Chapter 5

Exceptions

CSC 113
King Saud University
College of Computer and Information Sciences
Department of Computer Science

Dr. S. HAMMAMI

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.

Introduction to Exception Handling

- No matter how well designed a program is, there is always the chance that some kind of error will arise during its execution.
- A well-designed program should include code to handle errors and other exceptional conditions when they arise.
- Sometimes the best outcome can be when nothing unusual happens
- However, the case where exceptional things happen must also be prepared for
  - Java exception handling facilities are used when the invocation of a method may cause something exceptional to occur

Introduction to Exception Handling

- Java library software (or programmer-defined code) provides a mechanism that signals when something unusual happens
  - This is called *throwing an exception*
- In another place in the program, the programmer must provide code that deals with the exceptional case
  - This is called *handling the exception*

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

- The avgFirstN() method expects that $N > 0$.
- If $N = 0$, a *divide-by-zero* error occurs in *avg/N*.

```
/**
  * Precondition:  N > 0
  * Postcondition: avgFirstN() equals the average of (1+2+…+N)
  */
public double avgFirstN(int N) {
    duuble sum = 0;
    for (int k = 1; k <= N; k++)
        sum += k;
    return sum/N;      // What if N is 0 ??
} // avgFirstN()
```

Bad Design: Doesn't guard against divide-by-0.

Not Catching Exceptions

```
class AgeInputVer1 {
   private int age;
   public void setAge(String s) {
      age = Integer.parseInt(s);
   }
   public int getAge() {
      return age;
   }
}
```

```
public class AgeInputMain1 {
 public static void main( String[] args ) {
  AgeInputVer1 P = new AgeInputVer1( );
  P.setAge("nine");
 System.out.println(P.getAge());
 }
}
```

Error message for invalid input

```
Exception in thread "main"
java.lang.NumberFormatException: For input string: "nine"
at java.lang.NumberFormatException.forInputString(Unknown
Source)
at java.lang.Integer.parseInt(Unknown Source)
at java.lang.Integer.parseInt(Unknown Source)
at AgeInputVer1.setAge(AgeInputVer1.java:5)
at AgeInputMain1.main(AgeInputMain1.java:8)
```

Not Catching Exceptions

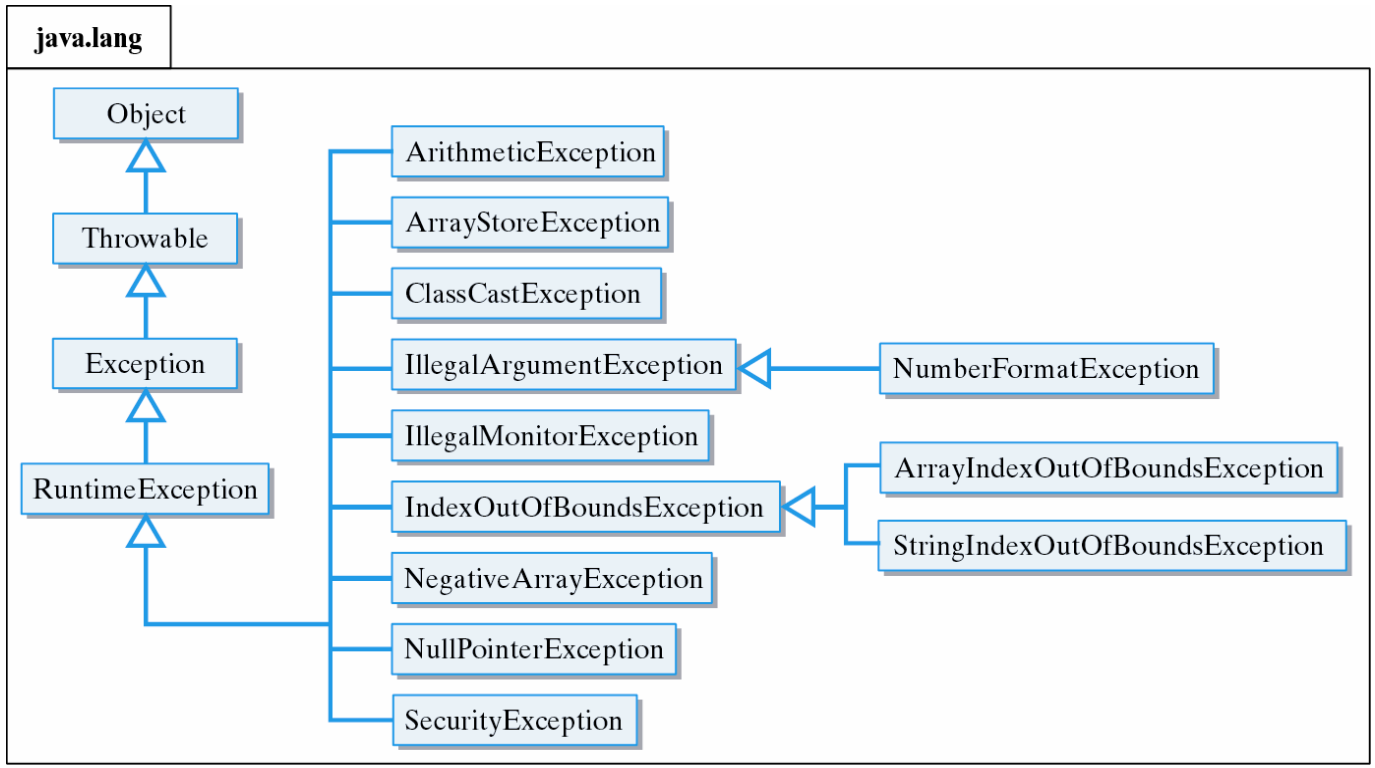
```
class AgeInputVer1 {
   private int age;
   public void setAge(String s) {
      age = Integer.parseInt(s);
   }
   public int getAge() {
      return age;
   }
}
```

```
public class AgeInputMain2 {
   public static void main( String[] args ) {
   AgeInputVer1 P = new AgeInputVer1( );
   P.setAge("9");
   System.out.println(P.getAge());
 }
}
```

