Objectives

- Collections
  - Maps
- Traversing
- Exceptions

On my Twitter feed:

> "Rather than teach everyone to code,
> let's teach them to think.
> The coding can come later; it's easier."
> - @rob_pike

Analysis of `equals` methods

```
public boolean equals(Object o){
   if(((Birthday) o).getDay() != this.getDay())
      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;
}
```

Using booleans in if statements

```
if( this.equals(that) == true ) {
   System.out.println("equal!");
}
```

```
if( this.equals(that) ) {
   System.out.println("equal!");
}
```

Review

- What are the 3 components of the Java Collection Framework?
- What data types can collections hold?
- How can we convert a primitive type into its respective Wrapper Object type?
- What is the syntax to say what type the collection holds?
- Why did I wait until now to show you Eclipse?

Eclipse

- Very helpful – *after* you know what you’re doing
  - Gives suggestions for fixes
    - You need to think through what the appropriate fix is

Eclipse Hints

- After you have written a method, type

```
/**
```

before the method, and then hit enter and the Javadocs template will be automatically generated for you

- Use command-spacebar for possible completions

Eclipse Discussion

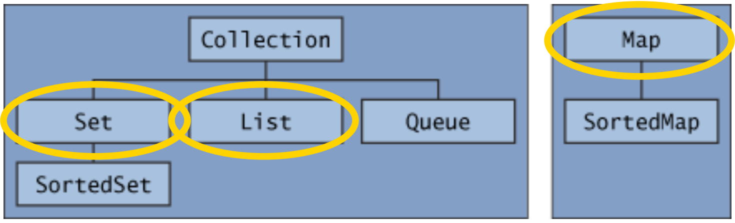
- Helpful hints
  - Control-spacebar
  - Format the file
  - Auto-templates for Javadocs

Review: Collections Framework

- **Interfaces**
  - Abstract data types that represent collections
  - Collections can be manipulated *independently* of implementation
- **Implementations**
  - Concrete implementations of collection interfaces
  - Reusable data structures
- **Algorithms**
  - Methods that perform useful computations on collections, e.g., searching and sorting
  - Reusable functionality
  - ***Polymorphic***: same method can be used on many different implementations of collection interface

Review: Core Collection Interfaces

- Encapsulate different types of collections



MAPS

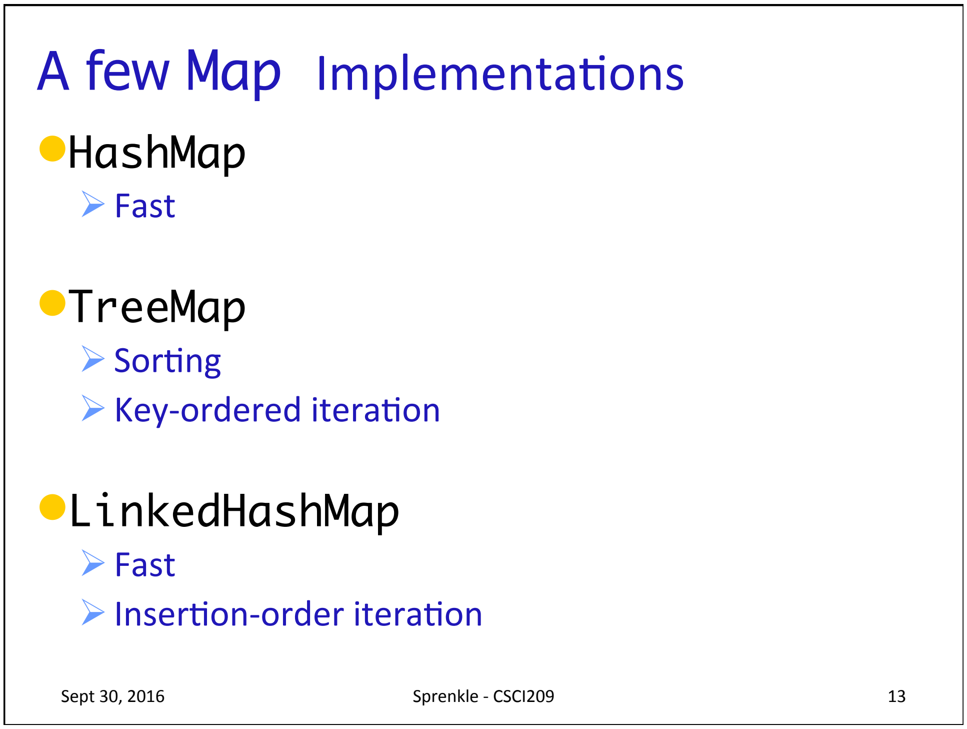
Maps

- Maps keys (of type <K>) to values (of type <V>)
- No duplicate keys
  - Each key maps to at most one value

Map Interface

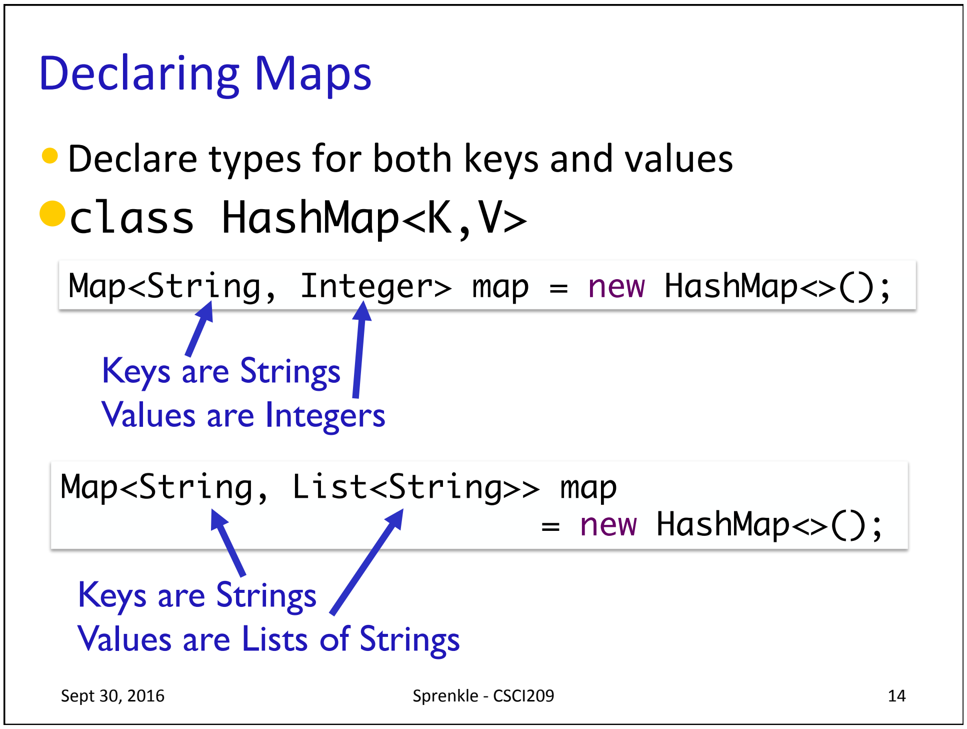
- `<V> put(<K> key, <V> value)`
  - Returns old value that key mapped to
- `<V> get(Object key)`
  - Returns value at that key (or null if no mapping)
- `Set<K> keySet()`
  - Returns the set of keys

And more …



A few Map Implementations

- HashMap
  - Fast
- TreeMap
  - Sorting
  - Key-ordered iteration
- LinkedHashMap
  - Fast
  - Insertion-order iteration



Declaring Maps

- Declare types for both keys and values
- class HashMap<K,V>

```
Map<String, Integer> map = new HashMap<>();
```

Keys are Strings
Values are Integers

```
Map<String, List<String>> map
                             = new HashMap<>();
```

Keys are Strings
Values are Lists of Strings

ALGORITHMS

Collections Framework’s Algorithms

- *Polymorphic algorithms*
- Reusable functionality
- Implemented in the `Collections` class
  - Static methods, 1st argument is the collection
  - Similar to `Arrays` class, which operates on arrays

Overview of Available Algorithms

- Sorting – optional `Comparator`
- Shuffling
- Searching – binarySearch

* Only `Lists`

- Routine data manipulation: reverse*, copy*, fill*, swap*, addAll
- Composition – frequency, disjoint
- Finding min, max

TRAVERSING COLLECTIONS

Traversing Collections: For-each Loop

- For-each loop:

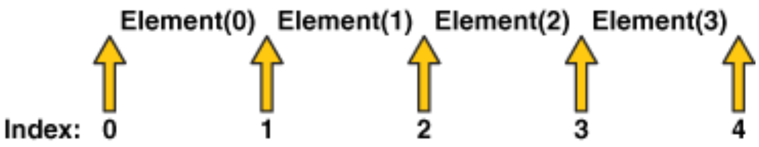
Or whatever data type is appropriate

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

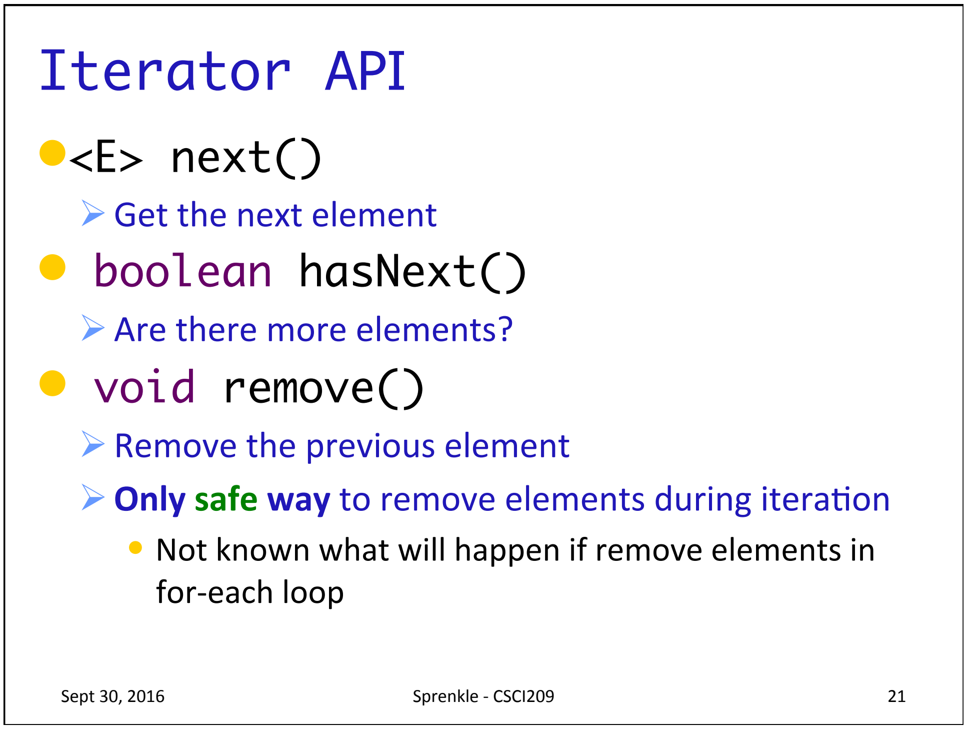
- Valid for all Collections
  - Maps (and its implementations) are not Collections
    - But, Map's keySet() is a Set and values() is a Collection

Iterator: Like a Cursor

- Always between two elements

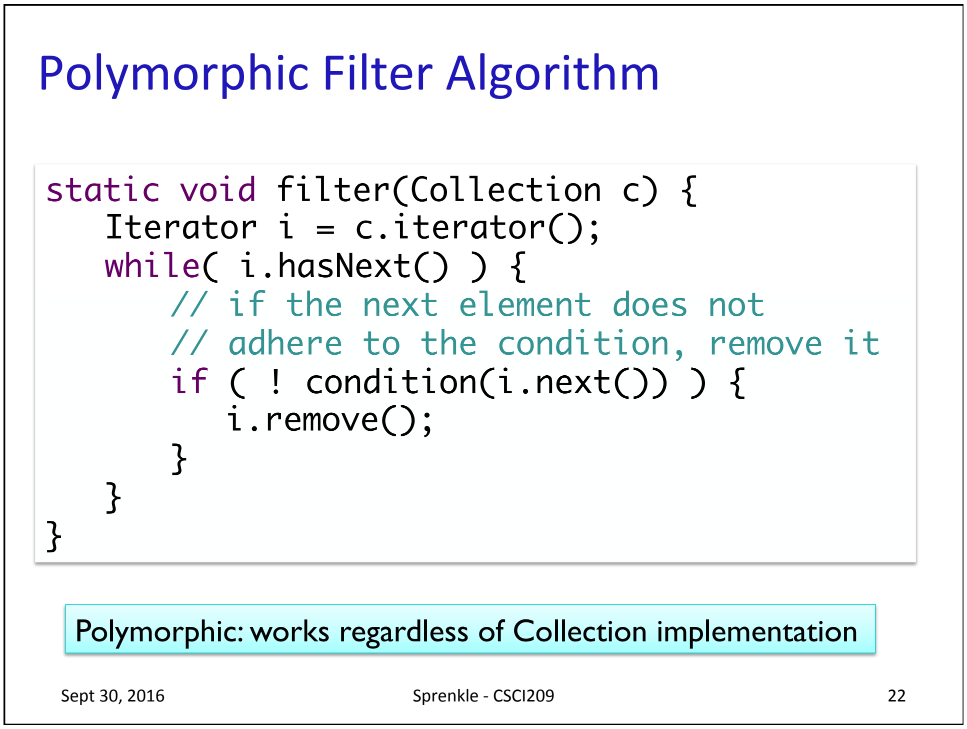


```
Iterator<Integer> i = list.iterator();
while( i.hasNext()) {
      int value = i.next();
      …
}
```



Iterator API

- <E> next()
  - Get the next element
- boolean hasNext()
  - Are there more elements?
- void remove()
  - Remove the previous element
  - **Only safe way** to remove elements during iteration
    - Not known what will happen if remove elements in for-each loop



Polymorphic Filter Algorithm

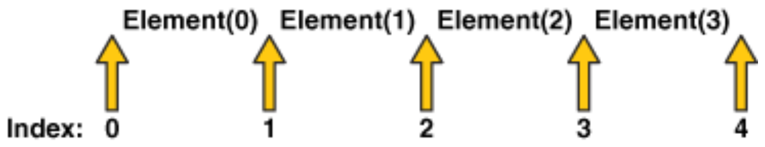
```
static void filter(Collection c) {
   Iterator i = c.iterator();
   while( i.hasNext() ) {
      // if the next element does not
      // adhere to the condition, remove it
      if ( ! condition(i.next()) ) {
         i.remove();
      }
   }
}
```

Polymorphic: works regardless of Collection implementation

Traversing Lists: `ListIterator`

- Methods to traverse list backwards too
  - `hasPrevious()`
  - `previous()`
- To get a `ListIterator`:
  - `listIterator(int position)`
    - Pass in `size()` as position to get at end of list

Key difference



How Not to Iterate

- Don’t use `get` to access `List`
  - If implementation is a `LinkedList`, performance is reeeeeally slow

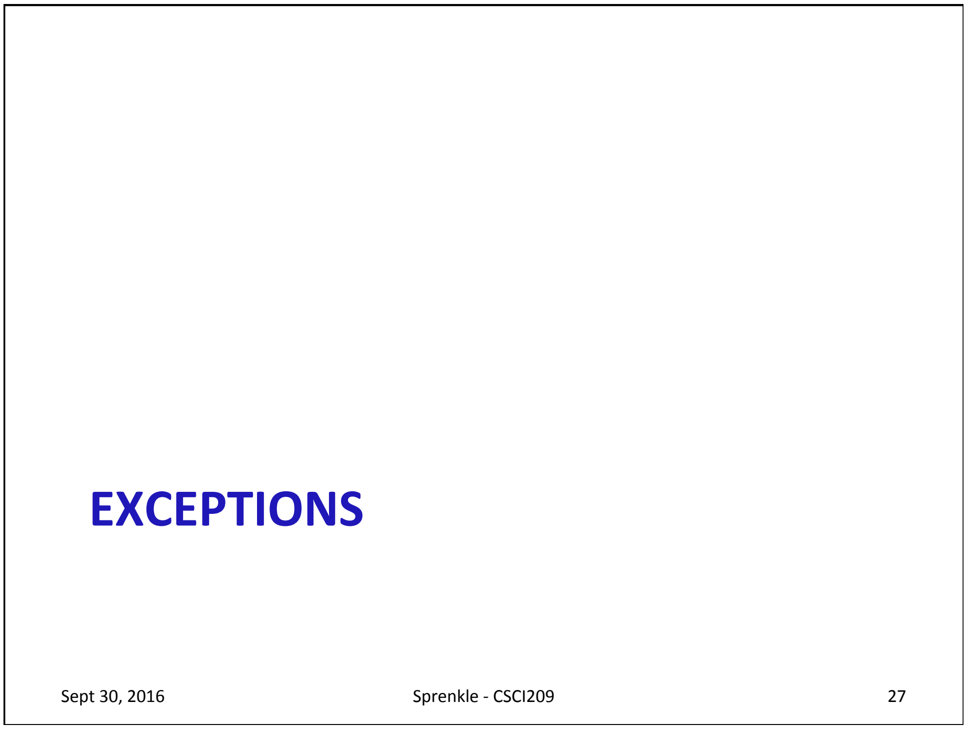
```
for (int i = 0; i < list.size(); i++) {
   count += list.get(i); // do something
}
```

