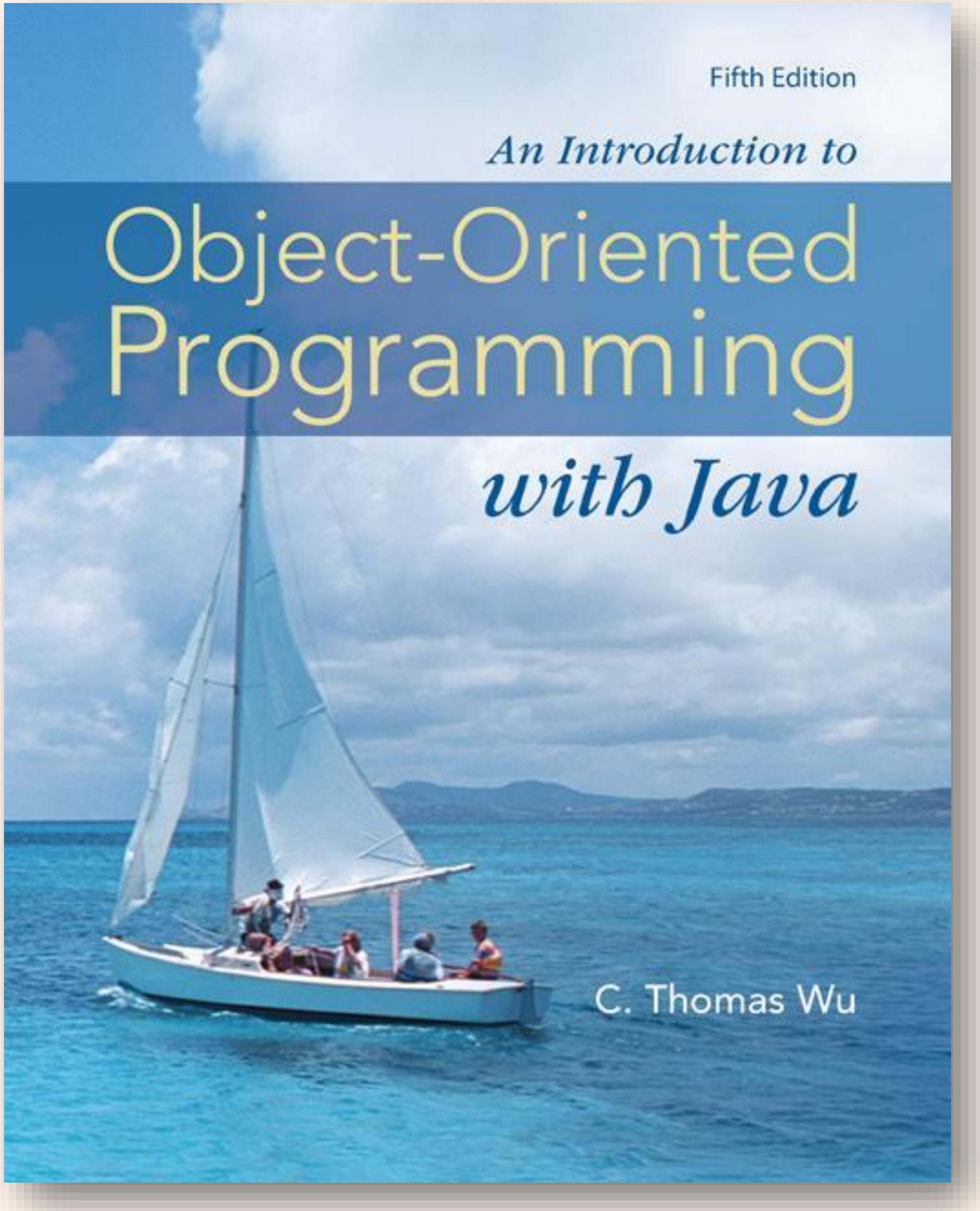


Chapter 8

Exceptions and Assertions

Animated Version

Objectives

- After you have read and studied this chapter, you should be able to
  - Improve the reliability of code by incorporating exception-handling and assertion mechanisms.
  - Write methods that propagate exceptions.
  - Implement the try-catch blocks for catching and handling exceptions.
  - Write programmer-defined exception classes.
  - Distinguish the checked and unchecked, or runtime, exceptions.

Definition

- An *exception* represents an error condition that can occur during the normal course of program execution.
- When an exception occurs, or is *thrown*, the normal sequence of flow is terminated. The exception-handling routine is then executed; we say the thrown exception is *caught*.

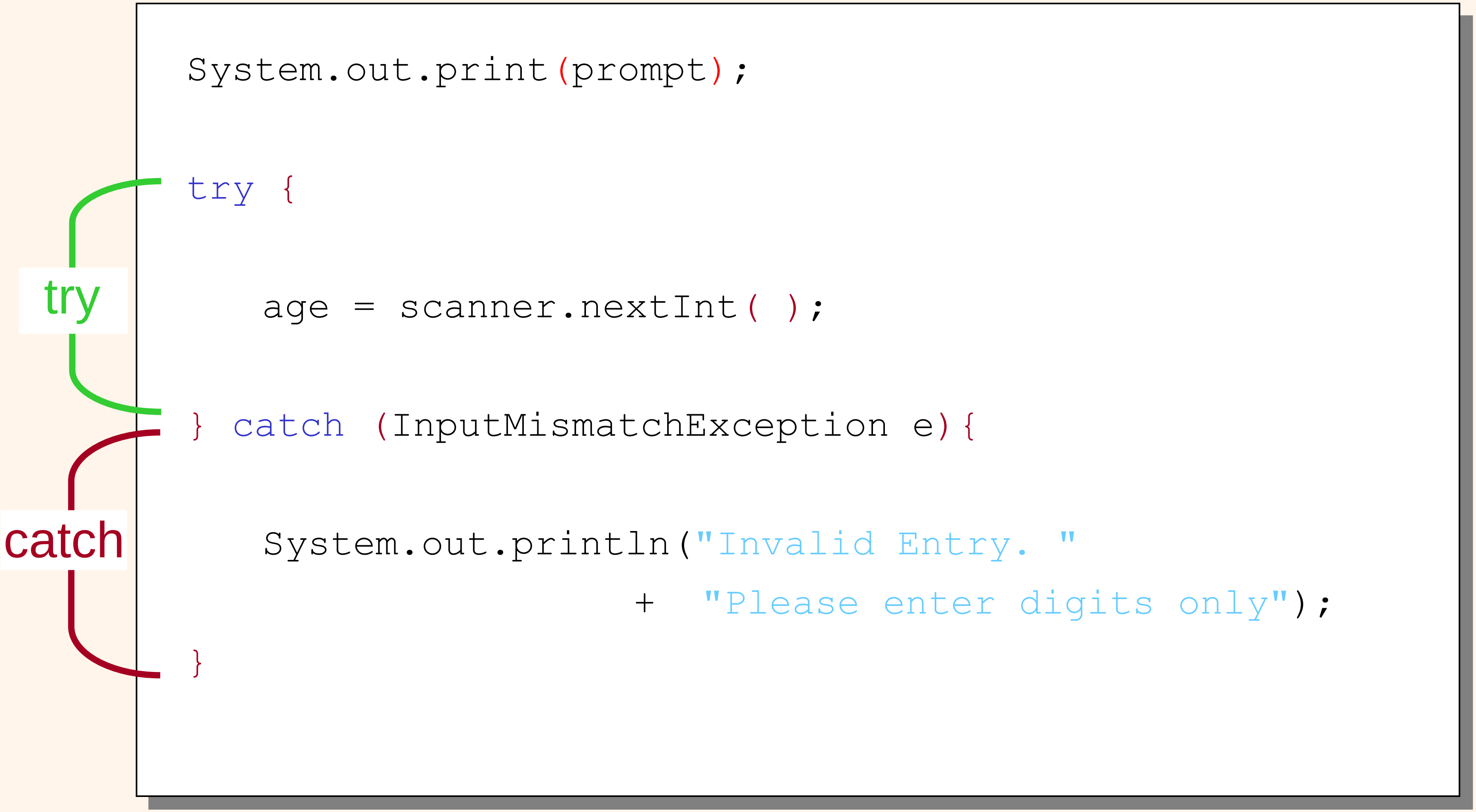
Not Catching Exceptions

```
Scanner scanner = new Scanner(System.in);
System.out.println(“Enter integer:");
int number = scanner.nextInt();
```

