Exception Blocks Good Practices

- Do not leave a catch block empty
  - Basically ignores that an error has happened, which doesn't fix the problem
- Include enough information in the exception to understand the error
  - You can create your own exceptions if existing ones don't have enough information for you
- Know which exceptions are thrown to your code
- Standardize your project's use of exceptions
- Catch specific exceptions when you can
  - Can include a more general catch-all exception after the specific ones

Sizing Exceptions

- How big should your try block be?
  - Only as much code as may fail in a consistent operation
- How detailed should your catch parameter be?
  - Be as specific as you can reasonably be

How much is too much?

- Some exceptions are very specific
  - Eg. Array index out of range
- Does that mean you should have every array access within a try block in case you have a bad index?
  - No. Use a try block on code where there is some external influence contributing to the error.
  - If your own logic is generating the error then find it in debugging or use assertions.

DALHOUSIE UNIVERSITY
Inspiring Minds

Assertions

- Could be validating input parameters in private methods
  - Callers of private methods should already know what good data is and be sending good data

- Used around branches
  - Body of “if” statements to state what should be
  - In and around loops
    - Before the loop – precondition
    - Inside the loop – loop invariant
    - After the loop -- postcondition

DALHOUSIE UNIVERSITY
Inspiring Minds

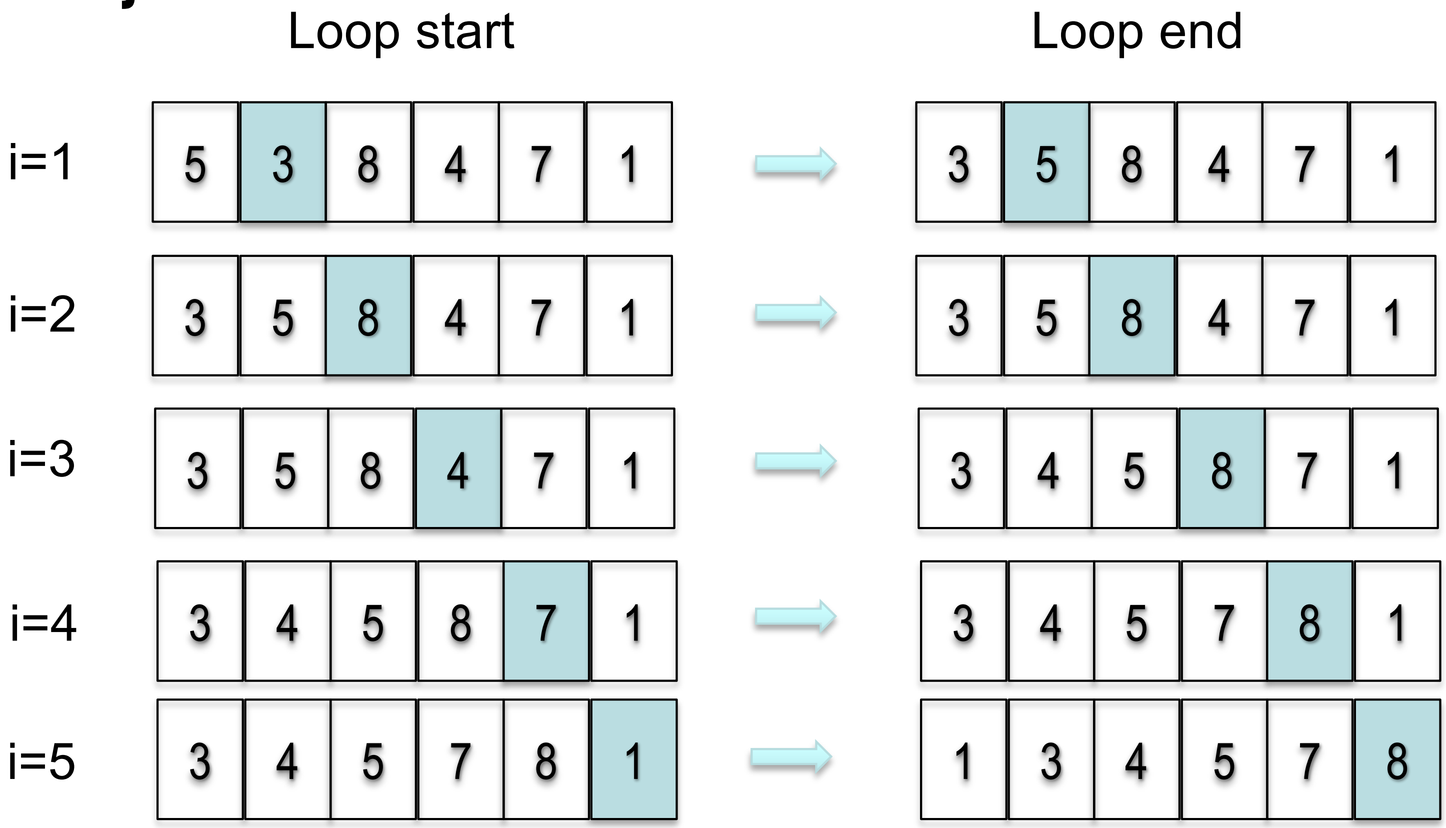
Contract Programming Example – Insertion Sort

```
insertionSort( int[ ] sortMe ) {
    …
    for (int i = 1; i < sortMe.length; i ++) {

        for (int j = i; (j > 0) && (sortMe[ j-1 ] > sortMe[ j ]); j--) {

            swap sortMe[ j-1 ] and sortMe[ j ]

        }

    }
    …
}
```

