Agenda

1. Exception Handling
2. What is Exception?
3. Exception Hierarchy
4. Ways of Handling Exception
5. Types of Exception
6. Creating Custom Exception

**package** com.revature;

**public** **class** ExceptionDemo {

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

**int** a = 5; //primitive int variable declaration & initialization

**int** b = 0;

System.***out***.println("a/b =" + (a/b));

//Ctrl+F11 - To run the java code in Eclipse

}

}

Output:

Exception in thread "main" java.lang.ArithmeticException: / by zero

at com.revature.ExceptionDemo.main(ExceptionDemo.java:9)

Exception – Is a condition in a java program which will lead to pre-mature closure/termination of java code.

When a line throws an exception, with in that block other lines of code will not get executed.

If handled properly, we can avoid pre-mature closure/termination of java program.

Two ways of Handling Exception

1. Using try/catch block (recommended)
2. Using throws keyword (passing the responsibility to JVM)

**package** com.revature;

**public** **class** ExceptionDemo {

**public** **static** **void** main(String[] args) **throws** ArithmeticException {

// **TODO** Auto-generated method stub

**int** a = 5; //primitive int variable declaration & initialization

**int** b = 0;

System.***out***.println("a/b = " + (a/b)); //this line will throw an exception

System.***out***.println(" Program Ends here");

//Ctrl+F11 - To run the java code in Eclipse

}

}

Output :

Exception in thread "main" java.lang.ArithmeticException: / by zero

at com.revature.ExceptionDemo.main(ExceptionDemo.java:9)

important keywords used in exception handling mechanism

1. throw – This keyword used to throw an exception object. It is used inside the method.
2. throws – This keyword is used to inform the JVM that the method might throw some exception. It is used in method signature only
3. try – This keyword is used to handle the code which might throw exception
4. catch – This keyword is used to catch the exception object and avoid pre-mature termination
5. finally – This keyword is used to do clean-up activities like closing resources (file/db connection/scanner…)

Types of Exception

1. Runtime or Un-checked Exception – exception handling code is not compulsory (All the sub-classes of RuntimeException class – IndexOutofBoundException, ArithmeticException, ClassCastException, IllegalArgumentException, NullPointerException)
2. Compile or Checked Exception - Exception handling code is compulsory. (ClassNotFoundException, FileNotFoundException, IOException, InstantiationException, IllegalAccessException, SQL Exception)

Types of Exception based on the creator

1. Built-in or Pre-defined Exceptions
2. User-defined or custom exceptions

In java “\” is a escape sequence. It will not be printed in the console if it is a single backslash.

Rules with respect to try/catch block

1. Try block must have either a catch block or finally block
2. Try with catch block is valid
3. Try with finally block is also valid (It’s not recommended) - It’s similar to using throws keyword
4. Catch block is actually handling the exception
5. Try block can be followed by multiple catch blocks
6. Similarly multiple exception object can be handled by single catch block
7. Finally block codes will get executed all the times irrespective of exception status

**package** com.revature;

**import** java.io.File;

**import** java.io.IOException;

**import** java.util.Scanner;

**public** **class** ExceptionHandlingRules {

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

// try block with catch block

**try** {

System.***out***.println("Inside first try block");

} **catch** (Exception e) {

System.***out***.println("Exception caught");

}

// try block with finally block

**try** {

System.***out***.println("Inside second try block");

} **finally** {

System.***out***.println("Inside finally block");

// resource clean-up code will be added

}

// try block with multiple catch block

**try** {

**int** a = 5;

**int** b[] = **new** **int**[5];

b[0] = 0;

// System.out.println(a / b[0]);

// System.out.println(b[5]);

File f = **new** File("");

f.createNewFile();

} **catch** (ArithmeticException e) {

System.***out***.println("ArithmeticException caught");

} **catch** (ArrayIndexOutOfBoundsException e) {

System.***out***.println("caught ArrayIndexOutOfBoundsException");

} **catch** (RuntimeException e) {

System.***out***.println("caught RuntimeException");

} **catch** (IOException e) {

System.***out***.println("IO Exception Caught " + e.getMessage());

} **catch** (Exception e) {

System.***out***.println("Exception Caught" + e.getMessage());

}

// try with single catch block handling multiple exceptions

**try** {

**int** a = 5;

**int** b[] = **new** **int**[5];

b[0] = 0;

// System.out.println(a / b[0]);

// System.out.println(b[5]);

File f = **new** File("");

f.createNewFile();

} **catch** (ArithmeticException | ArrayIndexOutOfBoundsException | IOException e) {

System.***out***.println("Exception caught " + e.getMessage());

}

//try with resource

Scanner scan = **new** Scanner(System.***in***);

**try** (Scanner input = **new** Scanner(System.***in***)) {

System.***out***.println("Inside try with resource block");

}**catch** (Exception e) {

// **TODO**: handle exception

} **finally** {

System.***out***.println("This is try with resource example. the Resource will be closed automatcally upon exit");

}

scan.close();

}

}

Output :

Inside first try block

Inside second try block

Inside finally block

IO Exception Caught The system cannot find the path specified

Exception caught The system cannot find the path specified

Inside try with resource block

This is try with resource example. the Resource will be closed automatcally upon exit

Custom Exception

1. Create a class called MyFirstException and extend the RuntimeException class.

**package** com.revature;

**public** **class** Starter {

**public** **static** **void** main(String[] args) **throws** MyFirstException {

// **TODO** Auto-generated method stub

**int** a = 25;

**if** (a > 25) {

**throw** **new** MyFirstException();

} **else** {

System.***out***.println(" a is less than or equal to 25");

