Objectives

- Collections wrap up
- Exceptions
- Eclipse

Change in Today's Office Hour

- 12:15-1:15 p.m. – already updated on Canvas

Review

- What are *wrapper* classes? When do we use them?
- I made the claim that this is the preferred way to create an object variable that adheres to an interface:

```
Interface variable = new Implementation();
Example: List<Card> hand = new ArrayList<>();
```

  - Why is that the preferred way? What is the design principle it adheres to?
  - Review Deck.java from the examples from Monday. Point to code snippets where Deck.java adheres to that design principle
- What are the components of the Java Collections Framework?
  - What are benefits of the Java Collections Framework?

SETS

`Set` Interface

- No duplicate elements
  - Needs to determine if two elements are “logically” the same (`equals` method)
- Models mathematical set abstraction

Set Interface

- `boolean add(<E> o)`
  - Add to set, only if not already present
- `int size()`
  - Returns the number of elements in the list
- And more! (`contains`, `remove`, `toArray`, …)
  - Note: no get method – can't get #3 from the set because sets aren't ordered.

Some Set Implementations

- `HashSet`
  - Implements set using *hash table*
    - `add`, `remove`, and `contains` each execute in O(1) time
  - Used more frequently
  - Faster than `TreeSet`
  - No ordering

- `TreeSet`
  - Implements set using a *tree*
    - `add`, `remove`, and `contains` each execute in O(log n) time
  - Sorts

MAPS

Maps

- Python called these *dictionaries*

- Maps keys (of type <K>) to values (of type <V>)

- No duplicate keys
  - Each key maps to at most one value

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

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

`MapExample.java`

ALGORITHMS

Collections Framework's Algorithms

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

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

Review: Traversing Collections: For-each Loop

- For-each loop:

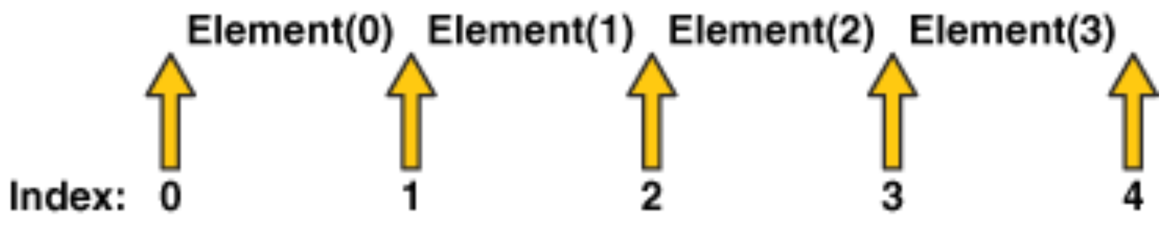
Or whatever data type is appropriate

```
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`

Traversing Lists: Iterator

- Always between two elements



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

Helpful to use if removing elements from list during iteration

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

Error Handling

- 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)

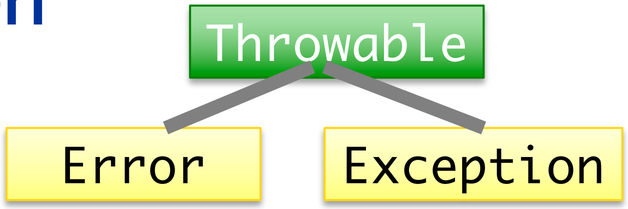
Java Method Behavior

- 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

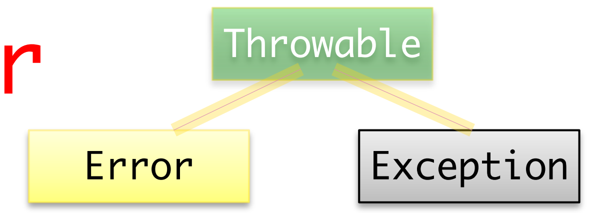
Similar to Python

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



Exception Classification: Error

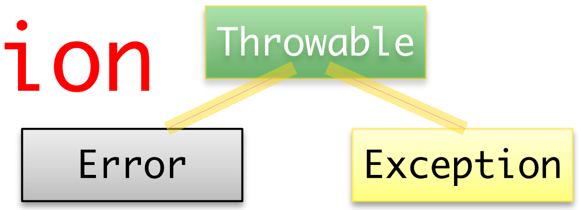


- An internal error
- Strong convention: reserved for JVM
  - JVM-generated when resource exhaustion or an internal problem
    - Example: Out of Memory error

When can that happen in Java?

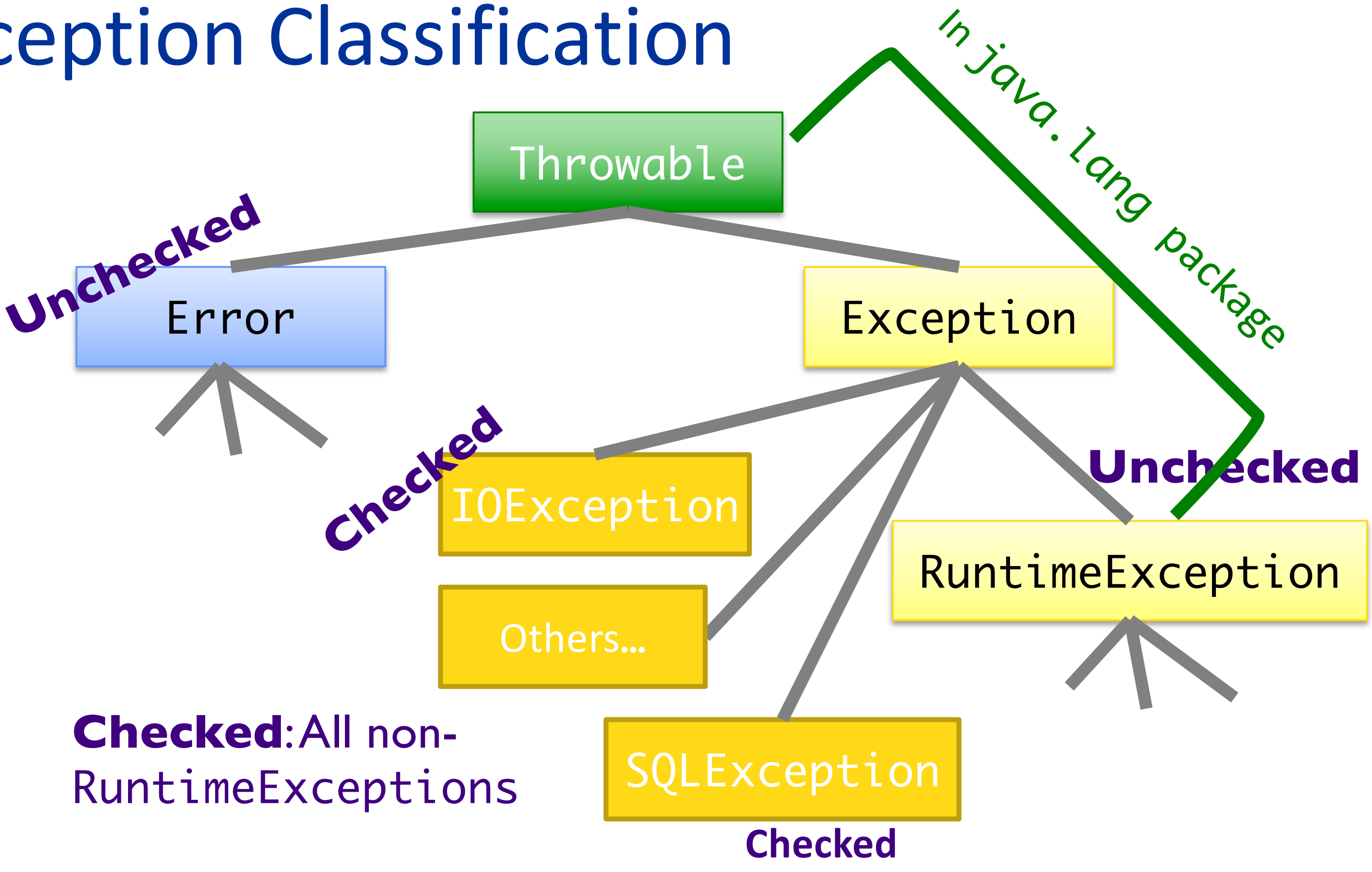
- Program’s code should not and can not throw an object of this type
- This is an example of an *Unchecked* exception

Exception Classification: `Exception`



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 that programmer handles them
   - 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 prevent RuntimeExceptions**
  - Often indicates programming error
  - E.g., precondition violations, not using API correctly, dividing by 0

Checked

- Any other exception
- For conditions from which caller can reasonably be expected to recover
- Compiler-enforced checking
  - Program MUST handle
  - Improves *reliability**

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, try to prevent
   - 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

eclipse

eclipse https://www.eclipse.org/

- Open source integrated development environment (IDE) for Java
- Described as "an open extensible IDE for anything and nothing in particular"
- Provides a robust Java development environment
- Incorporates popular software development tools like JUnit, Maven, and git
- Plugins allow extensibility

Project/Code Organization

- `workspace` directory contains all projects
  - Located in your home directory, unless you specified otherwise
- Use ***projects*** to organize your code
- Within a project
  - `src/` directory contains `.java` files
  - `bin/` directory contains `.class` files
    - Often hidden in GUI

Java Made Easier

- Creating class's basic functionality
  - See Source and Refactor menus
- Gives you a list of methods for an object
  - After you type object.
  - Then shows parameters for methods
- Automatically creates template of Javadoc
  - When you type /**
- Autocompletion of variables, methods
- Formatting code …
- Shows unused fields/variables
- Shows compiler errors
- …

Eclipse Demo

- Create a new `Birthday` class
  - Generate `main` method, Comments
- Demonstrate Source menu
  - Generate constructor, toString
  - Override `equals` method
- Create a String object, see methods used
- Demonstrate Refactor menu
  - Rename a field
  - Extract a method (month name)
- Run the Birthday Class (main)
  - Command line arguments
- Using git

Why can a Java IDE provide this functionality?

Eclipse Hints

- After you have written a method, type

  `/**`

  before the method, and then hit enter and the Javadocs comment template will be automatically generated for you
- Use `command-spacebar` for possible completions
- Use `command-shift-F` to format code

Eclipse Tradeoffs

- Very helpful – *after* you know what you're doing
  - You know
    - Code is compiled before executed
    - Structure of classes
    - How to fix errors
- Eclipse can handle the "routine" for you
  - That wasn't "routine" for you a few weeks ago
  - Help you focus on the important design considerations
- Gives suggestions for fixes
  - **You need to think through what the appropriate fix is**
    - Sometimes, it's "I'm not done yet"
  - Don't say "Eclipse made me do <something>"
- Eclipse is a beast (memory hog)
  - If you have less than ~8GB of memory, Eclipse will be slow

Looking Ahead

- Eclipse set up for Friday
- Change in today’s office hour: 12:15-1:15