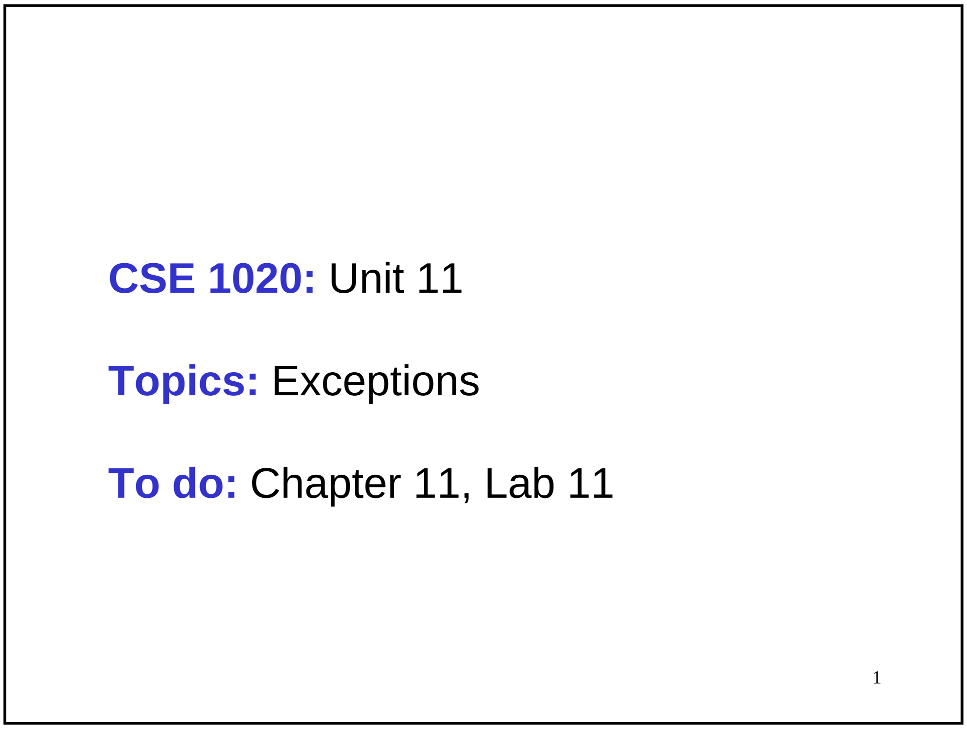
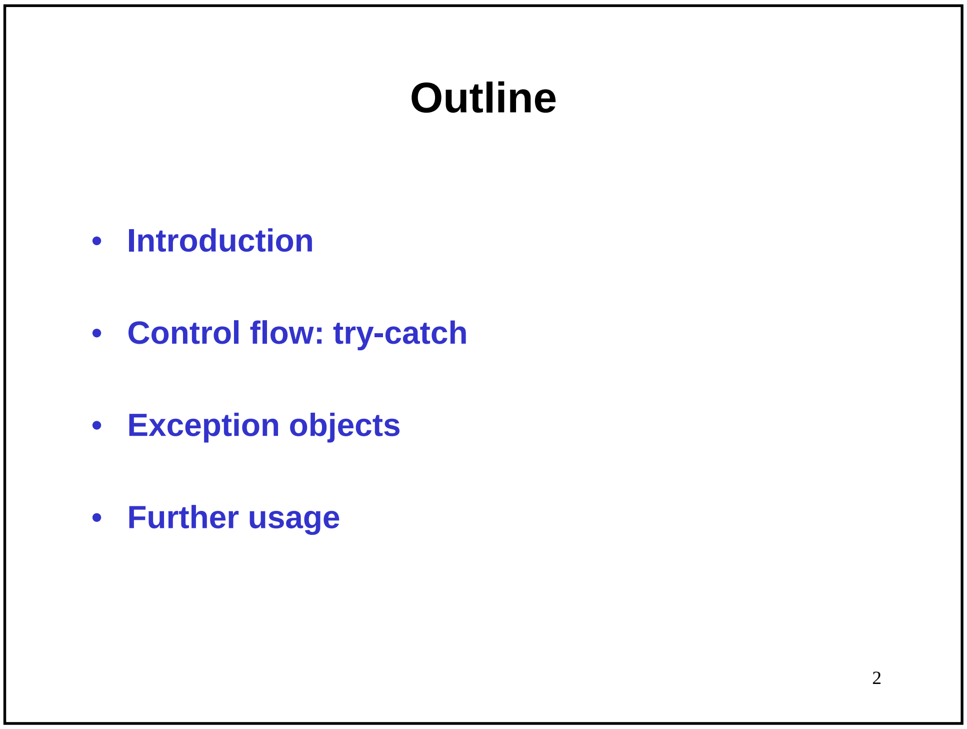


CSE 1020: Unit 11

Topics: Exceptions

To do: Chapter 11, Lab 11



Outline

- **Introduction**
- **Control flow: try-catch**
- **Exception objects**
- **Further usage**

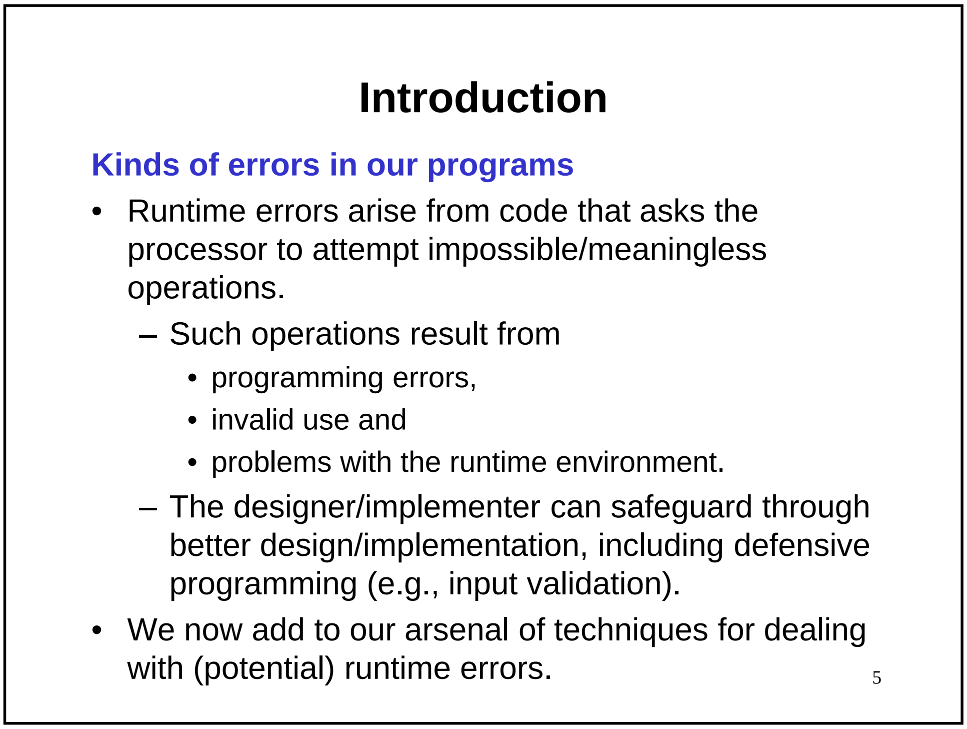
Outline

- **Introduction**
- **Control flow: try-catch**
- **Exception objects**
- **Further usage**

Introduction

Kinds of errors in our programs

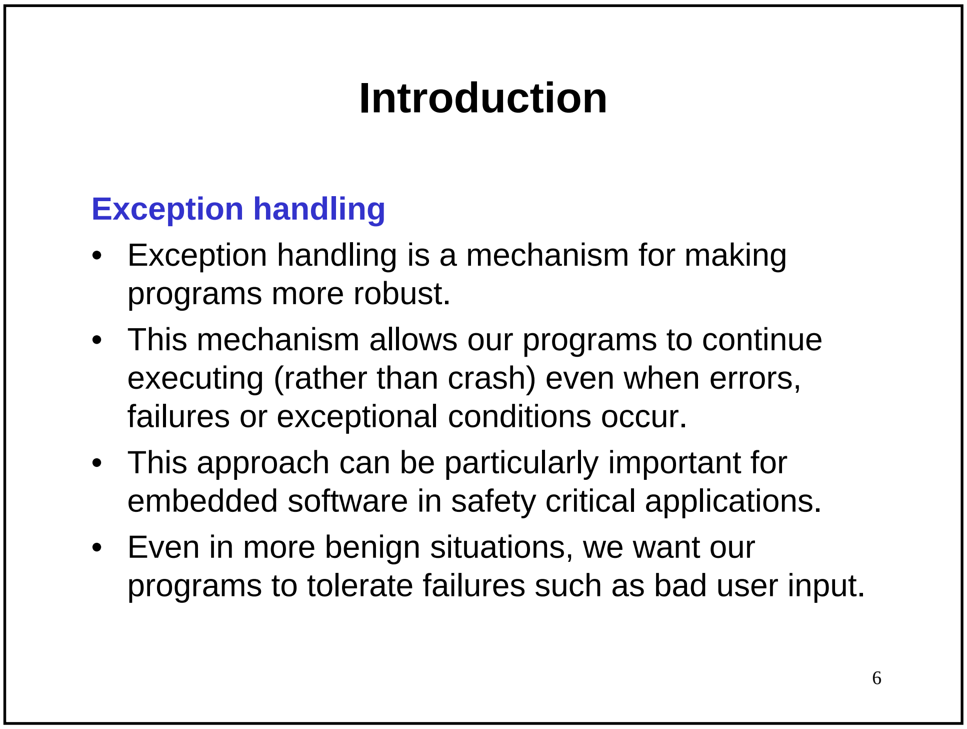
- Syntax errors arise from illegal (e.g., Java) code.
  - The compiler provides safeguards.
- Logic errors arise from conceptually incorrect code.
  - The designer/implementer must provide safeguard through clear thinking.



Introduction

Kinds of errors in our programs

- Runtime errors arise from code that asks the processor to attempt impossible/meaningless operations.
  - Such operations result from
    - programming errors,
    - invalid use and
    - problems with the runtime environment.
  - The designer/implementer can safeguard through better design/implementation, including defensive programming (e.g., input validation).
- We now add to our arsenal of techniques for dealing with (potential) runtime errors.

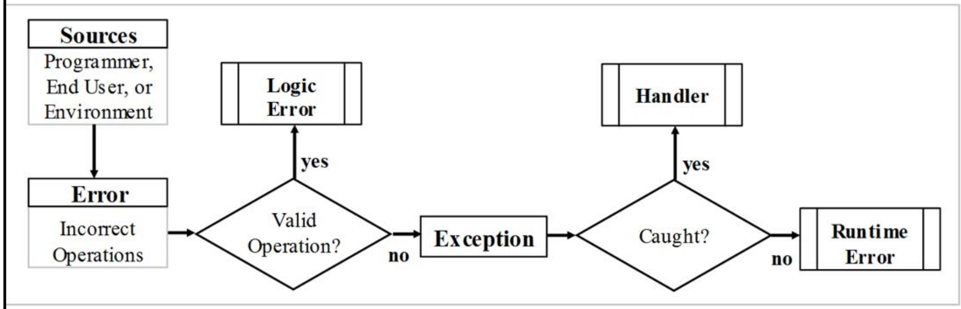


Introduction

Exception handling

- Exception handling is a mechanism for making programs more robust.
- This mechanism allows our programs to continue executing (rather than crash) even when errors, failures or exceptional conditions occur.
- This approach can be particularly important for embedded software in safety critical applications.
- Even in more benign situations, we want our programs to tolerate failures such as bad user input.

Introduction



Sources

Programmer, End User, or Environment

Error

Incorrect Operations

Valid Operation?

yes

Logic Error

no

Exception

Caught?

yes

Handler

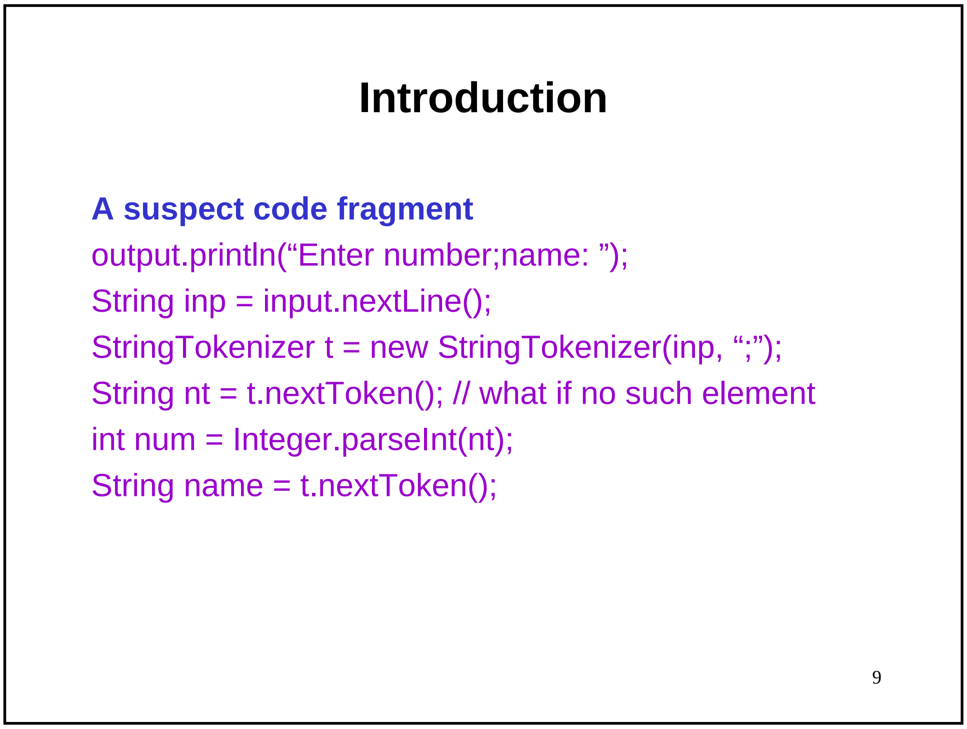
no

Runtime Error

Introduction

A suspect code fragment

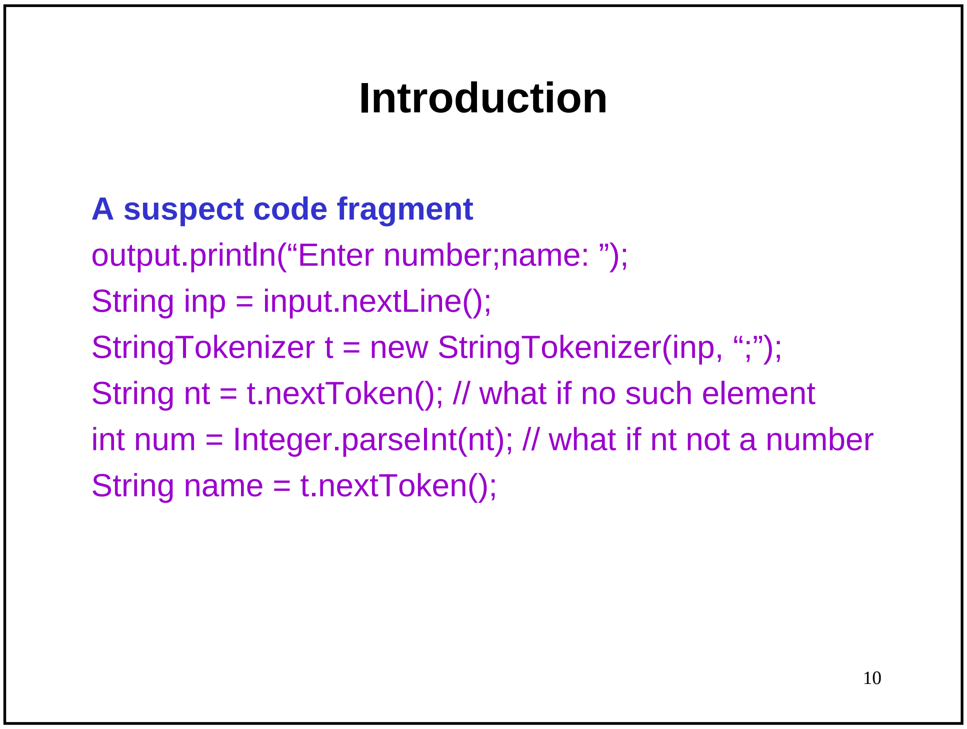
```
output.println(“Enter number;name: ”);
String inp = input.nextLine();
StringTokenizer t = new StringTokenizer(inp, “;”);
String nt = t.nextToken();
int num = Integer.parseInt(nt);
String name = t.nextToken();
```



Introduction

A suspect code fragment

```
output.println(“Enter number;name: ”);
String inp = input.nextLine();
StringTokenizer t = new StringTokenizer(inp, “;”);
String nt = t.nextToken(); // what if no such element
int num = Integer.parseInt(nt);
String name = t.nextToken();
```



Introduction

A suspect code fragment

```
output.println(“Enter number;name: ”);
String inp = input.nextLine();
StringTokenizer t = new StringTokenizer(inp, “;”);
String nt = t.nextToken(); // what if no such element
int num = Integer.parseInt(nt); // what if nt not a number
String name = t.nextToken();
```

Introduction

A suspect code fragment

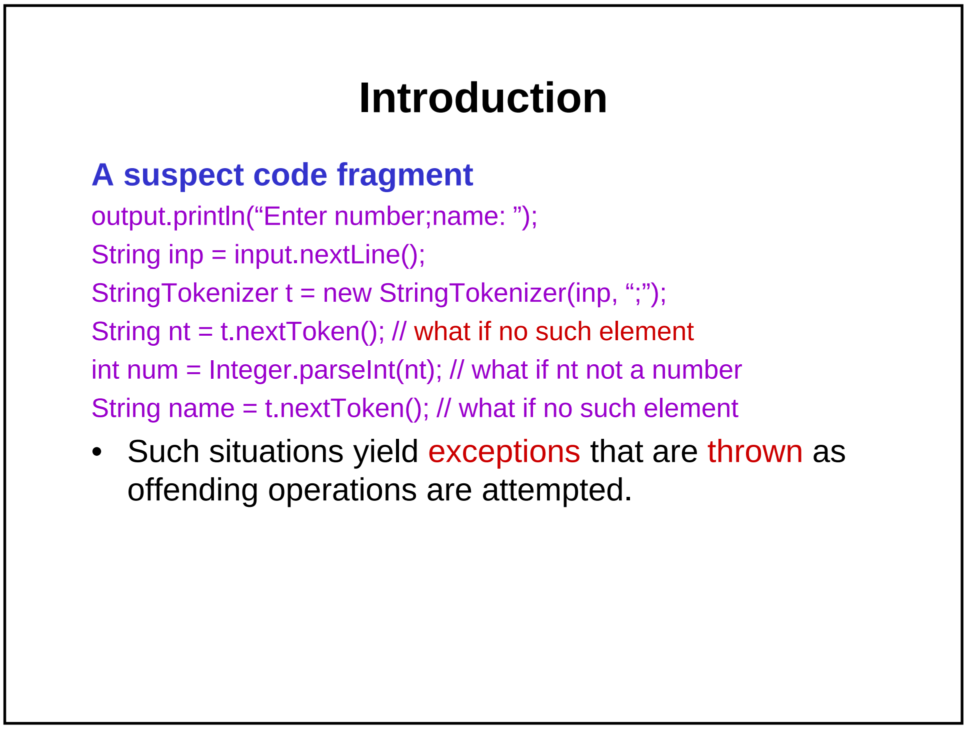
```
output.println(“Enter number;name: ”);
String inp = input.nextLine();
StringTokenizer t = new StringTokenizer(inp, “;”);
String nt = t.nextToken(); // what if no such element
int num = Integer.parseInt(nt); // what if nt not a number
String name = t.nextToken(); // what if no such element
```

Introduction

A suspect code fragment

```
output.println(“Enter number;name: ”);
String inp = input.nextLine();
StringTokenizer t = new StringTokenizer(inp, “;”);
String nt = t.nextToken(); // what if no such element
int num = Integer.parseInt(nt); // what if nt not a number
String name = t.nextToken(); // what if no such element
```

- Such situations yield exceptions that are thrown as offending operations are attempted.

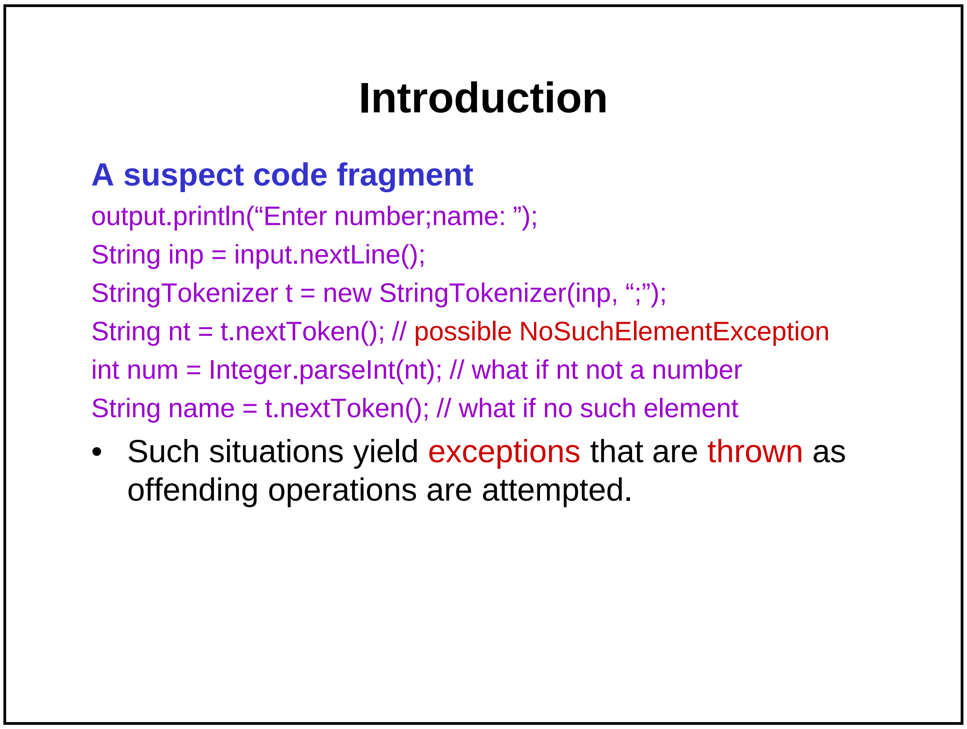


Introduction

A suspect code fragment

```
output.println(“Enter number;name: ”);
String inp = input.nextLine();
StringTokenizer t = new StringTokenizer(inp, “;”);
String nt = t.nextToken(); // what if no such element
int num = Integer.parseInt(nt); // what if nt not a number
String name = t.nextToken(); // what if no such element
```

- Such situations yield exceptions that are thrown as offending operations are attempted.



Introduction

A suspect code fragment

```
output.println(“Enter number;name: ”);
String inp = input.nextLine();
StringTokenizer t = new StringTokenizer(inp, “;”);
String nt = t.nextToken(); // possible NoSuchElementException
int num = Integer.parseInt(nt); // what if nt not a number
String name = t.nextToken(); // what if no such element
```

- Such situations yield exceptions that are thrown as offending operations are attempted.

Introduction

A suspect code fragment

```
output.println(“Enter number;name: ”);
String inp = input.nextLine();
StringTokenizer t = new StringTokenizer(inp, “;”);
String nt = t.nextToken(); // possible NoSuchElementException
int num = Integer.parseInt(nt); // what if nt not a number
String name = t.nextToken(); // what if no such element
```

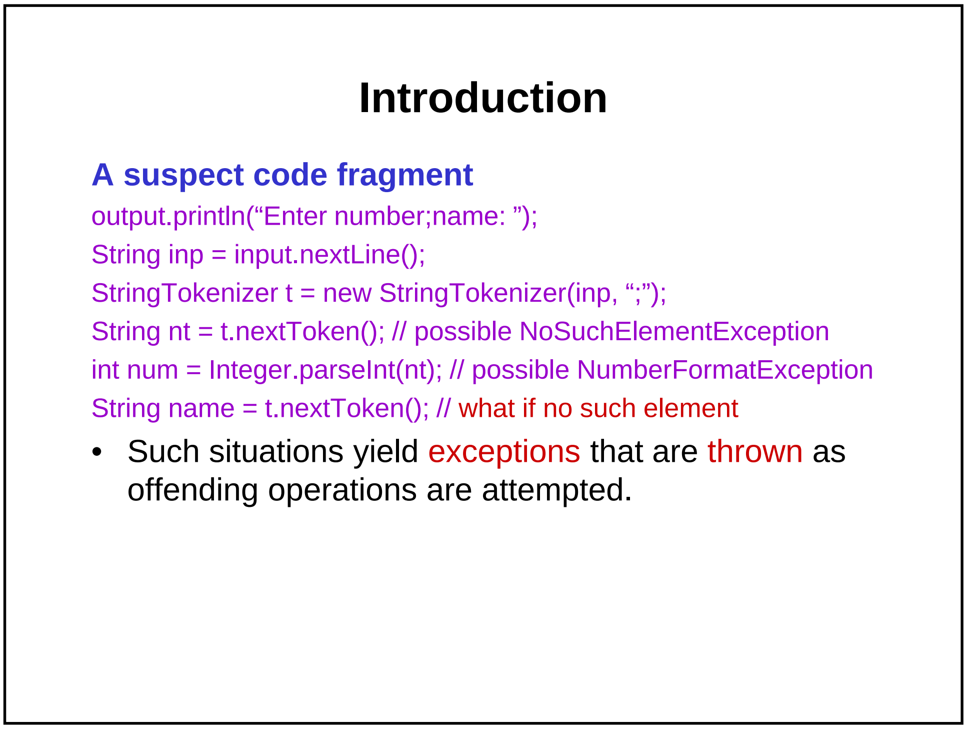
- Such situations yield exceptions that are thrown as offending operations are attempted.

Introduction

A suspect code fragment

```
output.println(“Enter number;name: ”);
String inp = input.nextLine();
StringTokenizer t = new StringTokenizer(inp, “;”);
String nt = t.nextToken(); // possible NoSuchElementException
int num = Integer.parseInt(nt); // possible NumberFormatException
String name = t.nextToken(); // what if no such element
```

- Such situations yield exceptions that are thrown as offending operations are attempted.

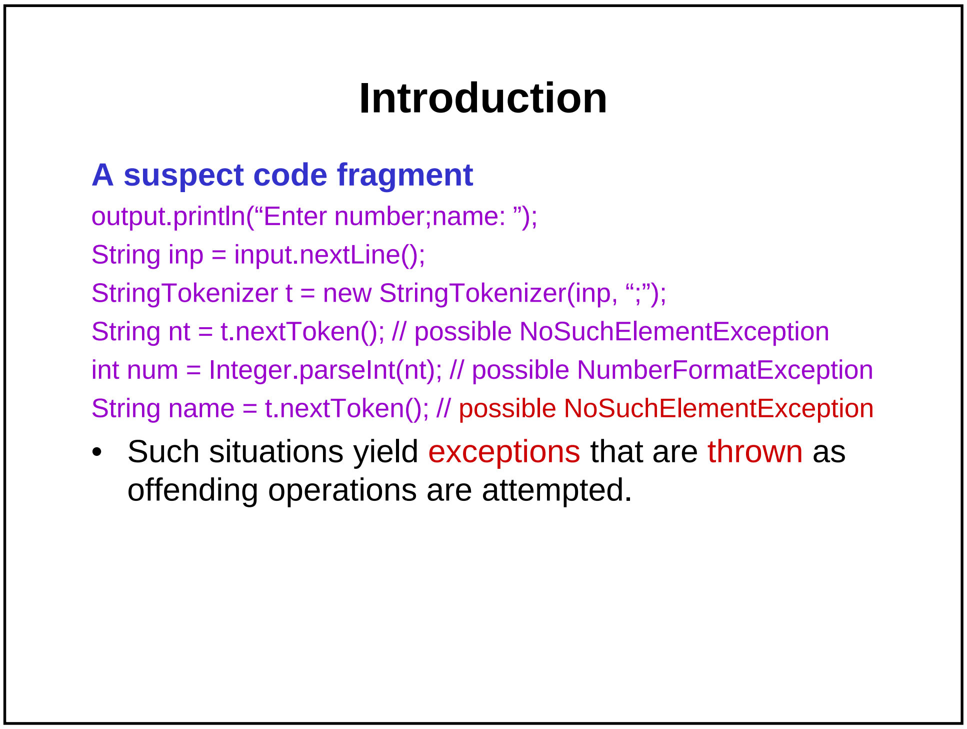


Introduction

A suspect code fragment

```
output.println(“Enter number;name: ”);
String inp = input.nextLine();
StringTokenizer t = new StringTokenizer(inp, “;”);
String nt = t.nextToken(); // possible NoSuchElementException
int num = Integer.parseInt(nt); // possible NumberFormatException
String name = t.nextToken(); // what if no such element
```

- Such situations yield exceptions that are thrown as offending operations are attempted.



Introduction

A suspect code fragment

```
output.println(“Enter number;name: ”);
String inp = input.nextLine();
StringTokenizer t = new StringTokenizer(inp, “;”);
String nt = t.nextToken(); // possible NoSuchElementException
int num = Integer.parseInt(nt); // possible NumberFormatException
String name = t.nextToken(); // possible NoSuchElementException
```

- Such situations yield exceptions that are thrown as offending operations are attempted.

Introduction

A suspect code fragment

```
output.println(“Enter number;name: ”);
String inp = input.nextLine();
StringTokenizer t = new StringTokenizer(inp, “;”);
String nt = t.nextToken(); // possible NoSuchElementException
int num = Integer.parseInt(nt); // possible NumberFormatException
String name = t.nextToken(); // possible NoSuchElementException
```

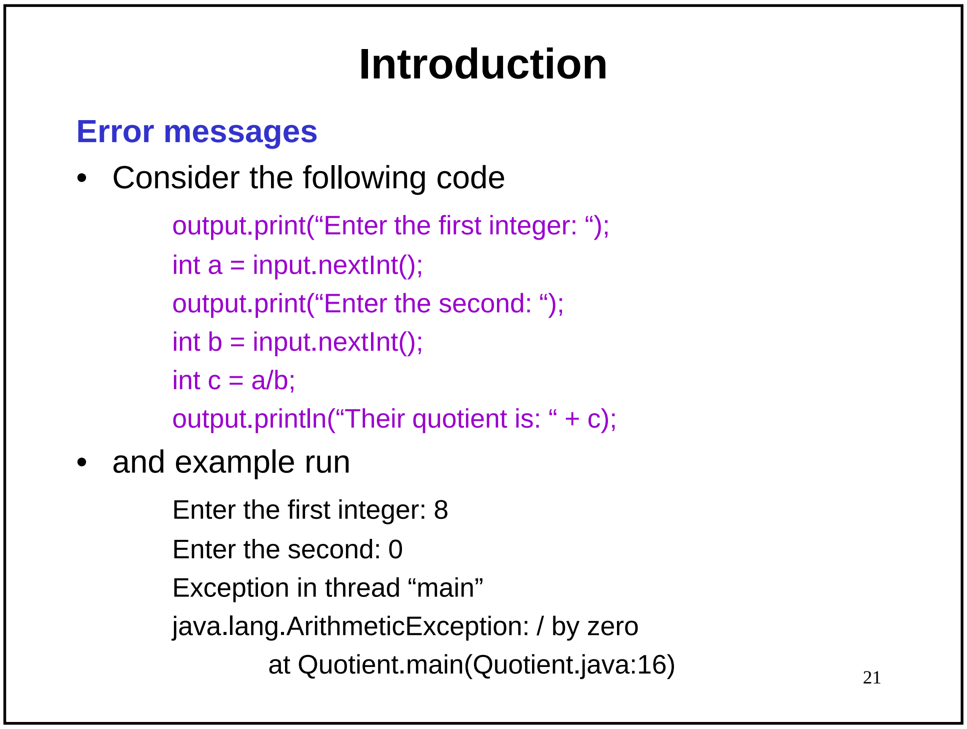
- Such situations yield exceptions that are thrown as offending operations are attempted.
- So, far we have seen these show up as error messages at runtime.

Introduction

Error messages

- Consider the following code

```
output.print(“Enter the first integer: “);
int a = input.nextInt();
output.print(“Enter the second: “);
int b = input.nextInt();
int c = a/b;
output.println(“Their quotient is: “ + c);
```



Introduction

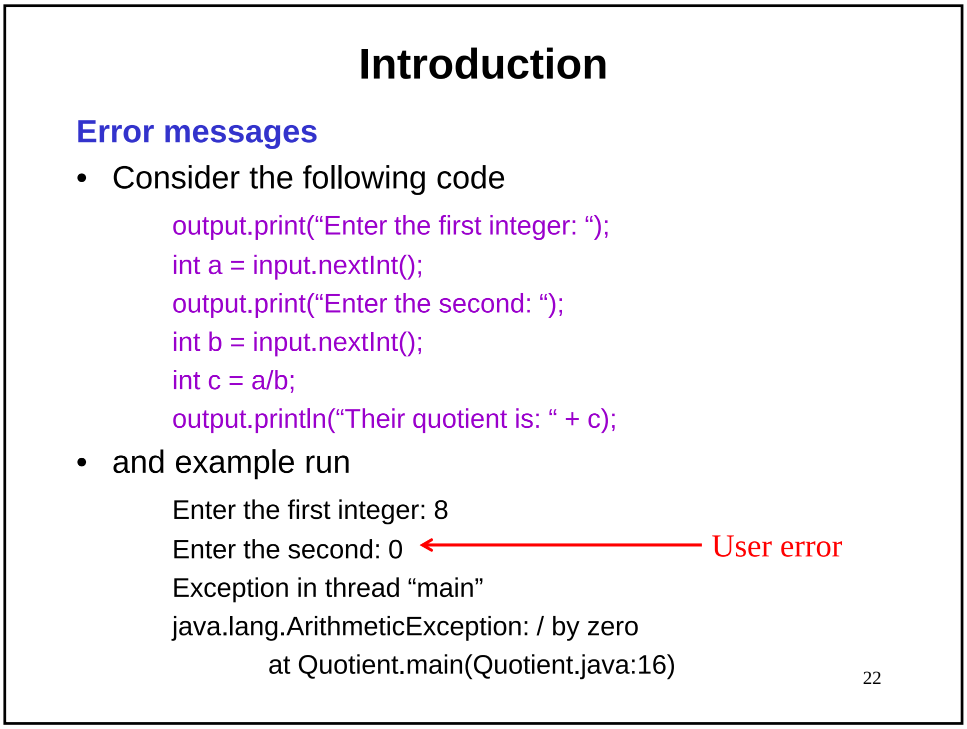
Error messages

- Consider the following code

```
output.print(“Enter the first integer: “);
int a = input.nextInt();
output.print(“Enter the second: “);
int b = input.nextInt();
int c = a/b;
output.println(“Their quotient is: “ + c);
```

- and example run

```
Enter the first integer: 8
Enter the second: 0
Exception in thread “main”
java.lang.ArithmeticException: / by zero
         at Quotient.main(Quotient.java:16)
```



Introduction

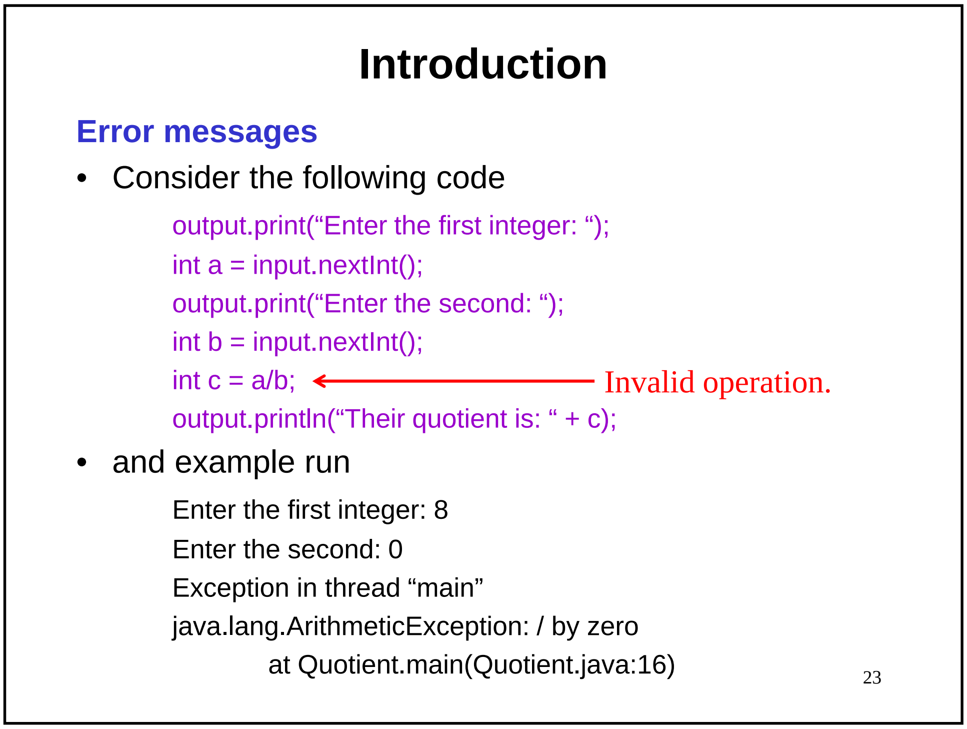
Error messages

- Consider the following code

```
output.print(“Enter the first integer: “);
int a = input.nextInt();
output.print(“Enter the second: “);
int b = input.nextInt();
int c = a/b;
output.println(“Their quotient is: “ + c);
```

- and example run

```
Enter the first integer: 8
Enter the second: 0  ←————————  User error
Exception in thread “main”
java.lang.ArithmeticException: / by zero
         at Quotient.main(Quotient.java:16)
```



Introduction

Error messages

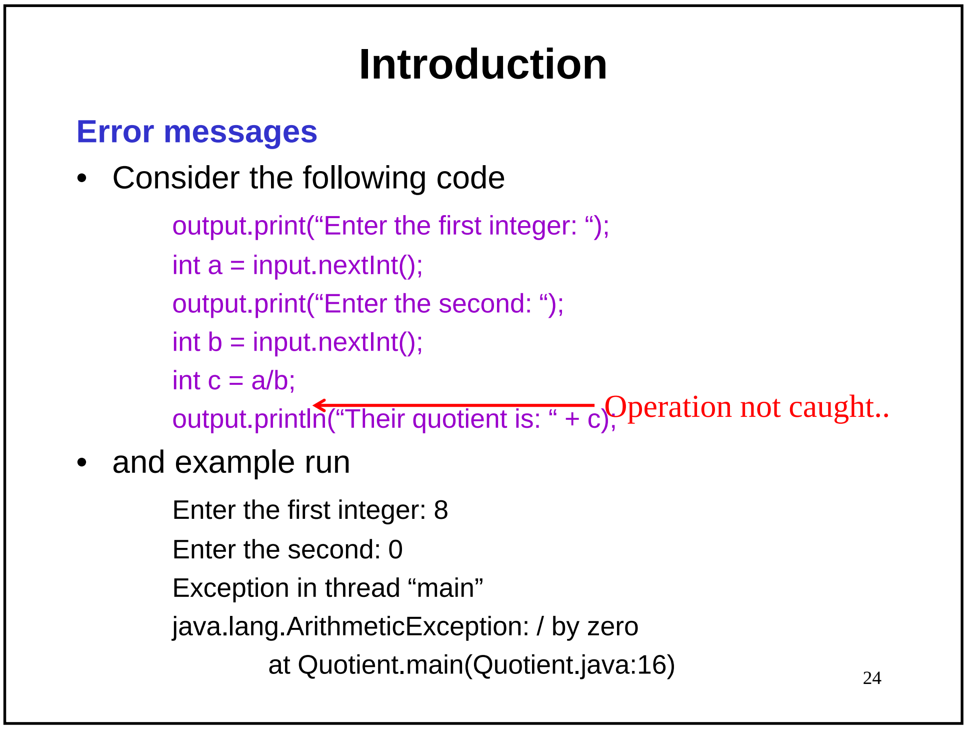
- Consider the following code

```
output.print("Enter the first integer: ");
int a = input.nextInt();
output.print("Enter the second: ");
int b = input.nextInt();
int c = a/b;
output.println("Their quotient is: " + c);
```

Invalid operation.

- and example run

```
Enter the first integer: 8
Enter the second: 0
Exception in thread "main"
java.lang.ArithmeticException: / by zero
        at Quotient.main(Quotient.java:16)
```



Introduction

Error messages

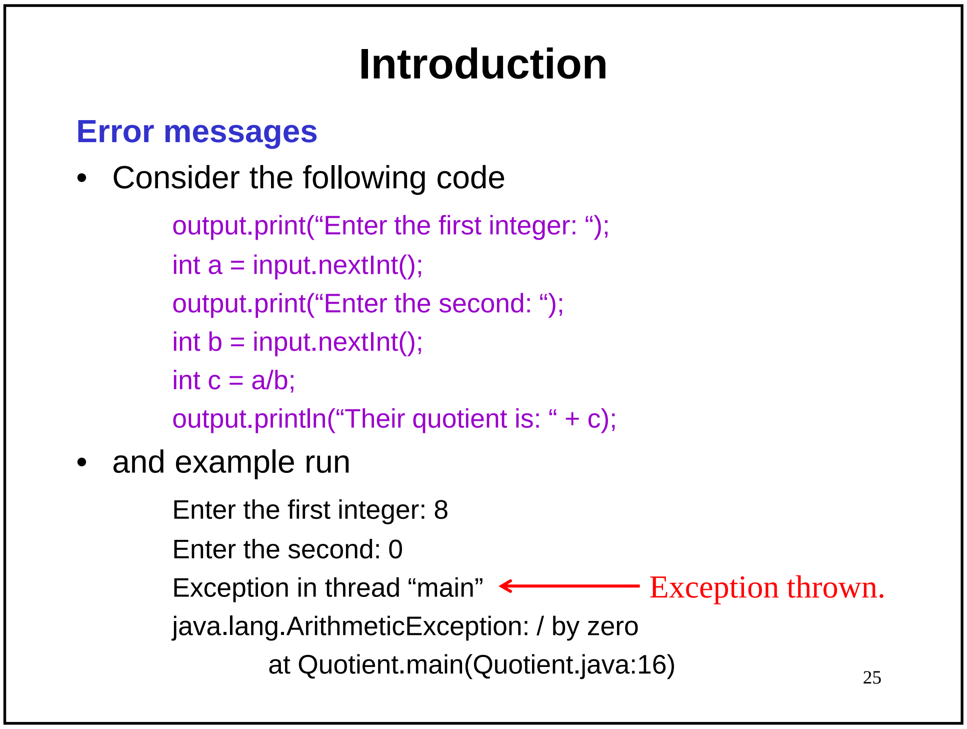
- Consider the following code

```
output.print("Enter the first integer: ");
int a = input.nextInt();
output.print("Enter the second: ");
int b = input.nextInt();
int c = a/b;
output.println("Their quotient is: " + c);
```

Operation not caught..

- and example run

```
Enter the first integer: 8
Enter the second: 0
Exception in thread "main"
java.lang.ArithmeticException: / by zero
        at Quotient.main(Quotient.java:16)
```



Introduction

Error messages

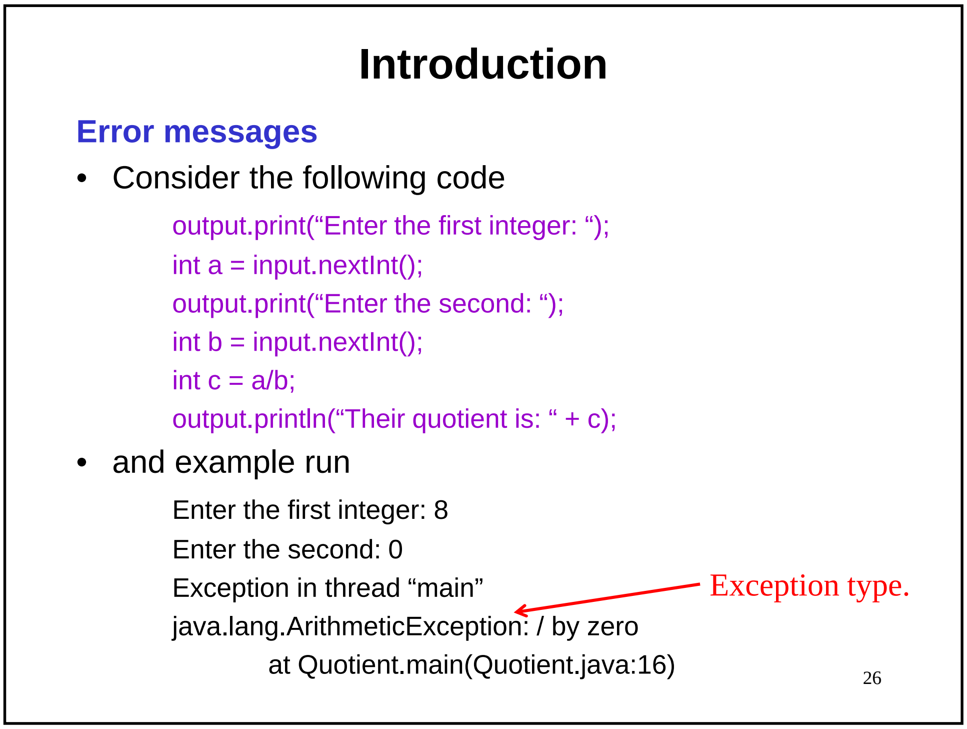
- Consider the following code

```
output.print("Enter the first integer: ");
int a = input.nextInt();
output.print("Enter the second: ");
int b = input.nextInt();
int c = a/b;
output.println("Their quotient is: " + c);
```

- and example run

```
Enter the first integer: 8
Enter the second: 0
Exception in thread "main"
java.lang.ArithmeticException: / by zero
        at Quotient.main(Quotient.java:16)
```

Exception thrown.



Introduction

Error messages

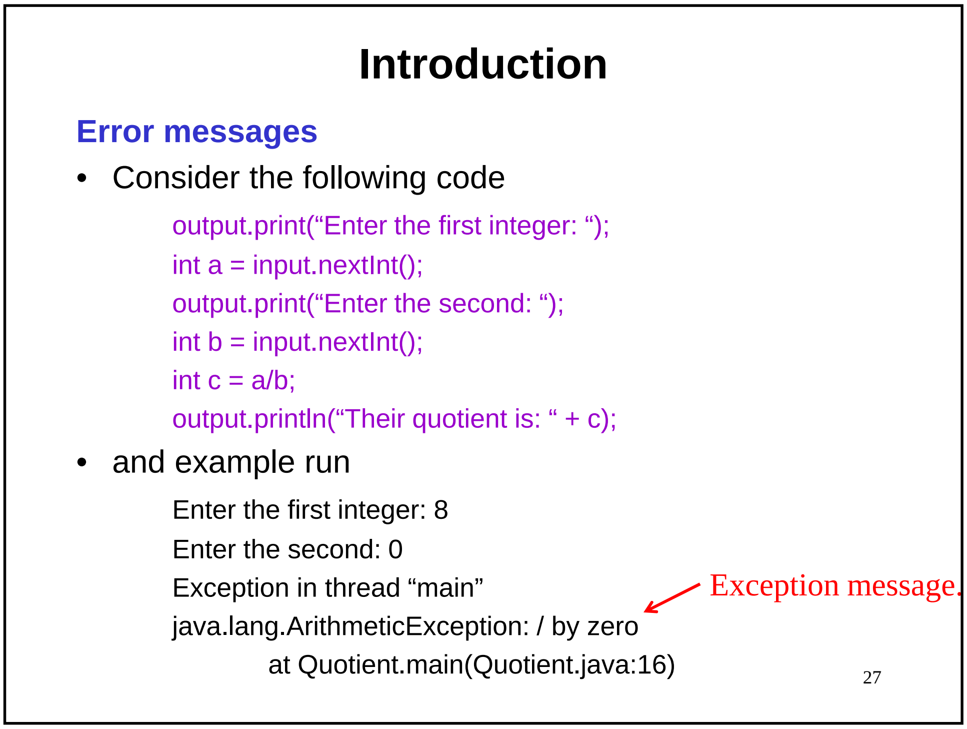
- Consider the following code

```
output.print("Enter the first integer: ");
int a = input.nextInt();
output.print("Enter the second: ");
int b = input.nextInt();
int c = a/b;
output.println("Their quotient is: " + c);
```

- and example run

```
Enter the first integer: 8
Enter the second: 0
Exception in thread "main"
java.lang.ArithmeticException: / by zero
        at Quotient.main(Quotient.java:16)
```

Exception type.



Introduction

Error messages

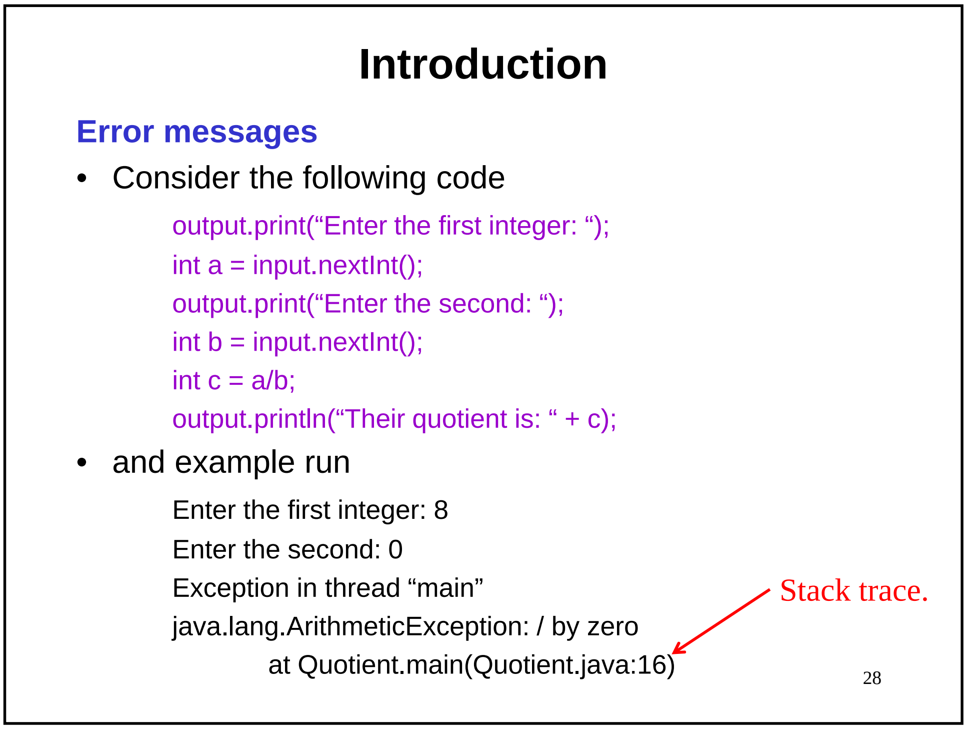
- Consider the following code

```
output.print("Enter the first integer: ");
int a = input.nextInt();
output.print("Enter the second: ");
int b = input.nextInt();
int c = a/b;
output.println("Their quotient is: " + c);
```

- and example run

```
Enter the first integer: 8
Enter the second: 0
Exception in thread "main"
java.lang.ArithmeticException: / by zero
        at Quotient.main(Quotient.java:16)
```

Exception message.



Introduction

Error messages

- Consider the following code

```
output.print("Enter the first integer: ");
int a = input.nextInt();
output.print("Enter the second: ");
int b = input.nextInt();
int c = a/b;
output.println("Their quotient is: " + c);
```

- and example run

```
Enter the first integer: 8
Enter the second: 0
Exception in thread "main"
java.lang.ArithmeticException: / by zero
        at Quotient.main(Quotient.java:16)
```

Stack trace.

Introduction

A suspect code fragment

```
output.println(“Enter number;name: ”);
String inp = input.nextLine();
StringTokenizer t = new StringTokenizer(inp, “;”);
String nt = t.nextToken(); // possible NoSuchElementException
int num = Integer.parseInt(nt); // possible NumberFormatException
String name = t.nextToken(); // possible NoSuchElementException
```

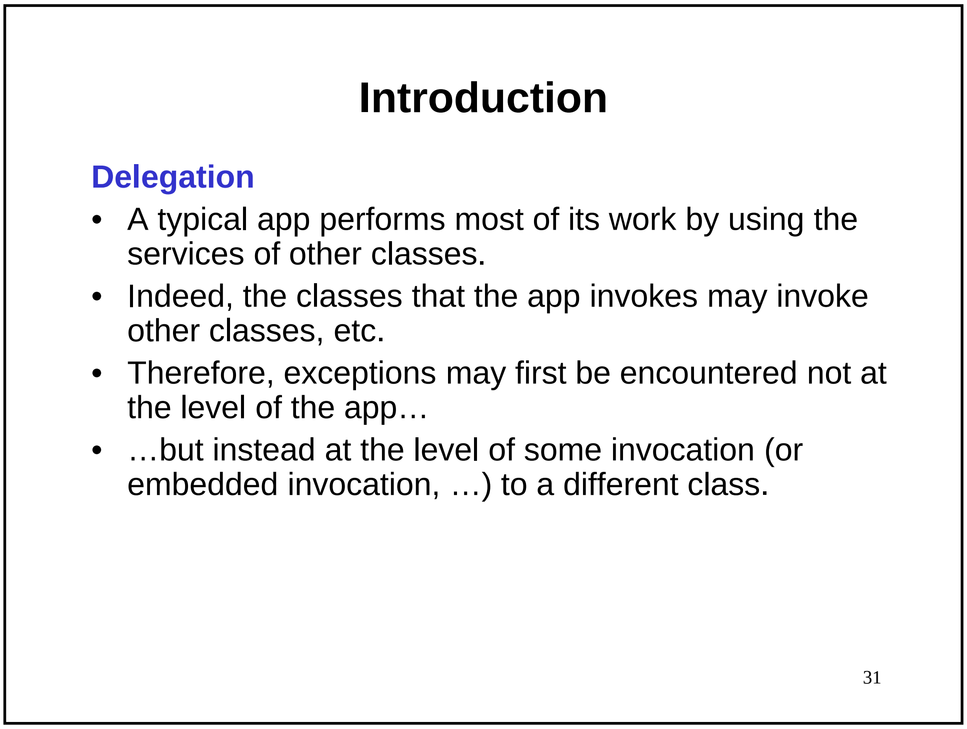
- Such situations yield exceptions that are thrown as offending operations are attempted.
- So, far we have seen these show up as error messages at runtime.

Introduction

A suspect code fragment

```
output.println(“Enter number;name: ”);
String inp = input.nextLine();
StringTokenizer t = new StringTokenizer(inp, “;”);
String nt = t.nextToken(); // possible NoSuchElementException
int num = Integer.parseInt(nt); // possible NumberFormatException
String name = t.nextToken(); // possible NoSuchElementException
```

- Such situations yield exceptions that are thrown as offending operations are attempted.
- So, far we have seen these show up as error messages at runtime.
- Now, we learn to deal such exceptions internal to our programs.

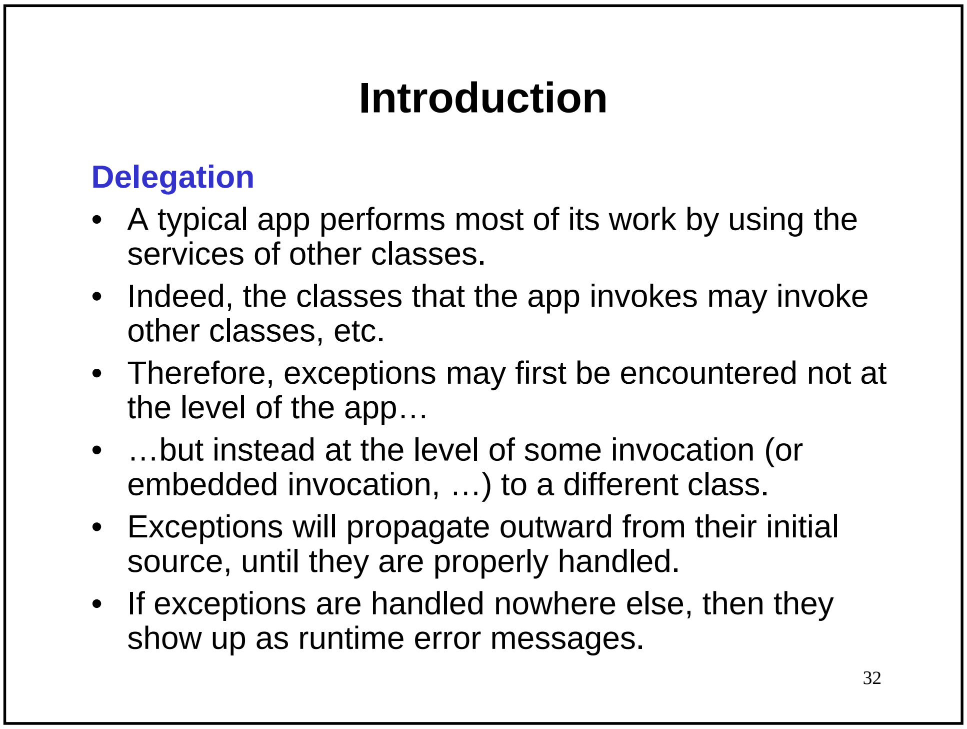


Introduction

Delegation

- A typical app performs most of its work by using the services of other classes.
- Indeed, the classes that the app invokes may invoke other classes, etc.
- Therefore, exceptions may first be encountered not at the level of the app…
- …but instead at the level of some invocation (or embedded invocation, …) to a different class.

