Chapter 8

Exceptions and Assertions

CS 180
Sunil Prabhakar
Department of Computer Science
Purdue University
Objectives

This week we will study how to
- Improve the reliability of code by incorporating exception-handling and assertion mechanisms.
- Write methods that propagate exceptions.
- Implement try-catch blocks for catching and handling exceptions.
- Write programmer-defined exception classes.
- Distinguish between checked and unchecked, or runtime, exceptions.
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, a situation such as this 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 exception handling mechanism is used instead.
- Unexpected (or unusual) cases are handled by a special type of control flow.
An exception represents an error condition that can occur during the normal course of program execution.

When an exception occurs, or is thrown, the normal sequence of flow is terminated. The exception-handling routine is then executed; we say the thrown exception is caught.
String inputStr;
int    age;

inputStr = JOptionPane.showInputDialog(null, "Age:");
age      = Integer.parseInt(inputStr);

Error message for invalid input

```
java.lang.NumberFormatException: ten
   at java.lang.Integer.parseInt(Integer.java:405)
   at java.lang.Integer.parseInt(Integer.java:454)
   at Ch8Sample1.main(Ch8Sample1.java:20)
```
inputStr = JOptionPane.showInputDialog(null, "Age:");

try {
    age = Integer.parseInt(inputStr);
} catch (NumberFormatException e) {
    JOptionPane.showMessageDialog(null, "'" + inputStr + "' is invalid
    Please enter digits only");
}
try-catch Control Flow

Exception

try {
  <t-stmt-1>
  <t-stmt-2>
  <t-stmt-3>
  Assume <t-stmt-3> throws an exception.
  <t-stmt-4>
  ...
  <t-stmt-n>
}
  catch (Exception e) {
    <c-stmt-1>
    ...
    <c-stmt-m>
  }
  <next stmt>

No Exception

try {
  <t-stmt-1>
  <t-stmt-2>
  <t-stmt-3>
  <t-stmt-4>
  ...
  <t-stmt-n>
}
  catch (Exception e) {
    <c-stmt-1>
    ...
    <c-stmt-m>
    Statements in the catch block are skipped.
  }
  <next stmt>

All statements in the try block are executed.
Exception object

- An exception is thrown by creating an exception object.
- Control passes to a matching catch block which is given the exception object as a parameter.
- Matching based upon class of exception.

```java
try {
    . . .
} catch (NumberFormatException e){
    System.out.println(e.getMessage());
    System.out.println(e.printStackTrace());
}
```

`e` is a catch block parameter corresponding to the exception object.
Exception information

- 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.
  - This is in reverse order of the calls
  - The last method call is listed first, main is last.
  - Hence the “stack” (like a stack of books).

```
java.lang.NumberFormatException: For input string: “ten”
at java.lang.Integer.parseInt(Integer.java:405)
at java.lang.Integer.parseInt(Integer.java:454)
at Ch8Sample1.main(Ch8Sample1.java:20)
```
Multiple `catch` Blocks

- A single try-catch statement can include multiple catch blocks, one for each type of exception.

```java
try {
    . . .
    age = Integer.parseInt(inputStr);
    . . .
    val = cal.get(id); //cal is a GregorianCalendar
    . . .
} catch (NumberFormatException e){
    . . .
} catch (ArrayIndexOutOfBoundsException e){
    . . .
}
```
**Multiple catch Control Flow**

**Exception**

```
try {
    <t-stmt-1>
    <t-stmt-2>
    <t-stmt-3>
    <t-stmt-4>
    ...
    <t-stmt-n>
}
<catch-block-1>
<catch-block-2>
<catch-block-3>
...
<catch-block-m>

<next stmt>
```

Assume `<t-stmt-3>` throws an exception and `<catch-block-3>` is the matching block.

Statements in the matching catch block are executed.

