Exceptions

EECS2030 B: Advanced
Object Oriented Programming
Fall 2018

CHEN-WEI WANG

Caller vs. Callee

- Within the body implementation of a method, we may call other methods.

```
1  class C1 {
2    void m1() {
3      C2 o = new C2();
4      o.m2(); /* static type of o is C2 */
5    }
6  }
```

- From **Line 4**, we say:
  - Method **C1.m1** (i.e., method `m1` from class `C1`) is the *caller* of method **C2.m2**.
  - Method **C2.m2** is the *callee* of method **C1.m1**.

Why Exceptions? (1.1)

```
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  }
```

- A negative radius is considered as an *invalid input value* to method `setRadius`.
- What if the *caller* of `Circle.setRadius` passes a negative value for `r`?
  - An error message is *printed to the console* (**Line 5**) to warn the ***caller*** of `setRadius`.
  - However, printing an error message to the console *does not force* the ***caller*** `setRadius` to stop and handle invalid values of `r`.

Why Exceptions? (1.2)

```
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  }
```

- **L4**: `CircleCalculator.main` is *caller* of `Circle.setRadius`
- A negative radius is passed to `setRadius` in **Line 4**.
- The execution *always flows smoothly* from **Lines 4** to **Line 5**, *even when there was an error* message printed from **Line 4**.
- It is not feasible to check if there is any kind of error message printed to the console right after the execution of **Line 4**.
- **Solution:** A way to force `CircleCalculator.main`, *caller* of `Circle.setRadius`, to realize that things might go wrong.
  ⇒ When things do go wrong, immediate actions are needed.

Why Exceptions? (2.1)

```
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; }
  }
}
```

- A negative deposit or withdraw amount is *invalid*.
- When an *error* occurs, a message is *printed to the console*.
- However, printing error messages does not force the *caller* of `Account.deposit` or `Account.withdraw` to stop and handle invalid values of `a`.

Why Exceptions? (2.2)

```
1  class Bank {
2    Account[] accounts; int numberOfAccounts;
3    Account(int id) { ... }
4    void withdrawFrom(int id, double a) {
5      for(int i = 0; i < numberOfAccounts; i ++) {
6        if(accounts[i].id == id) {
7          accounts[i].withdraw( a );
8        }
9      } /* end for */
10   } /* end withdraw */
11 }
```

- **L7**: ***Bank.withdrawFrom*** is *caller* of ***Account.withdraw***
- What if in **Line 7** the value of `a` is negative?
  Error message `Invalid withdraw` printed from method ***Account.withdraw*** to console.
- Impossible to force ***Bank.withdrawFrom***, the *caller* of ***Account.withdraw***, to stop and handle invalid values of `a`.

Why Exceptions? (2.3)

```
1  class BankApplication {
2    pubic static void main(String[] args) {
3      Scanner input = new Scanner(System.in);
4      Bank b = new Bank(); Account acc1 = new Account(23);
5      b.addAccount(acc1);
6      double a = input.nextDouble();
7      b.withdrawFrom(23, a );
8  }
```

- There is a chain of method calls:
  - ***BankApplication.main*** calls **Bank.withdrawFrom**
  - **Bank.withdrawFrom** calls ***Account.withdraw***.
- The actual update of balance occurs at the `Account` class.
  - What if in **Line 7** the value of `a` is negative?
    `Invalid withdraw` printed from ***`Bank.withdrawFrom`***, printed from ***`Account.withdraw`*** to console.
  - Impossible to force ***`BankApplication.main`***, the *caller* of ***`Bank.withdrawFrom`***, to stop and handle invalid values of `a`.
- **Solution:** Define error checking only once and let it *propagate*.

What is an Exception?

- An *exception* is an *event*, which
  - occurs during the *execution of a program*
  - *disrupts the normal flow* of the program's instructions
- When an error occurs within a method:
  - the method throws an exception:
    - first creates an *exception object*
    - then hands it over to the *runtime system*
  - the exception object contains information about the error:
    - type [e.g., `NegativeRadiusException`]
    - the state of the program when the error occurred

Exceptions in Java (1.1)

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

- A new kind of `Exception`: `InvalidRadiusException`
- For any method that can have this kind of error, we declare at that method's *signature* that it may *throw* an `InvalidRaidusException` object.

Exceptions in Java (1.2)

```
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; }
}
```

- As part of the *signature* of `setRadius`, we declare that it may *throw* an `InvalidRadiusException` object at runtime.
- Any method that calls `setRadius` will be forced to *deal with this potential error* .