Error message for invalid input

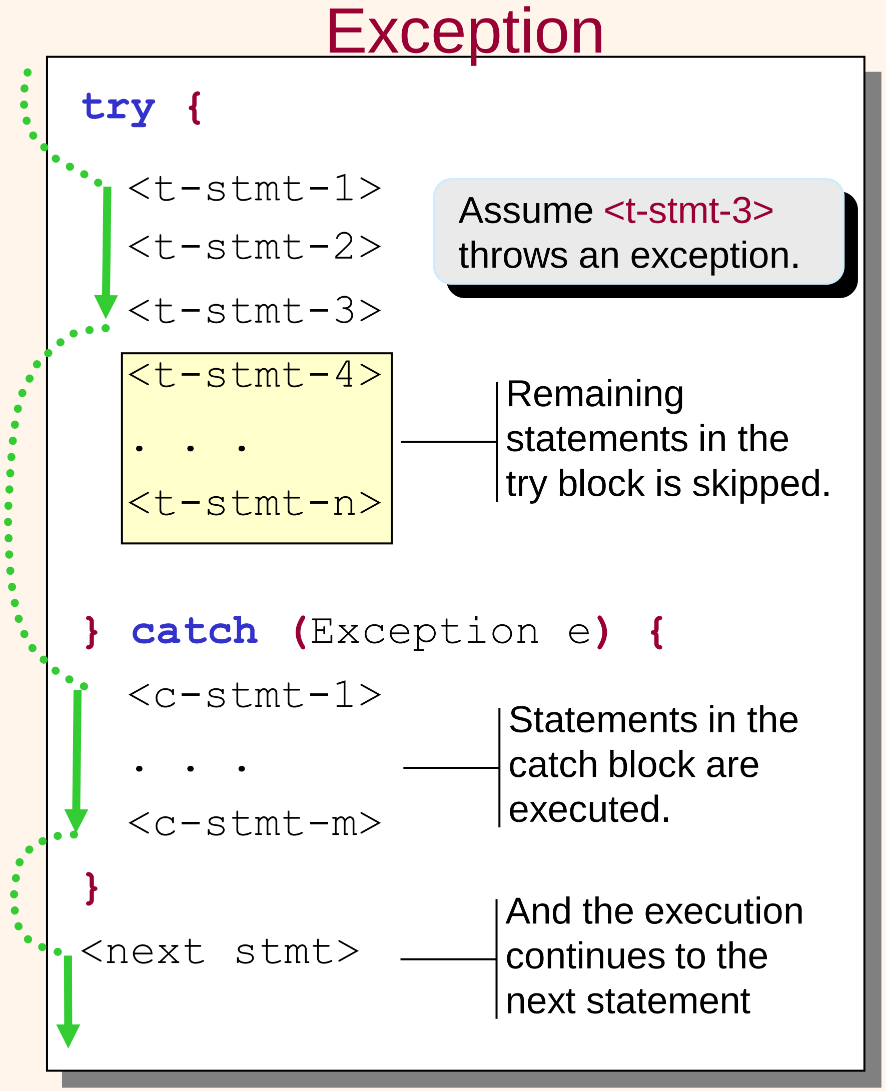
```
Exception in thread “main” java.lang.InputMismatchException
        at java.util.Scanner.throwFor(Scanner.java:819)
        at java.util.Scanner.next(Scanner.java:1431)
        at java.util.Scanner.nextInt(Scanner.java:2040)
        at java.util.Scanner.nextInt(Scanner.java:2000)
        at Ch8Sample1.main(Ch8Sample1.java:35)
```

Catching an Exception



```
System.out.print(prompt);

try {

    age = scanner.nextInt( );

} catch (InputMismatchException e){

    System.out.println("Invalid Entry. "
                            +  "Please enter digits only");
}
```

