Exception Handling

- when we write programs as part of an application, we may have to visualize the

challenges that can disrupt the normal flow of execution of the code.

- Once we know what are the different situations that can disrupt the flow of execution, we can take preventive measures to overcome these disruptions.
- In java, this mechanism comes in the form of Exception Handling.

In procedural programming, it is the responsibility of the programmer to ensure that the programs are error-free in all aspects
Errors have to be checked and handled
manually by using some error codes
But this kind of programming was very cumbersome and led to spaghetti code
Java provides an excellent mechanism
for handling runtime errors

- An exception is an event that occurs during the execution of a program that disrupts the

normal flow of instructions

- The ability of a program to intercept run-time errors, take corrective measures and continue execution is referred to as exception handling

There are various situations when an exception could occur:

- Attempting to access a file that does not exist
- Inserting an element into an array at a position that is not in its bounds
- Performing some mathematical operation that is not permitted
- Declaring an array using negative values

Uncaught Exceptions

```
class    Demo {

public static void main(String args[]) {

int x = 0;

int y = 50/x;  System.out.println("y    = " +y);

}

}
```

Although this program will compile, but when you execute it, the Java run-time-system will generate an exception and displays the following output on the console :

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

at Demo.main(Demo.java:4)
```

Exception Handling Techniques

- There are several built-in exception classes that are used to handle the very fundamental errors that may occur in your programs
- You can create your own exceptions also by extending the **Exception** class
- These are called user-defined exceptions, and will be used in situations that are unique to your applications

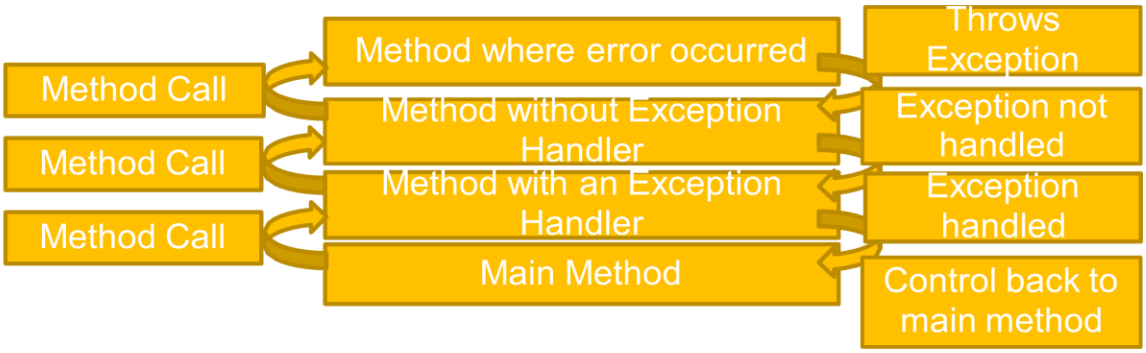
Handling Runtime Exceptions

- Whenever an exception occurs in a program, an object representing that exception is created and thrown in the method in which the exception occurred
- Either you can handle the exception, or ignore it
- In the latter case, the exception is handled by the Java run-time-system and the program terminates
- However, handling the exceptions will allow you to fix it, and prevent the program from terminating abnormally

Advantages - Exceptions

1. Separating Error-Handling Code from "Regular" Code

2. Propagating Errors Up the Call Stack



3. Grouping and Differentiating Error Types

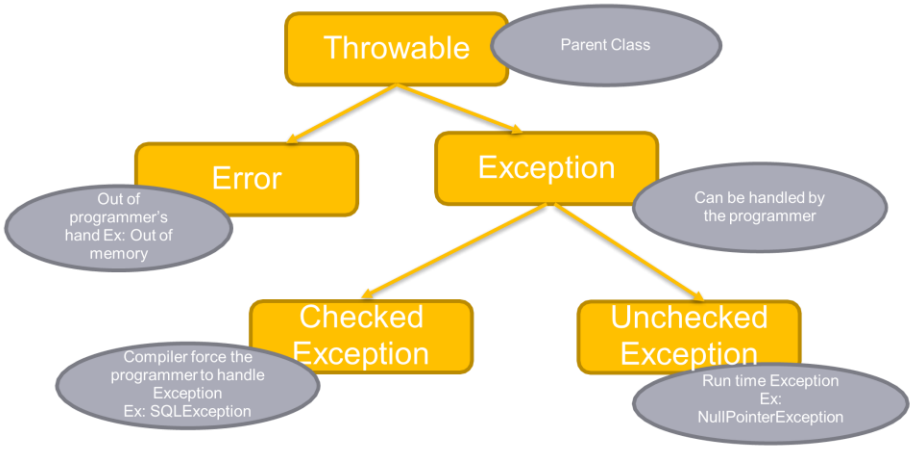
Exception Handling Keywords

Java's exception handling is managed using the following keywords: **try, catch, throw, throws and finally.**