Outer for loop assertions

- **for (int i = 1; i < sortMe.length; i ++) {**

```
for (int i = 1; i < sortMe.length; i ++) {
    …
}
```



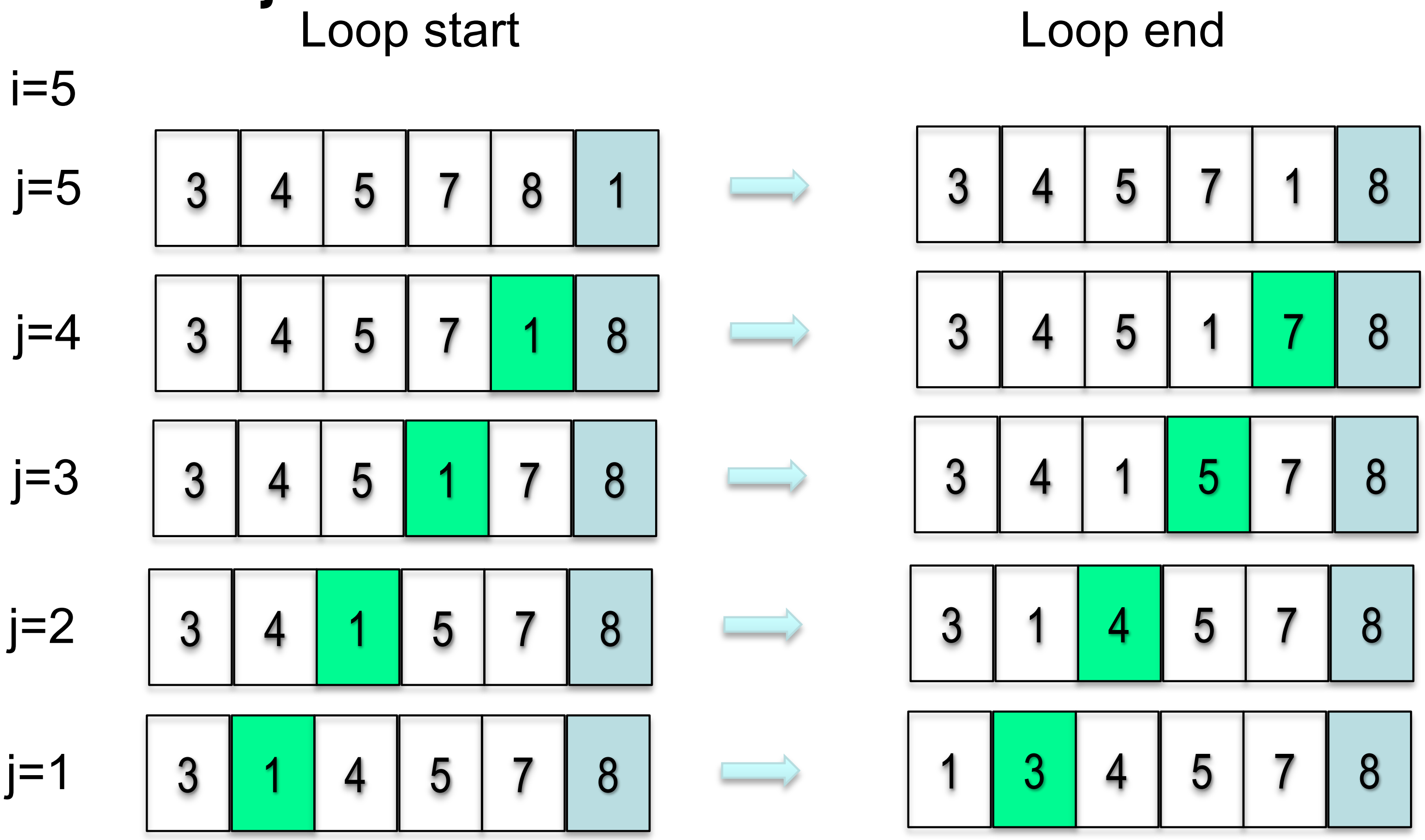
	Loop start							Loop end					
i=1	5	**3**	8	4	7	1	→	3	**5**	8	4	7	1
i=2	3	5	**8**	4	7	1	→	3	5	**8**	4	7	1
i=3	3	5	8	**4**	7	1	→	3	4	5	**8**	7	1
i=4	3	4	5	8	**7**	1	→	3	4	5	7	**8**	1
i=5	3	4	5	7	8	**1**	→	1	3	4	5	7	**8**

