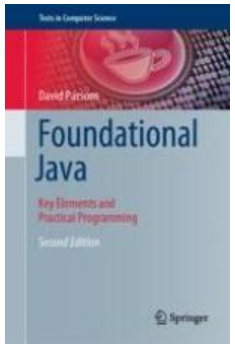


Chapter 11
Exploring the Java Libraries

Foundational Java

Key Elements and Practical Programming

Library Classes

- Reuse existing classes rather than re-inventing the wheel
- Object oriented programming offers reuse of existing classes
  - Save time by not having to implement the code.
  - Library code has already been extensively tested
- Use the Javadoc to find classes to reuse in JRE
- Other sources
  - commercial or open source projects
- Art of Java is assembling components from other sources

Frequently Used Classes in java.lang

- java.lang.Object
- java.lang.Math
- java.lang.System
- java.lang.Class
- Wrapper classes - java.lang.Integer etc...

The java.lang.Object Class

- As the root of the class hierarchy, the Object class is a superclass for all other classes
  - Every class inherits the methods that are defined in the Object class
- The Object class defines the default behavior for all objects through methods like:
  - equals(java.lang.Object) // returns a boolean
  - getClass() // returns a Class object
  - toString() // returns a String representation of the object
  - hashCode() // returns an integer for indexing hash tables
- 'wait', 'notify' and 'notifyAll' relate to multithreading
- Only other methods on the Object class are 'finalize' and 'clone'.

```
protected void finalize() throws Throwable
protected Object clone() throws CloneNotSupportedException
```

The ‘finalize’ Method

- Called on an object if it is garbage collected
- No guarantee that a given object will be garbage collected
  - no guarantee that this method will ever be called
- Provides an opportunity for an object to release any resources that it may be holding before it is disposed of
- ‘finalize’ has ‘protected’ visibility on the Object class
  - Not automatically available as public methods of subclasses

The ‘clone’ Method

- ‘clone’ makes a shallow copy of the current object
  - By default it only makes a shallow copy

```
Object object2 = object1;      // by default is equal to
Object object2 = object2.clone();
```

- ‘clone’ can be overridden to give a different behaviour
  - Override ‘clone’ to provide a deep copy of an object
  - Original object and copy are independent
- The basic implementation should always return ‘super.clone’.

```
@Override
public Object clone() throws CloneNotSupportedException
{
    return super.clone();
}
```

The Cloneable Interface

- The class being cloned must also implement the 'Cloneable' interface
  - Otherwise the CloneNotSupportedException will be thrown.

```
public class CloneExample implements Cloneable {…
```

- Implementing 'clone'

```
@Override
public Object clone() throws CloneNotSupportedException {
  CloneExample clone = (CloneExample)super.clone();
  int[] clonedArray = getArray();
  int[] copiedArray = new int[clonedArray.length];
  for(int i = 0; i < clonedArray.length; i++) {
    copiedArray[i] = clonedArray[i];
  }
  clone.setArray(copiedArray);
  return clone;
}
```

Exercise 11.1

- Override the ‘clone’ method for the Course class so that it makes a deep copy of its array of Modules
- Write a JUnit test case that shows that your clone method is, in fact, making a deep copy
  - This will involve cloning a course, changing the modules of the original course and testing that the clone remains unchanged

The java.lang.Math Class

- All the methods in the Math class are static methods
- These methods allow a user to construct and evaluate mathematical expressions
  - Work with overloaded data types or assume double parameters and return types

Method	Usage	Example
double Math.pow(double x, double y)	Returns the value of x raised to the power of y	Math.pow(2,3) // 2^3 = 8
double Math.ceil(double x)	Returns the smallest integer greater than or equal to x	Math.ceil(5.2) // = 6
double Math.sqrt(double x)	Returns the square root of x	Math.sqrt(9) // = 3

Math Class Constants

- The Math class also includes two public constants

```
public static final double E;   // the base of the natural logarithms
public static final double PI;  // the ratio of the circumference of a circle to its diameter
```

