Lecture 4



Java PROGRAMMING (2)

BY: DR. MOHAMMED ABDELFATTAH

Lecture Objectives

- Review
- HashSet
  - Add Items
  - Item Check
  - Remove Items
  - Remove All Items
  - Get Size
  - Looping
- HashMap
  - Add Items
  - Access Items
  - Remove Items
  - Remove All Items
  - Get Size
  - Looping
  - Get Keys/Values
- Iterator
- Looping Through Collections

Lecture Objectives

- Exceptions
  - try … catch
  - Finally
  - Multiple Catch Blocks
  - throw keyword
- Exception Categories
  - Checked Exceptions
  - Unchecked Exceptions
  - Errors
- Exception Hierarchy
- Exception Class' Methods
- Types of Exceptions
- Hierarchy of Exception classes
- Try with Resources
- Files
  - File Class' Methods
  - Create Files
  - Write To Files
  - Read Files
  - Get File Information
  - Delete Files/Folders

HashSet

A **HashSet** is a collection of items where every **item** is **unique**, and it is found in the **java.util** package.

Create a HashSet object called cars that will store strings:

```
import java.util.HashSet; // Import the HashSet class

HashSet<String> cars = new HashSet<String>();
```

Add Items

The HashSet class has many useful methods. For example, to add items to it, use the add() method.

```
// Import the HashSet class
import java.util.HashSet;

public class Main {
  public static void main(String[] args) {
    HashSet<String> cars = new HashSet<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("BMW");
    cars.add("Mazda");
    System.out.println(cars);
  }
}
```

//Output

[Volvo, Mazda, Ford, BMW]

Note: In the example above, even though BMW is added twice it only appears once in the set because every item in a set has to be unique.

Item Check

To check whether an item exists in a HashSet, use the contains() method.

```
import java.util.HashSet;

public class Main {
  public static void main(String[] args) {
    HashSet<String> cars = new HashSet<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("BMW");
    cars.add("Mazda");

    System.out.println(cars.contains("Mazda"));
  }
}
```

```
//Output
true
```

Remove Items

To remove an item, use the remove() method.

```
import java.util.HashSet;

public class Main {
  public static void main(String[] args) {
    HashSet<String> cars = new HashSet<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("BMW");
    cars.add("Mazda");
    System.out.println(cars);

    cars.remove("Volvo");
    System.out.println(cars);
  }
}
```

//Output

[Volvo, Mazda, Ford, BMW]

[Mazda, Ford, BMW]

Remove All Items

To remove an item, use the remove() method.

```
import java.util.HashSet;

public class Main {
  public static void main(String[] args) {
    HashSet<String> cars = new HashSet<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("BMW");
    cars.add("Mazda");
    System.out.println(cars);

    cars.clear();
    System.out.println(cars);
  }
}
```

```
//Output

[Volvo, Mazda, Ford, BMW]

[]
```

HashSet Size

To find out how many items there are, use the size() method.

```
import java.util.HashSet;

public class Main {
  public static void main(String[] args) {
    HashSet<String> cars = new HashSet<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("BMW");
    cars.add("Mazda");
    System.out.println(cars.size());
  }
}
```

```
//Output
4
```

Loop Through HashSet

Loop through the items of an HashSet with a for-each loop.

```
import java.util.HashSet;

public class Main {

  public static void main(String[] args) {
    HashSet<String> cars = new HashSet<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("BMW");
    cars.add("Mazda");

    for (String i : cars) {
      System.out.println(i);
    }
  }
}
```

```
//Output
Volvo
Mazda
Ford
BMW
```

Other Types

Items in an HashSet are actually objects. In the examples above, we created items (objects) of type "String". Remember that a String in Java is an object (not a primitive type). To use other types, such as int, you must specify an equivalent wrapper class: Integer. For other primitive types, use: Boolean for boolean, Character for char, Double for double, etc.

```
import java.util.HashSet;

public class Main {
  public static void main(String[] args) {
    // Create a HashSet object called numbers
    HashSet<Integer> numbers = new HashSet<Integer>();

    numbers.add(4);        // Add values to the set
    numbers.add(7);
    numbers.add(8);

    // Show which numbers between 1 and 10 are in the set
    for(int i = 3; i <= 10; i++) {
      if(numbers.contains(i)) {
        System.out.println(i + " was found in the set.");
      } else {
        System.out.println(i + " was not found in the set.");
      }
    }
  }
}
```

//Output

3 was not found in the set.

4 was found in the set.

5 was not found in the set.

6 was not found in the set.

7 was found in the set.

8 was found in the set.

9 was not found in the set.

10 was not found in the set.

HashMap

In the ArrayList lecture, you learned that Arrays store items as an ordered collection, and you have to access them with an index number (int type). A HashMap however, store items in "key/value" **pairs**, and you can access them by an index of another type (e.g. a String).

