ROAD-TO-NINJA

Core Java - Beginner (Part 3)

Collections, Exceptions, File I/O

Organised by :

Supported by :

ABOUT ME

Name :	**Mohd Azman Kudus**
Age :	30 years
Java exp :	7 years

Question?

AGENDA

- Integrated Development Environment (IDE)
- Breaking conditions
- Collections Data Structures
  - Set – HashSet, TreeSet
  - List – ArrayList
  - Map – HashMap, TreeMap
  - Queue & Stack
  - Iteration and modification

AGENDA

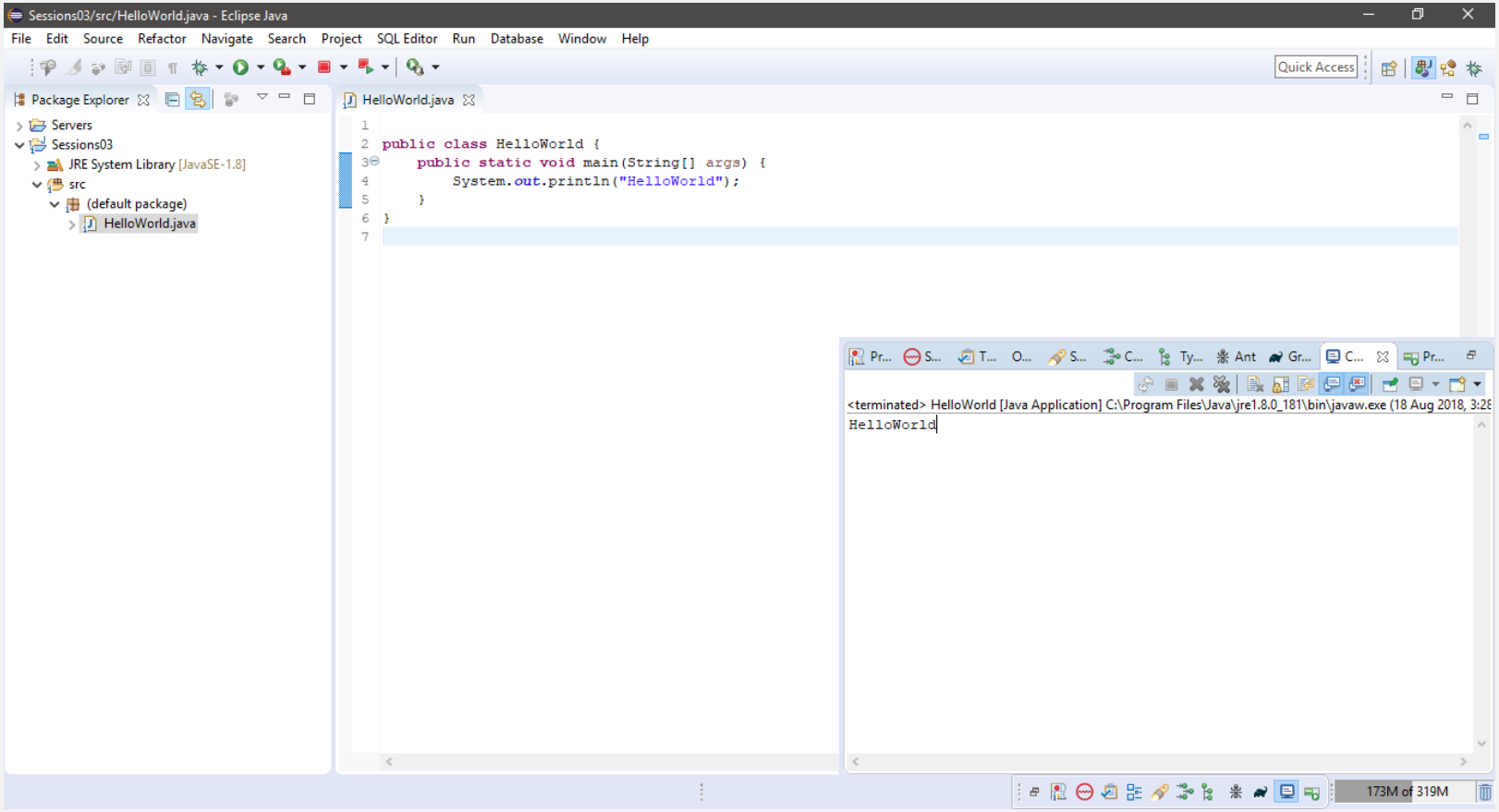
- Execution arguments
  - User defined arguments
  - JVM arguments & Properties
- Errors & Exception
  - Error & Unchecked exception
  - Checked exception
- File Input & Output

INTEGRATED DEVELOPMENT ENVIRONMENT

- Software that provides comprehensive facilities for software development
- Usually, it contains.
  1. Source code editor
  2. Build automation tool
  3. Debugger
  4. Artefacts / Library management
  5. Version control tool (CVS, SVN, Git, Mercurial)
  6. Project browser
  7. Intelligent code completion (IntelliSense)
  8. Code quality analyser (Lint)
  9. Multi language support
  10. Web browser

