Lecture 6

Monday September 23

- Guides
  - ~ Written Test
  - ~ Programming Test
- Review Session

Tentatively: 10am ~ 12 noon

Monday Sep. 30

Error Handling via Console Messages: Circles

```
1  class Circle {
2    double radius;
3    Circle() { /* radius defaults to 0 */ }
4    void setRadius(double r) {
5      if ( r < 0 ) { System.out.println( "Invalid radius." ); }
6      else { radius = r; }
7    }
8    double getArea() { return radius * radius * 3.14; }
9  }
```

Caller?

Callee?

```
1  class CircleCalculator {
2    public static void main(String[] args) {
3      Circle c = new Circle();
4      c.setRadius(-10);
5      double area = c.getArea();
6      System.out.println("Area: " + area);
7    }
8  }
```

print error to console

when 'r' is invalid, these two lines should not be executed.

Error Handling via Console Messages: Bank

```
class Account {
  int id; double balance;
  Account(int id) { this.id = id; /* balance defaults to 0 */ }
  void deposit(double a) {
    if ( a < 0 ) { System.out.println( "Invalid deposit." ); }
    else { balance += a; }
  }
  void withdraw(double a) {
    if ( a < 0 || balance - a < 0 ) {
      System.out.println( "Invalid withdraw." ); }
    else { balance -= a; }
  }
}
```

```
class Bank {
  Account[] accounts; int numberOfAccounts;
  Account(int id) { ... }
  void withdrawFrom(int id, double a) {
    for(int i = 0; i < numberOfAccounts; i ++) {
      if(accounts[i].id == id) {
        accounts[i].withdraw( a );
      }
    }
  } /
}
```

```
class BankApplication {
  pubic static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    Bank b = new Bank(); Account acc1 = new Account(23);
    b.addAccount(acc1);
    double a = input.nextDouble();
    b.withdrawFrom(23, a);
    System.out.println("Transaction Completed.");
  }
```

Caller?

Callee?

call stack

Account.withdraw

Bank.wf

BankApp.main

context	caller	callee
Account		
Bank		
BankApp		

Catch-or-Specify Requirement

1. **The "Catch" Solution:** A `try` statement that *catches and handles the exception*.

```
main(...) {
  Circle c = new Circle();
  try {
    c.setRadius(-10);
  }
  catch(NegativeRaidusException e) {
    ...
  }
}
```

The "Specify" Solution: A method that specifies as part of its *signature* that it *can throw* the exception (without handling that exception).

```
class Bank {
  Account[] accounts; /* attribute */
  void withdraw (double amount)
     throws InvalidTransactionException {
    ...
    accounts[i].withdraw(amount);
    ...
  }
}
```

Example: To Handle or Not To Handle?

context	caller	callee

```
class A {
  ma(int i) {
    if (i < 0) { /* Error */ }
    else { /* Do something. */ }
  } }
```

```
class B {
  mb(int i) {
    A oa = new A();
    oa.ma(i); /* Error occurs if i < 0 */
  } }
```

```
class Tester {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    int i = input.nextInt();
    B ob = new B();
    ob.mb(i); /* Where can the error be handled? */
  } }
```