}

**int** age = 15;

**if** (age < 18) {

**throw** **new** InvalidAgeException();

} **else** {

System.***out***.println("Valid Age");

}

}

}

Output :

a is less than or equal to 25

Exception in thread "main" com.revature.InvalidAgeException: Age is Invalid

at com.revature.Starter.main(Starter.java:19)

**package** com.revature;

**public** **class** InvalidAgeException **extends** RuntimeException {

@Override

**public** String getMessage() {

// **TODO** Auto-generated method stub

**return** "Age is Invalid";

}

}

**package** com.revature;

//User-defined exception or custom exception

**public** **class** MyFirstException **extends** Exception {

// ctrl+Space or cmd+space

@Override

**public** String getMessage() {

// **TODO** Auto-generated method stub

**return** "MyFirstException Object is thrown";

}

}

Collection API/Framework

API – Application Programming Interface

Class Employee {

-----

-----

}

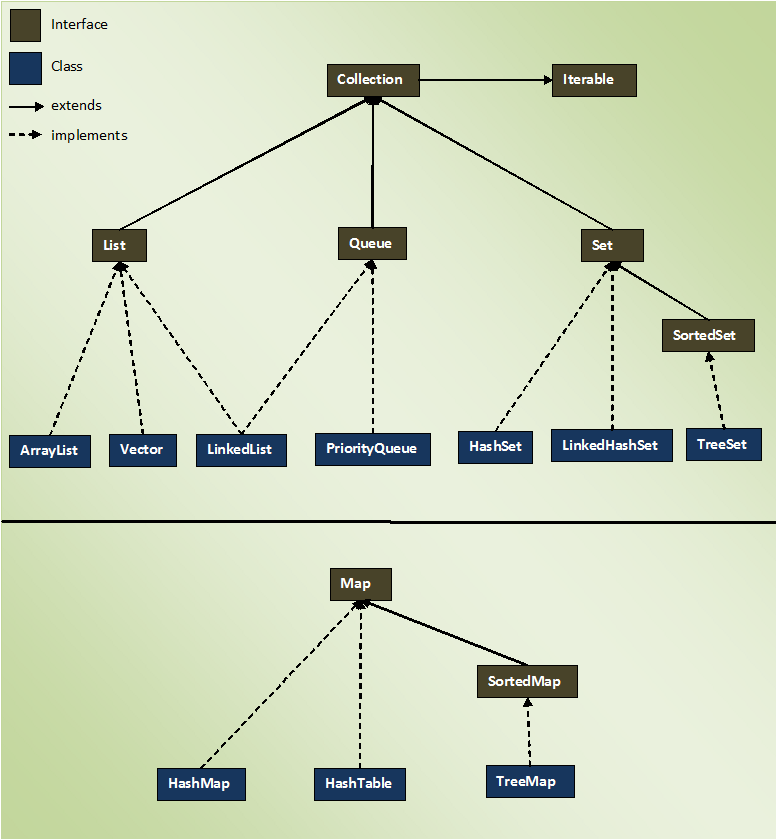
Employee.java – Reusing the existing code – Is called API

Collection (I) – java.util package

List(I)

Set(I)

Map(I)



In Collection API, there are few important interfaces

1. Collection (I) – Used to handle group of objects
2. List (I) – It allows duplicate values, allows null n number of time, maintains insertion order
3. Queue (I) – It implements FIFO (Data Structure) First In First Out
4. Set (I) – It won’t allow duplicates, allow null only one time. Will not maintain the insertion order
5. Map (I) – It store the data in key, value pair. Key should be unique.

List is dynamically growing array. (It’s size changes dynamically depending upon the number of element that we add)

ArrayList can be homo/heterogeneous

Usage of Collection

1. Bank has n number of customer accounts (25000 savings account, 28000 current account, 17000 deposit account – calculate interest for a particular period (3 months)

Boxing – Is the process of converting primitive data to it’s corresponding object representation.

Int a = 20;

Integer intObject = new Integer(a);

In java, there are 8 different primitive data types. In some places like in collection, we can’t use primitive data. We have to convert it into Objects

Usage of Wrapper classes

1. To Convert primitive to it’s corresponding Obj representation.

Tips :

1. What is the difference between final, finally & finalize

* Both final, finally are keywords where finalize is a method.
* Final is non-access (behaviour) modifier. Final variables can’t be re-assigned, final methods can’t be overridden, final class can’t be extended.
* Finally is a block in exception handling – used to do the clean up activity (close resources)
* Finalize is a method which will be called automatically by the JVM during the garbage collection

1. What is mean by auto-boxing & auto-unboxing. When it will happen in JAVA?

* Auto-Boxing is the process of converting primitive value to it’s corresponding object representation
* Unboxing is the reverse of boxing.
* While adding primitive to collection api auto-boxing will happen
* While reading/retrieving primitive data from collection auto-unboxing will happen



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 32 | 57 | 78 | 95 | 67 | 57 |

**package** com.revature;

**import** java.util.Set;

**import** java.util.TreeSet;

**public** **class** TreeSetDemo {

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

Set sortedSet = **new** TreeSet();

sortedSet.add(258);

sortedSet.add(45);

sortedSet.add(791);

sortedSet.add(347);

sortedSet.add(832);

sortedSet.add(743);

sortedSet.add(143);

sortedSet.add(258);

System.***out***.println(sortedSet);

}

}

Output :

[45, 143, 258, 347, 743, 791, 832]