Exception Handling

Objectives

- After you have read and studied this chapter, you should be able to
  - Improve the reliability of code by incorporating exception-handling.
  - Implement the try-catch blocks for catching and handling exceptions.
  - Write methods that propagate exceptions.
  - Distinguish the checked and unchecked exceptions.
  - Define Custom Exception class

Definition

- An *exception* represents an error condition that can occur during the normal course of program execution.
- Examples
  - Division by zero
  - Trying to open an input file that does not exist
  - An array index that goes out of bounds

Java's Mechanism of Exception Handling

- A Java exception is an object that describes an exceptional (error) condition has occurred in a piece of code.
- When an exceptional condition occur, an object representing that exception is created and *thrown in the method that caused the error.*
- The exception-handling codes can then be executed to catch & handle the exception object; we say the thrown exception object is *caught*.
- The normal sequence of flow is terminated if the exception object is thrown and not caught.

Exception thrown (not caught)

```
//somewhere in method main()
Scanner scan = new Scanner(System.in);
int  num = 0;
System.out.print("Enter an integer>");
num = scan.nextInt();
System.out.println("The number you entered is " +num);
```

Error (stack trace) message for invalid input

```
Enter an integer> three
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Unknown Source)
        at java.util.Scanner.next(Unknown Source)
        at java.util.Scanner.nextInt(Unknown Source)
        at java.util.Scanner.nextInt(Unknown Source)
        at ExceptionTest.main(ExceptionTest.java:11)
```

Java's Mechanism of Exception Handling

- **(**From the example) The execution of **main() stops** because once an exception has been thrown, it must be *caught by an exception handler and* dealt with immediately.
- If we have no exception handlers of our own, the exception is caught the Java run-time system.
- The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program.

Java's Mechanism of Exception Handling (continued)

- Java provides a number of exception classes to effectively represent certain common exceptions such as division by zero, invalid input, and file not found

Example:

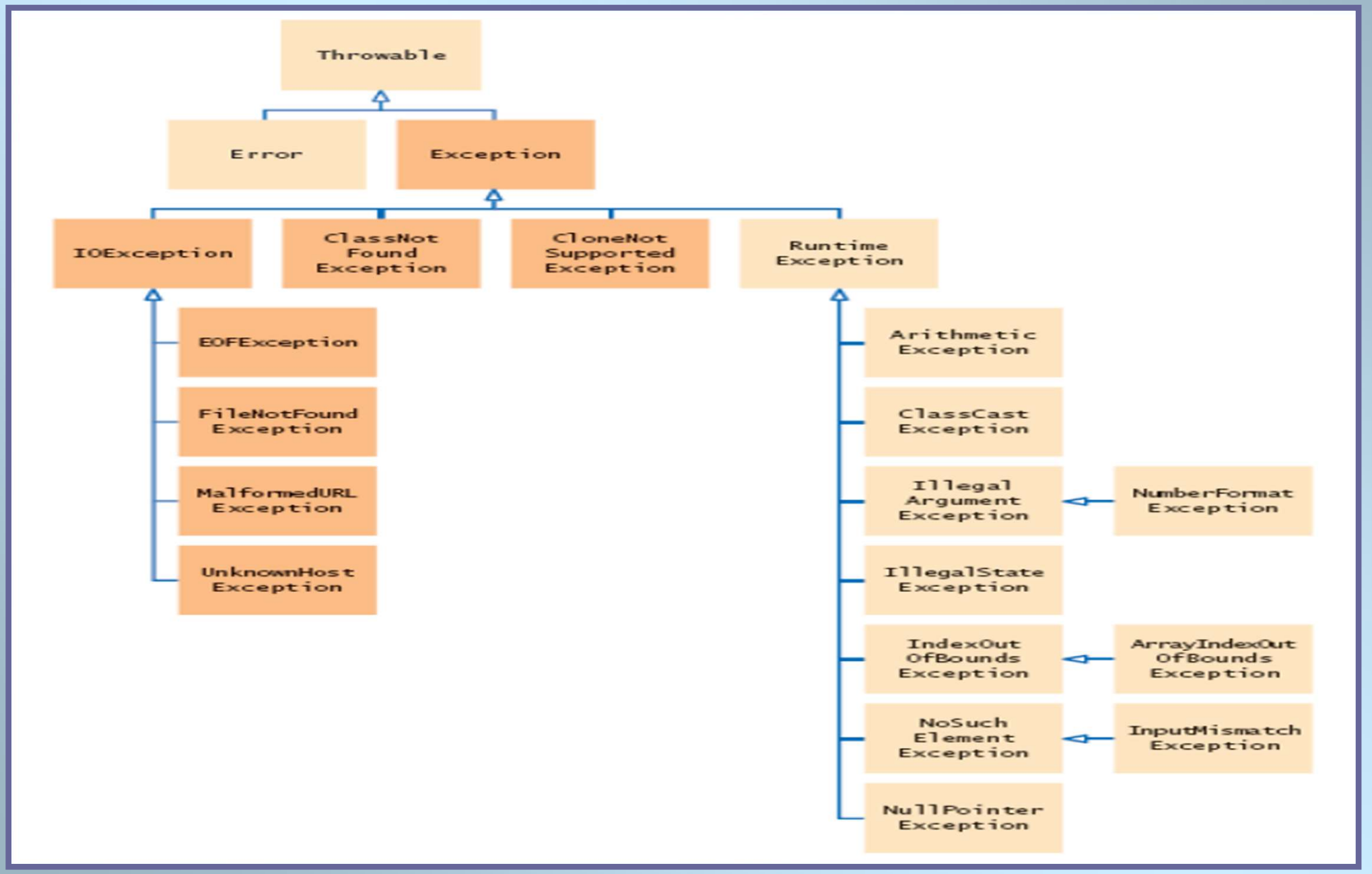
- When a `Scanner` object is used to input data into a program, any invalid input errors are handled using the `class InputMismatchException`
- When a division by zero exception occurs, the program creates an object of the `class ArithmeticException`

Exception Types

- All types of thrown exception objects are instances of the **Throwable** class or its subclasses.
- **Throwable is at the** top of the exception class hierarchy. Immediately below **Throwable are two subclasses, Error & Exception)** that partition exceptions into two distinct branches.
- Serious errors are represented by instances of the **Error** class or its subclasses (Note: we don't deal with **Error** type**, because these are typically created in response to catastrophic** failures that cannot usually be handled by your program.).
- Exceptional cases that common applications should handle are represented by instances of the **Exception** class or its subclasses.

