

SWEN 601

Software Construction

Exceptions & I/O



Activity: Getting Started

1. Begin by accepting the GitHub Classroom invitation for today's homework.
 - a. *The project may already contain some code!*
2. Create a session package. This is where you will write your solutions to today's activities.
3. Create a homework package. This is where you will implement your solution to the homework.

When you submit your homework, you will include your activities. You may earn up to a 10% bonus on the homework if you have completed all of the activities.

Do not submit code that **does not compile**. Comment it out if necessary.

Next Two Weeks

WEEK 13	SUN	MON	TUES	WEDS	THURS	FRI	SAT
QUIZ			Quiz #18		Quiz #19		
LECTURE			Exceptions & IO		Threads & Concurrency		
HOMEWORK	Hwk 18 Due (11:30PM)		Hwk 19 Assigned		Hwk 20 Assigned	Hwk 19 Due (11:30pm)	

WEEK 14	SUN	MON	TUES	WEDS	THURS	FRI	SAT
QUIZ			Quiz #20				
LECTURE			Thread Cooperation		No Class (Thanksgiving)		
HOMEWORK	Hwk 20 Due (11:30pm)		Hwk 21 Assigned				

Reversi!

- Today we will be implementing the game of Reversi.
 - Also known as Othello.
- This will involve creating classes to represent pieces, a board, and the game itself.
- But first, let's watch a video about how to play.



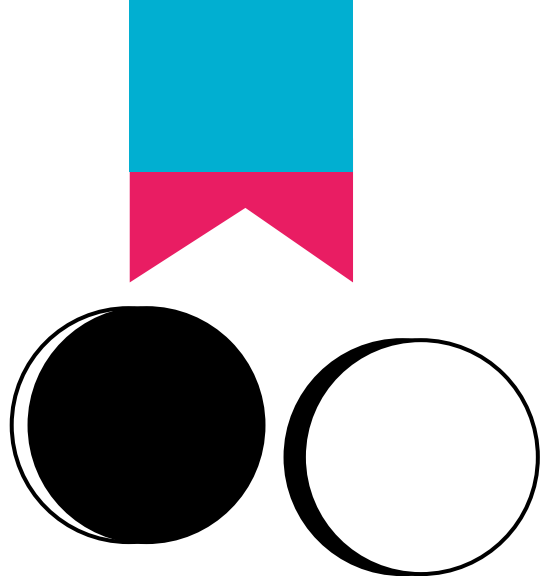
Advanced Enums

- We have already used basic enums a few times.
- There are some more **advanced** features of enums that we have not explored.
- For example, each value defined by the enum may be associated with one or more **other values**.
- You can also write **constructors** and **methods**.

Here is an example of an enum to represent boiling points. But it's not very useful without associating temperatures with each value. Let's change that.

```
public enum BoilingPoint {  
    FAHRENHEIT(212),  
    CELSIUS(100),  
    KELVIN(373);  
  
    private final int degrees;  
  
    BoilingPoint(int degrees) {  
        this.degrees = degrees;  
    }  
  
    public int getDegrees() {  
        return degrees;  
    }  
}
```

Notice that you need a semicolon (;) if any code follows the list of values.



Activity: A Piece enum

Reversi pieces are played with one of two colors face up.

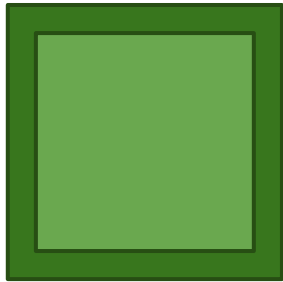
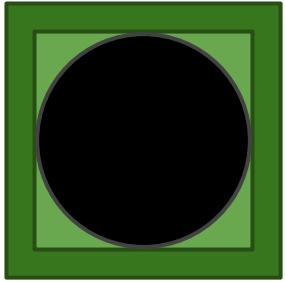
1. Define an enum to represent the two possible piece colors. (BLACK & WHITE).
2. Each color should have a symbol associated with it (e.g. “B” or “W”).
3. Write a method that will return the symbol.
4. Write a method that can be used to “flip” the piece (i.e. when it is called on a WHITE piece, it will return a BLACK piece and vice versa).



Activity: A Square Class

A Reversi board comprises squares. Make a Square class.

1. A Square starts empty but may be occupied by a Piece. Declare fields and write a constructor. How will you represent an “empty” square?
2. Write a method, `public void occupy(Piece piece)`, that will occupy the Square with a Piece.
3. Write a method, `public boolean occupied()`, that returns `true` if the square is occupied, and `false` otherwise.
4. Write a method, `public void flip()`, that will flip the Piece on the Square.
5. Write a `toString()` method that returns a space (“ ”) if the Square is empty, or the symbol for the Piece’s color if it is occupied.



