Lecture 15
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

• lec15.v1
  – Notice how less specific general exception raised by Scanner 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 not always the right thing to do.
  – Look again at lec15.v1
    • In particular, note handling of FileNotFoundException
  – lec15.v2
    • Note printStackTrace() method of Exception in Main1
    • Note differences in how FileNotFoundException handled in Main1 vs. Main2
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 cause NullPointerException
    • Exception generated is not very specific
    • Almost always have to look higher in the stack trace to see what the real problem is.

• lec15.v3
Finally always runs

• Question from last time:
  – Will finally block always run even if catch block causes control flow to break out of loop or return from method?
  – lec15.v4
Catch block order

• Catch blocks are evaluated in order.
  – First catch block where specific type of exception raised matches data type of exception variable.
    • I.e., exception thrown has “is-a” relationship with exception variable declared in catch block.
  – Can take advantage of this to catch related exceptions or to provide a general catch-all.
    • Java won’t let a more specific catch block follow a more general catch block that will match before it.
      – Essentially detects the code as unreachable.
      – Try reordering IOException and FileNotFoundException in lec15.v2
    – lec15.v5
Assignment 6

• Demonstrate UI
Observer / Observable

• Official Gang of Four description:
  – Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

• Observable
  – The object of “interest”
    • Represents data or state that may change in the future.

• Observer
  – The “interested” objects
    • Represents parts of system that need to be updated or may need to take action if/when observable object changes.
Use Cases

• User Interfaces
  – User interface elements like buttons, scrollbars, etc. are “observable”
    • State changes correspond to clicks, drags, etc.
  – Application objects that must respond to user’s interactions with the UI are “observers”

• Asynchronous Programming
  – Also known as “event-based” programming
  – May have well-defined actions corresponding to events that may occur, but can’t know in advance which event will occur.
Basic Observer/Observable

- Defining Observer as an interface allows any class to act as an observer for Observable.
- Notice that Observable can have more than one observer.
  - And that they don’t know about each other or order of update.
- Drawbacks to this simple model?

```java
interface Observer {
    void update();
}

class Observable {
    ArrayList<Observer> observers;

    void register(Observer o) {
        // Adds o to list of observers
        observers.add(o);
    }

    void unregister(Observer o) {
        // Takes o off list of observers
        observers.delete(o);
    }

    void notify() {
        // Trigger update method on all observers
        for (Observer o : observers) {
            o.update();
        }
    }
}
```
Better Observer/Observable

- Allows Observer to register with more than one Observable and then detect which one changed.
- Allows Observable to provide additional information about what changed.

```java
interface Observer {
    void update(Observable o, Object update_info);
}

class Observable {
    ArrayList<Observer> observers;

    void register(Observer o) {
        // Adds o to list of observers
        observers.add(o);
    }

    void unregister(Observer o) {
        // Takes o off list of observers
        observers.remove(o);
    }

    void notify(Object info) {
        // Trigger update method on all observers
        for (Observer o : observers) {
            o.update(this, info);
        }
    }
}
```
Observer/Observable in Java

• Java provides skeleton Observer / Observable that you can extend.
  – Don’t have to use them.
  – May contain more functionality than you really need.
  – The pattern is defined by relationship between objects and their interaction.
    • Not the specific method names and/or implementation.