31

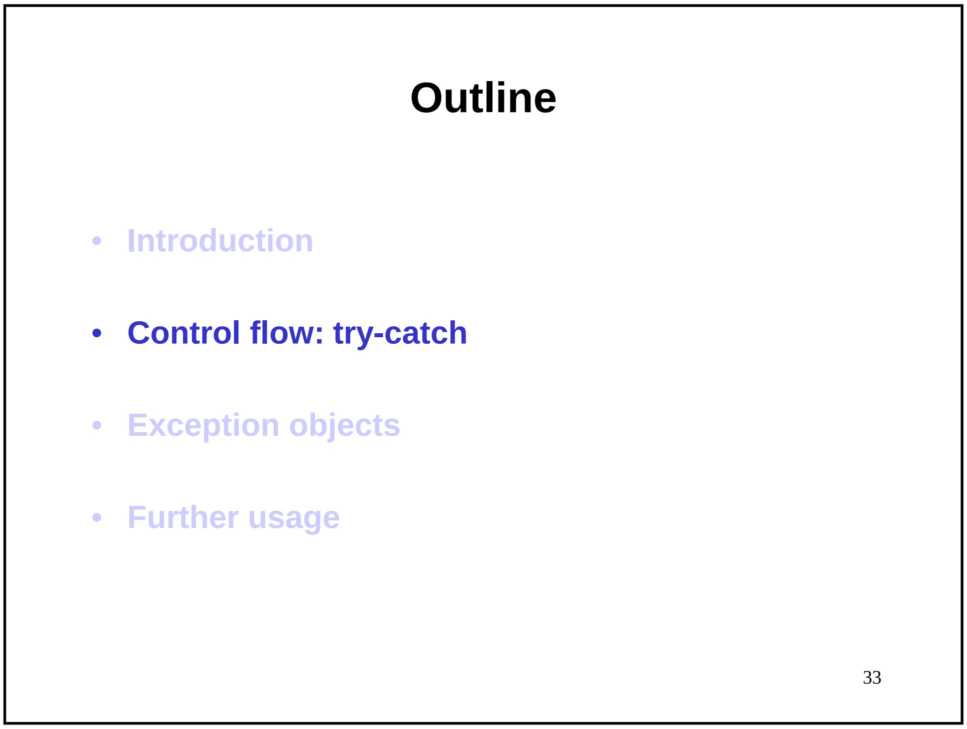


Introduction

Delegation

- A typical app performs most of its work by using the services of other classes.
- Indeed, the classes that the app invokes may invoke other classes, etc.
- Therefore, exceptions may first be encountered not at the level of the app…
- …but instead at the level of some invocation (or embedded invocation, …) to a different class.
- Exceptions will propagate outward from their initial source, until they are properly handled.
- If exceptions are handled nowhere else, then they show up as runtime error messages.

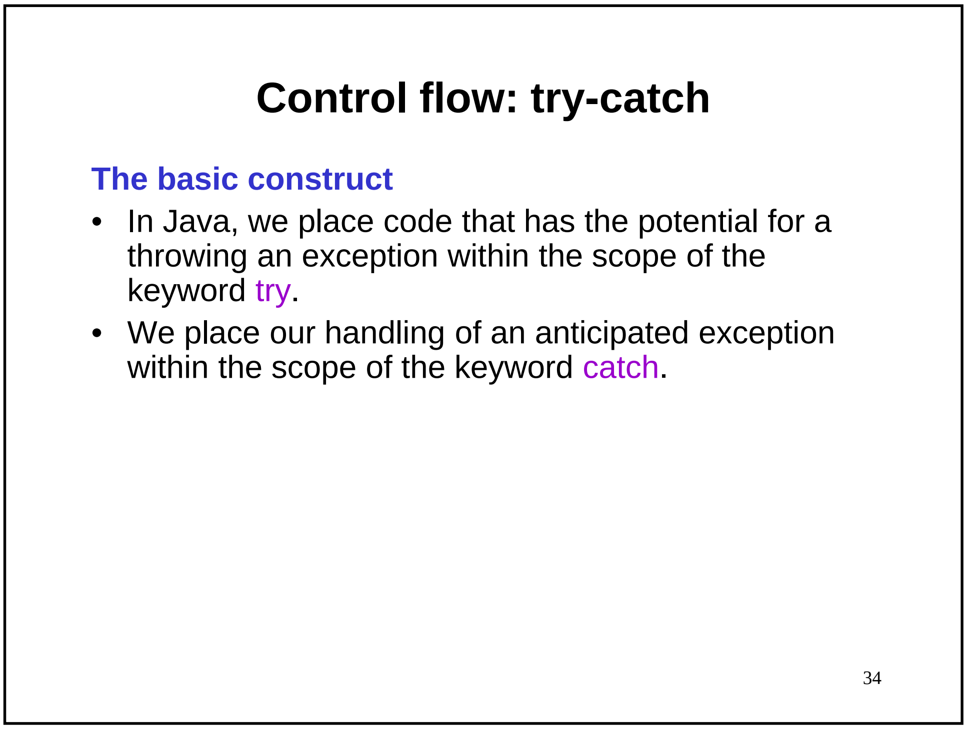
32



Outline

- Introduction
- Control flow: try-catch
- Exception objects
- Further usage

33



Control flow: try-catch

The basic construct

- In Java, we place code that has the potential for a throwing an exception within the scope of the keyword try.
- We place our handling of an anticipated exception within the scope of the keyword catch.

34

Control flow: try-catch

The basic construct

- In Java, we place code that has the potential for a throwing an exception within the scope of the keyword try.
- We place our handling of an anticipated exception within the scope of the keyword catch.

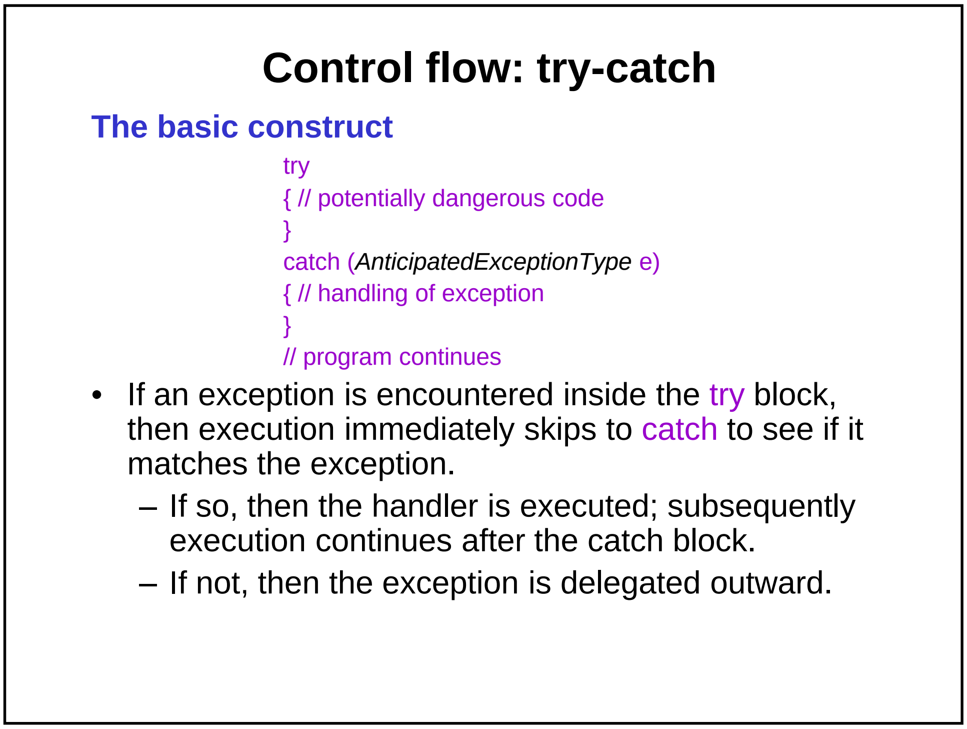
```
try
{ // potentially dangerous code
}
catch (AnticipatedExceptionType e)
{ // handling of exception
}
// program continues
```

Control flow: try-catch

The basic construct

```
try
{ // potentially dangerous code
}
catch (AnticipatedExceptionType e)
{ // handling of exception
}
// program continues
```

- If an exception is encountered inside the try block, then execution immediately skips to catch to see if it matches the exception.

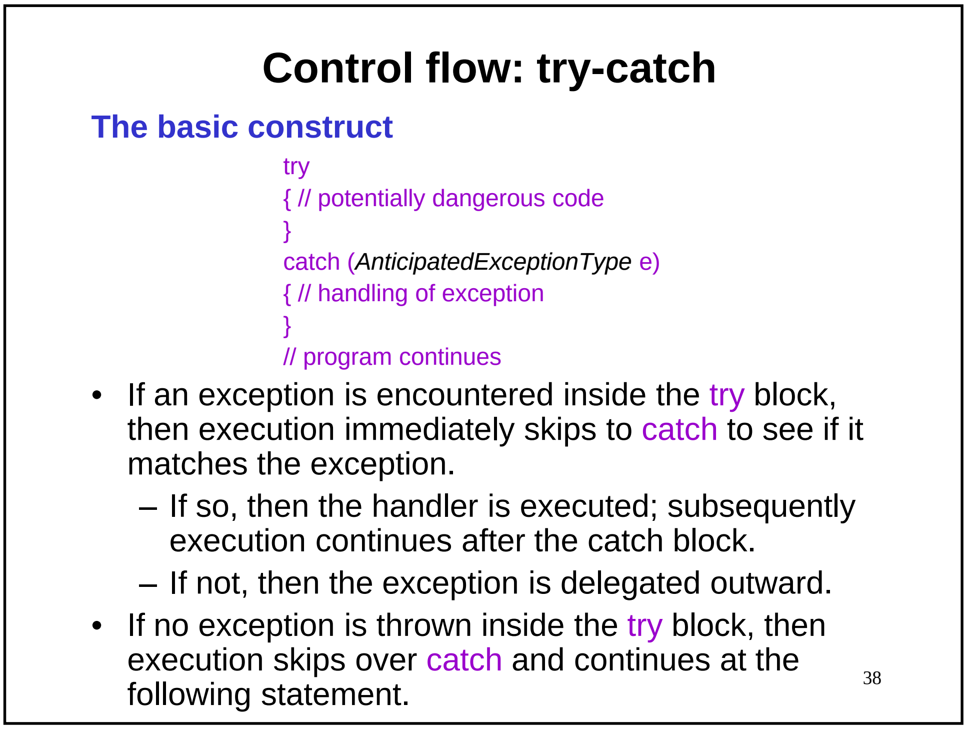


Control flow: try-catch

The basic construct

```
try
{ // potentially dangerous code
}
catch (AnticipatedExceptionType e)
{ // handling of exception
}
// program continues
```

- If an exception is encountered inside the try block, then execution immediately skips to catch to see if it matches the exception.
  - If so, then the handler is executed; subsequently execution continues after the catch block.
  - If not, then the exception is delegated outward.

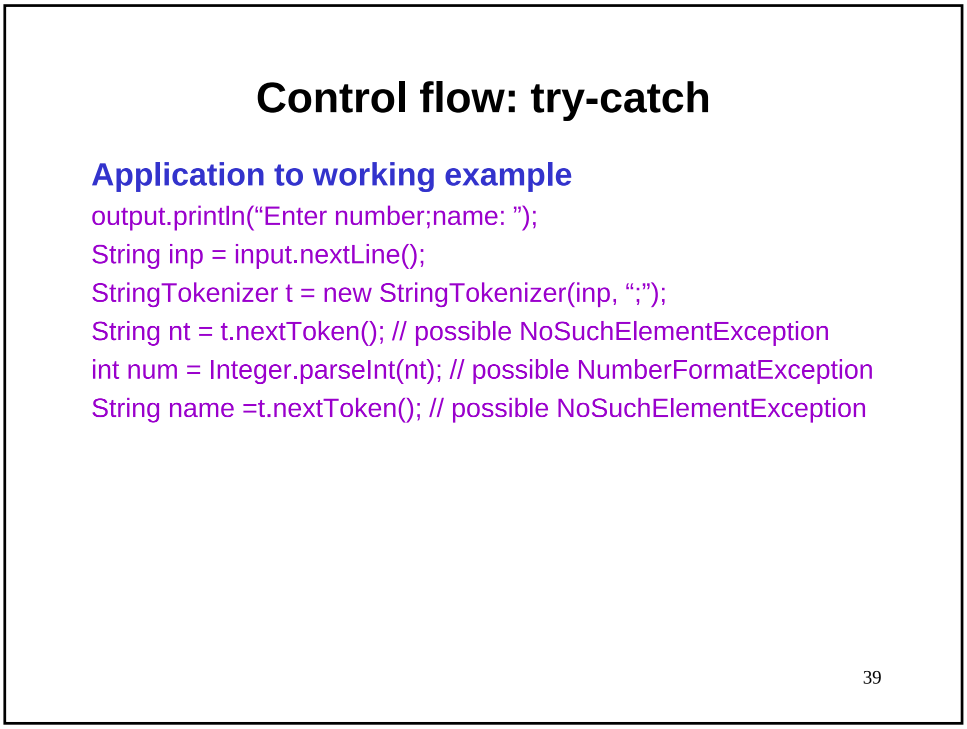


Control flow: try-catch

The basic construct

```
try
{ // potentially dangerous code
}
catch (AnticipatedExceptionType e)
{ // handling of exception
}
// program continues
```

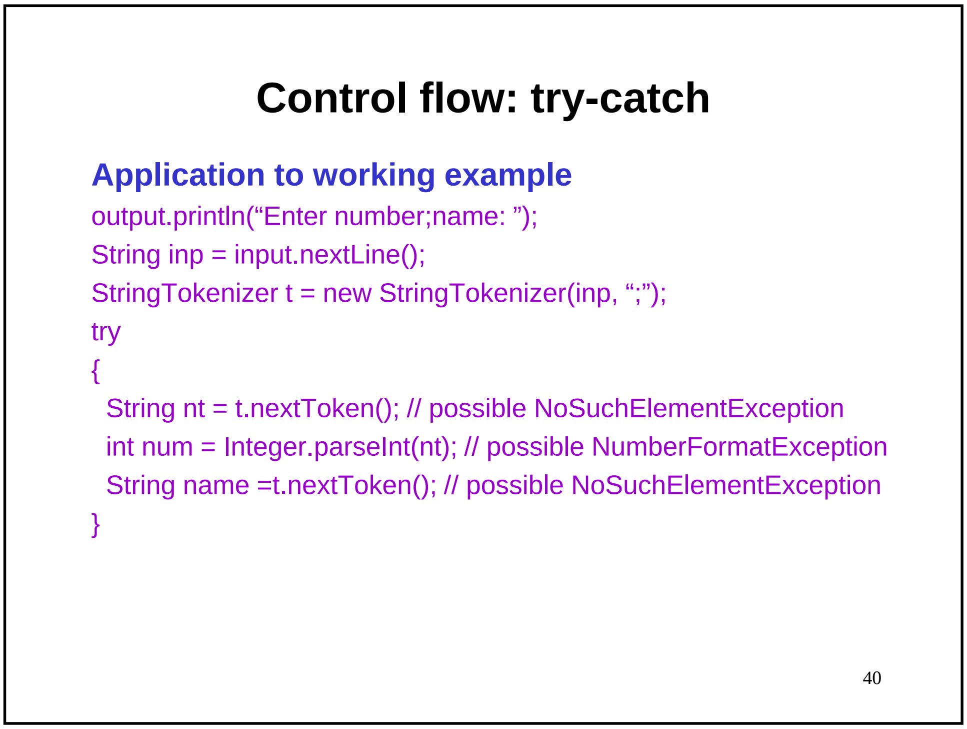
- If an exception is encountered inside the try block, then execution immediately skips to catch to see if it matches the exception.
  - If so, then the handler is executed; subsequently execution continues after the catch block.
  - If not, then the exception is delegated outward.
- If no exception is thrown inside the try block, then execution skips over catch and continues at the following statement.



Control flow: try-catch

Application to working example

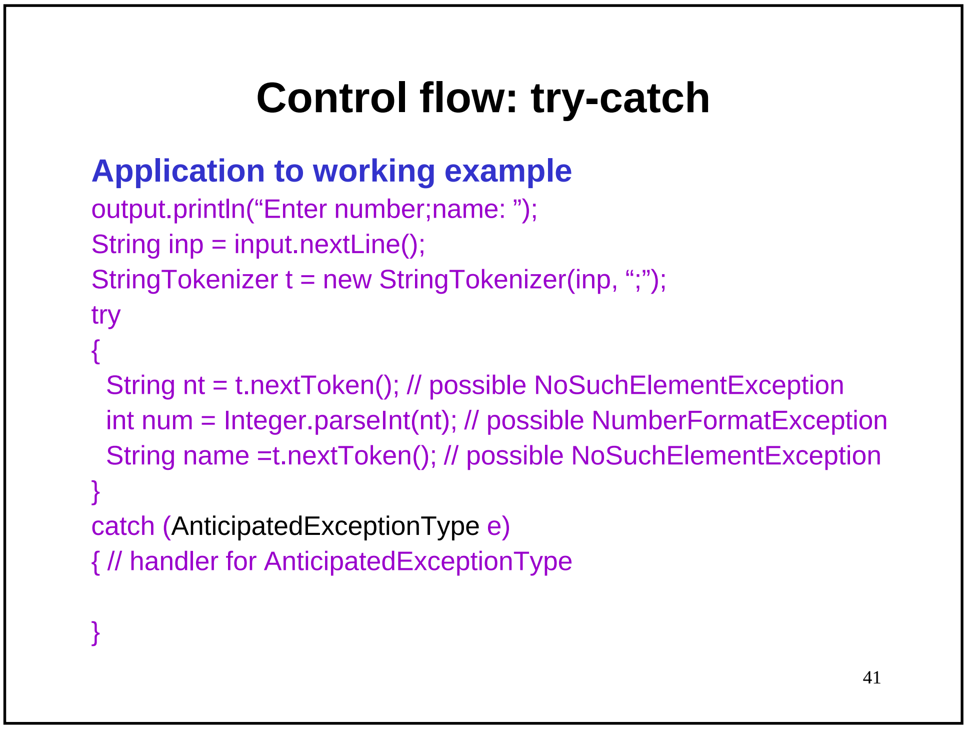
```
output.println(“Enter number;name: ”);
String inp = input.nextLine();
StringTokenizer t = new StringTokenizer(inp, “;”);
String nt = t.nextToken(); // possible NoSuchElementException
int num = Integer.parseInt(nt); // possible NumberFormatException
String name =t.nextToken(); // possible NoSuchElementException
```



Control flow: try-catch

Application to working example

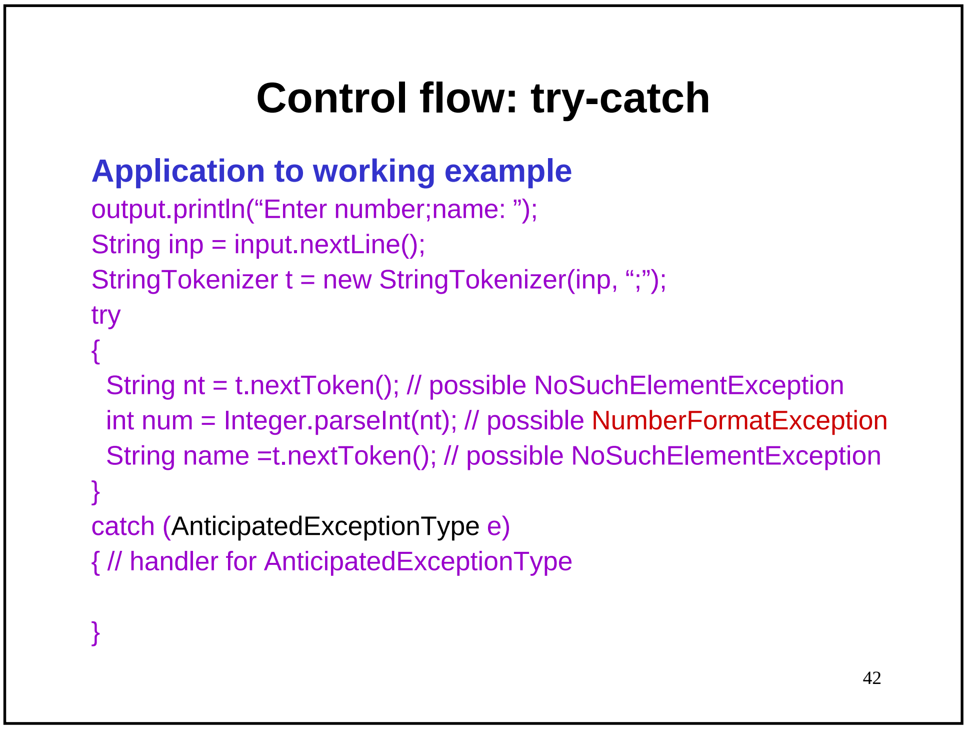
```
output.println(“Enter number;name: ”);
String inp = input.nextLine();
StringTokenizer t = new StringTokenizer(inp, “;”);
try
{
  String nt = t.nextToken(); // possible NoSuchElementException
  int num = Integer.parseInt(nt); // possible NumberFormatException
  String name =t.nextToken(); // possible NoSuchElementException
}
```



Control flow: try-catch

Application to working example

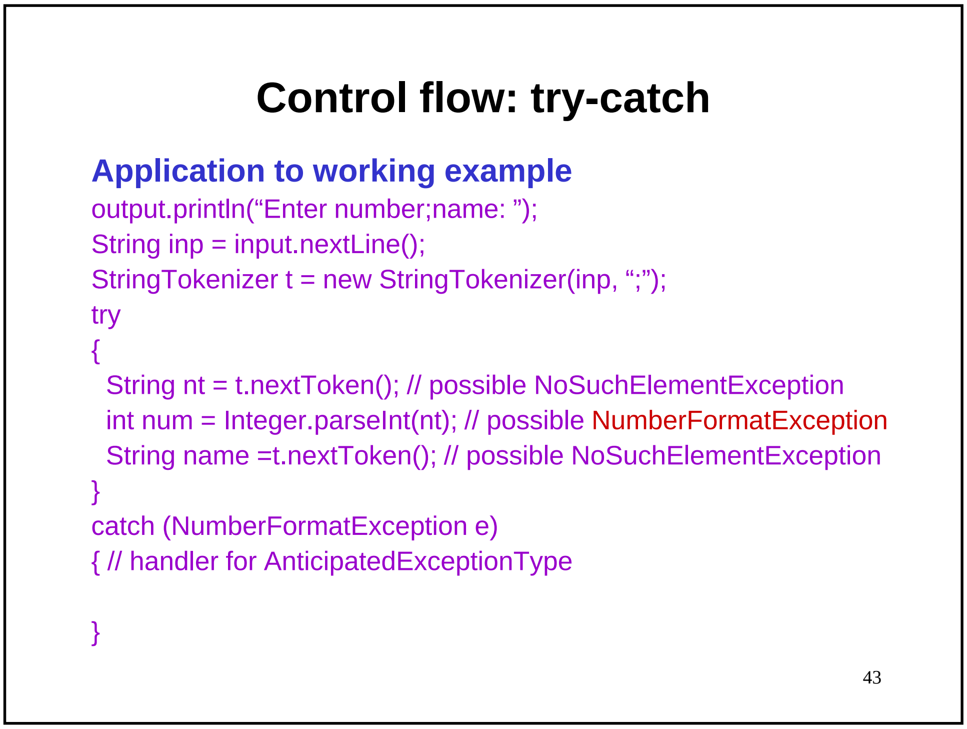
```
output.println("Enter number;name: ");
String inp = input.nextLine();
StringTokenizer t = new StringTokenizer(inp, ";");
try
{
  String nt = t.nextToken(); // possible NoSuchElementException
  int num = Integer.parseInt(nt); // possible NumberFormatException
  String name =t.nextToken(); // possible NoSuchElementException
}
catch (AnticipatedExceptionType e)
{ // handler for AnticipatedExceptionType

}
```



Control flow: try-catch

Application to working example

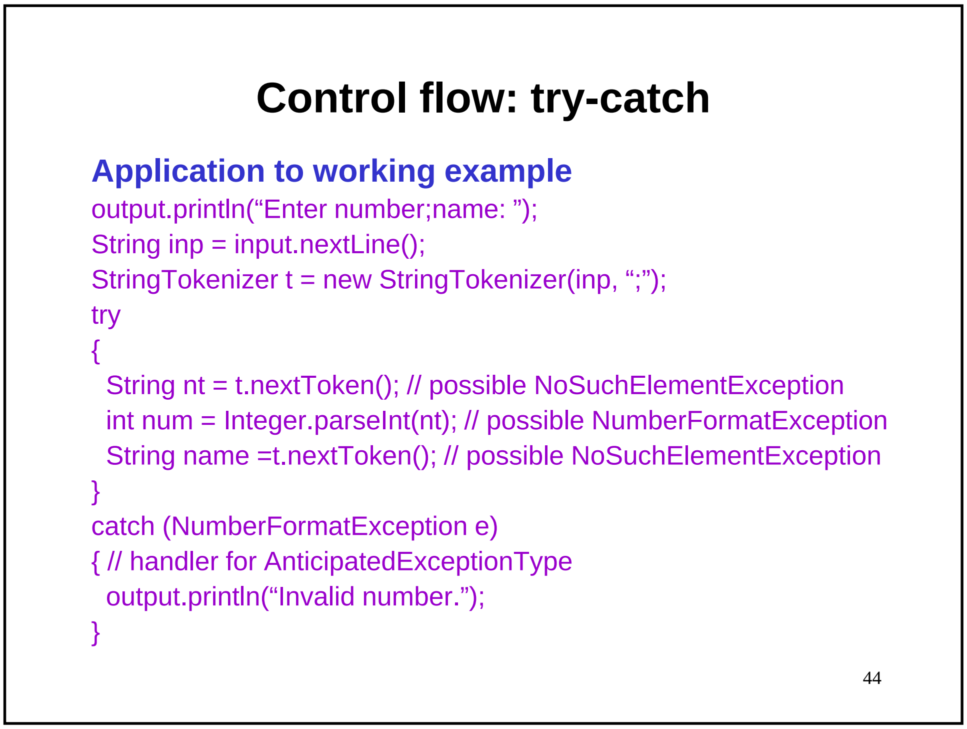
```
output.println("Enter number;name: ");
String inp = input.nextLine();
StringTokenizer t = new StringTokenizer(inp, ";");
try
{
  String nt = t.nextToken(); // possible NoSuchElementException
  int num = Integer.parseInt(nt); // possible NumberFormatException
  String name =t.nextToken(); // possible NoSuchElementException
}
catch (AnticipatedExceptionType e)
{ // handler for AnticipatedExceptionType

}
```



Control flow: try-catch

Application to working example

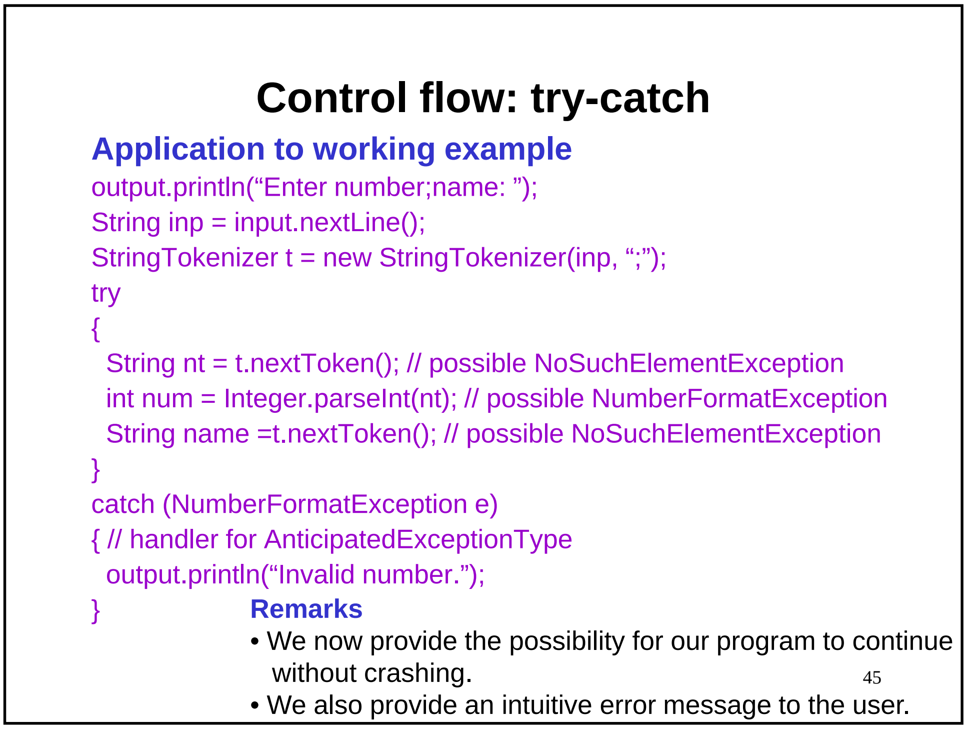
```
output.println(“Enter number;name: ”);
String inp = input.nextLine();
StringTokenizer t = new StringTokenizer(inp, “;”);
try
{
  String nt = t.nextToken(); // possible NoSuchElementException
  int num = Integer.parseInt(nt); // possible NumberFormatException
  String name =t.nextToken(); // possible NoSuchElementException
}
catch (NumberFormatException e)
{ // handler for AnticipatedExceptionType

}
```



Control flow: try-catch

Application to working example

```
output.println(“Enter number;name: ”);
String inp = input.nextLine();
StringTokenizer t = new StringTokenizer(inp, “;”);
try
{
  String nt = t.nextToken(); // possible NoSuchElementException
  int num = Integer.parseInt(nt); // possible NumberFormatException
  String name =t.nextToken(); // possible NoSuchElementException
}
catch (NumberFormatException e)
{ // handler for AnticipatedExceptionType
  output.println(“Invalid number.”);
}
```



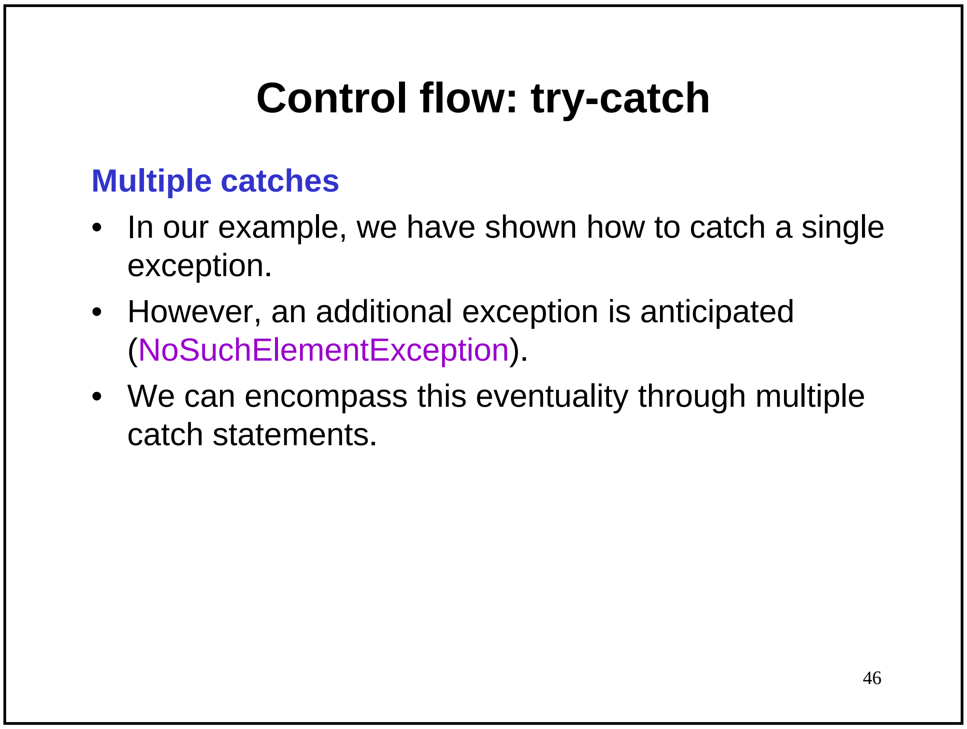
Control flow: try-catch

Application to working example

```
output.println(“Enter number;name: ”);
String inp = input.nextLine();
StringTokenizer t = new StringTokenizer(inp, “;”);
try
{
  String nt = t.nextToken(); // possible NoSuchElementException
  int num = Integer.parseInt(nt); // possible NumberFormatException
  String name =t.nextToken(); // possible NoSuchElementException
}
catch (NumberFormatException e)
{ // handler for AnticipatedExceptionType
  output.println(“Invalid number.”);
}
```

Remarks

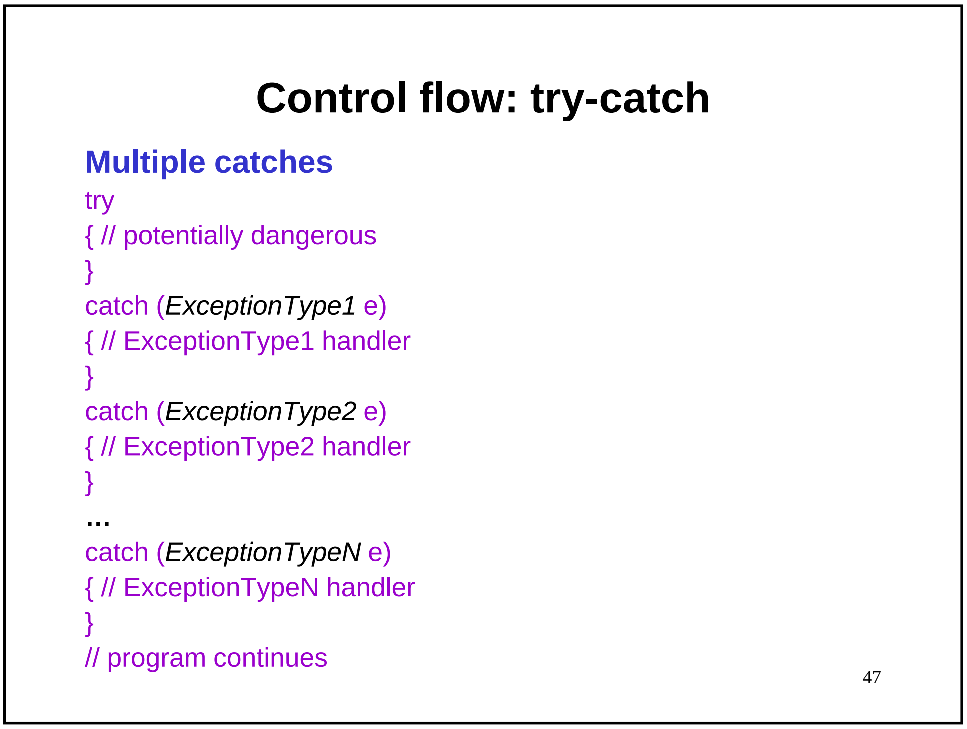
- We now provide the possibility for our program to continue without crashing.
- We also provide an intuitive error message to the user.



Control flow: try-catch

Multiple catches

- In our example, we have shown how to catch a single exception.
- However, an additional exception is anticipated (NoSuchElementException).
- We can encompass this eventuality through multiple catch statements.

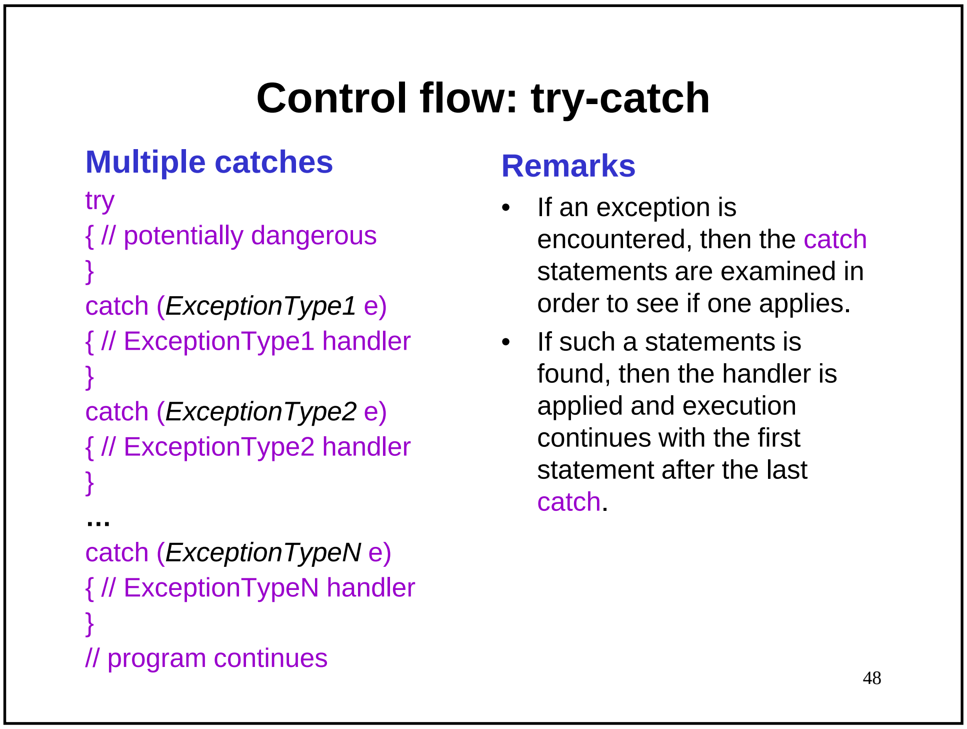


Control flow: try-catch

Multiple catches

```
try
{ // potentially dangerous
}
catch (ExceptionType1 e)
{ // ExceptionType1 handler
}
catch (ExceptionType2 e)
{ // ExceptionType2 handler
}
…
catch (ExceptionTypeN e)
{ // ExceptionTypeN handler
}
// program continues
```

47



Control flow: try-catch

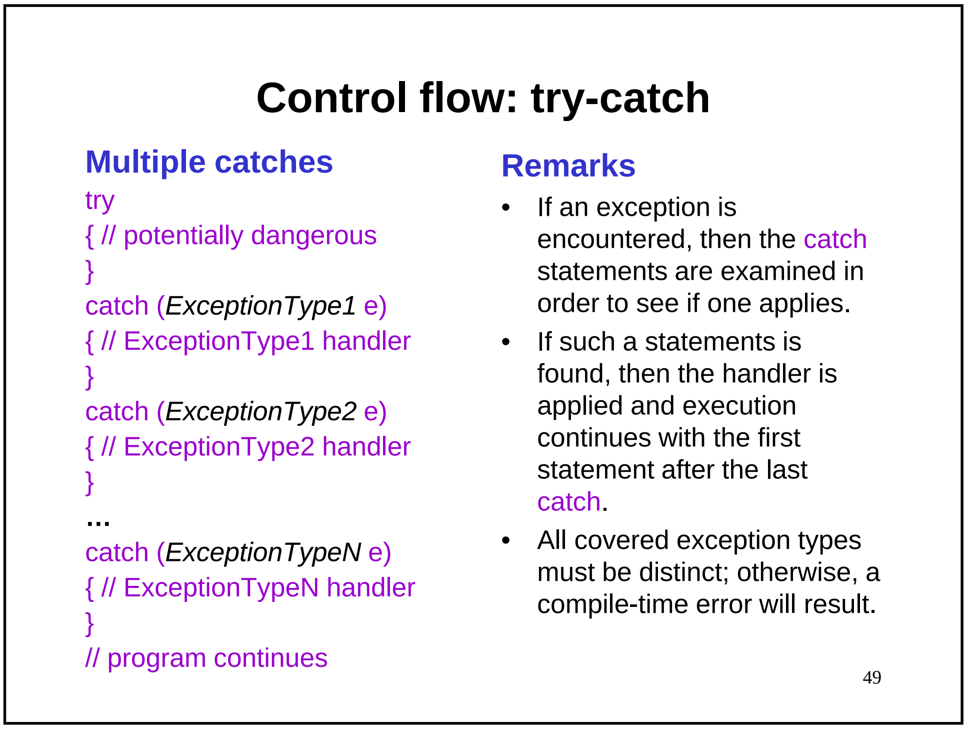
Multiple catches

```
try
{ // potentially dangerous
}
catch (ExceptionType1 e)
{ // ExceptionType1 handler
}
catch (ExceptionType2 e)
{ // ExceptionType2 handler
}
…
catch (ExceptionTypeN e)
{ // ExceptionTypeN handler
}
// program continues
```

Remarks

- If an exception is encountered, then the catch statements are examined in order to see if one applies.
- If such a statements is found, then the handler is applied and execution continues with the first statement after the last catch.

48



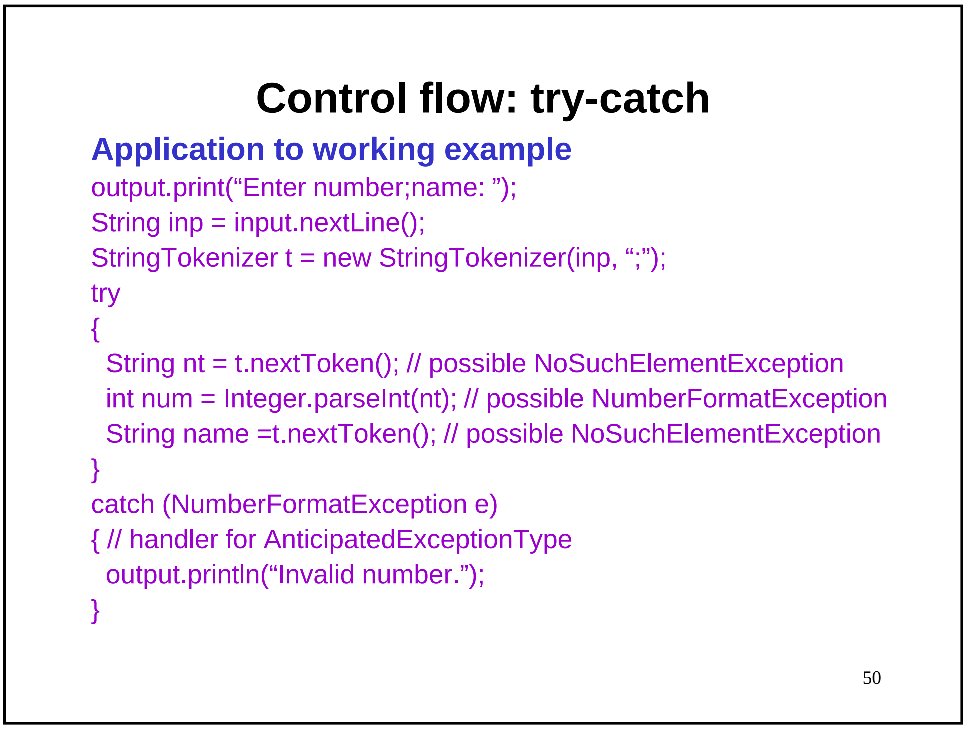
Control flow: try-catch

Multiple catches

```
try
{ // potentially dangerous
}
catch (ExceptionType1 e)
{ // ExceptionType1 handler
}
catch (ExceptionType2 e)
{ // ExceptionType2 handler
}
…
catch (ExceptionTypeN e)
{ // ExceptionTypeN handler
}
// program continues
```

Remarks

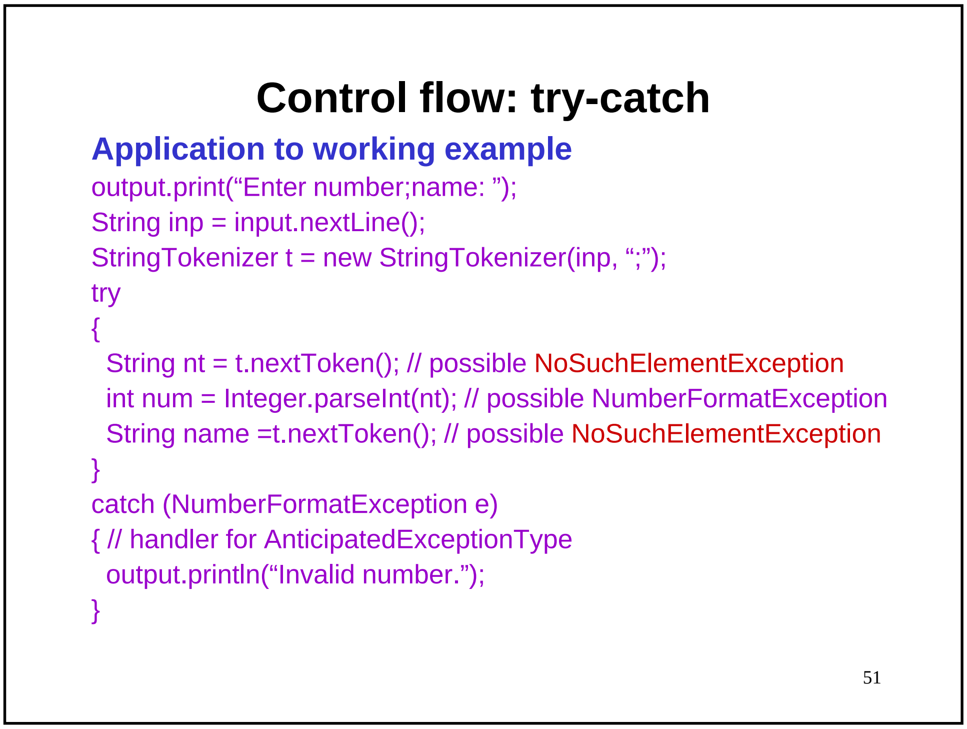
- If an exception is encountered, then the catch statements are examined in order to see if one applies.
- If such a statements is found, then the handler is applied and execution continues with the first statement after the last catch.
- All covered exception types must be distinct; otherwise, a compile-time error will result.



Control flow: try-catch

Application to working example

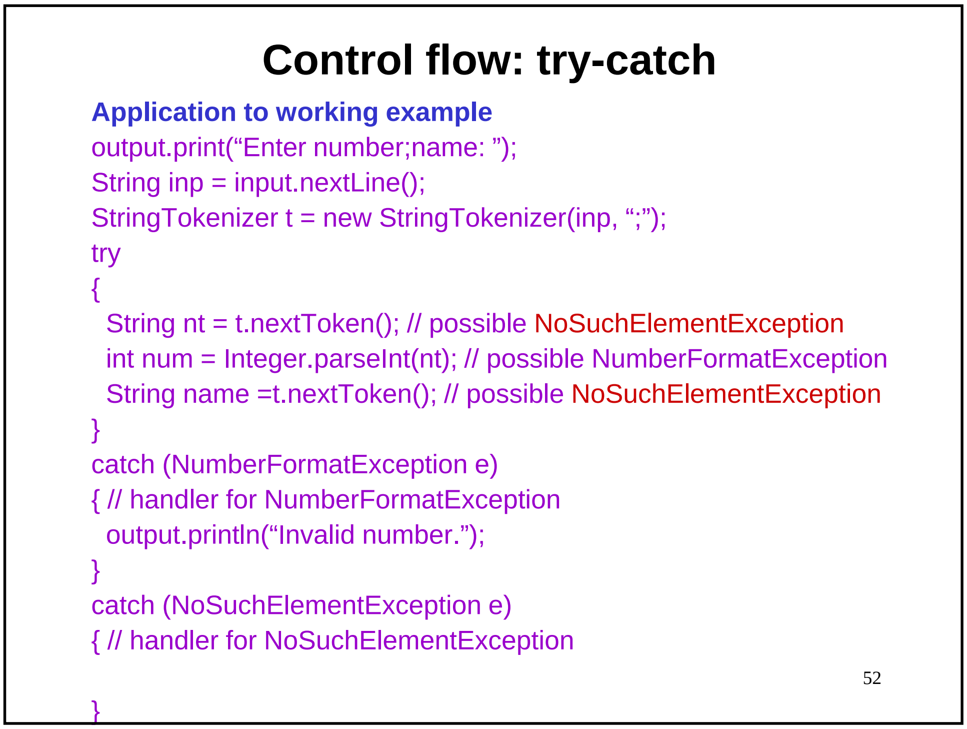
```
output.print(“Enter number;name: ”);
String inp = input.nextLine();
StringTokenizer t = new StringTokenizer(inp, “;”);
try
{
  String nt = t.nextToken(); // possible NoSuchElementException
  int num = Integer.parseInt(nt); // possible NumberFormatException
  String name =t.nextToken(); // possible NoSuchElementException
}
catch (NumberFormatException e)
{ // handler for AnticipatedExceptionType
  output.println(“Invalid number.”);
}
```



Control flow: try-catch

Application to working example

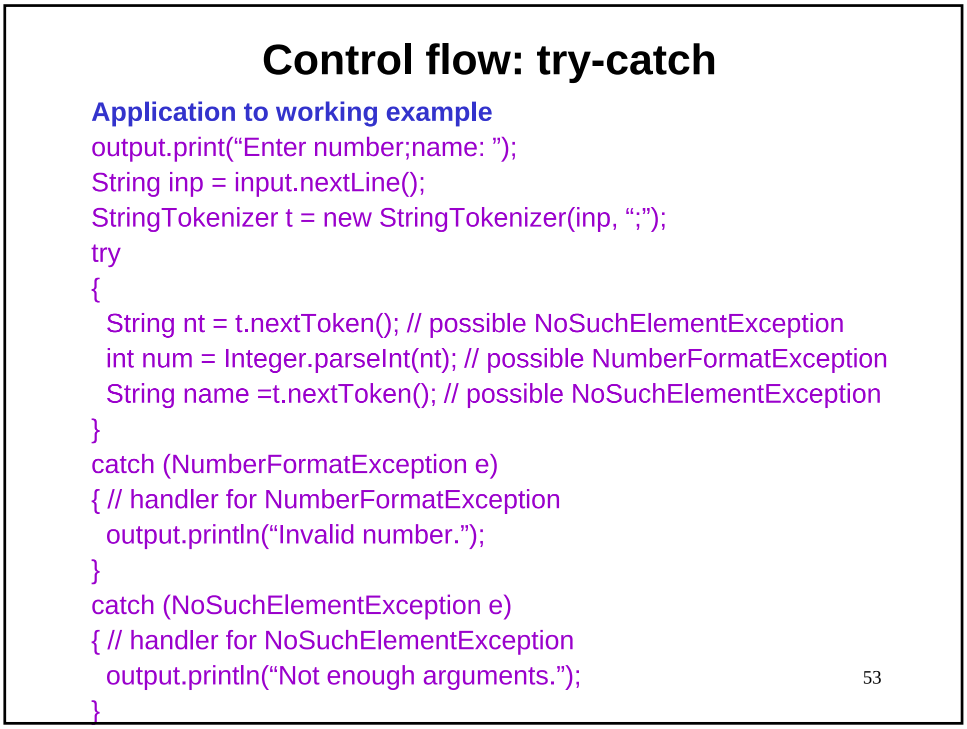
```
output.print("Enter number;name: ");
String inp = input.nextLine();
StringTokenizer t = new StringTokenizer(inp, ";");
try
{
  String nt = t.nextToken(); // possible NoSuchElementException
  int num = Integer.parseInt(nt); // possible NumberFormatException
  String name =t.nextToken(); // possible NoSuchElementException
}
catch (NumberFormatException e)
{ // handler for AnticipatedExceptionType
  output.println("Invalid number.");
}
```



Control flow: try-catch

Application to working example

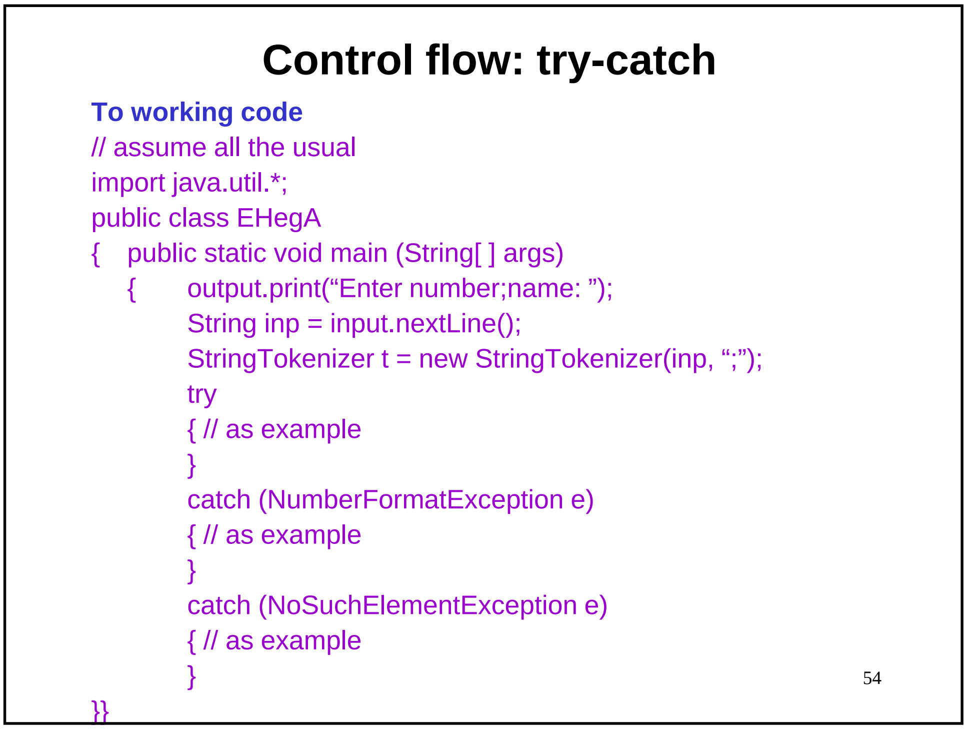
```
output.print("Enter number;name: ");
String inp = input.nextLine();
StringTokenizer t = new StringTokenizer(inp, ";");
try
{
  String nt = t.nextToken(); // possible NoSuchElementException
  int num = Integer.parseInt(nt); // possible NumberFormatException
  String name =t.nextToken(); // possible NoSuchElementException
}
catch (NumberFormatException e)
{ // handler for NumberFormatException
  output.println("Invalid number.");
}
catch (NoSuchElementException e)
{ // handler for NoSuchElementException

}
```



Control flow: try-catch

Application to working example

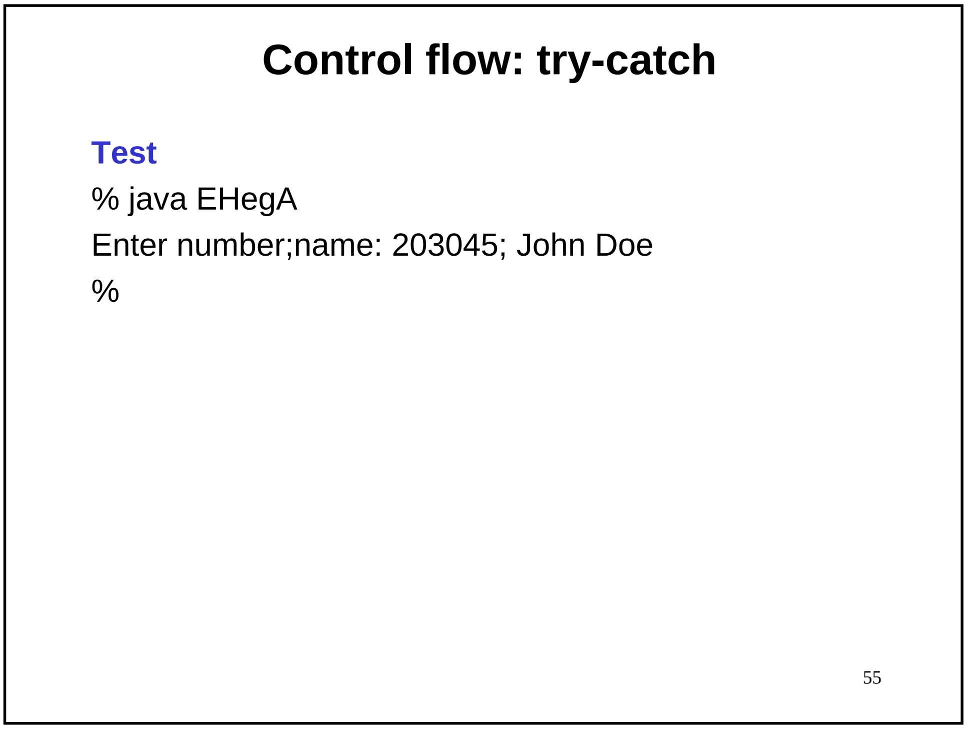
```
output.print("Enter number;name: ");
String inp = input.nextLine();
StringTokenizer t = new StringTokenizer(inp, ";");
try
{
  String nt = t.nextToken(); // possible NoSuchElementException
  int num = Integer.parseInt(nt); // possible NumberFormatException
  String name =t.nextToken(); // possible NoSuchElementException
}
catch (NumberFormatException e)
{ // handler for NumberFormatException
  output.println("Invalid number.");
}
catch (NoSuchElementException e)
{ // handler for NoSuchElementException
  output.println("Not enough arguments.");
}
```



Control flow: try-catch

To working code

```
// assume all the usual
import java.util.*;
public class EHegA
{   public static void main (String[ ] args)
    {       output.print("Enter number;name: ");
            String inp = input.nextLine();
            StringTokenizer t = new StringTokenizer(inp, ";");
            try
            { // as example
            }
            catch (NumberFormatException e)
            { // as example
            }
            catch (NoSuchElementException e)
            { // as example
            }
}}
```