Exceptions in Java (1.3)

```
1  class CircleCalculator1 {
2   public static void main(String[] args) {
3     Circle c = new Circle();
4     try {
5       c.setRadius(-10);
6       double area = c.getArea();
7       System.out.println("Area: " + area);
8     }
9     catch(InvalidRadiusException e) {
10      System.out.println(e);
11    }
12  } }
```

- **Lines 6** is forced to be wrapped within a *try-catch* block, since it may *throw* an `InvalidRadiusException` object.
- If an `InvalidRadiusException` object is thrown from **Line 6**, then the normal flow of execution is *interrupted* and we go to the `catch` block starting from **Line 9**.

Exceptions in Java (1.4.1)

Exercise: Extend `CircleCalculator1`: repetitively prompt for a new radius value until a valid one is entered (i.e., the `InvalidRadiusException` does not occur).

Exceptions in Java (1.4.2)

```
1  public class CircleCalculator2 {
2    public static void main(String[] args) {
3      Scanner input = new Scanner(System.in);
4      boolean inputRadiusIsValid = false;
5      while(!inputRadiusIsValid) {
6        System.out.println("Enter a radius:");
7        double r = input.nextDouble();
8        Circle c = new Circle();
9        try { c.setRadius(r);
10             inputRadiusIsValid = true;
11             System.out.print("Circle with radius " + r);
12             System.out.println(" has area: "+ c.getArea()); }
13       catch(InvalidRadiusException e) { print("Try again!"); }
14     } } }
```

- At **L7**, if the user's input value is:
  - Non-Negative: **L8** – **L12**. [`inputRadiusIsValid` set ***true***]
  - Negative: **L8**, **L9**, **L13**. [`inputRadiusIsValid` remains ***false***]

Exceptions in Java (2.1)

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

- A new kind of `Exception`: `InvalidTransactionException`
- For any method that can have this kind of error, we declare at that method's *signature* that it may *throw* an `InvalidTransactionException` object.

Exceptions in Java (2.2)

```
class Account {
  int id; double balance;
  Account() { /* balance defaults to 0 */ }
  void withdraw(double a) throws InvalidTransactionException {
    if (a < 0 || balance - a < 0) {
      throw new InvalidTransactionException("Invalid withdraw."); }
    else { balance -= a; }
  }
}
```

- As part of the *signature* of `withdraw`, we declare that it may *throw* an `InvalidTransactionException` object at runtime.
- Any method that calls `withdraw` will be forced to *deal with this potential error* .

Exceptions in Java (2.3)

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

- As part of the *signature* of `withdraw`, we declare that it may *throw* an `InvalidTransactionException` object.
- Any method that calls `withdraw` will be forced to *deal with this potential error* .
- We are *propagating* the potential error for the right party (i.e., `BankApplication`) to handle.

Exceptions in Java (2.4)

```
1  class BankApplication {
2   pubic static void main(String[] args) {
3     Bank b = new Bank();
4     Account acc1 = new Account(23);
5     b.addAccount(acc1);
6     Scanner input = new Scanner(System.in);
7     double a = input.nextDouble();
8     try {
9       b.withdraw(23, a);
10      System.out.println(acc1.balance); }
11    catch (InvalidTransactionException e) {
12      System.out.println(e); } } }
```

- **Lines 9** is forced to be wrapped within a *try-catch* block, since it may *throw* an `InvalidTransactionException` object.
- If an `InvalidTransactionException` object is thrown from **Line 9**, then the normal flow of execution is interrupted and we go to the `catch` block starting from **Line 11**.

Examples (1)

```
double r = ...;
double a = ...;
try{
  Bank b = new Bank();
  b.addAccount(new Account(34));
  b.deposit(34, 100);
  b.withdraw(34, a);
  Circle c = new Circle();
  c.setRadius(r);
  System.out.println(r.getArea());
}
catch(NegativeRadiusException e) {
  System.out.println(r + " is not a valid radius value.");
  e.printStackTrace();
}
catch(InvalidTransactionException e) {
  System.out.println(r + " is not a valid transaction value.");
  e.printStackTrace();
}
```

Example (2.1)

The `Integer` class supports a method for parsing Strings:

```
public static int parseInt(String s)
                throws NumberFormatException
```

e.g., `Integer.parseInt("23")` returns `23`

e.g., `Integer.parseInt("twenty-three")` throws a `NumberFormatException`

Write a fragment of code that prompts the user to enter a string (using `nextLine` from `Scanner`) that represents an integer.