9

Java's Exception Hierarchy

- *Unchecked exceptions*: belong to a subclass of RuntimeException and are not monitored by the compiler.



Some Important Exceptions

Class	Description
ArithmeticException	Division by zero or some other kind of arithmetic problem
ArrayIndexOutOfBounds-	An array index is less than zero or Exception greater than or equal to the array's length
FileNotFoundException	Reference to a unfound file IllegalArgumentException Method call with improper argument
IndexOutOfBoundsException	An array or string index out of bounds
NullPointerException	Reference to an object which has not been instantiated
NumberFormatException	Use of an illegal number format, such as when calling a method
StringIndexOutOfBoundsException	A String index less than zero or greater than or equal to the String's length

Catching an Exception

```
class AgeInputVer2 {
 private int age
 public void setAge(String s)
  {
    try {
        age = Integer.parseInt(s);
    } catch (NumberFormatException e){
        System.out.Println("age is invalid, Please enter digits only");
    }
  }
  public int getAge() {    return age;     }
}
```

try

catch

We are catching the number format exception, and the parameter e represents an instance of the **NumberFormatException** class

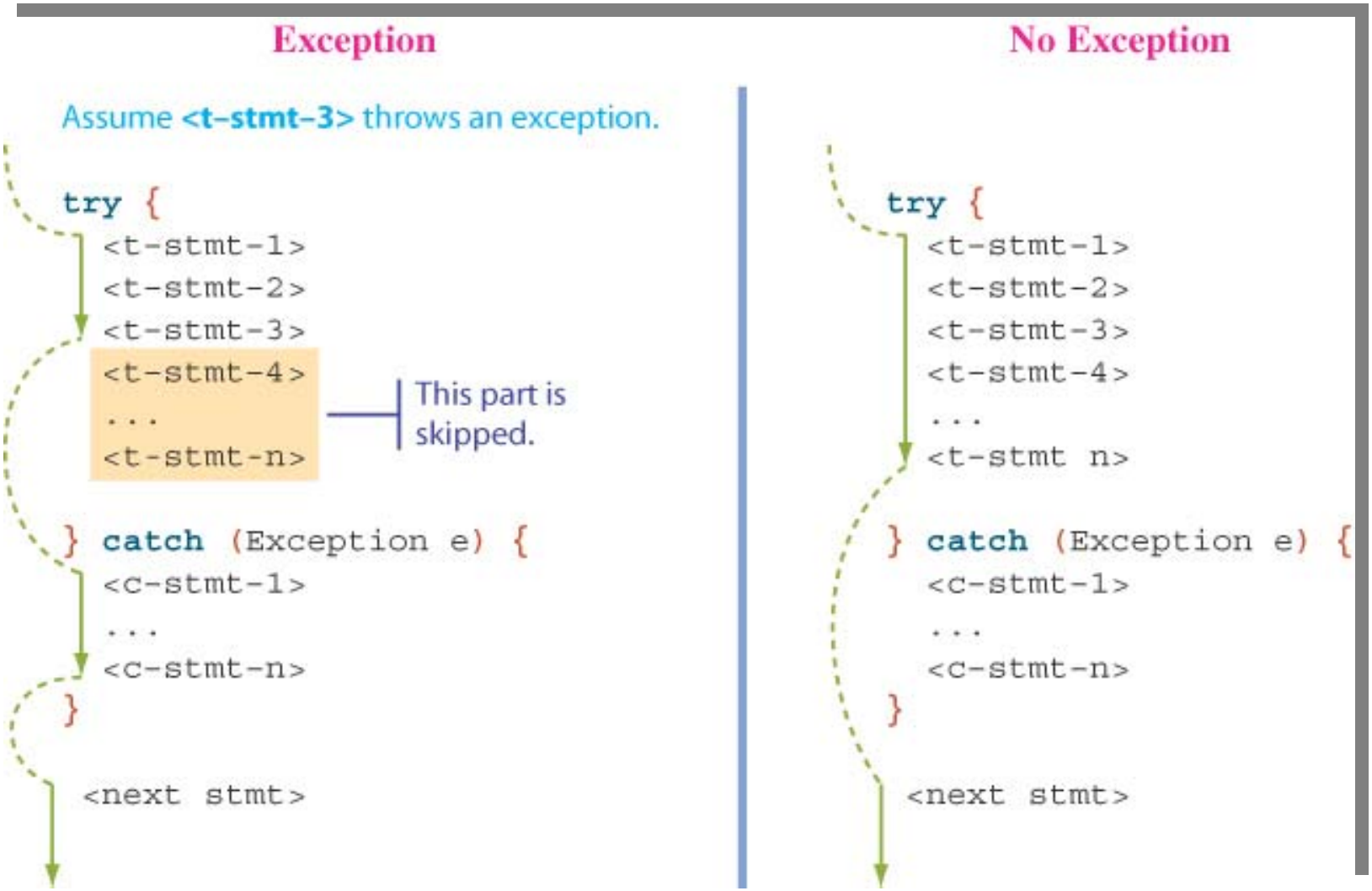
Catching an Exception

To accomplish this repetition, we will put the whole try-catch statement in side a loop:

```
import java.util.Scanner;
class AgeInputVer3 {
  private int age;
  public void setAge(String s) {
    String m =s;
    Scanner input = new Scanner(System.in);
    boolean ok = true;
     while (ok) {
       try {
          age = Integer.parseInt(m);
          ok = false;
       } catch (NumberFormatException e){
                System.out.println("age is invalid, Please enter digits only");
                m = input.next();
       }
   }
      public int getAge() {   return age;    }
}
```

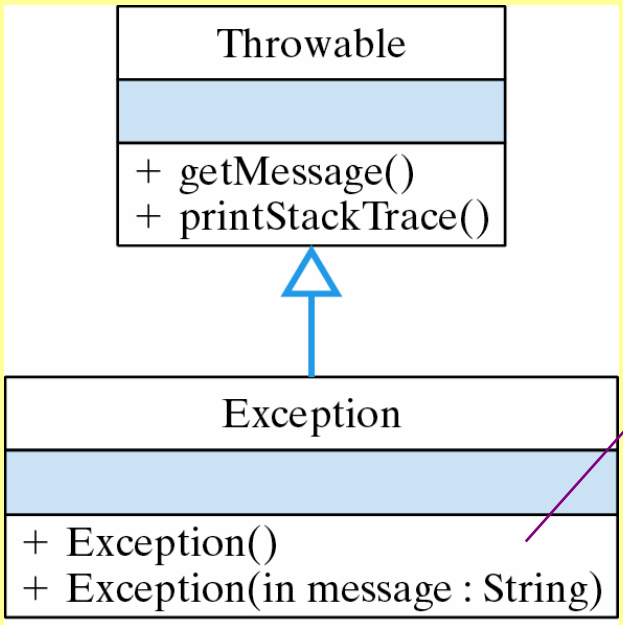
This statement is executed only if no exception is thrown by **parseInt**.