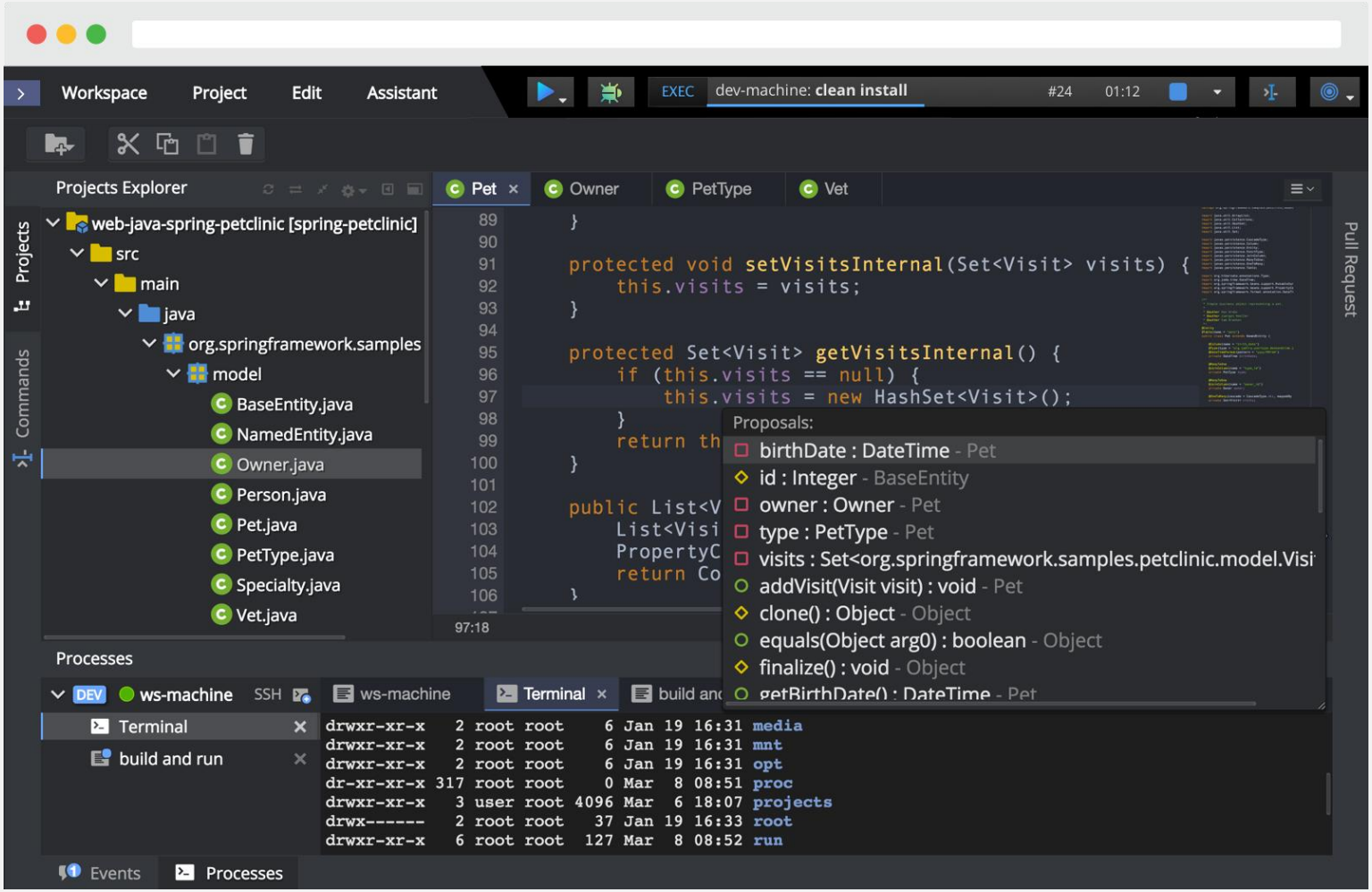
ECLIPSE

- Eclipse is one of the best Java IDE with rich features and extensible modules and plugins.
- Lightweight and not resource hungry.



ECLIPSE CHE

- Eclipse Che is a web based IDE, run almost exactly like Eclipse desktop version.



BREAKING CONDITION

- We can immediately exit a method.
- Keyword: return
- With or without value, depends on method definition.

```
public void exitMethod() {
    int a = 1
    if (a == 1) {
        return;
    }
    else {
        System.out.println("Not yet return");
    }

    System.out.println("Last before exit");
}
```

Method will stop here and back to caller if a equals to 1.

BREAKING CONDITION


```
public boolean returnTrueOrFalse() {
    int a = 1
    if (a == 1) {
        return true;
    }
    else {
        System.out.println("Not yet return");
    }

    System.out.println("Last before return");

    return false;
}
```

Method will stop here if a equals to 1. Then back to caller with value "true".

Default return value

BREAKING CONDITION


```
public int returnEqualsNumber() {
    int[] numbers = new int[] {1,2,3,4,5};
    int a = 3

    for (int num : numbers) {
        if (a == num) {
            return num;
        }
    }

    return 0;
}
```

Method will stop here if num equals to 3. Then exit loop and back to caller with value of "num"

BREAKING CONDITION

- We can exit a loop without iterating all elements.
- Keyword: break

```
public void breakTheLoop() {
    int[] numbers = new int[] {1,2,3,4,5};
    int a = 3

    for (int num : numbers) {
        if (num > a) {
            a = num;
            break;
        }
    }

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

Exit the loop when "num" greater than "a". Then proceed with the rest of the code.

BREAKING CONDITION

- Within a loop, we can skip processes and continue with next element.
- Keyword: continue

```
public void continueTheLoop() {
    int[] numbers = new int[5] {1,2,3,4,5};

    for (int num : numbers) {
        if (num == 3) {
            continue;
        }
        System.out.println(a);
    }

