

## Objectives

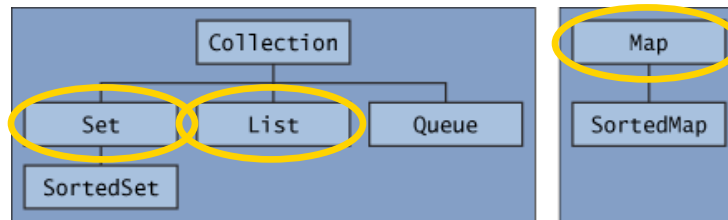
- Collections
  - Maps
- Traversing Collections
- Exception handling

## 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  $\langle K \rangle$ ) to values (of type  $\langle V \rangle$ )
- No duplicate keys
  - Each key maps to at most one value

Sept 30, 2016

Sprenkle - CSCI209

5

## Map Interface

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

And more ...

Sept 30, 2016

Sprenkle - CSCI209

6

## A few Map Implementations

- **HashMap**

- Fast

- **TreeMap**

- Sorting
- Key-ordered iteration

- **LinkedHashMap**

- Fast
- Insertion-order iteration

Sept 30, 2016

Sprenkle - CSCI209

7

## 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

Sept 30, 2016

Sprenkle - CSCI209

8

# ALGORITHMS

Sept 30, 2016

Sprenkle - CSCI209

9

## Collections Framework's Algorithms

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

Sept 30, 2016

Sprenkle - CSCI209

10

## Overview of Available Algorithms

- **Sorting** – optional Comparator
  - **Shuffling**
  - **Searching** – binarySearch
  - **Routine data manipulation**: reverse\*, copy\*, fill\*, swap\*, addAll
  - **Composition** – frequency, disjoint
  - **Finding min, max**
- \* Only Lists

## TRaversing Collections

## Two Ways to Iterate over Collections

- For-each loop
- Iterator

## Traversing Collections: For-each Loop

- For-each loop:

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

Or whatever data type is appropriate

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

## Traversing Collections: Iterator

- **Iterator**: Java Interface
- To get an **Iterator** from a **Collection** object:

```
Iterator<E> iterator()
```

- Returns an **Iterator** over the elements in this collection

- Example:

```
Iterator<String> iter = keys.iterator();
```

## 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

Sept 30, 2016

Sprenkle - CSCI209

17

## 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

Sept 30, 2016

Sprenkle - CSCI209

18

## 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



Sept 30, 2016

Sprenkle - CSCI209

19

## Enumeration

- Legacy class
- Similar to `Iterator`
- Example methods:
  - `boolean hasNextElements()`
  - `Object nextElement()`
- Longer method names
- Doesn't have remove operation

