# 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 Byte)** |
| **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

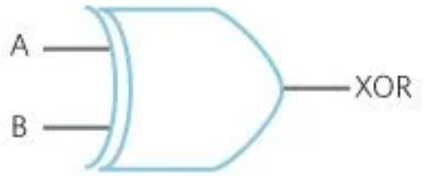
- < (**Less than**)
- > (**Greater Than**)
- <= (**Less than or equal to**)
- >= (**Greater than or equal to**)
- != (**Not equal to**)
- == (**equal to**)

**CyberxNuke**

# 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

## 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.

# JAVA COLLECTIONS

## Iterable Interface

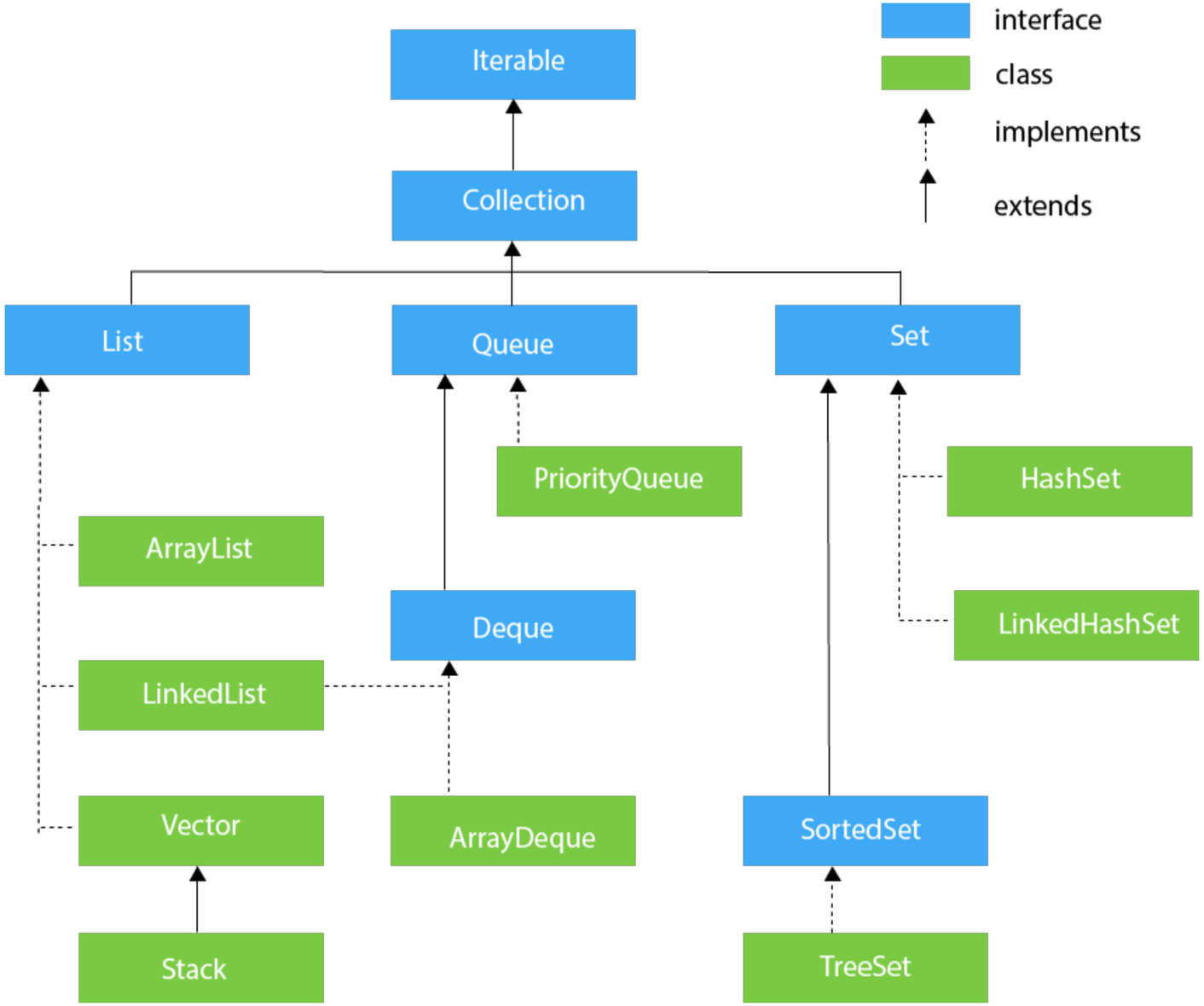
- The Iterable interface is the root interface for all the collection classes.
- The Collection interface extends the Iterable interface and therefore all the subclasses of Collection interface also implement the Iterable interface.
- It contains only one abstract method. i.e.,

```
Iterator<T> iterator()
```

# JAVA COLLECTIONS

- 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

## 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.

# 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.

# Java Collections

## LinkedList

- LinkedList implements the Collection 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.

# Java Collections

## Vector

- Vector uses a dynamic array to store the data elements.
- It is similar to ArrayList.
- It is synchronized and contains many methods that are not the part of Collection framework.

