Chapter 4

Dates, Times, Locales, and Resource Bundles

Business Training

1

Content

- **Dates, Times and Locales**
- **Dealing with Time Zones and Daylight Savings**
- **Formatting Dates and Times**
- **Properties Files**
- **Resource Bundles**

2

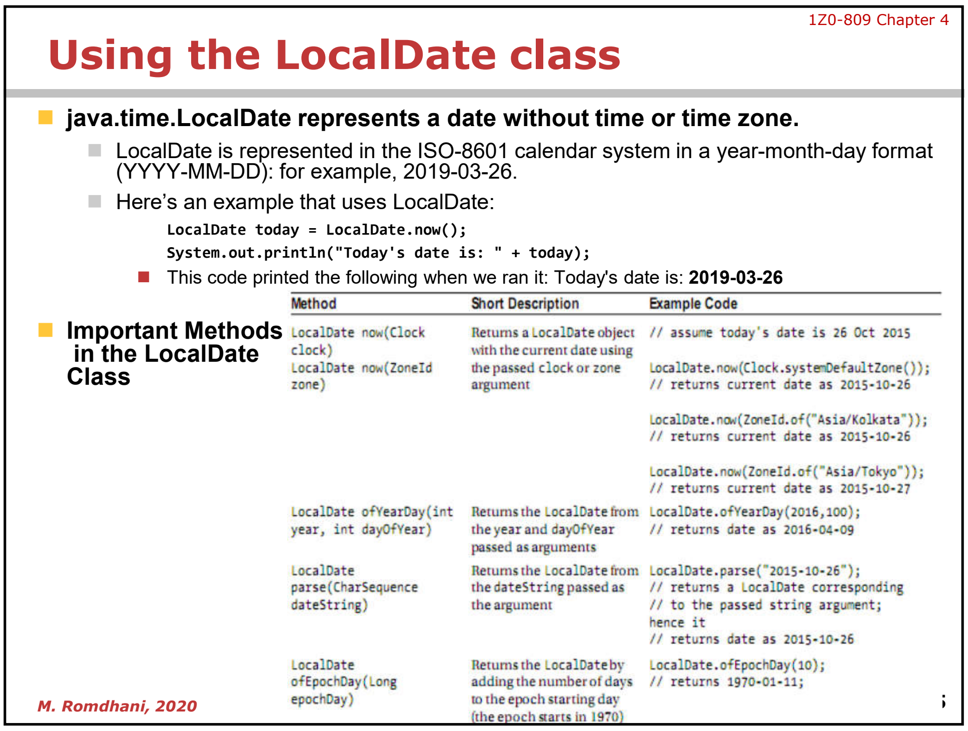
Dates, Times, and Locales

3

Working with Dates and Times

- **Here's an overview of how the classes in java.time are organized**
  - **Local dates and times**
    - These dates and times are local to your time zone and so don't have time-zone information associated with them. These are represented by classes java.time.LocalDate, java.time.LocalTime, and java.time.LocalDateTime.
  - **Zoned dates and times**
    - These dates and times include time-zone information. They are represented by classes java.time.ZonedDateTime and java.time.OffsetDateTime.
  - **Formatters for dates and times**
    - With java.time.format.DateTimeFormatter, you can parse and print dates and times with patterns and in a variety of styles.
  - **Adjustments to dates and times**
    - With java.time.temporal.TemporalAdjusters and java.time.temporal.ChronoUnit, you can adjust and manipulate dates and times by handy increments.
  - **Periods, Durations, and Instants**
    - java.time.Periods and java.time.Durations represent an amount of time, periods for days or longer and durations for shorter periods like minutes or seconds. java.time .Instants represent a specific instant in time, so you can, say, compute the number of minutes between two instants.

