Exceptions/Debugging/ Annotations/RegEx in Java

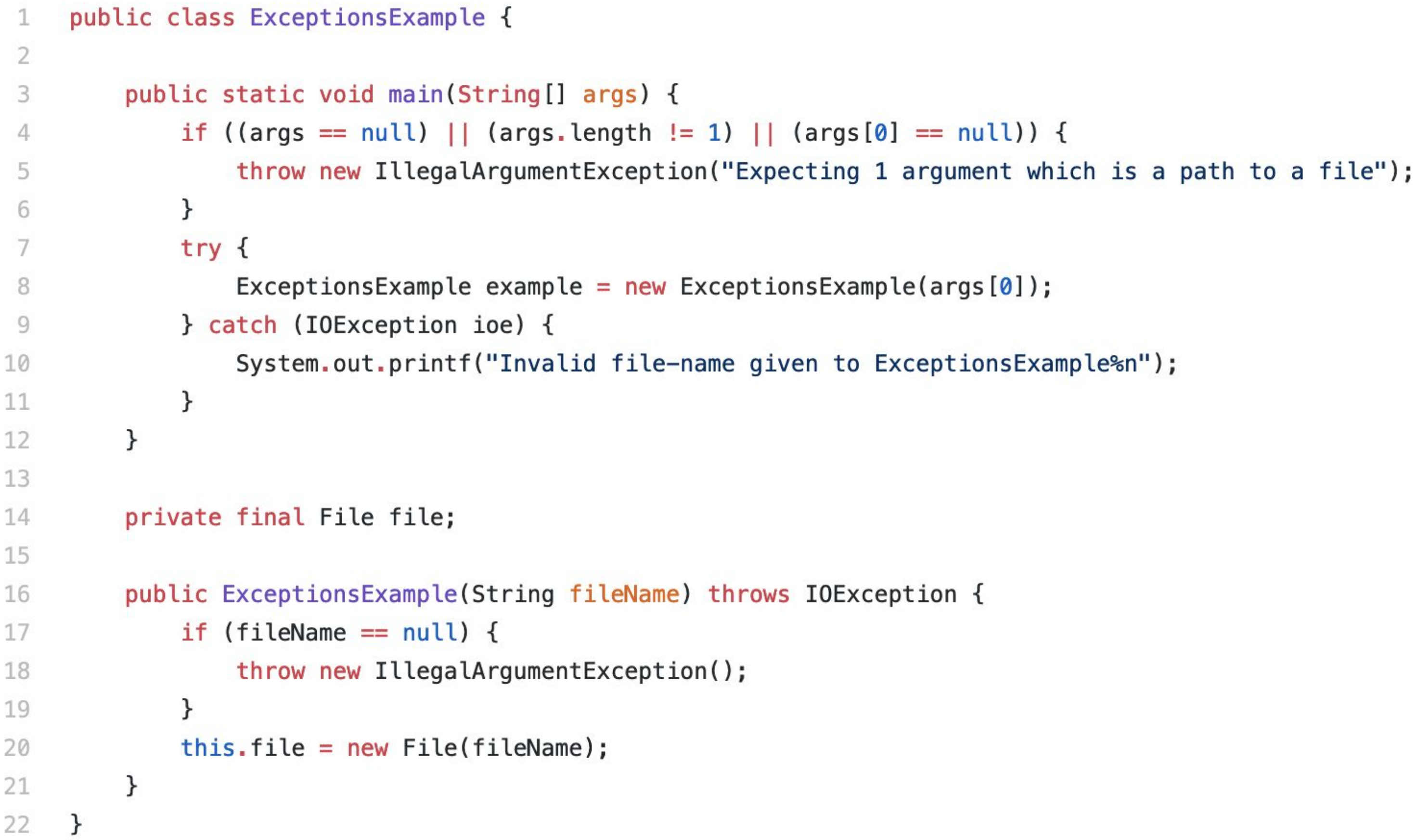
Exceptions Overview

- A prescription for handling errors in the Java language
- Not meant as flow control for normal execution; these are exceptions after all.
- There are two types of exceptions in Java
  - “Checked” exceptions
  - “Unchecked” exceptions

How not to handle exceptions when coding (in any language)

- Returning a “special” value
  - Returning `null` or `-1`
  - Can be confusing - type does not document the exception
  - Can be restrictive - what if the value you need to return is actually `-1`?
- Instead use exceptions (most languages have a notion of exceptions) but only if the case is indeed exceptional (not simply for flow-control)
- If an exception does not make sense, return a type (or wrapper type) detailing the issue

