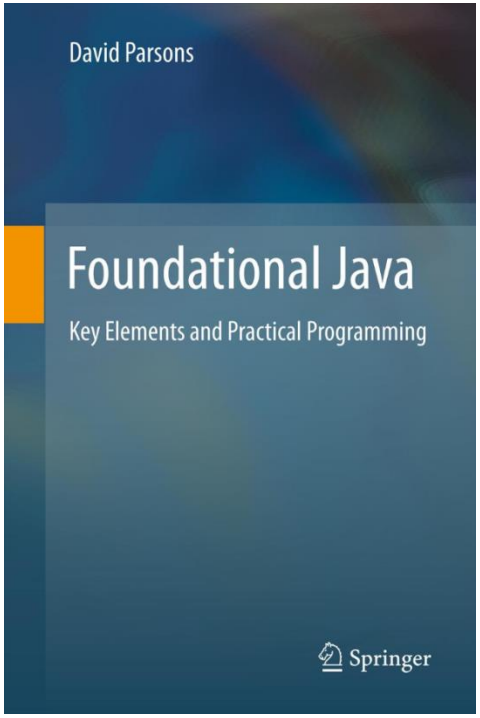


Chapter 11
Exploring the Java Libraries

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

```
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'

```
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	2.718281828459045
public static final double	PI	3.141592653589793

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

  `Date`

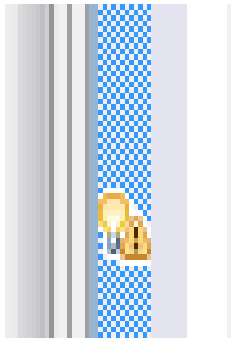
  `Calendar`

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

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

The Date Class

- Date has largely become immutable
  - Methods to manipulate dates are deprecated and now part of the Calendar class
- Deprecated methods are indicated in Eclipse with strikethrough text



```
Date date = new Date();
date.getDay();
```

The Calendar Class and Factory Methods

- Calendar represents a mutable date
- Does not have a public constructor

```
Calendar cal = new Calendar(); // will not compile
```

- Created using a Factory method

```
Calendar cal = Calendar.getInstance();
```

- The ‘clear’ method sets all the fields to appropriate zero or null values

```
cal.clear();
```

Calendar Methods

- Elements of the calendar can be set, e.g.

```
cal.set(year, month, day);
```

  - Using int parameters – use full year, month values are from 0 to 11, e.g.

```
cal.set(1970, 0, 1);
```

- The current date can be returned as a Date instance, using the getTime() method

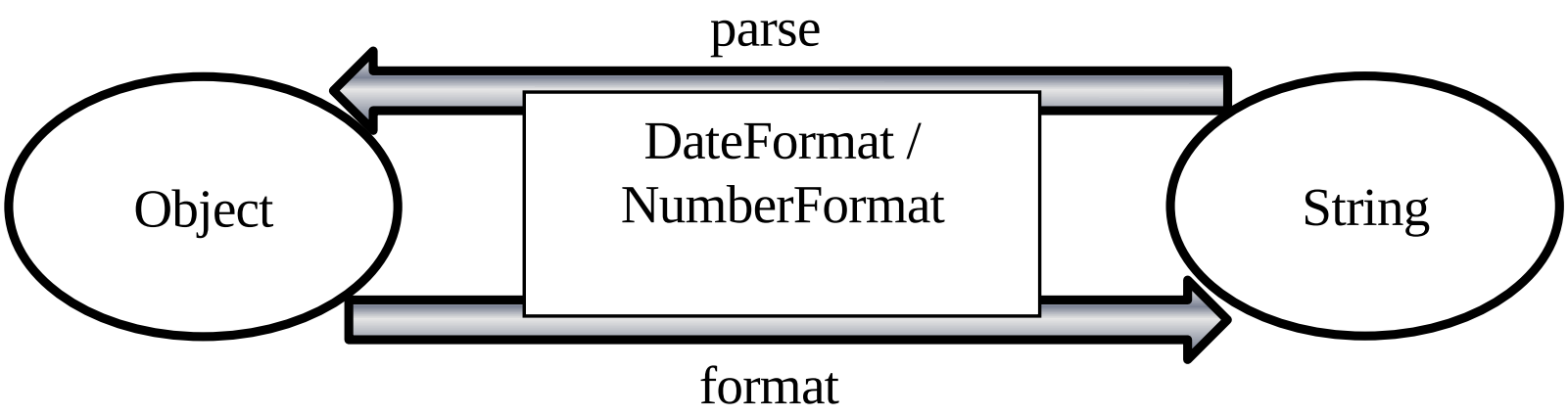
```
Date myDate = cal.getTime();
```

Formatter Classes in java.text

- DateFormat and NumberFormat classes
- Part of the java.text package
- Convert from Strings to objects (and primitive types) using ‘format’ methods
- Convert from objects (and primitive types) to Strings using ‘parse’ methods
- Customizable formatting

