Exception Handling

CS 180
Sunil Prabakar
Department of Computer Science
Purdue University
When things go wrong

- Good programs should be robust -- i.e., they should be able to handle exceptional situations.
- What happens if we are trying to input an integer value and the user enters *ten*, or *3.45*?
- A good program should tell the user to re-enter a valid integer.
- So far, this situation would result in the termination of our program when we execute `Integer.parseInt()` on this invalid string.
- How do we prevent this?
Handling errors

- One idea is to use if-then style tests whenever we expect that an error may arise.
- This is the style in C -- return values can signal the existence of an error.
- But this is clumsy, and inelegant.
- In Java, the \textit{exception handling mechanism} is used instead.
- Unexpected (or unusual) cases are handled by a special type of control flow.
Exceptions

- An *exception* is used to indicate that something unusual (that prevents regular processing) has occurred.

- When an exception occurs, or is *thrown*, an Exception object is created, and the normal sequence of flow is terminated.

- An exception handling mechanism is invoked which is responsible for handling or *catching* the thrown exception.
Uncaught Exceptions

- When an exception is thrown, and the program does not specify how to catch it, it causes the program to terminate:

```java
import javax.swing.*;
public class ReadInt{
    public static void main(String[] args){
        String inputStr;
        int i;
        inputStr = JOptionPane.showInputDialog(null, "Enter Deposit Amount");
        i = Integer.parseInt(inputStr);
    }
}
```
Catching an exception

```java
String inputStr;
int i;
inptStr = JOptionPane.showInputDialog(null, "Enter an integer");
try{
i = Integer.parseInt(inputStr);
} catch (Exception e){
System.out.println("Invalid integer");
}
```
Exception control-flow

try{
    stmt;
}
catch (Exception e){
    
}

Exception is thrown when executing this statement.
Exception object

- An exception is thrown by creating an Exception object.
- The exception object is passed to the catch block as a parameter.
- It contains details about the actual exception that was thrown.

```java
try {
    . . .
} catch (Exception e) {
    . . .
}
e is a catch block parameter corresponding to the exception object.
```
Exception object

- The exception object contains details about the exception.
  - The `getMessage()` method simply returns a string of text that describes the exception.
  - The `printStackTrace()` method gives us the order (and line numbers) in which methods had been called when the exception took place.
    - In reverse order of the calls
    - The last method call is listed first, main is last.
public class SafeInputHelper {
  static int getInt(String msg) {
    String str;
    int i;
    do {
      str = JOptionPane.showInputDialog(null, msg);
      try {
        i = Integer.parseInt(str);
        return i;
      } catch (NumberFormatException e) {
        System.out.println("Invalid integer format, please re-enter");
      }
    } while (true);
  }
}
The Exception Hierarchy

```
Exception

IOExcepton
RunTimeException
SQLException

NullPointerException
IllegalArgumentExcepton
ArithmeticException

NumberFormatException

Many more.
See Java API
```
Multiple catch Blocks

- If more than one type of exception can take place, we may want to handle each one differently.
- A single try-catch statement can include multiple catch blocks, one for each type of exception.
- Only the first matching catch block is executed.
- Matching is based on the class of the exception.
- Make sure to list classes lower in the hierarchy before listing classes higher up.
Multiple catch Blocks

```java
try {
    . . .
    i = Integer.parseInt(inputStr);
    . . .
} catch (NumberFormatException e) {
    . . . // code to handle NumberFormatExceptions.
} catch (NullPointerException e) {
    . . . // code to handle NullPointerExceptions.
} catch (Exception e) {
    . . . // code to handle all other exceptions.
}
```
Terminating a program

- It is possible to terminate a program at any point in its execution (maybe because a very serious error has occurred).
- This is achieved by calling `System.exit(0)`
- This call takes any integer value as a parameter.
- The program is immediately terminated.
The **finally** Block

- There are situations where we need to take certain actions regardless of whether an exception is thrown or not.
- We place statements that must be executed regardless of exceptions, in the **finally** block.
- Commonly used to perform cleanup (e.g., closing disconnecting from a database, or closing a network connection)
Exception control-flow

No exception

finally block is always executed.

Exception thrown

try{
    ...
    stmt;
    ...
} catch (Exception e){
    ...
}
finally {
    ...
}

Exception is thrown when executing this statement.
Salient points

- If multiple catch blocks are defined they are tested in order -- only the first that matches the thrown exception gets executed.
  - List them from more specific to general.
  - CAUTION: if A is a subclass of B, then an exception of class A is also an exception of class B!