If the user input is not a valid integer, then prompt them to enter again.

Example (2.2)

```
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 */
  }
}
```

Example: to Handle or Not to Handle? (1.1)

Consider the following three classes:

```
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?  */
  } }
```

Example: to Handle or Not to Handle? (1.2)

- We assume the following kind of error for negative values:

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

- The above kind of exception may be thrown by calling `A.ma`.
- We will see three kinds of possibilities of handling this exception:

  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.

Example: to Handle or Not to Handle? (2.1)

Version 1: Handle the exception in `B.mb`.

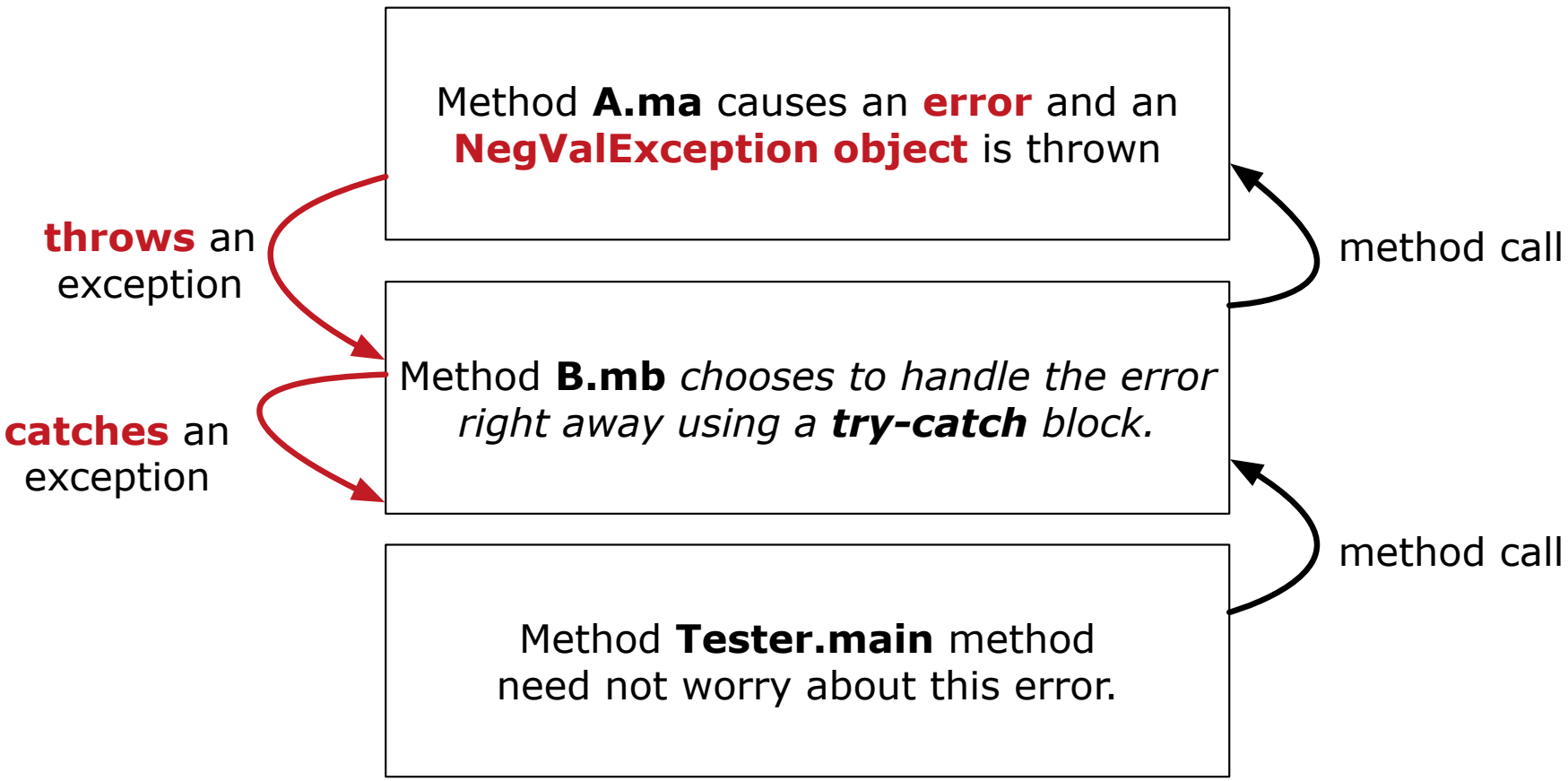
```
class A {
  ma(int i) throws NegValException {
    if(i < 0) { throw new NegValException("Error."); }
    else { /* Do something. */ }
  } }
```

```
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. */
  } }
```

Example: to Handle or Not to Handle? (2.2)

Version 1: Handle the exception in `B.mb`.



