JAVA

CORE

CyberxNuke
DAY 1

JAVA

CONTENTS

- Basic Intro. To Java
- JDK, JRE, JVM
- Data types
- Variables
- Operators
- Loops (all types of loops)
- Coding Standards

JAVA

INTRODUCTION TO JAVA

- Java is an object oriented programming language (OOP).
- It is write once - use anywhere type of programming language.
- Object Oriented Programming Concepts:
    - Abstraction
    - Encapsulation
    - Polymorphism
    - Inheritance

JDK, JRE, JVM

- **JDK**: Java Development Kit. It comprises of the development tools, the compiler and JRE.
- **JRE**: Java Runtime Environment. It comprises of the library classes and JVM.
- **JVM**: Java Virtual Machine. It is an interpreter and platform dependent. It converts the .class (bytecode) generated by the java compiler to machine language (binary).
- **JIT**: Just In Time Compiler. It compiles the frequently executed code (hot spots) during run time. This leads to substantial performance gains in execution.

VARIABLES & DATA TYPES

- Variable is a container to store data. Every variable is assigned memory according to it’s data type.
- Variable Types:
  - **Static**: A static variable can be accessed without creating the instance of a class. It is allocated memory only once.
  - **Instance**: An instance variable is accessible through an object/instance of a class. It is unique to that object.
  - **Local**: A local variable can be used inside the method where it is declared. It cannot be accessed outside its scope.

VARIABLES & DATA TYPES

- Variable is a container to store data. Every variable is assigned memory according to it’s data type.

- Primitive Data Types. They store the value:

int (4 Bytes)	**double (8 Bytes)**
short (2 Bytes)	**char (2 Bytes)**
long (8 Bytes)	**boolean (1 Bit)**
float (4 Bytes)	**Byte (1 Byte)**

VARIABLES & DATA TYPES

- Variable is a container to store data. Every variable is assigned memory according to it’s data type.
- Non-Primitive Data Types. They don’t store the value but they store the reference (address) to the value:

String
Arrays
Class
Interface

OPERATORS

- Arithmetic Operators
- Relational Operators
- Bitwise Operators
- Logical Operators
- Assignment Operators
- Miscellaneous Operators

OPERATORS

ARITHMETIC OPERATORS

- + (**Addition**)
- - (**Subtraction**)
- * (**Multiplication**)
- / (**Division**)
- % (**Remainder**)
- ++ (**Increment**)
- - - (**Decrement**)

OPERATORS

RELATIONAL OPERATORS

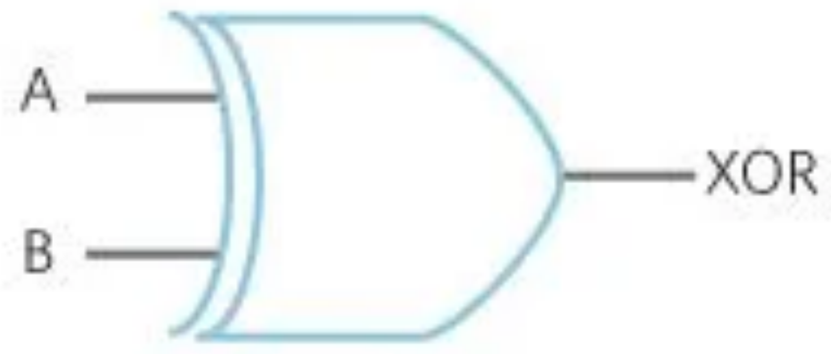
- < (**Less than**)
- > (**Greater Than**)
- <= (**Less than or equal to**)
- >= (**Greater than or equal to**)
- != (**Not equal to**)
- == (**equal to**)

OPERATORS

BITWISE OPERATORS

- & (**bitwise and**)
- | (**bitwise or**)
- ^ (**bitwise xor**)
- ~ (**bitwise compliment**)
- << (**Binary Left Shift**)
- >> (**Binary Right Shift**)
- >>> (**Shift right zero fill - unsigned**)
- >>> will always put a 0 in the left most bit, while >> will put a 1 or a 0 depending on what the sign of it is.

$X = A \oplus B$



A	B	XOR
0	0	0
0	1	1
1	0	1
1	1	0

OPERATORS

LOGICAL OPERATORS

- && (**Logical AND**)
- || (**Logical OR**)
- ! (**Logical NOT**)

OPERATORS

ASSIGNMENT OPERATORS

- = (**Assignment**)
- += (**Short Hand Addition**)
- -= (**Short Hand Subtraction**)
- *= (**Short Hand Multiplication**)
- /= (**Short Hand Division**)
- %= (**Short Hand Remainder**)
- &= (**Bitwise AND assignment**)
- |= (**Bitwise OR assignment**)
- ^= (**Bitwise XOR or exclusive OR assignment**)
- <<= (**Left Shift assignment**)
- >>= (**Right Shift assignment**)

OPERATORS

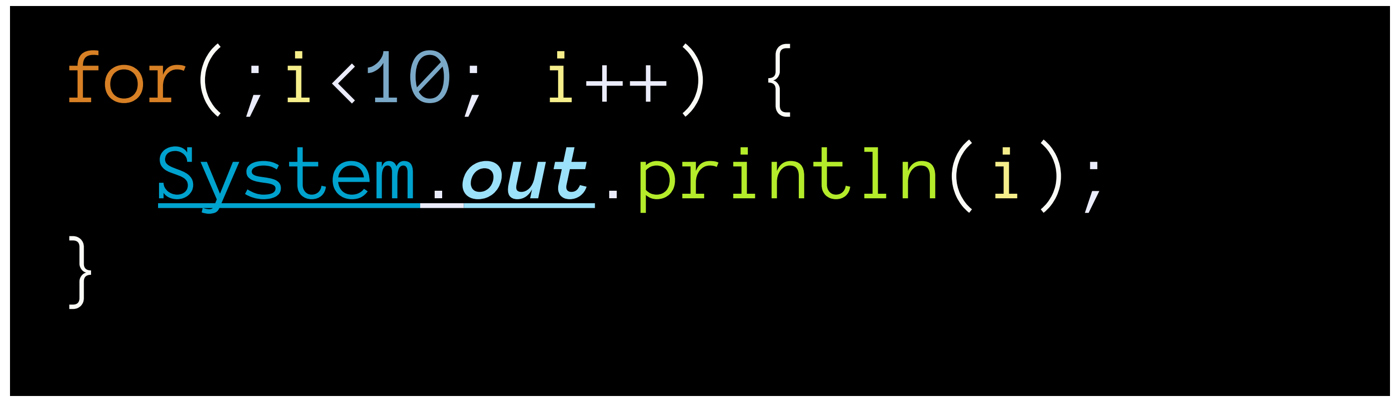
MISCELLANEOUS OPERATORS

- ? (**Conditional Operator** or **Ternary Operator**)
  - Used to evaluate boolean expressions.
  - **Example**
    - (3 > 2) ? True : False

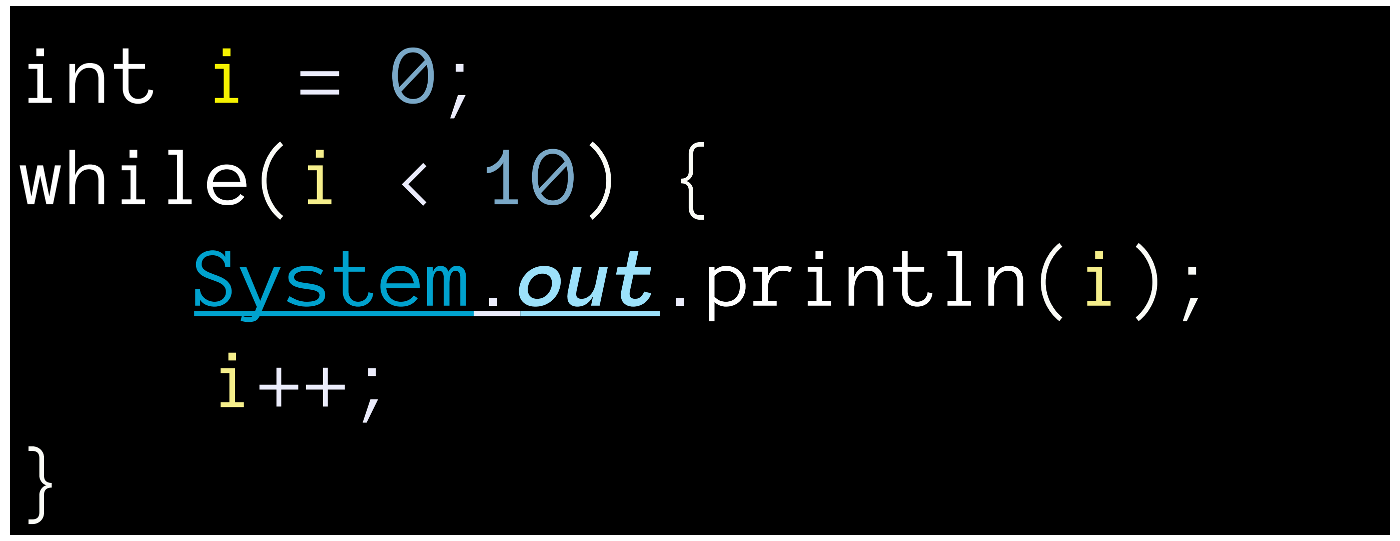
LOOPS

ENTRY CONTROLLED

- An entry controlled loop checks the condition before executing the body of the loop.
- **Example**: for, while



```
for(;i<10; i++) {
  System.out.println(i);
}
```