try-catch Control Flow



Exception

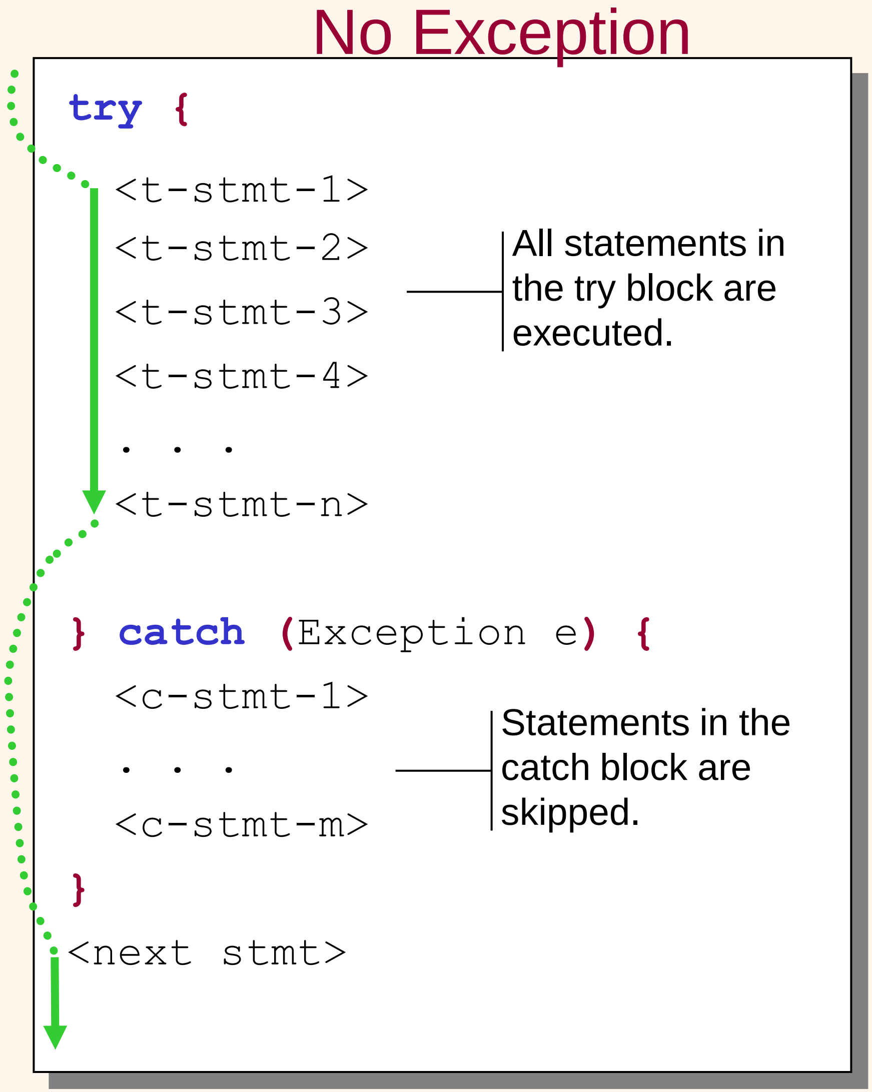
```
try {
    <t-stmt-1>
    <t-stmt-2>
    <t-stmt-3>
    <t-stmt-4>
    . . .
    <t-stmt-n>

} catch (Exception e) {
    <c-stmt-1>
    . . .
    <c-stmt-m>
}
<next stmt>
```

Assume <t-stmt-3> throws an exception.

Remaining statements in the try block is skipped.

Statements in the catch block are executed.

And the execution continues to the next statement



No Exception

```
try {
    <t-stmt-1>
    <t-stmt-2>
    <t-stmt-3>
    <t-stmt-4>
    . . .
    <t-stmt-n>

} catch (Exception e) {
    <c-stmt-1>
    . . .
    <c-stmt-m>
}
<next stmt>
```

All statements in the try block are executed.

Statements in the catch block are skipped.

Getting Information

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

```
try {
    . . .
} catch (InputMismatchException e){
    scanner.next(); //remove the leftover garbage char
    System.out.println(e.getMessage());
    e.printStackTrace();
}
```

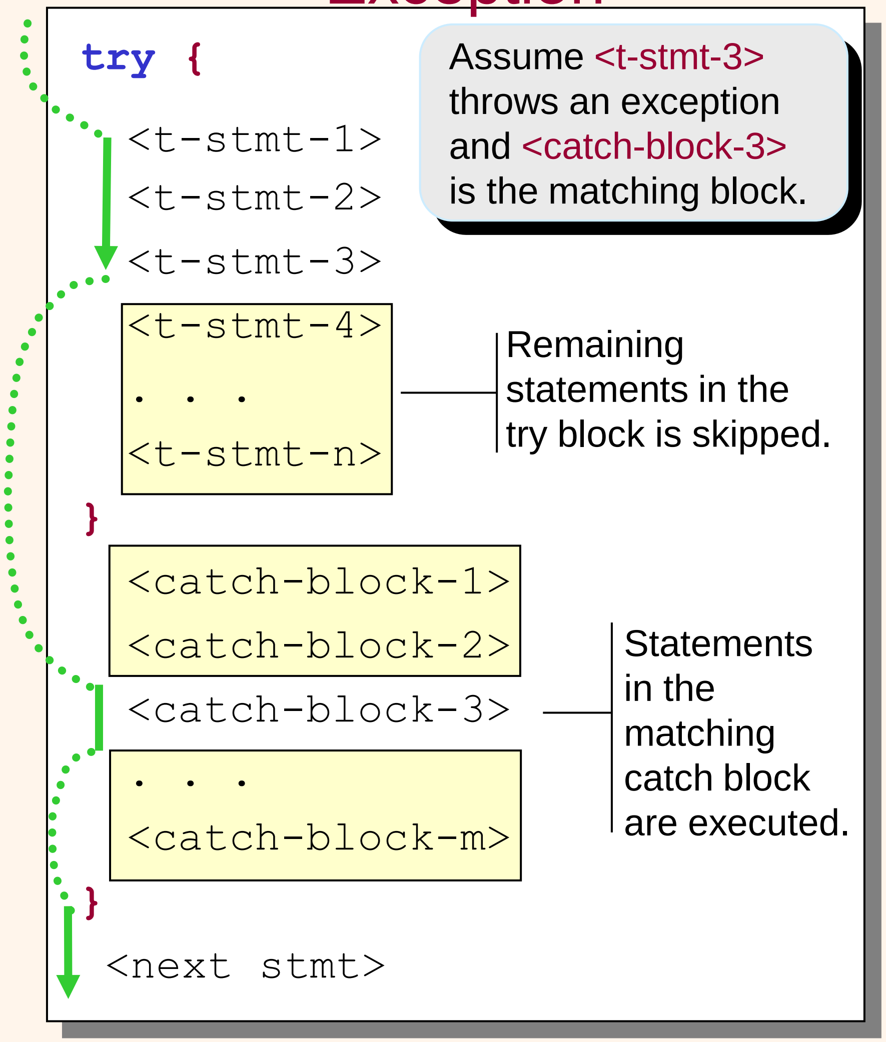
Multiple catch Blocks

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

```
try {
    . . .
    age = scanner.nextInt( );
    . . .
    val = cal.get(id); //cal is a GregorianCalendar
    . . .
} catch (InputMismatchException e){
    . . .
} catch (ArrayIndexOutOfBoundsException e){
    . . .
}
```

