Java Core APIs

Lesson Outlines

- Exceptions – specifics and handling
- Files – standard operations
- Functional Interfaces and Lambda Expressions
- Generics
- Collections
  - Basic Data Structures Implemented
  - Basic Interfaces and Implementations
  - Autoboxing and Autounboxing, Primitive Wrappers
  - Streams API

Exceptions

Exceptions

- Standard try-catch-finally block
- Try with resources (for autoclosable objects)
- Exception methods
  - message, stackTrace, cause

```java
try{
    //...
}
catch (Exception e){
    System.out.println(e.getMessage());
}
finally {
    //...
}
```

```java
try(Scanner sc = new Scanner(System.in)){
    int input = sc.nextInt();
}
catch (Exception e){
    //..
}
//no need for sc.close() in finally block
```

Exceptions

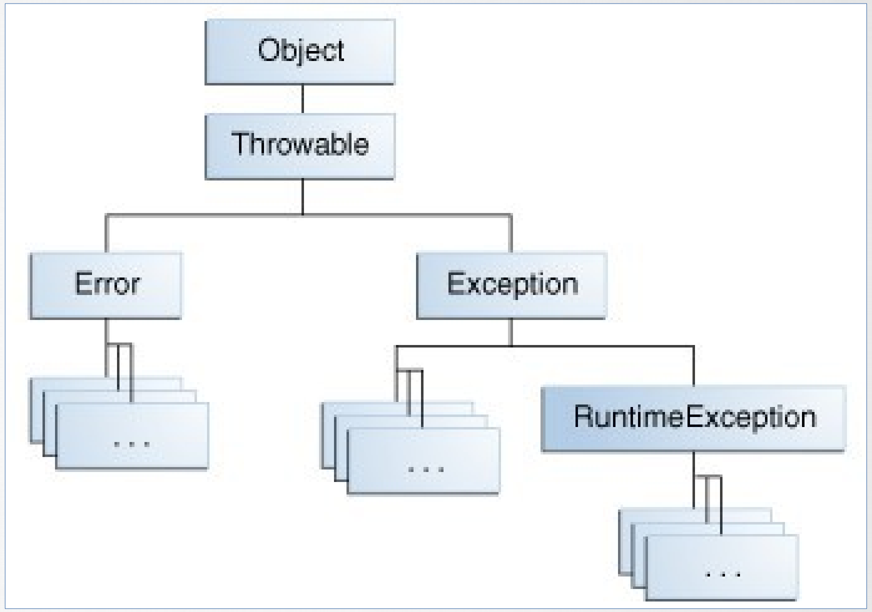
- Chained Exceptions (throwing Ex2 in a catch of Ex1)
- Wrapped Exceptions (cause)
- Manually throwing and Exception

```java
try {
    File f = new File( pathname: "krasi.txt");
    f.createNewFile();
}
catch (IOException e){
    throw new SecurityException("Security goes brrr", e);
}
```

```
Exception in thread "main" java.lang.SecurityException: Security goes brrr
    at exceptions.Demo.main(Demo.java:28)
Caused by: java.io.IOException
    at exceptions.Demo.main(Demo.java:25)
```

Exceptions

- Exceptions Hierarchy - Throwable
  - **Errors**
    - StackOverflow, OutOfMemoryError
  - **Checked Exceptions (extend Exception)**
    - Anticipated, out of our control
    - Catch-or-specify requirement
    - IOException, SQLException, etc
  - **Unchecked Exceptions (extend RuntimeException)**
    - Logic errors or improper use of an API
    - NullPointerException, IndexOutOfBoundsException, ClassCastException, etc



Exceptions

- Defining Own Exceptions
  - **Just create a class that extends Exception**
  - Override any constructor you want to use

```
public class UnauthorizedException extends Exception{

    public UnauthorizedException(String msg){
        super(msg);
    }

    public UnauthorizedException(String msg, Throwable cause){
        super(msg, cause);
    }
}
```

Files

Files

- In Java all files (including directories) are objects
  - **Part of the I/O API**
  - java.io.File – constructed with paths, has many utility methods

```
File file1 = new File( pathname: "krasi.txt");//relative path, in project dir.
File file2 = new File( parent: "..",  child: "test.txt");//relative path, in parent dir
File file3 = new File( pathname: "D:\\Nexo\\file.txt");//absolute path

boolean exists = file1.exists();
boolean isFile = file1.isFile();
if(file3.isDirectory()){
    File[] children = file3.listFiles();
}
file2.delete();
```

Files

- Reading and Writing operations - **Use File Streams**
  - InputStream – abstract class for reading
  - OutputStream – abstract class for writing
- Streams must be closed when not needed anymore
- Streams are unidirectional and read data only once
- EOF is read as byte value of -1

Files

- Stream types
  - Byte Streams – FileInputStream, FileOutputStream – work with byte[]

```
//if the file does not exist, the stream creates it
try(FileOutputStream fos = new FileOutputStream(new File( pathname: "krasi.txt"))){
    fos.write( b: '!');//works with bytes
}//auto closable after this block
```

  - Character Streams – FileReader, FileWriter – work with String

```
try(FileWriter writer = new FileWriter(new File( pathname: "krasi.txt"),  append: true);) {
    writer.write( str: "kak e?");
    writer.flush();
}
```

  - Object Streams – for serialization, pretty old and obsolate
  - Scanner and PrintStream – high level objects, commonly used. Have friendly methods.

