TA Session 9

Clarifications, Exceptions, Files

Clarifications

- You can change parameters' values via Debugger.
- Methods (including constructors) behave the same as data fields with the default modifier.
- Passing the testers is not the major issue.

Exception Classes

- As mentioned in the lecture, exceptions are **messages** that states that something went wrong.
- Exception are implemented as **java objects.**
- As such, exception classes usually do not contain any fields or methods
  - The calling class knows what the message is by the specific exception class
  - Each type of message has its own class
- Sometimes an exception object will specify more information (usually for debug purposes) such as a string specifying what exactly went wrong

Motivation

```
public static void sendMessage(Message m) {
    checkMessageValidity(m);

    establishCommunication();
    Response s = send(m);
    closeCommunication();

    checkMessageResponse(s);
}
```

Motivation cont.

```
public static void sendMessage(Message m) {
    checkMessageValidity(m);

    establishCommunication();
    Response s = send(m);
    closeCommunication();

    checkMessageResponse(s);
}
```

What if this line fails?

Motivation cont.

```
public static void sendMessage(Message m) {
    checkMessageValidity(m);

    establishCommunication();
    Response s = send(m);
    closeCommunication();

    checkMessageResponse(s);
}
```

What if this line fails?

Motivation cont.

```
public static void sendMessage(Message m) {
    checkMessageValidity(m);

    establishCommunication();
    Response s = send(m);
    closeCommunication();

    checkMessageResponse(s);
}
```

What if this line fails?

Motivation cont.

```
public static void sendMessage(Message m) {
    checkMessageValidity(m);

    establishCommunication();
    Response s = send(m);
    closeCommunication();

    checkMessageResponse(s);
}
```

What if this line fails?

Motivation cont.

```
public static void sendMessage(Message m) {
    checkMessageValidity(m);

    establishCommunication();
    Response s = send(m);
    closeCommunication();

    checkMessageResponse(s);
}
```

What if this line fails?

Motivation cont.

```
public static void sendMessage(Message m) {
    if (checkMessageValidity(m) == true) {
        if (establishCommunication() == true) {
            Response s = send(m);

            if (s != null) {
                System.out.println("Error sending");
            }
            else {
                closeCommunication();
                if (checkMessageResponse(s) == false) {
                    System.out.println("Error sending.");
                }
            }
        }
        else {
            System.out.println("Error sending");
        }
    }
    else {
        System.out.println("Error sending.");
    }
}
```

Using Exceptions

```
public static void sendMessage(Message m) {
    try {
        checkMessageValidity(m);

        establishCommunication();
        Response s = send(m);
        closeCommunication();

        checkMessageResponse(s);
    }
    catch( CommunicationException ex) {
        // ..
        System.out.println("Error sending.");
    }
}
```

Advantages of Using Exceptions

Advantage #1:

Separating Error-Handling Code from "Regular" Code

Exceptions Helps us to **Handle** Errors

```
public static void sendMessage(Message m) {
    try {
        checkMessageValidity(m);

        establishCommunication();
        Response s = send(m);
        closeCommunication();

        checkMessageResponse(s);
    }
    catch( CommunicationException ex) {
        // ..
        System.out.println("Error sending.");
    }
}
```

Exception Handling

Exceptions Helps us to **Handle** Errors

```
public static void sendMessage(Message m) {
    try {
        checkMessageValidity(m);

        establishCommunication();
        Response s = send(m);
        closeCommunication();

        checkMessageResponse(s);
    }
    catch( CommunicationException ex) {
        // ..
        System.out.println("Error sending.");
    }
}
```