Benefits of Collections Framework

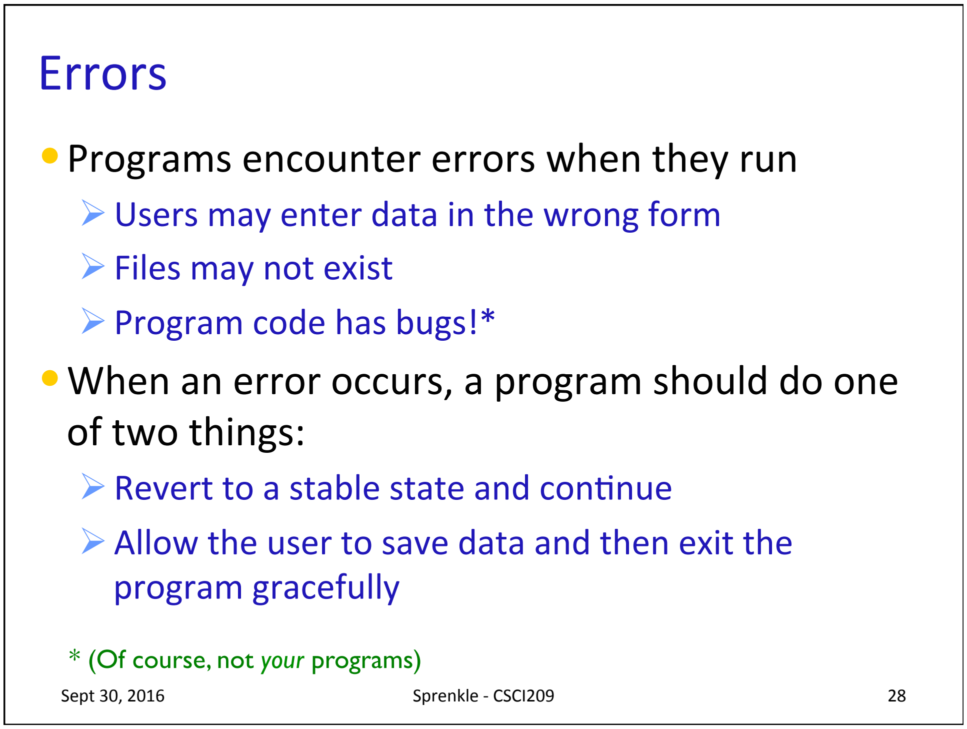
- ?

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



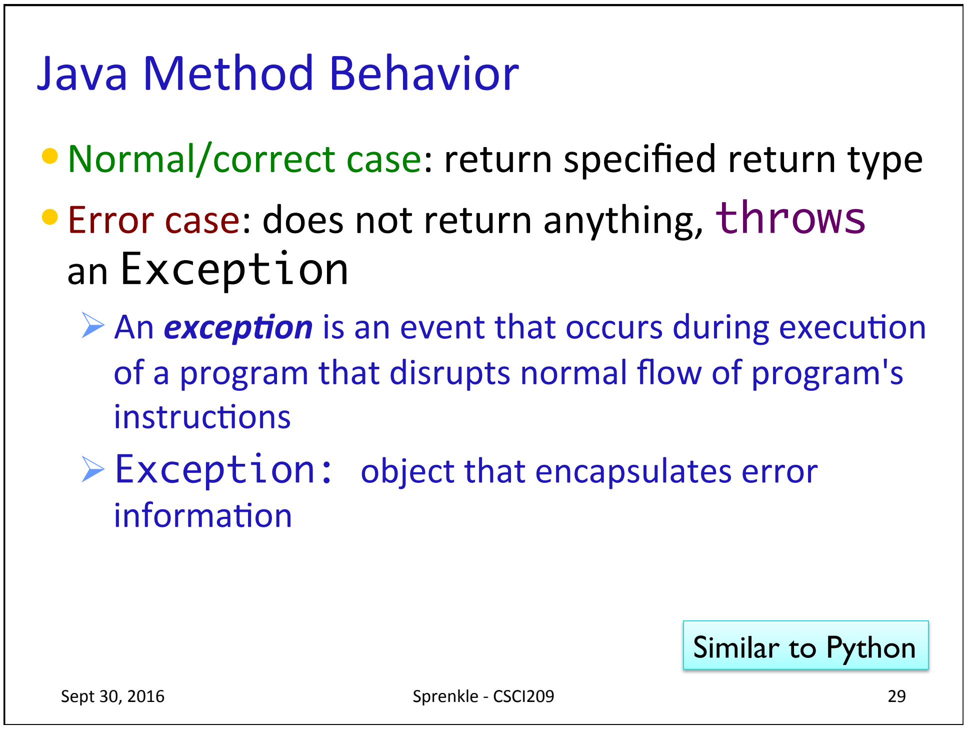
EXCEPTIONS



Errors

- Programs encounter errors when they run
  - ➢ Users may enter data in the wrong form
  - ➢ Files 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

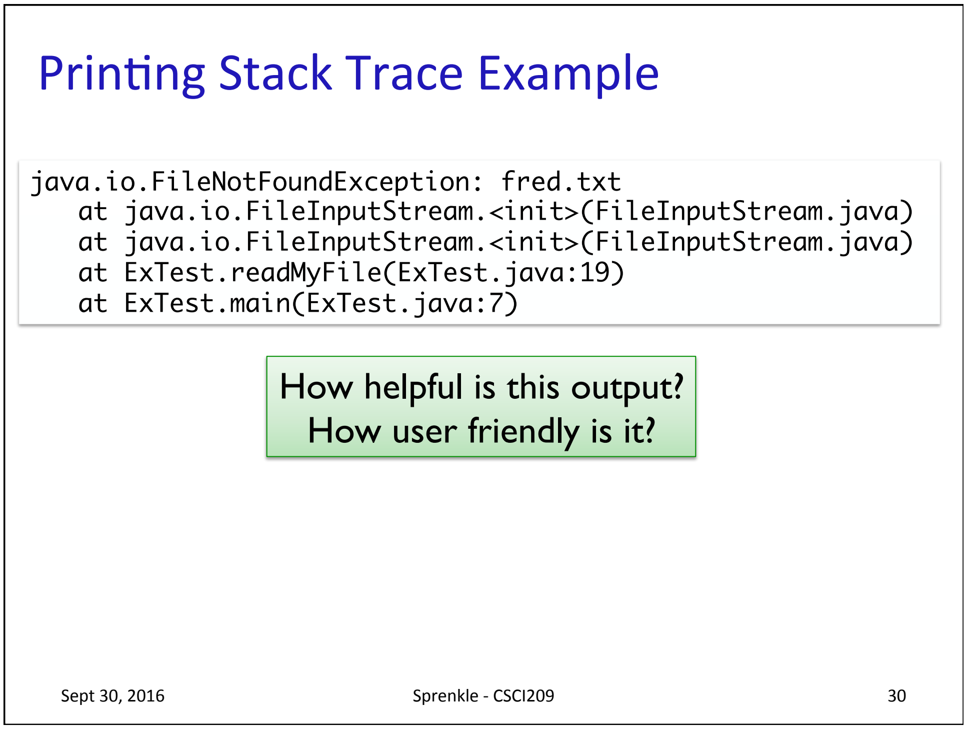
* (Of course, not *your* programs)



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



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?

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?

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

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

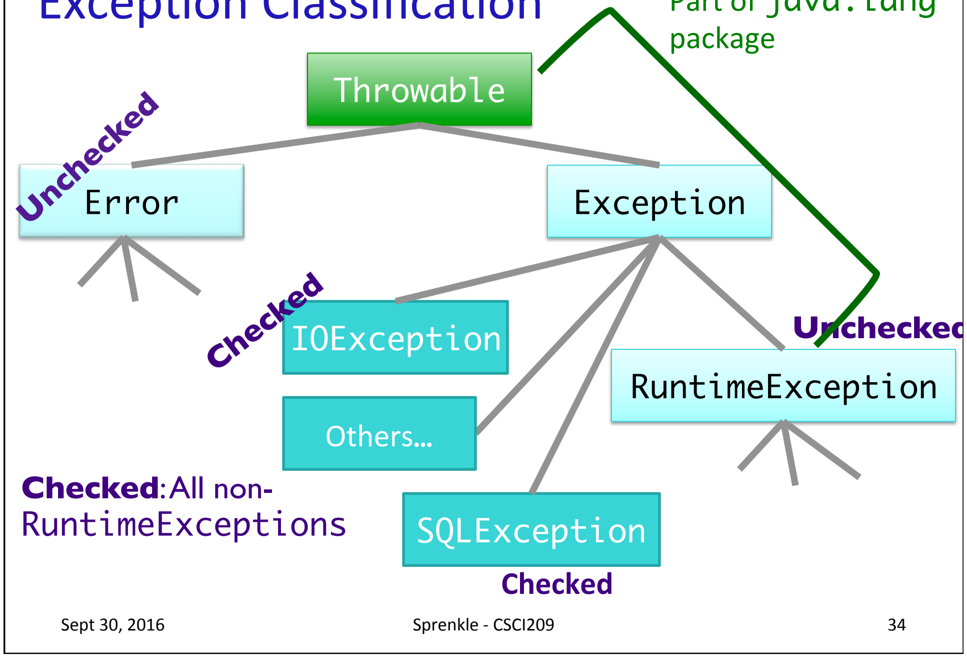
When can that happen in Java?

