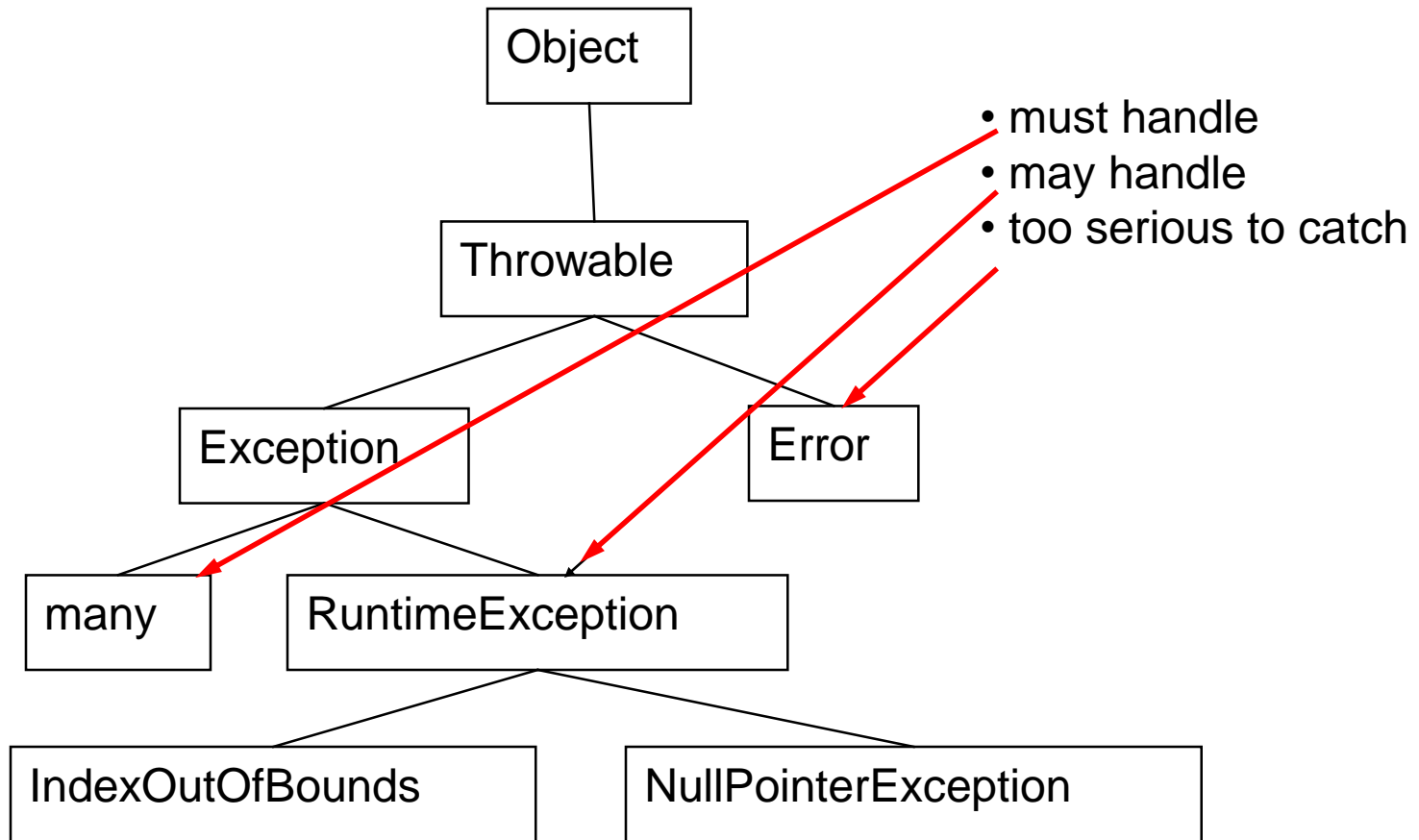


Java Exceptions

Intro to Exceptions

- What are exceptions?
 - Events that occur during the execution of a program that interrupt the normal flow of control.
- e.g. Opening a file, does not exist.