Version 1:
Handle it in `B.mb`

Version 2:
Pass it from `B.mb` and handle it in `Tester.main`

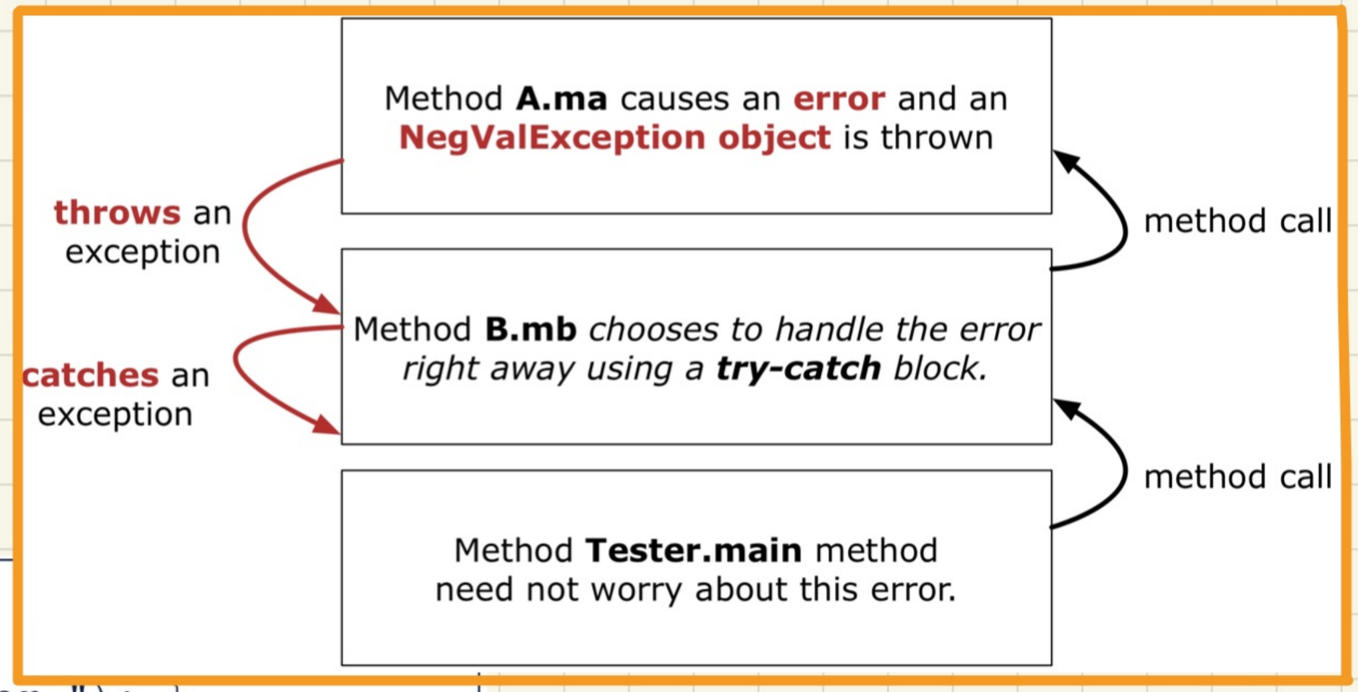
Version 3:
Pass it from `B.mb`, then from `Tester.main`, then throw it to the console.

```
class NegValException extends Exception {
  NegValException(String s) { super(s); }
}
```

call stack

A.ma
B.mb
Tester.main

Version 1:
Handle the Exception in B.mb




```
class A {
  ma(int i) throws NegValException {
    if(i < 0) { throw new NegValException("Error."); }
    else { /* Do something. */ }
  } }
```