try-catch Control Flow



The Exception Class: Getting Information

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



Simple: only constructor methods.

The Exception Class: Getting Information

We are catching the number format exception, and the parameter e represents an instance of the **NumberFormatException** class

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

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

```
For input string: "nine"
java.lang.NumberFormatException: For input string: "nine"
at java.lang.NumberFormatException.forInputString(Unknown Source)
at java.lang.Integer.parseInt(Unknown Source)
at java.lang.Integer.parseInt(Unknown Source)
at AgeInputVer1.setAge(AgeInputVer1.java:11)
at AgeInputMain1.main(AgeInputMain1.java:8)
```

`try-throw-catch` Mechanism

throw new
***ExceptionClassName*(*PossiblySomeArguments*);**

- When an exception is thrown, the execution of the surrounding **try** block is stopped
  - Normally, the flow of control is transferred to another portion of code known as the **catch** block
- The value thrown is the argument to the **throw** operator, and is always an object of some exception class
  - The execution of a **throw** statement is called *throwing an exception*

`try-throw-catch` Mechanism

- A throw statement is similar to a method call:
  - throw new ExceptionClassName(SomeString);
  - In the above example, the object of class ExceptionClassName is created using a string as its argument
  - This object, which is an argument to the throw operator, is the exception object thrown
- Instead of calling a method, a throw statement calls a catch block

`try-throw-catch` Mechanism

- When an exception is thrown, the catch block begins execution
  - The catch block has one parameter
  - The exception object thrown is plugged in for the catch block parameter
- The execution of the catch block is called catching the exception, or handling the exception
  - Whenever an exception is thrown, it should ultimately be handled (or caught) by some catch block

