teamnet transforming technology

An introduction to the Java Collections Framework

Trainers:

Cristian DUMITRU
Cătălin BÎNĂ
Radu HOAGHE
Andrei MARICA
Oana BEȘLIU
Mihail TUDOSE
Ioan DINU

Author: Bogdan ȘTEFAN

Document version: 1.3

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

2.1

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;
```

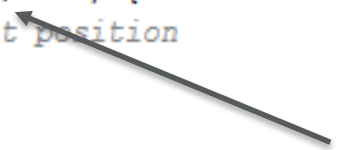
explicit position a.k.a. “the array **Index**”

explicit value

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/accessing (explicit):

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

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

Array(s) – Further notes on retrieval

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

```
// --- 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!");
```

Using "foreach"

Simpler access to references instead of "i"-based values

Array(s) – Printing

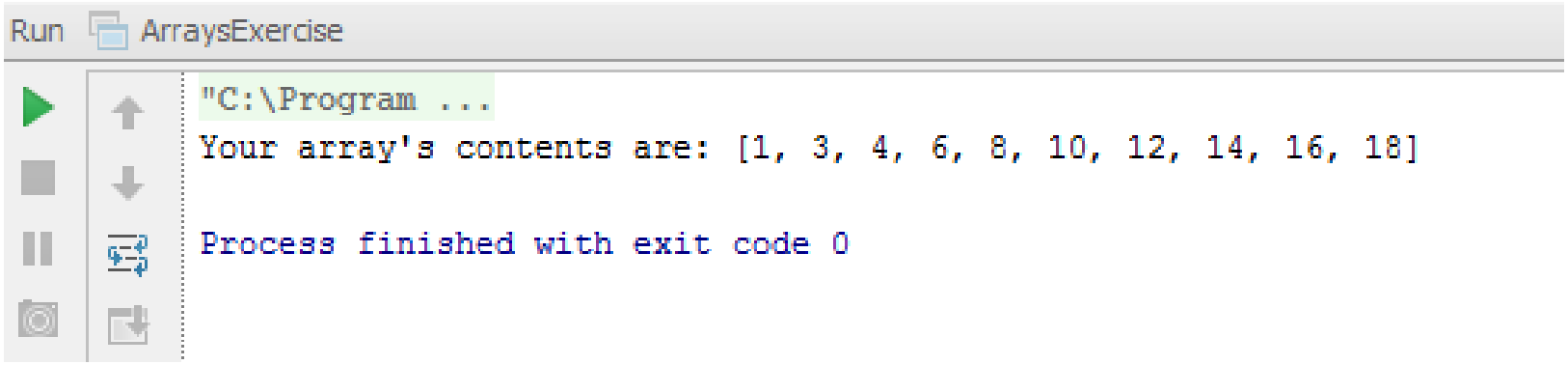
Printing (user friendly way):

```
// 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:



Array(s) – *"99 problemz"* ☺

- They are really really fast; specifically, they provide an *efficient* performance, however at *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!).

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

Changing requirements

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

Think in terms of

“*dynamic memory management*”…

OMG!!! Omagawd!!!

What do we do?!

What… do we… doo?!!

Image source: http://awesomegifs.com/2011/08/19/omg-cat-makes-you-say-omg/

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:

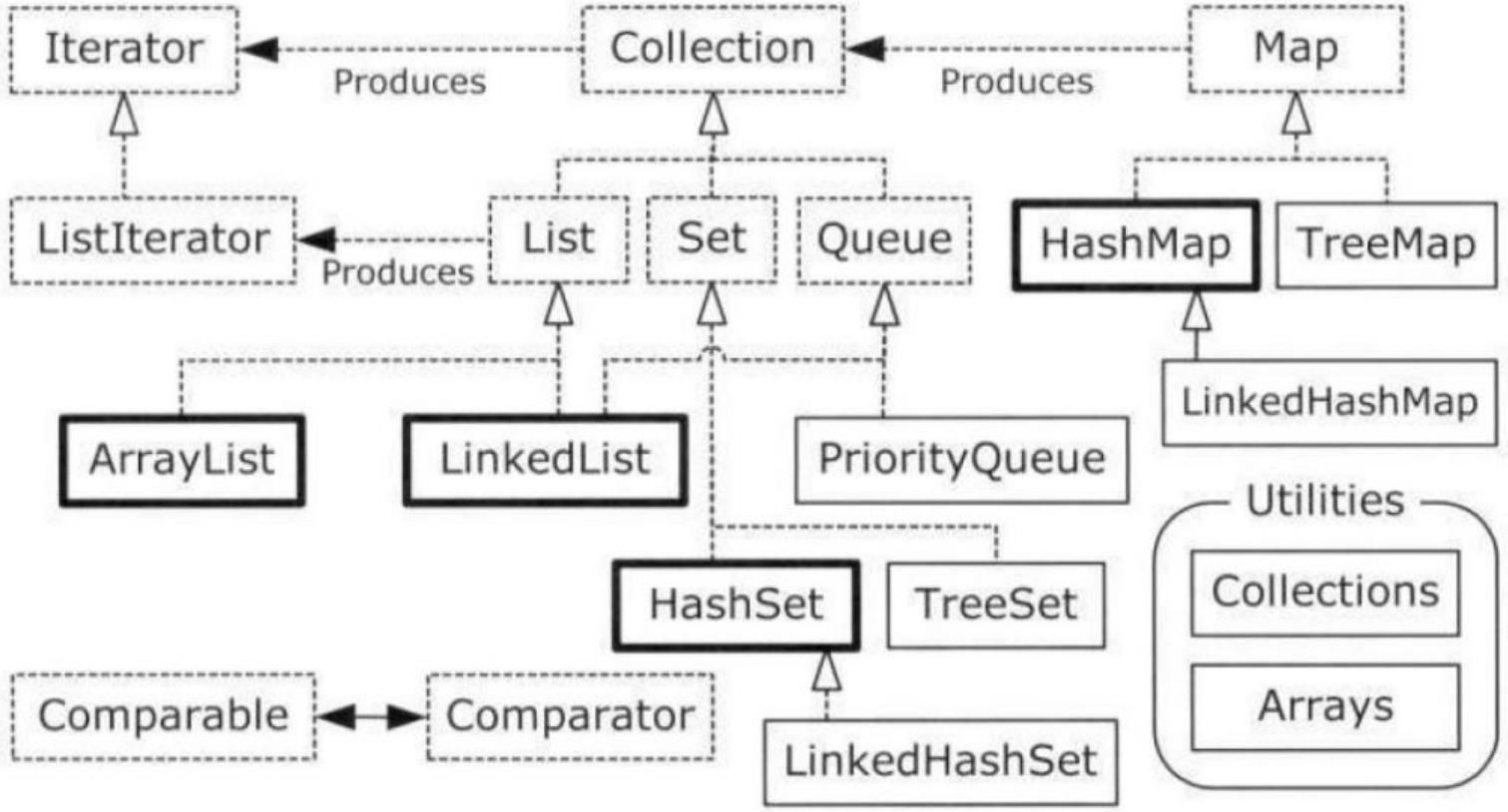


Image source: Thinking in Java (4th Edition), Bruce Eckel

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

java.util.Collection(s)

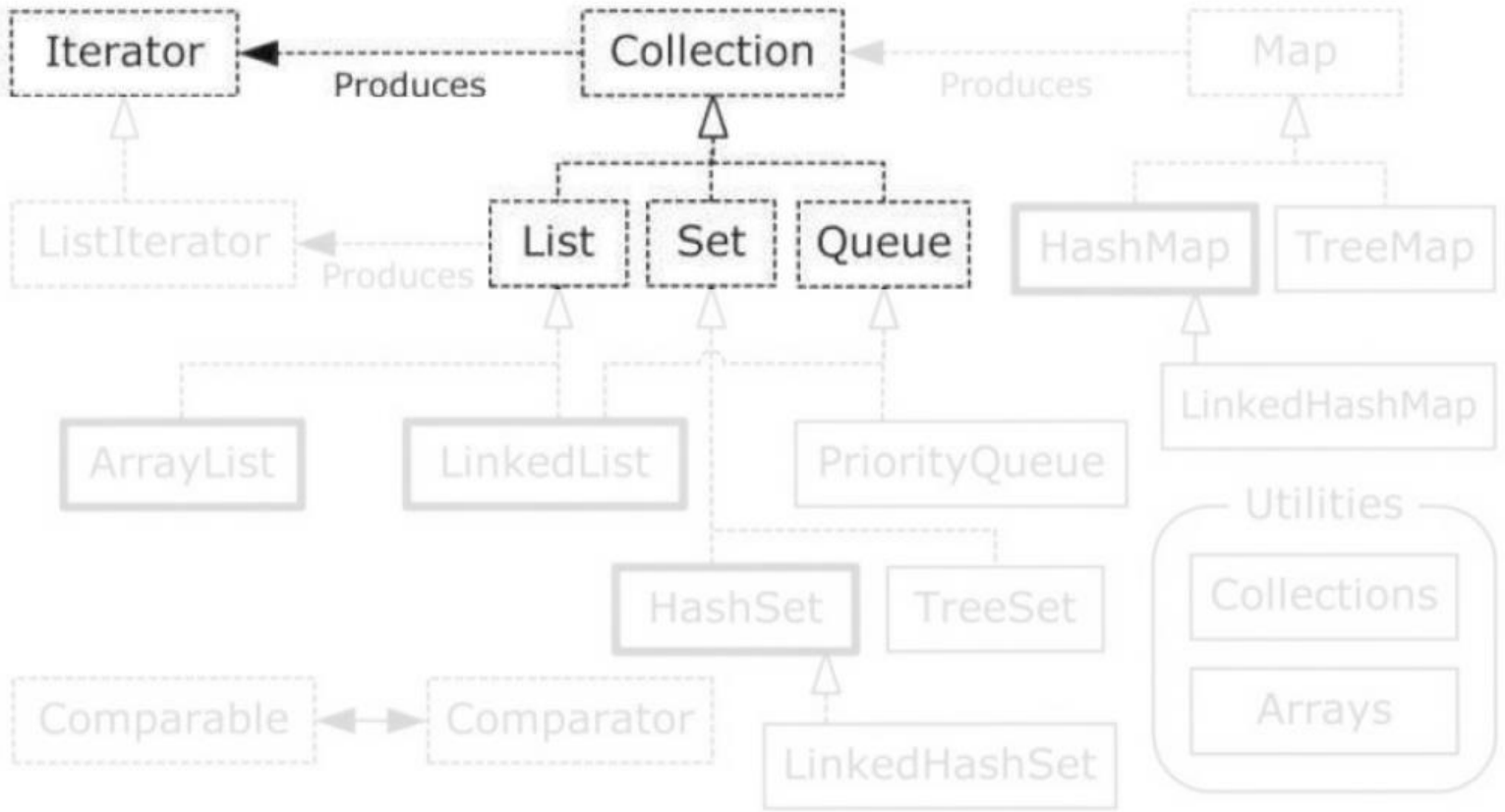
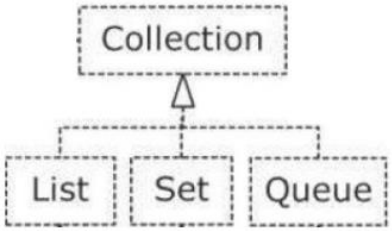


Image source: Thinking in Java (4th Edition), Bruce Eckel

java.util.Collection(s)



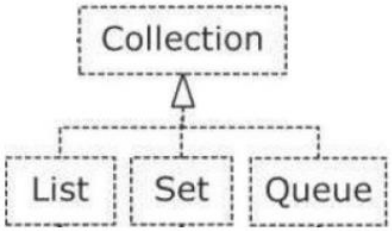
The basic single-item containers in Java are known as **collections**.