Example: to Handle or Not to Handle? (3.1)

Version 2: Handle the exception in `Tester.main`.

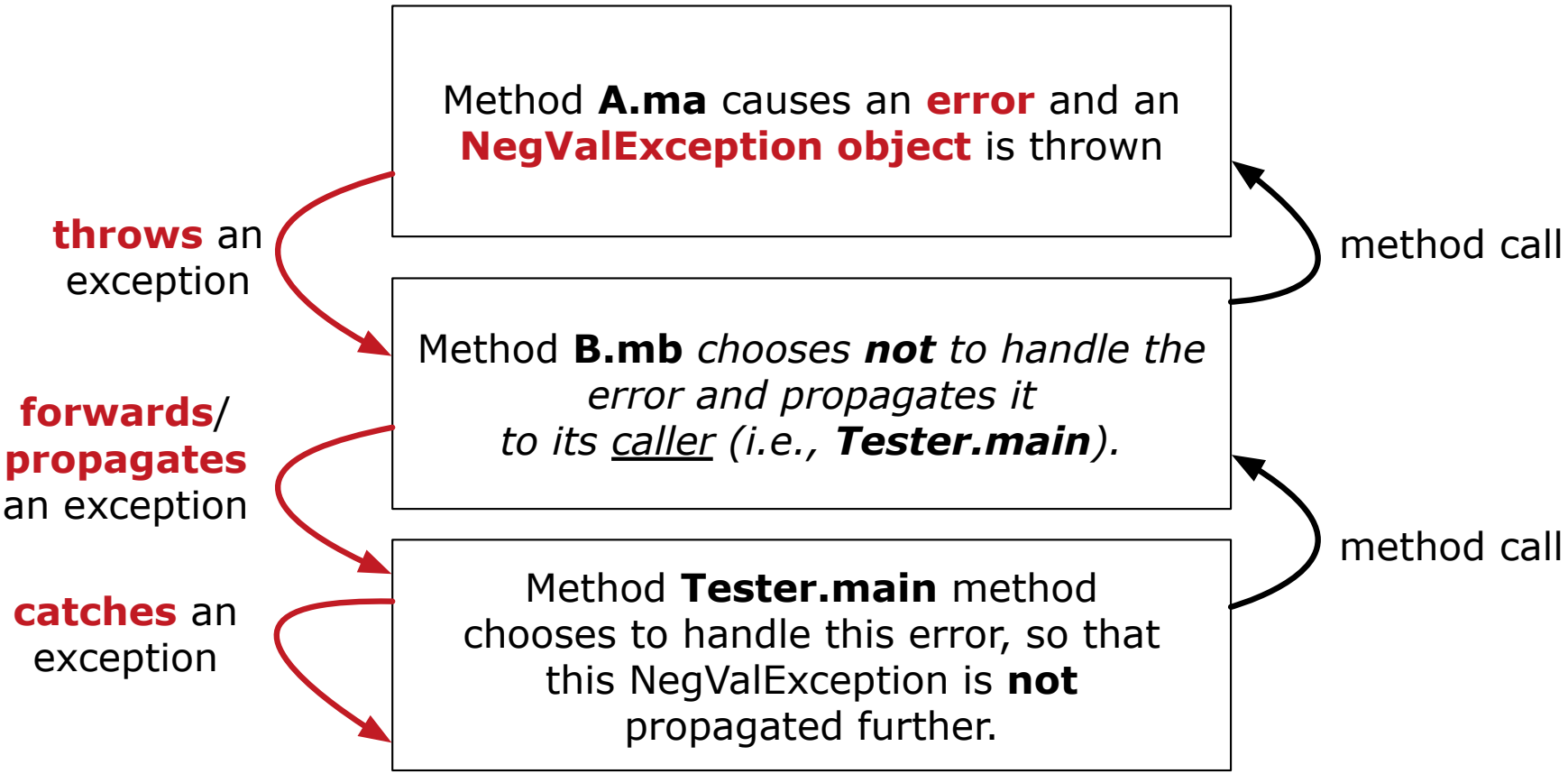
```
class A {
  ma(int i) throws NegValException {
    if(i < 0) { throw new NegValException("Error."); }
    else { /* Do something. */ }
  } }
```

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

```
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. */ }
  } }
```

Example: to Handle or Not to Handle? (3.2)

Version 2: Handle the exception in `Tester.main`.