What Could Go Wrong (so far)?

- Think about the Square class that we just finished writing. What are the sort of things that can go wrong?
- What happens if the player tries to occupy a square that is *already occupied*?
 - You can't play a piece on top of another piece...right?
- What if an *empty square* is flipped?
 - Will this cause a problem?
- So what should we do if one of these bad things happens?



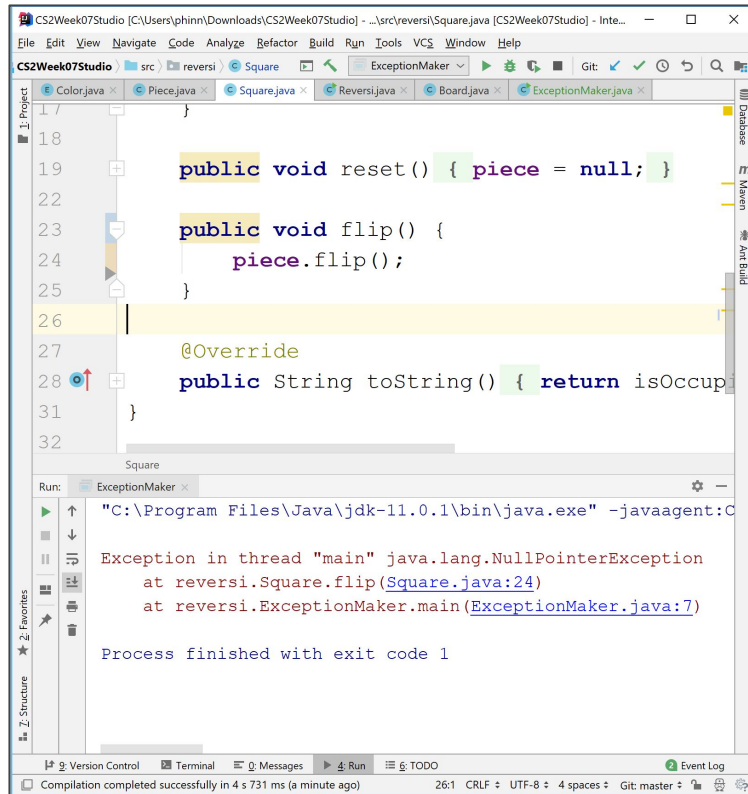
Up to this point in the semester, when an error has occurred in your program, it simply crashed.

Today we will talk about how to handle errors when they occur...

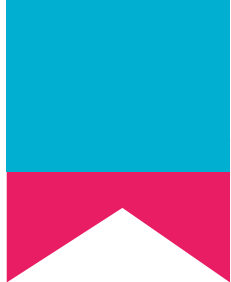
...and how to generate our own errors when necessary.

Debugging Exceptions

- When an exception is not **handled**, it **crashes** the program.
- IntelliJ shows you a **stack trace** detailing the problem.
 - The stack trace shows exactly which **lines of code** were executing when the error occurred.
 - IntelliJ helpfully **hyperlinks** them.
- **Debugging** from a stack trace is a combination of:
 - Understanding what can **cause** a specific exception to occur.
 - **Examining** the line(s) of code involved for possible **culprits**.



What causes a *null pointer exception*?
What might the culprit on line 24 of the Square class be?



Activity: Debugging Exceptions

Given the stack traces below, what you think the root cause of the exception is? Hint: There is **exactly one** possible cause for each.

```
31 public void copy(int[] a, int[] b) {  
32     for(int i=0; i<a.length; i++) {  
33         b[i] = a[i];  
34     }  
35 }  
36  
37 public int sum(int[] values) {  
38     int sum = 0;
```

Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException
at Example.copy([Example.java:33](#))
at Main.main([Main.java:10](#))

The length of b is less than a.

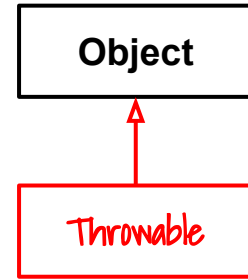
```
25 public boolean equals(Object o) {  
26     if(o instanceof Dog) {  
27         Dog d = (Dog)o;  
28         return d.name.equals(this.name);  
29     } else {  
30         return false;  
31     }  
32 }
```

Exception in thread "main"
java.lang.NullPointerException
at Dog.equals([Dog.java:28](#))
at Doggo.main([Doggo.java:15](#))

The name of Dog d is null.