One object is used as a key (index) to another object (value). It can store different types: String keys and Integer values, or the same type, like: String keys and String values.

```
import java.util.HashMap; // import the HashMap class

HashMap<String, String> capitalCities = new HashMap<String, String>();
```

Add Items



The **HashMap** class has many useful methods. For example, to add items to it, use the put() method.

```
import java.util.HashMap;          // Import the HashMap class

public class Main {

  public static void main(String[] args) {

    // Create a HashMap object called capitalCities
    HashMap<String, String> capitalCities = new HashMap<String, String>();

    // Add keys and values (Country, City)
    capitalCities.put("England", "London");
    capitalCities.put("Germany", "Berlin");
    capitalCities.put("Norway", "Oslo");
    capitalCities.put("USA", "Washington DC");

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

//Output
{USA=Washington DC, Norway=Oslo, England=London, Germany=Berlin}

Access Items

To access a value in the HashMap, use the get() method and refer to its key.

```
import java.util.HashMap;          // Import the HashMap class

public class Main {

  public static void main(String[] args) {

    // Create a HashMap object called capitalCities
    HashMap<String, String> capitalCities = new HashMap<String, String>();

    // Add keys and values (Country, City)
    capitalCities.put("England", "London");
    capitalCities.put("Germany", "Berlin");
    capitalCities.put("Norway", "Oslo");
    capitalCities.put("USA", "Washington DC");

    System.out.println(capitalCities.get("England"));
  }
}
```

//Output

London

Remove Items

To remove an item, use the remove() method and refer to the key.

```
import java.util.HashMap;          // Import the HashMap class

public class Main {

  public static void main(String[] args) {

    HashMap<String, String> capitalCities = new HashMap<String, String>();

    // Add keys and values (Country, City)
    capitalCities.put("England", "London");
    capitalCities.put("Germany", "Berlin");
    capitalCities.put("Norway", "Oslo");
    capitalCities.put("USA", "Washington DC");

    capitalCities.remove("England");
    System.out.println(capitalCities);
  }
}
```

//Output
{USA=Washington DC, Norway=Oslo, Germany=Berlin}

Remove All Items

```
import java.util.HashMap;     // Import the HashMap class

public class Main {

  public static void main(String[] args) {

    HashMap<String, String> capitalCities = new HashMap<String, String>();

    // Add keys and values (Country, City)
    capitalCities.put("England", "London");
    capitalCities.put("Germany", "Berlin");
    capitalCities.put("Norway", "Oslo");
    capitalCities.put("USA", "Washington DC");

    capitalCities.clear();
    System.out.println(capitalCities);
  }
}
```

```
//Output

{}
```

HashMap Size

To find out how many items there are, use the size() method.

```
import java.util.HashMap;      // Import the HashMap class

public class Main {

  public static void main(String[] args) {

    HashMap<String, String> capitalCities = new HashMap<String, String>();

    // Add keys and values (Country, City)
    capitalCities.put("England", "London");
    capitalCities.put("Germany", "Berlin");
    capitalCities.put("Norway", "Oslo");
    capitalCities.put("USA", "Washington DC");

    System.out.println(capitalCities.size());
  }
}
```

```
//Output

4
```

Loop Through HashMap: Print Keys

Loop through the items of a HashMap with a for-each loop. Use the keySet() method if you only want the keys, and use the values() method if you only want the values.

```
import java.util.HashMap;

public class Main {
  public static void main(String[] args) {
    HashMap<String, String> capitalCities = new HashMap<String, String>();
    // Add keys and values (Country, City)
    capitalCities.put("England", "London");
    capitalCities.put("Germany", "Berlin");
    capitalCities.put("Norway", "Oslo");
    capitalCities.put("USA", "Washington DC");

    // Print keys
    for (String i : capitalCities.keySet()) {
      System.out.println(i);
    }
  }
}
```

```
//Output
USA
Norway
England
Germany
```

Loop Through HashMap: Print Values

Loop through the items of a HashMap with a for-each loop. Use the keySet() method if you only want the keys, and use the values() method if you only want the values.

```
import java.util.HashMap;

public class Main {
  public static void main(String[] args) {
    HashMap<String, String> capitalCities = new HashMap<String, String>();
    // Add keys and values (Country, City)
    capitalCities.put("England", "London");
    capitalCities.put("Germany", "Berlin");
    capitalCities.put("Norway", "Oslo");
    capitalCities.put("USA", "Washington DC");

    // Print values
    for (String i : capitalCities.values()) {
      System.out.println(i);
    }
  }
}
```

```
//Output
Washingto
n DC
Oslo
London
Berlin
```

Loop Through HashMap: Print Keys & Values



Loop through the items of a HashMap with a for-each loop. Use the keySet() method if you only want the keys, and use the values() method if you only want the values.

```
import java.util.HashMap;

public class Main {
  public static void main(String[] args) {
    HashMap<String, String> capitalCities = new HashMap<String, String>();
    // Add keys and values (Country, City)
    capitalCities.put("England", "London");
    capitalCities.put("Germany", "Berlin");
    capitalCities.put("Norway", "Oslo");
    capitalCities.put("USA", "Washington DC");

    // Print keys and values
    for (String i : capitalCities.keySet()) {
      System.out.println("key: " + i + " value: " + capitalCities.get(i));
    }
  }
}
```

```
//Output
key: USA  value: Washington DC
key: Norway  value: Oslo
key: England  value: London
key: Germany value: Berlin
```

Other Types

Keys and values in a HashMap are actually objects. In the examples above, we used objects of type "String". Remember that a String in Java is an object (not a primitive type). To use other types, such as int, you must specify an equivalent wrapper class: Integer. For other primitive types, use: Boolean for boolean, Character for char, Double for double, etc.