    System.out.println("Done");
}
```

Skip the upcoming processes within the loop and move to the next value.

BREAKING CONDITION

- We can immediately terminate a program.
- Keyword: System.exit(value);

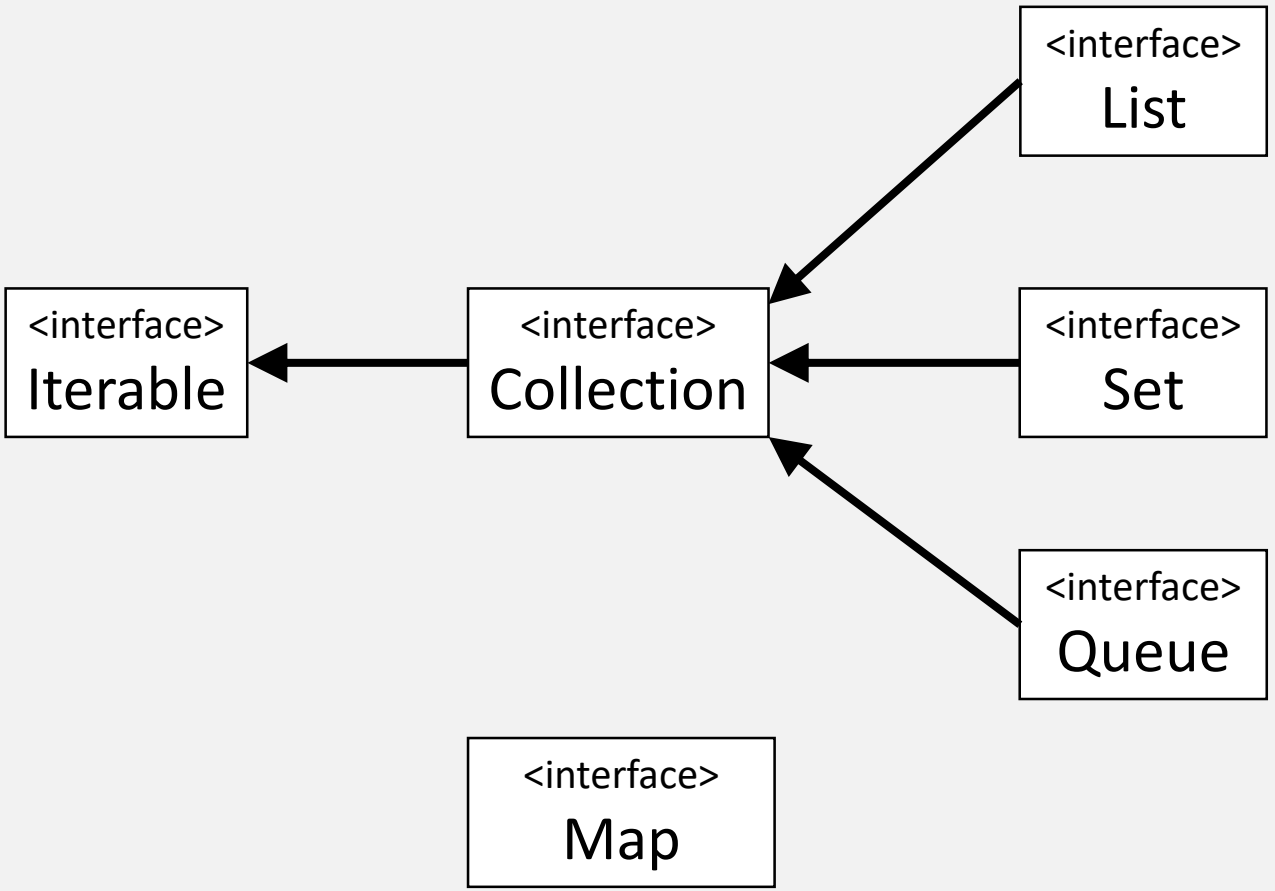
```
public void exitProgram() {
    int[] numbers = new int[5] {1,2,3,4,5};

    for (int num : numbers) {
        if (num == 3) {
            System.exit(0);
        }
        System.out.println(a);
    }

    System.out.println("Done");
}
```

Exit with code "0"

COLLECTIONS

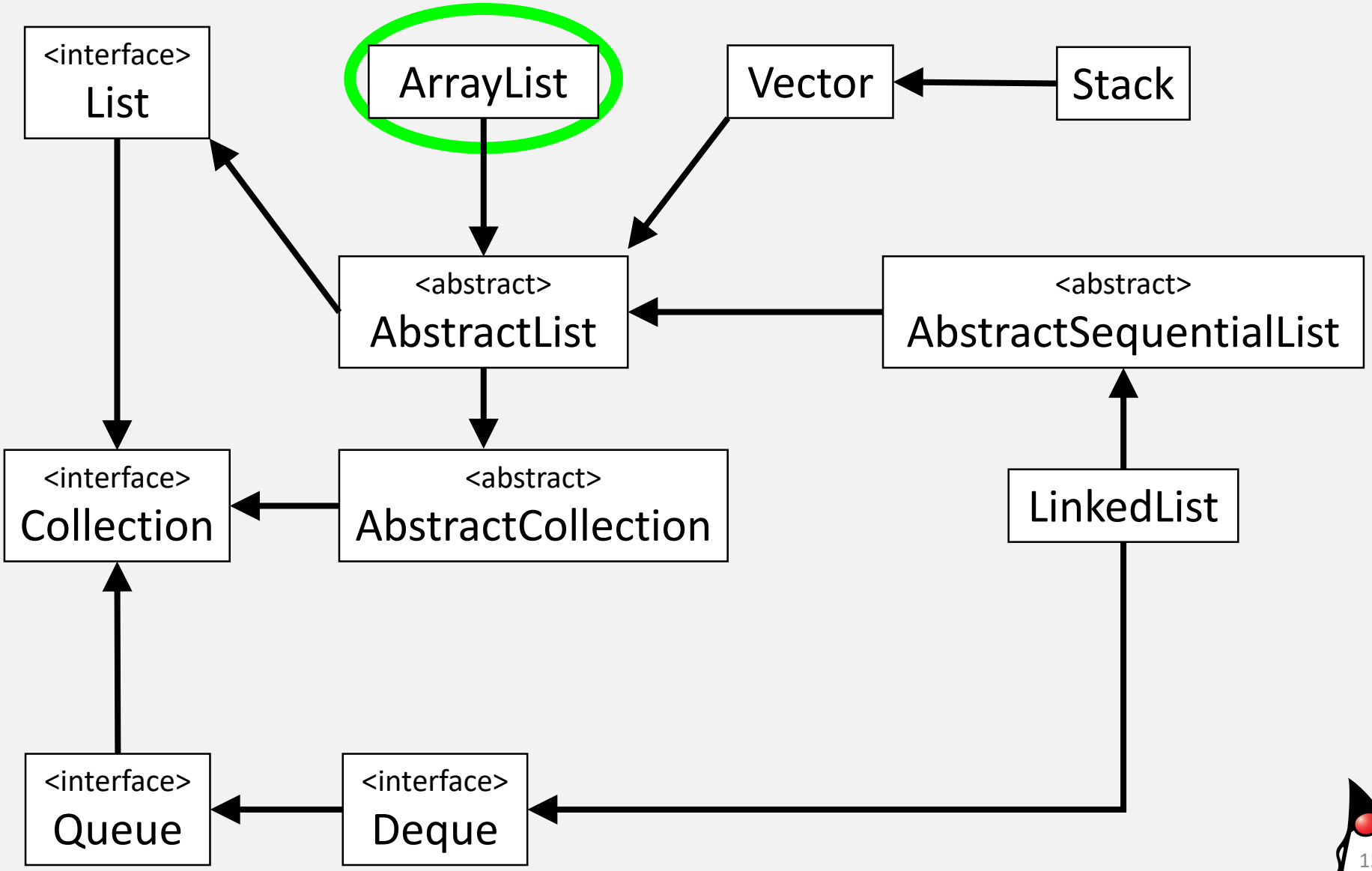
- Classpath: java.util.*
- Diamond operator for instantiation.

