Vietnam National University of HCMC
International University
School of Computer Science and Engineering

Package and Exception Handling

(IT069IU)

Nguyen Trung Ky

ntky@hcmiu.edu.vn

it.hcmiu.edu.vn/user/ntky

Previously,

- **Java Generic Collections**
  - Type-Wrapper Classes for Primitive Types
    - Autoboxing vs Auto-unboxing
  - List
    - ArrayList
    - Vector
    - LinkedList

Agenda Today

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

Packages

- A package is a grouping of related classes, interfaces 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;
```

Exceptions

- **Exception**: Error beyond the control of a program. When an exception occurs, the program will terminate abruptly.
- 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;
  - An accessed file does not exist or 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**).

Exceptions

- The following program causes an exception.

ExceptionDemo_1.java *

```
1   public class ExceptionDemo_1 {
2     public static void main (String[] args)
3     {   int x=5, y=0;
4         System.out.println(x/y);
5         System.out.println("Hello");
6     }
7   }
```

Output - Chapter04 (run)

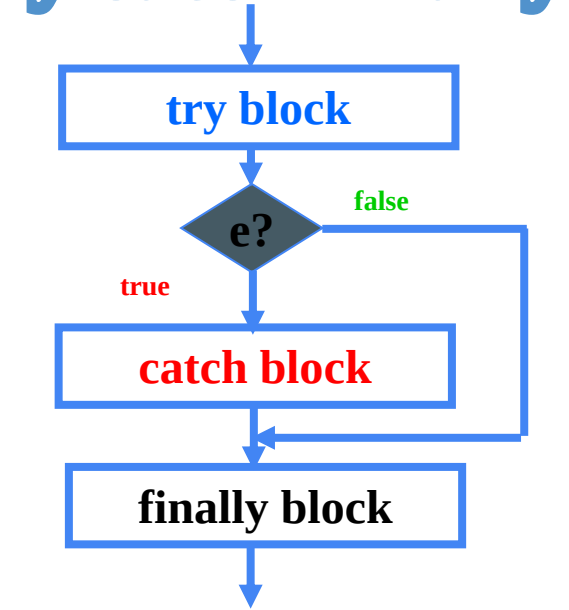
```
run:
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at ExceptionDemo_1.main(ExceptionDemo_1.java:4)
Java Result: 1
BUILD SUCCESSFUL (total time: 2 seconds)
```

Exceptions are pre-defined data (Exception classes) thrown by JVM and they can be caught by code in the program

Why we need Exception Handling

- Java exception handling is important because it helps maintain the normal, desired flow of the program even when unexpected events occur.
- If Java exceptions are not handled, programs may crash or requests may fail. This can be very frustrating for customers and if it happens repeatedly, you could lose those customers.

How to handle exceptions in Java: try catch finally



If no exception is thrown in the try *block, all* catch *blocks are bypassed*

```
try {
      < statements may cause exceptions >
}
```

```
catch  (  ExceptionType1  e1 ) {
    < statements handle the situation  1>
}
```

```
catch  (  ExceptionType2  e2) {
    < statements handle the situation  2>
}
```

```
finally {
    < statements are always executed >
}
```

If an exception arises, the first matching catch *block, if any, is executed, and the others are skipped*

Types of Exceptions

- java.lang.**Throwable** (implements java.io.Serializable)
  - java.lang.**Error**
  - java.lang.**Exception**
    - java.lang.**RuntimeException**

Checked Exceptions
(We must use the try catch blocks)

Unchecked- Exceptions
Program Bugs
(We may not use the try catch blocks)

Refer to the Java.lang documentation for more information.

```
1   public class ExceptionDemo_1 {
2       public static void main (String[] args)
3   {   int[] a= { 1,2,3,4,5};
4       int n=10;
5       for (int i=0;i<n;i++)
6           System.out.print("" + a[i] + ",");
7   }
8   }
9
```

```
Output - Chapter04 (run)
run:
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5
1,2,3,4,5,        at ExceptionDemo_1.main(ExceptionDemo_1.java:6)
Java Result: 1
BUILD SUCCESSFUL (total time: 1 second)
```

```
1   public class ExceptionDemo_1 {
2       public static void main (String[] args)
3   {   int[] a= { 1,2,3,4,5};
4       int n=10;
5       try
6       { for (int i=0;i<n;i++)
7           System.out.print("" + a[i] + ",");
8       }
9       catch(Exception e) // general exception
10      {   System.out.println(e);
11      }
12  }
```

```
Output - Chapter04 (run)
run:
1,2,3,4,5,java.lang.ArrayIndexOutOfBoundsException: 5
BUILD SUCCESSFUL (total time: 0 seconds)
```

Two Types of Exception

- Checked exception:
  - Checked exceptions represent errors outside the control of the program.
  - For example, caused by faults outside code like missing files, invalid class names, and networking errors.