```
import java.util.HashMap;

public class Main {
  public static void main(String[] args) {
    // Create a HashMap object called people
    HashMap<String, Integer> people = new HashMap<String, Integer>();

    // Add keys and values (Name, Age)
    people.put("John", 32);
    people.put("Steve", 30);
    people.put("Angie", 33);

    for (String i : people.keySet()) {
      System.out.println("key: " + i + " value: " + people.get(i));
    }
  }
}
```

```
//Output
key: Angie      value: 33
key: Steve      value: 30
key: John       value: 32
```

Iterator

An Iterator is an object that can be used to loop through collections, like ArrayList and HashSet. It is called an "iterator" because "**iterating**" is the technical term for looping.

- To use an Iterator, you must import it from the java.util package.

- The iterator() method can be used to get an Iterator for any collection.

Example

```
// Import the ArrayList class and the Iterator class
import java.util.ArrayList;
import java.util.Iterator;

public class Main {
  public static void main(String[] args) {

    // Make a collection
    ArrayList<String> cars = new ArrayList<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("Mazda");

    // Get the iterator
    Iterator<String> it = cars.iterator();

    // Print the first item
    System.out.println(it.next());
  }
}
```

```
//Output

Volvo
```

Looping Through Collections

To loop through a collection, use the hasNext() and next() methods of the Iterator.

```
import java.util.ArrayList;
import java.util.Iterator;

public class Main {
  public static void main(String[] args) {

    // Make a collection
    ArrayList<String> cars = new ArrayList<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("Mazda");

    // Get the iterator
    Iterator<String> it = cars.iterator();

    while(it.hasNext()) {
      System.out.println(it.next());
    }
  }
}
```

```
//Output
Volvo
BMW
Ford
Mazda
```

Removing Items From Collections



Iterators are designed to easily change the collections that they loop through. The **remove()** method can remove items from a collection while looping.

Note: Trying to remove items using a **for loop** or a **for-each** loop would not work correctly because the collection is changing size at the same time that the code is trying to loop.

For example, Use an iterator to remove numbers less than 10 from a collection.

Example



For example, Use an iterator to remove numbers less than 10 from a collection.

```
import java.util.ArrayList;
import java.util.Iterator;

public class Main {
  public static void main(String[] args) {
    ArrayList<Integer> numbers = new ArrayList<Integer>();
    numbers.add(12);
    numbers.add(8);
    numbers.add(2);
    numbers.add(23);

    Iterator<Integer> it = numbers.iterator();
    while(it.hasNext()) {
      Integer i = it.next();
      if(i < 10) {
        it.remove();
      }
    }
    System.out.println(numbers);
  }
}
```

//Output

[12, 23]

Why Exception Occurs?

An exception can occur for many different reasons. Following are some scenarios where an exception occurs.

- A user has entered an invalid data.
- A file that needs to be opened cannot be found.
- A network connection has been lost in the middle of communications or the JVM has run out of memory.

Exceptions

When executing Java code, different errors can occur: coding errors made by the programmer, errors due to wrong input, or other unforeseeable things.

When an error occurs, Java will normally stop and generate an error message. The technical term for this is: Java will throw an exception (throw an error).

- The **try** statement allows you to define a block of code to be tested for errors while it is being executed.
- The **catch** statement allows you to define a block of code to be executed, if an error occurs in the try block.
- The try and catch keywords come in pairs.

```
try
{
  //  Block of code to try
}
catch(Exception e)
{
  //  Block of code to handle errors
}
```

Example 1

This will generate an error, because **myNumbers[10]** does not exist.

```
public class Main
{
  public static void main(String[ ] args)
  {
    int[] myNumbers = {1, 2, 3};
    System.out.println(myNumbers[10]); // error!
   }
}
```

//Output

Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: Index 10
out of bounds for length 3 at Main.main(Main.java:4)

Example 2

If an error occurs, we can use **try...catch** to catch the error and execute some code to handle it.

```
public class Main
{
  public static void main(String[ ] args)
  {
      try
      {
        int[] myNumbers = {1, 2, 3};
        System.out.println(myNumbers[10]);
      }
      catch (Exception e)
      {
        System.out.println("Something went wrong.");
      }
  }
}
```

//Output
Something went wrong.

Finally

The finally statement lets you execute code, after **try...catch**, regardless of the result.

```
public class Main {
  public static void main(String[] args) {
    try {
      int[] myNumbers = {1, 2, 3};
      System.out.println(myNumbers[10]);
    }
    catch (Exception e) {
      System.out.println("Something went wrong.");
    }
    finally {
      System.out.println("The 'try catch' is finished.");
    }
  }
}
```

```
//Output
Something went wrong.
The 'try catch' is finished.
```

throw keyword

- The throw statement allows you to create a custom error.
- The **throw** statement is used together with an exception type.
- There are many exception types available in Java: ArithmeticException, FileNotFoundException, ArrayIndexOutOfBoundsException, SecurityException, etc.

Example

Throw an exception if age is below 18 (print "Access denied"). If age is 18 or older, print "Access granted“.