Exception Handling **?**

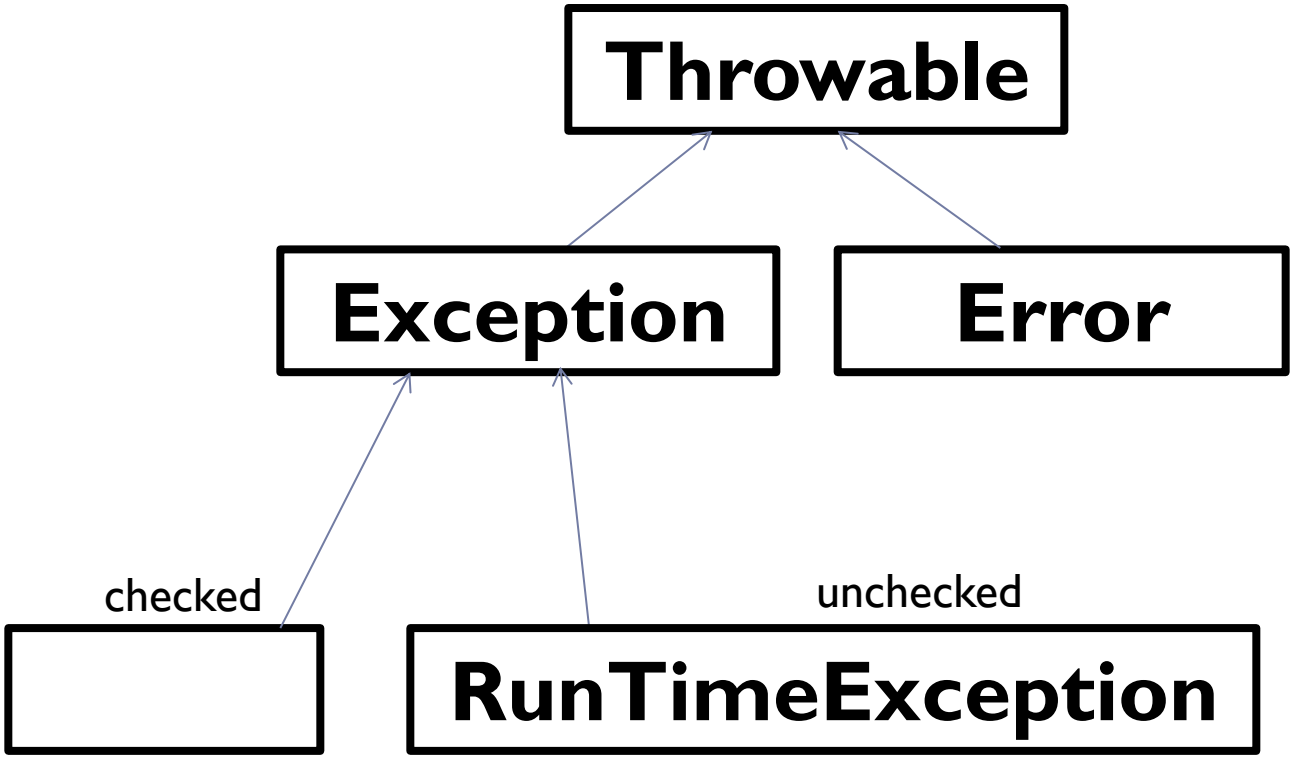
Exceptions Helps us to **Handle** Errors

```
public static void sendMessage(Message m) {
    try {
        checkMessageValidity(m);

        establishCommunication();
        Response s = send(m);
        closeCommunication();

        checkMessageResponse(s);
    }
    catch( CommunicationException ex) {
        // ..
        initializeIterativeTrials();
    }
}
```

