Exceptions

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.

An Introduction to Object-Oriented Programming with Java C. Thomas Wu

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*.

An Introduction to
Object-Oriented
Programming with **Java** C. Thomas Wu

Not Catching Exceptions

```java
String inputStr;
int     age;


inputStr = JOptionPane.showInputDialog(null, "Age:");
age      = Integer.parseInt(inputStr);
```

Error message for invalid input

```
java.lang.NumberFormatException: ten
        at java.lang.Integer.parseInt(Integer.java:405)
        at java.lang.Integer.parseInt(Integer.java:454)
        at Ch8Sample1.main(Ch8Sample1.java:20)
```

An Introduction to Object-Oriented Programming with **Java** C. Thomas Wu

Catching an Exception

```java
inputStr = JOptionPane.showInputDialog(null, "Age:");

try {

    age = Integer.parseInt(inputStr);

} catch (NumberFormatException e){

    JOptionPane.showMessageDialog(null, "'" + inputStr
        + "' is invalid\n"
        + "Please enter digits only");

}
```
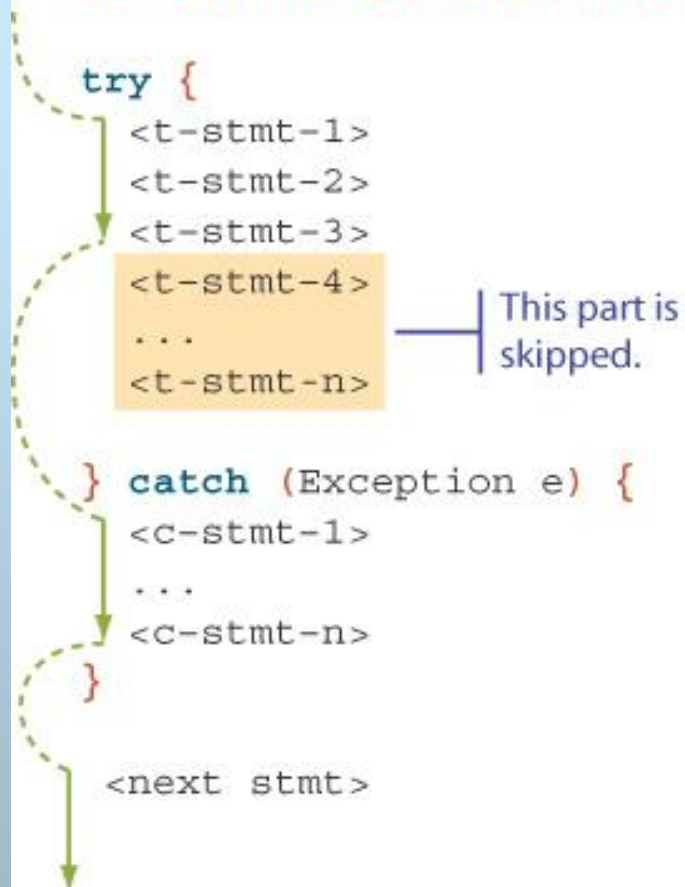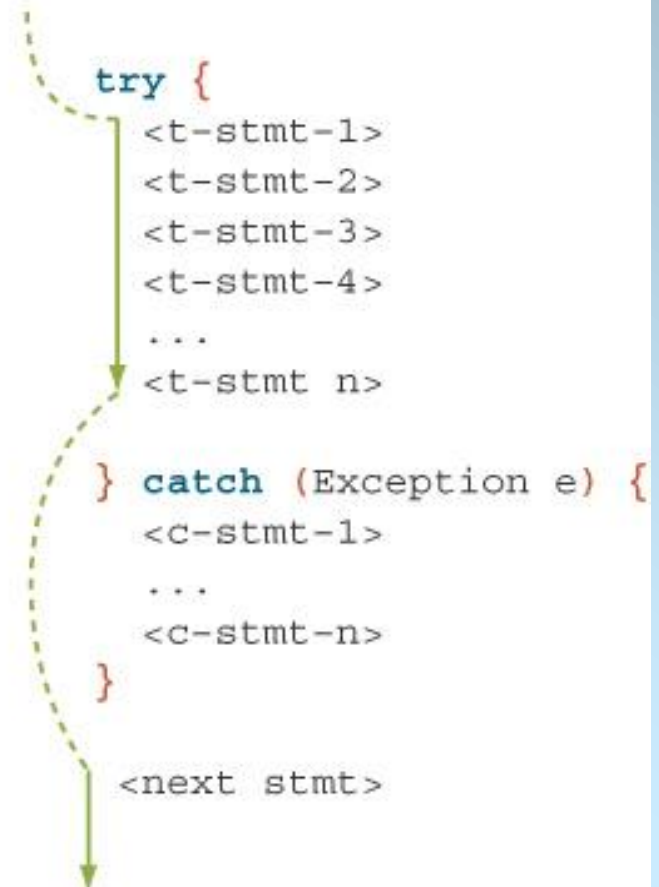
try