```
Collection<DataType> variable = new Collection<>();
```



COLLECTIONS - LIST

- Classpath: java.util.*



LIST - ARRAYLIST

- Sequential entry ordered values with integer indexing, begins with 0.
- Allow duplicate values.
- Accessible using any loop or Iterator

```
List<String> listA = new ArrayList<>();
listA.add("one");
listA.add("two");
listA.add("three");
listA.add("four");
listA.add("four");
listA.add("five");

listA.add(3, "six");

System.out.println(listA.size());
System.out.println(listA.get(0));
System.out.println(listA.get(3));

for (int i = 0; i < listA.size(); i++) {
    System.out.println(listA.get(i));
}
```

Instantiation

Add values

Get total count

Get element at specific index

Iterate

LIST - MODIFY

- Remove can be done outside loop or via Iterator and while loop.
- Add new element must done outside iterator or loop of the collection.
- Replace can be done anywhere

```
Iterator<String> iteratorListA = listA.iterator();
while (iteratorListA.hasNext()) {
    String a = iteratorListA.next();
    System.out.println("iterator-" + a);
    if (a.equals("three") || a.equals("four")) {
        System.out.println("remove " + a);
        iteratorListA.remove();
    }
    else if (a.equals("five")) {
        listA.set(listA.indexOf("five"), "ten");
}

listA.remove(1);

for (String a : listA) {
    System.out.println("final-" + a);
}

System.out.println(listA.size());
```

Every Collection has iterator

Check and get next element.

Remove element using iterator

Update value

Remove element directly from collection

LIST - FIND

- Lookup for elements in the collection
- Keyword: contains

```
if (listA.contains("two")) {
    System.out.println("two is here");
}
```

- Check if collection is empty
- Keyword: isEmpty

```
if (listA.isEmpty()) {
    System.out.println("nothing in here");
}
```

These methods return boolean value

LIST - FIND

- Find the index of the first occurrence of a known element.
- Return -1 if not found.

```
System.out.println(listA.indexOf("three"));
System.out.println(listA.indexOf("ten"));
```

This method return int value as the index/position of the value if found

- Find the index of the last occurrence of a known element.
- Return -1 if not found.

```
System.out.println(listA.lastIndexOf("three"));
System.out.println(listA.lastIndexOf("ten"));
```

LIST - SORT

- Sort using Collections.sort()

```
print(listA);
Collections.sort(listA);
print(listA);
```

Sort ascending order

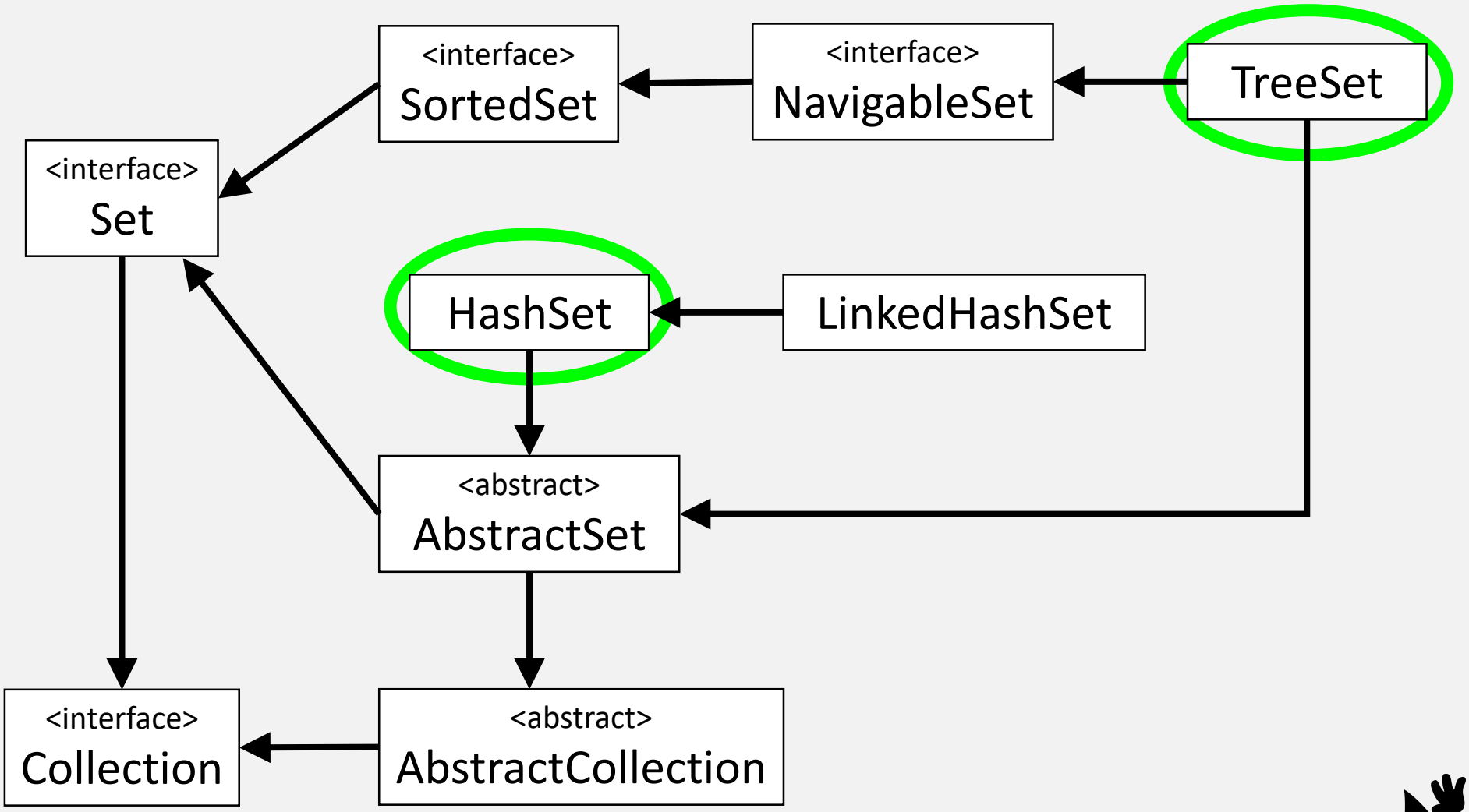
- Sort using Comparator