Files

- Files utility class (java.nio package)
  - Many utility methods for easier file manipulation

```java
Path path = Path.of( first: "nexo.txt");
//create
if(!Files.exists(path)) {
    Files.createFile(path);//Files.createDirectory(path);
    //write
    Files.writeString(path,  csq: "Как е хавата?");
}
Files.writeString(path,  csq: "Всичко наред ли е?", StandardOpenOption.APPEND);
//read
String text = Files.readString(path);
System.out.println(text);
//delete
Files.delete(path);
```

```java
Files.isReadable(path);
Files.isWritable(path);
Files.isExecutable(path);
```

```java
Files.readAllLines(Path.of( first: "copy.txt")).forEach(s -> System.out.println(s));
```

Files

- Copy Files, Move Files

```
Path original = Path.of( first: "nexo.txt");
if(Files.notExists(original)) {
    Files.createFile(original);
    Files.writeString(original,  csq: "some original text");
}
Path copy = Path.of( first: "copy.txt");
Files.copy(original, copy);

Path dir = Path.of( first: "myFolder");
Files.createDirectory(dir);
Files.move(original, Path.of(
        first: dir.toAbsolutePath()+File.separator+original.getFileName()),
        StandardCopyOption.REPLACE_EXISTING);
```

Functional Interfaces

Lambda Expressions

Functional Interfaces

- Introduced as a term in Java 8
- Must contain only a single abstract (unimplemented) method
- Can contain default and static methods

```java
public interface IRobot {
    void shoot();
}
```

- Since it has only one method, we can implement this using Java Lambda Expression

```java
public static void main(String[] args) {
    practiceShooting(() -> System.out.println("Pew Pew"));
}

public static void practiceShooting(IRobot shooter){
    System.out.println("A wild robot appeared");
    shooter.shoot();
}
```

Lambda Expressions

- First Step into functional programming in Java
- It is a function that can be created without belonging to any class
- Can be passed around and executed on demand

- **Actually it is an instance of an anonymous class that implements a functional interface**

- Used to implement simple event listeners / callbacks
- Used in functional programming with the **Java Stream API**

Lambda Expressions

- Main Advantage – Concise and Expressive
- Old:

```java
button.addActionListener(
   new ActionListener() {
      @Override
      public void actionPerformed(ActionEvent e) {
         doSomethingWith(e);
      }
});
```

New:

```java
button.addActionListener(e -> doSomethingWith(e));
```

- Main Syntax is **(parameters)** → **{body}**

- The Compiler uses the context of the expression to determine the types of parameters.

Lambda Expressions

- You write what looks like a function

```
Arrays.sort(testStrings, (s1, s2) -> s1.length() - s2.length());
taskList.execute(() -> downloadSomeFile());
someButton.addActionListener(event -> handleButtonClick());
double d = MathUtils.integrate(x -> x*x, 0, 100, 1000);
```

- You get an instance of a class that implements the interface that was expected in that place

- Replace this:

```
new SomeInterface() {
    @Override
    public SomeType someMethod(args) { body }
}
```

with this:

```
(args) -> { body }
```

Lambda Expressions

- Type Inferencing
  - Basic Lambda:
    - (Type1 var1, Type2 var2) → {method body}
  - Lambda with type inferencing
    - (var1, var2) → {method body}
- Implied Return Values and omitted parens for single param
  - Basic Lambda:
    - (var1) → {return (something);}
  - Lambda with expression for body
    - var1 → something;

Lambda Expressions

- Common usa cases - Listeners

```
button1.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        setBackground(Color.BLUE);
        }
    });
button2.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        setBackground(Color.GREEN);
        }
    });
button3.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        setBackground(Color.RED);
        }
    });
```

```
button1.addActionListener(event -> setBackground (Color.BLUE));
button1.addActionListener(event -> setBackground (Color.GREEN));
button1.addActionListener(event -> setBackground (Color.RED));
```

Lambda Expressions

- Common usa cases - Sort Collections

```
ArrayList<User> users = new ArrayList<User>();
//fill the collection ...
//sort by age ascending
users.sort(((u1, u2) -> u1.getAge() - u2.getAge()));
//sort by name ascending
users.sort(((u1, u2) -> u1.getName().compareTo(u2.getName())));
```

Built-in Functional Interfaces

- Predicate<T>
- Supplier<T>
- Function<T,R>
- Consumer<T>

```
//Function example
Function<Integer, Integer> tripple = x -> x*3;
System.out.println(tripple.apply( t: 5));//prints 15
//Predicate example
Predicate<Integer> greaterThanOne = i -> (i > 1);
Predicate<Integer> lesserThanTen = i -> (i < 10);
System.out.println(greaterThanOne.test( t: 10));//true
System.out.println(lesserThanTen.test( t: 20));//false
System.out.println(greaterThanOne.and(lesserThanTen).test( t: 5));//true
//Supplier example
Supplier<Double> randomValue = () -> Math.random();
System.out.println(randomValue.get());//random number
//Consumer example
Consumer<Integer> displayPretty = a -> System.out.println("Value = " + a);
displayPretty.accept( t: 10);//Value = 10
```