Exception Classes



Exception Flow

Class Coffee Vending Machine

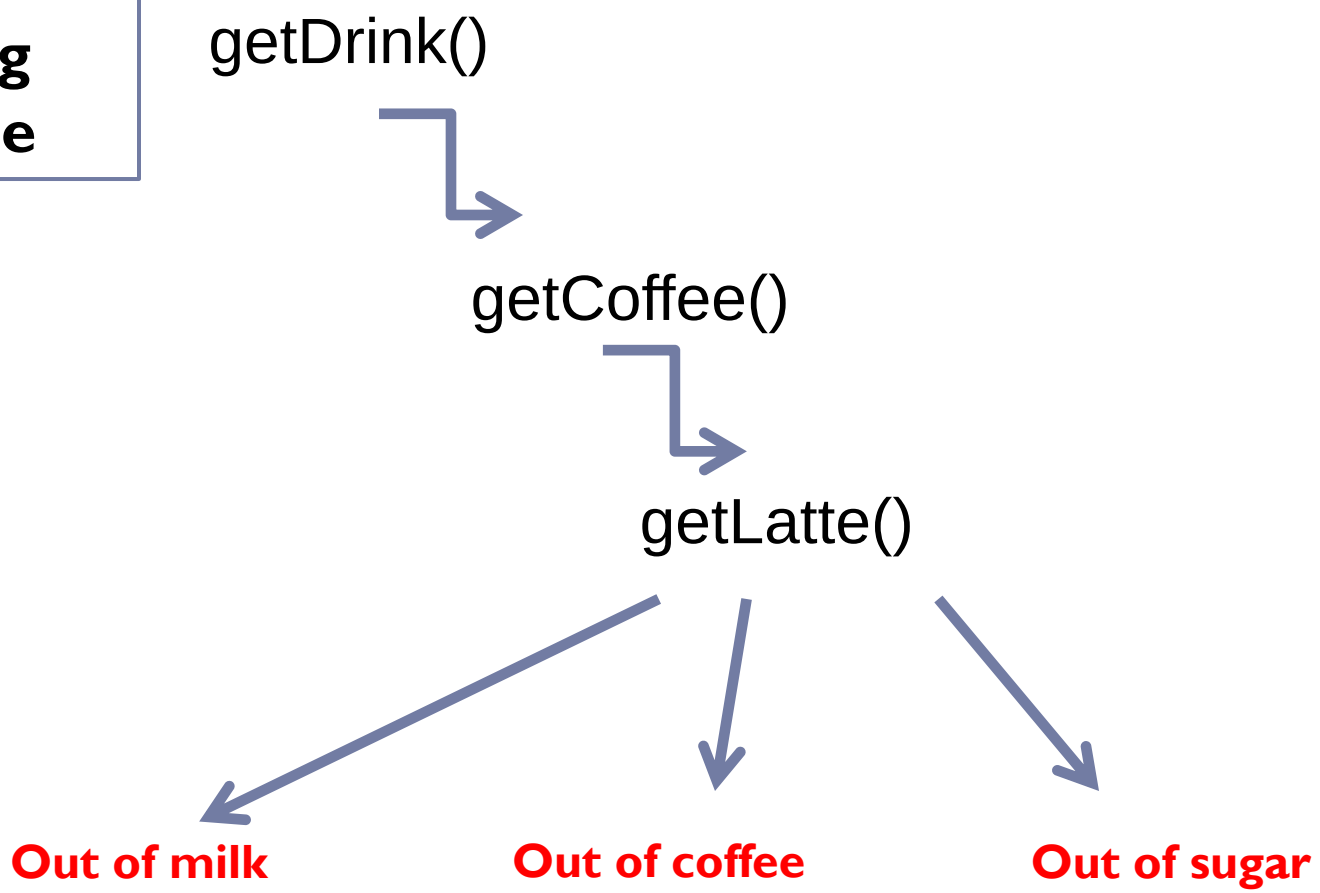
getDrink()

getCoffee()

getLatte()

Exception Flow

Class Coffee Vending Machine



Exception Classes

- New exception classes that we write will almost always be **checked**
  - I.e., extend the **Exception** class
- Exception classes that we write may have their own hierarchy
  - Allows to give similar treatment to several types of errors

Defining A New Exception Class

```
public class LatteException extends Exception{
   private static final long serialVersionUID = 1L;

   public LatteException () {
       super("Problem with ordering coffee");
   }
   public LatteException (String s){
       super(s);
   }
}
```

Default Ctor

Message Ctor

- **serialVersionUID –** later in the course

Throwing exceptions

```
Latte getLatte() throws LatteException {
  if (milk.isEmpty() || suagr.isEmpty())
      throw new LatteException ();
}
```

Using Exception Hierarchy

```
class LatteException extends Exception{…}

class OutOfMilkException extends LatteException {…}
class OutOfSugarException extends LatteException {…}
```

