Part 1: Composition, Aggregation, and Delegation
Part 2: Exceptions

Lecture 11

COMP 401, Fall 2018

Layers of Abstraction

- Simple objects
  - Object state (i.e., fields) are basic data types
- As objects become more complex…
  - Encapsulated object state is complex enough to require additional “layers of abstraction”.
  - Object state is modeled by a combination of other objects.
  - Recall early examples with Triangle and Point

Composition and Aggregation

- Two design techniques for creating an object that encapsulates other objects.
  - Whole / part relationship
  - Any specific situation is not necessarily strictly one or the other.
- In a nutshull…
  - Composition
    - The individual parts that make up the whole are "owned" solely by the whole.
      - They don't otherwise have a reason for being.
  - Aggregation
    - The individual parts that make up the whole may also exist on their own outside of the whole.
      - Or even as a part of other objects.

Example of Aggregation

- lec11.ex01
  - Course
    - Models a course at the university
    - Encapsulates Room, Professor, and a list of Student objects

Characteristics of Aggregation

- Encapsulated objects provided externally
  - As parameters to constructor
  - Getters and setters for these components often provided.
- Encapsulated objects may be independently referenced outside of the aggregating object
  - Including possibly as part of another aggregation.

Example of Composition

- lec11.ex02
  - Car
    - Two implementations: HondaOdyssey and Porche911
    - Both encapsulate an implementation of Horn and an implementation of Engine
  - Implementations of Engine written with inheritance.
    - EngineImpl provides most of the implementation
      - Abstract class
    - ManualEngine and AutomaticEngine are subclasses
      - Override setGear()

Characteristics of Composition

- Encapsulated objects created internally
  - Usually within the constructor
  - No setters and often no getters
- Encapsulated objects do not make sense outside of the abstraction.
  - Not shared with other abstractions
- Functionality / state of encapsulated objects only accessible through the composition.

Delegation

- Claiming an “is-a” relationship with an interface but relying on another object to actually do the work.
  - Independent of the concepts of aggregation or composition.
- Delegation example
  - lec11.ex03

Exceptions

Exception Handling

- What is an exception?
  - Unexpected (or at least unusual or abnormal)
  - Disruptive
  - Possibly fatal

Before exception handling...

- Strategy 1: Global Error Code
  - Well-documented global variable
  - Set to some sort of code when something goes wrong.
  - Onus on programmer to check the code when appropriate.
  - lec11.ex04

Before exception handling...

- Strategy 2: Special return value.
  - Specific return value(s) that are interpreted as errors.
  - Common conventions:
    - Procedures (i.e., does not produce a result)
      - 0 indicates success (i.e., no error)
      - Less than zero indicates some type of error.
      - Mapping of values to types of error documented with procedure.
    - Functions (i.e., expected to produce a value)
      - If reference type is expected, then null signals error.
      - If value type is expected, choose some “out of range” value to signal an error.
  - lec11.ex05

Drawbacks to ad-hoc approaches

- Inconsistent
- May not have an “out-of-range” value to use to signal error condition.
- Puts onus on programmer to properly detect and handle return value as an error.
- Limited information about the error.
- Difficult to extend in future development.

Exception Handling

- Formal mechanism for detecting and dealing with exceptions
  - Most modern OO languages provide it.
    - Java, C++, C#, Python, Ruby, etc.
- Two parts:
  - Throwing
    - Signaling that an exception has occurred.
    - Also known as “raising” an exception.
  - Catching
    - Handling an exception when it occurs.

Benefits of Exception Handling

- Promotes good software engineering.
  - Consistency
  - Modularity
  - Separation of concerns
  - Extensibility
- Provides an abstraction hierarchy for error information.
  - Information about when and why the error occurred is encapsulated into an object.
- Improves code organization
  - Separates error handling code from “normal” code.
  - Provides a facility for ensuring that critical code executes no matter what happens.

Exception Handling in Java

- Two kinds of exceptions:
  - Checked exceptions
    - Possibly valid situations that system should have some way of dealing with.
    - Examples:
      - Trying to open a file that doesn't exist.
      - Reading past the end of a file.
      - The printer is out of ink.
  - Unchecked exceptions
    - Also known as "runtime" exceptions.
    - Exceptions that should never happen and usually indicate a bug or flaw in logic.
      - Out of bounds indexing of an array.
      - Illegal cast of an object reference.
  - Terms "checked" and "unchecked" are a little misleading.
    - Should think of it more as "compiler will check on how you are dealing with the possibility of this exception at compile time" vs. not that.
  - In either situation, the syntax and mechanisms of throwing and catching are the same.

Runtime Exceptions

- To signal a runtime exception…
  - Create a RuntimeException object and then "throw" it.
    - Constructor accepts a string message.
  - Usually done on the same line.
- Default behavior is to end program and print information about the exception to the console.
- lec11.ex06

Catching Exceptions

- May not want the program to end.
  - May be able to recover or otherwise deal with the problem.
- try-catch blocks
  - Put code that might generate the exception inside a “try” block.
  - Follow try block with one or more “catch” blocks.
  - Each catch block is associated with a specific exception type and declares a variable that is set to the exception object if that type of exception is raised.
    - Exception object has methods to get information about the error.
- lec11.ex07

Extending RuntimeException

- You can create your own runtime exception types by subclassing from runtime exception.
- Allows you to encapsulate custom information about the exception.
- lec11.ex08

