Advanced Object-Oriented Programming

Exceptions

Dr. Kulwadee Somboonviwat

International College, KMITL

kskulwad@kmitl.ac.th

Exceptions

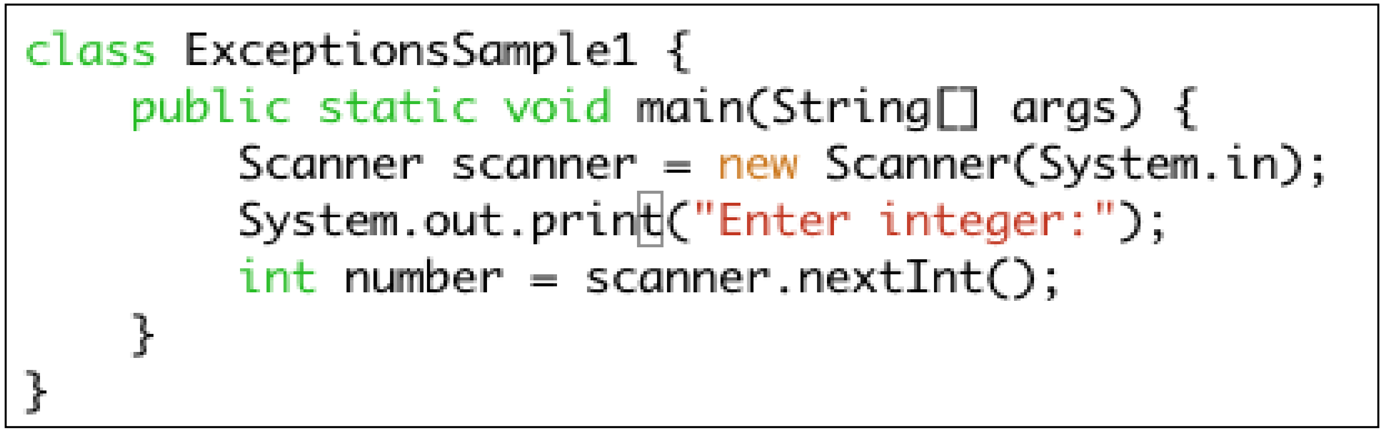
- Definition
- Catching Exceptions
- Propagating Exceptions
- Throwing Exceptions
- Classification of Exceptions
- Programmer-defined Exceptions

Definition

- An *exception* represents an error condition that can occur during the normal course of program execution.

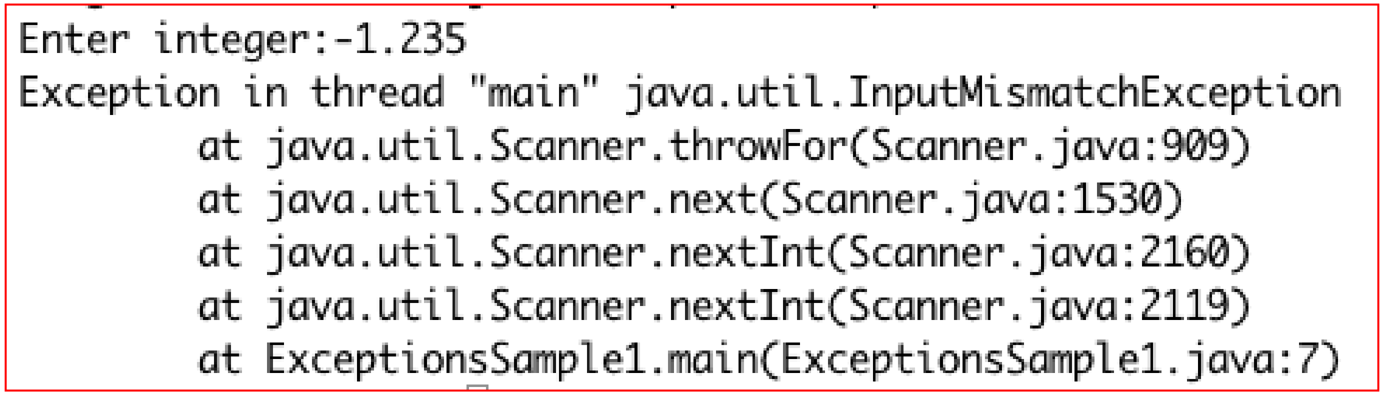
- When an exception occurs, or is *thrown*, the normal sequence of flow is terminated. The exception-handling routine is then executed; we say the thrown exception is *caught*.

Not Catching Exceptions



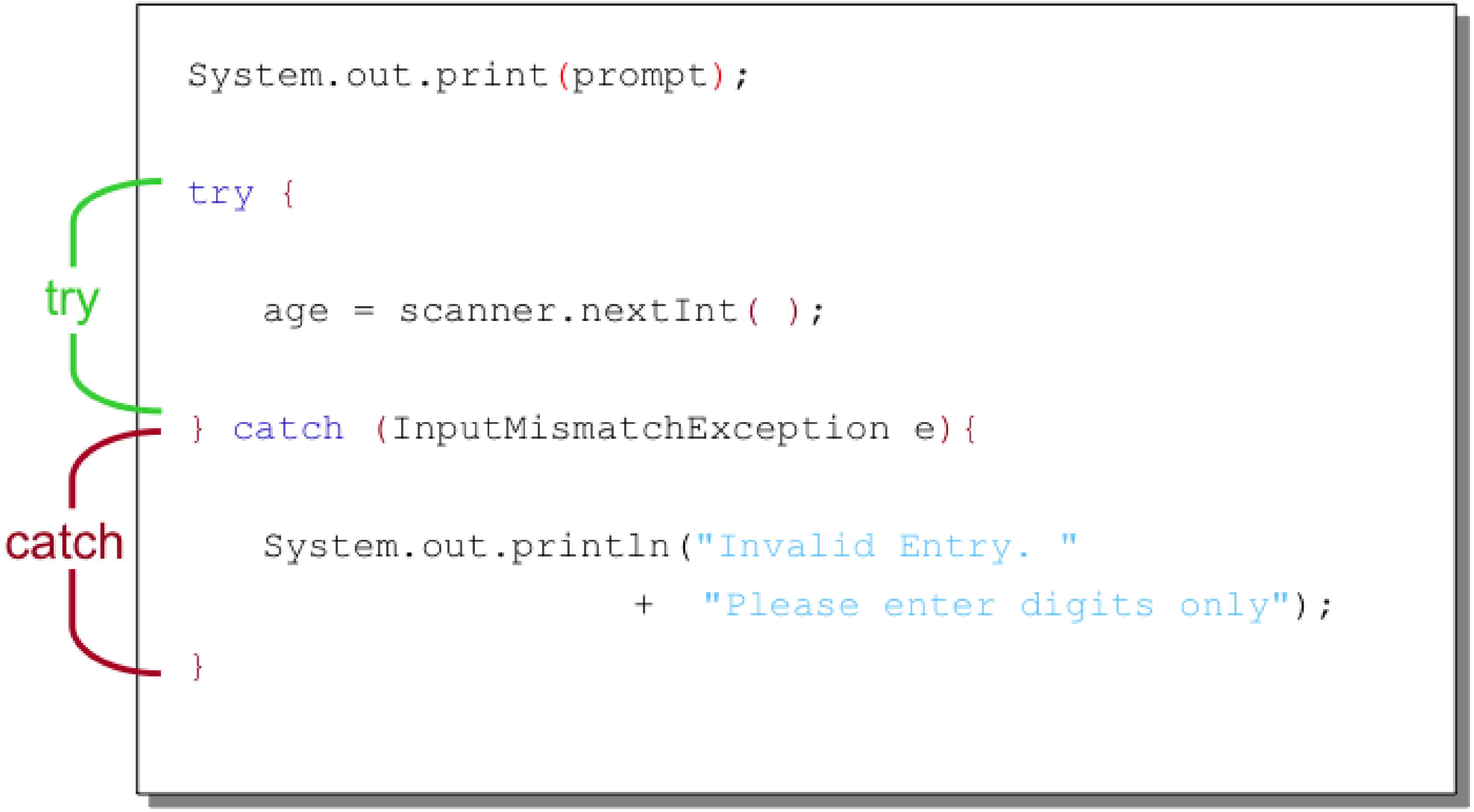
```
class ExceptionsSample1 {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter integer:");
        int number = scanner.nextInt();
    }
}
```