Exception Example

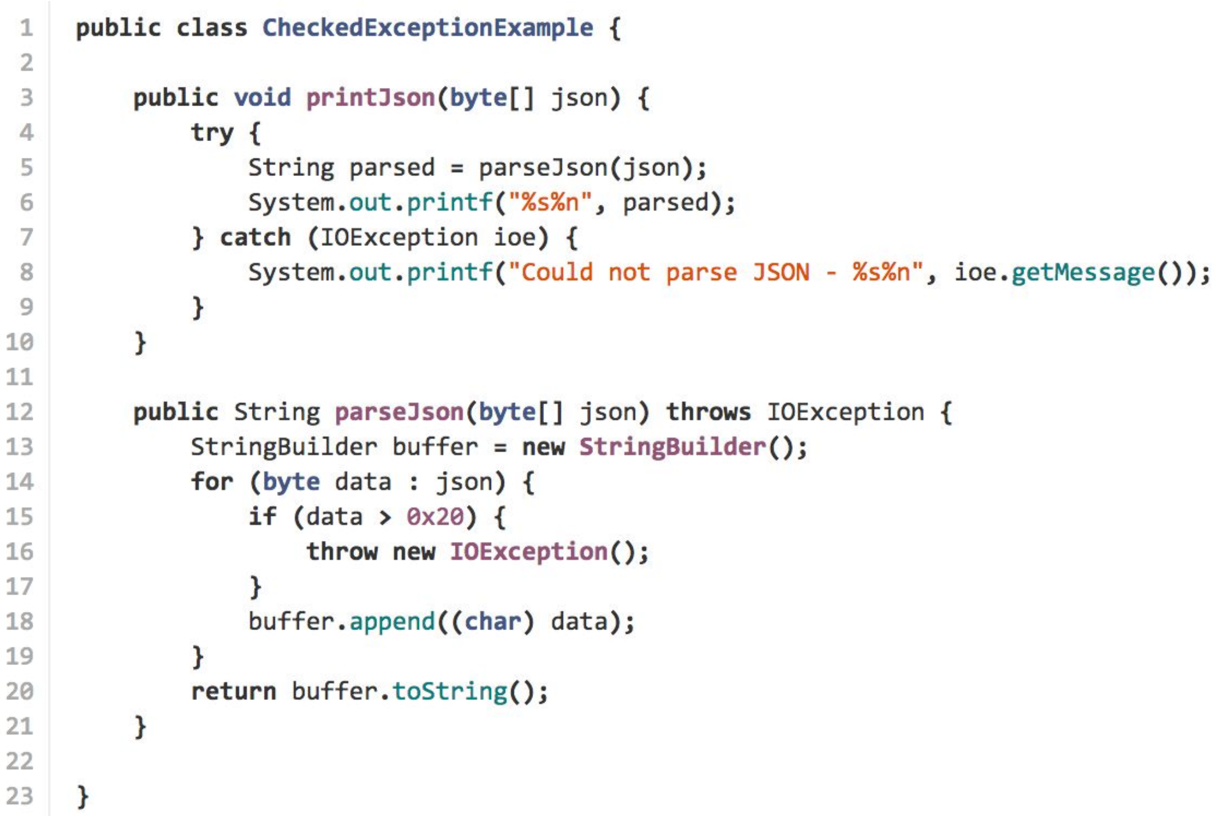


```java
public class ExceptionsExample {

    public static void main(String[] args) {
        if ((args == null) || (args.length != 1) || (args[0] == null)) {
            throw new IllegalArgumentException("Expecting 1 argument which is a path to a file");
        }
        try {
            ExceptionsExample example = new ExceptionsExample(args[0]);
        } catch (IOException ioe) {
            System.out.printf("Invalid file-name given to ExceptionsExample%n");
        }
    }

    private final File file;

    public ExceptionsExample(String fileName) throws IOException {
        if (fileName == null) {
            throw new IllegalArgumentException();
        }
        this.file = new File(fileName);
    }
}
```

Checked Exceptions

- Must be caught or rethrown
- Catching or re-throwing is statically “checked” by the compiler
- In general, only use if the program can recover from this failure. Does this exception have a graceful route to recovery.
  - Otherwise, code becomes bloated with exception handling that isn’t aiding in recovery but simply appeasing the compiler.

Checked Exception Example

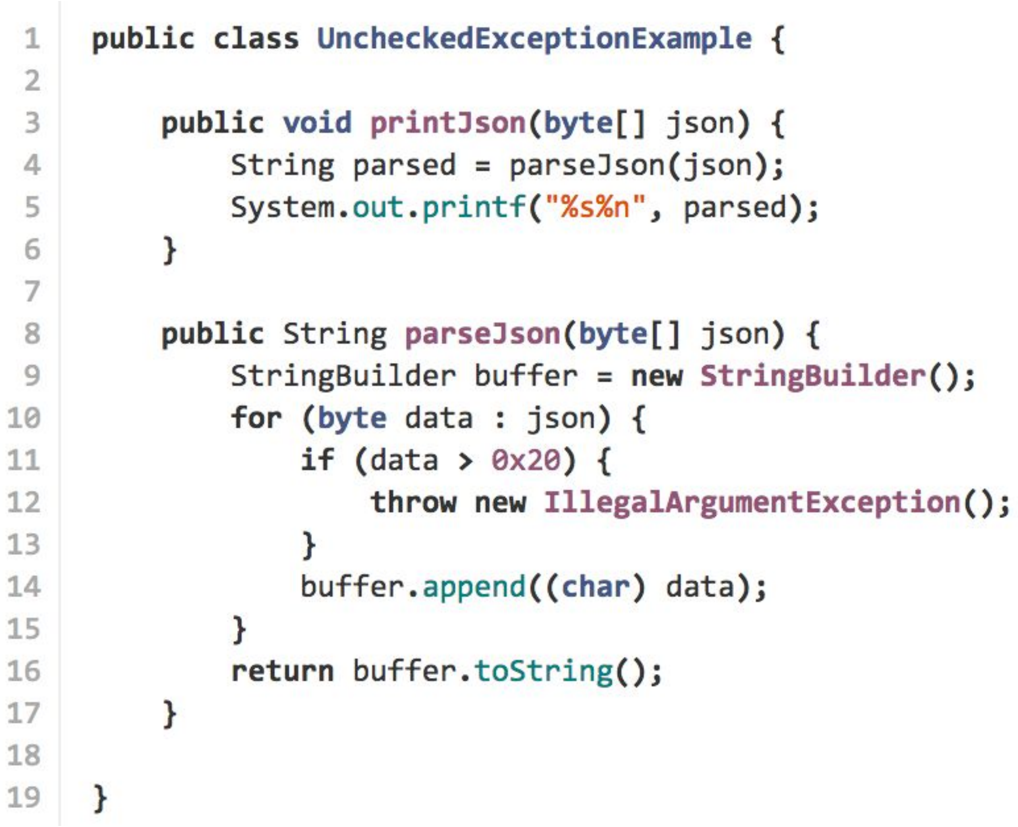


```java
public class CheckedExceptionExample {

    public void printJson(byte[] json) {
        try {
            String parsed = parseJson(json);
            System.out.printf("%s%n", parsed);
        } catch (IOException ioe) {
            System.out.printf("Could not parse JSON - %s%n", ioe.getMessage());
        }
    }

    public String parseJson(byte[] json) throws IOException {
        StringBuilder buffer = new StringBuilder();
        for (byte data : json) {
            if (data > 0x20) {
                throw new IOException();
            }
            buffer.append((char) data);
        }
        return buffer.toString();
    }

}
```