Multiple catch Control Flow

Exception



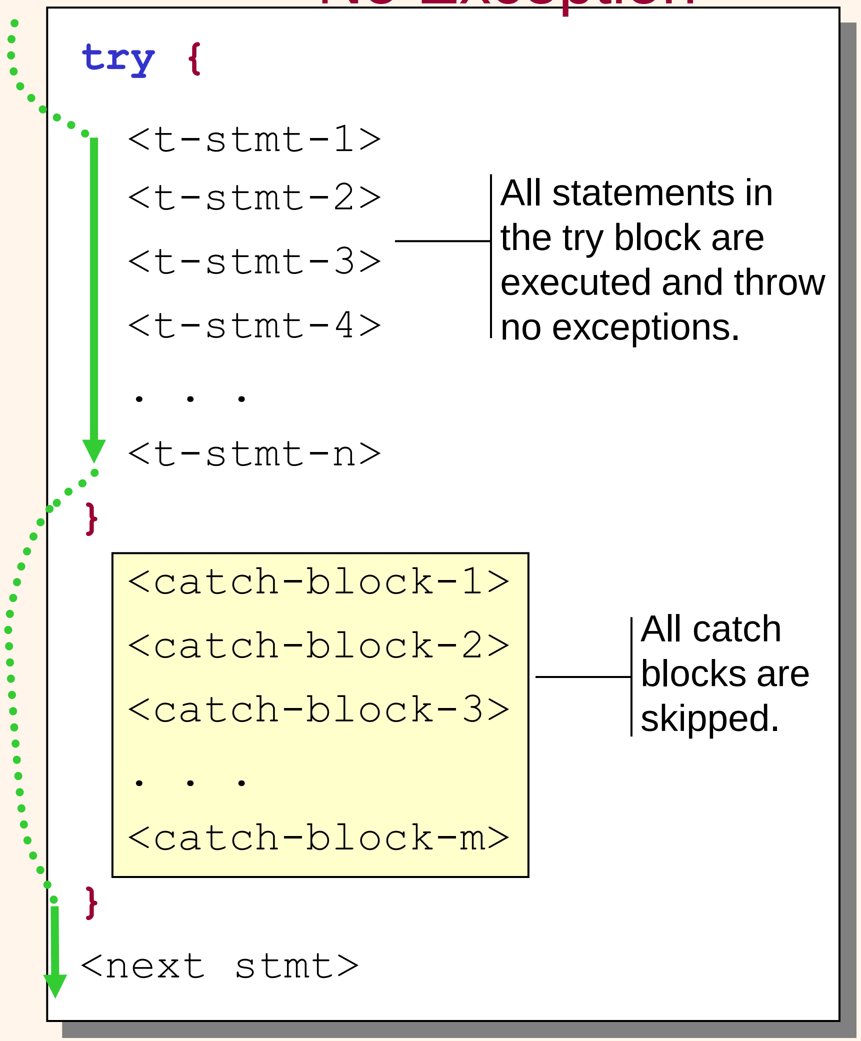
```
try {
    <t-stmt-1>
    <t-stmt-2>
    <t-stmt-3>
    <t-stmt-4>
    .  .  .
    <t-stmt-n>
}
    <catch-block-1>
    <catch-block-2>
    <catch-block-3>
    .  .  .
    <catch-block-m>
}
   <next stmt>
```

Assume <t-stmt-3> throws an exception and <catch-block-3> is the matching block.

Remaining statements in the try block is skipped.

Statements in the matching catch block are executed.

No Exception



```
try {
    <t-stmt-1>
    <t-stmt-2>
    <t-stmt-3>
    <t-stmt-4>
    .  .  .
    <t-stmt-n>
}
    <catch-block-1>
    <catch-block-2>
    <catch-block-3>
    .  .  .
    <catch-block-m>
}
<next stmt>
```

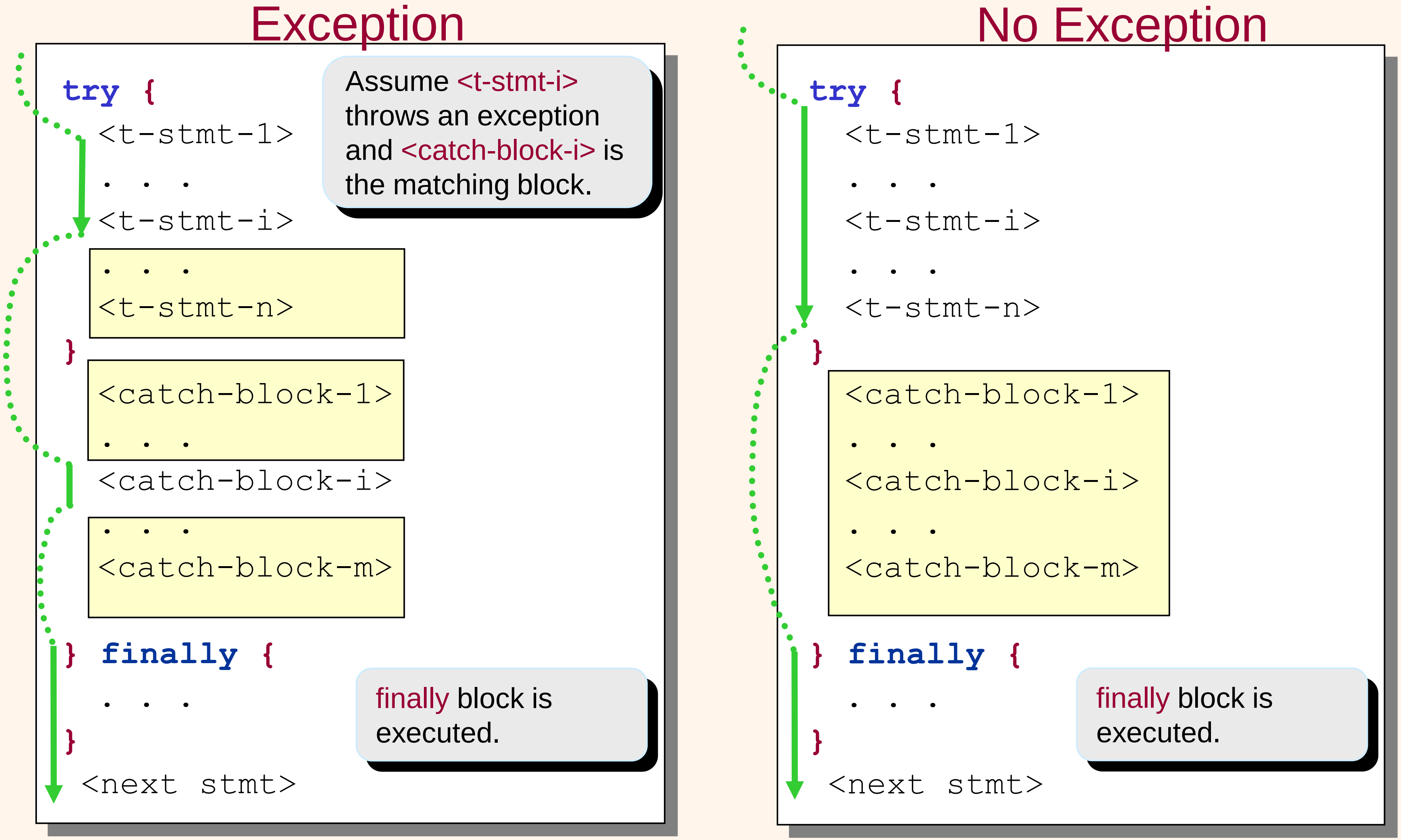
All statements in the try block are executed and throw no exceptions.