catch

An Introduction to Object-Oriented Programming with **Java** C. Thomas Wu

try-catch Control Flow

Exception

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

```
try {
    <t-stmt-1>
    <t-stmt-2>
    <t-stmt-3>
    <t-stmt-4>          This part is
    ...                 skipped.
    <t-stmt-n>

} catch (Exception e) {
    <c-stmt-1>
    ...
    <c-stmt-n>
}

    <next stmt>
```

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-n>
}

    <next stmt>
```

An Introduction to Object-Oriented Programming with **Java** C. Thomas Wu

Getting Information

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

```java
try {

    . . .

} catch (NumberFormatException e){

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

}
```

An Introduction to Object-Oriented Programming with **Java** C. Thomas Wu

Multiple catch Blocks

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

```java
try {

    . . .

    age = Integer.parseInt(inputStr);

    . . .

    val = cal.get(id); //cal is a GregorianCalendar

    . . .

} catch (NumberFormatException e){

    . . .

} catch (ArrayIndexOutOfBoundsException e){

    . . .

}
```

An Introduction to Object-Oriented Programming with Java   C. Thomas Wu
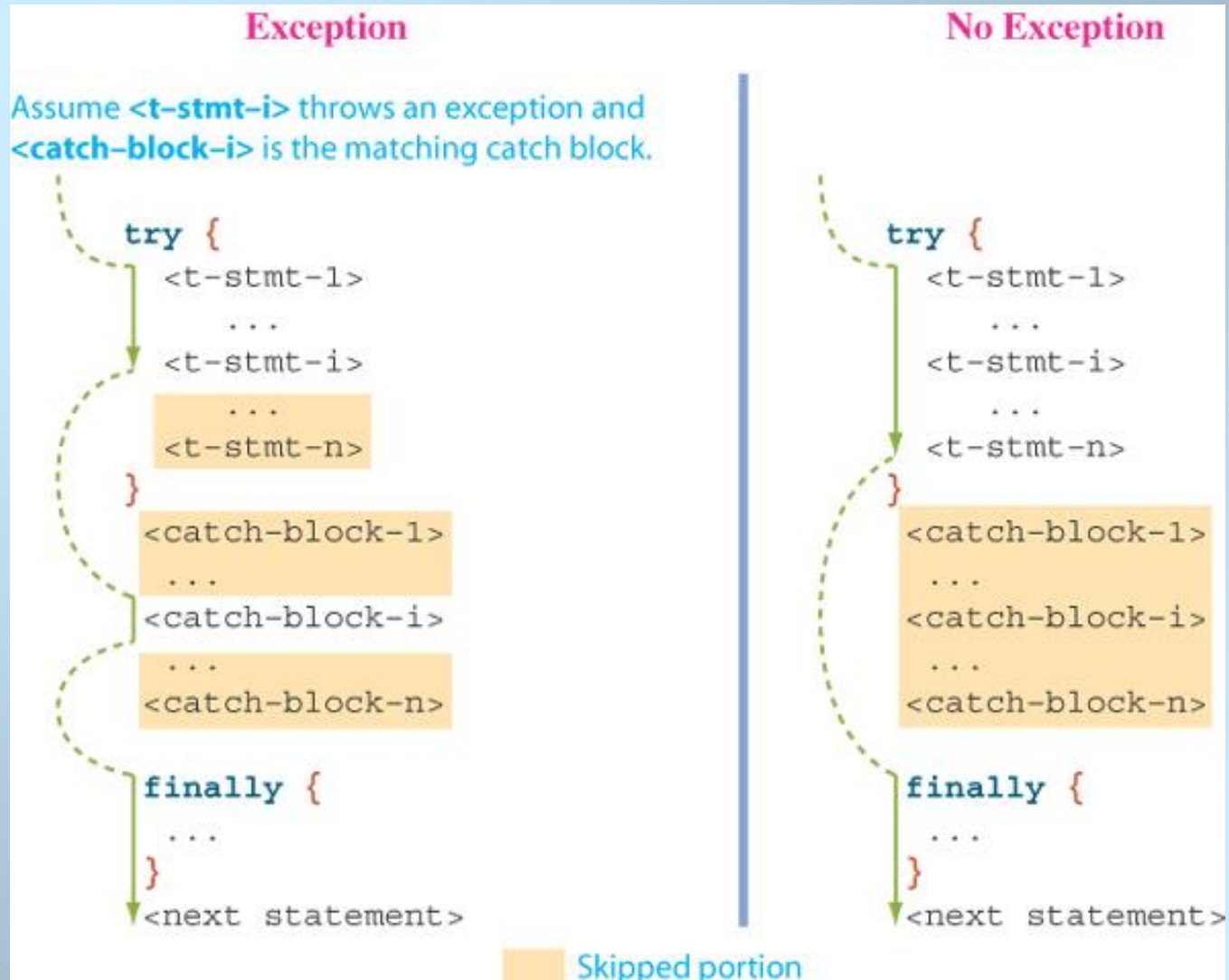
Multiple catch Control Flow
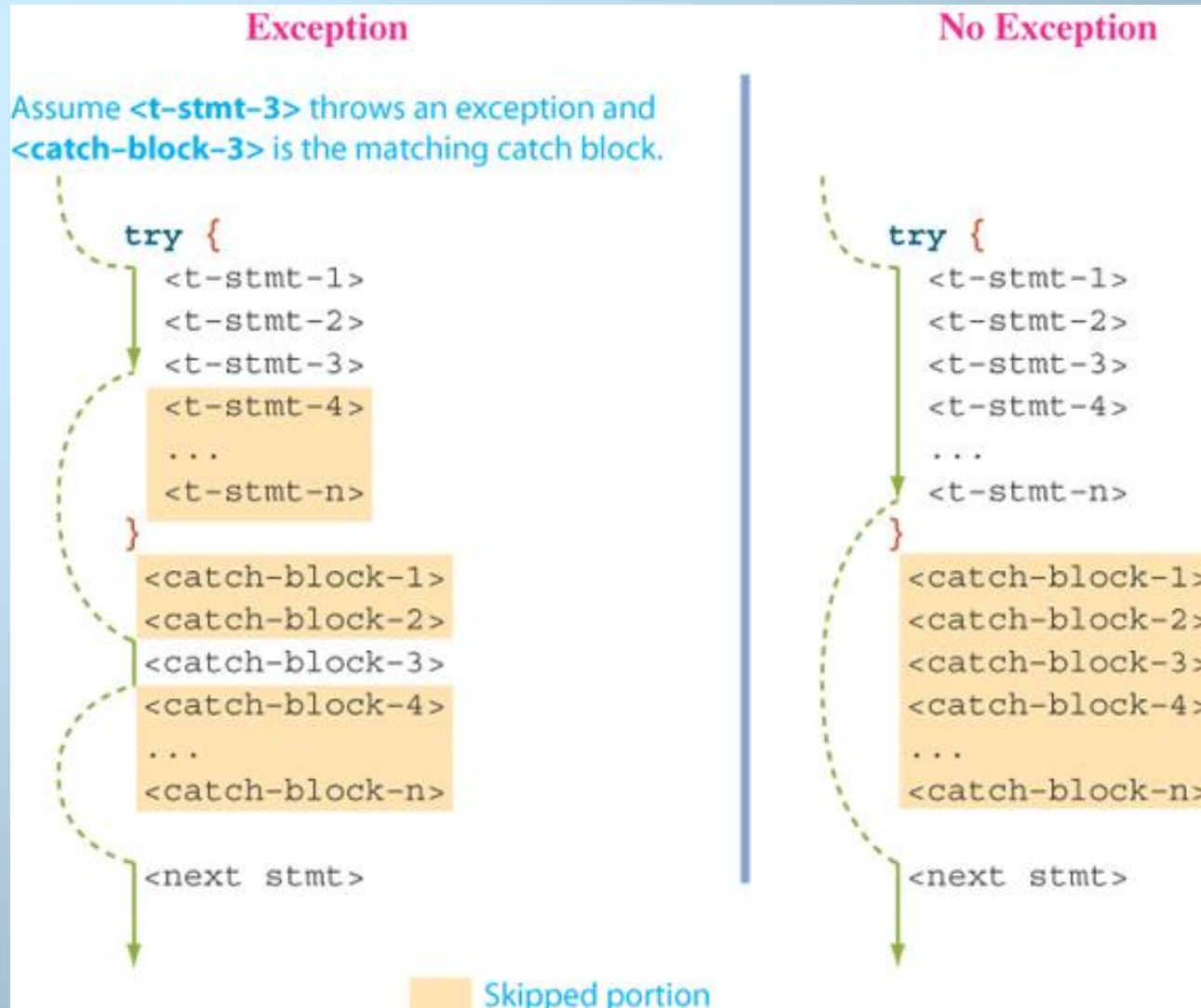
The finally Block

- 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

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.