Format and Parse

- 'format' methods convert from objects or primitive types to Strings
- 'parse' methods convert from Strings to objects or primitive types



Formatting Dates

- 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
```

Built-In Date Formats

- Built in formats are specified as static final fields in the DateFormat class
  - SHORT, MEDIUM, LONG, FULL
- To set a specific pattern, use the factory method ‘getDateInstance(int)’ and pass one of the four constants as the parameter, e.g.

```
DateFormat longDateFormat = DateFormat.getDateInstance(DateFormat.LONG);
```

January 1, 1970

Applying Format Patterns

- Custom format patterns can be applied
- Cast the DateFormat down to SimpleDateFormat

```
SimpleDateFormat customDateFormat =
        (SimpleDateFormat) DateFormat.getDateInstance();
```

- SimpleDateFormat has an 'applyPattern' method
  - uses special characters
- This pattern is day, month, year, separated by forward slashes (case is significant)

```
customDateFormat.applyPattern("dd/MM/yy");
```

01/01/70

Date and Time Patterns

Letter	Date or Time Component	Presentation	Examples
G	Era designator	Text	AD
y	Year	Year	1996; 96
M	Month in year	Month	July; Jul; 07
w	Week in year	Number	27
W	Week in month	Number	2
D	Day in year	Number	189
d	Day in month	Number	10
F	Day of week in month	Number	2
E	Day in week	Text	Tuesday; Tue
a	Am/pm marker	Text	PM
H	Hour in day (0-23)	Number	0
k	Hour in day (1-24)	Number	24
K	Hour in am/pm (0-11)	Number	0
h	Hour in am/pm (1-12)	Number	12
m	Minute in hour	Number	30
s	Second in minute	Number	55
S	Millisecond	Number	978
z	Time zone	General time zone	Pacific Standard Time; PST; GMT-08:00
Z	Time zone	RFC 822 time zone	-0800

Date Format Examples

- Using three ‘M’ characters displays an abbreviated month name.

```
customDateFormat.applyPattern("dd-MMM-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

```
customDateFormat.applyPattern("EEEE dd MMMM, yyyy");
```

Thursday 01 January, 1970

Parsing Dates

- The DateFormat's 'parse' method can be used to convert Strings to Dates
- The String pattern used in the 'applyPattern' method determines the way that dates are parsed
- The number of characters is not important when parsing

```
SimpleDateFormat parseDateFormat = new SimpleDateFormat("M/d/y");
```

Date Parsing Example

```
SimpleDateFormat parseDateFormat = new SimpleDateFormat("M/d/y");
try
{
  Date d = parseDateFormat.parse("10/22/2012");
  System.out.println(d);
}
catch(ParseException e)
{
  e.printStackTrace();
}
```

Mon Oct 22 00:00:00 NZDT 2012

Exercise 11.3

- Use a Calendar to create and set a specific date
- Get a Date from the Calendar (using the 'getTime' method)
- 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(" ");
```

Formatting and Parsing Numbers

- Use a NumberFormat object from a 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 Number Class

- 'parse' methods of NumberFormat class parse Strings into numbers
- Return instances of the Number class
  - Superclass of the number wrapper classes
- Has various methods to return primitive numbers
  - 'byteValue', 'doubleValue', 'intValue' etc.
- Some element of truncation or rounding is possible

```
try {
  Number num = numFormat.parse("1234.5");
  System.out.println(num.doubleValue());
  System.out.println(num.intValue());
}
catch (ParseException e) {
  e.printStackTrace();
}
```

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"
```

Parsing Currency

- In this example, we parse a String into a Number and then return the ‘doubleValue’.

```
try
{
  // must be a parseable string in the local currency
  value = dollarFormat.parse("$5,432.10").doubleValue();
  System.out.println(value);    // 5432.1
}
catch (ParseException e)
{
  e.printStackTrace();
}
```

Handling Different Currencies

- Java provides for different formats based on country
- The Locale class defines formatting options for numbers, dates and currencies

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

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

  - Locales do nothing to convert between currencies

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

- 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

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
  - Date and Calendar classes from java.util
  - Classes from java.text to format and parse dates, numbers and currencies
- Reuse existing library classes as much as possible
- Become familiar with using the Javadoc to explore classes and methods