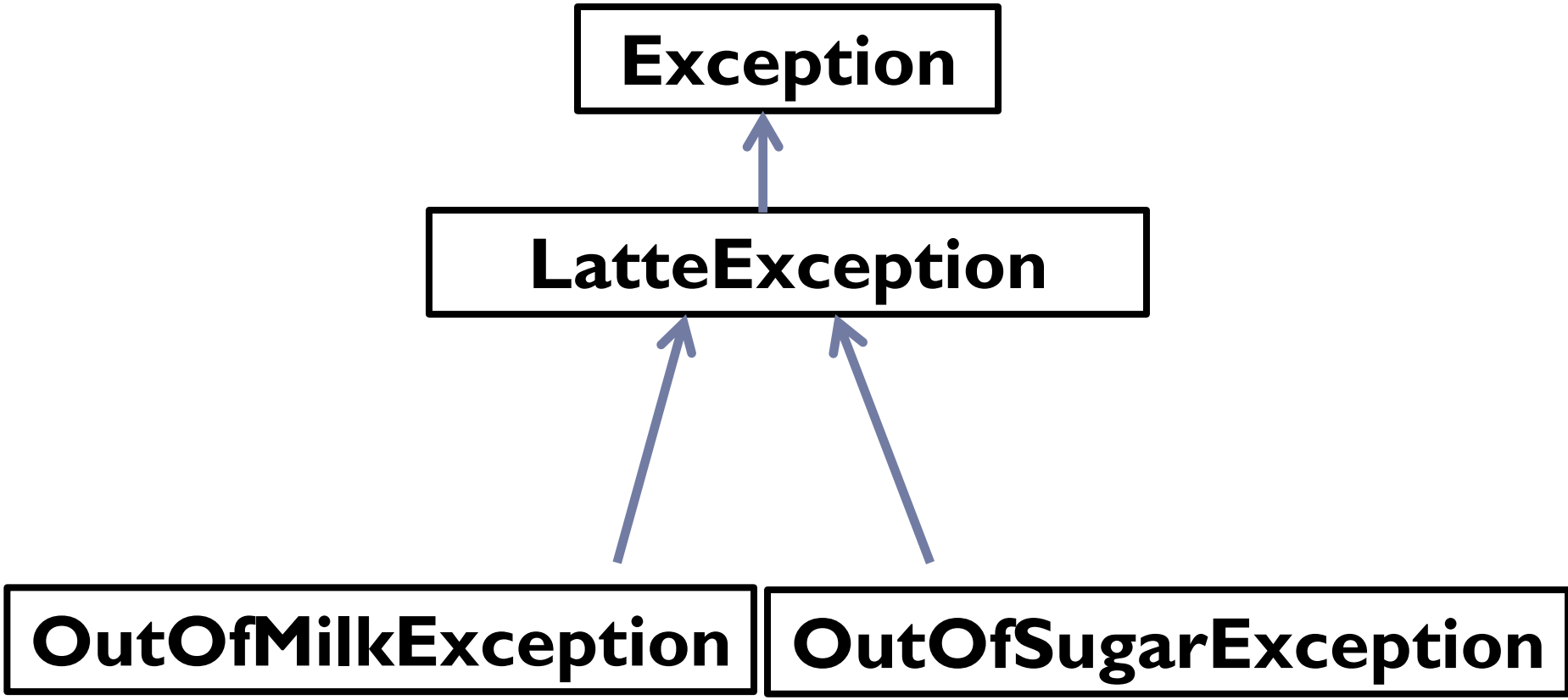
Using Exception Hierarchy

```
class LatteException extends Exception{…}

class OutOfMilkException extends LatteException {…}
class OutOfSugarException extends LatteException {…}
```

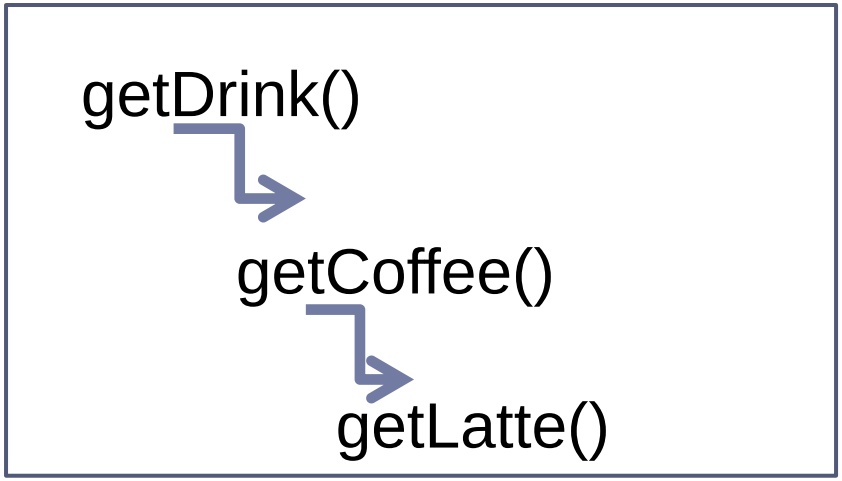
```
private Latte getLatte() throws LatteException {
   if (milk.isEmpty())
       throw new OutOfMilkException ();
   if (suagr.isEmpty())
       throw new OutOfSugarException ();
   //...
}
```