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

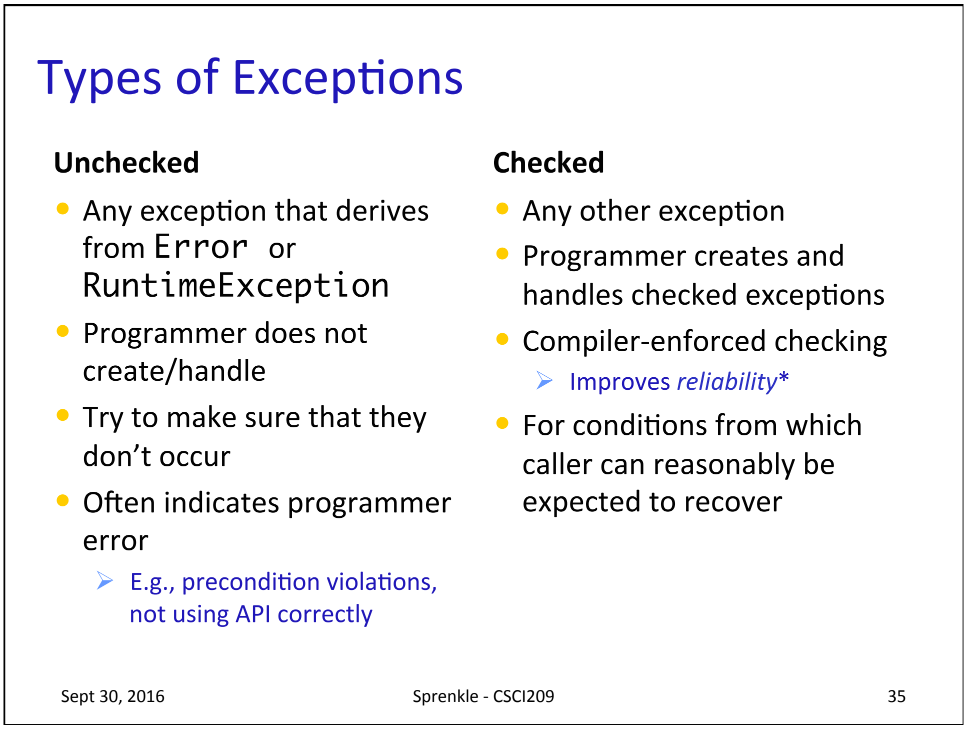
Exception Classification: Exception

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

Exception Classification



Checked: All non-`RuntimeExceptions`



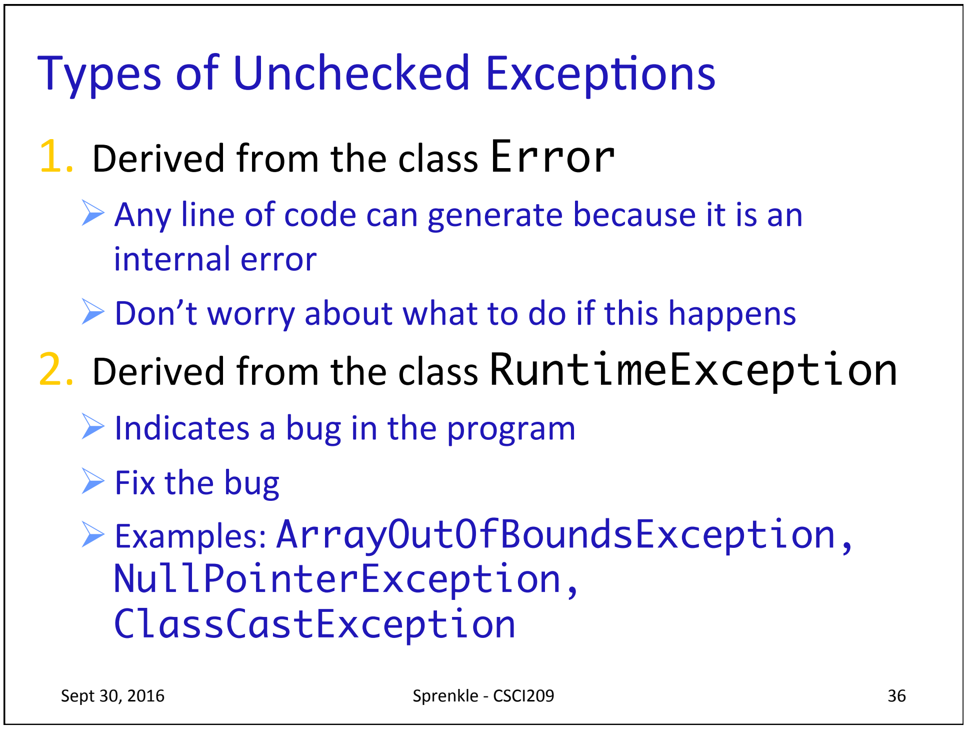
Types of Exceptions

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, 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

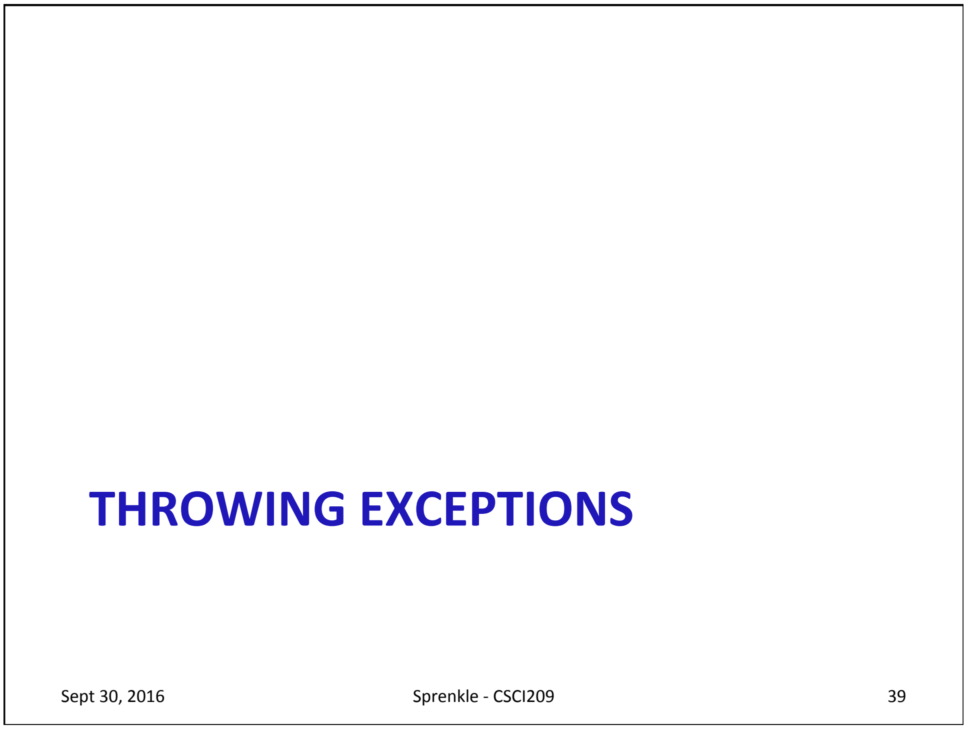
1. Derived from the class `Error`
   - Any line of code can generate because it is an internal error
   - Don’t worry about what to do if this happens
2. Derived from the class `RuntimeException`
   - Indicates a bug in the program
   - Fix the bug
   - Examples: `ArrayOutOfBoundsException`, `NullPointerException`,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

Checked Exceptions

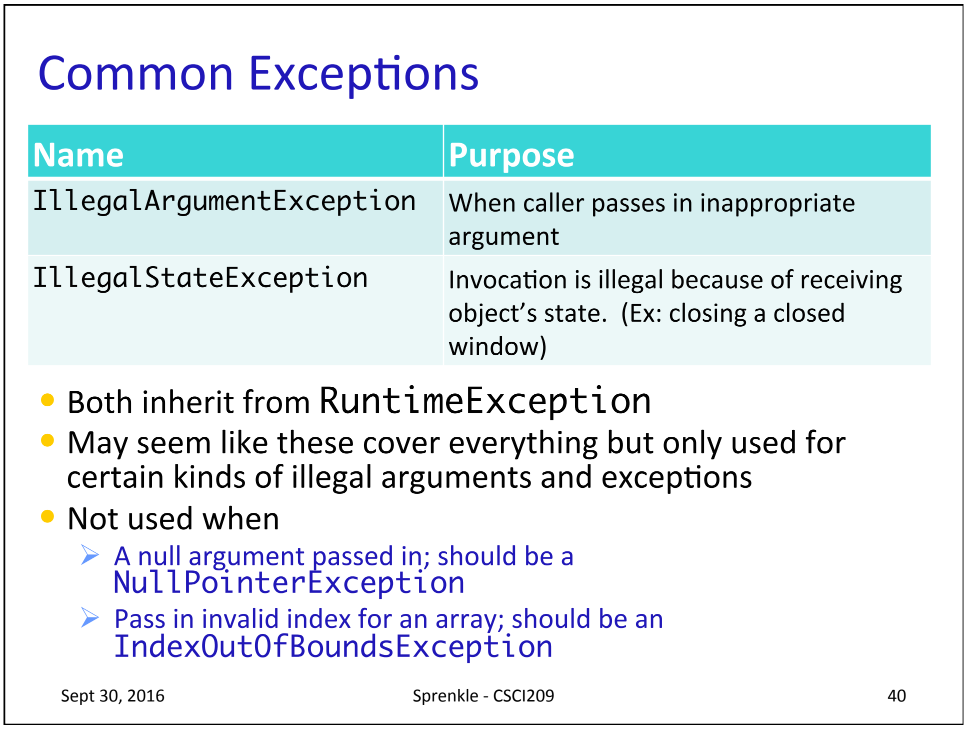
- Need to be handled by your program
    - Compiler-enforced
    - Improves reliability*
- For each method, tell the compiler:
    - What the method returns
    - What could possibly go wrong
        - *Advertise* the exceptions that a method throws
        - Helps users of your interface know what method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

Discussion: Why Checked and Unchecked Exceptions?

