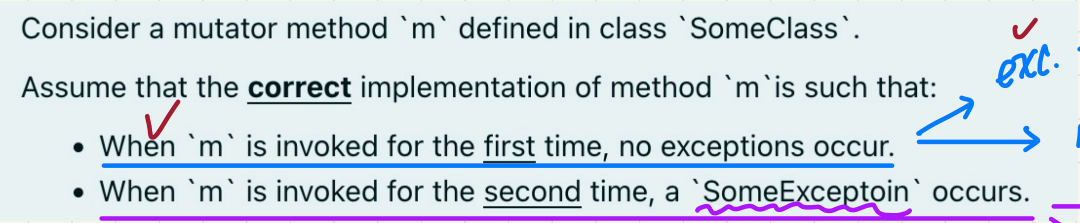
Final Exam

Review Q&A



Consider a mutator method `m` defined in class `SomeClass`.

Assume that the **correct** implementation of method `m`is such that:

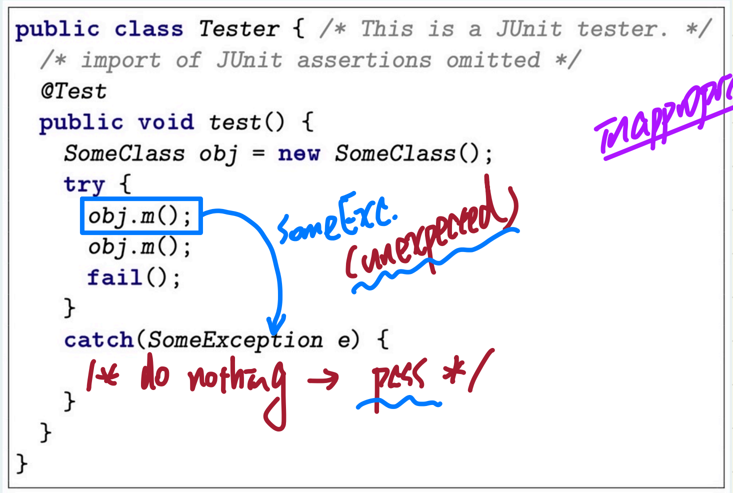
- When `m` is invoked for the first time, no exceptions occur.
- When `m` is invoked for the second time, a `SomeExceptoin` occurs.

✓ exc. → fail

no exc. → pass

SomeEx. → pass

no exc.
exc other than SomeEx. → fail

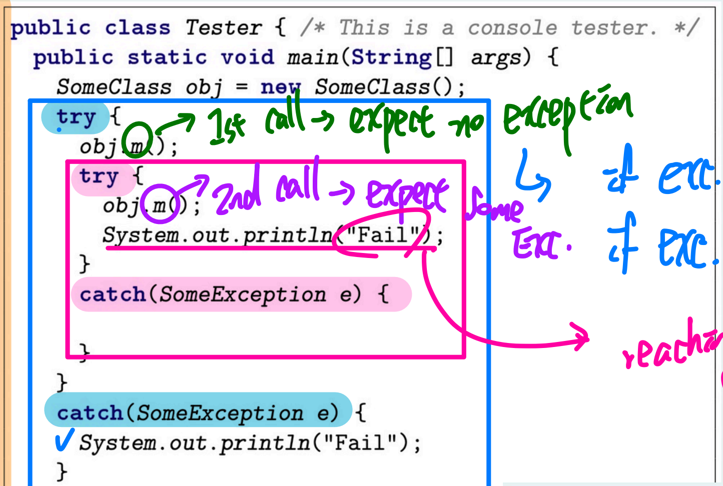


```
public class Tester { /* This is a JUnit tester. */
  /* import of JUnit assertions omitted */
  @Test
  public void test() {
    SomeClass obj = new SomeClass();
    try {
      obj.m();
      obj.m();
      fail();
    }
    catch(SomeException e) {
      /* do nothing → pass */
    }
  }
}
```

