

Session 05

Package, Exceptions

(<http://docs.oracle.com/javase/tutorial/essential/exceptions/index.html>)

Objectives

- Packages
- Exception Handling
 - try block
 - catch block
 - finally block
 - custom exception class

Packages

- A *package* is a grouping of related classes, interfaces, enumerations, and annotation types providing access protection and name space (set of pre-defined names) management.
- Syntax to create a new package:
`package [package name];`
 - This statement must be the **first line** in the source file.
 - There can be only one package statement in each source file, and it applies to all types in the file.
 - *The compiler will read source code line-by-line from the beginning of the source file. So, the first work must be carried out is creating the folder and the folder name is the package name. The package information will be added to classes in this package.*

Using Packages Members

- To use a public package member from outside its package, we can:

- Refer to the member by its fully qualified name

```
graphics.Rectangle myRect = new graphics.Rectangle();
```

- Import the package member

```
import graphics.Rectangle;
```

```
...
```

```
Rectangle myRectangle = new Rectangle();
```

- Import the member's entire package

```
import graphics.*;
```

```
...
```

```
Rectangle myRectangle = new Rectangle();
```

- 2 packages can contain 2 classes which have the same name

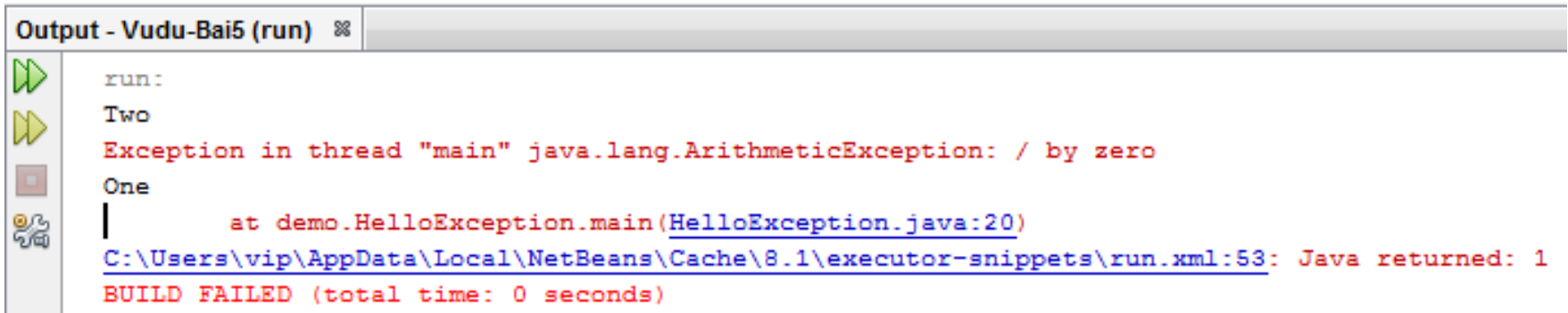
```
pkg1.ClassA obj1;
```

```
pkg2.ClassA obj2;
```

Example

```
public class HelloException {  
    public static void main(String[] args) {  
        System.out.println("Two");  
        // The division is ok  
        int value = 10 / 2;  
        System.out.println("One");  
        // divide by zero  
        // error encountered in here  
        value = 10 / 0;  
        //this line is not executed  
        System.out.println("Let's go!");  
    }  
}
```

Output



The screenshot shows an IDE output window titled "Output - Vudu-Bai5 (run)". On the left side of the window, there is a vertical toolbar with icons for running (a green play button), stepping through code (a yellow play button), stopping (a red square), and debugging (a magnifying glass over a bug). The output text is as follows:

```
run:
Two
Exception in thread "main" java.lang.ArithmeticException: / by zero
One
|       at demo.HelloException.main(HelloException.java:20)
|       C:\Users\vip\AppData\Local\NetBeans\Cache\8.1\executor-snippets\run.xml:53: Java returned: 1
BUILD FAILED (total time: 0 seconds)
```

Fix it

```
public class HelloCatchException {  
    public static void main(String[]  
args) {  
        System.out.println("Two");  
        // The division is ok  
        int value = 10 / 2;  
        System.out.println("One");  
    }  
}
```

```
try { // divide by zero
    // error encountered in here
    value = 10 / 0;
    //this line is not executed
    System.out.println("Value =" + value);
} catch (ArithmeticException e) {
    // this line in catch is executed
    System.out.println("Error: " +
e.getMessage());
    // this lines in catch are executed
    System.out.println("Ignore...");
}
// this line is executed
System.out.println("Let's go!");
}}
```


Exceptions

- When a program is executing something occurs that is not quite normal from the point of view of the goal at hand.
- For example:
 - a user might type an invalid filename;
 - a file might contain corrupted data;
 - a network link could fail;
 - ...
- Circumstances of this type are called *exception conditions* in Java and are represented using objects (All exceptions descend from the `java.lang.Throwable`).