```
try {
// code comes here
}
catch(TypeofException          obj) {
//handle the exception
}
finally  {
//code to be executed before the program ends
}
```

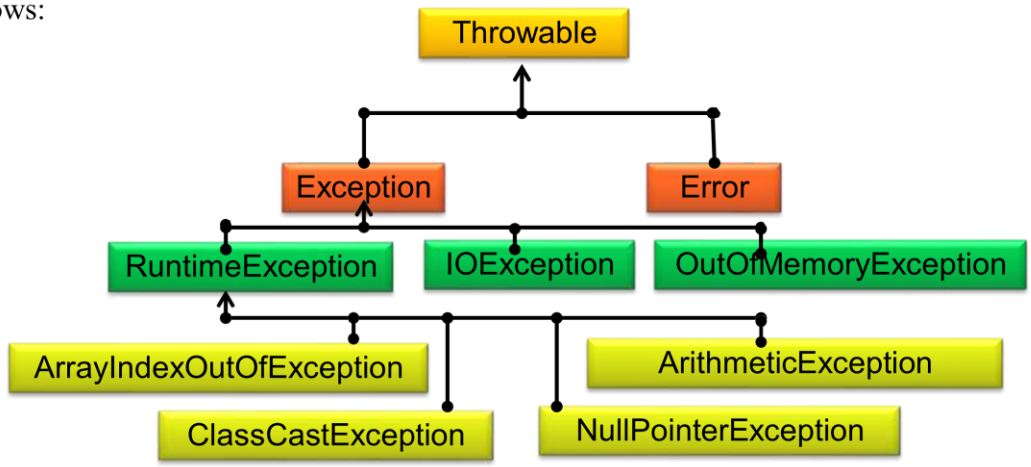
Exception_Handling – Types

Exception in a Nutshell



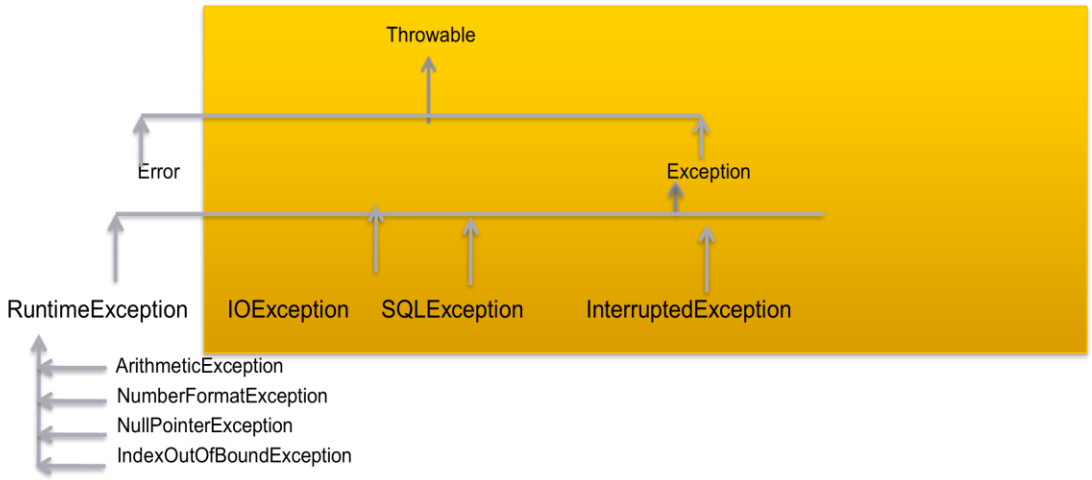
Exception Types

Exceptions are implemented in Java through a number of classes. The exception hierarchy is as follows:



Checked and Unchecked Exceptions

Checked Exceptions



Checked Exception (Contd.).

- A checked exception is an exception that usually happens due touser error or it is an error situation that cannot be foreseen by the programmer
- A checked exception must be handled using a try or catch or at least declared to be thrown using throws clause. Non compliance of this rule results in a compilation error

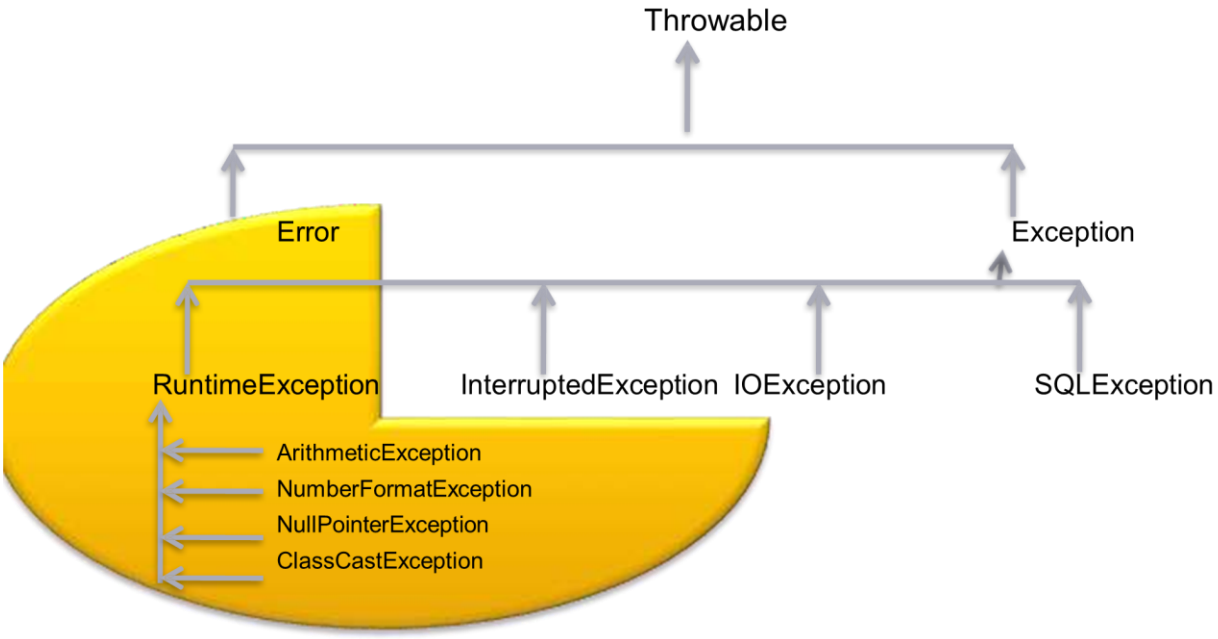
Ex: FileNotFoundException

- If you try to open a file using

FileInputStream fx = new FileInputStream("A1.txt");

- During execution, the system will throw a FileNotFoundException, if the file A1.txt is not located, which may be beyond the control of a programmer

Unchecked Exception



Unchecked Exceptions (Contd.).

- An unchecked exception is an exception, which could have been avoided by the programmer
- The class RuntimeException and all its subclasses are categorized as Unchecked Exceptions
- If there is any chance of an unchecked exception occuring in the code, it is ignored during compilation

---

Error

- Error is not considered as an Exception
- Errors are problems that arise beyond the control of the programmer or the user
- A programmer can rarely do anything about an Error that occurs during the execution of a program
- This is the precise reason Errors are typically ignored in the code
- Errors are also ignored by the compiler
- Ex : Stack Overflow

Exception Handling – Try-Catch Block

Try-Catch Block

Multiple Catch Block

Nested Try Block

Try-Catch Block

- Any part of the code that can generate an error should be put in the **try** block
- Any error should be handled in the c**atch** block defined by the **catch** clause
- This block is also called the **catch block**, or the **exception handler**
- The corrective action to handle the exception should be put in the **catch** block

How to Handle exceptions?