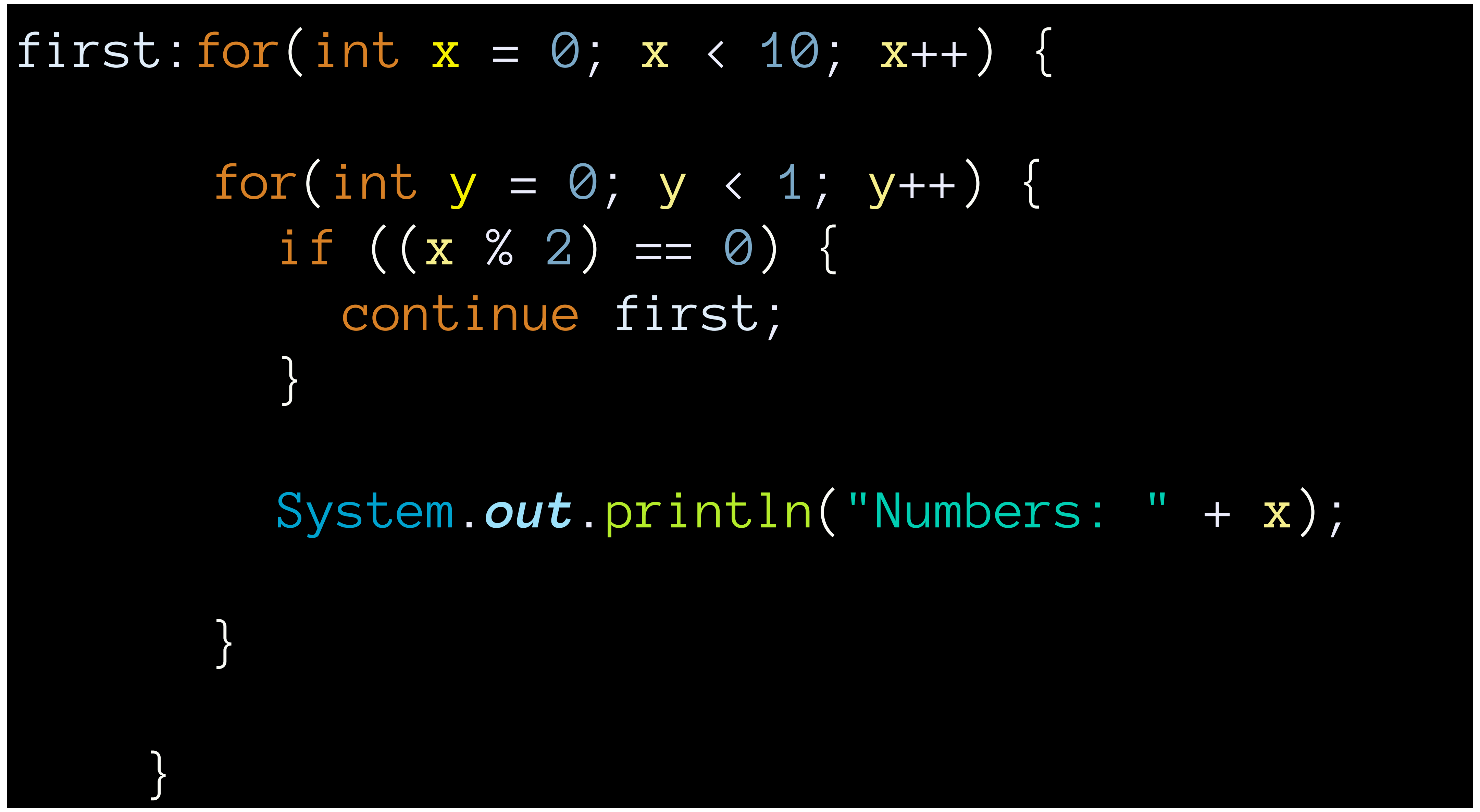


```
int i = 0;
while(i < 10) {
    System.out.println(i);
     i++;
}
```

LOOPS

CONTINUE KEYWORD

- Continue keyword skips the loop and continues with next iteration in the loop.
- **Example:** continue

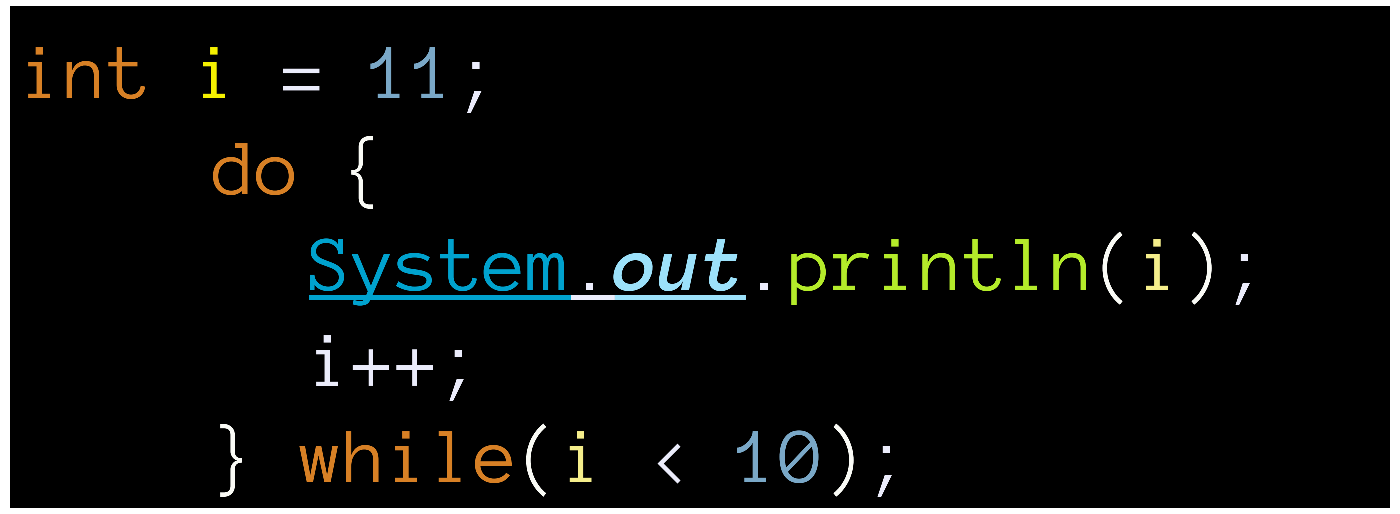


```
first:for(int x = 0; x < 10; x++) {

      for(int y = 0; y < 1; y++) {
        if ((x % 2) == 0) {
          continue first;
        }

        System.out.println("Numbers: " + x);

      }

    }
```

LOOPS

EXIT CONTROLLED

- An exit controlled loop checks the condition after executing the body of the loop. So, it is guaranteed to execute at least once.
- **Example**: do while.



```
int i = 11;
    do {
       System.out.println(i);
       i++;
    } while(i < 10);
```