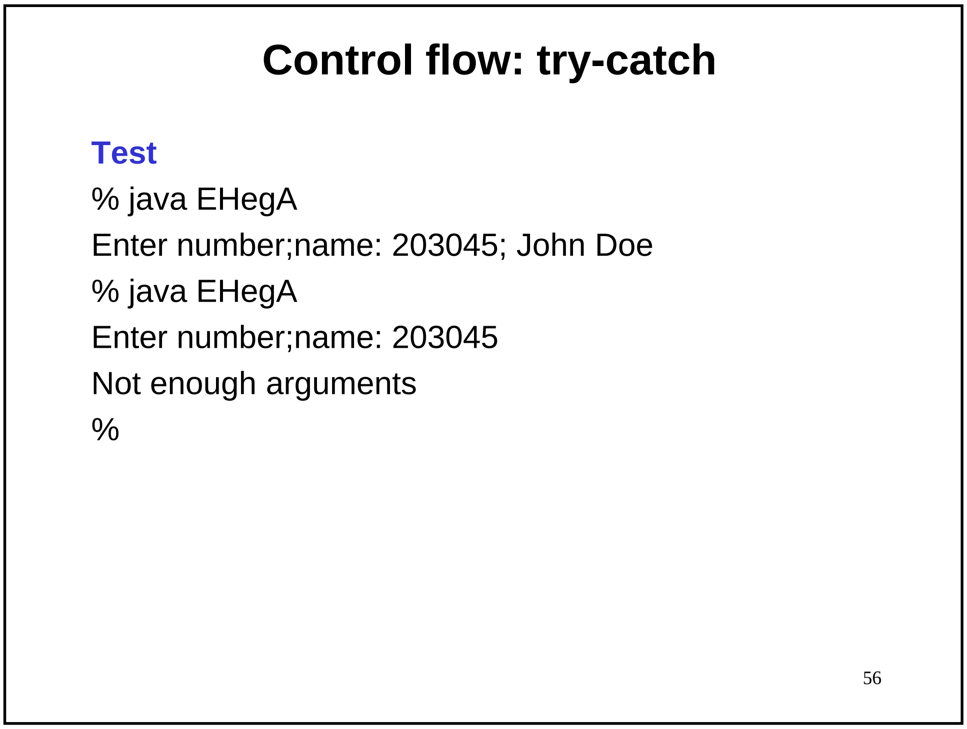


Control flow: try-catch

Test

```
% java EHegA
Enter number;name: 203045; John Doe
%
```

55

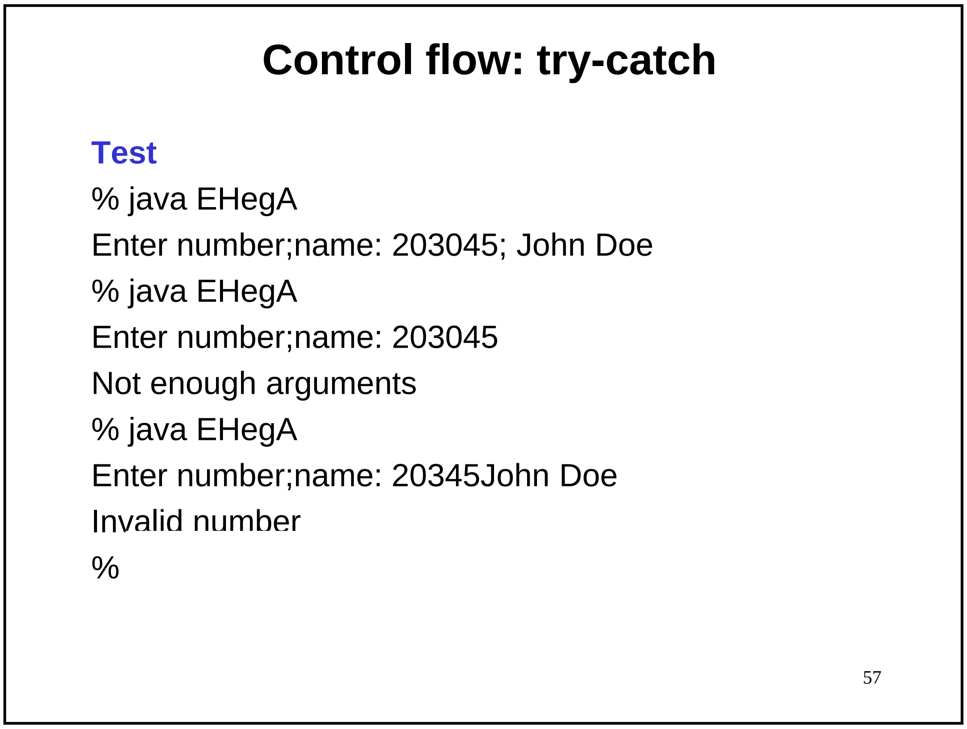


Control flow: try-catch

Test

```
% java EHegA
Enter number;name: 203045; John Doe
% java EHegA
Enter number;name: 203045
Not enough arguments
%
```

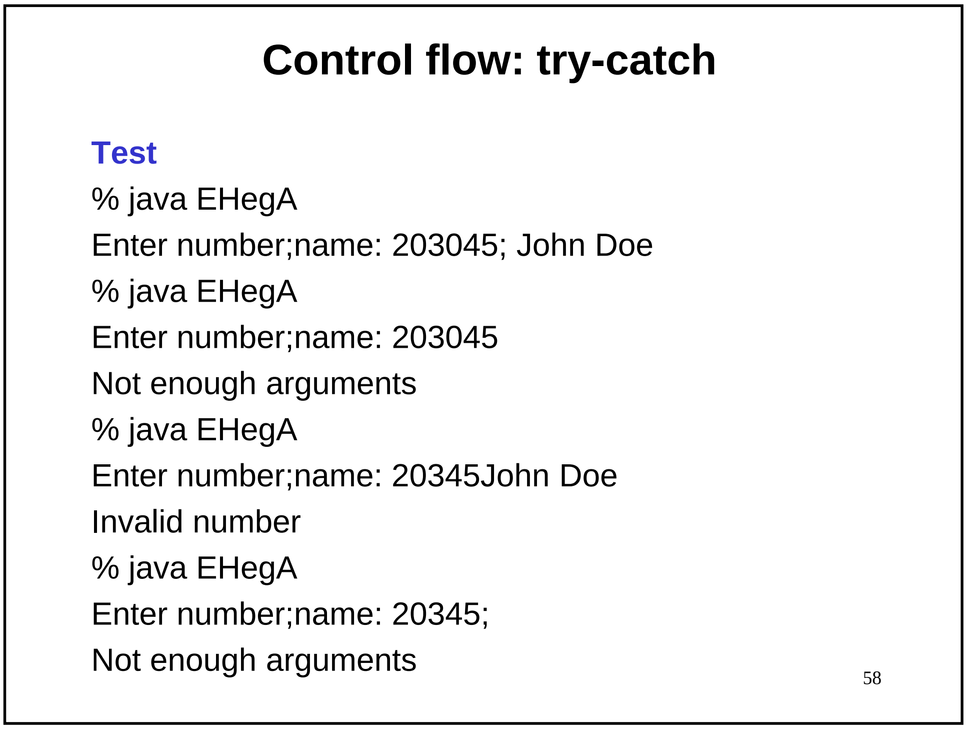
56



Control flow: try-catch

Test

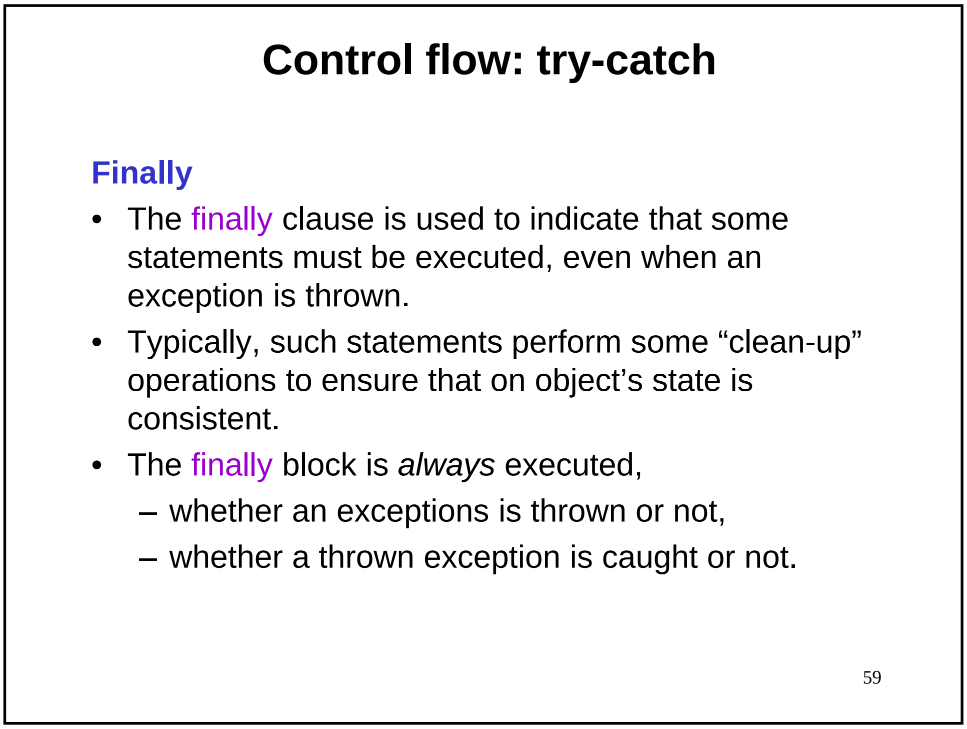
```
% java EHegA
Enter number;name: 203045; John Doe
% java EHegA
Enter number;name: 203045
Not enough arguments
% java EHegA
Enter number;name: 20345John Doe
Invalid number
%
```



Control flow: try-catch

Test

```
% java EHegA
Enter number;name: 203045; John Doe
% java EHegA
Enter number;name: 203045
Not enough arguments
% java EHegA
Enter number;name: 20345John Doe
Invalid number
% java EHegA
Enter number;name: 20345;
Not enough arguments
```

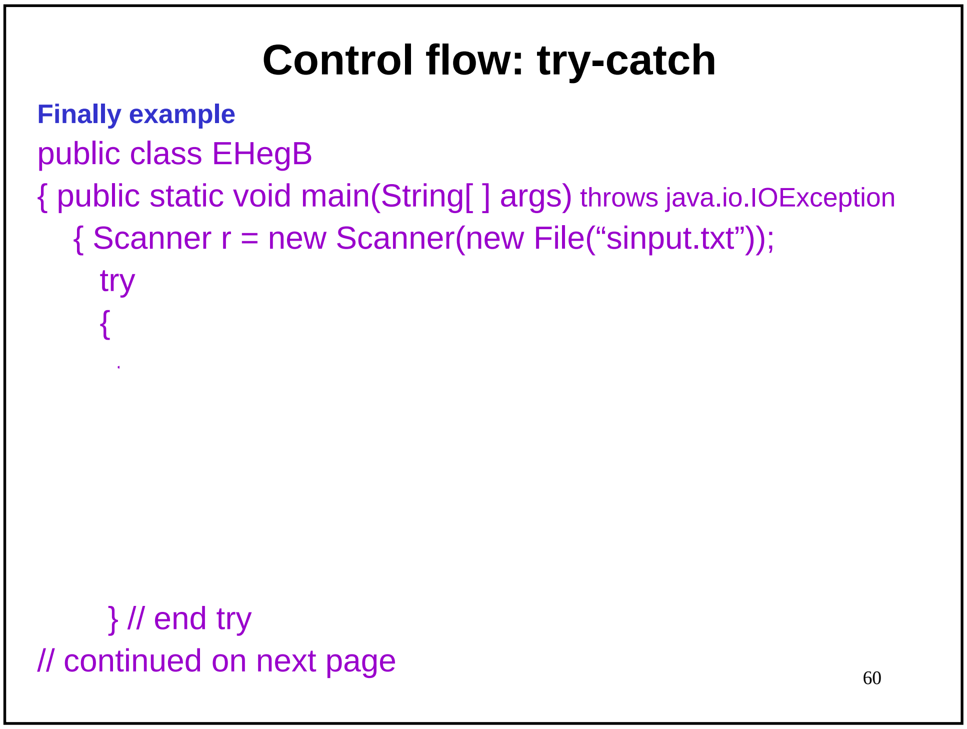


Control flow: try-catch

Finally

- The finally clause is used to indicate that some statements must be executed, even when an exception is thrown.
- Typically, such statements perform some “clean-up” operations to ensure that on object’s state is consistent.
- The finally block is *always* executed,
  - whether an exceptions is thrown or not,
  - whether a thrown exception is caught or not.

59

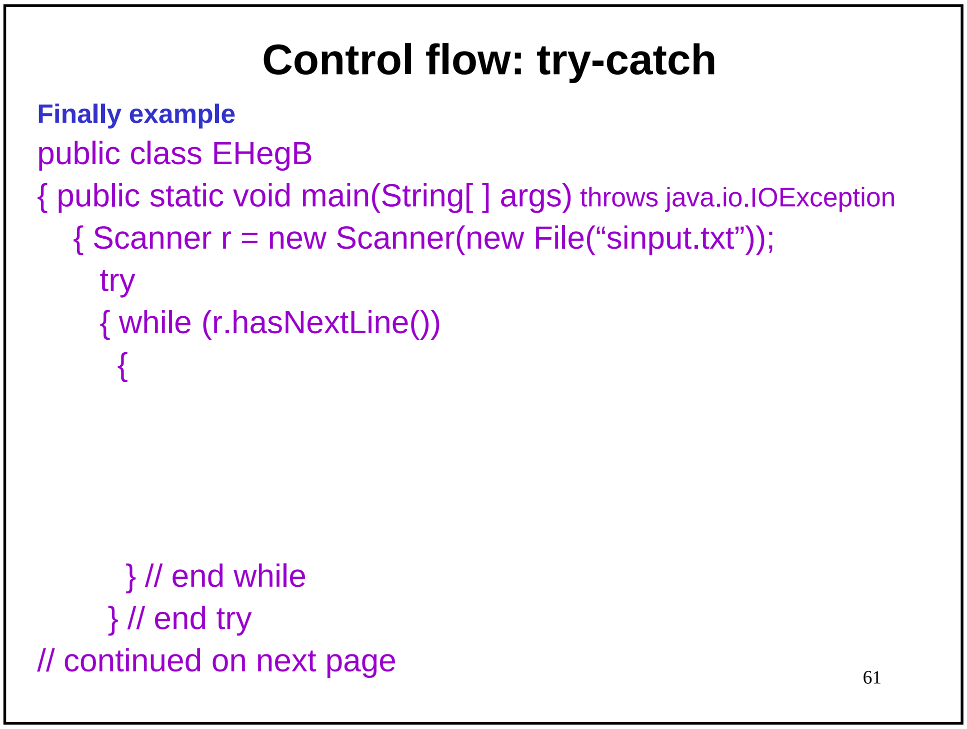


Control flow: try-catch

Finally example

```
public class EHegB
{ public static void main(String[ ] args) throws java.io.IOException
   { Scanner r = new Scanner(new File(“sinput.txt”));
     try
     {
       .





      } // end try
// continued on next page
```

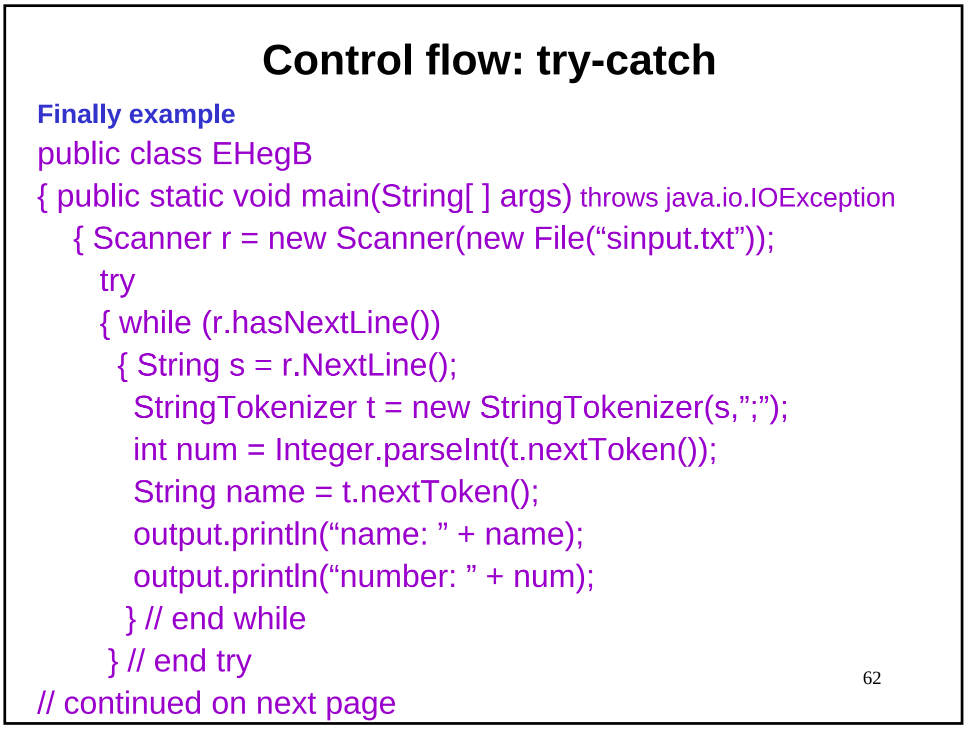
60



Control flow: try-catch

Finally example

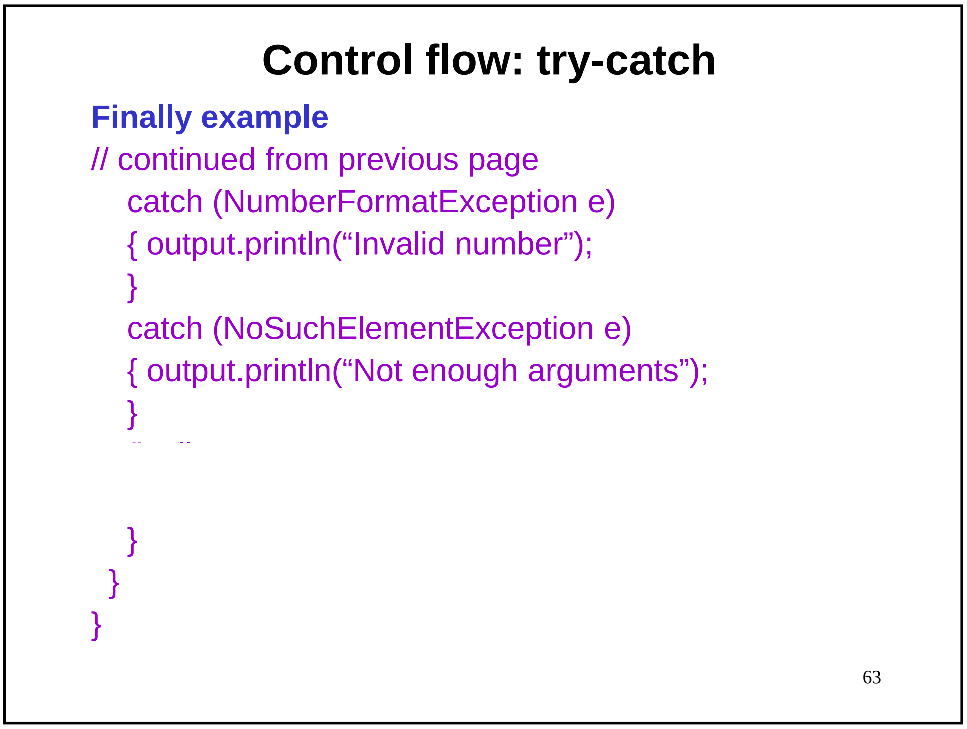
```
public class EHegB
{ public static void main(String[ ] args) throws java.io.IOException
   { Scanner r = new Scanner(new File(“sinput.txt”));
      try
      { while (r.hasNextLine())
        {




         } // end while
       } // end try
// continued on next page
```



Control flow: try-catch

Finally example

```
public class EHegB
{ public static void main(String[ ] args) throws java.io.IOException
   { Scanner r = new Scanner(new File(“sinput.txt”));
      try
      { while (r.hasNextLine())
        { String s = r.NextLine();
          StringTokenizer t = new StringTokenizer(s,”;”);
          int num = Integer.parseInt(t.nextToken());
          String name = t.nextToken();
          output.println(“name: ” + name);
          output.println(“number: ” + num);
         } // end while
       } // end try
// continued on next page
```

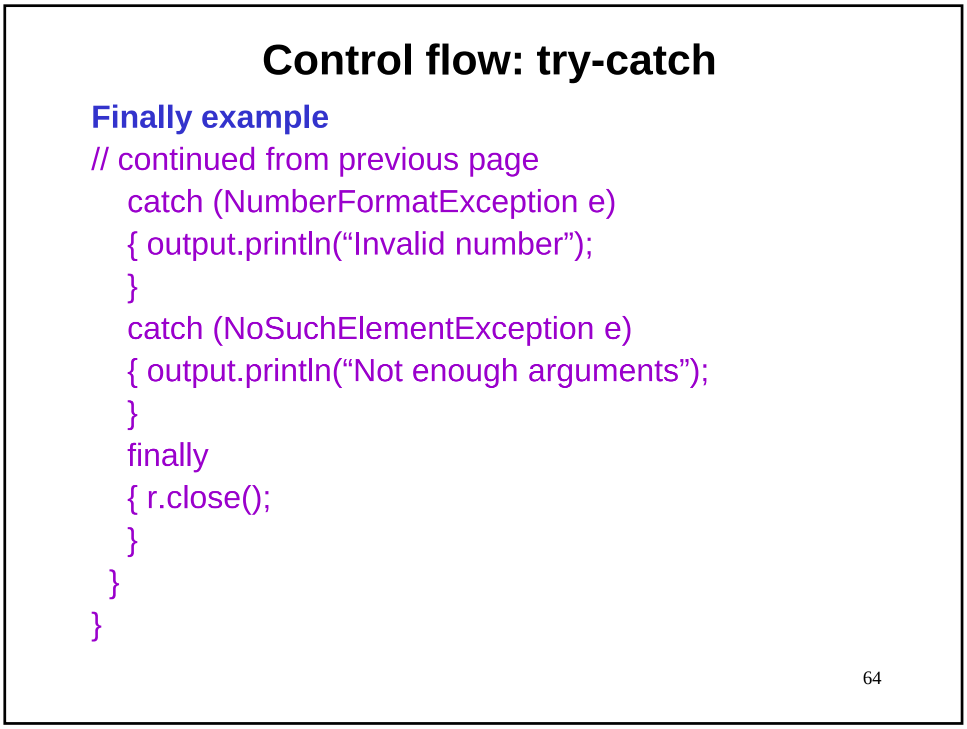


Control flow: try-catch

Finally example

```
// continued from previous page
    catch (NumberFormatException e)
    { output.println(“Invalid number”);
    }
    catch (NoSuchElementException e)
    { output.println(“Not enough arguments”);
    }



    }
  }
}
```

63



Control flow: try-catch

Finally example

```
// continued from previous page
    catch (NumberFormatException e)
    { output.println(“Invalid number”);
    }
    catch (NoSuchElementException e)
    { output.println(“Not enough arguments”);
    }
    finally
    { r.close();
    }
  }
}
```

64

Control flow: try-catch

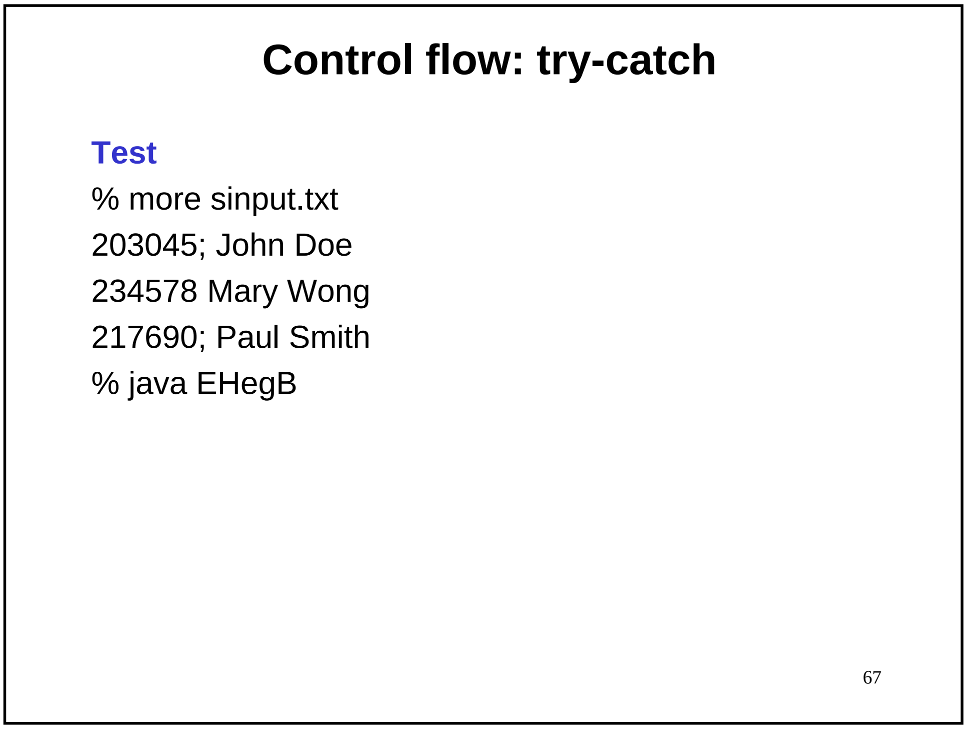
Test

```
% more sinput.txt
```

Control flow: try-catch

Test

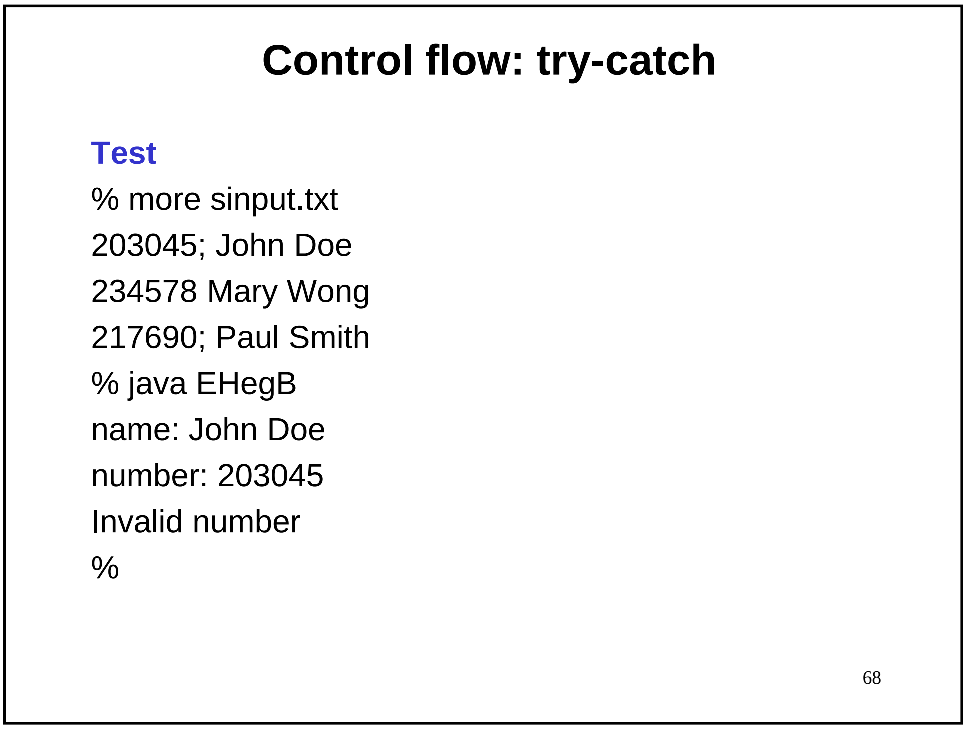
```
% more sinput.txt
203045; John Doe
234578 Mary Wong
217690; Paul Smith
%
```



Control flow: try-catch

Test

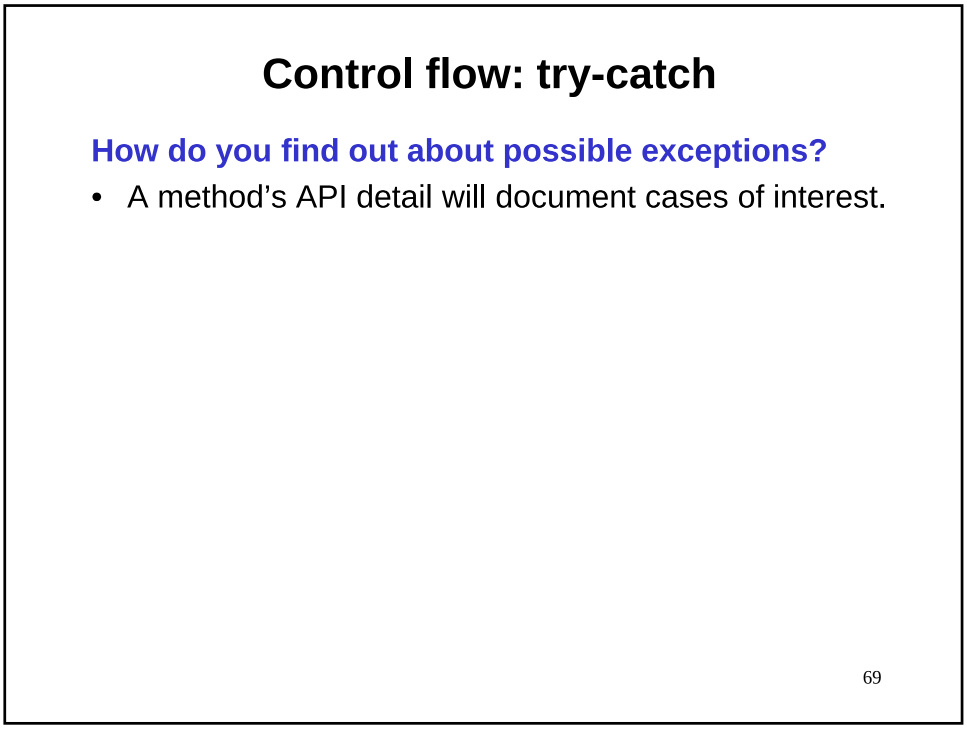
```
% more sinput.txt
203045; John Doe
234578 Mary Wong
217690; Paul Smith
% java EHegB
```



Control flow: try-catch

Test

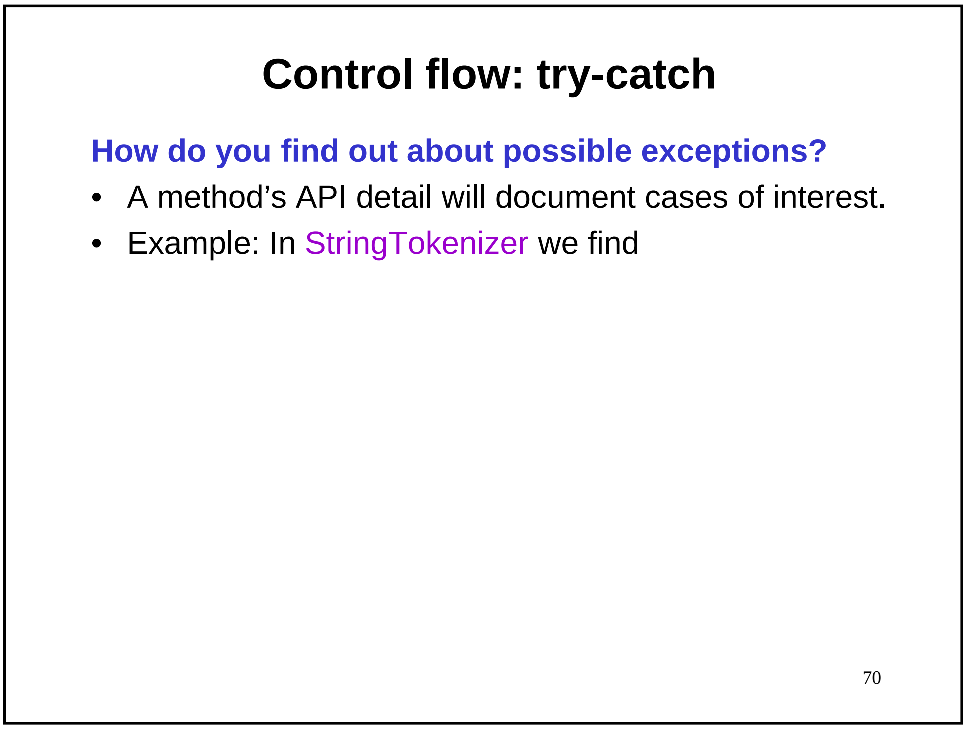
```
% more sinput.txt
203045; John Doe
234578 Mary Wong
217690; Paul Smith
% java EHegB
name: John Doe
number: 203045
Invalid number
%
```



Control flow: try-catch

How do you find out about possible exceptions?

- A method’s API detail will document cases of interest.



Control flow: try-catch

How do you find out about possible exceptions?

- A method’s API detail will document cases of interest.
- Example: In StringTokenizer we find

Control flow: try-catch

Method Detail

nextToken

public String nextToken(String delim)

Returns the next token in this string tokenizer's string. First, the set of characters considered to be delimiters by this StringTokenizer object is changed to be the characters in the string delim. Then the next token in the string after the current position is returned. The current position is advanced beyond the recognized token. The new delimiter set remains the default after this call.

Parameters:

delim - the new delimiters.

Returns:

the next token, after switching to the new delimiter set.

Throws:

NoSuchElementException - if there are no more tokens in this tokenizer's string.

Control flow: try-catch

How do you find out about possible exceptions?

- A method’s API detail will document cases of interest.
- Example: In Integer we find

Control flow: try-catch

Method Detail

parseInt

public static int parseInt(String s)

Parses the string argument as a signed decimal integer. The characters in the string must all be decimal digits, except that the first character may be an ASCII minus sign '-' ('\u002D') to indicate a negative value. The resulting integer value is returned, exactly as if the argument and the radix 10 were given as arguments to the parseInt(java.lang.String, int) method.

Parameters:

s - a String containing the int representation to be parsed

Returns:

the integer value represented by the argument in decimal.

Throws:

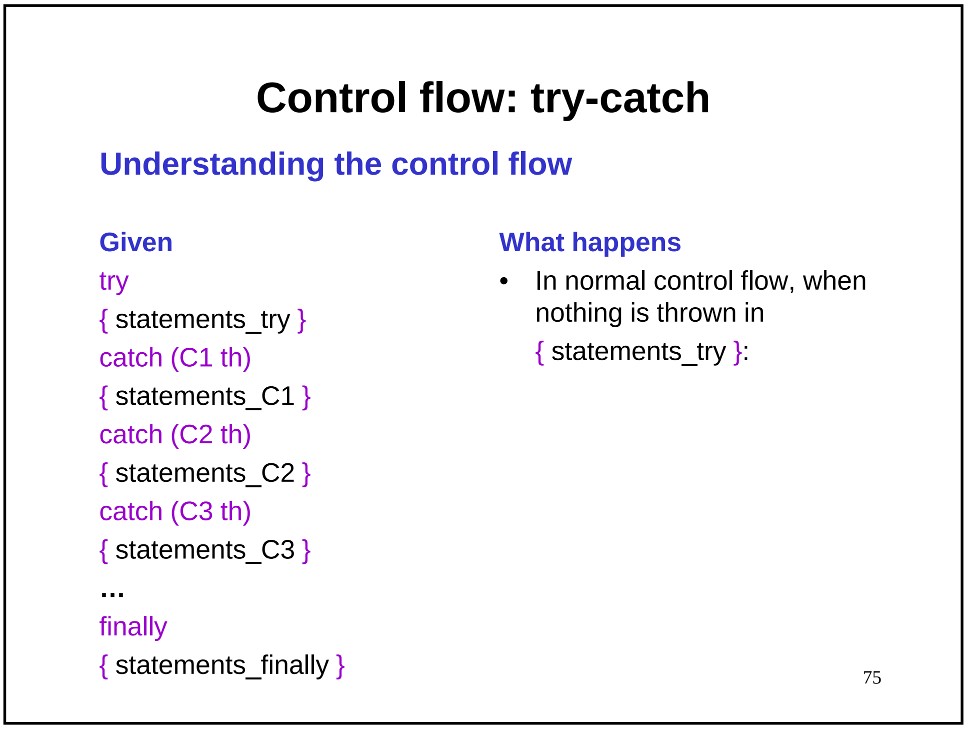
NumberFormatException - if the string does not contain a parsable integer.

Control flow: try-catch

Understanding the control flow

Given

```
try
{ statements_try }
catch (C1 th)
{ statements_C1 }
catch (C2 th)
{ statements_C2 }
catch (C3 th)
{ statements_C3 }
…
finally
{ statements_finally }
```



Control flow: try-catch

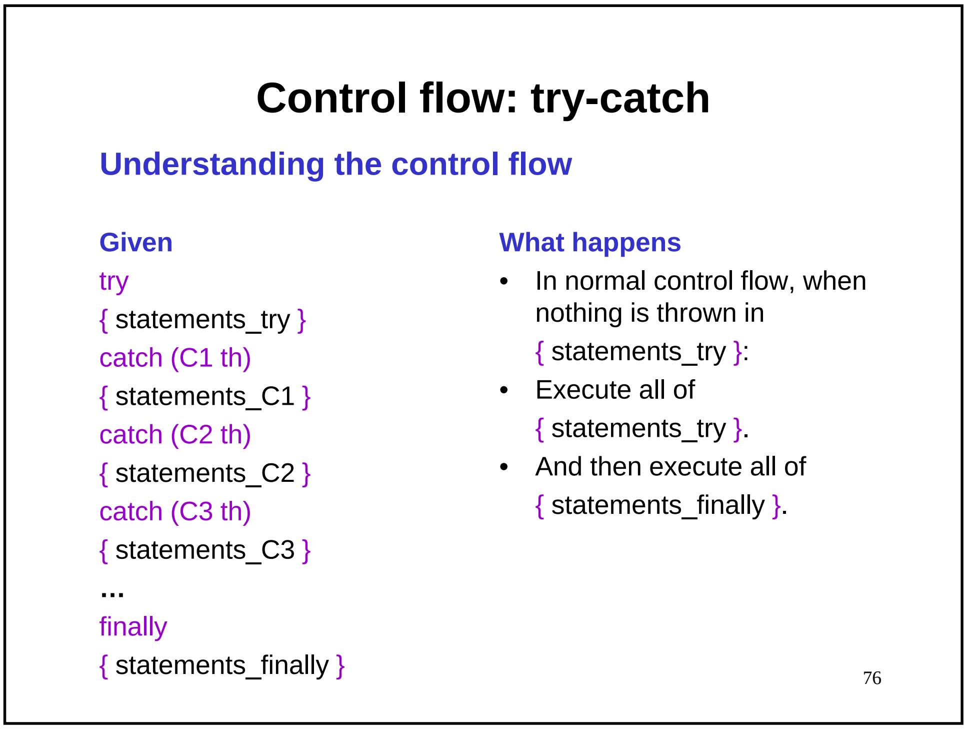
Understanding the control flow

Given

```
try
{ statements_try }
catch (C1 th)
{ statements_C1 }
catch (C2 th)
{ statements_C2 }
catch (C3 th)
{ statements_C3 }
…
finally
{ statements_finally }
```

What happens

- In normal control flow, when nothing is thrown in
  { statements_try }:



Control flow: try-catch

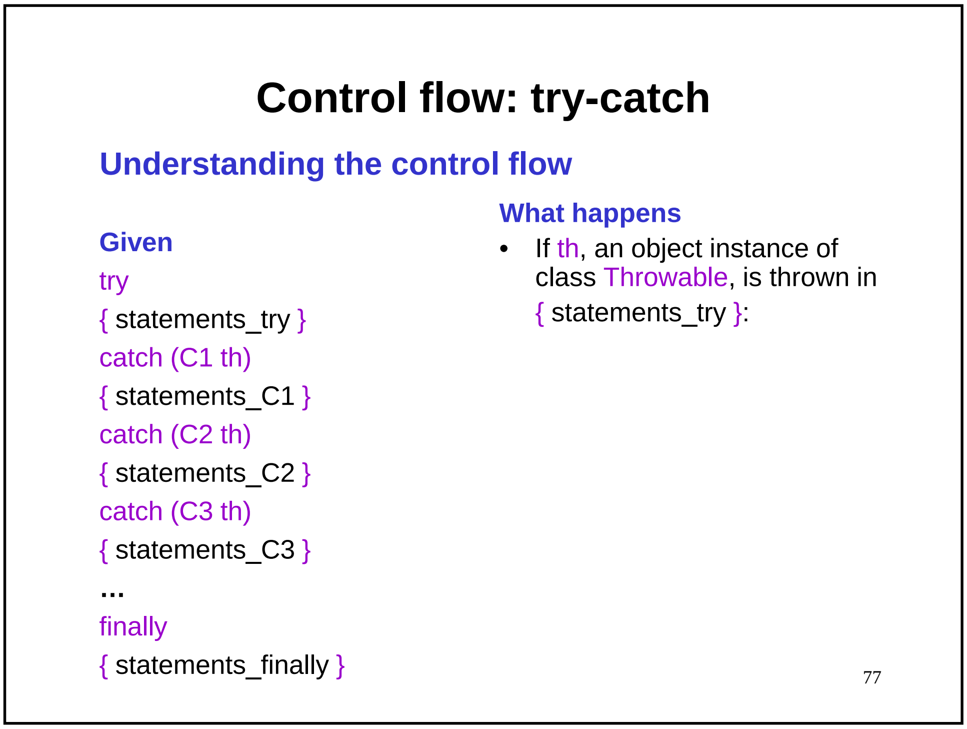
Understanding the control flow

Given

```
try
{ statements_try }
catch (C1 th)
{ statements_C1 }
catch (C2 th)
{ statements_C2 }
catch (C3 th)
{ statements_C3 }
…
finally
{ statements_finally }
```

What happens

- In normal control flow, when nothing is thrown in
  { statements_try }:
- Execute all of
  { statements_try }.
- And then execute all of
  { statements_finally }.



Control flow: try-catch

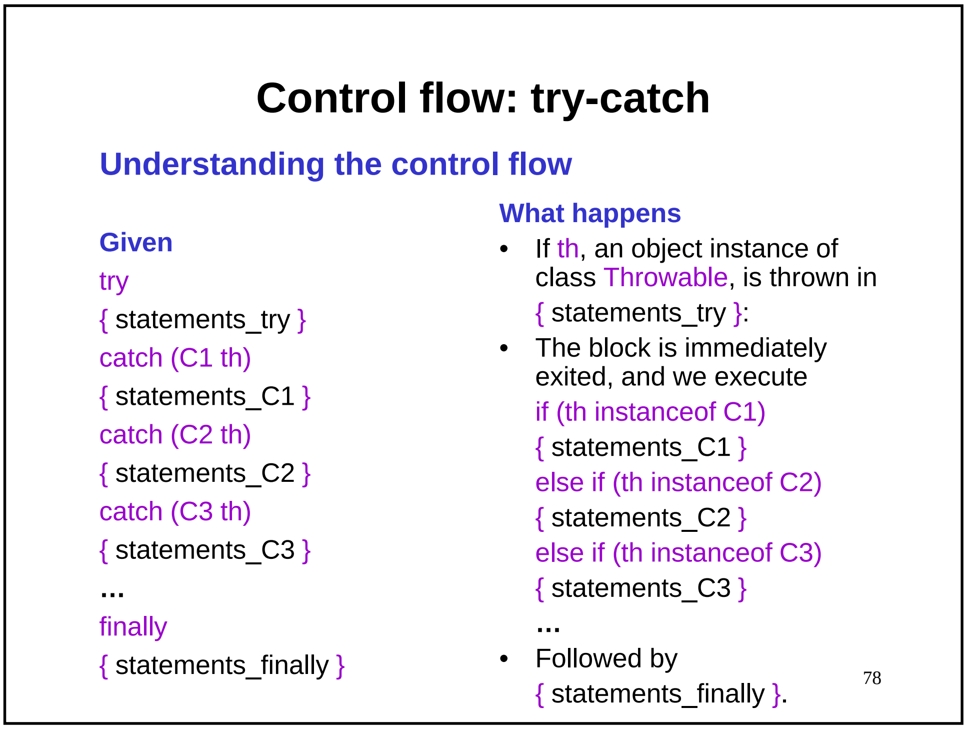
Understanding the control flow

Given

```
try
{ statements_try }
catch (C1 th)
{ statements_C1 }
catch (C2 th)
{ statements_C2 }
catch (C3 th)
{ statements_C3 }
…
finally
{ statements_finally }
```

What happens

- If th, an object instance of class Throwable, is thrown in { statements_try }:



Control flow: try-catch

Understanding the control flow

Given

```
try
{ statements_try }
catch (C1 th)
{ statements_C1 }
catch (C2 th)
{ statements_C2 }
catch (C3 th)
{ statements_C3 }
…
finally
{ statements_finally }
```

What happens

- If th, an object instance of class Throwable, is thrown in { statements_try }:
- The block is immediately exited, and we execute

```
if (th instanceof C1)
{ statements_C1 }
else if (th instanceof C2)
{ statements_C2 }
else if (th instanceof C3)
{ statements_C3 }
…
```

- Followed by { statements_finally }.

Control flow: try-catch

Understanding the control flow

Given

```
try
{ statements_try }
catch (C1 th)
{ statements_C1 }
catch (C2 th)
{ statements_C2 }
catch (C3 th)
{ statements_C3 }
…
finally
{ statements_finally }
```

What happens

- We see that C1, C2, C3, … must be classes.
- Indeed, they must be subclasses of Throwable.

Control flow: try-catch

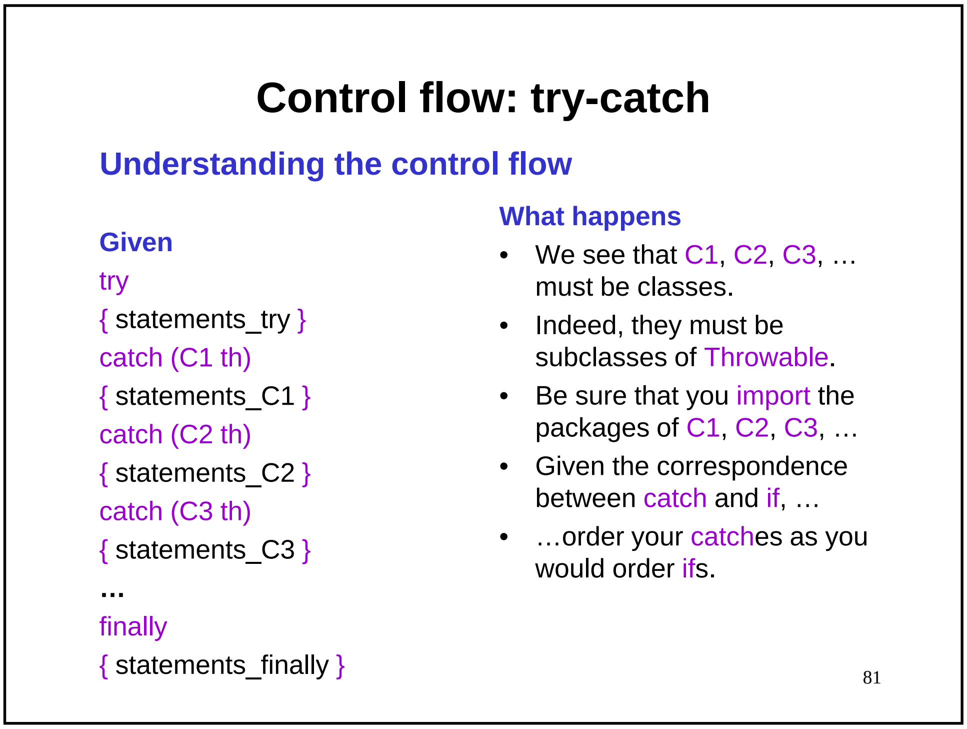
Understanding the control flow

Given

```
try
{ statements_try }
catch (C1 th)
{ statements_C1 }
catch (C2 th)
{ statements_C2 }
catch (C3 th)
{ statements_C3 }
…
finally
{ statements_finally }
```

What happens

- We see that C1, C2, C3, … must be classes.
- Indeed, they must be subclasses of Throwable.
- Be sure that you import the packages of C1, C2, C3, …



Control flow: try-catch

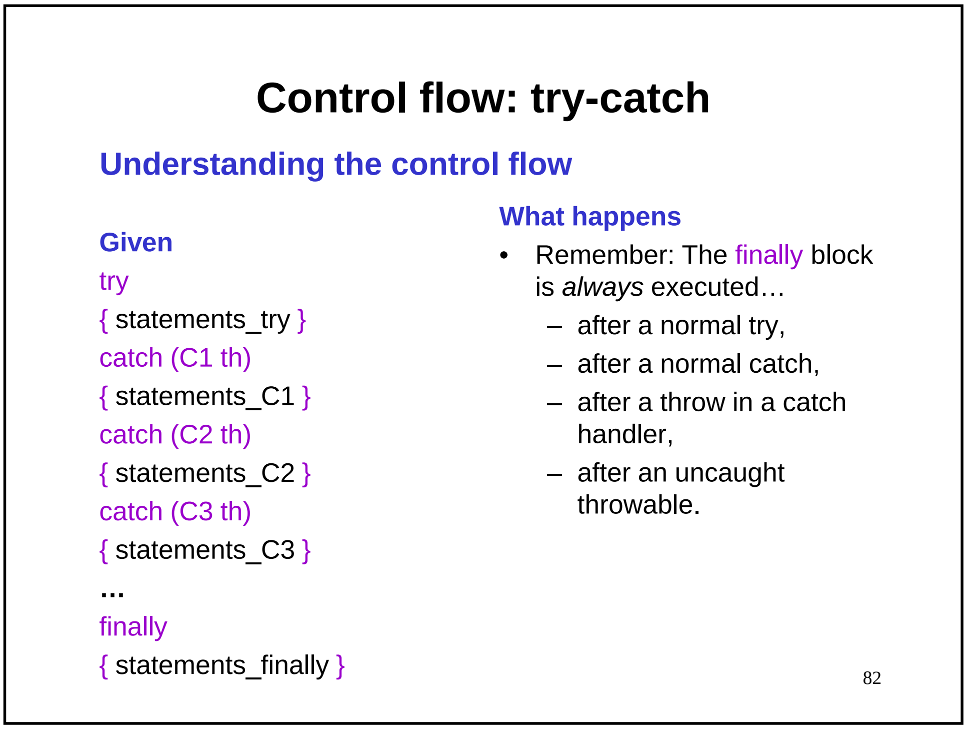
Understanding the control flow

Given

```
try
{ statements_try }
catch (C1 th)
{ statements_C1 }
catch (C2 th)
{ statements_C2 }
catch (C3 th)
{ statements_C3 }
…
finally
{ statements_finally }
```

What happens

- We see that C1, C2, C3, … must be classes.
- Indeed, they must be subclasses of Throwable.
- Be sure that you import the packages of C1, C2, C3, …
- Given the correspondence between catch and if, …
- …order your catches as you would order ifs.



Control flow: try-catch

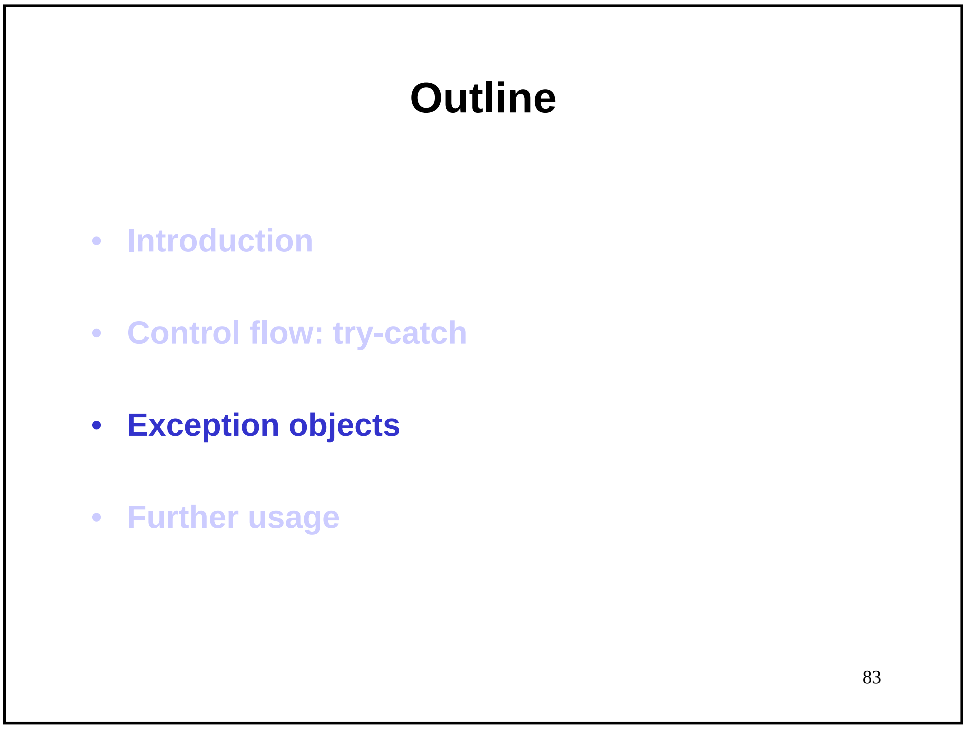
Understanding the control flow

Given

```
try
{ statements_try }
catch (C1 th)
{ statements_C1 }
catch (C2 th)
{ statements_C2 }
catch (C3 th)
{ statements_C3 }
…
finally
{ statements_finally }
```

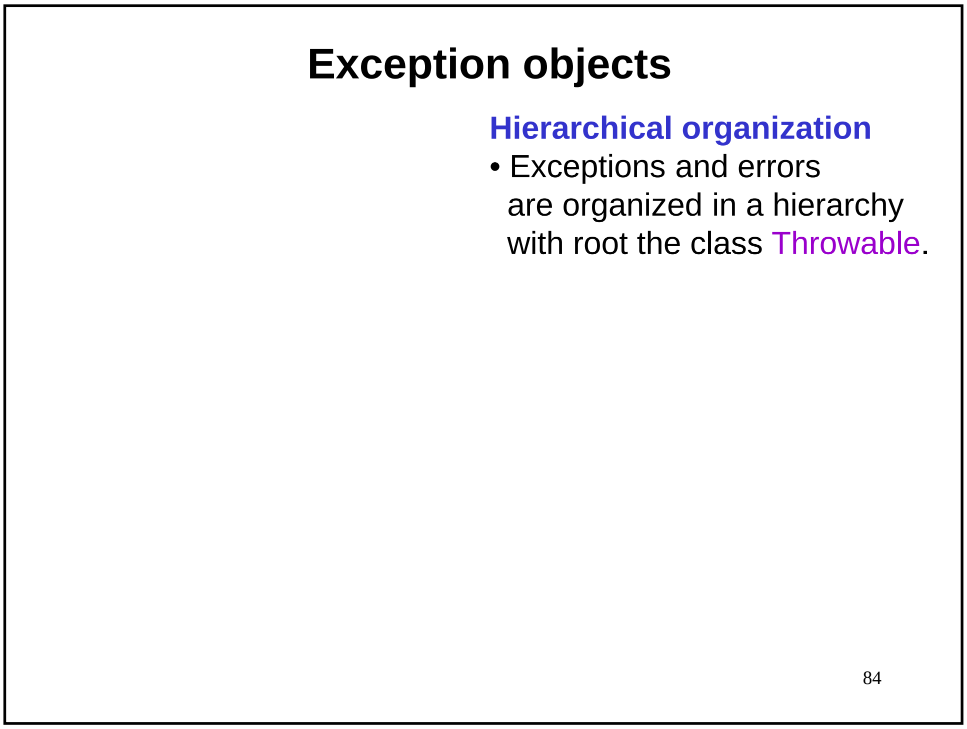
What happens

- Remember: The finally block is *always* executed…
  - after a normal try,
  - after a normal catch,
  - after a throw in a catch handler,
  - after an uncaught throwable.



Outline

- Introduction
- Control flow: try-catch
- **Exception objects**
- Further usage

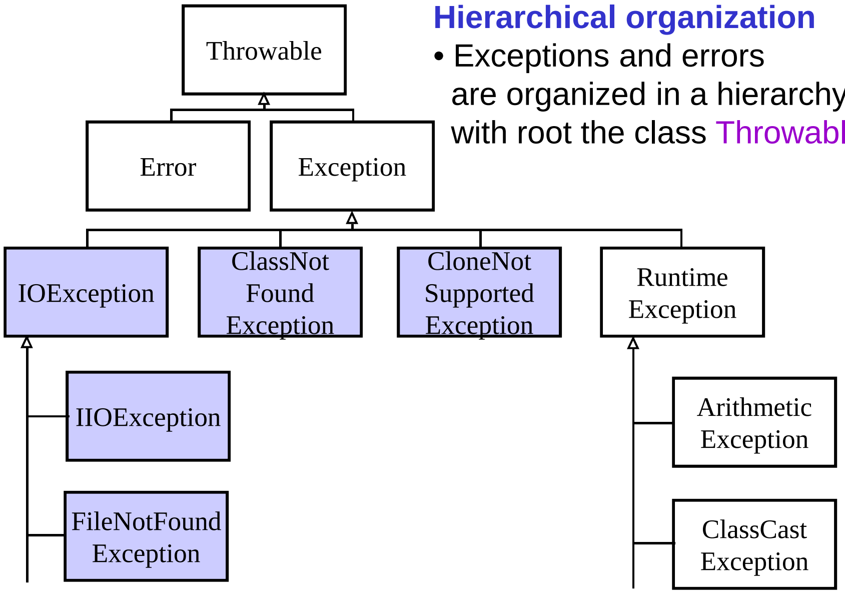


Exception objects

Hierarchical organization

- Exceptions and errors are organized in a hierarchy with root the class Throwable.