Throwable Hierarchy

- There are over 60 classes in the hierarchy.



Java's Mechanism of Exception Handling (continued)

- Java exception handling is managed via five keywords:
  - **try,**
  - **catch,**
  - **throw,**
  - **throws,**
  - **finally.**

Catching an Exception

Use try{} and catch{} block

To catch an exception:

- Put code that might throw an exception inside a **try{}** block.
- Put code that handles the exception inside a **catch{}** block.

```
try {

  // statements, some of which might
  // throw an exception
}
catch ( SomeExceptionType ex ) {
  // statements to handle this
  // type of exception
}
```

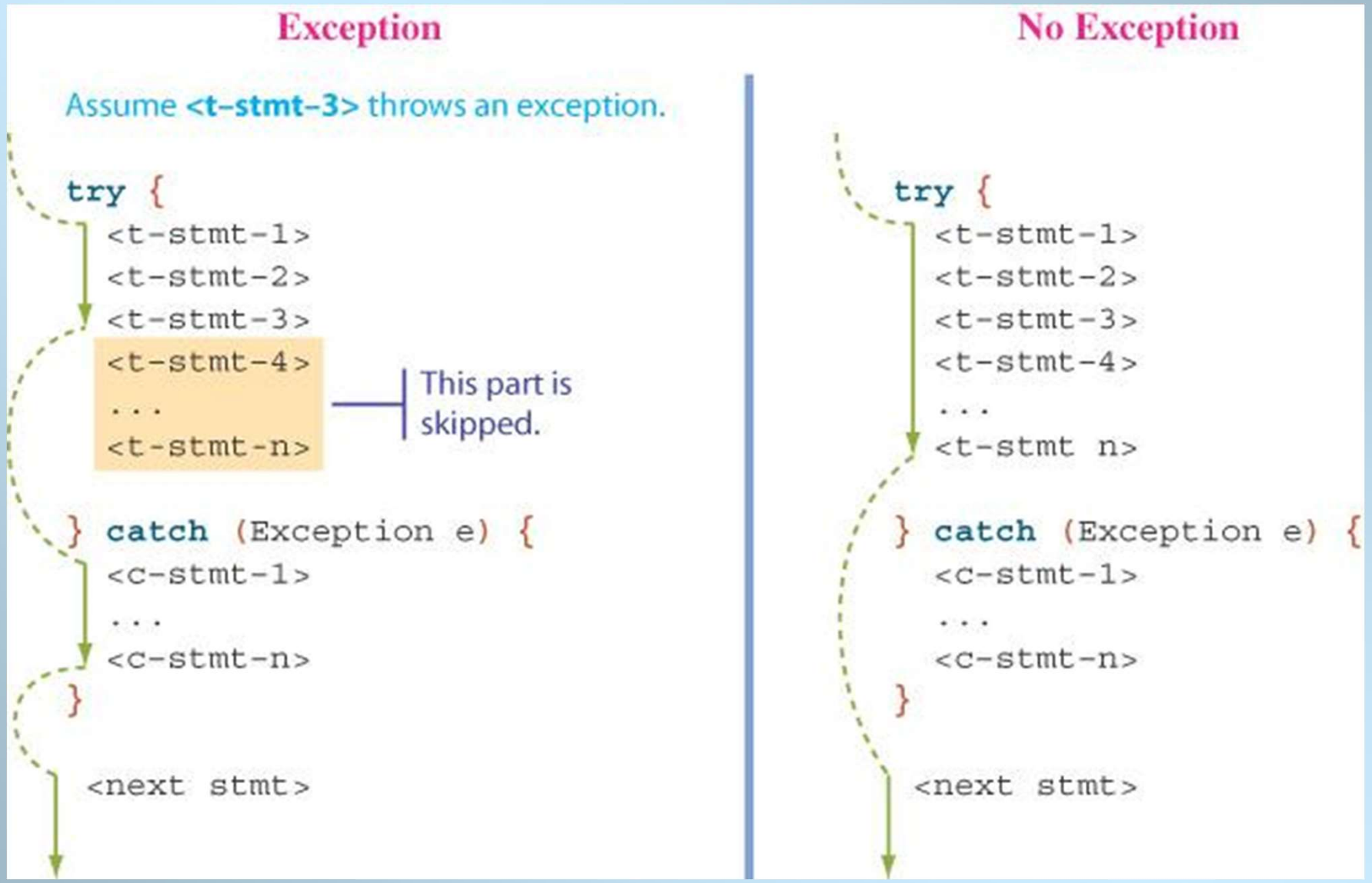
Catching an Exception

```
Scanner scan = new Scanner(System.in);
int  num = 0;
try {
    System.out.print("Enter an integer>");
        num = scan.nextInt();

} catch (InputMismatchException e){

  System.out.println("Invalid input! Please enter
digits only");
}
System.out.println("The number you entered is " +num);
```

try

catch

try-catch Control Flow



Getting Information

- There are two methods we can call to get information about the thrown exception:
  - **getMessage**
  - **printStackTrace**