**No Exception**

```
try {
    <t-stmt-1>
    <t-stmt-2>
    <t-stmt-3>
    <t-stmt-4>
    ...
    <t-stmt-n>
}
<catch-block-1>
<catch-block-2>
<catch-block-3>
...
<catch-block-m>

<next stmt>
```

All statements in the try block are executed and throw no 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.
try-catch-finally Control Flow

Exception

```java
try {
    <t-stmt-1>
    . . .
    <t-stmt-i>
    . . .
    <t-stmt-n>
}
<catch-block-1>
. . .
<catch-block-i>
. . .
<catch-block-m>
finally
. . .
<next stmt>
```

Assume `<t-stmt-i>` throws an exception and `<catch-block-i>` is the matching block.

No Exception

```java
try {
    <t-stmt-1>
    . . .
    <t-stmt-i>
    . . .
    <t-stmt-n>
}
<catch-block-1>
. . .
<catch-block-i>
. . .
<catch-block-m>
finally
. . .
<next stmt>
```

finally block is executed.
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.
Propagating Exceptions

- Instead of catching a thrown exception by using the try-catch statement, we can propagate the thrown exception back to the caller of our method.

- Any method that creates or passes on an exception must declare this possibility.

- For this, the method header must include the reserved word `throws` followed by a list of the classes of exceptions that may be propagated.

```java
public int getAge( ) throws NumberFormatException {
    . . .
    int age = Integer.parseInt(inputStr);
    . . .
    return age;
}
```
Throwing Exceptions

- We can write a method that throws an exception directly, i.e., this method is the origin of the exception.
- Use the `throw` keyword to create a new instance of the Exception or its subclasses.
- The method header includes the reserved word `throws`.

```java
public void doWork(int num) throws Exception {
    . . .
    if (num != val) throw new Exception("Invalid val");
    . . .
}
```
private static final String DEFAULT_MESSAGE = "Your age:"

public AgeInputVer3() {
}

public int getAge() {
    return getAge(DEFAULT_MESSAGE);
}

public int getAge(String prompt) {
    String inputStr;
    int age;
    while (true) {
        inputStr = JOptionPane.showInputDialog(null, prompt);
        try {
            age = Integer.parseInt(inputStr);
            if (age < 0) {
                throw new Exception("Negative age is invalid");
            }
            return age; //input okay so return the value & exit
        } catch (NumberFormatException e) {
            JOptionPane.showMessageDialog(null, "'" + inputStr + "' is invalid\n" + "Please enter digits only");
        } catch (Exception e) {
            JOptionPane.showMessageDialog(null, "Error: " + e.getMessage());
        }
    }
}
public static void main(String[] args) {
    GregorianCalendar today;
    int age, thisYear, bornYr, answer;

    AgeInputVer3 input = new AgeInputVer3();
    age = input.getAge("How old are you?");

    today = new GregorianCalendar();
    thisYear = today.get(Calendar.YEAR);

    bornYr = thisYear - age;

    answer = JOptionPane.showConfirmDialog(null,
        "Already had your birthday this year?",
        '',
        JOptionPane.YES_NO_OPTION);

    if (answer == JOptionPane.NO_OPTION) {
        bornYr--;
    }
    JOptionPane.showMessageDialog(null, "You are born in " + bornYr);
}
Exception Thrower

- When a method may throw an exception, either directly or indirectly, we call the method an *exception thrower*.

- Every exception thrower must be one of two types:
  - catcher
  - propagator
Types of Exception Throwers

- An *exception catcher* is an exception thrower that includes a matching `catch` block for the thrown exception.

- An *exception propagator* does not contain a matching `catch` block.

- A method may be a catcher of one exception and a propagator of another.
Sample Call Sequence

Method A

```java
try {
    B();
} catch (Exception e) {
    . . .
}
```

Method B

```java
try {
    C();
} catch (Exception e) {
    . . .
}
```

Method C

```java
D();
```

Method D

```java
if (cond) {
    throw new Exception();
}
```

Stack Trace

A → B → C → D → C → B → A
Exception handling

Once an exception is thrown:

- If the exception was thrown within a try block, and there is a matching catch block, then control passes to that catch block.
- Otherwise, a matching catch block is searched in reverse order of calls (stack).

- If a matching catch block is found it is executed and then control goes to the statement that follows that try-catch statement.
- Otherwise the system will handle the exception.
Exception Types

- All types of thrown errors are instances of the **Throwable** class or its subclasses.

