

UNIT-IV

Packages, Exception handling

Unit – IV	Contact Hours = 8 Hours
<p>Packages: Package fundamentals, packages and member access, importing packages, static import.</p> <p>Exception handling: the exception hierarchy, exception handling fundamentals, exception types, uncaught exceptions, using try and catch, multiple catch clauses, catching subclass exceptions, nested try, throw, throws, finally, Java's built-in exceptions, creating your own exception subclasses.</p>	

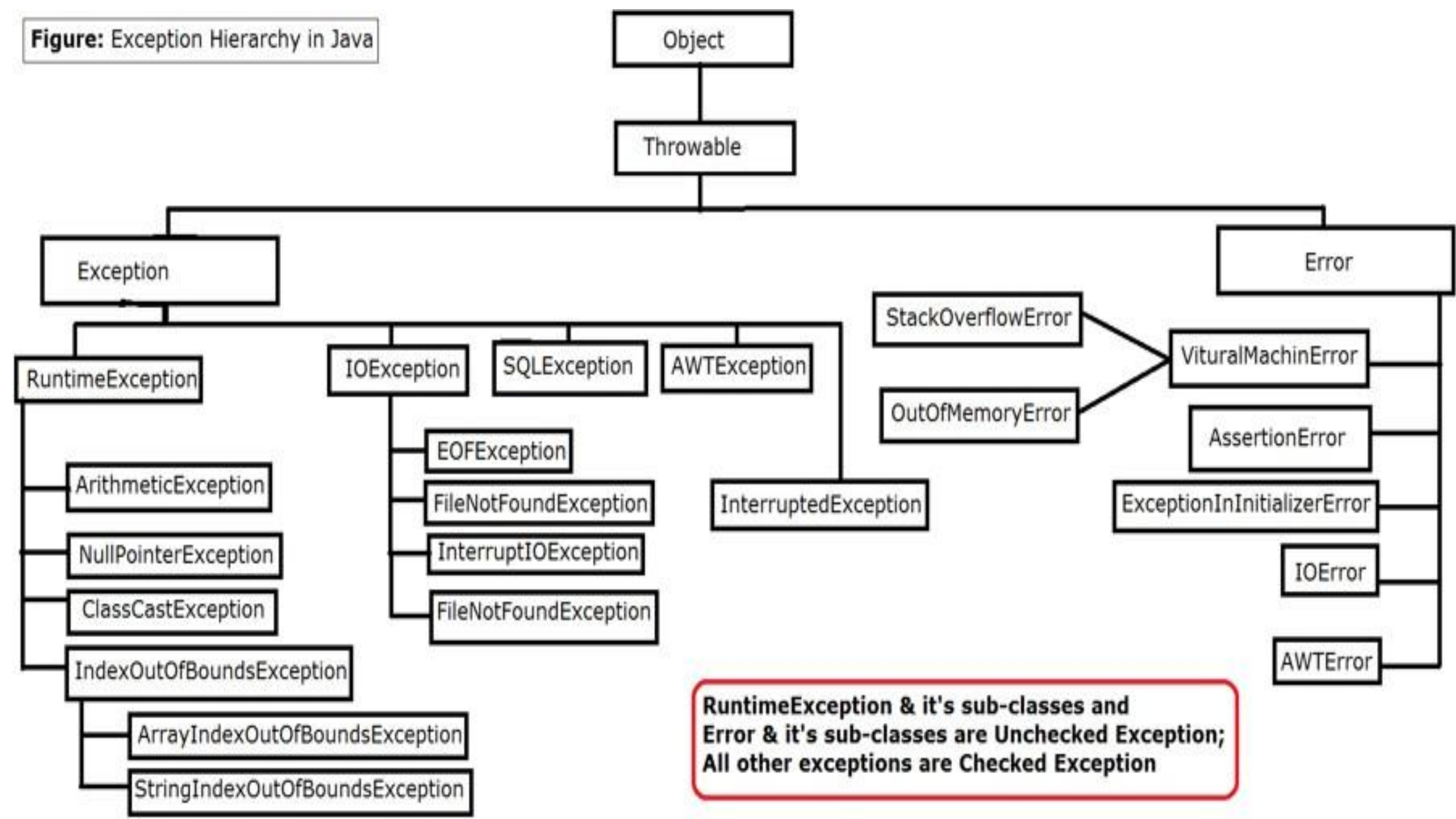
Exception handling:

- The exception hierarchy:
- Exception handling fundamentals: Java exception handling is managed via five keywords: **try**, **catch**, **throw**, **throws**, and **finally**
- Exception types
- uncaught exceptions
- using try and catch
- multiple catch clauses catching subclass exceptions, nested try, throw, throws,
- finally, Java's built-in exceptions,
- creating your own exception subclasses.