```java
public int getAge( ) throws NumberFormatException {

    . . .

    int age = Integer.parseInt(inputStr);

    . . .

    return age;

}
```

An Introduction to Object-Oriented Programming with **Java** C. Thomas Wu

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.

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

    . . .

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

    . . .

}
```

An Introduction to Object-Oriented Programming with **Java** C. Thomas Wu

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:
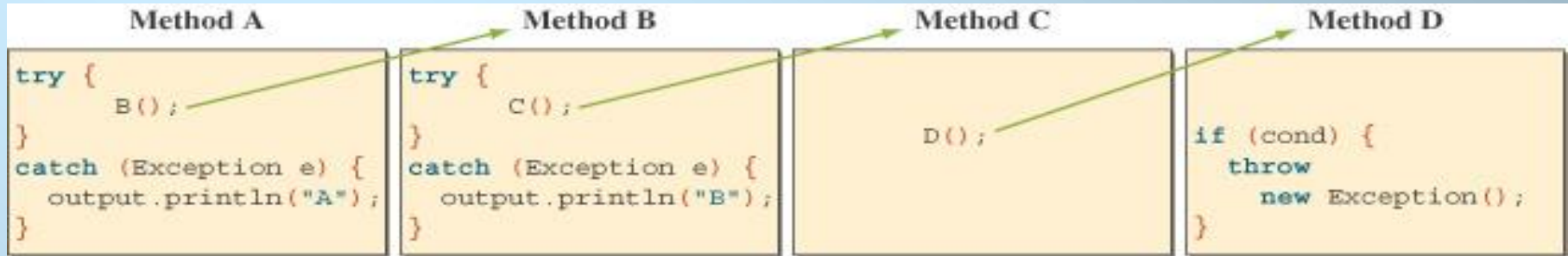 - catcher.
  - propagator.