Unchecked Exceptions

- Need not be caught, rethrown or declared
- In general, only use when the program cannot be expected to recover from the failure.
- Callers can choose to catch these exceptions (and recover) they’re just not required by the compiler.

Unchecked Exceptions Example



```java
public class UncheckedExceptionExample {

    public void printJson(byte[] json) {
        String parsed = parseJson(json);
        System.out.printf("%s%n", parsed);
    }

    public String parseJson(byte[] json) {
        StringBuilder buffer = new StringBuilder();
        for (byte data : json) {
            if (data > 0x20) {
                throw new IllegalArgumentException();
            }
            buffer.append((char) data);
        }
        return buffer.toString();
    }

}
```

Throwing Exceptions

- Done by using keyword `throw` (i.e., `throw new IllegalArgumentException()`)
- Must declare checked exceptions as being ‘thrown’
- Do not have to but can declare unchecked exceptions
  - Not often done. If you do this it does not mean the caller must catch the exception
- Cannot override a method and throw a more generic exception
  - But can override and throw a more specific exception
- If overriding a method without a declared checked exception, cannot add one

Catching Exceptions

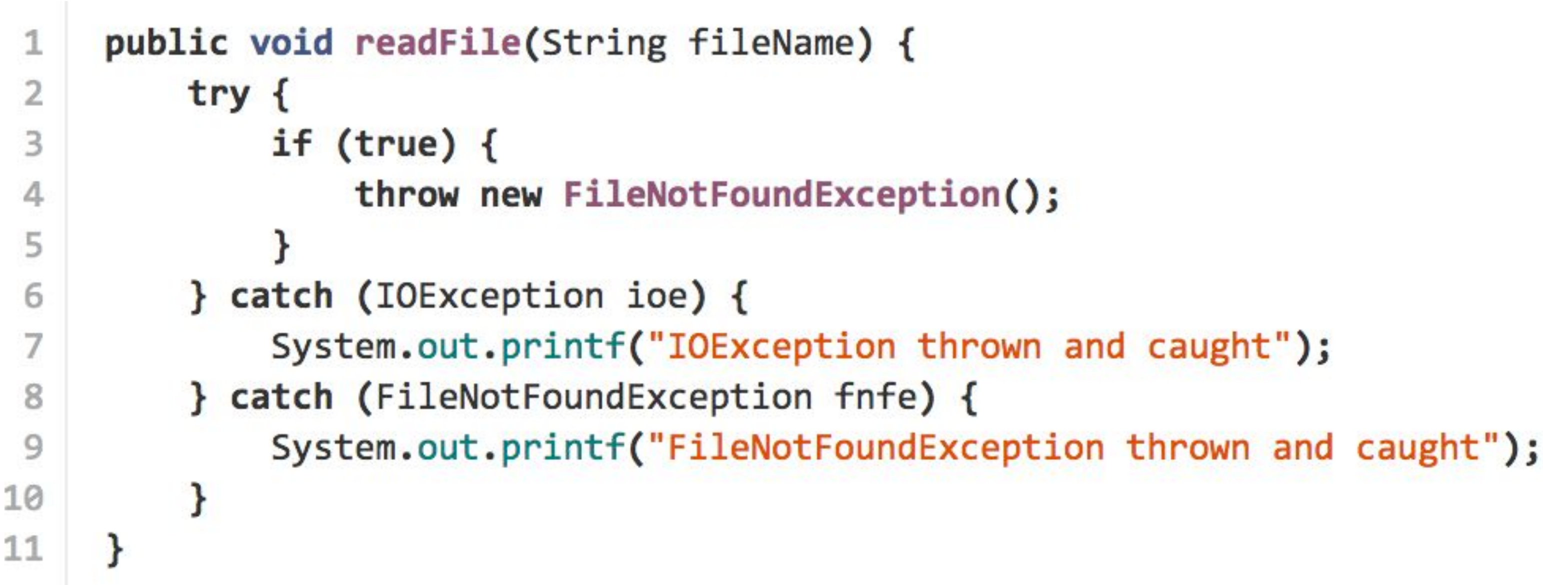
- Use the try/catch mechanism provided by Java
- If you do not catch the type thrown it is handled by the caller
    - If the exception is checked you must either catch it or declare your method as throwing it
    - If the exception is unchecked and not handled up the call-stack, then the JVM stops because of the “uncaught” exception
- If no exception is thrown within the `try` block then the catch block is not executed. Additionally, if the type of the `catch` block does not match the thrown exception then the `catch` block is not executed.
- Multiple catch blocks are handled in order of declaration
- Cannot catch checked exceptions which are not thrown.

