Objectives

- Exceptions
- Eclipse

1

EXCEPTIONS

2

Errors

- 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)

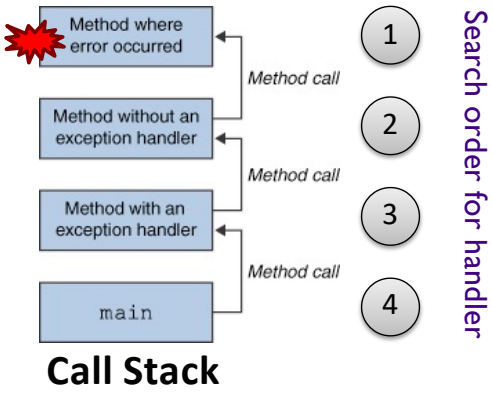
Java Method Behavior

- 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

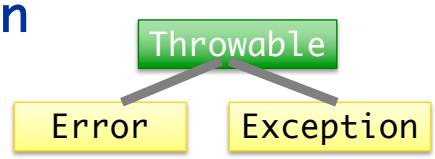
Handling Exceptions

- 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



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

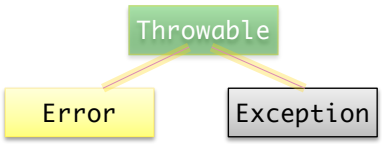


Printing Stack Trace Example

```
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

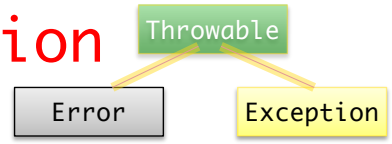


- An internal error
- Strong convention: reserved for JVM
  - JVM-generated when resource exhaustion or an internal problem
    - Example: Out of Memory error

When can that happen in Java?

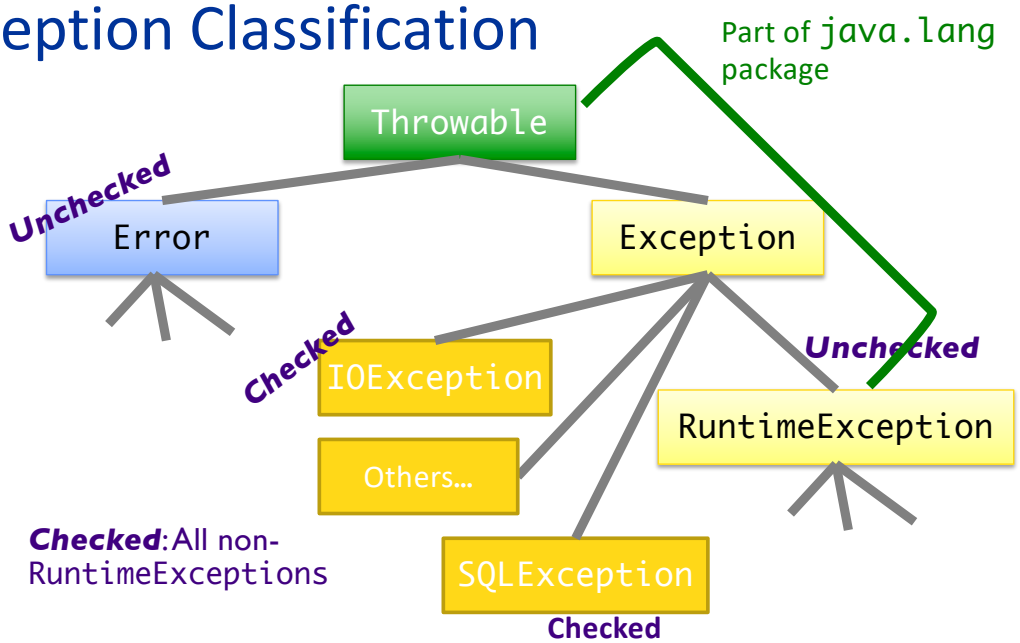
- Program's code should not and can not throw an object of this type
- This is an example of an *Unchecked* exception

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
   - Examples: `IOException`, `SQLException`

Exception Classification



Categories of Exceptions

Unchecked

- Any exception that derives from `Error` or `RuntimeException`
- Programmer does not necessarily create/handle
- Try to prevent them
- 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 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
   - Examples: `ArrayOutOfBoundsException`, `NullPointerException`, `ClassCastException`

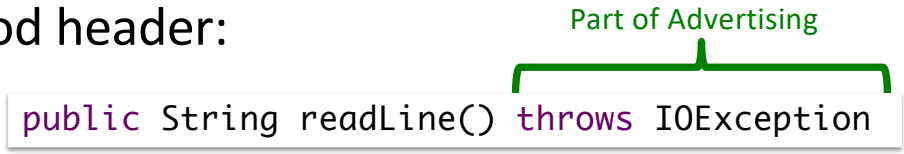
Checked Exceptions

- 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

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:



- Interpreting the header: `readLine` will
  - return a String (if everything went right)
  - throw an `IOException` (if something went wrong)

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

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

Example: Throwing An Exception We Created

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

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

Equivalent in Python?

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

Common Exception Classes

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`

Birthday Error Handling Discussion

- Design decision:
  - Since month and day are not independent, should be set together rather than separately
- Check all the error cases before setting the instance variables
  - Don't want an inconsistent resulting birthday
- `IllegalArgumentException` is appropriate
  - Programming error
  - Should catch those errors before executing program

Goal: Failure Atomicity

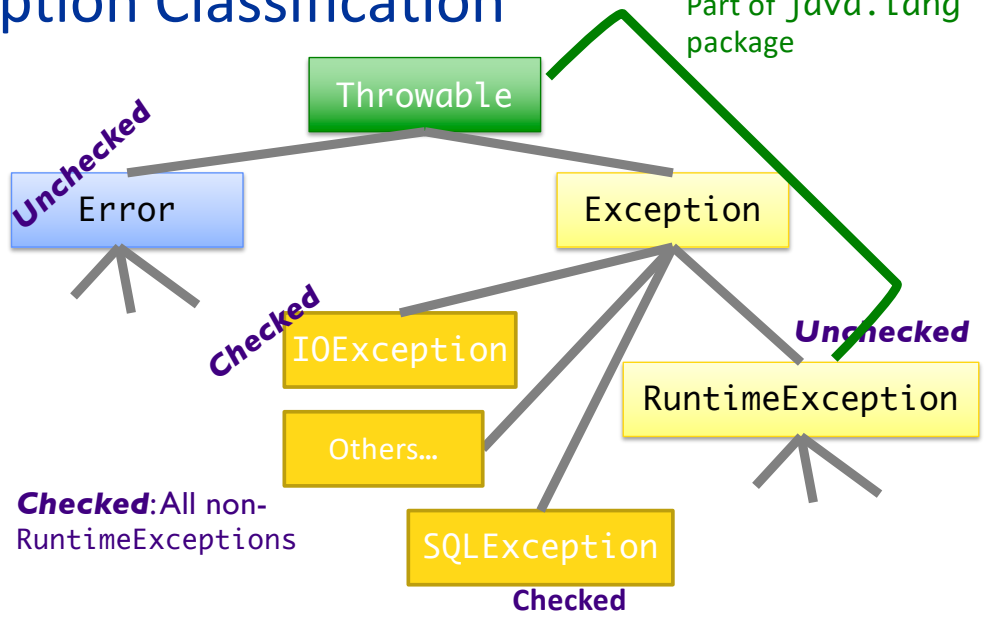
- After an object throws an exception, the object should be in a well-defined, usable state
  - A failed method invocation should leave object in state prior to invocation
- Approaches:
  - Check parameters/state before performing operation(s)
  - Do the failure-prone operations first
  - Use recovery code to "rollback" state
  - Apply to temporary object first, then copy over values

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

CATCHING EXCEPTIONS

Exception Classification



Catching Exceptions

- After we throw an exception, some part of program needs to ***catch*** it
- What does it mean to catch an exception?
  - Program 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:

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
}
```