- Why do we have exceptions that the compiler doesn't force the programmer to check?
    - Think about examples of unchecked exceptions (`ArrayOutOfBoundsException`, `NullPointerException`, `ClassCastException`) and when those exceptions can occur



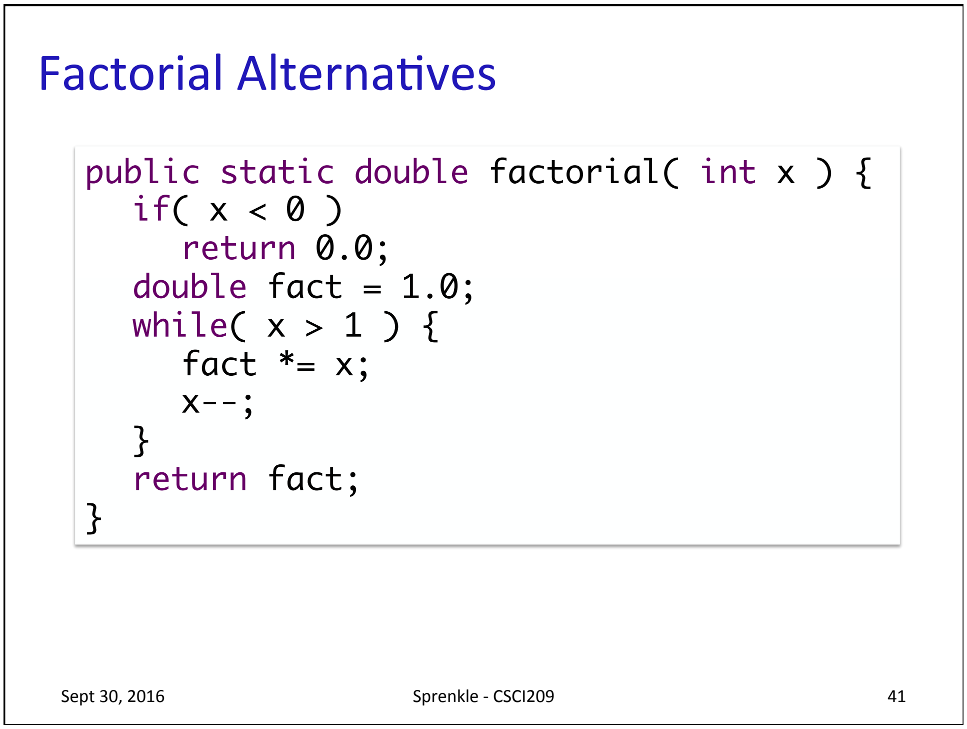
THROWING EXCEPTIONS



Common Exceptions

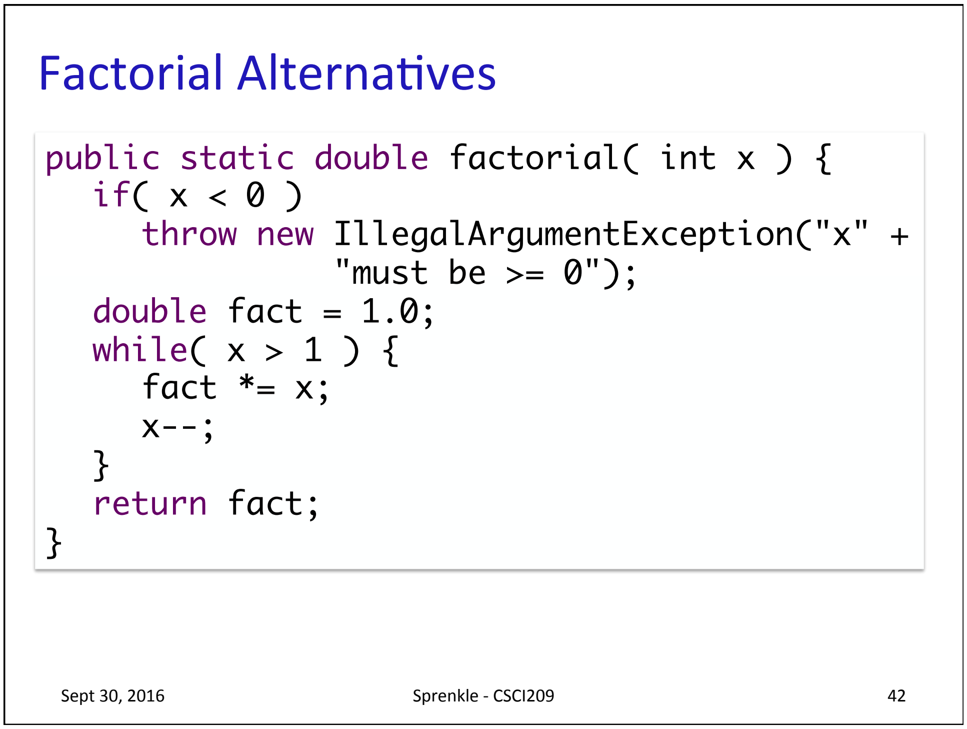
Name	Purpose
`IllegalArgumentException`	When caller passes in inappropriate argument
`IllegalStateException`	Invocation is illegal because of receiving object's state. (Ex: closing a closed window)

- 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`



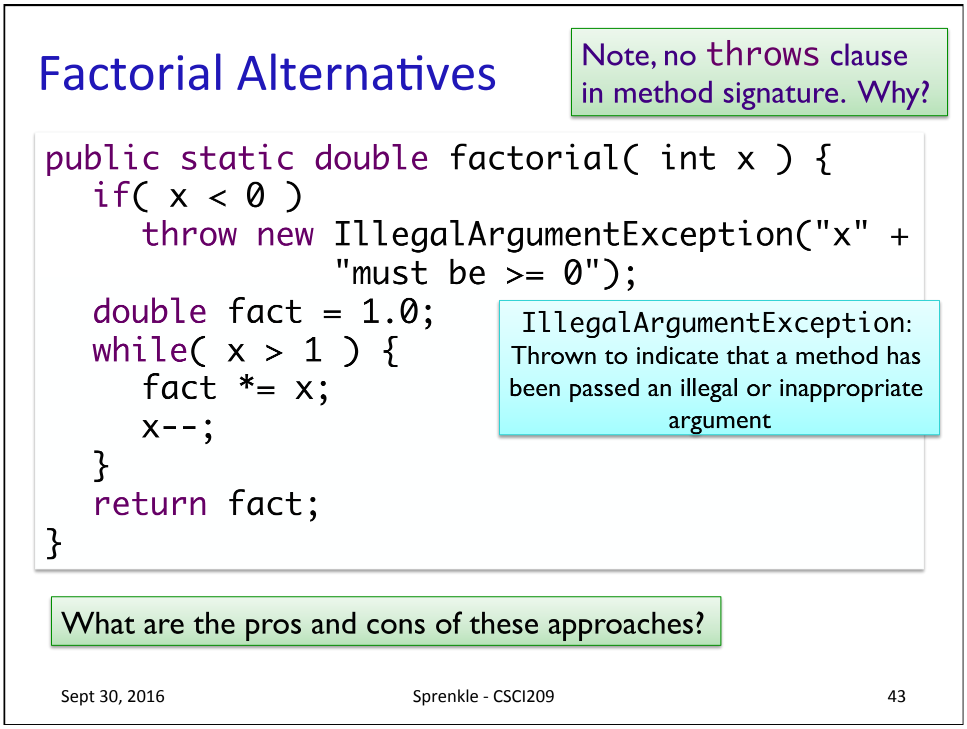
Factorial Alternatives

```
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