The Exception Hierarchy

- All exception classes are derived from a class called **Throwable**.
- When an exception occurs in a program an object of some type of exception class is generated.
- There are 2 direct subclasses of **Throwable**: **exception** and **error**.
- **Exception**: is a Runtime error which will occur during program execution.
- **Error** are related to errors that are beyond the control, which occur in JVM itself.
- **Errors** are not handled by programmer, errors occur due to software or computer fault.
- Exception are of two types :
 - 1) **Checked exception**(detected during compile time)
EX: ClassNotFoundException, IOException, InterruptedException, SQLException, AWTException
 - 2) **Unchecked exception**(detected during run time)
EX: ArithmeticException, ArrayIndexOutOfBoundsException, NullPointerException, NumberFormatException

Figure: Exception Hierarchy in Java



Difference between Exception & Error

Exception	Error
1. Exception occurs because of our programs	1. Error occurs because of lack of system resources.
2. Exceptions are recoverable i.e. programmer can handle them using try-catch block	2. Errors are not recoverable i.e. programmer can handle them to their level
3. Exceptions are of two types : <ul style="list-style-type: none">■ Compile Time Exceptions or Checked Exceptions■ Runtime Exceptions or Unchecked Exceptions	3. Errors are only of one type : <ul style="list-style-type: none">■ Runtime Exceptions or Unchecked Exceptions

1. At the top of the hierarchy is the **Throwable** class, which is the superclass of all **exceptions and errors**. The **Error** class is a subclass of Throwable and represents serious errors that are usually not recoverable, such as `StackOverflowError` and `OutOfMemoryError`.
2. The **Exception** class is another subclass of **Throwable** and is the superclass of all checked exceptions. The **RuntimeException** class is a subclass of Exception and is the superclass of all unchecked exceptions. The remaining subclasses of Exception represent specific checked exceptions, such as **IOException** and **SQLException**.

Exceptions can be categorized into two types: checked exceptions and unchecked exceptions.

Checked Exceptions: Checked exceptions are the exceptions that are checked at **compile-time**. If a method throws a checked exception, then the method must either handle the exception using a try-catch block or declare the exception using the throws keyword. Examples of checked exceptions include `IOException`, `ClassNotFoundException`, `SQLException`, etc.

Unchecked Exceptions: these exceptions occur at **runtime** and are not required to be declared in a method's throws clause. Examples of unchecked exceptions include `NullPointerException`, `ArrayIndexOutOfBoundsException`, `ArithmeticException`, etc.

Exception-Handling Fundamentals

- Java exception handling is managed via five keywords: **try**, **catch**, **throw**, **throws**, and **finally**.
- Program statements that you want to monitor for exceptions are contained within a **try** block. If an exception occurs within the try block, it is thrown.
- Code can catch this exception (using **catch**) and handle it in some rational manner
- System-generated exceptions are automatically thrown by the Java run-time system.
- To manually throw an exception, use the keyword **throw**.
- Any exception that is thrown out of a method must be specified as such by a **throws** clause

1. Using try and catch

- This is **the general form of an exception-handling block**:

```
try {  
    // block of code to monitor for errors  
}  
catch (ExceptionType1 exOb) {  
    // exception handler for ExceptionType1  
}  
  
catch (ExceptionType2 exOb) {  
    // exception handler for ExceptionType2  
}
```

- *ExceptionType* is the type of exception that has occurred
- When an exception is thrown, it is caught by its corresponding **catch** clause
- There can be more than one catch clause associated with a **try** block
- The type of the exception determines which **catch** is executed
- If the exception type specified by a catch matches that of the exception, then that catch clause is executed. All other catch clauses are bypassed
- If no exception is thrown by a try block, no catch clause will be executed and program control will resume after the catch.

Exception can be printed in 3 ways

```
public class Main{  
    public static void main(String[] args) {  
        try{  
            int a=100,b=0,c;  
            c=a/b;  
  
        }  
        catch(ArithmeticException e){  
            e.printStackTrace();    //prints full object i.e exception name ,description and stacktrace  
            System.out.println(e); //prints exception name and description  
        }  
    }  
}
```

```
java.lang.ArithmeticException: / by zero  
    at Main.main(Main.java:14)  
java.lang.ArithmeticException: / by zero  
java.lang.ArithmeticException: / by zero  
/ by zero
```

```
class Example{  
    public static void main(String[] args) {  
        int d,a;  
        try{  
            d=0;  
            a=25/d; //throws an exception object of type ArithmeticException  
            System.out.println("What happens to this statement?");  
        }  
        catch(ArithmeticException e){  
            //e is the reference variable which will hold the exception  
            //object thrown by the catch clause  
            System.out.println("Division by zero error!!");  
        }  
        System.out.println("Executes after the catch statement");  
    }  
}
```

OUTPUT:

Division by zero error!!

Executes after the catch statement

Key points on Exception Handling

- The code which may generate errors need to be contained within **try** block.
- When an exception occurs, the exception is thrown out of the **try** block making the **try** block to terminate. Then that exception will be caught by the **catch** block.
- Hence, the call to **println()** inside the **try** block in the above code is never executed. Once an exception is thrown, program control transfers out of the **try** block into the **catch** block.
- **catch** is not “called,” so execution never “returns” to the **try** block from a **catch**. Thus, the line “What happens to this statement .” is not displayed.
- Once the **catch** statement has executed, program control continues with the next line in the program following the entire **try/catch** mechanism.

Example 2:

```
class Example{  
    public static void main(String[] args) {  
        int d,a;  
        try{  
            d=2;  
            a=25/d; //No exception is thrown  
            System.out.println("What happens to this statement?");  
        }  
        catch(ArithmeticException e){ //control does not enter catch  
            //as no exception is thrown  
            System.out.println("Division by zero error!!");  
        }  
        System.out.println("Executes after the catch statement");  
    }  
}
```

OUTPUT:

What happens to this statement?

