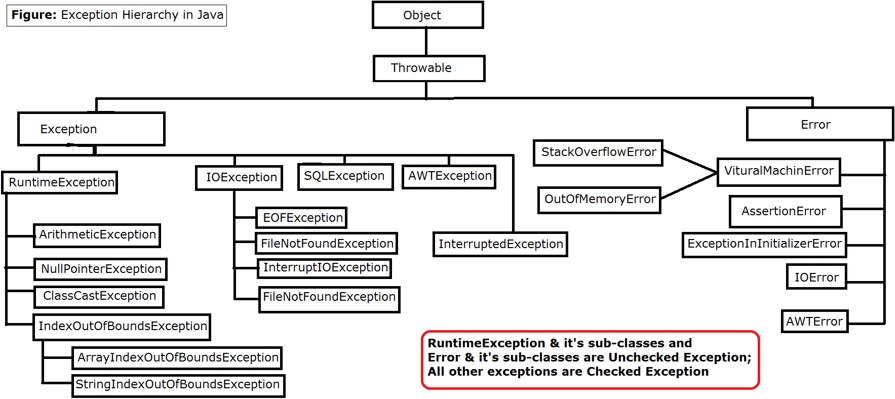
**The Exception Hierarchy**

# Exception Handling

* 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

## 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.

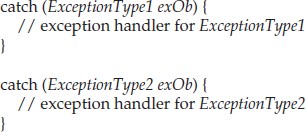
* Code can catch this exception (using **catch**) and handle it in some rational manner
* System-generated exceptions are automatically thrown by the Java run-time system.
* To manually throw an exception, use the keyword **throw**.
* Any exception that is thrown out of a method must be specified as such by a **throws**

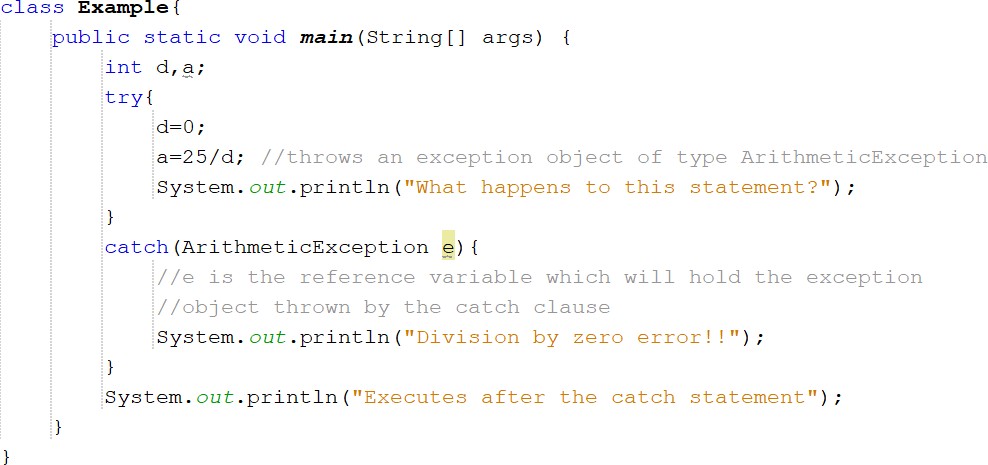
clause

### Using try and catch

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





* *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.

### OUTPUT:



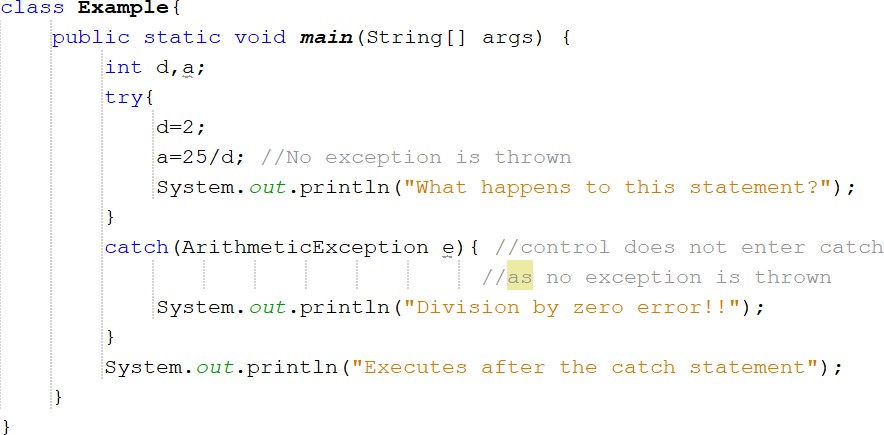
**Key points on Exception Handling**

* 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.