SomeExc. (unexpected)

inappropriate

appropriate



```
public class Tester { /* This is a console tester. */
  public static void main(String[] args) {
    SomeClass obj = new SomeClass();
    try {
      obj.m();
      try {
        obj.m();
        System.out.println("Fail");
      }
      catch(SomeException e) {

      }
    }
    catch(SomeException e) {
      System.out.println("Fail");
    }
  }
}
```

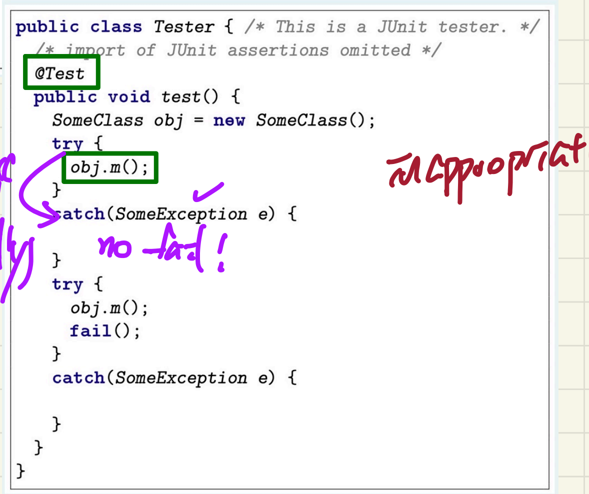
1st call → expect no exception

↳ if exc. occurred → caught and "Fail" printed

if exc. not occurred → proceed with normal flow

2nd call → expect Some Exc.

reaching this line means ≠ the expected SomeEx did not occur.

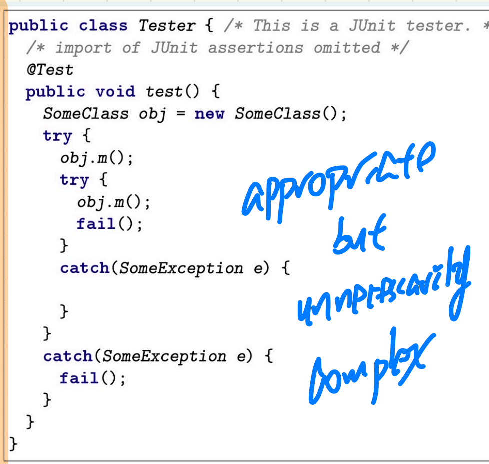


```
public class Tester { /* This is a JUnit tester. */
  /* import of JUnit assertions omitted */
  @Test
  public void test() {
    SomeClass obj = new SomeClass();
    try {
      obj.m();
    }
    catch(SomeException e) {

    }
    try {
      obj.m();
      fail();
    }
    catch(SomeException e) {

    }
  }
}
```

SomeEx occurred (unexpectedly)

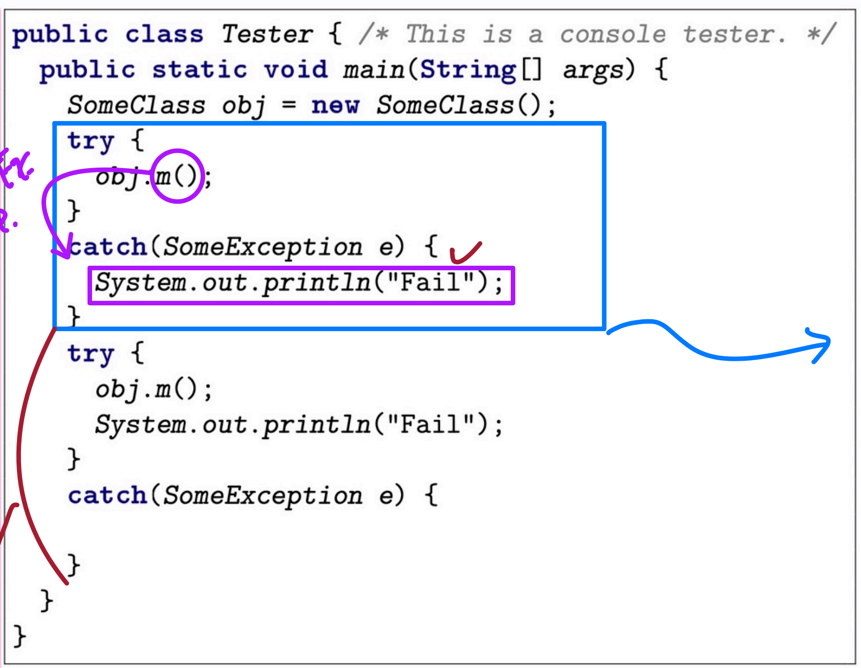
no fail!