Error message for invalid input



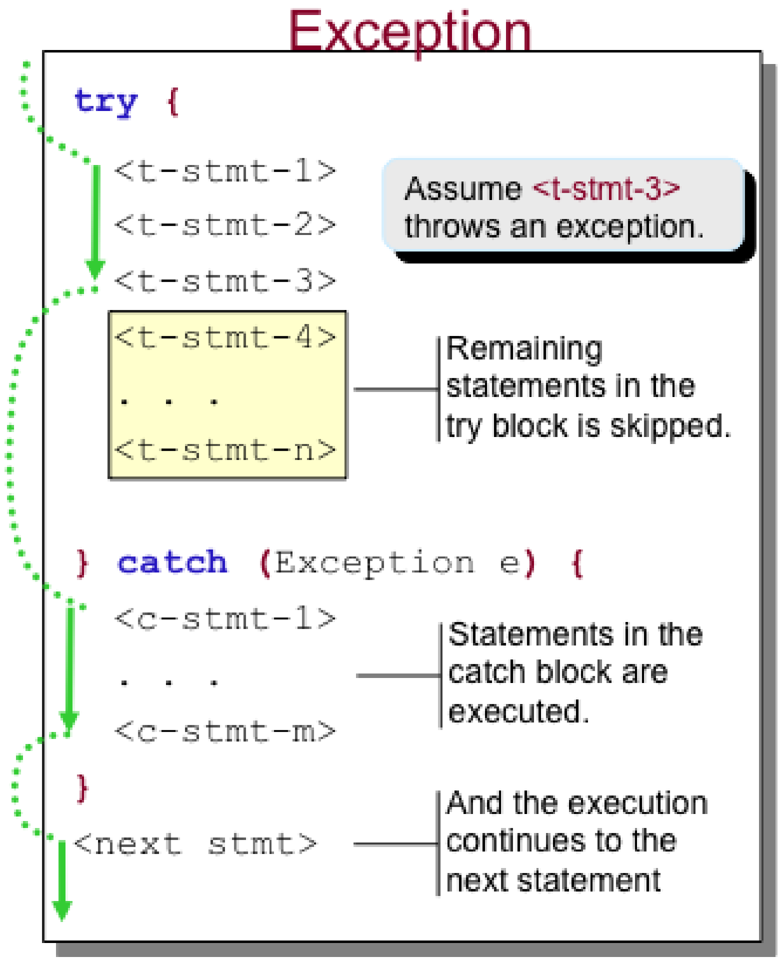
```
Enter integer:-1.235
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Scanner.java:909)
        at java.util.Scanner.next(Scanner.java:1530)
        at java.util.Scanner.nextInt(Scanner.java:2160)
        at java.util.Scanner.nextInt(Scanner.java:2119)
        at ExceptionsSample1.main(ExceptionsSample1.java:7)
```

Catching an Exception



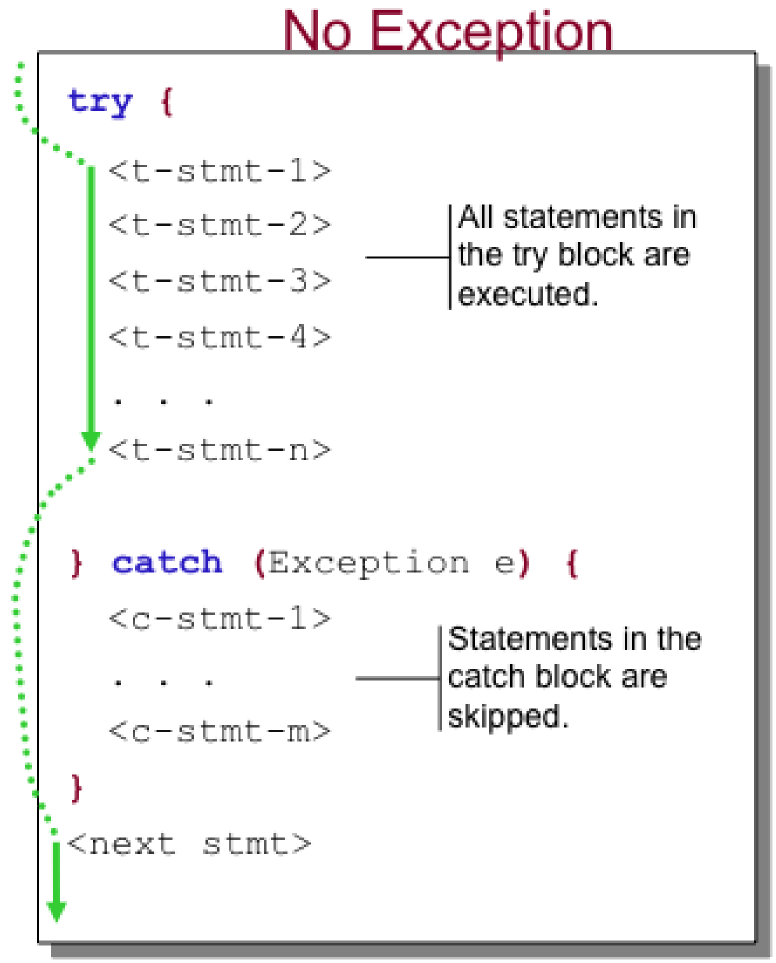
```
System.out.print(prompt);

try {

    age = scanner.nextInt( );

} catch (InputMismatchException e){

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

try-catch control flow



