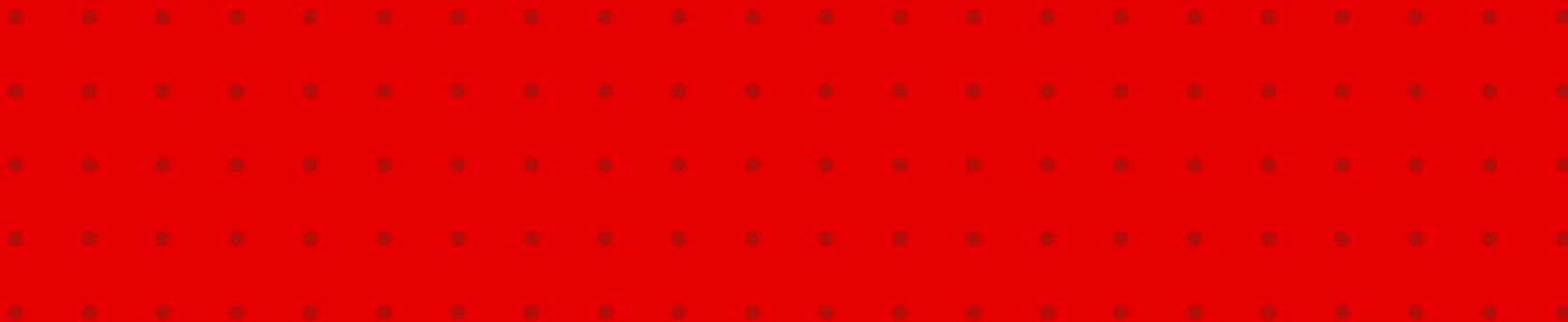




An introduction to the Java Collections Framework



Bogdan ȘTEFAN
Radu HOAGHE

@teamnet.ro

- •
- **Outline** •
- •

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

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

2

Containers... the Java way

2

But first, a word about Tuples!

•
• **Tuples** •
• •

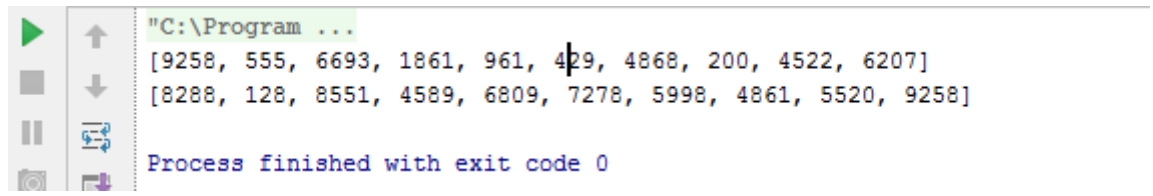
- ❑ They represent **ordered** (not sorted!) sequences of elements.
 - ❑ They are **immutable** (i.e. they cannot be changed at element level).
 - ❑ **Not part of Java**, by default.
 - ❑ Their purpose: ???
- Let's find out... 😊

- •
- **Tuples** – Quick example •
- •

Problem formulation:

```
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

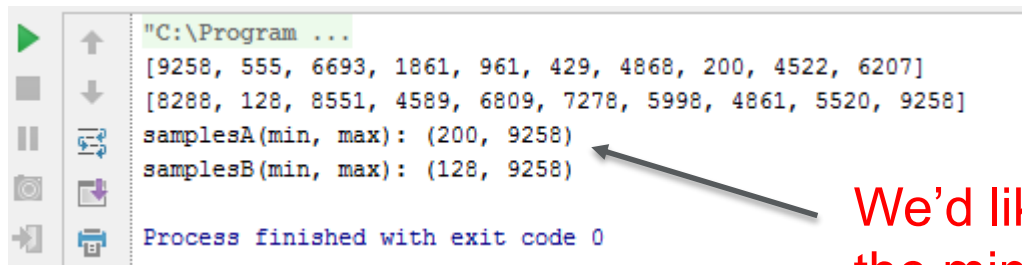


• **Tuples** – Quick example (2)

Problem formulation:

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

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



We'd like *to* compute both
the minimum
and the maximum...

using a single method!

• **Tuples** – Quick example (3)

We define a 2-Tuple to hold our data:

```
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 holders for
1st and 2nd element*

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

```
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 + ")";
    }
}
```

add a 3rd holder item
using inheritance

Tuples – Quick example (3)

We define a 2-Tuple to hold our data:

```
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 holders for
1st and 2nd element*

Build our algorithm, using above type

```
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

- •
- **Tuples** – Conclusions • • • • • • • • • • • • • • • •
- •

- ☐ They represent ordered (not sorted!) sequences of elements.
- ☐ They are immutable (they cannot be changed at element level).
- ☐ **Not part of Java**, by default.
- ☐ Their purpose: they allow **multi-return** in methods/functions.

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, };
```

Annotations for the declaration:

- type
- variable name
- “new” operator
- type[desired_size]
- initializer

Setting values explicitly:

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

Annotations for setting values:

- explicit position
- explicit value


- •
- **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” property
always available



Retrieval (explicit):

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

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

- •
- **Array(s) – Printing** •
- •

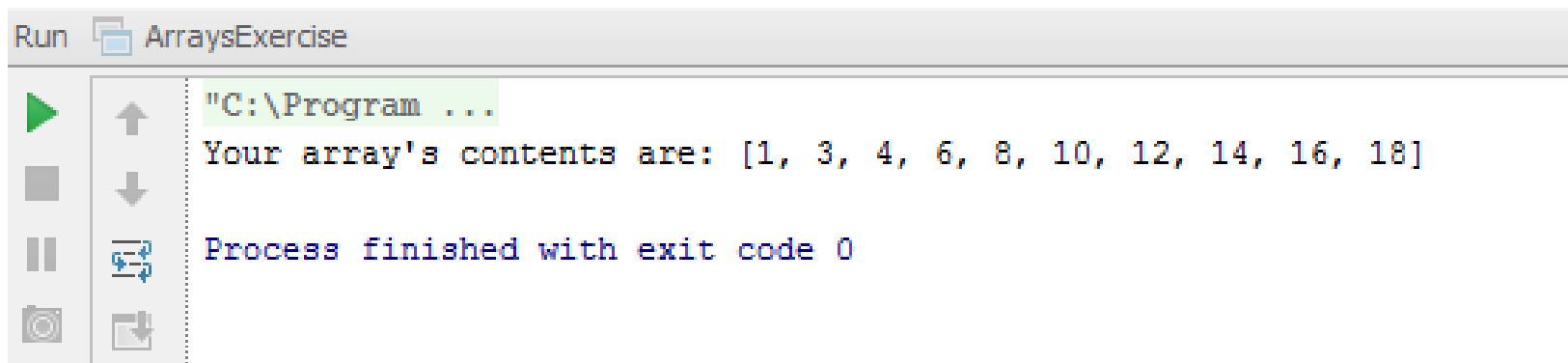
Printing (user friendly way):