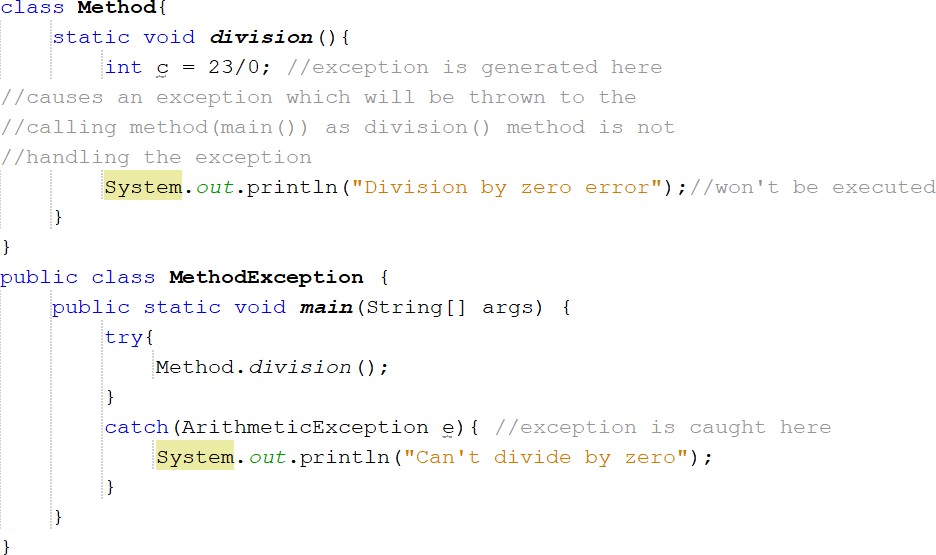
Ex:2



### OUTPUT:



* + 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

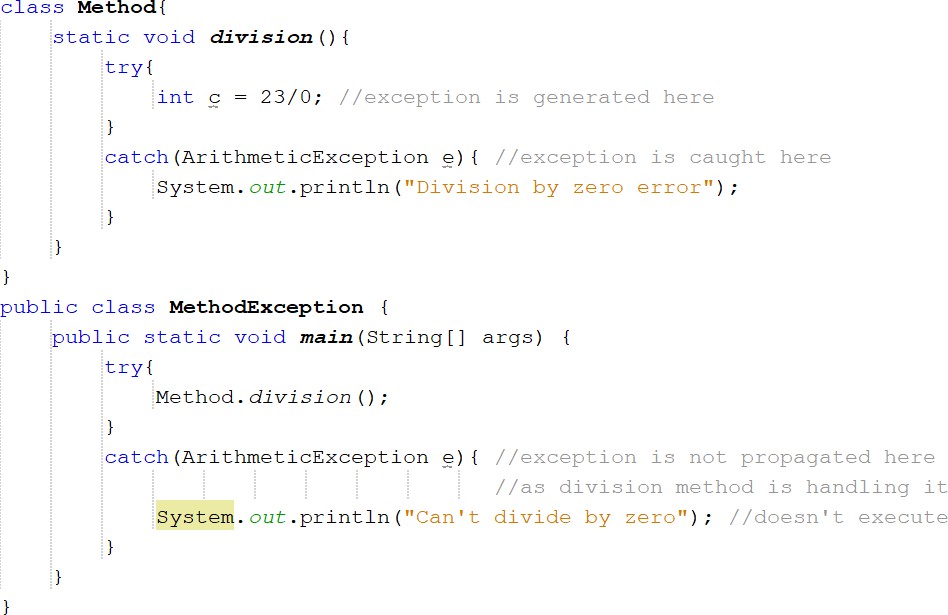


OUTPUT:



* **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().

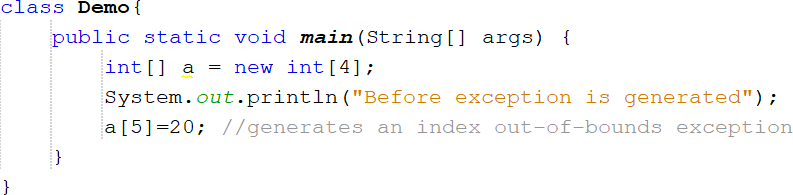
For Example,

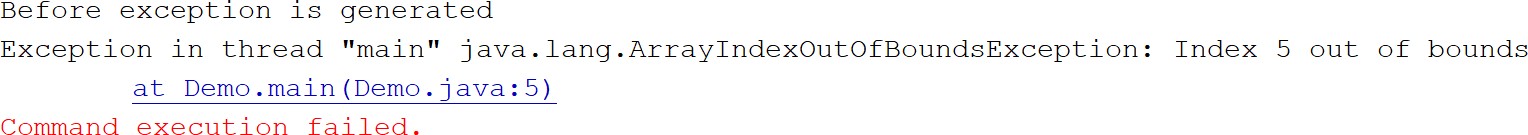


OUTPUT:

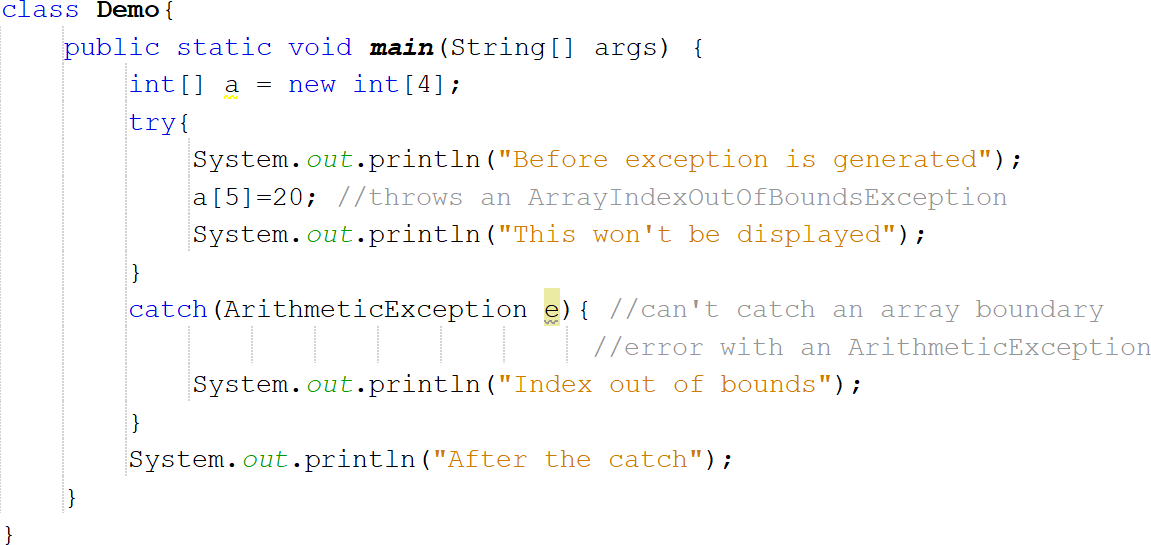


## The Consequences of an Uncaught Exception

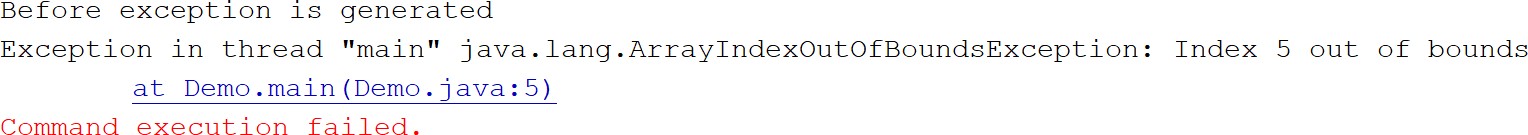
* If the program does not catch an exception, then it will be caught by the JVM
* But, JVM default exception handler terminates execution and displays an error message followed by list of method calls that cause the exception (referred as stack trace) “abnormal termination”
* For example,