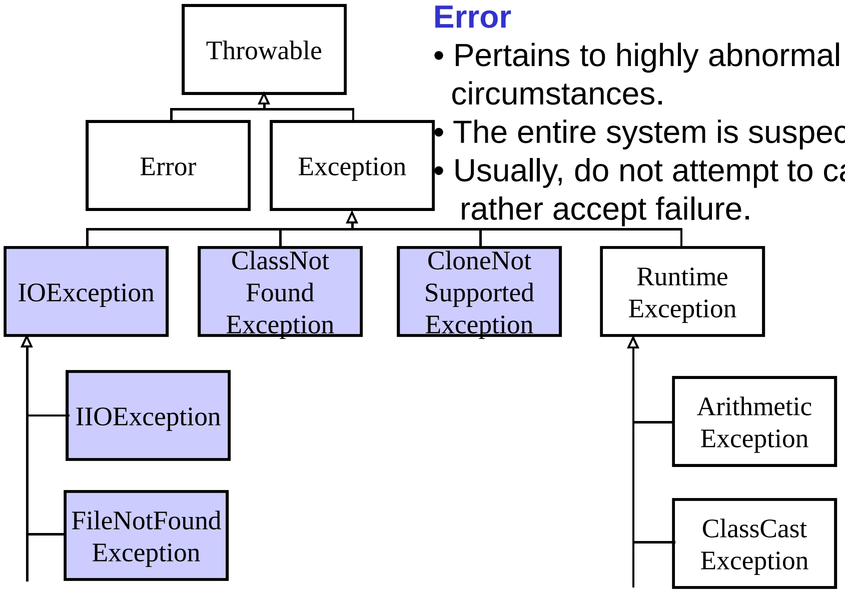
Exception objects



Hierarchical organization

- Exceptions and errors are organized in a hierarchy with root the class Throwable.

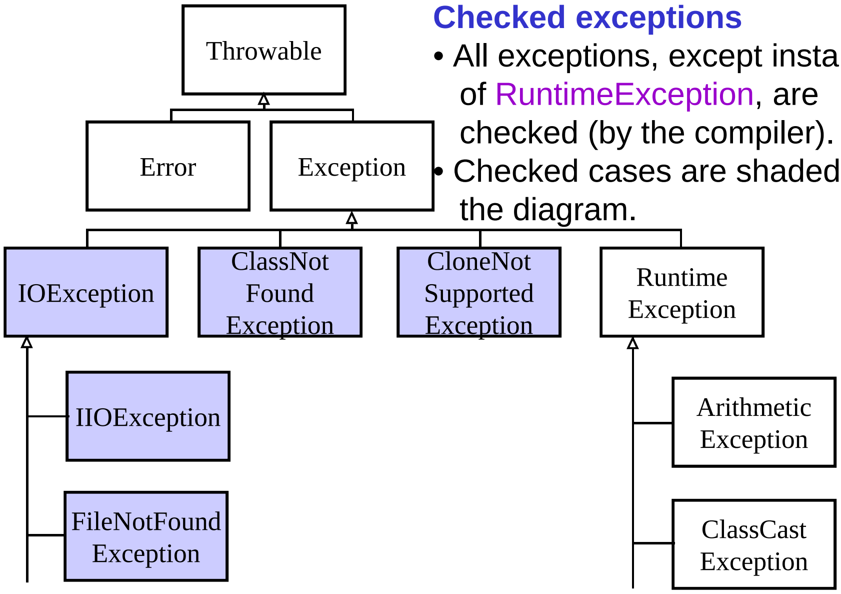
Exception objects



Error

- Pertains to highly abnormal circumstances.
- The entire system is suspect.
- Usually, do not attempt to catch; rather accept failure.

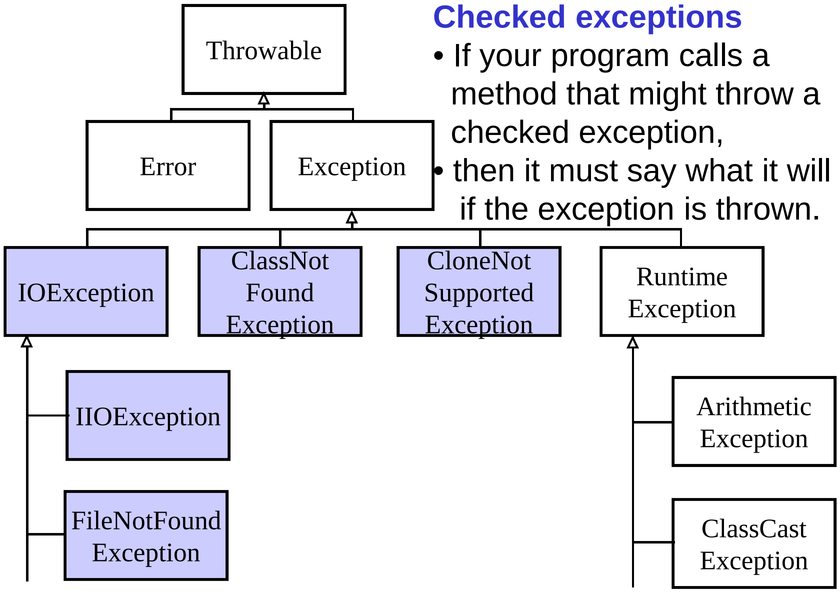
Exception objects



Checked exceptions

- All exceptions, except instances of RuntimeException, are checked (by the compiler).
- Checked cases are shaded in the diagram.

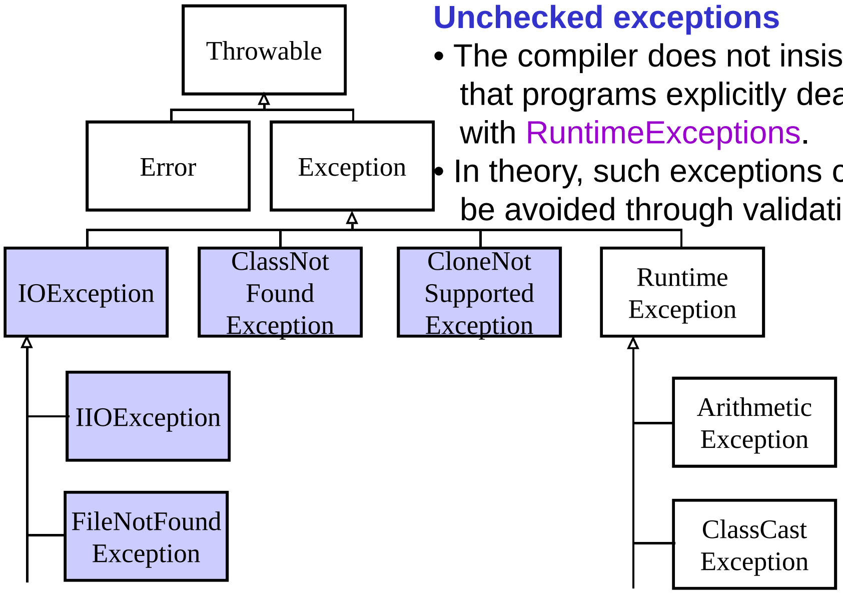
Exception objects



Checked exceptions

- If your program calls a method that might throw a checked exception,
- then it must say what it will do if the exception is thrown.

Exception objects



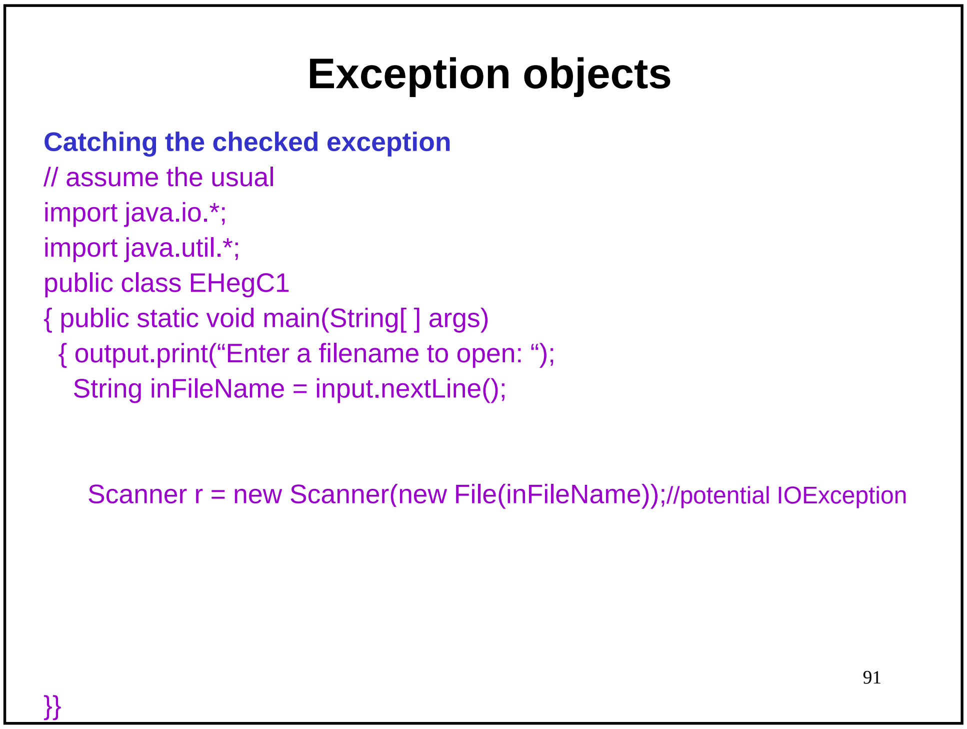
Unchecked exceptions

- The compiler does not insist that programs explicitly deal with RuntimeExceptions.
- In theory, such exceptions can be avoided through validation.

Exception objects

Explicitly dealing with a checked exception

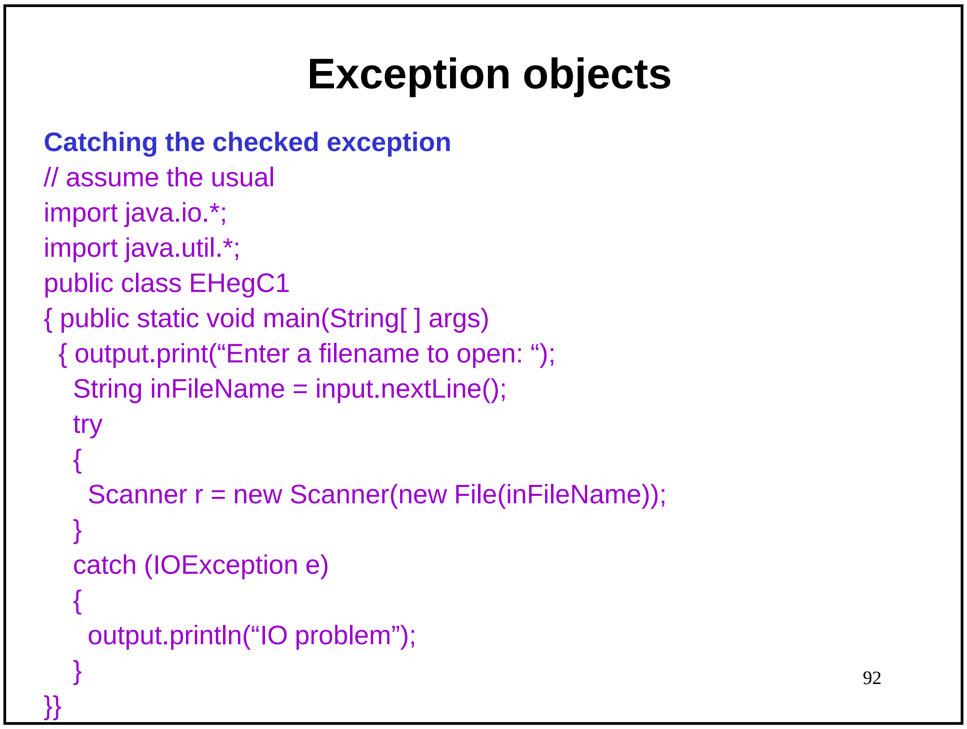
- We have noted that the compiler insists that a program that might confront a checked exception must explicitly deal with this possibility.
- (Reminder: To find out which exceptions might be thrown by a method, check its API detail.)
- There are two possible approaches
  1. Catch the anticipated checked exception.
  2. Explicitly acknowledge that your code (app) might, in turn, throw the checked exception (delegation outward).



Exception objects

Catching the checked exception

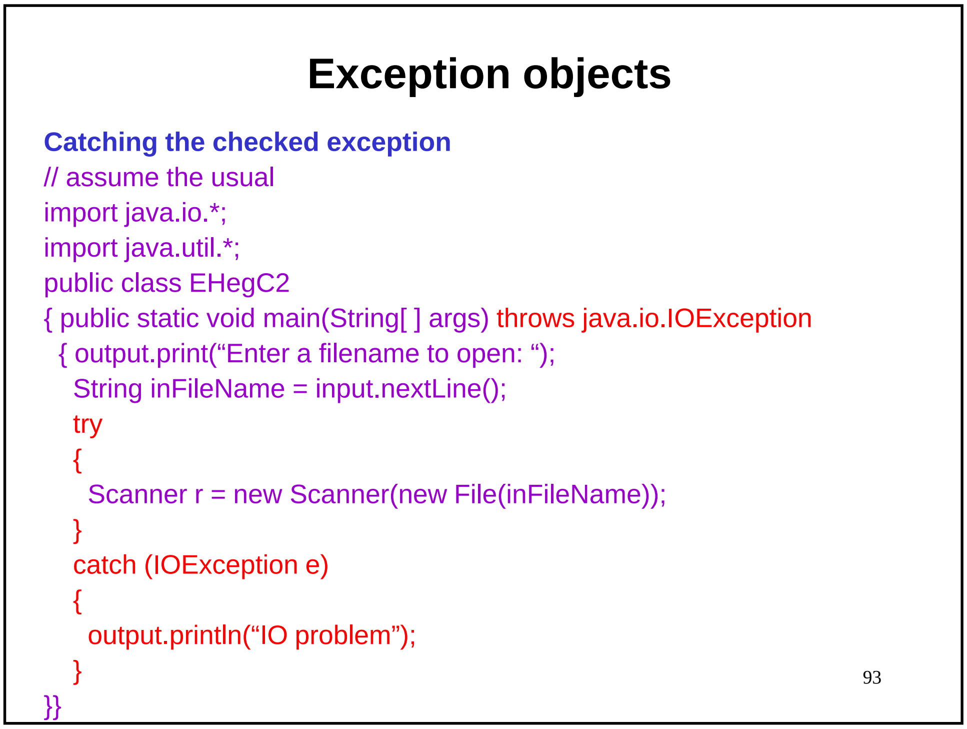
```
// assume the usual
import java.io.*;
import java.util.*;
public class EHegC1
{ public static void main(String[ ] args)
  { output.print(“Enter a filename to open: “);
    String inFileName = input.nextLine();


      Scanner r = new Scanner(new File(inFileName));//potential IOException




}}
```



Exception objects

Catching the checked exception

```
// assume the usual
import java.io.*;
import java.util.*;
public class EHegC1
{ public static void main(String[ ] args)
  { output.print(“Enter a filename to open: “);
    String inFileName = input.nextLine();
    try
    {
      Scanner r = new Scanner(new File(inFileName));
    }
    catch (IOException e)
    {
      output.println(“IO problem”);
    }
}}
```

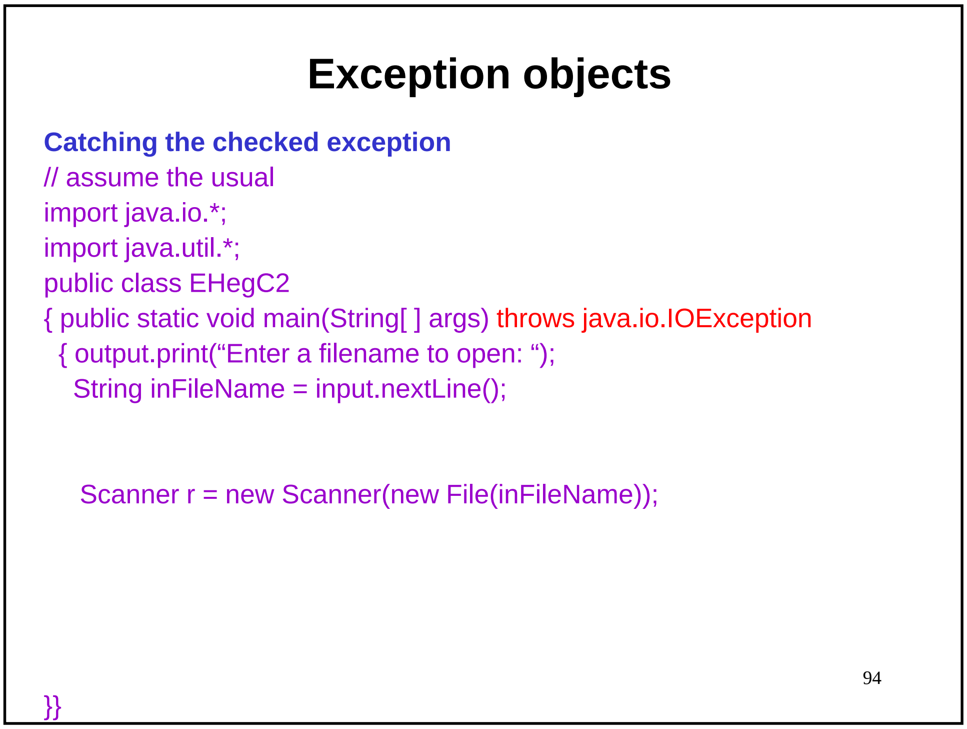


Exception objects

Catching the checked exception

```
// assume the usual
import java.io.*;
import java.util.*;
public class EHegC2
{ public static void main(String[ ] args) throws java.io.IOException
  { output.print(“Enter a filename to open: “);
    String inFileName = input.nextLine();
    try
    {
      Scanner r = new Scanner(new File(inFileName));
    }
    catch (IOException e)
    {
      output.println(“IO problem”);
    }
}}
```

93



Exception objects

Catching the checked exception

```
// assume the usual
import java.io.*;
import java.util.*;
public class EHegC2
{ public static void main(String[ ] args) throws java.io.IOException
  { output.print(“Enter a filename to open: “);
    String inFileName = input.nextLine();


     Scanner r = new Scanner(new File(inFileName));




}}
```

94

Exception objects

Catching the checked exception

```
// assume the usual
import java.io.*;
import java.util.*;
public class EHegC2
{ public static void main(String[ ] args) throws java.io.IOException
  { output.print(“Enter a filename to open: “);
    String inFileName = input.nextLine();


     Scanner r = new Scanner(new File(inFileName));




}}
```

Exception objects

Catching the checked exception

```
// assume the usual
import java.io.*;
import java.util.*;
public class EHegC2
{ public static void main(String[ ] args) throws java.io.IOException
  { output.print(“Enter a filename to open: “);
    String inFileName = input.nextLine();
    Scanner r = new Scanner(new File(inFileName));
    // Do something with r and don’t forget to close when done.
}}
```

Exception objects

Dealing with unchecked exceptions

- We have noted that the compiler *does not* insist that a program that might confront an unchecked exception (RuntimeException) deal with this possibility.
- Essentially, this is because runtime errors typically can be prevented through validation and the designers of Java chose to leave that option available to individual programmers.

Exception objects

Dealing with unchecked exceptions

- We have noted that the compiler *does not* insist that a program that might confront an unchecked exception (RuntimeException) deal with this possibility.
- Essentially, this is because runtime errors typically can be prevented through validation and the designers of Java chose to leave that option available to individual programmers.
  - Check to see if a String contains characters consistent with an int before attempting parseInt
  - Check to see if a user input has appropriate number of values,
  - Etc...

Exception objects

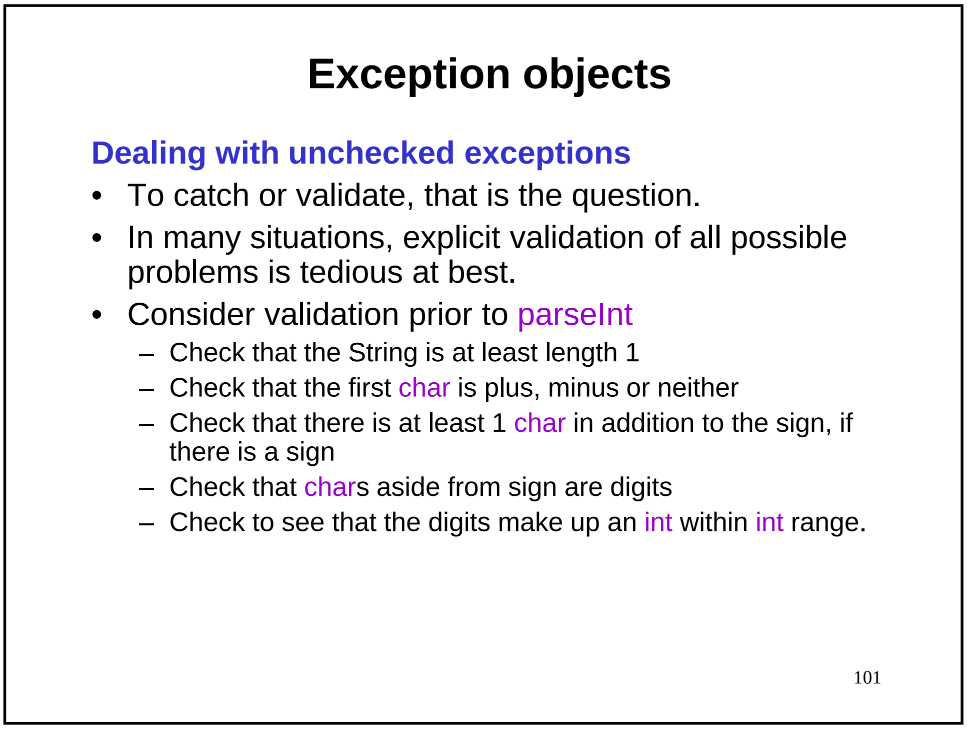
Dealing with unchecked exceptions

- We have noted that the compiler *does not* insist that a program that might confront an unchecked exception (RuntimeException) deal with this possibility.
- Essentially, this is because runtime errors typically can be prevented through validation and the designers of Java chose to leave that option available to individual programmers.
  - Check to see if a String contains characters consistent with an int before attempting parseInt
  - Check to see if a user input has appropriate number of values,
  - Etc...
- By checking these exceptions, it would force redundant code on those who opted for validation.

Exception objects

Dealing with unchecked exceptions

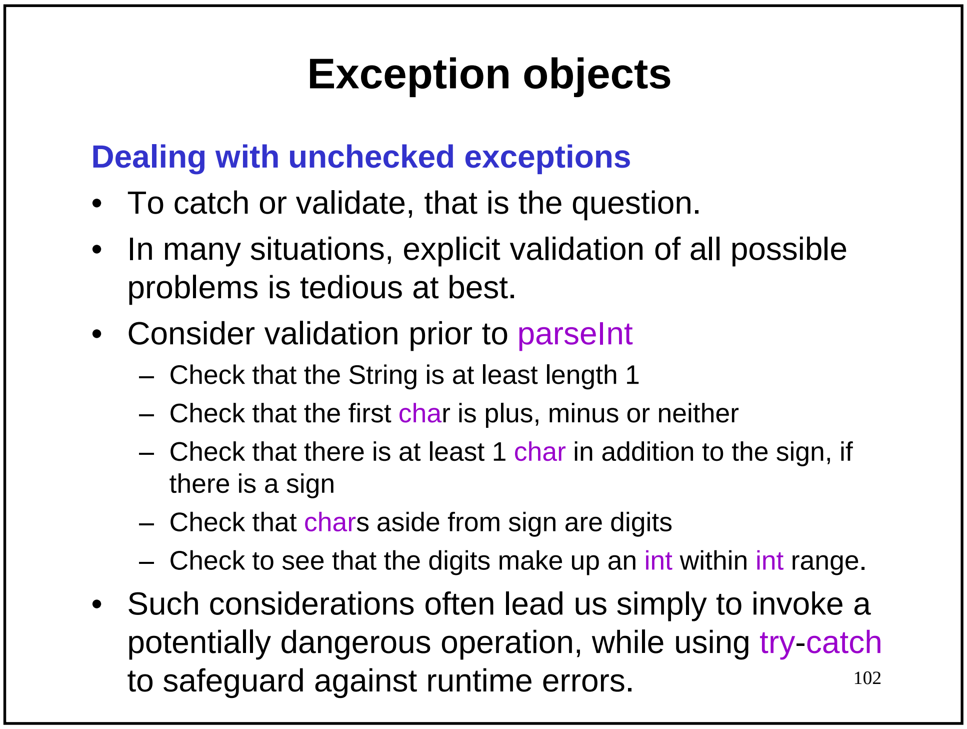
- To catch or validate, that is the question.
- In many situations, explicit validation of all possible problems is tedious at best.



Exception objects

Dealing with unchecked exceptions

- To catch or validate, that is the question.
- In many situations, explicit validation of all possible problems is tedious at best.
- Consider validation prior to parseInt
  - Check that the String is at least length 1
  - Check that the first char is plus, minus or neither
  - Check that there is at least 1 char in addition to the sign, if there is a sign
  - Check that chars aside from sign are digits
  - Check to see that the digits make up an int within int range.



Exception objects

Dealing with unchecked exceptions

- To catch or validate, that is the question.
- In many situations, explicit validation of all possible problems is tedious at best.
- Consider validation prior to parseInt
  - Check that the String is at least length 1
  - Check that the first char is plus, minus or neither
  - Check that there is at least 1 char in addition to the sign, if there is a sign
  - Check that chars aside from sign are digits
  - Check to see that the digits make up an int within int range.

- Such considerations often lead us simply to invoke a potentially dangerous operation, while using try-catch to safeguard against runtime errors.

Exception objects

Dealing with unchecked exceptions

- Following is the API summary of the RuntimeException class.

Exception objects

Dealing with unchecked exceptions

- Following is the API summary of the RuntimeException class.
- In java.lang we find

public class RuntimeException
extends Exception

RuntimeException is the superclass of those exceptions that can be thrown during the normal operation of the Java Virtual Machine. A method is not required to declare in its throws clause any subclasses of RuntimeException that might be thrown during the execution of the method but not caught.

Exception objects

Dealing with unchecked exceptions

- Following are some of the most common subclasses of the RuntimeException class.

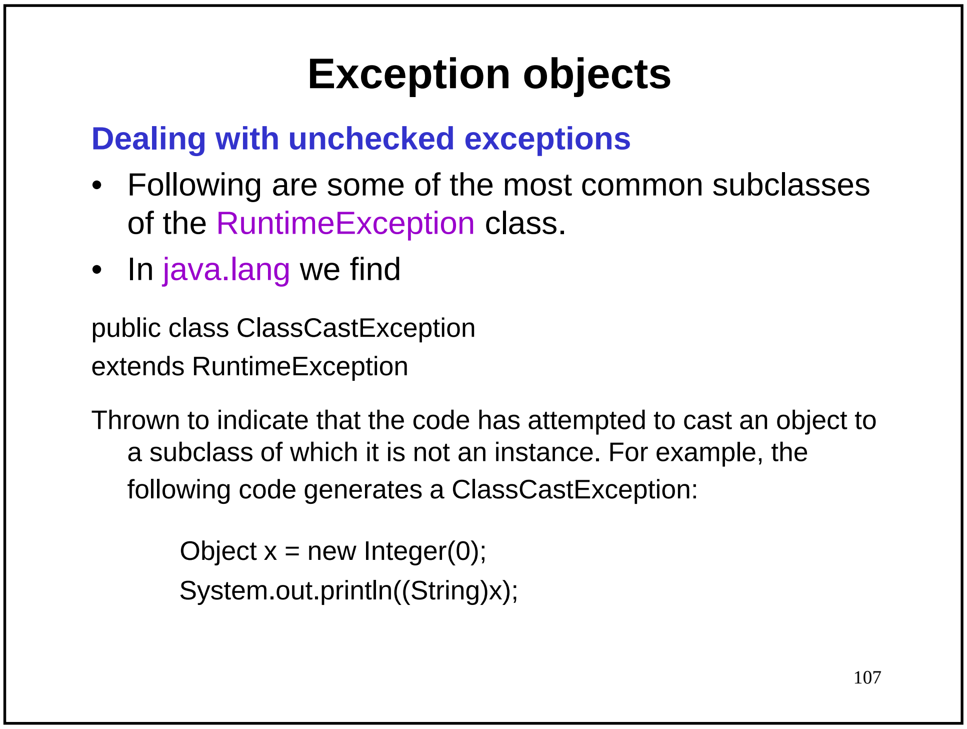
Exception objects

Dealing with unchecked exceptions

- Following are some of the most common subclasses of the RuntimeException class.
- In java.lang we find

public class ArithmeticException
extends RuntimeException

Thrown when an exceptional arithmetic condition has occurred. For example, an integer "divide by zero" throws an instance of this class.



Exception objects

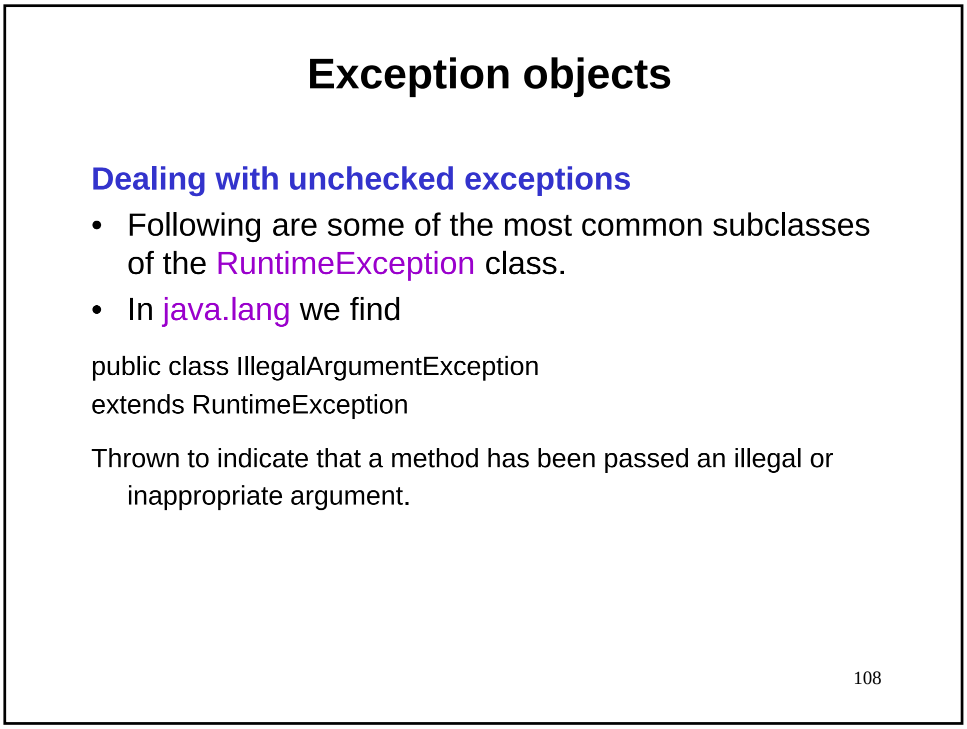
Dealing with unchecked exceptions

- Following are some of the most common subclasses of the RuntimeException class.
- In java.lang we find

public class ClassCastException
extends RuntimeException

Thrown to indicate that the code has attempted to cast an object to a subclass of which it is not an instance. For example, the following code generates a ClassCastException:

```
Object x = new Integer(0);
System.out.println((String)x);
```



Exception objects

Dealing with unchecked exceptions

- Following are some of the most common subclasses of the RuntimeException class.
- In java.lang we find

public class IllegalArgumentException
extends RuntimeException

Thrown to indicate that a method has been passed an illegal or inappropriate argument.

Exception objects

Dealing with unchecked exceptions

- Following are some of the most common subclasses of the RuntimeException class.
- In java.lang we find

public class IndexOutOfBoundsException
extends RuntimeException

Thrown to indicate that an index of some sort (such as to an array, to a string, or to a vector) is out of range. Applications can subclass this class to indicate similar exceptions.

Exception objects

Dealing with unchecked exceptions

- Following are some of the most common subclasses of the RuntimeException class.
- In java.util we find

public class NoSuchElementException
extends RuntimeException

Thrown by the nextElement method of an Enumeration to indicate that there are no more elements in the enumeration.

Exception objects

Dealing with unchecked exceptions

- Following are some of the most common subclasses of the RuntimeException class.
- In java.lang we find

public class NullPointerException
extends RuntimeException

Thrown when an application attempts to use null in a case where an object is required. These include:

Calling the instance method of a null object.
Accessing or modifying the field of a null object.
Taking the length of null as if it were an array.
Accessing or modifying the slots of null as if it were an array.
Throwing null as if it were a Throwable value.

Applications should throw instances of this class to indicate other illegal uses of the null object.

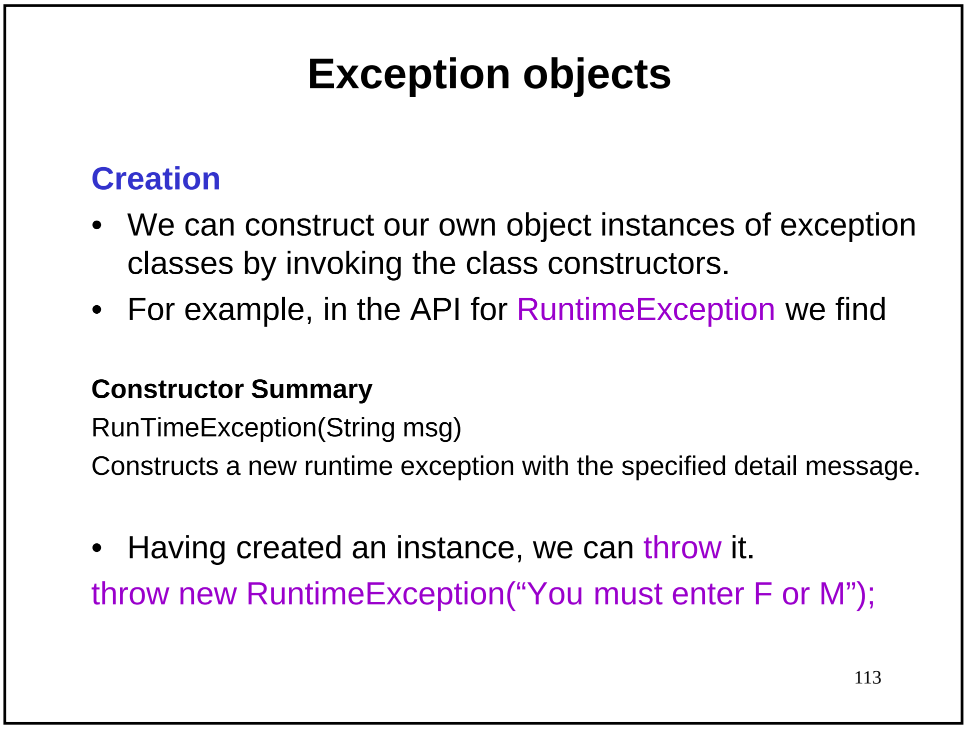
Exception objects

Dealing with unchecked exceptions

- Following are some of the most common subclasses of the RuntimeException class.
- In java.lang we find

public class NumberFormatException
extends IllegalArgumentException

Thrown to indicate that the application has attempted to convert a string to one of the numeric types, but that the string does not have the appropriate format.



Exception objects

Creation

- We can construct our own object instances of exception classes by invoking the class constructors.
- For example, in the API for RuntimeException we find

Constructor Summary

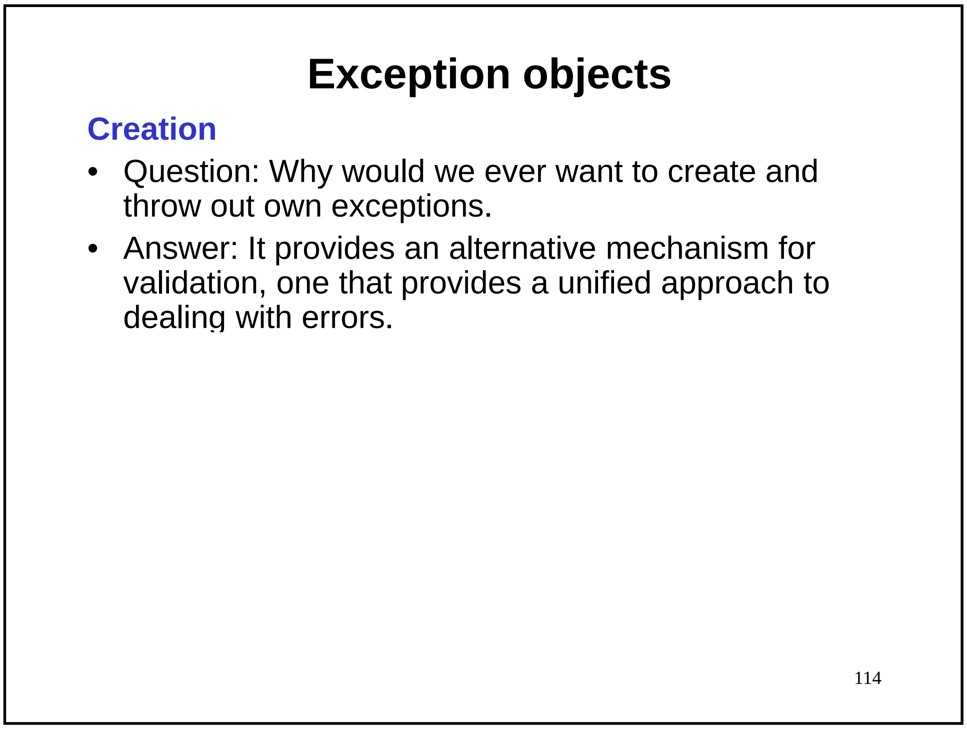
RunTimeException(String msg)

Constructs a new runtime exception with the specified detail message.

- Having created an instance, we can throw it.

throw new RuntimeException(“You must enter F or M”);

113



Exception objects

Creation

- Question: Why would we ever want to create and throw out own exceptions.
- Answer: It provides an alternative mechanism for validation, one that provides a unified approach to dealing with errors.

114

Exception objects

Creation

- Question: Why would we ever want to create and throw out own exceptions.
- Answer: It provides an alternative mechanism for validation, one that provides a unified approach to dealing with errors.
- Example

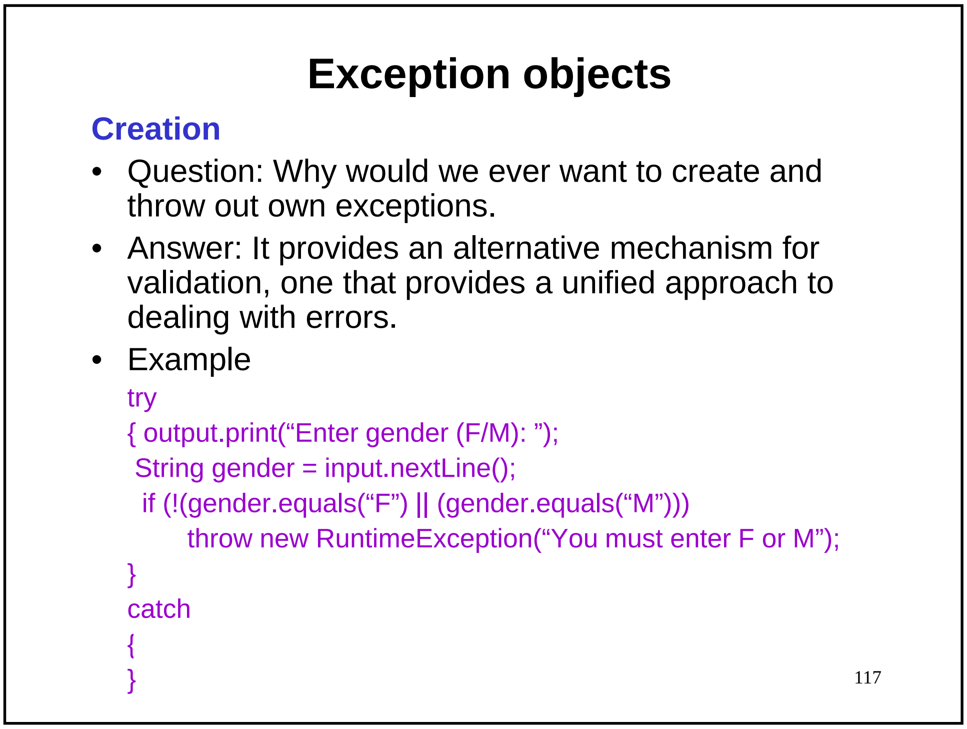
```
try
{



}
catch
{
}
```

Exception objects

Creation

- Question: Why would we ever want to create and throw out own exceptions.
- Answer: It provides an alternative mechanism for validation, one that provides a unified approach to dealing with errors.
- Example

```
try
{ output.print(“Enter gender (F/M): ”);
 String gender = input.nextLine();


}
catch
{
}
```



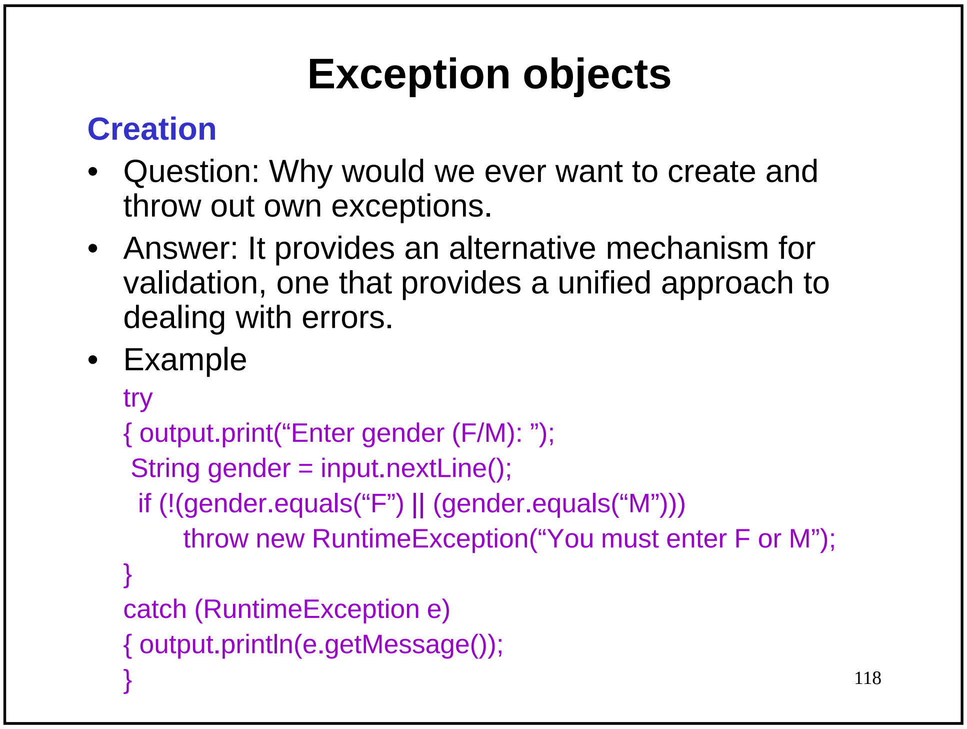
Exception objects

Creation

- Question: Why would we ever want to create and throw out own exceptions.
- Answer: It provides an alternative mechanism for validation, one that provides a unified approach to dealing with errors.
- Example

```
try
{ output.print(“Enter gender (F/M): ”);
 String gender = input.nextLine();
  if (!(gender.equals(“F”) || (gender.equals(“M”)))
        throw new RuntimeException(“You must enter F or M”);
}
catch
{
}
```

117



Exception objects

Creation

- Question: Why would we ever want to create and throw out own exceptions.
- Answer: It provides an alternative mechanism for validation, one that provides a unified approach to dealing with errors.
- Example

```
try
{ output.print(“Enter gender (F/M): ”);
 String gender = input.nextLine();
  if (!(gender.equals(“F”) || (gender.equals(“M”)))
        throw new RuntimeException(“You must enter F or M”);
}
catch (RuntimeException e)
{ output.println(e.getMessage());
}
```

118

Exception objects

Method invocation

- Since all exception classes are subclasses of Throwable, they inherit its features (methods and attributes).
- As usual, to find out about these features we consult the API.
- Let's consider two that often are of use.

Exception objects

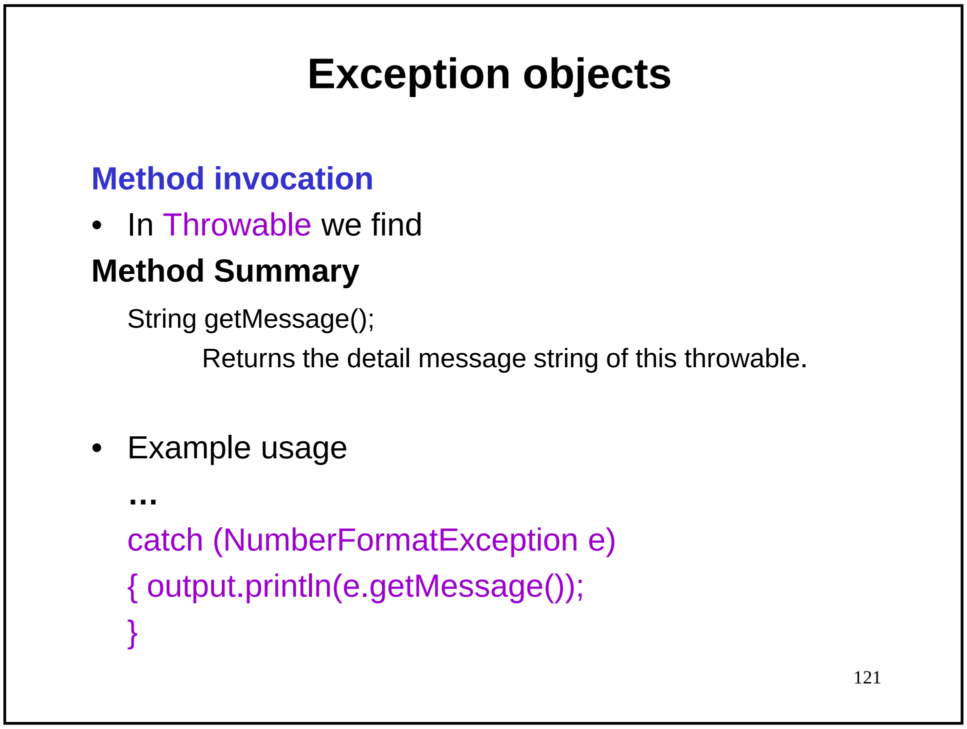
Method invocation

- In Throwable we find

Method Summary

String getMessage();

Returns the detail message string of this throwable.



Exception objects

Method invocation

- In Throwable we find

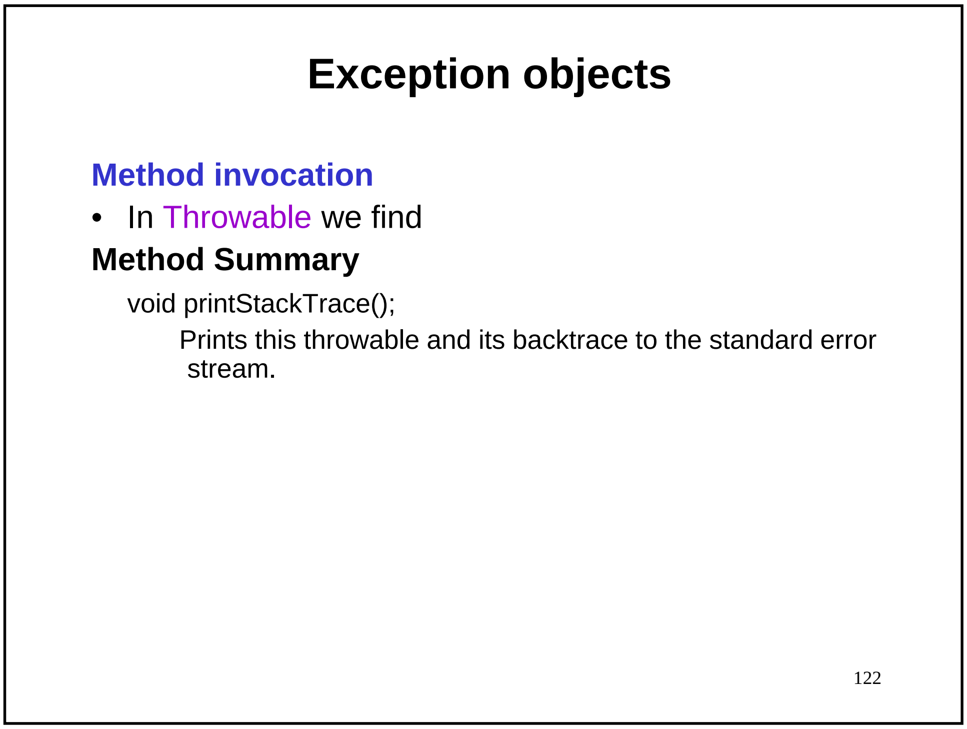
Method Summary

String getMessage();

Returns the detail message string of this throwable.

- Example usage

```
…
catch (NumberFormatException e)
{ output.println(e.getMessage());
}
```



Exception objects

Method invocation

- In Throwable we find

Method Summary

void printStackTrace();

Prints this throwable and its backtrace to the standard error stream.

Exception objects

Method invocation

- In Throwable we find

Method Summary

void printStackTrace();

Prints this throwable and its backtrace to the standard error stream.

- Example usage

```
…
catch (NumberFormatException e)
{ output.println(e.getStackTrace());
}
```

Exception objects

Recall

- Consider the following code

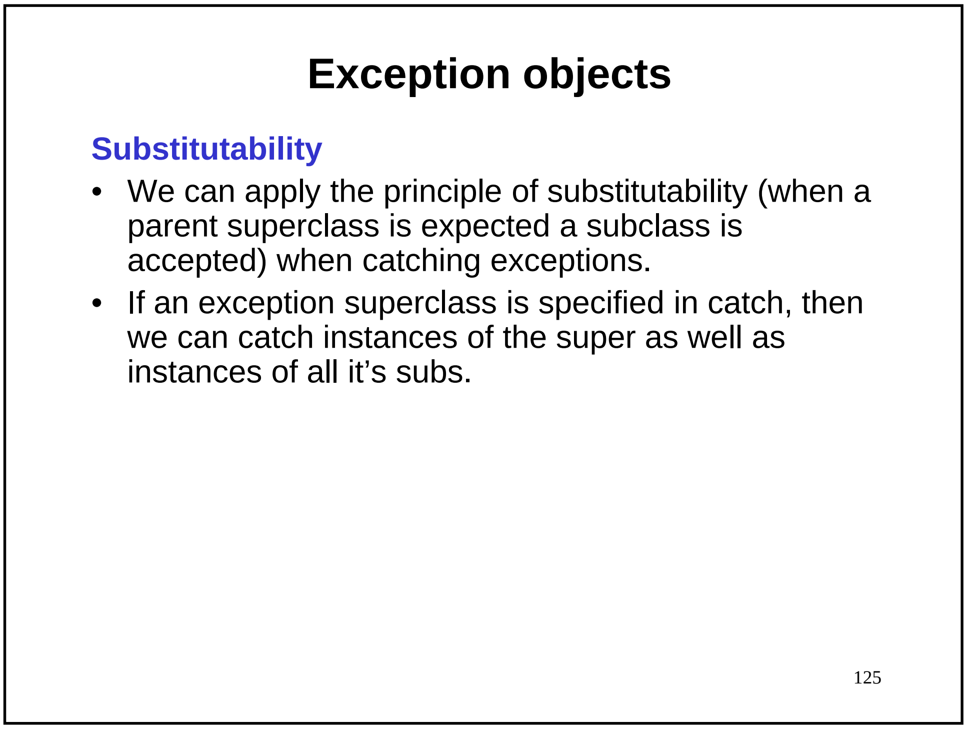
```
output.print("Enter the first integer: ");
int a = input.nextInt();
output.print("Enter the second: ");
int b = input.nextInt();
int c = a/b;
output.println("Their quotient is: " + c);
```

- and example run

```
Enter the first integer: 8
Enter the second: 0
Exception in thread "main"
java.lang.ArithmeticException: / by zero
          at Quotient.main(Quotient.java:16)
```

Exception message.

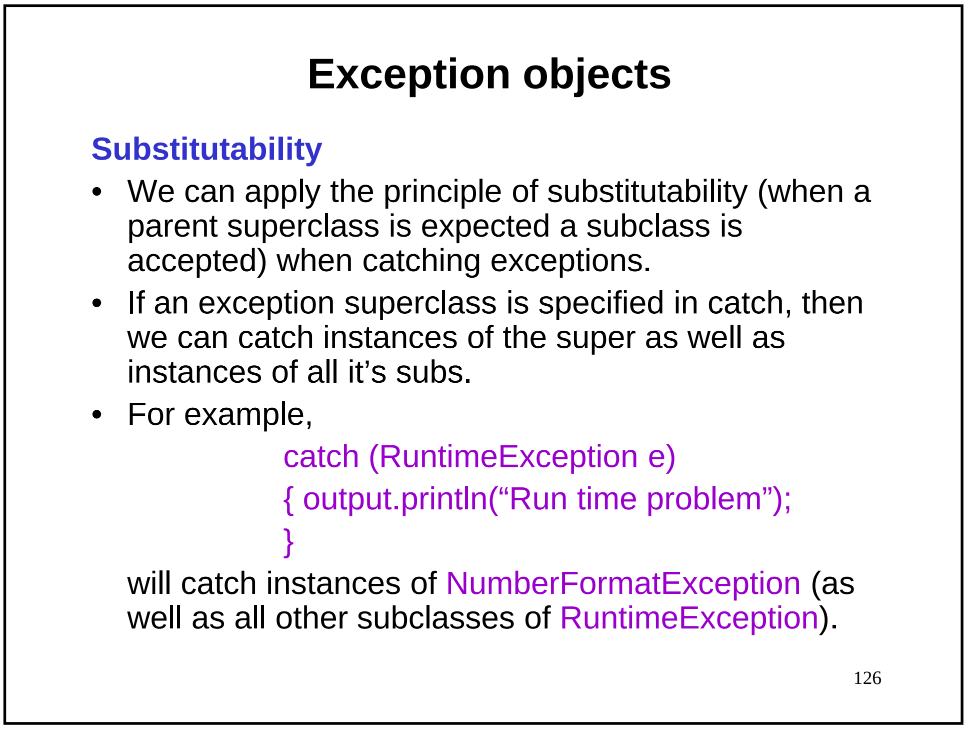
Stack trace.



Exception objects

Substitutability

- We can apply the principle of substitutability (when a parent superclass is expected a subclass is accepted) when catching exceptions.
- If an exception superclass is specified in catch, then we can catch instances of the super as well as instances of all it's subs.



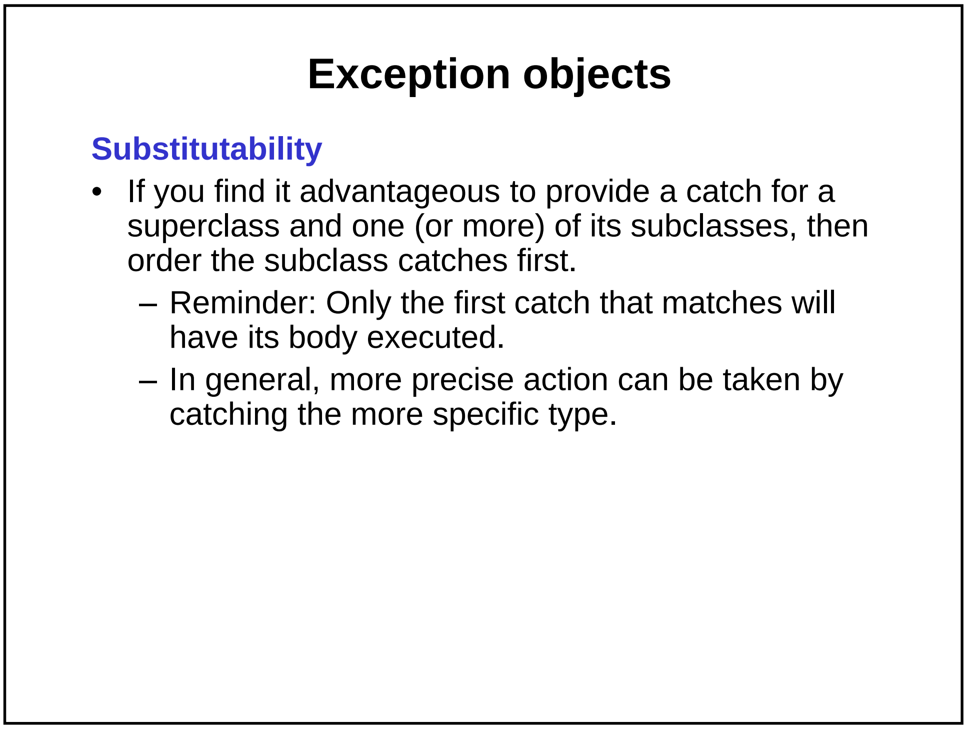
Exception objects

Substitutability

- We can apply the principle of substitutability (when a parent superclass is expected a subclass is accepted) when catching exceptions.
- If an exception superclass is specified in catch, then we can catch instances of the super as well as instances of all it's subs.
- For example,

```
catch (RuntimeException e)
{ output.println("Run time problem");
}
```

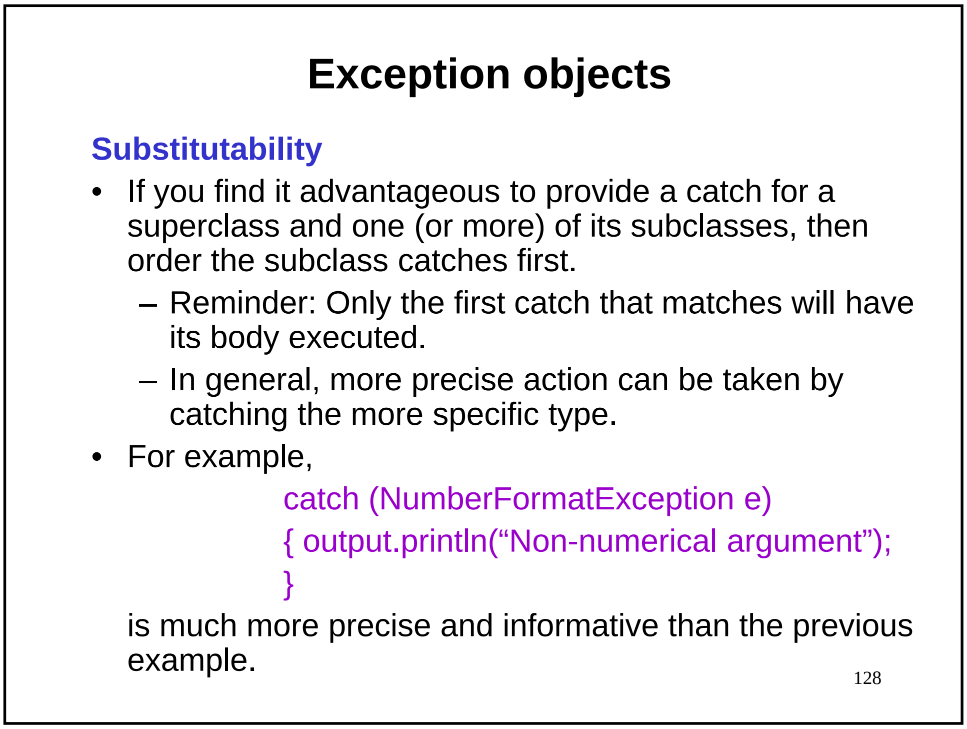
will catch instances of NumberFormatException (as well as all other subclasses of RuntimeException).