- Even if there is a return from the try or catch blocks, the finally block is executed before returning!

- If no matching catch block is found for an exception, the finally block gets executed
Caution: order of catch blocks

```java
try {
    ... 
    i = Integer.parseInt(inputStr);
    ... 
} catch (Exception e) {
    ... // code to handle general exceptions.
} catch (NullPointerException e) {
    ... // code to handle NullPointerExceptions.
} catch (NumberFormatException e) {
    ... // code to handle NumberFormatExceptions.
}
```

Will never get executed!
Propagating exceptions

- If an exception occurs and there is no matching catch block, then the exception is propagated.
  - control passes to the caller
  - if the caller has no matching catch block, the same happens
  - eventually, if the main method does not handle the exception, the runtime system handles it.
Exception handling

```java
public static void main(String[] args){
    ...
    a.methodA();
    ...
}

public void methodA(){
    try{
        methodB();
    } catch (NumberFormatException e){
        . . .
    }
    . . .
}

public void methodB(){
    stmt;
}
```

NumberFormatException is thrown when executing this statement.

Exception is handled.

Exception is propagated.
Exception handling

```java
public static void main(String[] args){
    ...
    a.methodA();
    ...
}
```

**Exception handling**

```java
public void methodA(){
    try{
        methodB();
    } catch (NumberFormatException e){
        ...
    }
    ...
}
```

```java
public void methodB(){
    stmt;
}
```

NullPointerException is thrown when executing this statement.

Program is terminated.

Exception is propagated.

Exception is propagated.
Types of exceptions

Two main types of exceptions

- **Checked** exceptions
- **Unchecked** exceptions.

Unchecked exceptions are those that can be thrown during the normal operation of the Java Virtual Machine

- Captured under the RuntimeException class in the hierarchy.
- `NullPointerException`, `ArithmeticException`, `IndexOutOfBoundsException`, etc.
Types of exceptions (cont.)

- Unchecked exceptions need not be explicitly handled (as we have done so far)
  - If unhandled, will lead to program termination.

- Checked exceptions must be explicitly handled by the program.
  - Any method that could result in a checked exception being thrown must either:
    - Handle it with a `try-catch` block, OR
    - Propagate and explicitly declare this possibility.
A method that propagates an unchecked exception must declare this possibility:
- the method header must include the reserved word `thrown` followed by a list of the classes of exceptions that may be propagated
- declaring runtime exceptions is optional

```java
public int accessDB() throws SQLException {
    // code that accesses some database
    ...
}
```
Handling Unchecked Exceptions

`parseInt` throws `NumberFormatException` (see API).

```java
void methodA() {
    try {
        int i = Integer.parseInt(s);
    } catch (NumberFormatException e) {
        . . .
    }
}
```

```java
void methodB() throws NumberFormatException {
    int i = Integer.parseInt(s);
}
```

Propagators

Catcher
Handling Checked Exceptions

Scanner(File) throws FileNotFoundException (see API).

```java
void methodA() {
    try {
        scanner = new Scanner(f);
    } catch (FileNotFoundException e) {
        // ...
    }
}
```

```java
void methodB() throws FileNotFoundException {
    scanner = new Scanner(f);
}
```

Catcher

Propagator
We can throw an exception at any point in our code.

To do this, we create an exception object and throw it.

If this is a checked exception, we must declare that we throw this exception (unless we catch the exception).

```java
public float squareRoot(float value) throws Exception {
    . . .
    if (value<0) 
        throw new Exception("Imaginary numbers not yet supported");
    . . .
}
```
Defining Custom Exceptions

- Should only need to do this if we want to capture extra information, or if you want to handle this class in a special fashion.
- Must extend an exception class. Good idea to extend the Exception class.
- Define a default constructor.
- Call the parent’s constructor as the first call in the constructor for the new exception: `super(msg);`
Let us assume that our application often needs to input several streams of integers in ascending order with a minimum jump between values.

- each stream has its own starting point and minimum jump.