**OUTPUT:**

* 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,



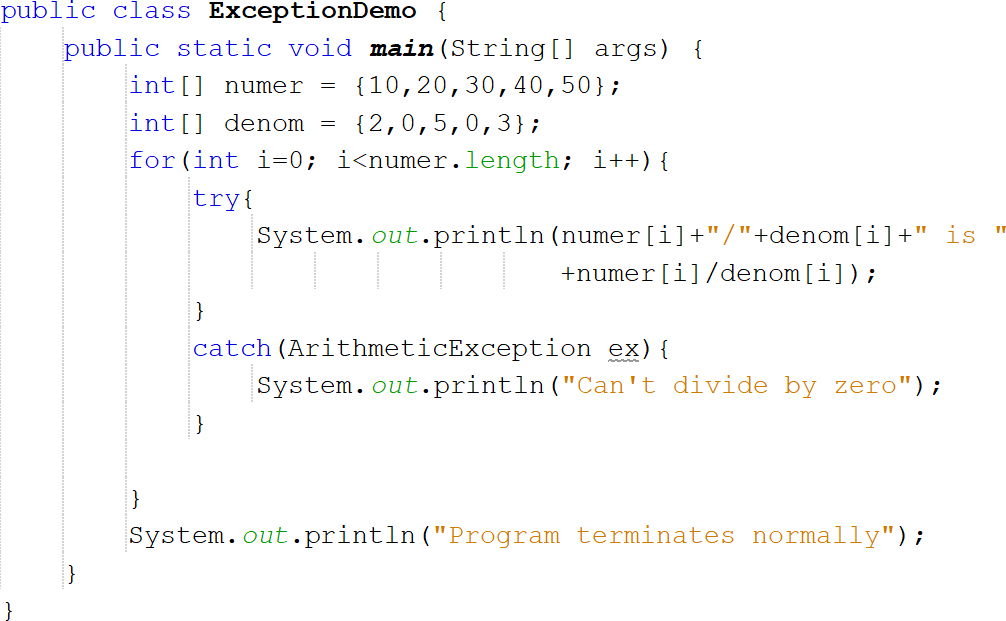
### OUTPUT:



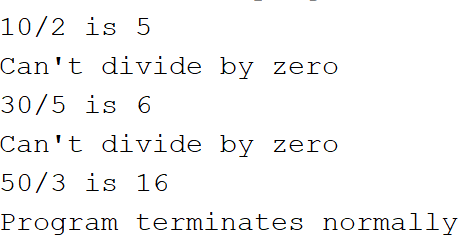
* The above program tries to catch an array boundary error with a catch for ArithmeticException, whereas ArrayOutOfBoundsException is generated when the array boundary is overrun.

## 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,



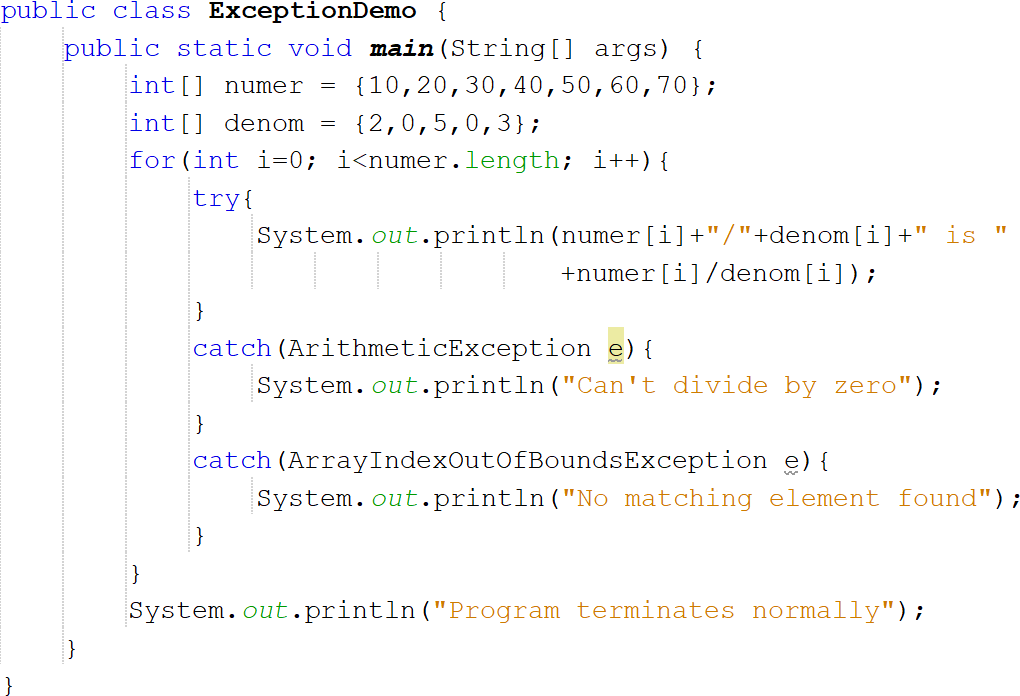
### OUTPUT:



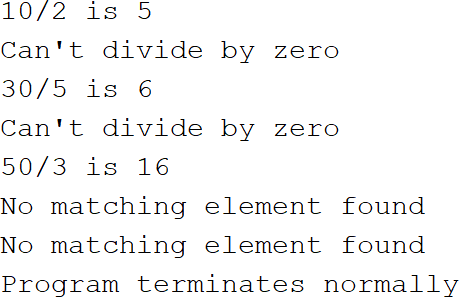
* In the above program, if a division by zero occurs, an ArithmeticException is generated. This exception is handled by reporting the error and then continue with execution
* Thus, attempting to divide by zero does not cause an abrupt termination of the program. Instead, it is handled and program execution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



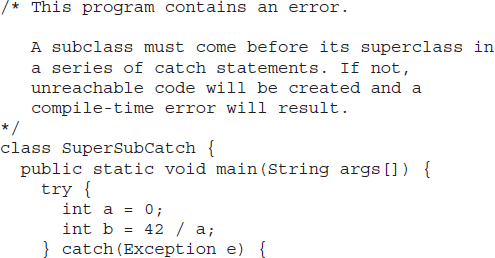
### OUTPUT:

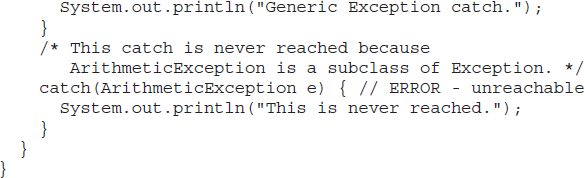


* Each catch responds only to its own type of exception
* **catch** clauses are checked in the order in which they occur in a program