- Invoke using the class:

```
Math.PI
Math.E
```

From the Javadoc

java.lang.Math

public static final double	E	2.71828182845904 5
public static final double	PI	3.14159265358979 3

Exercise 11.2

- Use the Math.pow and Math.sqrt methods to calculate the hypotenuse (longest side) of a right-angled triangle
- According to Pythagoras, the square of the hypotenuse is equal to the sum of the squares of the other two sides
- Your code needs to:
  - Calculate the squares of the two shorter sides
  - Add these squares together
  - Find the square root of this value; this will be the length of the longest side
- Use a ‘test first’ approach
  - Begin by writing a JUnit test case that expects a correct answer
  - e.g. the hypotenuse of a right-angled triangle with side lengths of 12 and 5 is 13
  - Once you have written the tests, write the unit under test

The java.lang.System Class

- Like the Math class, all the methods in the System class are static methods
- These methods provide platform-independent access to underlying system functions
- The System class also has static fields
  - ‘in’, ‘out’, and ‘err’ represent standard input, standard output, and standard error output respectively

Using System.err

- ‘try’ block uses ‘System.out’
- ‘catch’ block uses ‘System.err’
- System.err is the default for stack traces

```
public static void main(String[] args) {
    try  {
      System.out.println("About to do some arithmetic");
      int x = 1;
      int y = x/0;
    }
    catch(ArithmeticException e)  {
      System.err.println("Oh dear...");
      e.printStackTrace(System.err);
    }
  }
```

Wrapper Classes

- For each of the primitive data types there exists a corresponding class
  - Byte, Short, Character, Integer, Long, Float, Double, Boolean
- This allows Java to construct an object whose state reflects the value of a given primitive data type
  - The object serves to “wrap” the primitive data type
- Wrapper classes have various fields and methods appropriate to their types

```
Boolean aBoolean = Boolean.TRUE;
aBoolean.equals(new Boolean(true));          // true
```

```
Character aCharacter = new Character('c');
aCharacter.isDigit();                        // false
```

Data Conversion With Wrapper Classes

- Wrapper classes can be used to convert strings to numbers
  - The number classes have static ‘parse…’ methods that convert in one step
  - e.g. the Integer class has a parseInt method

```
int year = Integer.parseInt("1066");
```

Wrappers and Collections

- Java collection classes such as ArrayList can only hold objects
- If you want to store a particular primitive data type in a collection, the primitive must be put into a wrapper object before being added to it:

```
int myInt = 25;                                  // cannot be added to a Java collection
Integer myInteger = new Integer(myInt);          // myInteger can be added to a collection
```

- Can be done automatically using ‘autoboxing’
- Wrapper classes have overloaded constructors that allow objects to be created from different types of data

```
Integer int1 = new Integer(42);
Integer int2 = new Integer("42");
```

Classes in the java.util Package

- This package contains utility classes
- It includes the Collection classes that we will look at later
- It also includes the classes

  `Locale`

  `Currency`

- Unlike `java.lang`, classes from `java.util` must be explicitly imported

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

Factory Methods

- In the code examples that follow, there are several examples of factory methods
- Factory methods are static
- A factory method is a common design pattern for creating objects without calling a constructors - they returning a newly created object based on some criteria
- A good example of this is where the configuration of a newly created object can vary depending on the international ‘locale’ in which the object is being created

The Locale Class

- Many elements of data vary according to their international locale
  - Language
  - Character set
  - Datelines
  - Time zones
  - Number and currency format
- This means that some operations in a program are “locale-sensitive”
  - The role of the Locale class is to support the internationalization of applications.
- There a several different ways of creating a Locale object, but the simplest is to use the “forLanguageTag” factory method
  - The possible set of language tags is defined in the IETF BCP 47 standard
  - The language tag for United States English for example is “en-US”.

