

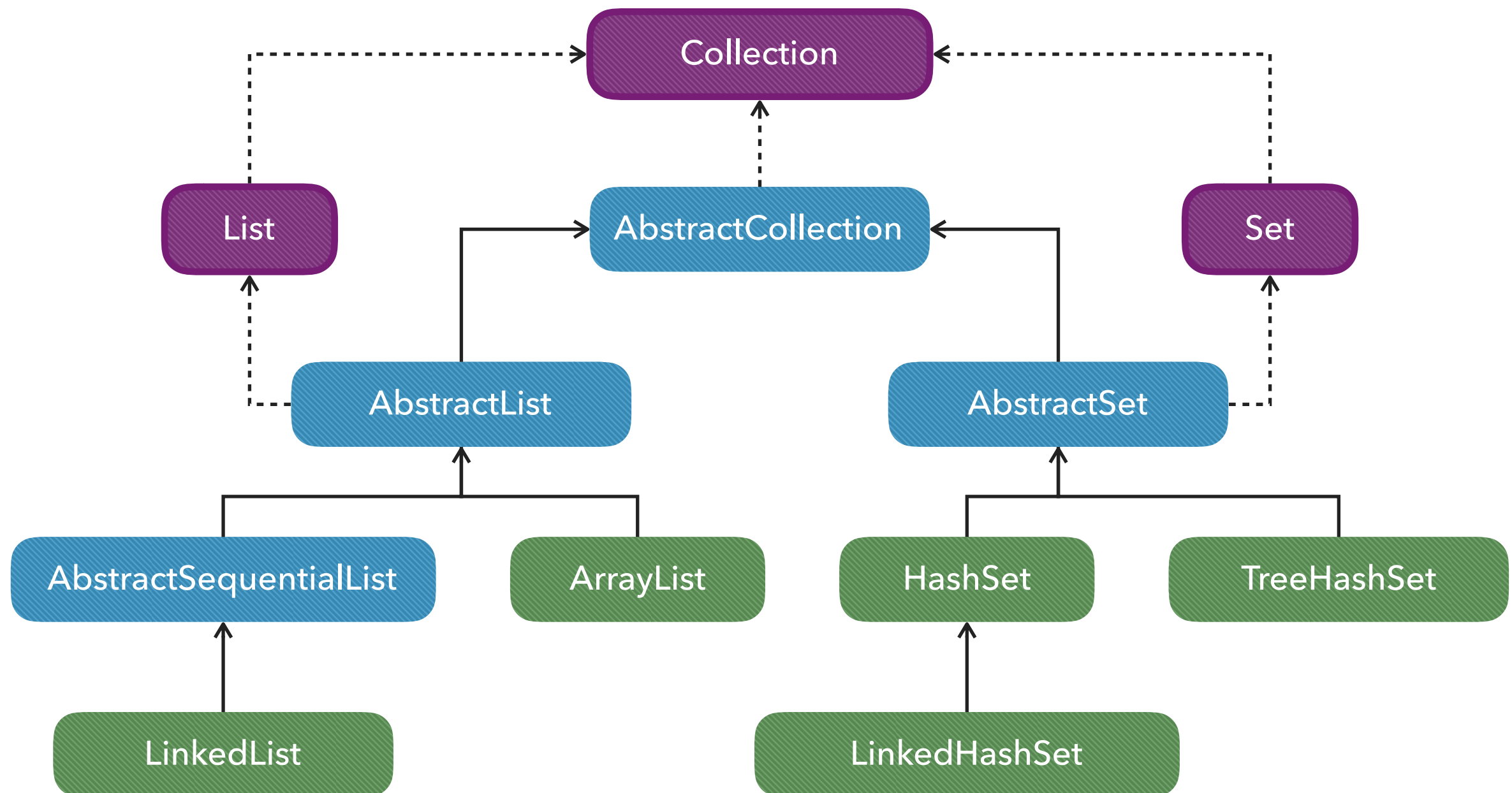
COLLECTIONS API

OVERVIEW

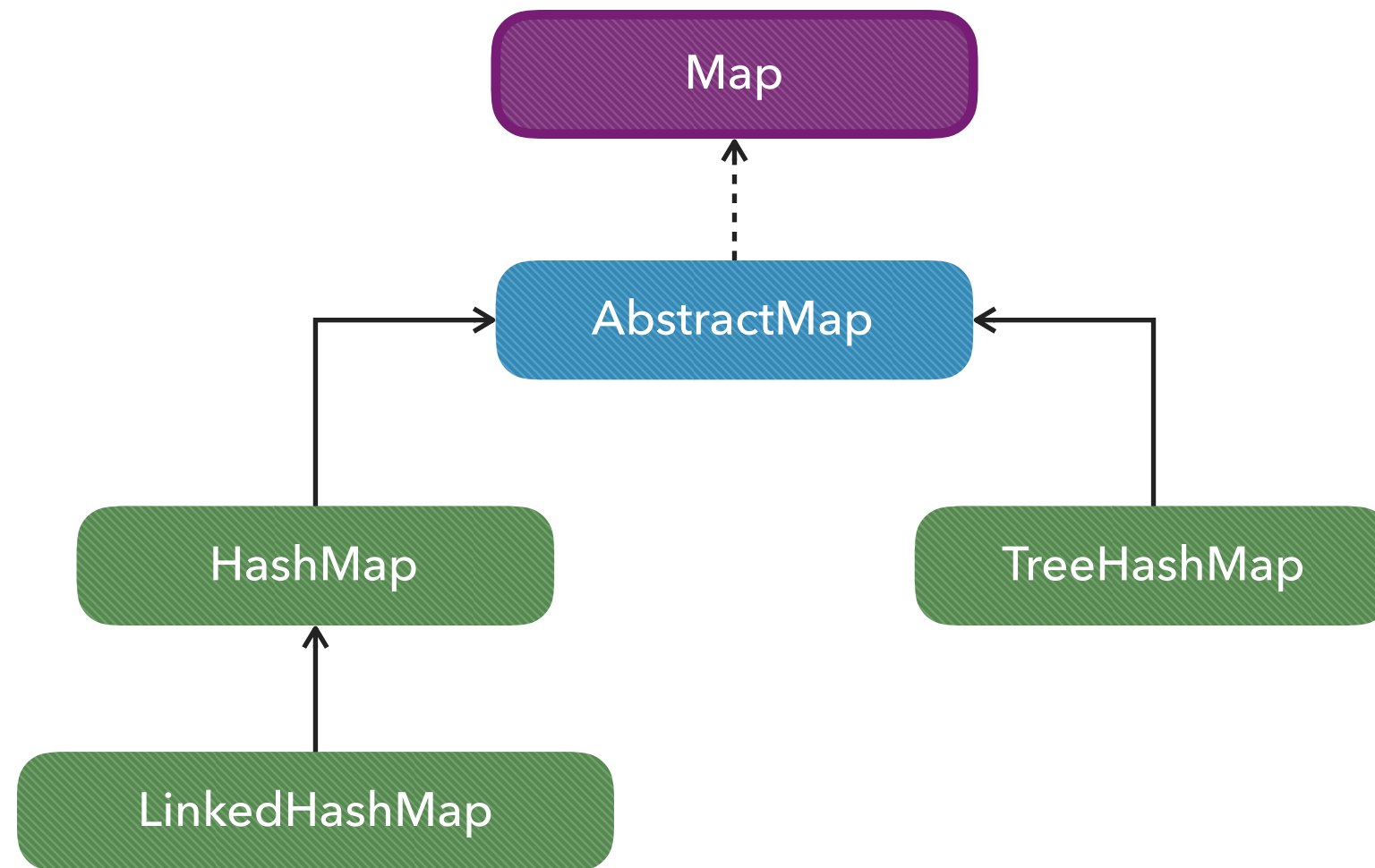
REASONING

- ▶ Plain data structures (e.g. arrays) are **simple** and **fast**, but **cumbersome** to work with
- ▶ Initially Java **provided** some tools to store and manipulate group of objects, but they **lacked** unifying theme
- ▶ Language developers wanted to **design** such framework, that would **meet** several goals
 - ▶ **High** performance
 - ▶ Support high degree of **interoperability** and **abstraction**
 - ▶ **Extend** and **adapt** collections easily