Method References

- Used as a shorter and more readable alternative for a lambda expression which only calls an existing method
  - Reference to a Static Method

```java
public class Demo1 {

    public static void main(String[] args) {
        //old
        Function<Double, Double> calcPrice1 = rawPrice -> Demo1.applyVAT(rawPrice);
        //new
        Function<Double, Double> calcPrice2 = Demo1::applyVAT;
        System.out.println(calcPrice1.apply( t: 100.00));//120.0
        System.out.println(calcPrice2.apply( t: 399.99));//479.9
    }

    public static Double applyVAT(Double price){
        return price*1.2;//add 20%, can be extracted as constant
    }
}
```

Method References

- Used as a shorter and more readable alternative for a lambda expression which only calls an existing method
  - Reference to an Instance Method

```
ArrayList<User> users = new ArrayList<User>();
//fill the collection ...
//sort by age ascending old
users.sort(((u1, u2) -> u1.getAge() - u2.getAge()));
//sort by age ascending new
users.sort(Comparator.comparingInt(User::getAge));
//sort by name descending old
users.sort(((u1, u2) -> u2.getName().compareTo(u1.getName())));
//sort by name descending new
users.sort((Comparator.comparing(User::getName).reversed()));
```

Generics

Definition and general usage

- Add a way to specify concrete types to general purpose classes and methods that operated on Object before
- Most commonly used on Collections

```java
List list = new ArrayList();
list.add(5);//accepts Object
System.out.println(list.get(0) + 6);//list.get(0) returns Object
```

```java
List<Integer> list = new ArrayList();
list.add(5);//accepts Integer only
System.out.println(list.get(0) + 6);//list.get(0) returns Integer
```

- Aim to reduce bugs and improve readability and reliability of code when we use such container classes
- Made the ‘for-each’ loop possible

```java
List<Integer> list = new ArrayList<>();
//uses generics, otherwise would assume Object
for(Integer i : list){
    System.out.println(i*2);//no need to check or cast
}
```

- Avoid annoying type casting when getting an element from the collection

Definition and general usage

- Generics is not restricted to the predefined classes in the Java API's. It is possible to generify your own Java classes using **className<T>**
- The **<T>** is a type token that signals that this class can have a type set when instantiated.
- If no type is specified when instancing the class, **Object** is taken by default

```java
public class Cage<T> {
    private T prisoner;
    Cage() {}
    Cage(T prisoner){
        this.prisoner = prisoner;
    }
    public T getPrisoner(){
        return prisoner;
    }
}
```

```java
Cage cage = new Cage();
cage.get
  getPrisoner()                Object
```

Definition and general usage

- If you do specify a type, then T stands for the type specified for this instance only!

```java
Cage cage = new Cage();

Cage<Bird> birdCage = new Cage<>(new Bird());
Bird prisoner1 = birdCage.getPrisoner();

Cage<Monkey> monkeyCage = new Cage<>(new Monkey());
Monkey prisoner2 = monkeyCage.getPrisoner();
```

m getPrisoner() Monkey

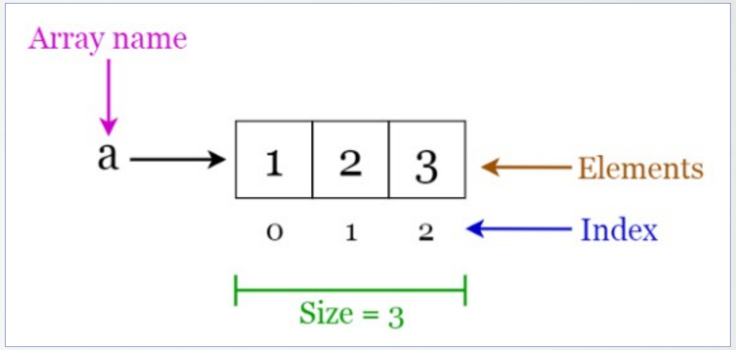
Use Tab to overwrite the current identifier with the chosen variant Next Tip

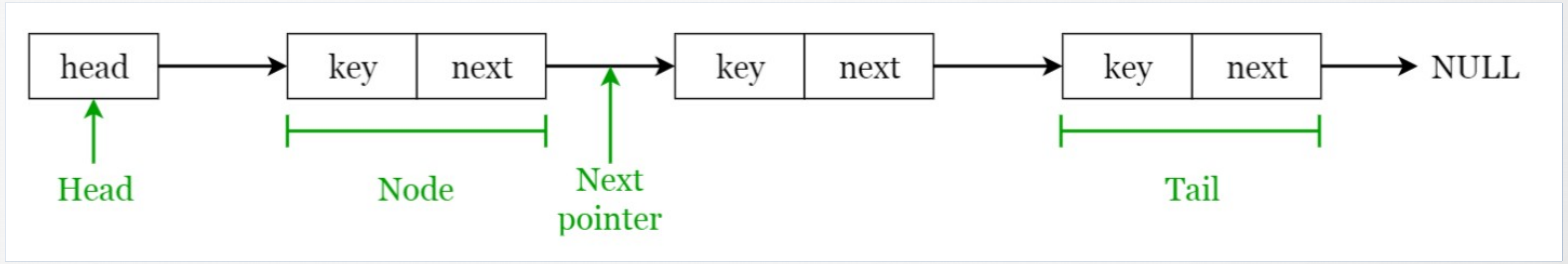
- No casting necessary anymore. The cage is versatile and elegant