Example: to Handle or Not to Handle? (4.1)

Version 3: Handle in neither of the classes.

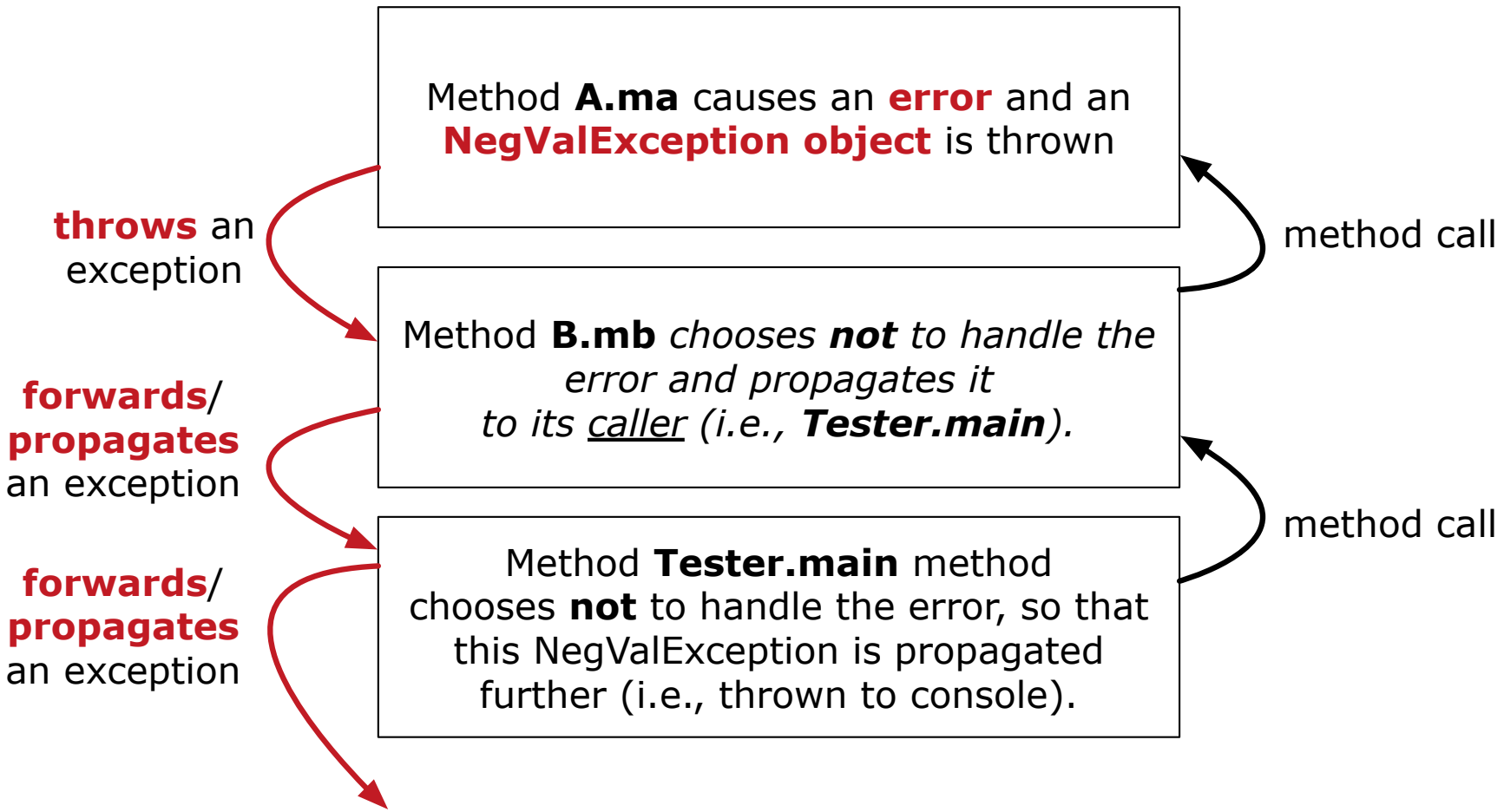
```
class A {
  ma(int i) throws NegValException {
    if(i < 0) { throw new NegValException("Error."); }
    else { /* Do something. */ }
  } }
```

```
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);
  } }
```