```
class ExceptDemo{
public static void main(String args[]){  int x, a;
try{
x = 0;
a = 22 / x;
System.out.println("This will be bypassed.");
}
catch (ArithmeticException e){  System.out.println("Division by zero.");
```

```
}

System.out.println("After catch statement.");

}

}
```

Multiple Catch Statements

- A single block of code can raise more than one exception
- You can specify two or more **catch** clauses, each catching a different type of execution
- 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

```
class MultiCatch{

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

int l = args.length;  System.out.println("l = " +l);  int b = 42 / l;

int arr[] = { 1 };  arr[22] = 99;

}

catch(ArithmeticException e){  System.out.println("Divide by 0: "+ e);

}

catch(ArrayIndexOutOfBoundsException e){  System.out.println("Array index oob: "+e);

}

System.out.println("After try/catch blocks.");

}

}
```

Multiple Catch Statements involving Exception Superclasses & Subclasses

- When you use multiple catch statements, it is important to remember that exception subclasses must come before any of their exception superclasses
- This is because a catch statement that uses a superclass will catch exceptions of that type as well as exceptions of its subclasses
- Thus, a subclass exception would never be reached if it came after its superclass that manifests as an **unreachable code error**

Nested try Statements

- The **try** statement can be nested
- If an inner **try** statement does not have a **catch** handler for a particular exception, the outer

block's catch handler will handle the exception

- This continues until one of the **catch** statement succeeds, or until all of the nested **try**

statements are exhausted

- If no catch statement matches, then the Java runtime system will handle the exception

Syntax

```
try
{
statement 1;
statement 2;
try
{
statement 1;
statement 2;
}
catch(Exception e)
{
```

```
}
}
catch(Exception e)
{
}
```

Example for nested try

```
class Nested_Try{
public static void main(String args[]){  try{
try{
System.out.println("Arithmetic Division");  int b =39/0;
} catch(ArithmeticException e){  System.out.println(e);
}
try{
int a[]=new int[5];  System.out.println("Accessing Array Elements");
a[5]=4;
} catch(ArrayIndexOutOfBoundsException e)  System.out.println(e);
}
System.out.println ("Inside Parent try");
} catch(Exception e) {
System.out.println("Exception caught");
}
System.out.println("Outside Parent try");
}
}
```

Finally Clause

- When an exception occurs, the execution of the program takes a non-linear path, and could bypass certain statements
- A program establishes a connection with a database, and an exception occurs
- The program terminates, but the connection is still open
- To close the connection, **finally** block should be used

The finally block is guaranteed to execute in all circumstances

```
import java.io.*;
class FinallyDemo{
static void funcA() throws FileNotFoundException
{
try{
System.out.println("inside funcA( )");  throw new FileNotFoundException( );
}
finally{
System.out.println
("inside finally of funA( )");
}
}
static void funcB(){
try{
System.out.println("inside funcB( )");
}
```