Collections

Data Structures Implemented

- ArrayList
  - Add/remove at end – O(1)
  - Add/remove at start/mid – O(N)
  - Get element - O(1)

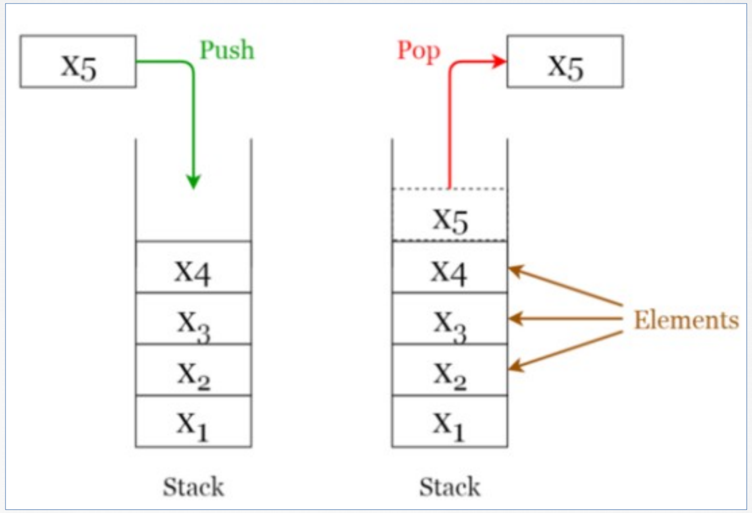


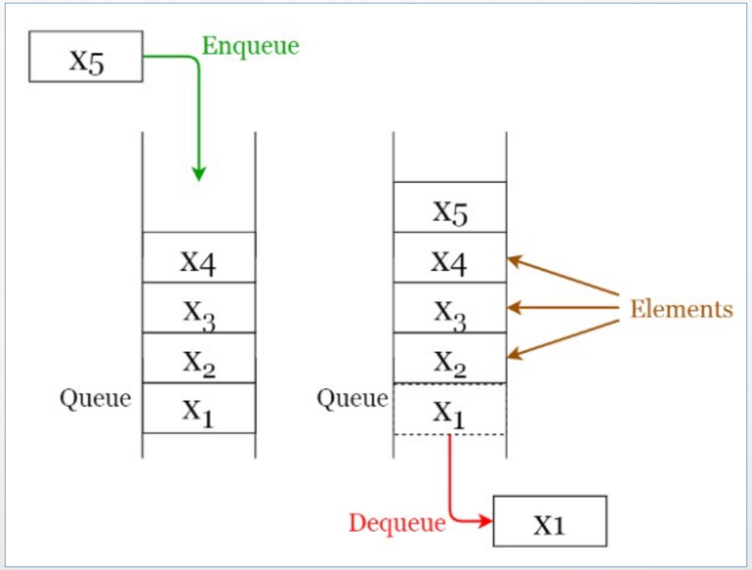


- LinkedList
  - Add/remove at end – O(1)
  - Add/remove at start – O(1)
  - Add/remove at mid - O(N)
  - Get element at mid - O(N)

