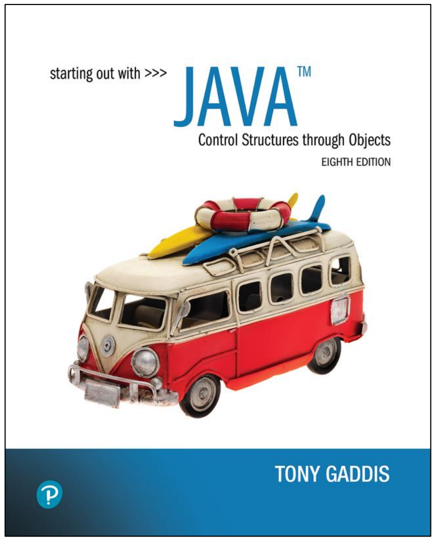
Starting Out with Java Control Structures Through Objects

Eighth Edition



Chapter 11

Exceptions and Advanced File I/O

Chapter Topics

- Chapter 11 discusses the following main topics:
  - Handling Exceptions
  - The Extended try-with-resources Statement
  - Throwing Exceptions
  - Advanced Topics:
    - Binary Files,
    - Random Access Files, and
    - Object Serialization

Handling Exceptions (1 of 5)

- An exception is an unexpected error that occurs while a program is running.
- Exceptions cause the program to halt if the error is not properly dealt with.

```
int[] numbers = { 1, 2, 3 };
System.out.println(numbers[3]);
```

The error message gives information about the exception

```
Exception java.lang.ArrayIndexOutOfBoundsException:
Index 3 out of bounds for length 3
      at (BadArray.main(BadArray.java:15)
```

Handling Exceptions (2 of 5)

- When an exception occurs, an **exception object** is created in memory.
- The exception object contains information about the error that occurred.
- The process of creating an exception object is known as **throwing an exception**.
- When an exception is thrown, the application halts unless the application contains code to handle the exception.

Handling Exceptions (3 of 5)

- Some exceptions are easy to avoid.
- Example: array index out-of-bounds

Bad:

```
void displayElement(int[] numbers, int index)
{
   System.out.println(numbers[index]);
}
```

Better:

```
void displayElement(int[] numbers, int index)
{
   if (index >= 0 && index < numbers.length)
      System.out.println(numbers[index]);
   else
      System.out.println("Invalid index.");
}
```

Handling Exceptions (4 of 5)

- Some exceptions are easy to avoid.
- Example: division by zero

Bad:

```
quotient = number1 / number2;
```

Better:

```
if (number2 != 0)
   quotient = number1 / number2;
else
   System.out.println("Cannot divide by zero!");
```

Handling Exceptions (5 of 5)

- Some exceptions cannot be avoided
  - Example: The `Scanner` class's `nextInt` method expects to read an `int`.
    - If the `nextInt` method reads a value that cannot be stored as an `int`, an exception is thrown.

The `try` Statement (1 of 8)

- You use the `try` statement to handle an exception and prevent the program from crashing.
- General format:

```
try
{
   try block statements...
}
catch (ExceptionType variable)
{
   catch block statements...
}
```

The `try` Statement (2 of 8)

- You use the `try` statement to handle an exception and prevent the program from crashing.
- General format:

```
try
{
   try block statements...
}
catch (ExceptionType variable)
{
   catch block statements...
}
```

The try block contains one or more statements that can potentially throw an exception.

The `try` Statement (3 of 8)

- You use the `try` statement to handle an exception and prevent the program from crashing.
- General format:

```
try
{
   try block statements...
}
catch (ExceptionType variable)
{
   catch block statements...
}
```

If an exception of *ExceptionType* is thrown, the program jumps to the catch clause.

The `try` Statement (4 of 8)

- You use the `try` statement to handle an exception and prevent the program from crashing.
- General format:

```
try
{
   try block statements...
}
catch (ExceptionType variable)
{
   catch block statements...
}
```

The catch parameter will reference the exception object, and the statements in the catch block will execute.