```
public class Main {
  static void checkAge(int age) {
    if (age < 18) {
      throw new ArithmeticException("Access denied - You must be at least 18 years old.");
    }
    else {
      System.out.println("Access granted - You are old enough!");
    }
  }

  public static void main(String[] args) {
    checkAge(15); // Set age to 15 (which is below 18...)
  }
}
```

Note: If age was 20, you would not get an exception.

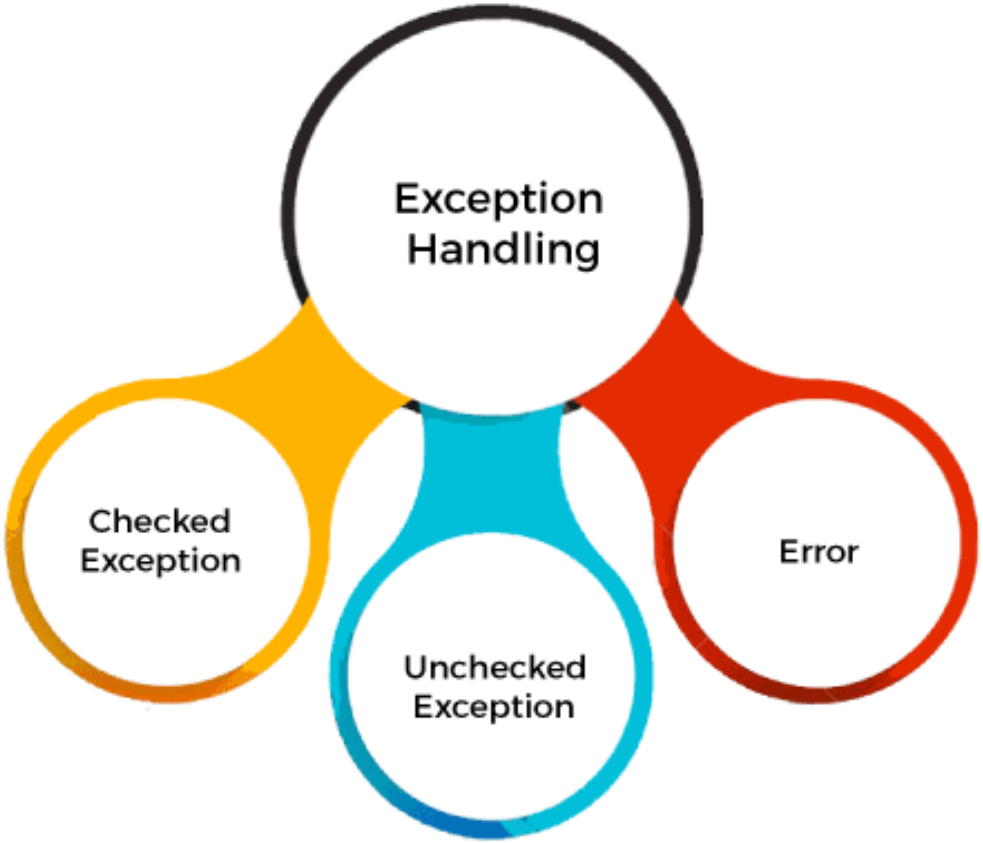
//Output

```
Exception in thread "main"
java.lang.ArithmeticException: Access
denied - You must be at least 18 years old.
        at Main.checkAge(Main.java:4)
        at Main.main(Main.java:12)
```

Exception Categories

Based on these, we have the following categories of Exceptions. You need to understand them to know how exception handling works in Java.

- Checked exceptions
- Unchecked exceptions
- Errors



Checked Exceptions

A **checked exception** is an exception that is checked (notified) by the compiler at compilation-time, these are also called as **compile time exceptions**. These exceptions cannot simply be ignored, the programmer should take care of (handle) these exceptions.

```
import java.io.File;
import java.io.FileReader;

public class Main {
    public static void main(String args[]) {
        File file = new File("E://file.txt");
        FileReader fr = new FileReader(file);
    }
}
```

//Output

Main.java:7: error: unreported exception FileNotFoundException; must be caught or declared to be thrown
FileReader fr = new FileReader(file);

Unchecked Exceptions

An **unchecked** exception is an exception that occurs at the time of execution. These are also called as **Runtime Exceptions**. These include programming bugs, such as logic errors or improper use of an API. Runtime exceptions are ignored at the time of compilation.

