Objectives

- Exceptions

Review

1. What are the components of the Java Collections framework?
2. For each discussed interface, name one implementation
3. What is the preferred way to use Java Collections?
   - Why is that the preferred way?
4. What are the two main ways to iterate through a collection?
5. What are benefits of the Java Collections framework?

Review: Traversing Collections

- For-each loop:

Or whatever data type is appropriate

```
for (Object o : collection)
    System.out.println(o);
```

- Valid for all `Collections`
  - `Maps` (and its implementations) are not `Collections`
    - But, `Map`'s `keySet()` is a `Set` and `values()` is a `Collection`

Review: Traversing Lists: Iterator

- Always between two elements


```
Iterator<Integer> i = list.iterator();
while( i.hasNext()) {
    int value = i.next();
    …
}
```

Benefits of Collections Framework

- **Provides common, well-known interface**
  - Allows interoperability among unrelated APIs
  - Reduces effort to learn and to use new APIs for different implementations
- **Reduces programming effort:** provides useful, reusable data structures and algorithms
- **Increases program speed and quality:** provides high-performance, high-quality implementations of data structures and algorithms; interchangeable implementations → tuning
- **Reduces effort to design new APIs:** use standard collection interface for your collection
- **Fosters software reuse:** New data structures/algorithms that conform to the standard collection interfaces are reusable

EXCEPTIONS

Errors

- Programs encounter errors when they run
  - Users may enter data in the wrong form
  - File may not exist
  - 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

* (Of course, not *your* programs)

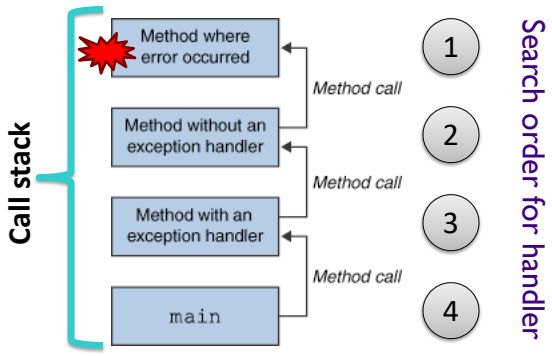
Java Method Behavior

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

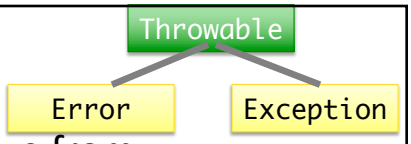
Similar to Python

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



Throwable



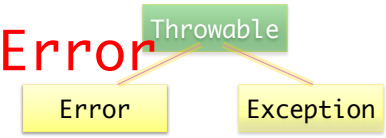
- All exceptions indirectly derive from `Throwable`
  - Child classes: `Error` and `Exception`
- Important `Throwable` methods
  - `getMessage()`
    - Detailed message about error
  - `printStackTrace()`
    - Prints out where problem occurred and path to reach that point
  - `getStackTrace()`
    -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

Printing 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?

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?

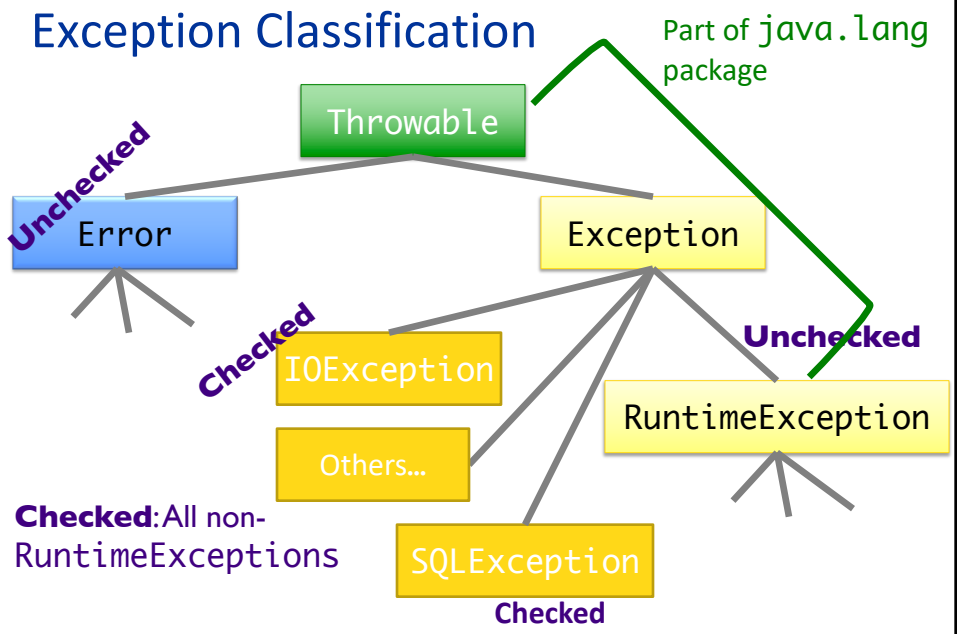
- Program's code should not and can not throw an object of this type
- This is an example of an *Unchecked* exception

Exception Classification: Exception

Throwable
Error
Exception

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 these exceptions
     - Compiler enforces
   - Examples: `IOException`, `SQLException`