Throwable

- All **exceptions** in Java are represented as **objects**.
- The base class for all exceptions is [java.lang.Throwable](#).
- The Throwable class defines many useful methods, including:
 - `printStackTrace()` - prints a stack trace with the class and method names, and line numbers that culminated in the error.
 - `getMessage()` - returns a message that provides more detail about the error.
 - `getCause()` - sometimes exceptions are caused by other exceptions. If so, this method is used to get the cause.

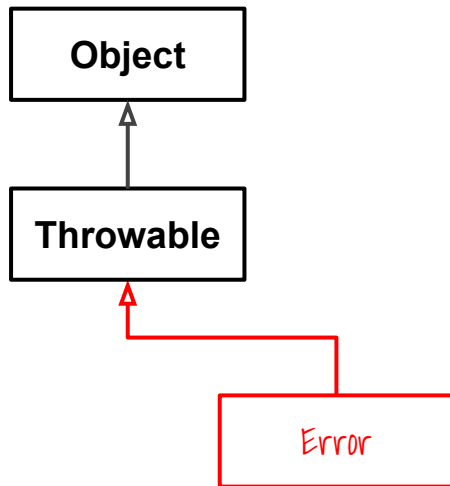


The [java.lang.Throwable](#) class, which extends `Object`, class is at the top of a class hierarchy that defines several different types of exceptions.

Any error that may ultimately cause your program to crash is a descendant of `Throwable`.

Error

- An [Error](#) is a special kind of Throwable that indicates that a **serious** (and perhaps **unrecoverable**) error has occurred.
- Some examples include:
 - [OutOfMemoryError](#) - occurs when the JVM has used up all of the memory allocated to it by the operating system.
 - [IOException](#) - a serious IO error.
 - [StackOverflowError](#) - The JVM has run out of stack space.
 - [AssertionError](#) - assertions are enabled, and an assertion has failed.

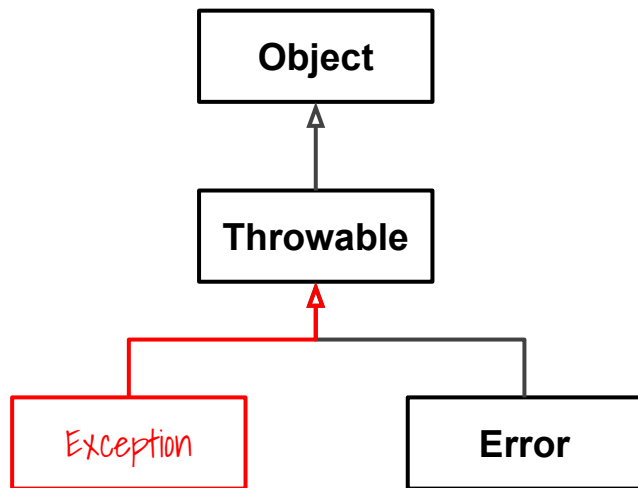


In many cases, an Error indicates a problem that is so severe that the program will not be able to recover and execute normally.

It is for this reason that many programs do not try to handle Errors.

Checked Exceptions

- The [java.lang.Exception](#) class is the base class for all **checked** exceptions.
- A **checked** exception is one that must be **handled explicitly** by the program in some way.
 - This is why they are called “checked” exceptions.
- Unlike Errors, most checked exceptions are **not** so serious that the program can't recover.

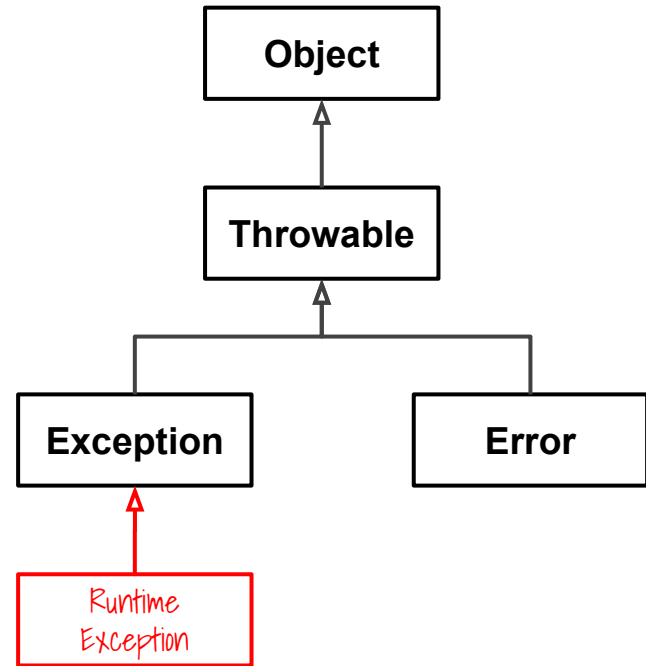


A checked exception is one that must be handled explicitly by the program in some way.