The Locale Class

- This example creates a Locale for the United States using the “forLanguageTag” method to create a Locale for that region.

```
Locale USLocale = Locale.forLanguageTag("en-US");
```

- The Locale class provides several constants that can be used instead of having to create a Locale object
- There are constants based on both country and language
  - Locale.FRANCE is the constant for the country of France
  - Locale.FRENCH is the constant for the French language

The Currency Class

- The Currency class represents information about international currencies
- Create a Currency object with the “getInstance” method and a Locale object

```
Locale USLocale = Locale.forLanguageTag("en-US");
Currency USCurrency = Currency.getInstance(usLocale);
```

- Alternatively, we can use one of the Locale constants, for example

```
Currency UKCurrency = Currency.getInstance(Locale.UK);
```

- The information available from a Currency object includes several items of data

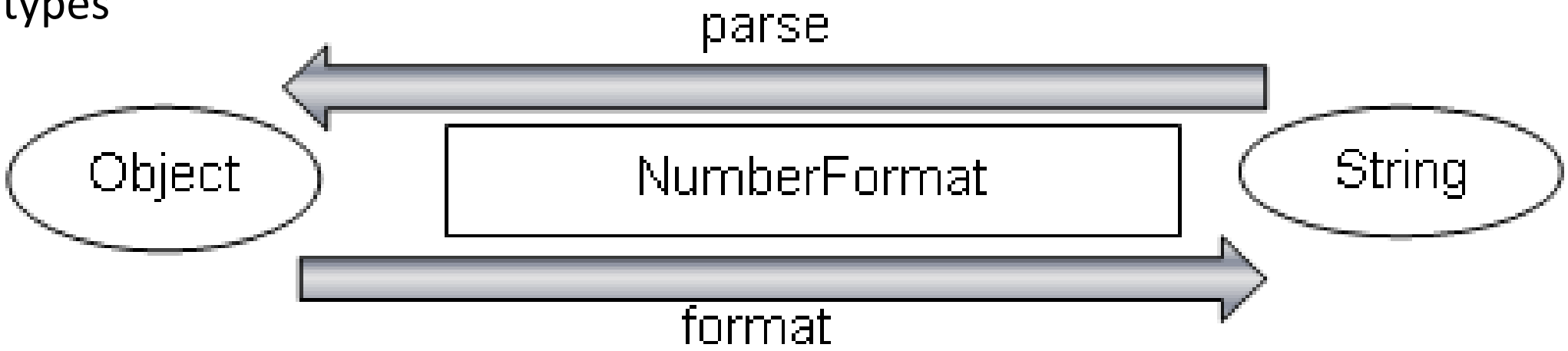
```
System.out.println(UKCurrency.getNumericCode());     // 826
System.out.println(UKCurrency.getDisplayName());     // British Pound
System.out.println(UKCurrency.getSymbol());          // £
System.out.println(UKCurrency.getCurrencyCode());    // GBP
```

Exercise 11.3

- Create a “Money” class that can represent an amount of money in a specific currency.
- Use the services of the Currency class to help implement your Money class.
- Add overloaded static “getInstance” factory methods to your class, one that takes one argument (and amount of money) and defaults to the current locale and another that that takes a Locale object as a second argument.
- Add a “toString” method to your Money class that returns the amount of money using the currency symbol at the front and the currency code at the end, for example:
  - $250USD
- Write a “main” method that creates Money objects from different locales and displays their amounts.

NumberFormat Classes in the “java.text” Package

- The NumberFormat classes in the “java.text” package provide a tool for formatting numbers as Strings and converting Strings to numbers.
- These classes include “format” methods that make it easy to customize number formatting
- The same classes have “parse” methods to convert Strings to objects or primitive types



Formatting and Parsing Numbers

- Create a NumberFormat object from the “getNumberInstance” factory method

```
NumberFormat numFormat = NumberFormat.getNumberInstance();
```

- Has default formats
  - default number of decimal places
  - will round the result

```
String s1 = numFormat.format(1234.56789);          // "1,234.568”
```

- Another default behaviour is to remove trailing zeros

```
String s2 = numFormat.format(1234.00);             // "1,234"
```

Number Formats

- Format behaviour can be configured
- e.g. specify number of digits after the decimal point using the “setMaximumFractionDigits” method

```
numFormat.setMaximumFractionDigits(2);
s1 = numFormat.format(1234.56789); // "1,234.57”
```

- The format methods of the NumberFormat class need to be passed something that can be formatted as a number
- If the argument cannot be formatted then a java.lang.IllegalArgumentException will be thrown at runtime.