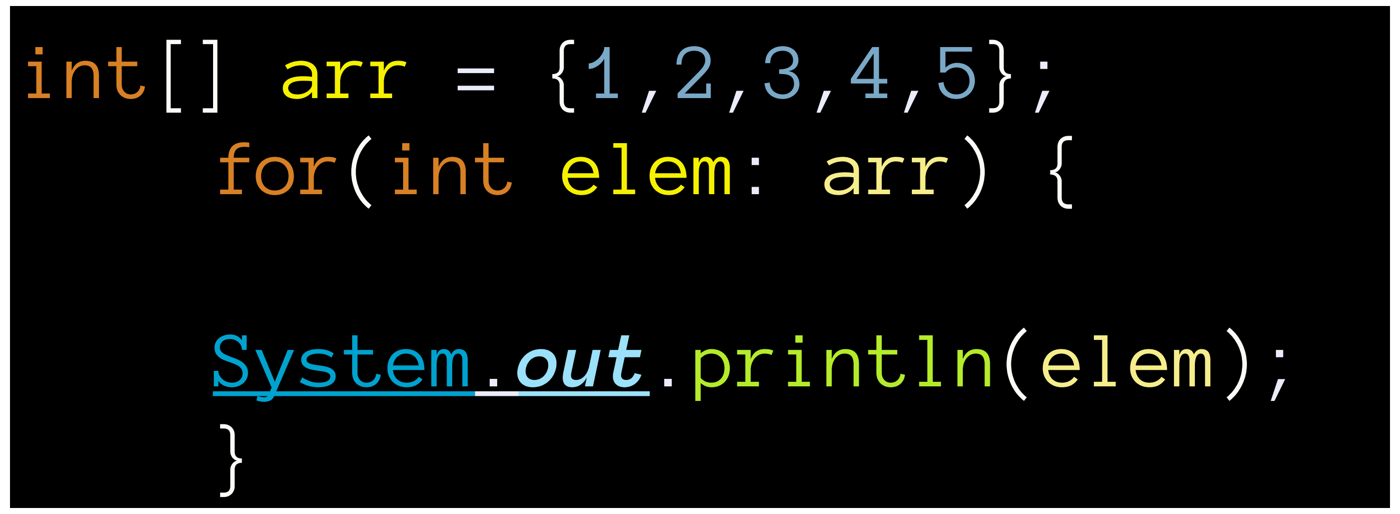
- **Output**: 11

LOOPS

FOR EACH

- For-each loop uses a loop variable to iterate over a collection like array, ArrayList etc.

- **Example**: for-each.



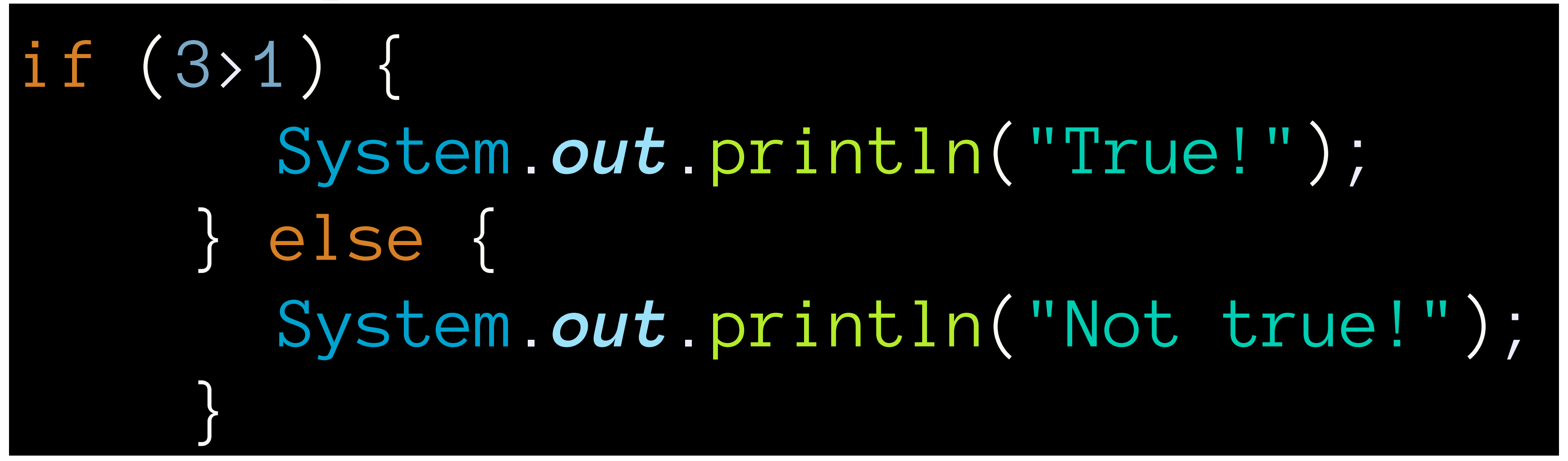
```
int[] arr = {1,2,3,4,5};
    for(int elem: arr) {

    System.out.println(elem);
    }
```

IF-ELSE

CONDITION

- If-else condition is used to perform an action based on the condition. Conditional operators can be used in conjunction with operands as conditions.

- **Example**: if-else.

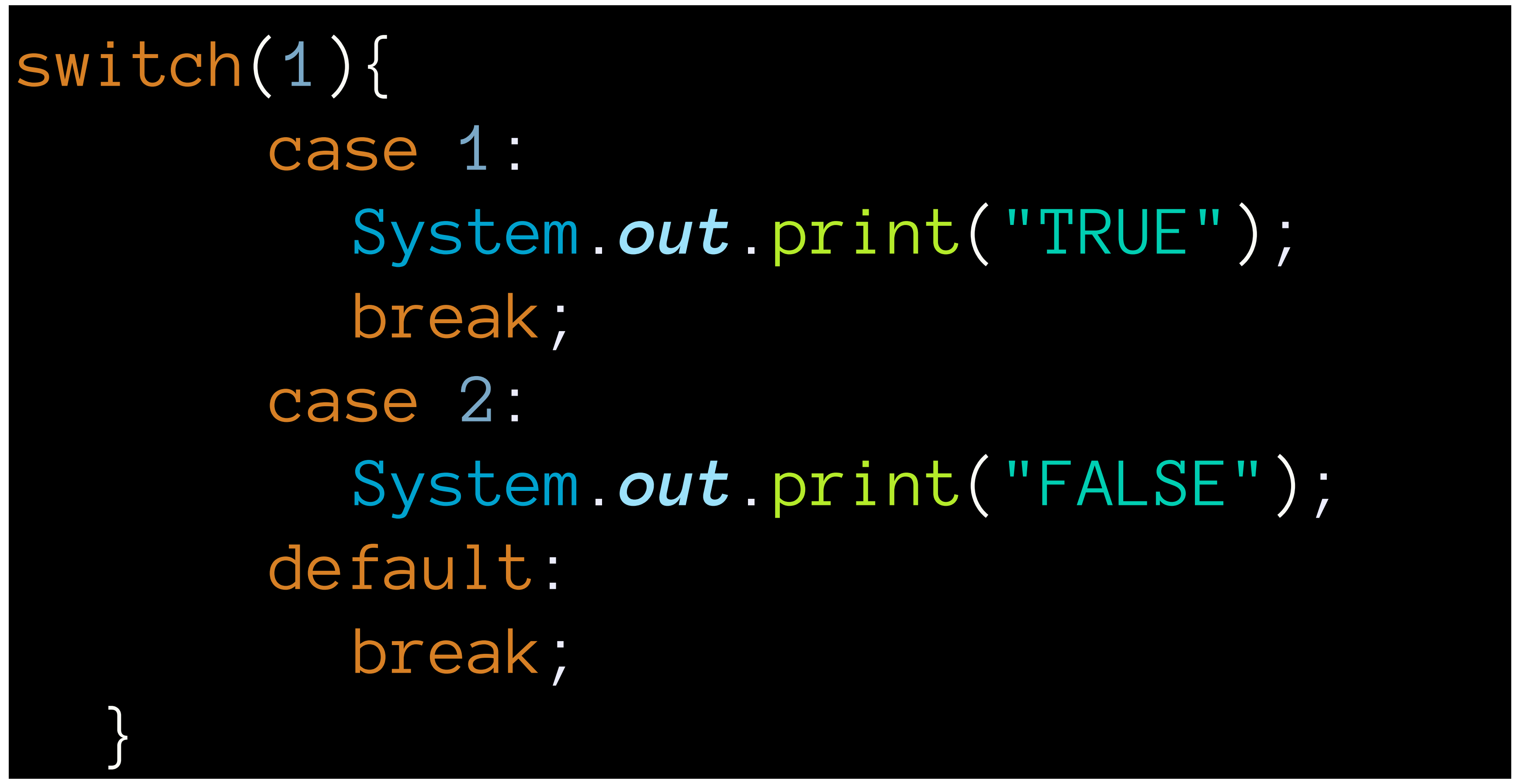


```
if (3>1) {
      System.out.println("True!");
    } else {
      System.out.println("Not true!");
    }
```

SWITCH CASE

CONDITION

- Switch case can be used to perform an action based on the given condition.
- **Example**: switch.