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*).

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

java.util.Collection(s)



Why would one use such data structures?

Advantages:

- No expansion limits - theoretically.

  (making them *perfect* for dynamic memory management)

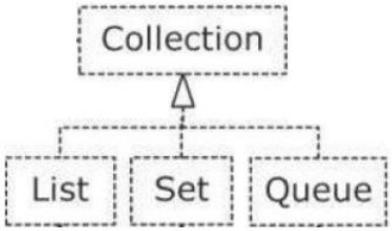
Disadvantages:

- 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) ☺

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

java.util.Collection(s)



Java collections can *initially* be split into:

- `Lists`
- `Sets`
- `Queues`

(these are all just root *interfaces*)

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

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

2.2

Lists in Java

2.2

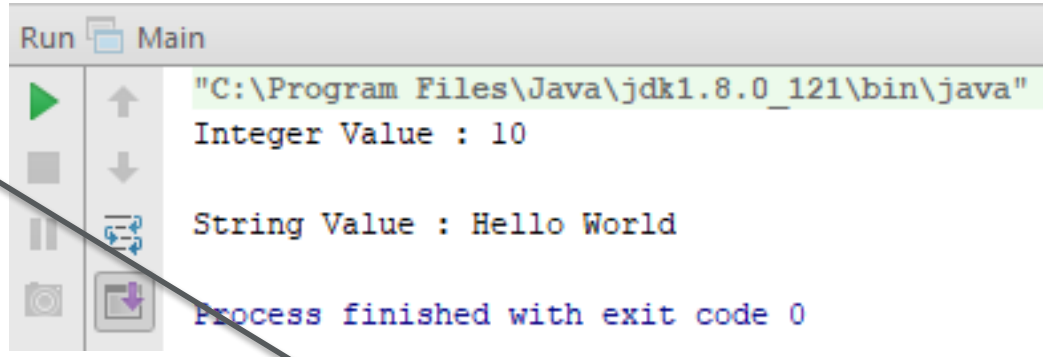
But first, a word about Generics!

Generics

- Introduced in Java 5
- They add a way to specify **concrete** types to general purpose classes and methods that operated on Object before
- They provide **compile-time type safety** that allows programmers to catch invalid types at compile time
- So, why do we need them?

Generics – Quick example

```
public class Box<T> {
    private T t;

    public void add(T t) {
        this.t = t;
    }

    public T get() {
        return t;
    }
}

public static void main(String[] args) {
    Box<Integer> integerBox = new Box<Integer>();
    Box<String> stringBox = new Box<String>();

    integerBox.add(new Integer(10));
    stringBox.add(new String("Hello World"));

    System.out.printf("Integer Value :%d\n\n", integerBox.get());
    System.out.printf("String Value :%s\n", stringBox.get());
}
```



Type parameter

Creating two boxes that contain two different types

Generics with Collections

- Generics are very useful to specify the **type** of elements contained by a Collection
- The type of any inserted element is checked at **compile-time**
- Thus, by using generics we can’t have mixed types elements in the same Collection (e.g. Strings and Integers)

Generics with Collections - example

```
List list = new ArrayList();

list.add(new Integer(2));
list.add("a String");

Integer integer = (Integer) list.get(0);

String string   = (String) list.get(1);
```

List of any Object

Adding Objects to list

Retrieving Objects from list

```
List<String> strings = new ArrayList<String>();

strings.add("a String");
strings.add("another String");

String aString = strings.get(0);
String anotherString = strings.get(1);
```

List of any String objects

Adding String objects to list

Retrieving Objects from list

2.2

Now, let’s get back to Lists!

java.util.List(s)

