CS11 – Java

Fall 2014-2015

Lecture 4

Java **`File`** Objects

- Java represents files with **`java.io.File`** class
  - Can represent either absolute or relative paths
- Absolute paths start at the root directory of the filesystem
  - e.g. “C:\Documents and Settings\Donnie Pinkston\Desktop\Foo.java”
    - Note: “\” characters must be escaped in Java strings!
  - e.g. “/home/donnie/Desktop/Foo.java”
- Relative paths start from the current directory
  - Can use “.” to mean the current directory
  - “..” means the parent of the current directory

Java **`File`** Objects (2)

- **`java.io.File`** provides several constants
  - **`File.separator`** is a **`String`** containing the name separator that appears in paths
    - On Windows, set to "\\". On Unix variants, set to "/"
  - Also **`File.separatorChar`**, a **`char`** constant
- Also have constants for path separators
  - **`File.pathSeparator`** is a **`String`** containing the separator between path components
    - On Windows, set to ";". On Unix variants, set to ":"
  - Useful when you must programmatically generate a classpath or other collection of file/directory paths

Creating File Objects

- File constructor is very easy to use
  - **`File(String pathname)`**
    - Specify a relative or absolute path to the file
  - **`File(File parent, String child)`**
    - Assumes that parent is a directory
    - Creates a new File object to reference a file child in the directory parent
  - **`File(String parent, String child)`**
    - Same as previous constructor
- These constructors don’t test whether the files actually exist!

Examining File Objects

- Many helpful methods to examine files, such as:
  - **`boolean exists()`**
    - Is there a file or directory on the filesystem corresponding to the File object?
  - **`boolean isFile()`**
    - Is the File object a “normal” file? (checks that it’s not a directory, and also some system-specific checks)
  - **`boolean isDirectory()`**
    - Is the File object a directory?
  - **`boolean canRead()`**
    - Does the file exist, and can it be read by the application?
  - **`boolean canWrite()`**
    - Does the file exist, and can it be written by the application?
  - **`long length()`**
    - Reports a file’s length.

Manipulating File Objects

- Can perform basic file operations, such as:
  - **`boolean delete()`**
    - Delete a file, or a directory (if it's empty). Returns true if successful, false if not.
  - **`boolean renameTo(File dest)`**
    - Rename a file or directory to a different location
    - May not succeed if moving the file across filesystems, or if destination file already exists, etc.

Navigating the Filesystem

- Can also use **`File`** to navigate the filesystem:
  - **`File[] File.listRoots()`**
    - Static method that returns an array of **`File`** objects specifying the system's root directories
  - **`File[] listFiles()`**
    - Instance method that returns an array of **`File`** objects within a directory
  - (will talk about Java arrays in a future class)
- Can also specify filters to **`listFiles()`** method
  - Implement **`FilenameFilter`** or **`FileFilter`** interface to exclude files based on some criteria

Java Stream IO

- Java provides a stream-based IO mechanism
- **`java.io.InputStream`**, **`java.io.OutputStream`**
  - Abstract base-classes that specify all operations that streams should provide
- Usually open an input- or output-stream via some specific mechanism
  - e.g. open a file and get an input-stream
  - e.g. open a network connection; get an output-stream for sending, an input-stream for receiving

Java Stream IO (2)

- **`InputStream`** methods:
  - **`read()`** for reading one or more bytes
    - A blocking method: will not return until more data is available, or it knows that a read will definitely fail
  - **`available()`** reports how many bytes can be read without blocking
  - **`close()`** closes the input stream
    - Releases any resources associated with the stream
- **`OutputStream`** methods:
  - **`write()`** for writing one or more bytes
  - **`flush()`** to force any internal Java write-buffers to be written out to the OS (may be buffered by OS though)
  - **`close()`** closes the output stream

Java Stream IO (3)

- **`InputStream`** and **`OutputStream`** are byte streams
  - The values actually transferred are bytes
  - Often not suitable for text-based data! (Particularly locale-specific data.)
- **`java.io.Reader`** and **`java.io.Writer`** interfaces work with character data
  - Basically same operations as **`InputStream`** and **`OutputStream`**, but with **`char`** values

Java Stream IO (4)

- Java stream API supports composing streams
- Example: read lines of a text file

```
FileInputStream fis =
    new FileInputStream("foo.txt");
```

  - `FileInputStream` derives from `InputStream`

```
InputStreamReader isr =
    new InputStreamReader(fis);
```

  - Wrap the input-stream with a Reader to read character data

```
BufferedReader br =
    new BufferedReader(isr);
```