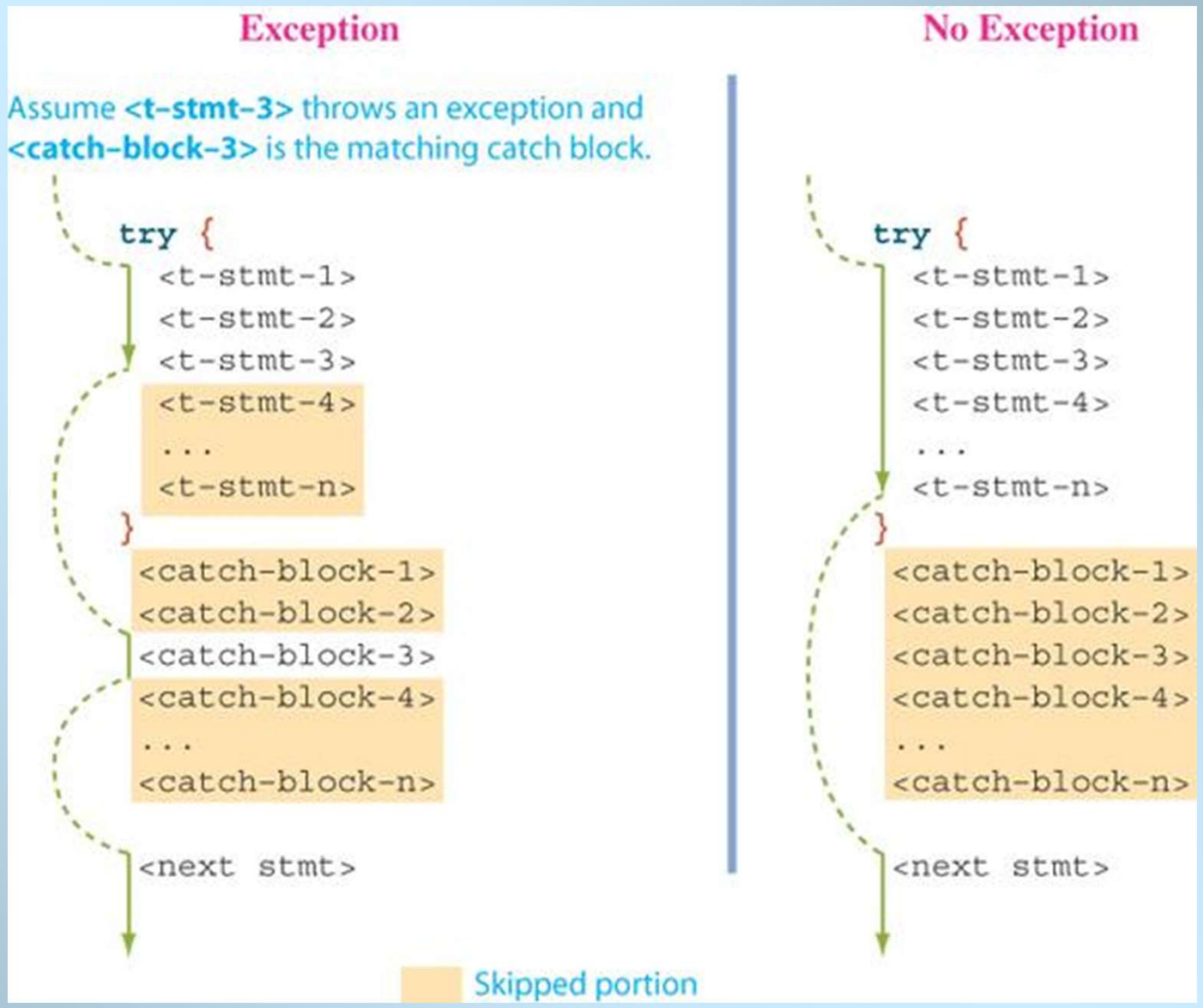
```
try {
        . . .
} catch (NumberFormatException e){

        System.out.println(e.getMessage());
        System.out.println(e.printStackTrace());
}
```

Multiple catch Blocks

- A single try-catch statement can include multiple catch blocks, one for each type of exception.

```
try {
        . . .
        age = Integer.parseInt(inputStr);
        . . .
        val = cal.get(id);
        . . .
} catch (NumberFormatException e){
        . . .
} catch (ArrayIndexOutOfBoundsException e){
        . . .
}
```

Multiple catch Control Flow



-Only one catch{} block gets control (the first block that matches the type of the exception).
-If no catch{} block matches the exception, none is picked (just as if there were no try{} block.)

Multiple catch Control Flow

• The **most specific exception types** should appear first in the structure, followed by the **more general exception types**.