Exception objects

Substitutability

- If you find it advantageous to provide a catch for a superclass and one (or more) of its subclasses, then order the subclass catches first.
  - Reminder: Only the first catch that matches will have its body executed.
  - In general, more precise action can be taken by catching the more specific type.



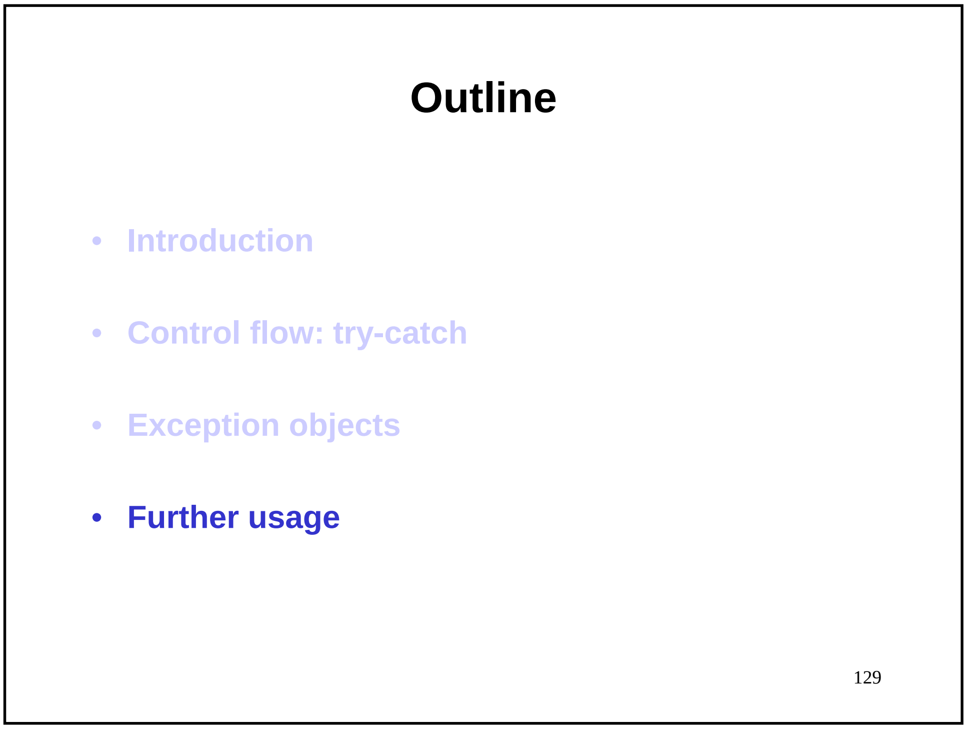
Exception objects

Substitutability

- If you find it advantageous to provide a catch for a superclass and one (or more) of its subclasses, then order the subclass catches first.
  - Reminder: Only the first catch that matches will have its body executed.
  - In general, more precise action can be taken by catching the more specific type.
- For example,

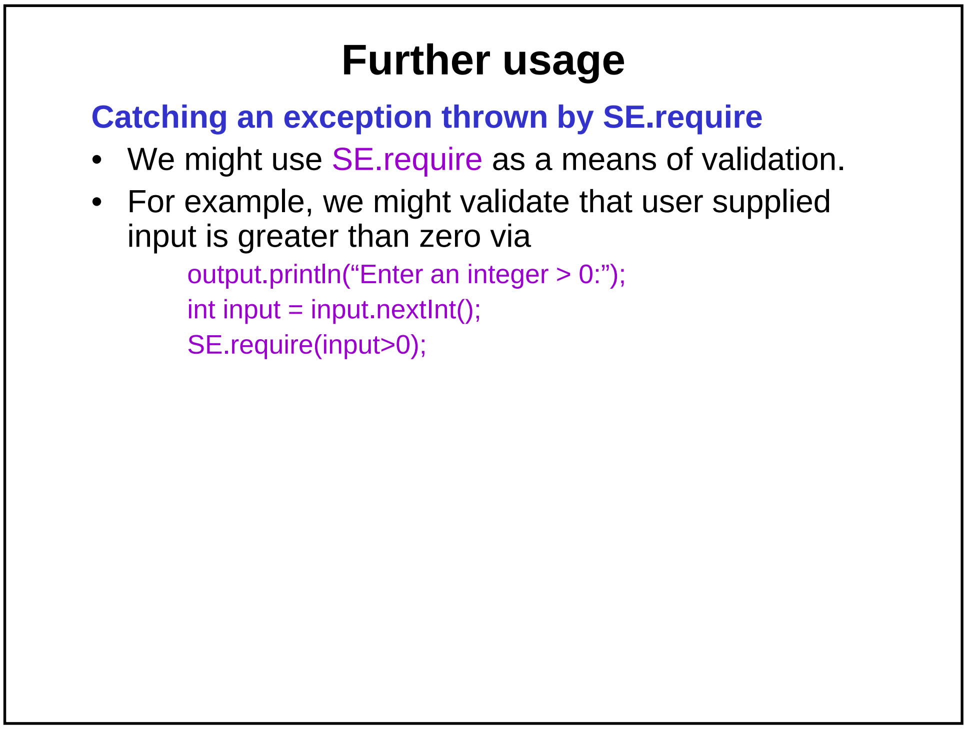
```
catch (NumberFormatException e)
{ output.println(“Non-numerical argument”);
}
```

is much more precise and informative than the previous example.



Outline

- Introduction
- Control flow: try-catch
- Exception objects
- **Further usage**



Further usage

Catching an exception thrown by SE.require

- We might use SE.require as a means of validation.
- For example, we might validate that user supplied input is greater than zero via

```
output.println("Enter an integer > 0:");
int input = input.nextInt();
SE.require(input>0);
```

Further usage

Catching an exception thrown by SE.require

- We might use SE.require as a means of validation.
- For example, we might validate that user supplied input is greater than zero via

```
output.println(“Enter an integer > 0:”);
int input = input.nextInt();
SE.require(input>0);
```

- This approach verifies that the user input is as required; however, when !(input>0) a somewhat cryptic error message appears on standard out

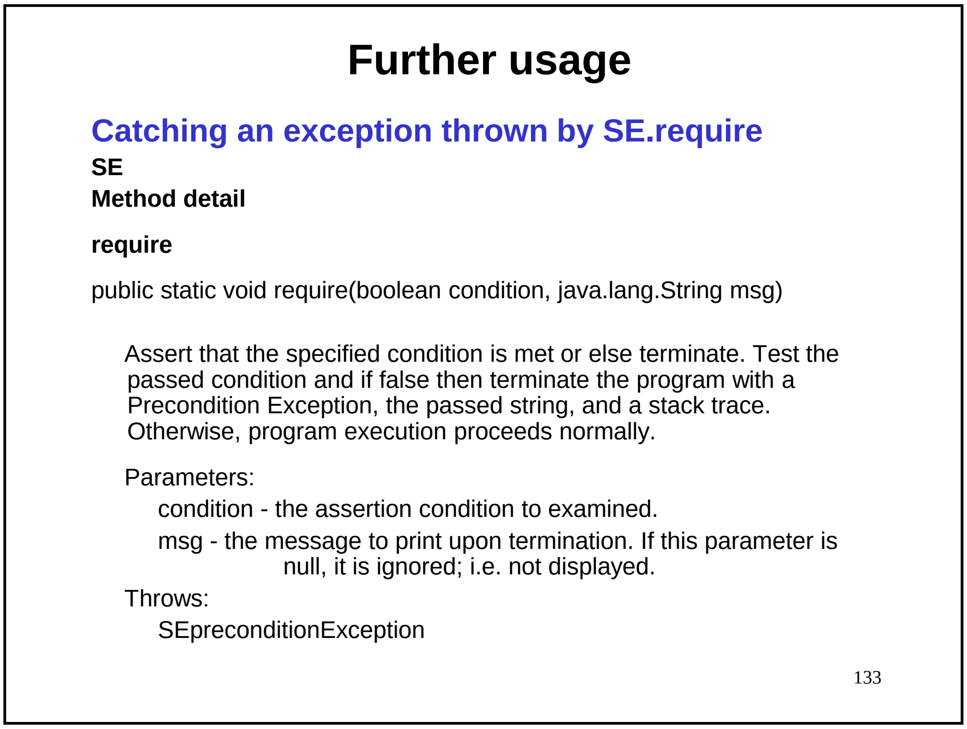
```
Exception in thread “main” type.lang.SEpreconditionException
        at type.lang.SE.require(SE.java:78)
        at type.lang.SE.require(SE.java:89)
        at Test.main(Test.java:281)
```

and the program crashes.

Further usage

Catching an exception thrown by SE.require

- We can now recognize that SE.require throws an exception…
- … and we are now in a better position to exploit this fact through try-catch.
- Let’s take a look at the method detail for require in class SE.



Further usage

Catching an exception thrown by SE.require

SE

Method detail

require

public static void require(boolean condition, java.lang.String msg)

Assert that the specified condition is met or else terminate. Test the passed condition and if false then terminate the program with a Precondition Exception, the passed string, and a stack trace. Otherwise, program execution proceeds normally.

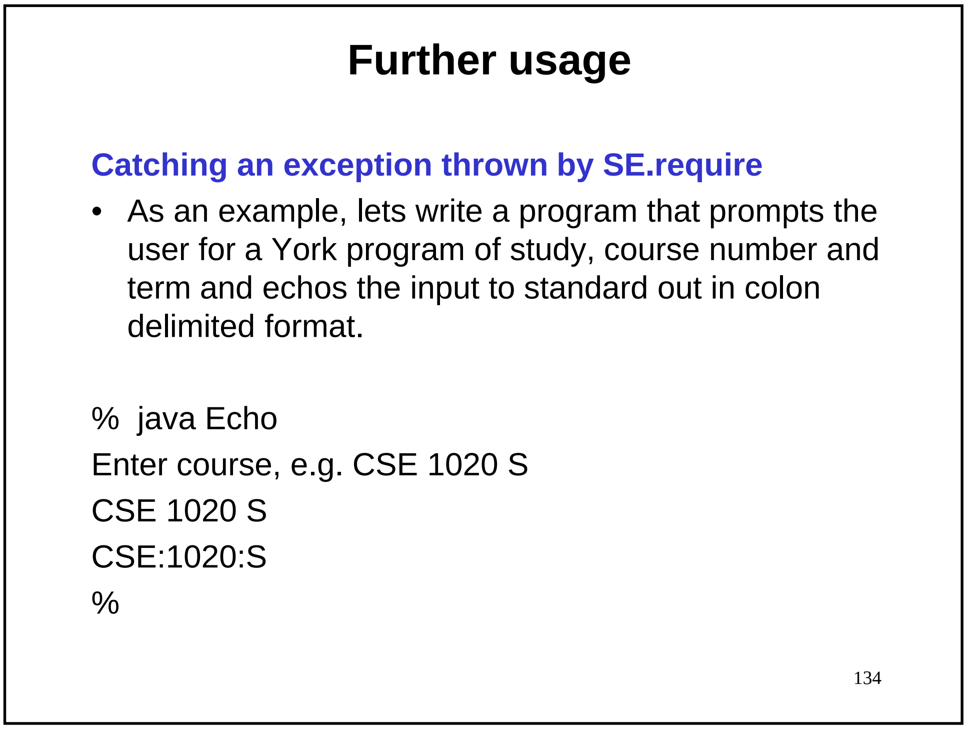
Parameters:

condition - the assertion condition to examined.

msg - the message to print upon termination. If this parameter is null, it is ignored; i.e. not displayed.

Throws:

SEpreconditionException

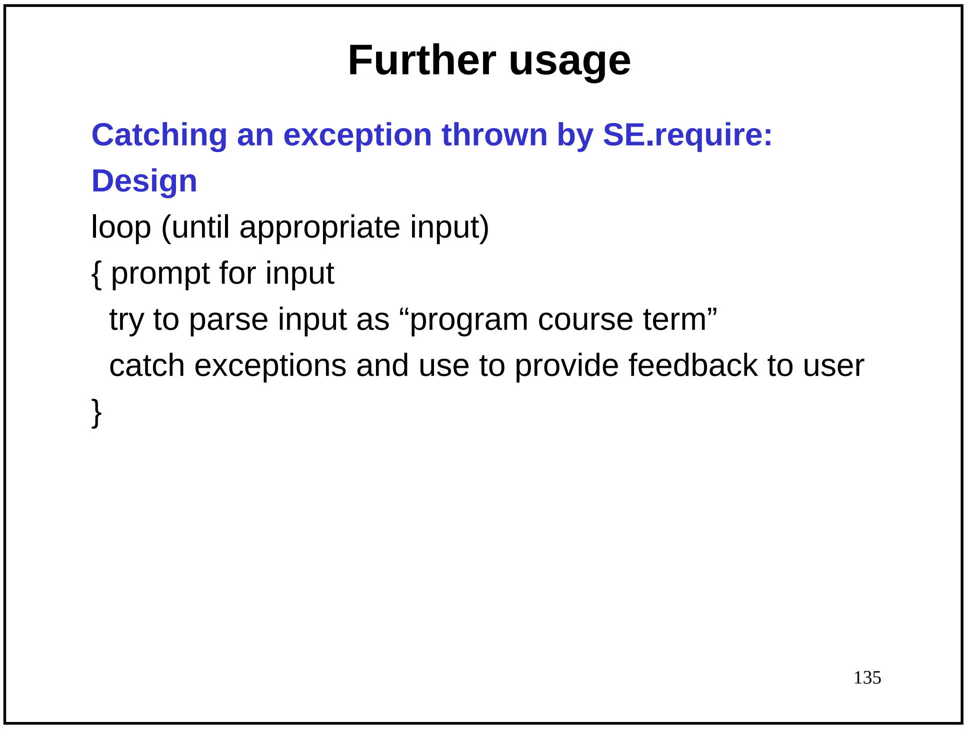


Further usage

Catching an exception thrown by SE.require

- As an example, lets write a program that prompts the user for a York program of study, course number and term and echos the input to standard out in colon delimited format.

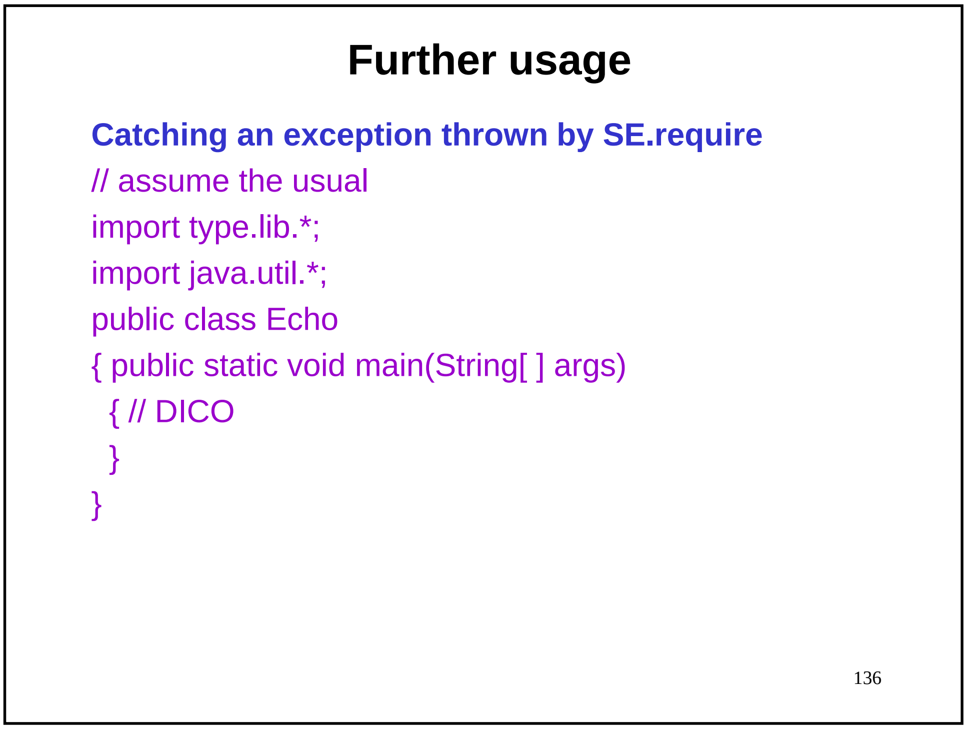
```
%  java Echo
Enter course, e.g. CSE 1020 S
CSE 1020 S
CSE:1020:S
%
```



Further usage

Catching an exception thrown by SE.require: Design

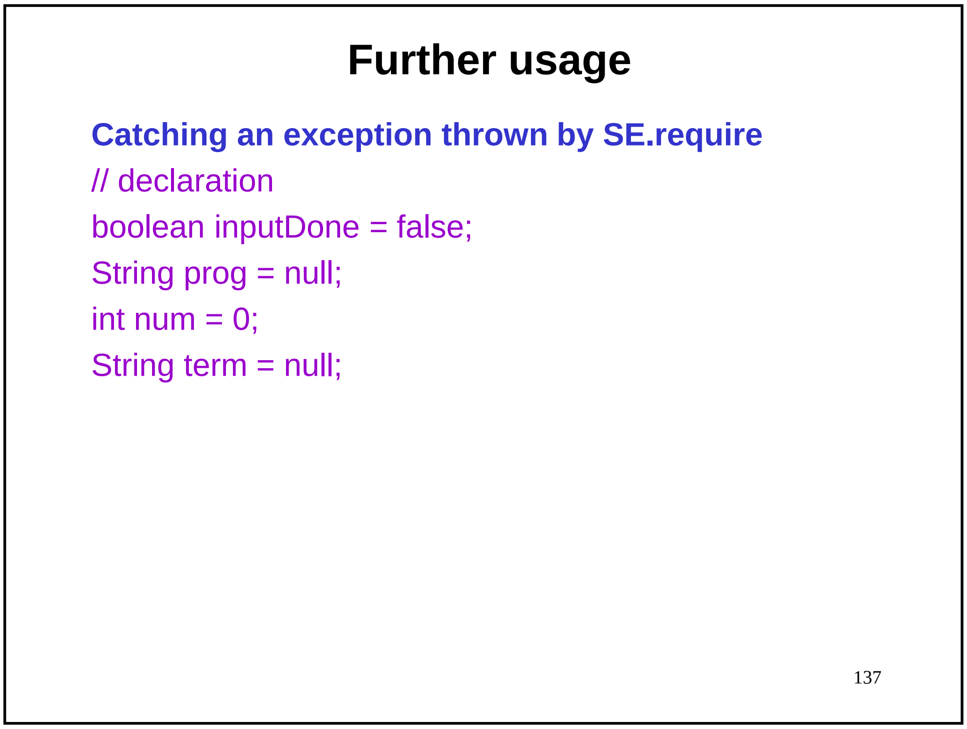
```
loop (until appropriate input)
{ prompt for input
  try to parse input as “program course term”
  catch exceptions and use to provide feedback to user
}
```



Further usage

Catching an exception thrown by SE.require

```
// assume the usual
import type.lib.*;
import java.util.*;
public class Echo
{ public static void main(String[ ] args)
  { // DICO
  }
}
```

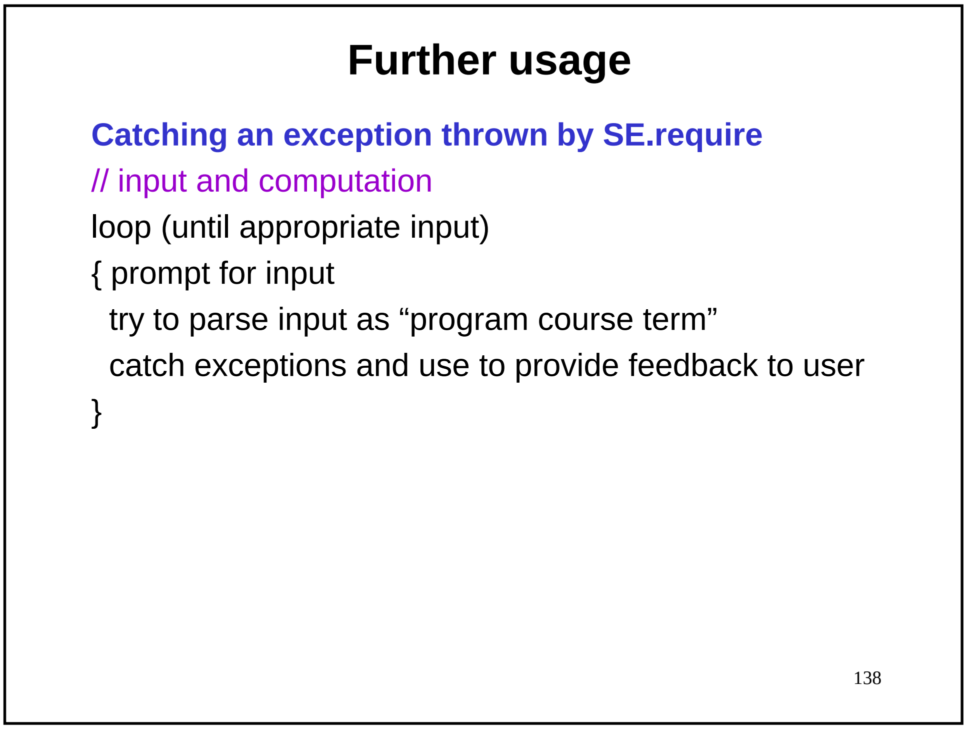


Further usage

Catching an exception thrown by SE.require

```
// declaration
boolean inputDone = false;
String prog = null;
int num = 0;
String term = null;
```

137

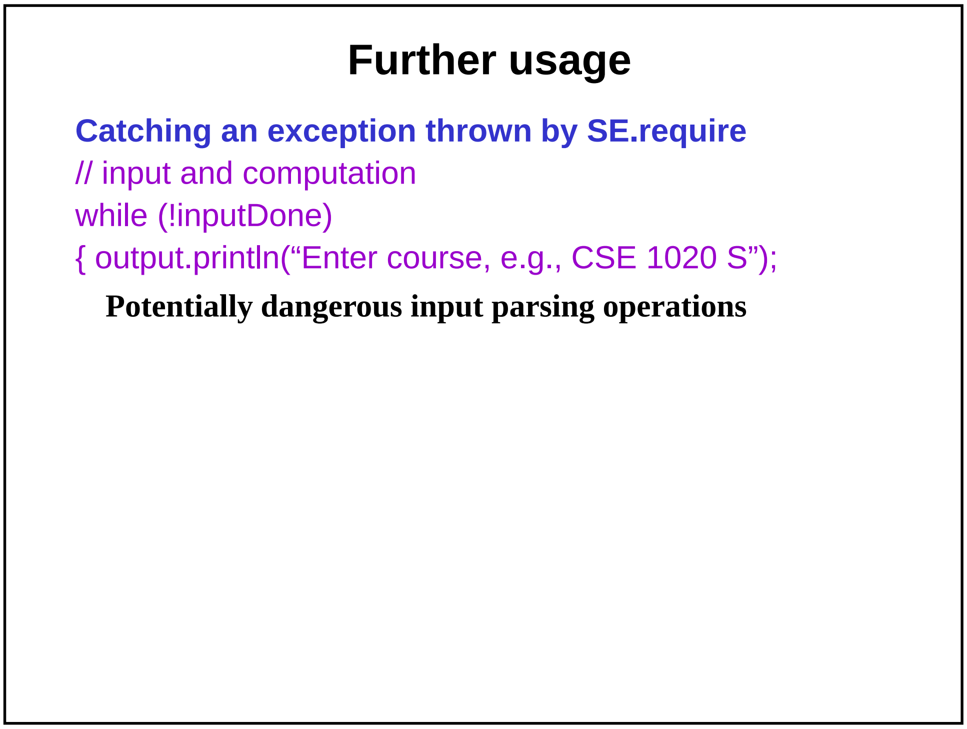


Further usage

Catching an exception thrown by SE.require

```
// input and computation
loop (until appropriate input)
{ prompt for input
  try to parse input as “program course term”
  catch exceptions and use to provide feedback to user
}
```

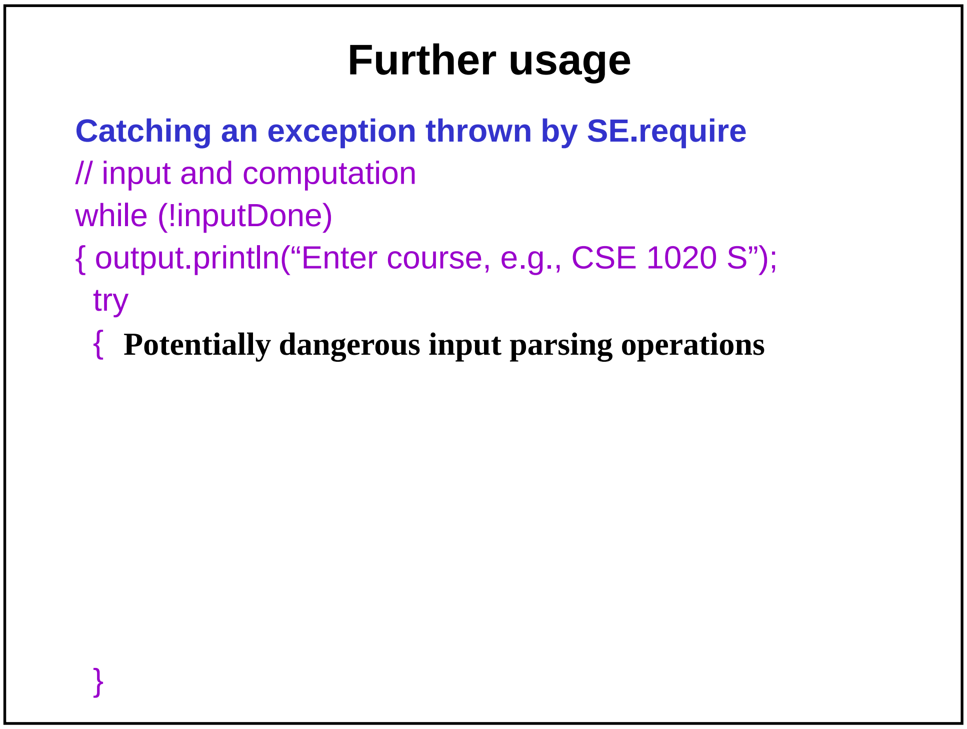
138



Further usage

Catching an exception thrown by SE.require

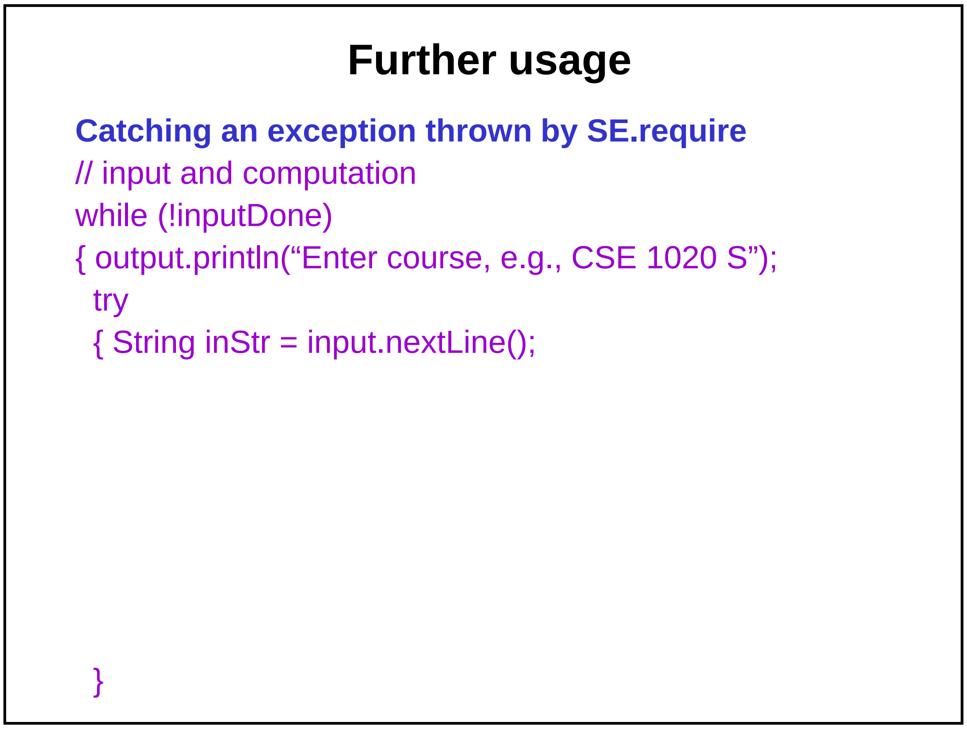
```
// input and computation
while (!inputDone)
{ output.println(“Enter course, e.g., CSE 1020 S”);
   Potentially dangerous input parsing operations
```



Further usage

Catching an exception thrown by SE.require

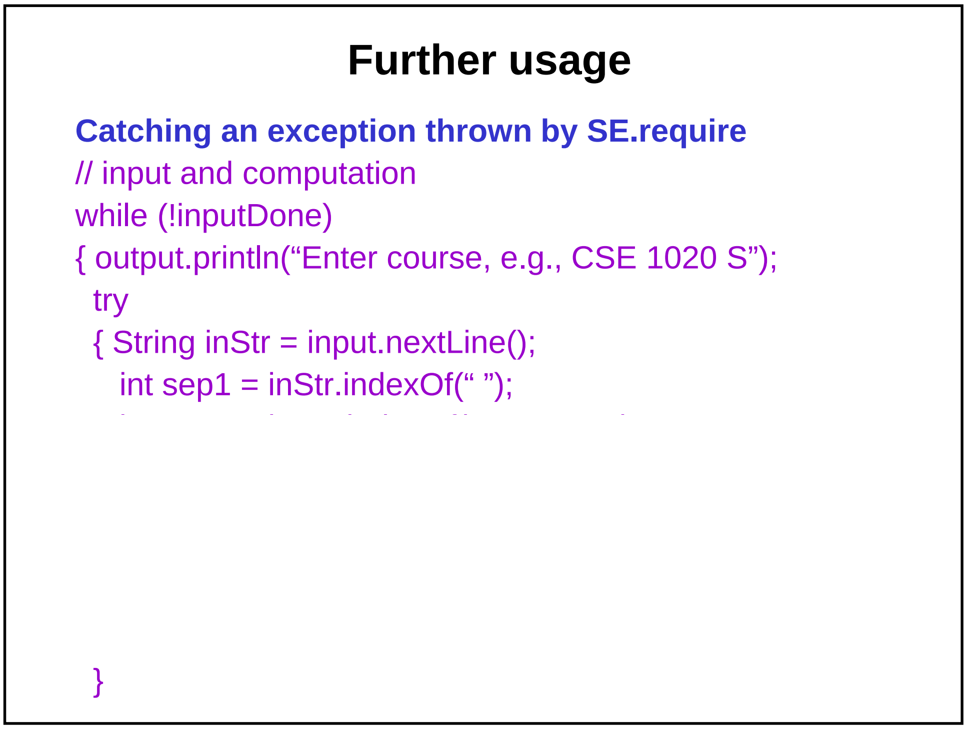
```
// input and computation
while (!inputDone)
{ output.println(“Enter course, e.g., CSE 1020 S”);
  try
  {  Potentially dangerous input parsing operations







  }
```



Further usage

Catching an exception thrown by SE.require

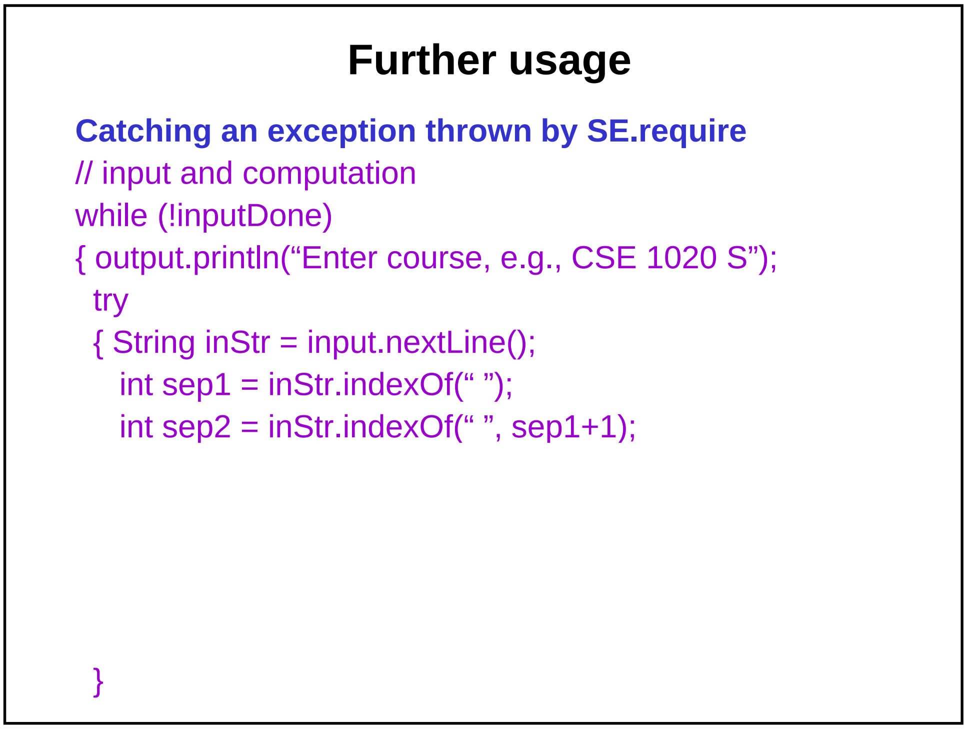
```
// input and computation
while (!inputDone)
{ output.println(“Enter course, e.g., CSE 1020 S”);
  try
  { String inStr = input.nextLine();









  }
```



Further usage

Catching an exception thrown by SE.require

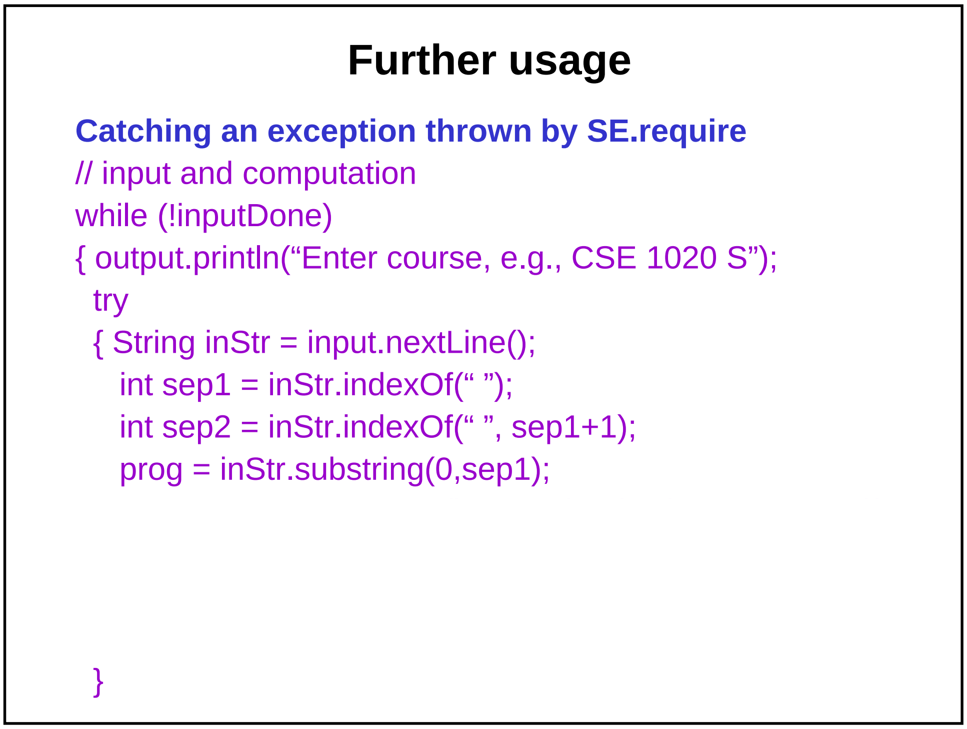
```
// input and computation
while (!inputDone)
{ output.println(“Enter course, e.g., CSE 1020 S”);
  try
  { String inStr = input.nextLine();
     int sep1 = inStr.indexOf(“ ”);








  }
```



Further usage

Catching an exception thrown by SE.require

```
// input and computation
while (!inputDone)
{ output.println(“Enter course, e.g., CSE 1020 S”);
  try
  { String inStr = input.nextLine();
     int sep1 = inStr.indexOf(“ ”);
     int sep2 = inStr.indexOf(“ ”, sep1+1);







  }
```



Further usage

Catching an exception thrown by SE.require

```
// input and computation
while (!inputDone)
{ output.println(“Enter course, e.g., CSE 1020 S”);
  try
  { String inStr = input.nextLine();
     int sep1 = inStr.indexOf(“ ”);
     int sep2 = inStr.indexOf(“ ”, sep1+1);
     prog = inStr.substring(0,sep1);





  }
```

Further usage

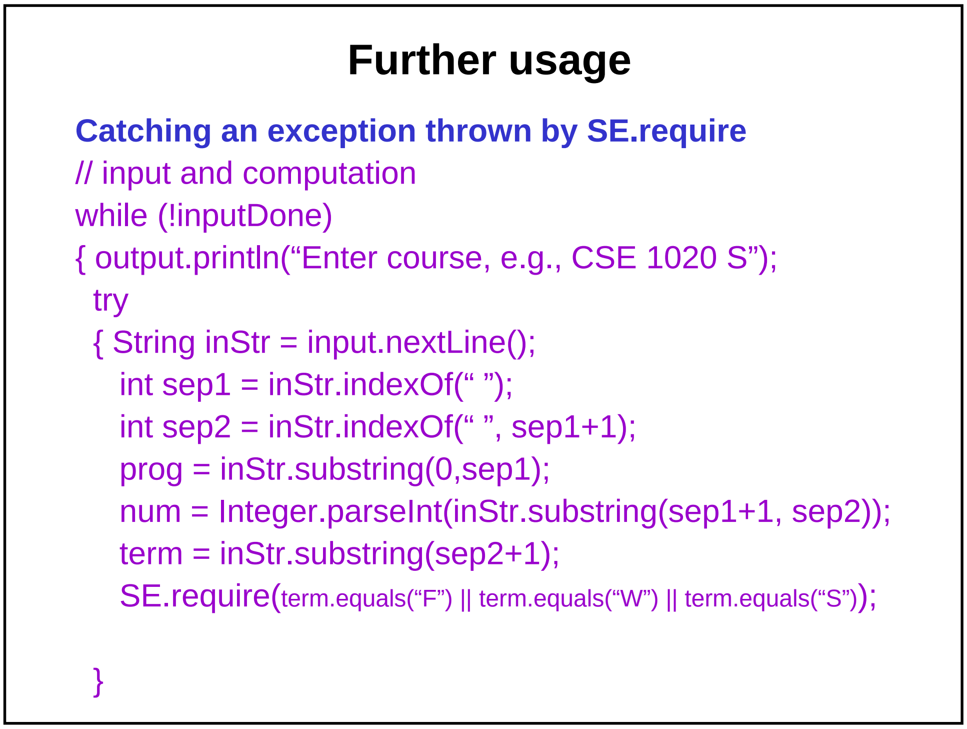
Catching an exception thrown by SE.require

```
// input and computation
while (!inputDone)
{ output.println(“Enter course, e.g., CSE 1020 S”);
  try
  { String inStr = input.nextLine();
     int sep1 = inStr.indexOf(“ ”);
     int sep2 = inStr.indexOf(“ ”, sep1+1);
     prog = inStr.substring(0,sep1);
     num = Integer.parseInt(inStr.substring(sep1+1, sep2));



  }
```

Further usage

Catching an exception thrown by SE.require

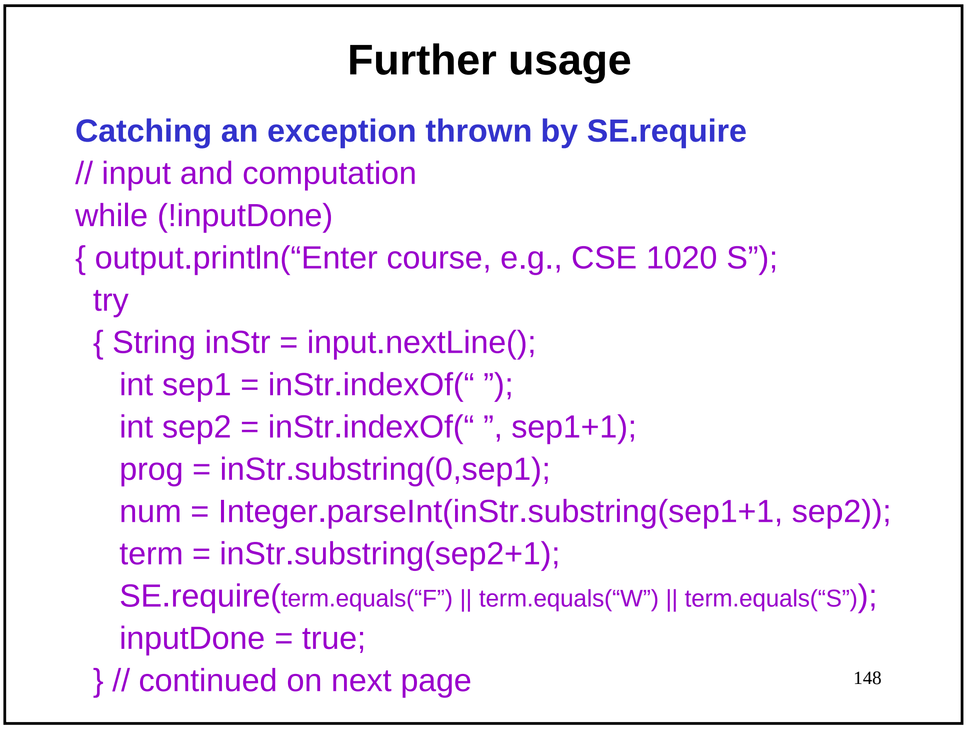
```
// input and computation
while (!inputDone)
{ output.println(“Enter course, e.g., CSE 1020 S”);
  try
  { String inStr = input.nextLine();
     int sep1 = inStr.indexOf(“ ”);
     int sep2 = inStr.indexOf(“ ”, sep1+1);
     prog = inStr.substring(0,sep1);
     num = Integer.parseInt(inStr.substring(sep1+1, sep2));
     term = inStr.substring(sep2+1);


  }
```



Further usage

Catching an exception thrown by SE.require

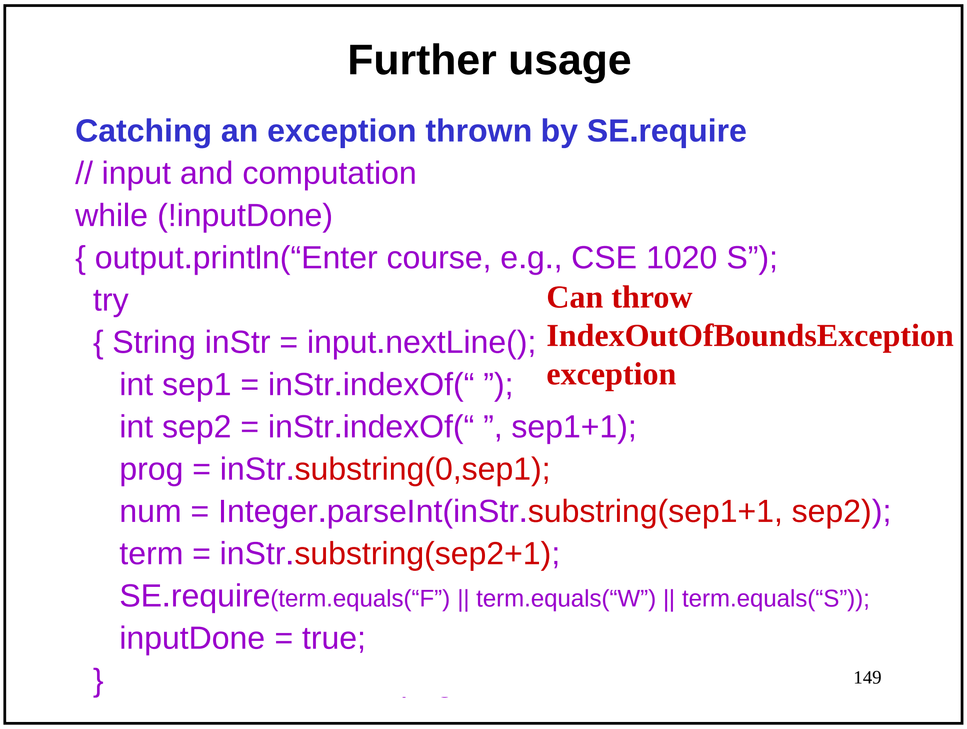
```
// input and computation
while (!inputDone)
{ output.println("Enter course, e.g., CSE 1020 S");
  try
  { String inStr = input.nextLine();
     int sep1 = inStr.indexOf(" ");
     int sep2 = inStr.indexOf(" ", sep1+1);
     prog = inStr.substring(0,sep1);
     num = Integer.parseInt(inStr.substring(sep1+1, sep2));
     term = inStr.substring(sep2+1);
     SE.require(term.equals("F") || term.equals("W") || term.equals("S"));

  }
```



Further usage

Catching an exception thrown by SE.require

```
// input and computation
while (!inputDone)
{ output.println("Enter course, e.g., CSE 1020 S");
  try
  { String inStr = input.nextLine();
     int sep1 = inStr.indexOf(" ");
     int sep2 = inStr.indexOf(" ", sep1+1);
     prog = inStr.substring(0,sep1);
     num = Integer.parseInt(inStr.substring(sep1+1, sep2));
     term = inStr.substring(sep2+1);
     SE.require(term.equals("F") || term.equals("W") || term.equals("S"));
     inputDone = true;
  } // continued on next page
```

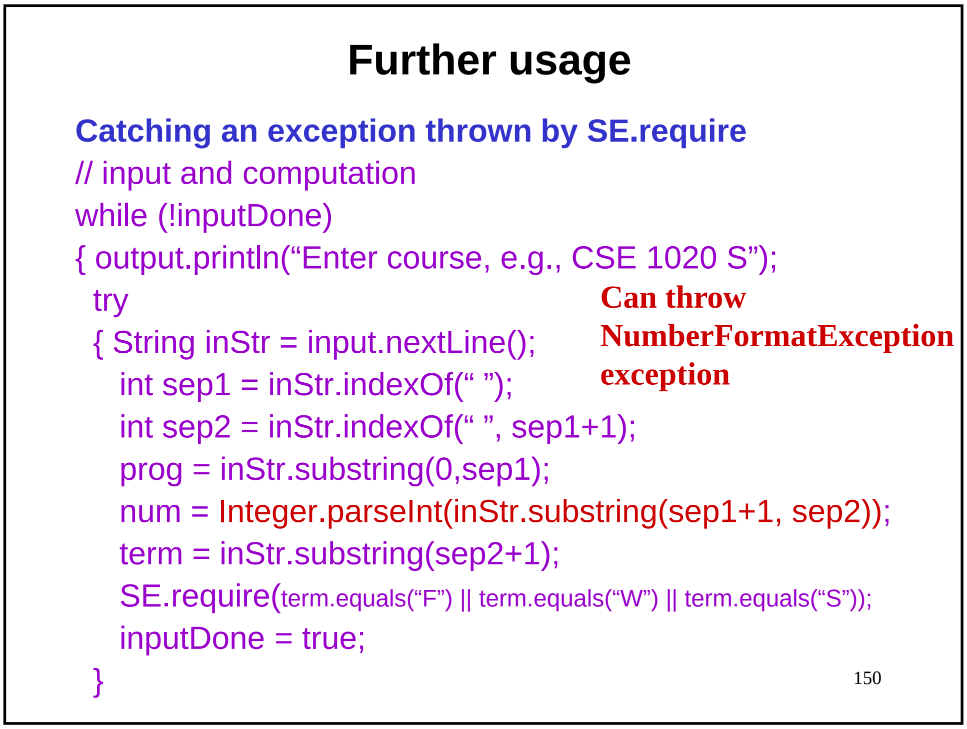


Further usage

Catching an exception thrown by SE.require

```
// input and computation
while (!inputDone)
{ output.println("Enter course, e.g., CSE 1020 S");
  try
  { String inStr = input.nextLine();
     int sep1 = inStr.indexOf(" ");
     int sep2 = inStr.indexOf(" ", sep1+1);
     prog = inStr.substring(0,sep1);
     num = Integer.parseInt(inStr.substring(sep1+1, sep2));
     term = inStr.substring(sep2+1);
     SE.require(term.equals("F") || term.equals("W") || term.equals("S"));
     inputDone = true;
  }
```

Can throw IndexOutOfBoundsException exception

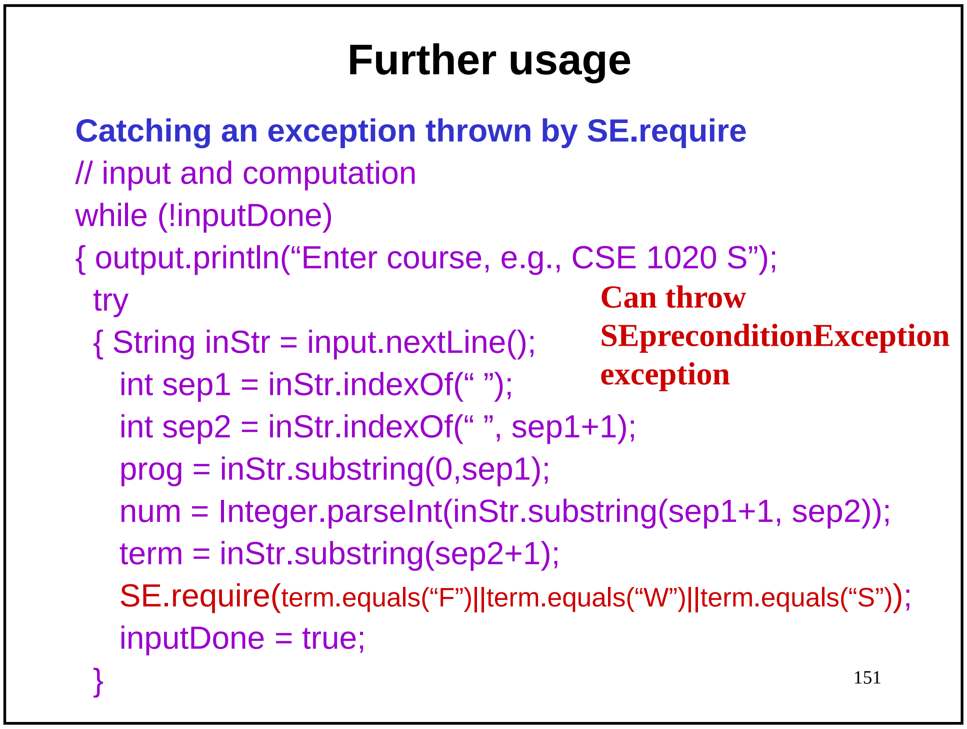


Further usage

Catching an exception thrown by SE.require

```
// input and computation
while (!inputDone)
{ output.println("Enter course, e.g., CSE 1020 S");
  try
  { String inStr = input.nextLine();
     int sep1 = inStr.indexOf(" ");
     int sep2 = inStr.indexOf(" ", sep1+1);
     prog = inStr.substring(0,sep1);
     num = Integer.parseInt(inStr.substring(sep1+1, sep2));
     term = inStr.substring(sep2+1);
     SE.require(term.equals("F") || term.equals("W") || term.equals("S"));
     inputDone = true;
  }
```

Can throw NumberFormatException exception



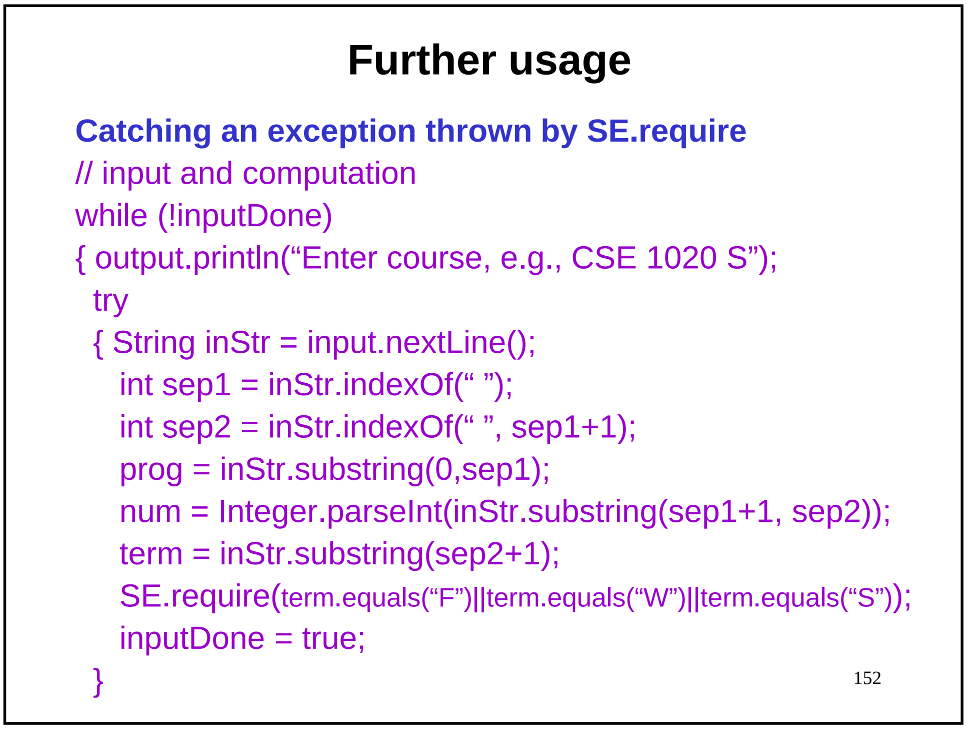
Further usage

Catching an exception thrown by SE.require

```
// input and computation
while (!inputDone)
{ output.println("Enter course, e.g., CSE 1020 S");
  try
  { String inStr = input.nextLine();
    int sep1 = inStr.indexOf(" ");
    int sep2 = inStr.indexOf(" ", sep1+1);
    prog = inStr.substring(0,sep1);
    num = Integer.parseInt(inStr.substring(sep1+1, sep2));
    term = inStr.substring(sep2+1);
    SE.require(term.equals("F")||term.equals("W")||term.equals("S"));
    inputDone = true;
  }
```

Can throw SEpreconditionException exception

151



Further usage

Catching an exception thrown by SE.require

```
// input and computation
while (!inputDone)
{ output.println("Enter course, e.g., CSE 1020 S");
  try
  { String inStr = input.nextLine();
    int sep1 = inStr.indexOf(" ");
    int sep2 = inStr.indexOf(" ", sep1+1);
    prog = inStr.substring(0,sep1);
    num = Integer.parseInt(inStr.substring(sep1+1, sep2));
    term = inStr.substring(sep2+1);
    SE.require(term.equals("F")||term.equals("W")||term.equals("S"));
    inputDone = true;
  }
```