Executes after the catch statement

- In the above program, no exception is thrown, hence catch clause is not executed

Ex 3: Exception generated by a method called from within try block

```
class Method{
    static void division() {
        int c = 23/0; //exception is generated here
        //causes an exception which will be thrown to the
        //calling method(main()) as division() method is not
        //handling the exception
        System.out.println("Division by zero error");//won't be executed
    }
}

public class MethodException {
    public static void main(String[] args) {
        try{
            Method.division();
        }
        catch(ArithmeticException e){ //exception is caught here
            System.out.println("Can't divide by zero");
        }
    }
}
```

OUTPUT:

Can't divide by zero

- **division()** is called from within try block, the exception that it generates is caught by the catch in main()
- if **division()** had caught the exception itself, it never would have been passed back to main().

For Example,

```
class Method{
    static void division(){
        try{
            int c = 23/0; //exception is generated here
        }
        catch(ArithmeticException e){ //exception is caught here
            System.out.println("Division by zero error");
        }
    }
}

public class MethodException {
    public static void main(String[] args) {
        try{
            Method.division();
        }
        catch(ArithmeticException e){ //exception is not propagated here
            //as division method is handling it
            System.out.println("Can't divide by zero"); //doesn't execute
        }
    }
}
```

OUTPUT:

Division by zero error

The Consequences of an Uncaught Exception

- If the program does not catch an exception, then it will be caught by the JVM
- But, JVM default exception handler terminates execution and displays an error message followed by list of method calls that cause the exception (referred as stack trace)□
“abnormal termination”
- For example,

```
class Demo{  
    public static void main(String[] args) {  
        int[] a = new int[4];  
        System.out.println("Before exception is generated");  
        a[5]=20; //generates an index out-of-bounds exception  
    }  
}
```

OutPut

Before exception is generated

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds
at Demo.main(Demo.java:5)

Command execution failed.

- As you can see in the output, program terminates as soon as exception occurs
- The type of the exception must match the type specified in a catch. If it does not, the exception would not be caught.
- For example,

```
class Demo{  
    public static void main(String[] args) {  
        int[] a = new int[4];  
        try{  
            System.out.println("Before exception is generated");  
            a[5]=20; //throws an ArrayIndexOutOfBoundsException  
            System.out.println("This won't be displayed");  
        }  
        catch(ArithmeticException e){ //can't catch an array boundary  
                                     //error with an ArithmeticException  
            System.out.println("Index out of bounds");  
        }  
        System.out.println("After the catch");  
    }  
}
```

- The above program tries to catch an array boundary error with a catch for `ArithmeticException`, whereas `ArrayOutOfBoundsException` is generated when the array boundary is overrun.

OUTPUT

Before exception is generated

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds

at Demo.main(Demo.java:5)

Command execution failed.

Exceptions Enable You to Handle Errors Gracefully

- One of the key benefits of exception handling is that it enables the program to respond to an error in a graceful way
- An exception handler can **prevent abrupt program termination**
- For example,

```
public class ExceptionDemo {  
    public static void main(String[] args) {  
        int[] numer = {10,20,30,40,50};  
        int[] denom = {2,0,5,0,3};  
        for(int i=0; i<numer.length; i++){  
            try{  
                System.out.println(numer[i]+"/"+denom[i]+" is "  
                                   +numer[i]/denom[i]);  
            }  
            catch(ArithmeticException ex){  
                System.out.println("Can't divide by zero");  
            }  
        }  
        System.out.println("Program terminates normally");  
    }  
}
```

OUTPUT

```
10/2 is 5  
Can't divide by zero  
30/5 is 6  
Can't divide by zero  
50/3 is 16  
Program terminates normally
```

- In the above program, if a division by zero occurs, an `ArithmeticException` is generated. This exception is handled by reporting the error and then continue with execution
- Thus, attempting to divide by zero does not cause an abrupt termination of the program. Instead, it is handled and program execution is allowed to continue

Using Multiple catch Clauses

- In some cases, more than one exception could be raised by a single piece of code.
- To handle this type of situation, you can specify two or more **catch** clauses, each catching a different type of exception.
- When an exception is thrown, each **catch** statement is inspected in order, and the first one whose type matches that of the exception is executed.
- After one **catch** statement executes, the others are bypassed, and execution continues after the **try/catch** block.

```
public class ExceptionDemo {  
    public static void main(String[] args) {  
        int[] number = {10,20,30,40,50,60,70};  
        int[] denom = {2,0,5,0,3};  
        for(int i=0; i<number.length; i++){  
            try{  
                System.out.println(number[i]+"/"+denom[i]+" is "  
                                   +number[i]/denom[i]);  
            }  
            catch(ArithmeticException e){  
                System.out.println("Can't divide by zero");  
            }  
            catch(ArrayIndexOutOfBoundsException e){  
                System.out.println("No matching element found");  
            }  
        }  
        System.out.println("Program terminates normally");  
    }  
}
```

OutPut

```
10/2 is 5  
Can't divide by zero  
30/5 is 6  
Can't divide by zero  
50/3 is 16  
No matching element found  
No matching element found  
Program terminates normally
```

- Each catch responds only to its own type of exception
- **catch** clauses are checked in the order in which they occur in a program

Catching Subclass Exceptions

- When you use multiple **catch** statements, it is important to remember that exception subclasses must come before any of their superclasses.
- This is because a **catch** statement that uses a superclass will catch exceptions of that type plus any of its subclasses. Thus, a subclass would never be reached if it comes after its superclass.

```
/* This program contains an error.
```

```

A subclass must come before its superclass in
a series of catch statements. If not,
unreachable code will be created and a
compile-time error will result.
```

```
*/
class SuperSubCatch {
    public static void main(String args[]) {
        try {
            int a = 0;
            int b = 42 / a;
        } catch(Exception e) {
```

```
        System.out.println("Generic Exception catch.");
```

```
    }
    /* This catch is never reached because
       ArithmeticException is a subclass of Exception. */
    catch(ArithmeticException e) { // ERROR - unreachable
        System.out.println("This is never reached.");
    }
}
```

