Book: Head First

Chapter-16:
Data Structures

Upcode Software
Engineer Team

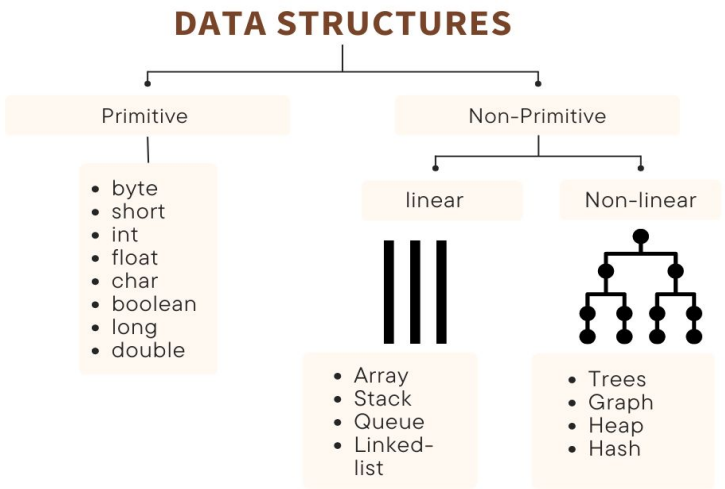
-2023-

CONTENT

1. Data Structures
2. Collection Framework
3. Generic Methods & Type
4. Summary

Data Structures (1/n)

- The **Java Collections Framework** has **a data structure** that should work for virtually anything you'll ever need to do.
- **A data structure** is a storage that is used to store and organize data. It is a way of arranging data on **a computer so that it can be accessed and updated efficiently**.



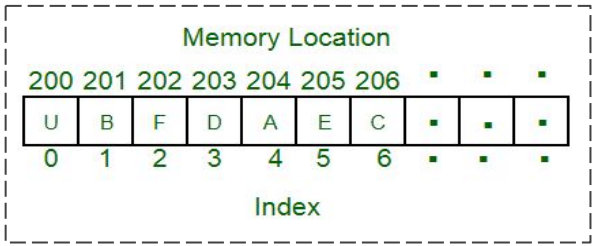
Data Structures (2/n)

- **Linear data structure:** Data structure in which data elements are arranged sequentially or linearly, where each element is attached to its previous and next adjacent elements, is called a linear data structure.

  Examples of linear data structures are array, stack, queue, linked list, etc.

  - **Static data structure:** Static data structure has a fixed memory size. It is easier to access the elements in a static data structure.

    An example of this data structure is an array.



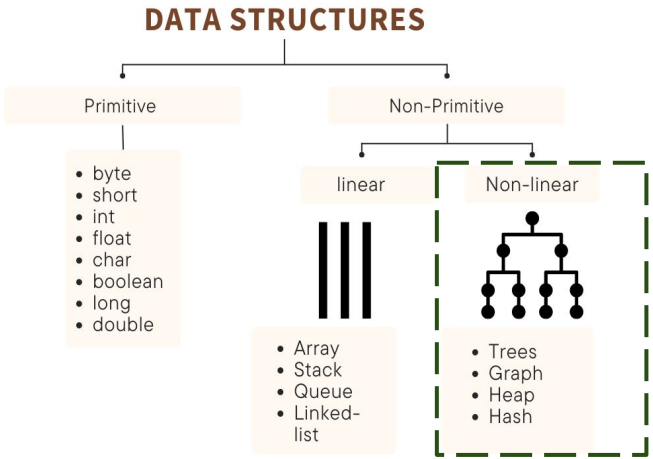
  - **Dynamic data structure:** In dynamic data structure, the size is not fixed. It can be randomly updated during the runtime which may be considered efficient concerning **the memory (space) complexity of the code.**

    *Examples of this data structure are **queue, stack, etc.***

Data Structures (3/n)