COLLECTIONS API HIERARCHY: COLLECTION



COLLECTIONS API HIERARCHY: MAP



1. COLLECTION CHARACTERISTICS

- ▶ Ordered
 - ▶ Whether it is **possible** to iterate over the elements of an ordered collection in a **predictable** order
- ▶ Uniqueness of elements
 - ▶ Some collections **do not** allow **duplicate** elements
- ▶ Thread safety
 - ▶ Whether it is **safe** to work with collection in **multithreaded** environment

2. COLLECTION CHARACTERISTICS

- ▶ Underlying storage structure
 - ▶ **Array** based storage
 - ▶ Fast to access but slow to remove or insert
 - ▶ **Linked-list** based storage
 - ▶ Efficient at removing or inserting but slower for access
 - ▶ **Hash** based storage
 - ▶ Reasonably efficient access
 - ▶ **Tree** based storage
 - ▶ Efficient for searching

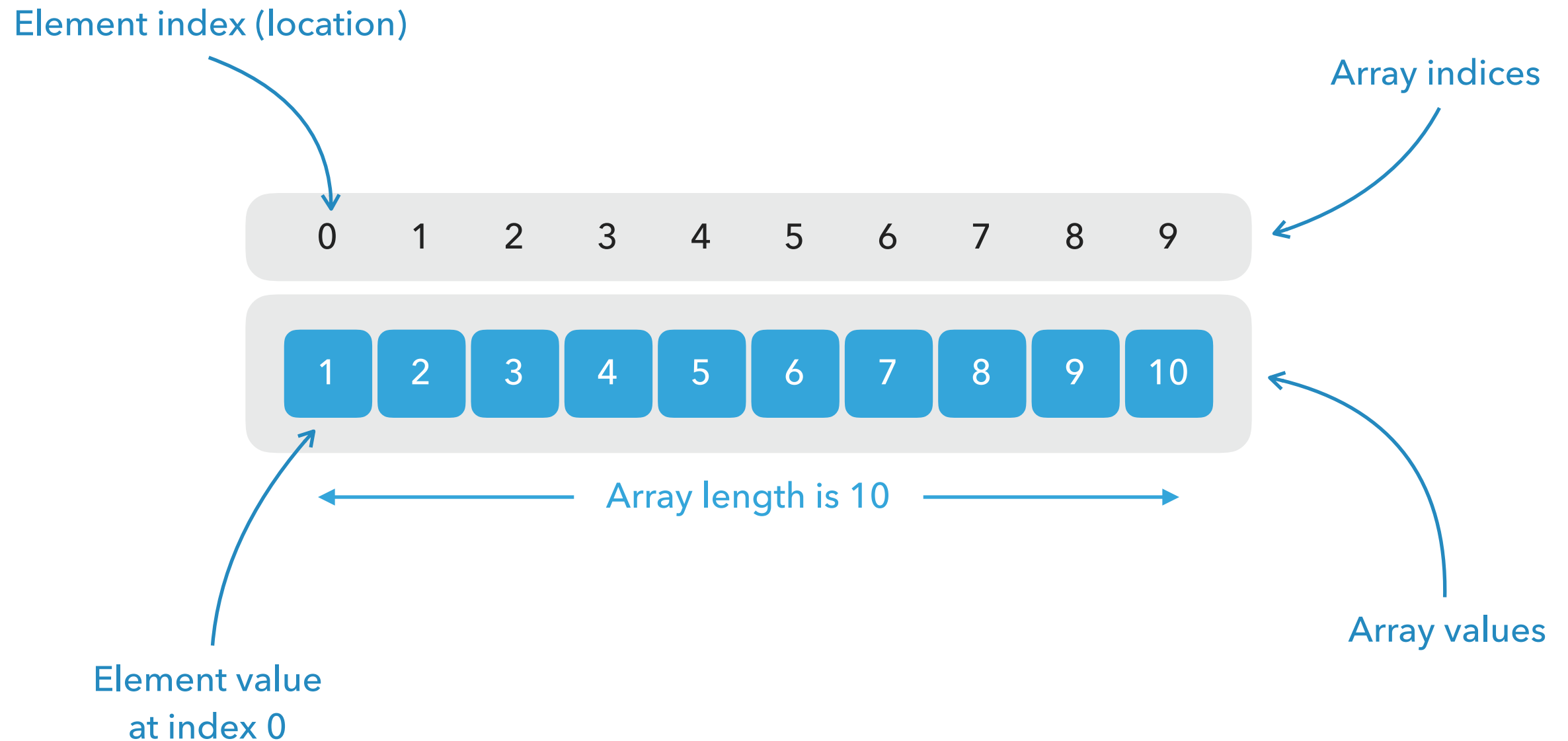
ARRAYLIST

INTERNALS

RESIZABLE ARRAY IMPLEMENTATION OF THE LIST INTERFACE

