Five-Minute Review

1. How do we construct a **`GPolygon`** object?
2. How does **`GCompound`** support decomposition for graphical objects?
3. What does *algorithmic complexity* mean?
4. Which operations does a **`HashMap`** support?
5. What is an efficient way to implement it?

Programming – Lecture 12

Files, Exception handling (Chapter 12.4)

- Text files vs. strings
- Opening a file
- Reading characters/lines
- Exception handling
- Selecting files interactively
- **`Scanner`** class
- Writing files

Text Files vs. Strings

- Both contain character data
- Permanent storage vs. temporary existence
- Sequential vs. random access

Opening a File

```
import java.io.*;

BufferedReader rd =
   new BufferedReader(
      new FileReader(filename));
```

Alternatives for finding the file:

1. Specify path (may use **`pwd`** to find out)
2. Put file into default working directory; see *Run* → *Run Configurations* → *Arguments* → *Working directory*
   Note: *workspace_loc* is location of work space; for a project, see *Properties* → *Resource* → *Linked Resources* → *Path Variables*
3. Change working directory

Reading Characters

```
int nLetters = 0;
while (true) {
   int ch = rd.read();
   if (ch == -1) break;
   if (Character.isLetter(ch))
      nLetters++;
}
```

Reading Lines

```
int maxLength = 0;
while (true) {
   String line = rd.readLine();
   if (line == null) break;
   maxLength =
      Math.max(maxLength,
               line.length());
}
```

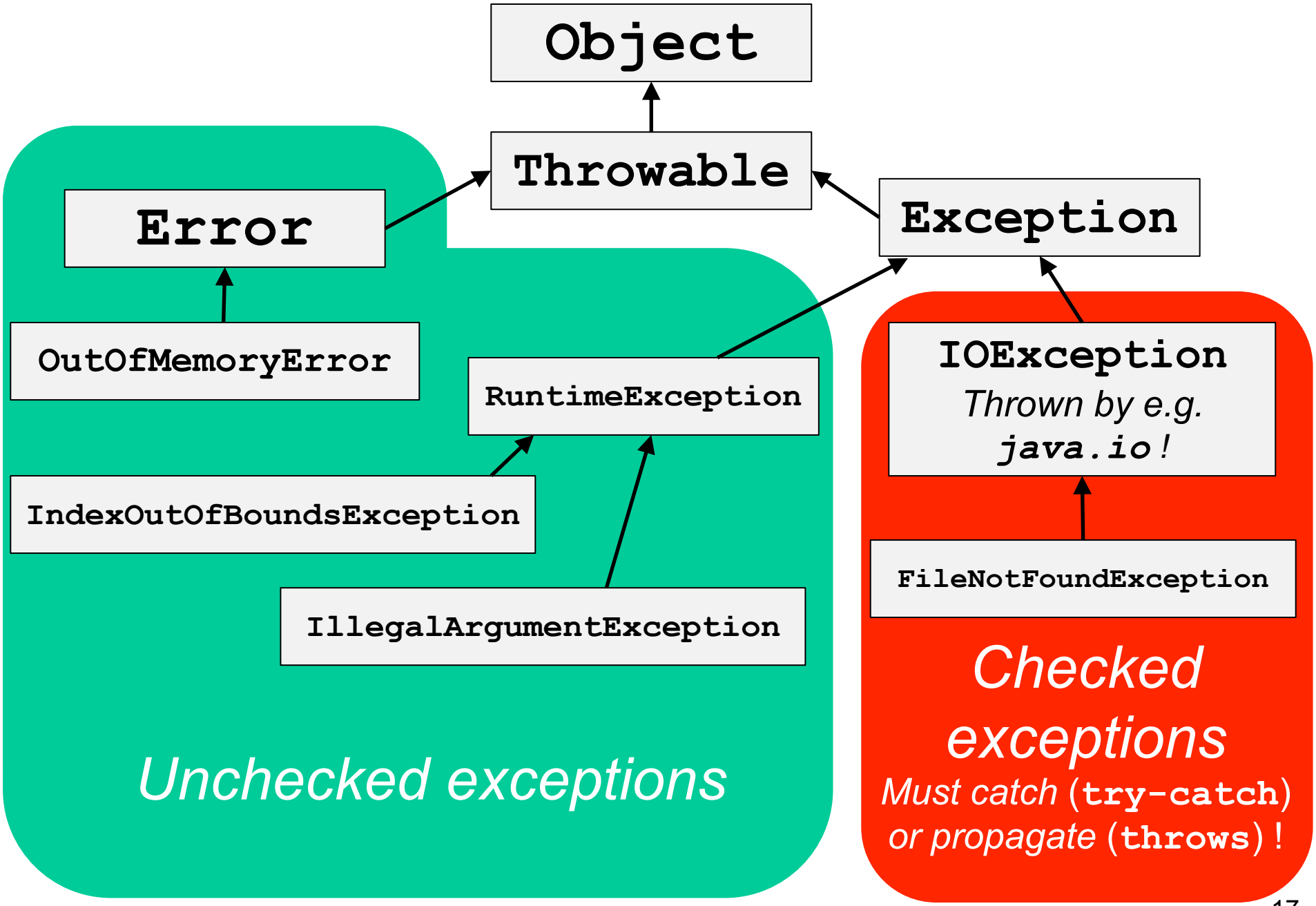
Exception Handling

```
try {
    code in which an exception might occur
} catch (type identifier) {
    code to respond to the exception
}
```

```
private void checkExpression(String prefixExp) {
  try {
    String infixExp = new PNParser(prefixExp).toString();
    println(prefixExp + " => " + infixExp);
  } catch (IllegalArgumentException ex) {
    println("\"" + prefixExp + "\" caused " + ex);
  }
}

public void run() {
  checkExpression("1");
  checkExpression("+");
  checkExpression("1 + 2");
  checkExpression("+ 1 2");
}
```

```
1 => 1
"+" caused java.lang.IllegalArgumentException: Premature
  end of tokens!
"1 + 2" caused java.lang.IllegalArgumentException: Too
  many tokens!
+ 1 2 => 1 + 2
```