```
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

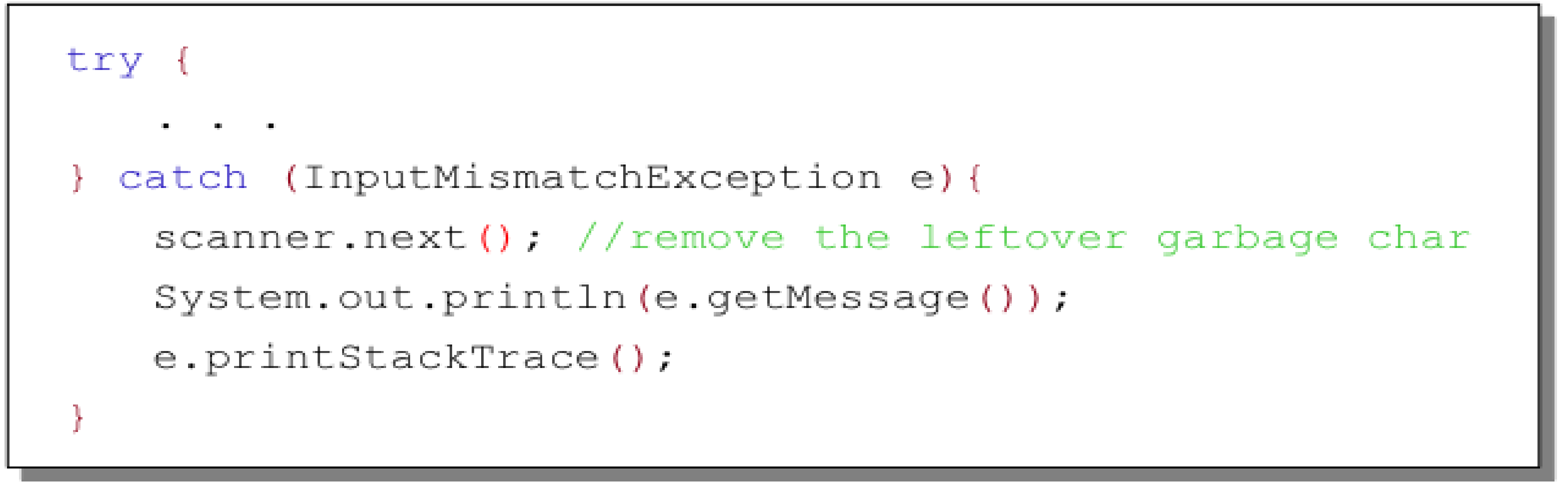


```
try {
    <t-stmt-1>
    <t-stmt-2>
    <t-stmt-3>
    <t-stmt-4>
    . . .
    <t-stmt-n>

} 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.

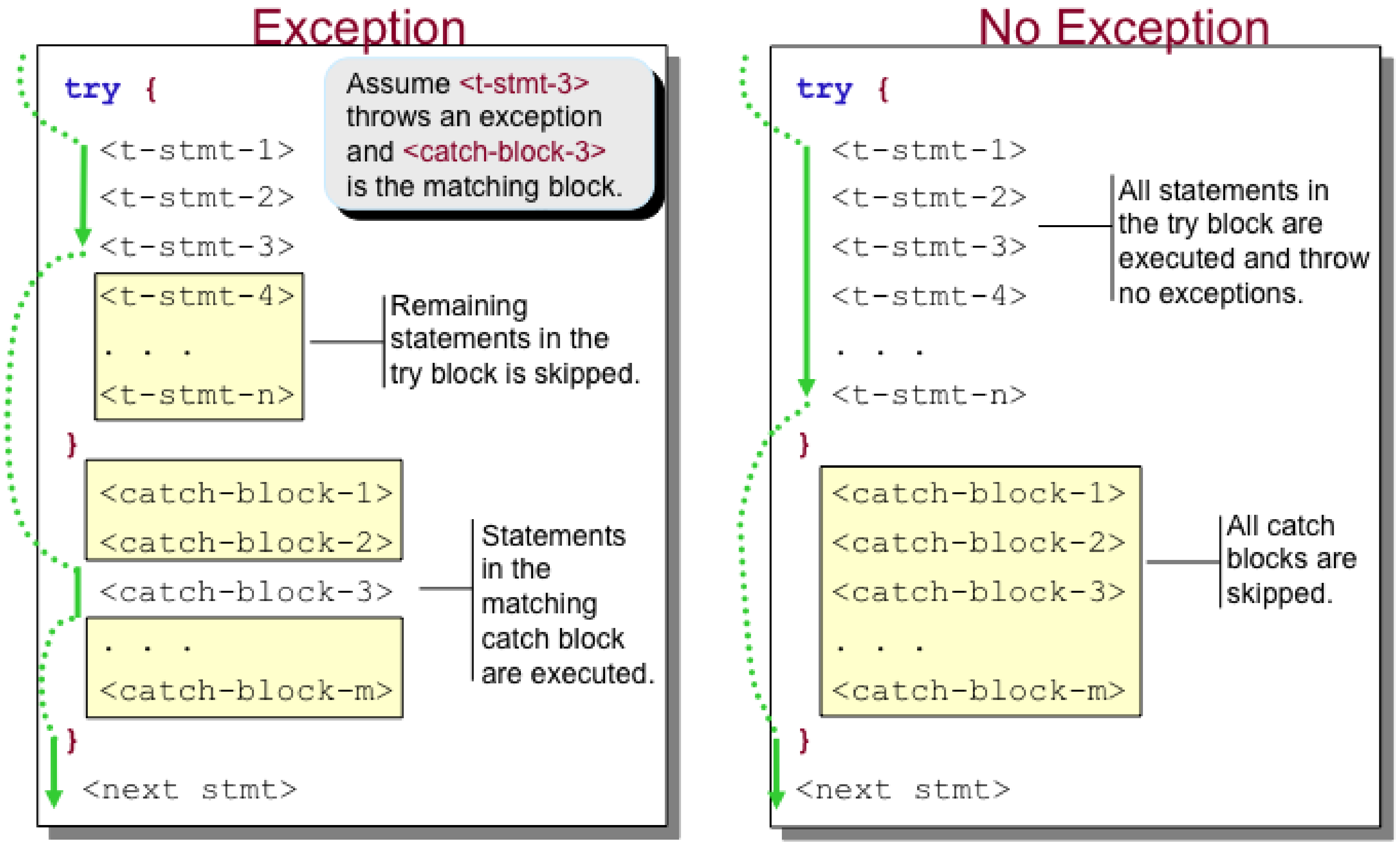