The `try` Statement (5 of 8)

- Example:

```
try
{
   Scanner keyboard = new Scanner(System.in);
   System.out.print("Enter your sales: ");
   double sales = keyboard.nextDouble();
   double commission = sales * COMMISSION_RATE;
   System.out.printf("Your commission: $%,.2f\n", commission);
}
catch(InputMismatchException e)
{
   System.out.println("Enter a valid numeric value.");
}
```

The `try` Statement (6 of 8)

- Example:

If the user enters a nonnumeric value, the `nextDouble()` method will throw an exception of the `InputMismatchException` type.

```
try
{
   Scanner keyboard = new Scanner(System.in);
   System.out.print("Enter your sales: ");
   double sales = keyboard.nextDouble();
   double commission = sales * COMMISSION_RATE;
   System.out.printf("Your commission: $%,.2f\n", commission);
}
catch(InputMismatchException e)
{
   System.out.println("Enter a valid numeric value.");
}
```

The `try` Statement (7 of 8)

- Example:

If the user enters a nonnumeric value, the `nextDouble()` method will throw an exception of the `InputMismatchException` type.

```
try
{
   Scanner keyboard = new Scanner(System.in);
   System.out.print("Enter your sales: ");
   double sales = keyboard.nextDouble();
   double commission = sales * COMMISSION_RATE;
   System.out.printf("Your commission: $%,.2f\n", commission);
}
catch(InputMismatchException e)
{
   System.out.println("Enter a valid numeric value.");
}
```

The program will jump to the `catch` clause and execute the statement in the catch block.

The `try` Statement (8 of 8)

- Example – when no exception is thrown:

If the try block executes without throwing exception, the program jumps to the statement immediately following the try/catch construct.

```
try
{
   Scanner keyboard = new Scanner(System.in);
   System.out.print("Enter your sales: ");
   double sales = keyboard.nextDouble();
   double commission = sales * COMMISSION_RATE;
   System.out.printf("Your commission: $%,.2f\n", commission);
}
catch(InputMismatchException e)
{
   System.out.println("Enter a valid numeric value.");
}
```

When an Exception Is Not Caught (1 of 3)

Suppose a statement in this try block throws a `NumberFormatException`? What happens?

```
try
{
   // Statements
}
catch(InputMismatchException e)
{
   // Code that responds to the exception…
}
```

When an Exception Is Not Caught (2 of 3)

Suppose a statement in this try block throws a `NumberFormatException`? What happens?

```
try
{
   // Statements
}
catch(InputMismatchException e)
{
   // Code that responds to the exception…
}
```

The exception will not be handled because the `catch` clause handles `InputMismatchExceptions`.

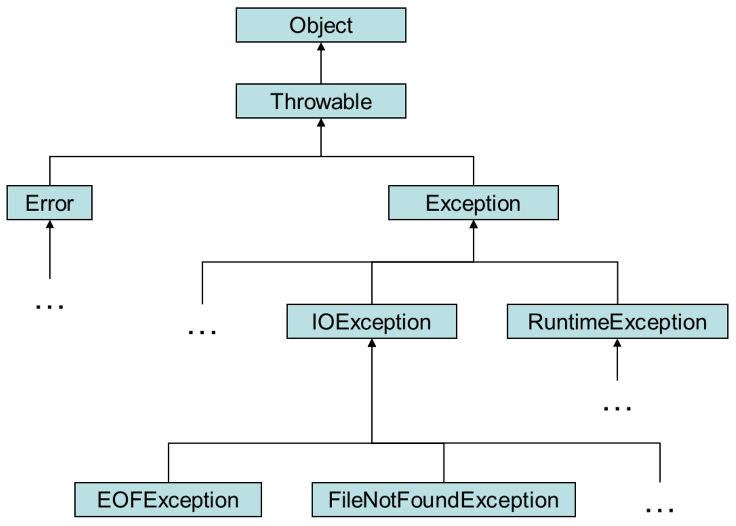
When an Exception Is Not Caught (3 of 3)

- If an exception is thrown in a try block, but the there is no catch parameter of the correct type to receive the exception object, the exception will not be handled.
- The result will be the same as if the `try` statement did not exist.

