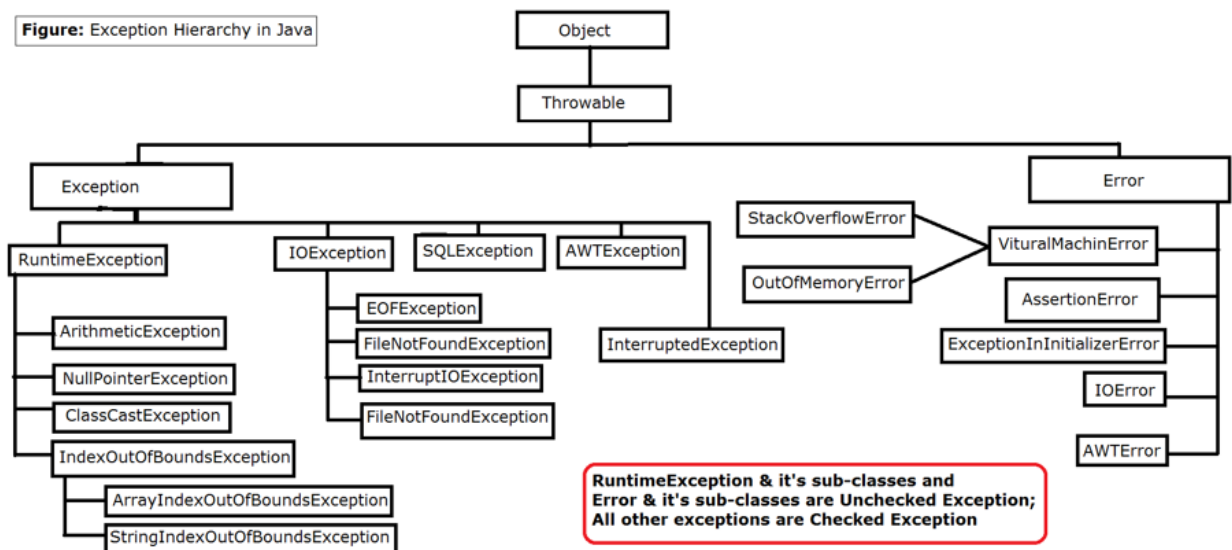


Exception Handling

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**.
- Exceptions of type Error are related to errors that are beyond the control, which occur in JVM itself.
- Errors that result from program activity are represented by subclasses of Exception

Figure: Exception Hierarchy in Java



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

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, then a try block ends normally
- If no exception is thrown by a try block, no catch clause will be executed and program control will resume after the catch.

```

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 “This will not be printed.” 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.

Ex: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");
    }
}
```

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.

- 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.

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[] number = {10,20,30,40,50};
        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 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[] numer = {10,20,30,40,50,60,70};
        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 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");
        }
    }
}

```

- First, **main()** sets up an exception context and then calls **division()**.
- The **division()** method then sets up another exception handling context and immediately throws a new instance of `ArithmeticException`, which is caught on the next line. The exception is then rethrown.

Throwable

| Method | Description |
|---|---|
| <code>Throwable fillInStackTrace()</code> | Returns a Throwable object that contains a completed stack trace. This object can be rethrown. |
| <code>String getLocalizedMessage()</code> | Returns a localized description of the exception. |
| <code>String getMessage()</code> | Returns a description of the exception. |
| <code>void printStackTrace()</code> | Displays the stack trace. |
| <code>void printStackTrace(PrintStream stream)</code> | Sends the stack trace to the specified stream. |
| <code>void printStackTrace(PrintWriter stream)</code> | Sends the stack trace to the specified stream. |
| <code>String toString()</code> | Returns a String object containing a complete description of the exception. This method is called by println() when outputting a Throwable object. |

```
// Using two Throwable methods.

class ExcTest {
    static void genException() {
        int[] nums = new int[4];

        System.out.println("Before exception is generated.");

        // generate an index out-of-bounds exception
        nums[7] = 10;
        System.out.println("this won't be displayed");
    }
}

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

        try {
            ExcTest.genException();
        }
        catch (ArrayIndexOutOfBoundsException exc) {
            // catch the exception
            System.out.println("Standard message is: ");
            System.out.println(exc);
            System.out.println("\nStack trace: ");
            exc.printStackTrace();
        }
        System.out.println("After catch.");
    }
}
```

The output from this program is shown here.