```
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;
}
```



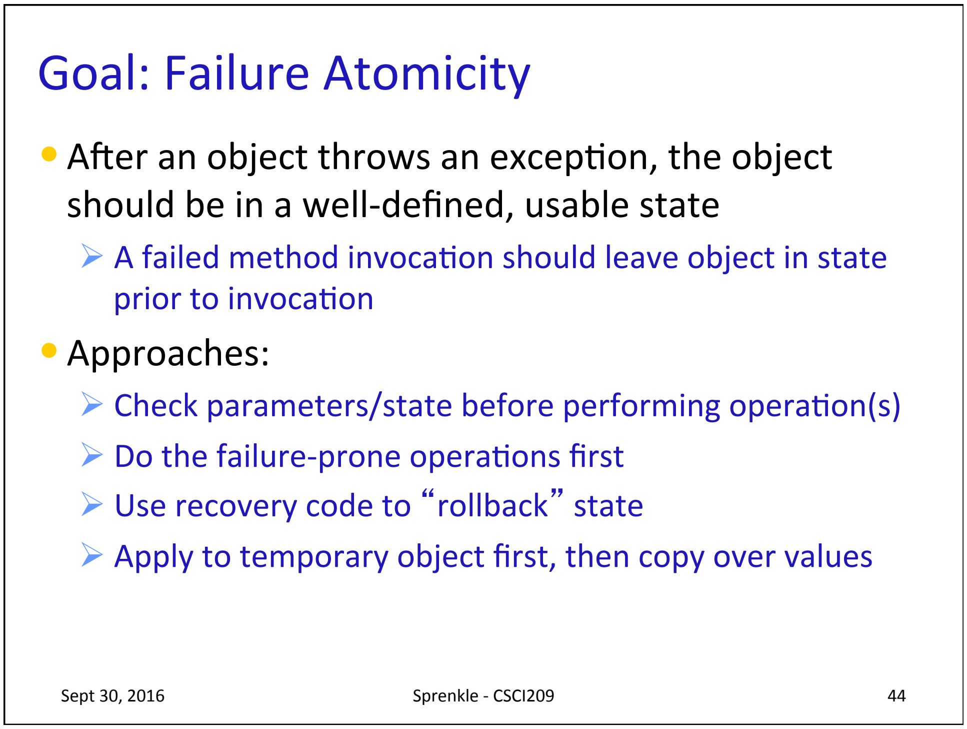
Factorial Alternatives

Note, no throws clause in method signature. 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;
}
```

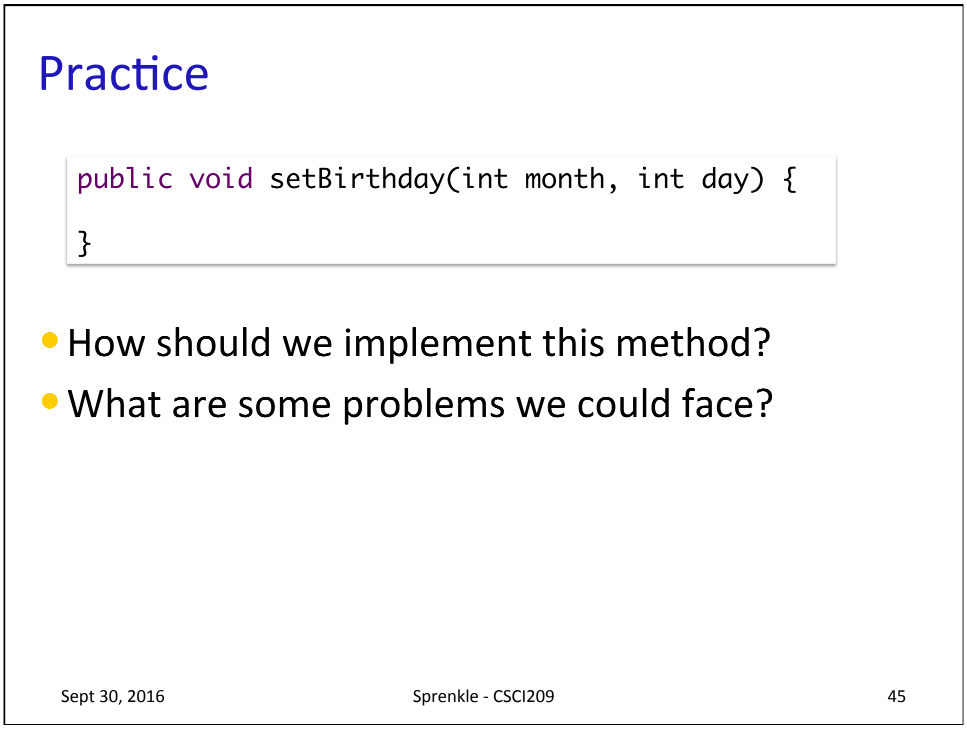
IllegalArgumentException: Thrown to indicate that a method has been passed an illegal or inappropriate argument

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

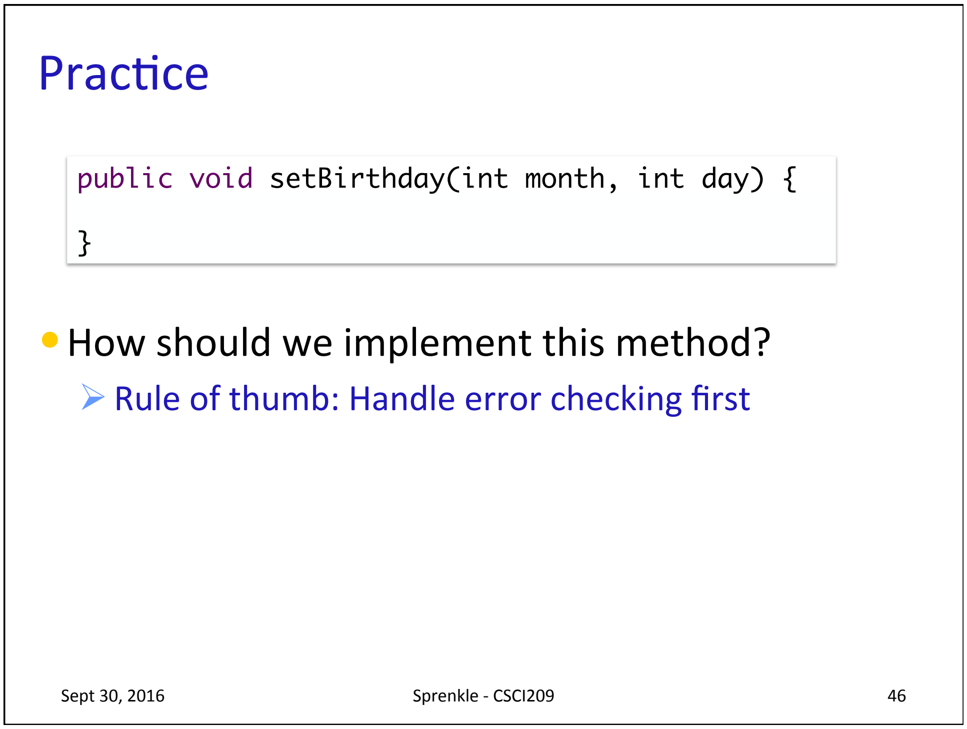


Practice

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

}
```

- How should we implement this method?
- What are some problems we could face?



Practice

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

}
```

- How should we implement this method?
  - Rule of thumb: Handle error checking first

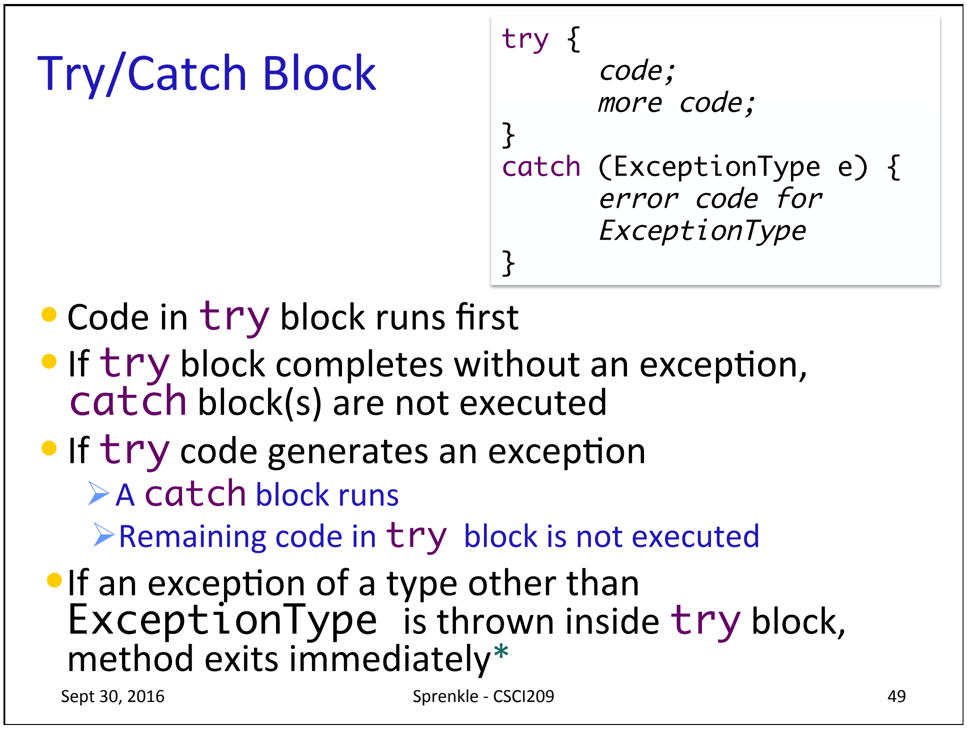
CATCHING EXCEPTIONS

Try/Catch Block

- The simplest way to catch an exception
- Syntax:

Python equivalent?

