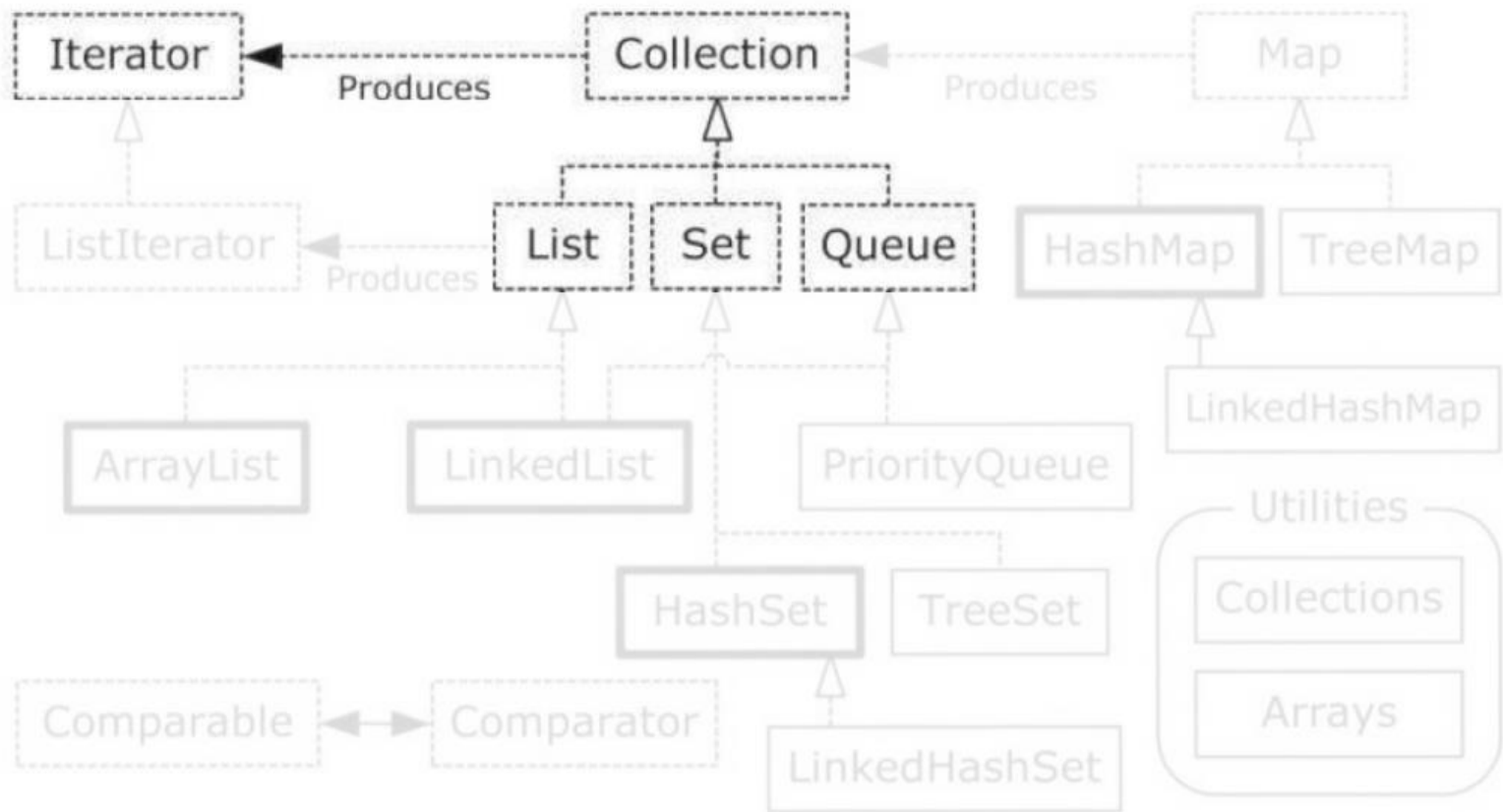
A first word about Java containers

Categories:

1. **Collections** – *sequences* which can hold **individual** elements based on one or more rules.

2. **Maps** – a group of **associated pairs** of elements (also known as a *dictionary*, in programming literature).

Container utilities: `java.util.`**`Arrays`** & **`Collections`** classes

тєαмпєт

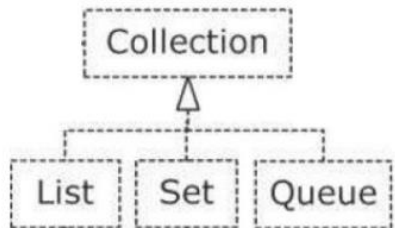Image source: Thinking in Java (4th Edition), Bruce Eckel

java.util.Collection

Collection

The basic single-item containers in Java are known as **collections**.

The `Collection` interface generalizes the idea of a sequence – a way of holding a group of objects.

Crudely put, a collection is a **container** that can **hold** any number of **objects** (possibly taking into account some *rules*).

TEAMNET

Image source: Thinking in Java (4th Edition), Bruce Eckel

java.util.Collection

Collection

Why would one use such data structures?

Advantages:

- No expansion limits - theoretically.

  (making them *perfect* for dynamic memory management)

Disadvantages:

- None

  (well, sort of - because they are *task specific* - this illustrates that they have *weaknesses of their own*, which you need to be aware of) ☺

TEAMNET

java.util.Collection(s)

Java collections can *initially* be split into:

- Lists
- Sets
- Queues

(these are all just *interfaces*)

Again, each comes with strengths and weaknesses, and is suitable for a specific task, as we'll see next.