Formatting Currency

- A special currency instance of the NumberFormat class can be created to enable the formatting and parsing of values that represent currency

```
NumberFormat dollarFormat = NumberFormat.getCurrencyInstance();
```

- The factory method ‘getCurrencyInstance’ uses your default locale to determine the type of currency and the format of the output
  - e.g. format a double unto a currency String (in a dollar locale).

```
double value = 1234.5;
System.out.println(dollarFormat.format(value)); // ”$1,234.50”
```

Specifying Currencies

- If you want to format currencies from specific locales, these can be passed as an argument to the “getCurrencyInstance” factory method.
- e.g. Germany (Euro)

```
NumberFormat euroFormat = NumberFormat.getCurrencyInstance(Locale.GERMANY);
System.out.println(euroFormat.format(1234.0));          // ”1.234,00 €”
```

- e.g. Japan (Yen)

```
NumberFormat yenFormat  = NumberFormat.getCurrencyInstance(Locale.JAPAN);
System.out.println(yenFormat.format(1234.0));           // ”¥1,234.00"
```

Parsing Numbers

- ‘parse’ methods of NumberFormat class parse Strings into numbers
- Return instances of the Number class
  - Superclass of the number wrapper classes such as Integer and Double
- Has various methods to return primitive numbers
  - ‘byteValue’, ‘doubleValue’, ‘intValue’ etc.
- Truncation or rounding is possible and “ParseException” may be thrown

```
NumberFormat numParser = NumberFormat.getNumberInstance();
try {
  Number num = numParser.parse("1234.5");
  System.out.println(num.doubleValue());       // 1234.5
  System.out.println(num.intValue());          // 1234
}
catch (ParseException e) {…}
```

Parsing Currencies

- Strings representing currency values can also be parsed into Number objects, which can then be used to return primitive values
- e.g. a String in the UK locale currency format
  - Parsed to a Number and converted to a double

```
NumberFormat parseCurrency = NumberFormat.getCurrencyInstance(Locale.UK);
Try {
  // must be a parseable string in the currency of the current Locale
  var currencyValue = parseCurrency.parse("£5,432.10").doubleValue();
  System.out.println(currencyValue);              // 5432.1
}
catch (ParseException e) {…}
```

Exercise 11.4

- Add a method to the BankAccount class (the one you created in Exercise 9.3) to return a formatted balance
- Use a currency instance of the NumberFormat class

Exercise 11.5

- Update your Money class so the “toString” method uses a currency instance of the NumberFormat class to format the amount.
- Append the currency code to the end.

Exercise 11.6

- In your BankAccount class, replace the double field that represents the balance of the account with a java.math.BigDecimal.
  - Use the Javadoc to find out how to use this class in your code so that the methods still work.
- Write a JUnit test case for your BankAccount class.

Dates and Times in the “java.time” Package

- The java.util.Date class has not been deprecated but since Java 8 there has been a newer package called “java.time” that contains a new set of date and time related classes
- The classes in this package are thread safe, with more consistent APIs that follow a functional programming style
- They make extensive use of factory methods, creating immutable objects with methods that frequently return modified copies of themselves so method calls can be chained together
- We will look at the basic features of
  - LocalDate
  - LocalTime
  - LocalDateTime

The LocalDate Class

- LocalDate returns the current date in the local time zone
- A LocalDate contains date-related fields such as “day-of-year”, “day-of-week” and “week-of-year”.
- A LocalDate can be created using the “now” factory method.

```
LocalDate today = LocalDate.now();
System.out.println(today);
```