inappropriate!



```
public class Tester { /* This is a JUnit tester. */
  /* import of JUnit assertions omitted */
  @Test
  public void test() {
    SomeClass obj = new SomeClass();
    try {
      obj.m();
      try {
        obj.m();
        fail();
      }
      catch(SomeException e) {

      }
    }
    catch(SomeException e) {
      fail();
    }
  }
}
```

appropriate but unnecessarily complex



```
public class Tester { /* This is a console tester. */
  public static void main(String[] args) {
    SomeClass obj = new SomeClass();
    try {
      obj.m();
    }
    catch(SomeException e) {
      System.out.println("Fail");
    }
    try {
      obj.m();
      System.out.println("Fail");
    }
    catch(SomeException e) {

    }
  }
}
```

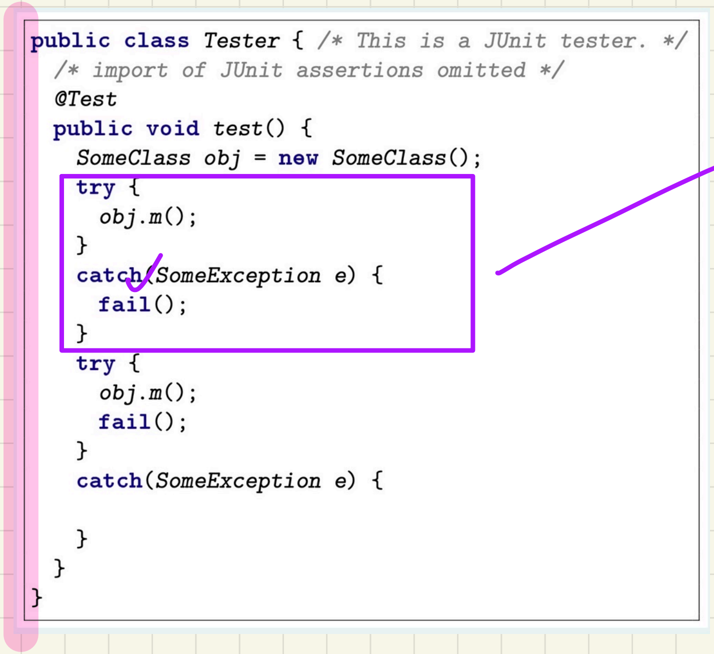
SomeEx unexpe.

✓

as soon as a first failure occurres, normal flow of exec should be interrupted

should not be continued if a "fail" occurs already.