## 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.





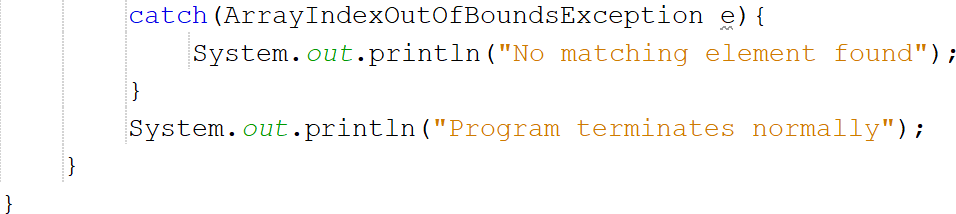
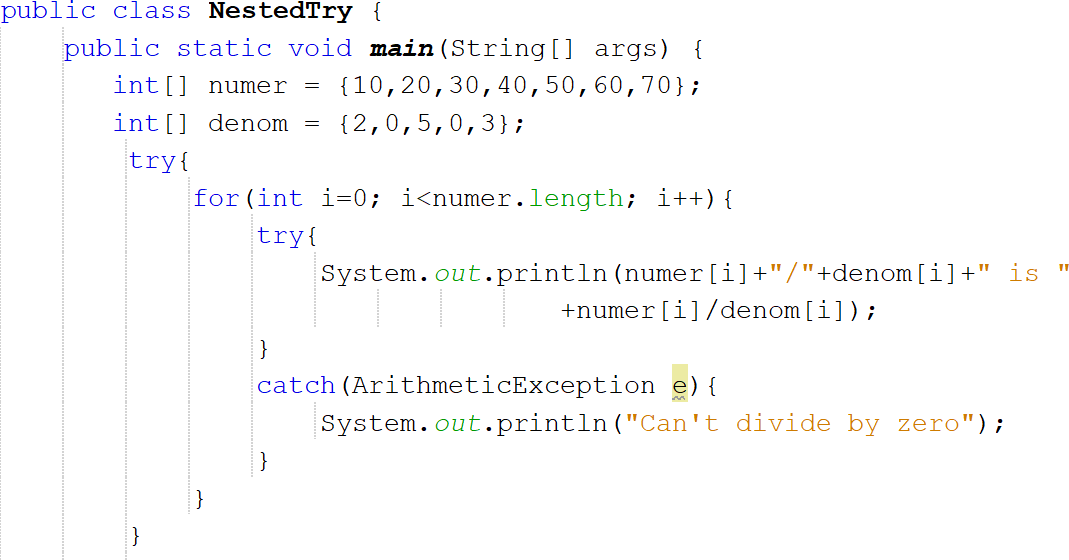
* **catch** statement will handle all **Exception**-based errors, including

**ArithmeticException**. This means that the second **catch** statement will never execute

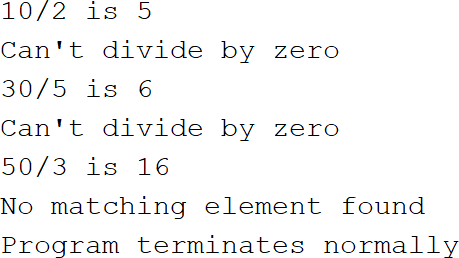
* To solve this problem, Superclass Exception class must be included as the last catch clause

## 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



### OUTPUT:



* 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

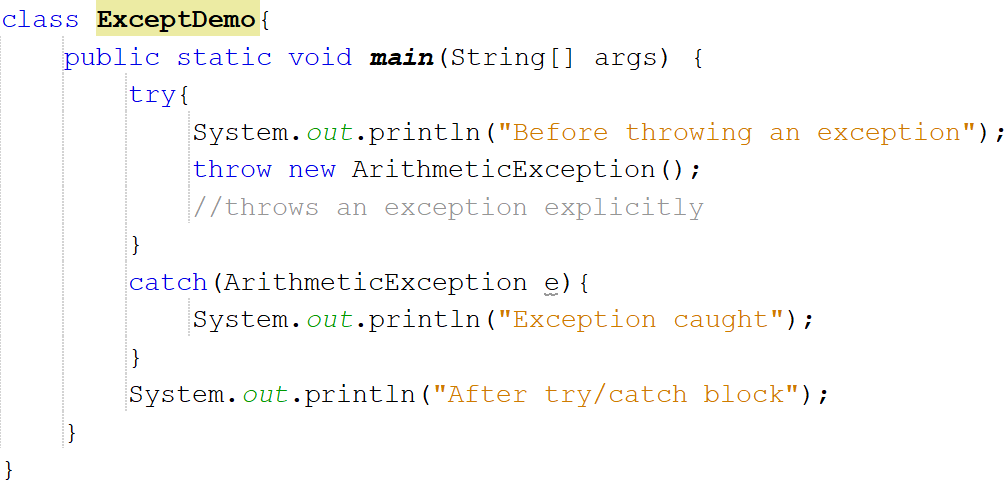
* Preceding examples have only been catching exceptions that are thrown by the Java run-time system.
* However, it is possible for the program to throw an exception explicitly, using the

**throw** statement.

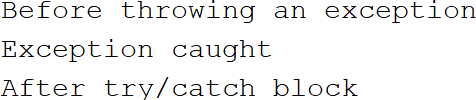
* The general form of **throw** is as follows:

throw exceptOb;

* exceptOb must be an object of an exception class derived from Throwable
* **throw** throws an object