TEAMNET

Short Quiz:
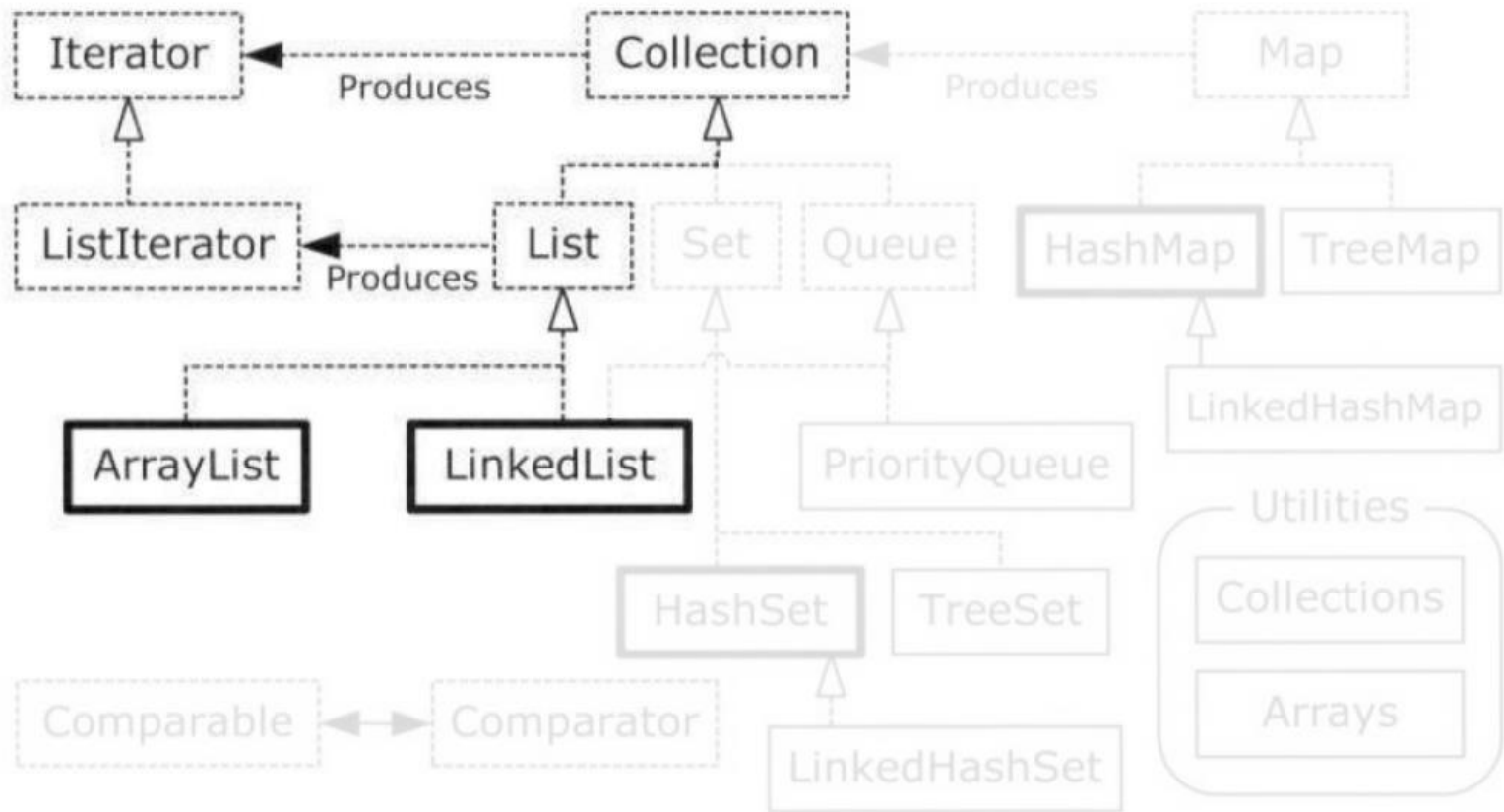
```java
public void omg(int w00t) {

    try {
        this.mustStillPayAttention(w00t);
    } catch (BrainMeltException o_O) {
        System.err.println(
                        o_O.getMessage());
    }
}

    "Let's take a 5 minute break, guys!" ☺
```

2.2

Lists in Java

Image source: Thinking in Java (4th Edition), Bruce Eckel

❑ They can hold single elements/objects.

❑ They **allow** duplicates to be inserted.

❑ They are **ordered**, by default (**not sorted** – careful here!).

❑ Adequate for **LIFO** and **FIFO** behavior (as **stack**s & **queue**s – later on this).

TeαmneT

Given the following:

```java
class Motherboard {

    private final String serialNumber;

    public Motherboard() { this.serialNumber = generateSerialNumber("MBD"); }

    public void listPartDetails() {
        System.out.println("I'm a " + this.getClass().getSimpleName()
                + "\nS/N: " + this.serialNumber);
    }
}

class CPU {

    private final String serialNumber;

    public CPU () {
        this.serialNumber = generateSerialNumber("CPU");
    }

    public void listPartDetails() {
        System.out.println("I'm a " + this.getClass().getSimpleName()
                + "\nS/N: " + this.serialNumber);
    }
}
```

Let's put them into practice:

```java
@SuppressWarnings("unchecked")
public static void main(String[] args) {

    // A quick declaration
    ArrayList partsList = new ArrayList();
    // Add some parts to our list
    partsList.add(new CPU());
    partsList.add(new CPU());
    partsList.add(new Motherboard());

    for (int i = 0; i < partsList.size(); i++) {
        // Retrieve and cast to CPUs
        ((CPU)partsList.get(i)).listPartDetails();
    }

}
```
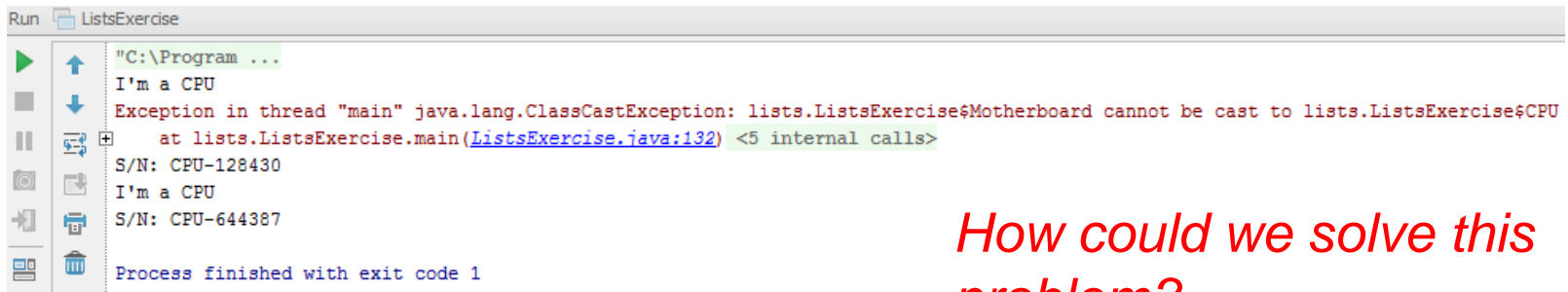
Simple declaration

Adding elements

Explicit retrieval by item
Index

```
Run    ListsExercise
  ▶   ↑    "C:\Program ...
           I'm a CPU
  ■   ↓    Exception in thread "main" java.lang.ClassCastException: lists.ListsExercise$Motherboard cannot be cast to lists.ListsExercise$CPU
  ‖   ⊞        at lists.ListsExercise.main(ListsExercise.java:132) <5 internal calls>
           S/N: CPU-128430
  ◎   ⊡    I'm a CPU
  ⇥   ⎙    S/N: CPU-644387
  ⊞   �🗑
           Process finished with exit code 1
```

How could we solve this problem?