All catch blocks are skipped.

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-catch-finally Control Flow



Exception

```
try {
  <t-stmt-1>
  . . .
  <t-stmt-i>
  . . .
  <t-stmt-n>
}
  <catch-block-1>
  . . .
  <catch-block-i>
  . . .
  <catch-block-m>
} finally {
  . . .
}
<next stmt>
```

Assume <t-stmt-i> throws an exception and <catch-block-i> is the matching block.

finally block is executed.

No Exception

```
try {
  <t-stmt-1>
  . . .
  <t-stmt-i>
  . . .
  <t-stmt-n>
}
  <catch-block-1>
  . . .
  <catch-block-i>
  . . .
  <catch-block-m>
} finally {
  . . .
}
<next stmt>
```

finally block is executed.

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 InputMismatchException {
    . . .
    int age = scanner.nextInt( );
    . . .
    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.
- The method header includes the reserved word throws.

```
public void doWork(int num) throws Exception {

   . . .

   if (num != val) throw new Exception("Invalid val");

   . . .

}
```

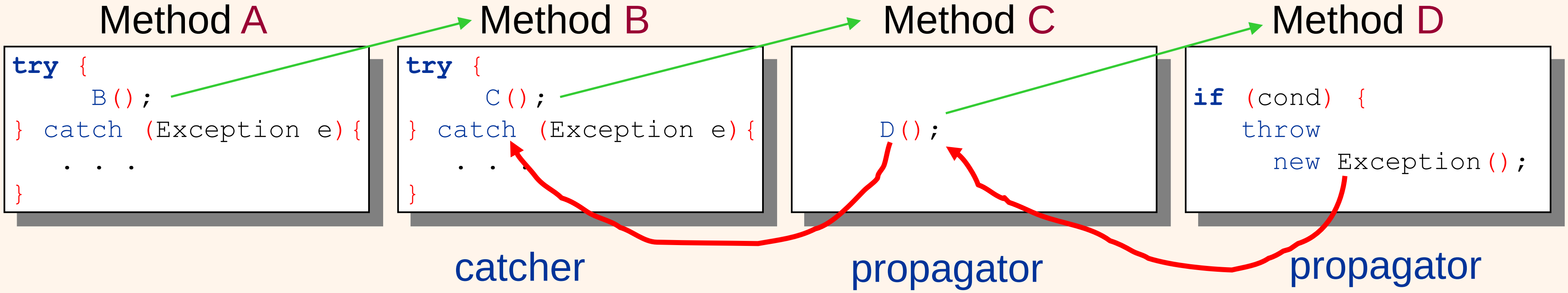
Exception Thrower

- 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.
- A method may be a catcher of one exception and a propagator of another.

Sample Call Sequence



Method A

```
try {
     B();
} catch (Exception e){
    . . .
}
```

Method B

```
try {
     C();
} catch (Exception e){
    . . .
}
```

Method C

```
      D();
```

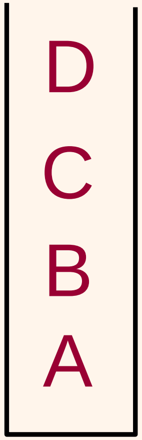
Method D

```
if (cond) {
  throw
    new Exception();
```