Data Structures Implemented

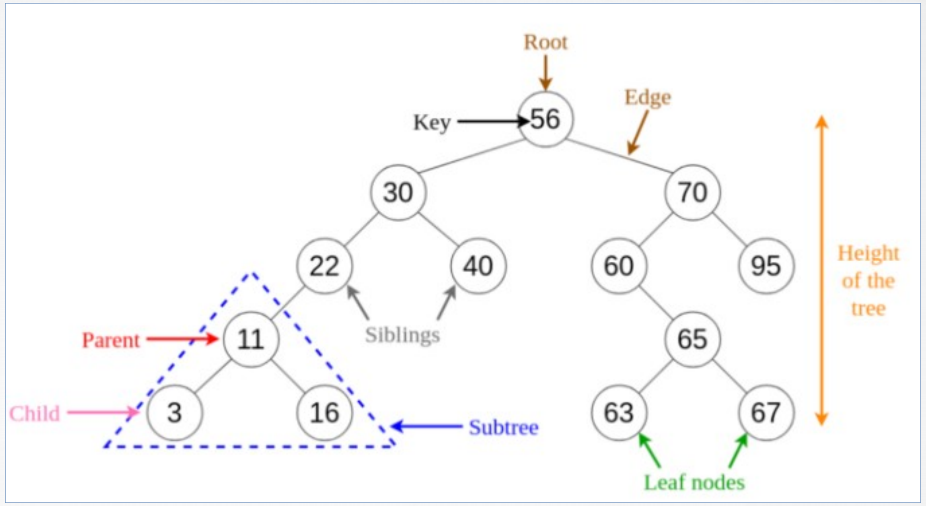
- Stack
- Queue



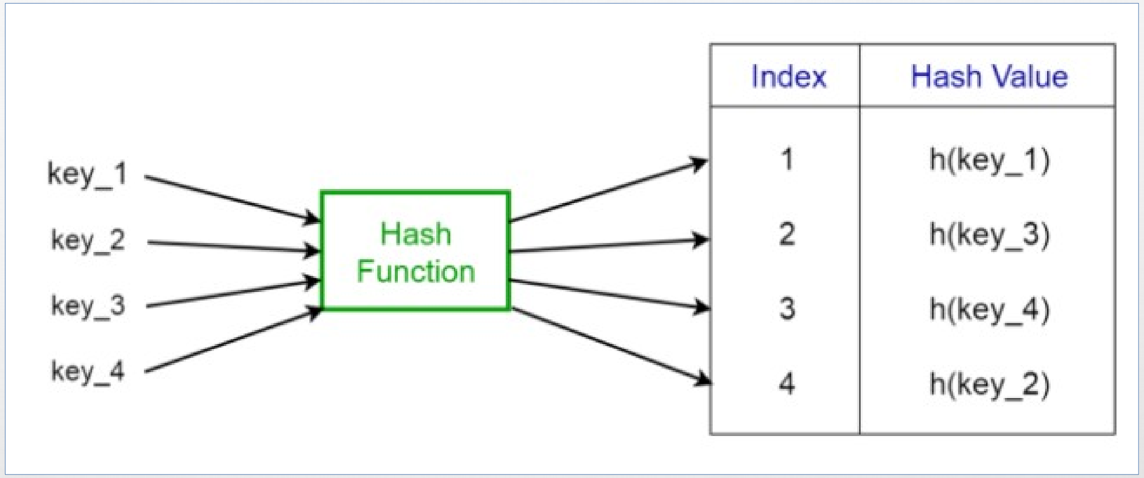


Data Structures Implemented

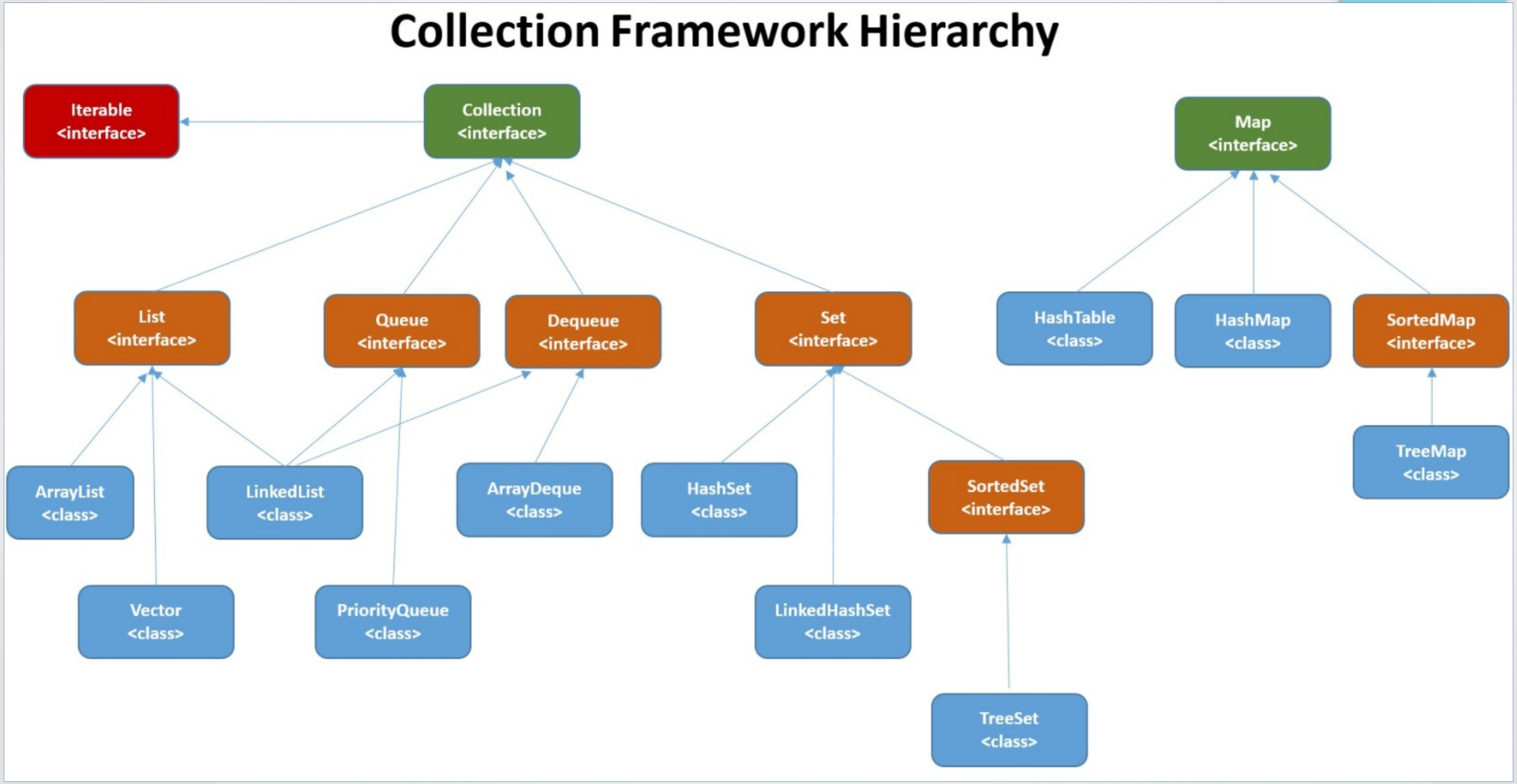
- Tree (BST)
  - All Operations are O(log)
  - Elements are sorted
  - Elements need to be comparable