  - By default, “toString” returns the date in YYY-MM-DD format, for example 2020-04-01
- A LocalDate is immutable and value-based
  - LocalDate objects should be compared using the “equals” method, not the “==” operator.
- Methods can create modified copies of the date, for example “minusDays”

```
LocalDate yesterday = LocalDate.now().minusDays(1);
```

Setting the Date

- Dates can be created for a specific date by passing year, month and day arguments to the “of” method.
- The month can be passed as an integer in the range 1 – 12 or one of the enums (enumerated types) defined in the Month class that use the names of the months rather than their numbers.

```
LocalDate dateOfBirth = LocalDate.of(2012, Month.JANUARY, 1);
```

- Further methods can be called to create modified copies of the immutable date

```
LocalDate twentyFirst = dateOfBirth.plusYears(21);
```

The LocalTime Class

- LocalTime represents a time with no date information
- It has a “now” method that creates an object representing the time of its creation and an “of” method that allows its time to be set using arguments, for example:

```
LocalTime time = LocalTime.now();
LocalTime specifiedTime = LocalTime.of(12,15);
```

- The “of” method used above specifies the hours and minutes, but there are other versions that take different numbers of arguments
- The output from the example LocalTime objects
  - the current time includes the seconds and nanoseconds):

```
20:10:37.441414200
12:15
```

The LocalDateTime Class

- LocalDateTime combines both a date and a time
- The syntax is similar to that used with LocalDate and LocalTime, with “now” and “of” methods

```
LocalDateTime dateTimeNow = LocalDateTime.now();
LocalDateTime pastDateTime = LocalDateTime.of(1970, 1, 1, 0, 0);
```

- Calling “toString” on a LocalDateTime shows that it contains both date and a time, e.g.

```
2020-04-02T20:19:06.498307
```

Formatting Dates and Times

- Classes in the “java.time.format” package can be used to format dates and times
- The DateTimeFormatter class provides many formatting options for both dates and times
- The “ofLocalizedDate” method creates a DateTimeFormatter object that deals only with the date and can be passed one of the FormatStyle constants of SHORT, MEDIUM, LONG or FULL to specify the output format

```
DateTimeFormatter formatter =
DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT);
```

- Passing a DateTimeFormatter object to the “format” method of the date returns a formatted date as a String:

```
String dateText = date.format(formatter);        // 31/03/20
```

Other Date Formats

- There are other built in formats defined by constants
  - MEDIUM
  - LONG
  - FULL
- For example, this will create a formatter using the FULL format style:

```
DateTimeFormatter formatter = DateTimeFormatter.ofLocalizedDate(FormatStyle.FULL);
```

- Displaying the same date in FULL format will look like this:

```
Thursday, 1 January 1970
```

Formatting LocalTime

- The “ofLocalizedTime” factory method can be used to display elements of the time
- Only the SHORT and MEDIUM formats will work with a LocalTime because the LONG and FULL formats require more detail than is included in a LocalTime.

```
LocalTime time = LocalTime.now();
DateTimeFormatter timeFormatter =
DateTimeFormatter.ofLocalizedTime(FormatStyle.MEDIUM);
String timeText = time.format(timeFormatter);
System.out.println(timeText);
```

- The output from the MEDIUM format, looks something like this:

```
11:04:05 pm
```

ZonedDateTime

- For the longer formats that include time zone information, a ZonedDateTime object needs to be used, since this is the only class that handles time zones.

```
ZonedDateTime zonedTime = ZonedDateTime.now();
DateTimeFormatter zonedTimeFormatter = DateTimeFormatter.ofLocalizedDateTime(FormatStyle.FULL);
String zonedTimeText = zonedTime.format(zonedTimeFormatter);
System.out.println(zonedTimeText);
```

- This is the FULL format of the ZonedDateTime

```
Thursday, 2 April 2020 at 8:31:25 pm New Zealand Daylight Time
```