152

Further usage

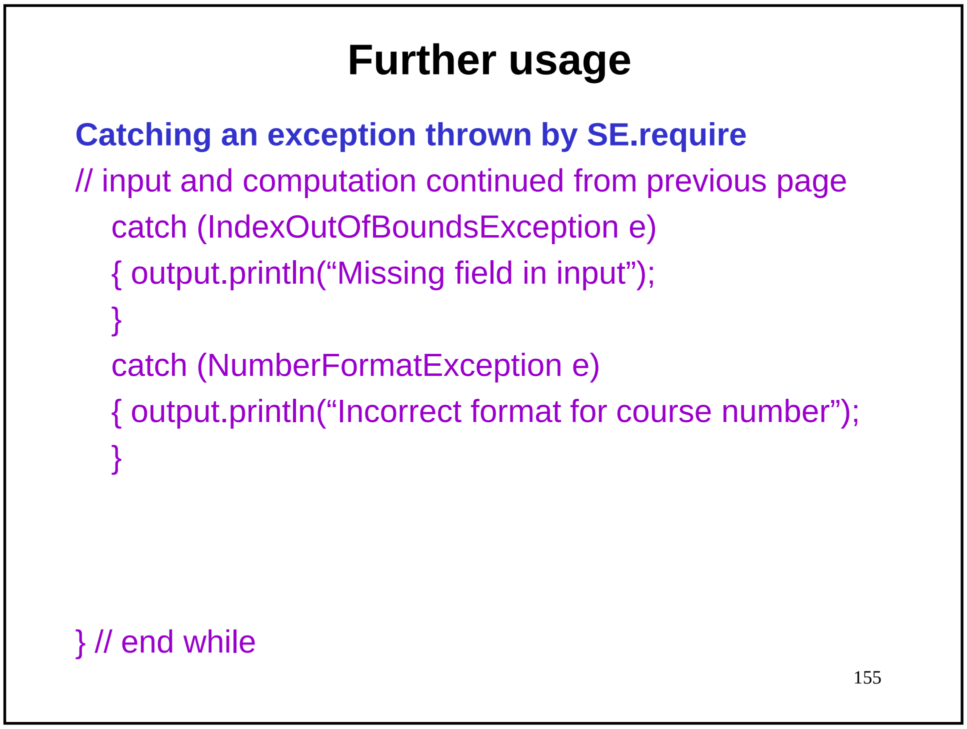
Catching an exception thrown by SE.require

```
// input and computation continued from previous page









} // end while
```

Further usage

Catching an exception thrown by SE.require

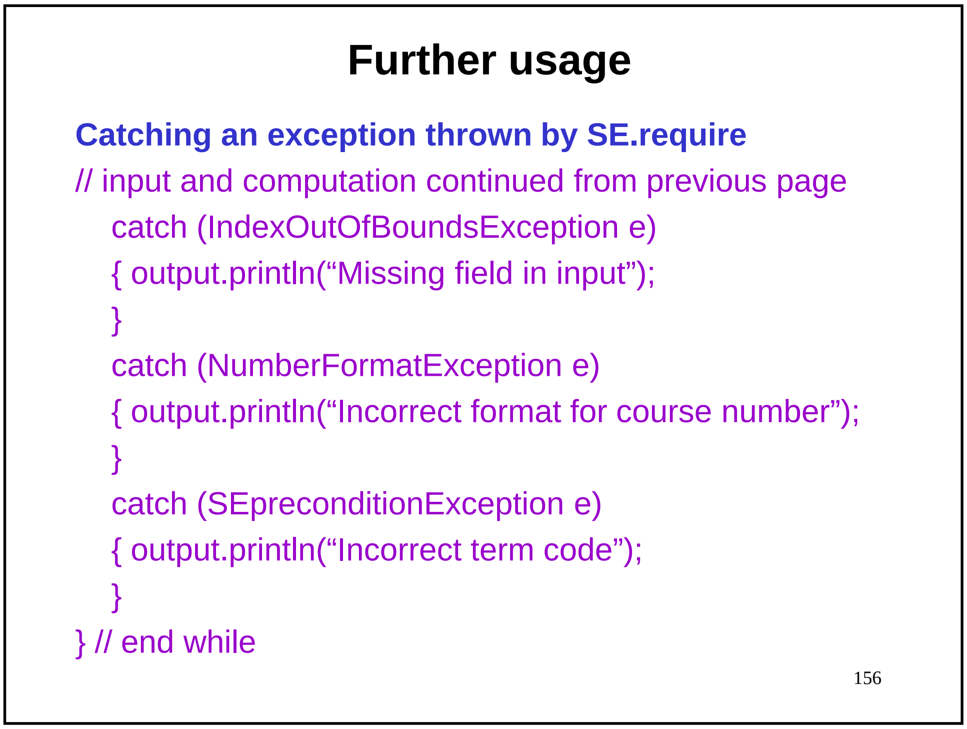
```
// input and computation continued from previous page
   catch (IndexOutOfBoundsException e)
   { output.println(“Missing field in input”);
   }






} // end while
```



Further usage

Catching an exception thrown by SE.require

```
// input and computation continued from previous page
    catch (IndexOutOfBoundsException e)
    { output.println(“Missing field in input”);
    }
    catch (NumberFormatException e)
    { output.println(“Incorrect format for course number”);
    }




} // end while
```



Further usage

Catching an exception thrown by SE.require

```
// input and computation continued from previous page
    catch (IndexOutOfBoundsException e)
    { output.println(“Missing field in input”);
    }
    catch (NumberFormatException e)
    { output.println(“Incorrect format for course number”);
    }
    catch (SEpreconditionException e)
    { output.println(“Incorrect term code”);
    }
} // end while
```

Further usage

Catching an exception thrown by SE.require

```
// output
output.println(prog + “:” + num + “:” + term);
```

Further usage

Catching an exception thrown by SE.require: Test

```
% java Echo
```

Further usage

Catching an exception thrown by SE.require: Test

```
% java Echo
Enter course, e.g., CSE 1020 S
```

Further usage

Catching an exception thrown by SE.require: Test

```
% java Echo
Enter course, e.g., CSE 1020 S
CSE 1020
```

Further usage

Catching an exception thrown by SE.require: Test

```
% java Echo
Enter course, e.g., CSE 1020 S
CSE 1020
Missing field in input
Enter course, e.g., CSE 1020 S
```

Further usage

Catching an exception thrown by SE.require: Test

```
% java Echo
Enter course, e.g., CSE 1020 S
CSE 1020
Missing field in input
Enter course, e.g., CSE 1020 S
CSE 1020S
```

Further usage

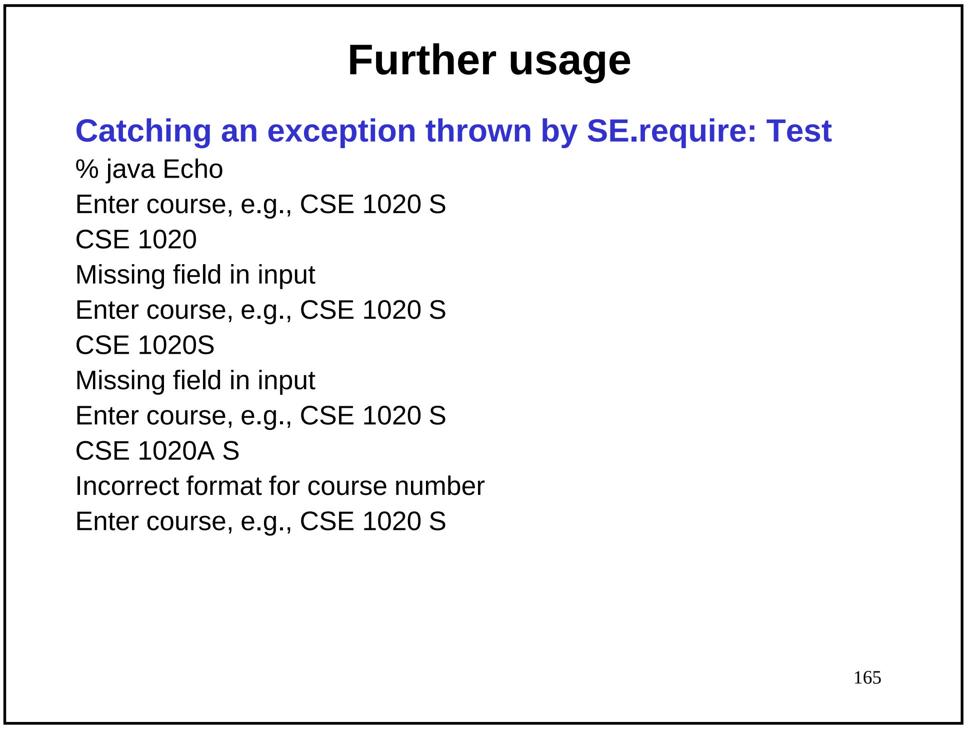
Catching an exception thrown by SE.require: Test

```
% java Echo
Enter course, e.g., CSE 1020 S
CSE 1020
Missing field in input
Enter course, e.g., CSE 1020 S
CSE 1020S
Missing field in input
Enter course, e.g., CSE 1020 S
```

Further usage

Catching an exception thrown by SE.require: Test

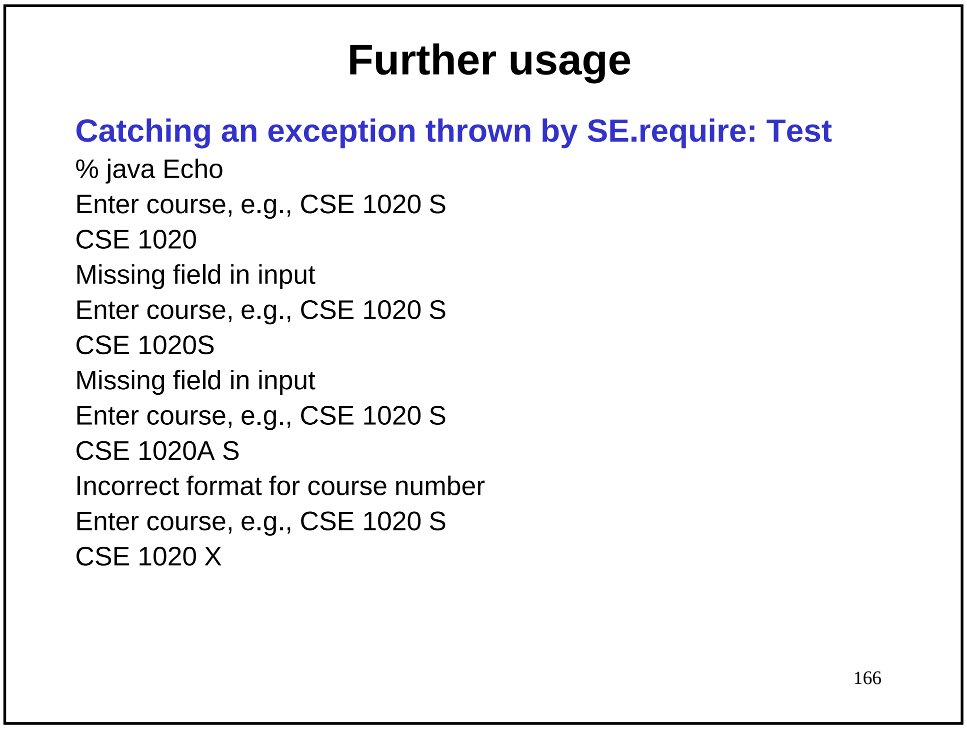
```
% java Echo
Enter course, e.g., CSE 1020 S
CSE 1020
Missing field in input
Enter course, e.g., CSE 1020 S
CSE 1020S
Missing field in input
Enter course, e.g., CSE 1020 S
CSE 1020A S
```



Further usage

Catching an exception thrown by SE.require: Test

```
% java Echo
Enter course, e.g., CSE 1020 S
CSE 1020
Missing field in input
Enter course, e.g., CSE 1020 S
CSE 1020S
Missing field in input
Enter course, e.g., CSE 1020 S
CSE 1020A S
Incorrect format for course number
Enter course, e.g., CSE 1020 S
```



Further usage

Catching an exception thrown by SE.require: Test

```
% java Echo
Enter course, e.g., CSE 1020 S
CSE 1020
Missing field in input
Enter course, e.g., CSE 1020 S
CSE 1020S
Missing field in input
Enter course, e.g., CSE 1020 S
CSE 1020A S
Incorrect format for course number
Enter course, e.g., CSE 1020 S
CSE 1020 X
```

Further usage

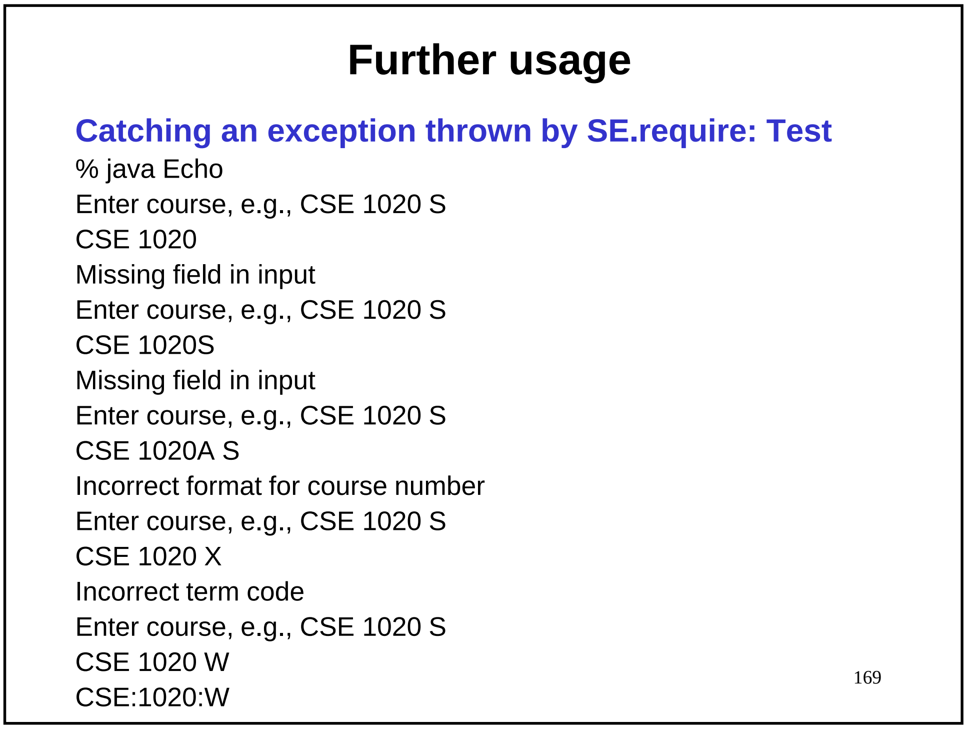
Catching an exception thrown by SE.require: Test

```
% java Echo
Enter course, e.g., CSE 1020 S
CSE 1020
Missing field in input
Enter course, e.g., CSE 1020 S
CSE 1020S
Missing field in input
Enter course, e.g., CSE 1020 S
CSE 1020A S
Incorrect format for course number
Enter course, e.g., CSE 1020 S
CSE 1020 X
Incorrect term code
Enter course, e.g., CSE 1020 S
```

Further usage

Catching an exception thrown by SE.require: Test

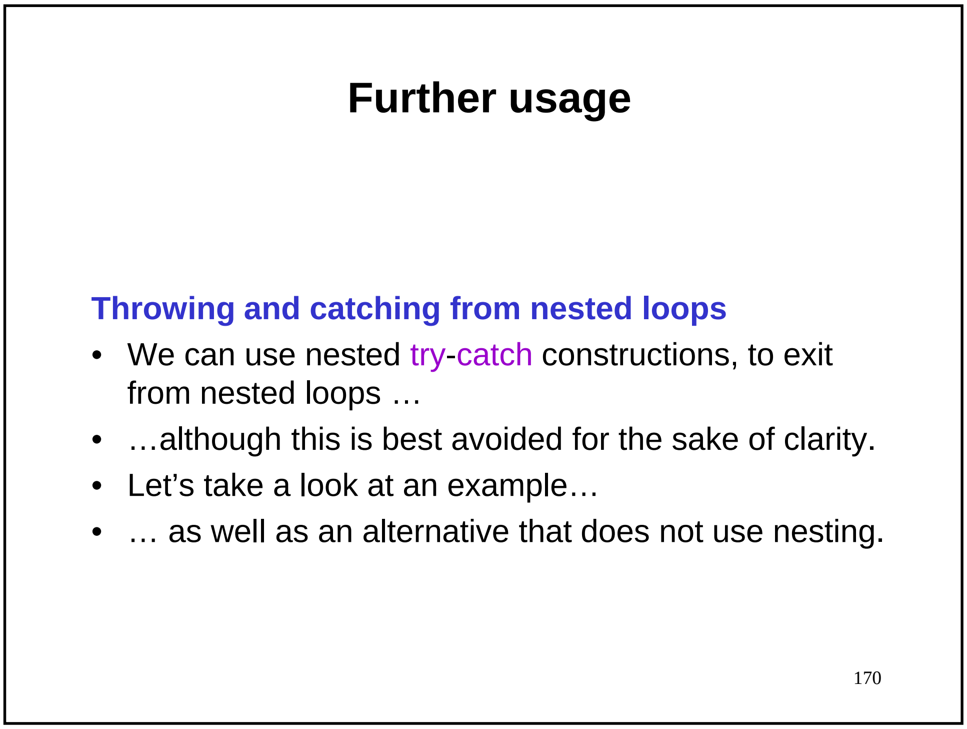
```
% java Echo
Enter course, e.g., CSE 1020 S
CSE 1020
Missing field in input
Enter course, e.g., CSE 1020 S
CSE 1020S
Missing field in input
Enter course, e.g., CSE 1020 S
CSE 1020A S
Incorrect format for course number
Enter course, e.g., CSE 1020 S
CSE 1020 X
Incorrect term code
Enter course, e.g., CSE 1020 S
CSE 1020 W
```



Further usage

Catching an exception thrown by SE.require: Test

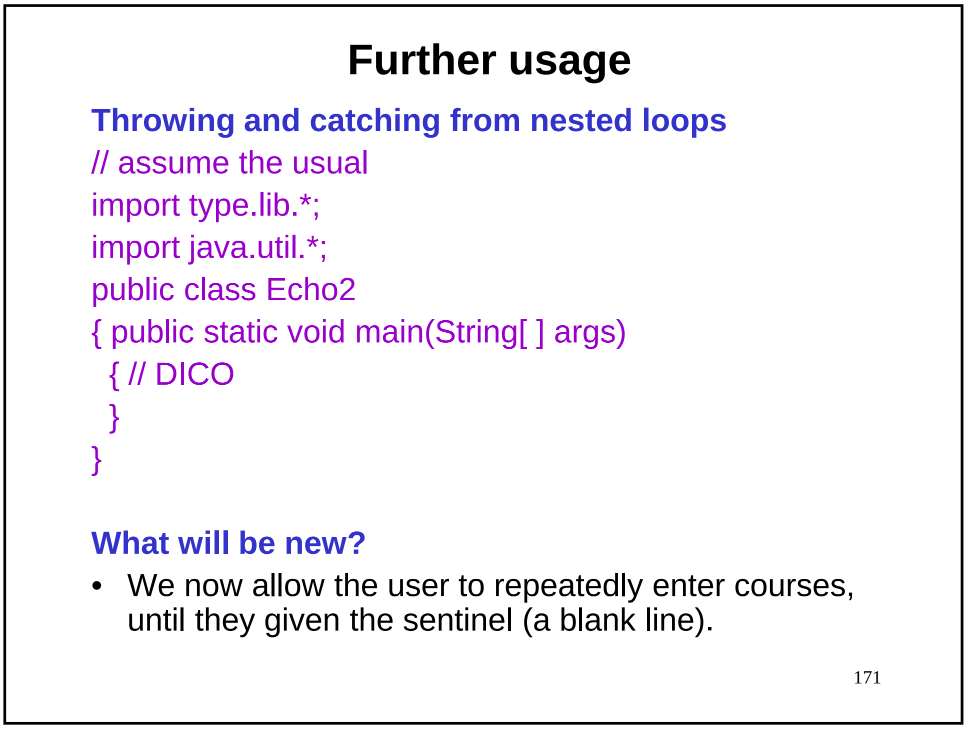
```
% java Echo
Enter course, e.g., CSE 1020 S
CSE 1020
Missing field in input
Enter course, e.g., CSE 1020 S
CSE 1020S
Missing field in input
Enter course, e.g., CSE 1020 S
CSE 1020A S
Incorrect format for course number
Enter course, e.g., CSE 1020 S
CSE 1020 X
Incorrect term code
Enter course, e.g., CSE 1020 S
CSE 1020 W
CSE:1020:W
```



Further usage

Throwing and catching from nested loops

- We can use nested try-catch constructions, to exit from nested loops …
- …although this is best avoided for the sake of clarity.
- Let’s take a look at an example…
- … as well as an alternative that does not use nesting.



Further usage

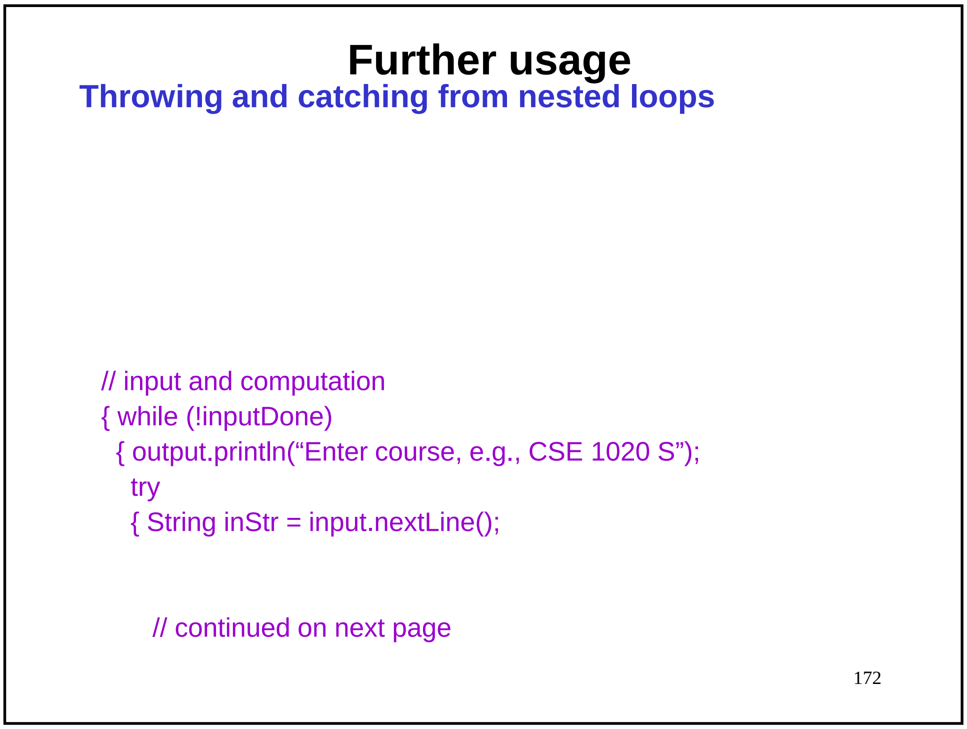
Throwing and catching from nested loops

```
// assume the usual
import type.lib.*;
import java.util.*;
public class Echo2
{ public static void main(String[ ] args)
  { // DICO
  }
}
```

What will be new?

- We now allow the user to repeatedly enter courses, until they given the sentinel (a blank line).

171



Further usage

Throwing and catching from nested loops

```
// input and computation
{ while (!inputDone)
  { output.println(“Enter course, e.g., CSE 1020 S”);
    try
    { String inStr = input.nextLine();


       // continued on next page
```

172

Further usage

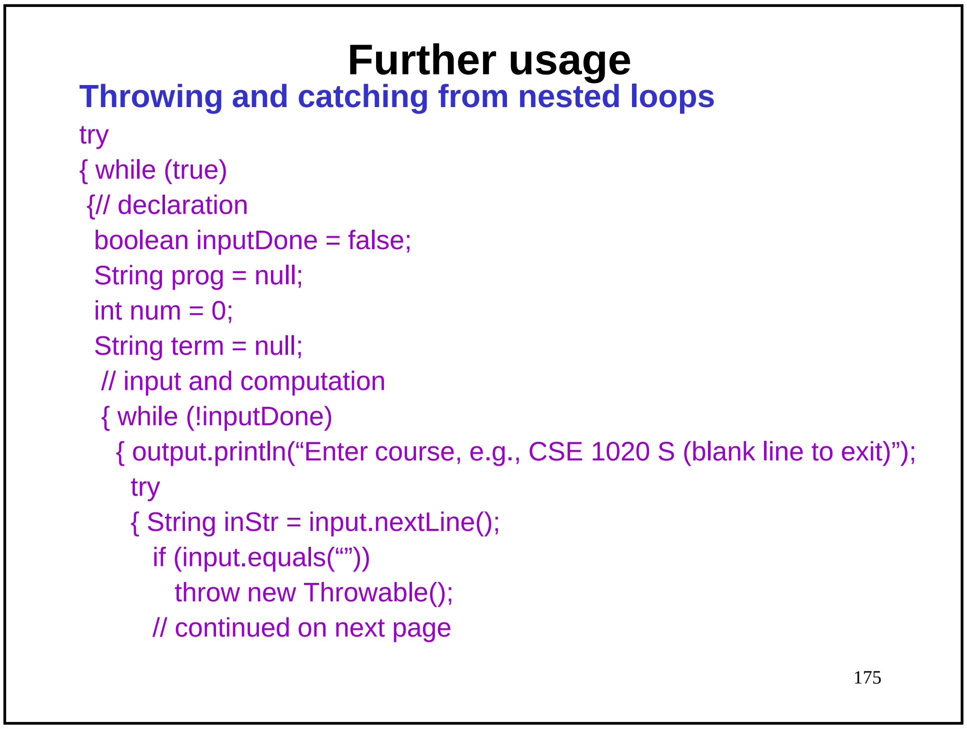
Throwing and catching from nested loops

```
  while (true)
 {// declaration
  boolean inputDone = false;
  String prog = null;
  int num = 0;
  String term = null;
   // input and computation
   { while (!inputDone)
     { output.println(“Enter course, e.g., CSE 1020 S (blank line to exit)”);
       try
       { String input = input.nextLine();



          // continued on next page
```

Further usage

Throwing and catching from nested loops

```
try
{ while (true)
 {// declaration
  boolean inputDone = false;
  String prog = null;
  int num = 0;
  String term = null;
   // input and computation
   { while (!inputDone)
     { output.println(“Enter course, e.g., CSE 1020 S (blank line to exit)”);
       try
       { String inStr = input.nextLine();
          if (input.equals(“”))
             throw new Throwable();
          // continued on next page
```

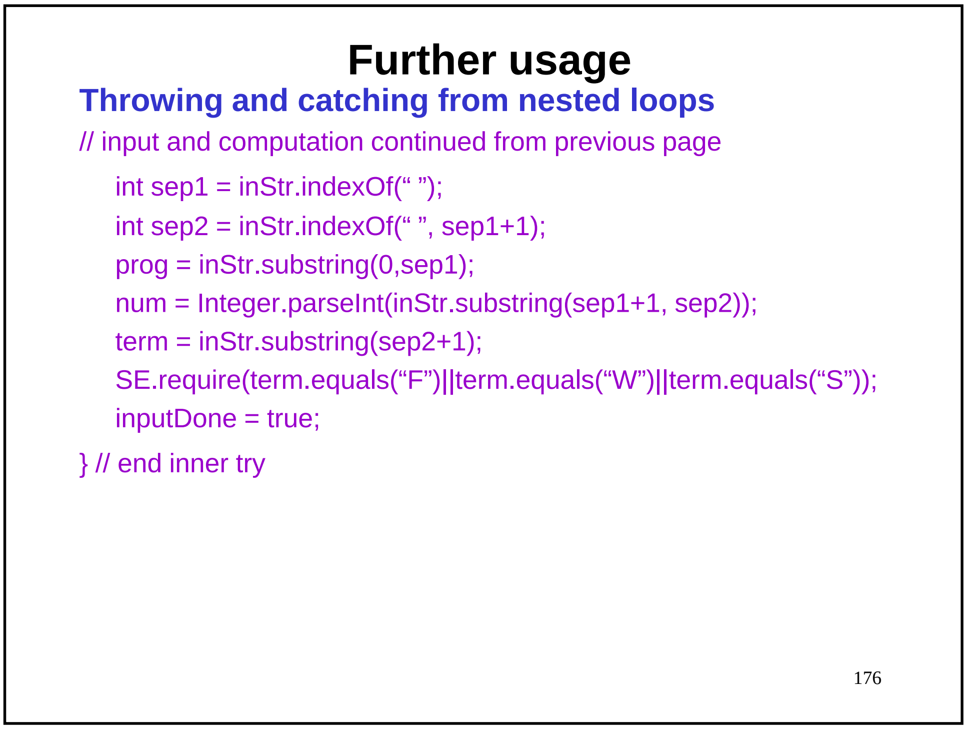


Further usage

Throwing and catching from nested loops

```
try
{ while (true)
 {// declaration
  boolean inputDone = false;
  String prog = null;
  int num = 0;
  String term = null;
   // input and computation
   { while (!inputDone)
     { output.println(“Enter course, e.g., CSE 1020 S (blank line to exit)”);
       try
       { String inStr = input.nextLine();
          if (input.equals(“”))
             throw new Throwable();
          // continued on next page
```

175

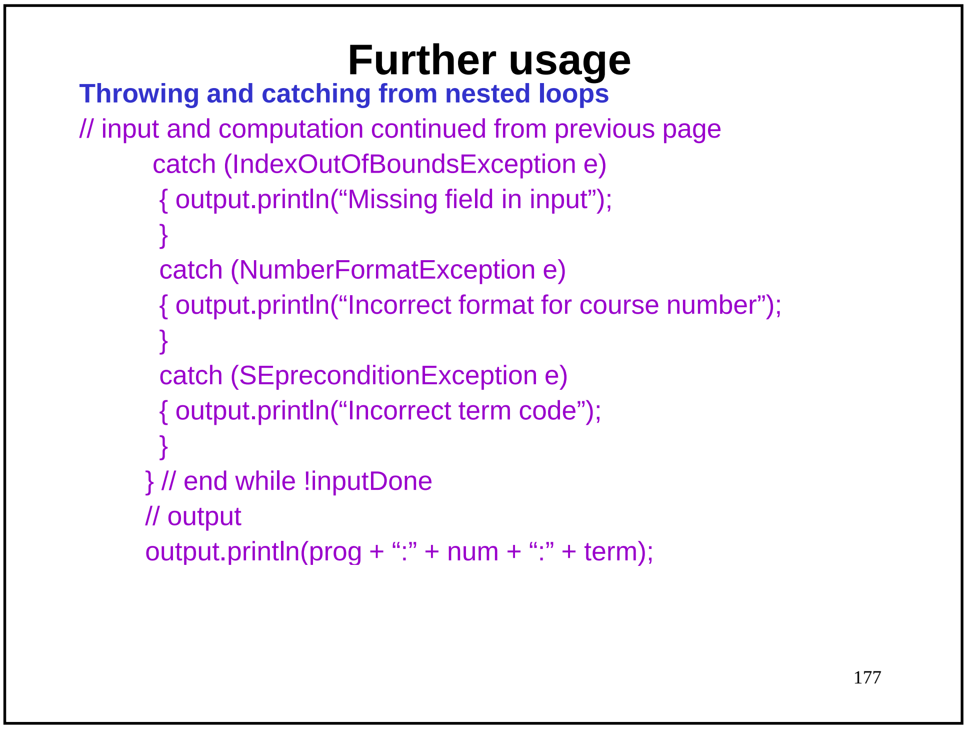


Further usage

Throwing and catching from nested loops

```
// input and computation continued from previous page
     int sep1 = inStr.indexOf(“ ”);
     int sep2 = inStr.indexOf(“ ”, sep1+1);
     prog = inStr.substring(0,sep1);
     num = Integer.parseInt(inStr.substring(sep1+1, sep2));
     term = inStr.substring(sep2+1);
     SE.require(term.equals(“F”)||term.equals(“W”)||term.equals(“S”));
     inputDone = true;
} // end inner try
```

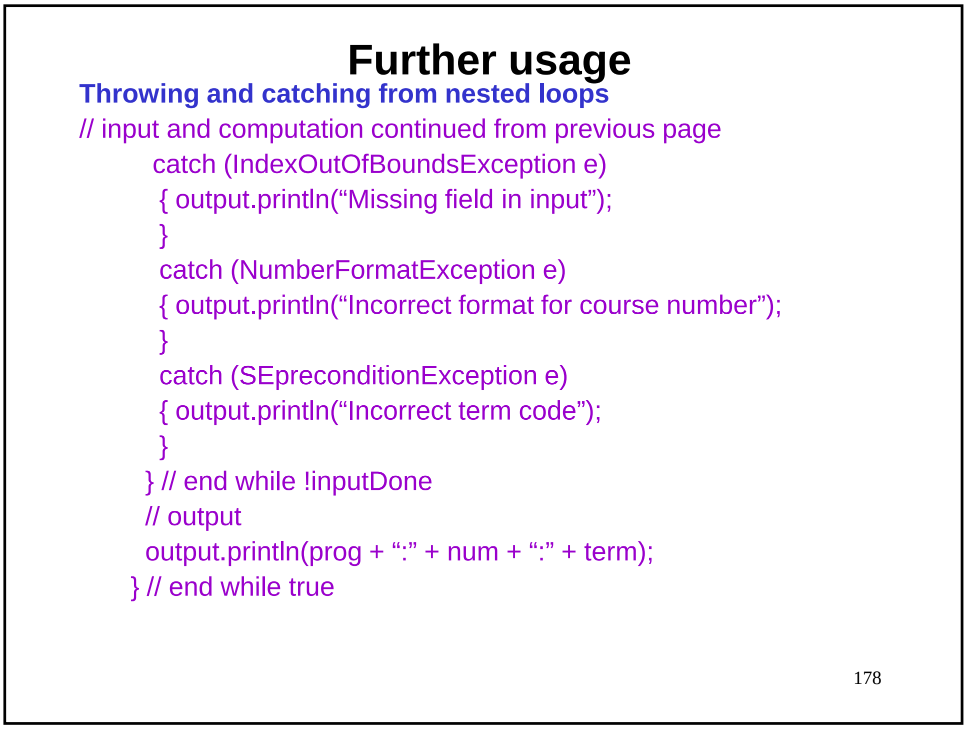
176



Further usage

Throwing and catching from nested loops

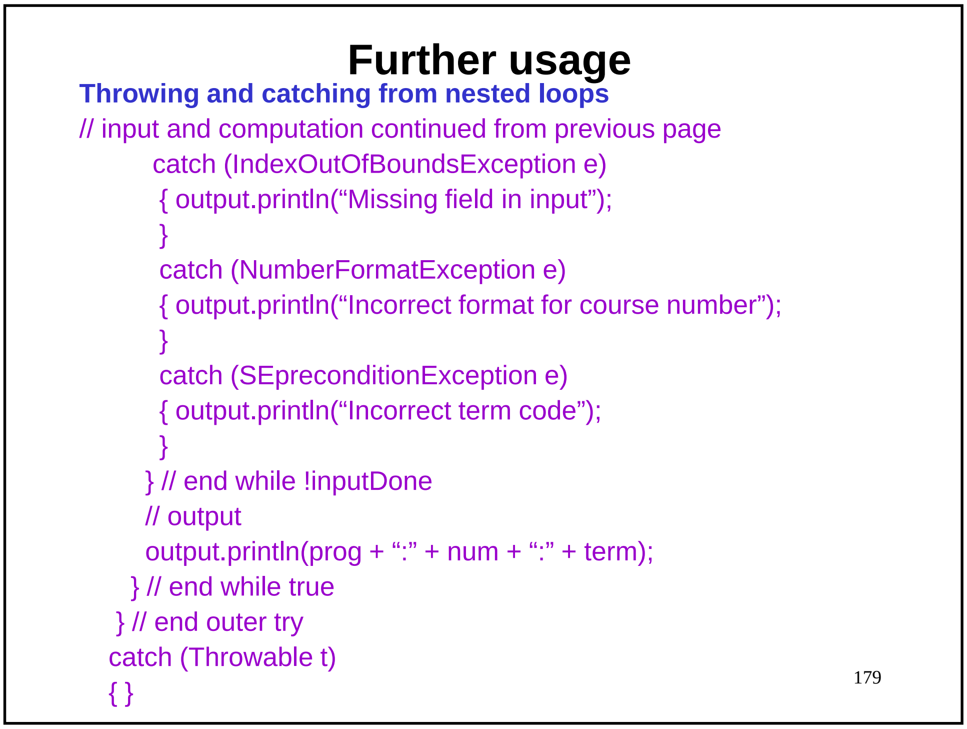
```
// input and computation continued from previous page
          catch (IndexOutOfBoundsException e)
           { output.println(“Missing field in input”);
           }
           catch (NumberFormatException e)
           { output.println(“Incorrect format for course number”);
           }
           catch (SEpreconditionException e)
           { output.println(“Incorrect term code”);
           }
         } // end while !inputDone
         // output
         output.println(prog + “:” + num + “:” + term);
```



Further usage

Throwing and catching from nested loops

```
// input and computation continued from previous page
          catch (IndexOutOfBoundsException e)
           { output.println(“Missing field in input”);
           }
           catch (NumberFormatException e)
           { output.println(“Incorrect format for course number”);
           }
           catch (SEpreconditionException e)
           { output.println(“Incorrect term code”);
           }
         } // end while !inputDone
         // output
         output.println(prog + “:” + num + “:” + term);
       } // end while true
```

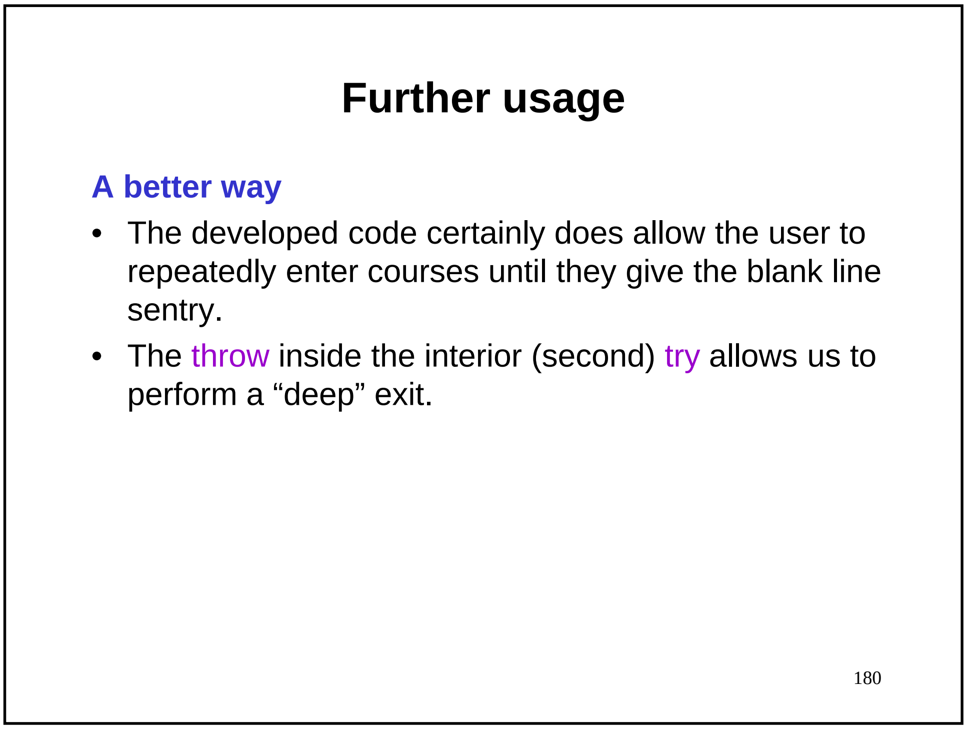


Further usage

Throwing and catching from nested loops

```
// input and computation continued from previous page
          catch (IndexOutOfBoundsException e)
           { output.println(“Missing field in input”);
           }
           catch (NumberFormatException e)
           { output.println(“Incorrect format for course number”);
           }
           catch (SEpreconditionException e)
           { output.println(“Incorrect term code”);
           }
         } // end while !inputDone
         // output
         output.println(prog + “:” + num + “:” + term);
       } // end while true
     } // end outer try
    catch (Throwable t)
    { }
```

179



Further usage

A better way

- The developed code certainly does allow the user to repeatedly enter courses until they give the blank line sentry.
- The throw inside the interior (second) try allows us to perform a “deep” exit.

180

Further usage

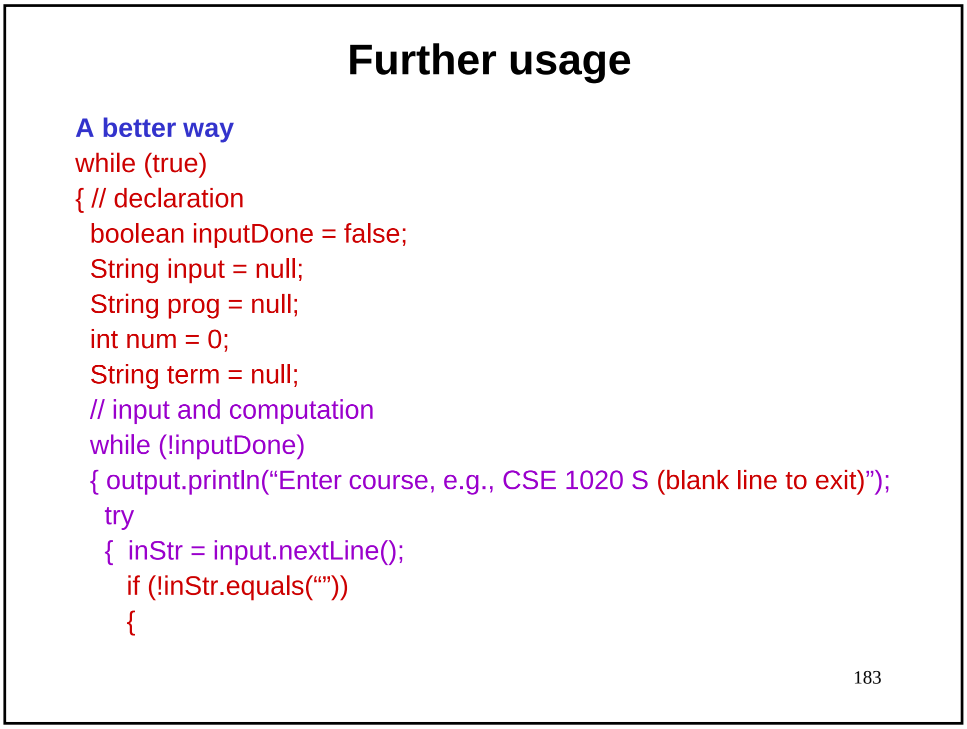
A better way

- The developed code certainly does allow the user to repeatedly enter courses until they give the blank line sentry.
- The throw inside the interior (second) try allows us to perform a “deep” exit.
- This approach, however, is not considered good style as it makes the code difficult to read and understand.
- We can do better by moving the check for the sentry outward in conjunction with break.
- Let’s take a look.

Further usage

A better way: Return to the original version of Echo

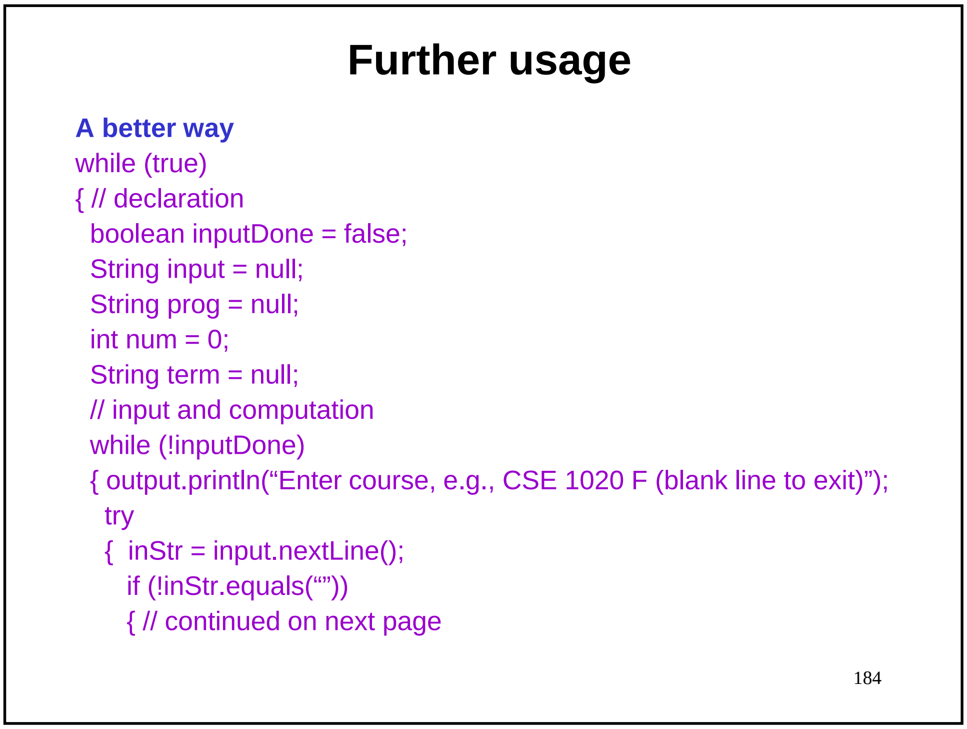
```
// input and computation
while (!inputDone)
{ output.println(“Enter course, e.g., CSE 1020 S”);
  try
  {  inStr = input.nextLine();
     int sep1 = inStr.indexOf(“ ”);
     int sep2 = inStr.indexOf(“ ”, sep1+1);
     prog = inStr.substring(0,sep1);
     num = Integer.parseInt(inStr.substring(sep1+1, sep2));
     term = inStr.substring(sep2+1);
     SE.require(term.equals(“F”)||term.equals(“W”)||term.equals(“S”));
     inputDone = true;
  } // continued on next page
```



Further usage

A better way

```
while (true)
{ // declaration
  boolean inputDone = false;
  String input = null;
  String prog = null;
  int num = 0;
  String term = null;
  // input and computation
  while (!inputDone)
  { output.println(“Enter course, e.g., CSE 1020 S (blank line to exit)”);
    try
    {  inStr = input.nextLine();
       if (!inStr.equals(“”))
       {
```



Further usage

A better way

```
while (true)
{ // declaration
  boolean inputDone = false;
  String input = null;
  String prog = null;
  int num = 0;
  String term = null;
  // input and computation
  while (!inputDone)
  { output.println(“Enter course, e.g., CSE 1020 F (blank line to exit)”);
    try
    {  inStr = input.nextLine();
       if (!inStr.equals(“”))
       { // continued on next page
```

Further usage

A better way

```
      // continued from previous page
      int sep1 = inStr.indexOf(“ ”);
      int sep2 = inStr.indexOf(“ ”, sep1+1);
      prog = inStr.substring(0,sep1);
      num = Integer.parseInt(inStr.substring(sep1+1, sep2));
      term = inStr.substring(sep2+1);
      SE.require(term.equals(“F”)||term.equals(“W”)||term.equals”S”));

    inputDone = true;
  } // end try
```

Further usage

A better way

```
      // continued from previous page
      int sep1 = inStr.indexOf(“ ”);
      int sep2 = inStr.indexOf(“ ”, sep1+1);
      prog = inStr.substring(0,sep1);
      num = Integer.parseInt(inStr.substring(sep1+1, sep2));
      term = inStr.substring(sep2+1);
      SE.require(term.equals(“F”)||term.equals(“W”)||term.equals”S”));
    } // end if
    inputDone = true;
  } // end try
```

Further usage

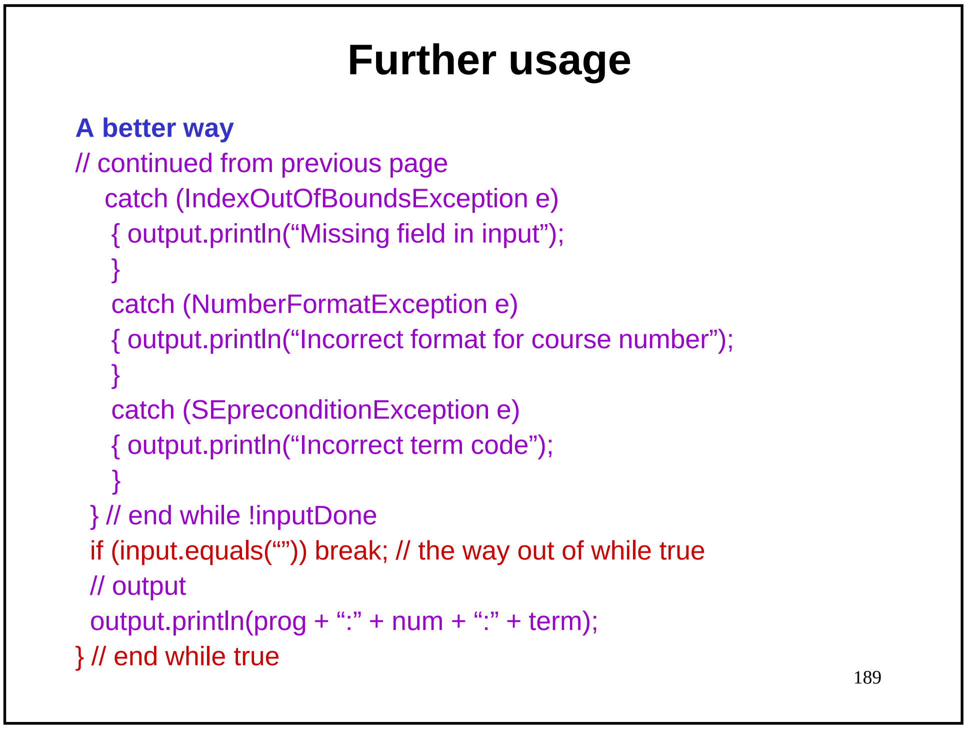
A better way

```
        // continued from previous page
        int sep1 = inStr.indexOf(“ ”);
        int sep2 = inStr.indexOf(“ ”, sep1+1);
        prog = inStr.substring(0,sep1);
        num = Integer.parseInt(inStr.substring(sep1+1, sep2));
        term = inStr.substring(sep2+1);
        SE.require(term.equals(“F”)||term.equals(“W”)||term.equals”S”));
      } // end if
      inputDone = true;
   } // end try
   // continued on next page
```

Further usage

A better way

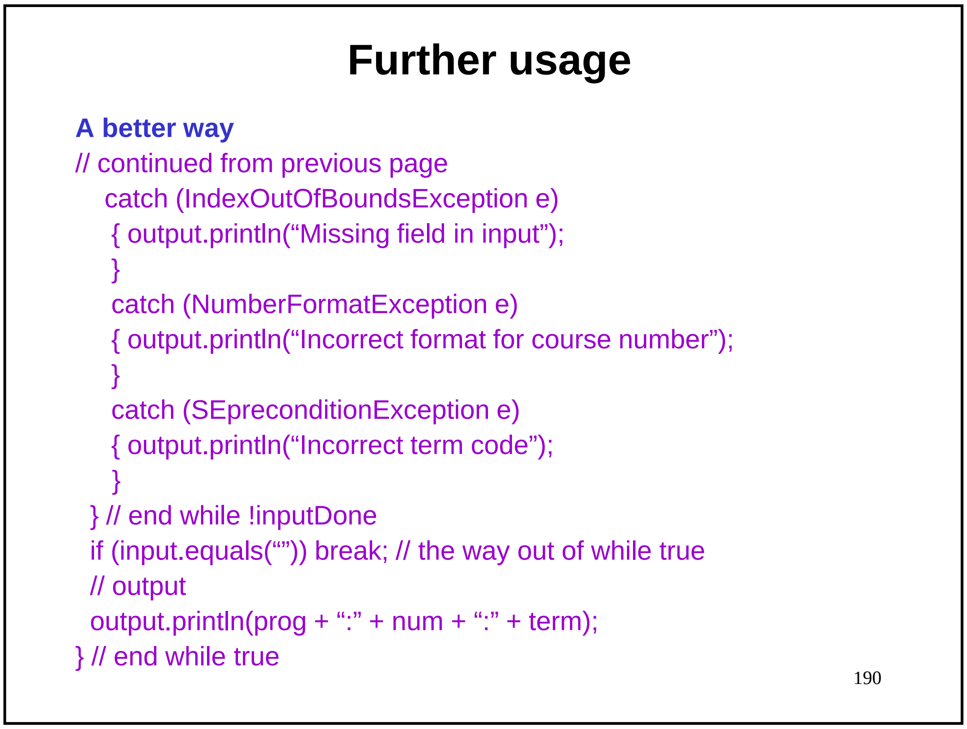
```
// continued from previous page
    catch (IndexOutOfBoundsException e)
     { output.println(“Missing field in input”);
     }
     catch (NumberFormatException e)
     { output.println(“Incorrect format for course number”);
     }
     catch (SEpreconditionException e)
     { output.println(“Incorrect term code”);
     }
  } // end while !inputDone

  // output
  output.println(prog + “:” + num + “:” + term);
```



Further usage

A better way

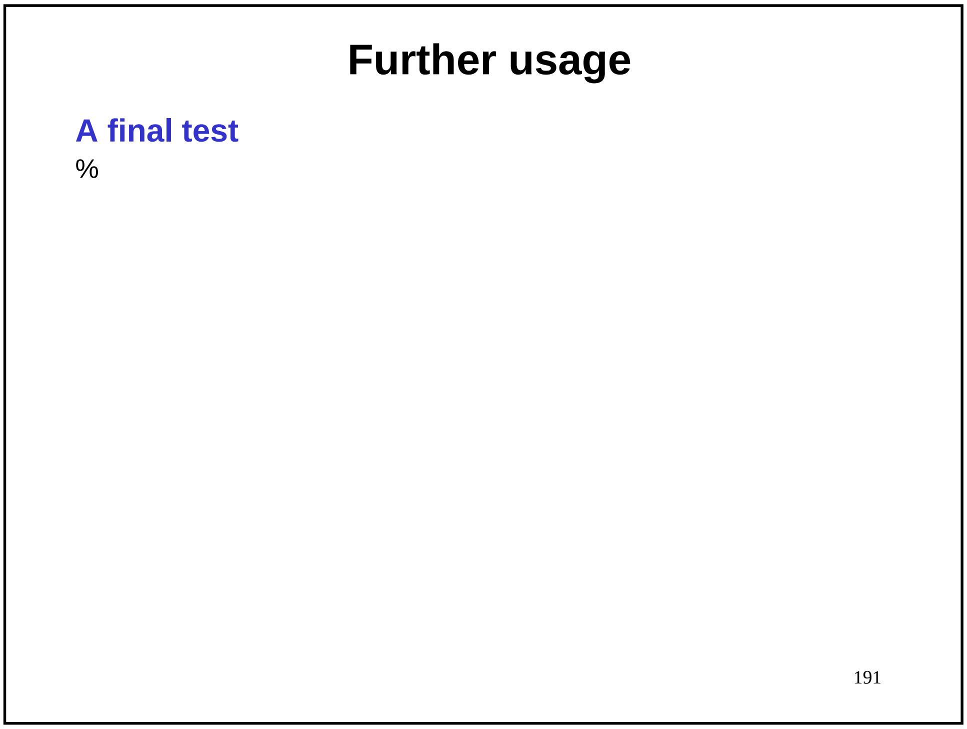
```
// continued from previous page
   catch (IndexOutOfBoundsException e)
    { output.println(“Missing field in input”);
    }
    catch (NumberFormatException e)
    { output.println(“Incorrect format for course number”);
    }
    catch (SEpreconditionException e)
    { output.println(“Incorrect term code”);
    }
  } // end while !inputDone
  if (input.equals(“”)) break; // the way out of while true
  // output
  output.println(prog + “:” + num + “:” + term);
} // end while true
```



Further usage

A better way

```
// continued from previous page
   catch (IndexOutOfBoundsException e)
    { output.println(“Missing field in input”);
    }
    catch (NumberFormatException e)
    { output.println(“Incorrect format for course number”);
    }
    catch (SEpreconditionException e)
    { output.println(“Incorrect term code”);
    }
  } // end while !inputDone
  if (input.equals(“”)) break; // the way out of while true
  // output
  output.println(prog + “:” + num + “:” + term);
} // end while true
```