Sept 30, 2016

Sprenkle - CSCI209

20

## 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
}
```

## Synchronized Collection Classes

- For multiple threads sharing same collection
- Slow down typical programs
  - Avoid for now
- e.g., `Vector`, `Hashtable`
- See `java.util.concurrent`

Another example: `StringBuffer` is synchronized, whereas `StringBuilder` is not

## Benefits of Collections Framework

- ??

## 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)

Sept 30, 2016

Sprenkle - CSCI209

25

## 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

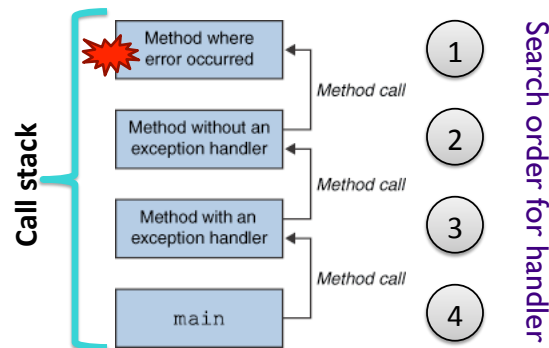
Sept 30, 2016

Sprenkle - CSCI209

26

## 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

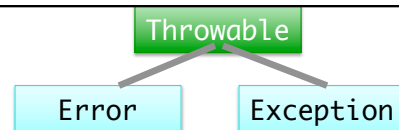


Sept 30, 2016

Sprenkle - CSCI209

27

## 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

Sept 30, 2016

Sprenkle - CSCI209

28

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

When can that happen in Java?

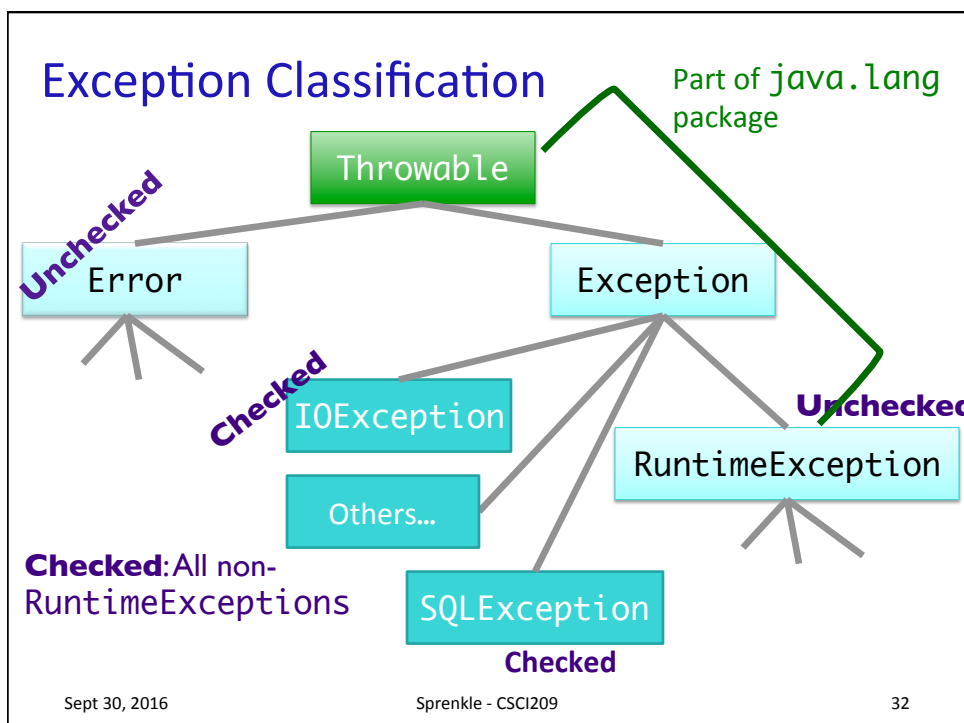
## Exception Classification: **Exception**

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`

Sept 30, 2016

Sprenkle - CSCI209

31





## 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

Sept 30, 2016

Sprenkle - CSCI209

33

## 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`

Sept 30, 2016

Sprenkle - CSCI209

34


## 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)

Sept 30, 2016

Sprenkle - CSCI209

37

## 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

Sept 30, 2016

Sprenkle - CSCI209

38

## 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

Sept 30, 2016

Sprenkle - CSCI209

39

## Throwing An Exception We Created

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

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

Equivalent in Python?

Sept 30, 2016

Sprenkle - CSCI209

40

## 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

Sept 30, 2016

41

## Common Exceptions

Name	Purpose
<code>IllegalArgumentException</code>	When caller passes in inappropriate argument
<code>IllegalStateException</code>	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`

Sept 30, 2016

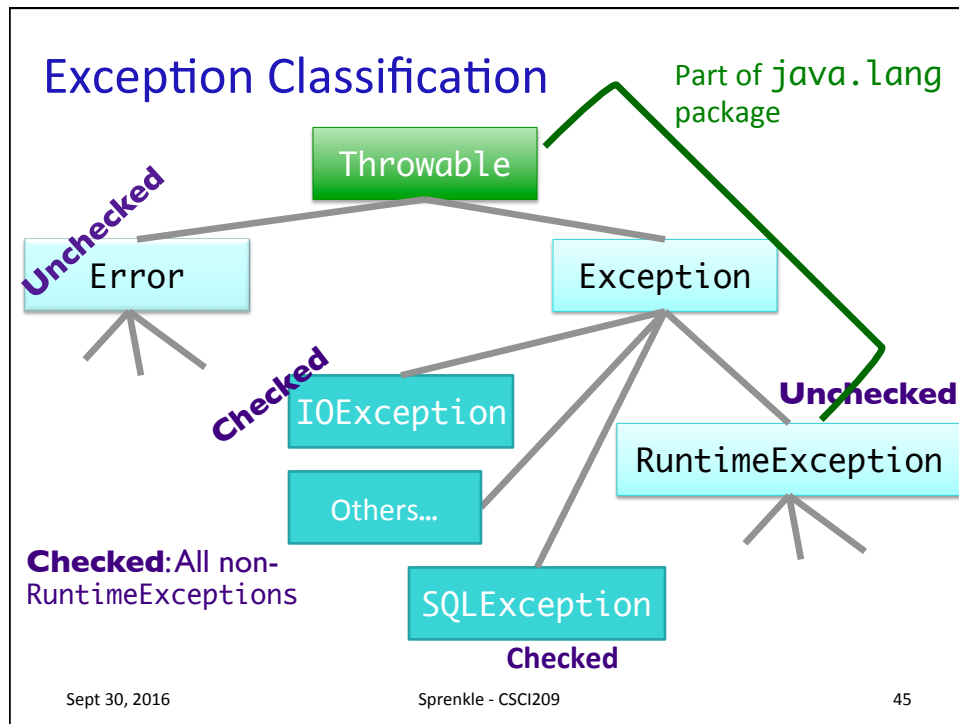
Sprenkle - CSCI209

42

## 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

## CATCHING EXCEPTIONS



## Catching Exceptions

- After we throw an exception, some part of program needs to *catch* it
  - 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:

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

Python equivalent?

Sept 30, 2016

Sprenkle - CSCI209

47

## 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\*

Sept 30, 2016

Sprenkle - CSCI209

48



## 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

Sept 30, 2016

Sprenkle - CSCI209

49

## 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

Sept 30, 2016

Sprenkle - CSCI209

50

## 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

Sept 30, 2016

Sprenkle - CSCI209

51

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