See also https://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html

ReverseFile

```
import acm.program.*;
import acm.util.*;
import java.io.*;
import java.util.*;

/** This program prints the lines from a file in reverse order */
public class ReverseFile extends ConsoleProgram {

   public void run() {
      println("This program reverses the lines in a file.");
      BufferedReader rd = openFileReader("Enter input file: ");
      String[] lines = readLineArray(rd);
      for (int i = lines.length - 1; i >= 0; i--) {
         println(lines[i]);
      }
   }

/*
 * Implementation note: The readLineArray method on the next slide
 * uses an ArrayList internally because doing so makes it possible
 * for the list of lines to grow dynamically.  The code converts
 * the ArrayList to an array before returning it to the client.
 */
```

ReverseFile

```
/*
 * Reads all available lines from the specified reader and returns
 * an array containing those lines.  This method closes the reader
 * at the end of the file.
 */
   private String[] readLineArray(BufferedReader rd) {
      ArrayList<String> lineList = new ArrayList<String>();
      try {
         while (true) {
            String line = rd.readLine();
            if (line == null) break;
            lineList.add(line);
         }
         rd.close();
      } catch (IOException ex) {
         throw new ErrorException(ex);
      }
      String[] result = new String[lineList.size()];
      for (int i = 0; i < result.length; i++) {
         result[i] = lineList.get(i);
      }
      return result;
   }
```

ReverseFile

```
/*
 * Requests the name of an input file from the user and then opens
 * that file to obtain a BufferedReader.  If the file does not
 * exist, the user is given a chance to reenter the file name.
 */
   private BufferedReader openFileReader(String prompt) {
      BufferedReader rd = null;
      while (rd == null) {
         try {
            String name = readLine(prompt);
            rd = new BufferedReader(new FileReader(name));
         } catch (IOException ex) {
            println("Can't open that file.");
         }
      }
      return rd;
   }

}
```

Selecting Files Interactively

```
import javax.swing.*;

int result;
JFileChooser chooser;
do {
   chooser = new JFileChooser();
   result = chooser.
      showOpenDialog(this);
} while (result !=
      JFileChooser.APPROVE_OPTION);
```

Selecting Files Interactively

```
try {
   BufferedReader rd = new
      BufferedReader(new FileReader(
        chooser.getSelectedFile()));
} catch (IOException ex) {
   println("Can't open that file.");
}
```

Scanner

`new Scanner(`*reader*`)`**	Creates a new **`Scanner`** object from the reader.
`next()`	Returns the next whitespace-delimited token as a string.
`nextInt()`	Reads the next integer and returns it as an **`int`**.
`nextDouble()`	Reads the next number and returns it as a **`double`**.
`nextBoolean()`	Reads the next Boolean value (**`true`** or **`false`**).
`hasNext()`	Returns **`true`** if the scanner has any more tokens.
`hasNextInt()`	Returns **`true`** if the next token scans as an integer.
`hasNextDouble()`	Returns **`true`** if the next token scans as a number.
`hasNextBoolean()`	Returns **`true`** if the next token is either **`true`** or **`false`**.
`close()`	Closes the scanner and the underlying reader.

Writing Files

```
PrintWriter wr =
   new PrintWriter(
      new FileWriter(filename));

wr.println("My first line");

wr.close();
```

Summary

- There are different types of files, we are particularly interested in text files
- Files are permanently stored, accessed sequentially
- A file must first be opened, and later be closed
- The **`Scanner`** class facilitates to read in data
- Exceptions are caught and handled with **`try`**/**`catch`**
- Thrown exceptions are propagated up the call stack until the next enclosing handler
- Exceptions outside of the **`RuntimeException`** hierarchy, such as **`IOException`**, must be caught by the application