```
FileInputStream fis = null;
try {
    fis = new FileInputStream("B:/myfile.txt");
} catch (FileNotFoundException e) {
    e.printStackTrace();
    rollbar.error(e, "Hello, Rollbar");
}
```

  - Must be handled by either the try-catch mechanism or the throws-declaration mechanism during compile time.

Two Types of Exception

- Unchecked exception:
  - In contrast to a checked exception, an unchecked exception represents an error in programming logic, not an erroneous situation that might reasonably occur during the proper use of an API.

  - For example, if we divide a number by 0, Java will throw ArithmeticException.

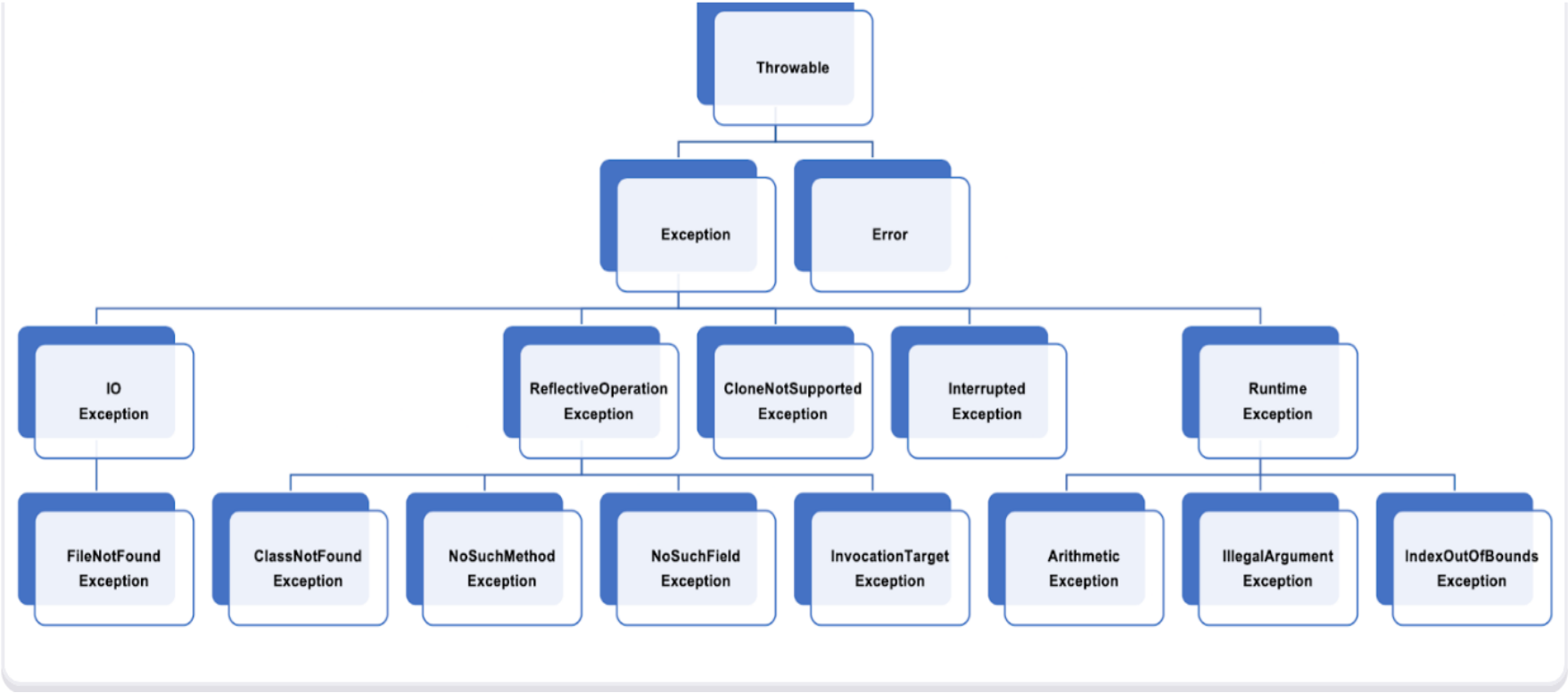
```
public class Main {
    public static void main(String[] args) {
        int a = 10, b = 0;
        System.out.println(a/b);
    }
}
```

Compare checked and unchecked exceptions

Compare checked vs. unchecked exceptions

Criteria	Unchecked exception	Checked exception
Purpose	Unanticipated errors in logic that show up at runtime	Anticipated problems associated with the normal use of an API
Ancestry	Includes RuntimeException	Does not include RuntimeException
Handling	Exception handling semantics are not required	Must be handled in a try-and-catch block, or be thrown by the invoking method
Extension	Can be customized by extending RuntimeException	Can be customized by extending java.lang.Exception
List of examples	NullPointerException, ClassCastException, ArithmeticException, DateTimeException, ArrayStoreException	ClassNotFoundException, SocketException, SQLException, IOException, FileNotFoundException

Hierarchy of Exception Classes



Catching specific/general-level exception

ExceptionDemo_1.java