More About Exception Classes

- Exception objects are created from classes in the Java API.
- All of the exception classes in the hierarchy are derived from the `Throwable` class.
- `Error` and `Exception` are derived from the `Throwable` class.

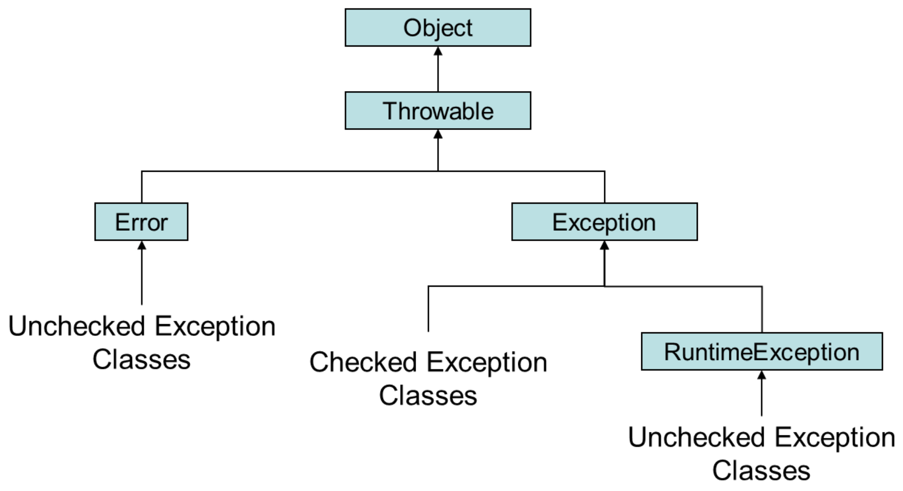
Exception Classes



Checked and Unchecked Exceptions (1 of 3)

- **Checked Exceptions:**
  - Cannot be ignored
  - Caused by unavoidable events (example: file not found)
  - The compiler makes sure your code handles checked exceptions
- **Unchecked Exceptions:**
  - Can typically be prevented (example: division by zero)
  - The compiler does not require you to handle unchecked exceptions

Checked and Unchecked Exceptions (2 of 3)


Object
Throwable
Error
Exception
Unchecked Exception Classes
Checked Exception Classes
RuntimeException
Unchecked Exception Classes

Checked and Unchecked Exceptions (3 of 3)

- All exceptions that are **not** derived from `Error` or `RuntimeException` are checked exceptions
- Unchecked exceptions are those that are derived from the `Error` class or the `RuntimeException` class
  - `RuntimeException` serves as a superclass for exceptions that result from programming errors – these can usually be avoided with properly written code
  - Exceptions derived from `Error` are thrown when a critical error occurs – it's best not to handle these

Handling Unchecked Exceptions

- If a method contains code that can throw a checked exception, the method must meet ONE of these requirements:
  - It must handle the exception with a `try` statement, or
  - It must have a `throws` clause in the method header

Preventing Checked Exceptions

- You are not required to handle checked exceptions or declare them with a `throws` clause
- Checked exceptions are usually preventable:
  - Make sure array subscripts are within range
  - Do not divide by zero
  - Make sure files exist before opening them for reading
  - etc.