Contract Programming Example – Insertion Sort

```
insertionSort( int[ ] sortMe ) {
                                          assert isSorted( sortMe, 0, 0 ) ;
   …
   for (int i = 1; i < sortMe.length; i ++) {
                                          assert isSorted( sortMe, 0, i-1 ) ;
      for (int j = i; (j > 0) && (sortMe[ j-1 ] > sortMe[ j ]); j--) {

         swap sortMe[ j-1 ] and sortMe[ j ]

      }
                                          assert isSorted( sortMe, 0, i ) ;
   }
                                          assert isSorted( sortMe, 0, sortMe.length-1 ) ;
   …
}
```

Inner for loop assertions

-
  ```
  for (int j = i; (j > 0) && (sortMe[ j-1 ] > sortMe[ j ]); j--) {
          swap sortMe[ j-1 ] and sortMe[ j ]
      }
  ```



i=5

	Loop start		Loop end
j=5	3 4 5 7 8 1	→	3 4 5 7 1 8
j=4	3 4 5 7 1 8	→	3 4 5 1 7 8
j=3	3 4 5 1 7 8	→	3 4 1 5 7 8
j=2	3 4 1 5 7 8	→	3 1 4 5 7 8
j=1	3 1 4 5 7 8	→	1 3 4 5 7 8

Contract Programming Example – Insertion Sort

```
insertionSort( int[ ] sortMe ) {
  ...                                      assert isSorted( sortMe, 0, 0 ) ;
  for (int i = 1; i < sortMe.length; i ++) {
                                           assert isSorted( sortMe, 0, i-1 ) ;
    for (int j = i; (j > 0) && (sortMe[ j-1 ] > sortMe[ j ]); j--) {
                                           assert isSorted( sortMe, j, i ) ;
      swap sortMe[ j-1 ] and sortMe[ j ]
                                           assert isSorted( sortMe, j-1, i ) ;
    }
                                           assert isSorted( sortMe, 0, i ) ;
  }
                                           assert isSorted( sortMe, 0, sortMe.length-1 ) ;
  ...
}
```

DALHOUSIE UNIVERSITY
Inspiring Minds

Programming paradigms

- **Procedural programming** C, Fortran, Cobol
  - Generally focuses on the operations, steps, and transformations needed to achieve an outcome
- **Object oriented programming** Java, C++, Python
  - Focuses on the data, concepts, or elements around which computation is happening
- **Functional programming** Lisp, ML, Haskell, OCaml
  - Program flow modeled as a composition of function calls
- **Logic programming** Prolog
  - Focus on the rules behind all the computation and let the running environment look to combine rules as they apply to reach an answer.

DALHOUSIE UNIVERSITY *Inspiring Minds*