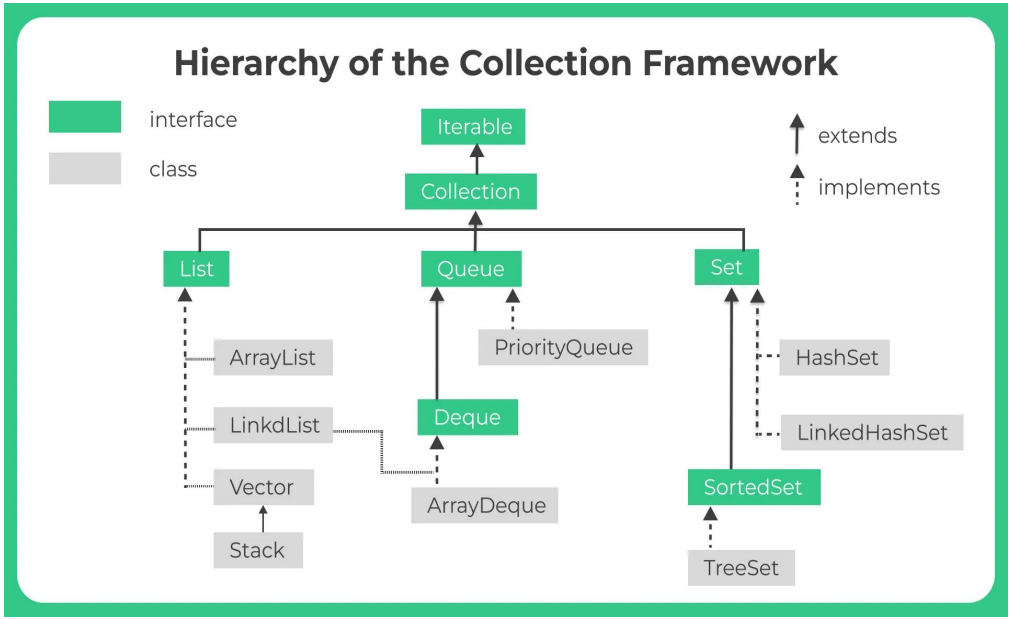
- **Non-linear data structure:** Data structures where data elements are not placed sequentially or linearly are called non-linear data structures. In a non-linear data structure, **we can't traverse all the elements in a single run only**.

  *Examples of non-linear **data structures are trees and graphs.***



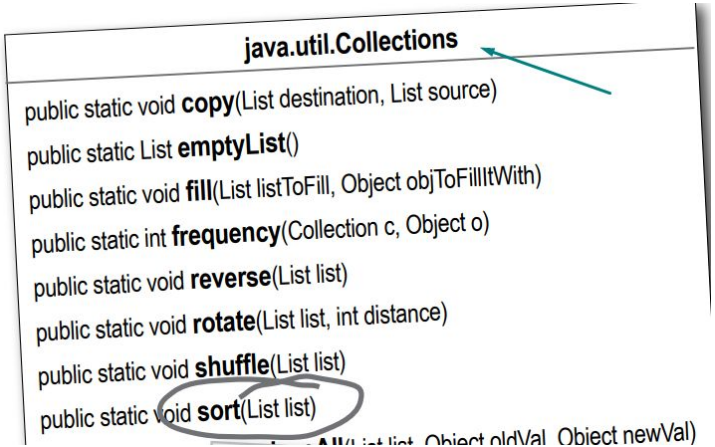
Collection Framework (1/n)

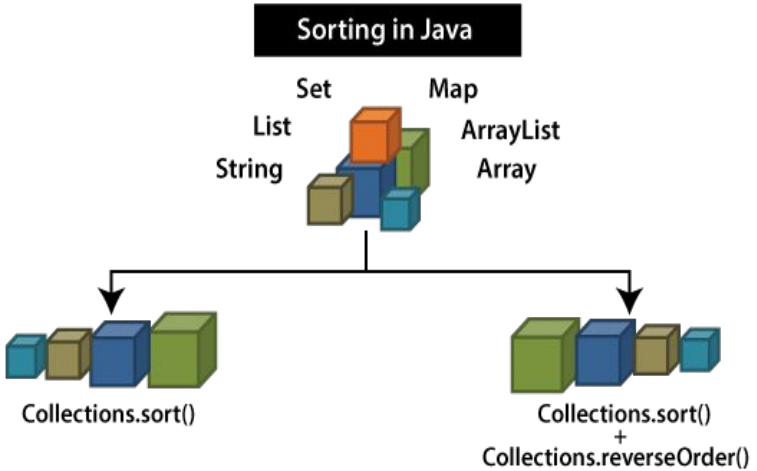
- **ArrayList** is not the only collection
- a **TreeSet** instead of an **ArrayList**, the Strings would automatically land in the right place, alphabetically sorted.