- **catch** statement will handle all **Exception**-based errors, including **ArithmeticException**. This means that the second **catch** statement will never execute
- To solve this problem, **Superclass Exception** class must be included as the last catch clause

Nested try blocks

- A **try** statement can be inside the block of another **try**.
- An exception generated within the inner try block that is not caught by a catch associated with that try is propagated to the outer try block

```
public class NestedTry {  
    public static void main(String[] args) {  
        int[] numer = {10,20,30,40,50,60,70};  
        int[] denom = {2,0,5,0,3};  
        try{  
            for(int i=0; i<numer.length; i++){  
                try{  
                    System.out.println(numer[i]+"/"+denom[i]+" is "  
                                         +numer[i]/denom[i]);  
                }  
                catch (ArithmeticException e) {  
                    System.out.println("Can't divide by zero");  
                }  
            }  
        }  
        catch (ArrayIndexOutOfBoundsException e) {  
            System.out.println("No matching element found");  
        }  
        System.out.println("Program terminates normally");  
    }  
}
```

OUTPUT:

```
10/2 is 5
Can't divide by zero
30/5 is 6
Can't divide by zero
50/3 is 16
No matching element found
Program terminates normally
```

- In this example, an exception that can be handled by the inner try, a divide by zero allows the program to continue.
- An array error boundary error is caught by the outer try which terminates the program

Throwing an Exception

- Preceding examples have only been catching exceptions that are thrown by the **Java run-time system**.
- However, it is possible for the program to **throw an exception explicitly**, using the **throw** statement.
- The general form of **throw** is as follows:
throw exceptOb;
- **exceptOb** must be an object of an exception class derived from Throwable
- **throw** throws an object

```
class ExceptDemo{  
    public static void main(String[] args) {  
        try{  
            System.out.println("Before throwing an exception");  
            throw new ArithmeticException();  
            //throws an exception explicitly  
        }  
        catch(ArithmeticException e){  
            System.out.println("Exception caught");  
        }  
        System.out.println("After try/catch block");  
    }  
}
```

OUTPUT

Before throwing an exception

Exception caught

After try/catch block

Rethrowing an Exception

- An exception caught by one catch can be rethrown so that it can be caught by an outer catch

For example,

```
class ReDemo{  
    static void division() {  
        try{  
            int c = 20/0;  
        }  
        catch(ArithmeticException ex){  
            System.out.println("Rethrows the exception");  
            throw ex; //rethrowing the exception which is  
                    //already caught by this catch  
        }  
    }  
}
```

```
public class ReThrow {  
    public static void main(String[] args) {  
        try{  
            ReDemo.division();  
            System.out.println("This will be executed after "  
                               + "returning from the division method");  
        }  
        catch(ArithmeticException e){//rethrown exception is caught here  
            System.out.println("Division by zero");  
        }  
    }  
}
```

```
class Main {  
    public static void main(String args[]) {  
        try {  
            demoproc();  
        } catch (NullPointerException e) {  
            System.out.println("Recought: " + e);  
        }  
    }  
    static void demoproc() {  
        try {  
            throw new NullPointerException("demo");  
        } catch (NullPointerException e) {  
            System.out.println("Caught inside demoproc.");  
            throw e; // rethrow the exception  
        }  
    }  
} //end main
```

Caught inside demoproc.

Recought: java.lang.NullPointerException: demo

Using finally

- In Java, the **finally block** is used in combination with a **try and/or catch block** to provide a **final piece of code that is always executed**, regardless of whether an exception is thrown or not.
- The finally block is optional, but when it is used, it will be executed after any catch block(s), and will always be executed, even if an exception is thrown or the program exits abnormally.
- The general form of try/catch that includes finally is

```
try
{
    // statements to try
}
catch(Exception e)
{
    // actions that occur if exception was thrown
}
finally
{
    // actions that occur whether catch block executed or not
}
```

- The **finally block** is typically used to **close resources that were opened in the try block**, such as files, network connections, and database connections.
- This ensures that the resources are properly released, regardless of whether an exception was thrown or not.

```

class ReDemo{
    static void division() {
        try{
            int c = 20/0;
        }
        catch(ArithmeticException ex){
            System.out.println("Catches an exception");
            return; //even if the method returns finally block will execute
        }
        finally{
            System.out.println("This executes even when the method returns");
        }
    }
}

public class ReThrow {
    public static void main(String[] args) {
        try{
            ReDemo.division();
            System.out.println("This will be executed after "
                               + "returning from the division method");
        }
        catch(ArithmeticException e){//rethrown exception is caught here
            System.out.println("Division by zero");
        }
    }
}

```

OUTPUT

Catches an exception

This executes even when the method returns

This will be executed after returning from the division method

- In the above example, finally block is executed even after the method returns


```
class ReDemo{
    static void division() {
        try{
            int c = 20/0;
        }
        catch(ArithmeticException ex){
            System.out.println("Catches an exception");
            return; //even if the method returns finally block will execute
        }
        finally{
            System.out.println("This executes even when the method returns");
        }
    }
}

public class ReThrow {
    public static void main(String[] args) {
        try{
            ReDemo.division();
            System.out.println("This will be executed after "
                               + "returning from the division method");
        }
        catch(ArithmeticException e){//rethrown exception is caught here
            System.out.println("Division by zero");
        }
    }
}
```