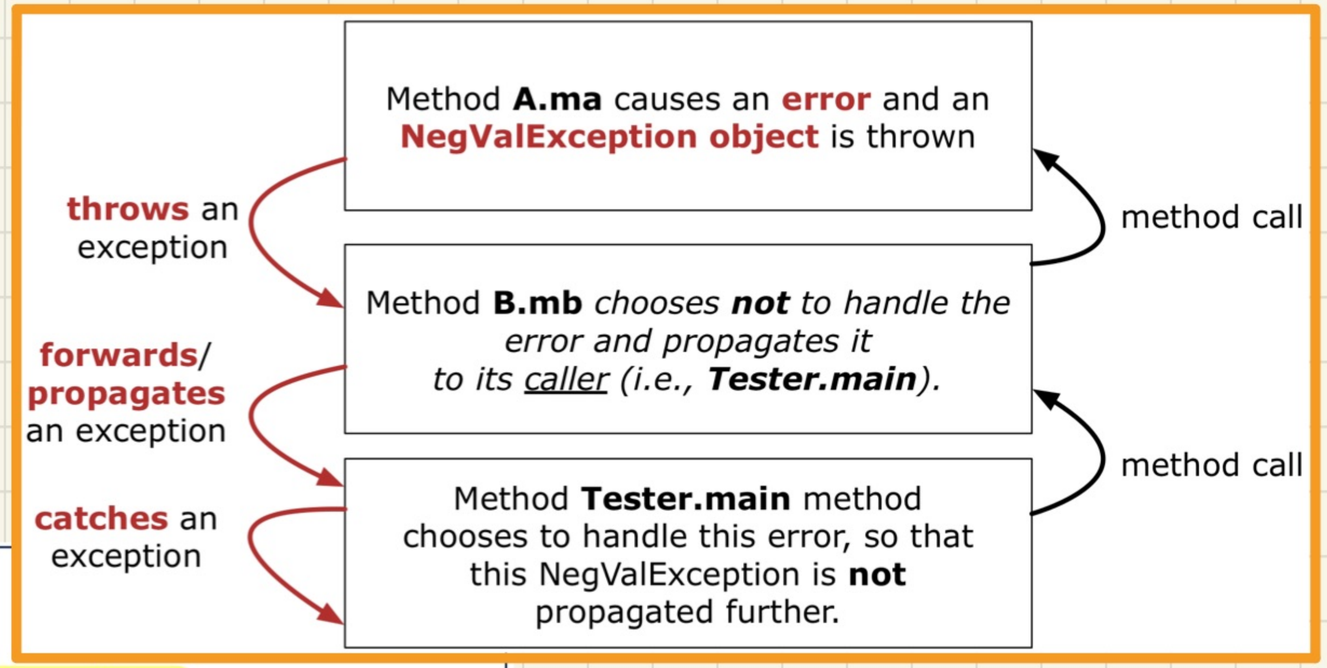
```
class B {
  mb(int i) {
    A oa = new A();
    try { oa.ma(i); }
    catch(NegValException nve) { /* Do something. */ }
  } }
```

```
class Tester {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    int i = input.nextInt();
    B ob = new B();
    ob.mb(i); /* Error, if any, would have been handled in B.mb.
  } }
```

no exception handling is necessary ∵ it's handled in B already.

Version 2:

Handle the Exception in Tester.main




```
class A {
  ma(int i) throws NegValException {
    if(i < 0) { throw new NegValException("Error."); }
    else { /* Do something. */ }
  } }
```

where we signal the error.

```
class B {
  mb(int i) throws NegValException {
    A oa = new A();
    oa.ma(i);
  } }
```

handle if specify

any caller of B.mb should handle this exception

```
class Tester {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    int i = input.nextInt();
    B ob = new B();
    try { ob.mb(i); }
    catch(NegValException nve) { /* Do something. */ }
  } }
```