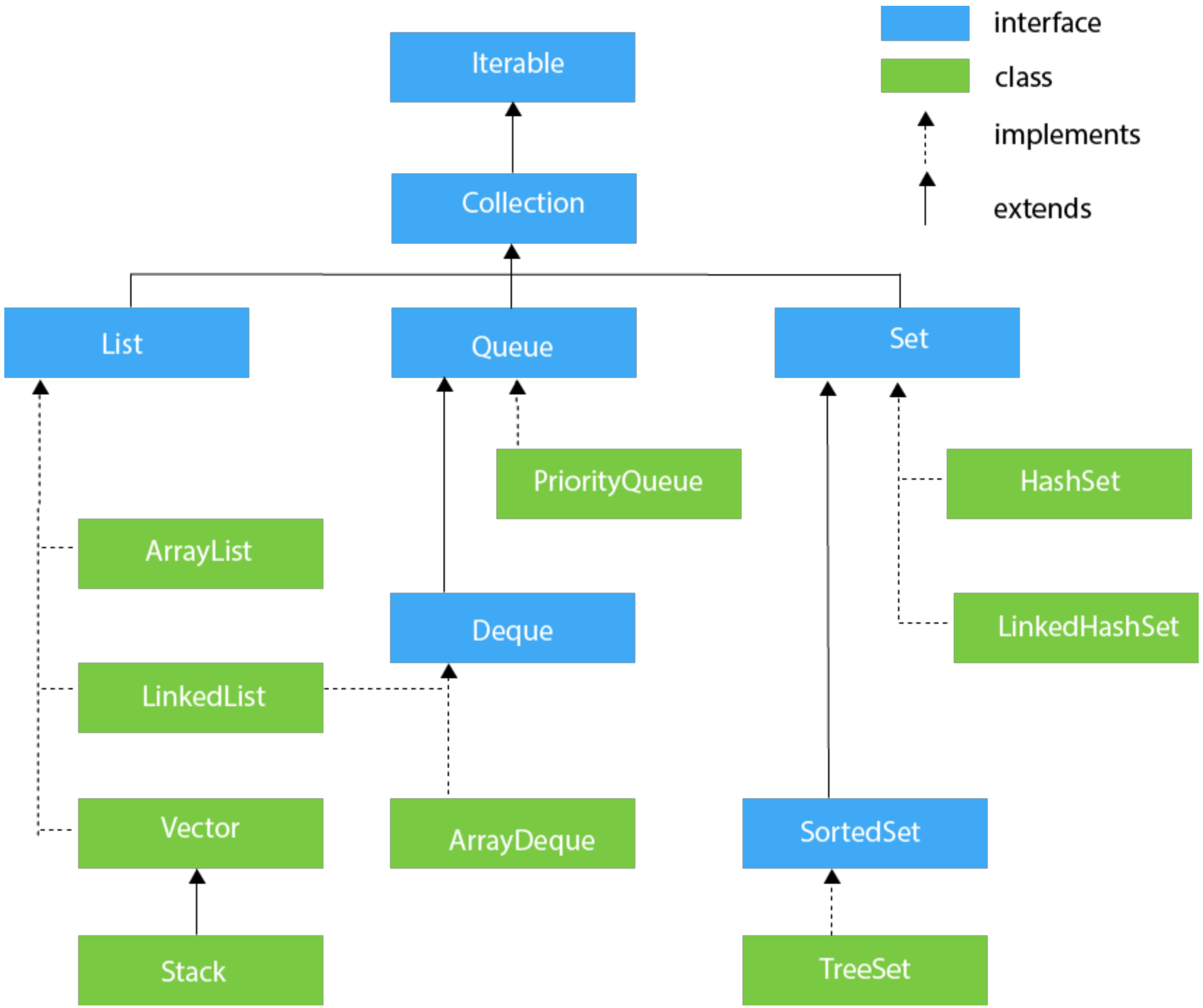
```
switch(1){
      case 1:
        System.out.print("TRUE");
        break;
      case 2:
        System.out.print("FALSE");
      default:
        break;
  }
```

JAVA COLLECTIONS

What is collection framework?

- Collection is a framework that provides an architecture for storing and manipulating objects.
- It represents a single unit of objects.
- It implements the root interface - Iterable.
- It provides both interfaces (List, Queue, Set) and classes (ArrayList, LinkedList etc.).
- It provides methods like Add()
  Size()
  Remove()
  Clear()
  Iterator().

JAVA COLLECTIONS



JAVA COLLECTIONS

Iterator Interface

- Iterator interface provides the facility of iterating the elements in a forward direction only.
- `public boolean hasNext()`

  It returns true if the iterator has more elements otherwise it returns false.
- `public Object next()`

  It returns the element and moves the cursor pointer to the next element.
- `public void remove()`

  It removes the last elements returned by the iterator. It is less used.

```
public interface Iterator<E> {
    /**
     * Returns {@code true} if the iteration has more elements.
     * (In other words, returns {@code true} if {@link #next} would
     * return an element rather than throwing an exception.)
     *
     * @return {@code true} if the iteration has more elements
     */
    boolean hasNext();

    /**
     * Returns the next element in the iteration.
     *
     * @return the next element in the iteration
     * @throws NoSuchElementException if the iteration has no more
elements
     */
    E next();


     …
     …

    default void remove() {
        throw new UnsupportedOperationException("remove");
    }
```

JAVA COLLECTIONS

Iterable Interface

- The Iterable interface is the root interface for all the collection classes.

- The Collection interface extends the Iterable interface and therefore all the subclasses of Collection interface also implement the Iterable interface.

- It contains only one abstract method. i.e.,
- It also provides the implementation for the forEach() method.

  - `Iterator<T> iterator()`

```
public interface Iterable<T> {
    /**
     * Returns an iterator over elements of type
{@code T}.
     *
     * @return an Iterator.
     */
    Iterator<T> iterator();


default void forEach(Consumer<? super T> action) {
    Objects.requireNonNull(action);
    for (T t : this) {
        action.accept(t);
    }
}


}
```

JAVA COLLECTIONS

Collection Interface

```
public interface Collection<E> extends Iterable<E> {
  int size();
  boolean isEmpty();
  boolean contains(Object o);
  boolean add(E e);
  boolean remove(Object o);
  boolean addAll(Collection<? extends E> c);
  boolean retainAll(Collection<?> c);
  void clear();

  …
  …
  …
  }
```

JAVA COLLECTIONS

List Interface

- List interface is the child interface of Collection interface.
- It inhibits a list type data structure in which we can store the ordered collection of objects. It can have duplicate values.
- List interface is implemented by the classes ArrayList, LinkedList, Vector, and Stack.
  - `List numbers = new ArrayList();`

```
public interface List<E> extends Collection<E> {

   int size();
   boolean isEmpty();
   boolean contains(Object o);
   boolean add(E e);
   boolean remove(Object o);
   boolean containsAll(Collection<?> c);
   boolean addAll(Collection<? extends E> c);
   boolean addAll(int index, Collection<? extends E> c);
   boolean removeAll(Collection<?> c);

   // Some extra methods provided by the list interface
   int indexOf(Object o);
   ListIterator<E> listIterator();

   …
   …
   …
}
```