```
print(listA);
Collections.sort(listA, new Comparator() {
    @Override
    public int compare(Integer o1, Integer o2) {
        return o2.compareTo(o1);
    }
});
print(listA);
```

Sorting condition

COLLECTIONS - SET

- Classpath: java.util.*



SET - HASHSET

- Unordered values without indexing.
- Does not allow duplicate values.
- Accessible using for-each or Iterator and while loop

```
Set<String> setA = new HashSet<>();
setA.add("one");
setA.add("two");
setA.add("three");
setA.add("four");
setA.add("four");
setA.add("five");

System.out.println(setA.size());

for (String a : setA) {
    System.out.println(a);
}

Iterator<String> iteratorSetA = setA.iterator();
while (iteratorSetA.hasNext()) {
    String a = interatorSetA.next();
    System.out.println(a);
}
```

Instantiation

Add values

Get total count

Iterate

SET - TREESET

- Ascending ordered values without indexing.
- Does not allow duplicate values.
- Accessible using for-each or Iterator and while loop.

```
TreeSet<String> setB = new TreeSet<>();
setB.add("one");
setB.add("two");
setB.add("three");
setB.add("four");
setB.add("four");
setB.add("five");
```

Instantiation

Add values

- Upon instantiation, we can define the condition of ordering using Comparator.

```
TreeSet<Integer> setB = new TreeSet<>(new Comparator<Integer>() {
    @Override
    public int compare(Integer o1, Integer o2) {
        return o1.compareTo(o2);
    }
});
```

Sorting condition

SET - FIND

- Lookup for elements in the collection

```
if (setB.contains("two")) {
    System.out.println("two is here");
}
```

- Check if collection is empty

```
if (setB.isEmpty()) {
    System.out.println("nothing in here");
}
```

These methods return boolean value

SET - MODIFY

- Remove only can be done via Iterator and while loop.
- Add new element must done outside iterator or loop of the collection.

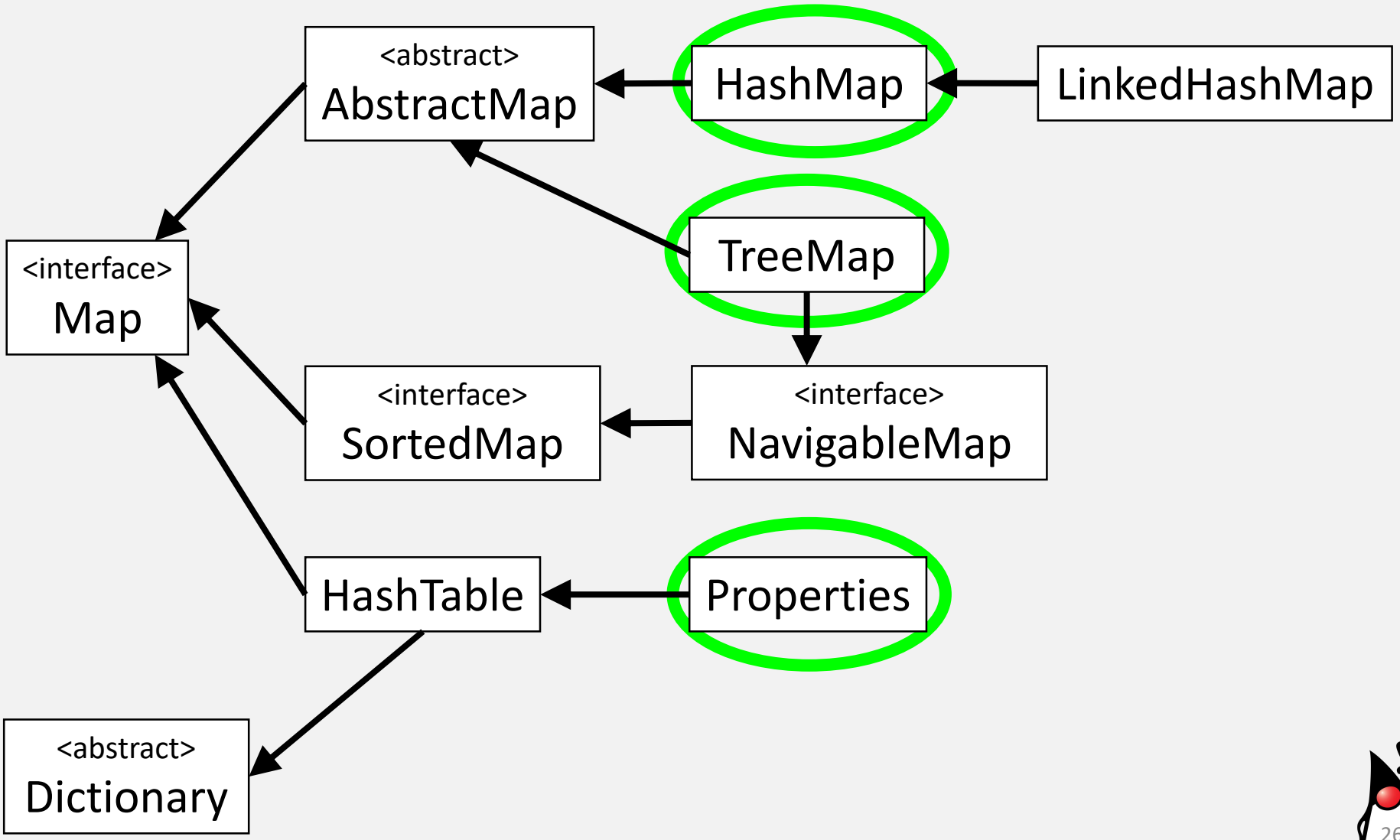
```
Iterator<String> iteratorSetA = setA.iterator();
while (iteratorSetA.hasNext()) {
    String a = iteratorSetA.next();
    System.out.println("iterator-" + a);
    if (a.equals("three") || a.equals("four")) {
        System.out.println("remove " + a);
        iteratorSetA.remove();
    }
}