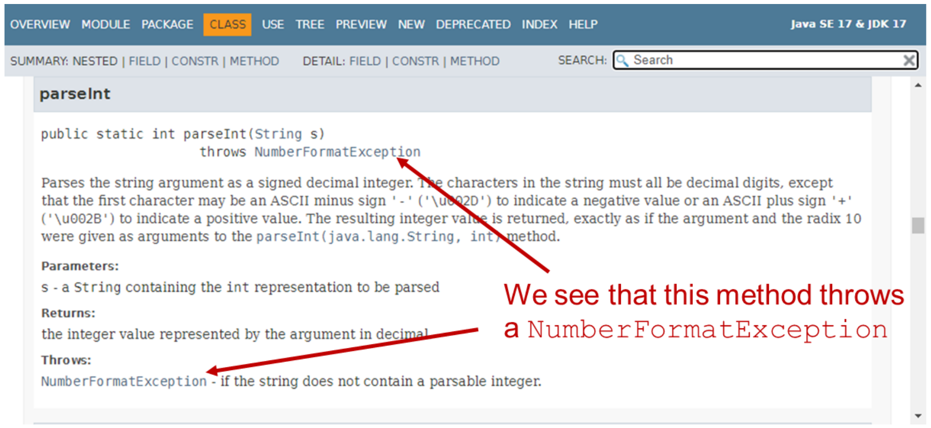
Exceptions and Packages

- The exception classes in the Java API are organized in packages.
  - `ArrayIndexOutOfBoundsException` is in the `java.lang` package
  - `InputMismatchException` is in the `java.util` package
  - `FileNotFoundException` is in the `java.io` package
- To handle an exception that is not in `java.lang`, you will need the appropriate `import` statement in your program
  - Example:

```
import java.util.InputMismatchException;
```

Exceptions and the Java API Documentation (1 of 2)

- The official Java API documentation is your best resource for discovering which exceptions are thrown by the API methods


OVERVIEW MODULE PACKAGE CLASS USE TREE PREVIEW NEW DEPRECATED INDEX HELP
Java SE 17 & JDK 17
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD
SEARCH: Search

parseInt

```
public static int parseInt(String s)
                    throws NumberFormatException
```

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 or an ASCII plus sign `'+'` (`'\u002B'`) to indicate a positive 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

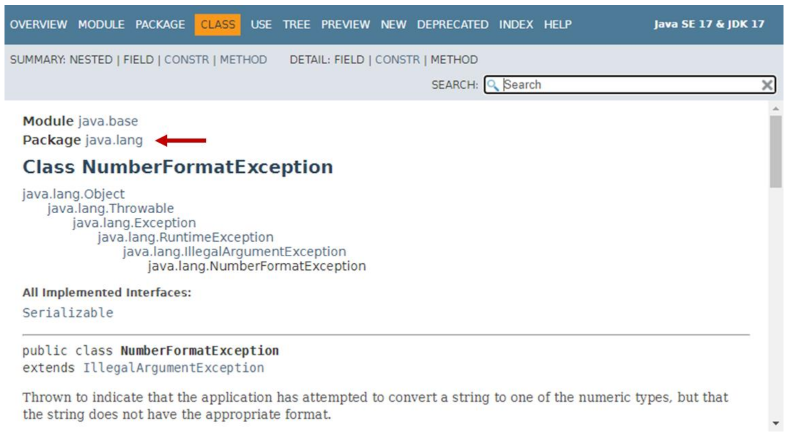
Returns:
the integer value represented by the argument in decimal

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

We see that this method throws a `NumberFormatException`

Exceptions and the Java API Documentation (2 of 2)

- Click the name of an exception to see its documentation and learn the name of its package


OVERVIEW MODULE PACKAGE CLASS USE TREE PREVIEW NEW DEPRECATED INDEX HELP
Java SE 17 & JDK 17
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD
SEARCH: Search

Module java.base
Package java.lang

Class NumberFormatException

```
java.lang.Object
    java.lang.Throwable
        java.lang.Exception
            java.lang.RuntimeException
                java.lang.IllegalArgumentException
                    java.lang.NumberFormatException
```

All Implemented Interfaces:
`Serializable`

```
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.

The Stack Trace

- The **call stack** is an internal list of all the methods that are currently executing.
- A **stack trace** is a list of all the methods in the call stack.
- It indicates:
  - the method that was executing when an exception occurred and
  - all of the methods that were called in order to execute that method.
- Example: ForceError.java

Passing Uncaught Exceptions Up the Stack Trace

- When code in a method throws an exception, the normal execution of the method stops
- If the exception is not handled with a try statement, control is passed to the previous method in the call stack
- If that method does not handle the exception, control is passed up the call stack to the previous method
- This continues until control reaches the `main` method
- If `main` does not handle the exception, the program is halted

Retrieving the Default Error Message

- Each exception object has a `getMessage` method returns the exception's default error message as a string.