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

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.
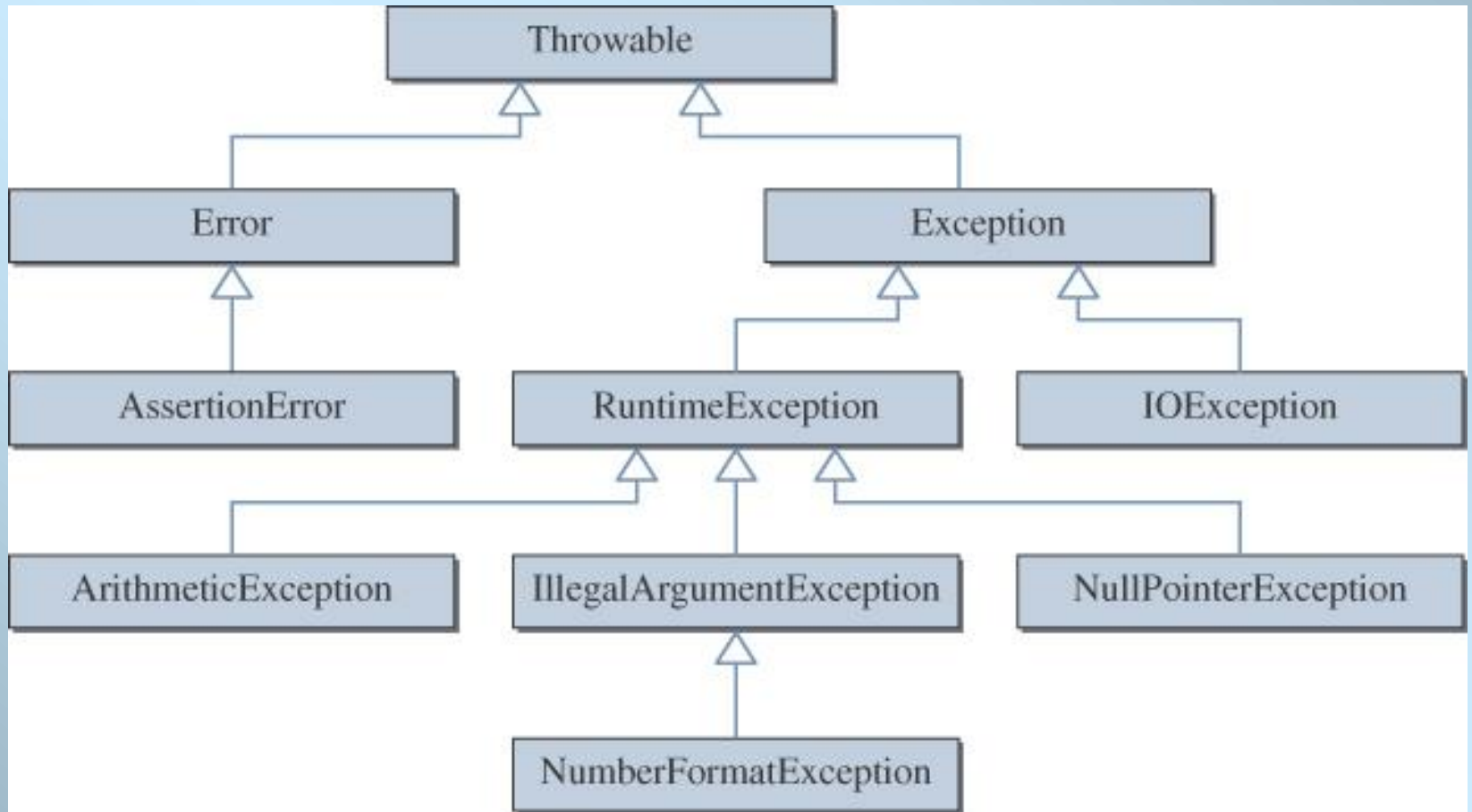
Checked vs. Runtime

- 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.
  - E.g. trying to divide a number by zero – ArithmeticException; trying to convert a string with letters to an integer - NumberFormatException