`try-throw-catch` Mechanism

- When a **`try`** block is executed, two things can happen:
  1. No exception is thrown in the try block
     - The code in the **`try`** block is executed to the end of the block
     - The **`catch`** block is skipped
     - The execution continues with the code placed after the **`catch`** block

- 2. An exception is thrown in the try block and caught in the catch block

  The rest of the code in the try block is skipped

  Control is transferred to a following catch block (in simple cases)

  The thrown object is plugged in for the catch block parameter

  The code in the catch block is executed

  The code that follows that catch block is executed (if any)

```
public class CalcAverage {
  public double avgFirstN(int N ){
    double sum = 0;
    try {
       if (N <=0)
        throw new Exception("ERROR: Can't average 0 elements");
       for (int k = 1; k <= N; k++)
         sum += k;
       return sum/N;
  }
}

public class CalcAverage {
 public double avgFirstN(int N ){
  double sum = 0;
  try {
    if (N <0)
    throw new Exception("ERROR: Can't average negative elements");
    for (int k = 1; k <= N; k++)
      sum += k;
    return sum/N;
  }
  catch(ArithmeticException e)
  {
   System.out.println(e.getmessage());
   System.out.println(“N=Zero is an valid denomiattor, please try again ");
  }
  catch (Exception e) {System.out.println e.getmessage() + “Please enter
                                                        positive integer”);}
  }
}
```

Multiple catch Blocks

- A **try** block can potentially throw any number of exception values, and they can be of differing types
  - In any one execution of a **try** block, at most one exception can be thrown (since a throw statement ends the execution of the **try** block)
  - However, different types of exception values can be thrown on different executions of the **try** block

- Each **catch** block can only catch values of the exception class type given in the **catch** block heading

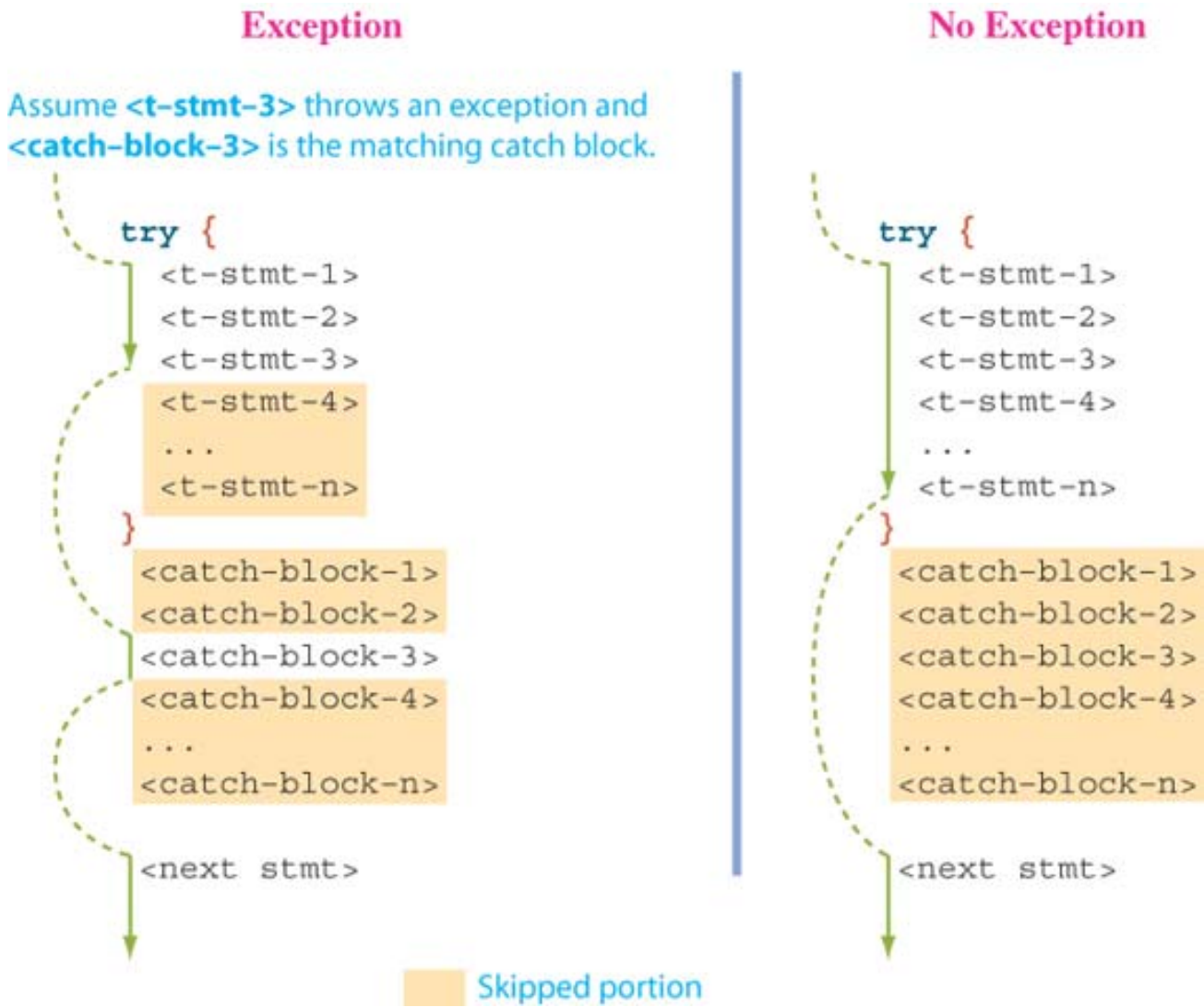
- Different types of exceptions can be caught by placing more than one **catch** block after a **try** block
  - Any number of **catch** blocks can be included, but they must be placed in the correct order