Exception Class hierarchy



Throwable hierarchy

- All exceptions extend the class `Throwable`, which splits into two branches:

Error and Exception

- `Error`: internal errors and resource exhaustion inside the Java runtime system. Little you can do.
- `Exception`: splits further into two branches. `RuntimeException` and other exceptions

Focus on the `Exception` branch

- Two branches of `Exception`
 - exceptions that derived from `RuntimeException`
 - examples: a bad cast, an out-of-array access
 - happens because errors exist in your program. Your fault.
 - those not in the type of `RuntimeException`
 - example: trying to open a malformed URL
 - program is good, other bad things happen. Not your fault.

Focus on the `Exception` branch

- Checked exceptions vs. unchecked exceptions
 - *Unchecked exceptions*: exceptions derived from the class `Error` or the class `RuntimeException`
 - *Checked exceptions*: all other exceptions that are not unchecked exceptions
 - If they occur, they must be dealt with in some way.
 - The compiler will check whether you provide exception handlers for checked exceptions which may occur

Approaches to handling an exception

1. Prevent the exception from happening
2. Catch it in the method in which it occurs, and either
 - a. Fix up the problem and resume normal execution
 - b. Rethrow it
 - c. Throw a different exception
3. Declare that the method throws the exception
4. With 1. and 2.a. the caller never knows there was an error.
5. With 2.b., 2.c., and 3., if the caller does not handle the exception, the program will terminate and display a stack trace

Example

```
public class myexception{  
    public static void main(String args[]){  
        try{  
            File f = new File("myfile");  
            FileInputStream fis = new FileInputStream(f);  
        }catch(FileNotFoundException ex){  
            File f = new File("Available File");  
            FileInputStream fis = new FileInputStream(f);  
        } finally{           // the finally block  
            } //continue processing here.  
    }  
}
```

- In this example we are trying to open a file and if the file does not exist we can do further processing in the catch block.
- The try and catch blocks are used to identify possible exception conditions. We try to execute any statement that might throw an exception and the catch block is used for any exceptions caused.
- If the try block does not throw any exceptions, then the catch block is not executed.
- The finally block is always executed irrespective of whether the exception is thrown or not.

Using throws clause

Use the throws to handle the exception in the calling function.

```
public class MyException{
    public static void main(String args[]){
        try{
            checkEx();
        } catch(FileNotFoundException ex){
        }
    }
    public void checkEx() throws FileNotFoundException{
        File f = new File("myfile");
        FileInputStream fis = new FileInputStream(f);
        //continue processing here.
    }
}
```

Using throws clause

- In this example, the main method calls the checkex() method and the checkex method tries to open a file, If the file is not available, then an exception is raised and passed to the main method, where it is handled.

Catching Multiple exceptions

```
public class myexception{
    public static void main(String args[]){
        try{
            File f = new File("myfile");
            FileInputStream fis = new FileInputStream(f);
        }
        catch(FileNotFoundException ex){
            File f = new File("Available File");
            FileInputStream fis = new FileInputStream(f);
        }catch(IOException ex){
            //do something here
        }
        finally{
            // the finally block
        }
        //continue processing here.
    }
}
```

Catching Multiple exceptions

- We can have multiple catch blocks for a single try statement.
- The exception handler looks for a compatible match and then for an exact match.
- In other words, in the example, if the exception raised was `myIOCustomException`, a subclass of `FileNotFoundException`, then the catch block of `FileNotFoundException` is matched and executed.
- If a compatible match is found before an exact match, then the compatible match is preferred.
- We need to pay special attention on ordering of exceptions in the catch blocks, as it can lead to mismatching of exception and unreachable code.
- We need to arrange the exceptions from specific to general.

Exception Handling Basics

- Three parts to Exception handling
 1. claiming exception
 2. throwing exception
 3. catching exception
- A method has the option of *throwing* one or more exceptions when specified conditions occur. This exception must be *claimed* by the method. Another method calling this method must either *catch* or re*throw* the exception. (unless it is a RuntimeException)

Claiming Exceptions

- Method declaration must specify every exception that the method potentially throws

MethodDeclaration throws Exception1,
Exception2, ..., ExceptionN

- Exceptions themselves are concrete subclasses of Throwable and must be defined and locatable in regular way.

Throwing Exception

- To throw an Exception, use the *throw* keyword followed by an instance of the Exception class

```
void foo() throws SomeException{  
    if (whatever) {...}  
    else{ throw new SomeException(...)}  
}
```

- Note that if a method foo has a throw clause within it, that the Exception that is thrown (or one of its superclasses) must be claimed after the signature.