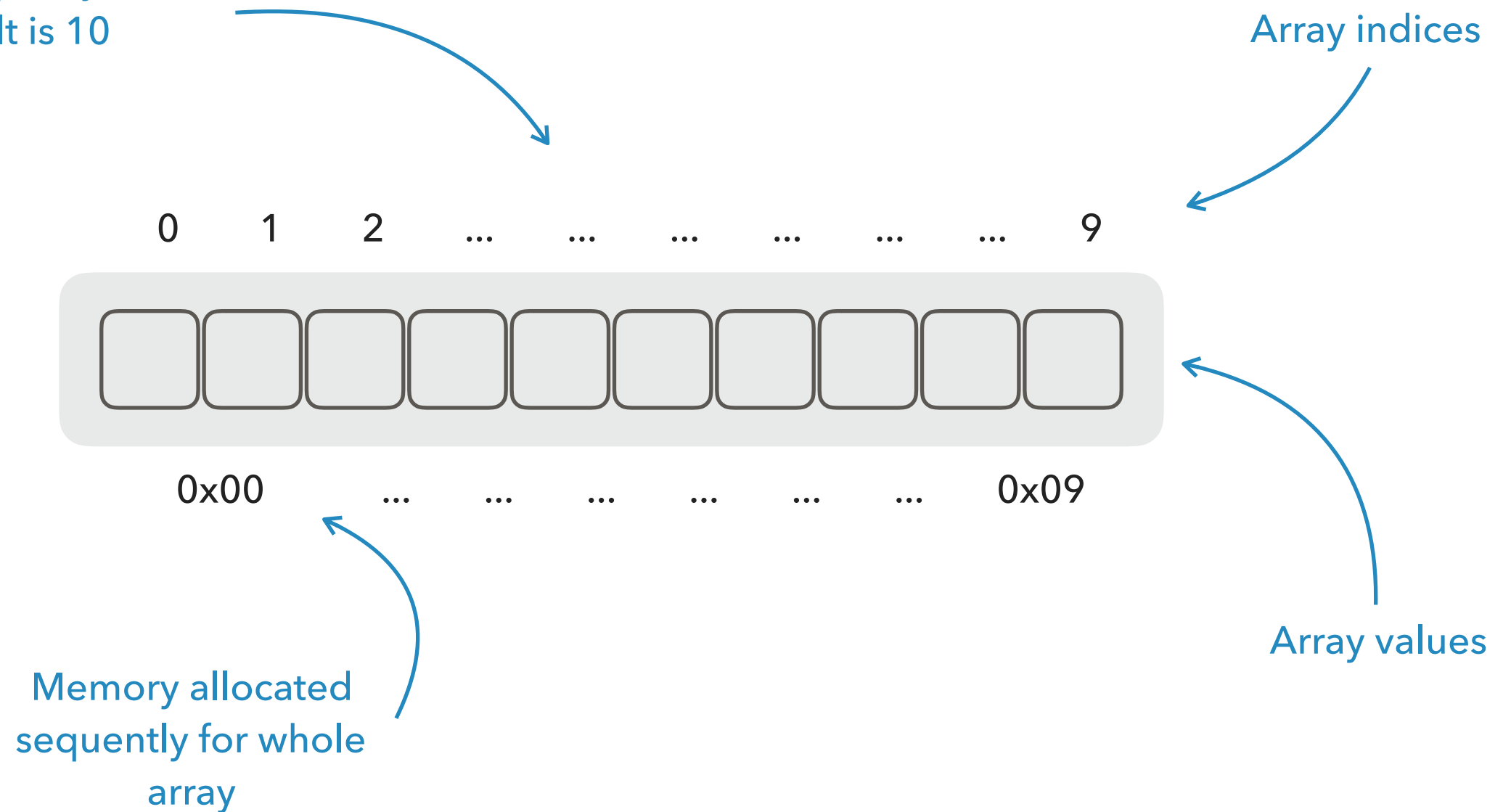
Java Documentation

ARRAYLIST INTERNAL DATA STRUCTURE REPRESENTATION



1. ARRAYLIST INSERTION PROCESS: INITIALIZATION

Initial array capacity (size)
by default is 10

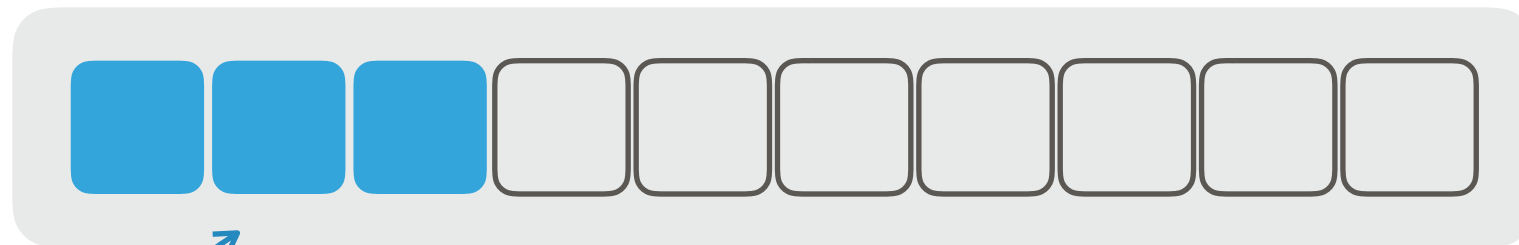


2. ARRAYLIST INSERTION PROCESS: ADDING ELEMENTS

Array capacity (size)
remains unchanged

Array indices

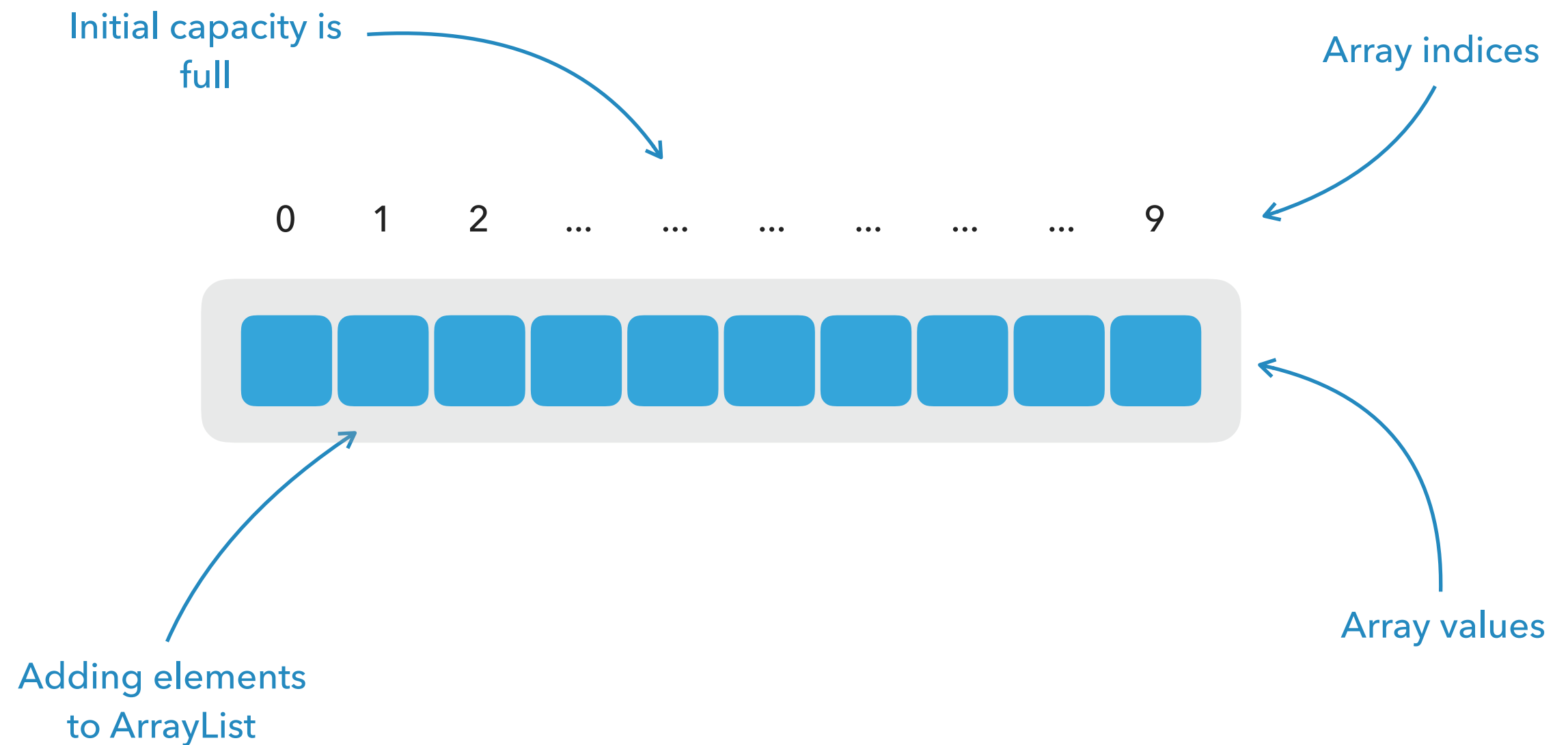
0 1 2 9



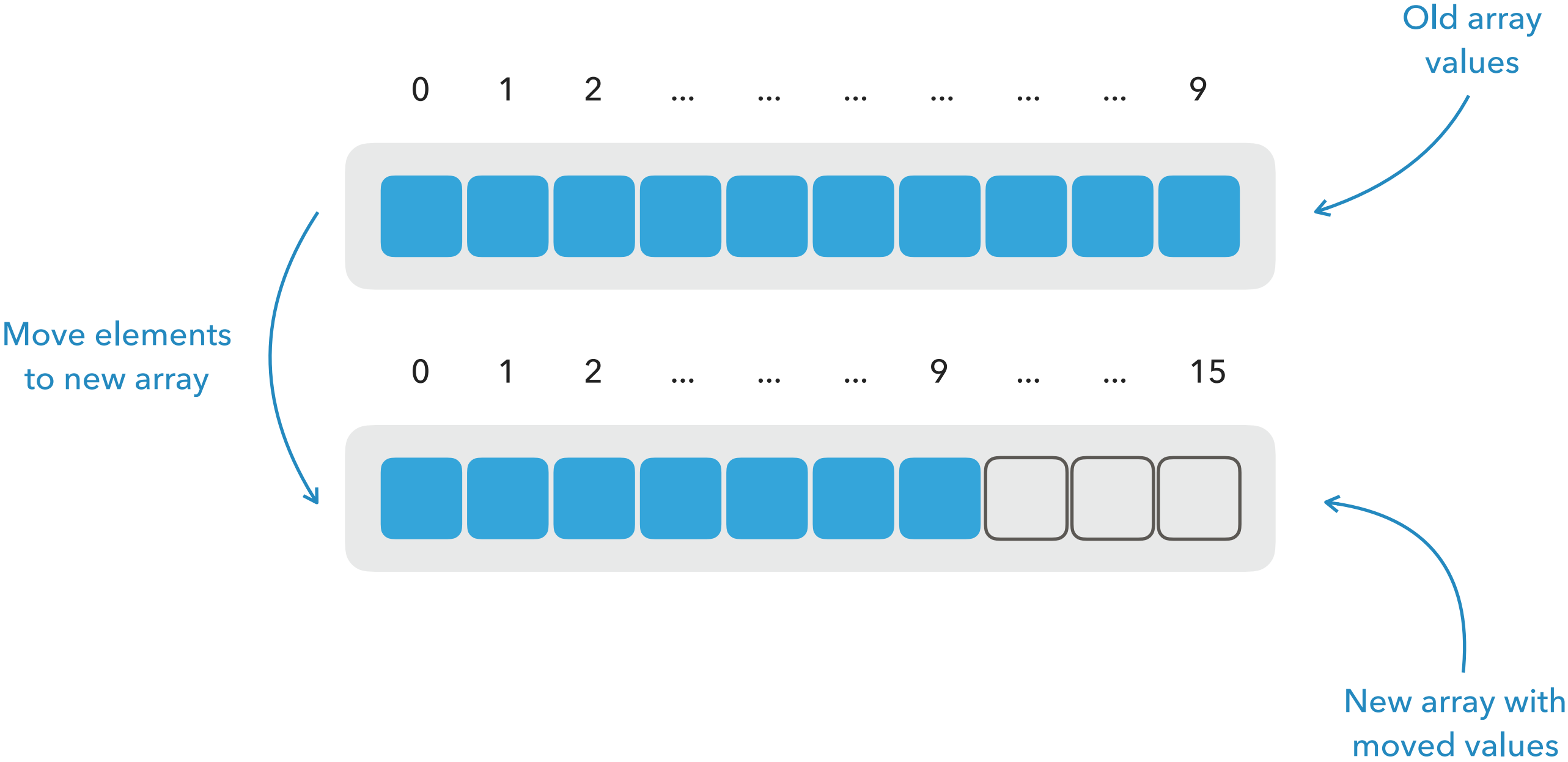
Adding elements
to ArrayList

Array values

3. ARRAYLIST INSERTION PROCESS: HIT CAPACITY CAP



4. ARRAYLIST INSERTION PROCESS: COPY TO NEW ARRAY



ARRAYLIST CAPACITY INCREMENT EQUATION

Increase capacity by roughly 50%



```
int newCapacity = (oldCapacity * 3) / 2 + 1;
```

ARRAYLIST (WITH DEFAULT CAPACITY): CODE EXAMPLE

Code

```
List<String> scaryStories = new ArrayList<>();  
scaryStories.add("Your browser history is public");  