Catching Exceptions

```
try {  
    // Exception-throwing code  
}  
catch (Exception_type name) {  
    // Exception-handling code  
}
```

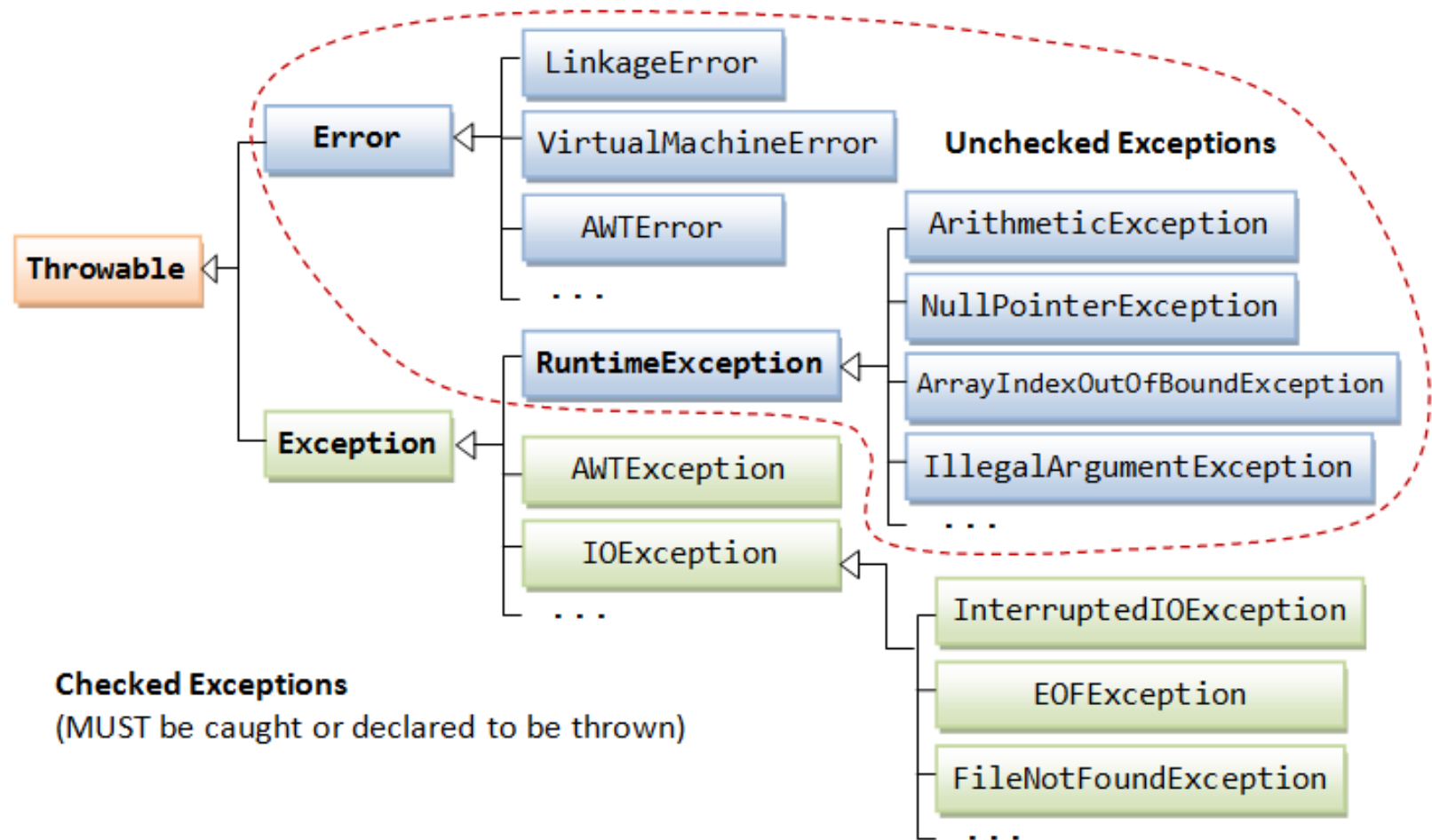
Example 1

```
public class Exception1 {  
    public static void main(String[]  
args) {  
        String sNum = "CTB";  
        String sDate = "10/03/2016";  
        int num = Integer.parseInt(sNum);  
        SimpleDateFormat f = new  
SimpleDateFormat("dd/MM/yyyy");  
        Date d = f.parse(sDate);  
    }  
}
```