Catching Exceptions

- The third piece of the picture is catching exceptions.
- This is what you will do with most commonly, since many of java's library methods are defined to throw one or more runtime exception.
- Catching exceptions:
 - When a method is called that throws an Exception e.g. `SomeException`, it must be called in a try-catch block:

```
try
{
    foo();
}
catch(SomeException se)
{
    ...
}
```


Example1

```
import java.io.*;
```

```
public class Exception1{  
    public static void main(String[] args){  
        InputStream f;  
        try{  
            f = new FileInputStream("foo.txt");  
        }  
        catch(FileNotFoundException fnfe){  
            System.out.println(fnfe.getMessage());  
        }  
    }  
}
```

Example2

```
import java.io.*;
public class Exception2{
    public static void main(String[] args){
        InputStream fin;
        try{
            fin = new FileInputStream("foo.txt");
            int input = fin.read();
        }
        catch(FileNotFoundException fnfe){
            System.out.println(fnfe.getMessage());
        }
        catch(IOException ioe){
            System.out.println(ioe.getMessage());
        }
    }
}
```

Catching exceptions

- Checked exceptions handling is strictly enforced. If you invoke a method that lists a checked exception in its throws clause, you have three choices
 1. Catch the exception and handle it
 2. Declare the exception in your own `throws` clause, and let the exception pass through your method (you may have a `finally` clause to clean up first)
 3. Catch the exception and map it into one of your exceptions by throwing an exception of a type declared in your own `throws` clause

try/catch clause (1)

- If no exception occurs during the execution of the statements in the `try` clause, it finishes successfully and all the `catch` clauses are skipped
- If any of the code inside the `try` block throws an exception, either directly via a `throw` or indirectly by a method invoked inside it
 1. The program skips the remainder of the code in the `try` block
 2. The `catch` clauses are examined one by one, to see whether the type of the thrown exception object is compatible with the type declared in the `catch`.
 3. If an appropriate `catch` clause is found, the code inside its body gets executed and all the remaining `catch` clauses are skipped.
 4. If no such a `catch` clause is found, then the exception is thrown into an outer `try` that might have a `catch` clause to handle it
- A `catch` clause with a superclass `exceptionType` cannot precede a `catch` clause with a subclass `exceptionType`

finally clause

- You can use a `finally` clause without a `catch` clause
- Sometimes the `finally` clause can also throw an exception

Example

```
public boolean searchFor(String file,  
String word)  
    throws StreamException  
{  
    Stream input = null;  
    try {  
        some code which may throw an  
StreamException  
    } finally {  
        input.close(); // this may throw anIOException  
    }  
}
```

finally clause

- You may want to do some actions whether or not an exception is thrown. `finally` clause does this for you

```
Graphics g = image.getGraphics();
try {
    //1
    code that might throw exceptions
    //2
} catch (IOException e) {
    //3
    show error dialog (// some code which may throw
exceptions)
    //4
} finally {
    g.dispose(); (// some code which will not throw
exceptions)
    //5
} //6
```

- No exception is thrown: 1, 2, 5, 6
- An exception is thrown and caught by the `catch` clause
 - The `catch` clause doesn't throw any other exception: 1, 3, 4, 5, 6
 - The `catch` clause throws an exception itself: 1, 3, 5, and the exception is thrown back to the caller of this method
- An exception is thrown but not caught by the `catch` clause: 1, 5, and the exception is thrown back to the caller of this method

Creating new exception types

- Exceptions are objects. New exception types should extend `Exception` or one of its subclasses
- Why creating new exception types?
 1. describe the exceptional condition in more details than just the string that `Exception` provides

E.g. suppose there is a method to update the current value of a named attribute of an object, but the object may not contain such an attribute currently. We want an exception to be thrown to indicate the occurring of if such a situation

```
public class NoSuchAttributeException extends Exception {  
    public String attrName;  
    public NoSuchAttributeException (String name) {  
        super("No attribute named \"" + name + "\" found");  
        attrName = name;  
    }  
}
```

2. the type of the exception is an important part of the exception data – programmers need to do some actions **exclusively** to one type of exception conditions, not others

Throwing Your Own Exceptions

- There are many exceptions and you should get to know them