JAVA COLLECTIONS

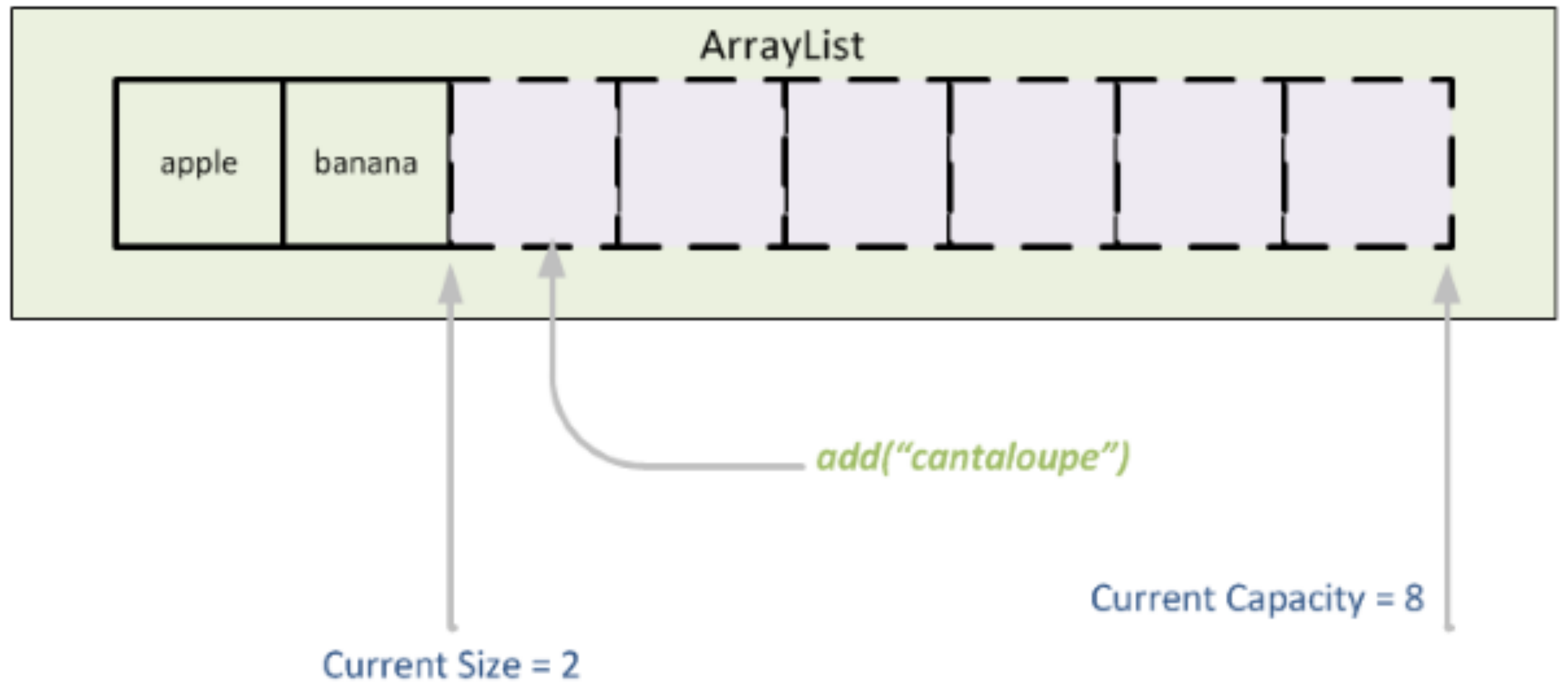
ArrayList

- The ArrayList class implements the List interface.

- It uses a dynamic array to store the duplicate element of different data types.

- The ArrayList class maintains the insertion order and is non-synchronized.

- The elements stored in the ArrayList class can be randomly accessed.

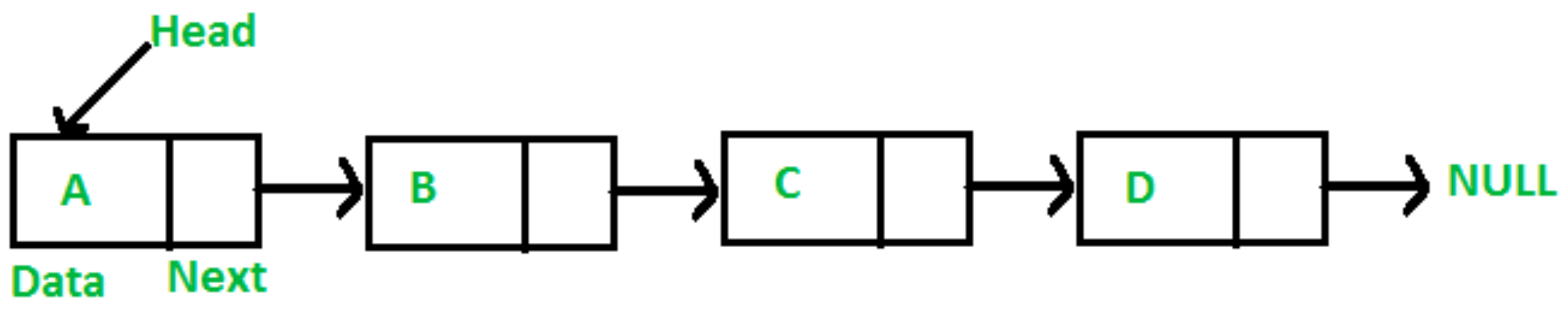


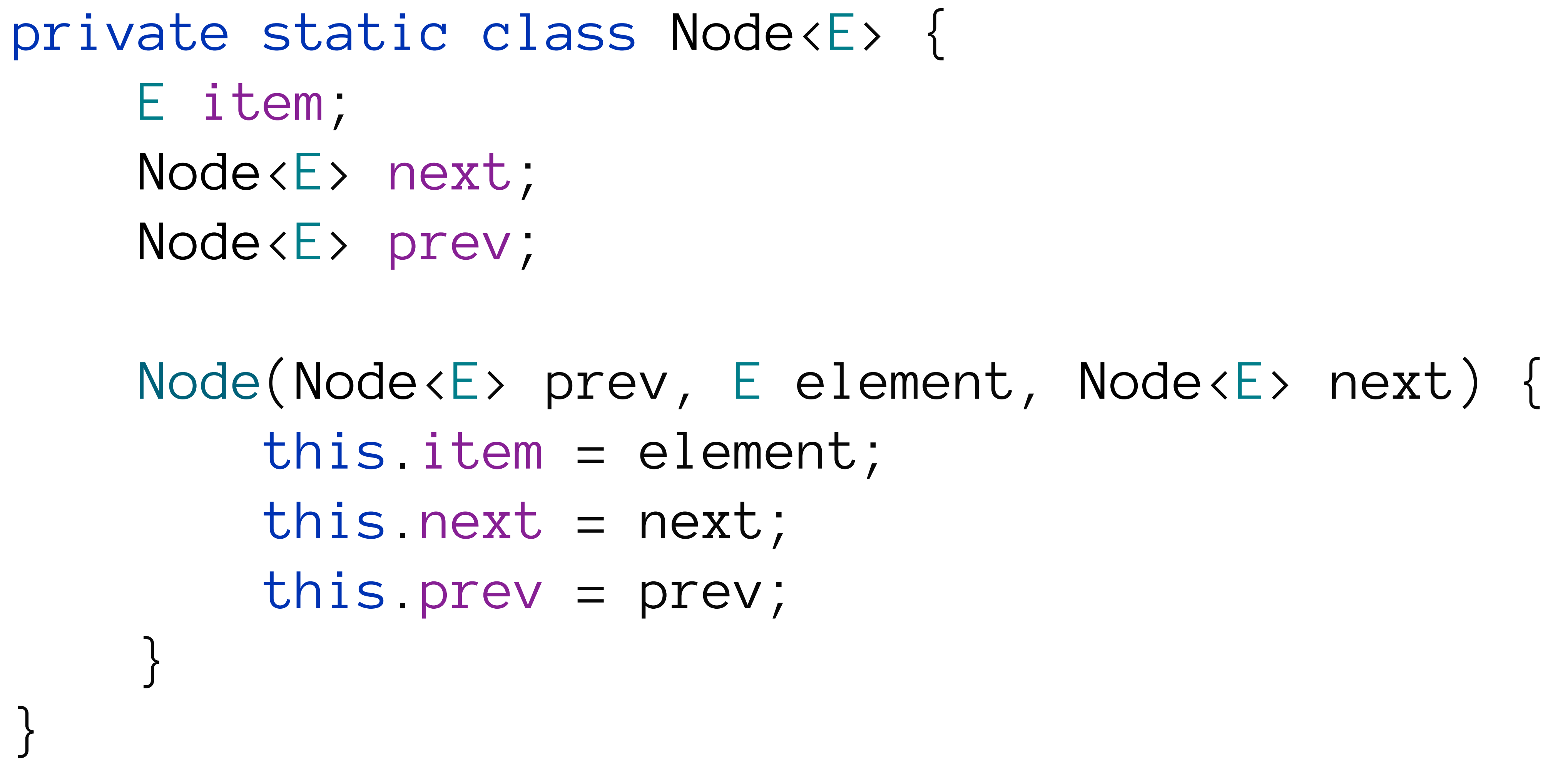
```
public class ArrayList<E> extends AbstractList<E>
        implements List<E>, RandomAccess, Cloneable,
java.io.Serializable
{

    transient Object[] elementData; // non-private to simplify
   nested class access

     @java.io.Serial
    private static final long serialVersionUID = 8683452581122892189L;

     /**
      * Default initial capacity.
      */
     private static final int DEFAULT_CAPACITY = 10;

      …
      …
      …

}
```