Getting Exceptions Information

- 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

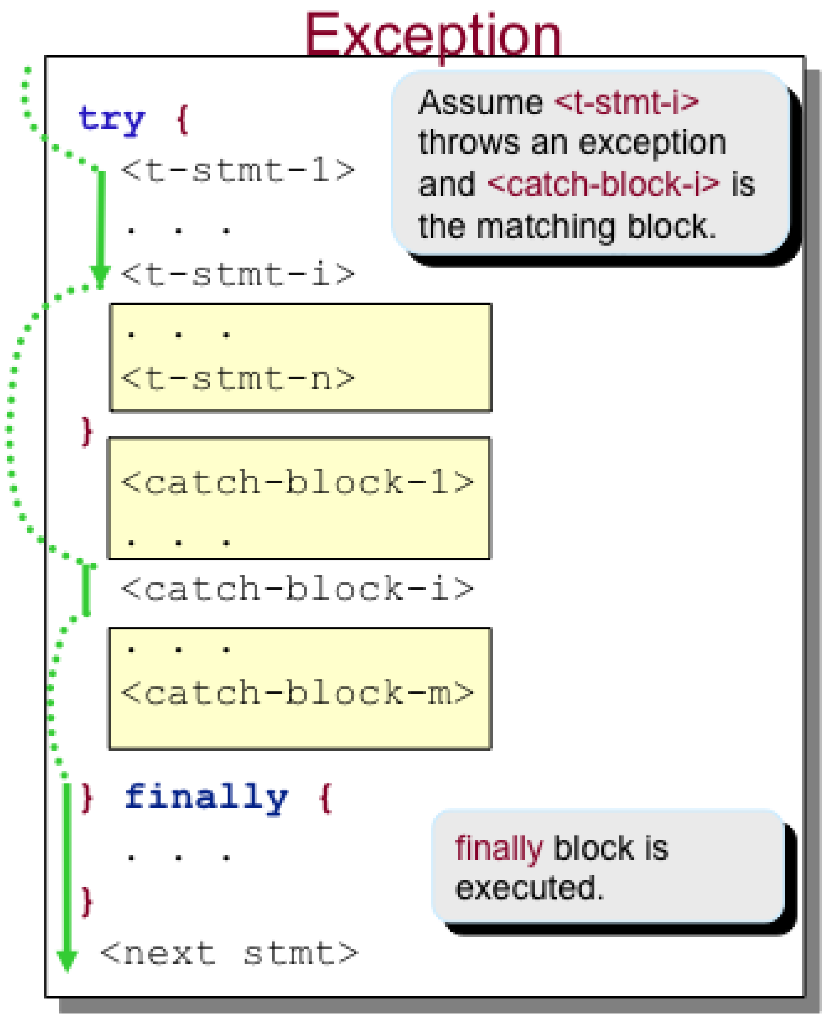


The finally Block

- 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

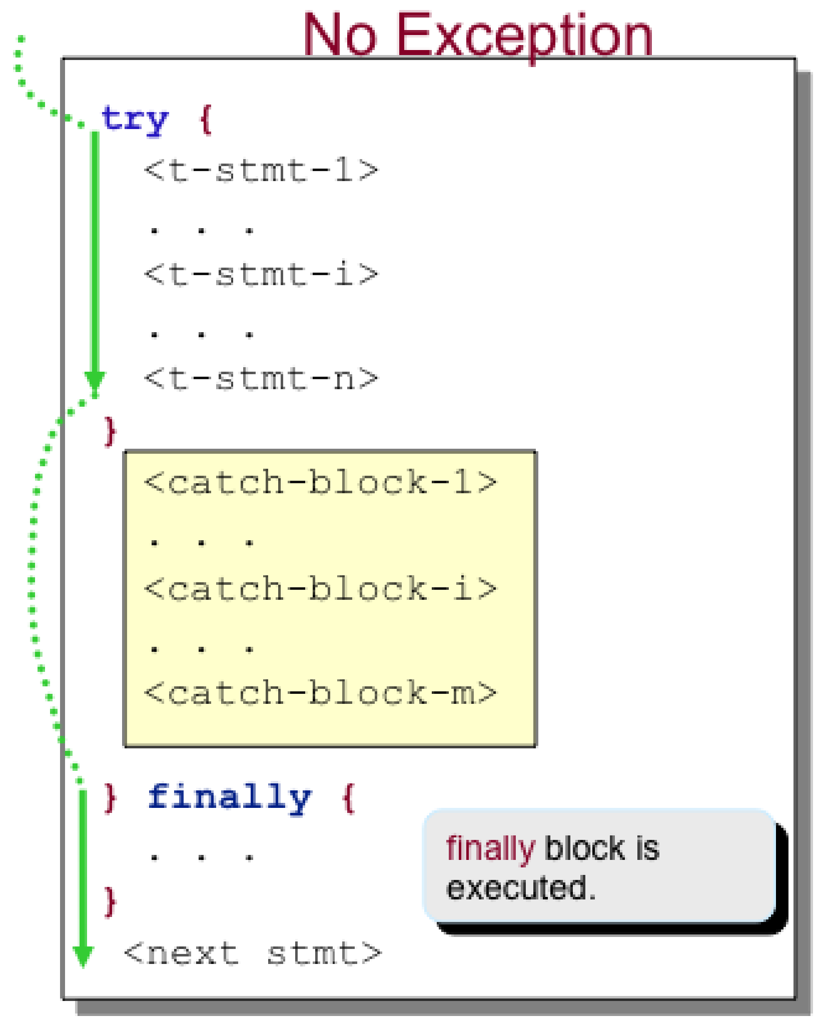


Exception