Exceptions are Throwables are Objects

- “Exceptions” in Java are actually a subtype of `Throwable`
- There are two main types that extend `Throwable`
  - `Exception`
    - Includes checked and unchecked exceptions.
    - Unchecked exceptions should extend from `RuntimeException`
  - `Error`
    - An exception indicating a system failure (i.e., no more memory).
    - In general, should not extend this class and very rarely throw it yourself

Hierarchy When Catching

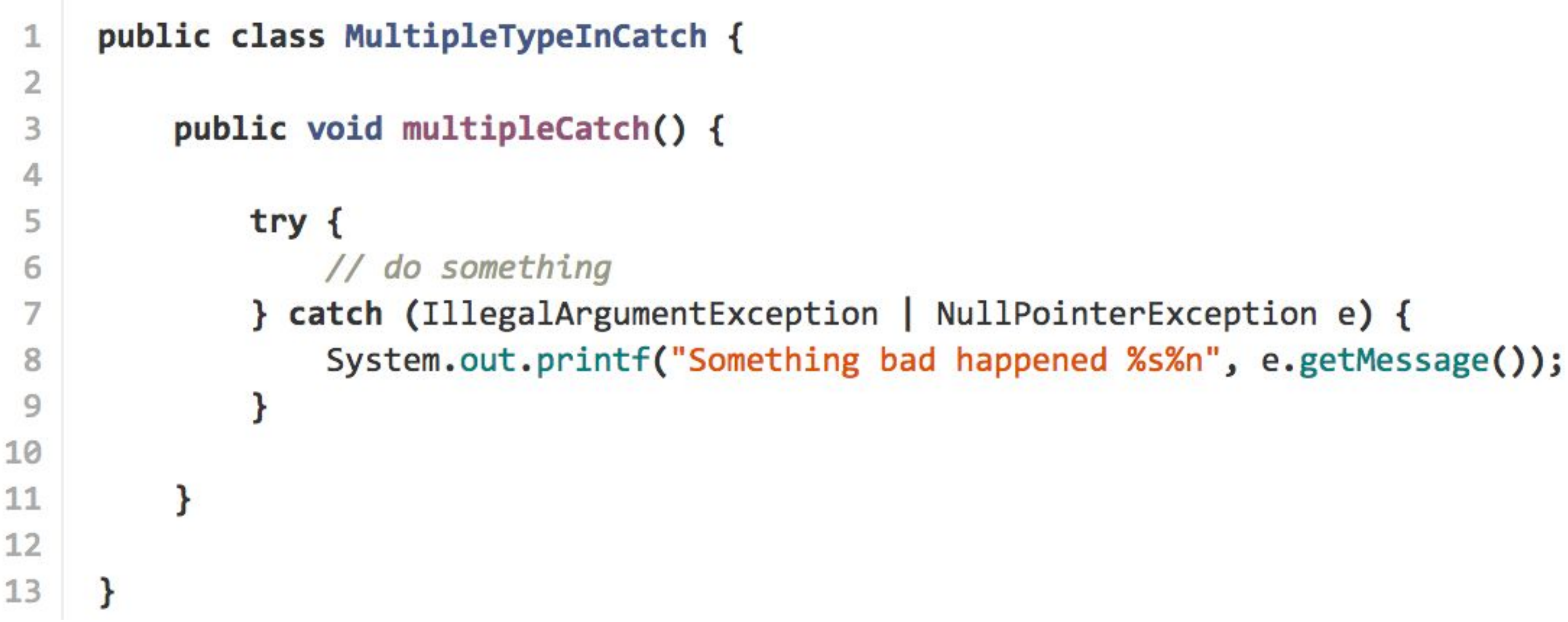
- More generic exception in hierarchy can handle more specific.
  - I.e., parent types can catch all of their children
  - Because of this, order matters. What does the following print?



```
public void readFile(String fileName) {
    try {
        if (true) {
            throw new FileNotFoundException();
        }
    } catch (IOException ioe) {
        System.out.printf("IOException thrown and caught");
    } catch (FileNotFoundException fnfe) {
        System.out.printf("FileNotFoundException thrown and caught");
    }
}
```

Multiple Exception Types in Catch

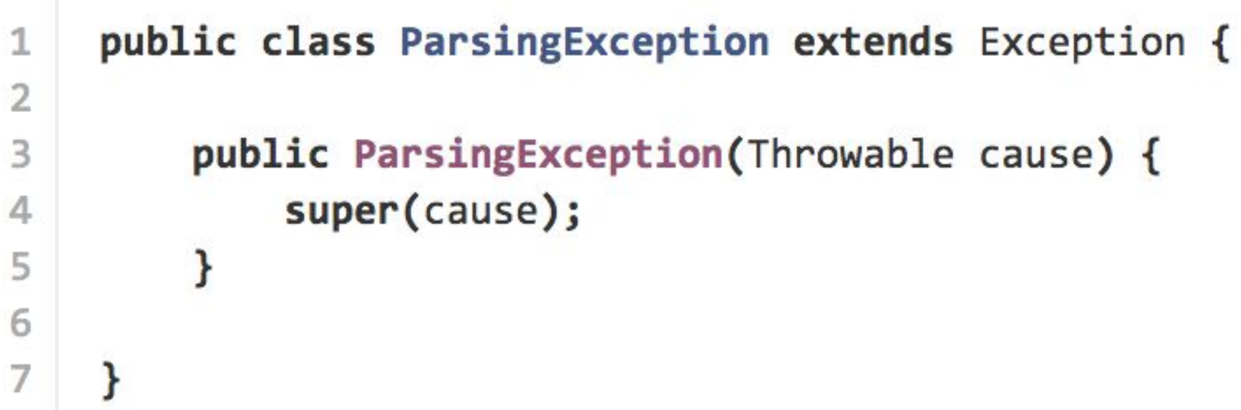
- Java 1.7 added ability to handle disparate exception types in the same catch statement.



```java
public class MultipleTypeInCatch {

    public void multipleCatch() {

        try {
            // do something
        } catch (IllegalArgumentException | NullPointerException e) {
            System.out.printf("Something bad happened %s%n", e.getMessage());
        }

    }

}
```

Custom Exceptions

- You can extend `Throwable` / `Exception` / `RuntimeException` / etc to create your own exception type.

