

Using Exceptions

Recitation 3





Programming Special Cases

- What should `"Hi".indexOf('a')` return?
 - -1
- This method requires checking if the value is -1 most times you use it
- Slightly inefficient
- Very annoying and error prone



Exception vs. Error

- Error
 - Mistake on the part of the programmer
- Exception
 - A special type of object in Java used to handle either special behavior in the program or programmer error

Exception Syntax

```
try {  
    for (int i = 0; i < args.length; i++) {  
        switch (args[i]) {  
            case "--file":  
                filename = args[i+1];  
                i++; // advance past filename  
                // as a one-liner: filename = args[++i];  
                break;  
            ...  
        }  
    }  
} catch (ArrayIndexOutOfBoundsException e) {  
    print_usage_message();  
    return;  
}
```

```
static int f(int a) {  
    if(a < 0 || a >= arr.length) {  
        throw new ArrayIndexOutOfBoundsException();  
    }  
    return arr[0];  
}
```




Throwable

- If `e` has type `Throwable`, `throw e` is a valid statement
- Allows for a new type of control flow that goes “up” to the “lowest” try catch block
- The exception is then bound and in scope in the catch block
- All exceptions implement `Throwable`



Finally Block And Multiple Catches

- Every try block must have at least one catch
- Multiple catches correspond to different exceptions
- Finally block contains code that gets run regardless of which or if any exceptions are caught
 - Good for essential clean up like closing resources



```
boolean g(int a) {  
    int b;  
    try {  
        b = f(a);  
    }  
    catch(ArrayIndexOutOfBoundsException aioobe) {  
        aioobe.printStackTrace();  
    }  
    catch(NullPointerException | IOException e) {  
        e.printStackTrace();  
    }  
    finally {  
        b = 7;  
    }  
    return b == 7;  
}
```



Checked Exception

- Required to be handled by either try catch or passing the exception further along as indicated in method header
- Represents unusual but unpreventable circumstance
- Useful to factor out code for rare cases
- Required to be checked
- Ex. IOException for if some input unexpectedly closes
- Commonly get thrown and caught in correctly written code



Unchecked Exception

- Not required to be handled
- Usually represent some kind of programmer error
improper use of array or calling method on null
- Ex. NullPointerException
- Can be thrown and caught in correctly written code but unlikely



Specifying Partial Functions

```
/** Returns: length of nth side of a triangle  
 *   Requires: 0 <= n <= 2  
 */  
double sideLength(int n);
```

What happens if the client calls `sideLength(-1)`?



Partial Functions: Checks clause

```
/** Returns: length of nth side of a triangle  
 * Checks:  $0 \leq n \leq 2$   
 */  
double sideLength(int n);
```

Better, but what exception will be thrown when the check fails?



Partial Functions: Assert example

```
/** Returns: length of nth side of a triangle
 * Checks: 0 <= n <= 2 (assert)
 */
double sideLength(int n) {
    assert n >= 0 && n <= 2;
    ...
}
```

Better, but if an exception is thrown, it still indicates a problem in the client code.



Partial Functions: Make them total

```
/** Returns: length of nth side of a triangle. Throws  
 *      OutOfBoundsException if n < 0 or n > 2.  
 */  
double sideLength(int n) throws OutOfBoundsException;
```

The exception is now expected behavior in some situations and must be handled by the client.