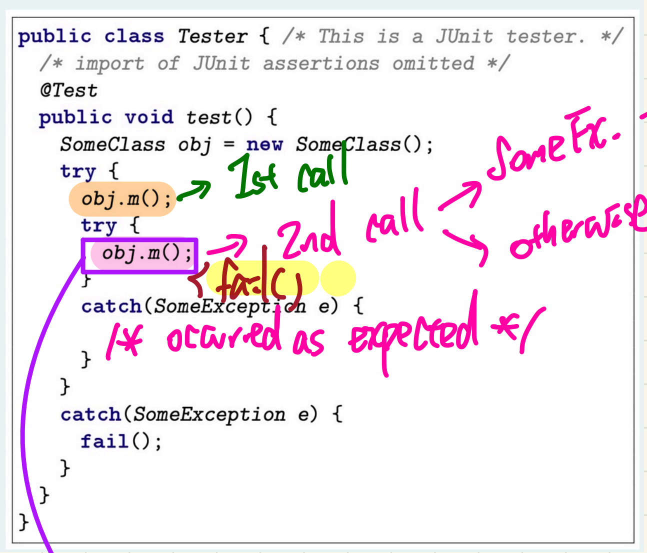
(need to have nested try-catch)



```
public class Tester { /* This is a JUnit tester. */
  /* import of JUnit assertions omitted */
  @Test
  public void test() {
    SomeClass obj = new SomeClass();
    try {
      obj.m();
    }
    catch(SomeException e) {
      fail();
    }
    try {
      obj.m();
      fail();
    }
    catch(SomeException e) {

    }
  }
}
```

as soon as the first "fail" occurs, the flow is interrupted.

(no need to have nested try-catch)



```
public class Tester { /* This is a JUnit tester. */
  /* import of JUnit assertions omitted */
  @Test
  public void test() {
    SomeClass obj = new SomeClass();
    try {
      obj.m();
      try {
        obj.m();
      }
      catch(SomeException e) {
      }
    }
    catch(SomeException e) {
      fail();
    }
  }
}
```

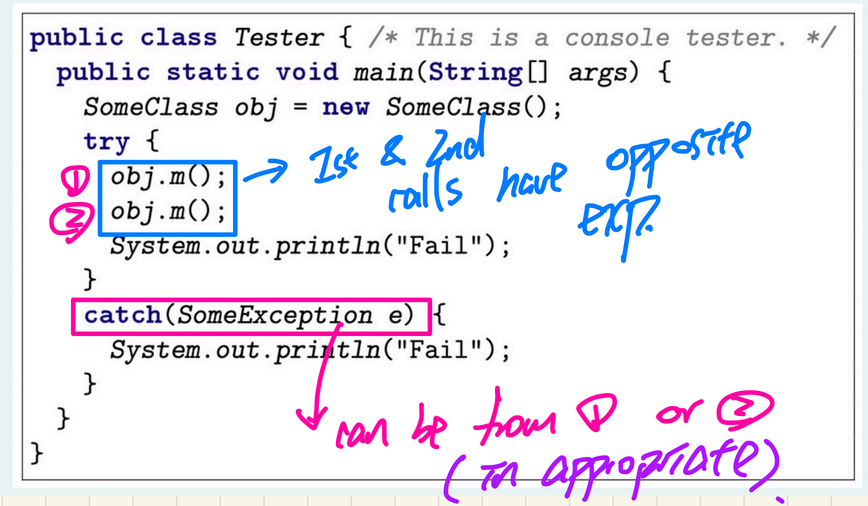
1st call

2nd call → SomeEx. → pass

otherwise → fail

fail()

/* occurred as expected */



```
public class Tester { /* This is a console tester. */
  public static void main(String[] args) {
    SomeClass obj = new SomeClass();
    try {
      obj.m();
      obj.m();
      System.out.println("Fail");
    }
    catch(SomeException e) {
      System.out.println("Fail");
    }
  }
}
```

① ② → 1st & 2nd calls have opposite exp.

can be from ① or ② (inappropriate)

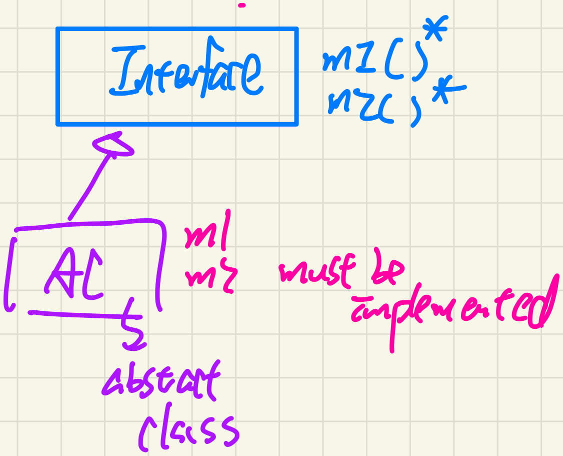
if the expected SomeEx. did not occur, the test would not fail!