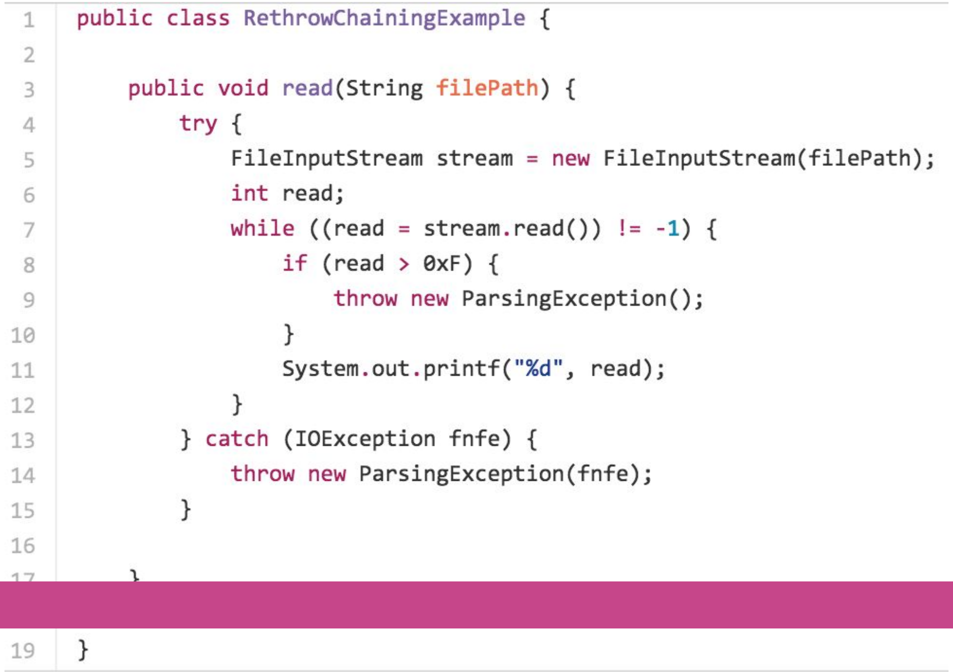


```java
public class ParsingException extends Exception {

    public ParsingException(Throwable cause) {
        super(cause);
    }

}
```

- Not often necessary. Prefer existing exceptions (NullPointerException, IllegalArgumentException, IOException, etc)

Rethrowing / Chaining Pattern

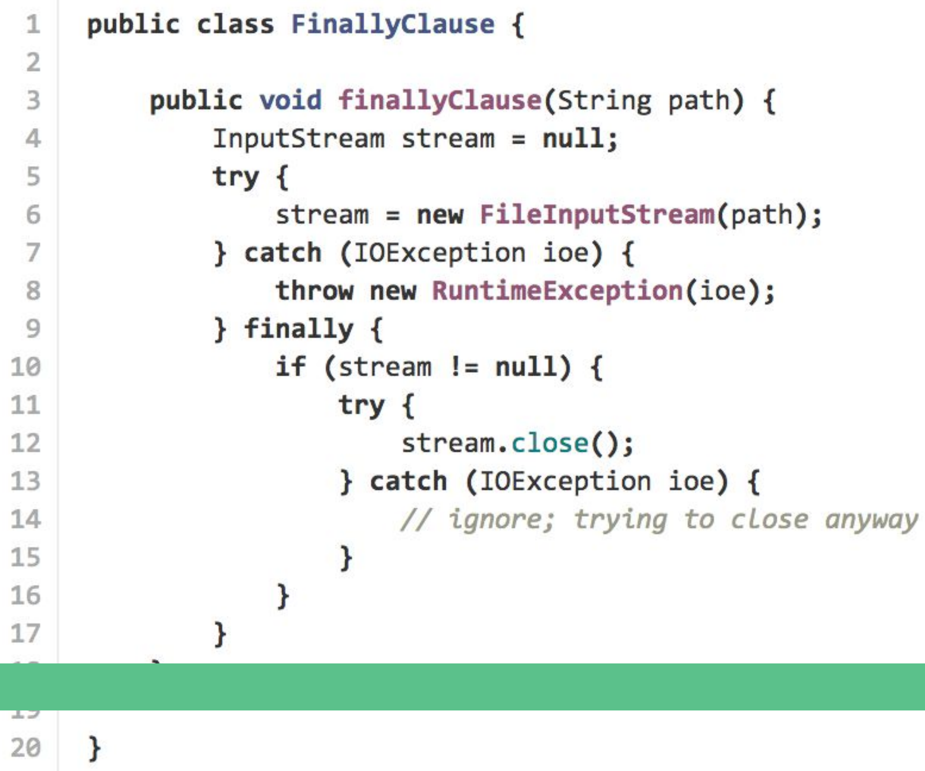
- To simplify your method signature (i.e., API) only throw one checked exception. Catch others and rethrow as one type.
    - Not always necessary, use discretion.



```java
public class RethrowChainingExample {

    public void read(String filePath) {
        try {
            FileInputStream stream = new FileInputStream(filePath);
            int read;
            while ((read = stream.read()) != -1) {
                if (read > 0xF) {
                    throw new ParsingException();
                }
                System.out.printf("%d", read);
            }
        } catch (IOException fnfe) {
            throw new ParsingException(fnfe);
        }

    }
}
```