Example 2

```
public class Excepton2 {  
    public static void main(String[] args) {  
        String s = "Subject - OOP";  
        try{  
            System.out.println("Before");  
            System.out.println(s.substring(50));  
        }  
        catch (StringIndexOutOfBoundsException e) {  
            System.out.println("Error:"+e.toString());  
        }  
        System.out.println("After");  
    }  
}
```

Exception Classes



Multiple catch blocks

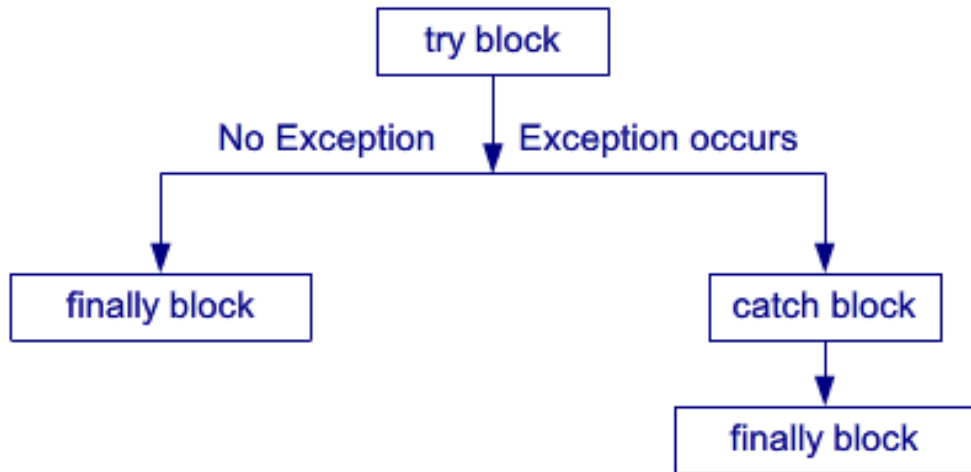
- try block may contain code that throws different exception types. This can even happen if the block contains only a single line of code, because a method is allowed to throw different types to indicate different kinds of trouble.

- Multiple catch blocks

```
try {  
    // do something ...  
} catch (Exception1 e) {  
    //Exception handler for Exception1  
} catch (Exception2 e) {  
    //Exception handler for Exception2  
} finally {  
    // this finally always executed  
    // to do something here  
}
```

The *finally* Block

- The last catch block associated with a try block may be followed by a finally block.



- The finally block's code is guaranteed to execute in nearly all circumstances excepts:
 - The death of the current thread
 - Execution of `System.exit()`
 - Turning off the computer

Declaring Exceptions

- There is a way to call exception-throwing methods without enclosing the calls in try blocks. A method declaration may end with the throws keyword, followed by an exception type, or by multiple exception types followed by commas.

```
private int throwsTwo() throws IOException, AWTException  
{...}
```

Of course, methods that call throwsOne() or throwsTwo() must either enclose those calls in try blocks or declare that they, too, throw the exception types.

Two Kinds of Exception

- *Checked exception*
 - Must be handled by either the try-catch mechanism or the throws-declaration mechanism.
- Runtime exception (*Unchecked exception*)
 - The right time to deal with runtime exceptions is when you're designing, developing, and debugging your code. Since runtime exceptions should never be thrown in finished code.