TEAMNET

Fix by adding a rule: establish *bounds*

```java
@SuppressWarnings("unchecked")
public static void main(String[] args) {

    // A quick declaration
    ArrayList partsList = new ArrayList();
    // Add some parts to our list
    partsList.add(new CPU());
    partsList.add(new CPU());
    partsList.add(new Motherboard());

    for (int i = 0; i < partsList.size(); i++) {
        // Retrieve and cast to CPUs
        ((CPU)partsList.get(i)).listPartDetails();
    }

}
```

```java
@SuppressWarnings("unchecked")
public static void main(String[] args) {

    // A bounded list (can hold only CPU)
    ArrayList<CPU> partsList = new ArrayList<CPU>();
    // Add some parts to our list
    partsList.add(new CPU());
    partsList.add(new CPU());
    // ! partsList.add(new Motherboard()); // Not allowed anymore

    for (CPU part : partsList) {
        // Easier retrieval as well
        part.listPartDetails();
    }

}
```

```
"C:\Program ...
I'm a CPU
S/N: CPU-307516-CP815915-
I'm a CPU
S/N: CPU-859552-CP59231-

Process finished with exit code 0
```

Hey, what about the poor Motherboard? ☹

TEAMNET

Easy fix: *lowering* the *bounds,* through *polymorphism*

```
public static void main(String[] args) {

    // A bounded list (can hold any Part)
    ArrayList<Part> partsList = new ArrayList<Part>();
    // Add some parts to our list
    partsList.add(new CPU());
    partsList.add(new CPU());
    partsList.add(new Motherboard()); // Allowed now
    partsList.add(new Motherboard());

    for (Part part : partsList) {
        // Easier retrieval as well
        part.listPartDetails();
    }
}
```

Both **CPU** and **Motherboard** are *some kind* of **Part**

```
"C:\Program ...
I'm a CPU
S/N: CPU-385229-CP62893-
I'm a CPU
S/N: CPU-73361-CP811429-
I'm a Motherboard
S/N: MBD-36526-CP22908-
I'm a Motherboard
S/N: MBD-603444-CP97315-

Process finished with exit code 0
```

TEAMNET

java.util.List(s)

Most often used **Lists** are:
- ArrayList
- LinkedList
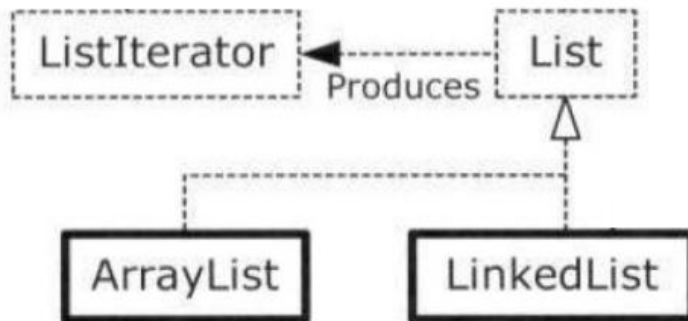
Legacy:
- Vector

(may be old school, but it offered thread-safety – now replaced by CopyOnWriteArrayList)

When and why would one use such data structures?

TEAMNET

java.util.ArrayList

The most basic type of sequence.

Excels at **random**ly **access**ing elements.

The drawback: **slower** when **insert**ing elements in the **middle**.

ListIterator ◄ ---- List
Produces

ArrayList   LinkedList

java.util.LinkedList

A general purpose sequence: can be used as a **stack**, as a **queue** and **double-ended queue**.

Larger feature set than an `ArrayList`.

Best for *sequential* access; **inexpensive insertions** and **deletions** in the middle.

The drawback: **slow** for **random access**.

ListIterator ◄----- Produces ----- List

ArrayList   LinkedList

TEAMNET

The most **common operations** you will do with/on a **List** are:

- `add(obj)` (at the end)
- `addAll(collection)`
- `insert(atPosition)`
- `contains(obj)`
- `get(position)`
- `remove(position/obj)`
- `iterator()`
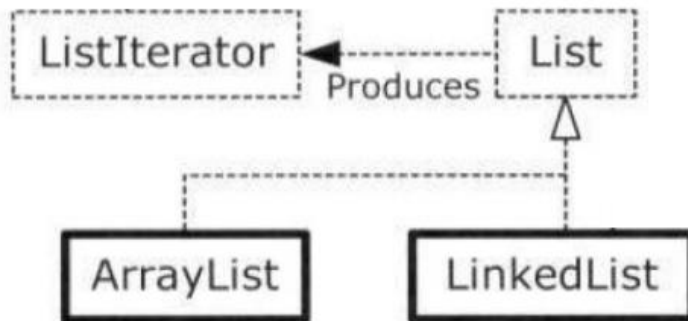
2.2.1

A case for Iterators

java.util.Iterator
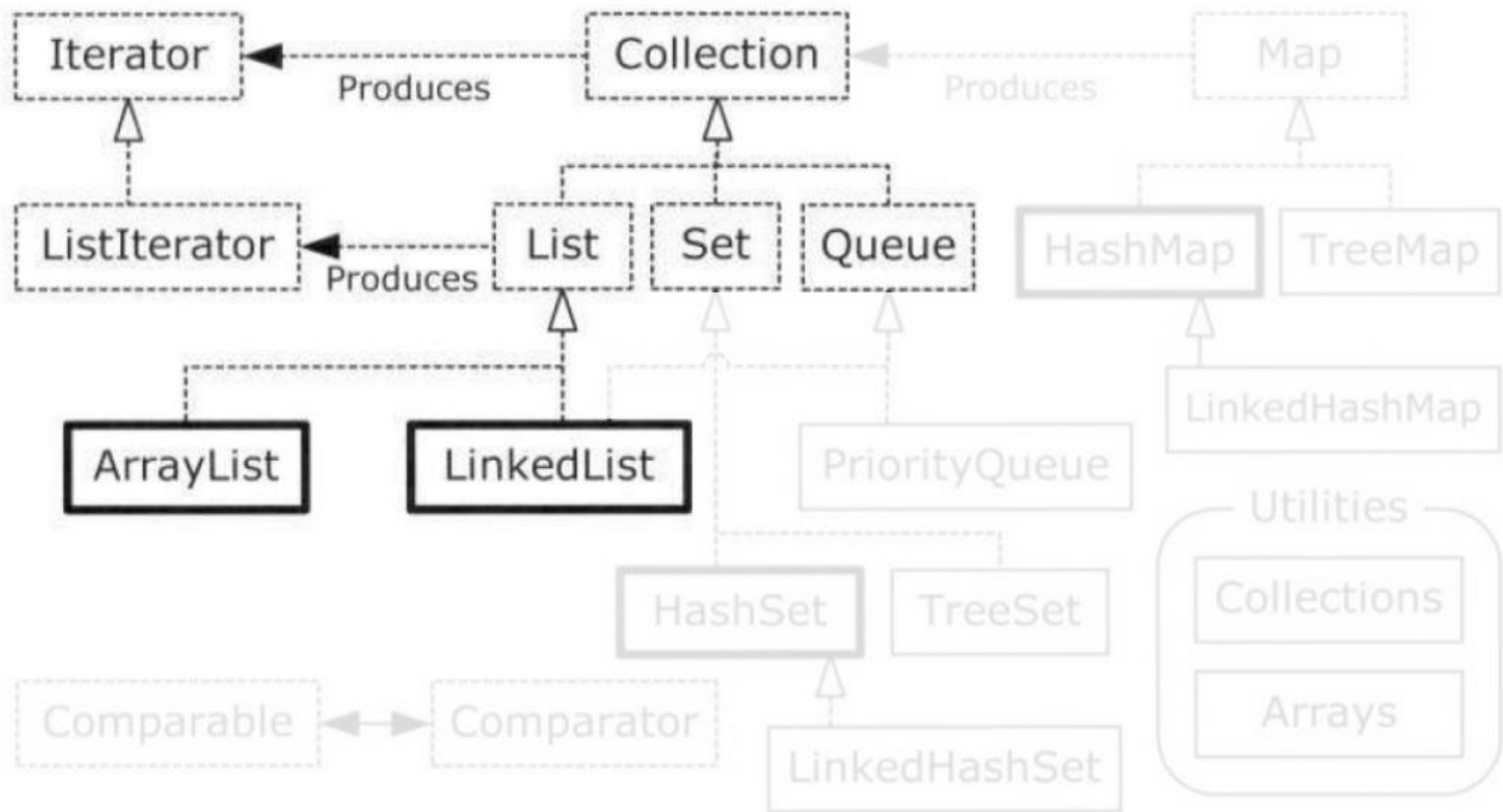
java.util.Iterator – Notes (1)