Using Exception Hierarchy



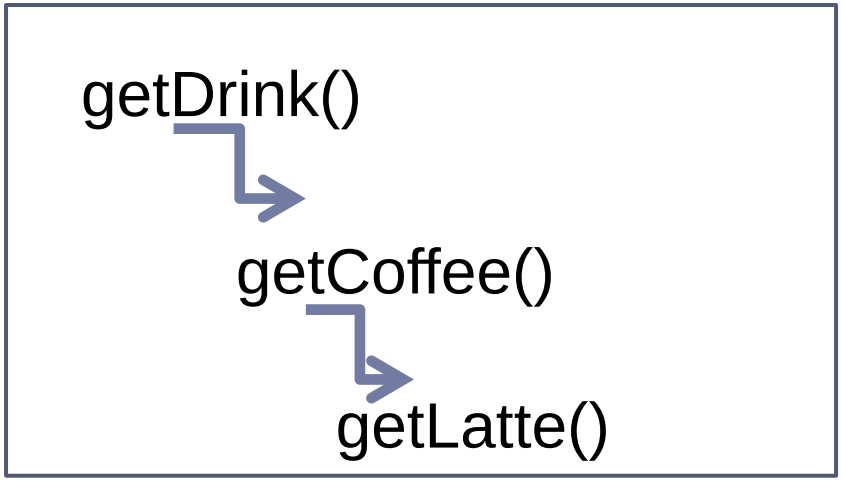
Catching Exceptions: Specific Exception

```
Public getCoffe(){
   try{
      // Making Latte
      Latte l = getLatte();
      //...
   } catch(OutOfMilkException e){
      orderMilk();
   }
}
```



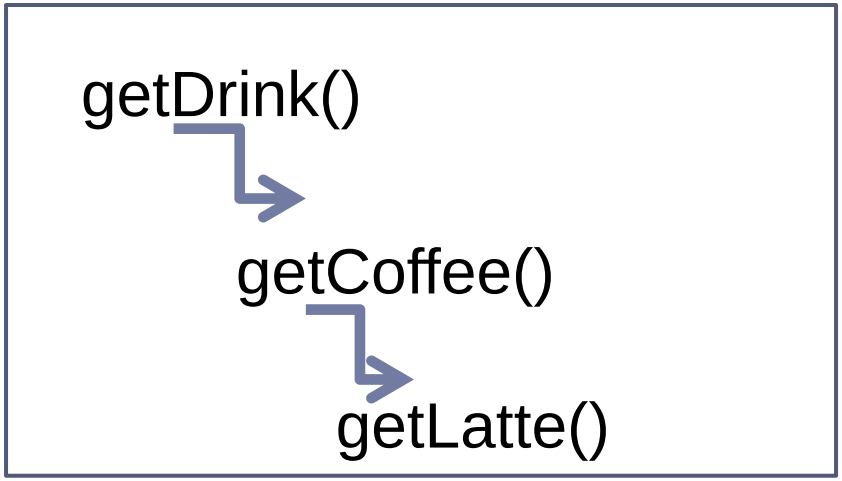
Catching Exceptions: Specific Exception

```
Public getCoffe(){
   try{
      // Making Latte
      Latte l = getLatte();
      //...
   } catch(OutOfMilkException e){
      orderMilk();
   }
   .........
}
```



Catching Exceptions: General Exception

```
Public getCoffe(){
   try{
      // Making Latte
      Latte l = getLatte();
      //...
   } catch(LatteException e){
      checkLatteSystem();
   }
}
```