```
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.

try-with-resources statement

- Java SE 7 provides a shortcut to the code pattern:

```
open a resource
try
{
    work with the resource
}
finally
{
    close the resource
}
```

Try-with-resource

```
try (Resource res = ...)
{
    work with res
}
```

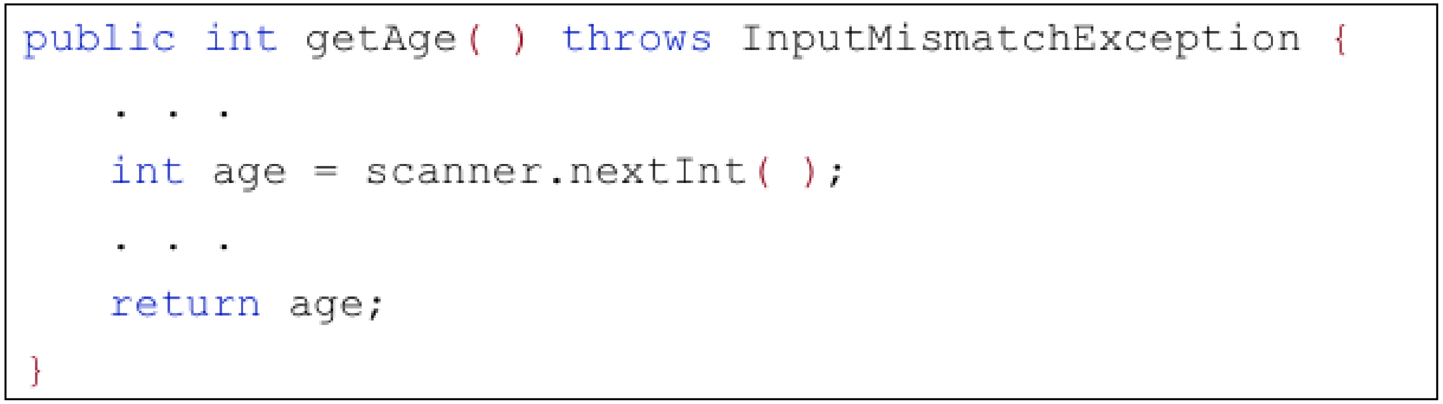
When the try block exits, then res.close() is called automatically.

```
import java.util.*;
import java.io.*;
public class NoTryWithSample {
    public static void main(String[] args) {
        Scanner in = null;
        PrintWriter out = null;
        try  {
            try {
                in = new Scanner(new FileInputStream("words.txt"));
                out = new PrintWriter("out.txt");
                while (in.hasNext())
                    out.println(in.next().toUpperCase());
            }
            catch (Exception e) {
                System.err.println(e.getMessage());
            }
        }
        finally {
            try {
                in.close();
                out.close();
            }
            catch (Exception e) {
                System.err.println(e.getMessage());
            }
        }
    }
}
```

```
import java.util.*;
import java.io.*;

class TryWithSample {
    public static void main(String[] args) {
        try
            (Scanner in = new Scanner(new FileInputStream("words.txt"));
             PrintWriter out = new PrintWriter("out.txt"))
        {
            while (in.hasNext())
                out.println(in.next().toUpperCase());
        }
        catch (FileNotFoundException e)
        {
            e.printStackTrace();
        }
    }
}
```

Propagating Exceptions

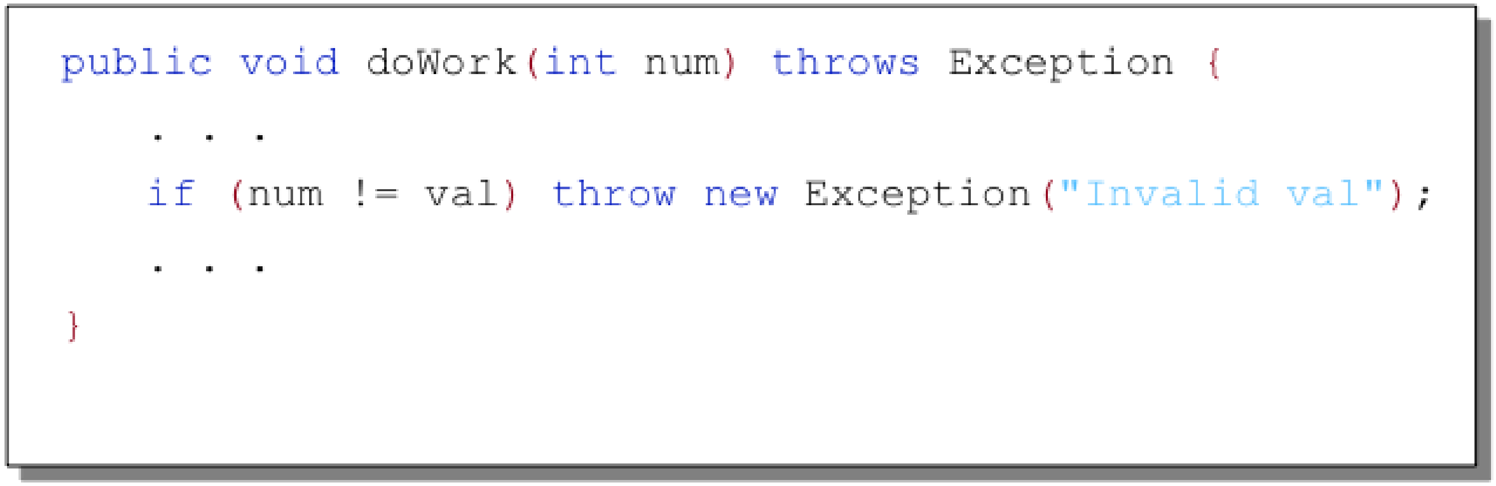
- Instead of catching a thrown exception by using the try-catch statement, we can propagate the thrown exception back to the caller of our method.
- The method header includes the reserved word throws.



```
public int getAge( ) throws InputMismatchException {
    . . .
    int age = scanner.nextInt( );
    . . .
    return age;
}
```