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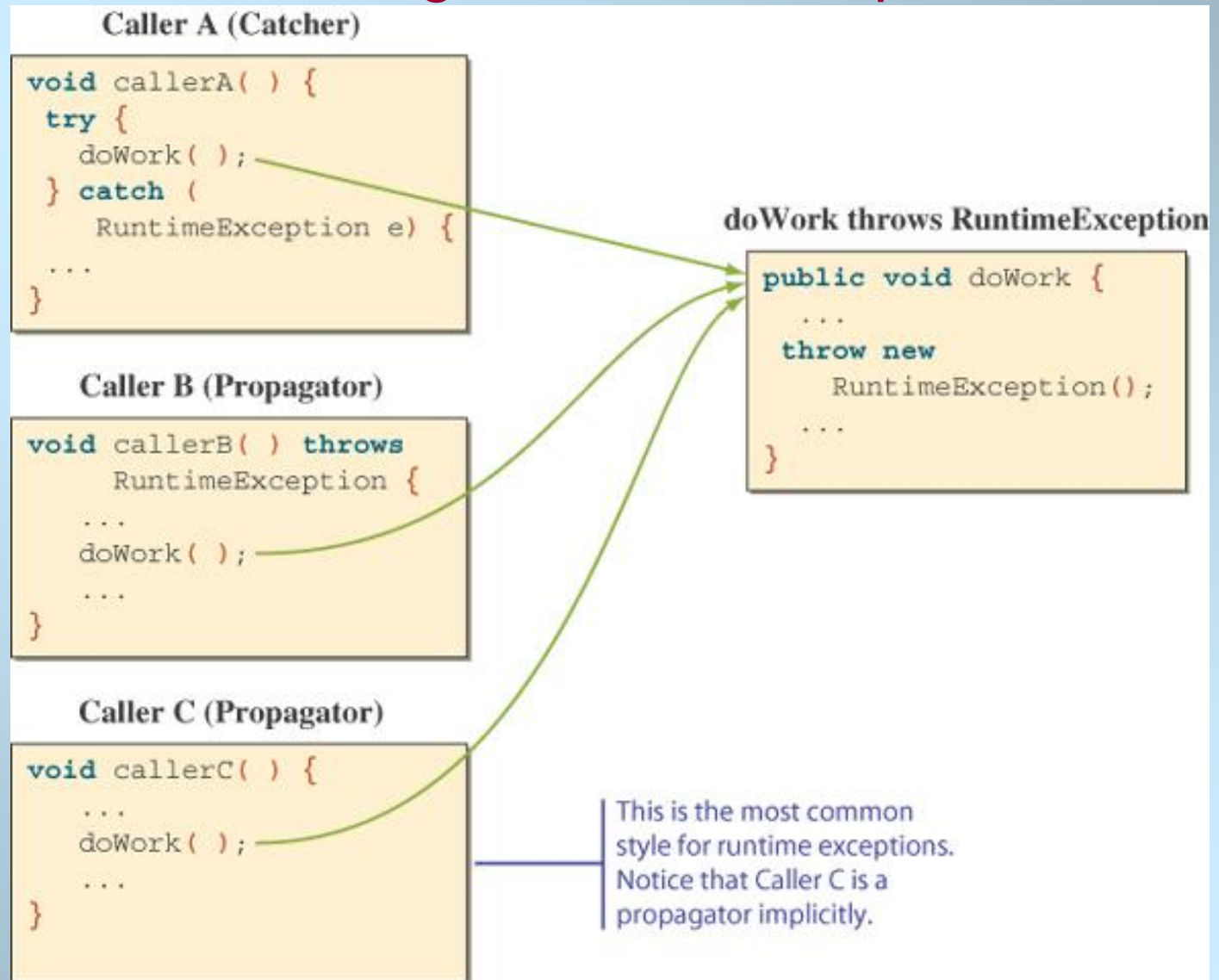
```
void callerB( )
      throws Exception {
  ...
  doWork( );
  ...
}
```

doWork throws Exception

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

An Introduction to Object-Oriented Programming with Java C. Thomas Wu

Handling Runtime Exceptions

Caller A (Catcher)

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

doWork throws RuntimeException

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

Caller B (Propagator)

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

Caller C (Propagator)

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

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

An Introduction to Object-Oriented Programming with **Java** C. Thomas Wu

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.

An Introduction to Object-Oriented Programming with **Java** C. Thomas Wu

END