```
// 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:



The screenshot shows an IDE's Run console window. The title bar says "Run" and "ArraysExercise". On the left is a vertical toolbar with icons for running, stepping through, and other debugging actions. The main area of the console displays the following text:

```
"C:\Program ...
Your array's contents are: [1, 3, 4, 6, 8, 10, 12, 14, 16, 18]
Process finished with exit code 0
```

- •
- **Changing requirements** • • • • • • • • • • • • • • • •
- •

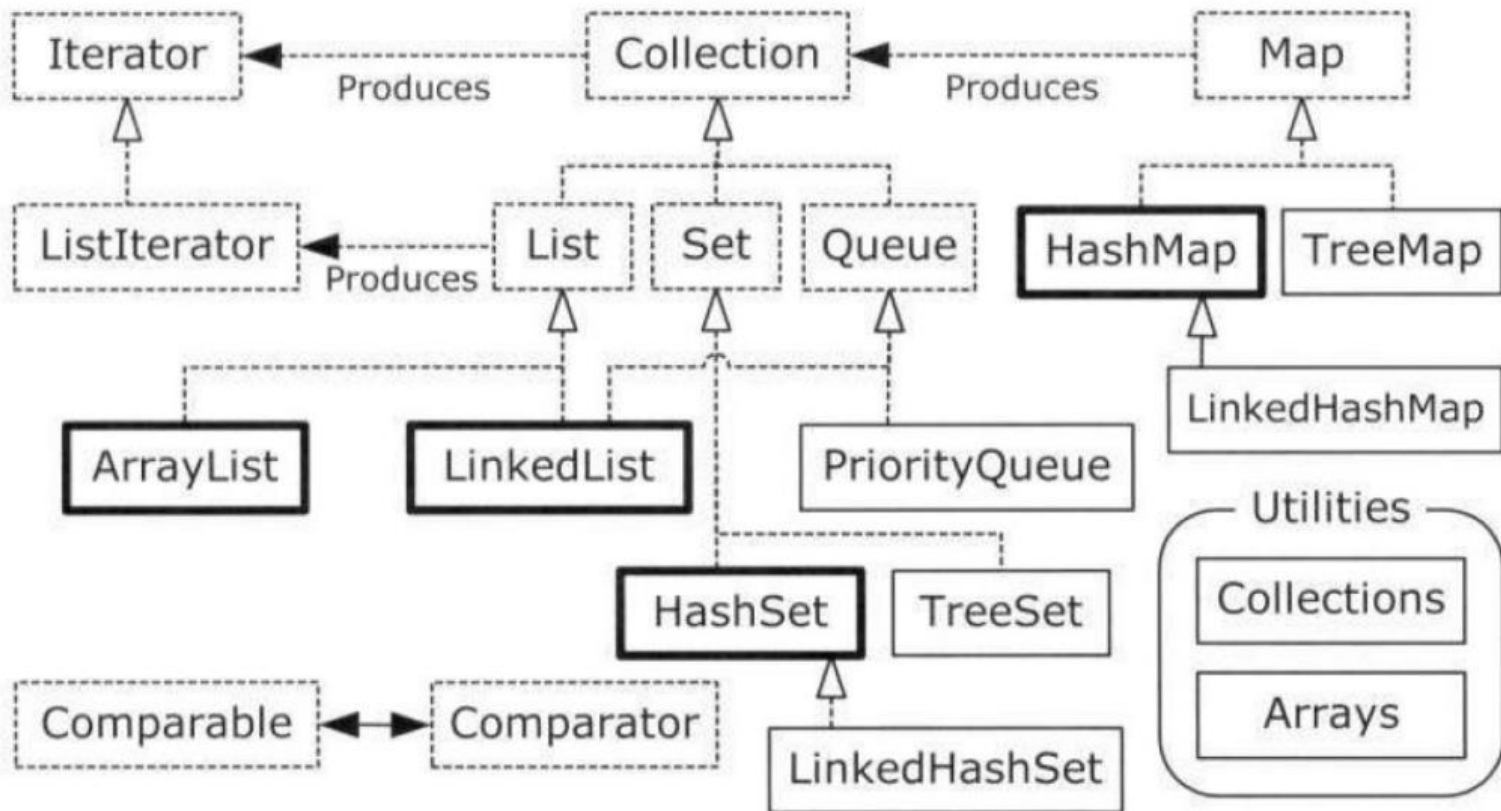
❑ What if we want to deal with any known number of “items”, dynamically at run-time?

❑ 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**?

The java.util “toolbox”

Here’s an overview of the most often used Java containers:



- •
- **A first word about Java containers** • • • • • • • •
- •

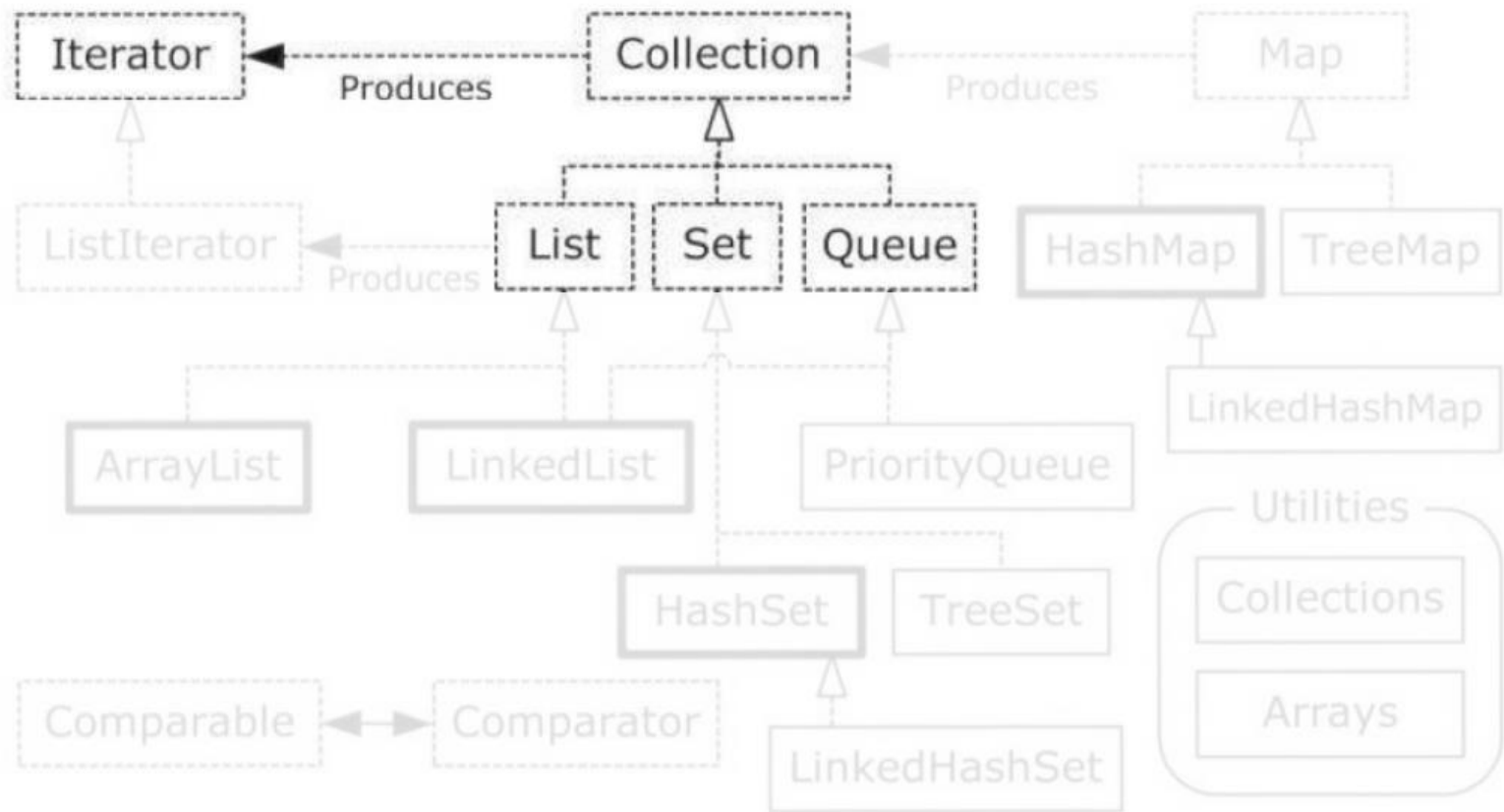
Categories:

- 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).

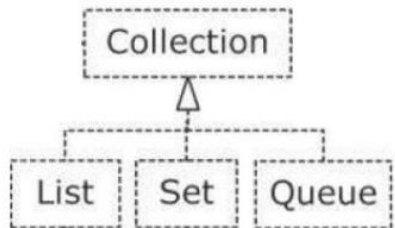
Container utilities: `java.util.Arrays` & `Collections` classes

TECMNET

java.util.Collection(s)



java.util.Collection(s)

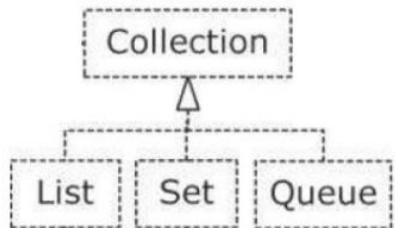


The basic single-item containers in Java are known as **collections**.

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*).

• **java.util.Collection(s)**



Why would one use such data structures?

Advantages:

- No expansion limits

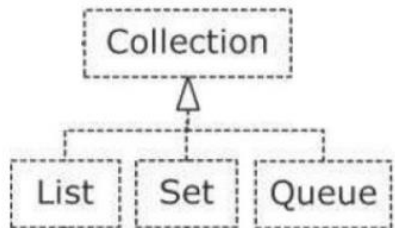
(making them *perfect* for dynamic memory management)

Disadvantages:

- None

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

• **java.util.Collection(s)**



Java collections can *initially* be split into:

- Lists
- Sets
- Queues

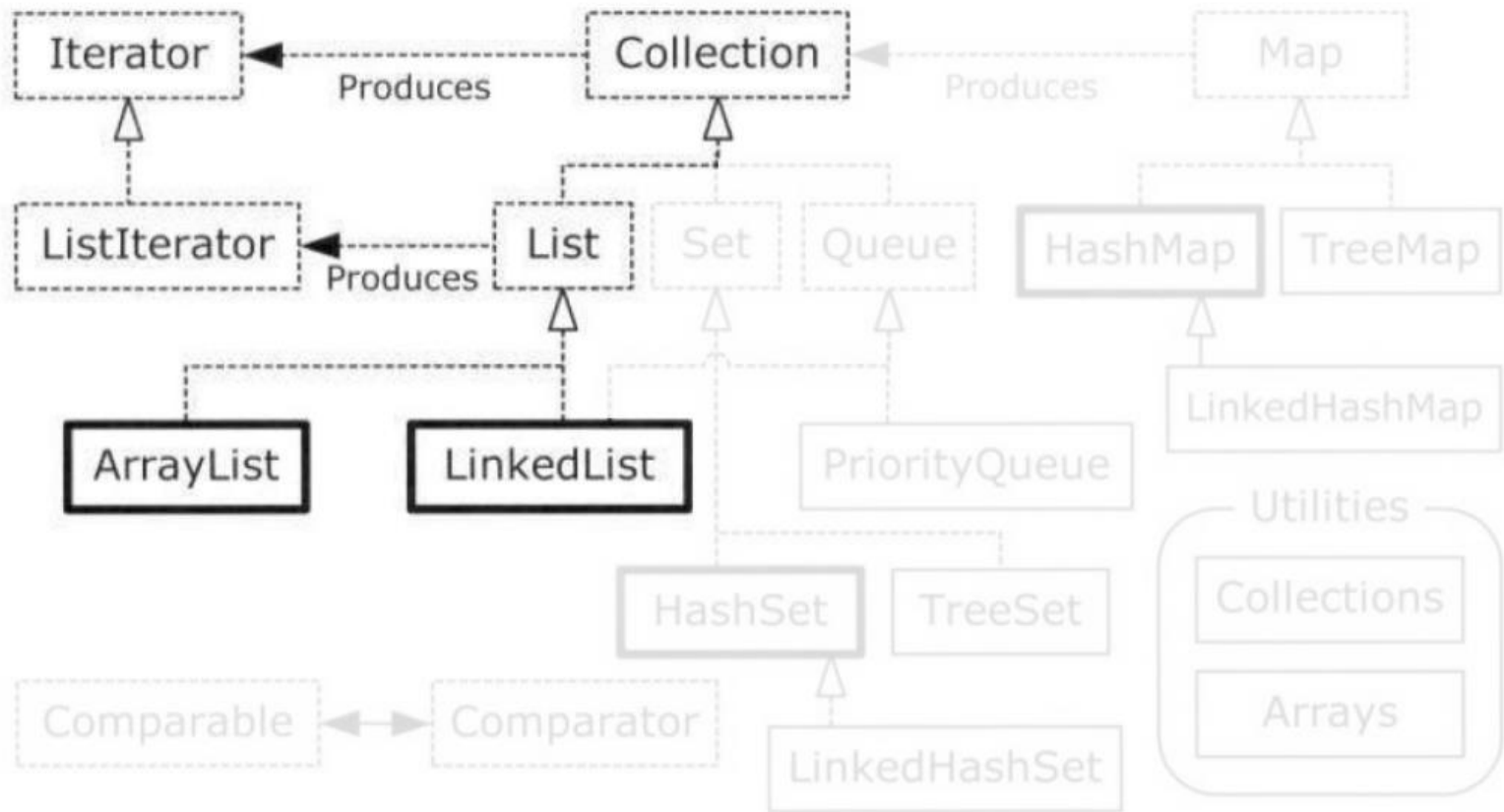
(these are all just *root* interfaces)