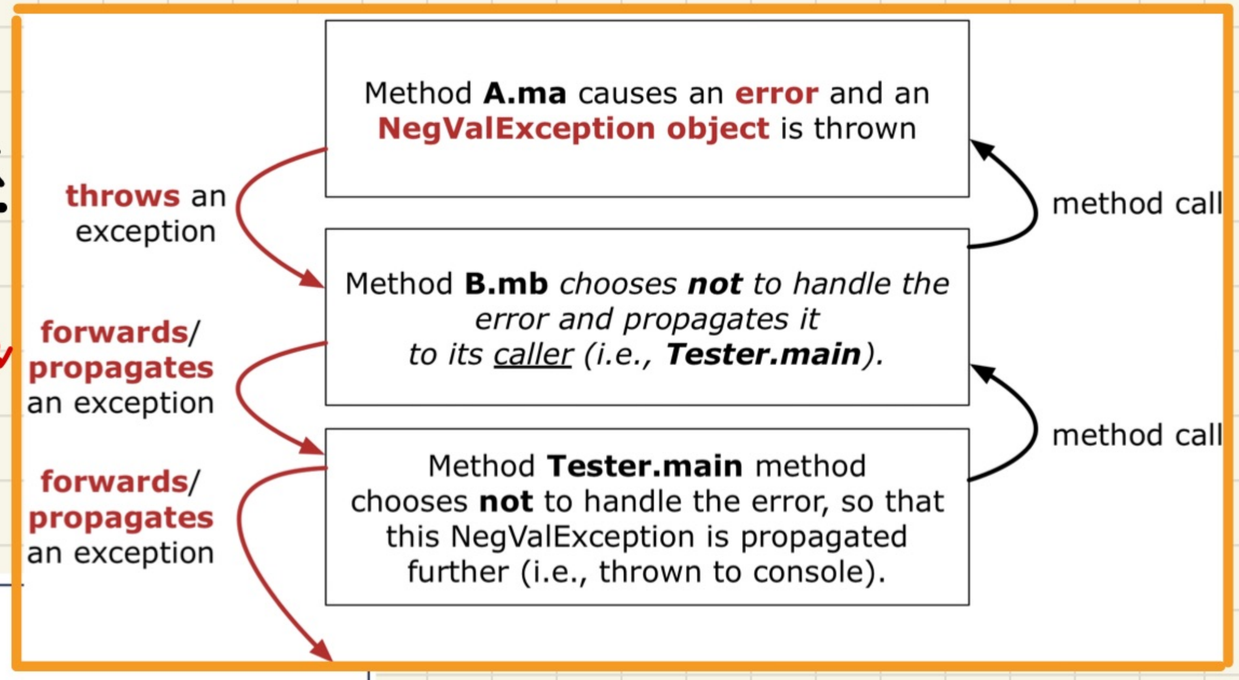
may throw NegValExcep

Version 3:

Handle in Neither Classes on Call Stack

throw : signal an error

throws : → warn any potential caller that they must handle this error.




```
class A {
  ma(int i) throws NegValException {
    if(i < 0) { throw new NegValException("Error."); }
    else { /* Do something. */ }
  } }
```

```
class B {
  mb(int i) throws NegValException {
    A oa = new A();
    oa.ma(i);
  } }
```

```
class Tester {
  public static void main(String[] args) throws NegValException {
    Scanner input = new Scanner(System.in);
    int i = input.nextInt();
    B ob = new B();
    ob.mb(i);
  } }
```

Error Handling via Exceptions: Circles (Version 1)

```
public class InvalidRadiusException extends Exception {
  public InvalidRadiusException(String s) {
    super(s);
  }
}
```

Case 1: Valid radius 5.

Case 2: Invalid radius -4.

```
class Circle {
  double radius;
  Circle() { /* radius defaults to 0 */ }
  void setRadius(double r) throws InvalidRadiusException {
    if (r < 0) {
      throw new InvalidRadiusException("Negative radius.");
    }
    else { radius = r; }
  }
  double getArea() { return radius * radius * 3.14;
}
```

```
class CircleCalculator1 {
  public static void main(String[] args) {
    Circle c = new Circle();
    try {
      c.setRadius(-4);
      double area = c.getArea();
      System.out.println("Area: " + area);
    }
    catch(InvalidRadiusException e) {
      System.out.println(e);
    }
  }
}
```

IRE thrown

Error Handling via Exceptions: Circles (Version 2)

```
public class InvalidRadiusException extends Exception {
  public InvalidRadiusException(String s) {
    super(s);
  }
}
```

```
class Circle {
  double radius;
  Circle() { /* radius defaults to 0 */ }
  void setRadius(double r) throws InvalidRadiusException {
    if (r < 0) {
      throw new InvalidRadiusException("Negative radius.");
    }
    else { radius = r; }
  }
  double getArea() { return radius * radius * 3.14; }
}
```

Test Case:

User enters **-5**

Then user enters **10**

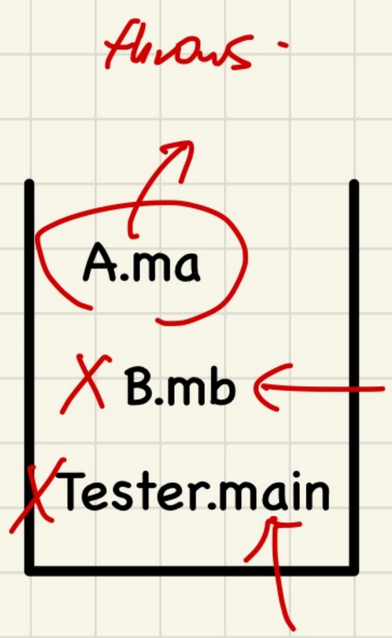
if we are able to move from L1 to L2 what does it mean?

```
public class CircleCalculator2 {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    boolean inputRadiusIsValid = false;
    while (!inputRadiusIsValid) {
      System.out.println("Enter a radius:");
      double r = input.nextDouble();
      Circle c = new Circle();
      try { c.setRadius(r);
            inputRadiusIsValid = true;
            System.out.print("Circle with radius " + r);
            System.out.println(" has area: "+ c.getArea()); }
      catch(InvalidRadiusException e) { print("Try again!"); }
    } } }
```

What to Do When an Exception is Thrown?

After a method *throws an exception*, the *runtime system* searches the corresponding *call stack* for a method that contains a block of code to *handle* the exception.

- This block of code is called an *exception handler*
  - An exception handler is **appropriate** if the *type* of the *exception object thrown* matches the *type* that can be handled by the handler.
  - The exception handler chosen is said to *catch* the exception.
- The search goes from the *top* to the *bottom* of the call stack:
  - The method in which the *error* occurred is searched first.
  - The *exception handler* is not found in the current method being searched ⇒ Search the method that calls the current method, and *etc.*
  - When an appropriate *handler* is found, the *runtime system* passes the exception to the handler.
- The *runtime system* searches all the methods on the *call stack* without finding an **appropriate** *exception handler*
  ⇒ The program terminates and the exception object is directly "thrown" to the console!



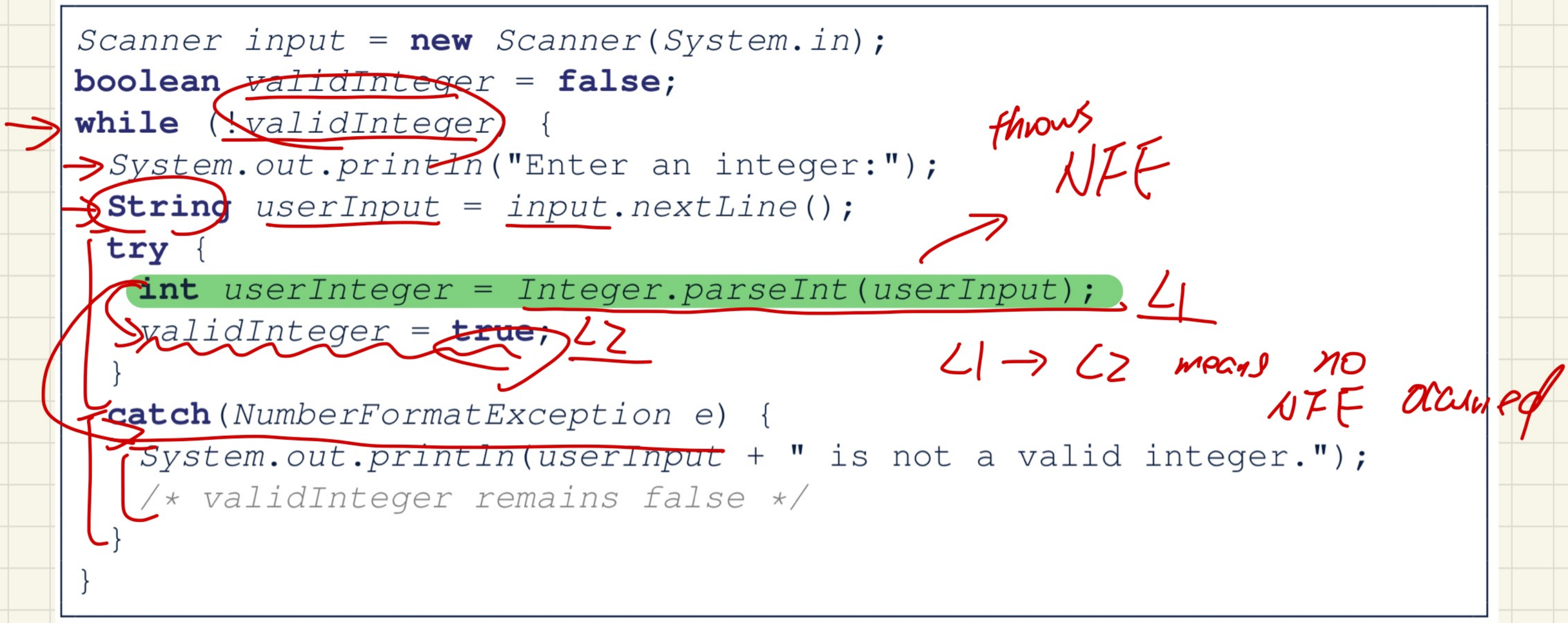
Read from user :