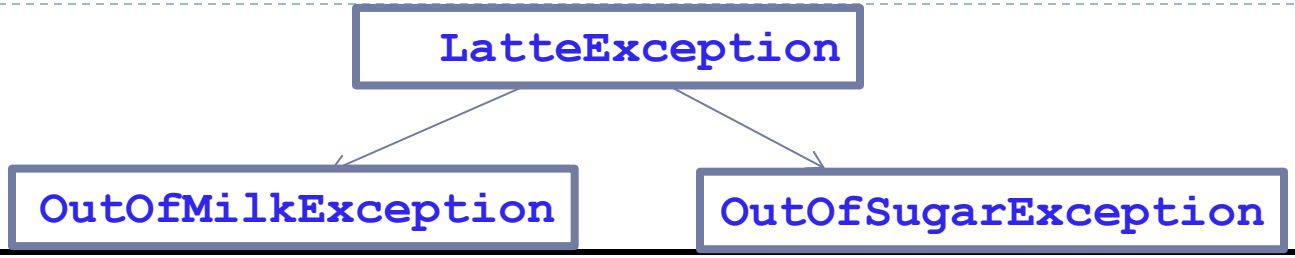


Advantages of Using Exceptions

Advantage #2:

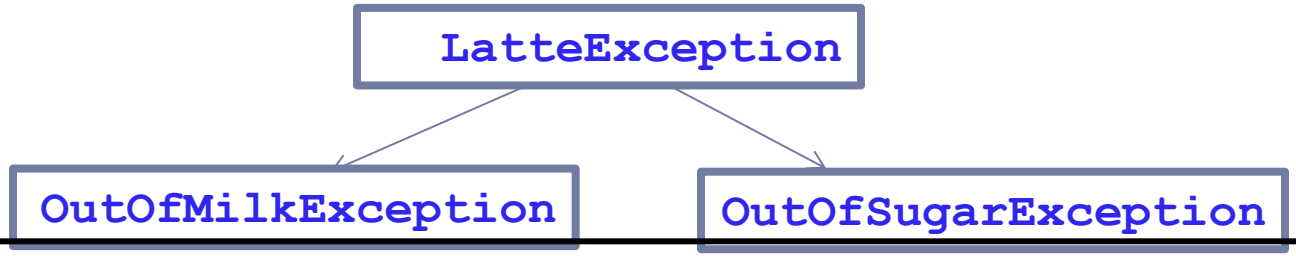
Grouping and Differentiating Error Types

What's Wrong Here?




```
try{
    // Making Latte
    Latte l = getLatte();
    //...
} catch(LatteException e){
    //...
} catch(OutOfMilkException e){
    //...
}
```

What's Wrong Here?




```
try{
    // Making Latte
    Latte l = getLatte();
    //...
} catch(LatteException e){
    //...
} catch(OutOfMilkException e){
    //...
}
```