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

2.2

Lists in Java

`java.util.List(s)`



- •
- **java.util.List(s)** – Notes • • • • • • • • • • • • • • • •
- •

- ❑ They can hold single elements.
- ❑ They **allow** duplicates to be inserted.
- ❑ They are **ordered**, by default (**not sorted** – careful here!).
- ❑ Adequate for FIFO and LIFO behavior (as **stacks** & **queues** – later on this).

- •
- **java.util.List(s)** – Quick example • • • • • • • • • •
- •

Given the following:

```
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);  
    }  
}
```

java.util.List(s) – Quick example (2)

Let's put them into practice:

```
@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();
```

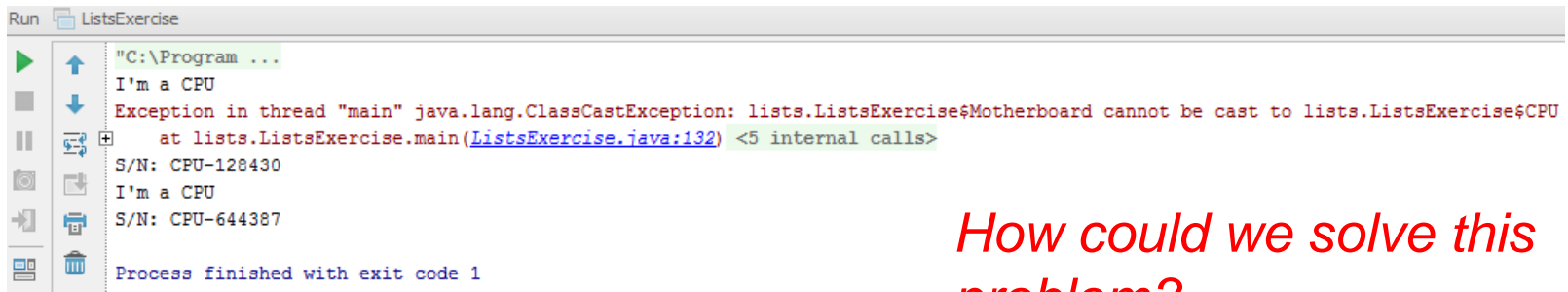
```
    }
```

```
}
```

Simple declaration

Adding elements

Retrieval







The screenshot shows an IDE window titled "Run ListsExercise". The output console displays the following text:

```
"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?

java.util.List(s) – Quick example (3)

Fix by adding a rule: establish *bounds*

<pre>@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(); } }</pre>	   	<pre>@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(); } }</pre>
--	---	--

```
"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
```

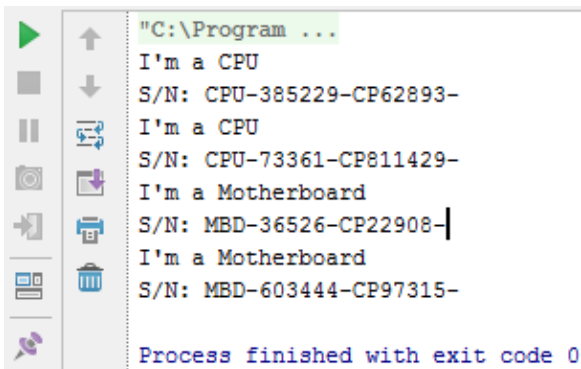
Hey, what about the poor Motherboard? ☹️

• **java.util.List(s)** – Quick example (4)

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();  
    }  
}
```

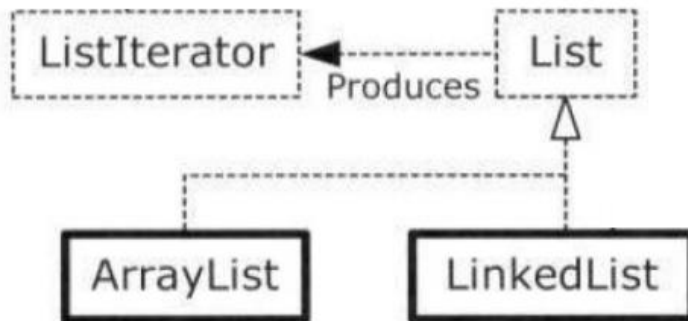
Both **CPU** and **Motherboard** are some kind of **Part**



`java.util.List(s)`

Most often used **Lists** are:

- ArrayList
- LinkedList



Legacy:

- Vector

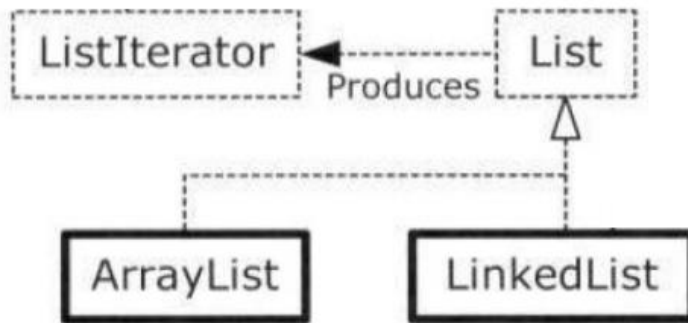
(may be old school, but it offered thread-safety – replaced by `CopyOnWriteArrayList`)

When and why would one use such data structures?

java.util.ArrayList

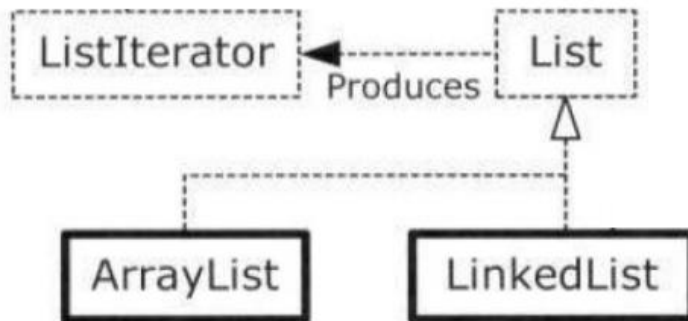
Excels at **randomly accessing** elements.

The drawback: **slower**
when **inserting** elements in
the **middle**.



java.util.LinkedList

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



Larger feature set than an `ArrayList`.