Create a helper class to input such values: AscendingInputHelper

This class throws a new type of exception that signals that the ascending rule was violated: AscendingException.
class AscendingInputHelper {
    int lastValue; // the previous input for this sequence
    int minimumIncrement; // the minimum increment required

    public AscendingInputHelper(int start, int minInc) {
        lastValue = start;
        minimumIncrement = minInc;
    }

    // Propagate exception
    public int getNextInt() throws AscendingException {
        int i;
        i = SafeInputHelper.getInt(); // Get the next integer from the user
        if (i < lastValue + minimumIncrement) // if invalid ascend, throw exception
            throw new AscendingException(i, lastValue, minimumIncrement);
        lastValue = i;
        return i;
    }
}

class AscendingInputHelper {
  int lastValue; // the previous input for this sequence
  int minimumIncrement; // the minimum increment required

  public AscendingInputHelper(int start, int minInc) {
      lastValue = start;
      minimumIncrement = minInc;
  }

  // Propagate exception
  public int getNextInt() throws AscendingException {
      int i;
      i = SafeInputHelper.getInt(); // Get the next integer from the user
      if (i < lastValue + minimumIncrement) // if invalid ascend, throw exception
          throw new AscendingException(i, lastValue, minimumIncrement);
      lastValue = i;
      return i;
  }
}
public class AscendingException extends Exception {
    private int lastEntry, errorEntry, minimumIncrement;
    private static final String ERROR_MSG = "Invalid Ascending Sequence";

    public AscendingException(int badEntry, int last, int inc) {
        this(AscendingException.ERROR_MSG, badEntry, last, inc);
    }

    public AscendingException(String msg, int badEntry, int last, int inc) {
        super(msg);
        errorEntry = badEntry;
        lastEntry = last;
        minimumIncrement = inc;
    }

    public int getLastEntry() { return lastEntry; }
    public int getErrorEntry() { return errorEntry; }
    public int getMinimumIncrement() { return minimumIncrement; }
}
class TestAscendingInput {
    public static void main(String args[]) {
        int total = 0, newValue;

        AscendingInputHelper ascInput = new AscendingInputHelper(0, 3);
        while (true) {
            try {
                newValue = ascInput.getNextInt();
                total += newValue;
            } catch (AscendingException e) {
                JOptionPane.showMessageDialog(null, "Error with order of input\n" +
                        e.getMessage() + "\nEntered value: " + e.getErrorEntry() +
                        "\nPrevious value: " + e.getLastEntry() +
                        "\nMinimum Increment required: " + e.getMinimumIncrement());
                System.out.println("Total of valid inputs: " + total);
                System.exit(0);
            }
        }
    }
}
Assertions

- Exceptions handle unexpected behavior during execution.
- Sometimes programs fail due to logical errors in the code.
- Assertions are a mechanism available to detect logical errors.
- An assertion is essentially a sanity check regarding the state of data at a given point in the program.
Assertions

- The syntax for the `assert` statement is:

  ```python
  assert <boolean expression>;
  ```

  where `<boolean expression>` represents the condition that must be true if the code is working correctly.

- If the expression results in `false`, an `AssertionError` (a subclass of `Error`) is thrown.
public double deposit(double amount) {
    double oldBalance = balance;
    balance += amount;
    assert balance > oldBalance;
}

public double withdraw(double amount) {
    double oldBalance = balance;
    balance -= amount;
    assert balance < oldBalance;
}
Second Form

- The assert statement may also take the form:

```
assert <boolean expression>: <expression>;
```

where `<expression>` represents the value passed as an argument to the constructor of the `AssertionError` class. The value serves as the detailed message of a thrown exception.
public double deposit(double amount) {
    double oldBalance = balance;
    balance += amount;
    assert balance > oldBalance :
        "Serious Error – balance did not " +
        " increase after deposit";
}
class AscendingInputAssert{
    public static void main(String args[])
    {
        int minIncrement = 3, lastValue = 0, newValue, total = 0;

        while(true)
        {
            try{
                newValue = SafeInputHelper.getInt("Enter next value in sequence");
                assert (newValue-lastValue)>=minIncrement;
                total += newValue;
                lastValue = newValue;
            } catch (AssertionError e) {
                JOptionPane.showMessageDialog(null, "Invalid increment: terminating");
                System.out.println("Total of valid inputs = " + total);
                System.exit(0);
            }
        }
    }
}
Compiling Programs with Assertions

- Before Java 2 SDK 1.4, the word `assert` is a valid non-reserved identifier. In version 1.4 and after, the word `assert` is treated as a regular identifier to ensure compatibility.

- To enable the assertion mechanism, compile the source file using

  ```bash
  javac -source 1.4 <source file>
  ```
Running Programs with Assertions

- To run the program with assertions enabled, use

  `java -ea <main class>`

- If the `-ea` option is not provided, the program is executed without checking assertions.
Different Uses of Assertions

- **Precondition assertions** check for a condition that must be true before executing a method.

- **Postcondition assertions** check conditions that must be true after a method is executed.

- A **control-flow invariant** is a third type of assertion that is used to assert the control must flow to particular cases.