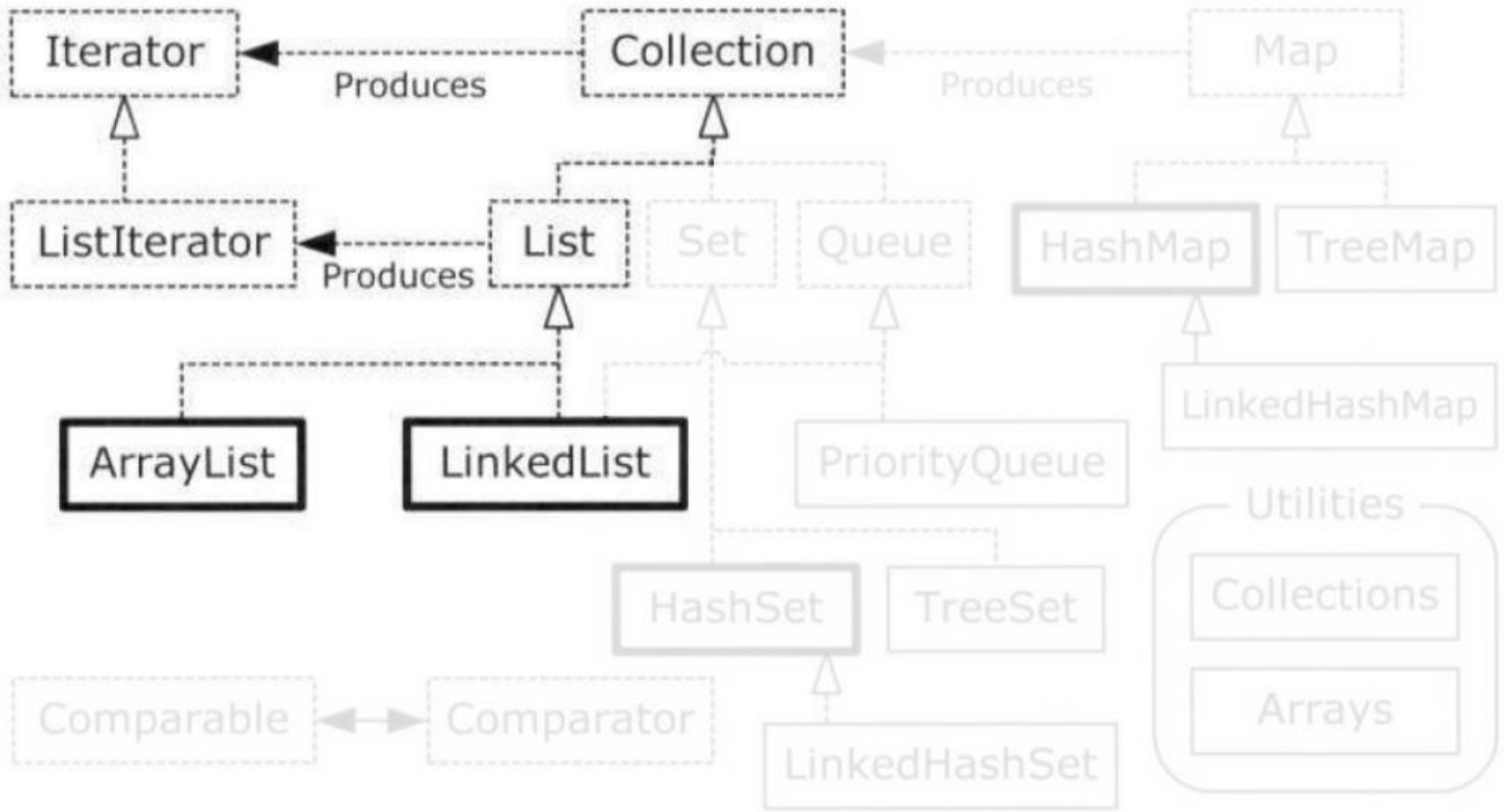


Image source: Thinking in Java (4th Edition), Bruce Eckel

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 **LIFO** and **FIFO** behavior (as **stack**s & **queue**s – 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)

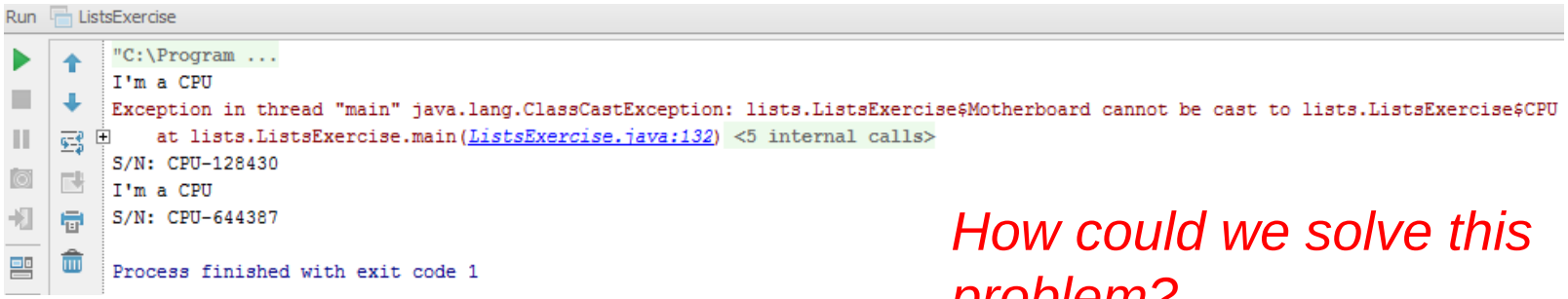
Let's put them into practice:

```
@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

Explicit retrieval by item *Index*



How could we solve this problem?

java.util.List(s) – Quick example (3)

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? ☹

java.util.List(s) – Quick example (4)

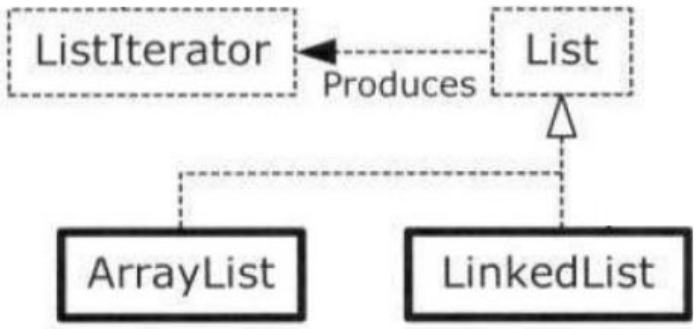
Easy fix: *lowering* the *bounds*, through *polymorphism*

```
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**

```
"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
```

java.util.List(s)



Most often used **Lists** are:

- `ArrayList`
- `LinkedList`

Legacy:

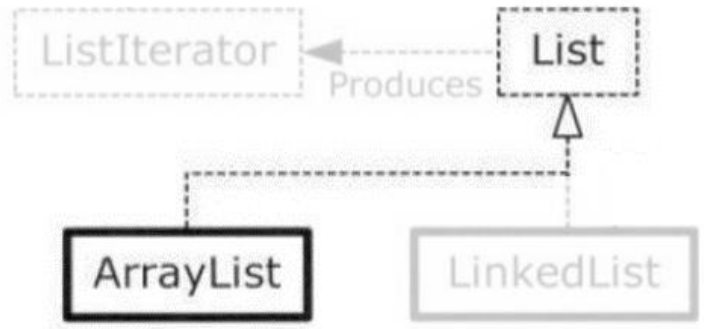
- `Vector`

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

When and why would one use such data structures?

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

java.util.ArrayList



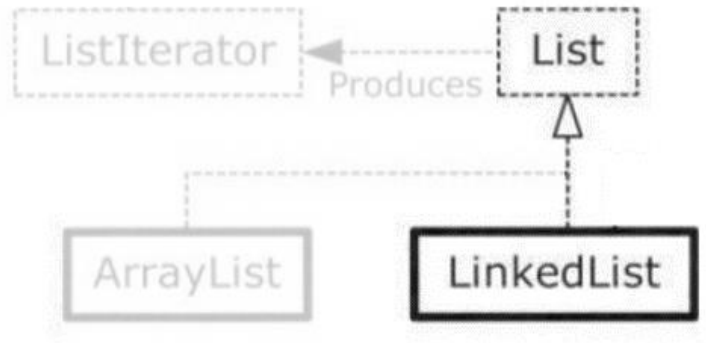
The most basic type of sequence.