```
public class Main {

  public static void main(String args[]) {

    int num[] = {1, 2, 3, 4};

    System.out.println(num[5]);

 }

}
```

//Output

```
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsExcept
ion: Index 5 out of bounds for length 4
      at Main.main(Main.java:7)
```

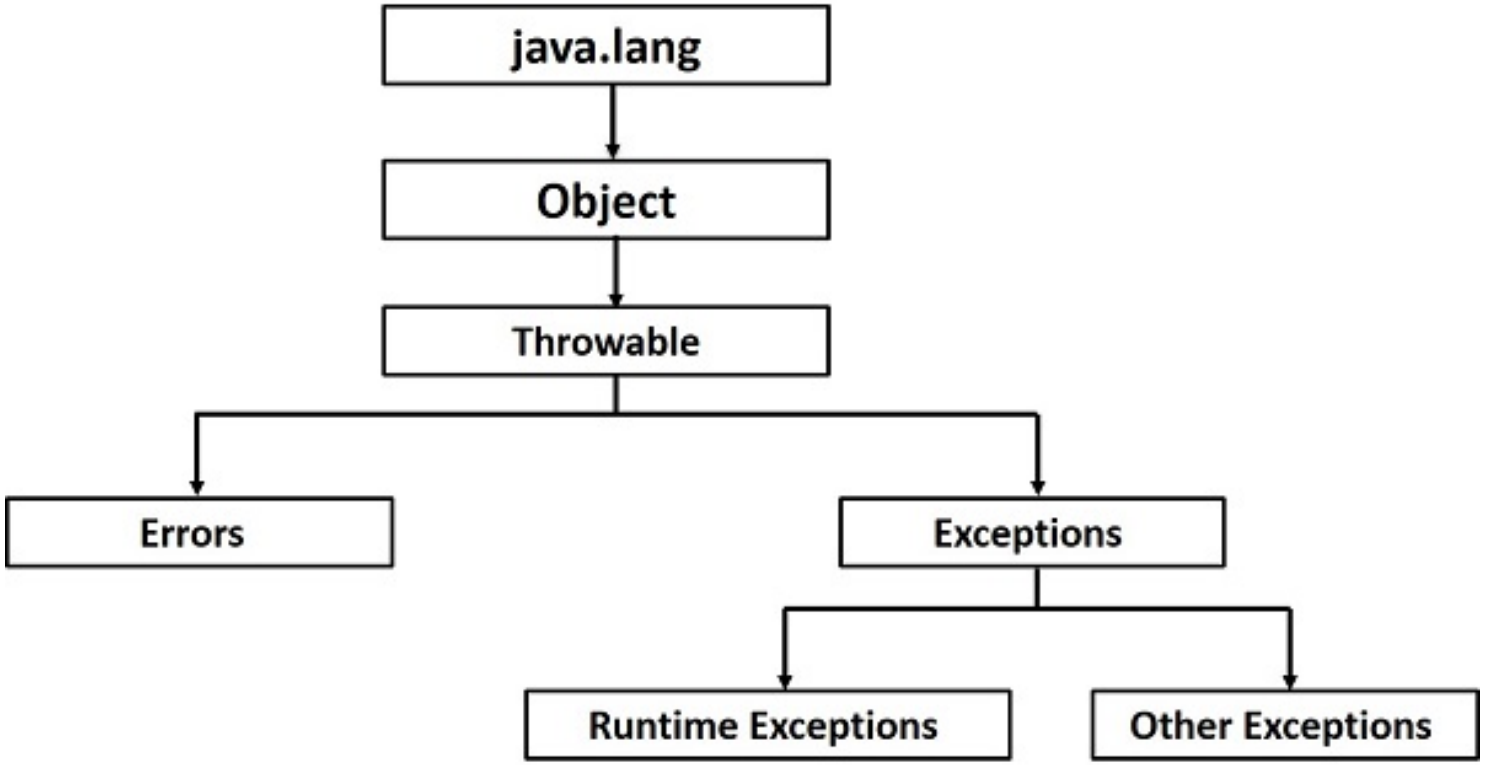
Errors



- These are not exceptions at all, but problems that arise beyond the control of the user or the programmer.
- Errors are typically ignored in your code because you can rarely do anything about an error.
- For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.

Exception Hierarchy

All exception classes are subtypes of the **java.lang.Exception** class. The exception class is a subclass of the Throwable class. Other than the exception class there is another subclass called Error which is derived from the **Throwable class**.



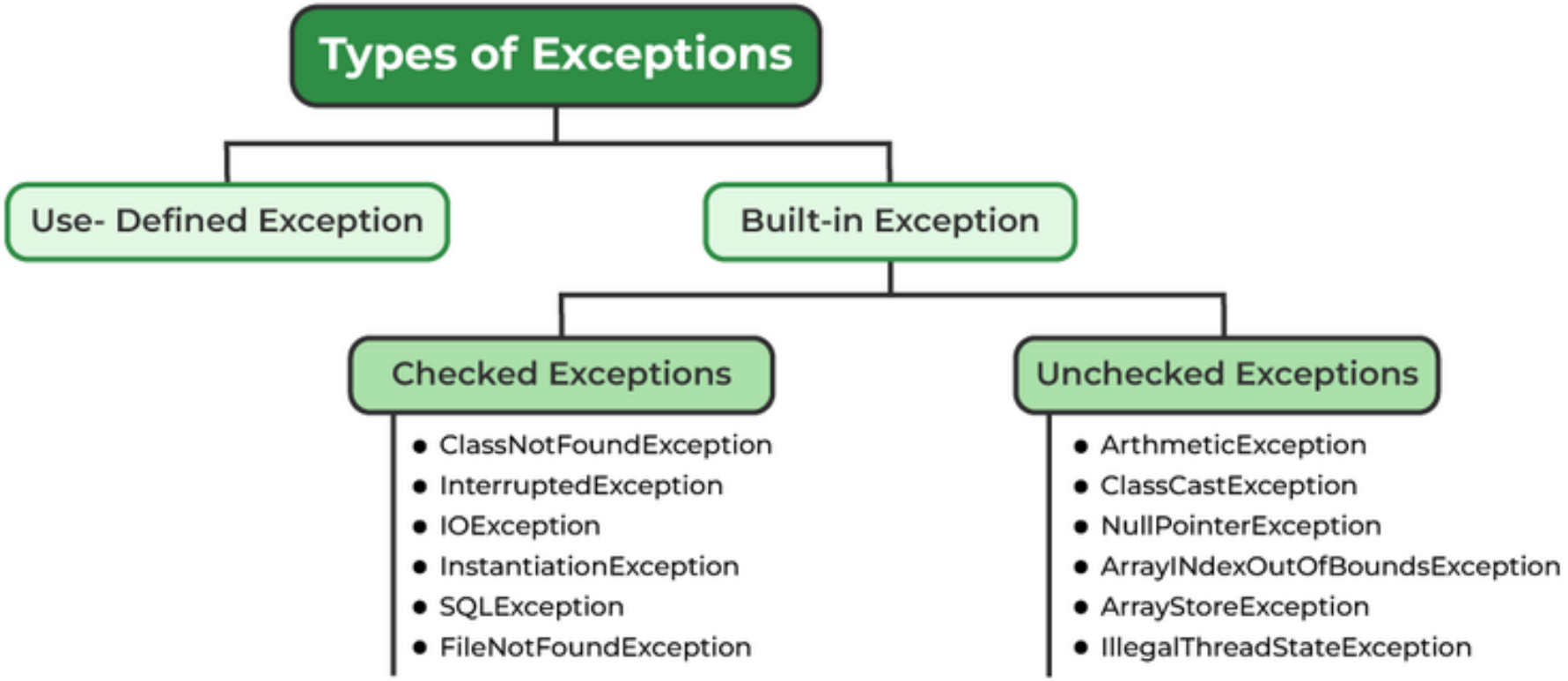
Exception Class Methods