Exception Classification



Checked: All non-RuntimeExceptions

Categories of Exceptions

Unchecked

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

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

1. Derived from the class `Error`
   - Any line of code can generate because it is an internal JVM error
   - Don't worry about what to do if this happens
2. Derived from the class `RuntimeException`
   - Indicates a bug in the program
   - Fix the bug
   - Examples: `ArrayOutOfBoundsException`, `NullPointerException`, `ClassCastException`

Checked Exceptions

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

THROWING EXCEPTIONS

Methods and Exceptions Example

- `BufferedReader` has method `readLine()`
  - Reads a line from a *stream*, such as a file or network connection
- Method 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: in Javadoc, document under what conditions each exception is thrown
  - `@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 = in.readLine();
     return str1;
}
```

Throws an IOException

- readData calls readLine, which can throw an IOException
- If readLine throws this exception to our method
  - readData *throws* the exception as well
  - Whoever calls readData will handle exception

Throwing An Exception We Created

```
if (grade < 0 || grade > 100) {
     throw new IllegalArgumentException();
}
```

Equivalent in Python?

1. Create a new object of class **IllegalArgumentException**
   - Class derived from **RuntimeException**
2. throw it
   - Method ends at this point
   - Calling method handles exception

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 (grade < 0 || grade > 100) {
      throw new IllegalArgumentException(
            "Grade is not in valid range (0-100)");
}
```

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 everything 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`

`Birthday.java`

Birthday Error Handling Discussion

- Design decision:
  - Since month and day are not independent, should be set together rather than separately
- Check all the error cases before setting the instance variables
  - Don't want an inconsistent resulting birthday
- IllegalArgumentException is appropriate
  - Programming error
  - Should catch those errors before executing program

Goal: Failure Atomicity

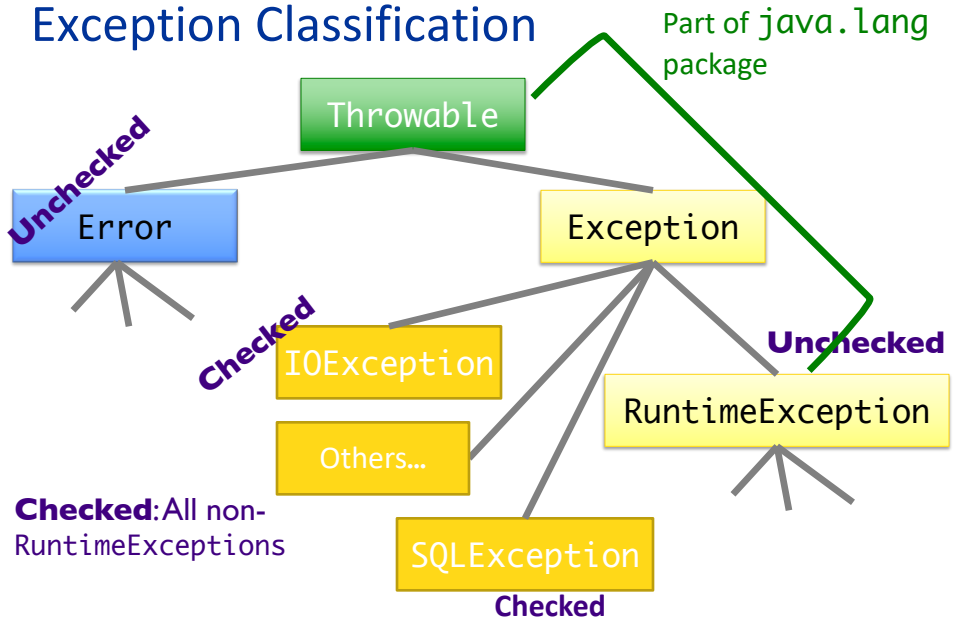
- After an object throws an exception, the object should be in a well-defined, usable state
  - A failed method invocation should leave object in state prior to invocation
- Approaches:
  - Check parameters/state before performing operation(s)
  - Do the failure-prone operations first
  - Use recovery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

Javadoc Guidelines about @throws

- Always report if throw **checked** exceptions
- Report any unchecked exceptions that the caller might reasonably want to catch
  - Exception: `NullPointerException`
  - Allows caller to handle (or not)
  - Document exceptions that are independent of the underlying implementation
- `Errors` should **not** be documented as they are unpredictable

CATCHING EXCEPTIONS

Exception Classification



Checked: All non-RuntimeExceptions

Catching Exceptions

- After we throw an exception, some part of program needs to ***catch*** it
- What does it mean to catch an exception?
  - Program knows how to deal with the situation that caused the exception
  - Handles the problem—hopefully gracefully, without exiting

Try/Catch Block

- The simplest way to catch an exception
- Syntax:

Python equivalent?

```
try {
      code;
      more code;
}
catch (ExceptionType e) {
      error code for ExceptionType;
}
catch (ExceptionType2 e) {
      error code for ExceptionType2;
}
…
```


Looking Ahead

- Assignment 7 – due Wednesday
  - 9 people still hadn't accepted the invitation
- Exam 1 – Friday
  - Online, timed exam: 70 minutes
    - No class Friday – office hours during that time
    - Open: Friday, 9:30 a.m. – Sunday, 11:59 p.m.
  - Open book/notes/slides – but **do not** rely on that
    - NOT open internet
  - Prep document online
  - 3 sections: Very Short Answer, Short Answer, Coding