- Serious errors are represented by instances of the **Error** class or its subclasses. These are usually not caught by the program and result in termination of execution.

- Exceptional cases that common applications should handle are represented by instances of the **Exception** class or its subclasses.
There are over 60 classes in the hierarchy.

See Java API
There are two types of exceptions:
- Checked.
- Unchecked.

A **checked exception** is an exception that is checked at compile time.

All other exceptions are **unchecked**, or **runtime, exceptions**. As the name suggests, they are detected only at runtime.

The **Error** and **RuntimeException** classes (and descendents) are unchecked. All others are checked.
Different Handling Rules

- When calling a method that can throw checked exceptions
  - use the **try-catch** statement and place the call in the **try** block, or
  - modify the method header to include the appropriate **throws** clause.

- When calling a method that can throw runtime exceptions, it is **optional** to use the try-catch statement or modify the method header to include a throws clause.
Handling Checked Exceptions

**Caller A (Catcher)**

```java
void callerA() {
    try {
        doWork();
    } catch (Exception e) {
        ...
    }
}
```

**doWork throws Exception**

```java
public void doWork throws Exception {
    ...
    throw new Exception();
    ...
}
```

**Caller B (Propagator)**

```java
void callerB() throws Exception {
    ...
    doWork();
    ...
}
```
Handling RunTime Exceptions

Caller A (Catcher)

```java
void callerA() {
    try {
        doWork();
    } catch (RunTimeException e) {
        …
    }
}
```

doWork throws RunTimeException

Caller B (Propagator)

```java
void callerB() throws RunTimeException {
    …
    doWork();
    …
}
```

Caller C (Propagator)

```java
void callerC() {
    …
    doWork();
    …
}
```

This is the most common style for runtime exceptions. CallerC implicitly propagates the exception.
Propagation Example

- Suppose the user of AgeInputVer sets upper and lower limits on ages.
- Throw an exception when this is violated.
- Do not catch it within AgeInputVer since it is a user-defined condition -- let the user handle the exception!
- AgeInputVer4 -- also checks for valid bounds.
public AgeInputVer4() throws IllegalArgumentException {
    setBounds(DEFAULT_LOWER_BOUND, DEFAULT_UPPER_BOUND);
}

public AgeInputVer4(int low, int high) throws IllegalArgumentException {
    if (low > high) {
        throw new IllegalArgumentException(  
            "Low (" + low + ") was " +  
            "larger than high(" + high + ")"  
        );
    } else {
        setBounds(low, high);
    }
}

public int getAge() throws Exception {
    return getAge(DEFAULT_MESSAGE);
}
public int getAge(String prompt) throws Exception {
    String inputStr;
    int    age;

    while (true) {
        inputStr  = JOptionPane.showInputDialog(null, prompt);
        try {
            age = Integer.parseInt(inputStr);
            if (age < lowerBound || age > upperBound) {
                throw new Exception("Input out of bound");
            }
            return age; //input okay so return the value & exit
        } catch (NumberFormatException e) {
            JOptionPane.showMessageDialog(null, "'" + inputStr
                    + "' is invalid
                    + "Please enter digits only");
        }
    }
}

private void setBounds(int low, int high) {
    lowerBound = low;
    upperBound = high;
}
import javax.swing.*;

class Ch8TestAgeInputVer4 {
    public static void main(String[] args) {
        int entrantAge;

        try {
            AgeInputVer4 input = new AgeInputVer4(15, 18);

            // AgeInputVer4 input = new AgeInputVer4(20, 18);

            entrantAge = input.getAge("Your Age:");
            JOptionPane.showMessageDialog(null, "Input Okay"); // TEMP

        } catch (IllegalArgumentException e) {
            System.out.println("Internal Error:" + e.getMessage);
        }

        } catch (Exception e) {
            JOptionPane.showMessageDialog(null, 
                "Sorry, you do not qualify to enter" + 
                " the junior competition");
        }
    }
}
Programmer-Defined Exceptions

- Using the standard exception classes, we can use the getMessage method to retrieve the error message.
- By defining our own exception class, we can pack more useful information.
  - for example, we may define an OutOfStock exception class and include information such as how many items to order
- Create new exception as a subclass of Exception.
- Constructor should call constructor of Exception and pass the string to it.
- Include a default constructor.
Example

- AgeInputException is defined as a subclass of Exception and includes public methods to access three pieces of information it carries: lower and upper bounds of valid age input and the (invalid) value entered by the user.

