CS415
INTRODUCTION TO COMPUTER SCIENCE
FALL 2017

14 EXCEPTIONS AND FILE I/O
CHAPTER 18

LAST TIME

- Strings
  - Concatenation
  - String methods
- Text I/O
- Scanner

PREVIEW

- Exceptions
  - I/O Exceptions
  - Runtime Exceptions
  - User-defined exceptions
- File class

PROGRAM FAILURES

- Lots of things can go wrong while executing any computer program:
  - A user can provide incorrect input files
  - An interactive user can enter bad data
  - A valid data file can get corrupted
  - A hardware failure can occur
  - Although extremely remote, there is a tiny possibility that your program might have a bug!

ROBUST SOFTWARE

- Robust software is good because of how it handles failure; it requires
  - good failure detection
    - problems should be detected as soon as possible
    - before they infect many parts of the program
  - appropriate failure response
    - ignore error?
    - recover from error?
    - Stop program?

PROGRAM FAILURE

- Program failure is normal behavior
  - all users are fallible
  - hardware is fallible
  - (some) programmers are fallible (not us, of course)
- Robust software is good because of how it handles failure; it requires
  - good failure detection
  - appropriate failure response (recovery, if possible)

FAILURE DETECTION

- Failure detection should occur at the earliest possible point

ROBUST SOFTWARE

- One purpose of classes in a program is to modularize the code.
- Each class can then be considered in isolation; classes can be written independent of any particular application.
- Problem: what if one class detects a problem how should it respond?
  - Since it doesn't know about the application it doesn't know how to respond.
  - It somehow needs to notify a part of the application that can respond.

DETECTION RESPONSIBILITY

- Who should be responsible for failure detection?
  - Hardware
  - Operating system
  - Programming language runtime environment
  - Application program

HARDWARE FAILURE DETECTION

- Some failures must be detected by hardware
  - memory failure -- parity memory
  - I/O error detection (pulling a USB stick out?)
  - power failure detection
  - memory protection violations
    - a user task is limited to a restricted region of memory
  - virtual memory page faults
    - user code references a virtual address, which is mapped dynamically to a physical address, or to a disk block if virtual block not in physical memory.

OS FAILURE DETECTION

- OS provides key information to hardware for
  - memory protection errors
  - virtual memory page faults
- OS I/O support identifies lots of failures
  - non-existent files
  - file type errors
  - file block addressing errors

PROGRAMMING LANGUAGE RUNTIME ENVIRONMENT

- Many programming languages provide extensive runtime environments offering many services to the application
  - Lots of failure detection built in to these environments
- Java's Virtual Machine architecture provides even more opportunity for effective failure detection
  - e.g., Null Pointer Exception, coercion errors

FAILURE RESPONSE RESPONSIBILITY

- All components participate in failure response; those able to detect it may not be best able to respond to it.
  - Hardware detects an I/O write error, software can select an alternate block for the write
  - Hardware detects a page fault, software can read the virtual memory page into physical memory
  - Java Runtime Environment (JRE) detects a null pointer reference; application may be able to recover.

```
Need a framework for error handling
```


ERROR HANDLING FRAMEWORK

- Our motivation is driven by handling errors, but the problem is more generic
  - One person's error is another's convenience
  - We can write code that "treats" something as an error because it might make the code simpler
- We define an Exception as a condition that interrupts the normal execution flow
  - A telephone call can interrupt your execution
  - It may or may not be an error

EXCEPTIONS IN JAVA

- Java provides a clean, elegant mechanism for defining and handling exceptions
- There are a number of pre-defined Exceptions
  - NullPointerException (many of you have seen this one!)
  - ClassNotFoundException
  - ArrayIndexOutOfBoundsException
  - and more
- Users can define their own Exceptions

WHAT IS A JAVA EXCEPTION?

- Each Java Exception is defined by a class
- Each Exception event is represented by an instance of the class
  - Suppose you fail to initialize a reference variable, and you use that variable to call a method

```
Rectangle r;
r.setColor( Color.BLUE );
```

The JRE traps the null reference and creates an instance of the *NullPointerException* class which contains information about the exception.