scaryStories.add("You didn't kill that spider");  
  
for (String story : scaryStories) { System.out.println(story); }
```

Console output

```
Your browser history is public  
You didn't kill that spider
```

ARRAYLIST (WITH SPECIFIED CAPACITY): CODE EXAMPLE

Code

```
List<String> scaryStories = new ArrayList<>(15);  
scaryStories.add("Your browser history is public");  
scaryStories.add("You didn't kill that spider");  
  
for (String story : scaryStories) { System.out.println(story); }
```

Console output

```
Your browser history is public  
You didn't kill that spider
```


ARRAYLIST CHARACTERISTICS: RECAP

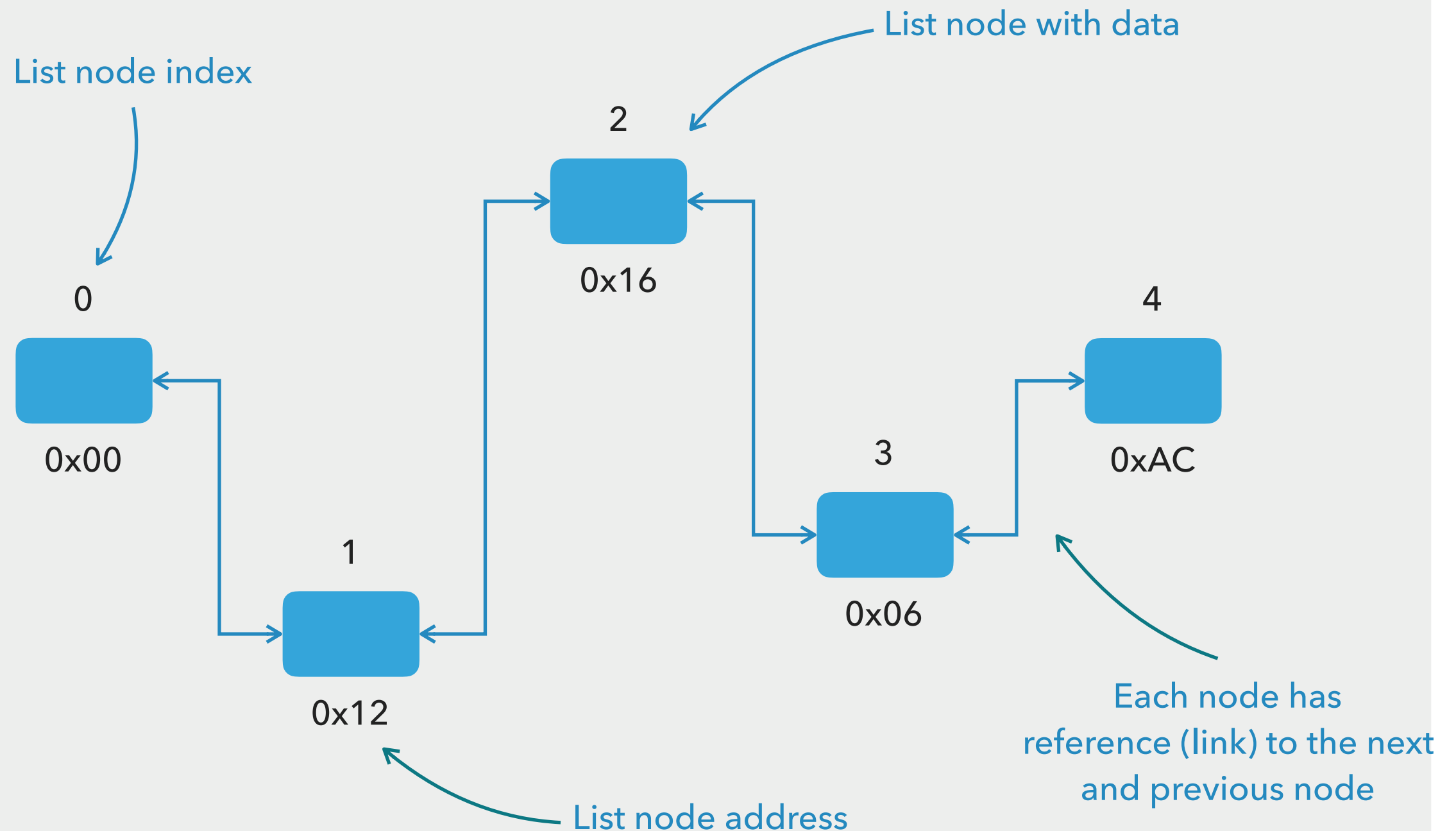
- ▶ It is a **resizable** array, also called a dynamic array
- ▶ It internally uses an **array** to store the elements
- ▶ It **allows** duplicate values
- ▶ It is an **ordered** collection
- ▶ It can store only **non-primitive** values