We will talk about specifically how to do this presently.

Unchecked Exceptions

- The `java.lang.RuntimeException` class is the base class for all **unchecked** exceptions.
- An **unchecked** exception does **not** need to be explicitly handled by the program **but** if it is not handled, it will **crash** the program.
- Some examples:
 - [`ArrayIndexOutOfBoundsException`](#) - if you try to access an invalid index in an array.
 - [`NullPointerException`](#) - if you try to use state or behavior with a null reference.
 - [`ClassCastException`](#) - if you try to cast a reference into the wrong class type.



Most of the problems you have seen so far this semester are Errors or Runtime Exceptions. This is why you didn't need to do anything special to handle them (and your program crashed because of it).

Custom Exceptions

- Sometimes a programmer writes code to **detect errors** and **throw** an exception.
 - It is often the case that an exception class already exists that can be reused.
 - In other cases, you want to make your own, **custom exception** class.
- You start by extending Exception for **checked** exceptions or RuntimeException for **unchecked** exceptions.
 - **Checked** exceptions make the possible errors in your code more **explicit**, and are therefore usually the way to go.

```
public class ACheckedException
    extends Exception {

    public ACheckedException(String message) {
        super(message);
    }

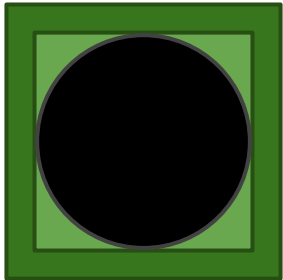
    public ACheckedException(Throwable cause) {
        super(cause);
    }

    public ACheckedException(String message,
        Throwable cause) {
        super(message, cause);
    }
}
```

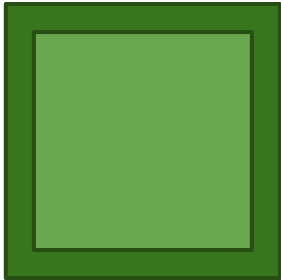
Custom exceptions usually define one or more constructors to accept a *detailed error message*, another Throwable (the *root cause*), or both.



Activity: A Custom Exception



Players of Reversi can try to make several different kinds of bad moves. Make a *custom exception* to use when a bad move is made.

- 
1. Name it `BadMoveException`.
 2. It should be a *checked* exception.
 3. It should define at least one constructor that takes a `String` message.

Throwing Exceptions

- A method that causes an exception is said to **throw** an exception.
- If the exception is **unchecked** (i.e. an **error** or a **runtime exception**), nothing special needs to be done.
- If the exception is **checked**, it must be declared as part of the **method signature** using **throws**.
- The method throws the exception by creating an instance of the exception and using **throw**.

```
1 public void getRoot(int x)
2     throws ACheckedException {
3
4     if(x < 0) {
5         throw new ACheckedException(
6             "x cannot be negative!");
7     }
8     return Math.sqrt(x);
9 }
```

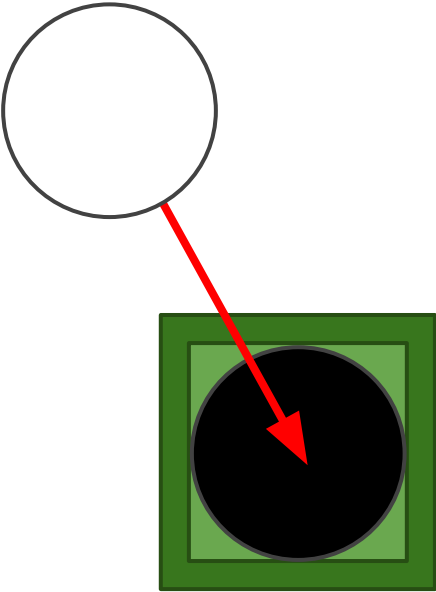
Note that *errors* and *unchecked exceptions* can also be declared using `throws`, but it is not required by the compiler.