### OUTPUT:

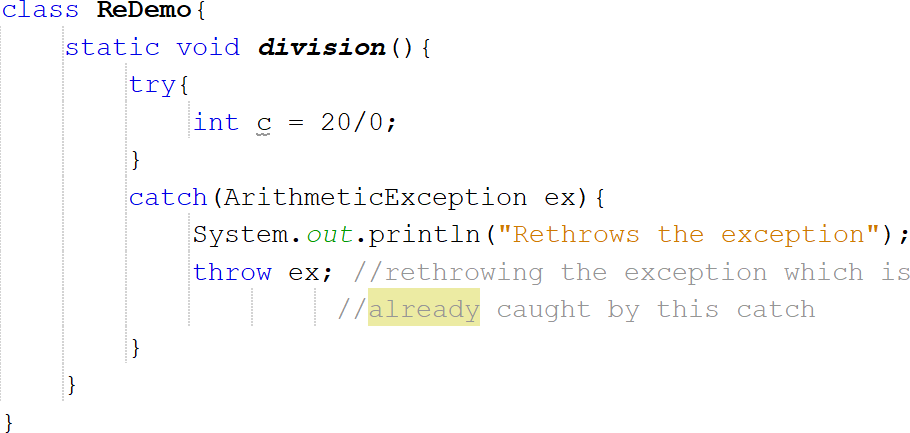


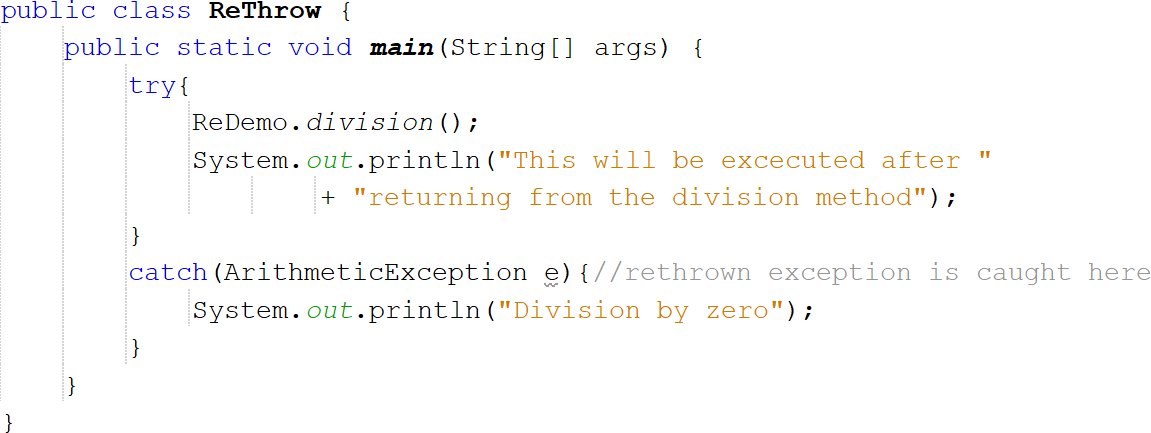
**Rethrowing an Exception**

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

### catch

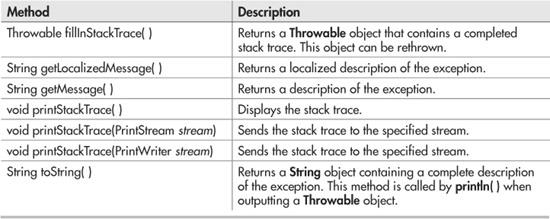
**For example,**

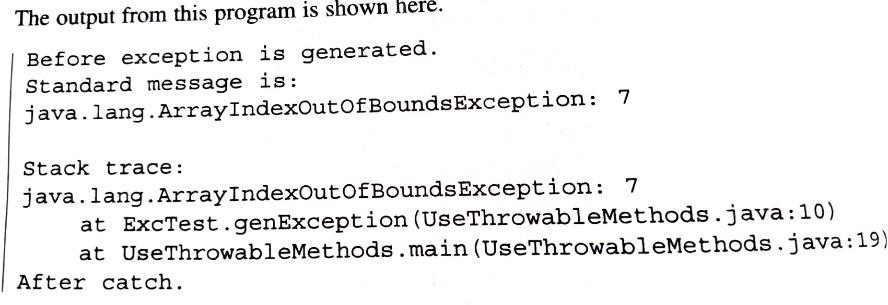
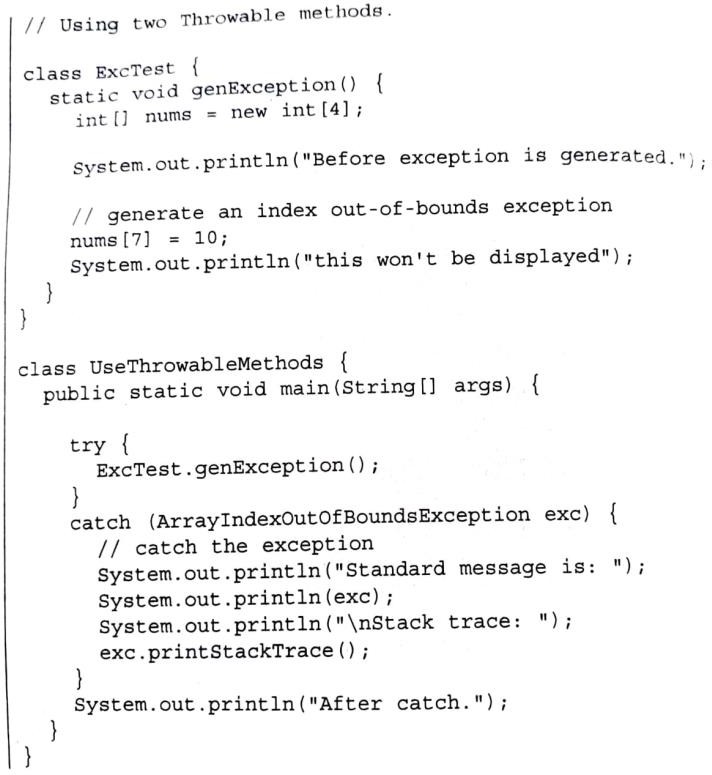




* First, **main( )** sets up an exception context and then calls **division( )**.
* The **division()** method then sets up another exception handling context and immediately throws a new instance of ArithmeticException, which is caught on the next line. The exception is then rethrown**.**

## Throwable





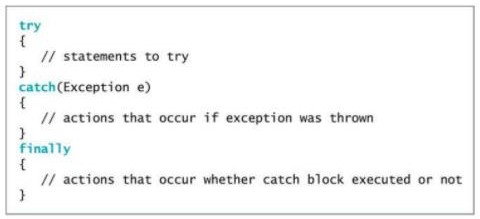
**Using finally**

* 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**

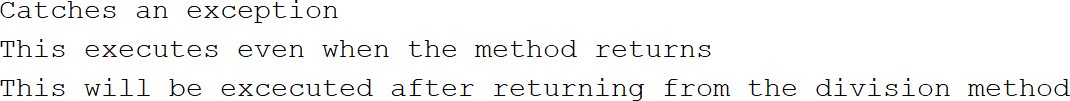
or a **finally** clause

* The general form of **try/catch** that includes finally is

Example,



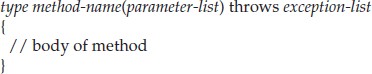
### OUTPUT:



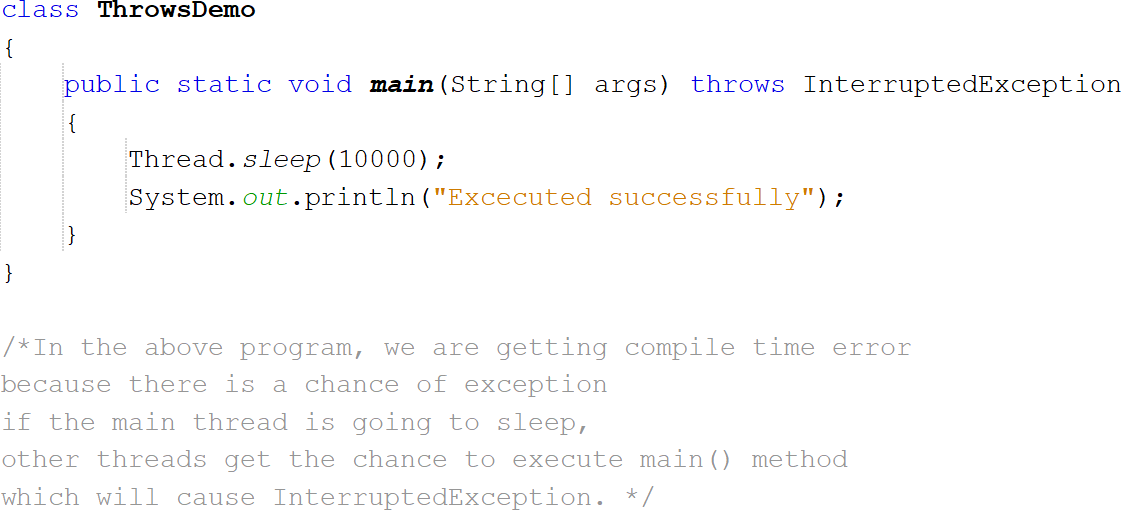
* In the above example, finally block is executed even after the method returns

## 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:



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



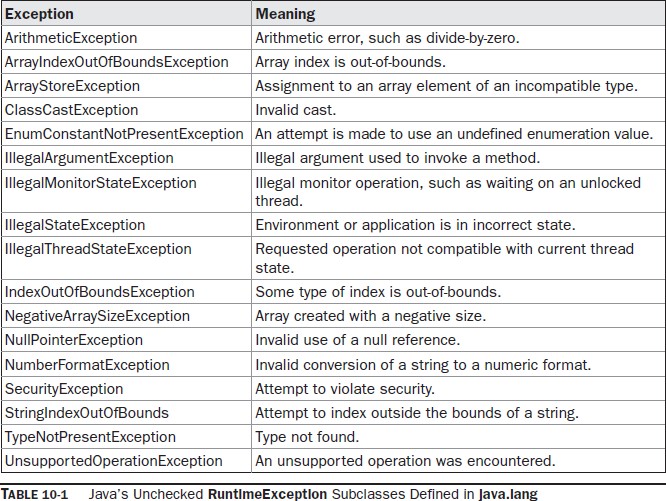
* In the above example**, main( )** throws **InterruptedException** but does not handle it
* **main( )** must define a **try**/**catch** statement that catches this exception.

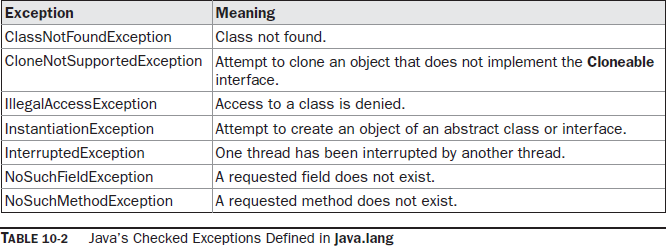
### Important Points about throws:

* **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 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**.

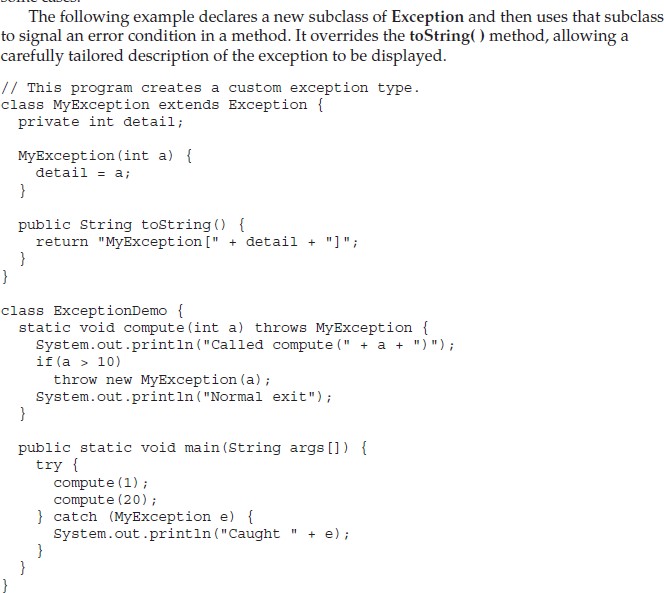
catch(final ArithmeticException | ArrayIndexOutOfBoundsException e) Example



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

## Creating Exception Subclasses

* Two commonly used **Exception** constructors are:
* The first form creates an exception that has no description.
* The second form lets you specify a description of the exception



* 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** objec

# 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:

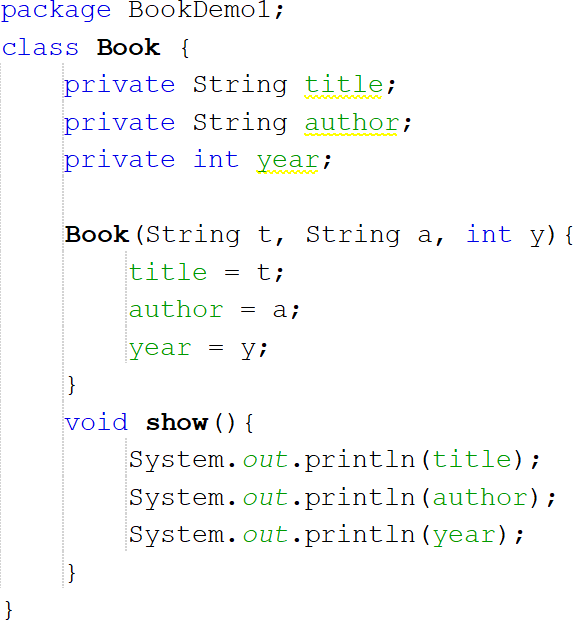
1. First, by default, the run-time system uses the current working directory as its starting point. If the package is in a subdirectory of the current directory, it will be found.