LINKEDLIST INTERNALS

DOUBLY-LINKED LIST IMPLEMENTATION OF THE LIST AND DEQUE INTERFACES

Java Documentation

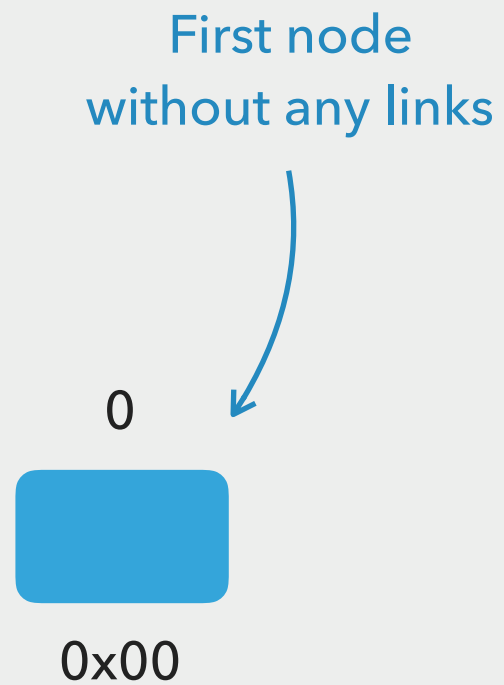
LINKEDLIST INTERNAL DATA STRUCTURE REPRESENTATION



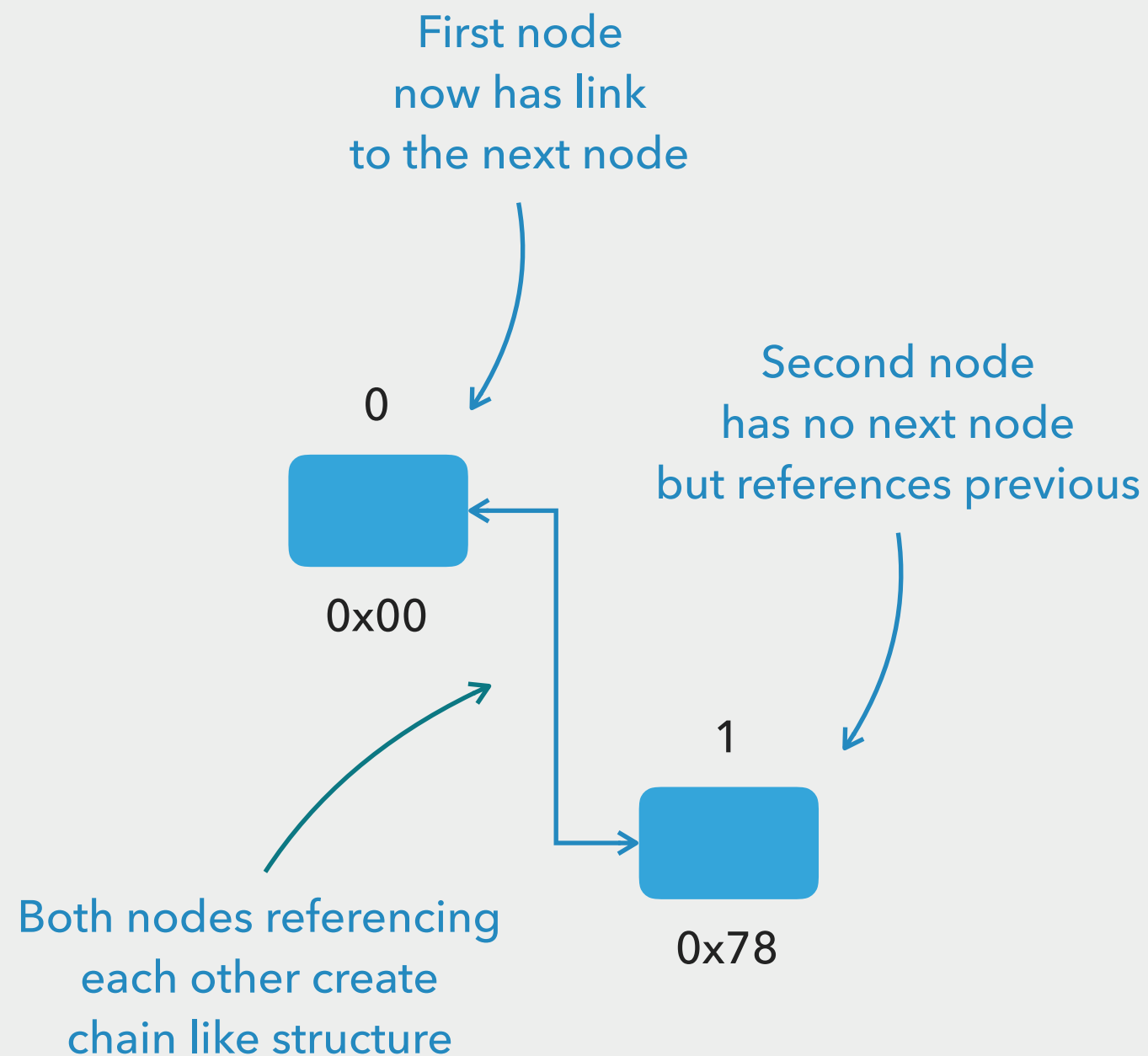
1. LINKEDLIST INSERTION PROCESS: INITIALIZATION

It starts totally empty..

2. LINKEDLIST INSERTION PROCESS: ADDING FIRST ELEMENT



2. LINKEDLIST INSERTION PROCESS: ADDING NTH ELEMENTS



LINKEDLIST: CODE EXAMPLE

Code

```
List<String> things = new LinkedList<>();  
things.add("Computer");  
things.add("Coffee");  
  
for (String thing: things) { System.out.println(thing); }
```

Console output

```
Computer  
Coffee
```


LINKEDLIST CHARACTERISTICS: RECAP

- ▶ Internally uses **distinct** objects which are **referencing** each other
- ▶ It **allows** duplicate and null values
- ▶ It is an **ordered** collection
- ▶ It can store only **non-primitive** values

ARRAYLIST AND LINKEDLIST KEY DIFFERENCES

- ▶ Memory consumption:
 - ▶ LinkedList **consumes more memory** than an ArrayList because it also stores the next and previous **references** along with the **data**
- ▶ Accessing data:
 - ▶ An element can be **accessed** in an ArrayList in $O(1)$ time (directly **by index**)
 - ▶ It takes $O(n)$ time to access an element in a LinkedList (**traverse** to the desired element though references)
- ▶ Addition or removal:
 - ▶ ArrayList is **usually slower**, because the elements in the ArrayList needs to be **shifted** if element is added or removed in the middle (capacity changes matter as well)
 - ▶ LinkedList is **faster** because only references must be changed