If the exception is thrown all the way up to main, the stack trace would look something like this...

```
Exception in thread "main"
  ACheckedException: x cannot be negative!
    at Roots.getRoot(Roots.java:5)
    at Roots.main(Roots.java:12)
```

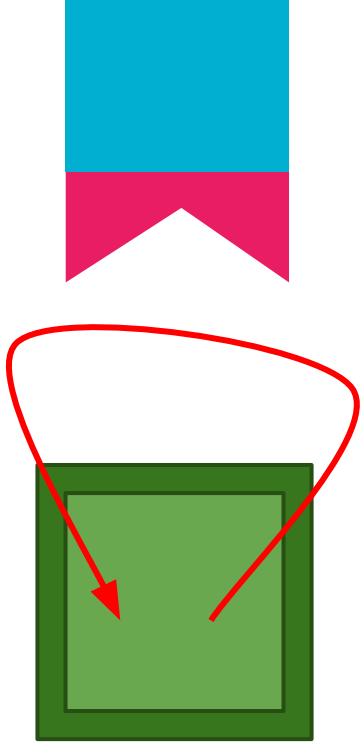


Activity: Bad Move: Occupy



A player may try to occupy a Square that is already occupied. Update the occupy method so that it checks for this and **throws** an exception.

1. Don't forget to declare that the method **throws** your custom exception.
 - a. How will you determine whether or not to **throw** an exception?
2. If appropriate, create a new BadMoveException with a specific error message and **throw** it.



Activity: Bad Move: Flip

A player may try to flip a Square that is empty. Update the `flip` method so that it checks for this and throws an exception.

1. Don't forget to declare that the method **throws** your custom exception.
 - a. How will you determine whether or not to **throw** an exception?
2. If appropriate, create a new `BadMoveException` with a specific error message and **throw** it.

Handling Exceptions: Rethrowing

- If a method declares that it throws a checked exception, a caller must **explicitly handle** the exception in some way.
- One way to do this is to simply **rethrow** the exception.
 - This passes responsibility for handling the exception to the **next** method in the call stack.
- A calling method may **rethrow** an exception by **declaring** that it **throws** an exception of a compatible type.
 - If the exception occurs, it is rethrown **automatically**.

```
1 public void example()  
2     throws ACheckedException {  
3     // ... maybe throws an exception  
4 }  
5  
6 public void someOtherMethod()  
7     throws ACheckedException {  
8  
9     example();  
10  
11 }
```

If one method calls another that declares that it may throw a checked exception, the second method may **rethrow** the exception.

If the first method throws an Error or an unchecked exception, nothing special is needed.

In either case, the exception will be transparently rethrown by the caller. But checked exceptions must be declared using throws.

Handling Exceptions: **try/catch**

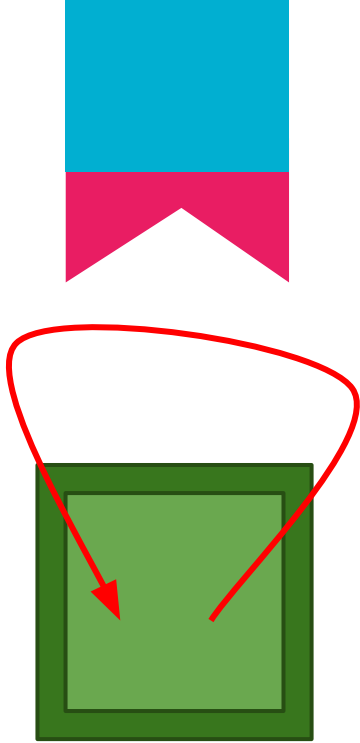
- If an exception or error of any kind makes it all the way up the call stack (i.e. to main), the program will **crash**.
- The only way to prevent a crash is to **catch** the exception using a **try/catch**.
- The code that *may* cause an exception is enclosed in a **try** block.
- A catch indicates the **specific type** of exception it can **handle** and the **code** to execute if it happens.

```
1 public void tryCatchExample() {  
2     try {  
3         anExceptionThrowingMethod(-1);  
4     } catch(ACheckedException e) {  
5         System.out.println("Oops!");  
6     }  
7 }  
8  
9  
10  
11 }
```

A **try** block encloses code that may throw an exception (checked or unchecked).

A **catch** block specifies the exact kind of exception that it can catch. There may be more than one different catch block!

If an exception occurs in the **try** block, the code in the **catch** block that **most closely** matches the exception is executed.



Activity: Making A Square Test

Create a new JUnit 5 test for your Square class.