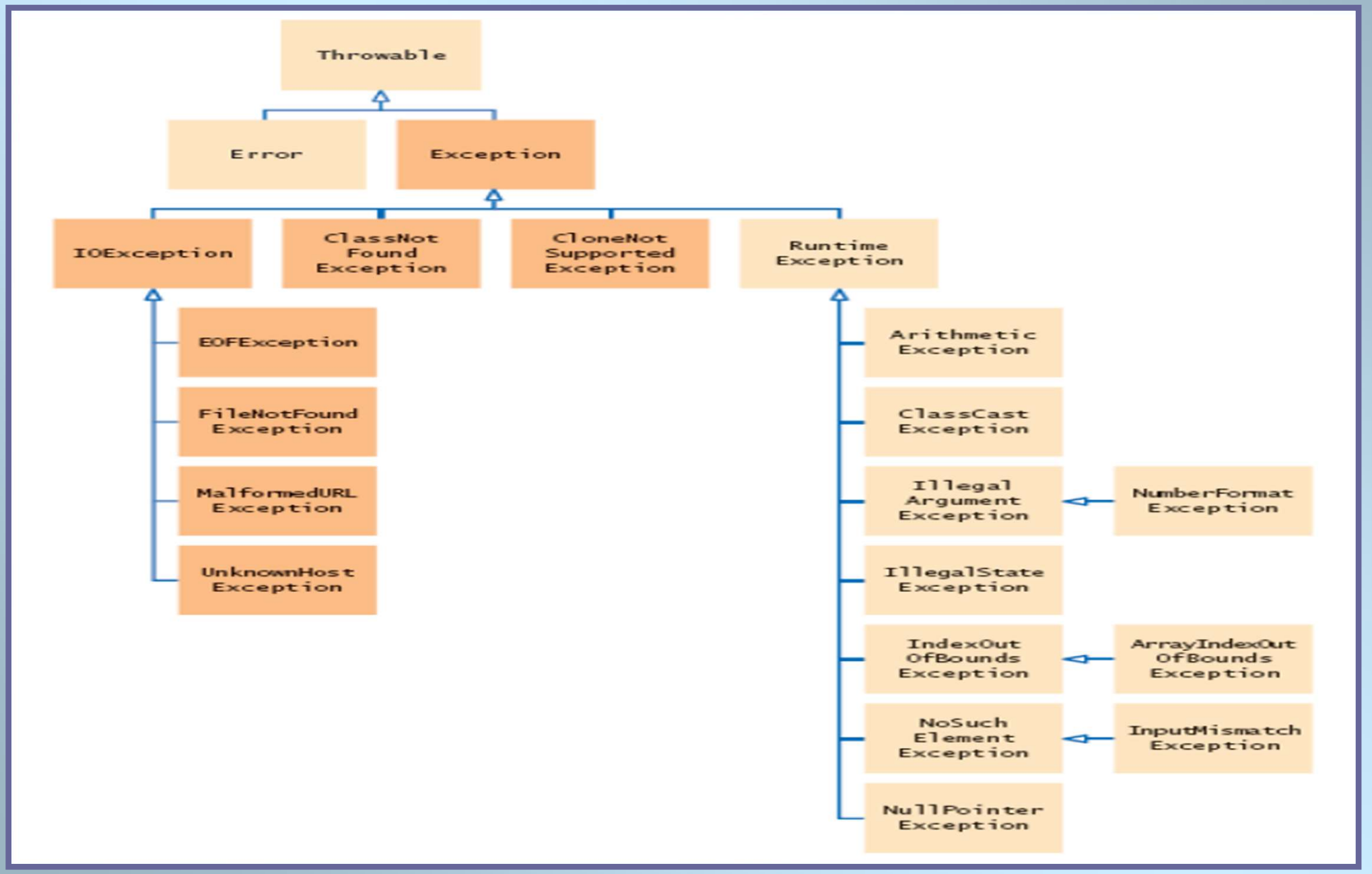
A child class **should appear before** any of its ancestors. If class A is not an ancestor or descendant of class B, then it doesn't matter which appears first.

Eg. the following is wrong:

```
try {
    ...

} catch (Exception ex ) {
   . . .

} catch (ArithmeticException ex ) {
   . . .
}
```

Throwable Hierarchy

- There are over 60 classes in the hierarchy.



Multiple catch Control Flow

Question: Is this ok?:

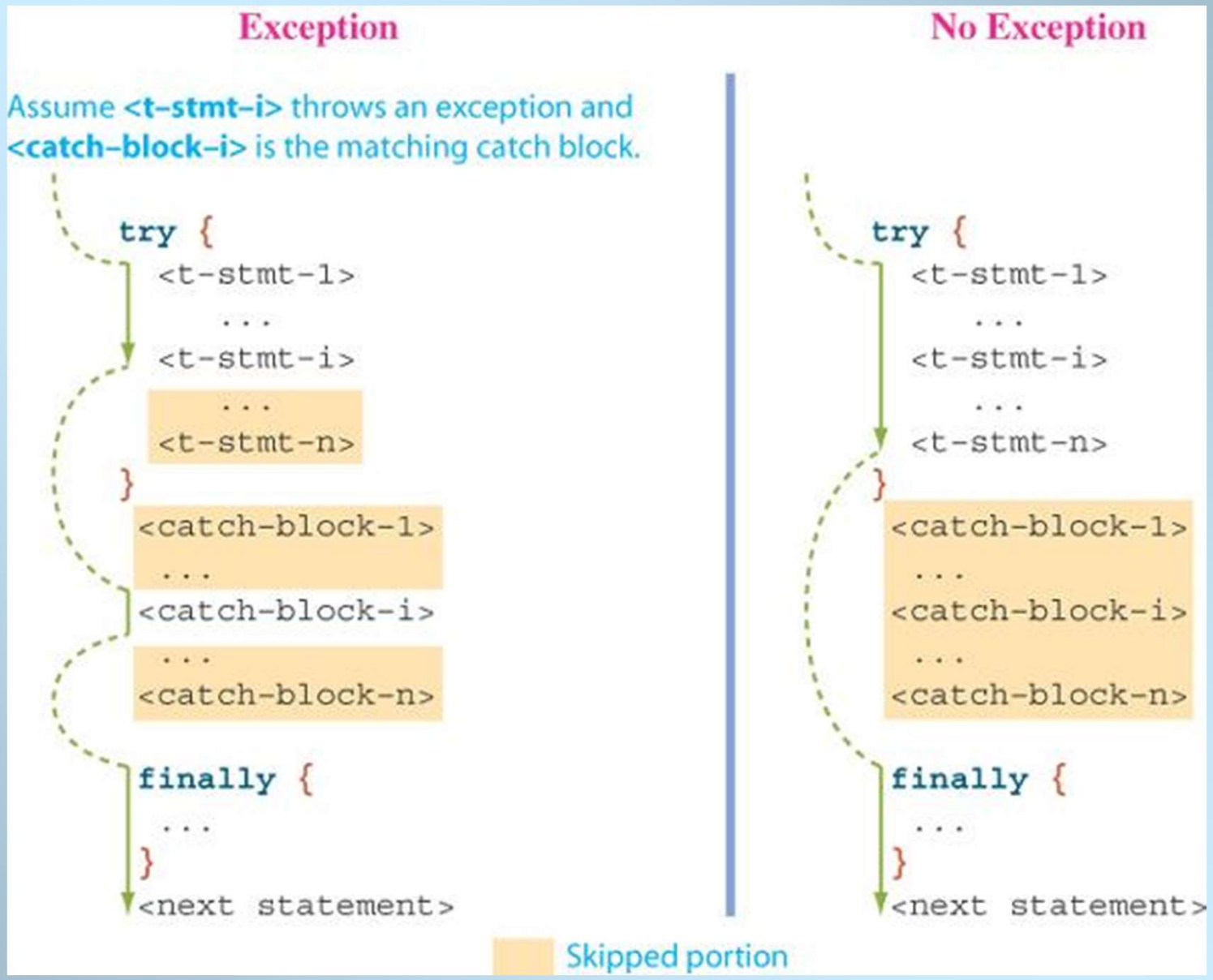
```
try {
    ...

} catch (NumberFormatException ex ) {
   . . .

} catch (ArithmeticException ex ) {
   . . .
}
```