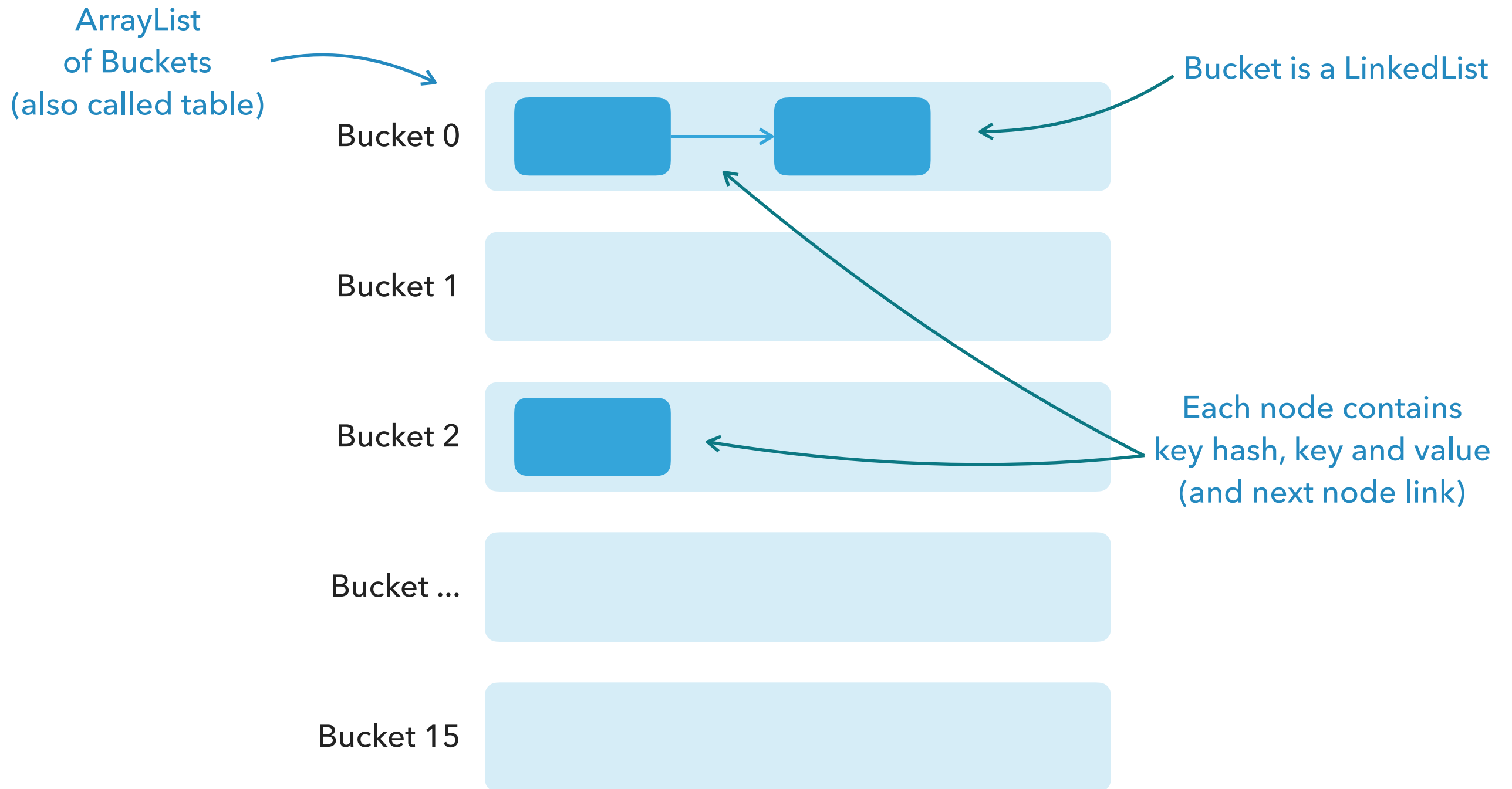
HASHMAP

INTERNALS

HASH TABLE BASED IMPLEMENTATION OF THE MAP INTERFACE

Java Documentation

HASHMAP INTERNAL DATA STRUCTURE REPRESENTATION



HASHING 101

- ▶ Hash function is a function that produces determined value
- ▶ For every argument there is unique hash produced
- ▶ Whenever hash function is invoked with the same argument more than once, the hash value is consistently the same
- ▶ Equal arguments should return equal hashes
- ▶ Whenever two different arguments return equal hashes, it is called collision
- ▶ Hash function is one way: original value cannot be obtained or calculated from hash

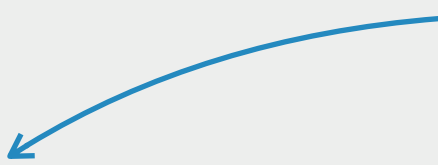
HASHCODE AND EQUALS CONTRACT

- ▶ Developers should **override both methods** in order to achieve a **fully working** equality mechanism
- ▶ If two objects are **equal** according to the *equals()* method, then calling the *hashCode()* method on each of the two objects must produce the **same integer result**

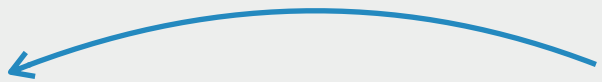
1. HASHCODE AND EQUALS CONTRACT: CODE EXAMPLE

```
public class Bag {  
  
    private String brand;  
    private String material;  
  
    public Bag(String brand, String material) {  
        this.brand = brand;  
        this.material = material;  
    }  
  
    @Override  
    public boolean equals(Object o) {  
        if (this == o) return true;  
        if (o == null || getClass() != o.getClass()) return false;  
        Bag bag = (Bag) o;  
        return Objects.equals(brand, bag.brand) &&  
            Objects.equals(material, bag.material);  
    }  
  
    @Override  
    public int hashCode() {  
        return Objects.hash(brand, material);  
    }  
}
```

Overriding equals
method by specifying
which fields to compare



Overriding hashCode
method by specifying
which fields to hash



2. HASHCODE AND EQUALS CONTRACT: CODE EXAMPLE

Code

```
Bag mk = new Bag("Michael Kors", "suede");  
Bag gucci = new Bag("Gucci", "leather");  
  
System.out.println("Michael Kors = " + mk.hashCode());  
System.out.println("Gucci = " + gucci.hashCode());
```

Console output

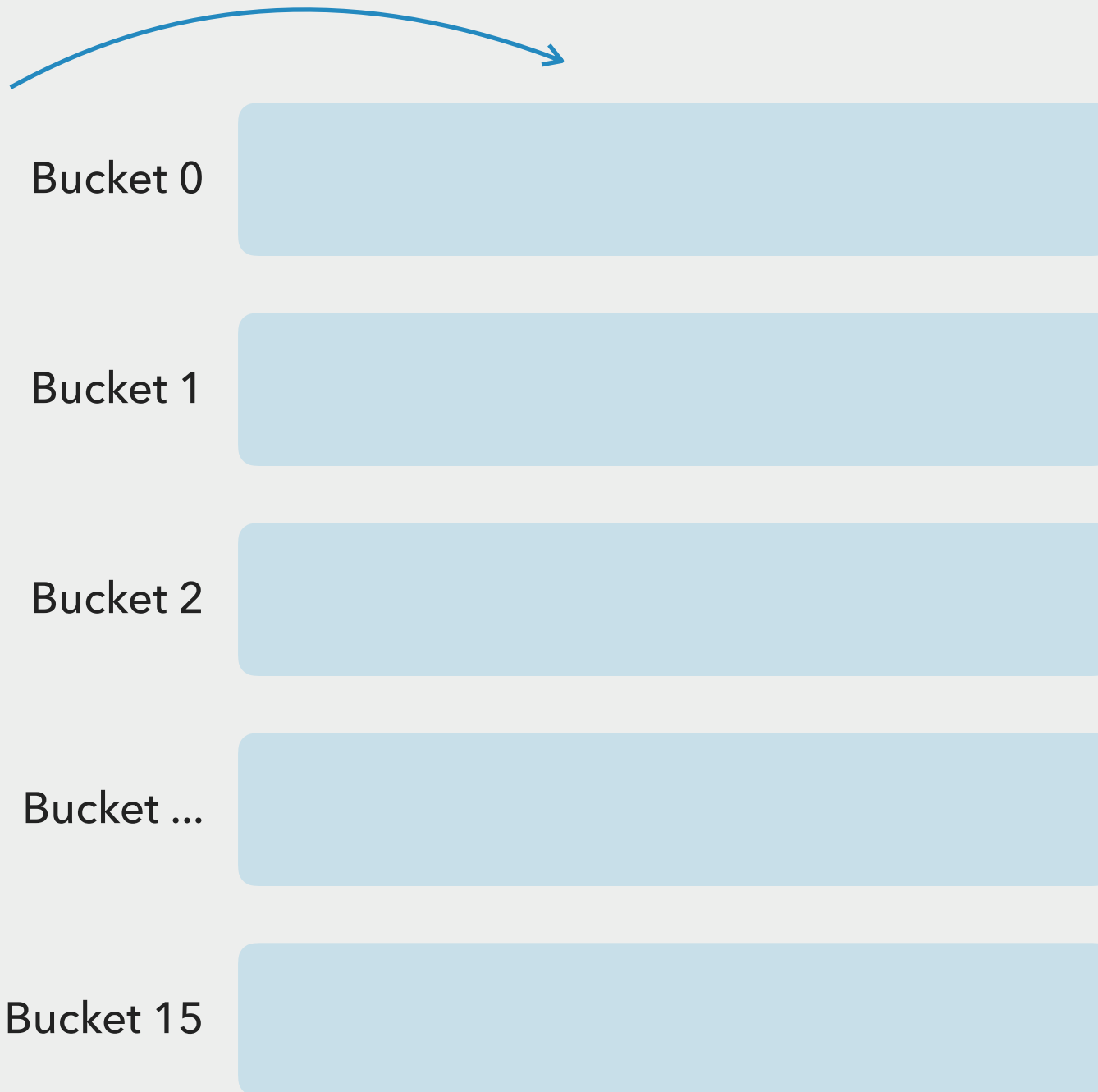
```
Michael Kors = 1944981575  
Gucci = -2100362481
```