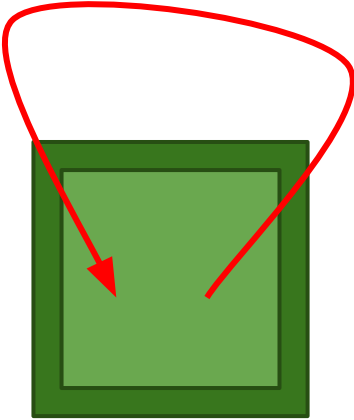
1. Create a testsrc folder and mark it as **Test Sources Root**.
2. Open the Square class and click your cursor into the name of the class.
3. Use **Alt-Enter** and select the **Create Test** option.
4. Make sure to select **JUnit5** as the **Testing library**.
5. Click the **Fix** button to install the library.



Activity: Testing for Exceptions

Add a test to verify that trying to flip an empty Square causes a `BadMoveException`.

1. Write the new test method.
2. Create an empty Square.
3. Call the `flip()` method on the Square - a compiler error should warn you that you are not handling the checked `BadMoveException`.
4. Surround the call in a `try/catch` block.
5. In the catch block assert that the message matches what you were expecting.
6. In the `try` block, use the `fail()` method in the event that an exception does not occur.



The JUnit `fail()` method can be used to force a test to fail. In this case, if an exception is **not** thrown, the test should fail.

Handling Exceptions: **finally**

- So what if there is some code that **must** execute **whether or not** an exception occurs?
 - For example, code to insure that **resources** being used are **cleaned up** even if an error occurs.
- A **finally** block may be used to specify code that will be executed last, **whether** the code in a **try** block causes an exception or **not**.
 - The code in a **finally** block is **always** executed, even if an exception is rethrown.

```
1 public void codeMustExecute()  
2     throws ACheckedException {  
3     try {  
4         throwsACheckedException();  
5     }  
6     finally {  
7         codeThatMustBeExecuted();  
8         regardlessOfExceptionOrNot();  
9     }  
10 }  
11 }
```

A **finally** block is used to specify code that **has to** execute even if an exception occurs.

There can only be one **finally** block, and the **finally** block must always come last (after any catch blocks).

A normal **try** block **must** be followed by a **catch** block or a **finally** block, but can be followed by multiple **catch** blocks and up to one **finally** block.

Tracing Exceptions

<pre>1 public static void getRoot(int x) 2 throws ACheckedException { 3 System.out.print("Root!"); 4 if(x < 0) { 5 throw new ACheckedException(6 "x cannot be negative!"); 7 } 8 double root = Math.sqrt(x); 9 System.out.println(root); 10 }</pre>	<pre>1 public void tryCatchExample() { 2 3 try { 4 System.out.print("Calling"); 5 anExceptionThrowingMethod(); 6 System.out.print(" Done!"); 7 } catch(ACheckedException e) { 8 System.out.print(" Error!"); 9 } 10 11 }</pre>	<pre>1 public void codeMustExecute() 2 throws ACheckedException { 3 try { 4 System.out.print("Calling"); 5 throwsACheckedException(); 6 System.out.print(" Done!"); 7 } finally { 8 codeThatMustBeExecuted(); 9 regardlessOfExceptionOrNot(); 10 System.out.print(" Finally!"); 11 } 12 }</pre>
called with x = 16 <i>Root 4.0</i>	no exception <i>Calling Done!</i>	no exception <i>Calling Done! Finally!</i>
called with x = -1 <i>Root</i> <i><stack trace> x cannot be negative!</i>	exception on line 5 <i>Calling Error!</i>	exception on line 5 <i>Calling Finally!</i>

QUESTIONS? !

Files

- Like nearly everything else in Java, **files** are represented as objects.
- The `java.io.File` class represents a **file** or a **directory** in the file system.
- A file is usually created with a name or a path, e.g. `"C:\bobby.txt"`
- It provides many useful methods, including:
 - `getAbsolutePath()` - gets the full path to the file in the file system.
 - `getParent()` - gets the path to the file's parent directory.
 - `exists()` - indicates whether or not a real file actually exists.
 - `length()` - gets the number of bytes of data in the file.
 - many more.

```
File absolute =  
    new File("/Users/Bobby/lab01.zip");  
  
File relative = new File("myfile.txt");
```

There are a number of different ways to create files in Java, but the most common is to use a String with the file name.

If the String does not include a full path, then Java assumes that the path is **relative** to the program's working directory.

The working directory is the directory from which you ran the program. In IntelliJ, this is the root folder for your project by default.



Activity: Files

Create a new class, Files, that you will use to experiment with files.