```
String str = "abcde";

try
{
  System.out.println("Converting " + str + " to an int.");
  int number = Integer.parseInt(str);
}
catch (NumberFormatException e)
{
  System.out.println("Conversion error: " +
                     e.getMessage());
}
```

The Scope of Variables Declared in a `try` Statement

- If you declare a variable inside a try block or a catch block, no statement outside that block will be able to access the variable
- If you need to access a variable in both the try block and a catch block, you must declare the variable outside the `try/catch` statement

Handling Exceptions Polymorphically (1 of 2)

- When handling exceptions, you can use a polymorphic reference as a parameter in the `catch` clause.
- Most exceptions are derived from the `Exception` class.
- A `catch` clause that uses a parameter variable of the `Exception` type is capable of catching any exception that is derived from the `Exception` class.

Handling Exceptions Polymorphically (2 of 2)

```
try
{
   number = Integer.parseInt(str);
}
catch (Exception e)
{
   System.out.println("The following error occurred: "
                     + e.getMessage());
}
```

- The `Integer` class's `parseInt` method throws a `NumberFormatException` object.
- The `NumberFormatException` class is derived from the `Exception` class.

Using Multiple `catch` Clauses to Handle Multiple Exceptions (1 of 2)

- A `try` statement can have multiple `catch` clauses
- This is useful when the try block contains code that can throw more than one type of exception

```
try
{
   try block statements . . .
}
catch (ExceptionType1 variable)
{
   catch block statements . . .
}
catch (ExceptionType2 variable)
{
   catch block statements . . .
}
catch (ExceptionType3 variable)
{
   catch block statements . . .
}
```

Using Multiple `catch` Clauses to Handle Multiple Exceptions (2 of 2)

```
try
{
   statement
   statement
   statement
}
catch (InputMismatchException e)
{
   System.out.println("You have entered invalid input.");
}
catch (NumberFormatException e)
{
   System.out.println("Invalid number format.");
}
catch (IndexOutOfBoundsException e)
{
   System.out.println("Invalid index.");
}
```

When an exception is thrown, the `catch` clauses are searched from top to bottom

The Order of Multiple `catch` Clauses (1 of 3)

- When catching multiple exceptions that are **related to each other through inheritance**, the order of the `catch` clauses is important
- The `catch` clauses must appear in the order of most specific exception classes first, and most general exception classes last
- In other words, you must catch exception subclasses before you can catch their superclasses.

The Order of Multiple `catch` Clauses (2 of 3)

- The following code has an error because `NumberFormatException` is a subclass of `IllegalArgumentException`

```
try
{
   number = Integer.parseInt(str);
}
catch (IllegalArgumentException e)
{
   System.out.println("Bad number format.");
}
catch (NumberFormatException e)
{
   System.out.println(str + " is not a number.");
}
```

`NumberFormatException` must be caught before `IllegalArgumentException`

The Order of Multiple `catch` Clauses (3 of 3)

- This is the correct order of the `catch` clauses

```
try
{
   number = Integer.parseInt(str);
}
catch (NumberFormatException e)
{
   System.out.println(str + " is not a number.");
}
catch (IllegalArgumentException e)
{
   System.out.println("Bad number format.");
}
```

Handle Each Exception Only Once in a `try` Statement

- A `try` statement can have only one `catch` clause for each specific type of exception

```
try
{
   number = Integer.parseInt(str);
}
catch (NumberFormatException e)
{
   System.out.println(str + " is not a number.");
}
catch (NumberFormatException e)
{
   System.out.println("Bad number format.");
}
```

Error

Handling Multiple Exceptions with One `catch` Clause

- You can specify more than one exception in a `catch` clause:

```
try
{
}
catch(NumberFormatException | InputMismatchException ex)
{
}
```

Separate the exceptions with the `|` character.

Pearson

The `finally` Clause (1 of 2)

- The `try` statement may have an optional `finally` clause.
- If present, the `finally` clause must appear after all the `catch` clauses.

