Exception Handling

MEET 2011
Scenario

```java
public class BankAccount
{
public void withdraw(double amount)
{
    if (amount > balance)
        // Now what?... ...
}
}
```
What is an Exception?

- Special conditions that change the normal flow of program execution.
- Some can be the programmer’s fault.
- Some can be out of the programmer’s control.
Exception Properties

- Can't be overlooked
- Various types exist for various failures
- Can be handled by a competent handler
- we “throw” an exception object to indicate a failure

```java
throw new XxxException(. . .);
```
Hierarchy of Exception Class

Figure 1  The Hierarchy of Exception Classes
Throwing an Exception

• Syntax:
  
  ```java
  throw exceptionObject;
  ```

  Example:

  ```java
  throw new IllegalArgumentException();
  ```

  Purpose: To throw an exception and transfer control to a handler for this exception type
public class BankAccount
{
    public void withdraw(double amount)
    {
        if (amount > balance)
        {
            throw new IllegalArgumentException("Amount exceeds balance");
        }
        balance = balance - amount;
    }
    ...
}
Practice Problem

- Modify the `BankAccount` class to throw an `IllegalArgumentException`
  - when the account is constructed with a negative balance
  - when a negative amount is deposited
  - or when an amount that is not between 0 and the current balance is withdrawn.

Write a test program that causes all three exceptions to occur and that catches them all. Your tester class should be called `BankAccountTester`. 
Exception Specification

Purpose: To indicate the exceptions that a method can throw.

Syntax:
```java
accessSpecifier returnType methodName
   (parameterType parameterName, . . .) throws
   ExceptionClass, ExceptionClass . . .
```

Example:
```java
public void read(BufferedReader in) throws IOException
```

public class Coin
{
    public void read(BufferedReader in) throws IOException
    {
        value = Double.parseDouble(in.readLine());
        name = in.readLine();
    }
    ...
}
Quick Review

• Suppose a method calls the FileReader constructor and the read method of the FileReader class, which can throw an IOException. Which throws specification should you use?
Checked vs. Unchecked Exceptions

Checked

• The compiler checks that you are aware of the exception
• Used for errors that can happen even in correct programs
• Checked exceptions can, at compile time, reduce the incidence of unhandled exceptions surfacing at runtime in a given application;

Unchecked

• “Your fault” errors (in most cases)
• Often represent scenarios that do not allow for recovery
Checked vs. Unchecked Exceptions

Checked

Checked exceptions are subclasses of Exception that are not subclasses of RuntimeException

Unchecked

Unchecked exceptions extend the class RuntimeException or Error.
Checked vs. Unchecked Exceptions

- Throwable
  - Error
  - Exception
    - IOException
      - EOFException
      - FileNotFoundException
    - ClassNotFoundException
    - CloneNotSupportedException
      - UnsupportedOperationException
    - RuntimeException
      - ArithmeticException
      - ClassCastException
Quick Review

• Why is a NullPointerException not a checked exception?
Specify Checked Statements

• To declare that a method should be terminated when a checked exception occurs within it, tag the method with a `throws` specifier.

```java
public class DataSet {
    public void read(String filename) throws FileNotFoundException {
        FileReader reader = new FileReader(filename);
        Scanner in = new Scanner(reader);
        ...
    }
    ...
}
```
Designing Your Own Exception Types

• if (amount > balance)
  throw new InsufficientFundsException(. . .);
• Make it an unchecked exception--programmer could have avoided it by calling getBalance first
• Extend RuntimeException
• Supply two constructors
public class InsufficientFundsException extends RuntimeException {

    public InsufficientFundsException()
    {
    }

    public InsufficientFundsException(String reason)
    {
        super(reason);
    }
}
Catching Exceptions

Statements in **try** block are executed:

- If no exceptions occur, **catch** clauses are skipped
- If exception of matching type occurs, execution jumps to **catch** clause
- If exception of another type occurs, it is thrown to the calling method
- If **main** doesn't catch an exception, the program terminates with a stack trace
Syntax: General Try Block

Purpose:

To execute one or more statements that may generate exceptions. If an exception of a particular type occurs, then stop executing those statements and instead go to the matching catch clause. If no exception occurs, then skip the catch clauses.
try Syntax: General Try Block

```java
{
    statement
    statement
    ...
}
catch (ExceptionClass exceptionObject)
{
    statement
    statement
    ...
}
catch (ExceptionClass exceptionObject)
{
    statement
    statement
    ...
}
...
Syntax: General Try Block

• Example:

```java
try
{
    System.out.println("What is your name?");
    String name = console.readLine();
    System.out.println("Hello,"+name +"!");
}
catch (IOException exception)
{
    exception.printStackTrace();
    System.exit(1);
}
```
The finally Clause

- Exception terminates current method
- Danger: Can skip over essential code
- Example:
  ```java
  BufferedReader in;
  in = new BufferedReader(
      new FileReader(filename));
  purse.read(in);
  in.close();
  ```
- Must execute `in.close()` even if exception happens
- Use `finally` clause for code that must be executed "no matter what"
The **finally** Clause

```java
BufferedReader in = null;
try {
    in = new BufferedReader(
        new FileReader(filename));
    purse.read(in);
} finally {
    if (in !=null) in.close();
}
```
The **finally** Clause

- Executed when `try` block comes to normal end
- Executed if a statement in `try` block throws an exception, before exception is thrown out of `try` block
- Can also be combined with `catch` clauses
Purpose

- To execute one or more statements that may generate exceptions, and to execute the statements in the finally clause whether or not an exception occurred.
Syntax

```python
try
{
    statement
    statement
    ...
}
finally
{
    statement
    statement
    ...
}
```
Syntax: Example

BufferedReader in = null;
try {
    in = new BufferedReader(new FileReader(filename));
    purse.read(in);
}
f{
    if (in !=null) in.close();
}
A Complete Example

• Program
  • reads coin descriptions from file
  • adds coins to purse
  • prints total
• What can go wrong?
  • File might not exist
  • File might have data in wrong format
• Who can detect the faults?
  • main method of PurseTest interacts with user
  • main method can report errors
  • Other methods pass exceptions to caller
The read method of the Coin class

- Distinguishes between expected and unexpected end of file

```java
public boolean read(BufferedReader in) throws IOException {
    String input = in.readLine();
    if (input == null) // normal end of file
        return false;
    value = Double.parseDouble(input);
    // may throw unchecked NumberFormatException
    name = in.readLine();
    if (name == null) // unexpected end of file
        throw new EOFException("Coin name expected");
    return true;
}
```
The read method of the Purse class

- Unconcerned with exceptions
- Just passes them to caller

```java
public void read(BufferedReader in) throws IOException {
    boolean done = false;
    while (!done) {
        Coin c = new Coin();
        if (c.read(in)) add(c);
        else done = true;
    }
}
```
The `readFile` method of the `Purse` class

- **finally** clause closes files if exception happens

```java
public void readFile(String filename)
    throws IOException
{
    BufferedReader in = null;
    try
    {
        in = new BufferedReader(
            new FileReader(filename));
        read(in);
    }
    finally
    {
        if (in != null)
            in.close();
    }
}
```
User interaction in `main`

- If an exception occurs, user can specify another file name

```java
boolean done = false;
String filename = JOptionPane.showInputDialog("Enter file name");
while (!done)
{
    try
    {
        Purse myPurse = new Purse();
        myPurse.readFile(filename);
        System.out.println("total=" + myPurse.getTotal());
        done = true;
    }
    ...
}
```
User interaction in main

...catch (IOException exception)
    {
        System.out.println("Input/output error "+
                exception);
    }
catch (NumberFormatException exception)
    {
        exception.printStackTrace(); // error in file format
    }
if (!done)
{
    filename = JOptionPane.showInputDialog("Try another
            file:");
    if (filename == null)
        done =true;
}
Scenario

1. PurseTest.main calls Purse.readFile
2. Purse.readFile calls Purse.read
3. Purse.read calls Coin.read
4. Coin.read throws an EOFException
5. Coin.read has no handler for the exception and terminates immediately.
6. Purse.read has no handler for the exception and terminates immediately
7. Purse.readFile has no handler for the exception and terminates immediately after executing the finally clause and closing the file.
8. PurseTest.main has a handler for an IOException, a superclass of EOFException. That handler prints a message to the user. Afterwards, the user is given another chance to enter a file name. Note that the statement printing the purse total has been skipped.