Using throws

- In Java, the **throws** keyword is used in a **method signature** to declare that the **method may throw one or more exceptions**.
- When a method is declared with a throws clause, it is indicating that it is possible for the method to throw an exception during its execution, and that it is up to the calling code to handle that exception.
- A throws clause lists the types of exceptions that a method might throw.
- The general form a method that includes a **throws** clause:

```
type method-name(parameter-list) throws exception-list    public void myMethod() throws ExceptionType1, ExceptionType2, ... {  
{                                                         // Method code that may throw one of the declared exceptions  
// body of method                                         }  
}
```

- Here, *exception-list* is a comma-separated list of the exceptions that a method can throw


```
class ThrowsDemo
{
    public static void main(String[] args) throws InterruptedException
    {
        Thread.sleep(10000);
        System.out.println("Excecuted successfully");
    }
}
```

/*In the above program, we are getting compile time error because there is a chance of exception if the main thread is going to sleep, other threads get the chance to execute main() method which will cause InterruptedException. */

Important Points about throws:

- In the above example, **main()** throws **InterruptedException** but does not handle it
- **main()** must define a **try/catch** statement that catches this exception.

For example, consider the following method that opens a file and reads its contents:

```
public String readFile(String fileName) throws FileNotFoundException, IOException {  
    FileInputStream fis = new FileInputStream(fileName);  
    StringBuilder sb = new StringBuilder();  
    int ch;  
    while((ch = fis.read()) != -1) {  
        sb.append((char) ch);  
    }  
    fis.close();  
    return sb.toString();  
}  
  
public void myMethod() {  
    try {  
        String contents = readFile("file.txt");  
        System.out.println(contents);  
    } catch(FileNotFoundException e) {  
        // Handle the FileNotFoundException  
    } catch(IOException e) {  
        // Handle the IOException  
    }  
}
```

Difference between throw and throws key word in java

Throw:

- "Throw" is a keyword used to explicitly throw an exception in a program.
- When you encounter an exceptional situation or error in your code, you can use the "throw" keyword to throw an exception.

EX

```
throw new SomeException("This is an example exception");
```

Throws:

- "Throws" is a keyword used in the method declaration to indicate that the method might throw one or more types of exceptions.
- It is used in the method signature to specify the exceptions that the method can throw, but it doesn't actually throw the exceptions itself.

Example:

```
public void someMethod() throws IOException, InterruptedException {  
    // Method code that might throw IOException or InterruptedException  
}
```

Java's Built-in Exceptions

- java.lang is implicitly imported into all Java programs

Exception	Meaning
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
EnumConstantNotPresentException	An attempt is made to use an undefined enumeration value.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBoundsException	Attempt to index outside the bounds of a string.
TypeNotPresentException	Type not found.
UnsupportedOperationException	An unsupported operation was encountered.

TABLE 10-1 Java's Unchecked **RuntimeException** Subclasses Defined in **java.lang**

Exception	Meaning
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does not implement the Cloneable interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.

TABLE 10-2 Java's Checked Exceptions Defined in **java.lang**

Creating your own exception subclasses.

- In Java, we can create our own exceptions that are derived classes of the Exception class.
- Creating our own Exception is known as custom exception or user-defined exception.

Two commonly used **Exception** constructors are:

`Exception();`

`Exception(String msg);`

- The first form creates an exception that has no description.
- The second form lets you specify a description of the exception

Design a custom(user defined exception) exception for validating age attribute of citizen for voting.

```
import java.util.Scanner;

class InvalidAgeException extends Exception {
    InvalidAgeException(String s){
        super(s);
    }
}

public class CustExHandle {

    static void validate(int age)throws InvalidAgeException{
        if(age<18)
            throw new InvalidAgeException("Minor: Cannot vote");
        else
            System.out.println("Can vote");
    }
}
```

- `public static void main(String[] args) {`
- `int ageValid;`
- `try{`
- `Scanner sc = new Scanner(System.in);`
- `ageValid = sc.nextInt();`
- `validate(ageValid);`
- `}catch(Exception ex){`
- `System.out.println("Exception occurred: "+ex);`
- `}`
- `}`

Build a java program that accepts an integer from the user,if number entered is odd the program throws an user defined exception name OddNumber Exception.
Define the class specification for the OddNumber Exception class

Step 1: Define the OddNumberException class

The OddNumberException class will extend the Exception class and provide a constructor to initialize the exception with a custom message

```
// OddNumberException.java
public class OddNumberException extends Exception {
    // Constructor that accepts a custom message
    public OddNumberException(String message) {
        super(message);
    }
}
```