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 = x/y;
    . . .
} catch (NumberFormatException e){
    System.out.println("age is invalid, Please enter digits only");
} catch (ArithmeticException e){
    System.out.println("Illegal arithmetic operation");
}
```

Multiple catch Control Flow



Multiple catch Control Flow: Example

```
import java.util.*; //InputMismatchException;
//import java.util.ArithmeticException;
public class DividbyZero2
{   public static void main (String args[]) //throws ArithmeticException
   {  Scanner input = new Scanner(System.in);
      boolean done=false;
      do {
         try
         {System.out.print("Please enter an integer number :  ");
          int a =input.nextInt();
          System.out.print("Please enter an integer number :  ");
          int b =input.nextInt();
          int c=a/b;
          System.out.println("a ="+ a +"  b= "+b+ "   amd quotient ="+c);
          done=true;
         }
```

```
  catch (InputMismatchException var1 )
         {  System.out.println("Exception :"+ var1);
            System.out.println("please try again: ");
         }
  catch(ArithmeticException var2)
         {
            System.out.println("\nException :"+ var2);
            System.out.println("Zero is an valid denomiattor");
         }
 }while(!done);
}
}
```

```
Please enter an integer number :  car
Exception :java.util.InputMismatchException
You must enter an integer value, please try again:
Please enter an integer number :  14
Please enter an integer number :  0

Exception :java.lang.ArithmeticException: / by zero
Zero is an valid denomiattor, please try again
Please enter an integer number :  7
Please enter an integer number :  3
a =7  b= 3   amd quotient =2
```

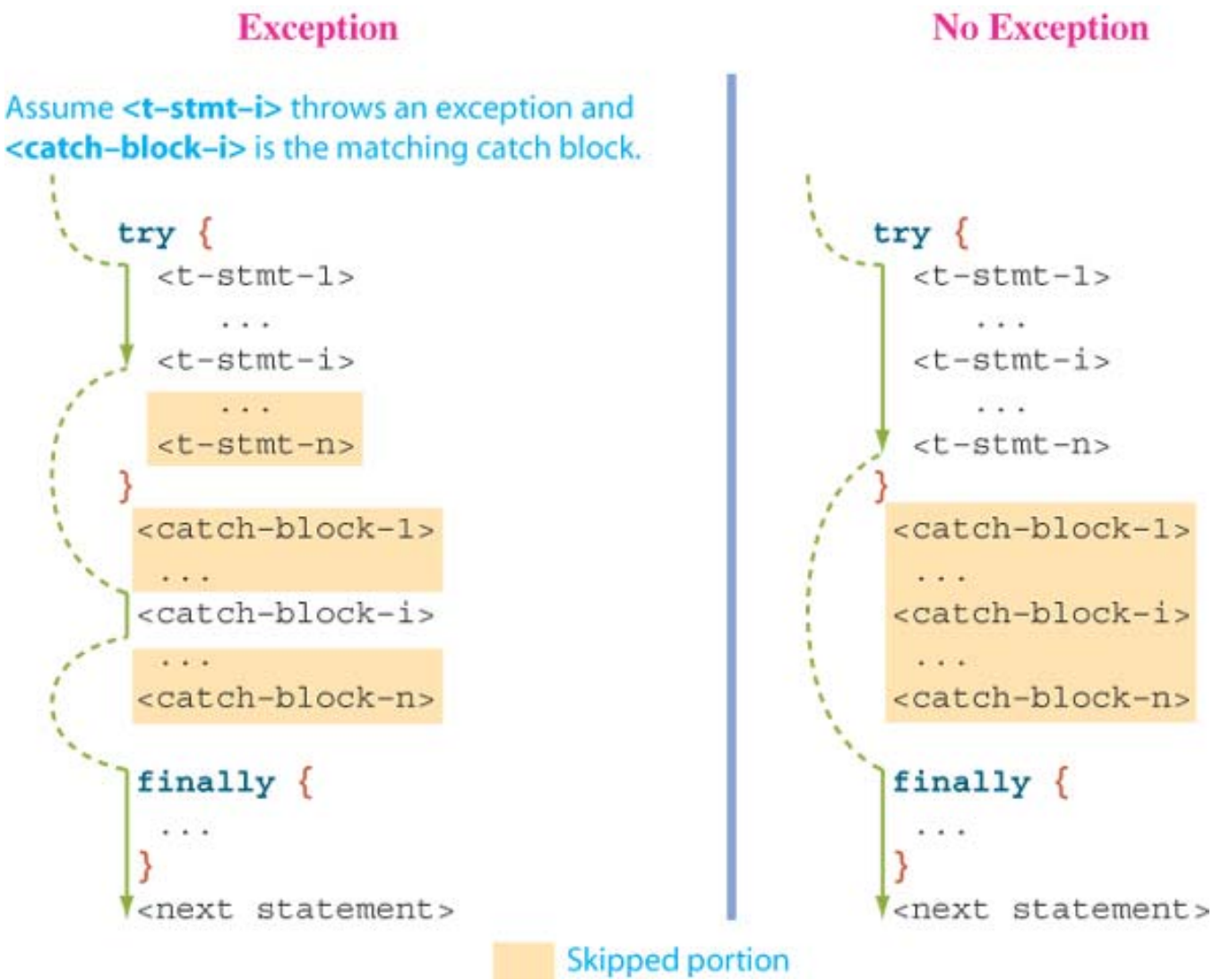
Pitfall: Catch the More Specific Exception First

- When catching multiple exceptions, the order of the **`catch`** blocks is important
  - When an exception is thrown in a **`try`** block, the **`catch`** blocks are examined in order
  - The first one that matches the type of the exception thrown is the one that is executed

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



try-catch-finally Control Flow

- If the **`try-catch-finally`** blocks are inside a method definition, there are three possibilities when the code is run:

  1. The **`try`** block runs to the end, no exception is thrown, and the **`finally`** block is executed

  2. An exception is thrown in the **`try`** block, caught in one of the **`catch`** blocks, and the **`finally`** block is executed

  3. An exception is thrown in the **`try`** block, there is no matching **`catch`** block in the method, the **`finally`** block is executed, and then the method invocation ends and the exception object is thrown to the enclosing method

Propagating Exceptions

Throwing an Exception in a Method

- Sometimes it makes sense to throw an exception in a method, but not catch it in the same method
  - Some programs that use a method should just end if an exception is thrown, and other programs should do something else
  - In such cases, the program using the method should enclose the method invocation in a try block, and catch the exception in a catch block that follows
- In this case, the method itself would not include try and catch blocks
  - However, it would have to include a throws clause

Declaring Exceptions in a `throws` Clause

- If a method can throw an exception but does not catch it, it must provide a warning
  - This warning is called a throws clause
  - The process of including an exception class in a throws clause is called declaring the exception