There is some difference there for checked and unchecked Exception

Suppose **checked exception** means it shows the **compile time exception**

ex: NoSuchMethod exception, NoSuch Field Exception , ClassNotFoundException

where as in **Unchecked Exception** shows **Runtime** Only

ex: Nullpointer Exception, Number Format Exception

Checked exceptions should always be caught

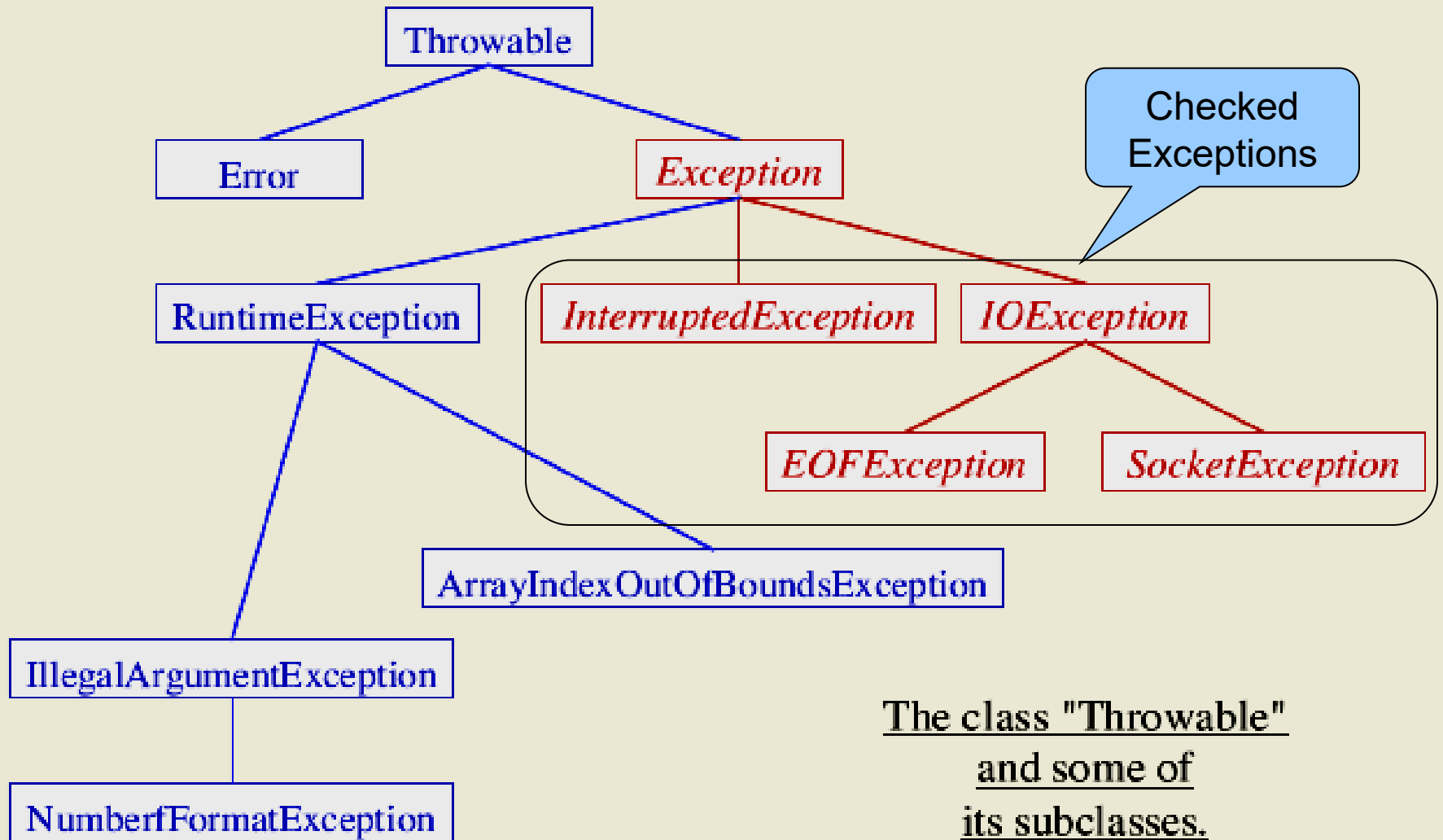
Runtime exceptions don't have to be caught.

Errors should never be caught.

The stack trace is recorded when the exception is constructed.

It is never appropriate for application programmers to construct and throw errors.