inappropriate

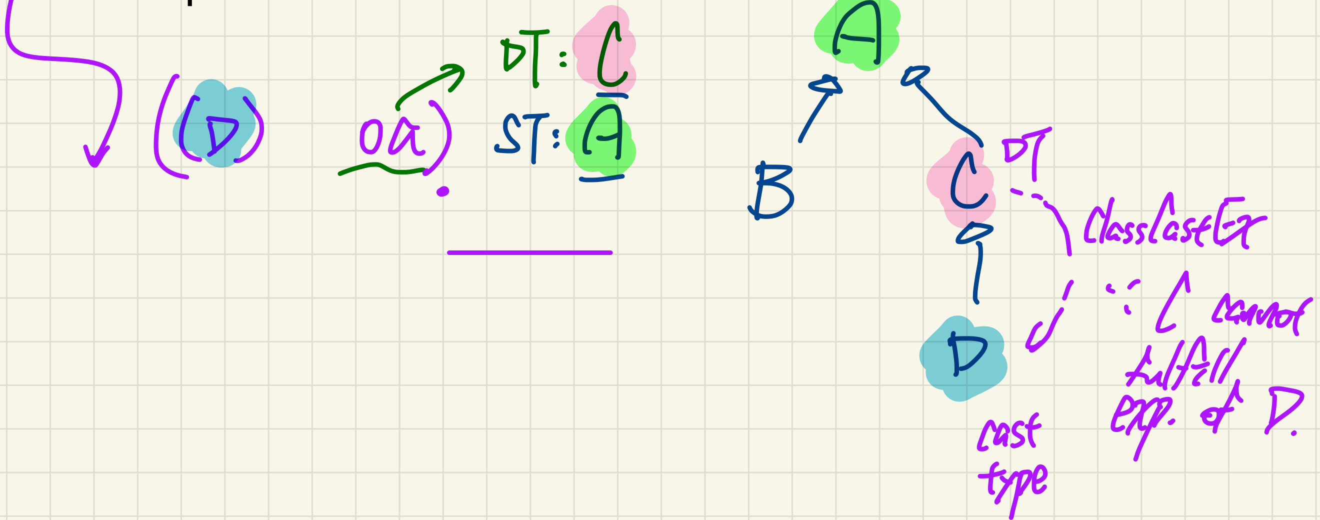
Past lab on recursion

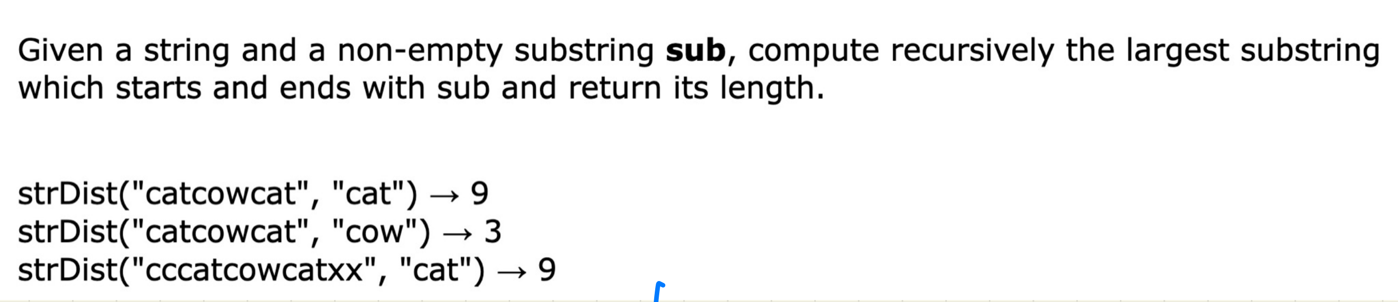
↳ only focus on arrays

↳ ArrayList not covered.