```
finally{
System.out.println
("inside finally of funB( )");
}
}
}
static void funcC() {  try{
System.out.println("inside funcC( )");
}
finally{
System.out.println
("inside finally of funcC( )");
}
}
public static void main(String args[]){
try{
funcA();
}
catch (Exception e){
System.out.println("Exception caught");
}
funcB( );
funcC( );
}
}
```

Significance of printStackTrace() method

- We can use the *printStackTrace()* method to print the program's execution stack
- This method is used for debugging

```
import java.io.*;
class PrintStackExample {
public static void main(String args[]) {
try {
m1();
}
catch(IOException e) {  e.printStackTrace();
}
}
```

Throw Clause

- System-generated exceptions are thrown automatically

- At times you may want to throw the exceptions explicitly which can be done using the **throw** keyword
- The exception-handler is also in the same block
- The general form of throw is:
    - throw **ThrowableInstance**
    - Here, **ThrowableInstance** must be an object of type **Throwable**, or a subclass of **Throwable**

```
class ThrowDemo{
public static void main(String args[]) {  try {
int age=Integer.parseInt(args[0]);
if(age < 18)
throw new ArithmeticException();
else
if(age >=60)
throw new ArithmeticException("Employee is retired");
}
catch(ArithmeticException e) {
System.out.println(e);
}
System.out.println("After Catch");
}
```

User Defined Exceptions

- Java provides extensive set of in-built exceptions
- But there may be cases where we may have to define our own exceptions which are

application specific

For ex: If we have are creating an application for handling the database of eligible voters, the age should be greater than or equal to 18

In this case, we can create a user defined exception, which will be thrown in case the age

entered is less than 18

- While creating user defined exceptions, the following aspects have to be taken care :
- defined exception class should extend from the Exception class and its subclass.
- If we want to display meaningful information about the exception, we should override the toString() method

Example:

```
class MyException extends Exception { public MyException() {

System.out.println("User defined Exception thrown");

}

public String toString() {

return "MyException Object : Age cannot be < 18 " ;

}

}

class MyExceptionDemo{ static int flag=0;

public static void main(String args[]) {

try {

int age=Integer.parseInt(args[0]);

if(age < 18)

throw new MyException();

}

catch(ArrayIndexOutOfBoundsException e) {

flag=1;
```

```
System.out.println("Exception : "+ e);
}
catch(NumberFormatException e) {  flag=1;
System.out.println("Exception : "+ e);
}
catch (MyExceptionClass e) {  flag=1;
System.out.println("Exception : "+ e);
}
if(flag==0)
System.out.println("Everything is fine");
}
}
```

Complete the code to print the message “Invalid Input” class InvalidInputException extends Exception {

```
InvalidInputException(String s) {
//Insert the code so that null is not printed
}
}
class Input {
void method() throws InvalidInputException {
throw new InvalidInputException("Invalid Input");
}
}
class TestInput {
public static void main(String[] args){
```

```
try {
new Input().method();
}
catch(InvalidInputException iie) {  System.out.println(iie.getMessage());
}
}
}
```

Throws Clause

- Sometimes, a method is capable of causing an exception that it does not handle
- Then, it must specify this behavior so that callers of the method can guard themselves

against that exception

- While declaring such methods, you have to specify what type of exception it may throw by using the **throws** keyword
- A **throws** clause specifies a comma-separated list of exception types that a method might throw:

type method-name(parameter list) throws exception-list

Implementing throws

```
import java.io.*;
class ThrowsDemo{
static void throwOne() throws FileNotFoundException{  System.out.println("Inside throwOne.");
throw new FileNotFoundException();
}
public static void main(String args[]) {  try{
throwOne();
```