❑ Any container must be able to accept as well as retrieve items.

(But you could say: well, we have **add()** and **get()** for exactly that.)

❑ However, the idea is to think at a higher-level, and thus, there is a drawback using the previous approach: **you need to program to the exact type of container**.

(What if we write code for a `List` and later decide it would apply to a `Set` as well – since both are collections after all ?)

(Or what if, we want, from the beginning, to write general purpose code that applies to every collection, no matter its type?)

❑ The concept of an `Iterator` (a design pattern) can be used to achieve this abstraction.

Teɑmnet

❑ An **iterator** is a *lightweight object* that **moves** through a **sequence**.

❑ It **selects each element** of that **sequence** without having the programmer worry about the underlying type's specific access logic (i.e. enforces *loose coupling*).

❑ A usual interaction with an iterator would look like:
1. Ask a `Collection` for an `Iterator`, by calling `.iterator()`
2. Get the next object in the sequence using `.next()`
3. See if there are more elements with `.hasNext()`
4. Remove the last element returned using `.remove()`

TEAMNET

```java
public static void main(String[] args) {
    List<Pet> pets = Pets.arrayList(12);
    // Iteration via iterator
    Iterator<Pet> it = pets.iterator();
    while (it.hasNext()) {
        Pet p = it.next();
        System.out.print(p.id() + ":" + p + " ");
    }
    System.out.println();
    // A simpler approach, when possible:
    for (Pet p : pets)
        System.out.print(p.id() + ":" + p + " ");
    System.out.println();
    // An Iterator can also remove elements:
    it = pets.iterator();
    for (int i = 0; i < 6; i++) {
        it.next();
        it.remove();
    }
    System.out.println(pets);
}
```

ask for the collection's Iterator

if there are elements in the sequence

retrieve an element

use *foreach* when *reading*

remove the current element

TEAMNET

java.util.Iterator – A (better) typical use case

```java
public class CrossContainerIteration {
    public static void display(Iterator<Pet> it) {
        while (it.hasNext()) {
            Pet p = it.next();
            System.out.print(p.id() + ":" + p + " ");
        }
        System.out.println();
    }

    public static void main(String[] args) {
        ArrayList<Pet> pets = Pets.arrayList(8);
        LinkedList<Pet> petsLL = new LinkedList<Pet>(pets);
        HashSet<Pet> petsHS = new HashSet<Pet>(pets);
        TreeSet<Pet> petsTS = new TreeSet<Pet>(pets);
        display(pets.iterator());
        display(petsLL.iterator());
        display(petsHS.iterator());
        display(petsTS.iterator());
    }
}
```
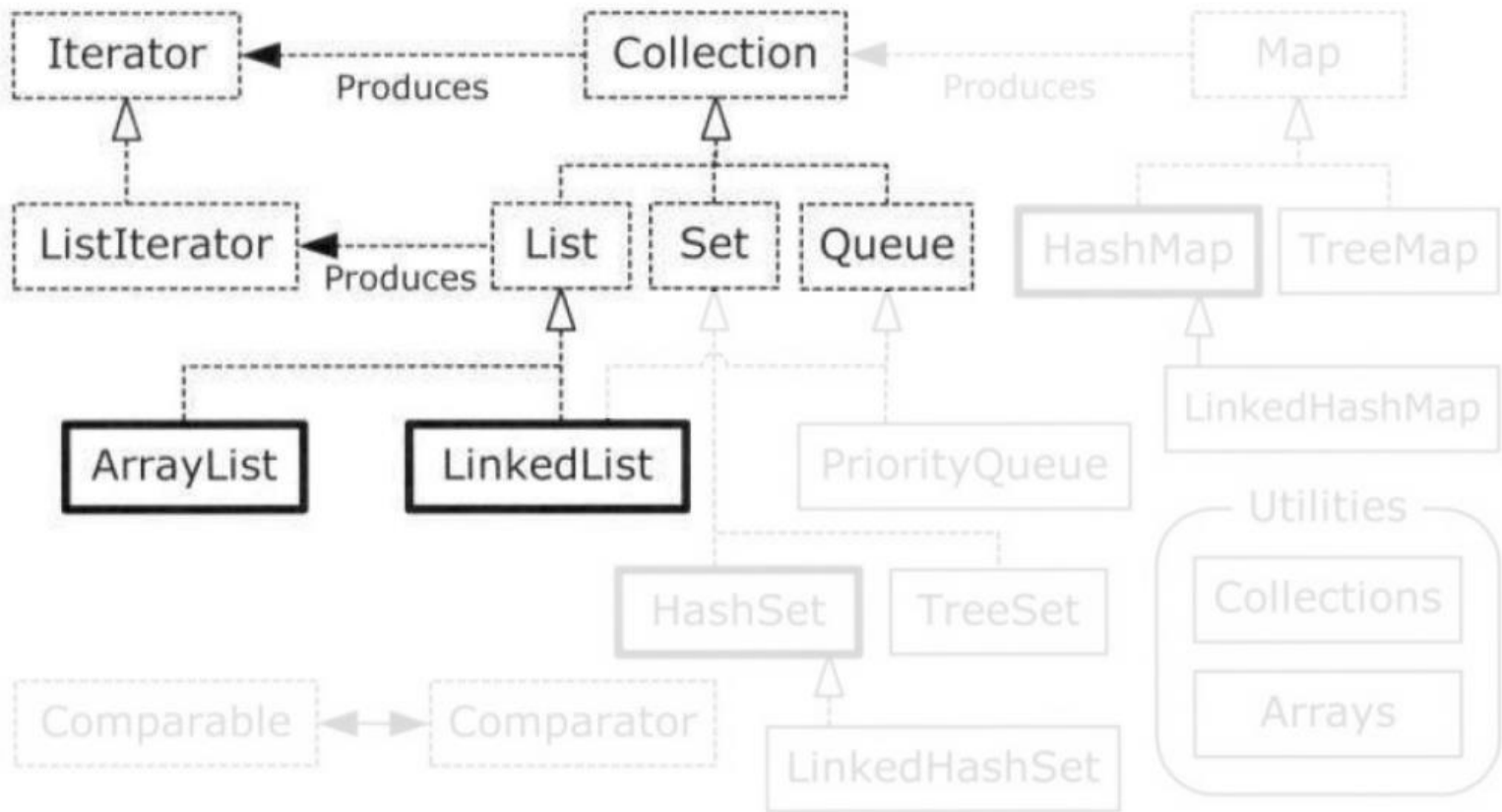
if there are elements in the sequence

retrieve an element via next()

hand-over each container's Iterator

TEAMNET

Image source: Thinking in Java (4th Edition), Bruce Eckel

java.util.ListIterator

❑ A more *powerful* iterator produced only by List implementations.