```
1   public class ExceptionDemo_1 {
2     public static void main (String[] args)
3     {   int x=6, y=0;
4         try
5         { System.out.println(x/y);
6           // other statements
7         }
8         catch( ArithmeticException e)
9         {   System.out.println(e);
10          y=2;
11        }
12        finally
13        { System.out.println("Hello");
14          System.out.println(x/y);
15        }
16    }
17  }
```

Output - Chapter04 (run)

```
run:
java.lang.ArithmeticException: / by zero
Hello
3
BUILD SUCCESSFUL (total time: 0 seconds)
```

ExceptionDemo_1.java *

```
1   public class ExceptionDemo_1 {
2     public static void main (String[] args)
3     {   int x=6, y=0;
4         try
5         { System.out.println(x/y);
6           // other statements
7         }
8         catch(Exception e) // general exception
9         {   e.printStackTrace();
10          y=2;
11        }
12        finally
13        { System.out.println("Hello");
14          System.out.println(x/y);
15        }
16    }
17  }
```

Output - Chapter04 (run)

```
run:
Hello
java.lang.ArithmeticException: / by zero
3
        at ExceptionDemo_1.main(ExceptionDemo_1.java:5)
BUILD SUCCESSFUL (total time: 0 seconds)
```

Throwing exceptions in methods

May we intentionally throw an exception? → YES