Following is some of the list of important methods available in the Throwable class.

#	Method & Description
1	public String **getMessage()** Returns a detailed message about the exception that has occurred. This message is initialized in the Throwable constructor.
2	public Throwable **getCause()** Returns the cause of the exception as represented by a Throwable object.
3	public String **toString()**
4	public **voidprintStackTrace()** Prints the result of toString() along with the stack trace to System.err, the error output stream.

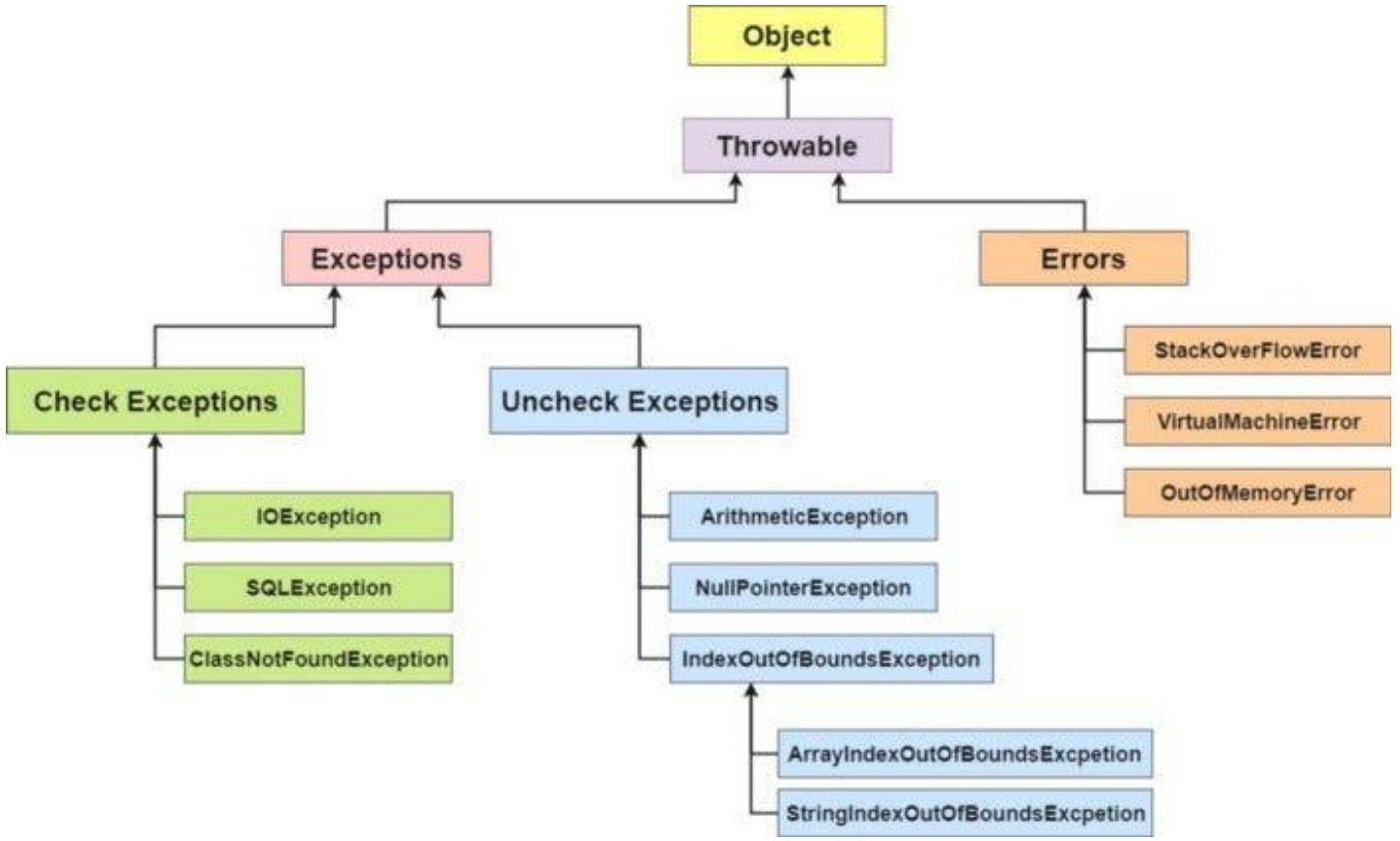
Types of Exceptions

Java defines several types of exceptions that relate to its various class libraries. Java also allows users to define their own exceptions.



Hierarchy of Exception classes

The java.lang.Throwable class is the root class of Java Exception hierarchy inherited by two subclasses: Exception and Error. The hierarchy of Java Exception classes is given below.



Example 1