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

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

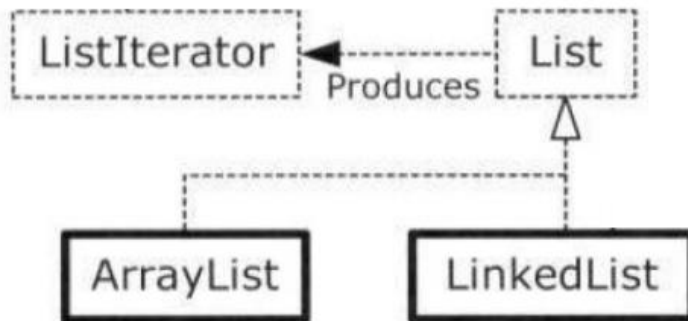
• • • • •

• java.util.List(s) • • • • •

• • • • •

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

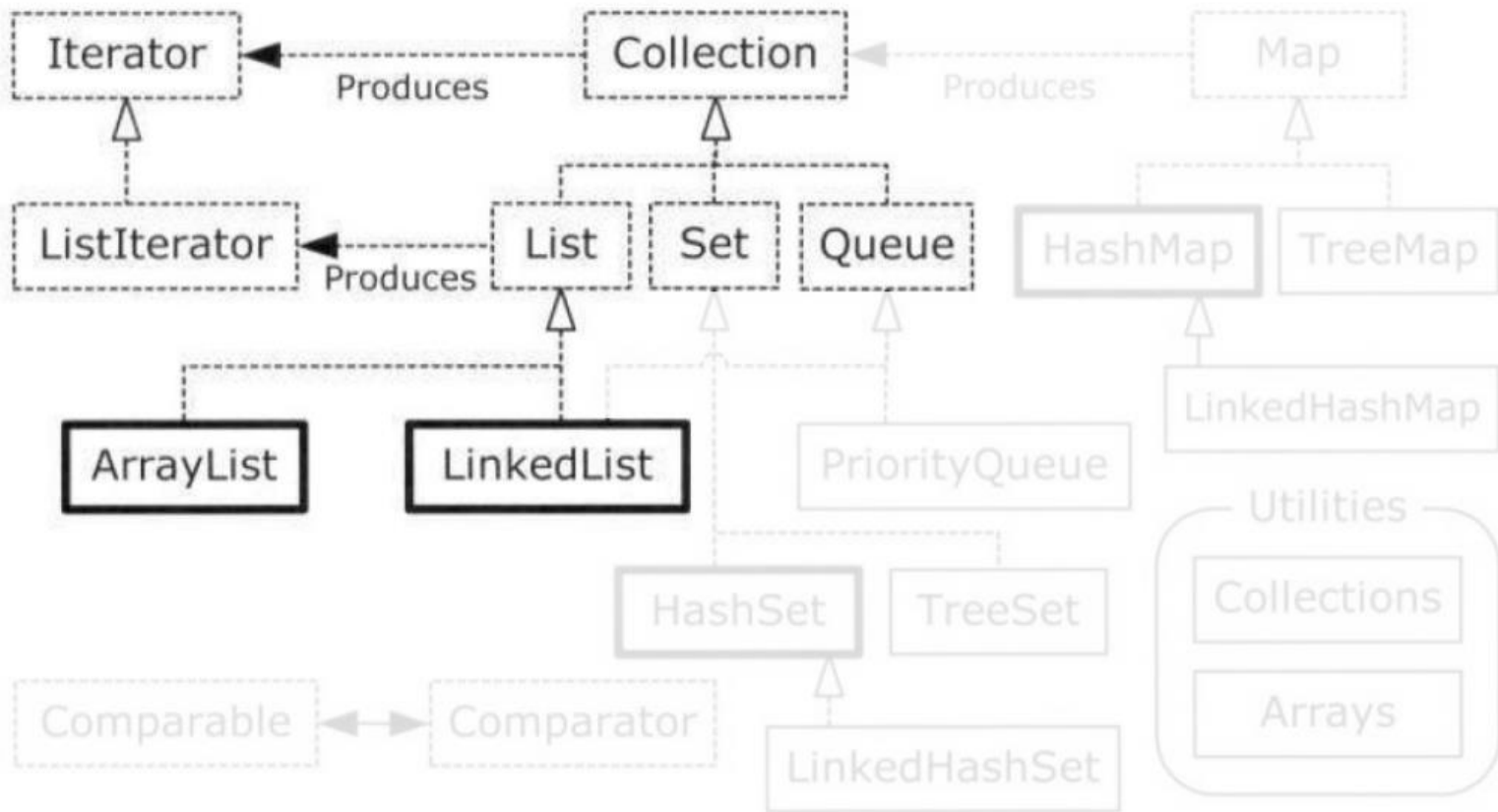
- add(obj) (at the end)
- addAll(collection)
- insert(atPosition)
- contains(obj)
- get(position)
- remove(position/obj)
- iterator()



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.

(Thus 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 containers after all ?)

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

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

- •
- **java.util.Iterator** – Notes (2) • • • • • • • • • • • •
- •

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

❑ It **selects each element** of that **sequence** without having the programmer worrying about the underlying type (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()`

