Objectives

- Exceptions

Review

- How do we specify that a class or a method cannot be subclassed/overridden?
- Compare and contrast abstract classes and interfaces
- When should a class be abstract?
- When should you create/use an interface?
- What is the keyword for defining your class to implement an interface?

Analysis of `equals` methods

```
public boolean equals(Object o){
   if(((Birthday) o).getDate() != this.getDate())
      return false;

   if( ((Birthday) o).getMonth() != this.getMonth())
      return false;
   return true;
}
```

```
public boolean equals(Object o) {
   Birthday other = (Birthday) o;
   if (this.month == other.month && this.day ==
other.day)
      return true;
   else
      return false;
}
```

EXCEPTIONS

Errors

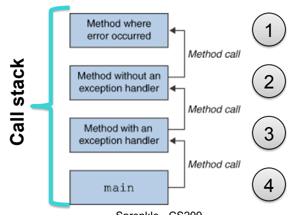
- Programs encounter errors when they run
  - Users may enter data in the wrong form
  - Files may not exist
  - Printers run out of paper in the middle of printing
  - Program code has bugs
- When an error occurs, a program should do one of two things:
  - Revert to a stable state and continue
  - Allow the user to save data and then exit the program gracefully

Java Method Behavior

- Normal/correct case: return specified return type
- Error case: does not return anything, `throws` an `Exception`
  - An ***exception*** is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.
  - `Exception:` object that encapsulates the error information

Handling Exceptions

- JVM's **exception-handling mechanism** searches for an **exception handler**—the error recovery code
  - Exception handler deals with a *particular* exception
  - Searches call stack for a method that can handle (or *catch)* the exception


Call stack
Method where error occurred
Method without an exception handler
Method with an exception handler
main
Method call
Method call
Method call
1
2
3
4

Throwable

- All exceptions indirectly derive from `Throwable`
  - Child classes: `Error` and `Exception`
- Important `Throwable` methods
  - `getMessage()`
    - Detailed message about error
  - `printStackTrace()`
    - Prints out where problem occurred and path to reach that point
  - `getStackTrace()`
    - Get the stack in non-text format

Stack Trace Example

```
java.io.FileNotFoundException: fred.txt
    at java.io.FileInputStream.<init>(FileInputStream.java)
    at java.io.FileInputStream.<init>(FileInputStream.java)
    at ExTest.readMyFile(ExTest.java:19)
    at ExTest.main(ExTest.java:7)
```

How helpful is this output?
How user friendly is it?

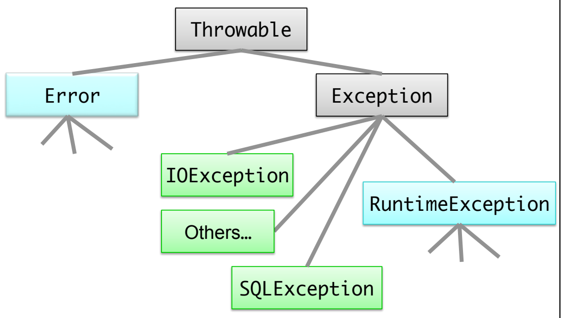
Stack Trace Example

```
java.io.FileNotFoundException: fred.txt
    at java.io.FileInputStream.<init>(FileInputStream.java)
    at java.io.FileInputStream.<init>(FileInputStream.java)
    at ExTest.readMyFile(ExTest.java:19)
    at ExTest.main(ExTest.java:7)
```

How helpful is this output?
How user friendly is it?

- Useful for debugging your code
- Generate/display user-friendly errors in finished product
  - Often requires "higher-level code" to handle exception

Exception Classification



Exception Classification: `Error`

- An internal error
- Strong convention: reserved for JVM
  - JVM-generated when resource exhaustion or an internal problem
    - Example: Out of Memory error (When can that happen in Java?)
- Program's code should not and can not throw an object of this type
- *Unchecked* exception

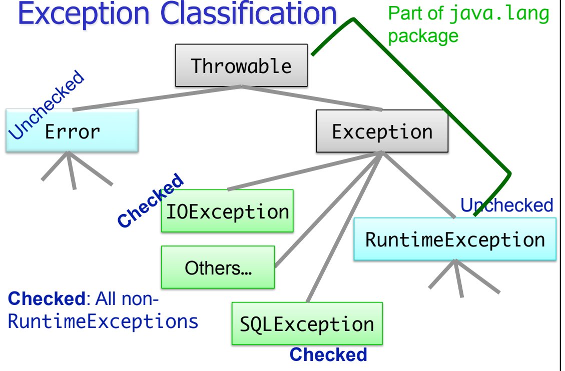
Exception Classifications

1. `RuntimeException` something that happens because of a programming error
   - **Unchecked** exception
   - Examples: `ArrayOutOfBoundsException, NullPointerException, ClassCastException`
2. **Checked** exceptions
   - A well-written application should anticipate and recover from
   - e.g., `IOException, SQLException`

Exception Classifications

- If something is *programmer's* fault → `RuntimeException`
- Otherwise, an `Error` or another `Exception`
- Common checked exception: `IOException`
  - Trying to read past the end of a file
  - Trying to open a bad URL
  - File not found
  - …