"23" String

↳ 23 int

More Example: Parsing Strings as Integers

```
Scanner input = new Scanner(System.in);
boolean validInteger = false;
while (!validInteger) {
 System.out.println("Enter an integer:");
 String userInput = input.nextLine();
 try {
  int userInteger = Integer.parseInt(userInput);
  validInteger = true;
 }
 catch(NumberFormatException e) {
  System.out.println(userInput + " is not a valid integer.");
  /* validInteger remains false */
 }
}
```



Review: Specify-or-Catch Principle

Approach 1 – Specify: Indicate in the method signature that a specific exception might be thrown.

Example 1: Method that throws the exception

```
class C1 {
  void m1(int x) throws ValueTooSmallException {
    if(x < 0) {
      throw new ValueTooSmallException("val " + x);
    }
  }
}
```

Example 2: Method that calls another which throws the exception

```
class C2 {
  C1 c1;
  void m2(int x) throws ValueTooSmallException {
    c1.m1(x);
  }
}
```

Review: Specify-or-Catch Principle

Approach 2 – Catch: Handle the thrown exception(s) in a try-catch block.

```
class C3 {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    int x = input.nextInt();
    C2 c2 = new c2();
    try {
      c2.m2(x);
    }
    catch(ValueTooSmallException e) { ... }
  }
}
```

Manual Test 1 from the Console

```
1  public class CounterTester1 {
2    public static void main(String[] args) {
3      Counter c = new Counter();
4      println("Init val: " + c.getValue());
5      try {
6        c.decrement();
7        println("Error: ValueTooSmallException NOT thrown.");
8      }
9      catch (ValueTooSmallException e) {
10       println("Success: ValueTooSmallException thrown.");
11     }
12   } /* end of main method */
13 } /* end of class CounterTester1 */
```

What if decrement is implemented correctly?

EXPECTED BEHAVIOUR :

Calling c.decrement() when c.value is 0 should trigger a ValueTooSmallException.

```
1  public class CounterTester1 {
2    public static void main(String[] args) {
3      Counter c = new Counter();
4      println("Init val: " + c.getValue());
5      try {
6        c.decrement();
7        println("Error: ValueTooSmallException NOT thrown.");
8      }
9      catch (ValueTooSmallException e) {
10       println("Success: ValueTooSmallException thrown.");
11     }
12   } /* end of main method */
13 } /* end of class CounterTester1 */
```

What if decrement is implemented incorrectly?