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

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

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

java.util.LinkedList



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

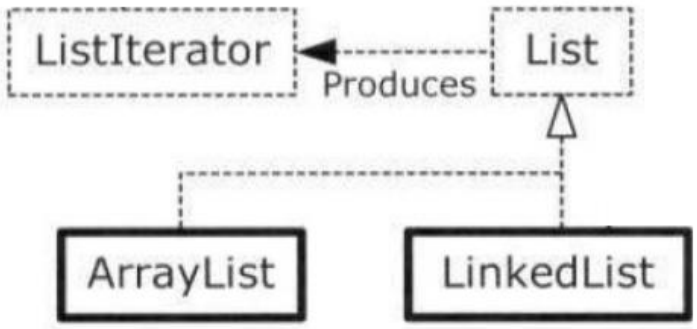
Larger feature set than an `ArrayList`.

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

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

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

java.util.List(s)



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

- `add(obj)` (at the end)
- `add(index, obj)`
- `addAll(collection)`
- `contains(obj)`
- `get(position)`
- `remove(position/obj)`
- `iterator()`

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

2.2.1

A case for Iterators

java.util.Iterator

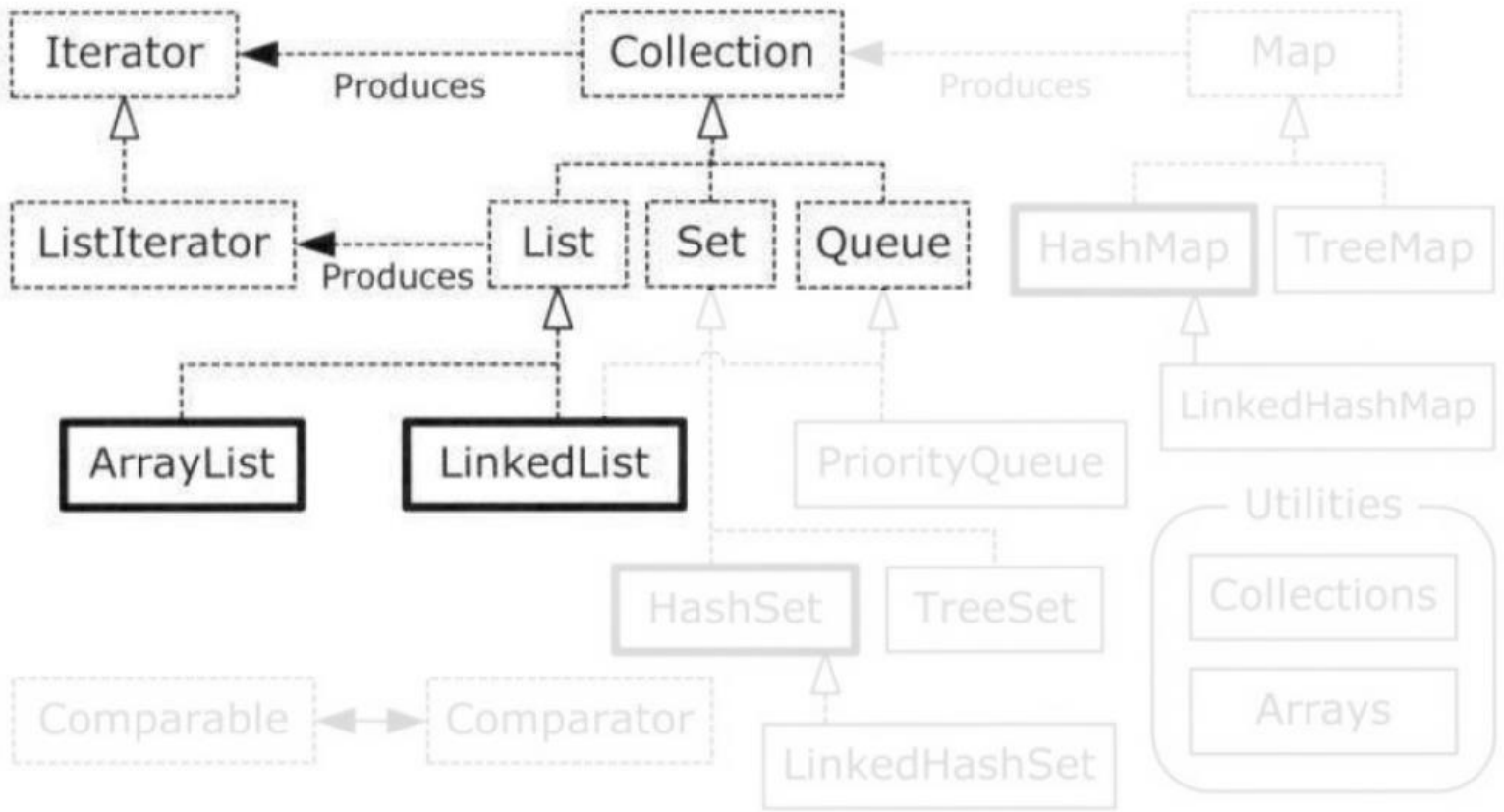


Image source: Thinking in Java (4th Edition), Bruce Eckel

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 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 worry 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