- HashTable
  - All operations are O(1)
  - Elements need hash
  - Elements are unordered



Basic Interfaces



ArrayList & LinkedList

- Implement **List** interface
- Use indexes
- Dinamyc size (autoalocate)
- We use ArrayList 99% of the time. Only prefer LinkedList when adding/removing from the beginning of the list

```java
List<String> list = new ArrayList();//new LinkedList()
list.add("Pesho");//adds to index 0
list.add(0, "Gosho");//adds to index 0, the others are shifted
list.add("Tosho");//adds to index 2 (since 2 items already in)
System.out.println(list.size());//3 elements
System.out.println(list.get(0));//get first
if(list.contains("Gosho")) {//true
    list.remove("Gosho");//removes, shifts the rest
}
list.clear();//removes every element
```

Stack & Queue

- Stack implements List, but adds stack-friendly methods
- Queue is an interface. Common implementation is the LinkedList & PriorityQueue

```java
Stack<String> stack = new Stack<>();
stack.push( item: "Pesho");
stack.push( item: "Gosho");
System.out.println(stack.peek());//retrieves but does not remove
while(!stack.empty()) {
    System.out.println(stack.pop());//retrieves and removes
}
```

```java
Queue<String> stack = new LinkedList<>();
stack.offer( e: "Pesho");
stack.offer( e: "Gosho");
System.out.println(stack.peek());//retrieves but does not remove
System.out.println(stack.poll());//retrieves and removes
```

Sets

- Provide a collection with guaranteed **unique** elements
- Implemented with Tree (TreeSet) and HashTable (HashSet)

```java
TreeSet<String> set = new TreeSet<>();
set.add("Pesho");
set.add("Pesho");//will be skipped
set.add("Ivan");
System.out.println(set);//Ivan, Pesho (sorted)
```

```java
HashSet<String> set = new HashSet<>();
set.add("Pesho");
set.add("Pesho");//will be skipped
set.add("Ivan");
System.out.println(set);//unordered
```

```java
LinkedHashSet<String> set = new LinkedHashSet<>();
set.add("Pesho");
set.add("Pesho");//will be skipped
set.add("Ivan");
System.out.println(set);//ordered (as inserted)
```

Sets

- TreeSet requires that the elements are **Comparable**
- Comparable<T> and Comparator<T>

```
TreeSet<String> set = new TreeSet<>((e1, e2) -> e1.length() - e2.length());
set.add("Ananas");
set.add("Banan");
set.add("Luk");
System.out.println(set);//Lik, Banan, Ananas
```

- If set has no Comparator or elements are not Comparable, a ClassCastException will occur

Sets

- HashSet requires that the elements are implementing **.hashCode** and **.equals**
- Two objects that are considered „the same“ should retrieve the same value for .**hashCode()** and their .**equals()** should return true
- Failing this will result in either duplicates entering the set or noone entering the set.

Maps

- Provide a collection with key-value pairs
- Implemented with Tree (TreeMap) and HashTable (HashMap)

```
TreeMap<String, Integer> map = new TreeMap<>();
map.put("Ananas", 3);
map.put("Banana", 5);
map.put("Banana", 87);//overrides the previous value of 5
map.put("Luk", 1);
System.out.println(map);//sorted by keys
```

```
HashMap<String, Integer> map = new HashMap<>();
map.put("Ananas", 3);
map.put("Banana", 5);
map.put("Banana", 87);//overrides the previous value of 5
map.put("Luk", 1);
System.out.println(map);//unordered
```

```
LinkedHashMap<String, Integer> map = new LinkedHashMap<>();
map.put("Ananas", 3);
map.put("Banana", 5);
map.put("Banana", 87);//overrides the previous value of 5
map.put("Luk", 1);
System.out.println(map);//preserves insertion order
```

Maps

- TreeMap requires that the key elements are Comparable

```
TreeMap<String, Integer> map = new TreeMap<>((e1, e2) -> e1.length() - e2.length());
map.put("Ananas", 3);
map.put("Banana", 5);
map.put("Banana", 87);//overrides the previous value of 5
map.put("Luk", 1);
System.out.println(map);//sorted by length
```

- If set has no Comparator or elements are not Comparable, a ClassCastException will occur

Maps

- HashMap requires that the **key** elements are implementing **.hashCode** and **.equals**
- Two **keys** that are considered „the same“ should retrieve the same value for .**hashCode()** and their .**equals()** should return true
- Failing this will result in either duplicates entering the set or noone entering the set.