Collection Framework (2/n)

- There IS a sort() method in the **Collections** class.
- It takes a **List, and since ArrayList implements the List interface**, **ArrayList IS-A List.**
- Thanks to polymorphism, you can pass an ArrayList to **a method declared to take List.**





Collection Framework (3/n) - Array/ ArrayList

- Array and ArrayList

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

// Arrays
Integer[] number = new
Integer[10];
for (int i=0; i<9; i++){
number[i]=i;
}
System.out.println(number[8]);

String[] gfg = new String[] {
"G", "E", "E", "K", "S" };
System.out.println(gfg.length);
System.out.println(gfg);
}
}
```

Q: But you CAN add something to an ArrayList at a specific index instead of just at the end—there's an overloaded add() method that takes an int along with the element to add. So wouldn't it be slower than inserting at the end?

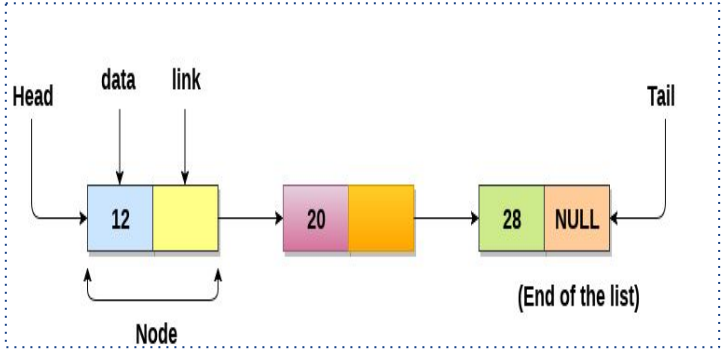
A: Yes, it's slower to insert something in an ArrayList somewhere *other* than at the end. So using the overloaded add(index, element) method doesn't work as quickly as calling the add(element)—which puts the added element at the end. But most of the time you use ArrayLists, you won't need to put something at a specific index.

Collection Framework (4/n) - LinkedList

- Firstly, **a Linked List** is a collection of things known as nodes that are kept **in memory at random.**
- Secondly, **a node has two fields: data saved at that specific address and a pointer to the next node in the memory**.
- The **null pointer is contained in the list's last node**

```
public class Node {
      int data;
      Node next;

      Node(int d){
            data =d;
            next = null;
      }
}
```



Node Sample Code

Collection Framework (5/n) - LinkedList

- a self-referential class in Java can be used to build a singly linked list of Java.
- SinglyLinkedList that encloses an inner self-referential class Node that has two fields: an integer data field and a Node type “next” field. In addition, the outer class contains a reference/pointer/link to the HEAD of the document.

```
public class LinkedListExamples {

Node head; // head of list

public void display(){
Node n = head;
      while (n!=null){
      System.out.println(n.data +
      "\n");
      n = n.next;
}
}
```

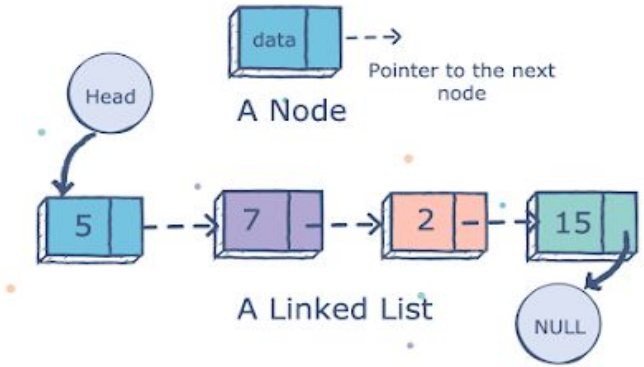
```
public static void main(String[] args) {
LinkedListExamples list = new
LinkedListExamples();
      list.head = new Node(100);
      Node second = new Node(2);
      Node third = new Node(332);