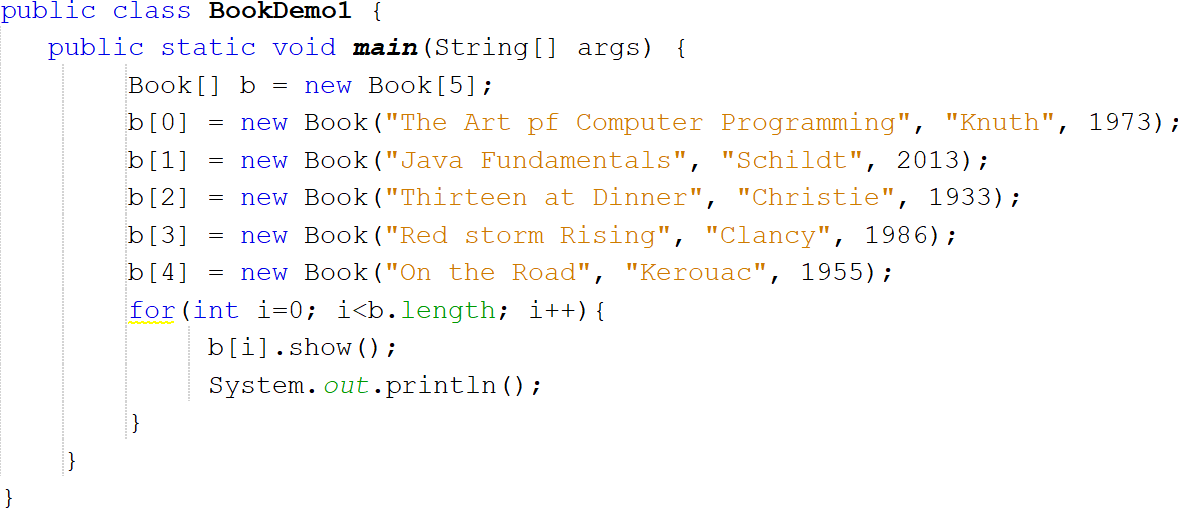
Example,

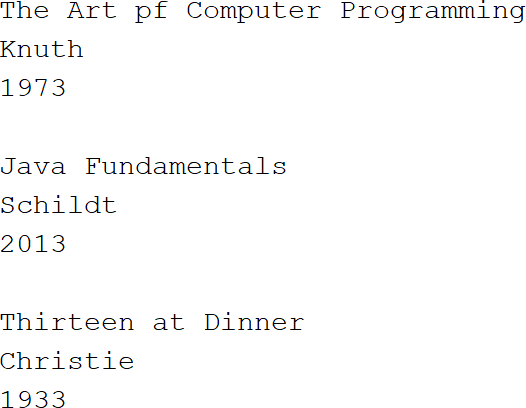
1. Second, a directory path or paths can be specified by setting the **CLASSPATH**

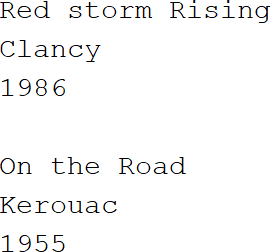
environmental variable.

1. Third, the -**classpath** option can be used **java** and **javac** to specify the path to the classes





**OUTPUT:**



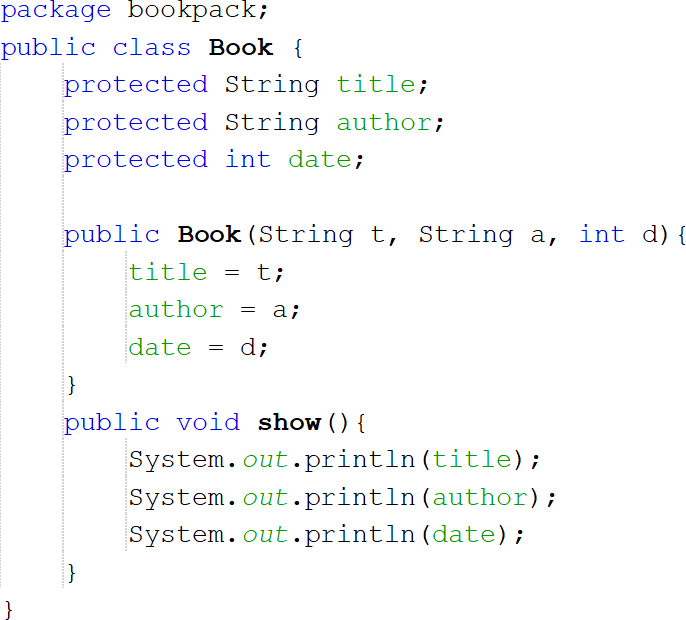
## Accessing a Package

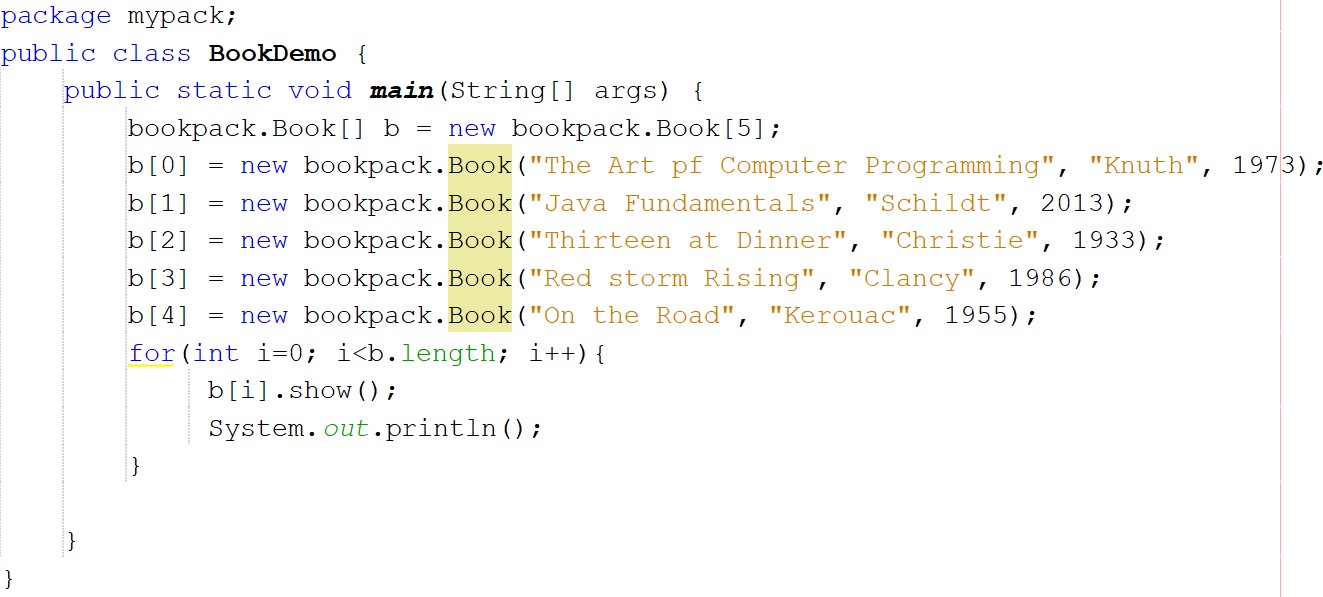
* In the above example, if **Book** and **BookDemo1** were in different packages, then **Book**