java.util.Iterator – Quick example

```
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

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

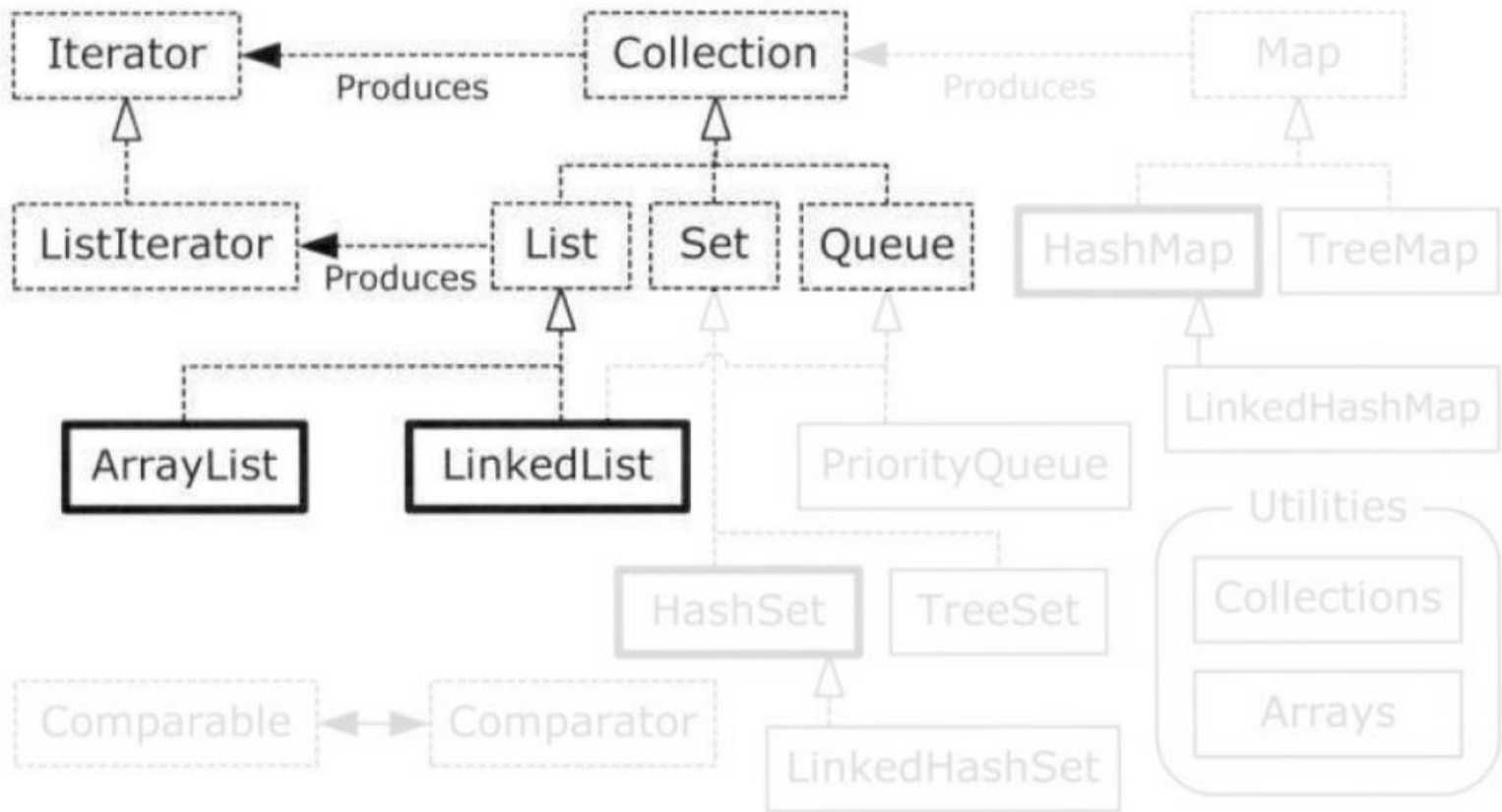
```
public class CrossContainerIteration {  
    public static void display(Iterator<Pet> it) {  
        while (it.hasNext()) {  
            Pet p = it.next();  
            System.out.print(p.id() + ":" + p + " ");  
        }  
        System.out.println();  
    }  
}  
  
public static void main(String[]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

ask for each
container's Iterator

java.util.ListIterator



- •
- **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.

java.util.ListIterator – Quick example

```
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 an element using set()

• •

- **java.util.List(s)** – Conclusions • • • • • • • • • •

• •

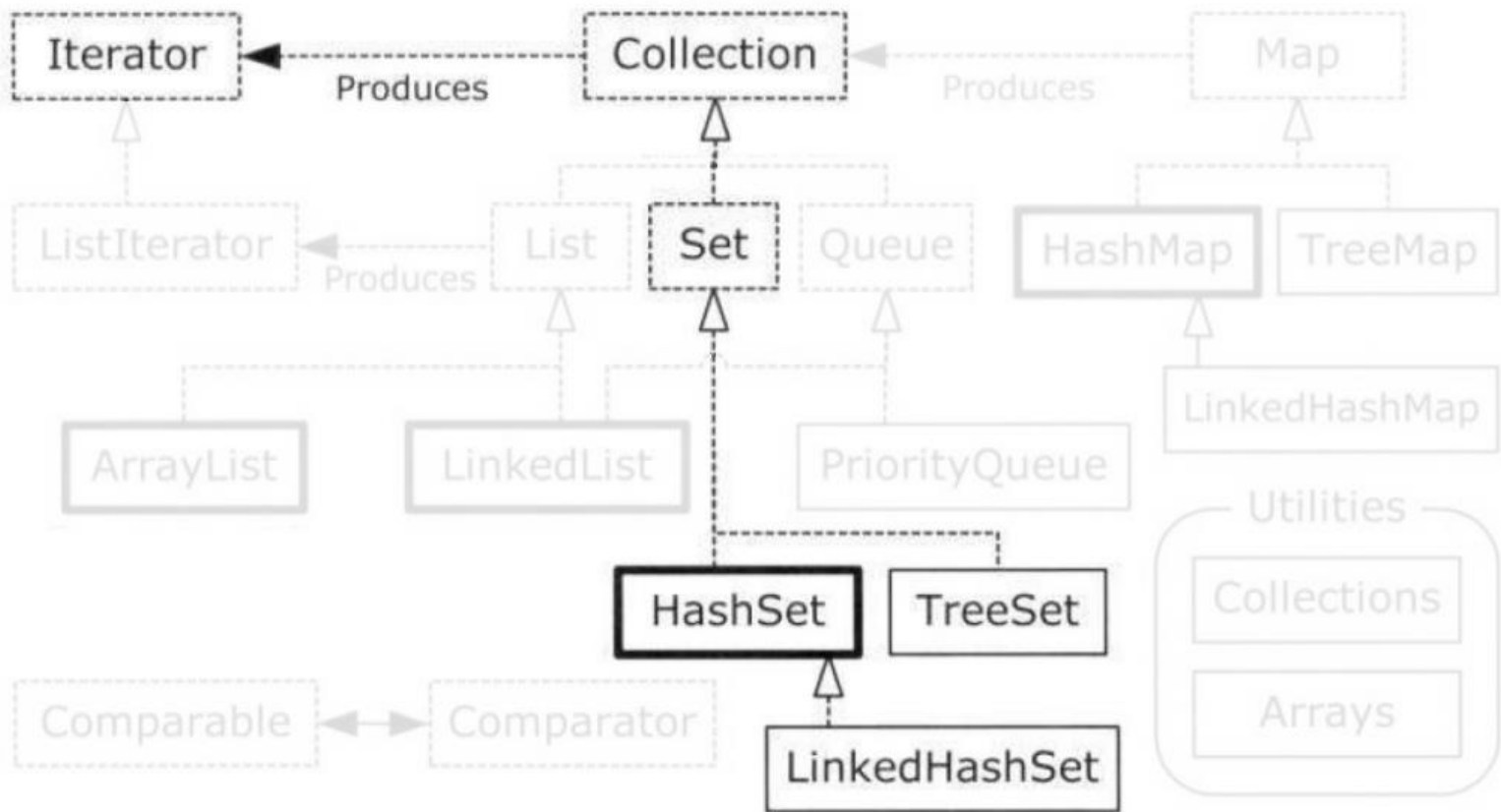
- ❑ They can associate numerical indexes to objects – thus, like arrays they are **ordered**.
- ❑ Automatic resizing to accommodate new items, if needed.
- ❑ ArrayLists excel at **random access** (direct retrieval).
- ❑ LinkedLists are **multi-purpose** lists; they offer **optimal sequential access**, as well as **insertions** and **deletions** in the middle.
- ❑ **Iterators** *unify access to containers* because they separate traversal of a sequence from underlying implementations.

A decorative grid of small, light gray dots arranged in a 5x20 pattern, spanning the width of the slide.

2.3

Sets in Java

java.util.Set(s)



• •

• **java.util.Set(s)** – Notes • • • • • • • • • • • • • • • •

• •

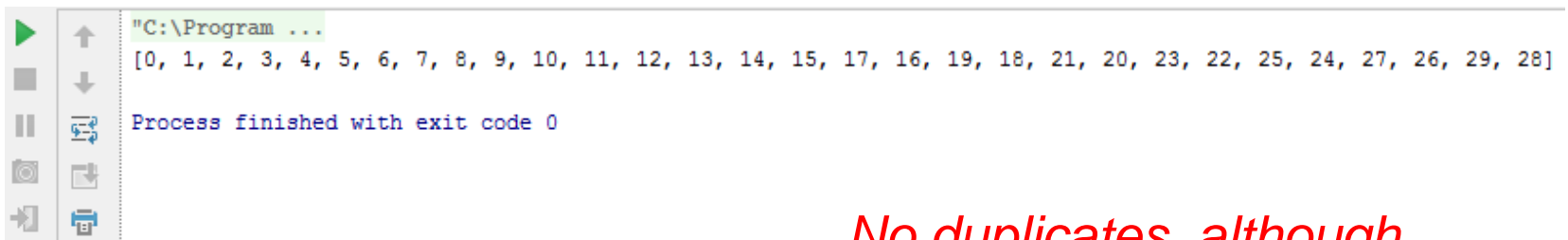
- ❑ Like lists, they can hold single elements.
- ❑ They **DO NOT** allow duplicates.
- ❑ Used for *querying* held elements, via `contains(obj)` method (e.g. *test for membership*).
- ❑ Because of this, lookup is typically the most important operation for a Set.

java.util.Set(s) – Quick example