```
try {
    ...
}
catch (Exception e) {
    ...
}
finally { ←
```

**FinallyTest.java**

Sept 30, 2016

Sprenkle - CSCI209

52

## 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

Sept 30, 2016

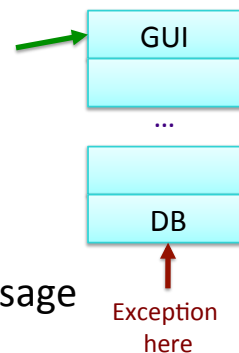
Sprenkle - CSCI209

53

## To Throw or Catch?

- Problem: lower-level exception propagated up to higher-level code
- Example: user enters account information and gets exception message “field exceeds allowed length in database”
  - Lost context
  - Lower-level detail polluting higher-level API

Handled here



**Solution:** higher-levels should catch lower-level exceptions and throw them in terms of higher-level abstraction

Sept 30, 2016

Sprenkle - CSCI209

54

## Exception Translation


```
try {
    // Call lower-level abstraction
}
catch (LowerLevelException ex) {
    // log exception ...
    throw new HigherLevelException(...);
}
```

- Special case: Exception Chaining

- When higher-level exception needs info from lower-level exception

```
try {
    // Call lower-level abstraction
}
catch (LowerLevelException cause) {
    // log exception ...
    throw new HigherLevelException(cause);
}
```

Most standard  
Exceptions have this  
constructor



Sept 29, 2016

Sprenkle - CSCI209

55

## Guidelines for Exception Translation

- Try to ensure that lower-level APIs succeed
  - Ex: verify that your parameters satisfy invariants
- Insulate higher-level from lower-level exceptions
  - Handle in some reasonable way
  - Always log problem so admin can check
- If can't do previous two, then use exception translation

Sept 30, 2016

Sprenkle - CSCI209

56

## 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

Sept 30, 2016

Sprenkle - CSCI209

57

## 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 () {
}
```

Sept 30, 2016

Sprenkle - CSCI209

## Creating Our Own Exception Class

- Try to reuse an existing exception
  - Match in name as well as semantics
- If you cannot find a predefined Java `Exception` class that describes your condition, implement a new `Exception` class!

Sept 30, 2016

Sprenkle - CSCI209

59

## Creating Our Own Exception Class

```
public class FileFormatException extends IOException {
    public FileFormatException() {
        }
        public FileFormatException(String message) {
            super(message);
        }
        // other 2 standard constructors...
}

```

What happens in this constructor implicitly?

Is this a checked or unchecked exception?

- Can now throw exceptions of type **FileFormatException**

Sept 30, 2016

Sprenkle - CSCI209

60

## Guidelines for Creating Your Own Exception Classes

- Include accessor methods to get more information about the cause of the exception
  - “failure-capture information”
- Checked or unchecked exception?
  - Checked: *forces* API user to handle BUT more difficult to use API
    - Has to handle all checked exceptions
  - Use checked exception if exceptional condition cannot be prevented by proper use of API *and* API user can take a useful action afterward

Sept 30, 2016

Sprenkle - CSCI209

61

## Practice: Designing a New Exception Class

- Scenario: When an attempt to make a purchase with a gift card fails because card doesn't have enough money, throw a new exception that you created
- Recall that all Exceptions are Throwable, so they have the methods: `getMessage()`, `printStackTrace()`, `getStackTrace()`

- How would someone else use your class?
- What constructors, additional method(s) may you want to add for your exception class?

Sept 30, 2016

Sprenkle - CSCI209

62

## Benefits of Exceptions?

Sept 30, 2016

Sprenkle - CSCI209

63

## 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

Sept 30, 2016

Sprenkle - CSCI209

64