- These extra items are defined as data members of the new exception class, AgeInputException.

- AgeInputVer5.java
class AgeInputException extends Exception {

    private static final String DEFAULT_MESSAGE = "Input out of bounds";
    private int lowerBound;
    private int upperBound;
    private int value;

    public AgeInputException(int low, int high, int input) {
        this(DEFAULT_MESSAGE, low, high, input);
    }

    public AgeInputException(String msg, int low, int high, int input) {
        super(msg);
        if (low > high) {
            throw new IllegalArgumentException();
        }
        lowerBound = low;
        upperBound = high;
        value      = input;
    }
}

Calling constructor of parent.
import javax.swing.*;
class Ch8TestAgeInputVer5 {

    public static void main( String[] args ) {

        int entrantAge;

        try {
            AgeInputVer5 input = new AgeInputVer5(25, 50);
            entrantAge = input.getAge("Your Age:");

            //continue the processing
            JOptionPane.showMessageDialog(null, "Input Okay");
        } catch (AgeInputException e) {
            JOptionPane.showMessageDialog(null,
            "Error: " + e.value() + " is entered. It is " +
            "outside the valid range of [" + e.lowerBound() +
            ", " + e.upperBound() + "]");
        }
    }
}
public AgeInputVer5() throws IllegalArgumentException {
    setBounds(DEFAULT_LOWER_BOUND, DEFAULT_UPPER_BOUND);
}

public AgeInputVer5(int low, int high) throws IllegalArgumentException {
    if (low > high) {
        throw new IllegalArgumentException(
            "Low (" + low + ") was " +
            "larger than high(" + high + ")");
    } else {
        setBounds(low, high);
    }
}

private void setBounds(int low, int high) {
    lowerBound = low;
    upperBound = high;
}

public int getAge(String prompt) throws AgeInputException {
    String inputStr;
    int    age;

    while (true) {
        inputStr = JOptionPane.showInputDialog(null, prompt);
        try {
            age = Integer.parseInt(inputStr);