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

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

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

java.util.ListIterator

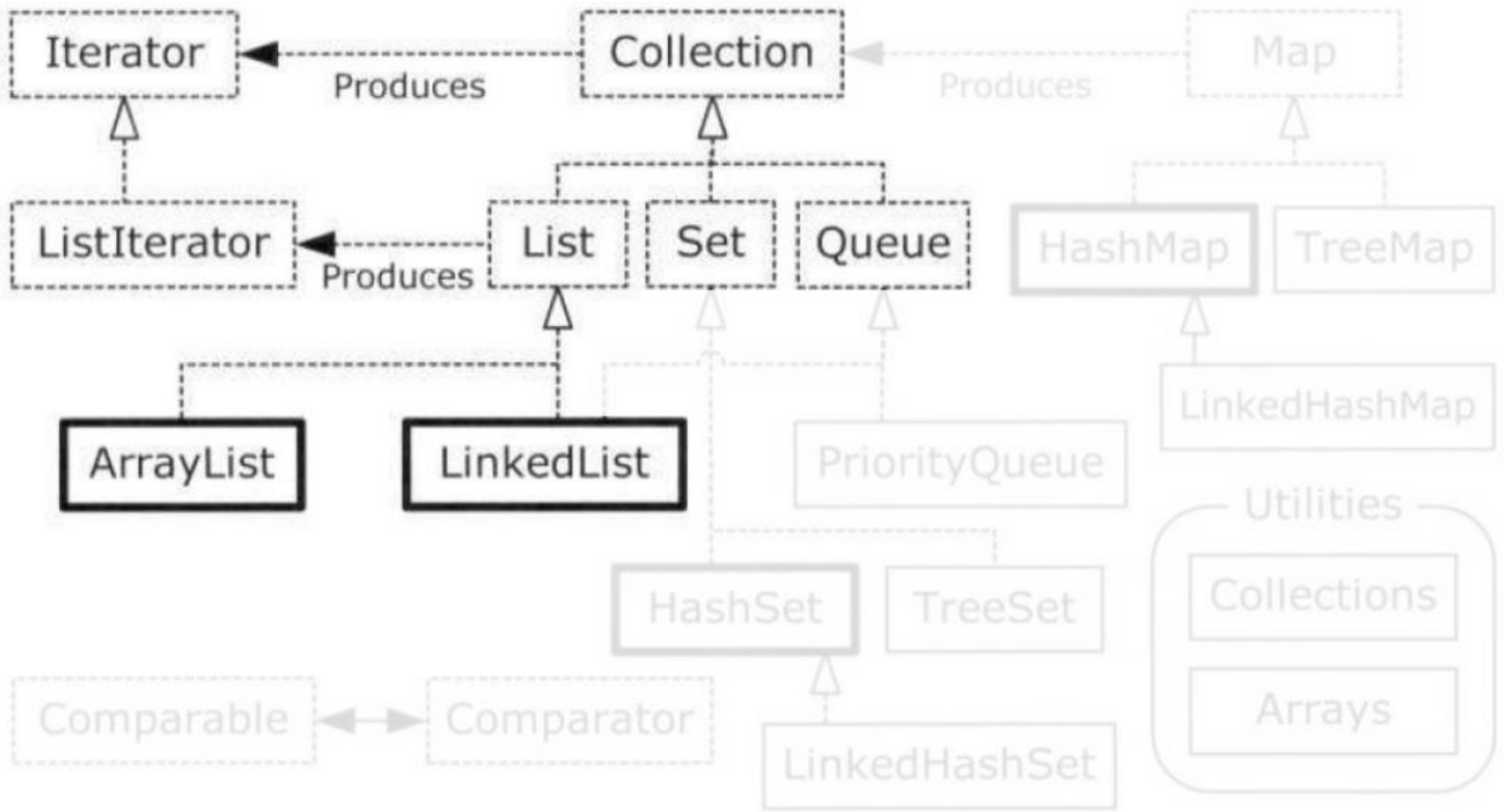


Image source: Thinking in Java (4th Edition), Bruce Eckel

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.

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 current iterator 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.
- **Iterator**s *unify access to containers* because they separate traversal of a sequence from underlying implementations.

2.3

Sets in Java

java.util.Set(s)