Iterating over collections

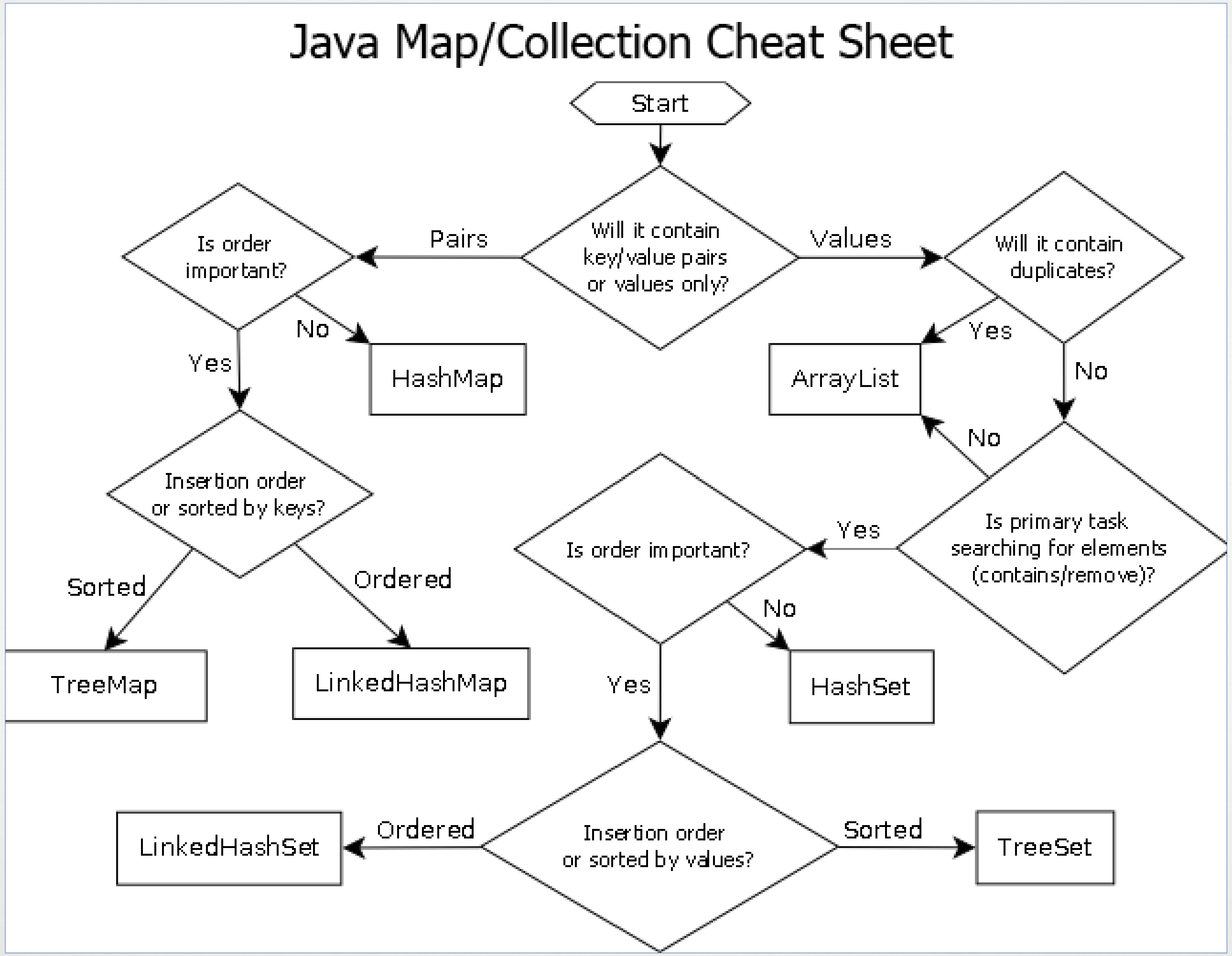
- Foreach loop
  - Super convenient
  - No index usage

```java
ArrayList<String> list = new ArrayList<>();
//fill list
for(String s : list){
    System.out.println(s);
}
//applied for arrays and all collections
```

- Iterator
  - Can delete during iteration

```java
ArrayList<String> list = new ArrayList<>();
//fill list
for(Iterator<String> it = list.iterator(); it.hasNext();){
    System.out.println(it.next());
}
//applied for arrays and all collections
```

Collections Cheat Sheet



Autoboxing

Autoboxing

- Collections can contain only reference types, because they need some methods from Object as well as some interface implementations
- Primitives do not qualify for elements of collections
- Wrapper classes come into play

primitive type	wrapper class
byte	Byte
short	Short
int	Integer
long	Long
double	Double
float	Float
char	Character
boolean	Boolean

- Autoboxing

```
Integer x = 5;
```

- Autounboxing

```
int x = Integer.valueOf(5);
```

Autoboxing

- Collections can use wrappers as generic types and thus accept primitives that are autoboxed/unboxed when needed

```
ArrayList<Integer> list = new ArrayList<>();
list.add(3);//primitive 3 boxed to Integer
list.add(61);
int element = list.get(1);//Integer 61 unboxed to primitive int
list.add(list.get(0) + list.get(1));//boxing and unboxing -> 64
System.out.println(list);
```

Stream API

Characteristics of Streams

- Not related to InputStreams and OutputStreams
- Bring memory efficiency and readability
- Functional programming inspired
- Designed for lambdas
- Can easily be output as arrays or lists
- Can be easily made to work in multithreading environment

Stream Processing

- A Java Stream is a component that is capable of **internal iteration** of its elements, meaning it can iterate its elements **itself**.

