Java Exceptions

Rui S. Moreira

Some useful links:
http://java.sun.com/docs/books/tutorial/essential/exceptions

Problems during program execution

- During execution, programs may find run-time **exceptions** or **errors** conditions
- When executing, some classes may generate (**throw**) exceptions or errors which sinalize a problem
- Developers my **catch** those exceptions and errors and (if possible) write code to recover

Exception example

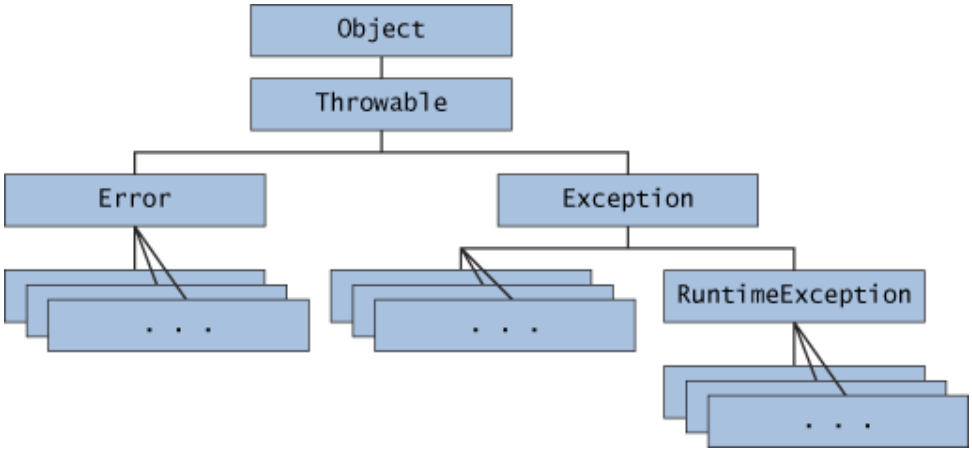
```
public class TestExceptionApp {

  public static void main(String args[]){

        //Declare/create array with 3 elements (0..2)
        String astr[]={"um", "dois", "tres"};

        // Print array elements
        for (int i=0; i<4; i++){
            // The following line throws an Exception!
            // Why? (try to go beyond array limit)
            System.out.println("astr["+i+"] = "+astr[i]);
        }
  }
}
```


Class hierarchy (java.lang package)



Throwable:

super-class of all objects that may be **thrown** & **caught** through exception mechanisms

Errors

- **Error**
  - Unrecoverable run-time errors, e.g.,
    - `VirtualMachineError, OutOfMemoryError, StackOverflowError,` *etc.*
  - Define serious problems, i.e., situations that compromise the execution of the rest of the program
  - Programs can find errors during their execution from which must/cannot recover

Exceptions

- **RunTimeException**
  - recoverable design/implementation problems, e.g.,
    - `NullPointerException, ClassCastException, ArithmeticException,` *etc.*
  - exceptional conditions that should not happen in correctly implemented programs; usually do not compromise the execution of the rest of the program
  - developers can provide code to identify these situations, e.g. usually just reporting exception message

Exceptions

- **Other Exceptions**
  - recoverable run-time problems usually associated with execution context/environment, e.g.
    - `IOException, FileNotFoundException, MalformedUrlException,` *etc.*
  - exceptional (although predictable) conditions that programs can find during their execution and that may be recovered
  - developers can provide code to handle these situations, i.e., recover control, correct the situation (if possible) or retry given operation and continue program execution

Catch Exception example

```
public class TestExceptionApp {
  public static void main(String args[]){
    try { // Inside try we put the "protected-code"
       //Declare/create array with 3 elements (0..2)
       String astr[]={"um", "dois", "tres"};
       // Print array elements
       for (int i=0; i<4; i++){
            // This line throws ArrayIndexOuOfBoundException
            System.out.println("astr["+i+"] = "+astr[i]);
       }
    } catch (Exception e){ // Inside catch we put code for
       // handling the exception, e.g., print out the
       // exception info/message
       System.out.println("main(): exception caught "+e);
    } finally { // Inside finally we put code always executed
       // whether an exception is thrown/caught or not
       // (except if System.exit() called inside protected-code)
    }
  }
}
```