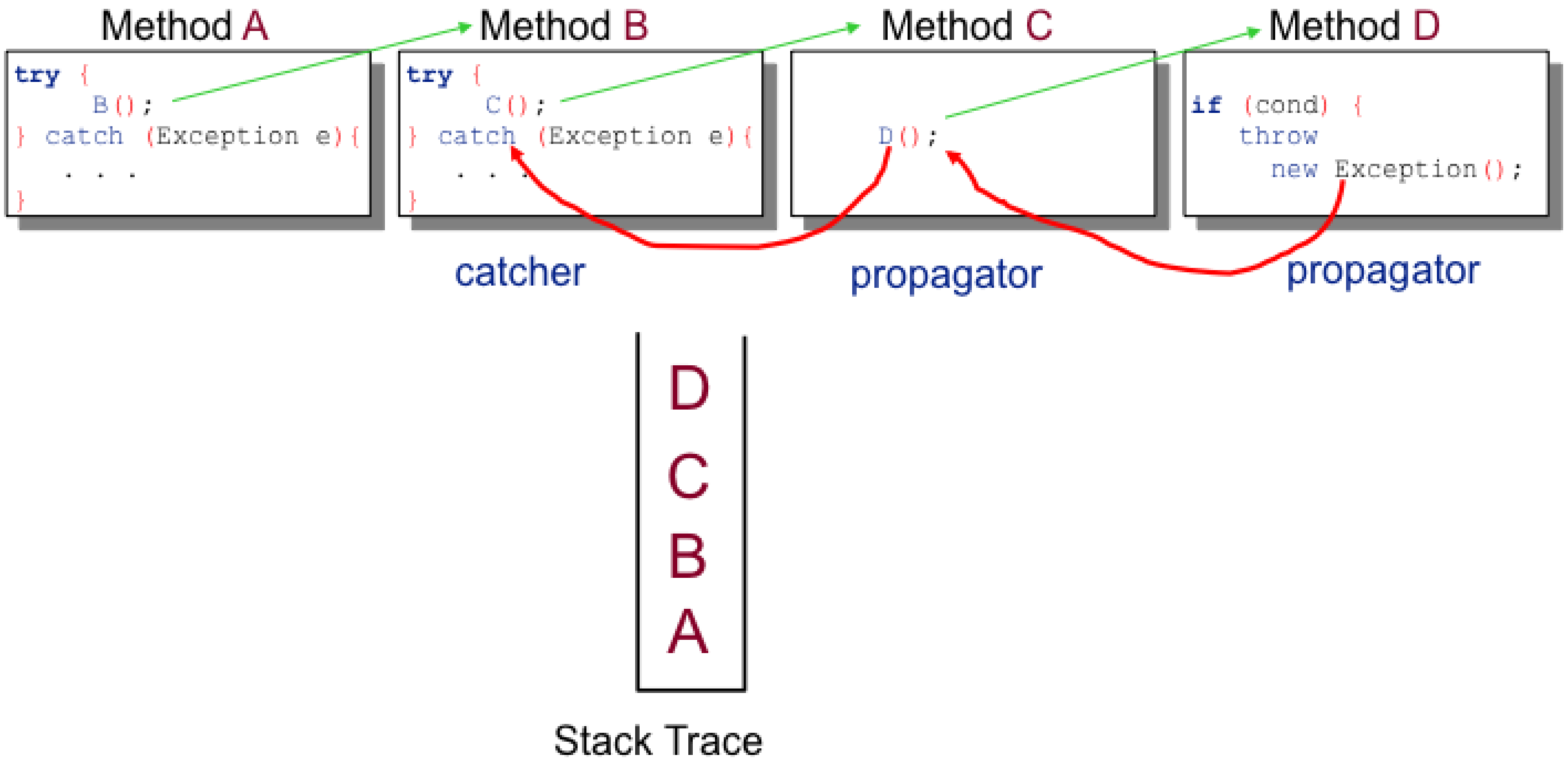
Throwing Exceptions

- 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");
    . . .
}
```

Sample call sequence



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

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