      list.head.next = second;
      second.next = third;
      list.display();
}
}
```

Collection Framework (6/n)

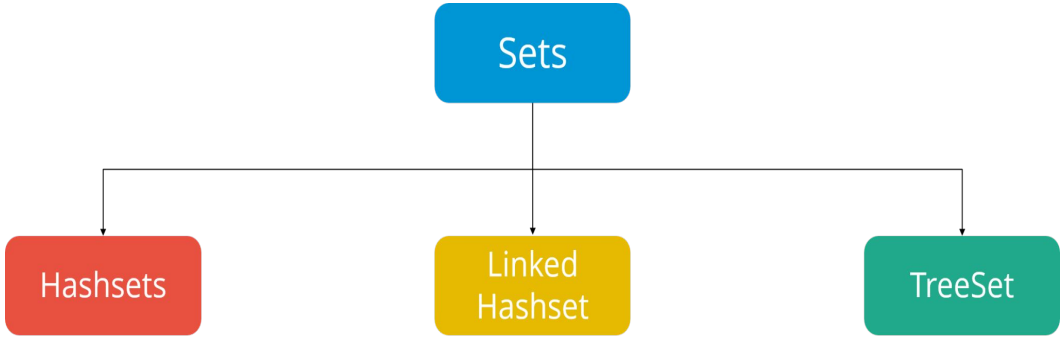
Q: I see there's a LinkedList class, so wouldn't *that* be better for doing inserts somewhere in the middle? At least if I remember my Data Structures class from college...

A: Yes, good spot. The LinkedList *can* be quicker when you insert or remove something from the middle, but for most applications, the difference between middle inserts into a LinkedList and ArrayList is usually not enough to care about unless you're dealing with a *huge* number of elements. We'll look more at LinkedList in a few minutes.



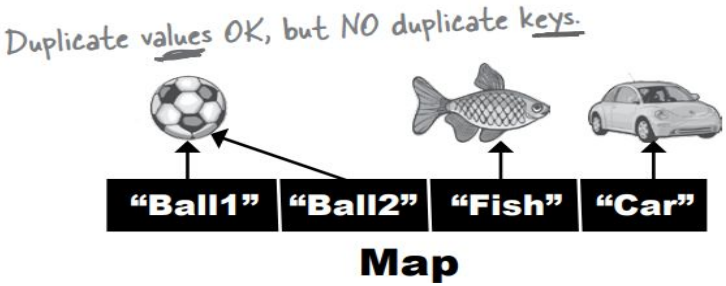
Collection Framework (7/n) - SET

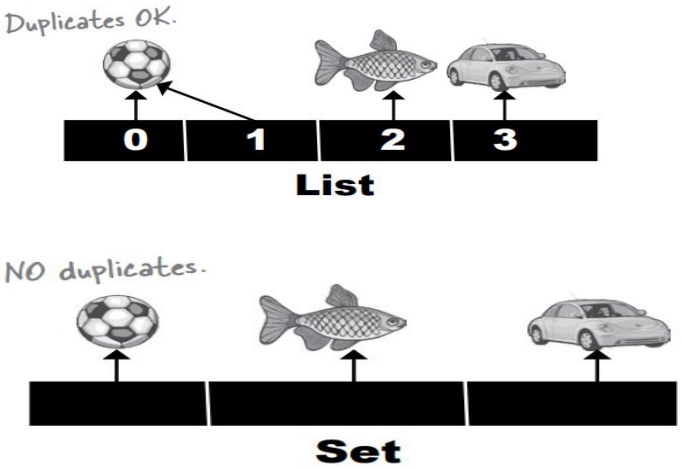
- The sorting all works, but now we have duplicates.