setA.add("six");

for (String a : setA) {
    System.out.println("final-" + a);
}

System.out.println(setA.size());
```

Check and get next element.

Remove element using iterator

COLLECTIONS - MAP

- Classpath: java.util.*



MAP - HASHMAP

- Unordered sets of key and value. Indexed by the key.
- Keys are unique. If add using key which already exists, it will replace the value.
- Accessible using any loop and Iterator

```
HashMap<Integer, String> mapA = new HashMap<>();
mapA.put(1, "one");
mapA.put(2, "two");
mapA.put(3, "three");
mapA.put(4, "four");
mapA.put(5, "five");
mapA.put(2, "twelve");

System.out.println(mapA.size());
System.out.println(mapA.get(2));

for (Entry<Integer, String> entryMapA : mapA.entrySet()) {
    Integer key = entryMapA.getKey();
    String value = entryMapA.getValue();
    System.out.println(key + " = " + value);
}
```

Instantiation

Add values

Get total count

Get element at specific index

Iterate

MAP - HASHMAP

```
Set<Entry<Integer, String>> entrySet = mapA.entrySet();
Iterator<Entry<Integer, String>> iteratorEntryMapA = entrySet.iterator();
while (iteratorEntryMapA.hasNext()) {
    Entry<Integer, String> entryMapA = iteratorEntryMapA.next();
    Integer key = entryMapA.getKey();
    String value = entryMapA.getValue();
    System.out.println(key + " = " + value);
}

for (String key : mapA.keyset()) {
    System.out.println("key = " + mapA.get(key));
}
```

Iterate for each element/entry.

Iterate based on keys in the map

MAP - TREEMAP

- Ascending ordered sets of key and value. Indexed by the key.
- Keys are unique. If add using key which already exists, it will replace the value.
- Accessible using any loop and Iterator

```
TreeMap<Integer, String> mapA = new TreeMap<>();
mapA.put(1, "one");
mapA.put(2, "two");
mapA.put(3, "three");
mapA.put(4, "four");
mapA.put(5, "five");
mapA.put(2, "twelve");
```

Instantiation

Add values

- Upon instantiation, we can define the condition of ordering using Comparator.

```
TreeMap<Integer, String> mapA = new TreeMap<>(new Comparator<Integer>() {
    @Override
    public int compare(Integer o1, Integer o2) {
        return o1.compareTo(o2);
    }
});
```

Sorting condition

MAP - FIND

- Lookup for elements in the collection

```
if (mapA.containsKey(2)) {
    System.out.println("two is here");
}

if (mapA.containsValue("two")) {
    System.out.println("two is here");
}
```

Lookup using key

Lookup using value

- Check if collection is empty

```
if (mapA.isEmpty()) {
    System.out.println("nothing in here");
}
```

These methods return boolean value

MAP - MODIFY

- Remove only can be done via Iterator and while loop.
- Add new element must done outside iterator or loop of the collection.

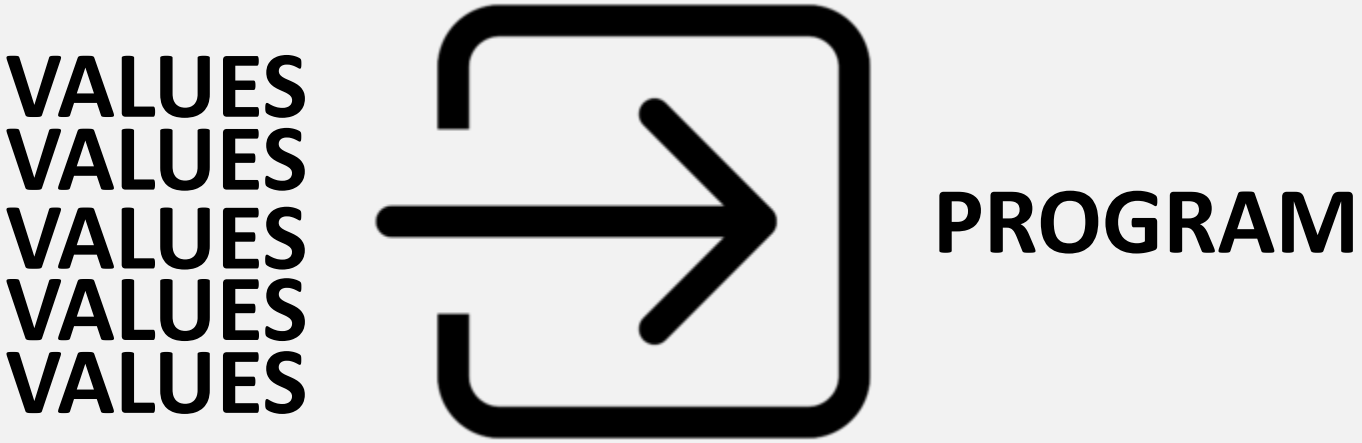
```
Set<Entry<Integer, String>> entrySet = mapA.entrySet();
Iterator<Entry<Integer, String>> iteratorEntryMapA = entrySet.iterator();
while (iteratorEntryMapA.hasNext()) {
    Entry<Integer, String> entryMapA = iteratorEntryMapA.next();
    Integer key = entryMapA.getKey();
    String value = entryMapA.getValue();
    if (key == 1 || value.equals("four") {
        iteratorEntryMapA.remove();
    }
}

mapA.put(44, "fourfour");

for (String a : setA) {
    System.out.println("final-" + a);
}

System.out.println(setA.size());
```

Remove element using iterator

Get set of entries, iterator of the set, then iterate

ARGUMENTS

- One of the input channels to pass data from command line into the program (main method).
- Two types:
  1. User arguments - User input data
  2. JVM arguments - Input properties/configuration values for internal program usage. Usually referred as "System Properties"



ARGUMENTS - USER

- Input parameters from java command.