The Exception Inheritance Hierarchy



Some runtime (unchecked) exceptions

The types of exceptions that ***need not be included*** in a methods **throws** list are called **Unchecked Exceptions**. They **can be foreseen and prevented by a programmer and, generally speaking, will never occur in a high-quality program**:

ArithmeticException

ArrayIndexOutOfBoundsException

ClassCastException

IndexOutOfBoundsException

IllegalStateException

NullPointerException

SecurityException

Unchecked exception

```
public class Exception3 {  
    public static void main(String[] args) {  
        String s = "Subject: OOP";  
        int x=12, y=0;  
        try{  
            System.out.println("Before");  
            System.out.println(s.substring(50));  
            System.out.println("x/y = "+ x/y);  
        } catch (StringIndexOutOfBoundsException  
e) {  
            System.out.println("Error:"+e.toString());  
        }  
    }  
}
```

```
catch (ArithmeticException e) {  
    System.out.println("Error: "+  
        e.getMessage() ); }  
catch (Exception e) {  
    e.printStackTrace();  
} finally {  
    System.out.println("always  
executed");  
}  
System.out.println("after");  
}  
}
```

Some checked exceptions

The types of exceptions that must be included in a methods **throws** list if that method can generate one of these exceptions and does not handle it itself are called **Checked Exceptions**. **Checked exceptions can happen at any time, cannot be prevented and therefore the language enforces to deal with them.**

ClassNotFoundException

CloneNotSupportedException

IllegalAccessException

InstantiationException

InterruptedException

NoSuchFieldException

NoSuchMethodException

Checked Exceptions

```
public class Exception4 {  
    public static void  
    main(String[] args) {  
        File file = new  
        File("example.txt");  
        try{  
            file.createNewFile();  
        } catch (IOException ex) {  
            System.out.println(ex);  
        }  
    }  
}
```


try-catch-finally

- A try-block must be accompanied by at least one catch-block or a finally-block.
- You can have multiple catch-blocks. Each catch-block catches only one type of exception.
- The order of catch-blocks is important. A subclass must be caught (and placed in front) before its superclass. Otherwise, you receive a compilation error "exception XxxException has already been caught"

```
try{
    System.out.println("Before");
    System.out.println(s.substring(50));
    System.out.println("x/y = "+ x/y);
} catch (Exception e) {
    e.printStackTrace();
} catch (StringIndexOutOfBoundsException e) {
    System.out.println(e);
} catch (ArithmeticException e) {
    System.out.println(e);
} finally{
    System.out.println("always executed");
}
System.out.println("after");
```

throw and throws

- The throw keyword in Java is used to explicitly throw an exception from a method or any block of code. We can throw either checked or unchecked exception.
- throws is a keyword in Java which is used in the signature of method to indicate that this method might throw one of the listed type exceptions. The caller to these methods has to handle the exception using a try-catch block.

```
class ThrowExcep1 {  
    static void fun() {  
        try{  
            throw new NullPointerException("demo");  
        } catch(NullPointerException e) {  
System.out.println("Caught inside fun().");  
            throw e; // rethrowing the exception  
        }  
    }  
  
    public static void main(String args[]) {  
        try{  
            fun();  
        } catch(NullPointerException e) {  
System.out.println("Caught in main.");  
        }  
    }  
}
```

```
class ThrowsExecp2 {  
    static void fun() throws IllegalAccessException  
    {  
        System.out.println("Inside fun(). ");  
        throw new IllegalAccessException("demo");  
    }  
    public static void main(String args[]) {  
        try{  
            fun();  
        }  
        catch(IllegalAccessException e) {  
            System.out.println("caught in main.");  
        }  
    }  
}
```

Creating Your Own Exception Classes

- Decide whether you want a checked or a runtime exception.
 - Checked exceptions should extend `java.lang.Exception` or one of its subclasses.
 - Runtime exceptions should extend `java.lang.RuntimeException` or one of its subclasses

Creating Your Own Exception Classes...

- The argument list of these constructors should include
 - A message
 - A cause
 - A message and a cause

Exception Occurrence

```
class ThrowStatement extends Exception {  
    public static void exp(int ptr) {  
        if (ptr == 0)  
            throw new NullPointerException();  
    }  
    public static void main(String[] args) {  
        int i = 0;  
        ThrowStatement.exp(i);  
    }  
}
```

```
java.lang.NullPointerException  
at ThrowStatement.exp(ThrowStatement.java:4)  
at ThrowStatement.main(ThrowStatement.java:8)
```