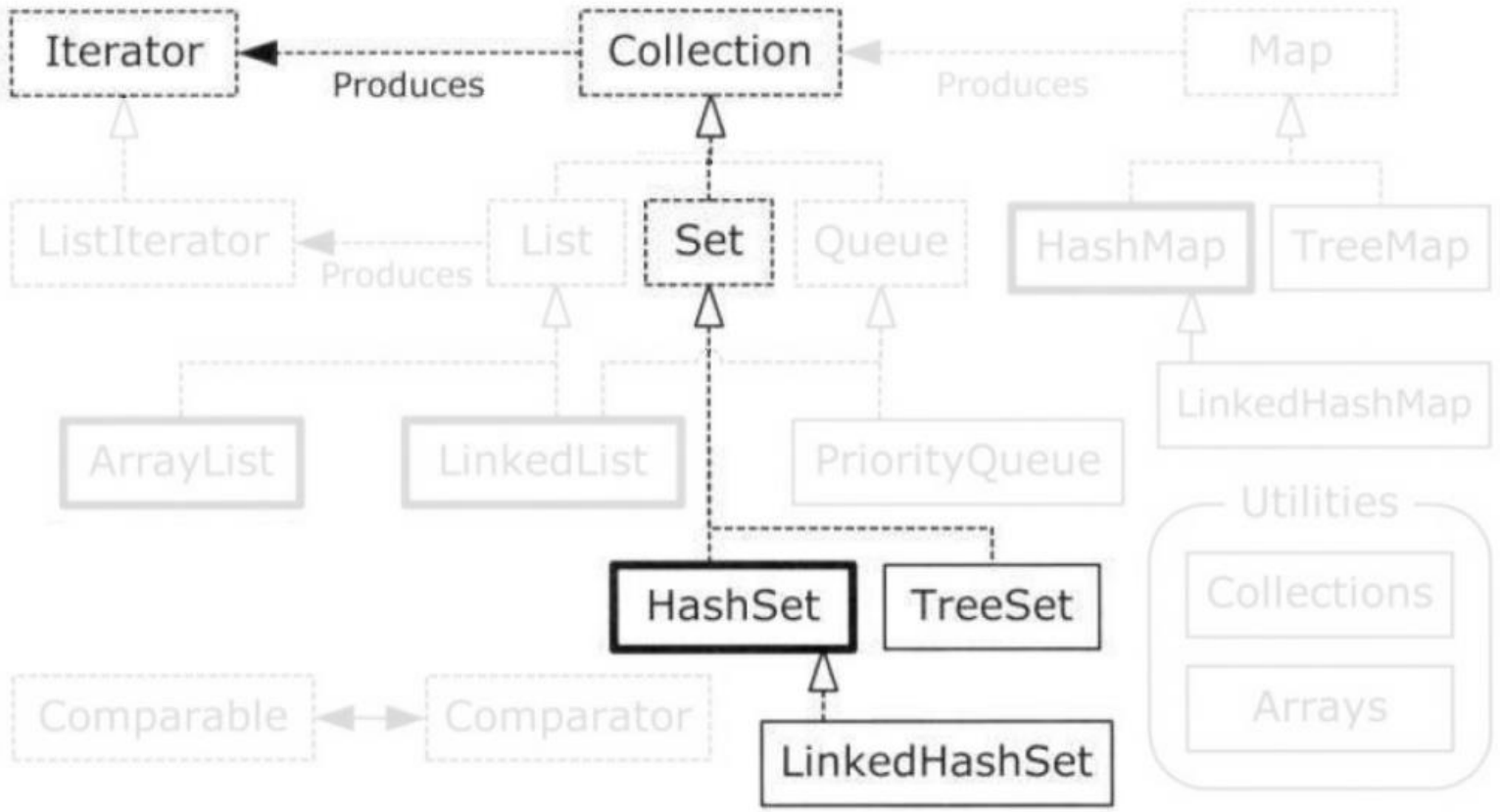


Image source: Thinking in Java (4th Edition), Bruce Eckel

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

```
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);

}
```

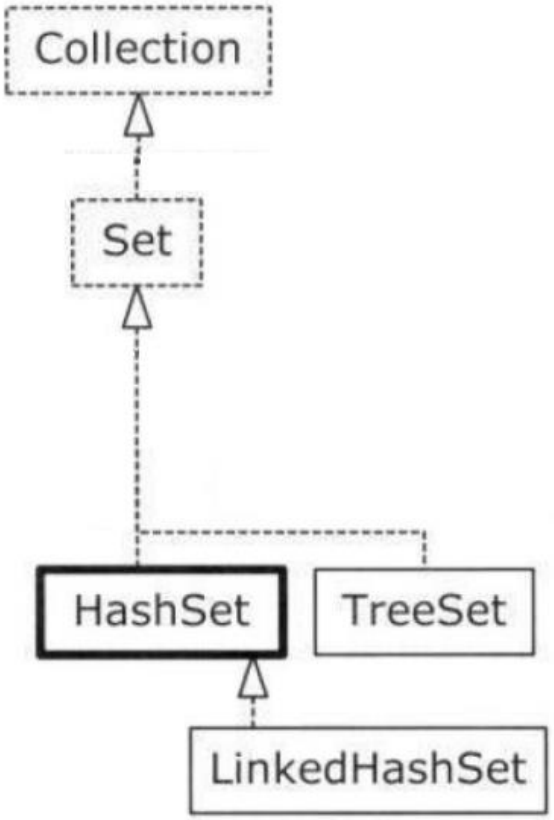
typical declaration

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.

java.util.Set(s)



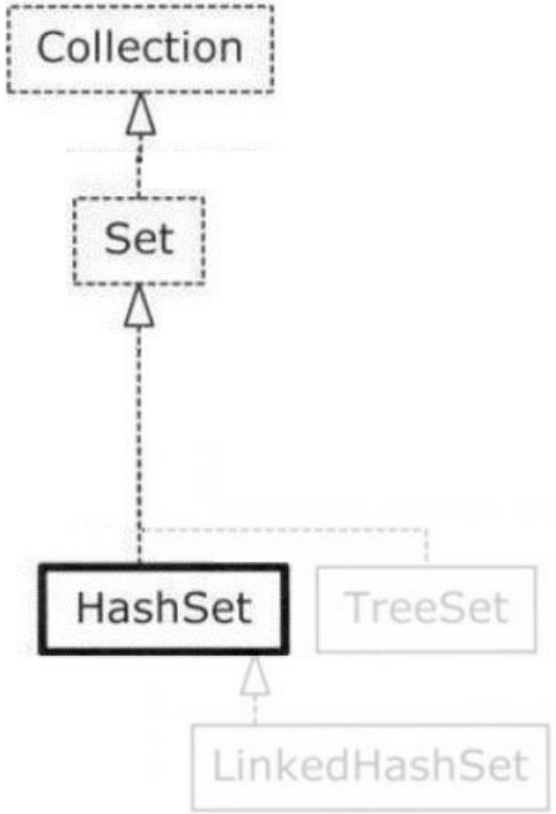
Sets are available in many flavors. The **three** most used are:

- `HashSet`
- `LinkedHashSet`
- `TreeSet`

When and why would one use such data structures?