```
public class HelloArguments {
    public static void main(String[] args) {
        System.out.println(args.length);
        for (String a : args) {
            System.out.println(a);
        }
    }
}
```

Count number of arguments

Show values passed by arguments

```
> java HelloArguments Hello
> java HelloArguments Hello World
> java HelloArguments "Hello World"
```

1 argument

2 arguments

1 argument

User arguments must be defined after main class

ARGUMENTS - JVM

- Input parameters from JVM parameters in java command.
- Keyword: -D
- Also known as system Properties

```
public class HelloArguments {
    public static void main(String[] args) {
        System.getProperty("message1");
        System.getProperty("message2", "none");
    }
}
```

System property

System property with fallback value

JVM arguments must be defined before main class

```
> java -Dmessage1="Hello" -Dmessage2="World" HelloArguments
> java -Dmessage1="Hello" HelloArguments
> java HelloArguments
```

PROPERTIES

- Key value store like a Map, without any diamond operator.
- Key always a String meanwhile value can be any Object
- Can be from system Properties (JVM) or user defined.

```
public class HelloArguments {
    public static void main(String[] args) {
        Properties prop = System.getProperties();
        for (Entry<Object, Object> entryProp : mapA.entrySet()) {
            String key = entryProp.getKey().toString();
            String value = entryMapA.getValue().toString();
            System.out.println(key + " = " + value);
        }
    }
}
```

Just like Map

```
Properties prop2 = new Properties();
prop2.setProperty("message1", "Hello");
prop2.setProperty("message2", "World");
```

No diamond operator

ERRORS & EXCEPTIONS

EXCEPTION

- Programming errors but recoverable via exception handling.
- Keyword: try-catch, try-catch-finally, try-with-resource
- 2 types:
  1. Checked Exception
  2. Unchecked Exception

Checked Exceptions	Unchecked Exceptions
They are known at compile time.	They are known at run time.
They are checked at compile time.	They are not checked at compile time. Because they occur only at run time.
These are compile time exceptions.	These are run time exceptions.
If these exceptions are not handled properly in the application, they give compile time error.	If these exceptions are not handled properly, they don't give compile time error. But application will be terminated prematurely at run time.
All sub classes of java.lang.Exception Class except sub classes of RunTimeException are checked exceptions.	All sub classes of RunTimeException and sub classes of java.lang.Error are unchecked exceptions.

CHECKED EXCEPTION

- Compiler will show compilation error.

```
Path path = Paths.get("tmp/test.txt");
BufferedReader reader = Files.newBufferedReader(path);
```

Unhandled exception type IOException

This statement caused the exception

TRY-CATCH

- Handle with **try-catch**
- Which ever part that cause exception in "try" block, will be handled and run processes in "catch" block will be executed.

```
Path path = Paths.get("tmp/test.txt");
try {
    BufferedReader reader = Files.newBufferedReader(path);

}
catch (IOException e) {
    System.out.println("Error...!!!");
}
```

Do these first.

Do these if any exception raised

TRY-CATCH-FINALLY

- Handle with **try-catch-Finally**
- Which ever part that cause exception in "try" block, will be handled and run processes in "catch" block will be executed.
- "finally" block in both conditions:
  1. after "try" block executed without any exception.
  2. after "catch" block executed when there is an exception in "try" block.

```
Path path = Paths.get("tmp/test.txt");
BufferedReader reader;
try {
    reader = Files.newBufferedReader(path);
}
catch (IOException e) {
    e.printStackTrace();
}
finally {
    reader.close();
}
```

Do these first.

Print errors stack if there is an exception

Do these after complete above process

TRY-WITH-RESOURCES

- Handle with **try-with-resources**.
- Any opened resources will be closed automatically after end of try-catch block instead of using "finally" block

```
Path path = Paths.get("tmp/test.txt");
try (BufferedReader reader = Files.newBufferedReader(path)) {
    reader.readLine();
}
catch (IOException e) {
    e.printStackTrace();
}
```

Only applicable to object that implements AutoCloseable

No need to define "finally" block

THROW

- Handle with **throws**
- Exceptions also can be thrown to the caller. It can be from:
  - an exception from then existing exceptions handling
  - a self-defined exception

```
public int readFile(Path path) throws IOException {
    BufferedReader reader = Files.newBufferedReader(path);
    reader.readLine();
}
```

Whenever exception occurs at this statement, exception will be thrown to the caller.

UNCHECKED EXCEPTION

- Compiler will not show compilation error. But during execution, JVM will stop and show error.

```
int number = Integer.parseInt("abc");
```

No compilation error.

- Handle with try-catch

```
try {
    int number = Integer.parseInt("abc");
}
catch (NumberFormatException e) {
    System.out.println("Not a number");
}
```

- Handle with throws

```
public int convertStringToNumber(String str) throws NumberFormatException
{
    return Integer.parseInt(str);
}
```

Handle like checked exception

UNCHECKED EXCEPTION

- We can throw a new self-defined exception
- E.g.: If something not satisfied, stop the subprocess immediately and tell the problem.