```
}
catch (FileNotFoundException e){   System.out.println("Caught " + e);
}
}
}
```

Rule governing overriding method with throws

- The overriding method must NOT throw checked exceptions that are new or broader than

those declared by the overridden method

For eg : A method that declares(throws) an SQLException cannot be overriden by a method that declares an IOException, Exception or any other exception unless it is a subclass of SQLException

- In other words, if a method declares to throw a given exception, the overriding method in a

subclass can only declare to throw the same exception or its subclass

This rule does not apply for unchecked exceptions

Final Keyword

- The **final** keyword used in context of behavioral restriction on:
    - variables
    - methods
    - classes
- Using final on variables to make them behave as constants which we have seen in earlier module.
- When a variable is made final – it can be initialized only once either by
    - Declaration and initialization

final int x=10;

    - Using constructor

System allows you to set the value only once; after which it can't be changed

The Role of the Keyword final in Inheritance

- The **final** keyword has two important uses in the context of a class hierarchy. These

uses are highlighted as follows:

- Using final to Prevent Overriding
    - While method overriding is one of the most powerful feature of object oriented design, there may be times when you will want to prevent certain critical methods in a superclass from being overridden by its subclasses.
    - Rather, you would want the subclasses to use the methods as they are defined

in the superclass.

    - This can be achieved by declaring such critical methods as final.

The Role of the Keyword final in Inheritance (Contd

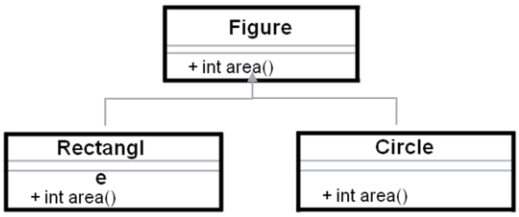
Using final to Prevent Inheritance

- Sometimes you will want to prevent a class from being inherited.
- This can be achieved by preceding the class declaration with final.
- Declaring a class as final implicitly declares all of its methods as final too.
- It is illegal to declare a class as both abstract and final since an abstract class is incomplete by itself and relies upon its subclasses to provide concrete and complete implementations.

Abstract classes

Abstract Classes

- Let us see the below example of Figure class extended by Rectangle and Circle.



- In the above example area() for Figure being more generic we cannot define it. At the level of rectangle or Circle we can give the formula for area.

- Often, you would want to define a superclass that declares the structure of a given

abstraction without providing the implementation of every method

- The objective is to:
  - Create a superclass that only defines a generalized form that will be shared by all of

its subclasses

  - leaving it to each subclass to provide for its own specific implementations
  - Such a class determines the nature of the methods that the subclasses ***must implement***
  - Such a superclass is unable to create a meaningful implementation for a method or methods

- The class **Figure** in the previous example is such a superclass.

  Figure is a pure geometrical abstraction

  You have only kinds of figures like **Rectangle**, **Triangle** etc. which actually are subclasses of class **Figure**

  The class **Figure** has no implementation for the **area( )** method, as there is no way to determine the area of a **Figure**

The **Figure** class is therefore a partially defined class with no implementation for the

area( ) method

The definition of **area()** is simply a placeholder

The importance of abstract classes:

they define a generalized form (possibly some generalized methods with no implementations) *that will be shared by all of its subclasses*, so that *each subclass can provide specific implementations* of such methods.

abstract method – It's a method declaration with no definition

a mechanism which shall ensure that a subclass must compulsorily override such methods.

Abstract method in a superclass has to be overridden by all its subclasses.

The subclasses cannot make use of the abstract method that they inherit directly(without overriding these methods).

These methods are sometimes referred to as subclasses' responsibility as they have no implementation specified in the superclass

- To use an abstract method, use this general form: **abstract type name(parameter- list);**
- Abstract methods do not have a body
- Abstract methods are therefore characterized by the lack of the opening and closing braces that is customary for any other normal method
- This is a crucial benchmark for identifying an abstract class
- area method of Figure class made Abstract.

public abstract int area();

Any class that contains one or more abstract methods **must** also be declared abstract

- It is perfectly acceptable for an abstract class to implement a concrete method
- You cannot create objects of an abstract class
- That is, an abstract class cannot be instantiated with the new keyword
- Any subclass of an abstract class must *either implement all of the abstract methods in the superclass, or be itself declared abstract.*

Revised Figure Class – using abstract

- There is no meaningful concept of area( ) for an undefined two-dimensional geometrical

abstraction such as a Figure

- The following version of the program declares area( ) as abstract inside class Figure.
- This implies that class Figure be declared abstract, and all subclasses derived from class Figure must override area( ).