1. Write a static method that takes a String filename and:
 - a. **Creates** a new File object with the specified filename.
 - b. Prints the **absolute path** to the file.
 - c. Prints a message indicating whether or not the file **exists**.
 - d. If the file does exist, print the **number of bytes** of data in the file.
2. Write a main method and test your method with several files in your project directory.



IOException

- Many of the methods that interact with the file system may throw a [java.io.IOException](#).
 - If you try to read a file that **does not exist**.
 - If you try to write a file that is **locked** (in use by another program).
 - If you run out of **disk space**.
 - If you don't have **permission** to read from or write to a file.
 - etc.
- IOException is a **checked** exception, and so most code that interacts with files must **explicitly** handle it.

```
public void writeData()  
    throws IOException {  
    // ...  
}  
  
public void readData() {  
    try {  
        // do file stuff ...  
    } catch(IOException ioe) {  
        // handle exception  
    }  
}
```

This means that most of the methods that you implement to write to or read from files will need to throw or catch IOException.

IOException is the base class for many other exceptions, but catching or throwing IOException will handle those as well.

Writers

- There are two basic kinds of data:
 - **characters** - ASCII characters
 - **bytes** - unsigned, 8-bit integers (0-255)
- In Java, the [java.io.Writer](#) class defines the **API** for writing characters to output.
- It defines many methods, including:
 - `write(char c)` - writes a single character.
 - `write(char[] c)` - writes an array of characters.
 - `flush()` - forces buffered data to be written.
 - `close()` - closes the resource.
- [java.io.FileWriter](#) is a child of `Writer` that can write **characters** to a **file**.

```
File f = new File("out.txt");
try {
    FileWriter fw1 = new FileWriter(f);
    FileWriter fw2 =
        new FileWriter("some_file.txt");
    fw2.write('a');
} catch(IOException ioe) {
    System.out.print("couldn't write!");
}
```

A `FileWriter` can be created with a `File` or a `String` with the name of the file.

If a file with the specified name does not already exist, it will be **created**. If it does exist, it will be **overwritten**.

Creating or using a `FileWriter` can potentially cause an `IOException`. The exception will need to be handled.



Activity: Writers

Create a new class, `InputOutput`, and add a `static` method called `writeSquares` that accepts a `List<Square>` and a filename as parameters.

1. Create a `FileWriter` that writes to a file with the specified name.
2. Write each `Square` out to the file as a single character using its `toString()` method.
3. Throw `IOException`.
4. Don't forget to close the file!
5. Add a `main` method. Create a list of 3 squares (one with a white piece, one empty, and one with a black piece). Write the squares out to a file.
6. If an exception occurs, catch it and print the message before exiting with an error code (i.e. `System.exit(1)`).



Readers

- In Java, the [java.io.Reader](#) class defines the API for reading characters from input.
- It defines many methods, including:
 - `read()` - reads and returns a single character as an `int`. Returns `-1` if there is no character to read.
 - `read(char[] buffer)` - reads up to `buffer.length` characters into the provided buffer. Returns the number of characters that were read. If the number is **zero or less**, it means that all of the data has been read.
 - `close()` - closes the resource.
- [java.io.FileReader](#) is a child of `Reader` that can read **characters** from a **file**.

```
File file = new File("in.txt");
FileReader fr1 = new FileReader(file);
FileReader fr2 =
    new FileReader("some_file.txt");

int n = 0;
char[] buffer = new char[1024];
while(true) {
    n = fr2.read(buffer);
    if(n <= 0) {
        break;
    }
}
```

If a `FileReader` can be created with a `File` or a `String` with the name of the file.

If a file with the specified name does not already exist, a `FileNotFoundException` will be thrown. You can't read from a file that isn't there!

Reading is unpredictable and you never know how many characters will be read at once, so reading is usually done in a loop.



Activity: Readers

Add a **static** method to your InputOutput class named `readSquares` that accepts a filename as a parameter and returns a `List<Square>`.

1. Create a `FileReader` that reads from a file with the specified name.
2. For each character in the file, create a new `Square` with the corresponding piece.
3. Throw `IOException`.
4. Don't forget to close the file!
5. In your `main` method, call your `readSquares` method and print the squares out.
6. If an exception occurs, catch it and print the message before exiting with an error code (i.e. `System.exit(1)`).