❑ Apart from the forward version of the general implementation, a `ListIterator` is **bidirectional**; traversal can be done both ways.

❑ Can also produce **indexes** of the **next** and **previous** elements, relative to where the iterator is pointing in the list.

❑ It can replace the last element visited, using the `set()` method.

TEAMNET

java.util.ListIterator – Quick example

```java
public static void main(String[] args) {
    List<Pet> pets = Pets.arrayList(8);
    ListIterator<Pet> it = pets.listIterator();
    while (it.hasNext())
        System.out.print(it.next() + ", " + it.nextIndex() +
                ", " + it.previousIndex() + "; ");
    System.out.println();
    // Backwards:
    while (it.hasPrevious())
        System.out.print(it.previous().id() + " ");
    System.out.println();
    System.out.println(pets);
    it = pets.listIterator(3);
    while (it.hasNext()) {
        it.next();
        it.set(Pets.randomPet());
    }
    System.out.println(pets);
}
```

ask for the collection's Iterator

forward *facing*

access indexes

reverse direction

change current iterator element using set()

2.3

Sets in Java

Image source: Thinking in Java (4th Edition), Bruce Eckel
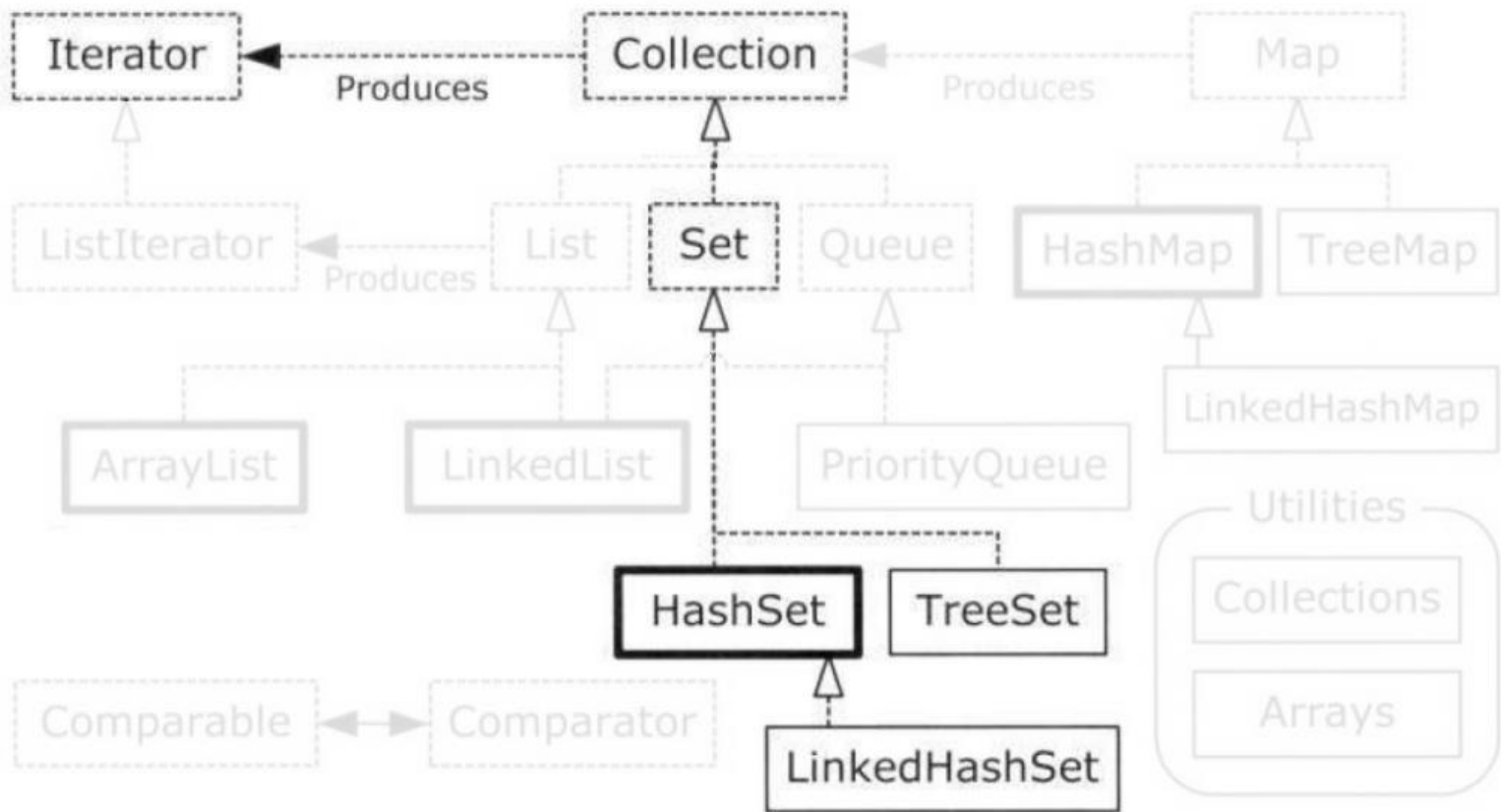
java.util.Set(s) – Notes
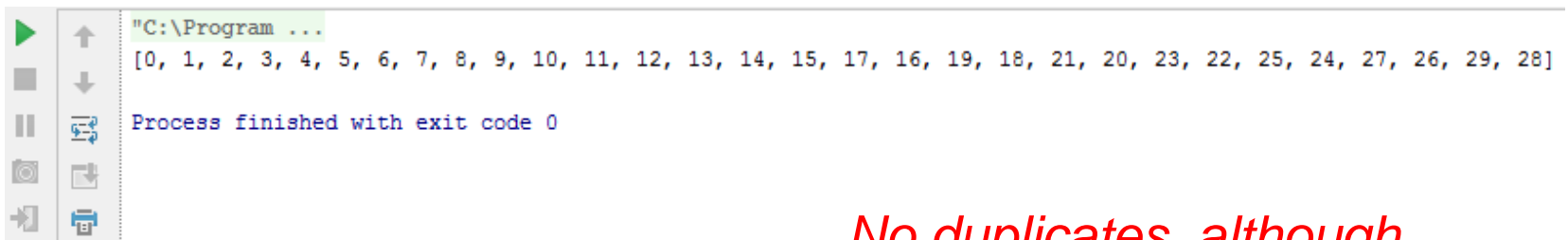
❑ Like lists, they can hold single elements.