Example: to Handle or Not to Handle? (4.2)

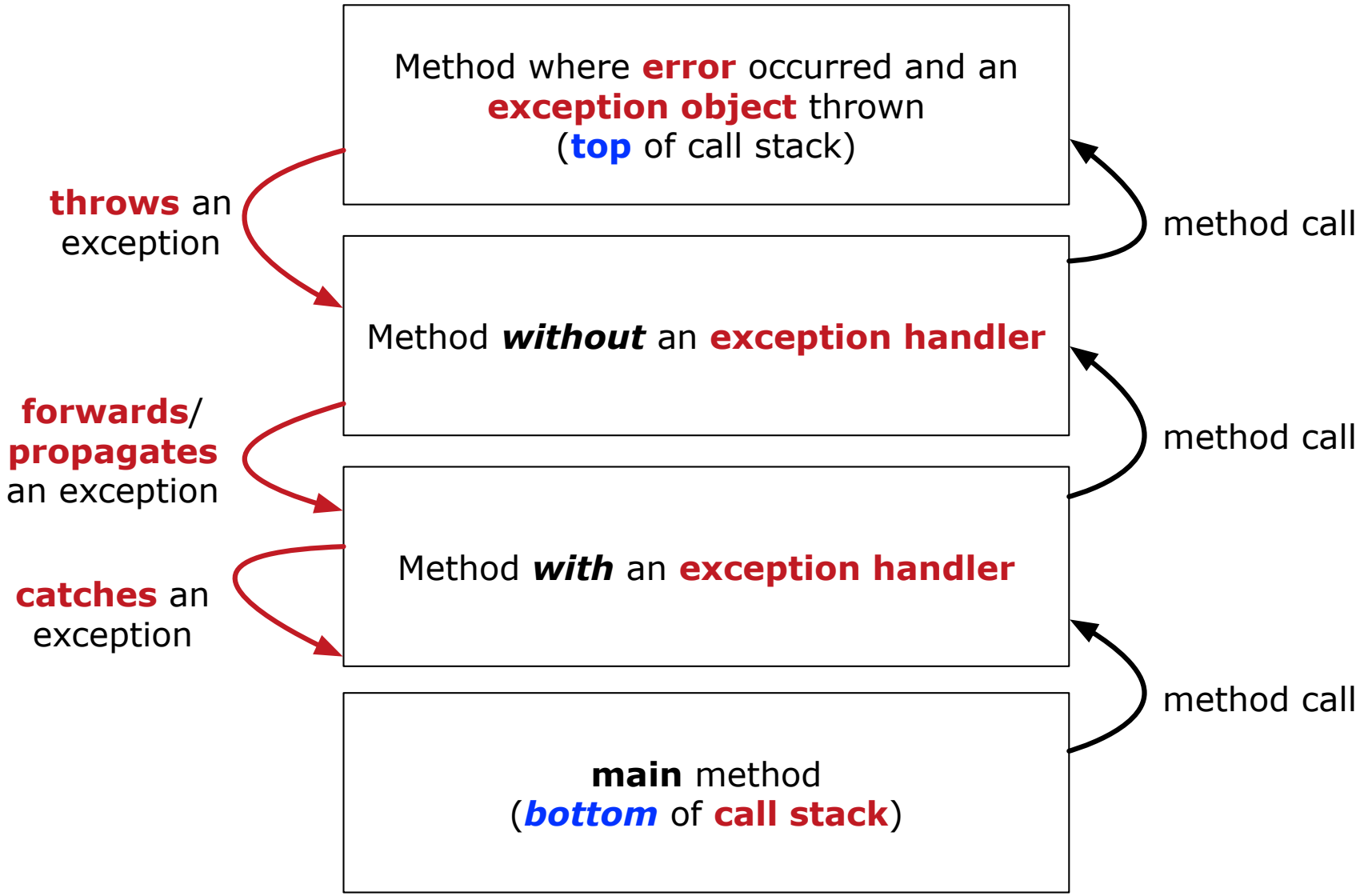
Version 3: Handle in neither of the classes.