The finally Block

- There are situations where we need to take certain actions regardless of whether an exception is thrown or not.
- We place statements that must be executed regardless of exceptions in the finally block.

```
try {
  // statements, some of which might
  // throw an exception
} catch ( SomeExceptionType ex ) {
 // statements to handle this
 // type of exception
} catch ( AnotherExceptionType ex ) {
  // statements to handle this
  // type of exception
} finally {
 // statements which will execute no matter
 // how the try block was exited.

}
```

try-catch-finally Control Flow



Propagating Exceptions

- Instead of catching a thrown exception by using the try-catch statement, we can propagate the thrown exception back to the caller of our method.
- The method header includes the reserved word throws.

```
public int getAge( ) throws NumberFormatException {
        . . .
        int age = Integer.parseInt(inputStr);
        . . .
        return age;
}
```

Throwing Exceptions

- We can write a method that throws an exception directly, i.e., this method is the origin of the exception.
- Use the throw reserved to create a new instance of the Exception or its subclasses.

```
public void doWork(int num) throws Exception {
        . . .
        if (num != val) throw new Exception("Invalid
val");
        . . .
}
```

Exception Thrower

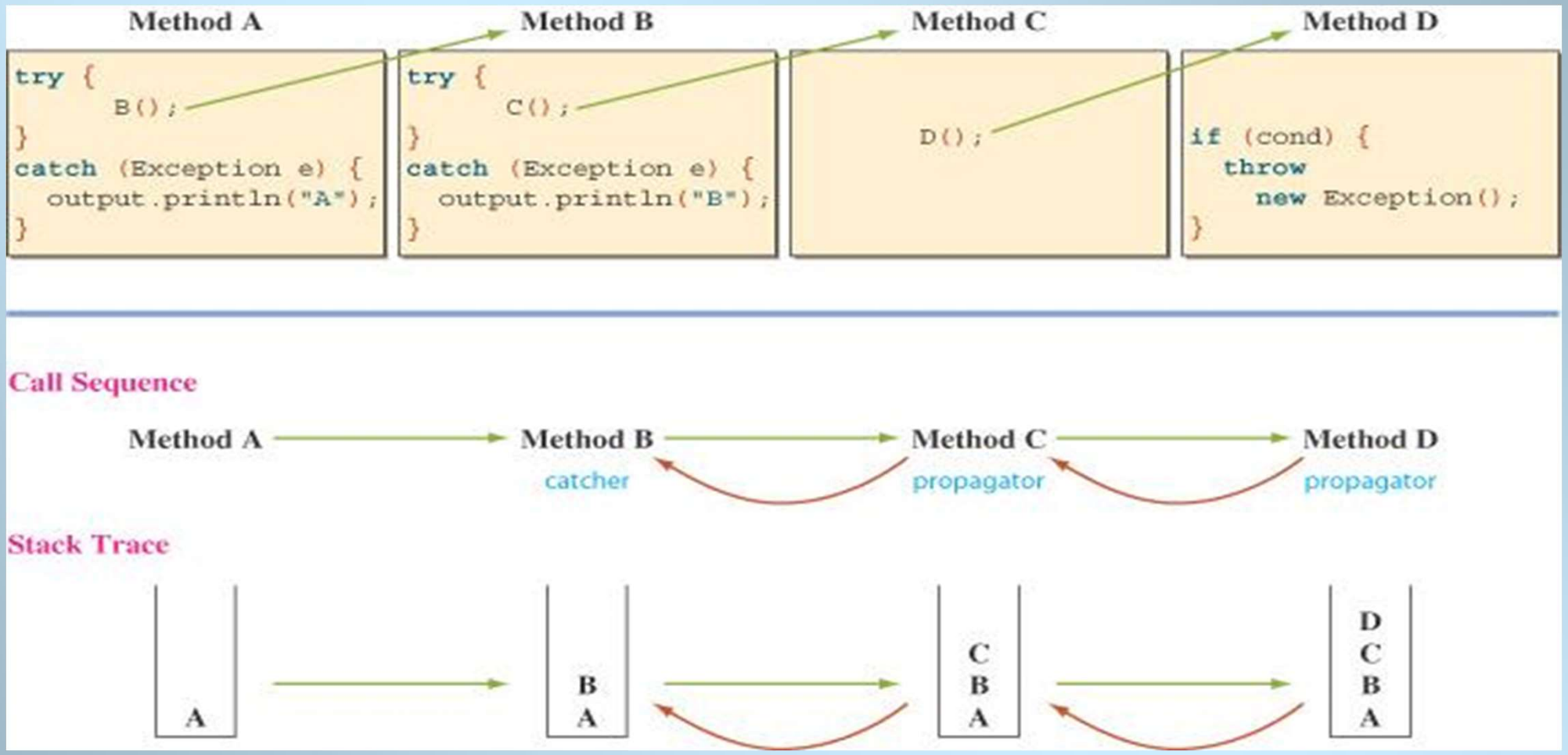
- When a method may throw an exception, either directly or indirectly, we call the method an *exception thrower*.