```
Before exception is generated.
Standard message is:
java.lang.ArrayIndexOutOfBoundsException: 7

Stack trace:
java.lang.ArrayIndexOutOfBoundsException: 7
    at ExcTest.genException(UseThrowableMethods.java:10)
    at UseThrowableMethods.main(UseThrowableMethods.java:19)
After catch.
```

Using finally

- A finally block will be executed whenever execution leaves a try/catch block, no matter what condition causes it
- Whether the try block ends normally, or because of an exception, the last code executed is that defined by **finally**
- The **finally** block is also executed if any code within the try block/ any of its **catch** clause return from a method
- The **finally** clause is optional. However, each **try** statement requires at least one **catch** or a **finally** clause
- 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
}
```

Example,

```
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

Using throws

- If a method generates an exception that it does not handle, it must declare that exception in a **throws** clause. This can be done by including a **throws** clause in the method's declaration.
- 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
{
    // 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. */

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

Important Points about throws:

- **throws** keyword is required only for checked exception and usage of **throws** keyword for unchecked exception is meaningless.
- **throws** keyword is required only to convince compiler and usage of **throws** keyword does not prevent abnormal termination of program.

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`

Exception Features Added by JDK 7 (FYI)

- Multi-catch feature
 - It allows 2 or more exceptions to be caught by the same catch clause
 - To create multi-catch, specify a list of exceptions within a single catch clause by separating each exception type in the list with the OR operator
 - Each multi catch parameter is implicitly **final**.

`catch(final ArithmeticException | ArrayIndexOutOfBoundsException e)`

Example

```

3 import java.io.IOException;
4 import java.sql.SQLException;
5
6 /** Copyright (c), AnkitMittal JavaMadeSoEasy.com */
7 public class ExceptionTest {
8     public static void main(String[] args) {
9
10         try{
11             int i=1;
12             if(i==1)
13                 throw new IOException();
14             else
15                 throw new SQLException();
16         }catch(IOException | SQLException ex){
17             System.out.println(ex + " handled ");
18         }
19     }
20 }
21
22
23 /*OUTPUT
24 java.io.IOException handled
25
26
27 */

```

- In the above program, single catch clause handles 2 types of Exceptions

Creating Exception Subclasses

- 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

The following example declares a new subclass of **Exception** and then uses that subclass to signal an error condition in a method. It overrides the **toString()** method, allowing a carefully tailored description of the exception to be displayed.

```
// This program creates a custom exception type.
class MyException extends Exception {
    private int detail;

    MyException(int a) {
        detail = a;
    }

    public String toString() {
        return "MyException[" + detail + "]";
    }
}

class ExceptionDemo {
    static void compute(int a) throws MyException {
        System.out.println("Called compute(" + a + ")");
        if(a > 10)
            throw new MyException(a);
        System.out.println("Normal exit");
    }

    public static void main(String args[]) {
        try {
            compute(1);
            compute(20);
        } catch (MyException e) {
            System.out.println("Caught " + e);
        }
    }
}
```

- This example defines a subclass of **Exception** called **MyException**. This subclass is quite simple: it has only a constructor plus an overloaded **toString()** method that displays the value of the exception.
- The **ExceptionDemo** class defines a method named **compute()** that throws a **MyException** object

PACKAGES

- It is helpful to group related pieces of a program together. This is accomplished by using a package.
- A package serves two purposes:
 - First, it provides a mechanism by which related pieces of a program can be organized as a unit. Classes defined within a package must be accessed through their package name.
 - Second, a package participates in Java's access control mechanism.
- Classes defined within a package can be made private to that package and not accessible by outside code.
- Hence, the package provides a means by which classes can be encapsulated.
- When you name a class, you are allocating a name from the **namespace**.
- No two classes can use the same name from the same **namespace**. This within a given namespace, each class name must be unique.
- In large programs, finding unique names for each class can be difficult. And also, you must avoid name collisions with code created by others
- The solution to these problems is the package, because it gives you a way to partition the namespace.
- When a class is defined within a package, the name of that package is attached to the class. Thus, avoiding name collisions with other classes that have the same name but are in other packages.

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.
- A class declared within that file will belong to the specified package.
- 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;`

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

Example,

```
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();
        }
    }
}
```

OUTPUT:

```
The Art pf Computer Programming
Knuth
1973

Java Fundamentals
Schildt
2013

Thirteen at Dinner
Christie
1933
```

Red storm Rising
Clancy
1986

On the Road
Kerouac
1955

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

Ex:

```
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 pf 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 |

Protected Members

- protected modifier creates a member that is accessible within its package and to subclasses in other packages

```

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 pf 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();
        }
        b[0].date=2014; //date has protected access in Book,
                       //so cannot be accessed in non subclass
    }
}

```

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

```

Ex:

```

package mypack;
import backpack.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

Importing Java's Standard Packages

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

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));
    }
}
```