HASHMAP USAGE OF EQUALITY CONTRACT

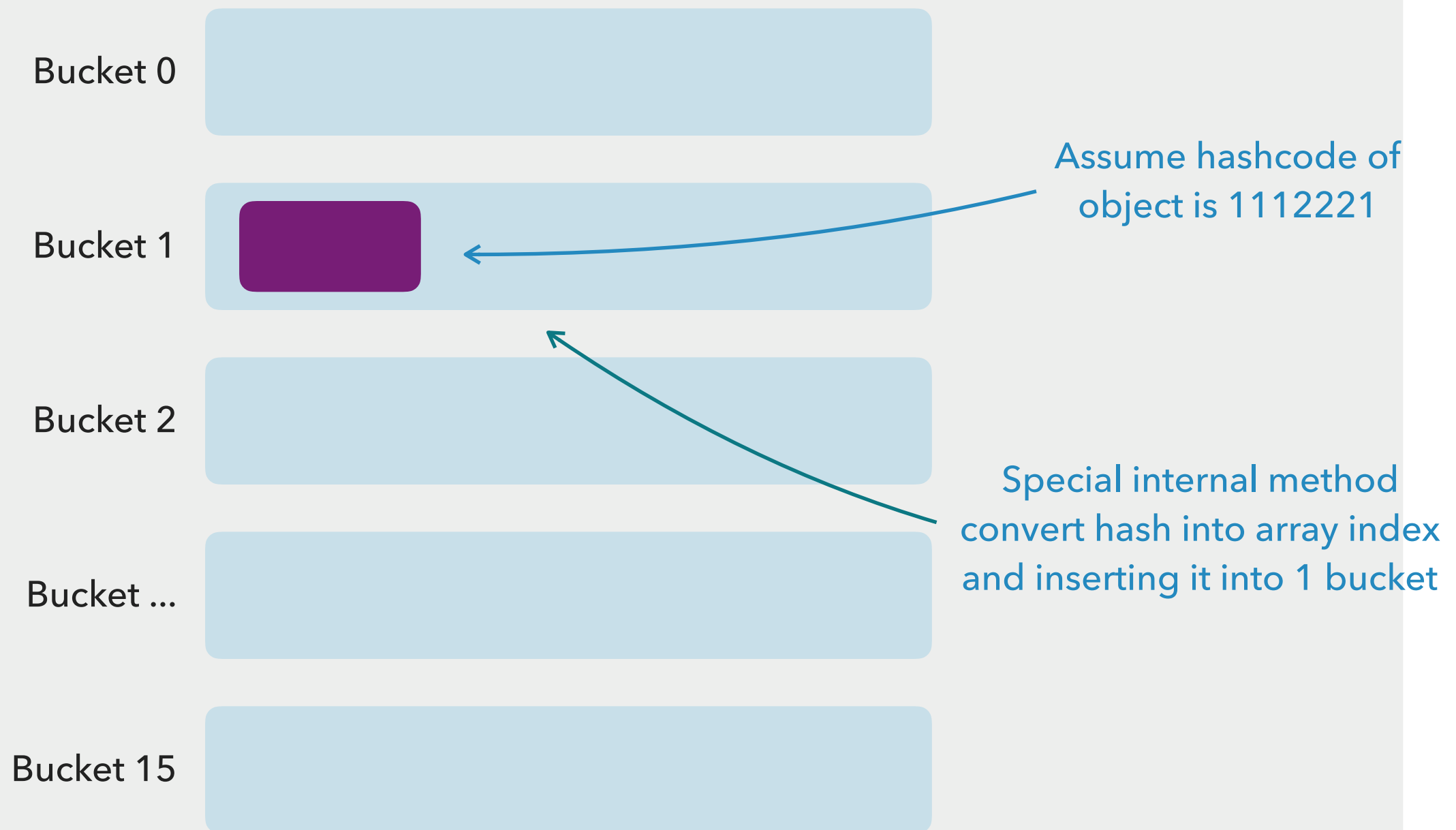
- ▶ An object's **hash** allows algorithms to put objects into **compartments** in order to increase lookup speed
- ▶ An object's **equal** method allows algorithms to **find exact** object in that **compartment**

1. HASHMAP INSERTION PROCESS: INITIALIZATION

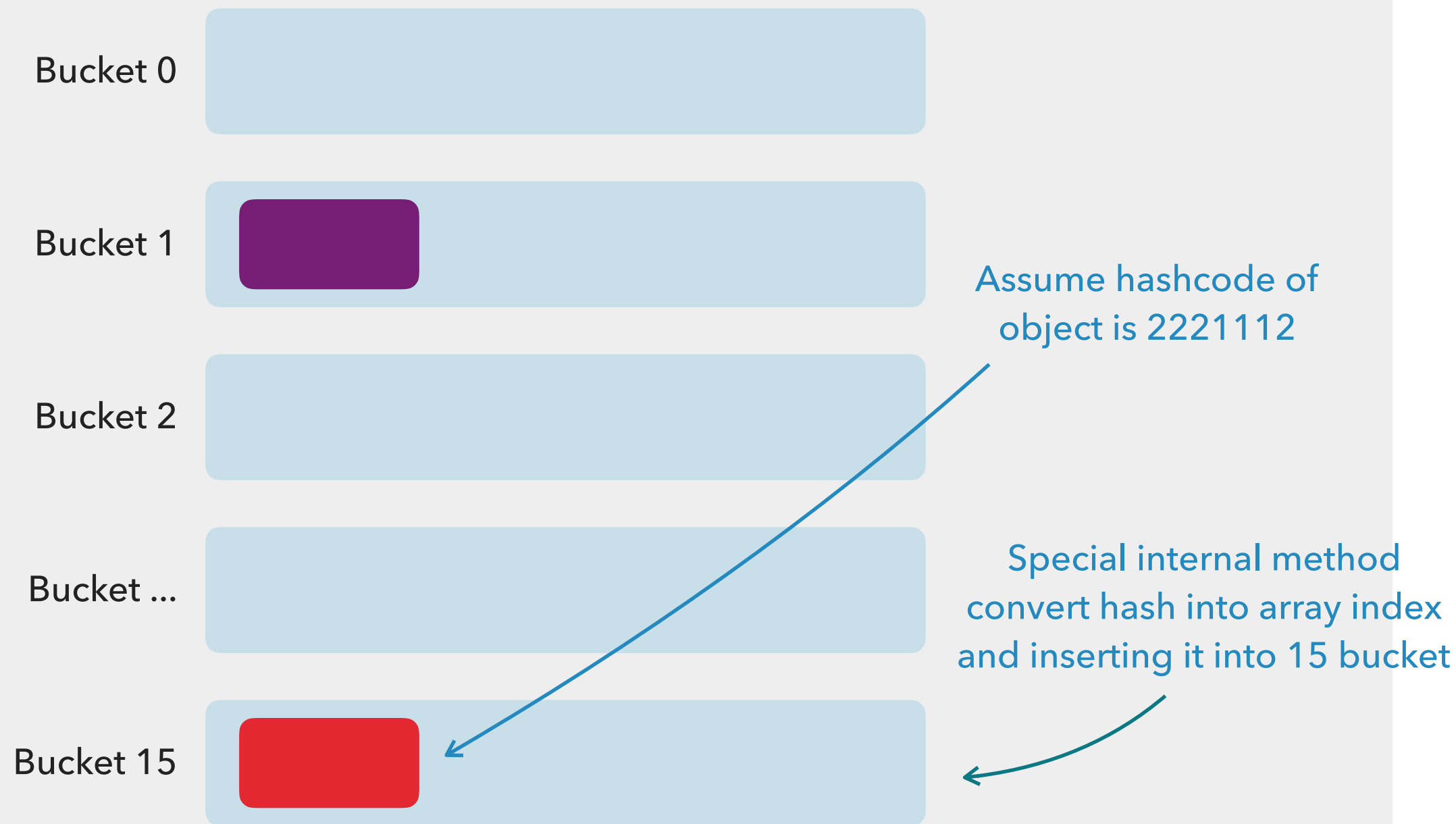
ArrayList with
empty 16 buckets
is created