            if (age < lowerBound || age > upperBound) {
                throw new AgeInputException("Input out of bound",
                                            lowerBound, upperBound, age);
            }
        }
        catch (NumberFormatException e) {
            JOptionPane.showMessageDialog(null, "'" + inputStr
                                             + "' is invalid\n                                             + "Please enter digits only");
        }
    }
    return age; //input okay so return the value & exit
}
Defining your own 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:

  ```java
  super(msg);
  ```
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;
}
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";
}
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

```
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.
Introducing Theory of Computation

- What are the limits of computers?
- What can a computer not compute
  - Today?
  - In 10 years?
  - Ever?
- These are some of the questions that the theory of computation tries to answer.
- Covered in CS483, CS182, CS380.
In order to answer such questions, we first begin with a model for a computer.

Actually, we begin by consider simple machines called automata.

These are really primitive beasts, and come in several species:

- 1-way finite automata
- 2-way finite automata
- Turing Machine
The Turing Machine

- Named after Alan Turing
- Consists of:
  - Input tape with symbols
  - A control consisting of a graph
  - Machine can read, the current symbol, write to the current place, move the tape left or right, and change its state.
- Looks primitive but is capable of computing anything that any modern computer can (no networking).
What are the limits of this machine?

- Can it compute the factorization of a prime number?
- Sure, but how long will it take?
  - 500 digits (life of universe with a supercomputer)
  - Security (cryptography relies on this :)
- How long is too long?
  - Polynomial time
- Can it compute the factorization using a better algorithm in polynomial time?
- Can we prove that there can be no fast algorithm for certain problems?
Guessing computers

- Consider a Turing Machine which can take one of many options at each step.
- If any guess gives us the correct answer, then we have solved the problem.
- Many problems (like factoring) can be solved by such a machine in polynomial time.
- But we don’t have such machines (yet!)
- Quantum computers MAY one day provide us such computers.
NP-Completeness

- Problems that can be solved in polynomial time by a “guessing” machine, but not by a Turing Machine are called NP-Complete.
- If any of these is solved efficiently, then all of them will be solved efficiently!
- Is this possible? I.e. is $P = NP$?
- We don’t know -- if you have the answer you can
  - Get the ACM Turing Award (~ Nobel Prize)
  - Get a great job!
These courses study the issues of computability.

What are the limits of computers?

When should we stop trying to find a fast algorithm?

If there is no (known) fast algorithm, what can we do?
  ○ Heuristics.

What happens if we get new types of computers?
Problem Statement

Implement a Keyless Entry System that asks for three pieces of information: resident’s name, room number, and a password.

- A password is a sequence of characters ranging in length from 4 to 8 and is unique to an individual dorm resident.
- If everything matches, then the system unlocks and opens the door.
- We assume no two residents have the same name.
- We use the provided support classes Door and Dorm.
- Sample resident data named samplelist.dat can be used for development.
Overall Plan

Tasks:
- To begin our development effort, we must first find out the capabilities of the Dorm and Door classes.
- Also, for us to implement the class correctly, we need the specification of the Resident class.

In addition to the given helper classes and the Resident class, we need to design other classes for this application.
- As the number of classes gets larger, we need to plan the classes carefully.
<table>
<thead>
<tr>
<th>Class</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ch8EntranceMonitor</td>
<td>The top-level control object that manages other objects in the program. This is an instantiable main class.</td>
</tr>
<tr>
<td>Door</td>
<td>The given predefined class that simulates the opening of a door.</td>
</tr>
<tr>
<td>Dorm</td>
<td>The given predefined class that maintains a list of Resident objects.</td>
</tr>
<tr>
<td>Resident</td>
<td>This class maintains information on individual dorm residents. Specification for this class is provided to us.</td>
</tr>
<tr>
<td>InputHandler</td>
<td>The user interface class for handling input routines.</td>
</tr>
<tr>
<td>JOptionPane</td>
<td>The standard class for displaying messages.</td>
</tr>
</tbody>
</table>
Class Relationships

- InputFrame
- Ch8EntranceMonitor (main class)
- Dorm
- JOptionPane
- Door
- Resident
We will develop this program in three steps:

1. Define the Resident class and explore the Dorm class. Start with a program skeleton to test the Resident class.
2. Define the user interface InputHandler class. Modify the top-level control class as necessary.
3. Finalize the code by making improvements and tying up loose ends.
Step 1 Design

- Explore the Dorm class
- Implement the Resident class, following the given specification
- Start with the skeleton main class
Step 1 Code

Directory: Chapter8/Step1

Source Files: Resident.java
Ch8EntranceMonitor.java
Step 1 Test

- The purpose of Step 1 testing is to verify that the Dorm class is used correctly to open a file and get the contents of the file.

- To test it, we need a file that contains the resident information. A sample test file called testfile.dat is provided for testing purpose.
  - This file contains information on four residents.
  - This file was created by executing the SampleCreateResidentFile program, which you can modify to create other test data files.
Step 2 Design

- Design and implement the InputHandler class.
- Modify the main class to incorporate the new class.
Step 2 Code

Directory: Chapter8/Step2

Source Files: Resident.java
             Ch8EntranceMonitor.java
             InputHandler.java
Step 2 Test

- The purpose of Step 2 testing is to verify the correct behavior of an InputHandler.

- We need to test both successful and unsuccessful cases.
  - We must verify that the door is in fact opened when the valid information is entered.
  - We must also verify that the error message is displayed when there’s an error in input.

- We should test invalid cases such as entering nonexistent name, correct name but wrong password, not entering all information, and so forth.
Step 3: Finalize

Possible Extensions

- Improve the user interface with a customized form window for entering three pieces of information.
- Terminate the program when the administrator enters a special code.