# Java Collections

## Stack

- The stack is the subclass of Vector.
- It implements the last-in-first-out data structure (LIFO).
- The stack contains all of the methods of Vector class and also provides its methods like
    - boolean push(),
    - boolean peek(),
    - boolean push(object o), which defines its properties.

# 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, Deque, and ArrayDeque which implements the Queue interface.

# Java Collections

## PriorityQueue

- The PriorityQueue class implements the Queue interface.
- It holds the elements or objects which are to be processed by their priorities.
- PriorityQueue doesn't allow null values to be stored in the queue.

# 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();
```

# 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.

# 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.

# 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<String>();
```

# Java Collections

## LinkedHashSet

- LinkedHashSet class represents the LinkedList implementation of Set Interface.
- It extends the HashSet class and implements Set interface.
- Like HashSet, It also contains unique elements. It maintains the insertion order and permits null elements.

# 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();
```

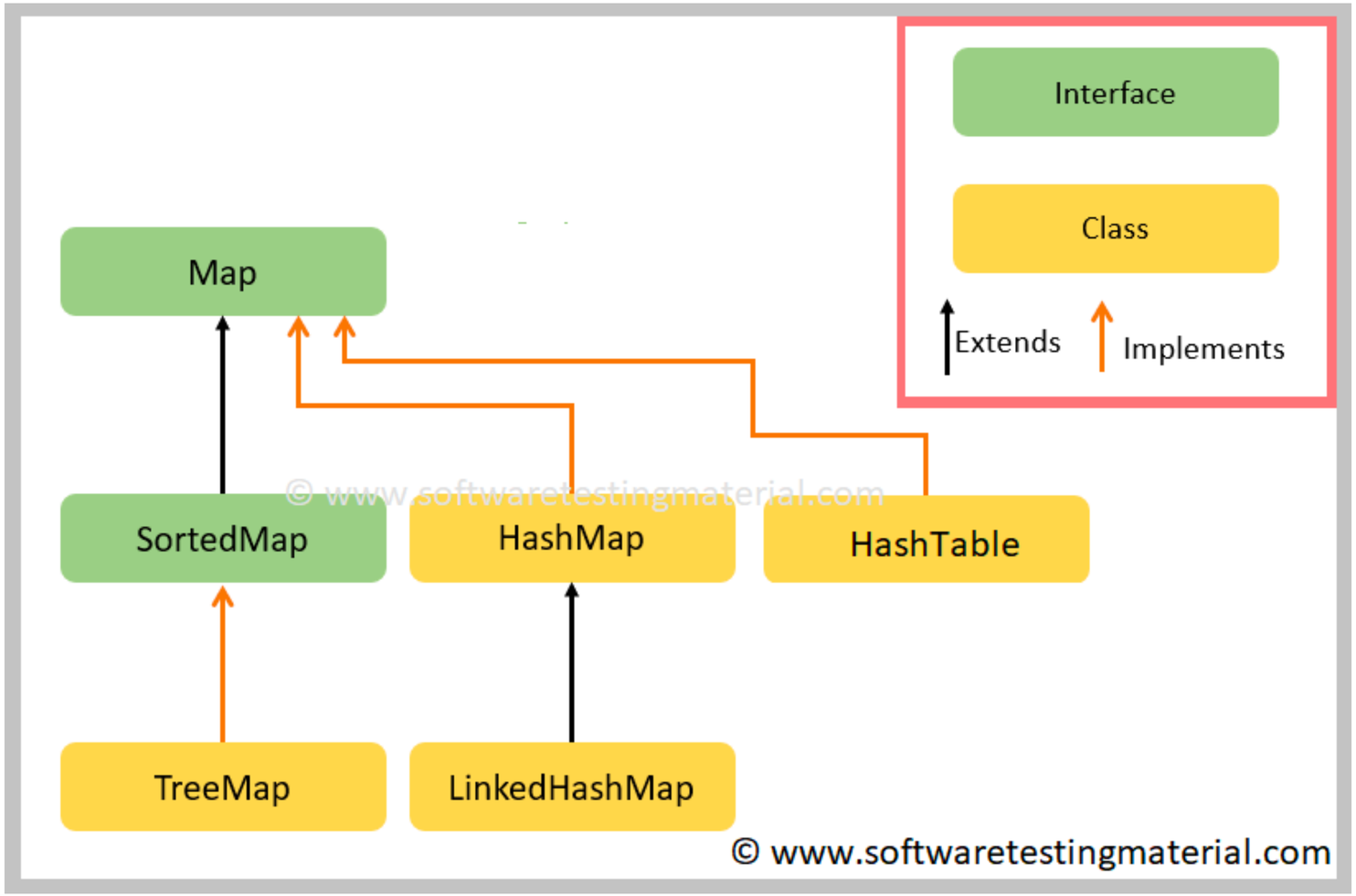
# Java Collections

## TreeSet

- TreeSet class implements the Set interface that uses a tree for storage.
- Like HashSet, TreeSet also contains unique elements.
- However, the access and retrieval time of TreeSet is quite fast.
- The elements in TreeSet stored in ascending order.

# Java Collections

## Map Interface



# Java Collections

## HashMap

- HashMap class implements Map Interface.
- 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>();
```

# 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