❑ They **DO NOT** allow duplicates.

❑ Used for *querying* held elements, via `contains(obj)` method (e.g. *test for membership*).

❑ Because of this, **lookup** is typically the most **important** operation for a `Set`.

java.util.Set(s) – Quick example

```java
public static void main(String[] args) {
    Random rand = new Random(47);

    Set<Integer> intSet = new HashSet<Integer>();

    for (int i = 0; i < 10000; i++)
        intSet.add(rand.nextInt(30));
    System.out.println(intSet);

}
```

typical declaration

add an integer
between 0 and 30

```
"C:\Program ...
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 17, 16, 19, 18, 21, 20, 23, 22, 25, 24, 27, 26, 29, 28]

Process finished with exit code 0
```

*No duplicates, although
10,000 integers were added.*

TEAMNET

Collection
↑
Set
↑
HashSet   TreeSet
↑
LinkedHashSet
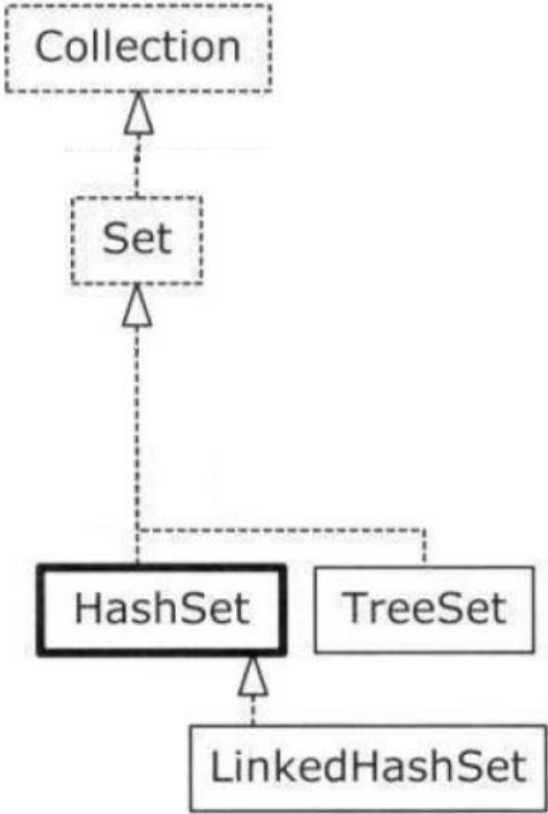
Sets are available in many flavors. The **three** most used are:

- HashSet
- LinkedHashSet
- TreeSet

When and why would one use such data structures?

Image source: Thinking in Java (4th Edition), Bruce Eckel
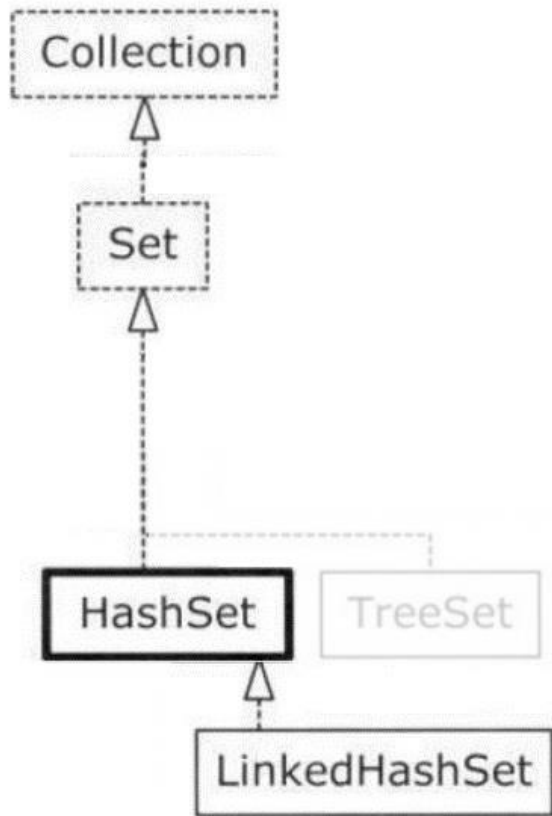
Collection → Set → HashSet, TreeSet, LinkedHashSet
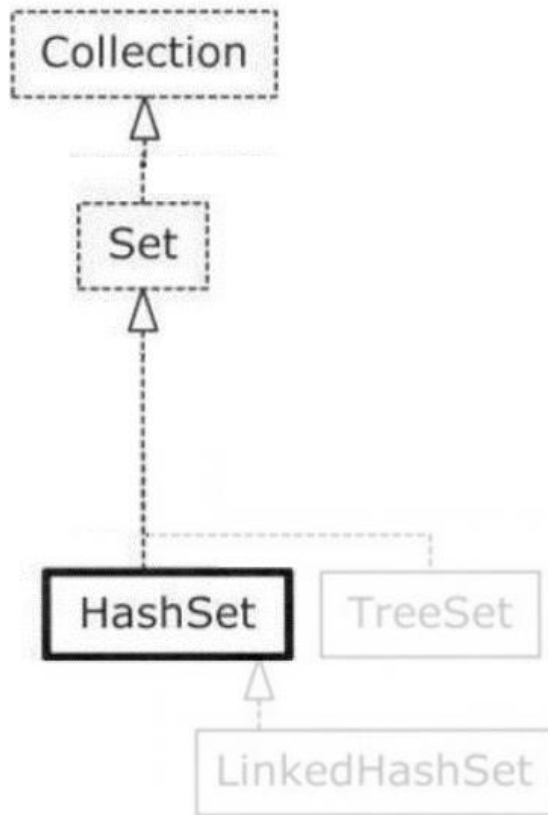
Used when **fast lookup time** is important.

Utilizes a *hashing function* for speed.
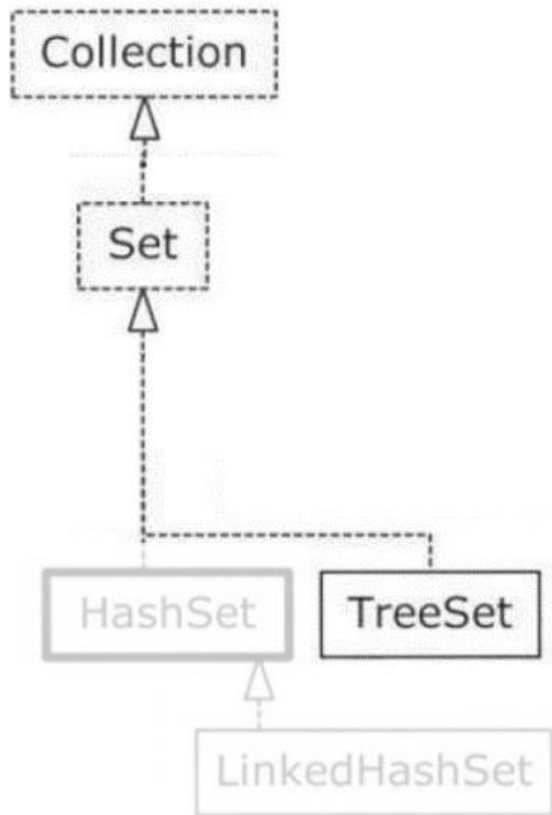
Because of this elements appears to be in no particular order.

