

# Lecture 30: Exceptions

Last time:

1. Packages

Today

1. Exceptions



# Project #6 Assigned!

- Project due Monday, 4/16 at 11 pm
- Project is **closed**
  - You must complete the project by yourself
  - Assistance can only be provided by teaching assistants (TAs) and instructors
  - You must not look at other students' code
- Start now!
  - Read entire assignment from beginning to end before starting to code
  - Check out assignment now from CVS
  - Follow the instructions *exactly*, as much of grading is automated

# Exceptions

- Programs can generate errors
  - **Arithmetic**  
Divide by zero, overflows, ...
  - **Object / Array**  
Using a null reference, illegal array index, ...
  - **File and I/O**  
Nonexistent file, attempt to read past the end of the file, (we'll see more about file I/O later in course), ...
  - **Application-specific**  
Errors particular to application (e.g., attempt to remove a nonexistent customer from a database)
- In Java: error = **exception**
- What to do when an error occurs?
  - Print an error message and terminate?
  - Return a default value? (Handle error in the code where the problem lies.)
  - Return “error code.”
  - Hard to give a general answer: modern languages cause “exceptions” to be thrown.

# Exception Behavior

- If program generates (“**throws**”) exception then default behavior is:
  - Java clobbers (“**aborts**”) the program
  - **Stack trace** is printed showing where exception was generated

- **Example**

```
public static int findIndex(int[] ints, int s){
    for (int ret = 0; ret <= ints.length; ret++)
        if (ints[ret] == s)
            return ret;

    return -1;
}
```

- **Consider:**

```
int[] ints = {0, 2, 1, 5};
```

- **Evaluating findIndex(ints, 6) results in:**

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4
at exceptionExamples.SearchExample.findIndex(SearchExample.java:13)
at exceptionExamples.SearchExample.main(SearchExample.java:27)
```

# Handling Exceptions

- Aborting program not always a good idea
  - E-mail: can't lose messages
  - E-commerce: must ensure correct handling of private info in case of crash
  - Antilock braking, air-traffic control: must recover and keep working
- **Java includes mechanisms for recovering from exceptions**

# Java Exception Terminology

- When an anomaly is detected during program execution, the JVM **throws** an exception
  - There are built-in exceptions
  - Users can also define their own (more later)
- To avoid crashing, a program can **catch** a thrown exception
- An exception generated by a piece of code can only be caught if the program is alerted. This process is called **trying** the piece of code

# Java Exceptions in Detail

- Exceptions are (special) **objects** in Java
  - They are created from classes
  - The classes are derived (“**inherit**”) from a special class, **Throwable**
  - We will learn more about inheritance, etc., later
- Every exception object / class has:
  - `Exception(String message)`  
Constructor taking an explanation as an argument
  - `String getMessage()`  
Method returning the embedded message of the exception
  - `void printStackTrace()`  
Method printing the call stack when the exception was thrown

# Exceptions, Classes and Types



- Exceptions are objects
- In Java, objects are created from classes
- Classes are also types
- Are there exception classes in Java? Yes! Some examples from the Java class library (mostly java.lang):
  - `ArithmeticException`: Used e.g. for divide by zero
  - `NullPointerException`: attempt to access an object with a null reference
  - `IndexOutOfBoundsException`: array or string index out of range
  - `ArrayStoreException`: attempting to store wrong type of object in array
  - `EmptyStackException`: attempt to pop an empty Stack (java.util)
  - `IOException`: attempt to perform an illegal input/output operation (java.io)
  - `NumberFormatException`: attempt to convert an invalid string into a number (e.g., when calling `Integer.parseInt( )`)
  - `RuntimeException`: general run-time error (subsumes above)
  - `Exception`: The most generic type of exception

# Handling Exceptions in Java

- Java uses **try-catch** blocks for exception handling. Syntax:

```
try {  
    ...  
}  
catch (<exntype1> e2) {  
    ...  
}  
catch(<exntype2> e2) {  
    ...  
}  
finally { // finally is optional  
    ...  
}
```

- **Meaning**

- Execute code after try ("**try block**")
- If an exception is generated, sequentially try each `catch` statement to see if exception matches given exception type (`exntype`)
- If a match is found:
  - Execute corresponding block
  - Resume after last `catch` / `finally`
- `finally` block executed after previous processing (even if exception not thrown, or thrown but not caught)

# Example

- Recall previous findIndex example
- What is output?

```
public static void main(String[] args) {  
    int[] ints = {0, 2, 1, 5};  
    try {  
        System.out.print ("Searching for 6.  ");  
        System.out.println ("Index is " + findIndex(ints, 6));  
    }  
    catch (IndexOutOfBoundsException e) {  
        System.out.println ("\nThere must be an array error.");  
        System.out.println ("Message is:  " + e.getMessage());  
    }  
}
```

- **Answer:**  
Searching for 6.  
There must be an array error.  
Message is: 4

# Exception Propagation

- In previous example:
  - Exception thrown in one method (findIndex) ...
  - ... but caught in another (main)
- Java uses **exception propagation** to look for exception handlers
  - When an exception occurs, Java pops back up the call stack to each of the calling methods to see whether the exception is being handled (by a try-catch block). This is **exception propagation**
  - The **first method** it finds that catches the exception will have its catch block executed. **Execution resumes normally** in the method after this catch block
  - If we get all the way back to main and no method catches this exception, Java catches it and **aborts** your program

# Throwing Exceptions in Java

- To throw an exception, use throw command:  
`throw e;`  
e must evaluate to an exception object
- You can create exceptions just like other objects, e.g.:

```
RuntimeException e = new RuntimeException  
    ("Uh oh");
```

- `RuntimeException` is a class
- Calling `new` this way invokes constructor for this class
- `RuntimeException` generalizes other kinds of exceptions (e.g. `ArithmeticException`)

# Example: What Is Output?

```
public class EasyExample {  
  
    public static void g(boolean b) {  
        System.out.println ("Entering g ...");  
        if (b) throw new RuntimeException("Oh no, you really messed up...");  
        System.out.println ("Exiting g ...");  
    }  
  
    public static void f(int i) {  
        System.out.println ("Entering f ...");  
        g(i == 0);  
        System.out.println ("Exiting f ...");  
    }  
  
    public static void main(String[] args) {  
        try {  
            f(1);  
        }  
        catch (RuntimeException e) {  
            System.out.print ("Runtime error.  Message:  ");  
            System.out.println (e.getMessage());  
        }  
    }  
}
```

# Example: Handling more than one exception?



```
public static void main(String[] args) {
    String d = "";
    try {
        d = JOptionPane.showInputDialog("Enter date: (mm/dd/yyyy)");
        int year = getYear(d);
        System.out.println("The year is " + year);
    }
    catch(IndexOutOfBoundsException e) {
        System.out.println("index error");
    }
    catch(NumberFormatException e) {
        System.out.println("Number format problem");
        return; // NOTE: finally block will still run!!
    }
    finally{
        System.out.println("Finally block is running now...");
    }
}
```