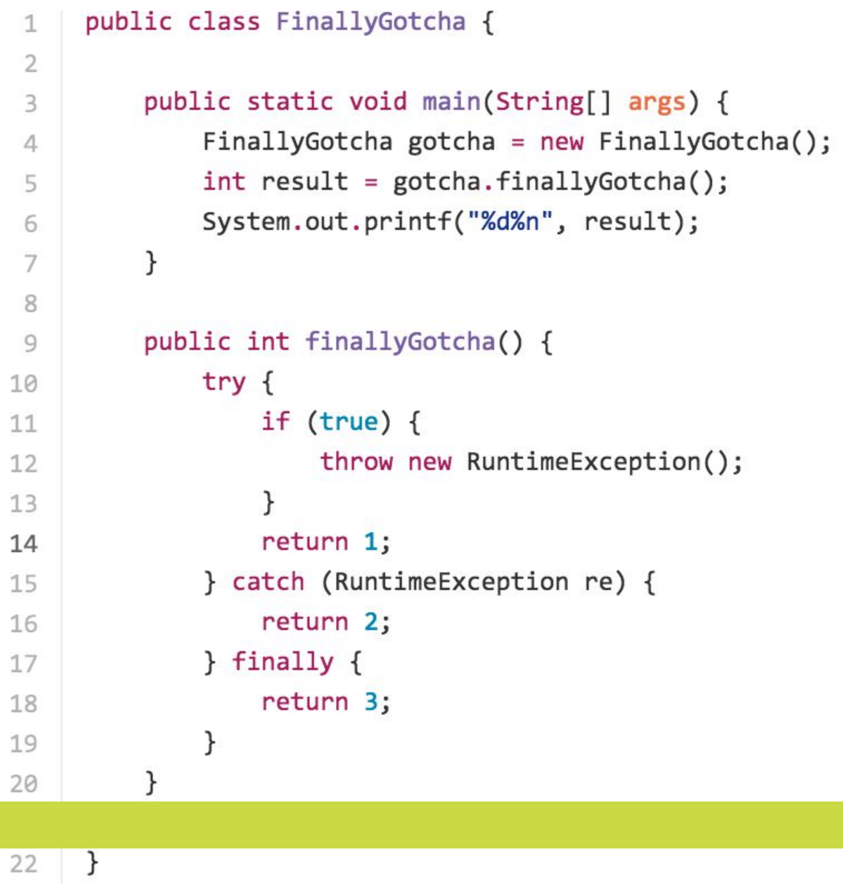
Finally Clause

- Used in conjunction with the try block, to allow code to be executed no matter whether an exception is thrown or not



```java
public class FinallyClause {

    public void finallyClause(String path) {
        InputStream stream = null;
        try {
            stream = new FileInputStream(path);
        } catch (IOException ioe) {
            throw new RuntimeException(ioe);
        } finally {
            if (stream != null) {
                try {
                    stream.close();
                } catch (IOException ioe) {
                    // ignore; trying to close anyway
                }
            }
        }
    }

}
```

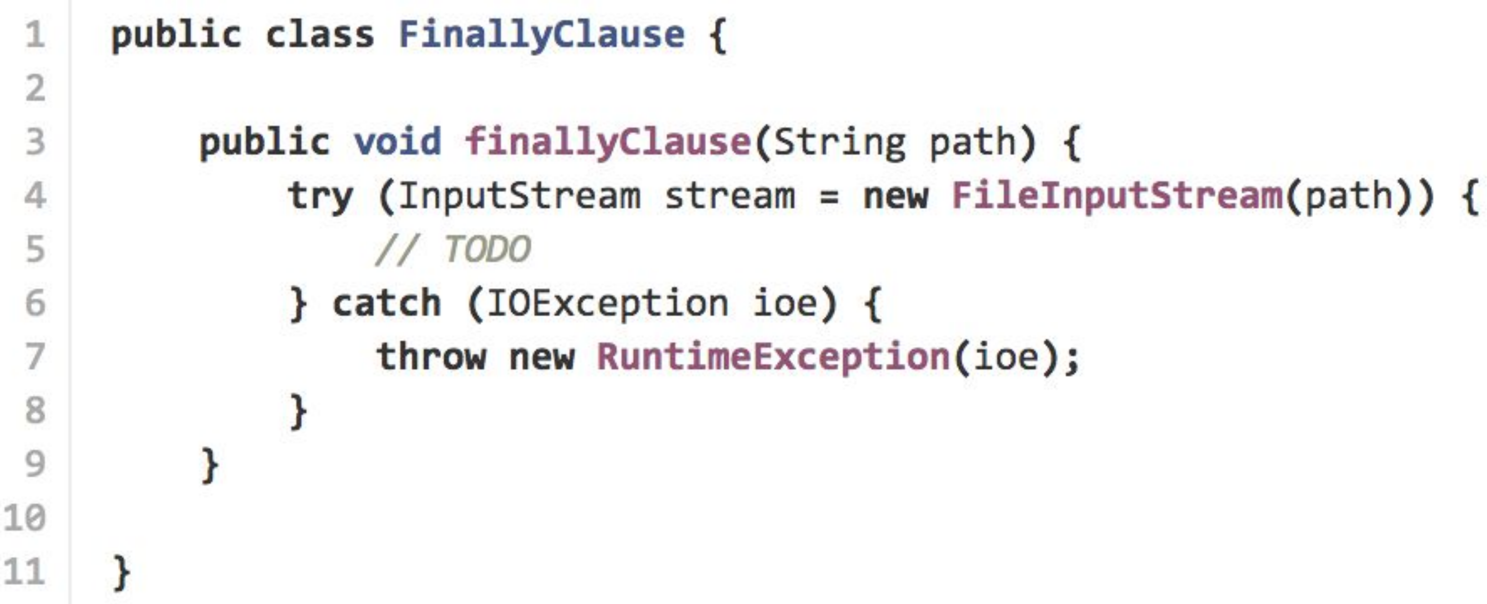
Finally Clause Gotcha



```
public class FinallyGotcha {

    public static void main(String[] args) {
        FinallyGotcha gotcha = new FinallyGotcha();
        int result = gotcha.finallyGotcha();
        System.out.printf("%d%n", result);
    }

    public int finallyGotcha() {
        try {
            if (true) {
                throw new RuntimeException();
            }
            return 1;
        } catch (RuntimeException re) {
            return 2;
        } finally {
            return 3;
        }
    }

}
```