  - Add buffering to reader so we can read whole lines of text

- (Java stream IO API is a little annoying…)

Java Stream IO and Exceptions

- **`File`** objects report some failures with a **`boolean`** result…
  - **`boolean delete()`**
  - **`boolean renameTo(File dest)`**

- Most stream IO operations report failures by *throwing exceptions*
  - Usually **`java.io.IOException`**, or some subclass of this exception

Exceptions

- Sometimes code can detect an error, but not necessarily resolve it
  - e.g. a **`FileInputStream`** can detect that the file can’t be opened, but what should it do?
- Several ways to indicate errors to the caller
  - Return a special error value
    - …unless it’s a constructor, which can’t return a value!
  - Throw an exception to signal the error
- An exception aborts the current computation
  - Execution transfers immediately to handler code

Throwing Exceptions

- Throwing exceptions is easy:

```
public double computeValue(double x) {
  if (x < 3.0) {
    throw new IllegalArgumentException(
      "x must be >= 3, got " + x);
  }
  return 0.5 * Math.sqrt(x – 3.0);
}
```

- A new exception object is created and then thrown
- Exception is populated with a stack-trace
  - Specifies where the exception object was created (*not* where it was thrown…)
  - Best to create the exception right when you throw it

Throwing Exceptions (2)

- When exception is thrown, execution *immediately* transfers to handler for that exception

```
public double computeValue(double x) {
  if (x < 3.0) {
    throw new IllegalArgumentException(
      "x must be >= 3, got " + x);
  }
  return 0.5 * Math.sqrt(x - 3.0);
}
```

- For above function, when exception is thrown, no more code inside the function is executed.
- Can specify an error message for exceptions
  - Should indicate what is expected, and what actually happened

Exception Handlers

- To handle an exception, code must catch it

```
void main(String[] args) {
  double x = getDouble();
  try {
    double result = computeValue(x);
    System.out.println("Result is " + result);
  }
  catch (IllegalArgumentException e) {
    System.out.println("Bad input:  " + e.getMessage());
  }
}
```

- Code inside **`try`** block *could* throw an exception…
- **`catch`** block will handle any errors that occur
  - **`IllegalArgumentException`** errors, that is…

Exception Handlers (2)

- If **`computeValue()`** throws, execution transfers *immediately* to catch-block with same exception type

```
void main(String[] args) {
  double x = getDouble();
  try {
    double result = computeValue(x);
    System.out.println("Result is " + result);
  }
  catch (IllegalArgumentException e) {
    System.out.println("Bad input:  " + e.getMessage());
  }
}
```

  - No result would be printed; the error is printed instead.

Exception Handlers (3)

- To catch exceptions from code that could throw, *must* enclose that code in a **`try`** block
  - A **`try`** block can only handle exceptions that occur within that block of code!
- Exception's type governs which **`catch`** block actually handles an exception
  - Specify one or more **`catch`** blocks immediately after the **`try`** block
  - *First* **`catch`** block with matching type will handle the exception
  - After **`catch`** block executes, execution resumes *after* try/catch statement (only one **`catch`** runs)

Java Exceptions

- Java has restrictions on exception handling:
  - Only objects of type **`java.lang.Throwable`** (and subclasses) can be thrown
  - In general, methods *must* declare what kinds of exceptions they throw
    - Another aspect of Java enforcing correctness
    - Forces programs to handle exceptions, or to explicitly declare what might be thrown

Java Exception Hierarchy

`Throwable`

Base-class for all throwable objects in Java

`Error`

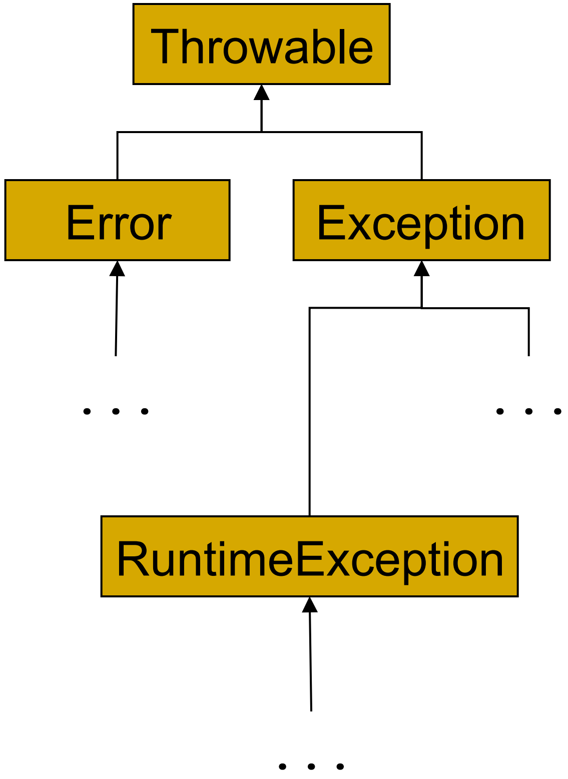
Serious issues in JVM; apps generally won't handle them