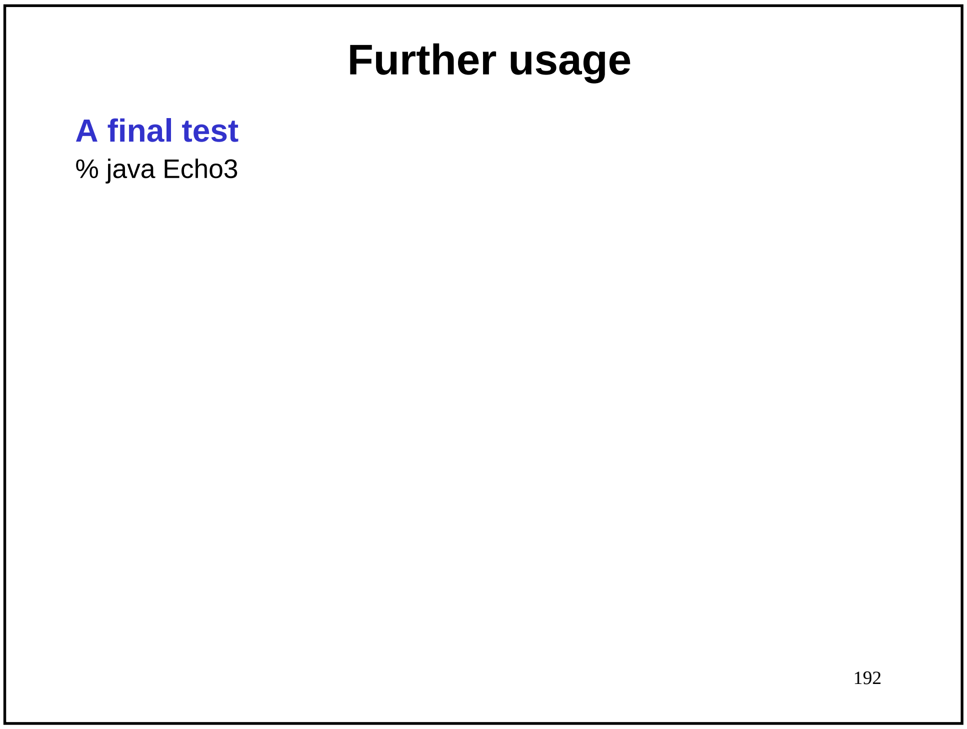


Further usage

A final test

```
%
```

191



Further usage

A final test

```
% java Echo3
```

192

Further usage

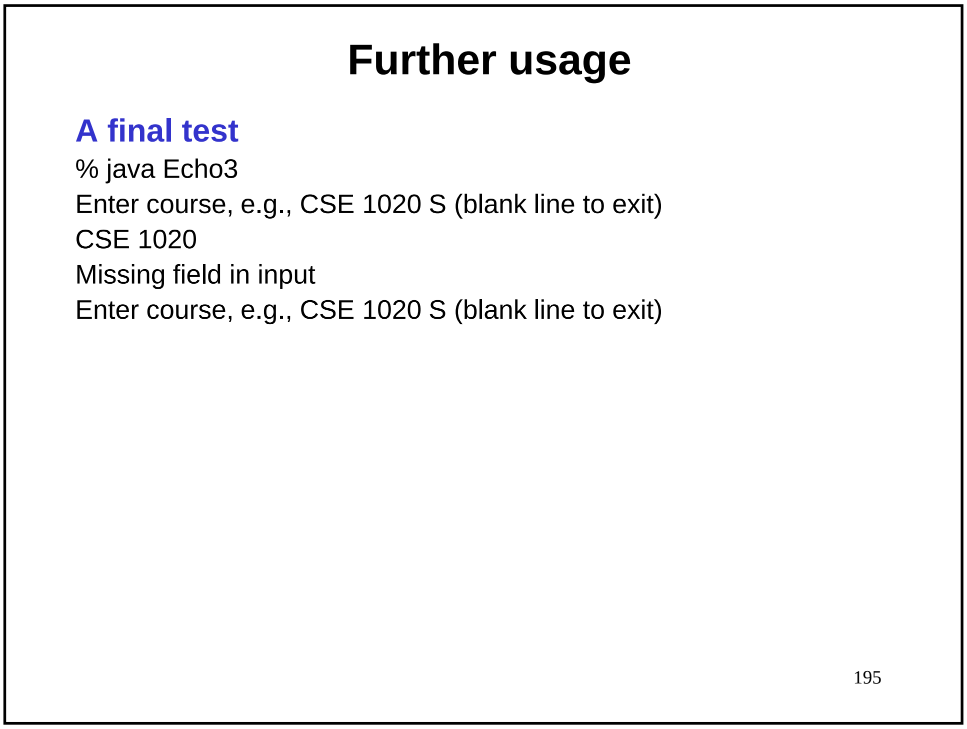
A final test

```
% java Echo3
Enter course, e.g., CSE 1020 S (blank line to exit)
```

Further usage

A final test

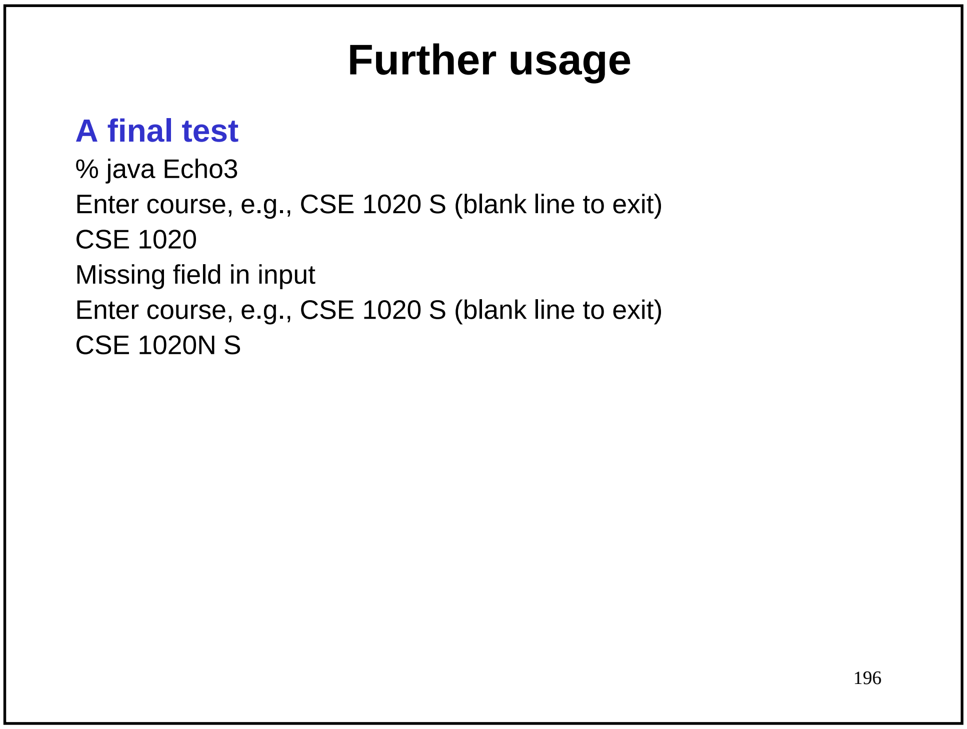
```
% java Echo3
Enter course, e.g., CSE 1020 S (blank line to exit)
CSE 1020
```



Further usage

A final test

```
% java Echo3
Enter course, e.g., CSE 1020 S (blank line to exit)
CSE 1020
Missing field in input
Enter course, e.g., CSE 1020 S (blank line to exit)
```



Further usage

A final test

```
% java Echo3
Enter course, e.g., CSE 1020 S (blank line to exit)
CSE 1020
Missing field in input
Enter course, e.g., CSE 1020 S (blank line to exit)
CSE 1020N S
```

Further usage

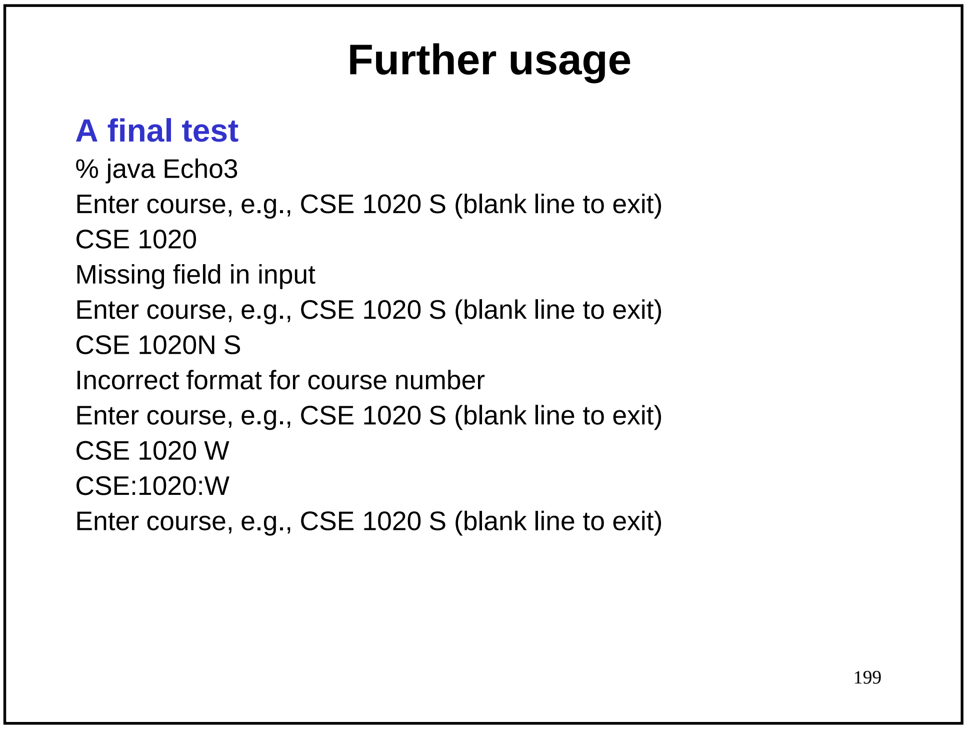
A final test

```
% java Echo3
Enter course, e.g., CSE 1020 S (blank line to exit)
CSE 1020
Missing field in input
Enter course, e.g., CSE 1020 S (blank line to exit)
CSE 1020N S
Incorrect format for course number
Enter course, e.g., CSE 1020 S (blank line to exit)
```

Further usage

A final test

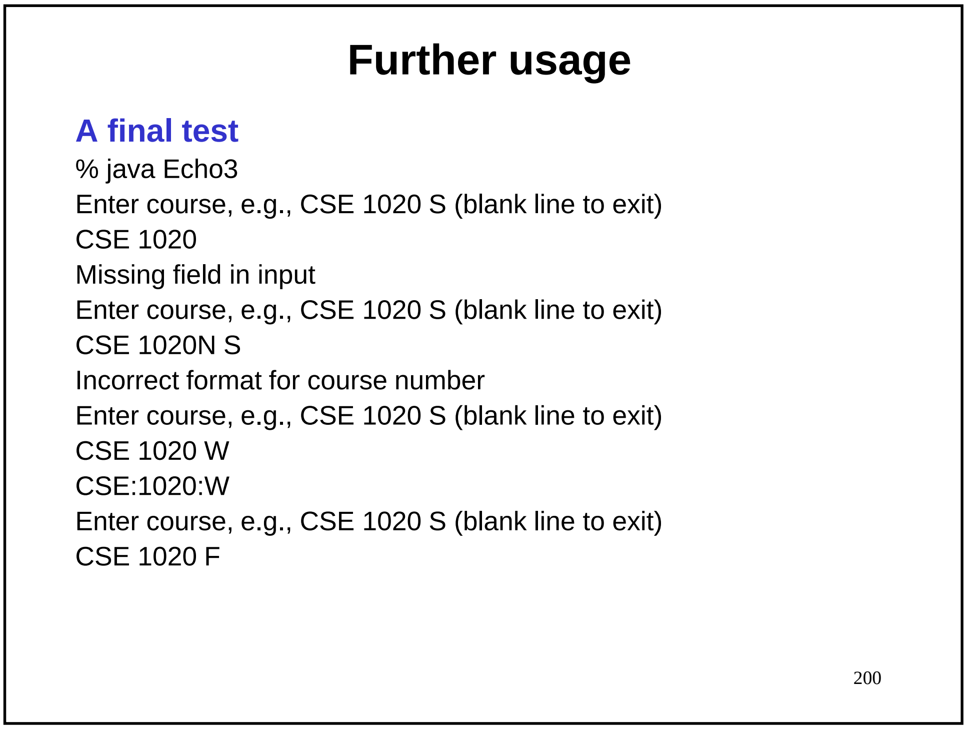
```
% java Echo3
Enter course, e.g., CSE 1020 S (blank line to exit)
CSE 1020
Missing field in input
Enter course, e.g., CSE 1020 S (blank line to exit)
CSE 1020N S
Incorrect format for course number
Enter course, e.g., CSE 1020 S (blank line to exit)
CSE 1020 W
```



Further usage

A final test

```
% java Echo3
Enter course, e.g., CSE 1020 S (blank line to exit)
CSE 1020
Missing field in input
Enter course, e.g., CSE 1020 S (blank line to exit)
CSE 1020N S
Incorrect format for course number
Enter course, e.g., CSE 1020 S (blank line to exit)
CSE 1020 W
CSE:1020:W
Enter course, e.g., CSE 1020 S (blank line to exit)
```



Further usage

A final test

```
% java Echo3
Enter course, e.g., CSE 1020 S (blank line to exit)
CSE 1020
Missing field in input
Enter course, e.g., CSE 1020 S (blank line to exit)
CSE 1020N S
Incorrect format for course number
Enter course, e.g., CSE 1020 S (blank line to exit)
CSE 1020 W
CSE:1020:W
Enter course, e.g., CSE 1020 S (blank line to exit)
CSE 1020 F
```

Further usage

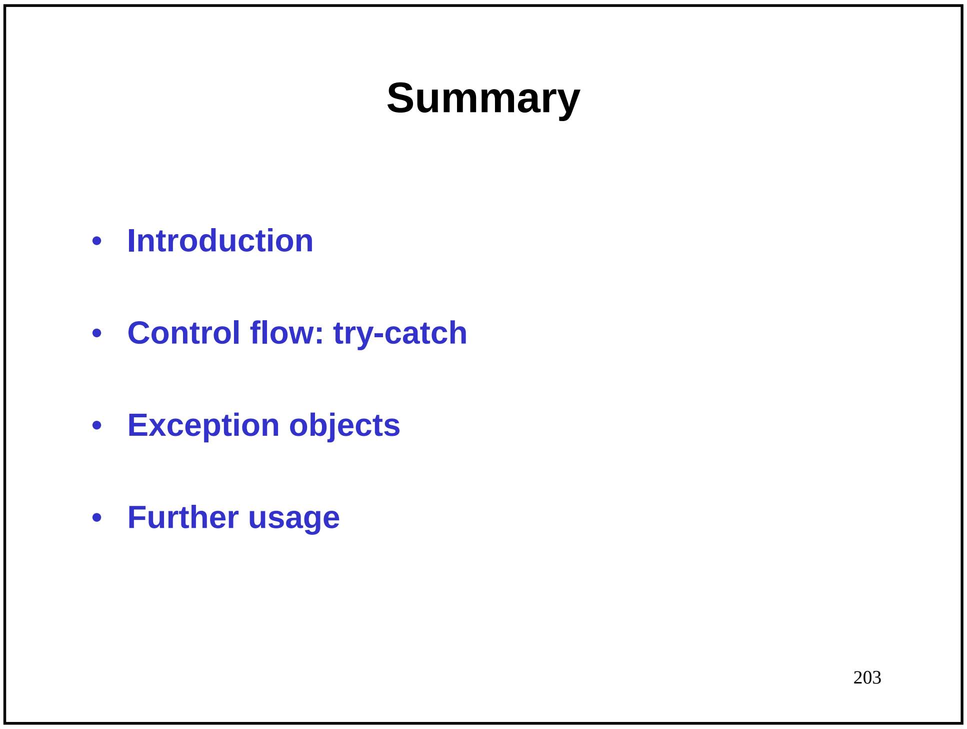
A final test

```
% java Echo3
Enter course, e.g., CSE 1020 S (blank line to exit)
CSE 1020
Missing field in input
Enter course, e.g., CSE 1020 S (blank line to exit)
CSE 1020N S
Incorrect format for course number
Enter course, e.g., CSE 1020 S (blank line to exit)
CSE 1020 W
CSE:1020:W
Enter course, e.g., CSE 1020 S (blank line to exit)
CSE 1020 F
CSE:1020:F
Enter course, e.g., CSE 1020 S
```

Further usage

A final test

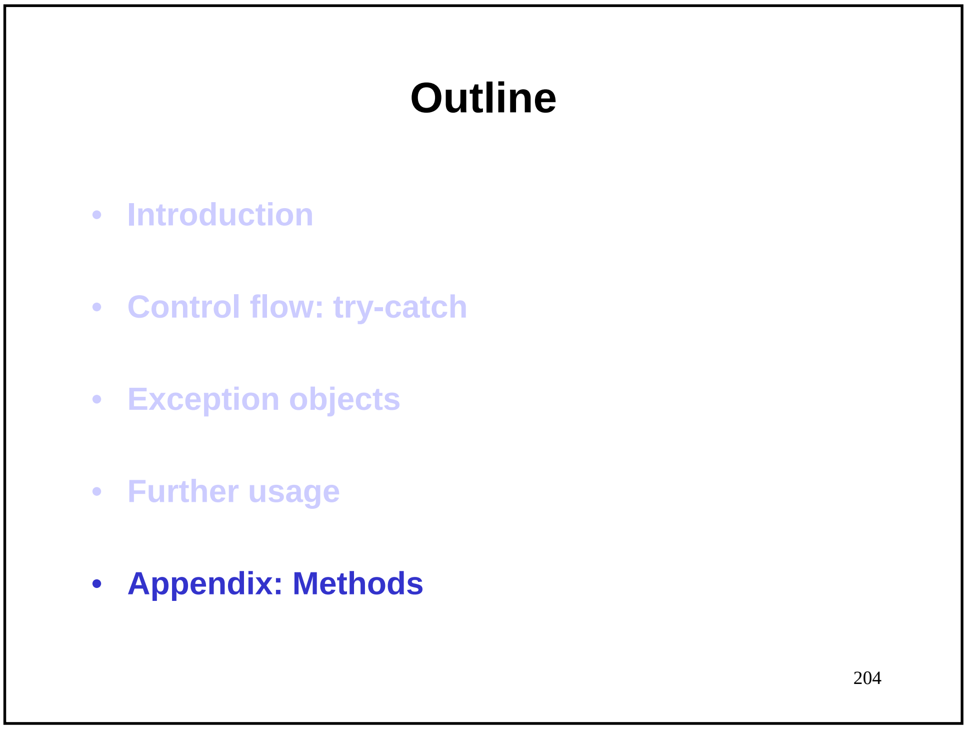
```
% java Echo3
Enter course, e.g., CSE 1020 S (blank line to exit)
CSE 1020
Missing field in input
Enter course, e.g., CSE 1020 S (blank line to exit)
CSE 1020N S
Incorrect format for course number
Enter course, e.g., CSE 1020 S (blank line to exit)
CSE 1020 W
CSE:1020:W
Enter course, e.g., CSE 1020 S (blank line to exit)
CSE 1020 F
CSE:1020:F
Enter course, e.g., CSE 1020 S

%
```



Summary

- Introduction
- Control flow: try-catch
- Exception objects
- Further usage

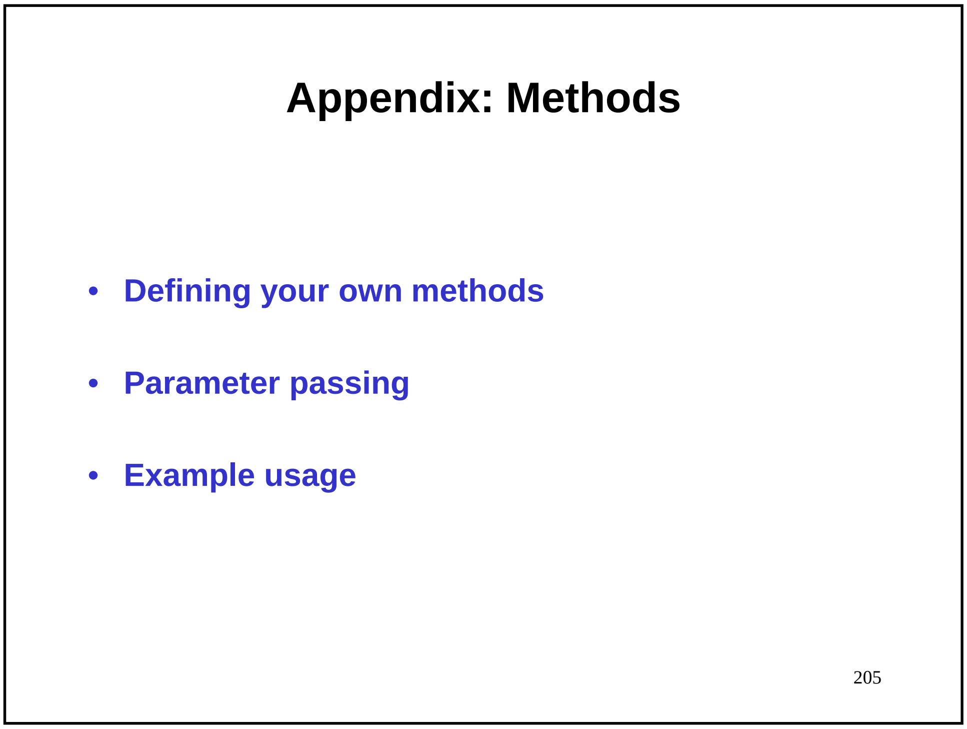
203



Outline

- Introduction
- Control flow: try-catch
- Exception objects
- Further usage
- Appendix: Methods

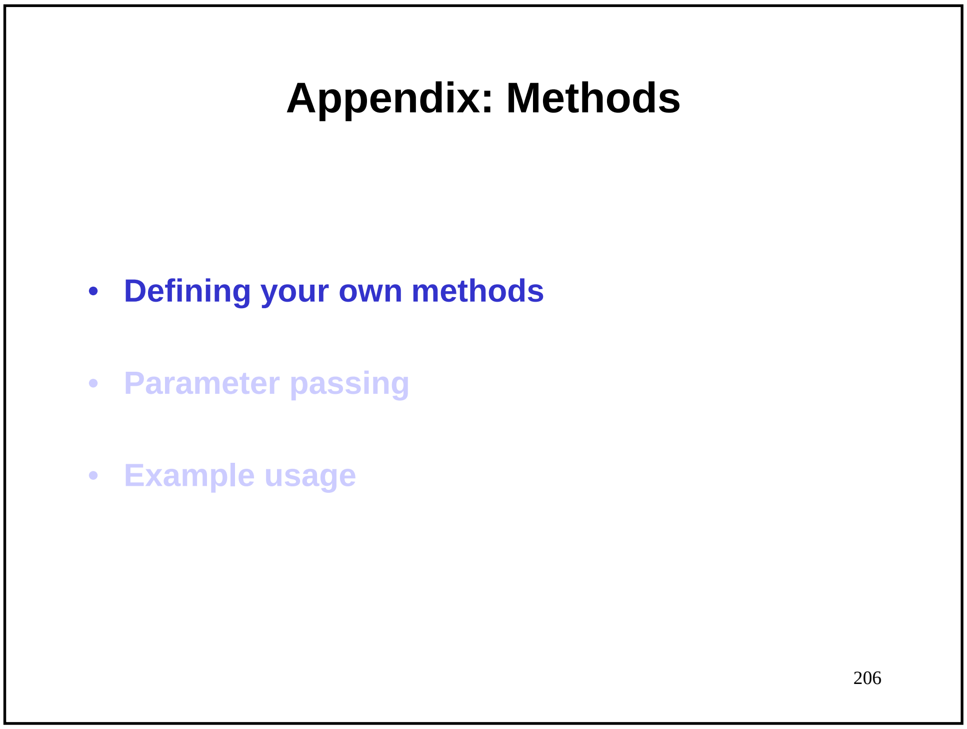
204



Appendix: Methods

- Defining your own methods
- Parameter passing
- Example usage

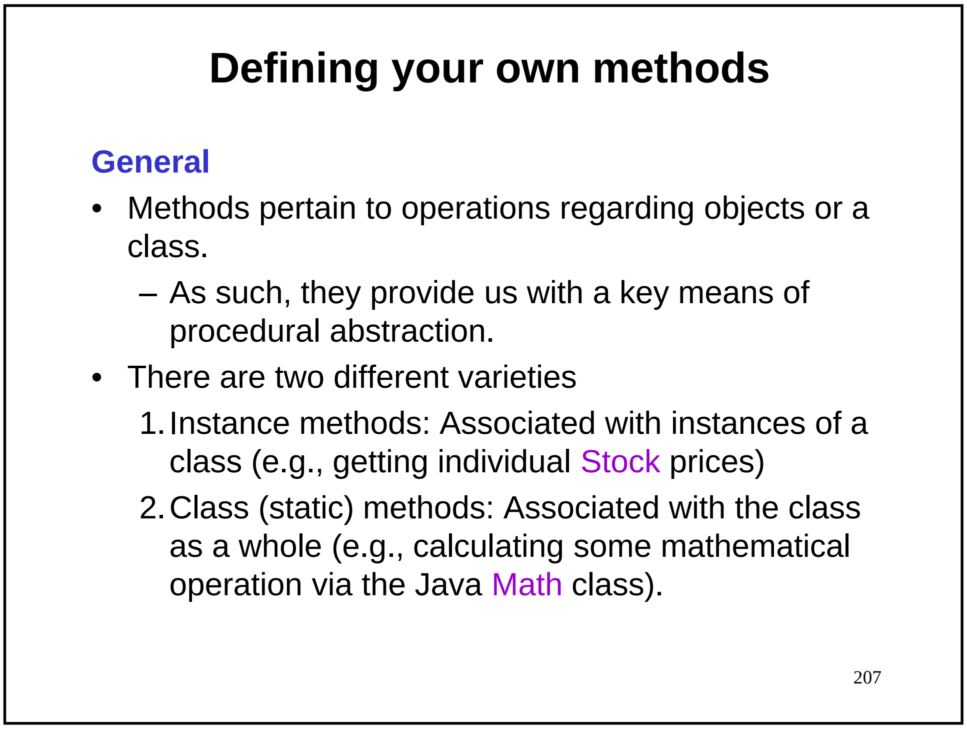
205



Appendix: Methods

- Defining your own methods
- Parameter passing
- Example usage

206

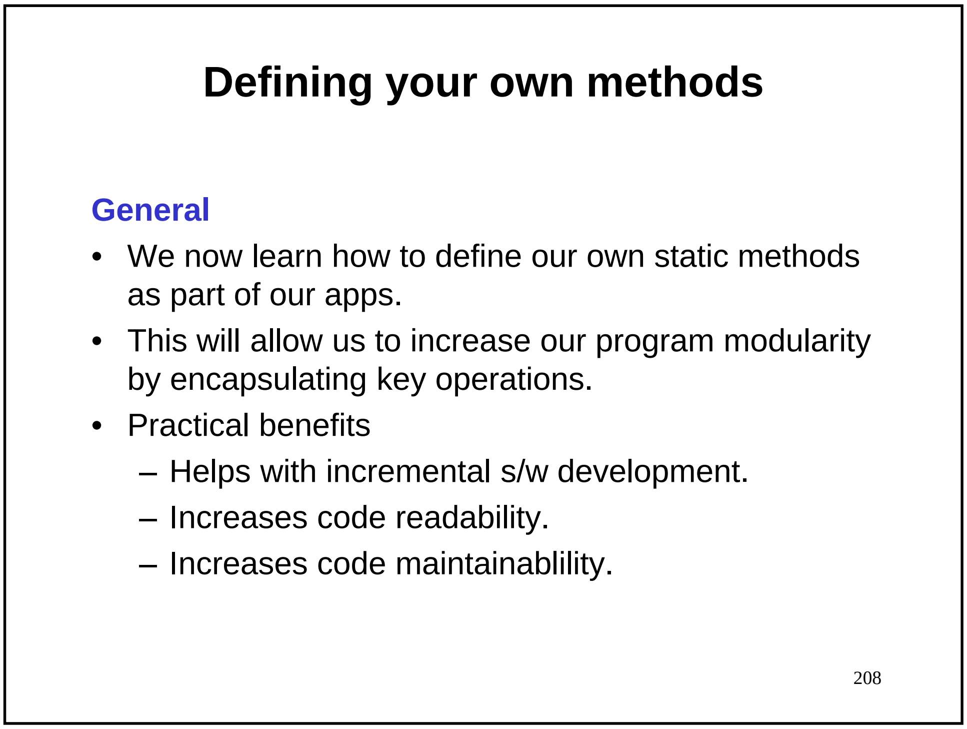


Defining your own methods

General

- Methods pertain to operations regarding objects or a class.
  - – As such, they provide us with a key means of procedural abstraction.
- There are two different varieties
  1. Instance methods: Associated with instances of a class (e.g., getting individual Stock prices)
  2. Class (static) methods: Associated with the class as a whole (e.g., calculating some mathematical operation via the Java Math class).

207



Defining your own methods

General

- We now learn how to define our own static methods as part of our apps.
- This will allow us to increase our program modularity by encapsulating key operations.
- Practical benefits
  - – Helps with incremental s/w development.
  - – Increases code readability.
  - – Increases code maintainablility.

208

Defining your own methods

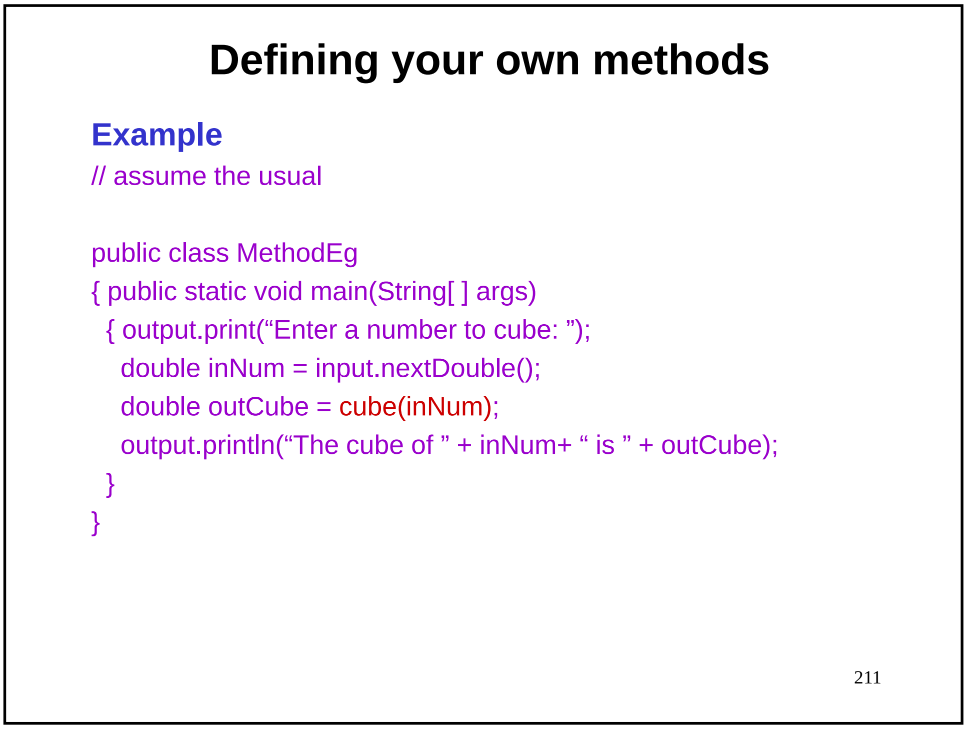
Example

```
// assume the usual

public class MethodEg
{ public static void main(String[ ] args)
  { output.print(“Enter a number to cube: ”);
    double inNum = input.nextDouble();
    double outCube = inNum * inNum * inNum;
    output.println(“The cube of ” + inNum + “ is ” + outCube);
  }
}
```

Defining your own methods

Example

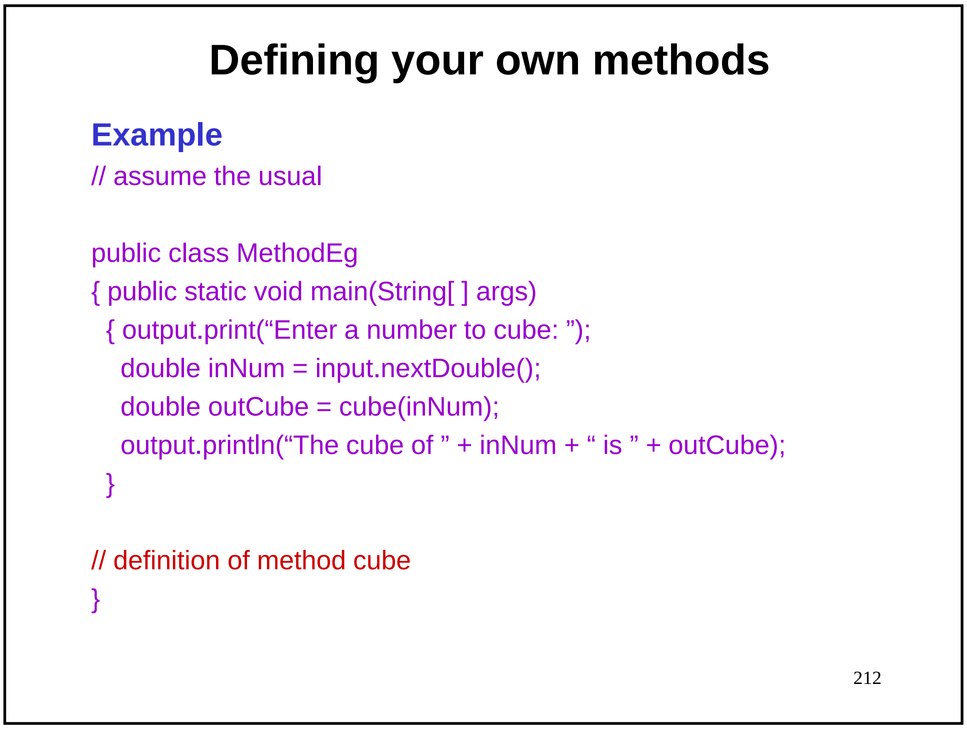
```
// assume the usual

public class MethodEg
{ public static void main(String[ ] args)
  { output.print(“Enter a number to cube: ”);
    double inNum = input.nextDouble();
    double outCube = inNum * inNum * inNum;
    output.println(“The cube of ” + inNum + “ is ” + outCube);
  }
}
```



Defining your own methods

Example

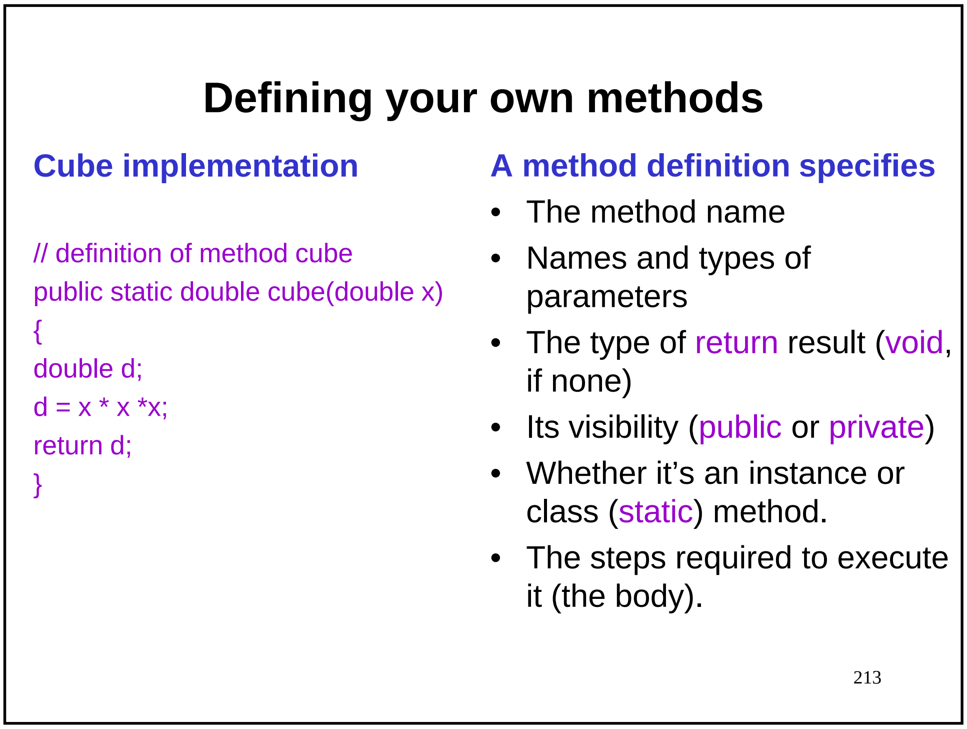
```
// assume the usual

public class MethodEg
{ public static void main(String[ ] args)
  { output.print(“Enter a number to cube: ”);
    double inNum = input.nextDouble();
    double outCube = cube(inNum);
    output.println(“The cube of ” + inNum+ “ is ” + outCube);
  }
}
```



Defining your own methods

Example

```
// assume the usual

public class MethodEg
{ public static void main(String[ ] args)
  { output.print(“Enter a number to cube: ”);
    double inNum = input.nextDouble();
    double outCube = cube(inNum);
    output.println(“The cube of ” + inNum + “ is ” + outCube);
  }

// definition of method cube
}
```



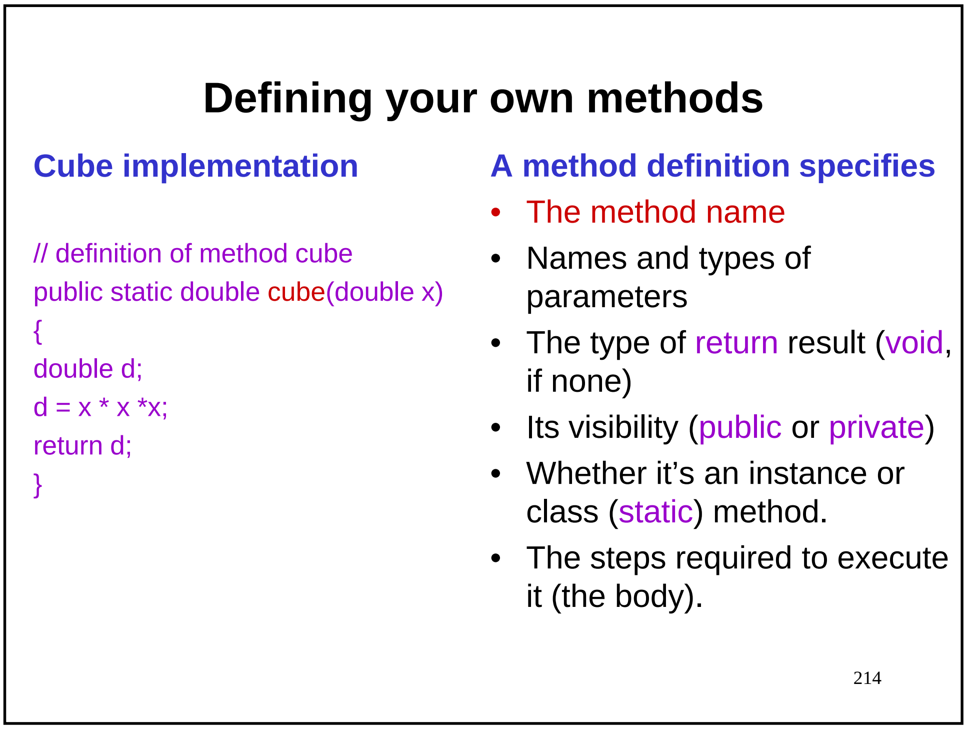
Defining your own methods

Cube implementation

```
// definition of method cube
public static double cube(double x)
{
double d;
d = x * x *x;
return d;
}
```

A method definition specifies

- The method name
- Names and types of parameters
- The type of return result (void, if none)
- Its visibility (public or private)
- Whether it's an instance or class (static) method.
- The steps required to execute it (the body).



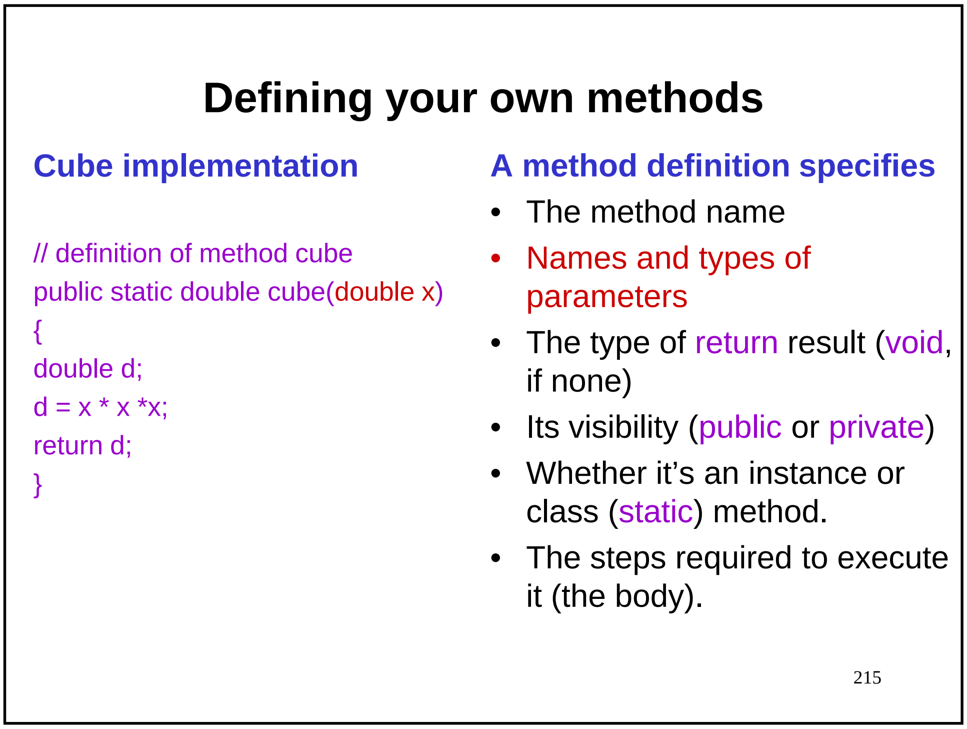
Defining your own methods

Cube implementation

```
// definition of method cube
public static double cube(double x)
{
double d;
d = x * x *x;
return d;
}
```

A method definition specifies

- The method name
- Names and types of parameters
- The type of return result (void, if none)
- Its visibility (public or private)
- Whether it's an instance or class (static) method.
- The steps required to execute it (the body).



Defining your own methods

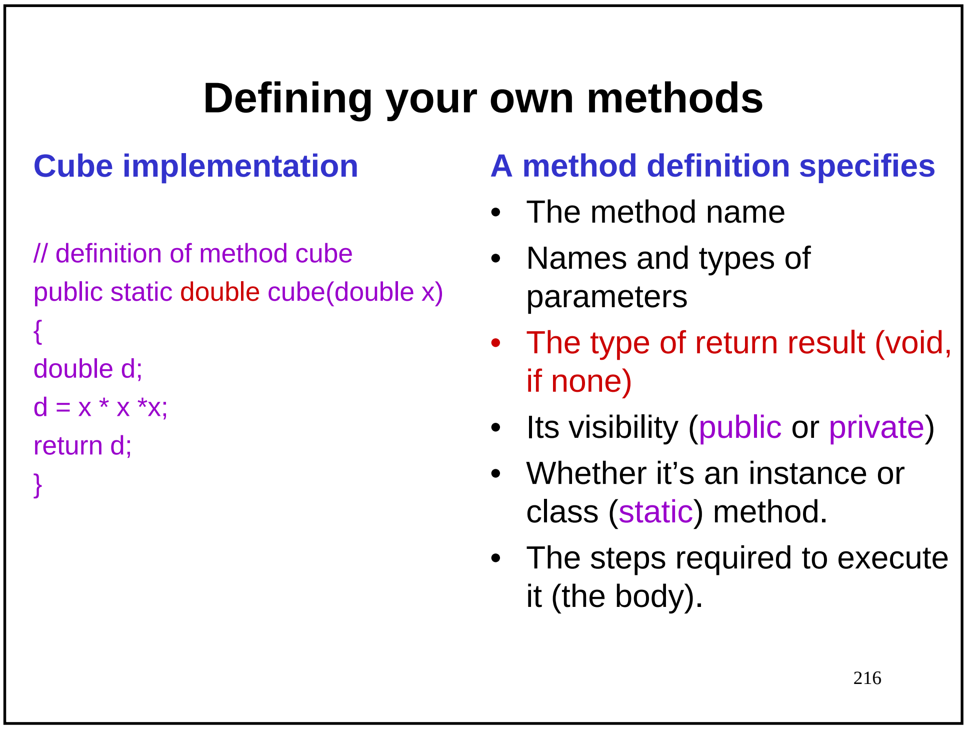
Cube implementation

```
// definition of method cube
public static double cube(double x)
{
double d;
d = x * x *x;
return d;
}
```

A method definition specifies

- The method name
- Names and types of parameters
- The type of return result (void, if none)
- Its visibility (public or private)
- Whether it’s an instance or class (static) method.
- The steps required to execute it (the body).

215



Defining your own methods

Cube implementation

```
// definition of method cube
public static double cube(double x)
{
double d;
d = x * x *x;
return d;
}
```

A method definition specifies

- The method name
- Names and types of parameters
- The type of return result (void, if none)
- Its visibility (public or private)
- Whether it’s an instance or class (static) method.
- The steps required to execute it (the body).

216

Defining your own methods

Cube implementation

```
// definition of method cube
public static double cube(double x)
{
double d;
d = x * x *x;
return d;
}
```

A method definition specifies

- The method name
- Names and types of parameters
- The type of return result (void, if none)
- Its visibility (public or private)
- Whether it’s an instance or class (static) method.
- The steps required to execute it (the body).

Defining your own methods

Cube implementation

```
// definition of method cube
public static double cube(double x)
{
double d;
d = x * x *x;
return d;
}
```

A method definition specifies

- The method name
- Names and types of parameters
- The type of return result (void, if none)
- Its visibility (public or private)
- Whether it’s an instance or class (static) method.
- The steps required to execute it (the body) between matched { and }.

Defining your own methods

Cube implementation

```
// definition of method cube
public static double cube(double x)
{
double d;
d = x * x *x;
return d;
}
```

A method definition specifies

- The method name
- Names and types of parameters
- The type of return result (void, if none)
- Its visibility (public or private)
- Whether it’s an instance or class (static) method.
- The steps required to execute it (the body) between matched { and }.

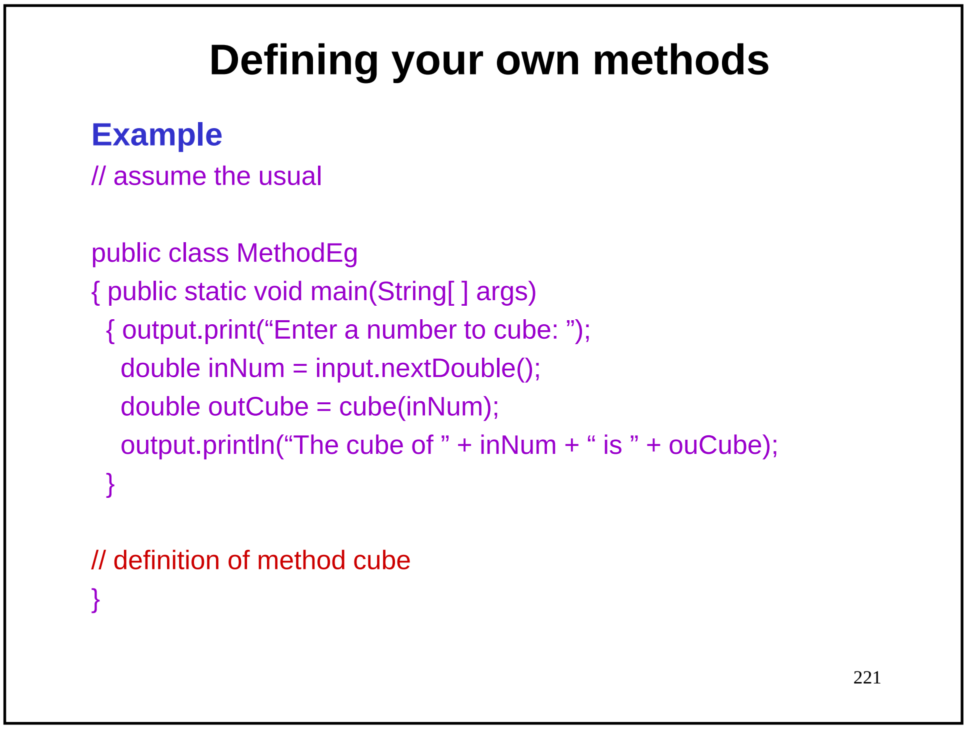
Defining your own methods

Cube implementation

```
// definition of method cube
public static double cube(double x)
{
double d;
d = x * x *x;
return d;
}
```

A method definition specifies

- The method name
- Names and types of parameters
- The type of return result (void, if none)
- Its visibility (public or private)
- Whether it’s an instance or class (static) method.
- The steps required to execute it (the body) between matched { and }.

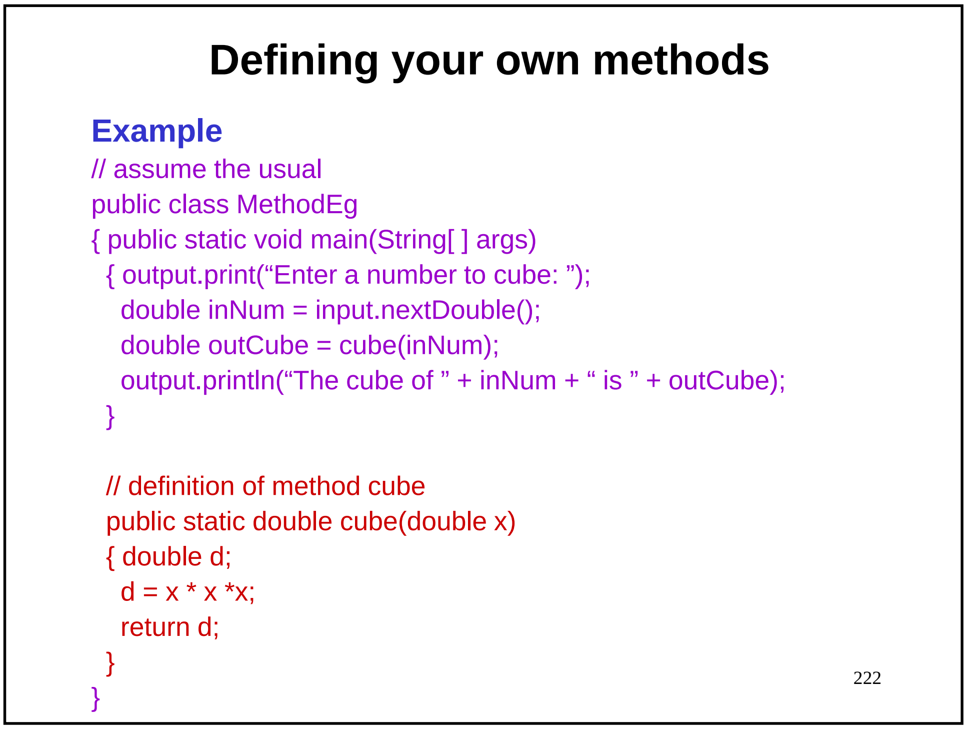


Defining your own methods

Example

```
// assume the usual

public class MethodEg
{ public static void main(String[ ] args)
  { output.print(“Enter a number to cube: ”);
    double inNum = input.nextDouble();
    double outCube = cube(inNum);
    output.println(“The cube of ” + inNum + “ is ” + ouCube);
  }

// definition of method cube
}
```

221



Defining your own methods

Example

```
// assume the usual
public class MethodEg
{ public static void main(String[ ] args)
  { output.print(“Enter a number to cube: ”);
    double inNum = input.nextDouble();
    double outCube = cube(inNum);
    output.println(“The cube of ” + inNum + “ is ” + outCube);
  }

  // definition of method cube
  public static double cube(double x)
  { double d;
    d = x * x *x;
    return d;
  }
}
```

222

Defining your own methods

Example

```
// assume the usual
public class MethodEg
{ public static void main(String[ ] args)
  { output.print(“Enter a number to cube: ”);
    double inNum = input.nextDouble();
    double outCube = cube(inNum);
    output.println(“The cube of ” + inNum + “ is ” + outCube);
  }

  // definition of method cube
  public static double cube(double x)
  { double d;
    d = x * x *x;
    return d;
  }
}
```

Defining your own methods

Parameters

- When we call a method, we often want to pass it some data.
- The data can then be used inside the method.
- We do this by having the method take parameters.
- Parameters are declared in the header of the method definition by specifying both their type and name.

Defining your own methods

Parameters

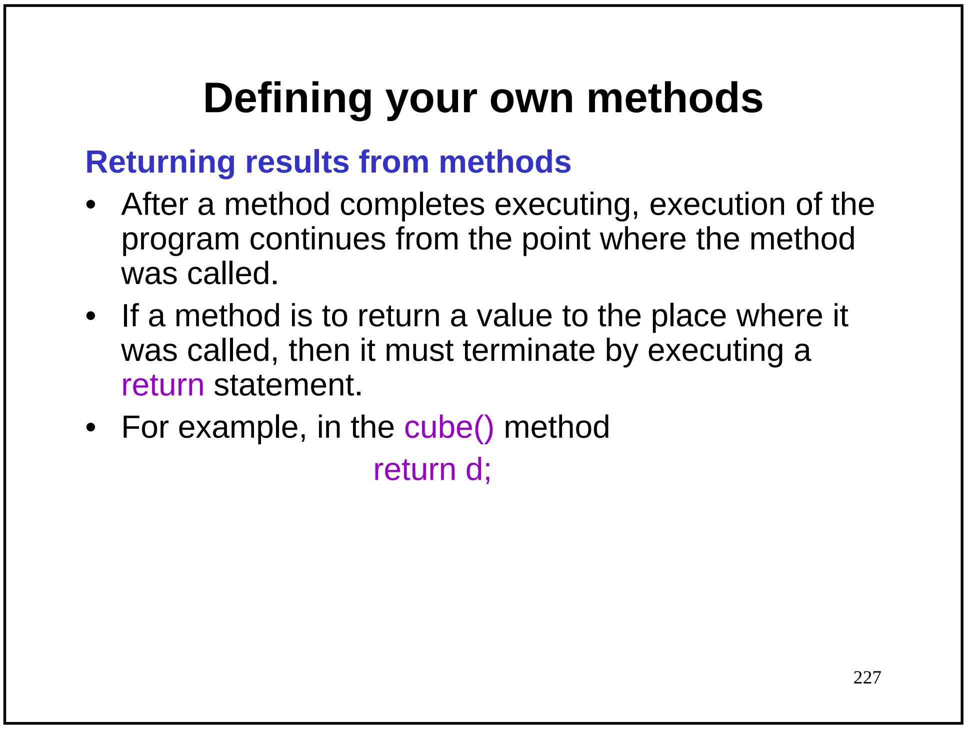
- When we call a method, we often want to pass it some data.
- The data can then be used inside the method.
- We do this by having the method take parameters.
- Parameters are declared in the header of the method definition by specifying both their type and name.
- For example, x is the parameter in

```
public static void cube(double x)
{
  double d;
  d = x * x *x;
  return d;
}
```

Defining your own methods

Returning results from methods

- After a method completes executing, execution of the program continues from the point where the method was called.
- If a method is to return a value to the place where it was called, then it must terminate by executing a return statement.