```
public int multiplyNumberBy10(int number) throws Exception {
    if (number < 0) {
        throw new Exception("Number must be positive");
    }
    return number * 10;
}
```

Throw will act like return but with Exception defined here

ERROR

- Unrecoverable exception and not related to the code
- Example: OutOfMemoryError, StackOverflowError
- **DO NOT CATCH...!!! Check your code again.**

FILE INPUT & OUTPUT

INPUT FILE

- Read file to get data to be processed.
- Must specify the file path.
- Can be read in binary mode or character mode.
- Binary mode will use FileInputStream or Files.newInputStream
- Character mode will use FileReader or Files.BufferedReader
- Use java.nio.* for better performance instead of java.io.*

PATH

- Location of a file.
- 2 types:
  1. Absolute path = full path
  2. Relative path = based on current working folder

- Relative path usually used the following notation.
  - Single period (.) as current directory
  - Double period (..) as parent directory

```
Unix/Linux/Mac:
/tmp/work/folder1/test1.txt
subfolder2/test2.txt
./subfolder3/test3.txt
../nextfolder4/test4.txt

Windows:
D:\temp\work\folder5\test5.txt
subfolder6\test6.txt
.\subfolder7\test7.txt
..\nextfolder8\test8.txt
```

BINARY INPUT

- Most of the time, it is referring to files that cannot be read by human.
- Because InputStream is an interface, human readable file can also be read using this method.
- Usually will involve buffer so that we can process data without waiting for full read.

```
byte[] buffer = new byte[8192];
int read = 0;
Path path = Paths.get("tmp/binary.jpg");
try (InputStream inStream = Files.newInputStream(path)) {
    while ((read = inStream.read(buffer)) != -1) {
        // do something
    }
}
catch (IOException e) {
    e.printStackTrace();
}
```

Buffer with size 8kB

Path to file to be read

Open the file

Read into buffer and do something with it

Exception handling if exception occurs

CHARACTER INPUT

- Human readable file.
- Does not involve buffer.
- Usually can be read by line or delimited by token.

```
Path path = Paths.get("tmp/test.txt");
try (BufferedReader reader = Files.newBufferedReader(path)) {
    String line;
    while ((line = reader.readLine()) != null) {
        System.out.println(line);
    }
}
catch (IOException e) {
    e.printStackTrace();
}
```

Path to file to be read

Open the file

Read by line and do something with it

Exception handling if exception occurs

BINARY OUTPUT

- Most of the time, it is referring to files that cannot be read by human.
- Because InputStream is an interface, human readable file can also be written using this method.
- Usually will involve buffer from the InputStream so that we can write data without waiting for full read.

```
Path path = Paths.get("out/binary.pdf");
try (OutputStream outStream = Files.newOutputStream(path)) {
    while ((read = inStream.read(buffer)) != -1) {
        outStream.write(buffer, 0, read);
    }
}
catch (IOException e) {
    e.printStackTrace();
}
```

Path to file to be read

Open the file as new/replace

Write everything in the buffer.

Exception handling if exception occurs

CHARACTER OUTPUT

- Human readable file and oes not involve buffer.
- Write just like System.out.print
- Usually two ways of writing a text file:
  1. New/Replace
  2. Append - Continue writing after the last character in the file.

CHARACTER OUTPUT - NEW

- Writing to new file.

```
Path path = Paths.get("out/new.txt");
try (BufferedWriter writer = Files.newBufferedWriter(path)) {
    writer.write("HelloWorld");
    writer.newLine();
}
catch (IOException e) {
    e.printStackTrace();
}
```

Writing destination

Write the characters

Open the file as new/replace

Exception handling if exception occurs

CHARACTER OUTPUT - APPEND

- Append to existing file.

```
Path path = Paths.get("out/append.txt");
try (BufferedWriter writer = Files.newBufferedWriter(path,
        StandardOpenOption.APPEND)) {
    writer.append("HelloAyam");
    writer.newline();
}
catch (IOException e) {
    e.printStackTrace();
}
```

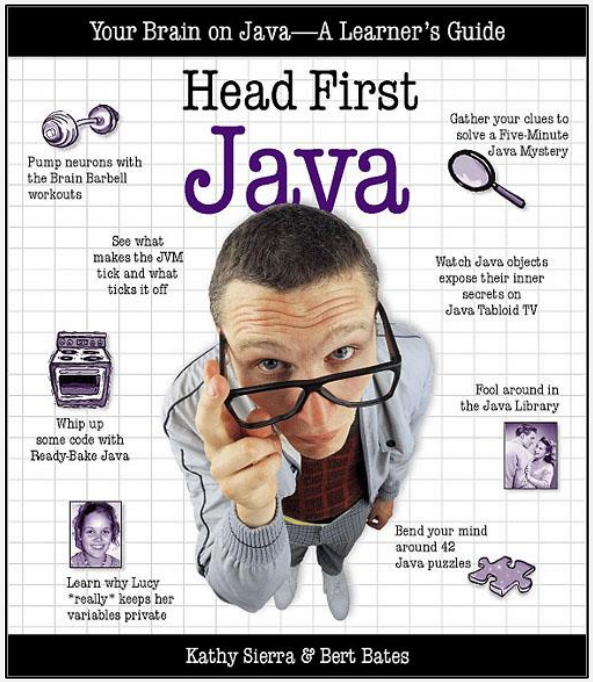
Writing destination

Write the characters

Open the file as append mode

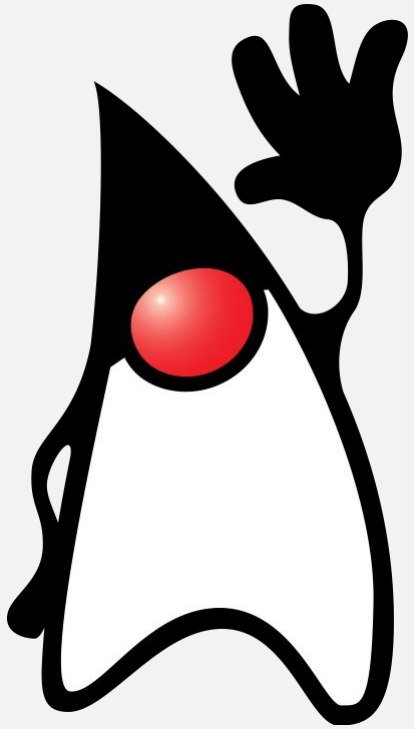
Exception handling if exception occurs

FURTHER READING



Head First Java (2nd Edition)

by Kathy Sierra (Author),
Bert Bates (Author)



THAT'S ALL FOR TODAY
SEE YOU IN THE NEXT CLASS