Compilation Error

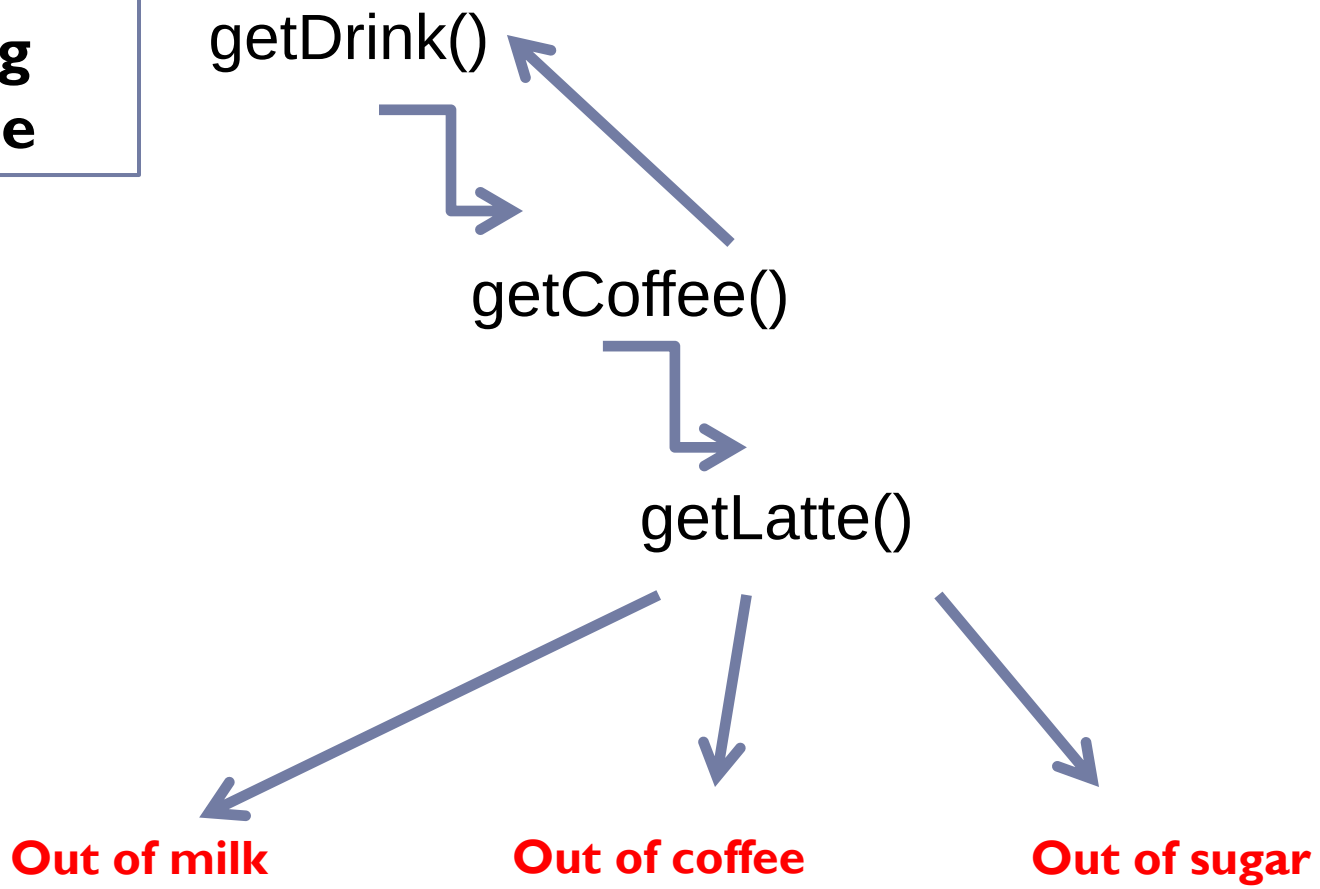
Exception Declaration

- Each method must declare the **checked** exceptions it throws.

```
public Latte getLatte() throws LatteException {
   if (milk.isEmpty())
       throw new OutOfMilkException ();
   if (suagr.isEmpty())
       throw new OutOfSugarException ();
   //...
}
```

Exception Flow

Class Coffee Vending Machine



Advantages of Using Exceptions

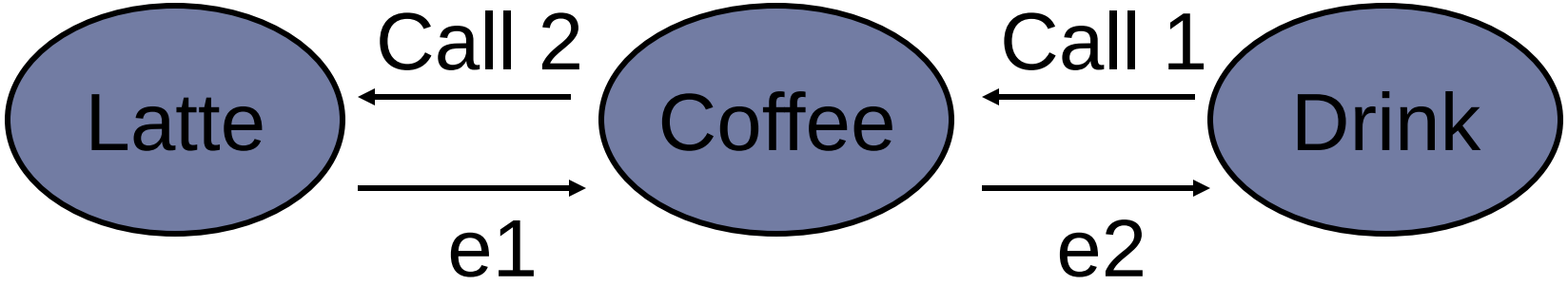
Advantage #3:

Propagating Errors Up the Call Stack

Chained Exceptions

- "translating" Exceptions from one class to another

```
try{
   //call some Latte method
}catch(LatteException e){
        throw new BadDrinkException("problem", e);
}
```



Handling More than One Type of Exception

- If we want to catch more than one exception but handle them all in the same way, we can use the "|" operator
  - Java 7 feature

```
try {
        // Making Latte
        Latte l = getLatte();
        //...
}catch(Exception1|Exception2 e){
        // Handle Exception1, Exception2 in the same way
        e.getMessage();
}
```