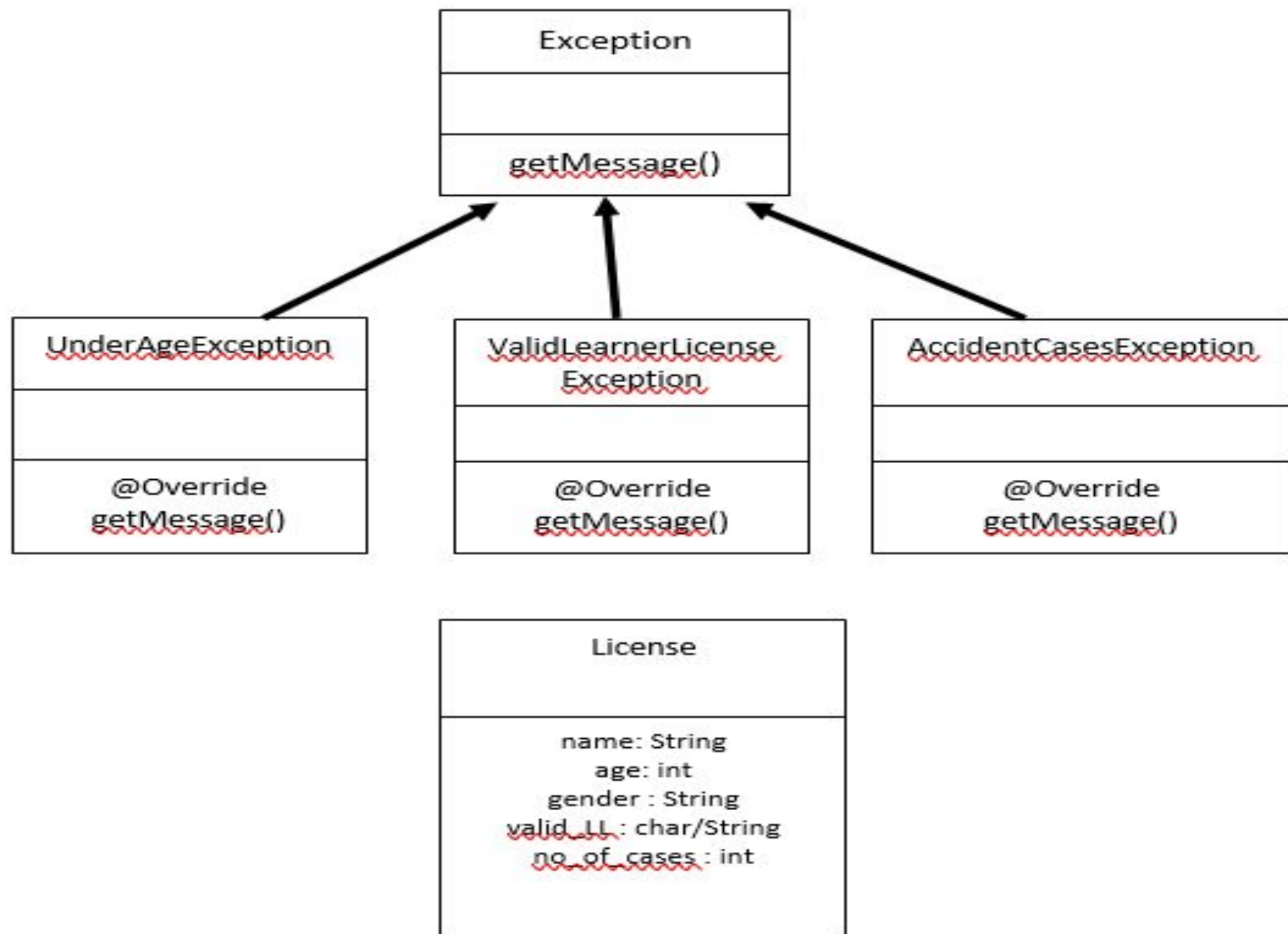
Step 2: Create the main program

The main program will prompt the user to enter an integer, check if it's odd, and throw the OddNumberException if it is.

```
try {  
    // Prompt the user to enter an integer  
    System.out.print("Enter an integer: ");  
    int number = scanner.nextInt();  
  
    // Check if the number is odd  
    if (number % 2 != 0) {  
        // Throw OddNumberException if the number is odd  
        throw new OddNumberException("The number entered is odd: " + number);  
    }  
  
    // Print the number if it's not odd  
    System.out.println("The number entered is even: " + number);  
  
} catch (OddNumberException e) {  
    // Handle the custom exception  
    System.out.println("Caught an exception: " + e.getMessage());  
} catch (Exception e) {  
    // Handle other exceptions like input mismatch  
    System.out.println("An error occurred: " + e.getMessage());  
} finally {  
    scanner.close(); // Close the scanner to prevent resource leak  
}  
}
```

8. Write a program to demonstrate the implementation of customized exception handling.

- Assume that you have received a request from the transport authority for automating the task of issuing the permanent license for two wheelers. The mandatory condition to issue the license are: 1) the applicant must over 18 years of age and 2) holder of a valid learner's license and 3) no accident cases in the last one year.
- Write a Java program that reads user details as required (use the Scanner class). Create user defined exceptions to check for the three conditions imposed by the transport authority. Based on the inputs entered by the user, decide and display whether or not a license has to be issued or an error message as defined by the user exception.



```
import java.util.Scanner;

class UnderAgeException extends Exception{

    UnderAgeException(String s){

        super(s) ;

    }

    @Override

    public String toString(){

        return "Sorry. You are too young for the license";

    }

}
```

```
class ValidLLR extends Exception{  
    ValidLLR(String s){  
        super(s);  
    }  
    @Override  
    public String toString(){  
        return "Sorry. You do not hold a valid LLR";  
    }  
}
```



```
class NumAccidents extends Exception{
    NumAccidents(String s){
        super(s);
    }
    @Override
    public String toString(){
        return "Sorry. There are accidents in last one year";
    }
}
```

```
class License{  
    String name;  
    int age, no_of_cases;  
    char gender;  
    char validLLR;  
    void readData(){  
        Scanner in=new Scanner(System.in);  
        System.out.println("Enter the name: ");  
        name = in.nextLine();  
        System.out.println("Enter the age: ");  
        age = in.nextInt();  
        System.out.println("Enter the gender: ");  
        gender = in.next().charAt(0);  
        System.out.println("Do you have Valid LLR (Y/N)? ");  
        validLLR = in.next().charAt(0);  
        System.out.println("How many number of cases in past one year? ");  
        no_of_cases = in.nextInt();  
    }  
}
```

```
public class TW8 {  
    public static void main(String[] args) {  
        License applicant =new License();  
        applicant.readData();  
        validateApplicant(applicant);  
    }  
    static void validateApplicant(License a){  
        try{  
            if(a.age<18)  
                throw new UnderAgeException("Underageexception:");  
            if(a.validLLR!='Y')  
                throw new ValidLLR("ValidLLRexception:");  
            if(a.no_of_cases>0)  
                throw new NumAccidents("Numberofaccidentsexception:");  
            System.out.println("Congrats!! Your license is being posted");  
        }  
    }  
}
```

```
catch (UnderAgeException e) {  
    System.out.println(e.getMessage()+e) ;  
}  
catch (ValidLLR e) {  
    System.out.println(e.getMessage()+e) ;  
}  
catch (NumAccidents e) {  
    System.out.println(e.getMessage()+e) ;  
}  
catch (Exception e) {  
    System.out.println(e.getMessage()+e) ;  
}  
  
}  
  
}
```