`Exception`

Standard run-of-the-mill problems that apps might want to handle

`RuntimeException`

Apps may or may not handle these. Usually indicate programming errors.



Checked Exceptions

- Checked exceptions:
  - Any subclass of **`Exception`** that doesn’t derive from **`RuntimeException`**
- Methods *must* specify checked exceptions they throw:

```
import java.io.IOException;

public String getQuote() throws IOException {
  ...
  if (errorOccurred)
    throw new IOException("An error occurred!");

  return quote;
}
```

  - Java compiler checks method’s code against specifications
  - Can also specify runtime exceptions, but not required

Checked Exceptions (2)

- A method may specify a base-class of what it throws

```
public String getQuote() throws IOException {
   ...
}
```

  - All these exceptions derive from **`IOException`**:
    - **`UnknownHostException`** (couldn’t resolve hostname)
    - **`EOFException`** (unexpected end of file)
    - **`SocketException`** (general socket problem)
  - The above method could also throw these without changing its exception specification
- Code can also catch the base-class type
  - e.g. could **`catch (IOException e)`** and handle the above exceptions

What Exceptions To Handle?

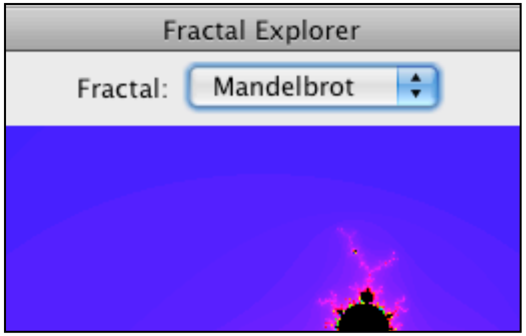
- Java API Documentation indicates which exceptions are thrown
  - API docs also say *when* they are thrown
- IO and networking libraries can throw many exceptions
- Threading libraries also can throw some exceptions
- Always very important to handle exceptions gracefully, to make your applications robust!

This Week’s Assignment

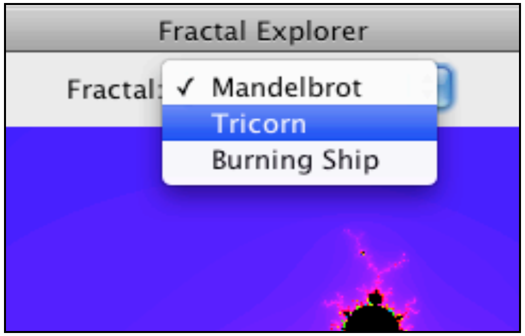
- This week, will add a few new features to your Fractal Explorer
    - The ability to render multiple fractals
        - A dropdown combobox will allow users to select which fractal to render
    - The ability to save the currently displayed fractal image to disk
- Both features shouldn’t be very hard to build
    - Can rely on various Java APIs to make these tasks very simple