Java Collections

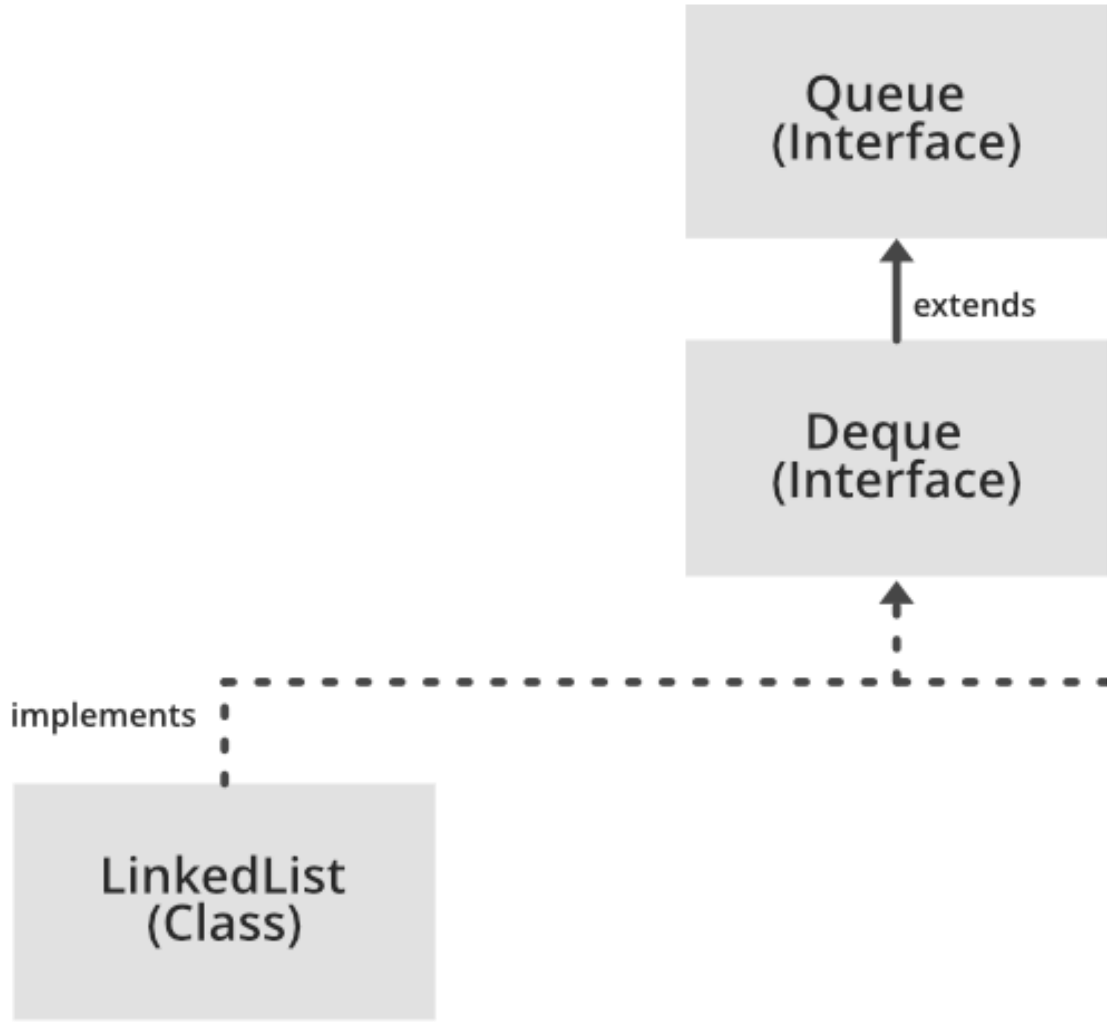
LinkedList

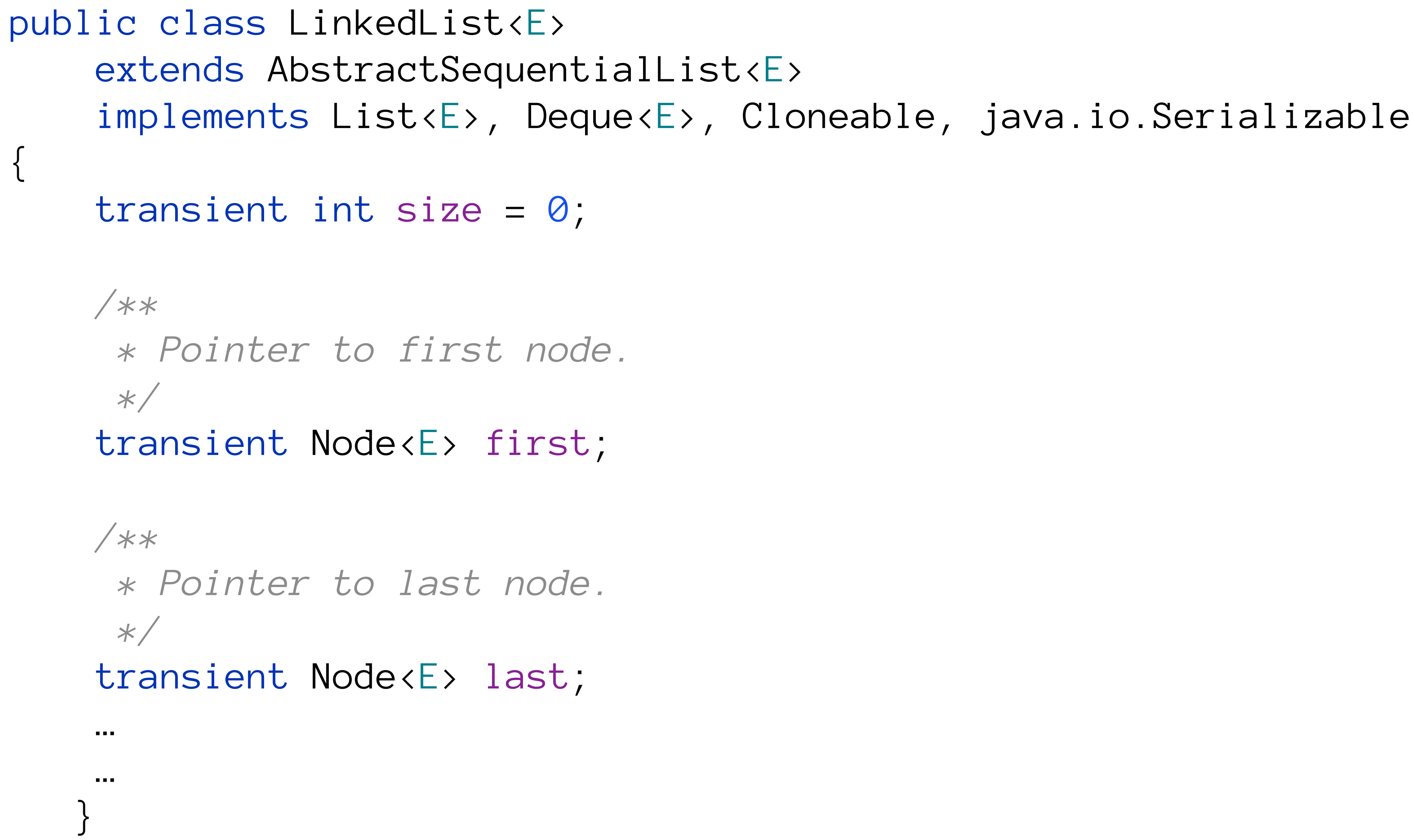
- LinkedList implements the List and the Deque interface.
- It uses a doubly linked list internally to store the elements.
- It can store the duplicate elements.
- It maintains the insertion order and is not synchronized.





```
private static class Node<E> {
    E item;
    Node<E> next;
    Node<E> prev;

    Node(Node<E> prev, E element, Node<E> next) {
        this.item = element;
        this.next = next;
        this.prev = prev;
    }
}
```