- **The sorted list contains duplicates**.
- *Collections* sort is a method of ***Java Collections*** **class used to sort a list**, which implements the *List* interface. **All the elements in the list must be mutually comparable**. If a list consists of string elements, then **it will be sorted in alphabetical order.**



Collection Framework (8/n)

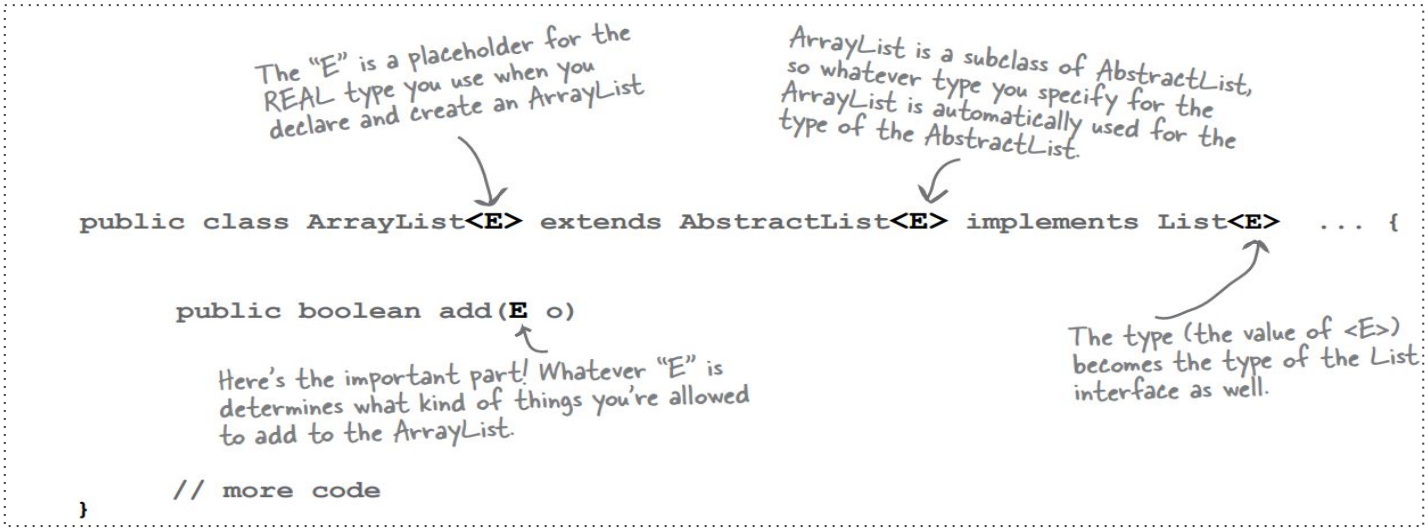
- **LIST** - when sequence matters Collections that know about index position.
- **SET** - when uniqueness matters Collections that do not allow duplicates.
- **MAP** - when finding something by key matters Collections that use key-value pairs





Generic Methods & Type (1/n)

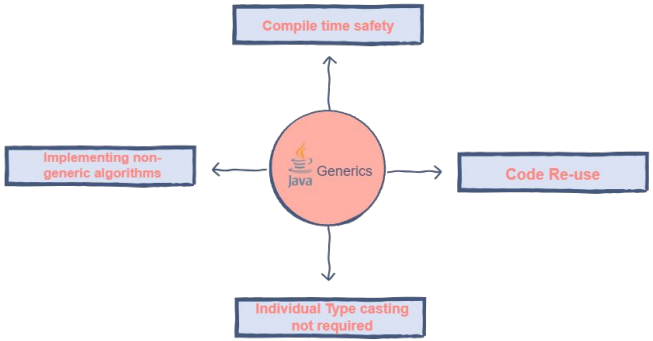
- We'll just say it right here—virtually all of the code you write that deals with **generics will be collection-related code**.