```
abstract class Figure{  double dimension1;  double dimension2;
Figure(double x, double y){  dimension1 = x;  dimension2 = y;
}
abstract double area();
}
class Rectangle extends Figure{  Rectangle(double x, double y){
super(x,y); }
double area(){
System.out.print("Area of rectangle is :");  return dimension1 * dimension2;
}
}
class Triangle extends Figure{  Triangle(double x, double y){  double area(){
System.out.print("Area for triangle is :");
return dimension1 * dimension2 / 2;
}
}
class FindArea{
public static void main(String args[]){  Figure fig;
```

```
Rectangle r = new Rectangle(9,5);  Triangle t = new Triangle(10,8);  fig = r;
System.out.println("Area of rectangle is :" + fig.area());
fig = t;
System.out.println("Area of triangle is :" + fig.area());
}
}
```

Packages:

Need for packages

▪ Till now, you did not use any package since all your classes were stored in the default package. Imagine a situation where all the classes are stored in one package. It would lead to tremendous confusion as it may lead to classes with similar names that is not allowed by the language.

This scenario is also referred to as a namespace collision.

Organizing classes into Packages

Packages are containers for classes and interfaces

▪ Classes and interfaces are grouped together in containers called packages

▪ To avoid namespace collision, we put the classes into separate containers called packages

▪ Whenever you need to access a class, you access it through its package by prefixing the class with the package name

Need for Packages

Packages are containers used to store the classes and interfaces into manageable units of code.

▪ Packages also help control the accessibility of your classes. This is also called as visibility control.

▪ Example:

```
package MyPackage;
class MyClass {...}
class YourClass{...}
```

Access Protection using Packages

Packages facilitate access-control

▪ Once a class is packaged, its accessibility is controlled by its package

▪ That is, whether other classes can access the class in the package depends on the access specifiers used in its class declaration

▪ There are four visibility control mechanisms packages offer:

▪ private

▪ no-specifier (default access)

▪ protected

▪ public

Packages & Access Control

Specifier	Accessibility
private	**Accessible in the same class only**
No-specifier (default access)	**Subclasses and non-subclasses in the same package**
protected	**Subclasses and non-subclasses in the same package, and subclasses in other packages**
public	**Subclasses and non-subclasses in the same package, as well as subclasses and non-subclasses in other packages. In other words, total visibility**

Specifier	Accessibility
private	same class only
No-specifier (default access)	same package only
protected	same package and subclasses
public	Anywhere in the program

Inbuilt Packages

java language has various in-built packages.

- java.lang, java.io, java.util, java.awt, java.applet, java.sql, javax.swing are some of the in-built packages.
- java.lang – Basic package which is automatically imported in all programs.
    - PrintWriter, String, StringBuilder, StringBuffer, All Wrapper Classes (totally 8),
    - Thread, Runnable,
    - Throwable , Exception etc
- java.io –Input / Output related classes are available here.
    - Scanner

- File , FileReader , FileWriter
- BufferedReader
- InputStreamReader
- IOException, FileNotFoundException etc

- java.util – Utility classes are available here. We can use these ready-made classes.
  - Date (to work with Date) , Calender (improved one)
  - Stack (LIFO) , Queue (FIFO) , Vector , ArrayList
  - Set, HashMap etc
- java.awt – Abstract Windowing Toolkit package.
  - Various classes like Button, Label, TextArea, Menu, MenuItem are available here.
- javax.swing – Swing package.
  - Various classes like JButton, JLabel, JTextArea, JMenu, JMenuItem, JTable are available here.
- java.sql – classes used for JDBC programming

  Various classes like Connection, DriverManager, ResultSet, SQLException are available here