- You can attach **listeners** to a Stream. The listeners are called once for each element in the stream. That way each listener gets to process each element in the stream

- The listeners of a stream form a chain. The first listener in the chain can process the element in the stream, and then return a new element for the next listener in the chain to process

Stream Processing

- A Stream is processed through a pipeline of operations
- A Stream starts with a source data structure
- Intermediate methods are performed on the Stream elements. They produce Streams and are not processed until the terminal method is called.
- The Stream is considered consumed when a terminal operation is invoked. No other operation can be performed on the Stream elements afterwards

Creating Streams

- From Individual values

```
Stream<Integer> str1 = Stream.of(4,6,2,4,2);
```

- From Arrays

```
String[] words = {"Peter", "Krasi", "Niki"};
Stream<String> str2 = Stream.of(words);
Stream<String> str3 = Arrays.stream(words);
```

- From Collections

```
List<String> list = List.of("Peter", "Krasi", "Niki");
Stream<String> str4 = list.stream();
```

Optional Class

- A container which may or may not contain a non-null value

- Common Methods
  - isPresent() – returns true if value is present
  - get() – returns value if present
  - orElse(T other) – returns value if present, or other
  - ifPresent(Consumer) – runs the lambda if value is present

Terminal, Non-terminal Operations

- **Non-terminal** stream operations add a listener to the stream without doing anything else
- **Terminal** stream operations start the internal iteration of the elements, call all the listeners, and return a result

```
List<String> list = List.of("Peter", "Krasi", "Niki");
long count = list.stream()
        .filter((word) -> word.length() >= 5)//non-terminal
        .map(String::toUpperCase)//non-terminal
        .count();//terminal
System.out.println(count);//2
```

Non-terminal Operations

- **filter**(Predicate) - filters out elements from a Java Stream
- **map**(Function) – converts/transforms elements in the Stream
- **distinct**() - eliminates duplicate element. equals() method is used for objects.
- **limit**(x) – returns only the first x elements
- **skip**(x) – returns a stream starting with element x
- **peek**(Consumer) – gives ability to do smth with each element without changing it

Non-terminal Operations

```java
List<String> list = List.of("User1","User2 ","   User3","User4  ",
                            "  User5","  User6  ","User7","User8",
                            "User9","User10","User11");
long count = list.stream()
        .filter((word) -> word.length() >= 5)//returns all longer than 5 symbols
        .map(String::toUpperCase)//transforms all to upper case
        .map(String::strip)//transforms all to be stripped of white spaces
        .distinct()//eliminates duplicates
        .limit(10)//takes only the first 10
        .skip(5)//skips the first 5, so gives from 6 to 10
        .peek(System.out::println)//displays each element
        .count();//returns the number of elements
System.out.println(count);//5
```

```
USER6
USER7
USER8
USER9
USER10
5
```

Terminal Operations

- Typically return a single value.
- Trigger internal iteration and listeners application.
- Result is returned after iteration and all listeners application.
- Ends the chaining of Stream instances from non-terminal operations.

Terminal Operations

- **anyMatch**(Predicate) – returns true if the predicate is true for **ONE** element. False if **NO** elements match the predicate test.
- **allMatch**(Predicate) – returns true if the predicate is true for **ALL** element. False if **ONE** element does not match the predicate test.
- **noneMatch**(Predicate) – returns true if **ALL** elements do not match the predicate. False if **ONE** element matches the predicate test.
- **forEach**(Consumer) – applies an operation, but is void and ends the Stream iterations.
- **findFirst**() - returns the first element from the Stream as an Optional.
- **findAny**() - returns a random element from the Stream as an Optional.

Terminal Operations

- **sorted**(Comparator) – sorts the elements based on the Comparator
- **min**(Comparator) – returns the smallest element based on the Comparator
- **max**(Comparator) – returns the biggest element based on the Comparator
- **count**() – returns the number of elements in the Stream
- **collect**(X) – collects the elements in a container, usually a Collection.
- **reduce**() - reduces all elements into a single element by applying a given operation and holding a total value. For example summing all elements

```
Stream<Integer> s = Stream.of(1,3,23,5);
Integer sum = s.reduce( t: 0, (value, total) -> value + total);//starts from 0
System.out.println(sum);//32
```

- **toArray**() - returns an array of Object containing all element of the Stream

Streams vs For Loop

- **Performance**
  - For small sets of data → for loop is better.
  - For large sets of data → parallel stream is better.
  - Parallel stream has a lot of overhead, should be used wisely.
  - "Performant code usually is not very readable, and readable code usually is not very performant."
- **Readability**
  - Depends on the point of view, but in general streams look more simple, scalable and maintanable.