```
      D();
```

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

Classification of Exceptions

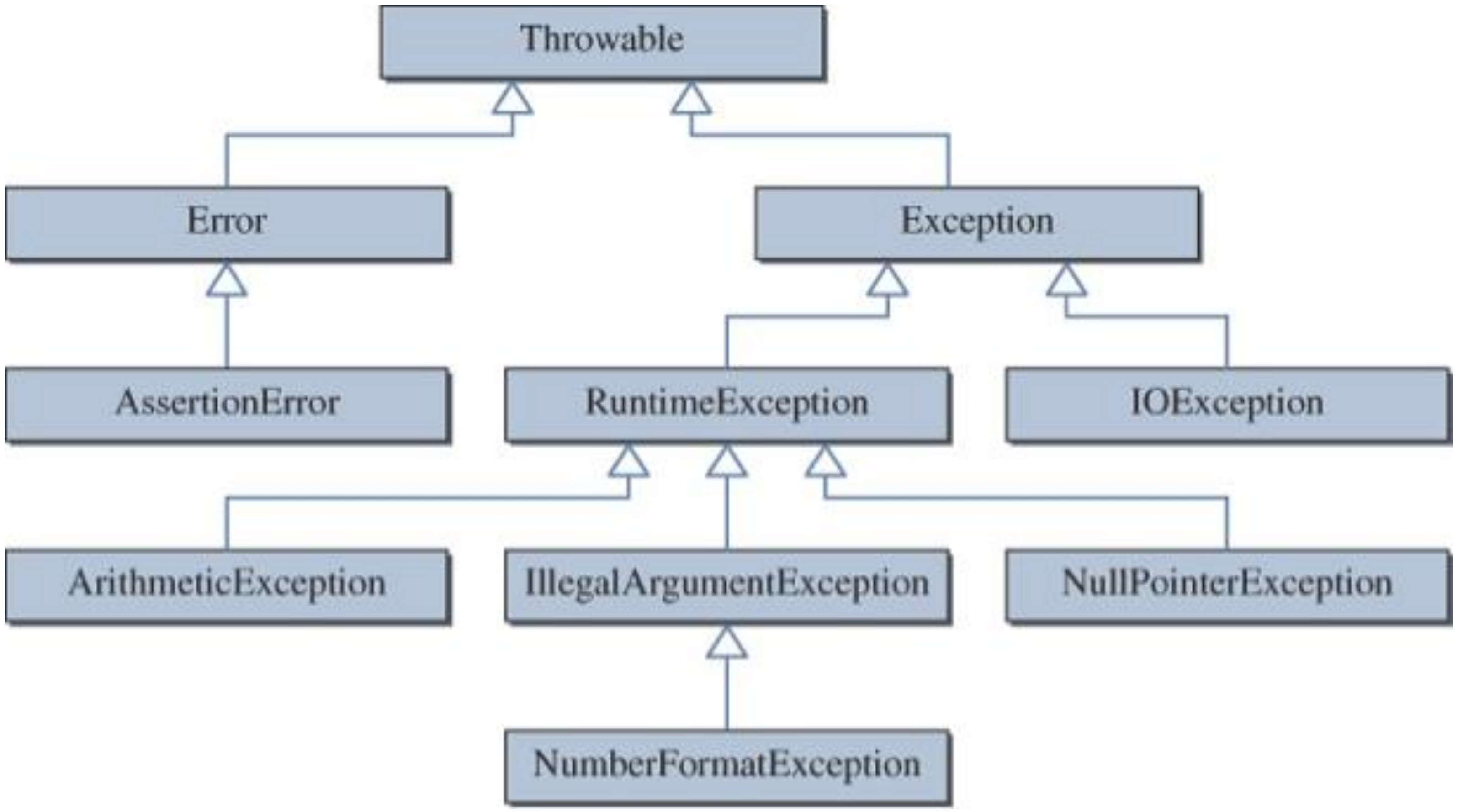
- All types of thrown errors are instances of the **Throwable** class or its subclasses.

- 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.

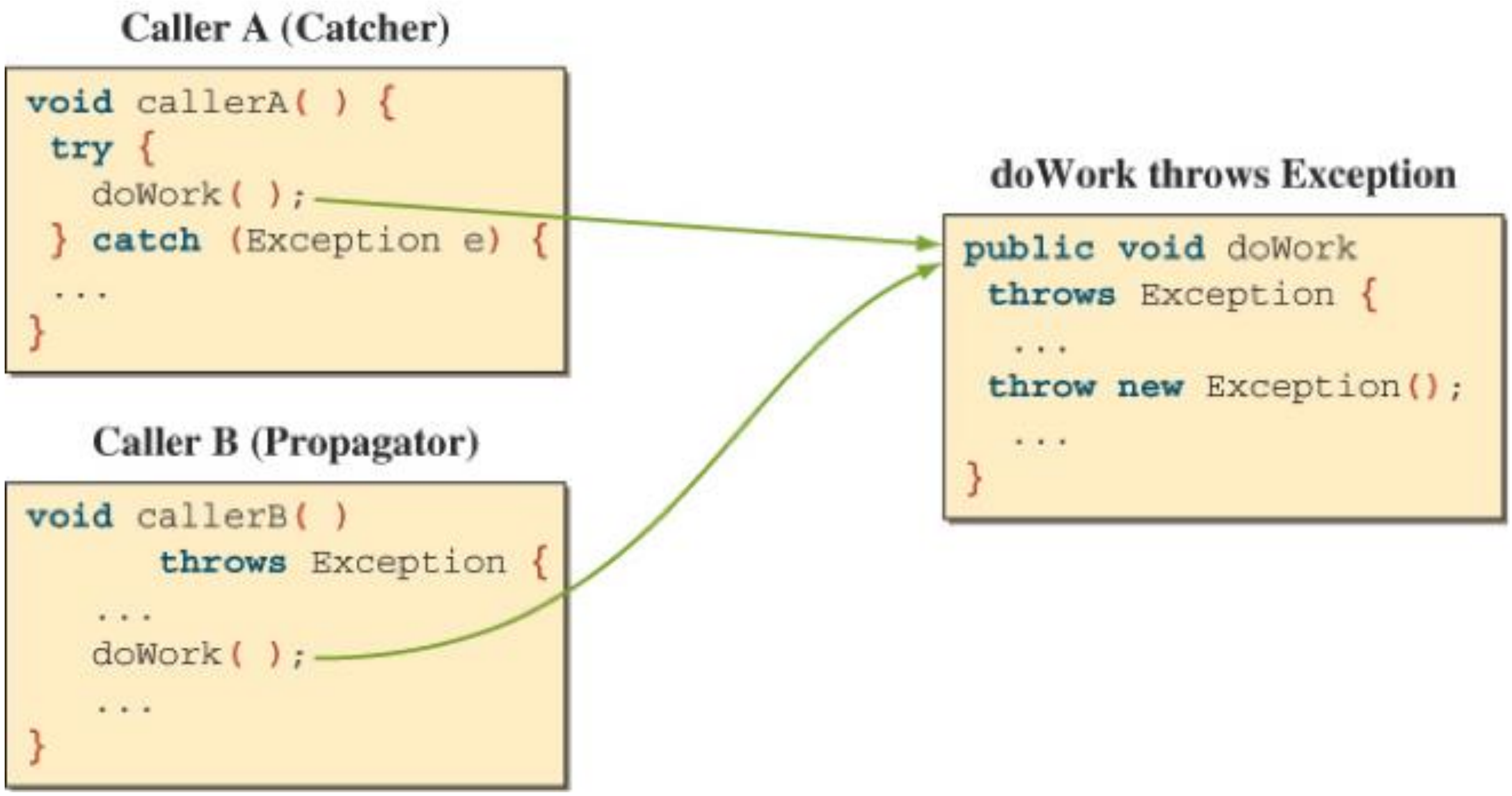
- All other exceptions are *unchecked*, or *runtime, exceptions*. As the name suggests, they are detected only at runtime.

Exception 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) {
  ...
}
```