EXCEPTION HANDLING

- How can we handle exceptions gracefully?
- The exception can occur anywhere during execution.
- We can create an exception object, but what happens to it? Where does control flow resume?
- Java lets application decide

```
void Shape myCopy( Shape s )
{
   Shape copy;
   copy = new Shape();
   copy.x = s.x;
   copy.y = s.y;
   copy.c = s.c.brighten();
   copy.angle = s.angle;
   copy.id = s.id;
   return copy;
}
```

```
Assume the c field of s is a null pointer;
NullPointerException occurs here!
```


TRY-CATCH

- Application can decide what happens to the Exception with the try-catch statement
  - try clause surrounds code that may throw an exception
  - catch clause says what to do if exception occurs
  - In this case, we "fail"; why?
    - copy is incomplete; angle and id were not copied
- This code also handles case where s is null.

```
void Shape myCopy( Shape s )
{
   Shape copy = null;
   try
   {
      copy = new Shape();
      copy.x = s.x;
      copy.y = s.y;
      copy.c = s.c.brighten();
      copy.angle = s.angle;
      copy.id = s.id;
   }
   catch( NullPointerException e)
   {
     // issue error?
     copy = null; // copy fails?
   }
   return copy;
}
```


CREATE A SCANNER FROM A FILE

- new FileReader( ) throws an IOException if the specified file does not exist. When this happens, we open System.in

```
try
{
   return new Scanner( new FileReader( filename ));
}
catch ( IOException ioex )
{
   System.err.println( "***Error--no such file: " + filename );
   System.err.println( "        opening System.in." );
   return new Scanner( System.in );
}
```


EXAMPLE FROM TEXT 18.6

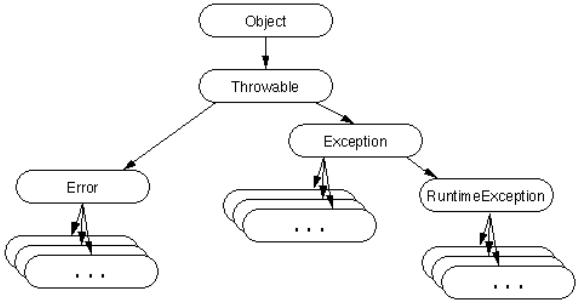
```
public EchoInteger2( )
{
    System.out.print( "Enter an integer, any integer: ");
    Scanner scanner = new Scanner( System.in );
    if ( scanner.hasNextInt( ) )
    {
        int number = scanner.nextInt( );
        System.out.println( "Your number is: " + number );
    }
    else
        System.out.println( "Not a number. Try again." );
}
```

- Instead of an explicit test to see if the next token is an integer, just let *nextInt()* throw an exception

EXAMPLE FROM TEXT 18.7

```
public EchoInteger3( )
{
    Scanner scanner = new Scanner( System.in );
    System.out.print( "Pick an integer, any integer: " );
    try
    {
        int number = scanner.nextInt( );
        System.out.println( "You said " + number );
    }
    catch ( InputMismatchException e )
    {
        System.out.println( "Not an integer, try again." );
    }
}
```


THROWABLE CLASS HIERARCHY



THROWABLE CLASS HIERARCHY

- Throwable is parent to Exception and Error
- Error classes describe serious problems that should not be caught by most applications
- Exception has two kinds of children
  - RuntimeExceptions
  - All other Exceptions, called checked exceptions

ERROR CLASS HIERARCHY

- Error classes describe serious problems that should not be caught by most applications:
  - VirtualMachineError
  - LinkageError
  - AWTError
  - etc.
- Error classes are a set of unchecked exceptions

EXCEPTION HIERARCHY

- Exception has two kinds of children
  - RuntimeExceptions - unchecked exceptions
    - Over 130 "known" RuntimeExceptions
  - All other Exceptions, called checked exceptions
    - Over 300 non-runtime exceptions

RUNTIME EXCEPTIONS

- Exceptions detected during program execution
- These are internal to the application, which is not expected to predict them or recover from them, but can try.
- They are a form of unchecked exception
  - There is no requirement that the application program specify how to handle them

RUNTIME EXCEPTIONS

- Common ones (with "Exception" omitted)
  - IndexOutOfBounds
    - ArrayIndexOutOfBounds (extends IndexOutOfBounds)
  - NoSuchElement
    - InputMisMatch (extends NoSuchElement)
  - NullPointer

CHECKED EXCEPTIONS

- All Exceptions other than RuntimeExceptions
- Exceptions that a well-written application should anticipate and recover from
  - FileNotFoundException, NoSuchMethodException, etc.
- Subject to the "Catch or Specify" requirement

CATCH OR SPECIFY

- For checked exceptions
  - Method must explicitly provide a *try-catch* block around any code that can generate such an exception
  - or specify explicitly that the calling method will handle it
- Otherwise, Java compiler generates an error!