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

- 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

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:
  1. Neither statement1 nor statement2 throws an exception
  2. statement1 throws an EOFException
  3. statement2 throws an EOFException
  4. statement1 throws an IOException

Practice: try/catch/finally Blocks

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

- Which statements run if:
  1. Neither statement1 nor statement2 throws an exception
     - 1, 2, 5
  2. statement1 throws an EOFException
     - 1,3,4,5
  3. statement2 throws an EOFException
     - 1,2,3,4,5
  4. statement1 throws an IOException
     - 1,5

Catching More Than One Exception Type

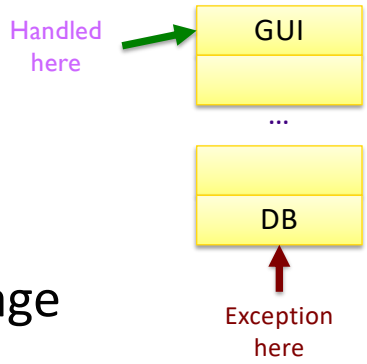
- Can catch multiple exception types in one catch block

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


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

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

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

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

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

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

41

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 and git
- Plugins allow extensibility

42

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

- Assignment 6 – due next Tuesday
  - Eclipse development practice
  - Collections practice
- Change in Tuesday office hours: 2-3 p.m.
  - Updated in Canvas site on Calendar