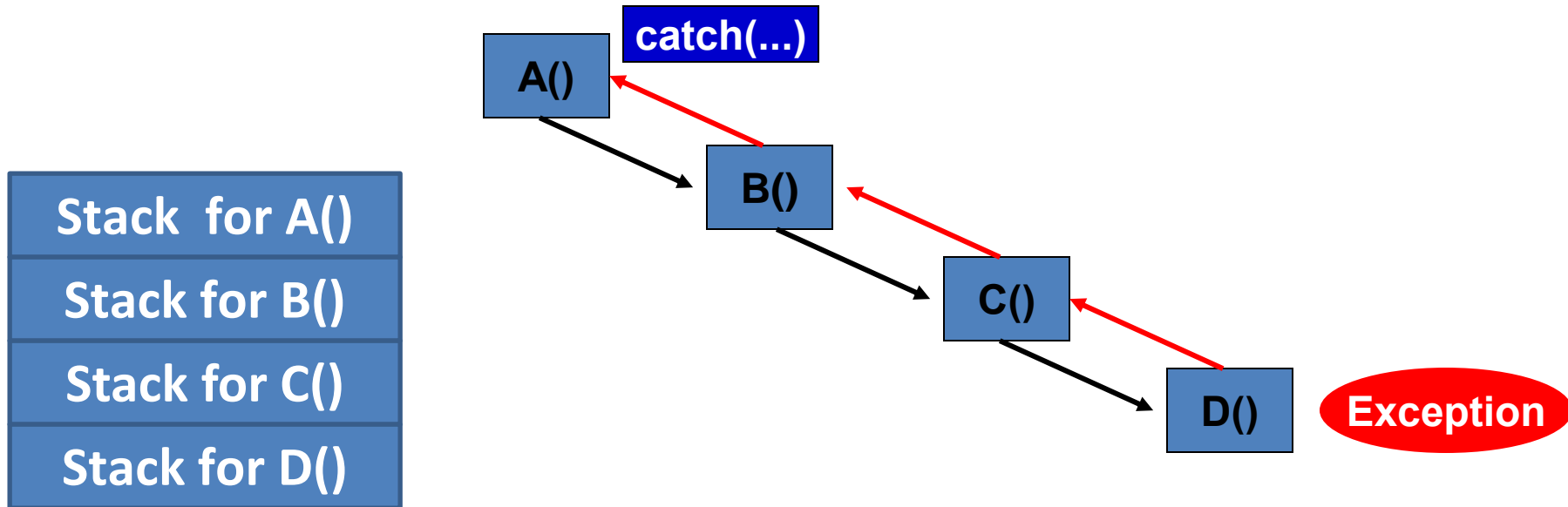

Exceptions and Overriding (1)

- When you extend a class and override a method, the Java compiler insists that all exception classes thrown by the new method must be the same as, or subclasses of, the exception classes thrown by the original method. In other words, an overridden method in a **sub class** **must not throw Exceptions not thrown in the base class**. Thus if the overriding method does not throw exceptions, the program will compile without complain.

Exceptions and Overriding (2)

```
class Disk {  
    void readFile() throws EOFException{  
    }  
class FloppyDisk extends Disk {  
    // ERROR!  
    void readFile() throws IOException {}  
}  
class DiskFix {  
void readFile() throws IOException {}  
}  
class FloppyDisk extends Disk {  
void readFile() throws EOFException {} //OK  
}
```

Exception Propagations



Stack trace

When an exception occurs at a method, program stack is containing running methods (method A calls method B,...). So, we can trace statements related to this exception.

Exception Propagations

```
1 public class ExceptionPropagate {
2     public void mA()
3     {
4         mB();
5     }
6     public void mB()
7     {
8         mC();
9     }
10    public void mC()
11    {
12        System.out.println(5/0);
13    }
14    public static void main(String[] args) {
15        ExceptionPropagate obj= new ExceptionPropagate();
16        obj.mA();
17    }
18 }
```

Output - FirstPrj (run) x

```
run:
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at ExceptionPropagate.mC(ExceptionPropagate.java:12)
    at ExceptionPropagate.mB(ExceptionPropagate.java:8)
    at ExceptionPropagate.mA(ExceptionPropagate.java:4)
    at ExceptionPropagate.main(ExceptionPropagate.java:16)
```

Java Result: 1

Example

```
public class ValidateException extends
    Exception {
    public ValidateException(String message) {
        super(message);
    }
}

private void validateID(String id)
    throws ValidateException{
    if(!id.matches("^ [Bb] {1} \\d {2} [A-Za-
z] {4} \\d {3} $" ) )
        throw new ValidateException ("ID
\\\""+id+\"\\ not valid");
    }
```

Summary

- Packages
- Exception Handling
- Multiple Handlers
- Code Finalization and Cleaning Up (finally block)
- Custom Exception Classes