```
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);  
}
```

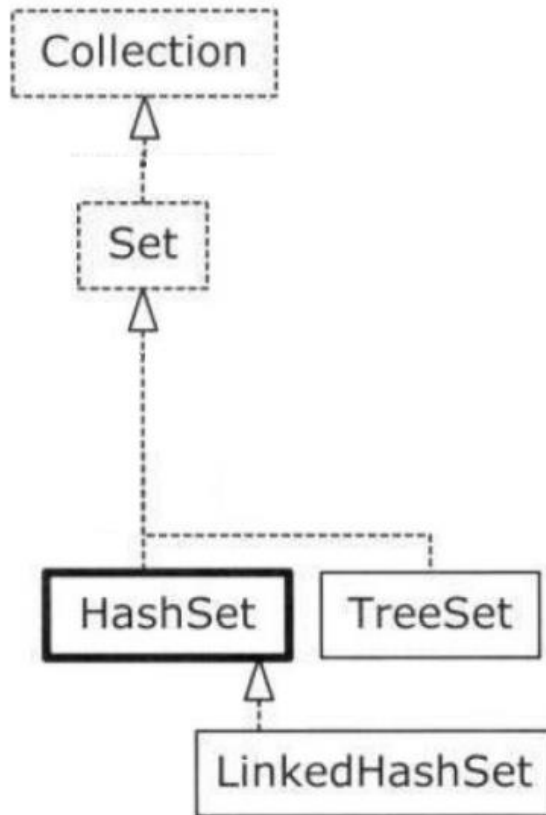
typical declaration

add an integer
between 0 and 30



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

```
java.util.Set(s)
```

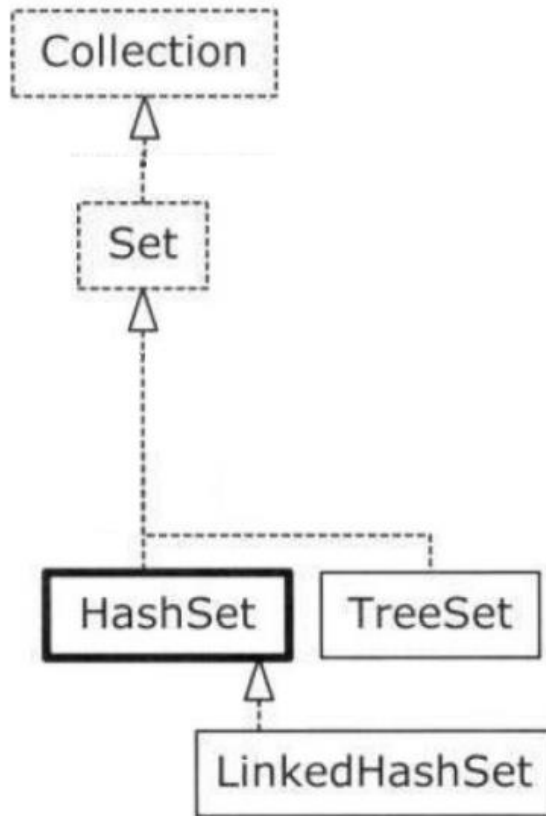


Sets are available in many flavors. The **three** most used are:

- HashSet
- LinkedHashSet
- TreeSet

When and why would one use such data structures?

java.util.HashSet

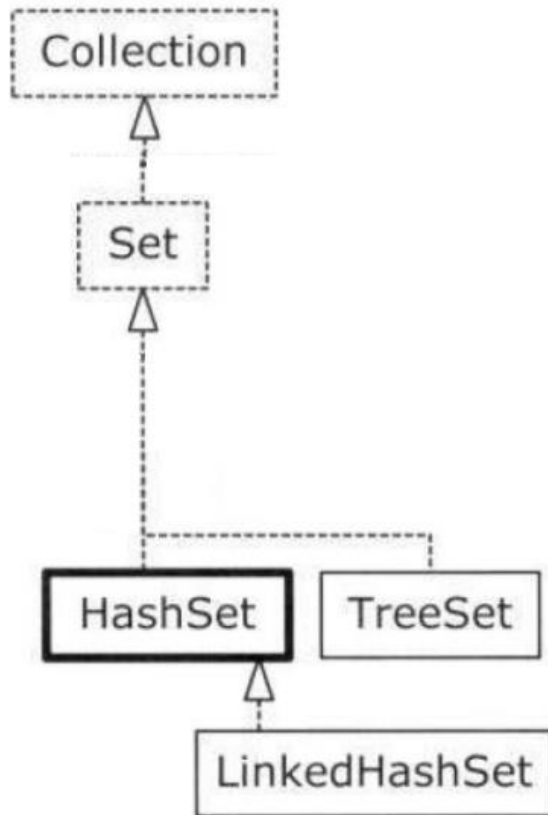


Used when fast lookup time is important.

Utilizes a hashing function for speed.

Order of elements appears to be maintained through custom heuristics.

• java.util.LinkedHashSet

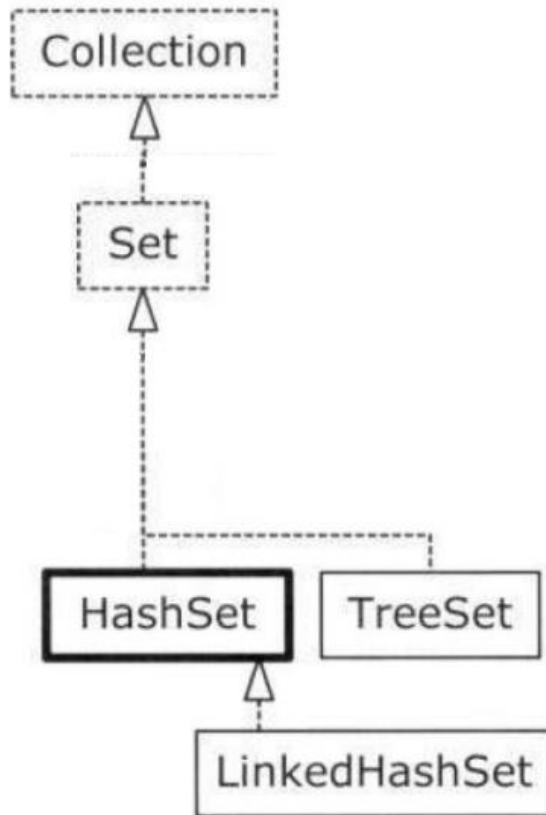


Typically as fast as `HashSet`, in matters of lookup speed.

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

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

java.util.TreeSet



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**.

java.util.TreeSet – Quick example