Exception Classification



Types of Exceptions

Unchecked

- Any exception that derives from `Error` or `RuntimeException`
  - Programmer does not create/handle
  - Try to make sure that they don't occur
  - Often indicates programmer error
    - E.g., precondition violations

Checked

- Any other exception
  - Programmer creates and handles checked exceptions
  - Compiler-enforced checking
    - Improves *reliability*
- For conditions from which caller can reasonably be expected to recover

Types of Unchecked Exceptions

- Derived from the class `Error`
  - Any line of code can generate because it is internal error
  - Don't worry about what to do if this happens
- Derived from the class `RuntimeException`
  - Indicates a bug in the program
  - Fix the bug

Checked Exceptions

- Need to be handled by your program
  - Compiler enforced
- *Advertise* the exceptions that a method throws
  - For each method, tell the compiler:
    - What the method returns
    - What could possibly go wrong
  - Helps users of your interface know what your method does and lets them decide how to handle exceptions

Discussion: Why Checked and Unchecked Exceptions?

- Why do we have exceptions that the compiler doesn't enforce that the programmer checks?
  - Think about examples of unchecked exceptions and when those exceptions can occur

THROWING EXCEPTIONS

Methods and Exceptions Example

- `BufferedReader` has method `readLine()`
  - Reads a line from a *stream*, such as a file or network connection
- Header:

Part of "Advertising"

```
public String readLine() throws IOException
```

- Interpreting the header: `readLine` will
  - return a String (if everything went right)
  - throw an `IOException` (if something went wrong)

Advertising Checked Exceptions

- Advertising: document under what conditions each exception is thrown in Javadoc
  - Use `@throws` tag
- Examples of when your method should advertise the ***checked*** exceptions that it may throw
  - Your method calls a method that throws a checked exception
  - Your method detects an error in its processing and decides to throw an exception

Example: Passing an Exception "Up"

```
public String readData(BufferedReader in)
   throws IOException {
      String str1;
      str1 = in.readLine();
      return str1;
}
```

Throws an `IOException`

- `readData()` calls a method that can throw an `IOException`
- `readLine()` will throw this exception to our method
  - Assuming we don't want to handle the exception, we *throw* the exception as well
  - Whoever calls `readData` will handle exception

Generating Our Own Exception

- If we have a program that reads a file byte-by-byte and we know in advance how big the file is…
- What do we do if we reach the EOF while we should still have data to read?
  - Generate our own `Exception` object!

Example: Throwing An Exception

Expected number of bytes

```
public String readBytes(BufferedReader in, int num_bytes)
     throws EOFException {
      while (. . .) {
        if (char_in == EOF) {
            if (number_read < num_bytes)
                  throw new EOFException();
        }
        . . .
      }
      . . .
}
```

Throwing An Exception

```
if (num_read < num_bytes)
     throw new EOFException();
```

- If we encounter an EOF, we make a new object of class `EOFException`
  - Class derived from `IOException`
- After making `Exception` object, we `throw` it
  - Method ends at this point
  - Calling method handles exception, which says that encountered an EOF before we should have

A More Descriptive Exception

- Four constructors for most Exception classes
  - Default (no parameters)
  - Takes a `String message`
    - Describe the condition that generated this exception more fully
  - 2 more

```
if (num_read < num_bytes) {
    String problem = "I read " + num_read +
     " when I should have read " + num_bytes;
    throw new EOFException(problem);
}
```

Best messages include all state that could have contributed to the problem

Common Exceptions

Name	Purpose
`IllegalArgumentException`	When caller passes in inappropriate argument
`IllegalStateException`	Invocation is illegal because of receiving object's state. (Ex: closing a closed window)

- Both inherit from `RuntimeException`
- May seem like these cover it all but only used for certain kinds of illegal arguments and exceptions
- Not used when
  - A null argument passed in; should be a `NullPointerException`
  - Pass in invalid index for an array; should be an `IndexOutOfBoundsException`

Factorial Alternatives

```
public static double factorial( int x ) {
   if( x < 0 )
      return 0.0;
   double fact = 1.0;
   while( x > 1 ) {
      fact *= x;
      x--;
   }
   return fact;
}
```

Factorial Alternatives

Note, no `throws` clause
Why?

```
public static double factorial( int x ) {
   if( x < 0 )
      throw new IllegalArgumentException("x" +
                  "must be >= 0");
   double fact = 1.0;
   while( x > 1 ) {
      fact *= x;
      x--;
   }
   return fact;
}
```

`IllegalArgumentException`: Thrown to indicate that a method has been passed an illegal or inappropriate argument.

What are the pros and cons of these approaches?

Goal: Failure Atomicity

- After an object throws an exception, the object should be in a well-defined, usable state
  - A failed method invocation should leave object in state prior to invocation
- Approaches:
  - Check parameters/state before performing operation(s)
  - Do the failure-prone operations first
  - Use recovery code to “rollback” state
  - Apply to temporary object first, then copy over values

Practice

- We discussed a similar method
- How should we implement this method?

```
public void setBirthday(int month, int day) {

}
```

Assignment 6

- Due Friday: Practice on Abstract classes, interfaces, packages, equals method