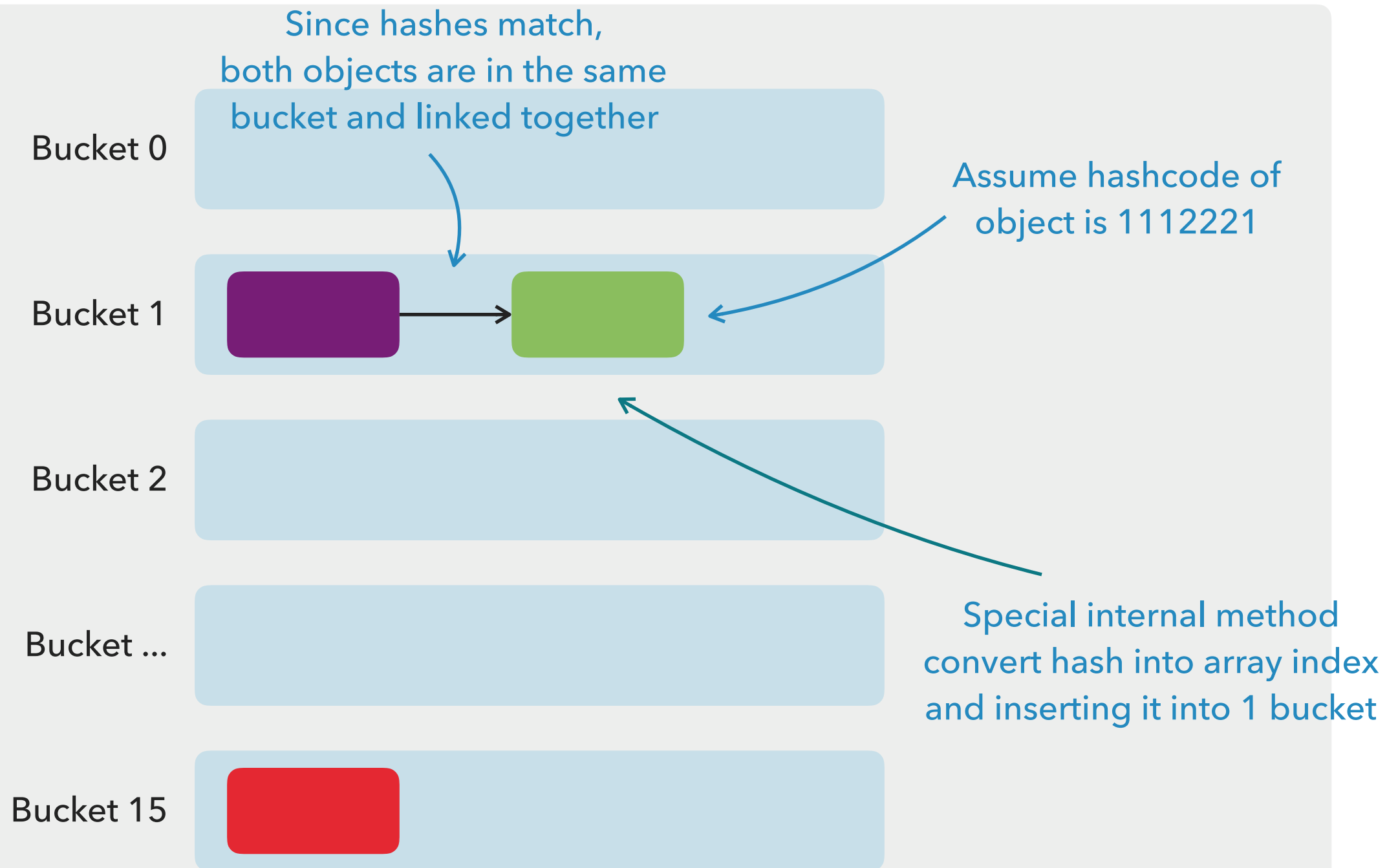
2. HASHMAP INSERTION PROCESS: ADDING ELEMENT



3. HASHMAP INSERTION PROCESS: ADDING ELEMENT



3. HASHMAP INSERTION PROCESS: ADDING ELEMENT



HASHMAP: CODE EXAMPLE

Code

```
Map<String, Integer> tableOfContents = new HashMap<>();  
tableOfContents.put("Introduction", 3);  
tableOfContents.put("Chapter 1", 15);  
tableOfContents.put("Chapter 2", 48);  
  
System.out.println(tableOfContents);
```

Console output

```
{Introduction=3, Chapter 1=15, Chapter 2=48}
```

HASHMAP CHARACTERISTICS: RECAP

- ▶ Internally uses `ArrayList` (with buckets as elements) and each bucket contains `LinkedList`
- ▶ It **doesn't allow** duplicate keys
- ▶ It **allows** single null key and multiple null values
- ▶ It is an **unordered** collection
- ▶ It can store only **non-primitive** values

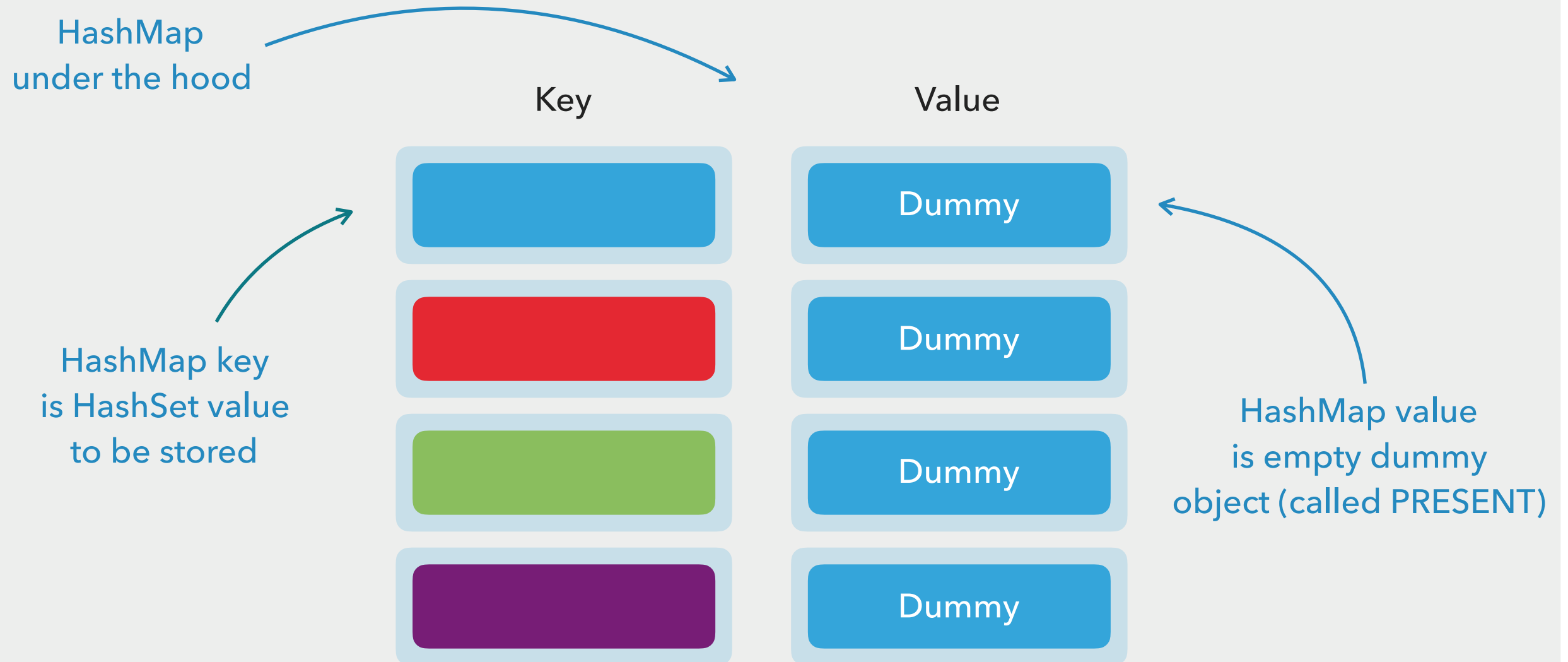
HASHSET

INTERNALS

**THIS CLASS IMPLEMENTS THE
SET INTERFACE, BACKED BY A
HASH TABLE**

Java Documentation

HASHMAP INTERNAL DATA STRUCTURE REPRESENTATION



HASHSET: CODE EXAMPLE

Code

```
Set<String> cities = new HashSet<>();  
cities.add("Riga"); cities.add("Ogre"); cities.add("Riga");  
  
System.out.println("cities = " + cities);
```

Console output

```
cities = [Riga, Ogre]
```

HASHSET CHARACTERISTICS: RECAP

- ▶ Internally uses `HashMap` to store its elements
- ▶ It **doesn't allow** duplicate values
- ▶ It **allows** single null value
- ▶ It is an **unordered** collection
- ▶ It can store only **non-primitive** values