Try With Resources (Java 1.7)

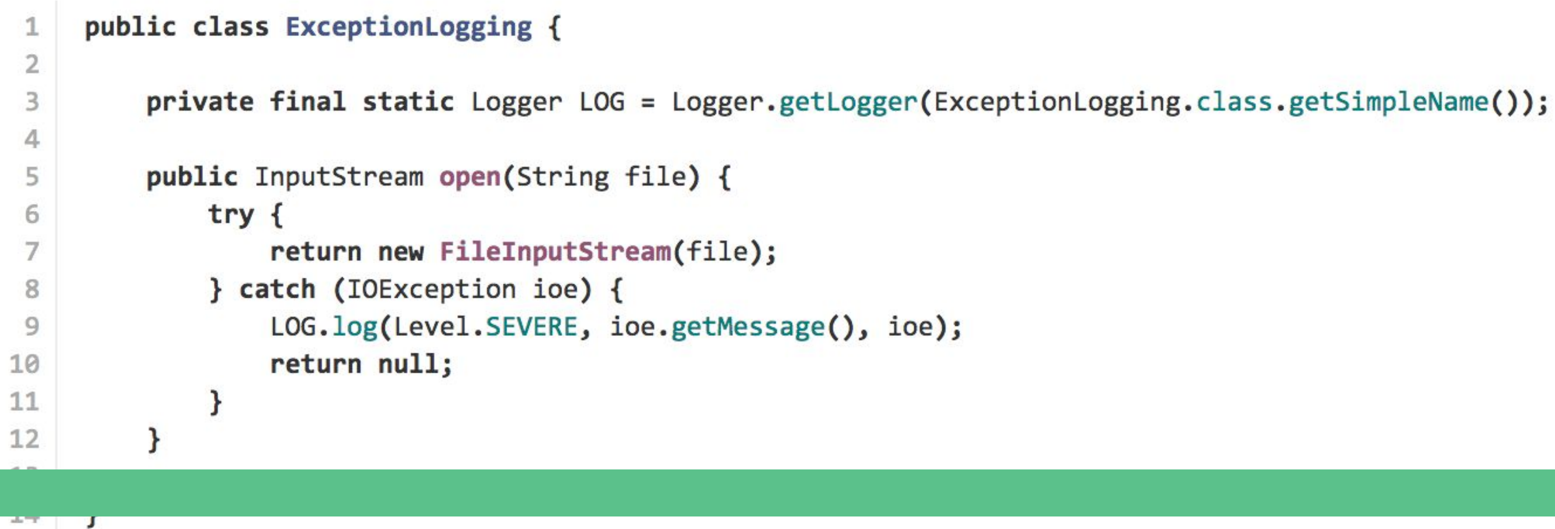
- Introduced in Java 1.7 - handy way to close resources.
  - Avoids the boilerplate code seen two slides previously
  - Can create your own using AutoCloseable interface



```java
public class FinallyClause {

    public void finallyClause(String path) {
        try (InputStream stream = new FileInputStream(path)) {
            // TODO
        } catch (IOException ioe) {
            throw new RuntimeException(ioe);
        }
    }

}
```

Exceptions and Logging

- **Never** print the exception stack-trace using `e.printStackTrace();`
  - As this often simply squashes the exception handling.
- Instead prefer a Logging implementation
  - Java provides a common API - **java.util.logger**
  - **log4j** is a popular library but outdated
  - Latest/best is **slf4j**



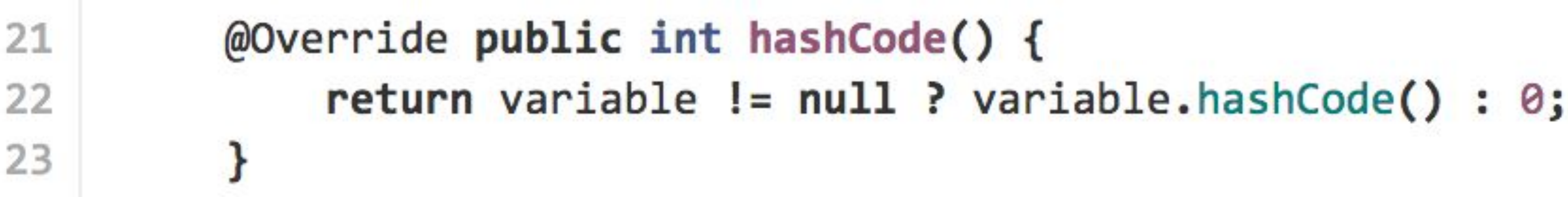
```java
public class ExceptionLogging {

    private final static Logger LOG = Logger.getLogger(ExceptionLogging.class.getSimpleName());

    public InputStream open(String file) {
        try {
            return new FileInputStream(file);
        } catch (IOException ioe) {
            LOG.log(Level.SEVERE, ioe.getMessage(), ioe);
            return null;
        }
    }

}
```

Debugging!