Common RunTimeExceptions

- **`ArithmeticException`**:
  - division by 0 (zero), e.g., `float x=0.0f, inv=1/x;`
- **`NullPointerException`**:
  - use a `null` reference variable, e.g., `Ponto p; p.setX(7);`
- **`ArrayIndexOutOfBoundException`**:
  - reference array element beyond array length
- **`NegativeArraySizeException`**:
  - create array with negative size
- **`SecurityException`**:
  - downloaded classes trying to access local files or trying to open a socket to another computer
- **`NumberFormatException`**:
  - read/convert illegal digits, e.g., `Integer.parseInt("10s");`
- **`ClassCastException`**:
  - wrong/illegal class type casting, e.g., given `ArrayList alist` storing several `Manager` objects throws an exception when casting:

```
Salesman s = (Salesman)alist.get(i);
```


Rule – **declare** or **handle** exceptions

- If we are implementing a method `readfile()` that may cause/detect an exception we must do one of 2 things:
  - **Declare** the exception:
    - announce that the method may throw an exception which should be caught by who uses it…

```
public void readfile() throws IOException {
        /*…*/
}
```

  - **Handle** the exception:
    - write the code to detect/catch and then handle the exception…

```
try {
   /* protected code here */
} catch (Exception e) {
   /* handle code here */
}
```

Create New Users' Exceptions

- Developers may create their own exception classes, e.g. by extending the *Exception* class, e.g.
  - Suppose we want a *OverWithdrawException* to alert situations where account owners try to take/withdraw more money than available from a given Account

```
public class OverWithdrawException extends Exception
{
   public OverWithdrawException(){
          // Automatically calls super();
   }
   public OverWithdrawException(String problem){
          super(problem);
   }
   //other constructors...
}
```


Throw User's Exceptions

- Developers may then launch/throw new *OverWithdrawException* whenever needed:

```
public class AccountSafe extends Account {

  public double withdraw(double amount) throws OverWithdrawException{
    if (super.getBalance() > amount) {
      setBalance(getBalance() - amount);
      return getBalance();
    } else {
      throw new
              OverWithdrawException("not enough money to withdraw!");
    }
  }
}
```

Exercise

- Create `IllicitDepositException` that alerts when account owners tries to make a negative deposit!?

```
public class IllicitDepositException extends Exception {
   // Code here (similar to OverWithdrawException)
}
```

- Now change `deposite()` method of `AccountSafe` class to `throw` this new `IllicitDepositException`

```
public double deposit(double amount) {
   // throw IllicitDepositException when amount < 0.0
}
```


Catch Exceptions Individually

- A single `try-catch-finally` clause may be used for catching individual exceptions:

```
public static void main(String args[]) {
   Client c = new Client("Alex", "Sesamo street");
   AccountSafe as = new AccountSafe("00730111", 5000.0, c);
   c.addAccount(as);
   try {
     as.withdraw(6000.0); // Will trigger an exception
   } catch (OverWithdrawException owe) {
     System.out.println(owe.toString());
   }
   try {
     as.deposit(-500.0); // also triggers an exception
   } catch (IllicitDepositException ide) {
     ide.printStackTrace();
   }
}
```

Catch Multiple Exceptions

- The same `try-catch-finally` clause may have several catches for multiple individual exceptions:

```
public static void main(String args[]) {
   try {
     Client c = new Client("Alex", "Sesamo street");
     AccountSafe as = new AccountSafe("00730111", 5000.0, c);
     c.addAccount(as);

     as.withdraw(6000.0); //May trigger an exception
     as.deposit(-500.0); //Also triggers an exception

   } catch (OverWithdrawException owe) {
     System.out.println(owe.toString());
   } catch (IllicitDepositException ide) {
     ide.printStackTrace();
   }
}
```


Catch Multiple Exceptions

- The same `try-catch-finally` clause may be used for catching several generic exceptions:

```
public static void main(String args[]) {
   try {
     Client c = new Client("Alex", "Sesamo street");
     AccountSafe as = new AccountSafe("00730111", 5000.0, c);
     c.addAccount(as);

     as.withdraw(6000.0); //May trigger an exception
     as.deposit(-500.0); //Also triggers an exception

   } catch (Exception e) {
     //catches any exception
     System.out.println(e.toString());
   }
}
```