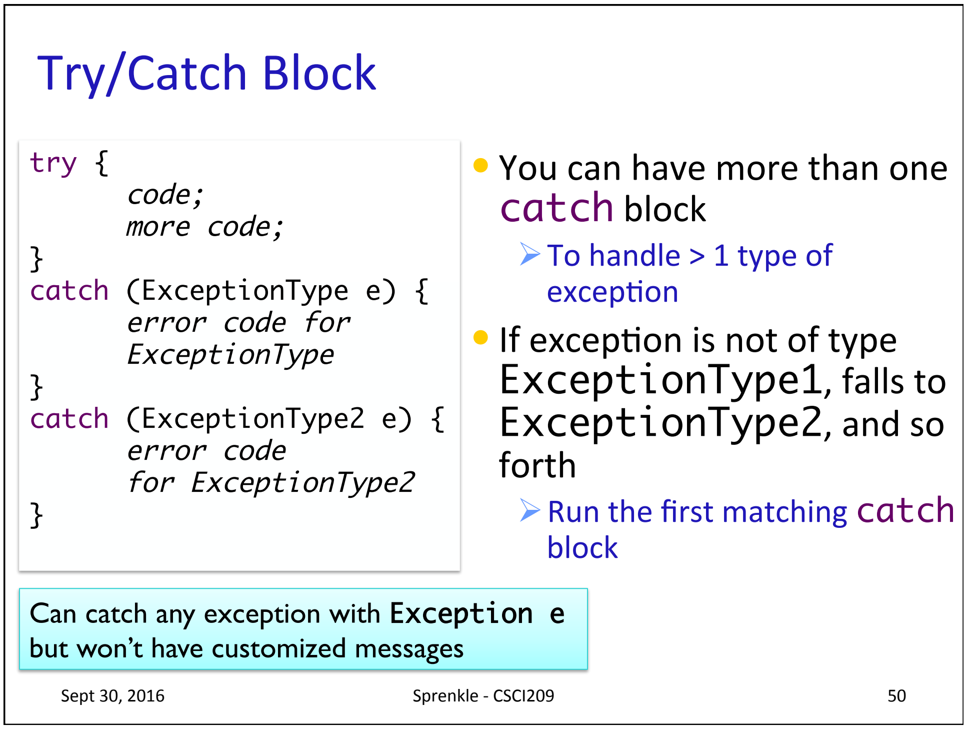
```
try {
      code;
      more code;
}
catch (ExceptionType e) {
      error code for ExceptionType;
}
catch (ExceptionType2 e) {
      error code for ExceptionType2;
}
…
```



Try/Catch Block

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

- Code in `try` block runs first
- If `try` block completes without an exception, `catch` block(s) are not executed
- If `try` code generates an exception
  - A `catch` block runs
  - Remaining code in `try` block is not executed
- If an exception of a type other than `ExceptionType` is thrown inside `try` block, method exits immediately*

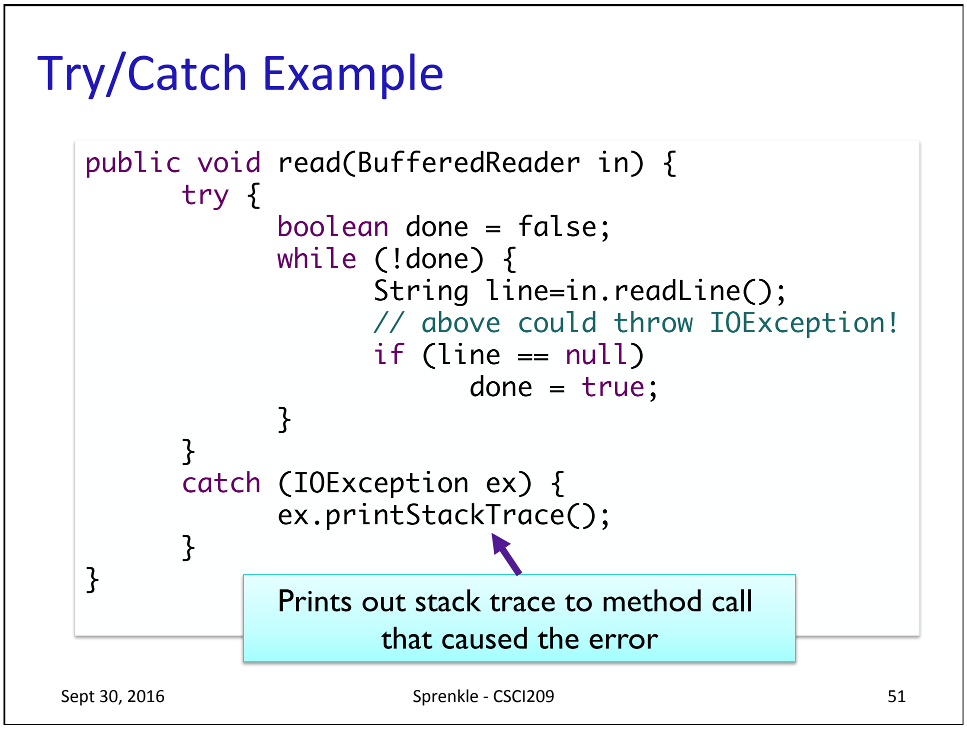


Try/Catch Block

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

- You can have more than one `catch` block
  - To handle > 1 type of exception
- If exception is not of type `ExceptionType1`, falls to `ExceptionType2`, and so forth
  - Run the first matching `catch` block

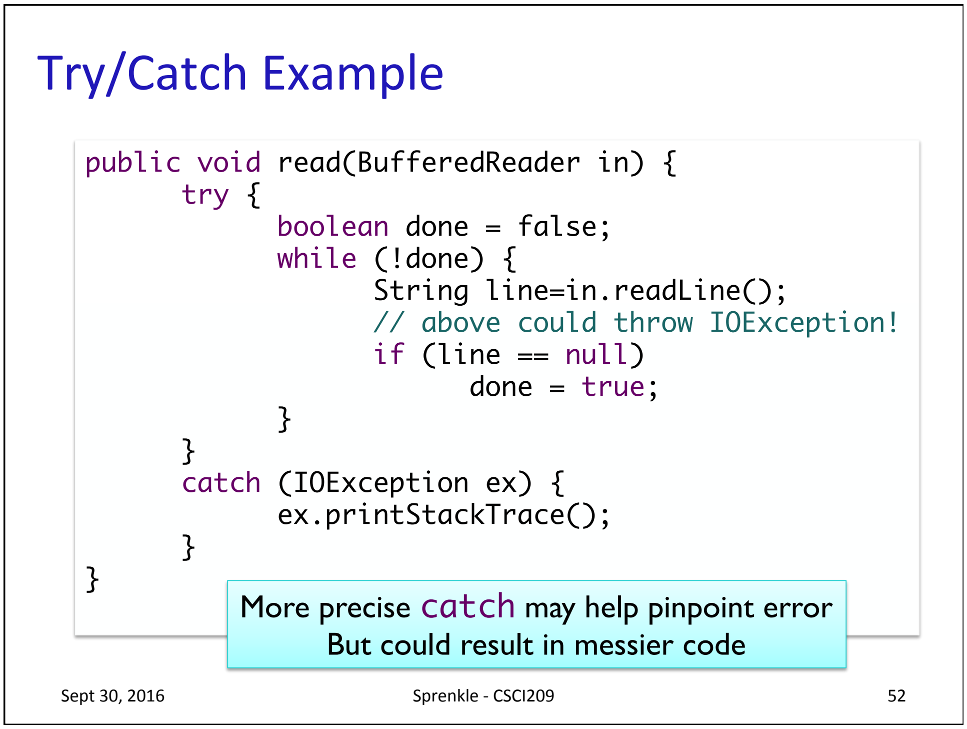
Can catch any exception with `Exception e` but won't have customized messages



Try/Catch Example

```
public void read(BufferedReader in) {
      try {
            boolean done = false;
            while (!done) {
                  String line=in.readLine();
                  // above could throw IOException!
                  if (line == null)
                        done = true;
            }
      }
      catch (IOException ex) {
            ex.printStackTrace();
      }
}
```

Prints out stack trace to method call that caused the error



Try/Catch Example

```
public void read(BufferedReader in) {
      try {
            boolean done = false;
            while (!done) {
                  String line=in.readLine();
                  // above could throw IOException!
                  if (line == null)
                        done = true;
            }
      }
      catch (IOException ex) {
            ex.printStackTrace();
      }
}
```

More precise `catch` may help pinpoint error
But could result in messier code

The finally Block

- Optional: add a finally block after all catch blocks
  - ➢ Code in finally block **always** runs after code in try and/or catch blocks
    - After try block finishes or, if an exception occurs, after the catch block finishes

```
try {
        …
}
catch (Exception e) {
        …
}
finally {
        …
}
```

- Allows you to clean up or do maintenance before method ends (one way or the other)
  - ➢ E.g., closing files or database connections

FinallyTest.java

Practice: try/catch/finally Blocks

```
try {
        statement1;
        statement2;
}
catch (EOFException e) {
        statement3;
        statement4;
}
finally {
        statement5;
}
```

- Which statements run if:
  - ➢ Neither statement1 nor statement2 throws an exception
  - ➢ statement1 throws an EOFException
  - ➢ statement2 throws an EOFException
  - ➢ statement1 throws an IOException

What to do with a Caught Exception?