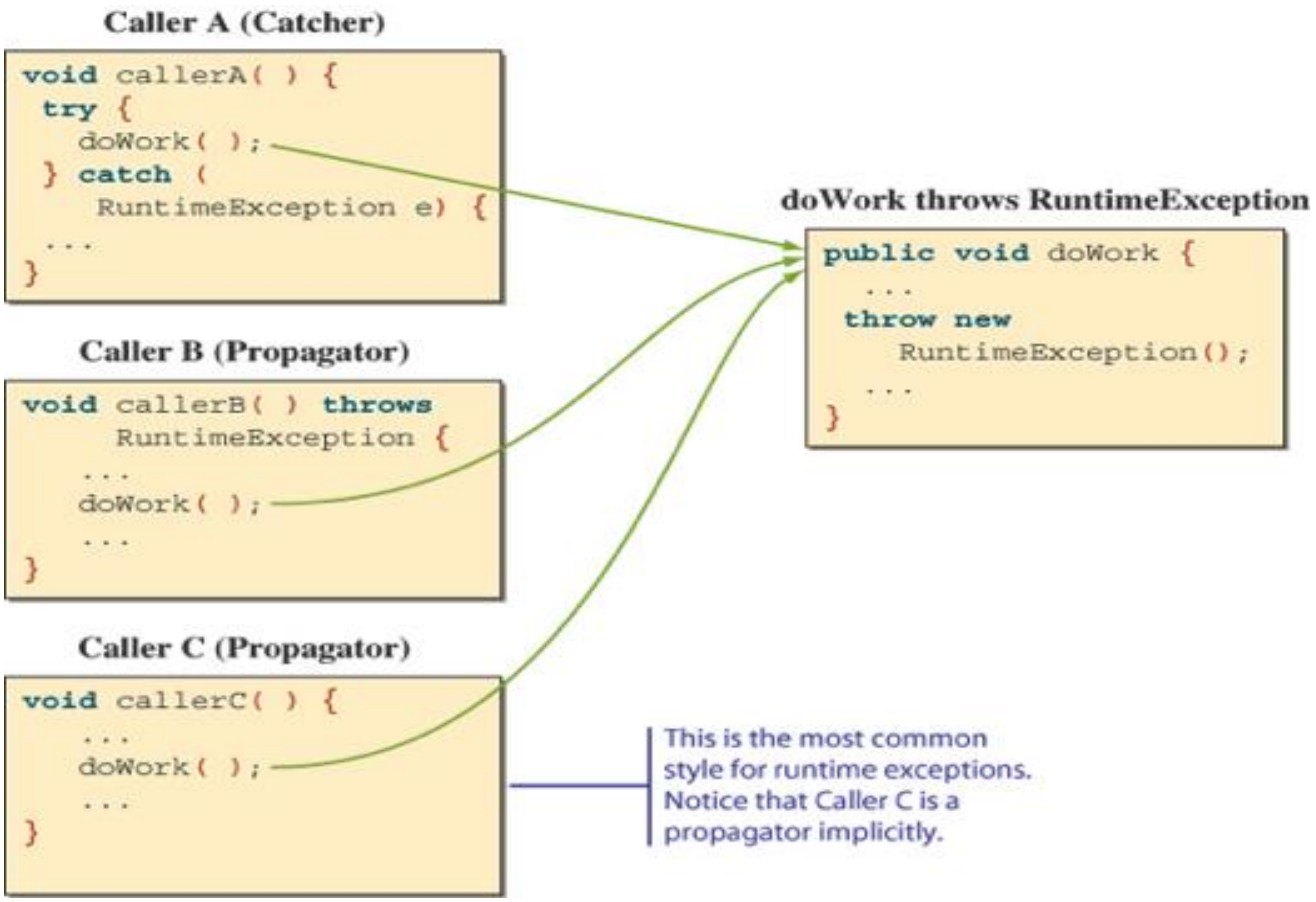
Caller B (Propagator)

```
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( );
   ...
}
```

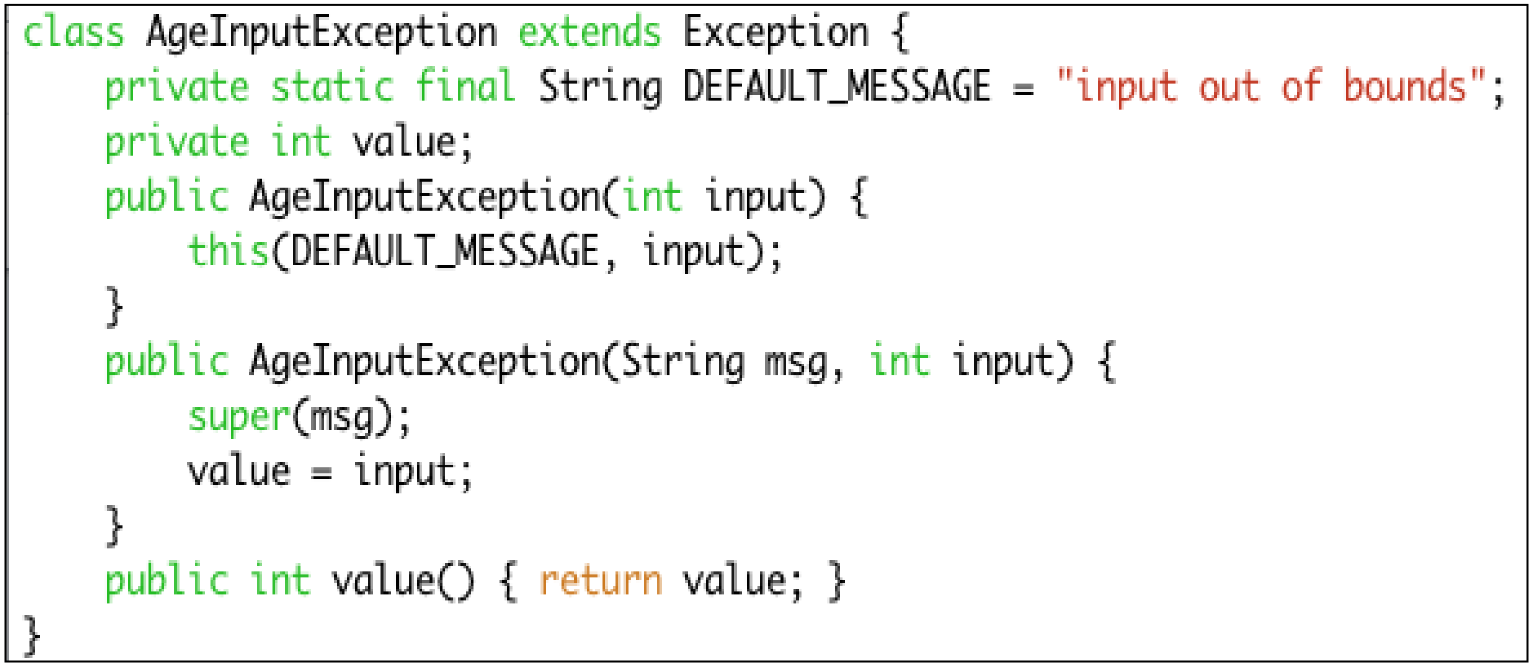
doWork throws RuntimeException

```
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



```
class AgeInputException extends Exception {
    private static final String DEFAULT_MESSAGE = "input out of bounds";
    private int value;
    public AgeInputException(int input) {
        this(DEFAULT_MESSAGE, input);
    }
    public AgeInputException(String msg, int input) {
        super(msg);
        value = input;
    }
    public int value() { return value; }
}
```