- **Generics** can be used in other ways, the main point of generics is to let you write **type-safe collections**.

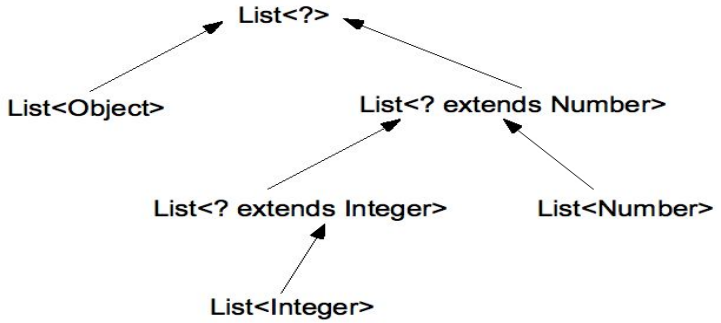


Generic Methods & Type (2/n)

- **Java Generic Type Naming** convention helps us understanding code easily and having a naming convention is one of the best practices of Java programming language.
- So generics also comes with its own naming conventions.
- Usually, type parameter names are single, uppercase letters to make it easily distinguishable from java variables. The most commonly used type parameter names are: E – Element (used extensively by the Java Collections Framework, for example ArrayList, Set etc.)
- **K – Key (Used in Map) N – Number T – Type V – Value (Used in Map) S,U,V etc. – 2nd, 3rd, 4th types**

Generic Methods & Type (3/n)

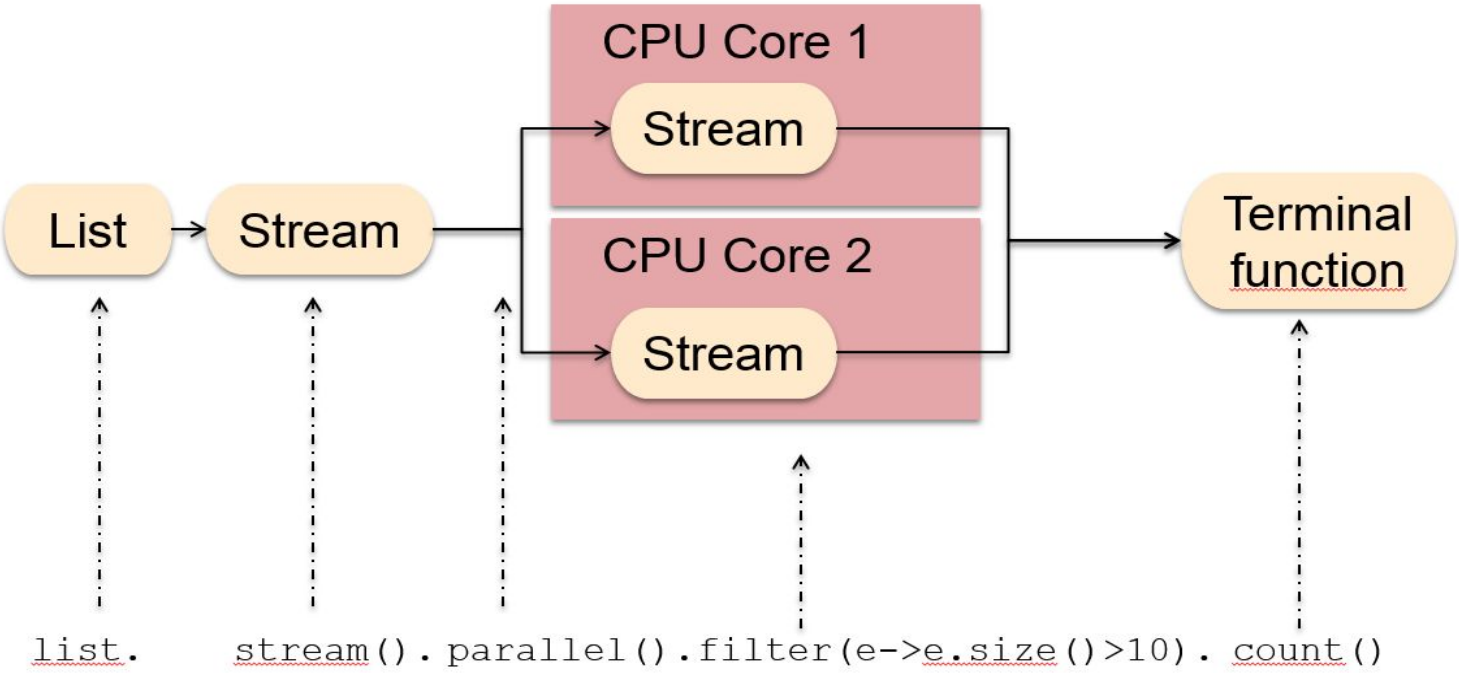




Taxonomy of the generic *List* types