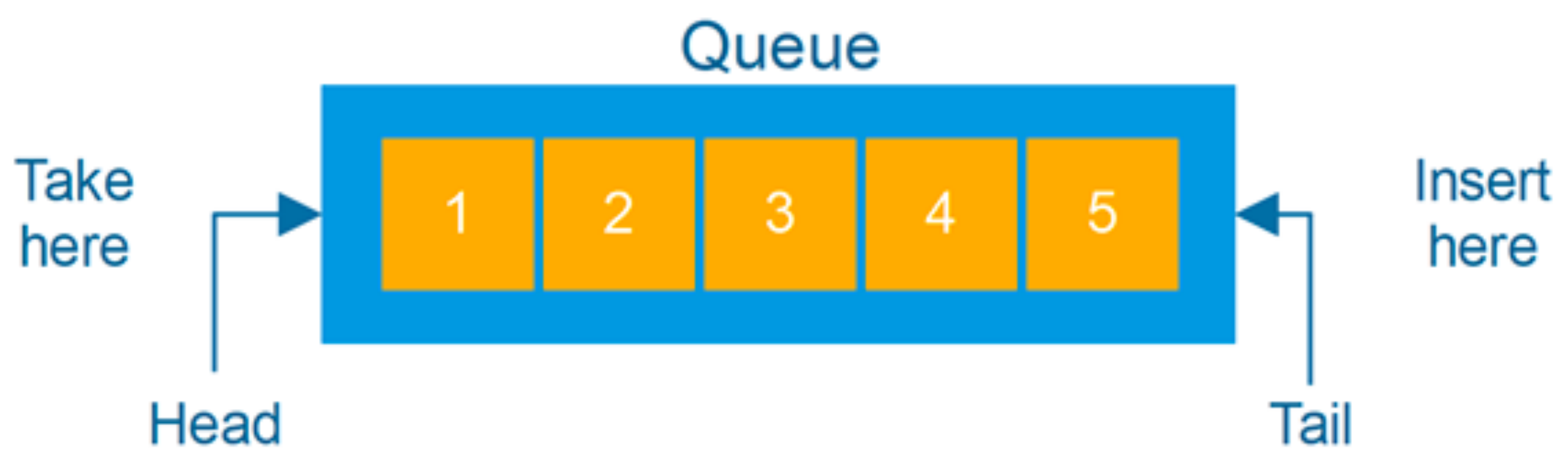




```
public class LinkedList<E>
    extends AbstractSequentialList<E>
    implements List<E>, Deque<E>, Cloneable, java.io.Serializable
{
    transient int size = 0;

    /**
     * Pointer to first node.
     */
    transient Node<E> first;

    /**
     * Pointer to last node.
     */
    transient Node<E> last;
    …
    …
  }
```

Java Collections

Queue



- Queue interface maintains the first-in-first-out order (FIFO).

- It can be defined as an ordered list that is used to hold the elements which are about to be processed.

- There are various classes like PriorityQueue and ArrayDeque which implements the Queue interface.

```
public interface Queue<E> extends Collection<E> {

  boolean add(E e);
  boolean offer(E e);
  E remove();
  E poll();
  E element();
  E peek();

  …
  …
  …

  }
```

Java Collections

PriorityQueue

- The PriorityQueue class implements the Queue interface.

- The elements of the priority queue are ordered according to the natural ordering, or by a Comparator provided at queue construction time, depending on which constructor is used.

- PriorityQueue doesn't allow null values to be stored in the queue.

```
public class PriorityQueue<E> extends AbstractQueue<E>
    implements java.io.Serializable {

    private static final int DEFAULT_INITIAL_CAPACITY = 11;

  public PriorityQueue() {
      this(DEFAULT_INITIAL_CAPACITY, null);
  }


  …
  …
  …

  }
```

Java Collections

Deque Interface

- Deque interface extends the Queue interface.
- In Deque, we can remove and add the elements from both the side.
- Deque stands for a double-ended queue which enables us to perform the operations at both the ends.

```
Deque dq = new ArrayDeque();
```

```
public interface Deque<E> extends Queue<E> {
   void addFirst(E e);
   void addLast(E e);
   boolean offerFirst(E e);
   boolean offerLast(E e);
   E removeFirst();
   E removeLast();
   E pollFirst();
   E pollLast();
   E getFirst();
   E getLast();
   E peekFirst();
   E peekLast();
   boolean removeFirstOccurrence(Object o);
   boolean removeLastOccurrence(Object o);
   boolean add(E e);
   boolean offer(E e);
   E remove();
   E poll();
   E element();
   E peek();
   boolean addAll(Collection<? extends E> c);
   void push(E e);
   E pop();
   boolean remove(Object o);
   boolean contains(Object o);
   int size();
   Iterator<E> iterator();
   Iterator<E> descendingIterator();
}
```

Java Collections

ArrayDeque

- ArrayDeque class implements the Deque interface.
- It facilitates us to use the Deque.
- Unlike queue, we can add or delete the elements from both the ends.
- ArrayDeque is faster than ArrayList and Stack and has no capacity restrictions.

```
public class ArrayDeque<E> extends AbstractCollection<E>
                           implements Deque<E>,
Cloneable, Serializable
{
  transient Object[] elements;
  transient int head;
  transient int tail;

/* Constructs an empty array deque with an initial
capacity sufficient to hold 16 elements.*/

public ArrayDeque() {
    elements = new Object[16 + 1];
}

  …
  …
  …
}
```

Java Collections

Set Interface

- It extends the Collection interface.
- It represents the unordered set of elements which doesn't allow us to store the duplicate items.
- We can store at most one null value in Set.
- Set is implemented by HashSet, LinkedHashSet, and TreeSet.

```
public interface Set<E> extends Collection<E> {
  boolean isEmpty();
  boolean contains(Object o);

  …
  …
  …
}
```

Java Collections

HashSet

- HashSet class implements Set Interface.
- It represents the collection that uses a hash table for storage.
- Hashing is used to store the elements in the HashSet.
- It contains unique items.

```
HashSet<String> set=new HashSet<Stri
ng>();
```

```
public class HashSet<E>
    extends AbstractSet<E>
    implements Set<E>, Cloneable, java.io.Serializable
{
     // The key-set of this map acts as the set.
     private transient HashMap<E,Object> map;

     public HashSet() {
         map = new HashMap<>();
     }


     …
     …
     …

     }
```

Java Collections

LinkedHashSet

- LinkedHashSet class represents the LinkedList implementation of Set Interface.

- It maintains a doubly-linked List across all elements

- It extends the HashSet class and implements Set interface.

- Contains unique elements.
- It maintains the insertion order
- It permits null elements.

```
public class LinkedHashSet<E>
    extends HashSet<E>
    implements Set<E>, Cloneable, java.io.Serializable {

  public LinkedHashSet() {
  // LoadFactor: The amount of capacity which is to be
  // exhausted for the DS to increase its capacity
      super(16, .75f, true);
  }

  …
  …
  …
  }
```

Java Collections

SortedSet Interface

- SortedSet is the alternate of Set interface that provides a total ordering on its elements.

- The elements of the SortedSet are arranged in the increasing (ascending) order.

- The SortedSet provides the additional methods that inhibit the natural ordering of the elements.

```
SortedSet<data-type> set = new TreeSet();
```

```
public class TreeSet<E> extends AbstractSet<E>
    implements NavigableSet<E>, Cloneable, java.io.Serializable
{

  private transient NavigableMap<E,Object> m;

  public TreeSet() {
      this(new TreeMap<>());
  }

  public TreeSet(Comparator<? super E> comparator) {
      this(new TreeMap<>(comparator));
}
```