catcher

propagator

propagator



D

C

B

A

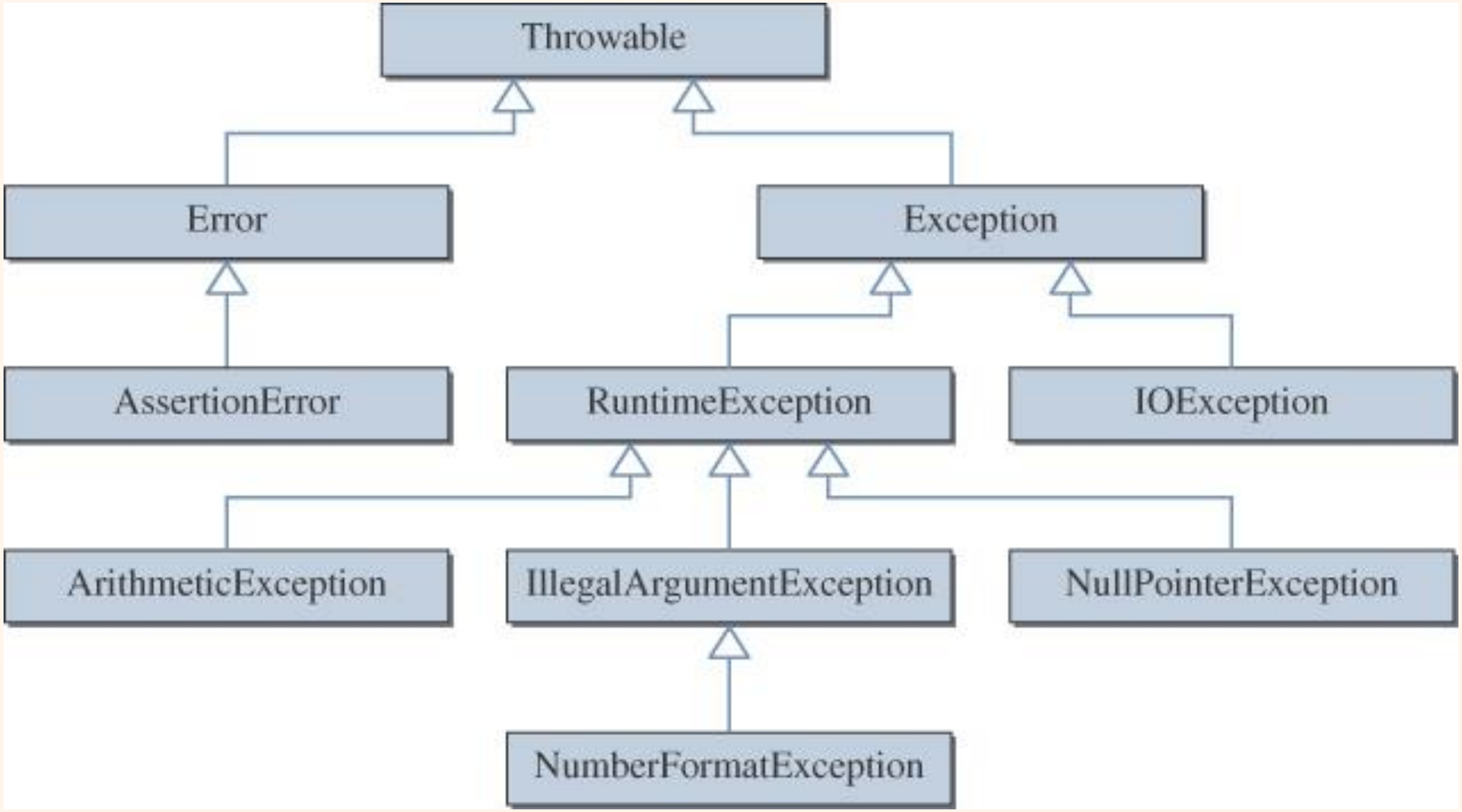
Stack Trace

Exception Types

- All types of thrown errors are instances of the **Throwable** class or its subclasses.
- Serious errors are represented by instances of the **Error** class or its subclasses.
- 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.



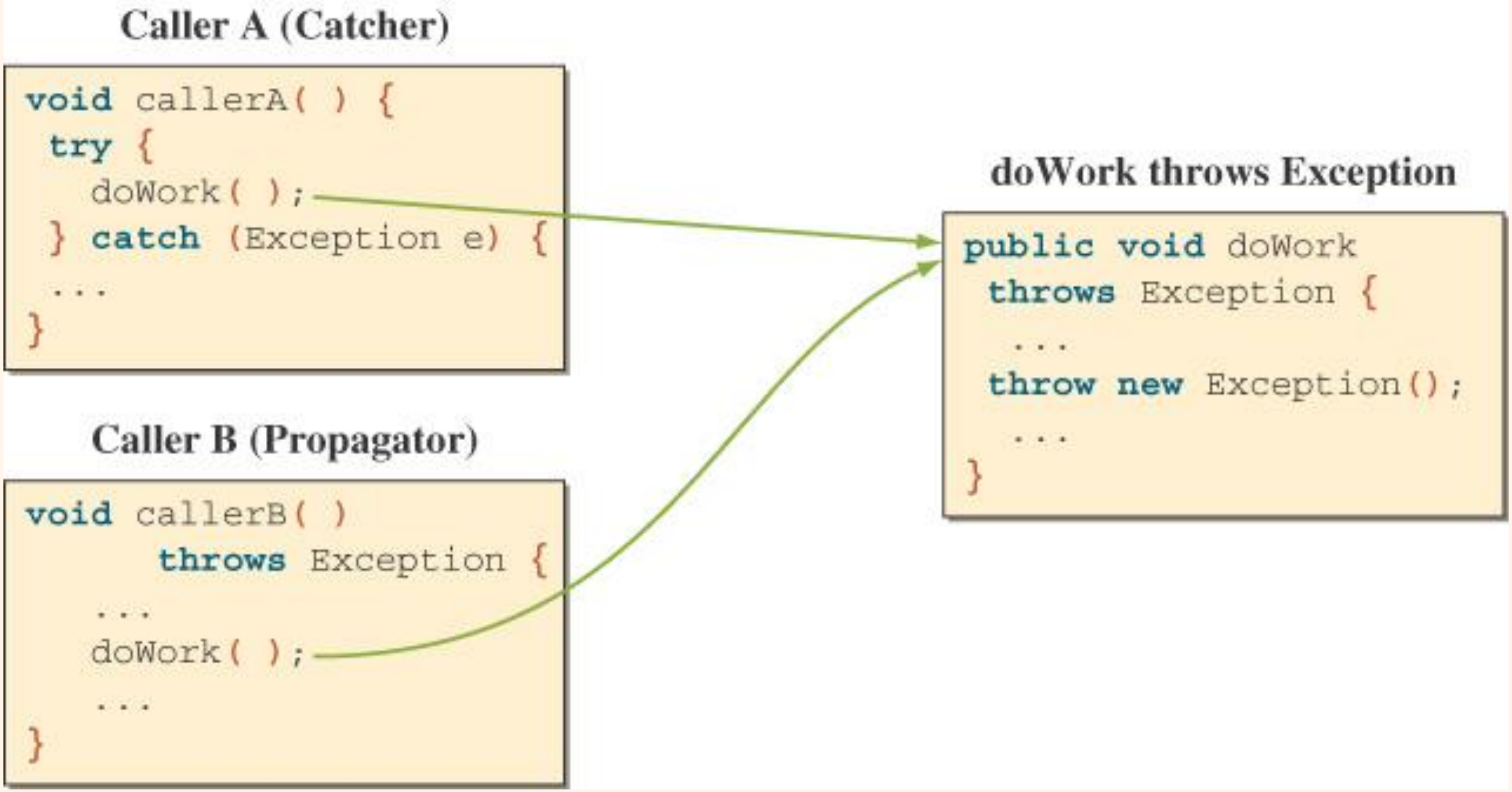
Checked vs. Runtime

- There are two types of exceptions:
  - Checked.
  - Unchecked.
- A *checked exception* is an exception that is checked at compile time.
- All other exceptions are *unchecked*, or *runtime, exceptions*. As the name suggests, they are detected only at runtime.