- Do early and often.
- Do not litter log statements into your program, instead use the debugger (jdb or an IDE).
- Always compile your code with **-Xlint:all**
- Do not debug code by adding a **main** method for testing. Test your code using unit-tests instead. Debugging by **main** methods pollutes your code and adds unnecessary dependencies

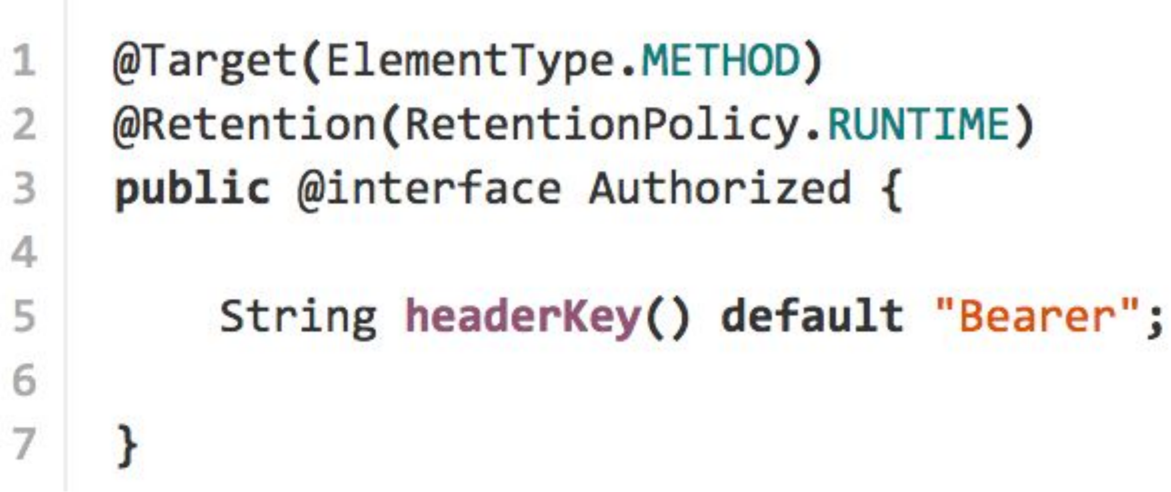
Annotations

- Form of meta programming
    - I.e., annotates your code with metadata which can be used.
- Adds metadata to your code which can be used by programmers for clarity as well as compilers/tooling to do static code analysis
    - We’ve already seen this with **`@Override`**



```
    @Override public int hashCode() {
        return variable != null ? variable.hashCode() : 0;
    }
```

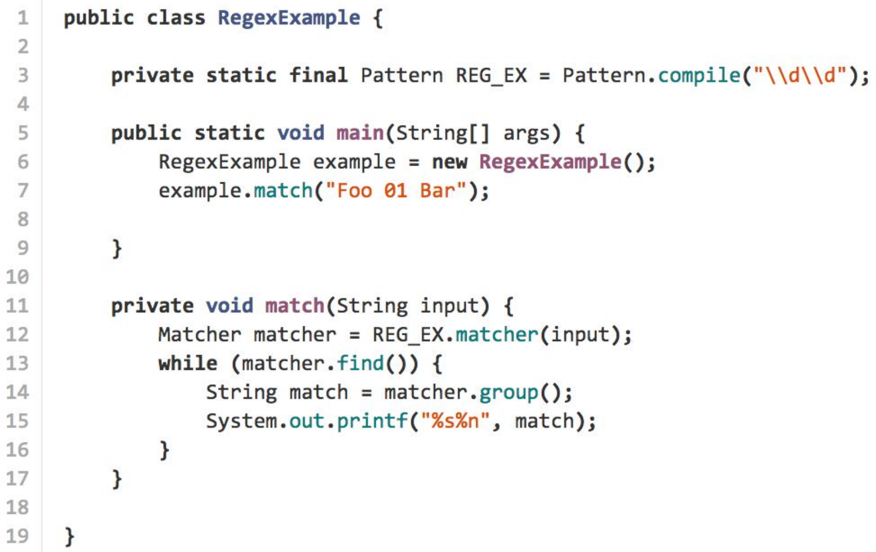
Creating Your Own



```java
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Authorized {

    String headerKey() default "Bearer";

}
```

Regular Expressions

- Encourage you to read the Java tutorial
  - http://docs.oracle.com/javase/tutorial/essential/regex/
- Similar to *Perl* in syntax
- Two main classes
  - `java.util.regex.Pattern`
  - `java.util.regex.Matcher`



```
public class RegexExample {

    private static final Pattern REG_EX = Pattern.compile("\\d\\d");

    public static void main(String[] args) {
        RegexExample example = new RegexExample();
        example.match("Foo 01 Bar");

    }

    private void match(String input) {
        Matcher matcher = REG_EX.matcher(input);
        while (matcher.find()) {
            String match = matcher.group();
            System.out.printf("%s%n", match);
        }
    }

}
```

Read Chapter 8

All sections except 8.9

Homework 6

https://github.com/NYU-CS9053/Spring-2019/homework/week6