Multiple Fractals

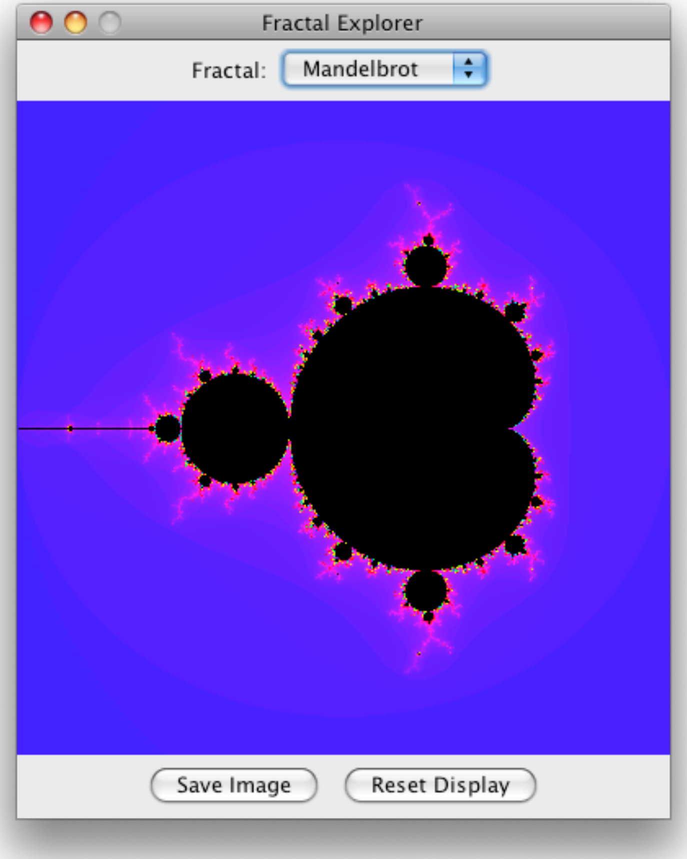
- Most GUI toolkits support dropdown combo-boxes
  - Allows user to choose from a list of options
- Provided by the Swing `JComboBox` class
  - Very easy to set up and use
  - Fires `ActionEvents` when the selection changes



Fractal: Mandelbrot

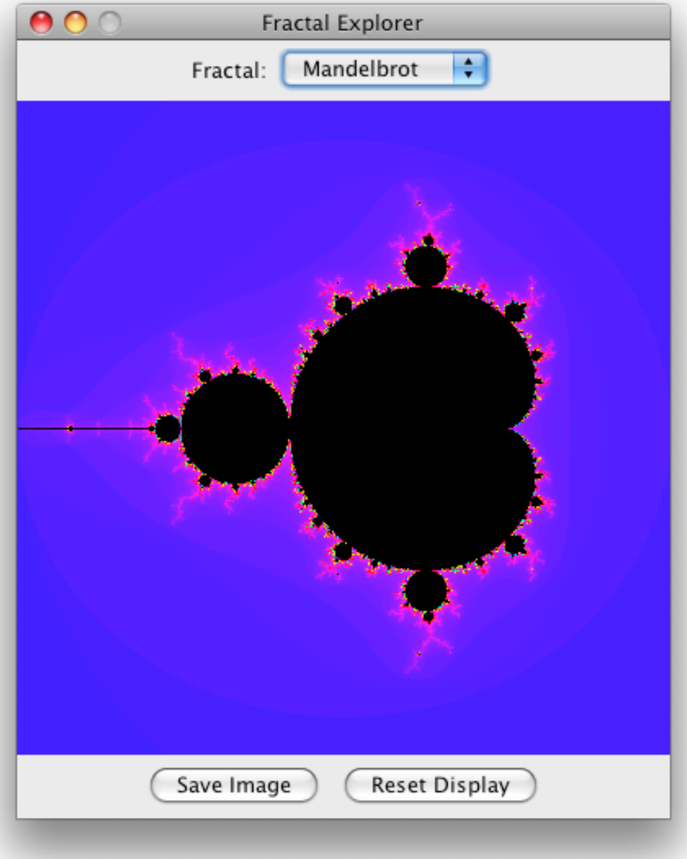


Saving Images



- Also add a button to your user interface, to save the current image
- Swing provides two helpful classes:
  - `JFileChooser` lets you select a file for opening or saving
  - `JOptionPane` can be used to show dialogs when things go wrong ☺

Saving Images (2)



- Now there are multiple sources of action events
- Generally, want to reduce total number of objects your programs create
- Goal:
  - Implement a single action-listener that can handle events from all sources

Action Commands

- Most components that fire **`ActionEvents`** also have an action-command field
  - Use this field to indicate the source's purpose or action

    ```
    JButton saveButton = new JButton("Save Image");
    saveButton.setActionCommand("save");
    ```

  - Other sources get their own action-commands too.
- Action-command value is provided in **`ActionEvent`**
  - **`getActionCommand()`** method on **`ActionEvent`**
  - Now **`ActionListener`** can listen to multiple sources, and perform the proper action based on the action-command

Multiple-Source Action Listeners

- Example action-listener implementation:

```
void actionPerformed(ActionEvent e) {
    String cmd = e.getActionCommand();

    if (e.getSource() == fractalChooser) {
        ...  // Get the fractal the user selected,
        ...  // and display it.
    }
    else if (cmd.equals("reset")) {
        ...  // Reset the fractal image.
    }
    else if (cmd.equals("save")) {
        ...  // Save the current fractal image.
    }
}
```