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

Handling Checked Exceptions



Caller A (Catcher)

```
void callerA( ) {
 try {
   doWork( );
 } catch (Exception e) {
  ...
}
```

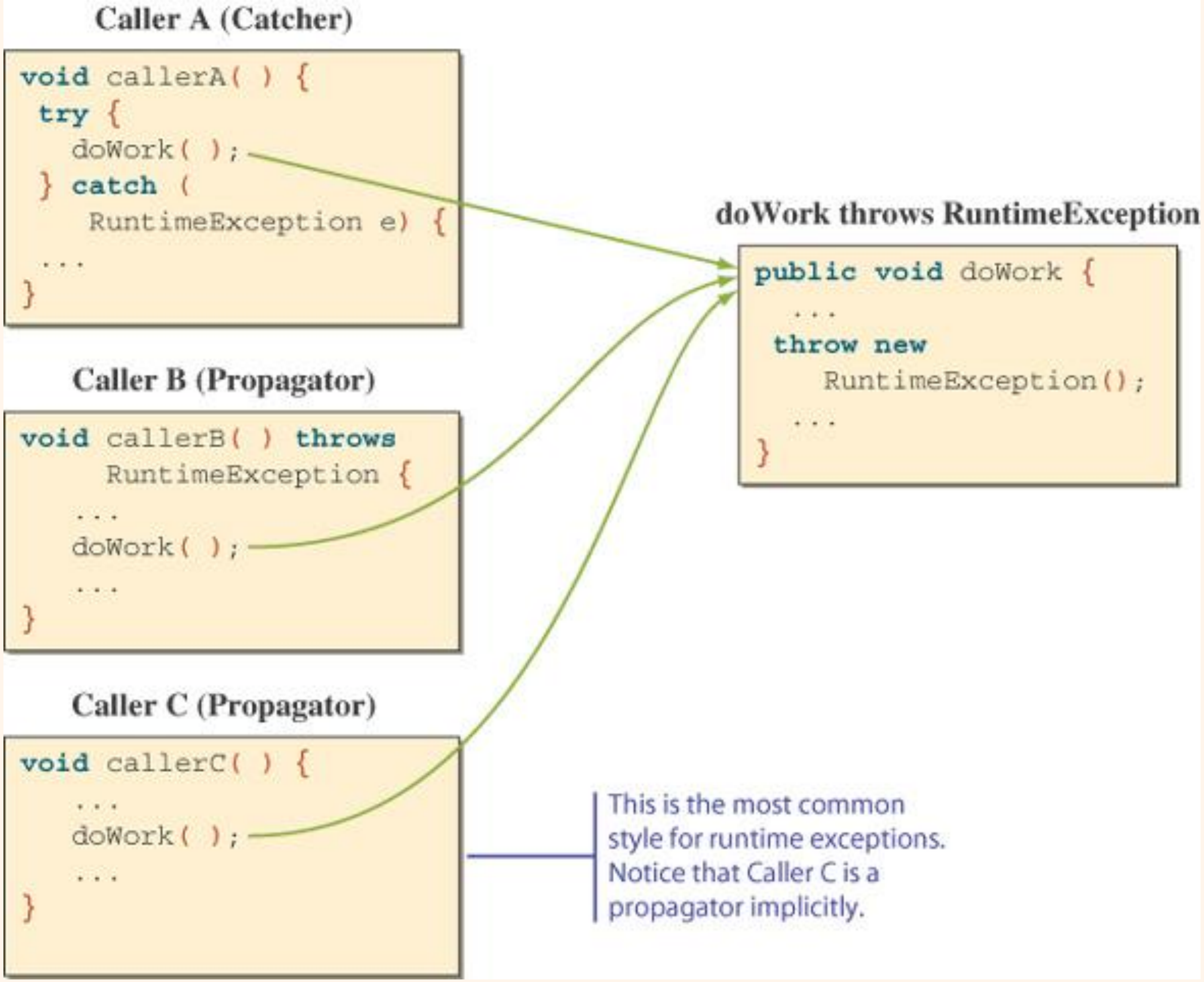
Caller B (Propagator)

```
void callerB( )
        throws Exception {
   ...
   doWork( );
   ...
}
```

doWork throws Exception

```
public void doWork
 throws Exception {
   ...
 throw new Exception();
   ...
}
```

Handling Runtime Exceptions



Caller A (Catcher)

```
void callerA( ) {
 try {
   doWork( );
 } catch (
    RuntimeException e) {
 ...
}
```

Caller B (Propagator)

```
void callerB( ) throws
     RuntimeException {
   ...
   doWork( );
   ...
}
```

Caller C (Propagator)

```
void callerC( ) {
   ...
   doWork( );
   ...
}
```

doWork throws RuntimeException

```
public void doWork {
  ...
 throw new
    RuntimeException();
  ...
}
```

This is the most common style for runtime exceptions. Notice that Caller C is a propagator implicitly.

Programmer-Defined Exceptions

- Using the standard exception classes, we can use the getMessage method to retrieve the error message.
- By defining our own exception class, we can pack more useful information
  - for example, we may define a OutOfStock exception class and include information such as how many items to order
- AgeInputException is defined as a subclass of Exception and includes public methods to access three pieces of information it carries: lower and upper bounds of valid age input and the (invalid) value entered by the user.

Assertions

- The syntax for the **assert** statement is

```
      assert <boolean expression>;
```

  where `<boolean expression>` represents the condition that must be true if the code is working correctly.

- If the expression results in **false**, an **AssertionError** (a subclass of **Error**) is thrown.

Sample Use #1

```
public double deposit(double amount) {
   double oldBalance = balance;
   balance += amount;
   assert balance > oldBalance;
}

public double withdraw(double amount) {
   double oldBalance = balance;
   balance -= amount;
   assert balance < oldBalance;
}
```

Second Form

- The assert statement may also take the form:

```
assert <boolean expression>: <expression>;
```

where `<expression>` represents the value passed as an argument to the constructor of the **AssertionError** class. The value serves as the detailed message of a thrown exception.

Sample Use #2

```
public double deposit(double amount) {

   double oldBalance = balance;

   balance += amount;

   assert balance > oldBalance :
      "Serious Error - balance did not " +
      " increase after deposit";
}
```

Running Programs with Assertions

- To run the program with assertions enabled, use

```
java -ea <main class>
```

- If the `-ea` option is not provided, the program is executed without checking assertions.

Different Uses of Assertions

- *Precondition assertions* check for a condition that must be true before executing a method.
- *Postcondition assertions* check conditions that must be true after a method is executed.
- A *control-flow invariant* is a third type of assertion that is used to assert the control must flow to particular cases.