Stack of Method Calls

- Execution of a Java project *starts* from the *`main` method* of some class (e.g., `CircleTester`, `BankApplication`).
- Each line of *method call* involves the execution of that method's *body implementation*
  - That method's body implementation may also involve *method calls*, which may in turn involve more *method calls*, and *etc*.
  - It is typical that we end up with *a chain of method calls* !
  - We call this chain of method calls a *call stack* . For example:
    - `Account.withdraw` [*top* of stack; *latest* called]
    - `Bank.withdrawFrom`
    - `BankApplication.main` [*bottom* of stack; *earliest* called]
  - The closer a method is to the *top* of the call stack, the *later* its call was made.

What to Do When an Exception Is Thrown? (1)



What to Do When an Exception Is Thrown? (2)

- 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!

The Catch or Specify Requirement (1)

Code (e.g., a method call) that might throw certain exceptions must be enclosed by one of the two ways:

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 Catch or Specify Requirement (2)

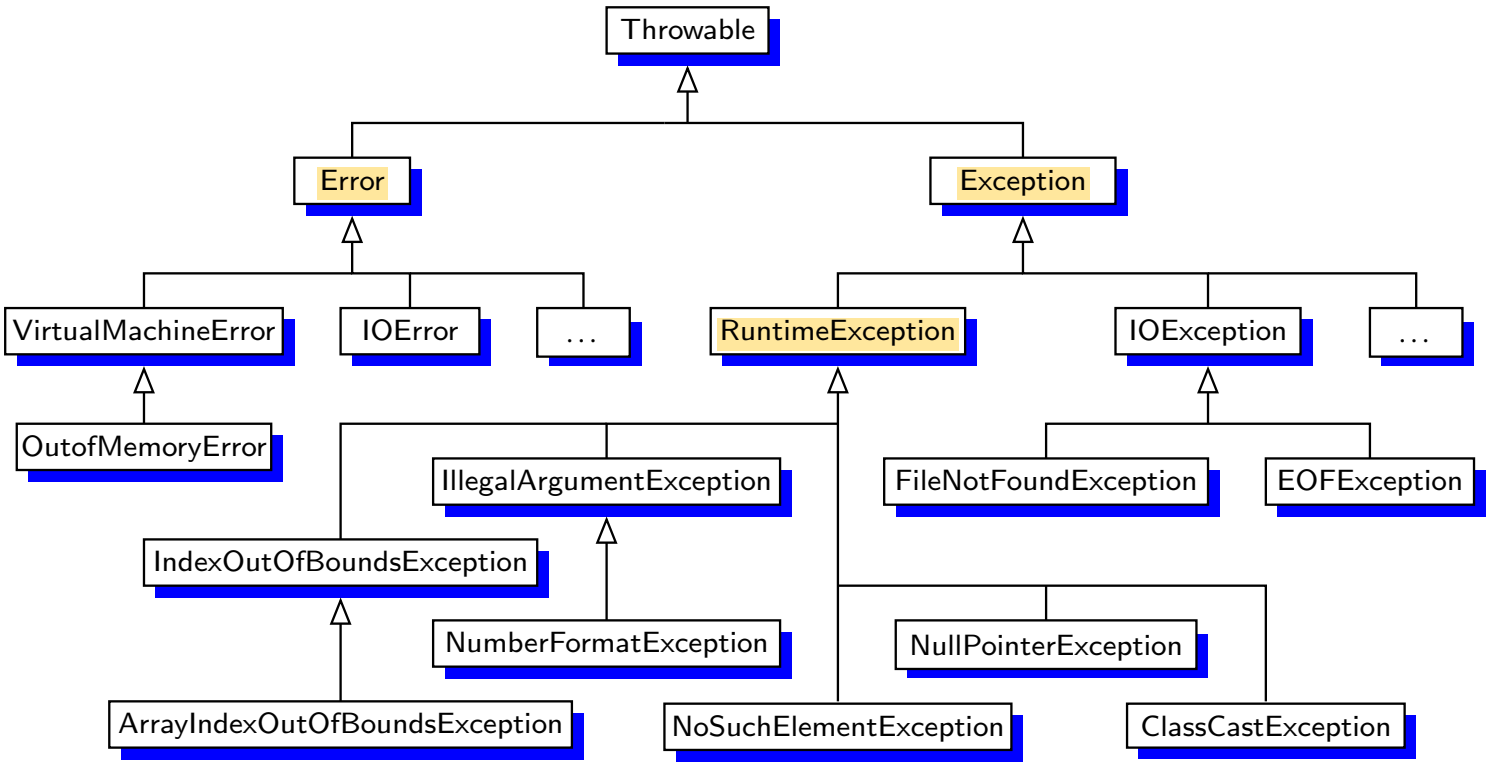
Code (e.g., a method call) that might throw certain exceptions must be enclosed by one of the two ways:

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

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

The Catch or Specify Requirement (3)

There are *three* basic categories of exceptions



Only one category of exceptions is subject to the *Catch or Specify Requirement*.