```
throws AnException  //throws clause
```

  - The following states that an invocation of aMethod could throw AnException

```
public void aMethod() throws AnException
```

Declaring Exceptions in a `throws` Clause

- If a method can throw more than one type of exception, then separate the exception types by commas

```
public void aMethod() throws AnException,AnotherException
```

- If a method throws an exception and does not catch it, then the method invocation ends immediately

Exception propagation and throws clause

Method **getDepend()** may throw a ***number format exception*** when converting a string to an integer, but it does not catch this exception.

The call to **getDepend()** occurs in the try block of method **main()**, so **main()** handles the exception in its catch block.

If main() did not have a catch block for number format exceptions, the exception would be handled by the JVM.

```
// postcondition: Returns int value of a numeric data string.
//   Throws an exception if string is not numeric.
public static int getDepend() throws NumberFormatException {
   String numStr = jnputnext();
   return Integer.parseInt(numStr);
}

// postcondition: Calls getDepend() and handles its exceptions.
public static void main(String[] args) {
  int children = 1;     // problem input, default is 1
  try {
       children = getDepend();
  }
  catch (NumberFormatException ex) {
     // Handle number format exception.
      System.out.println("Invalid integer“ + ex);
  }
}
```

Exception Thrower

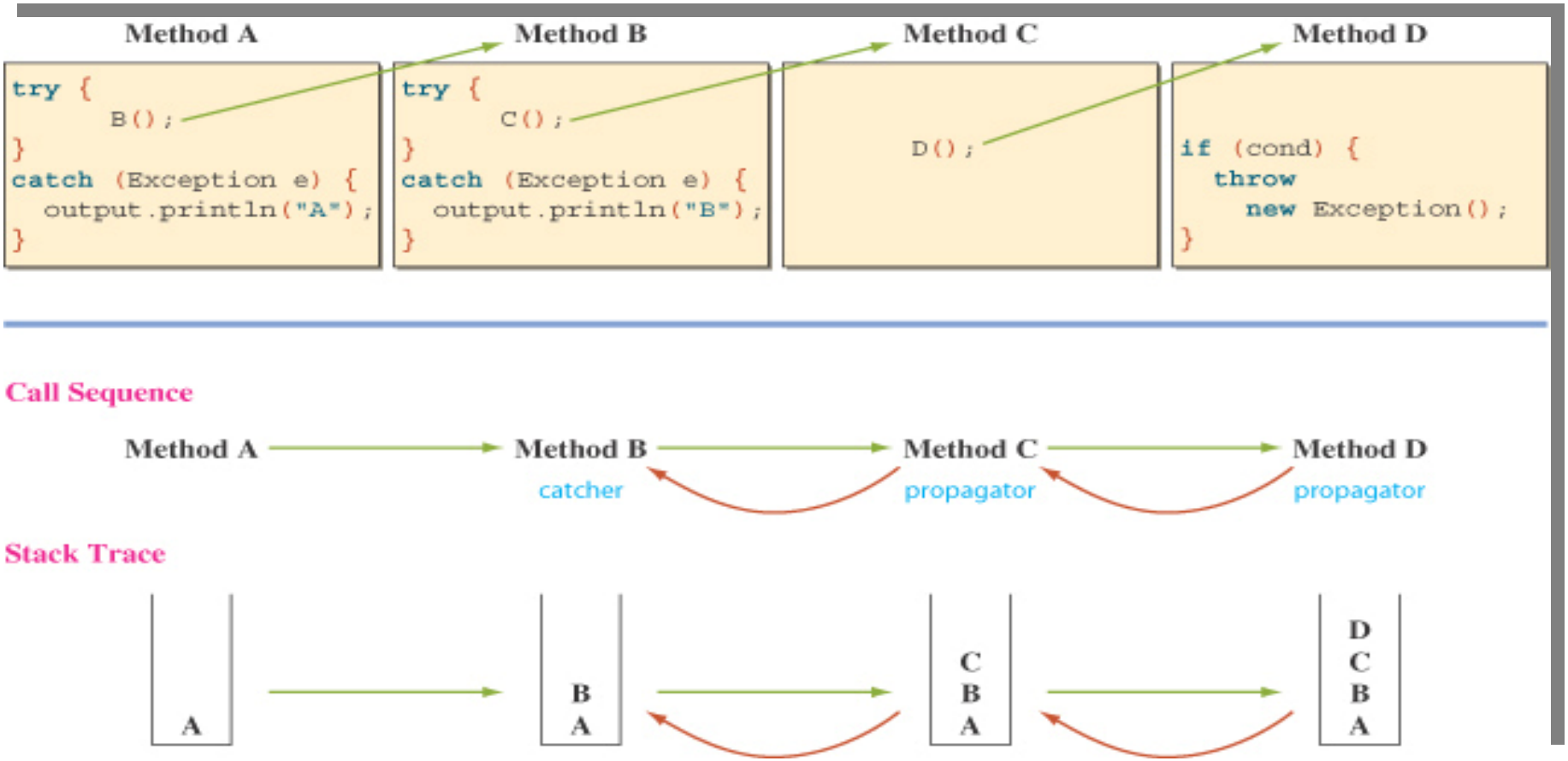
- When a method may throw an exception, either directly or indirectly, we call the method an *exception thrower*.

- 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 calls method B,
Method B calls method C,
Method C calls method D.

Every time a method is executed, the method's name is placed on top of the stack.

Sample Call Sequence

- When an exception is thrown, the system searches **down** the **stack** from the top, looking for the **first matching exception catcher**.
- Method D throws an exception, but **no** matching **catch block** exits in the method, so method D is an **exception propagator**.