```
1   public class ExceptionDemo_1 {
2     public int divide1(int a, int b) throws
3                           ArithmeticException
4     {  return a/b;
5     }
6     public int divide2(int a, int b)
7     {  if (b==0) throw new ArithmeticException
8                  ("Hey. Denominator:0");
9        return a/b;
10    }
11    public static void main (String[] args)
12    {  ExceptionDemo_1 obj= new ExceptionDemo_1();
13       try
14       { System.out.println(obj.divide1(6,0));
15       }
16       catch(Exception e) // general exception
17       {  System.out.println(e);
18       }
19    }
20  }
```

Output - Chapter04 (run)

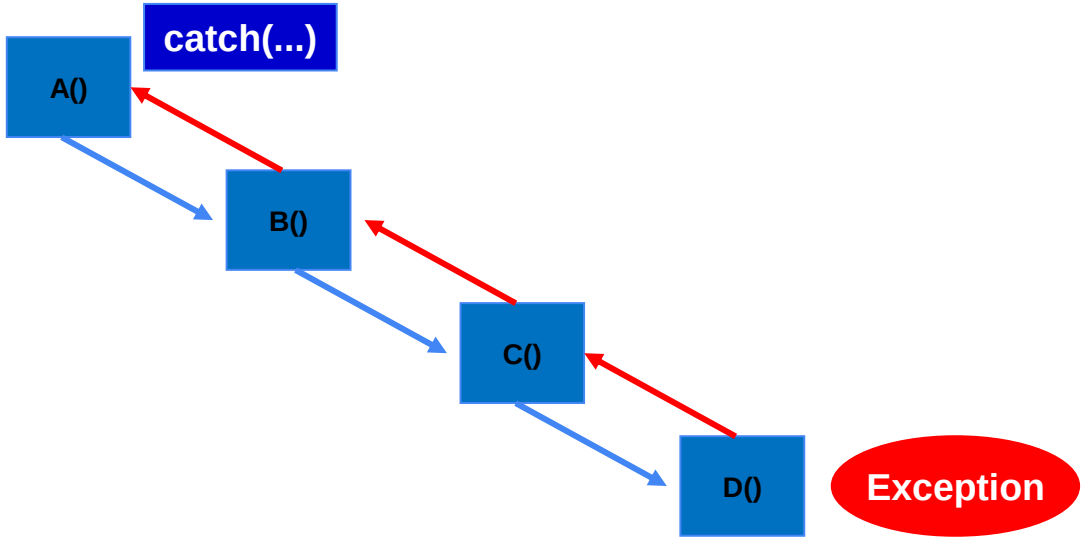
```
run:
java.lang.ArithmeticException: / by zero
BUILD SUCCESSFUL (total time: 0 seconds)
```

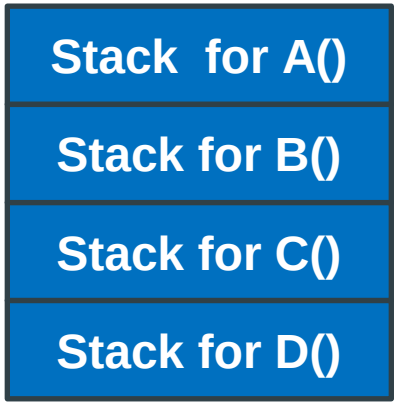
```
1   public class ExceptionDemo_1 {
2     public int divide1(int a, int b) throws
3                           ArithmeticException
4     {  return a/b;
5     }
6     public int divide2(int a, int b)
7     {  if (b==0) throw new ArithmeticException
8                  ("Hey. Denominator:0");
9        return a/b;
10    }
11    public static void main (String[] args)
12    {  ExceptionDemo_1 obj= new ExceptionDemo_1();
13       try
14       { System.out.println(obj.divide2(6,0));
15       }
16       catch(Exception e) // general exception
17       {  System.out.println(e);
18       }
19    }
20  }
```

Output - Chapter04 (run)

```
run:
java.lang.ArithmeticException: Hey. Denominator:0
BUILD SUCCESSFUL (total time: 0 seconds)
```

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

```java
public class ExceptionPropagate {
    public void mA()
    {
        mB();
    }
    public void mB()
    {
        mC();
    }
    public void mC()
    {
        System.out.println(5/0);
    }
    public static void main(String[]args){
        ExceptionPropagate obj= new ExceptionPropagate();
        obj.mA();
    }
}
```

Output - FirstPrj (run)

```
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
```

Catching Exceptions…

Using try…catch to input an integer 10<=n<=50

```
Scanner in = new Scanner(System.in);
boolean cont = true;
int n;
do {
   try {
        System.out.print("Enter a number: ");
        a = Integer.parseInt(in.nextLine());
        cont = false;
} catch (Exception e) {
        System.out.println("Required integer!");
     }
} while (cont == true|| n<10 || n>50);
```

The finally block

- A try block may optionally have a finally block associated with it.
- The code within a finally block is *guaranteed* to execute no matter what happens in the try/catch code that precedes it.
  - The try block executes to completion without throwing any exceptions whatsoever.
  - The try block throws an exception that is handled by one of the catch blocks.
  - The try block throws an exception that is ***not* handled by *any* of the catch blocks**

Nesting of try/catch blocks

- A try statement may be nested inside either the try or catch block of another try statement.

```
try {
  // Pseudo code.
  open a user-specified file
  }
  catch (FileNotFoundException e) {
       try {
           // Pseudo code.
           open a DEFAULT file instead ...
       }
       catch (FileNotFoundException e2) {
          // Pseudo code.
          attempt to recover ...
       }
}
```

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

Create your own exception class with it's constructor

```
class InvalidAge extends Exception{
    public InvalidAge(String mes) {
        super(mes);
    }
}
```

Creating Your Own Exception Classes

```
//Use it in some method
class MyClass{
    public void MyMethod(int a) throws InvalidAge{
        if(a<0)
            throw new InvalidAge("Age invalid!");
    }
}
```

Creating Your Own Exception Classes

```
//Using try-catch when this method is called
try {
        MyClass class1 = new MyClass();
        class1.MyMethod(-5);
  } catch (InvalidAge ex) {
        System.out.println(ex.getMessage());
  }
```

Thank you for your listening!

“Motivation is what gets you started.
Habit is what keeps you going!”

Jim Ryun