Dealing with more than one type

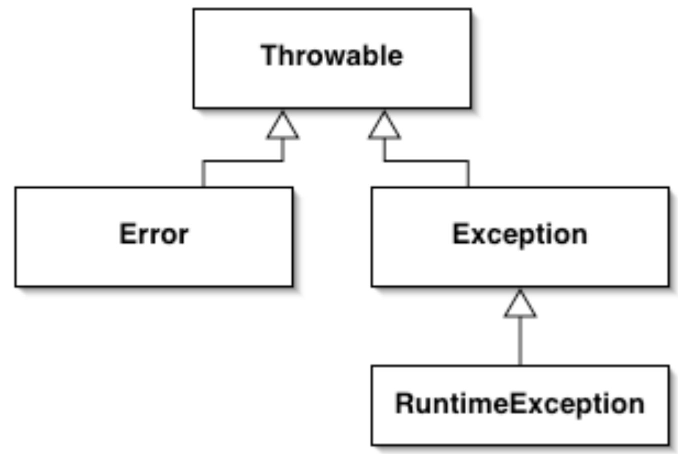
- Can specify different catch blocks for different types of exceptions.
  - An exception matches the first catch block with a declared exception variable that has an “is-a” relationship with the raised exception.
- lec11.ex09

The finally block

- Sometime we need some code to run no matter what happens.
  - Often this is the case in order to free up some system resource such as closing a file or closing a network connection.
- finally block will execute no matter what happens (exception or no).
- lec11.ex10

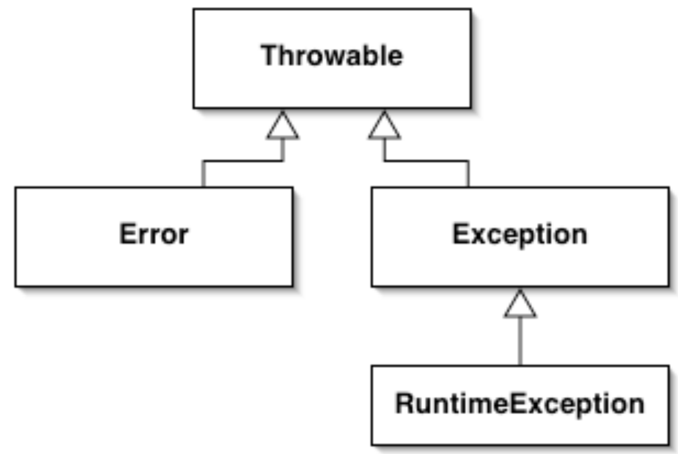
The Throwable Class Hierarchy

- Throwable
  - Superclass for all objects that can be "thrown"
- Error
  - Superclass for errors generally caused by external conditions.
- Exception
  - Superclass for exceptions generally caused by internal conditions.



Checked vs. Unchecked

- Error, RuntimeException, and their subclasses are “unchecked” exceptions.
- All other subclasses of Exception are “checked”.



Checked Exceptions

- Checked exceptions are subject to the “catch or specify” rule.
  - A method that could potentially throw a checked exception must declare the possibility as part of its method signature.
  - A method that calls another method that throws a checked exception must either:
    - Catch: Include code that catches the exception.
    - Specify: Declare the possibility of the exception as part of its own method signature.
- Why?
  - Forces user of method to explicitly handle the exception or pass the buck to whomever calls it.
  - Enforces good error handling of exceptional conditions that might normally arise.
    - In contrast to exceptions that really shouldn’t ever happen.
- lec11.ex11

General Principle: Be Specific

- Use an existing exception type.
  - There are lots.
  - If semantics of the exception match well, then go ahead and use it.
- Create your own exception type.
  - Subclass either RuntimeException or Exception
- lec11.ex12.v1
  - Notice how exceptions raised by Scanner or Song constructor transformed into context-specific exception for Playlist.
  - Also note how error message can be retrieved from exception object.
    - See handling of PlaylistFormatError in main()
    - See reference page for Exception for more.

General Principle: Catch Late

- Exceptions should rise to level where application has enough context to deal with them effectively.
  - Catching exception just because you can is not always the right thing to do.
    - Pass the buck unless response to this exception under all or nearly all circumstances is well-understood at this point.
  - Look again at lec11.ex12.v1
    - In particular, note handling of FileNotFoundException
  - lec11.ex12.v2
    - Note printStackTrace() method of Exception in Main1
    - Note differences in how FileNotFoundException handled in Main1 vs. Main2
      - Main1 not even given the chance because handled by playlist.
      - Main2 has ability to re-prompt for new filename and does so.

General Principle: Throw Early

- Validate values as early as possible.
  - Rather than waiting for exception generated by invalid values sent to other code.
    - Particularly apropos for null values that may later cause a NullPointerException
    - Exception generated by null pointer is not very specific
    - Almost always have to look higher in the stack trace to see what the real problem is.
- lec11.ex12.v3
  - Changed how Main2 reads filename in order to demonstrate this point.

Odds and Ends

- Remember that the scope of a variable is limited to the surrounding block.
  - Sometimes an issue if you declare a variable inside a try block but then need its value outside of the try block later.
- Sibling catch blocks can reuse the same variable name for the declared error object.
- A catch block associated with an exception class cannot precede a catch block associated with a subclass.
  - Results in unreachable code.
- try-catch-finally blocks can be nested.
- Interfaces must declare any checked exceptions thrown by any of its implementations.
- Finally always runs
  - Even if you "return" from within a try or catch block
  - lec11.ex13