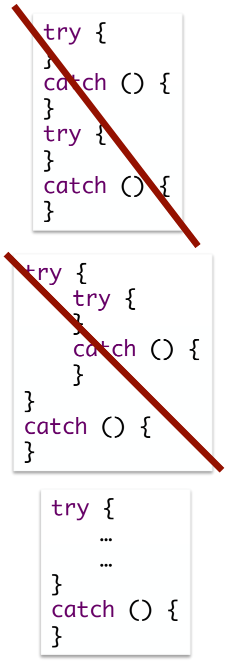
- Dump the stack after the exception occurs
  - What else can we do?
- Generally, two options:
  1. Catch the exception and recover from it
  2. Pass exception up to whoever called it

Summary: Methods Throwing Exceptions

- API documentation tells you if a method can throw an exception
  - If so, you **must** handle it
- If your method could possibly throw an exception (by generating it or by calling another method that could), advertise it!
  - If you can't handle every error, that's OK…let whoever is calling you worry about it
  - However, they can only handle the error if you advertise the exceptions you can't deal with

Programming with Exceptions

- Exception handling is slow
- Use one big `try` block instead of nesting `try-catch` blocks
  - Speeds up Exception Handling
  - Otherwise, code gets too messy
- Don't ignore exceptions (e.g., `catch` block does nothing)
  - Better to pass them along to higher calls

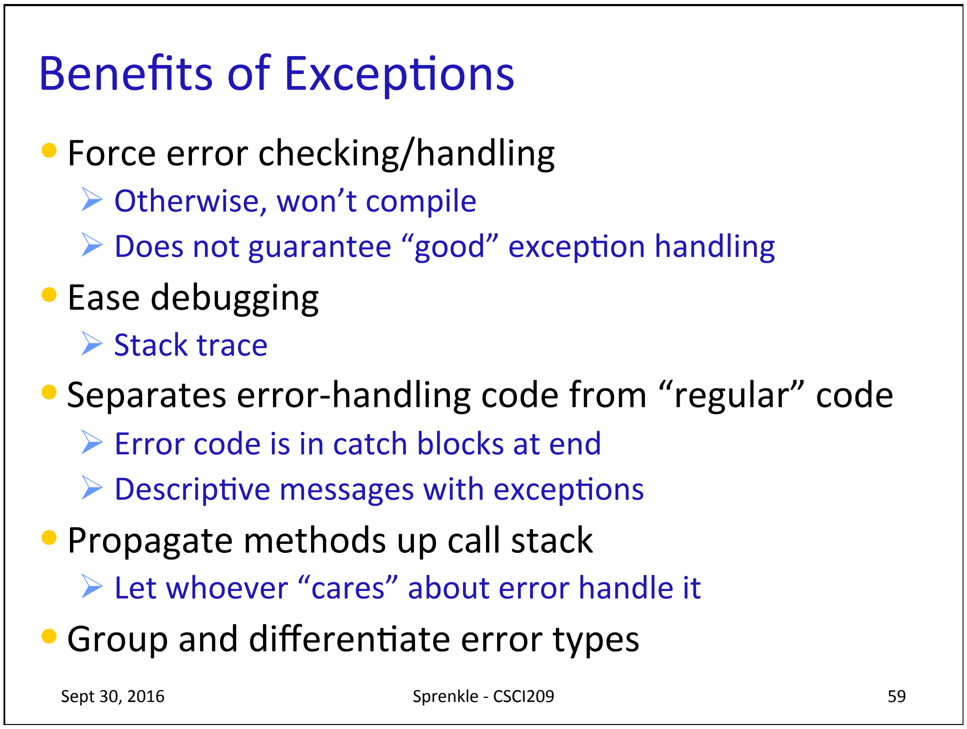


```
try {
}
catch () {
}
try {
}
catch () {
}
```

```
try {
     try {
     }
     catch () {
     }
}
catch () {
}
```

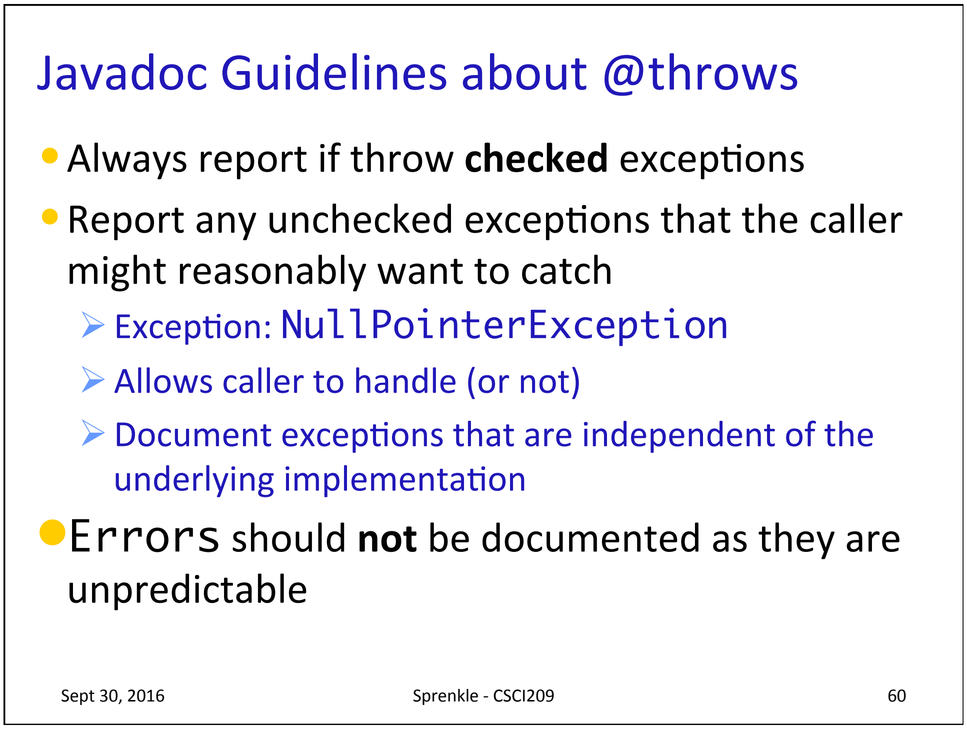
```
try {
     …
     …
}
catch () {
}
```

Benefits of Exceptions?



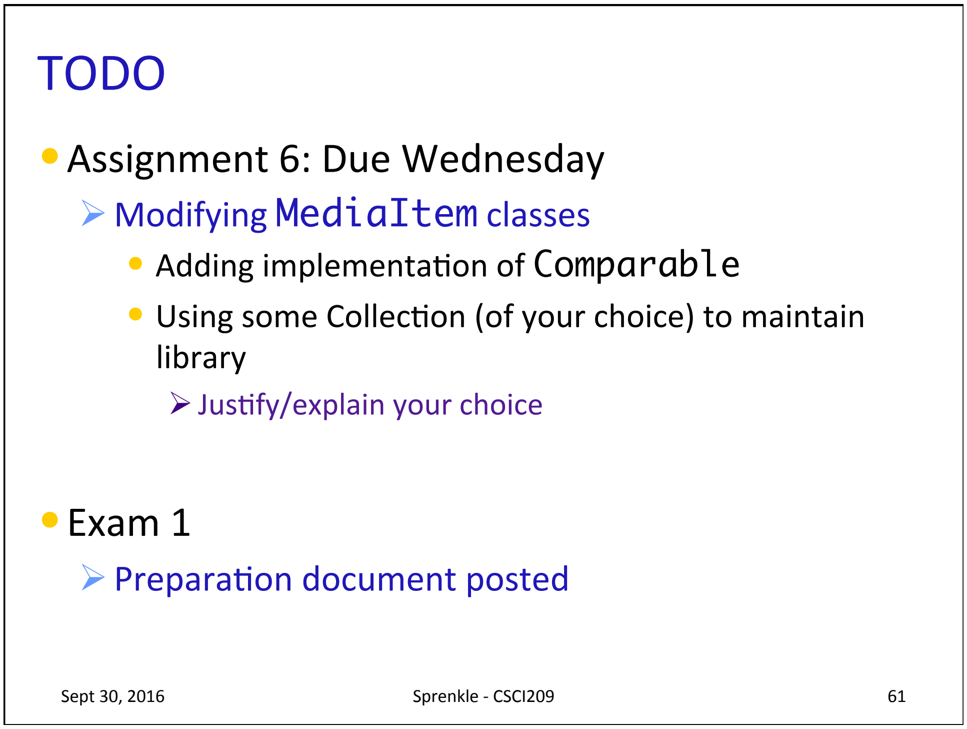
Benefits of Exceptions

- Force error checking/handling
  - Otherwise, won't compile
  - Does not guarantee "good" exception handling
- Ease debugging
  - Stack trace
- Separates error-handling code from "regular" code
  - Error code is in catch blocks at end
  - Descriptive messages with exceptions
- Propagate methods up call stack
  - Let whoever "cares" about error handle it
- Group and differentiate error types



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



TODO

- Assignment 6: Due Wednesday
  - Modifying `MediaItem` classes
    - Adding implementation of `Comparable`
    - Using some Collection (of your choice) to maintain library
      - Justify/explain your choice
- Exam 1
  - Preparation document posted