- Every exception thrower must be one of two types:
  - catcher.
  - propagator.

Types of Exception Throwers

- An *exception catcher* is an exception thrower that includes a matching **catch** block for the thrown exception.
- An *exception propagator* does not contain a matching **catch** block.
- Note: A method may be a catcher of one exception and a propagator of another.

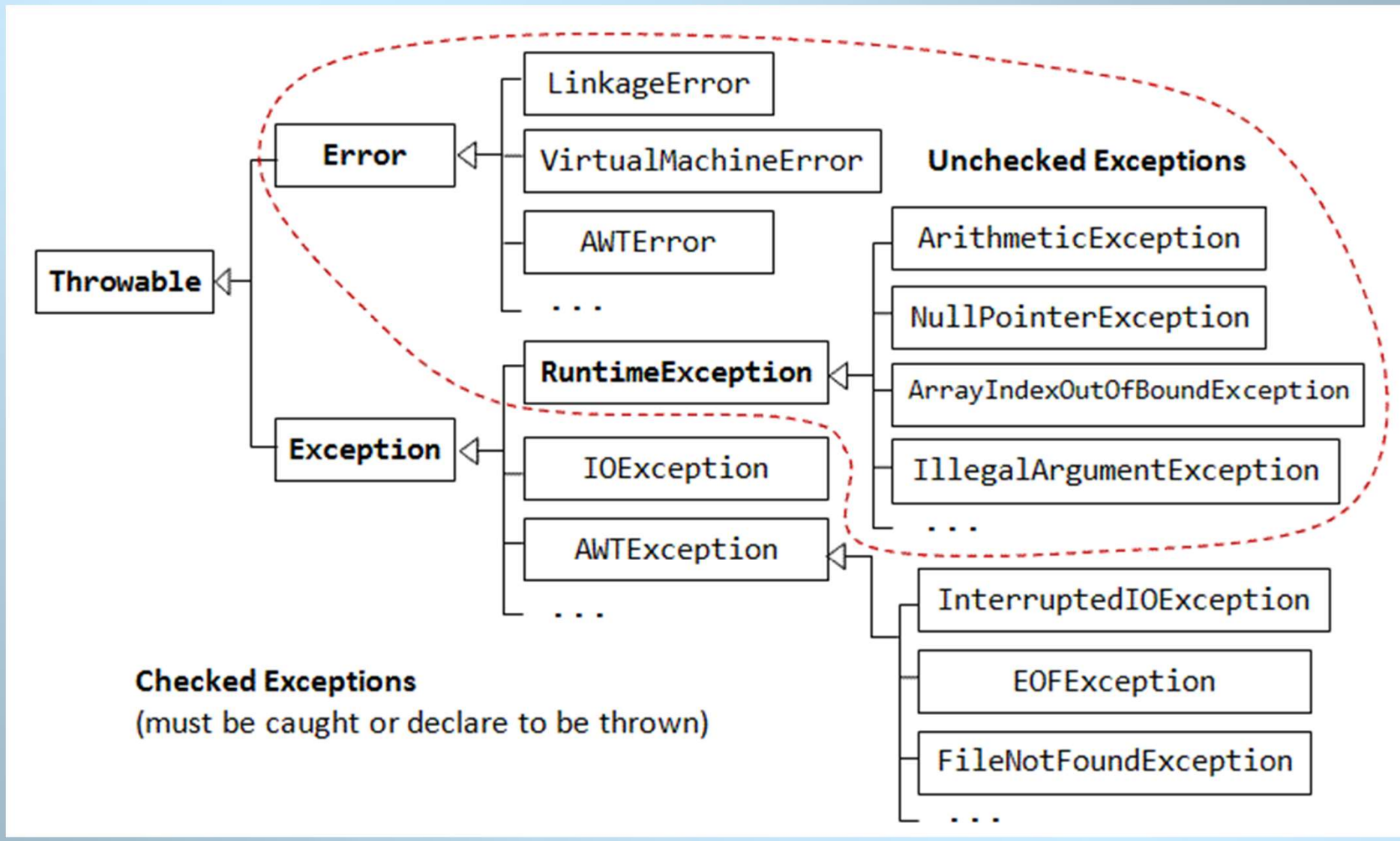
Sample Call Sequence



Checked vs. Unchecked Exception

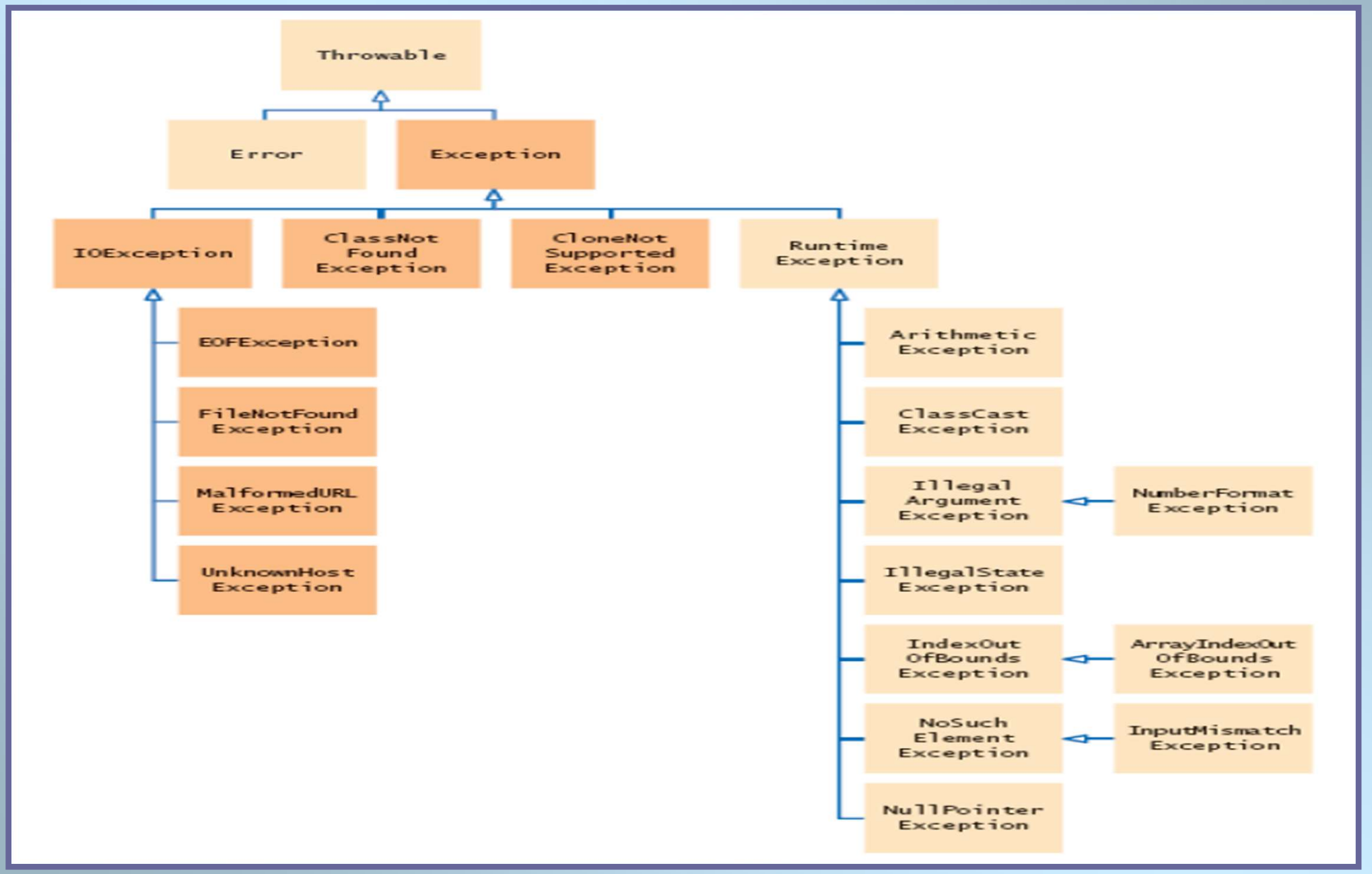
- There are two types of exceptions:
  - Checked.
  - Unchecked.
- A *checked exception* is an exception that is checked at compile time, so it must be handled.
- An *unchecked exception* is not checked at compile time, so no need to handle (optional).
- *unchecked* exception include Error and RuntimeException and their subclasses.
- *checked* exception include Exception and its subclasses.

Checked vs. Unchecked Exception



Throwable Hierarchy

- There are over 60 classes in the hierarchy.