Formatting Dates and Times

- We can format dates using a DateFormat object
  - factory methods rather than constructors
- 'getInstance' method creates a DateFormat object with default 'short' format

```
DateFormat defaultDateFormat = DateFormat.getInstance();
```

- Passing a Date object to the 'format' method returns a String containing the formatted date

```
System.out.println(defaultDateFormat.format(date));
```

```
1/1/70 12:00 AM
```

Applying Format Patterns

- In addition to the four standard formats, the DateTimeFormatter has an “ofPattern” method
  - allows you to specify exactly how you want the date and/or time to appear using special characters
- In this example, we apply a pattern so that the date appears in the order day, month, year, separated by forward slashes (case is significant here).

```
LocalDate date = LocalDate.of(1970,1,1);
DateTimeFormatter customDateFormat = DateTimeFormatter.ofPattern("dd/MM/yy");
String dateText = date.format(customDateFormat);
System.out.println(dateText);
```

- This pattern leads to the date being formatted as the following String:

```
01/01/70
```

Date and Time Patterns (Partial List)

Letter	Date or Time Component	Presentation	Examples
U	year	year	2004; 04
Y	year-of-era	year	2004; 04
D	day-of-year	number	189
M/L	month-of-year	number/text	7; 07; Jul; July; J
D	day-of-month	number	10
G	modified-julian-day	number	2451334
Q/q	quarter-of-year	number/text	3; 03; Q3; 3rd quarter
Y	week-based-year	year	1996; 96
W	week-of-week-based-year	number	27
W	week-of-month	number	4
E	day-of-week	text	Tue; Tuesday; T
e/c	localized day-of-week	number/text	2; 02; Tue; Tuesday; T
F	day-of-week-in-month	number	3
A	am-pm-of-day	text	PM
H	clock-hour-of-am-pm (1-12)	number	12
K	hour-of-am-pm (0-11)	number	0
K	clock-hour-of-day (1-24)	number	24
H	hour-of-day (0-23)	number	0
m	minute-of-hour	number	30

Date Format Examples

- The number of characters also affects how that element of the date is formatted
- If the number of pattern letters is four or more, the full form is used
- Using three ‘M’ characters displays an abbreviated month name.

```
DateTimeFormatter customDateFormat = DateTimeFormatter.ofPattern("dd-MMMM-yyyy");
```

```
01-Jan-1970
```

- Using four ‘M’ characters for the month would use the full month name:

```
01-January-1970
```

- This example includes the full day name

```
DateTimeFormatter customDateFormat = DateTimeFormatter.ofPattern("EEEE dd MMMM, yyyy");
```

```
Thursday 01 January, 1970
```

Parsing Dates

- The LocalDate's 'parse' method can be used to convert Strings to Dates
- the String pattern used in the "ofPattern" method of the DateTimeFormatter determines the way that dates are parsed
  - The data passed as the String argument must be in the expected format for the parse to succeed
  - The number of characters is not important when parsing

```
DateTimeFormatter dateFormat = DateTimeFormatter.ofPattern("M/d/y");
LocalDate parsedDate = LocalDate.parse("10/22/2012", dateFormat);
System.out.println(parsedDate);
```

- The output from this example is the standard "toString" output from the LocalDate object.

```
2012-10-22
```

Exercise 11.7

- Use a LocalDate and the “of” method to create and set a specific date.
- Format the Date using a consistent separation character (e.g. “/”).
- Use the ‘split’ method of the String class to split the formatted date into separate values and display them
- The ‘split’ method can be used to split a String using a separator String. It returns an array of Strings
  - in this example a space is used as the separator String

```
String st = new String("this is a test");
String[] split = st.split(" ");
```

Summary

- This chapter covered a small sample of classes from some of the packages in the Java libraries
  - Object, Math, System and the wrapper classes from java.lang
  - Locale and Currency classes from java.util
  - Classes from java.text to format and parse numbers and currencies
  - Classes from java.time for representing dates and times
  - Classes from java.time.format to format and parse dates and times
- Reuse existing library classes as much as possible
- Become familiar with using the Javadoc to explore classes and methods