PACKAGES

- Package is a way of organizing related classes and interfaces into a single unit.
- Packages help in avoiding naming conflicts, make the code more modular, and improve the overall organization of a large codebase.
- **Purpose of Packages:**
- **Organization:** Packages help organize related classes and interfaces into a single directory structure.
- **Encapsulation:** Packages provide a level of encapsulation by restricting the access to classes and interfaces outside the package.
i.e Classes defined within a package can be made private to that package and not accessible by outside code

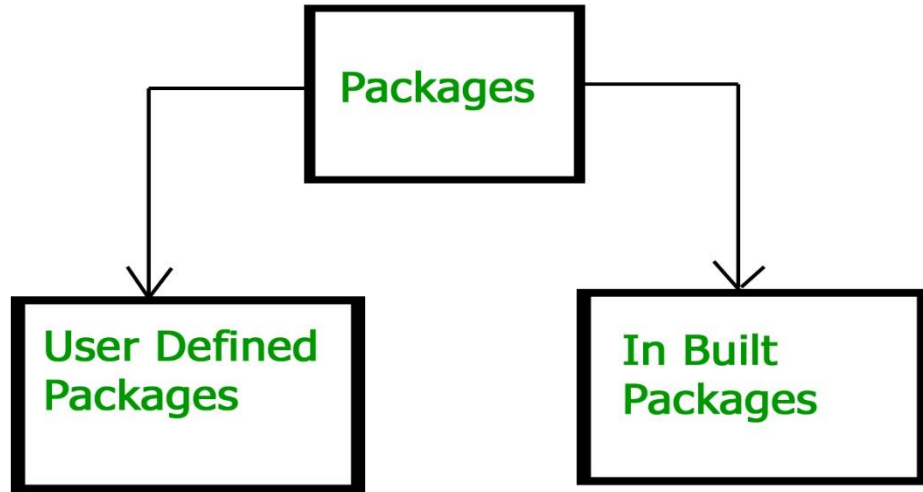
Defining a package

- ⌚ All classes in Java belong to some package.
- ⌚ When no package has been explicitly specified, the default or global package is used.
- ⌚ The default package has no name.
- ⌚ To create a package, use the package statement which is located at the top of a Java source file.
- ⌚ The **general form of the package statement is**.
 - **`package pkg;`**
- ⌚ For example:
 - **`package mypack;`**
- ⌚ Hierarchy of packages can be created by separating each package name from the one above it by use of a period.
- ⌚ The **general form of a multi leveled package statement is**
`package pack1.pack2.pack3.....packN;`

Built-in Packages

These packages consist of a large number of classes which are a part of Java **API**. Some of the commonly used built-in packages are:

- 1) **java.lang**: Contains language support classes(e.g classed which defines primitive data types, math operations). This package is automatically imported.
- 2) **java.io**: Contains classed for supporting input / output operations.
- 3) **java.util**: Contains utility classes which implement data structures like Linked List, Dictionary and support ; for Date / Time operations.
- 4) **java.applet**: Contains classes for creating Applets.
- 5) **java.awt**: Contain classes for implementing the components for graphical user interfaces (like button , ;menus etc).
- 6) **java.net**: Contain classes for supporting networking operations.



```
package myPackage;
public class MyClass {
    public void getNames(String s)
    {
        System.out.println(s);
    }
}
```

User-defined packages

These are the packages that are defined by the user. First we create a directory **myPackage** (name should be same as the name of the package). Then create the **MyClass** inside the directory with the first statement being the **package names**

Importing Java's Standard Packages

<i>Package</i>	<i>Provides classes for</i>
<code>java.applet</code>	programs (applets) that can be run from a web page
<code>java.awt</code>	Abstract Windowing Toolkit (AWT) – basic graphical user interface (GUI) components such as windows, fonts, colours, events, buttons and scroll bars
<code>java.io</code>	low-level input/output – for example, reading data from files or displaying on screen
<code>java.lang</code>	basic classes for the language – automatically imported and used by all Java programs
<code>java.net</code>	communication across a network, using clients, servers, sockets and URLs
<code>javax.swing</code>	creation of more sophisticated, platform-independent GUIs, building on the AWT capabilities
<code>java.util</code>	general utility classes, especially collection classes (data structures)