Finally Block

```
try{
      // Making Latte
      Latte l = getLatte();
      //...
} catch(LatteException e){
      //...
} Finally{
      //...
}
```

Exceptions and Packages

- Exceptions should be put in the same package as the classes that throw them
  - And not all in the same package
  - **IOException** resides in java.**io**
  - **EmptyStackException** resides in java.**util**
  - …
- This is because exceptions are **logically connected** to the classes that throw them
  - More than they are connected to one another

Bad usage of Exception Handling

- Using a computer language's error handling structures to perform normal program logic

```
try {
    int idx = 0;
    while (true) {
            displayProductInfo(prodnums[idx]);
            idx++;
    }
 } catch (IndexOutOfBoundsException e) {
    // Do some cleanup
}
```

Bad usage of Exception Handling

- Using a computer language's error handling structures to perform normal program logic

```
try {
    int idx = 0;
    while (true) {
        displayProductInfo(prodnums[idx]);
        idx++;
    }
 } catch (IndexOutOfBoundsException e) {
    // Do some cleanup
}
```

✘Heavy!

✘Unexpected!

✘Hides bugs!

Path

- Each file in our file system has a unique path
  - Starting from the **root**, down the directories structure, until the file name
  - A path structure changes between different Operating systems
    - Unix: /cs/course/current/oop/hello.txt
    - Windows: C:\My Documents\My Pictures\my_picture.jpg

Path

Properties

- Relative or absolute
  - Absolute: /cs/…/
  - Relative: myDir/…/
- Symbolic links
  - A file that is a reference to another file
  - Transparent to the user

File Class

- All information you need about files and directories in one class
- User-friendly
- This class **cannot** be used to read from/write to file!
- File myfile = **new** File(String path);

File Class

File myfile = new File(path);

- What is it?
  - **boolean** exists() – does a file\directory exist in *path?*
  - **boolean** isFile() – is it a file? (i.e., not a directory)
  - **boolean** isDirectory() – is it a directory? (i.e., not a file)
  - **boolean** isAbsolute() – is the path absolute (not relative)
  - **boolean** isHidden() – is the file hidden?

File Class

File myfile = new File(path);

- Where is it?
  - String getAbsolutePath() – get absolute path to file
  - String getParent() – get parent directory
- Describe it
  - **long** length() – size of file (in bytes)
  - **long** lastModified() – time of the last modification of this file (in miliseconds since 1/1/1970)
    - To be used with the Date class.

File Class

File myfile = new File(path);

- Change file
  - **boolean** delete() – delete the file
  - **boolean** renameTo(File dest) – move file to the path in *dest*
  - **boolean** mkdir() – create a directory named *path*
- Notice that there is no *copy* operation in the File class
  - See comment later

File Class

More

- Permissions
  - **boolean** canRead(), canWrite(), canExecute() – does file have reading/writing/execution permission
- Setting metadata
  - **boolean** setReadable/Writable/Executable(**boolean** value) – set reading/writing/execution permission
  - **boolean** setLastModified(long time) – set last modification date of this file

File Class

Directories

- If *path* points at a directory
  - File[] listFiles() – return all files and direct directories in the path.
  - String[] list() – same, but return String array

File Example

```
Import java.io.File
public void analyzePath( String path, String output){
    File file = new File("C:\\Dropbox\\OOP");

    System.out.println(file.getName());

    if (file.exists()) {
            System.out.println( "File exists!");
            System.out.println(file.getParent());
            if (file.canWrite()) {
                    System.out.println("Has Writing permissions!");
            } else {
                    System.out.println("Doesn't have writing permissions:(");
            }
```

File Example cont.

```
            if (file.isDirectory()) {
                        // Get all files in directory
                        String files[] = file.list();

                        for (String localFile : files) {
                                    System.out.println(localFile);
                        }
            } /* If isDirectory() */
    } /* If exists() */
}
```

java.nio2

- Java 7 introduced a new io packages for working with files (java.**n**io, version 2)
- This package includes various new file features
  - Getting/setting more metadata options
  - A copy operation
  - Much more flexibility in general
- In this course we only introduce the previous (simpler) approach (java.io.File)
  - You may use java.nio in your projects if you wish