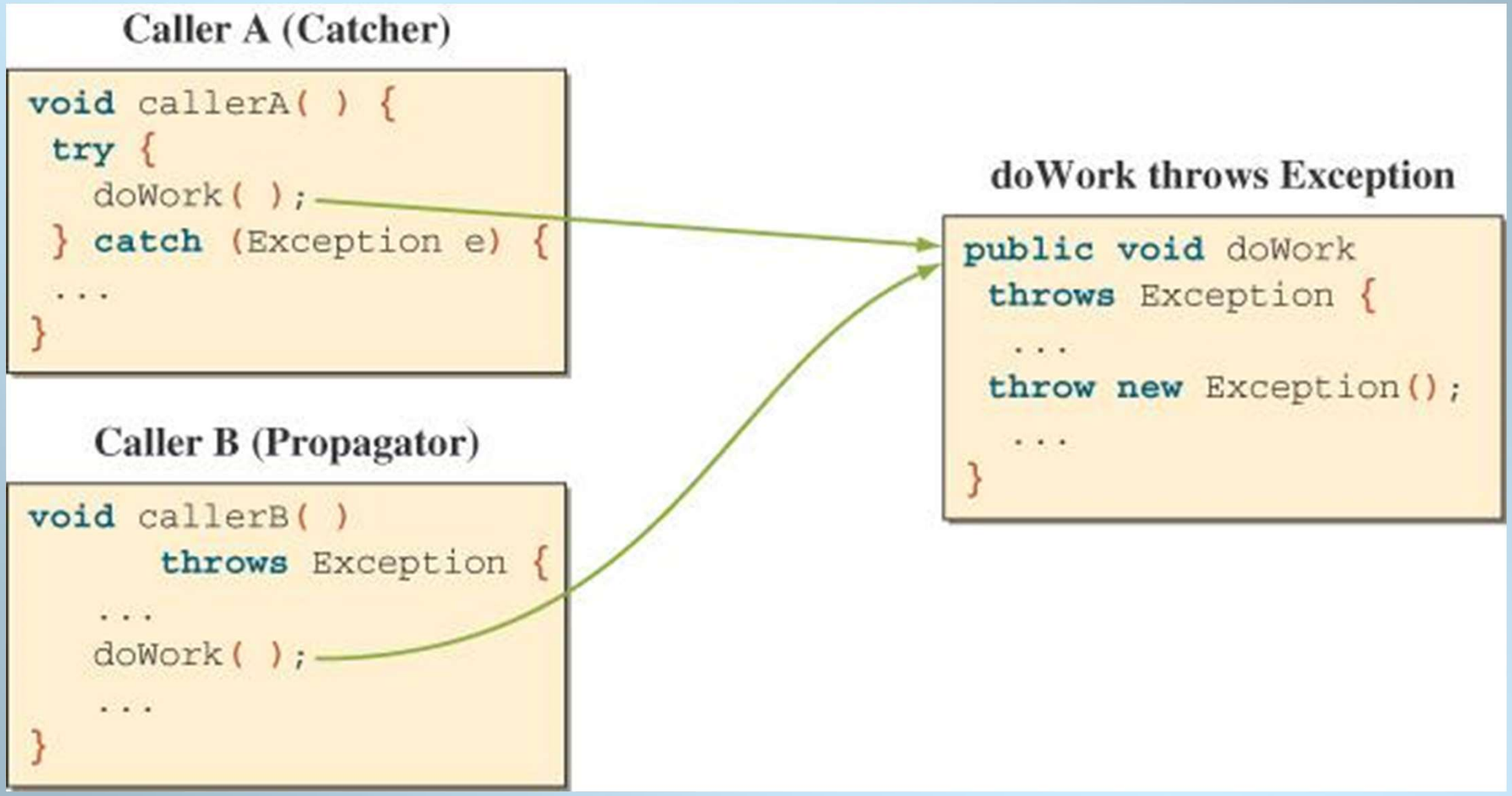


Different Handling Rules

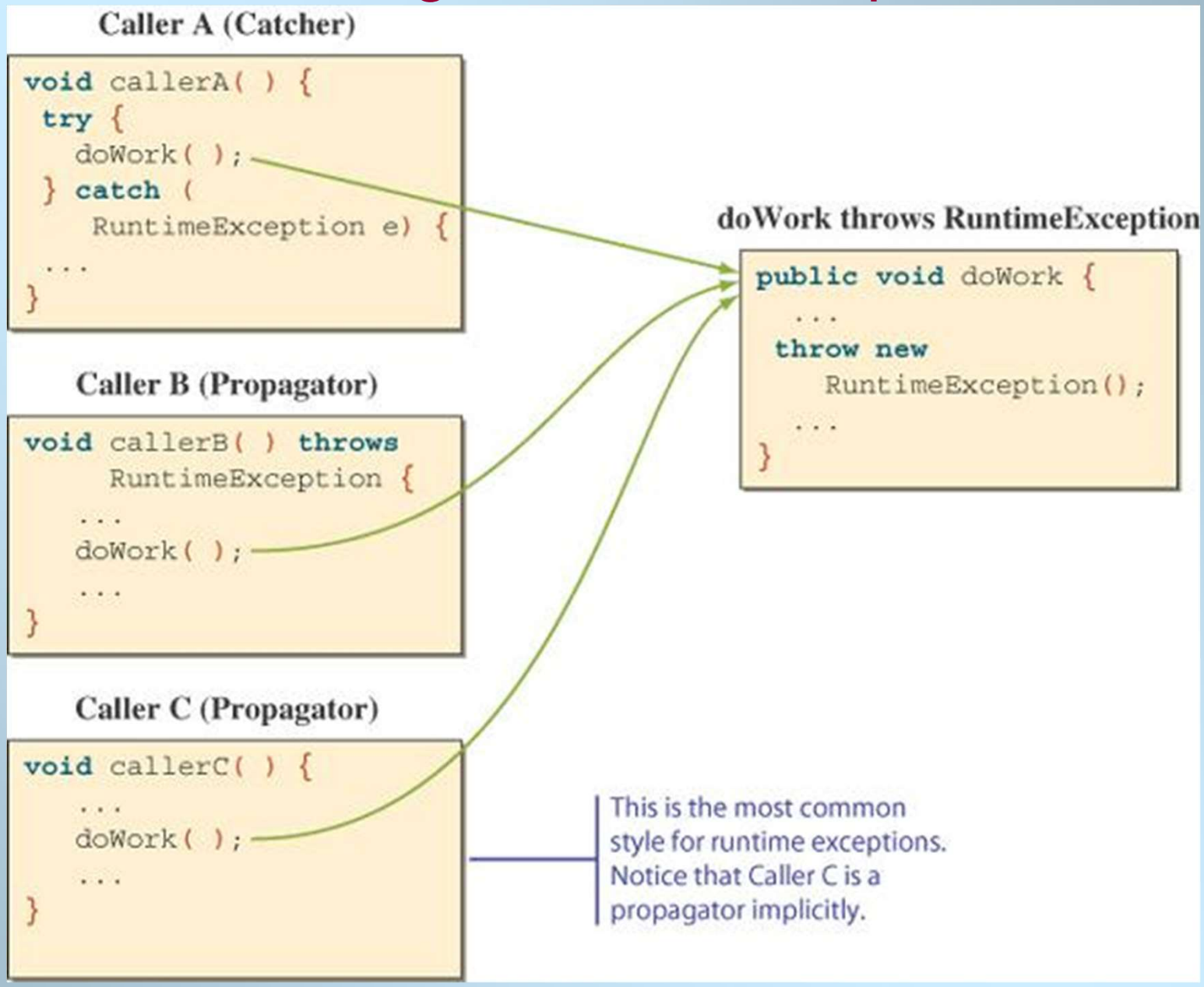
- When calling a method that can throw checked exceptions
  - use the **try-catch** statement and place the call in the **try** block, or (handle the exception)

    OR
  - modify the method header to include the appropriate **throws** clause. (declare the exception)
- When calling a method that can throw unchecked exceptions, it is optional to use the try-catch statement or modify the method header to include a throws clause.

Handling Checked Exceptions



Handling Runtime Exceptions



Defining Custom Exception Classes

- Use the exception classes in the API whenever possible.

- Define custom exception classes if the predefined classes are not sufficient.

- Define custom exception classes by extending Exception or a subclass of Exception.

Defining Custom Exception Classes

Example: Class **InvalidRadiusException** is defined as a subclass of class **Exception**:

```
public class InvalidRadiusException extends Exception
 {
        private double radius;
        /** Construct an exception */
        public InvalidRadiusException(double radius) {
                super("Invalid radius " + radius);
                 this.radius = radius;
         }

        /** Return the radius */
        public double getRadius() {
                return radius;
         }
}
```