TEAMNET

Typically as fast as `HashSet`, in matters of lookup speed.

Elements held **appear to be ordered** based on **insertion order**.

This is because the ordering is based on an **underlying linked list.**

Totally different paradigm than the previous two.

An importance is placed strictly on the principle of sorting of elements.

Sorting is made possible because of the underlying data structure: a **red-black tree**.

TEAMNET

Image source: Thinking in Java (4th Edition), Bruce Eckel
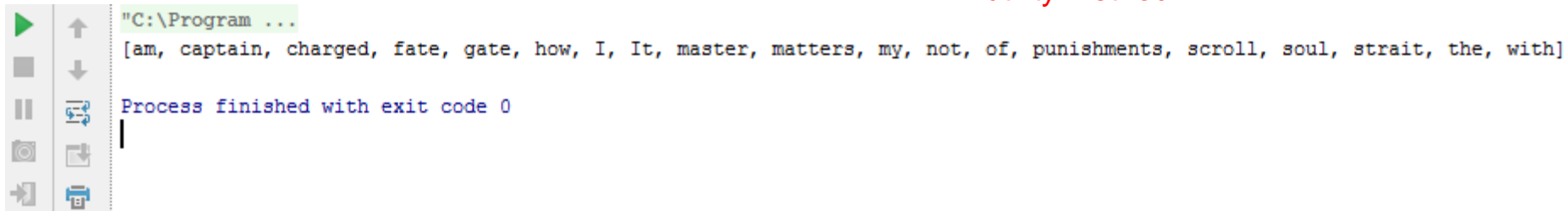
```java
public class SortedSetOfStrings {

    private static final  String poem =
            "It matters not how strait the gate,\n"+
            "How charged with punishments the scroll.\n"+
            "I am the master of my fate:\n"+
            "I am the captain of my soul.";

    public static void main(String[] args) {
        SortedSet<String> words =
                new TreeSet<String>(String.CASE_INSENSITIVE_ORDER);
        words.addAll(Arrays.asList(poem.split("\\W+")));
        System.out.println(words);

    }
}
```
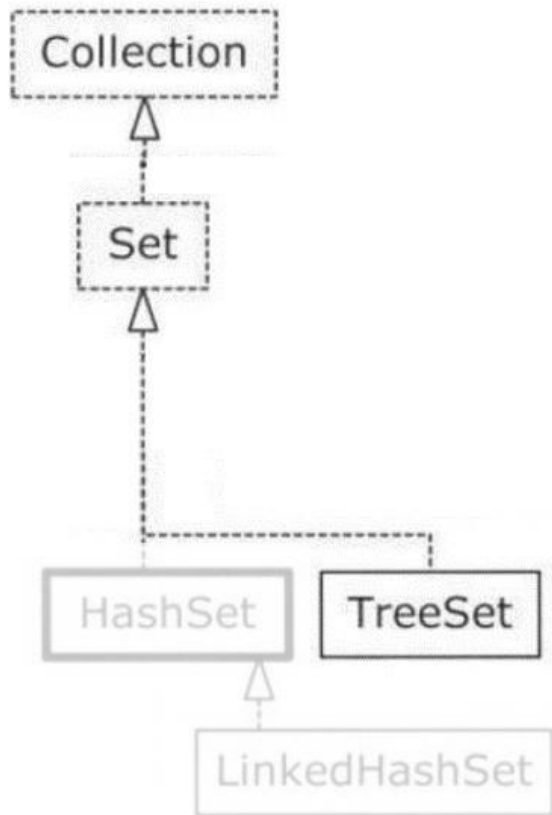
Notice the use of `SortedSet` interface

A comparator is given; not mandatory

Quick collection building via `Arrays.asList(T… a)` utility method

```
"C:\Program ...
[am, captain, charged, fate, gate, how, I, It, master, matters, my, not, of, punishments, scroll, soul, strait, the, with]

Process finished with exit code 0
```
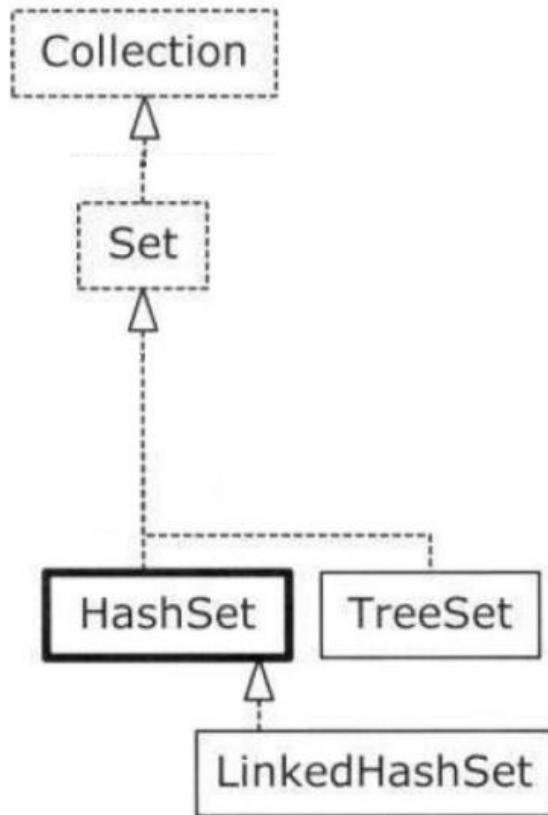
Thus, the elements in a `SortedSet` are guaranteed to be in **sorted order**.

This allows for the following interesting methods:

- `first()`
- `last()`
- `subSet(from, to)`
- `headSet(uptoElement)`
- `tailSet(fromElement)`

TeɑmneT

Image source: Thinking in Java (4th Edition), Bruce Eckel

The most **common operations** you will do with/on a **Set** are:

- add(obj)
- addAll(collection)
- contains(obj)
- iterator()
- remove(obj)

Image source: Thinking in Java (4th Edition), Bruce Eckel

java.util.Set(s) – Conclusions

❑ A `Set` **only accepts one** of each type of objects (no duplicates!).

❑ **Automatic resizing** to accommodate new items, if needed.

❑ `HashSet(s)` are best used for **fast lookup time**.

❑ `LinkedHashSet(s)` have similar lookup time, and maintain an **order** based on **insertion**.

❑ `TreeSet(s)` are a breed apart, focusing on a **sorting order** for held elements.