```
public class SortedSetOfStrings {
```

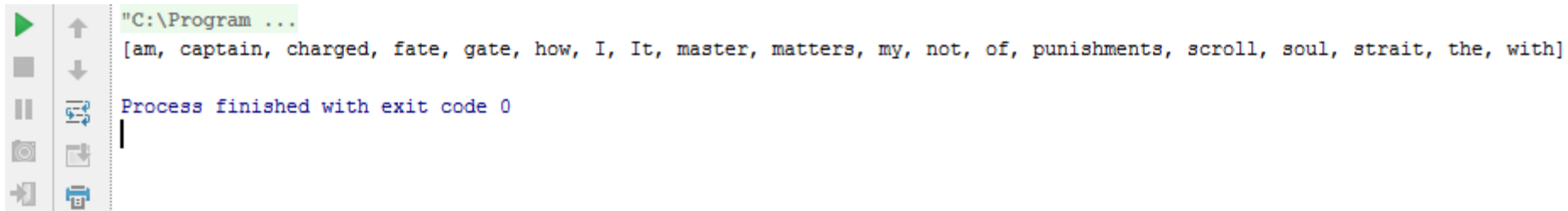
```
    private static final String poem =  
        "It matters not how strait the gate,\n"+  
        "How charged with punishments the scroll.\n"+  
        "I am the master of my fate:\n"+  
        "I am the captain of my soul.";
```

```
    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

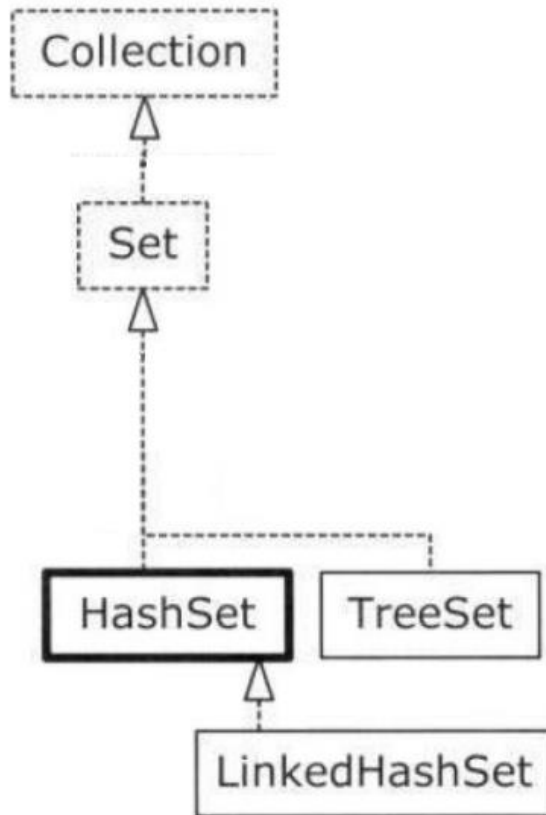
A comparator is
given; not
mandatory

Quick collection building
via Arrays.asList(...) utility



```
"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
```

java.util.TreeSet

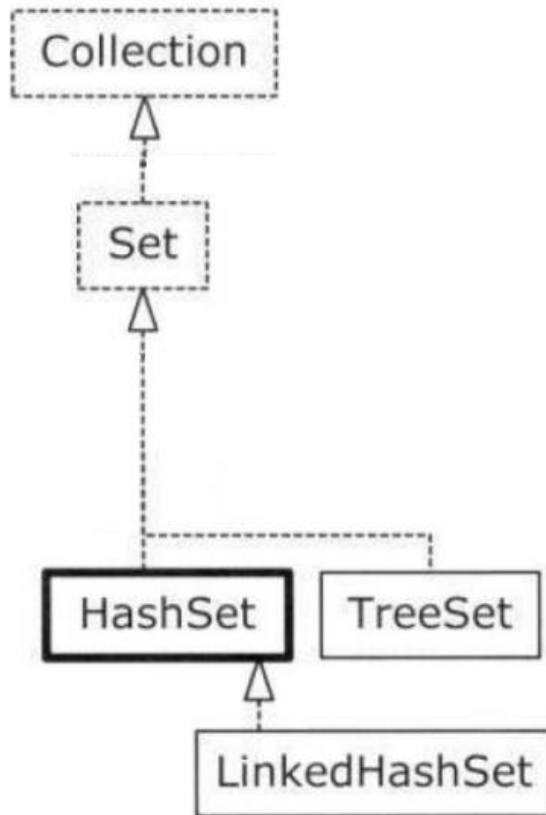


Thus, the elements in a `SortedSet` are guaranteed to be in **sorted order**.

This allows for the following interesting methods:

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

`java.util.Set(s)`



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

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

• •

- **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.