```
public class Main {
  public static void main(String args[])
  {
      try {
          int a = 30, b = 0;
          int c = a/b;  // cannot divide by zero
          System.out.println ("Result = " + c);
      }
      catch(ArithmeticException e) {
          System.out.println ("Can't divide a number by 0");
      }
  }
}
```

//Output
Can't divide a number by 0

Example 2

```
public class Main {

  public static void main (String[] args) {

    int a=5;
    int b=0;
      try{
        System.out.println(a/b);
      }
      catch(ArithmeticException e){
      System.out.println(e.toString());
    }

  }

}
```

```
e.getMessage();
```

```
//Output
java.lang.ArithmeticException: / by zero
```

Multiple Catch Blocks

Multiple catch blocks in Java are used to catch/handle multiple exceptions that may be thrown from a particular code section. A try block can have multiple catch blocks to handle multiple exceptions.

```
try {
  // Protected code
} catch (ExceptionType1 e1) {
  // Catch block
} catch (ExceptionType2 e2) {
  // Catch block
} catch (ExceptionType3 e3) {
  // Catch block
}
```

Example

```
public class Main {
  public static void main(String args[]) {
    try {
       int a[] = new int[2];
       int b = 0;
       int c = 1/b;
       System.out.println("Access element three :" + a[3]);
    }
    catch (ArithmeticException e) {
       System.out.println("ArithmeticException thrown  :" + e);
    }
    catch (ArrayIndexOutOfBoundsException e) {
       System.out.println("ArrayIndexOutOfBoundsException thrown  :" + e);
    }catch (Exception e) {
        System.out.println("Exception thrown  :" + e);
    }

    System.out.println("Out of the block");
 }
}
```

//Output

ArithmeticException thrown
:java.lang.ArithmeticException: / by zero

Out of the block

Try with Resources

- The Java try-with-resources statement is a try statement that is used for declaring one or more resources such as streams, sockets, databases, connections, etc. These resources must be closed while the program is being finished. The try-with-resources statement closes the resources at the end of the statement.
- The try-with-resources feature was **introduced in Java7**. The try-with-resources can also be a replacement for try-catch-finally with resources objects.
- The try-with-resources statement is also referred as "**Automatic Resource Management**" and it was introduced in Java7. This statement is a replacement of try-catch-finally-statement if you are working resources objects.

```
try(resources declarations) {
  // use of the resources
}
catch(Exception e) {
  // exception handling
}
```

Example

```
import java.io.FileReader;
import java.io.IOException;

public class Main {

   public static void main(String args[]) {
      try(FileReader fr = new FileReader("D://file.txt")) {
         char [] a = new char[50];
         fr.read(a);   // reads the contentto the array
         for(char c : a)
         System.out.print(c);   // prints the characters one by one
      } catch (IOException e) {
        System.out.print(e.getMessage());
      }
   }
}
```

//Output if file not exist
D:\file.txt (The system cannot find the file specified)

Nested Try Block

```
import java.io.File; import java.io.FileReader; import java.io.IOException;

public class Main {
   public static void main(String args[]) {
      FileReader fr = null;
      try {
         File file = new File("file.txt");
         fr = new FileReader(file);
         char [] a = new char[50];
         fr.read(a);              // reads the content to the array
         for(char c : a)
         System.out.print(c);    // prints the characters one by one
      } catch (IOException e) {
        System.out.print(e.getMessage());
      }finally {
         try {
            fr.close();
         } catch (IOException ex) {
          System.out.print(e.getMessage());
         }
      }
   }
}
```

Difference with “Try with Resources”

Files

File handling is an important part of any application. Java has several methods for creating, reading, updating, and deleting files.

- The File class from the **java.io package**, allows us to work with files.
- To use the File class, **create** an object of the class, and specify the filename or directory name.