Teamnet

3

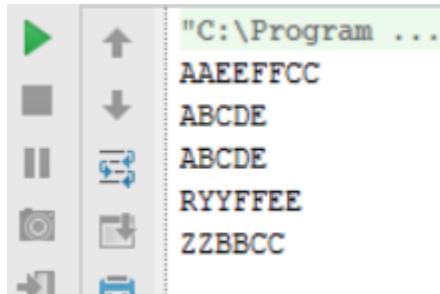## Container utility classes

Teamnet

java.util.Arrays

The most **common operations** you will do with the `java.util.Arrays` utility class are:

- `asList(T... a)`
- `toString(Object [] a)`
- `sort(Object [] a)`
- `fill(Object[] a, Object val)`

Sort an Array

```java
public class Main {

    public static void main(String[] args) {
        String[] stringArray = {"ZZBBCC", "ABCDE", "AAEEFFCC", "RYYFFEE", "ABCDE"};
        Arrays.sort(stringArray);

        for (String s : stringArray) {
            System.out.println(s);
        }

    }
}
```

```
"C:\Program ...
AAEEFFCC
ABCDE
ABCDE
RYYFFEE
ZZBBCC
```

TEAMNET

Fill an Array

```java
public class Main {

    public static void main(String[] args) {
        int arr[] = new int[] {1, 6, 3, 2, 9};
        // let us print the values
        System.out.println("Actual values: ");
        for (int value : arr) {
            System.out.println("Value = " + value);
        }
        // using fill for placing 18

        Arrays.fill(arr, 18);

        // let us print the values
        System.out.println("New values after using fill() method: ");
        for (int value : arr) {
            System.out.println("Value = " + value);
        }
    }
}
```

```
"C:\Program ...
Actual values:
Value = 1
Value = 6
Value = 3
Value = 2
Value = 9
New values after using fill() method:
Value = 18
Value = 18
Value = 18
Value = 18
Value = 18
```

The most **common operations** you will do with the `java.util.Collections` utility class are:

- `addAll(Collection c, T... elements);`
- `sort(List<?> list)`
- `shuffle(List<?> list)`
- `reverse(List<?> list)`
- `max(Collection<?> list)`
- `min(Collection<?> list)`
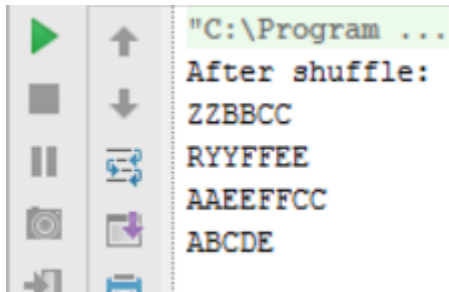- `frequency(Collection<?> c, Object o)`

Teαmneτ

Sort a List

```java
public class Main {

    public static void main(String[] args) {
        List<String> list = new ArrayList<String>();
        list.add("ZZBBCC");
        list.add("ABCDE");
        list.add("AAEEFFCC");
        list.add("RYYFFEE");

        Collections.sort(list);

        System.out.println("After sort: ");
        for(String s:list){
            System.out.println(s);
        }
    }
}
```
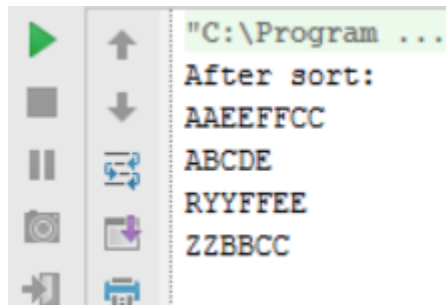
```
"C:\Program ...
After sort:
AAEEFFCC
ABCDE
RYYFFEE
ZZBBCC
```

TEAMNET

Shuffle a List

```java
public class Main {

    public static void main(String[] args) {
        List<String> list = new ArrayList<String>();
        list.add("ZZBBCC");
        list.add("ABCDE");
        list.add("AAEEFFCC");
        list.add("RYYFFEE");

        Collections.shuffle(list);

        System.out.println("After shuffle: ");
        for (String s : list) {
            System.out.println(s);
        }
    }
}
```

```
"C:\Program ...
After shuffle:
ZZBBCC
RYYFFEE
AAEEFFCC
ABCDE
```

TEAMNET

Reverse a List

```java
public class Main {

    public static void main(String[] args) {
        List<String> list = new ArrayList<String>();
        list.add("ZZBBCC");
        list.add("ABCDE");
        list.add("AAEEFFCC");
        list.add("RYYFFEE");

        Collections.reverse(list);

        System.out.println("After reverse: ");
        for (String s : list) {
            System.out.println(s);
        }
    }
}
```

```
"C:\Program ...
After reverse:
RYYFFEE
AAEEFFCC
ABCDE
ZZBBCC
```

TEAMNET
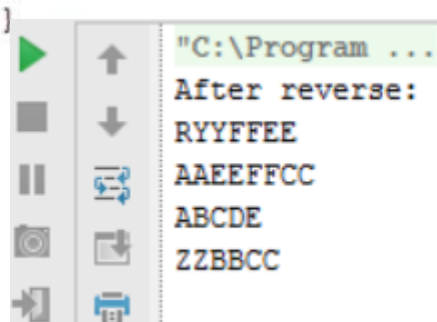
Max/Min from a Collection/List

```java
public class Main {

    public static void main(String[] args) {
        List<Integer> list = new ArrayList<Integer>();
        list.add(4);
        list.add(12);
        list.add(2);
        list.add(10);
        list.add(5);

        Integer maxName=Collections.max(list);
        System.out.println("Max from list is: "+maxName);

        Integer minName=Collections.min(list);
        System.out.println("Min from list is: "+minName);

    }
}
```

```
"C:\Program ...
Max from list is: 12
Min from list is: 2
```

Teαmnet
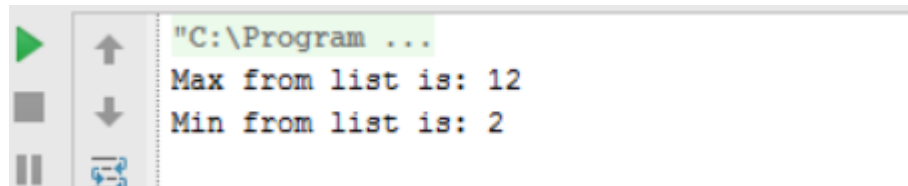
Frequency from a list

```java
public class Main {

    public static void main(String[] args) {
        List<String> list = new ArrayList<String>();
        list.add("ZZBBCC");
        list.add("ABCDE");
        list.add("AAEEFFCC");
        list.add("RYYFFEE");
        list.add("ABCDE");

        Integer frequency=Collections.frequency(list,"ABCDE");

        System.out.println("Frequency is "+frequency);
    }
}
```

```
"C:\Program ...
Frequency is 2
```
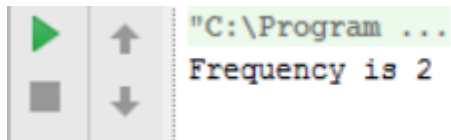
Teamnet

Thank you!

Questions or comments on these topics and more, are welcome:

Bogdan.ȘTEFAN,
Radu.HOAGHE,   @teamnet.ro
Ciprian.PARASCHIVESCU

We salute you! ☺