ORDERING IN A TREESET

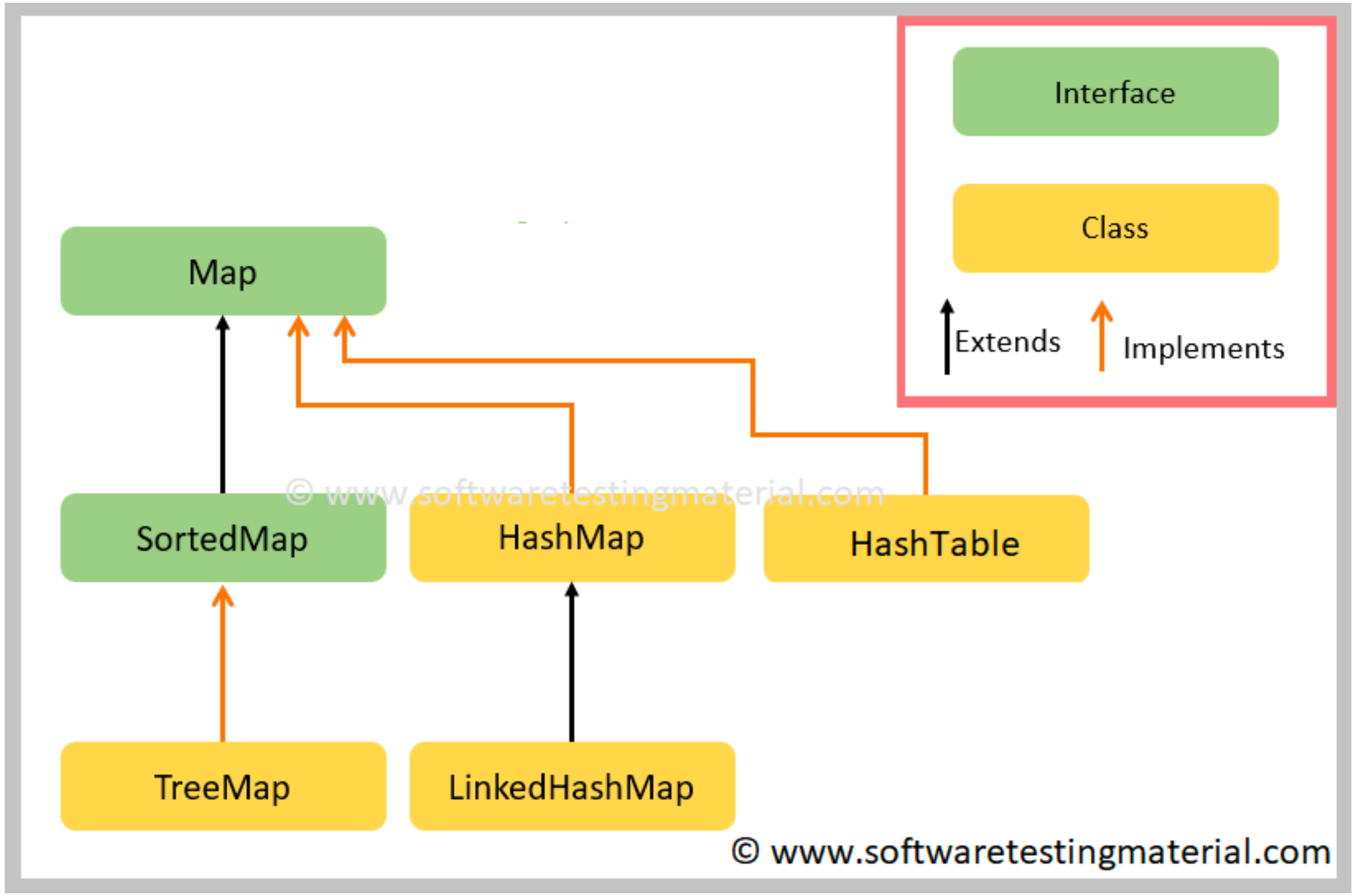
Numbers

Capitals

Small alphabets

Java Collections

Map Interface



Java Collections

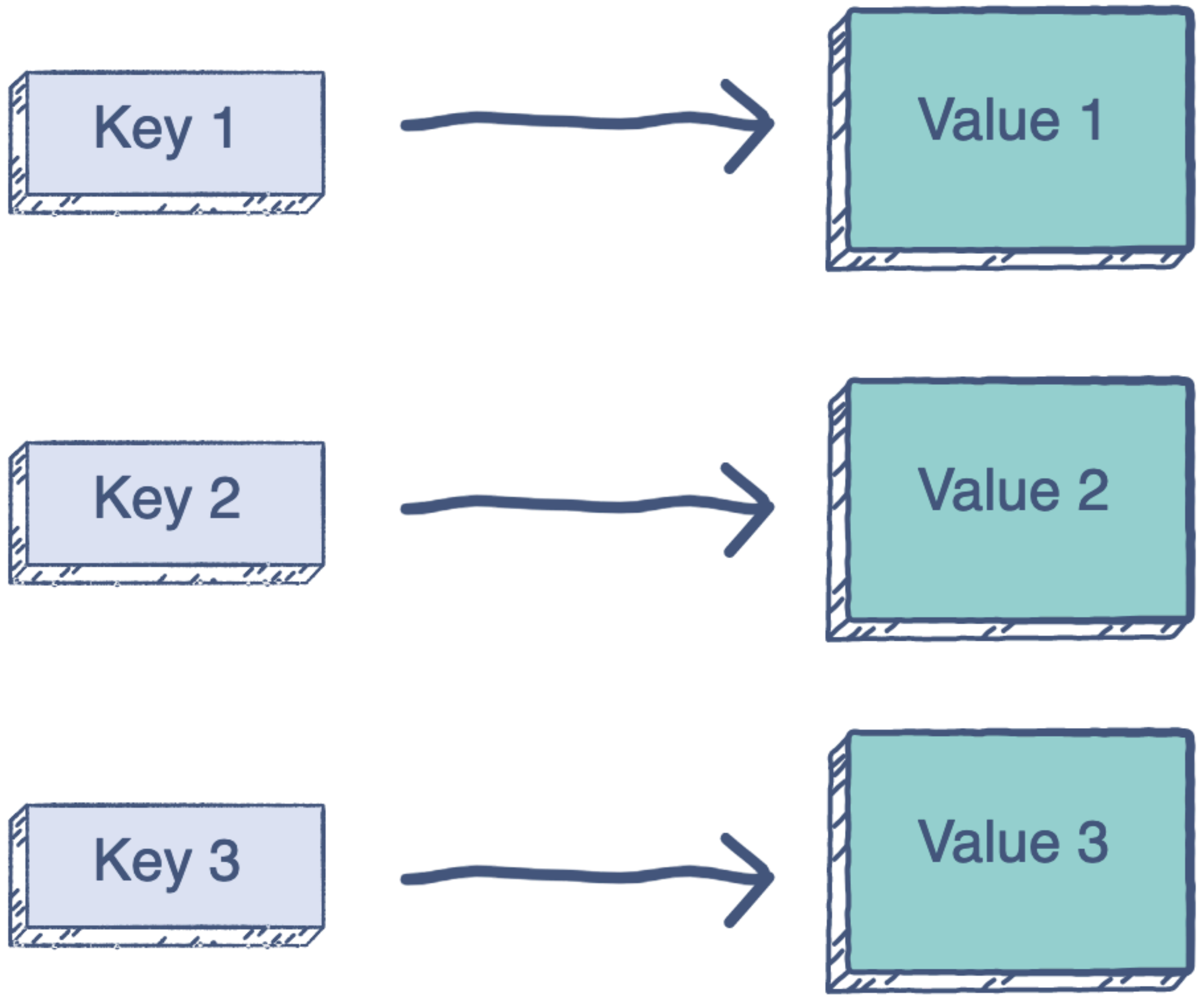
HashMap

- Java HashMap class implements the map interface by using a hash table.

- **HashMap** stores items as key/value pairs. Values can be accessed by indexes, known as keys, of a user-defined type.

- This class makes no guarantees as to the order of the map; in particular, it does not guarantee that the order will remain constant over time.

```
HashMap<Integer, String> set=new HashMap<Integer,
String>();
```



Java Collections

HashTable

- Hashtable was part of the original java.util and is a concrete implementation of a Dictionary.

- It is similar to HashMap, but is synchronized.

- Null values are not allowed.

- Hashtable stores key/value pairs in a hash table.

- The key is hashed, and the resulting hash code is used as the index at which the value is stored within the table.

```
public class Hashtable<K,V>
    extends Dictionary<K,V>
    implements Map<K,V>, Cloneable, java.io.Serializable {

/* To successfully store and retrieve objects from a Hashtable,
the objects used as -

keys must implement the hashCode method and the equals method. */

  public Hashtable() {
      this(11, 0.75f);
  }


  public Hashtable(int initialCapacity) {
      this(initialCapacity, 0.75f);
  }



  …
  …
  …
}
```

```
HashTable<Integer, String> hashTable=new HashTable<Integer, String>();
```

Java Collections

LinkedHashMap

- The **LinkedHashMap** is just like HashMap with an additional feature of maintaining an order of elements inserted into it.

- **Important Features of a LinkedHashMap:**
  - A LinkedHashMap contains values based on the key. It implements the Map interface and extends the HashMap class.
  - It contains only unique elements.
  - It may have one null key and multiple null values.
  - It is non-synchronized.

```
LinkedHashMap<Integer, String> set=new LinkedHashMap<Integer, String>();
```

JAVA 8

LAMDA EXPRESSIONS & FUNCTIONAL INTERFACES

- It is the first step into functional programming.

- It is a function which can be created without belonging to any class.

- A Java lambda expression can be passed around as if it was an object and executed on demand.

- Syntax of Lamda Expression:

```
(parameter_list) -> {function body}
```

- An interface with **only single abstract method** is called functional interface (or Single Abstract method interface).
    - Consumer Interface
    - Or define your own functional interface

```
StringFunction exclaim = (s) -> s + "!";
StringFunction end = (s) -> {
    return s + " XYZ";
};
```

```
@FunctionalInterface
interface StringFunction {
    String run(String x);
}
```

CODING STANDARDS

- Class and interface names should be in Camel Case. Avoid acronyms/ abbreviations.

- Use meaningful variable names.

- Don’t declare or execute multiple statements in the same line.

- Use getters, setters (**getX()**, **setX()**) to assign values to the variables. Set the access modifier of the variables to private.

Reference https://medium.com/@rhamedy/a-short-summary-of-java-coding-best-practices-31283d0167d3