```
try
{
   (try block statements...)
}
catch (ExceptionType ParameterName)
{
   (catch block statements...)
}
finally
{
   (finally block statements...)
}
```

Pearson

The `finally` Clause (2 of 2)

- The **finally block** is one or more statements,
  - that are always executed after the try block has executed and
  - after any catch blocks have executed if an exception was thrown.
- The statements in the finally block execute whether an exception occurs or not

Pearson

The Extended `try`-with-resources Statement (1 of 3)

- The `try`-with-resources statement can optionally have `catch` clauses and a `finally` clause
- This means that in addition to managing the opening and closing of a resource, it can handle exceptions that occur in the try block
- When a `try`-with-resources statement has a `catch` clause and/or a `finally` clause, it is known as an extended `try`-with-resources statement

Pearson

The Extended `try`-with-resources Statement (2 of 3)

- General format of a `try`-with-resources statement with two `catch` clauses:

```
try (Declaration statements for auto closeable objects)
{
   try block statements . . .
}
catch (ExceptionType1 variable)
{
   catch block statements . . .
}
catch (ExceptionType2 variable)
{
   catch block statements . . .
}
```

If a statement inside the try block throws an exception, the resources that were declared inside the parentheses are automatically closed.

The Extended `try`-with-resources Statement (3 of 3)

- General format of a `try`-with-resources statement with a `catch` clause and a `finally` clause:

```
try (Declaration statements for auto closeable objects)
{
   try block statements . . .
}
catch (ExceptionType variable)
{
   catch block statements . . .
}
finally
{
   finally block statements . . .
}
```

Handling `IOException` and its Subclasses (1 of 2)

- When working with files, you will need to handle exceptions of the `IOException` class, or one of its subclasses

Common File-Related Checked Exceptions

Exception Class	Package	Description
`IOException`	`java.io`	Indicates some sort of error occurred during an input or output operation.
`FileNotFoundException`	`java.io`	A subclass of `IOException`. Usually thrown by a class constructor (such as `PrintWriter` or `Scanner`) when a specified file is not found.
`EOFException`	`java.io`	A subclass of `IOException`. Indicates that the end of a file was unexpectedly reached during an input operation.

Handling `IOException` and its Subclasses (2 of 2)

Example from ReadFile.java

```
try (Scanner inFile = new Scanner(new File(filename)))
{
   while (inFile.hasNext())
   {
      input = inFile.nextLine();
      System.out.println(input);
   }
}
catch (FileNotFoundException e)
{
   System.out.println("That file is not found.");
}
```

See also SalesReport.java

Throwing Exceptions (1 of 2)

- You can write code that:
  - throws one of the standard Java exceptions, or
  - an instance of a custom exception class that you have designed.
- The `throw` statement is used to manually throw an exception.

```
throw new ExceptionType(MessageString);
```

- The `throw` statement causes an exception object to be created and thrown.

Pearson

Throwing Exceptions (2 of 2)

- The **MessageString** argument contains a custom error message that can be retrieved from the exception object's `getMessage` method.
- If you do not pass a message to the constructor, the exception will have a null message.

```
throw new Exception("Out of fuel");
```

  - **Note: Don't confuse the `throw` statement with the `throws` clause.**
- Example: DieExceptionDemo.java

Pearson

Creating Exception Classes (1 of 2)

- You can create your own exception classes by deriving them from the `Exception` class or one of its derived classes.
- See these examples:
  - BankAccount.java
  - NegativeStartingBalance.java
  - AccountTest.java

Pearson

Creating Exception Classes (2 of 2)

- Some examples of exceptions that can affect a bank account:
  - A negative starting balance is passed to the constructor.
  - A negative interest rate is passed to the constructor.
  - A negative number is passed to the deposit method.
  - A negative number is passed to the withdraw method.
  - The amount passed to the withdraw method exceeds the account's balance.
- We can create exceptions that represent each of these error conditions.

Pearson

`@exception` Tag in Documentation Comments

- General format

  `@exception` **ExceptionName Description**

- The following rules apply
  - The `@exception` tag in a method's documentation comment must appear after the general description of the method.
  - The description can span several lines. It ends at the end of the documentation comment (the `*/` symbol) or at the beginning of another tag.

Copyright