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:



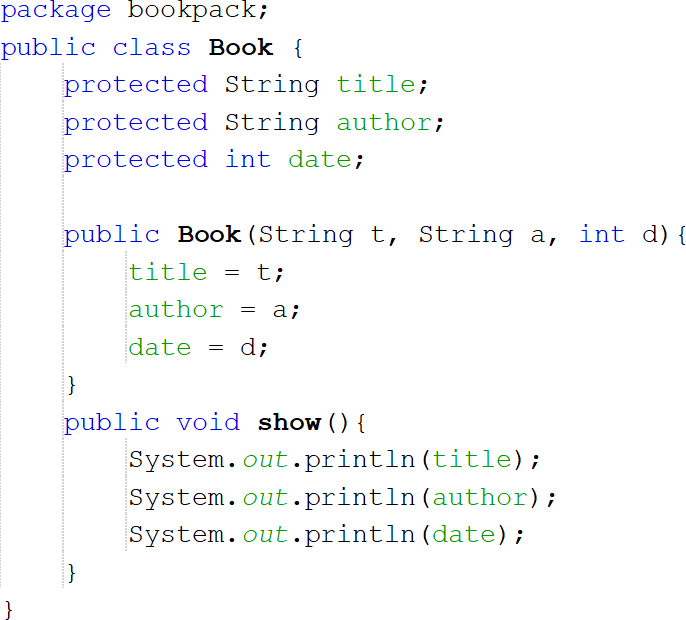


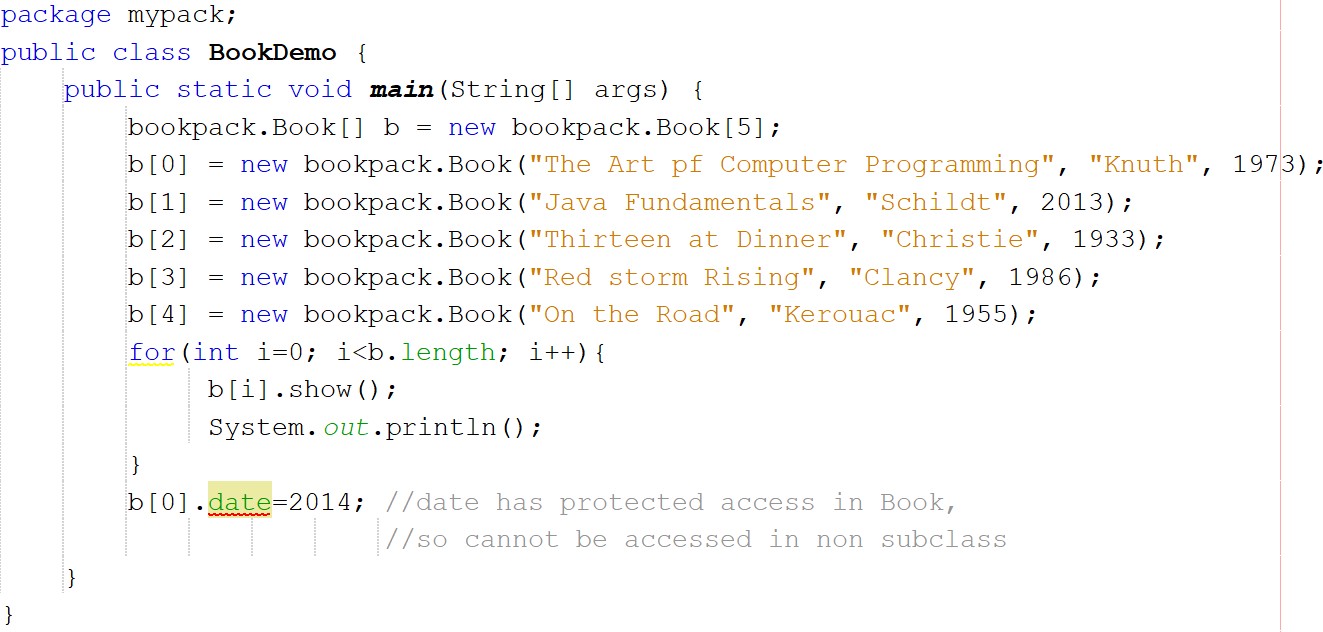
* To access **Book**, we must fully qualify its name to include its full package specification
* Without this specification, **Book** would not be found
* Syntax is,

packageName.className;

## Packages and Member Access

**Protected Members**

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



## 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

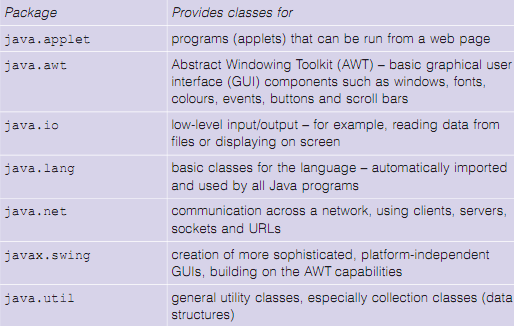
Ex:

import bookpack.Book; //Book class is imported from bookpack import bookpack.\*; //all of the classes in bookpack is imported

### Ex:

* No longer need to qualify Book with its package name

**Importing Java’s Standard Packages**



## 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.\*;