- The system then checks method C. C is also an **exception propagator**.
- Finally, the system locates the matching catch block in method B, and therefore, method B is the catcher for the exception thrown by method D.
- Method A also includes the matching catch block, but it will not be executed because the thrown exception is already caught by method B and method B does not propagate this exception.

```
void C() throws Exception {
   ….
}
```

```
void D() throws Exception {
   ….
}
```

Example

Consider the Fraction class. The setDenominator method of the Fraction class
was defined as follows:

```
public void setDenominator (int d)
{
   if (d = = 0)
   {
      System.out.println("Fatal Error");
      System.exit(1);
   }
   denominator = d;
}
```

Throwing an exception is a much better approach. Here's the modified method that throws an IlleglArgumentException when the value of 0 is passed as an argument:

```
public void setDenominator (int d)
                        throws IlleglArgumentException
{
   if (d = = 0)
   {
      throw new IlleglArgumentException ("Fatal Error");

   }
   denominator = d;
}
```

Programmer-Defined Exception Classes

- A **`throw`** statement can throw an exception object of any exception class
- Instead of using a predefined class, *exception classes can be programmer-defined*
  - These can be tailored to carry the precise kinds of information needed in the **`catch`** block
  - A different type of exception can be defined to identify each different exceptional situation
- Every exception class to be defined must be a sub-class of some already defined exception class
  - It can be a sub-class of any exception class in the standard Java libraries, or of any programmer defined exception class
- Constructors are the most important members to define in an exception class
  - They must behave appropriately with respect to the variables and methods inherited from the base class
  - Often, there are no other members, except those inherited from the base class
- The following exception class performs these basic tasks only

A Programmer-Defined Exception Class

Display 9.3 A Programmer-Defined Exception Class