CATCH OR SPECIFY

Catch

```
void pause()
{
  try
  {
    System.in.read();
  }
  catch ( IOException ex )
  {
    System.err.println(
       "read exception caught" );
  }
}
```

Specify

```
void pause() throws IOException
{
    System.in.read();
}
```

```
Specify simplifies this code, but now caller is required to
catch or specify also. It ripples up the call chain; can go all
the way up to main who can throw it up to the JRE.
```


CHECKED EXCEPTIONS

- From Java 5.0 documentation, there are over 300 "known" checked exceptions.
- Some common ones (omitting "Exception"):
  - FileNotFound, EOF, MalformedURL, Socket
    - These are just some of the IOExceptions
  - ClassNotFound, MethodNotFound, FieldNotFound
    - These are among the exceptions when a class is loaded and the loader can't find the class or there is a mismatch between the class definition at compile time and runtime.

USER-DEFINED EXCEPTIONS

- User code can define new exceptions by defining a new Exception class, by extending an existing one, usually Exception (for a checked exception) or RuntimeException (for an unchecked exception).
-

```
class NoSuchColorException extends Exception
{
  NoSuchColorException( String message )
  {
     super( message ); // seldom need anything except this
  }
}
```

- When exception occurs, create an instance of the class and throw it:

```
throw new NoSuchColorException( "Invalid color" );
```

USER-DEFINED EXCEPTIONS

- Suppose we have a method setColor(string) that sets an instance variable curColor to a color specified by a (valid) string.

```
public void setColor( String s );
{
    if ( s.equalsIgnoreCase( "red" ))
        curColor = Color.RED;
    else if ( s.equalsIgnoreCase( "green" ))
        curColor = Color.GREEN;
    ...
    else
        throw new NoSuchColorException( s );
    ....
}
```


USER-DEFINED EXCEPTIONS

- Calling code can now test for the exception

```
try
{
   String c = scanner.Next( );
   setColor( c );
}
catch ( NoSuchColorException nc )
{
   // curColor not set
   // Maybe issue error message and choose some default
}
```


FILE CLASS

- The *File* class encapsulates the notion of elements in a file system: files and directories
  - It does not represent data in the file, but access to it
  - It is a system-independent generalization of the notion of a file (or directory) path name.
    - Unix: /Users/rdb/cs415/slides/Exceptions.key
    - Windows: C:\Users\rdb\cs415\slides\Exceptions.key
    - Abstract: <prefix>, [Users, rdb, cs415, slides, Exceptions.key]
- *File* converts from a system dependent format to the abstract pathname and vice versa

FILE CLASS BASICS

- The *File* class provides lots of functionality, for now you only need a little:
  - Creating a *File* object from a command line argument:

```
// arg can be relative to current directory or absolute
//   i.e. “input.txt” or “/home/cs/rdb/cs415/p9/input.txt”
File myFile = new File( args[ 0 ] );
```

- Once you have a File object you can then create other objects from it for reading or writing to the file

READING FROM A FILE

```
File myFile = new File( args[ 0 ] );
```

- To read **lines or tokens** from the file open it with a Scanner and use the Scanner methods:

```
Scanner scan = new Scanner( myFile );
```

- To read **characters** open with a BufferedReader and use the read method.

```
buf = new BufferedReader( new FileReader( myFile ));
```

```
char ch = (char)buf.read();
```


SCANNING LINES FROM STANDARD INPUT

```
Scanner scan = new Scanner( System.in );

while(  scan.hasNextLine()  )
       System.out.println( scan.nextLine( ) );
```


SCANNING LINES FROM A FILE

```
 public void readFile( String fileName ) throws Exception
// still need to handle exceptions
{
    File file = new File( fileName );


    // next line throws exception
    Scanner scan = new Scanner( file );

    while(  scan.hasNextLine()  )
           System.out.println( scan.nextLine( ) );



}
```


WRITING A LINE TO A FILE

```
 public void writeFile( String fileName ) throws Exception
// still need to handle exceptions
 {
     File file = new File( fileName );
     PrintStream p;


     // Next line throws Exception
     p = new PrintStream( new FileOutputStream( file ) );

     p.println( "Hello File" );
     p.close( );



 }
```

REVIEW

- Exceptions
  - checked exceptions
  - unchecked exceptions (RuntimeExceptions)
  - try-catch
  - throws attribute of a method
  - throw statement
- File class