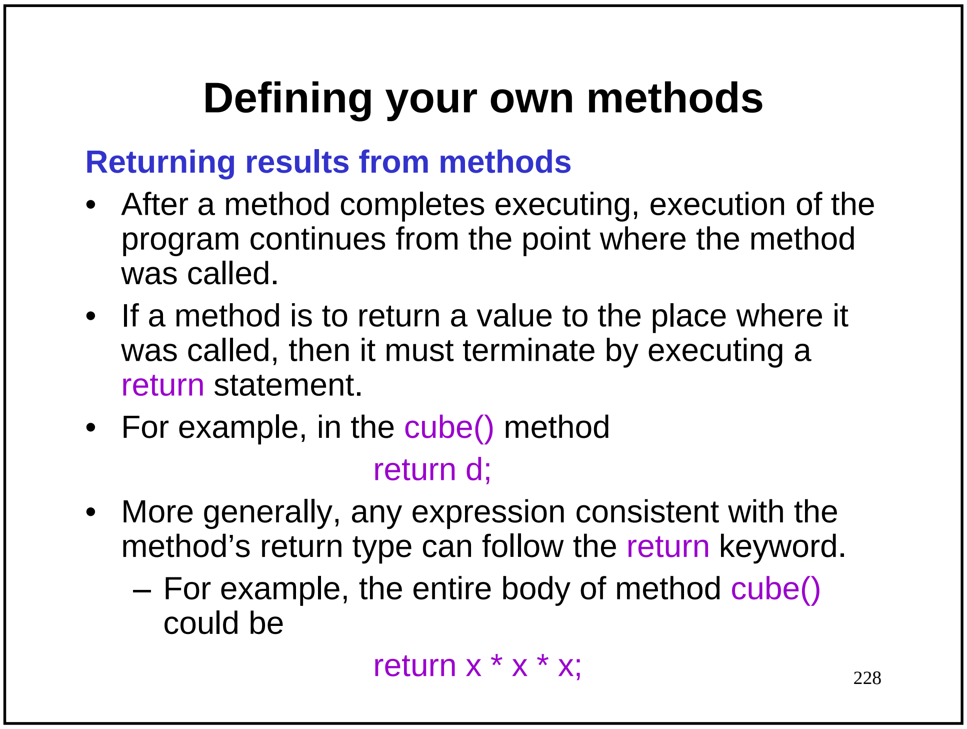


Defining your own methods

Returning results from methods

- After a method completes executing, execution of the program continues from the point where the method was called.
- If a method is to return a value to the place where it was called, then it must terminate by executing a return statement.
- For example, in the cube() method

```
return d;
```



Defining your own methods

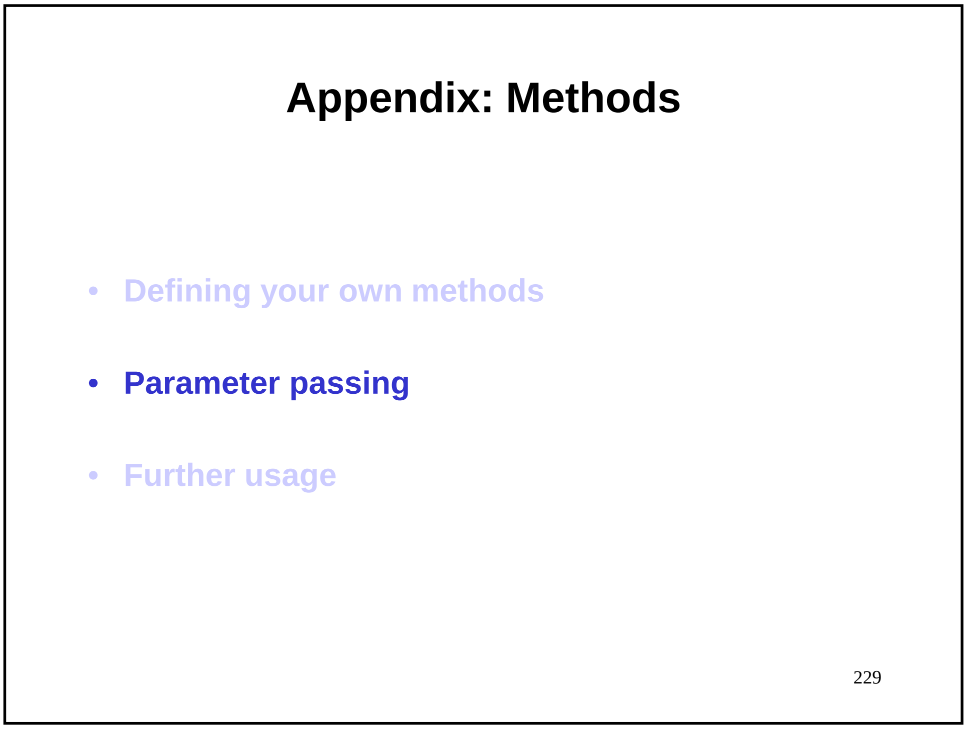
Returning results from methods

- After a method completes executing, execution of the program continues from the point where the method was called.
- If a method is to return a value to the place where it was called, then it must terminate by executing a return statement.
- For example, in the cube() method

```
return d;
```

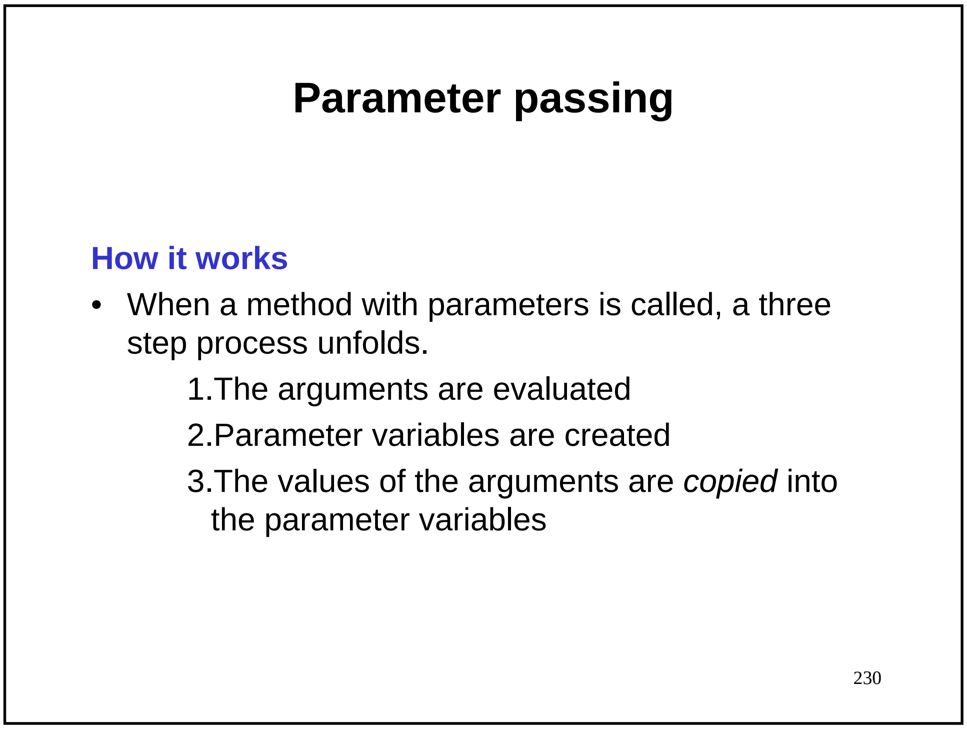
- More generally, any expression consistent with the method's return type can follow the return keyword.
  – For example, the entire body of method cube() could be

```
return x * x * x;
```



Appendix: Methods

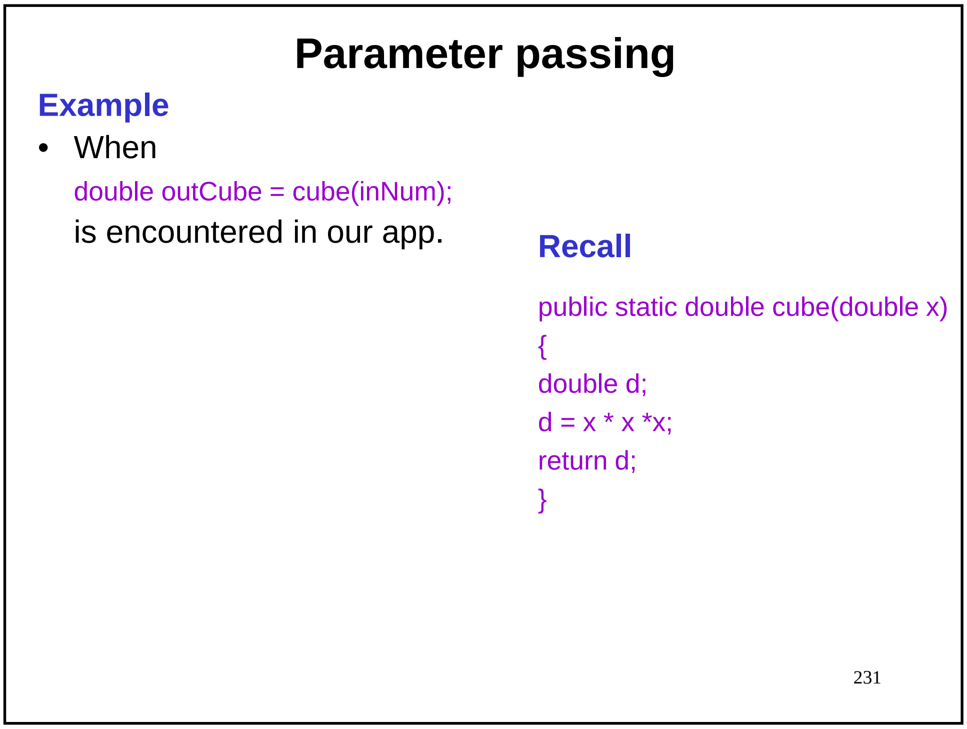
- Defining your own methods
- **Parameter passing**
- Further usage



Parameter passing

How it works

- When a method with parameters is called, a three step process unfolds.
    1. The arguments are evaluated
    2. Parameter variables are created
    3. The values of the arguments are *copied* into the parameter variables



Parameter passing

Example

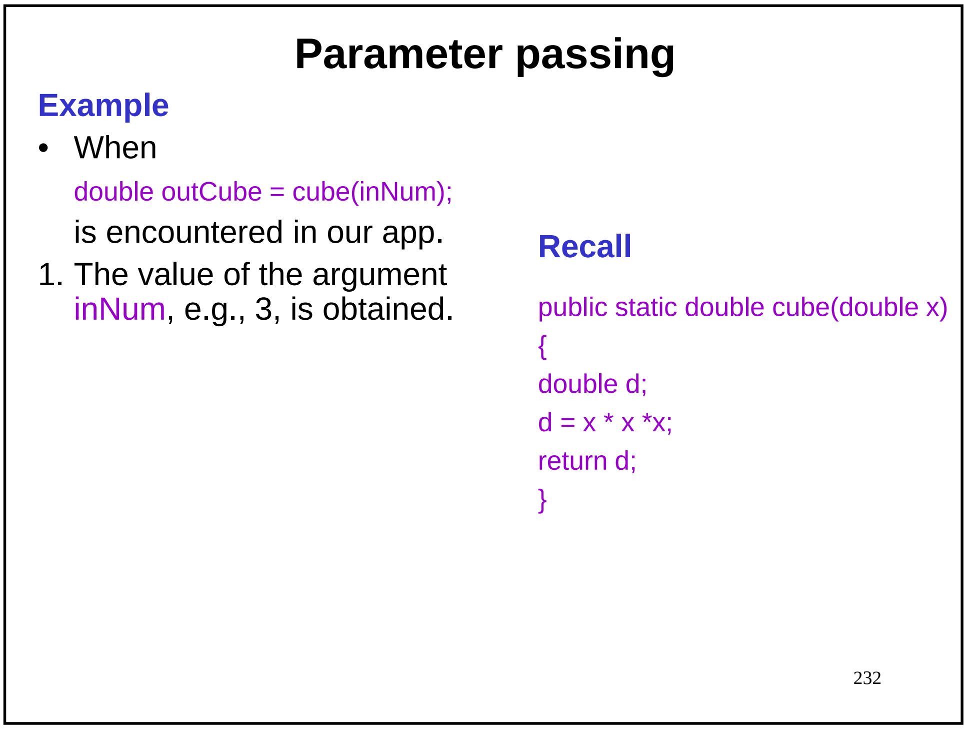
- When

  double outCube = cube(inNum);

  is encountered in our app.

Recall

```
public static double cube(double x)
{
double d;
d = x * x *x;
return d;
}
```



Parameter passing

Example

- When

  double outCube = cube(inNum);

  is encountered in our app.

1. The value of the argument inNum, e.g., 3, is obtained.

Recall

```
public static double cube(double x)
{
double d;
d = x * x *x;
return d;
}
```

Parameter passing

Example

- When

  `double outCube = cube(inNum);`

  is encountered in our app.

1. The value of the argument inNum, e.g., 3, is obtained.
2. A new formal parameter x is created (associated with a memory location).

Recall

```
public static double cube(double x)
{
double d;
d = x * x *x;
return d;
}
```

Parameter passing

Example

- When

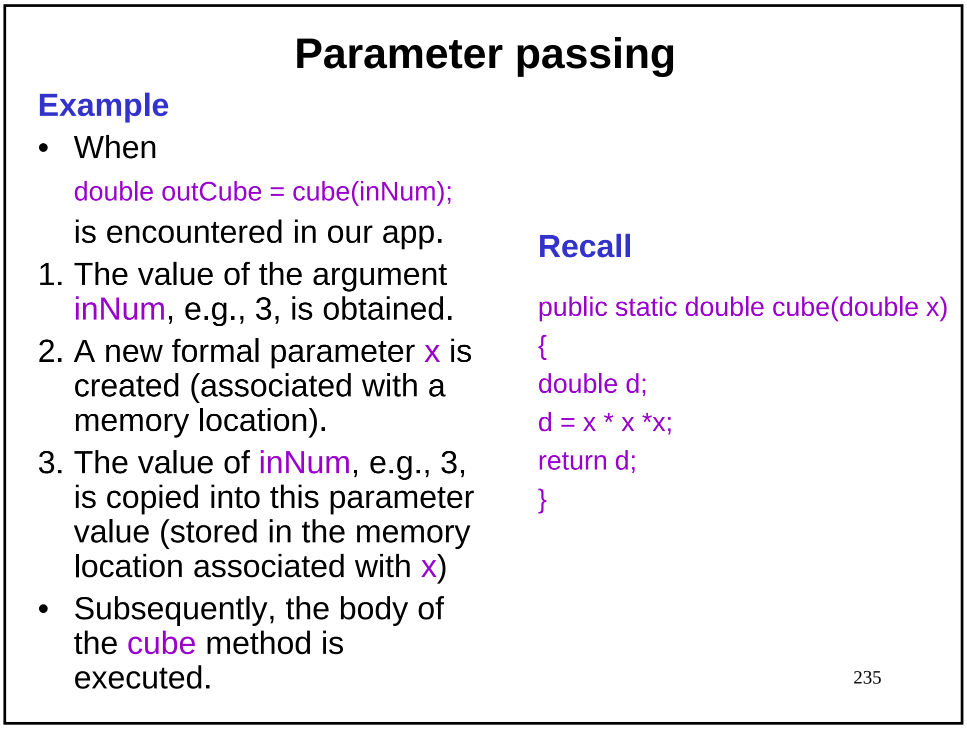
  `double outCube = cube(inNum);`

  is encountered in our app.

1. The value of the argument inNum, e.g., 3, is obtained.
2. A new formal parameter x is created (associated with a memory location).
3. The value of inNum, e.g., 3, is copied into this parameter value (stored in the memory location associated with x)

Recall

```
public static double cube(double x)
{
double d;
d = x * x *x;
return d;
}
```



Parameter passing

Example

- When

  ```
  double outCube = cube(inNum);
  ```

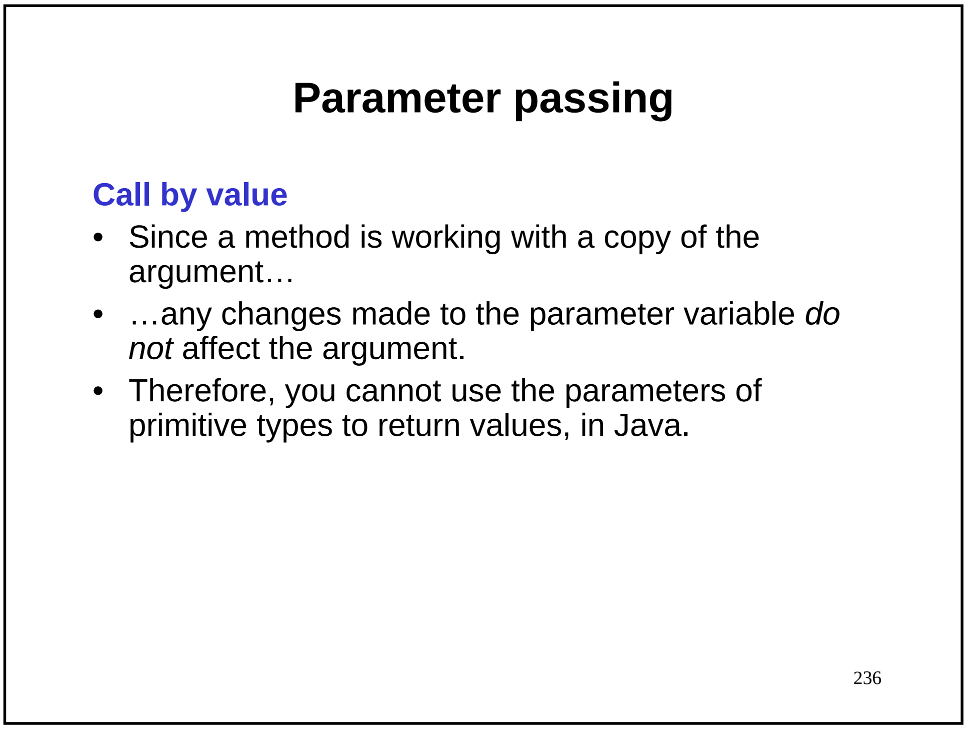
  is encountered in our app.

1. The value of the argument inNum, e.g., 3, is obtained.
2. A new formal parameter x is created (associated with a memory location).
3. The value of inNum, e.g., 3, is copied into this parameter value (stored in the memory location associated with x)

- Subsequently, the body of the cube method is executed.

Recall

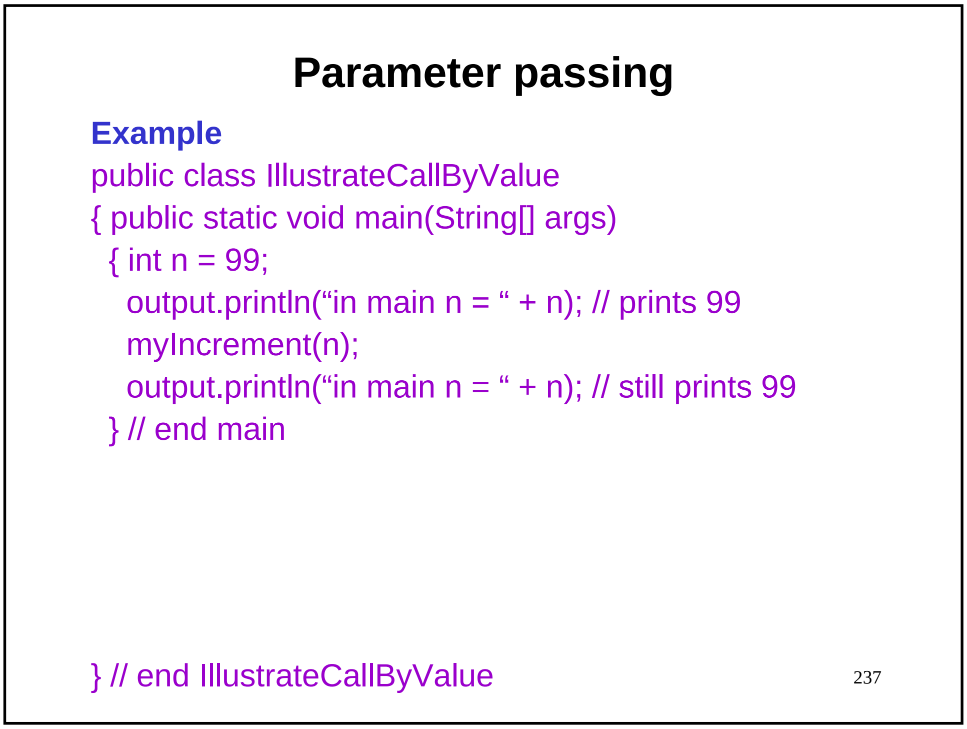
```
public static double cube(double x)
{
double d;
d = x * x *x;
return d;
}
```



Parameter passing

Call by value

- Since a method is working with a copy of the argument…
- …any changes made to the parameter variable *do not* affect the argument.
- Therefore, you cannot use the parameters of primitive types to return values, in Java.

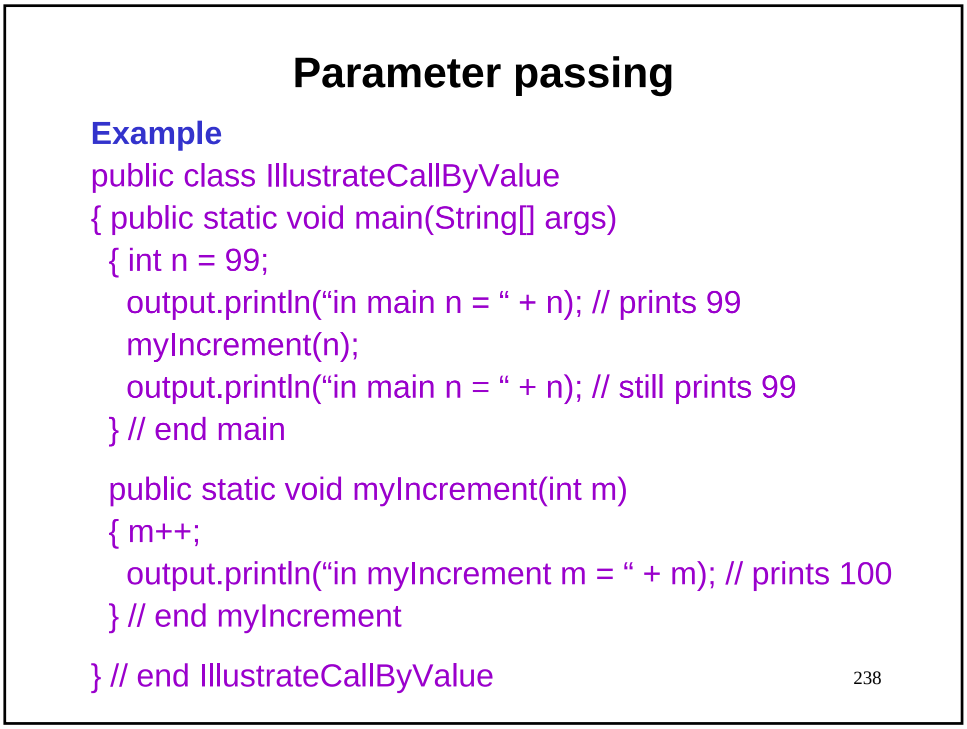


Parameter passing

Example

```
public class IllustrateCallByValue
{ public static void main(String[] args)
  { int n = 99;
    output.println(“in main n = “ + n); // prints 99
    myIncrement(n);
    output.println(“in main n = “ + n); // still prints 99
  } // end main






} // end IllustrateCallByValue
```

237



Parameter passing

Example

```
public class IllustrateCallByValue
{ public static void main(String[] args)
  { int n = 99;
    output.println(“in main n = “ + n); // prints 99
    myIncrement(n);
    output.println(“in main n = “ + n); // still prints 99
  } // end main

  public static void myIncrement(int m)
  { m++;
    output.println(“in myIncrement m = “ + m); // prints 100
  } // end myIncrement

} // end IllustrateCallByValue
```

238

Parameter passing

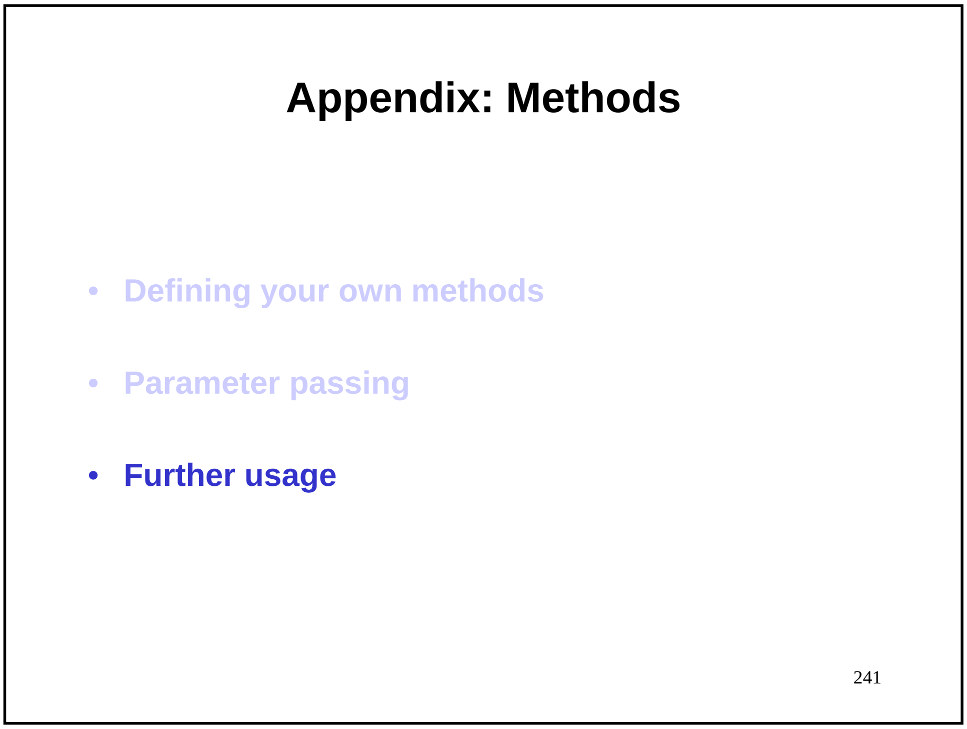
Call by value

- Since a method is working with a copy of the argument…
- …any changes made to the parameter variable *do not* affect the argument.
- Therefore, you cannot use the parameters of primitive types to return values, in Java.
- This mode of parameter passing is named call by value.
  - It is the value of the argument that is passed to the formal parameter…
  - …*not* the argument per se.

Parameter passing

When the argument is an object type

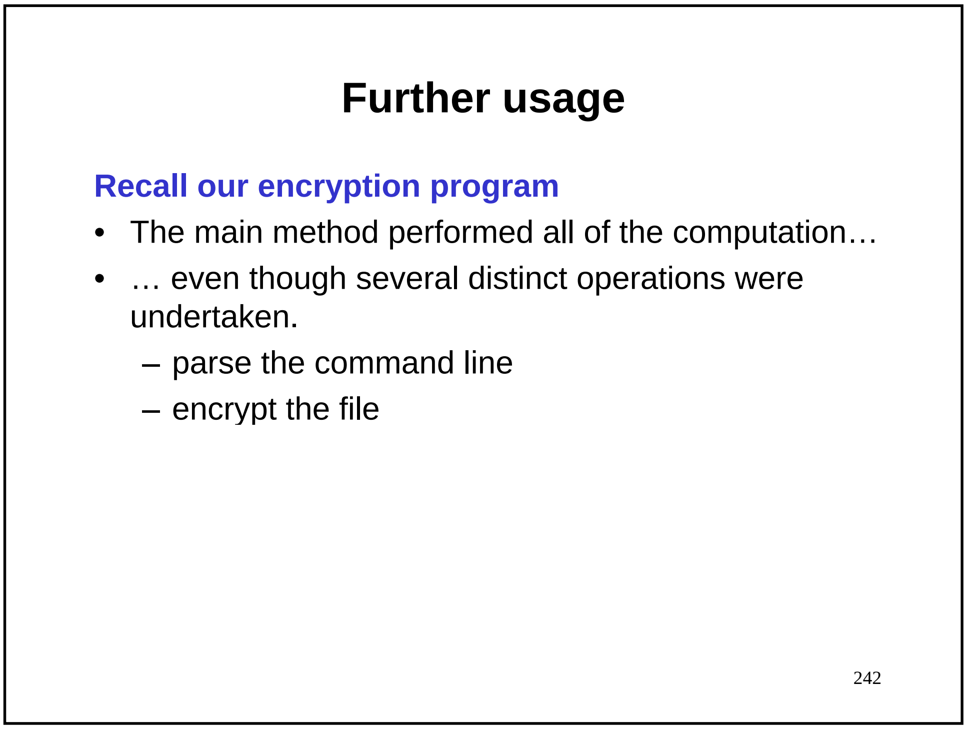
- When the type of a parameter is an object type…
- …only the reference in the argument gets copied in the parameter
- Both the argument and parameter refer to the same object.
- Thus, the method is working on the original object.
- Any change to its attributes persists when the method returns.
- Significantly, object parameters can be used by a method to send back results to the caller!



Appendix: Methods

- Defining your own methods
- Parameter passing
- **Further usage**

241

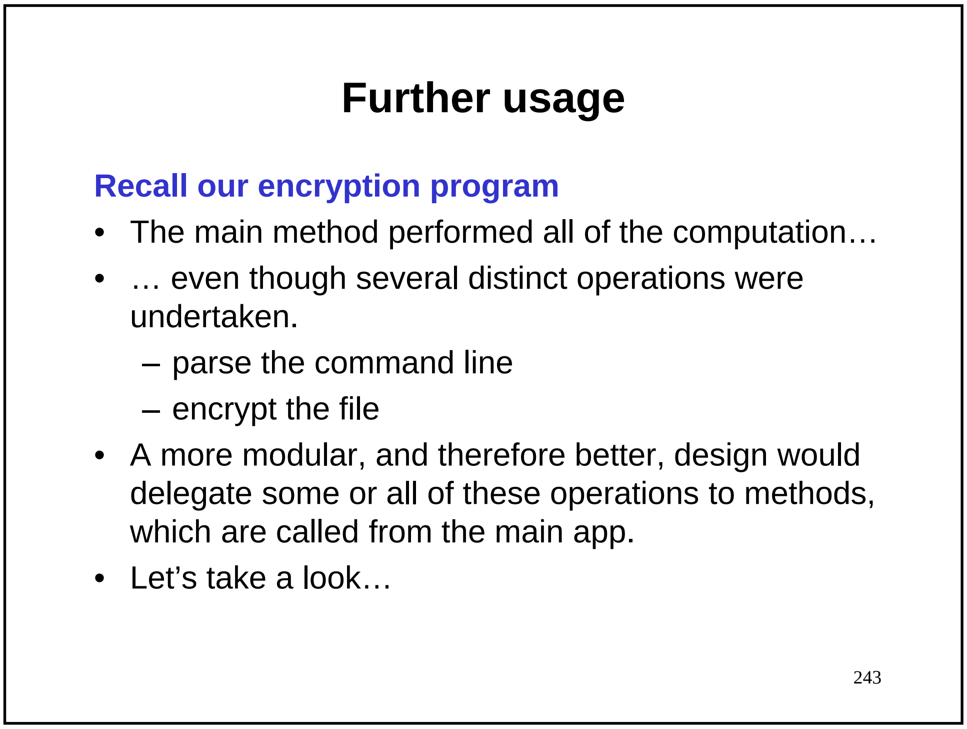


Further usage

Recall our encryption program

- The main method performed all of the computation…
- … even though several distinct operations were undertaken.
  - parse the command line
  - encrypt the file

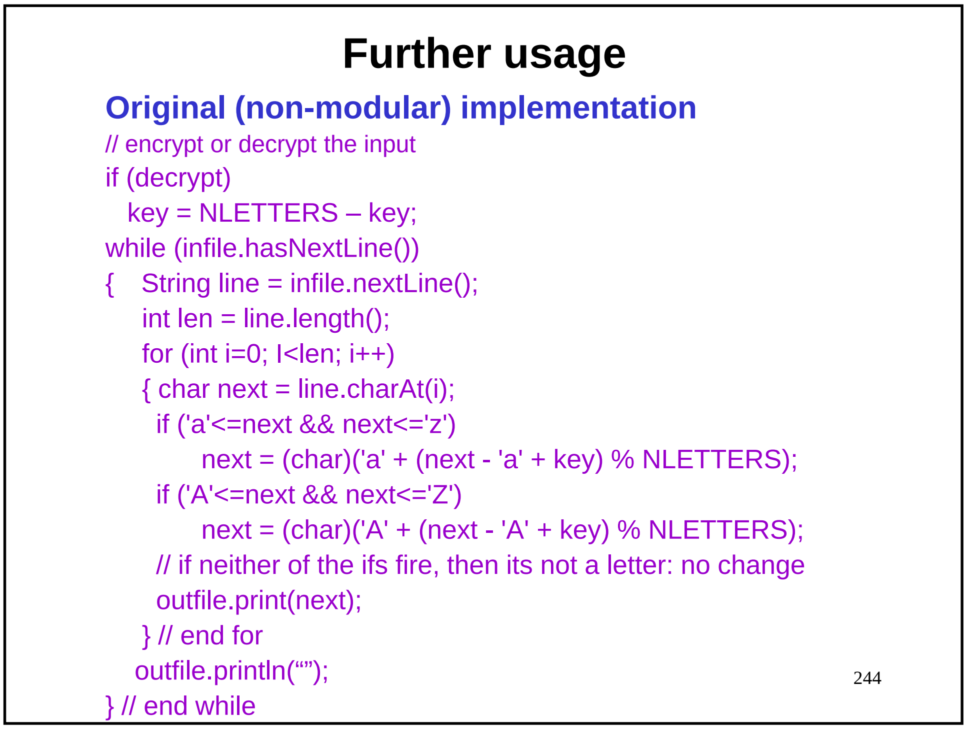
242



Further usage

Recall our encryption program

- The main method performed all of the computation…
- … even though several distinct operations were undertaken.
  - – parse the command line
  - – encrypt the file
- A more modular, and therefore better, design would delegate some or all of these operations to methods, which are called from the main app.
- Let’s take a look…



Further usage

Original (non-modular) implementation

```
// encrypt or decrypt the input
if (decrypt)
   key = NLETTERS – key;
while (infile.hasNextLine())
{    String line = infile.nextLine();
     int len = line.length();
     for (int i=0; I<len; i++)
     { char next = line.charAt(i);
       if ('a'<=next && next<='z')
             next = (char)('a' + (next - 'a' + key) % NLETTERS);
       if ('A'<=next && next<='Z')
             next = (char)('A' + (next - 'A' + key) % NLETTERS);
       // if neither of the ifs fire, then its not a letter: no change
       outfile.print(next);
     } // end for
    outfile.println(“”);
} // end while
```

Further usage

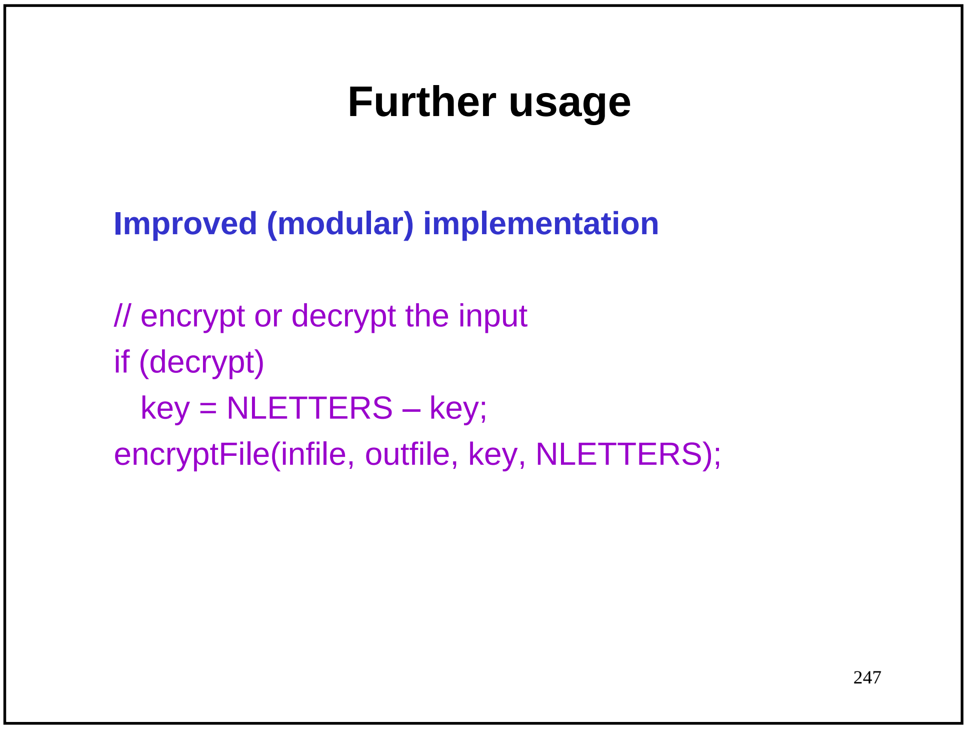
Original (non-modular) implementation

```
// encrypt or decrypt the input
if (decrypt)
  key = NLETTERS – key;
while (infile.hasNextLine())
{    String line = infile.nextLine();
     int len = line.length();
     for (int i=0; I<len; i++)
     { char next = line.charAt(i);
       if ('a'<=next && next<='z')
             next = (char)('a' + (next - 'a' + key) % NLETTERS);
       if ('A'<=next && next<='Z')
             next = (char)('A' + (next - 'A' + key) % NLETTERS);
       // if neither of the ifs fire, then its not a letter: no change
       outfile.print(next);
     } // end for
    outfile.println(“”);
} // end while
```

Further usage

Improved (modular) implementation

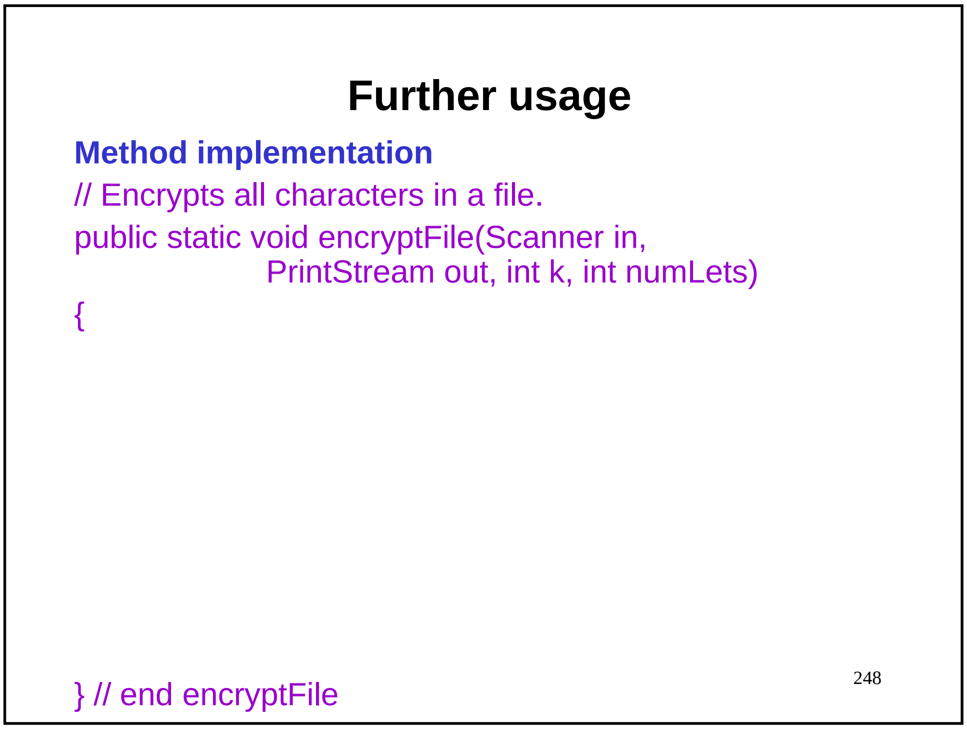
```
// encrypt or decrypt the input
if (decrypt)
  key = NLETTERS – key;
encryptFile(infile, outfile, key, NLETTERS);
```



Further usage

Improved (modular) implementation

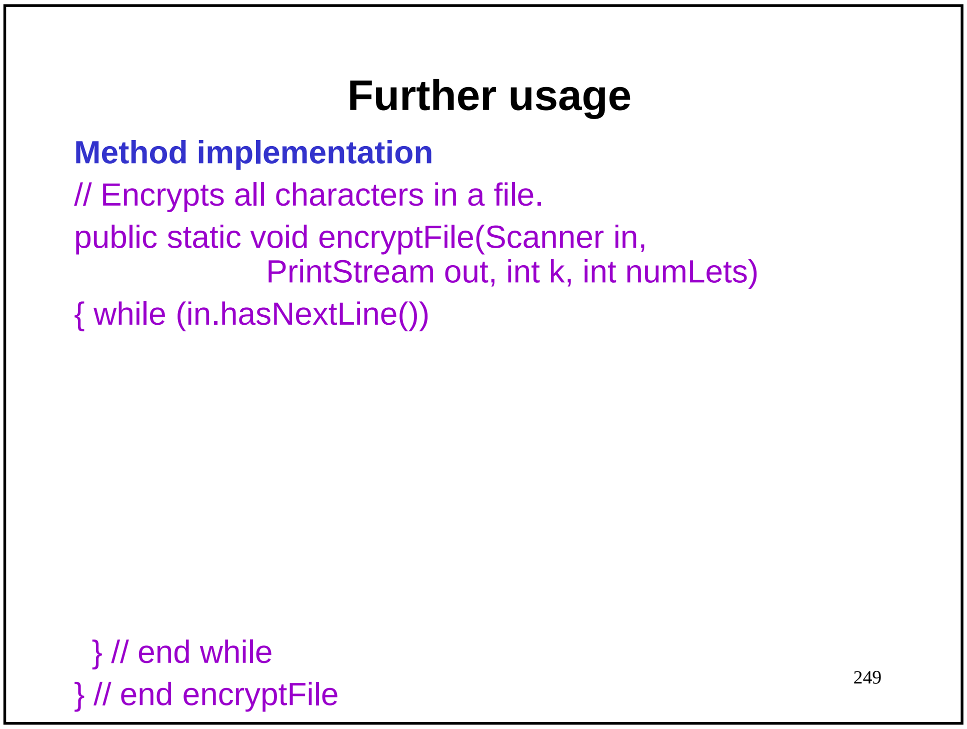
```
// encrypt or decrypt the input
if (decrypt)
   key = NLETTERS – key;
encryptFile(infile, outfile, key, NLETTERS);
```



Further usage

Method implementation

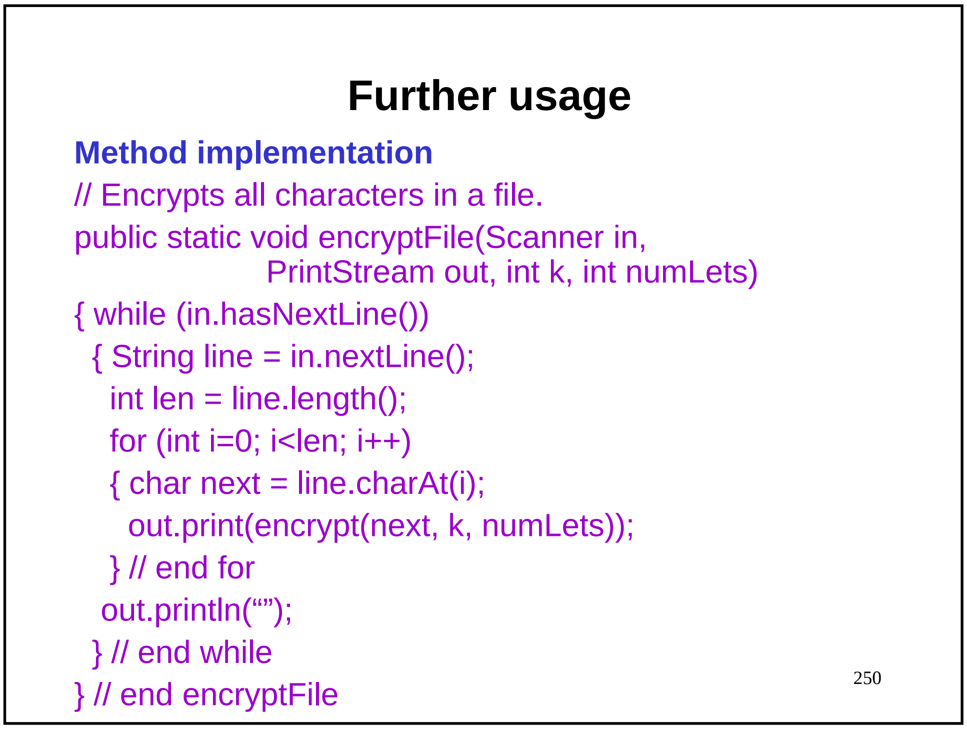
```
// Encrypts all characters in a file.
public static void encryptFile(Scanner in,
                     PrintStream out, int k, int numLets)
{








} // end encryptFile
```



Further usage

Method implementation

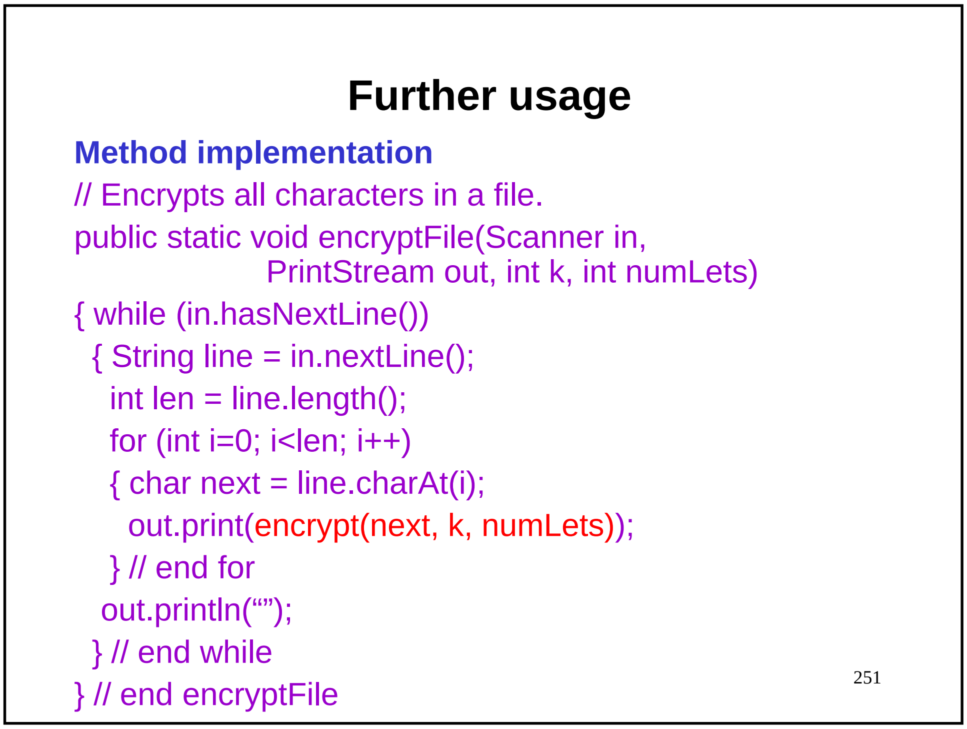
```
// Encrypts all characters in a file.
public static void encryptFile(Scanner in,
                     PrintStream out, int k, int numLets)
{ while (in.hasNextLine())







  } // end while
} // end encryptFile
```



Further usage

Method implementation

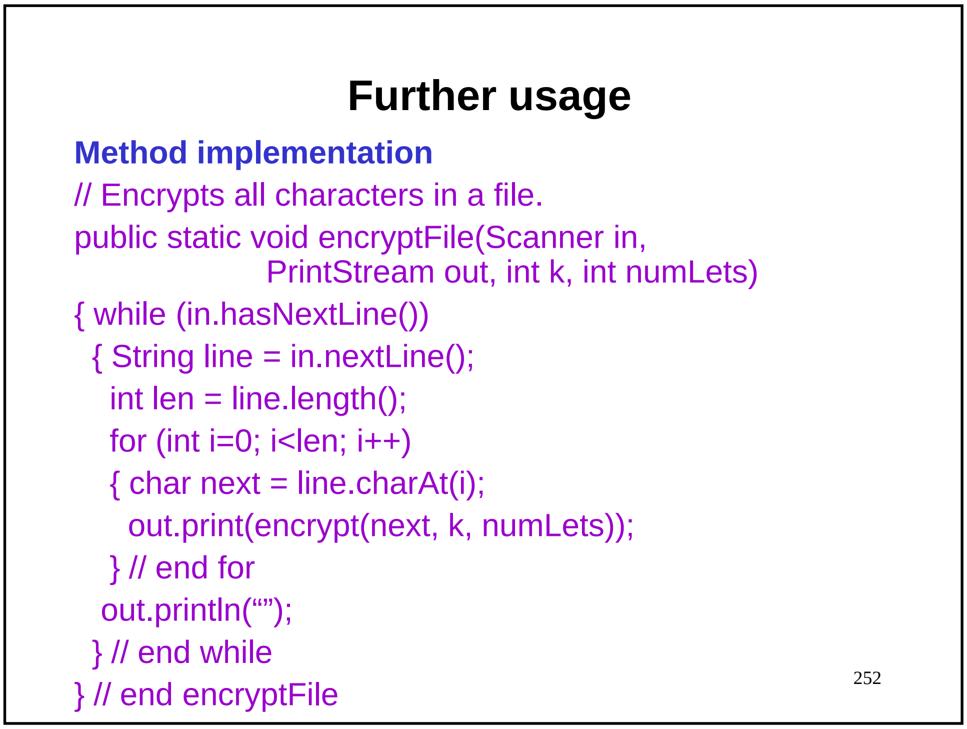
```
// Encrypts all characters in a file.
public static void encryptFile(Scanner in,
                     PrintStream out, int k, int numLets)
{ while (in.hasNextLine())
  { String line = in.nextLine();
    int len = line.length();
    for (int i=0; i<len; i++)
    { char next = line.charAt(i);
      out.print(encrypt(next, k, numLets));
    } // end for
   out.println(“”);
  } // end while
} // end encryptFile
```



Further usage

Method implementation

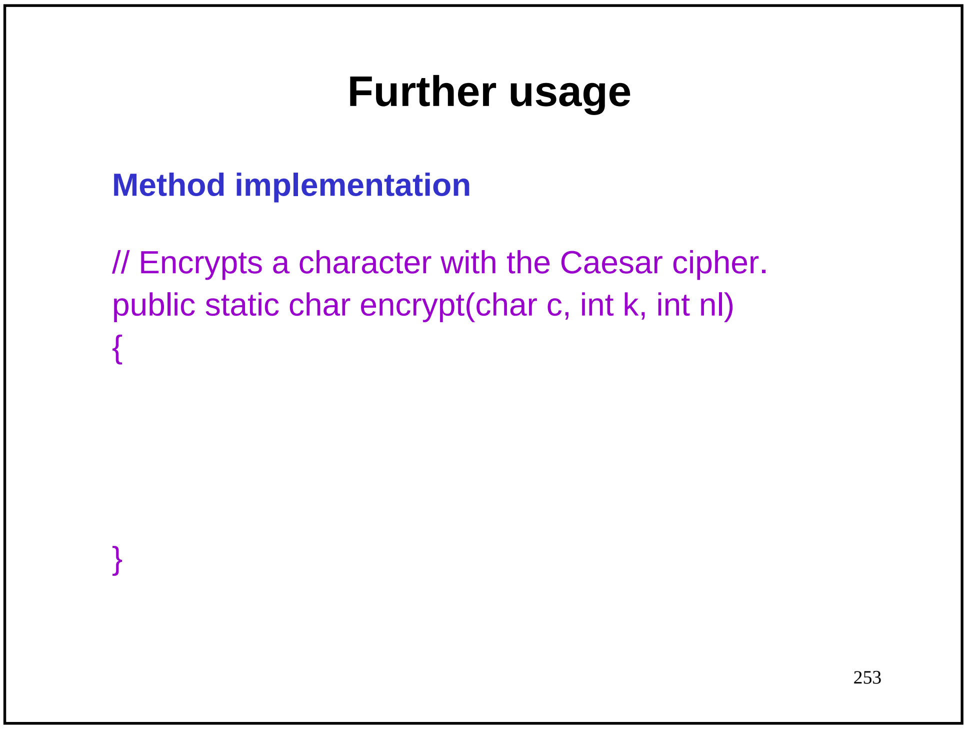
```
// Encrypts all characters in a file.
public static void encryptFile(Scanner in,
                     PrintStream out, int k, int numLets)
{ while (in.hasNextLine())
  { String line = in.nextLine();
    int len = line.length();
    for (int i=0; i<len; i++)
    { char next = line.charAt(i);
      out.print(encrypt(next, k, numLets));
    } // end for
   out.println(“”);
  } // end while
} // end encryptFile
```



Further usage

Method implementation

```
// Encrypts all characters in a file.
public static void encryptFile(Scanner in,
                     PrintStream out, int k, int numLets)
{ while (in.hasNextLine())
  { String line = in.nextLine();
    int len = line.length();
    for (int i=0; i<len; i++)
    { char next = line.charAt(i);
      out.print(encrypt(next, k, numLets));
    } // end for
   out.println(“”);
  } // end while
} // end encryptFile
```

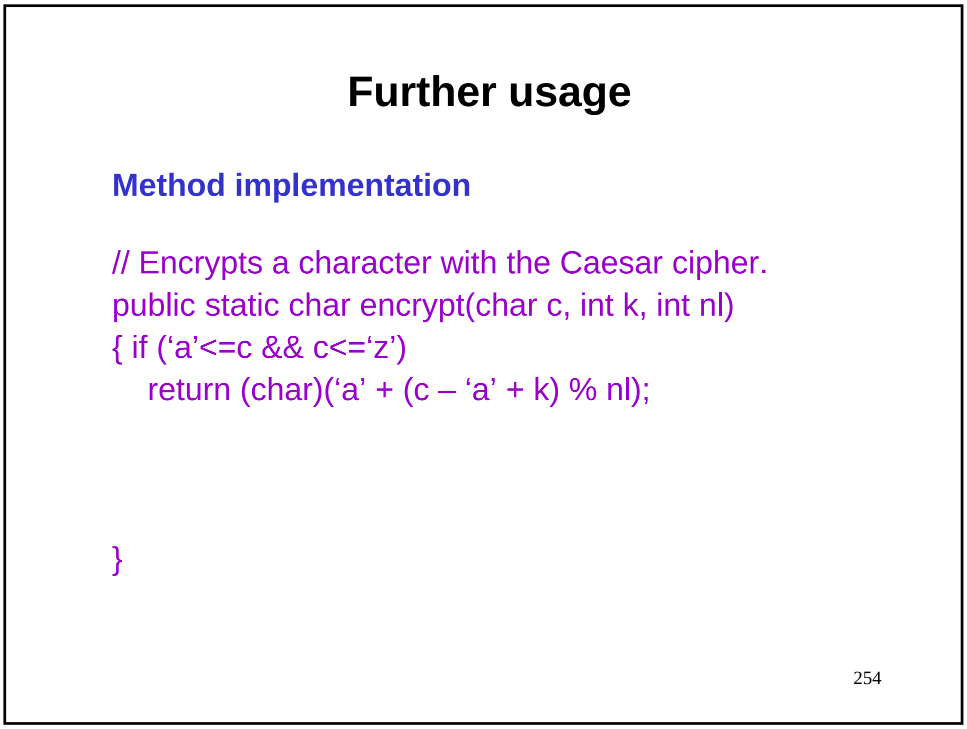


Further usage

Method implementation

```
// Encrypts a character with the Caesar cipher.
public static char encrypt(char c, int k, int nl)
{




}
```

253



Further usage

Method implementation

```
// Encrypts a character with the Caesar cipher.
public static char encrypt(char c, int k, int nl)
{ if (‘a’<=c && c<=‘z’)
    return (char)(‘a’ + (c – ‘a’ + k) % nl);



}
```

254

Further usage

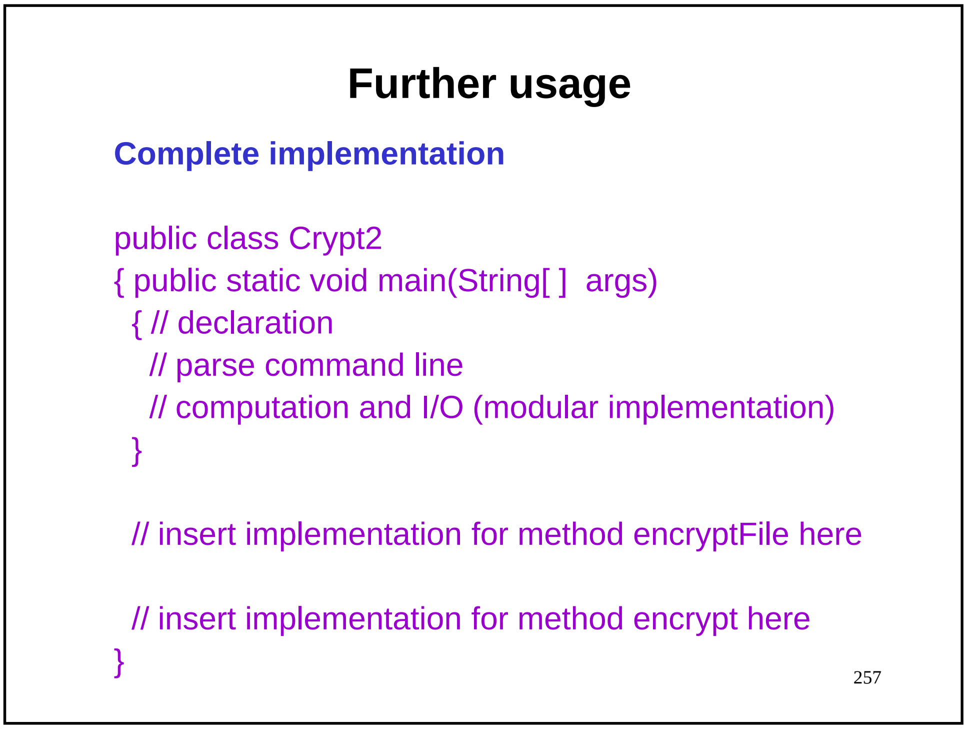
Method implementation

```
// Encrypts a character with the Caesar cipher.
public static char encrypt(char c, int k, int nl)
{ if (‘a’<=c && c<=‘z’)
    return (char)(‘a’ + (c – ‘a’ + k) % nl);
  if (‘A’<=c && c<=‘Z’)
    return (char)(‘A’ + (c – ‘A’ + k) % nl);

}
```

Further usage

Method implementation

```
// Encrypts a character with the Caesar cipher.
public static char encrypt(char c, int k, int nl)
{ if (‘a’<=c && c<=‘z’)
    return (char)(‘a’ + (c – ‘a’ + k) % nl);
  if (‘A’<=c && c<=‘Z’)
    return (char)(‘A’ + (c – ‘A’ + k) % nl);
  return c;
}
```

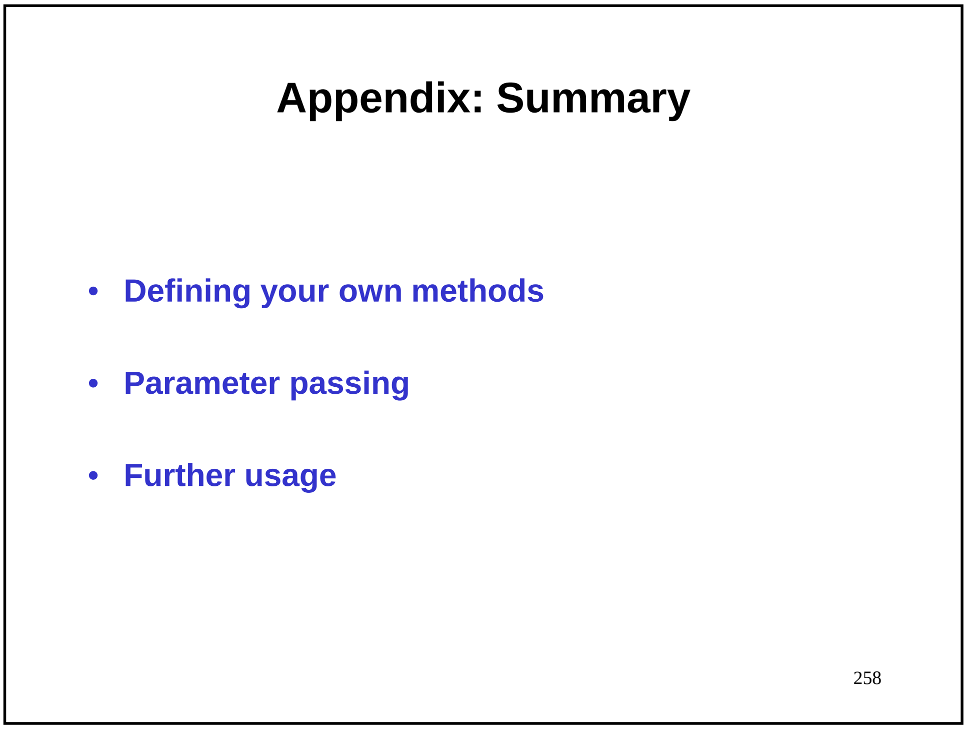


Further usage

Complete implementation

```
public class Crypt2
{ public static void main(String[ ]  args)
  { // declaration
    // parse command line
    // computation and I/O (modular implementation)
  }

  // insert implementation for method encryptFile here

  // insert implementation for method encrypt here
}
```

257



Appendix: Summary

- **Defining your own methods**
- **Parameter passing**
- **Further usage**

258