```
Set<Object> setOfAnyType = new HashSet<Object>();
setOfAnyType.add("abc"); //legal
setOfAnyType.add(new Float(3.0f)); //legal - <Object> can accept any type
```

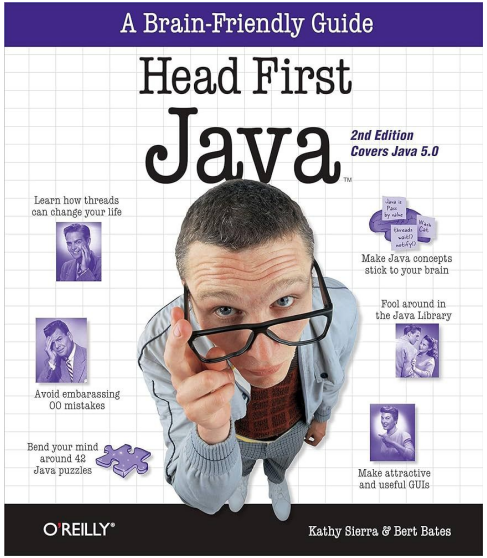
Stream API and CPU

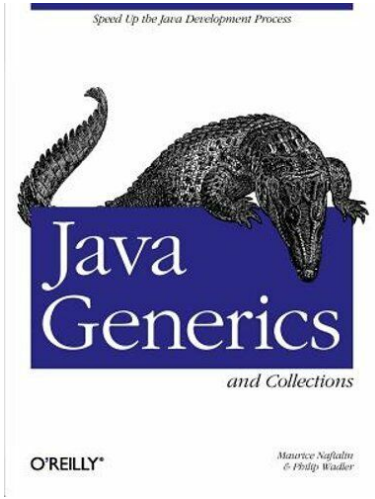


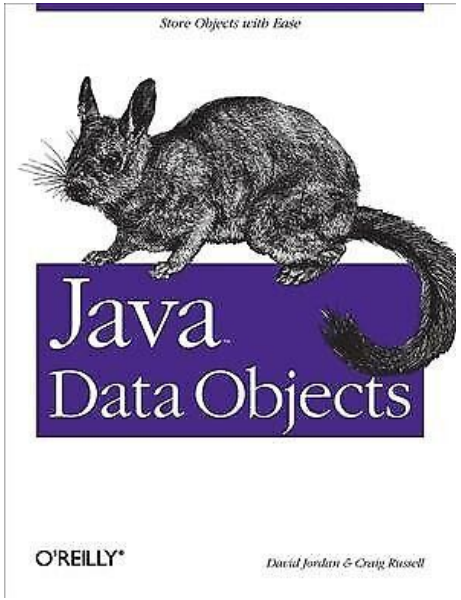
Summary

1. **Map** interface doesn't actually extend the Collection interface, but Map is still considered part of the "**Collection Framework**" (also known as the "Collection API").
2. Using a HashSet instead of ArrayList

Reference Book







Reference Resources?

1. **Head First (book)**
2. **Data Structure** (resource)
3. **Java Generics and Collections: Speed Up the Java Development Process by Naftalin**
4. **Java Generic** Blog

Thank you!

Presented by

Hamdamboy Urunov

(hamdamboy.urunov@gmail.com)