```
import java.io.File;  // Import the File class

File myObj = new File("filename.txt"); // Specify the filename
```

Note: There are many **available classes** in the Java API that can be used to read and write files in Java: FileReader, BufferedReader, Files, Scanner, FileInputStream, FileWriter, BufferedWriter, FileOutputStream, etc. Which one to use depends on the Java version you're working with and whether you need to **read bytes** or **characters**, and the **size** of the file/lines etc.

File Class' Methods

The File class has many useful methods for creating and getting information about files. For example.

Method	Type	Description
canRead()	Boolean	Tests whether the file is readable or not
canWrite()	Boolean	Tests whether the file is writable or not
createNewFile()	Boolean	Creates an empty file
delete()	Boolean	Deletes a file
exists()	Boolean	Tests whether the file exists
getName()	String	Returns the name of the file
getAbsolutePath()	String	Returns the absolute pathname of the file
length()	Long	Returns the size of the file in bytes
list()	String[]	Returns an array of the files in the directory
mkdir()	Boolean	Creates a directory

Create Files

To create a file in Java, you can use the **createNewFile()** method. This method returns a boolean value: **true** if the file was successfully created, and **false** if the file already exists. Note that the method is enclosed in a try...catch block. This is necessary because it throws an **IOException** if an error occurs (if the file cannot be created for some reason).

To create a file in a specific directory (requires permission), specify the path of the file and use double backslashes to escape the "****" character (for Windows).

```
File myObj = new File("C:\\Users\\MyName\\filename.txt");
```

On Mac and Linux you can just write the path, like: /Users/name/filename.txt

Example

```java
import java.io.File;  // Import the File class
import java.io.IOException;  // Import the IOException class to handle errors

public class Main {
  public static void main(String[] args) {
    try {
      File myObj = new File("filename.txt");
      if (myObj.createNewFile()) {
        System.out.println("File created: " + myObj.getName());
      } else {
        System.out.println("File already exists.");
      }
    } catch (IOException e) {
      System.out.println("An error occurred.");
      e.printStackTrace();
    }
  }
}
```

//Output

File created: filename.txt

Write To Files

In the following example, we use the FileWriter **class** together with its write() **method** to write some text to the file we created in the example above. Note that when you are done writing to the file, you should close it with the close() **method.**

```
import java.io.FileWriter;   // Import the FileWriter class
import java.io.IOException;  // Import the IOException class to handle errors

public class Main {
  public static void main(String[] args) {
    try {
      FileWriter myWriter = new FileWriter("filename.txt");
      myWriter.write("Borg Al Arab University!");
      myWriter.close();
      System.out.println("Successfully wrote to the file.");
    } catch (IOException e) {
      System.out.println("An error occurred.");
      e.printStackTrace();
    }
  }
}
```

//Output

Successfully wrote to the file.

Read Files

We use the **Scanner** class to read the contents of the text file.

```
import java.io.File;  // Import the File class
import java.io.FileNotFoundException;  // Import this class to handle errors
import java.util.Scanner; // Import the Scanner class to read text files

public class Main {
  public static void main(String[] args) {
    try {
      File myObj = new File("filename.txt");
      Scanner myReader = new Scanner(myObj);
      while (myReader.hasNextLine()) {
        String data = myReader.nextLine();
        System.out.println(data);
      }
      myReader.close();
    } catch (FileNotFoundException e) {
      System.out.println("Error: "+ e.getMessage());
      e.printStackTrace();
    }
  }
}
```

//Output

Borg Al Arab University!

Get File Information

To get more information about a file, use any of the File methods.

```
import java.io.File;  // Import the File class

public class Main {
  public static void main(String[] args) {
    File myObj = new File("filename.txt");
    if (myObj.exists()) {
      System.out.println("File name: " + myObj.getName());
      System.out.println("Absolute path: " + myObj.getAbsolutePath());
      System.out.println("Writeable: " + myObj.canWrite());
      System.out.println("Readable " + myObj.canRead());
      System.out.println("File size in bytes " + myObj.length());
    } else {
      System.out.println("The file does not exist.");
    }
  }
}
```

```
//Output
File name: filename.txt
Absolute path: E:\filename.txt
Writeable: true
Readable true
File size in bytes 52
```

Delete Files

To delete a file in Java, use the delete() method.

```
import java.io.File;  // Import the File class

public class Main {

  public static void main(String[] args) {

    File myObj = new File("filename.txt");
    if (myObj.delete()) {
      System.out.println("Deleted the file: " + myObj.getName());
    } else {
      System.out.println("Failed to delete the file.");
    }
  }

}
```

```
//Output
Deleted the file: filename.txt
```

Delete Folders

You can also delete a folder. However, it must be empty.

```
import java.io.File;

public class Main {

  public static void main(String[] args) {

    File myObj = new File("C:\\Test");
    if (myObj.delete()) {
      System.out.println("Deleted the folder: " + myObj.getName());
    } else {
      System.out.println("Failed to delete the folder.");
    }
  }

}
```

```
//Output
Deleted the folder: Test
```

Thanks

References:

https://www.w3schools.com & https://www.tutorialspoint.com & https://www.javatpoint.com