Exception Category (1): Checked Exceptions

- *Checked exceptions* are exceptional conditions that a well-written application should anticipate and recover from.
  - An application prompts a user for a circle radius, a deposit/withdraw amount, or the name of a file to open.
  - *Normally*, the user enters a positive number for radius/deposit, a not-too-big positive number for withdraw, and existing file to open.
  - When the user enters invalid numbers or file names, `NegativeRadiusException`, `InvalidTransactionException`, or `FileNotFoundException` is thrown.
  - A well-written program will *catch* this exception and notify the user of the mistake.
- *Checked exceptions* are:
  - subject to the *Catch or Specify Requirement*.
  - subclasses of `Exception` that are **not** *descendant classes* of `RuntimeException`.

Exception Category (2): Errors

- *Errors* are exceptional conditions that are *external* to the application, and that the application usually cannot anticipate or recover from.
  - An application successfully opens a file for input.
  - But the file cannot be read because of a hardware or system malfunction.
  - The unsuccessful read will throw `java.io.IOError`
- *Errors* are:
  - *not* subject to the *Catch or Specify Requirement*.
  - subclasses of `Error`

Exception Category (3): Runtime Exceptions

- *Runtime exceptions* are exceptional conditions that are *internal* to the application, and that the application usually cannot anticipate or recover from.
  - These usually indicate programming bugs, such as logic errors or improper use of an API.

  e.g., `NullPointerException`

  e.g., `ClassCastException`

  e.g., `ArrayIndexOutOfBoundException`
- *Runtime exceptions* are:
  - *not* subject to the *Catch or Specify Requirement*.
  - subclasses of `RuntimeException`
- *Errors* and *Runtime exceptions* are collectively known as *unchecked exceptions*.

Catching and Handling Exceptions

- To construct an *exception handler*:
  1. Enclose the code that might throw an exception within a `try` block.
  2. Associate *each possible kind of exception* that might occur within the `try` block with a `catch` block.
  3. Append an optional `finally` block.

```
try { /* code that might throw exceptions */ }
catch(ExceptionType1 e) { ... }
catch(ExceptionType2 e) { ... }
...
finally { ... }
```

- When an exception is thrown from Line *i* in the *try* block:
  - Normal flow of execution is *interrupted*: the rest of *try* block starting from Line *i* + 1 is skipped.
  - Each *catch* block performs an `instanceof` check on the thrown exception: the first matched *catch* block is executed.
  - The *finally* block is always executed after the matched `catch` block is executed.

Examples (3)

```
double r = ...;
double a = ...;
try{
  Bank b = new Bank();
  b.addAccount(new Account(34));
  b.deposit(34, a)
  Circle c = new Circle();
  c.setRadius(r);
  System.out.println(r.getArea());
}
catch(NegativeRadiusException e) {
  System.out.println(r + " is not a valid radius value.");
  e.printStackTrace();
}
catch(InvalidTransactionException e) {
  System.out.println(r + " is not a valid transaction value.");
  e.printStackTrace();
}
catch( Exception e) { /* any other kinds of exceptions */
  e.printStackTrace();
}
```

Examples (4): Problem?

```
double r = ...; double a = ...;
try{
  Bank b = new Bank();
  b.addAccount(new Account(34));
  b.deposit(34, 100);
  b.withdraw(34, a);
  Circle c = new Circle();
  c.setRadius(r);
  System.out.println(r.getArea());
}
/* Every exception object is a descendant of Exception.  */
catch( Exception e) {
  e.printStackTrace();
}
catch(NegativeRadiusException e) { /* Problem:  Not reachable!  */
  System.out.println(r + " is not a valid radius value.");
  e.printStackTrace();
}
catch(InvalidTransactionException e) { /* Problem:  Not reachable!  */
  System.out.println(r + " is not a valid transaction value.");
  e.printStackTrace();
}
```

Index (1)

Index (2)

Index (3)