Problem Statement

Implement a Keyless Entry System that asks for three pieces of information: resident's name, room number, and a password.

- *A password is any sequence of 8 or more characters and is unique to an individual dorm resident.*
- *If everything matches, then the system unlocks and opens the door.*
- *We assume no two residents have the same name.*
- *We use the provided support classes Door and Dorm.*
- *Sample resident data named sampleResidentFile can be used for development.*

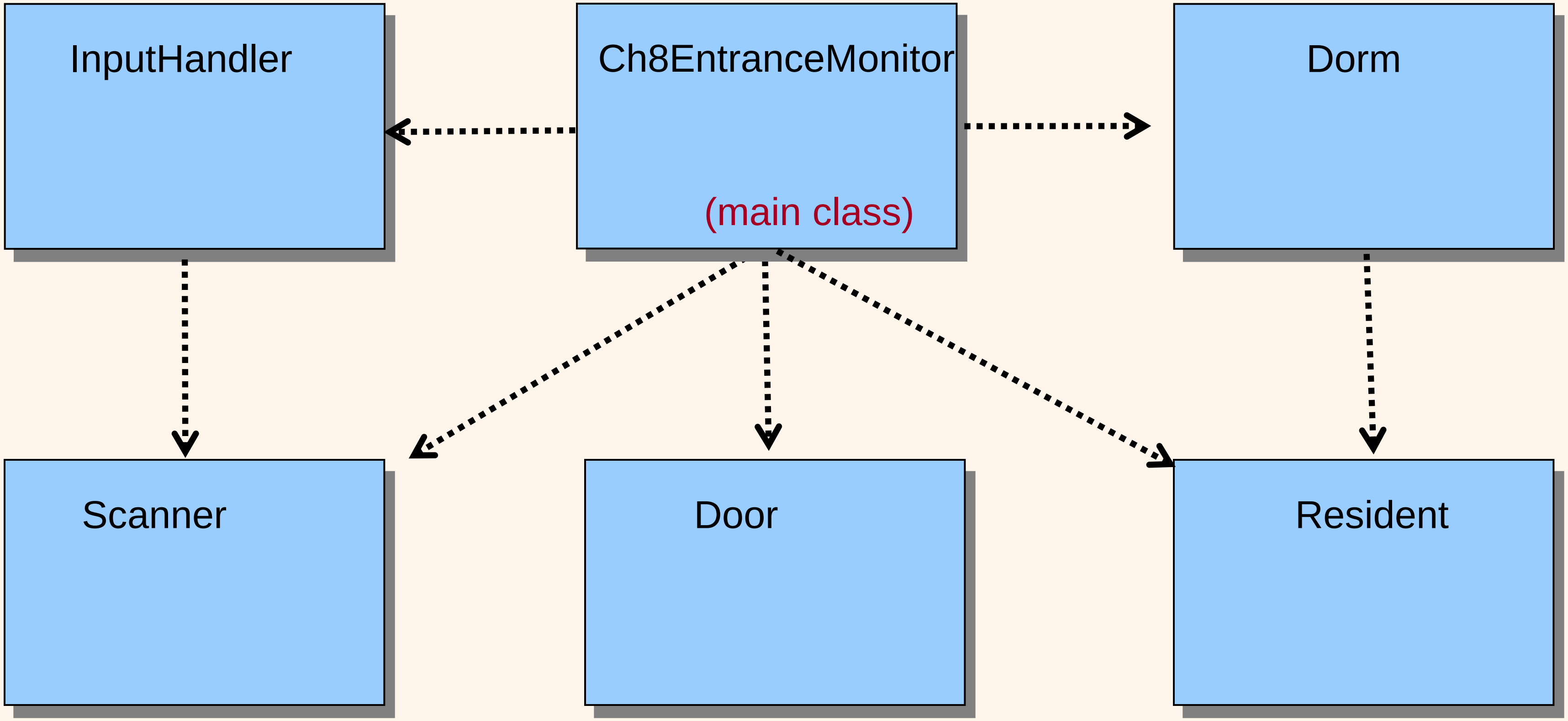
Overall Plan

- Tasks:
  - To begin our development effort, we must first find out the capabilities of the Dorm and Door classes.
  - Also, for us to implement the class correctly, we need the specification of the Resident class.
- In addition to the given helper classes and the Resident class, we need to design other classes for this application.
  - As the number of classes gets larger, we need to plan the classes carefully.

Design Document

Class	Purpose
Ch8EntranceMonitor	The top-level control object that manages other objects in the program. This is an instantiable main class.
Door	The given predefined class that simulates the opening of a door.
Dorm	The given predefined class that maintains a list of Resident objects.
Resident	This class maintains information on individual dorm residents. Specification for this class is provided to us.
InputHandler	The user interface class for handling input routines.
Scanner	The standard class for inputting data.

Class Relationships



Development Steps

- We will develop this program in three steps:

1. Define the Resident class and explore the Dorm class. Start with a program skeleton to test the Resident class.
2. Define the user interface InputHandler class. Modify the top-level control class as necessary.
3. Finalize the code by making improvements and tying up loose ends.

Step 1 Design

- Explore the Dorm class
- Implement the Resident class, following the given specification
- Start with the skeleton main class

Step 1 Code

Program source file is too big to list here. From now on, we ask you to view the source files using your Java IDE.

Directory: Chapter8/Step1

Source Files: Resident.java
Ch8EntranceMonitor.java

Step 1 Test

- The purpose of Step 1 testing is to verify that the Dorm class is used correctly to open a file and get the contents of the file.
- To test it, we need a file that contains the resident information. A sample test file called testfile.dat is provided for testing purpose.
  - This file contains information on four residents.
  - This file was created by executing the **SampleCreateResidentFile** program, which you can modify to create other test data files.

Step 2 Design

- Design and implement the InputHandler class.
- Modify the main class to incorporate the new class.

Step 2 Code

Directory: Chapter8/Step2

Source Files: Resident.java
Ch8EntranceMonitor.java
InputHandler.java

Step 2 Test

- The purpose of Step 2 testing is to verify the correct behavior of an InputHandler.
- We need to test both successful and unsuccessful cases.
  - We must verify that the door is in fact opened when the valid information is entered.
  - We must also verify that the error message is displayed when there's an error in input.
    - We should test invalid cases such as entering nonexistent name, corrent name but wrong password, not enetering all information, and so forth.

Step 3: Finalize

- Possible Extensions
  - Improve the user interface with a customized form window for entering three pieces of information.
  - Terminate the program when the administrator enters a special code