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

java.util.HashSet



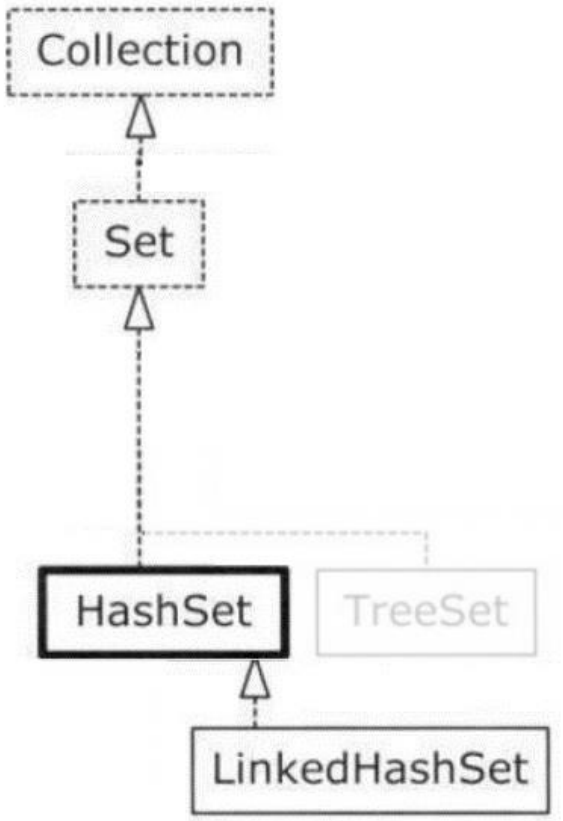
Used when fast lookup time is important.

Utilizes a hashing function for speed.

Order of elements appears to be maintained through custom heuristics.

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

java.util.LinkedHashSet



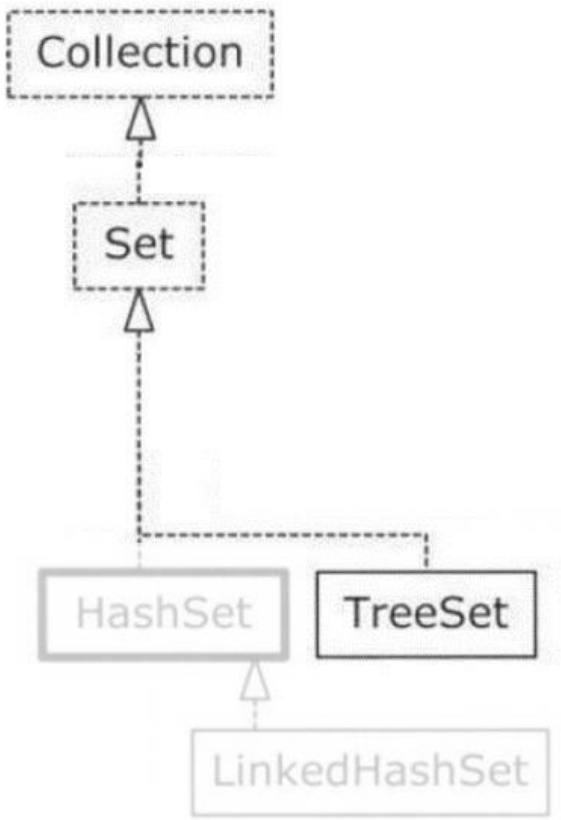
Typically as fast as HashSet, in matters of lookup speed.

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

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

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

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**.

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

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);
    }
}
```

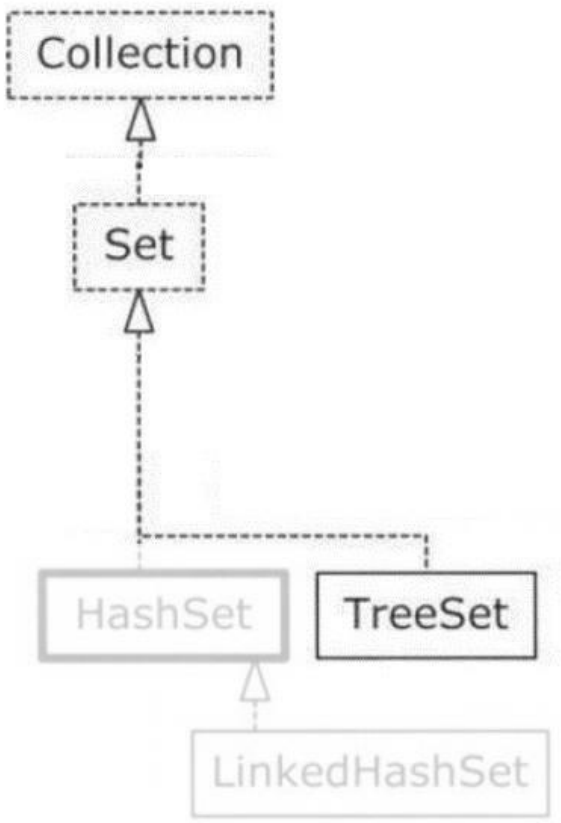
Notice the use of `SortedSet` interface

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



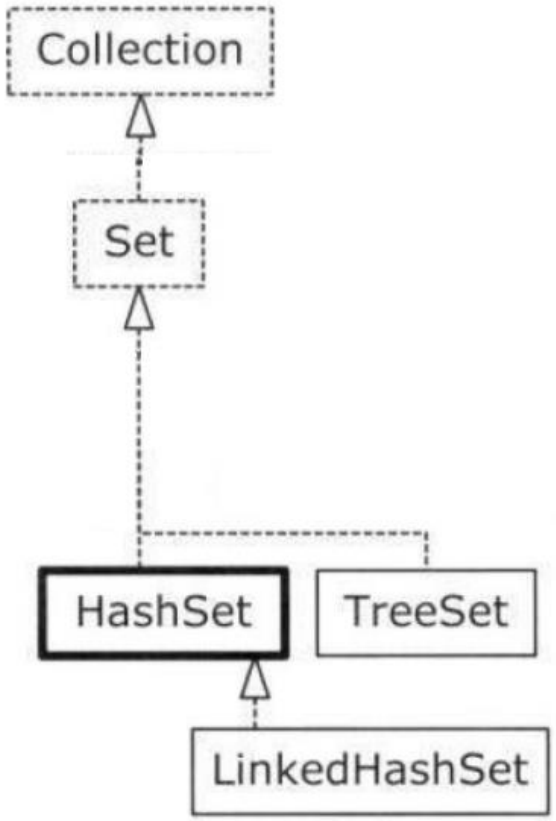
Thus, the elements in a `SortedSet` are guaranteed to be in **sorted order**.

This allows for the following interesting methods:

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

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

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)`

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.