BufferedReader & PrintWriter

- Up to this point, you have used the Scanner class to read input from the user, but it **hides** exceptions.
- Now that you understand how to handle exceptions, you should use the [java.io.BufferedReader](#) class instead.
 - It can be constructed with another Reader, e.g. a FileReader.
 - The readLine() method reads the next line of text.
 - It will throw a checked IOException if something goes wrong.
- Similarly, the [java.io.PrintWriter](#) class can be created with another Writer, e.g. a FileWriter.
 - It provides many methods to print values to the Writer, similar to System.out.

```
File file = new File("in.txt");
FileReader fr = new FileReader(file);

BufferedReader reader =
    new BufferedReader(fr);

// returns null if at end of file
String line = reader.nextLine();
```

```
File file = new File("out.txt");
FileWriter fw = new FileWriter(file);

PrintWriter writer =
    new PrintWriter(fw);

writer.print(123);
writer.print(false);
writer.println(456.789);
writer.println("This is a string.");
```

Output Streams

- There are two basic kinds of data:
 - **characters** - ASCII characters
 - **bytes** - unsigned, 8-bit integers (0-255)
- In Java, the [java.io.OutputStream](#) class defines the API for writing bytes to output.
- It defines many methods, including:
 - write(byte) - writes a single byte.
 - write(byte[] bytes) - writes an array of bytes.
 - flush() - forces buffered data to be written.
 - close() - closes the resource.
- [java.io.FileOutputStream](#) is a child of OutputStream that can write **bytes** to a **file**.

```
File file = new File("out.bin");  
  
FileOutputStream fos1 =  
    new FileOutputStream(file);  
  
FileOutputStream fos2 =  
    new FileOutputStream("a_file.txt");  
  
byte[] data = new byte[1024];  
fos2.write(data);
```

You will notice that the OutputStream looks and works a lot like a Writer.

The only real difference is that it writes **bytes** of data instead of characters.

The same rules still apply: a file will be created if it doesn't exist, and overwritten if it does. An IOException may be thrown.

Input Streams

- In Java, the [java.io.InputStream](#) class defines the API for reading bytes from input.
- It defines many methods, including:
 - `read()` - reads and returns single byte (0-255). Returns `-1` if there is no byte.
 - `read(byte[] buffer)` - reads up to `buffer.length` bytes into the provided buffer. Returns the number of bytes that were read. If the number is zero or less, it means that all of the data has been read.
 - `close()` - closes the resource.
- [java.io.FileInputStream](#) is a child of `InputStream` that can read **bytes** from a **file**.

```
File file = new File("in.txt");

FileInputStream fis1 =
    new FileInputStream(file);

FileInputStream fis2 =
    new FileInputStream("a_file.txt");

byte[] buffer = new byte[1024];
fis2.read(buffer);
```

You will notice that the `InputStream` looks and works a lot like a `Reader`.

The only real difference is that it reads **bytes** of data instead of characters.

The same rules still apply: if the file does not exist, and `FileNotFoundException` will be thrown.

Cleaning Up

- When interacting with files, it is very important to **clean up** after yourself.
 - You have all seen the *“Cannot delete this file because it is open in another program”* error message on your computer.
 - An open file is **locked**, which means that it cannot be read from or written to by other processes on your computer.
 - For this reason, it is important to make sure that you only open files for as long as necessary, and **close** them when they are no longer needed.
 - This is usually done in a **finally** block to make sure that it happens no matter what.

```
void readData() throws IOException {
    FileInputStream in = null;
    try {
        in = new FileInputStream("in.txt");
        byte[] data = new byte[1024];
        in.read(data);
    } finally {
        if(in != null) {
            try {
                in.close();
            } catch(IOException ioe) {
                // squash
            }
        }
    }
}
```

Properly ensuring that your streams are closed takes quite a bit of code.

Try-With-Resources

- It is considered a **best practice** to make sure that any I/O resource that you are using is **closed** when you are finished using it.
 - This prevents files from being **locked** so that other programs can't use them.
- Historically this was done in a **finally** block.
 - As demonstrated on the previous slide.
- As of Java 7 (2011), Java provides a special kind of **try** called a **try-with-resources**.
 - Any resources opened this way are **automatically closed**.
 - This kind of **try** does **not** need to be followed by a catch or finally.

```
try(FileInputStream in =  
    new FileInputStream("in.txt")) {  
    byte[] data = new byte[1024];  
    in.read(data);  
}  
// input stream is automatically  
// close when the try-with-resources  
// block ends
```

The resource opened using a **try-with-resources** must implement the [java.io.AutoClosable](https://docs.oracle.com/javase/7/docs/api/java/io/AutoClosable.html) interface.

The good news is that just about every I/O class you will use already does.

This is much easier to the old, error-prone way of closing resources.