Finding Packages and CLASSPATH

- The Java run-time system know where to look for packages that is created by doing following things:
 1. First, by default, the run-time system uses the current working directory as its starting point. If the package is in a subdirectory of the current directory, it will be found
 2. Second, a directory path or paths can be specified by setting the CLASSPATH environmental variable.
 3. Third, the -classpath option can be used java and javac to specify the path to the classes

```
package BookDemo1;



class Book {
    private String title;
    private String author;
    private int year;

    Book(String t, String a, int y){
        title = t;
        author = a;
        year = y;
    }

    void show() {
        System.out.println(title);
        System.out.println(author);
        System.out.println(year);
    }
}
```

```
public class BookDemo1 {
    public static void main(String[] args) {
        Book[] b = new Book[5];
        b[0] = new Book("The Art pf Computer Programming", "Knuth", 1973);
        b[1] = new Book("Java Fundamentals", "Schildt", 2013);
        b[2] = new Book("Thirteen at Dinner", "Christie", 1933);
        b[3] = new Book("Red storm Rising", "Clancy", 1986);
        b[4] = new Book("On the Road", "Kerouac", 1955);
        for(int i=0; i<b.length; i++){
            b[i].show();
            System.out.println();
        }
    }
}
```

Accessing a Package

-  In the above example, if **Book** and **BookDemo1** were in different packages, then **Book** would not have been accessible to other package
-  Following changes to be made to make **Book** available to other packages
 1. **Book** needs to be declared public
 2. Its constructor must be made public
 3. show() method needs to be public


```

package backpack;
public class Book {
    protected String title;
    protected String author;
    protected int date;

    public Book(String t, String a, int d) {
        title = t;
        author = a;
        date = d;
    }

    public void show() {
        System.out.println(title);
        System.out.println(author);
        System.out.println(date);
    }
}

```

```

package mypack;
public class BookDemo {
    public static void main(String[] args) {
        backpack.Book[] b = new backpack.Book[5];
        b[0] = new backpack.Book("The Art of Computer Programming", "Knuth", 1973);
        b[1] = new backpack.Book("Java Fundamentals", "Schildt", 2013);
        b[2] = new backpack.Book("Thirteen at Dinner", "Christie", 1933);
        b[3] = new backpack.Book("Red storm Rising", "Clancy", 1986);
        b[4] = new backpack.Book("On the Road", "Kerouac", 1955);
        for(int i=0; i<b.length; i++){
            b[i].show();
            System.out.println();
        }
    }
}

```



To access **Book**, we must fully qualify its name to include its full package specification



Without this specification, **Book** would not be found



Syntax is,

packageName.className;

Packages and Member Access

	default	private	protected	public
same class	yes	yes	yes	yes
same package subclass	yes	no	yes	yes
same package non-subclass	yes	no	yes	yes
different package subclass	no	no	yes	yes
different package non-subclass	no	no	no	yes

Importing Packages

- ⌚ Qualifying the name of the class with name of its package is tedious and tiresome.
- ⌚ import statement can be used to bring one or more members of a package into the scope
- ⌚ The **general form of the import statement** is
import pkg.className;
- ⌚ **pkg** is the name of the package with its full path, **className** is the name of the class being imported. '*' is used to import the entire contents of a package

Ex:

```
import backpack.Book; //Book class is imported from backpack
import backpack.*; //all of the classes in backpack is imported
```

```
package mypack;
import bookpack.Book;
public class BookDemo {
    public static void main(String[] args) {
        Book[] b = new Book[5]; //no need of qualify it with its package
                                //as import statement is used
        b[0] = new Book("The Art pf Computer Programming", "Knuth", 1973);
        b[1] = new Book("Java Fundamentals", "Schildt", 2013);
        b[2] = new Book("Thirteen at Dinner", "Christie", 1933);
        b[3] = new Book("Red storm Rising", "Clancy", 1986);
        b[4] = new Book("On the Road", "Kerouac", 1955);
        for(int i=0; i<b.length; i++){
            b[i].show();
            System.out.println();
        }
    }
}
```



No longer need to qualify Book with its package name

Static Import

- import statement can be used to import static members of a class/interface by following import with the keyword static
- Two general forms are
 1. `import static pkg.typeName.staticMemberName;`
 2. `import static pkg.typeName.*; //it imports all static members`

Ex:

```
import static java.lang.Math.sqrt;
```

```
import static java.lang.Math.pow;
```

OR

```
import static java.lang.Math.*;
```

```
import static java.lang.Math.*;
public class StaticDemo {
    public static void main(String[] args) {
        //no need to use Math classname to import the following methods
        System.out.println("The squareroot of the number is"+sqrt(25));
        System.out.println("The exponent value is "+pow(2, 4));
        System.out.println("The absolute value is "+abs(2.6));
    }
}
```

Advantages of using Packages in Java:

- 1. Modularity:** Packages provide a modular structure to the program. It helps in grouping related classes and interfaces, which makes the program easier to understand and maintain.
- 2. Access Control:** Protected and default have package level access control. A protected member is accessible by classes in the same package and its subclasses. A default member (without any access specifier) is accessible by classes in the same package only.
- 3. Namespace Management:** Packages provide a namespace management system that prevents naming conflicts between the classes and interfaces. For example there can be **two classes with name Employee** in two packages, `college.staff.cse.Employee` and `college.staff.ee.Employee`
- 4. Reusability:** Packages provide a way to reuse the code across different projects. The code can be packaged as a library and used across different programs.