Are descendant classes only useful if they have more expectations than their ancestor classes? If not, on page 5 of your written notes on "Static Types, Expectations, Dynamic Types, and Type Casts", when you cast (D) oa, it does not work because the "d" attribute is not declared in class C. But if we suppose that class D simply didn't have the "od.d" expectation (so only "a" and "c" attributes), would it still cause a ClassCastException even though D and C would have the same expectations?

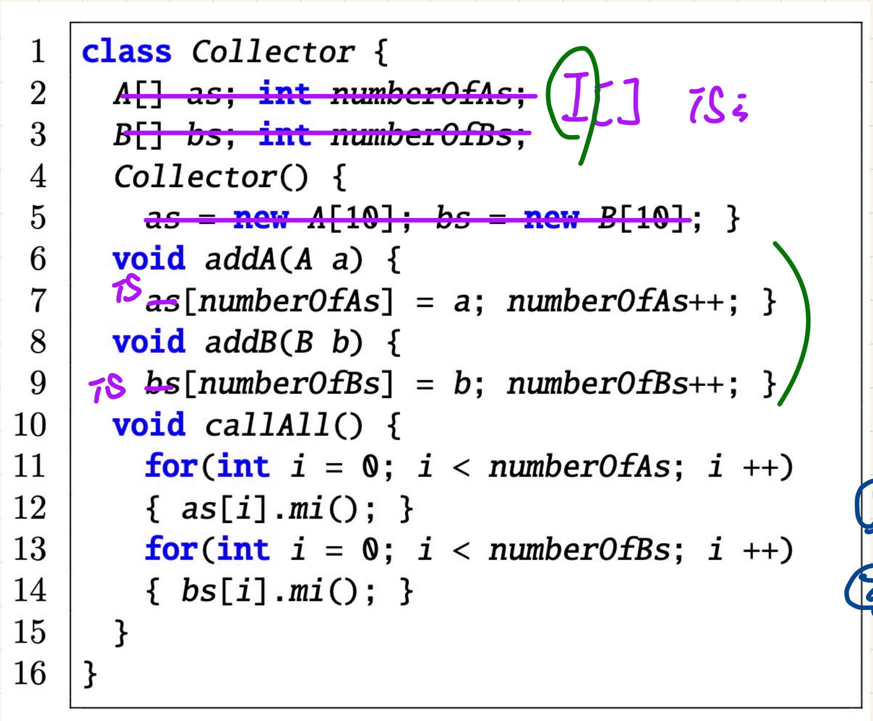




Given a string and a non-empty substring **sub**, compute recursively the largest substring which starts and ends with sub and return its length.

```
strDist("catcowcat", "cat") → 9
strDist("catcowcat", "cow") → 3
strDist("cccatcowcatxx", "cat") → 9
```

hints on recursive thinking to come!



```
class Collector {
  ~~A[] as; int numberOfAs;~~  I[]  is;
  ~~B[] bs; int numberOfBs;~~
  Collector() {
    ~~as = new A[10]; bs = new B[10]~~; }
  void addA(A a) {
  is
    ~~as~~[numberOfAs] = a; numberOfAs++; }
  void addB(B b) {
 is ~~bs~~[numberOfBs] = b; numberOfBs++; }
  void callAll() {
    for(int i = 0; i < numberOfAs; i ++)
    { as[i].mi(); }
    for(int i = 0; i < numberOfBs; i ++)
    { bs[i].mi(); }
  }
}
```

```
void addI(I i) {
  is[noi] = i; noi++
}
```

① c.addI(new A());

② c.addI(new B());