```
1  public class DivisionByZeroException extends Exception
2  {
3      public DivisionByZeroException()
4      {
5          super("Division by Zero!");
6      }

7      public DivisionByZeroException(String message)
8      {
9          super(message);
10     }
11 }
```

You can do more in an exception constructor, but this form is common.

super is an invocation of the constructor for the base class Exception.

Programmer-Defined Exception Class Guidelines

- Exception classes may be programmer-defined, but every such class must be a derived class of an already existing exception class
- The class **`Exception`** can be used as the base class, unless another class would be more suitable
- At least two constructors should be defined, sometimes more
- The exception class should allow for the fact that the method **`getMessage`** is inherited

Programmer-Defined Exceptions: AgeInputException

```
class AgeInputException extends Exception
{
   private static final String DEFAULT_MESSAGE = "Input out of bounds";
   private int lowerBoun, upperBound, value;
   public AgeInputException(int low, int high, int input)
{ this(DEFAULT_MESSAGE, low, high, input);   }
   public AgeInputException(String msg, int low, int high, int input)
    {     super(msg);
            if (low > high)    throw new IllegalArgumentException();
            lowerBound = low;   upperBound = high;    value      = input;
    }
    public int lowerBound()  {  return lowerBound;}
   public int upperBound()   {return  upperBound;}
    public int value() {  return value;}
}
```

Class AgeInputVer5 Uses AgeInputException

```
import java.util.Scanner;
class AgeInputVer5 {
   private static final String DEFAULT_MESSAGE = "Your age:";
   private static final int DEFAULT_LOWER_BOUND = 0;
   private static final int DEFAULT_UPPER_BOUND = 99;
   private int lowerBound, upperBound;
   public AgeInputVer5( ) throws IllegalArgumentException {
      setBounds(DEFAULT_LOWER_BOUND, DEFAULT_UPPER_BOUND);
   }
   public AgeInputVer5(int low, int high)   throws IllegalArgumentException
   {
      if (low > high)
      {    throw new IllegalArgumentException( "Low (" + low + ") was " +
             "larger than high(" + high + ")");
      } else setBounds(low, high);
   }
  public int getAge() throws AgeInputException
 {
 return      getAge(DEFAULT_MESSAGE);
   }
```

```
public int getAge(String prompt) throws AgeInputException  {
        Scanner T = new Scanner(System.in);
        String inputStr;    int    age;
        while (true)  {
           inputStr  = prompt;
           try
           {
              age = Integer.parseInt(inputStr);
              if (age < lowerBound || age > upperBound) {
                   throw new AgeInputException("Input out of bound  ",
                                        lowerBound, upperBound, age);
            }
            return age; //input okay so return the value & exit
           } catch (NumberFormatException e) {
              System.out.println("\n"+ inputStr + " is invalid age.");
              System.out.print("Please enter age as an integer value : ");
              prompt = T.next()+T.nextLine();
           }        } }
    private void setBounds(int low, int high) { lowerBound = low;   upperBound = high;
    }}
```

Main Using throws

```
public class TestAgeInputUsingThrows {
   public static void main( String[] args ) throws AgeInputException  {
      int   entrantAge=0;
      AgeInputVer5 input = new AgeInputVer5(25, 50);
      entrantAge   = input.getAge("Thirty");
       System.out.println("Input Okay ");  }  }
```

```
Thirty is invalid age.
Please enter age as an integer value : fourty

fourty is invalid age.
Please enter age as an integer value : 40
Input Okay
```

```
Thirty is invalid age.
Please enter age as an integer value : fourty

fourty is invalid age.
Please enter age as an integer value : 55
Exception in thread "main" AgeInputException: Input out of bound
at AgeInputVer5.getAge(AgeInputVersion5.java:42)
at TestAgeInputVer5.main(TestAgeInputVer5.java:7)
```

Main Using try-catch

```
public class Test2AgeInput {
   public static void main( String[] args )  {
      int   entrantAge;
      try   {
          AgeInputVer5 input = new AgeInputVer5(25, 50);
          entrantAge   = input.getAge("Thirty");
           System.out.println("Input Okay ");
       }
       catch (AgeInputException e)  {
       System.out.println("Error: " + e.value() + " is entered. It is " +  "outside the valid range of [" +
         e.lowerBound() +", " + e.upperBound() + "]"); }     }  }
```

```
Thirty is invalid age.
Please enter age as an integer value : fourty

fourty is invalid age.
Please enter age as an integer value : 40
Input Okay
```

```
Thirty is invalid age.
Please enter age as an integer value : fourty

fourty is invalid age.
Please enter age as an integer value : 55
Error: 55 is entered. It is outside the valid range of [25, 50]
```