4



1Z0-809 Chapter 4

Using the LocalDate class

- **java.time.LocalDate represents a date without time or time zone.**
  - LocalDate is represented in the ISO-8601 calendar system in a year-month-day format (YYYY-MM-DD): for example, 2019-03-26.
  - Here's an example that uses LocalDate:

```
LocalDate today = LocalDate.now();
System.out.println("Today's date is: " + today);
```

    - This code printed the following when we ran it: Today's date is: **2019-03-26**

- **Important Methods in the LocalDate Class**

Method	Short Description	Example Code

`LocalDate now(Clock clock)`
`LocalDate now(ZoneId zone)`
	Returns a `LocalDate` object with the current date using the passed `clock` or `zone` argument	`// assume today's date is 26 Oct 2015`
		
		`LocalDate.now(Clock.systemDefaultZone());`
		`// returns current date as 2015-10-26`
		
		`LocalDate.now(ZoneId.of("Asia/Kolkata"));`
		`// returns current date as 2015-10-26`
		
		`LocalDate.now(ZoneId.of("Asia/Tokyo"));`
		`// returns current date as 2015-10-27`

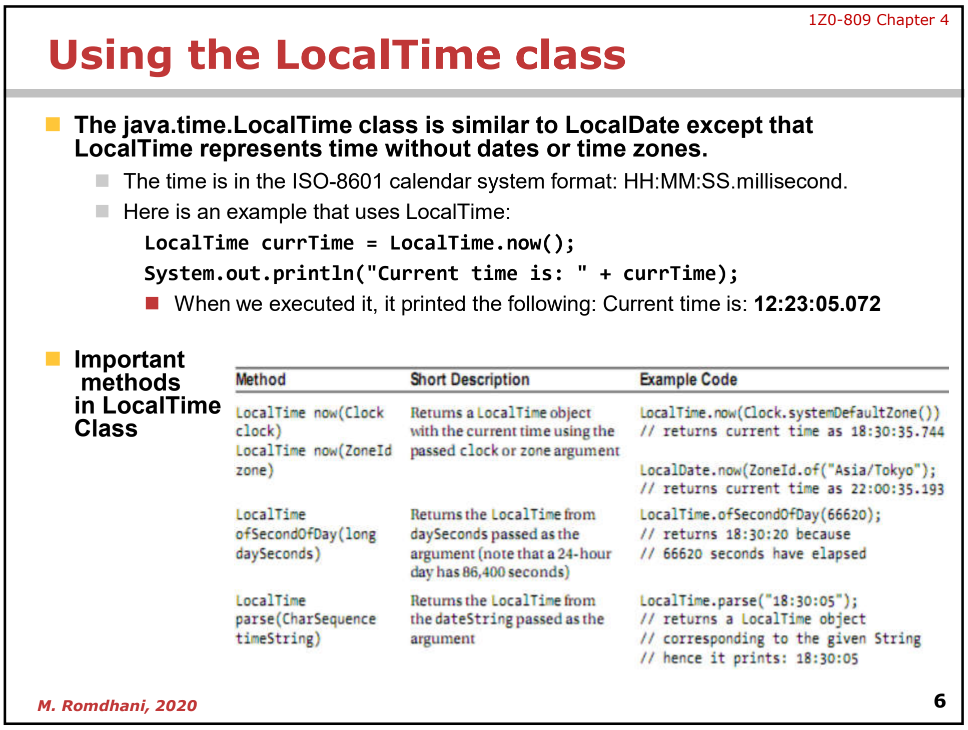
`LocalDate ofYearDay(int year, int dayOfYear)`	Returns the `LocalDate` from the year and `dayOfYear` passed as arguments	`LocalDate.ofYearDay(2016,100);`
		`// returns date as 2016-04-09`

`LocalDate parse(CharSequence dateString)`	Returns the `LocalDate` from the `dateString` passed as the argument	`LocalDate.parse("2015-10-26");`
		`// returns a LocalDate corresponding`
		`// to the passed string argument;`
		`hence it`
		`// returns date as 2015-10-26`

`LocalDate ofEpochDay(Long epochDay)`	Returns the `LocalDate` by adding the number of days to the epoch starting day (the epoch starts in 1970)	`LocalDate.ofEpochDay(10);`
		`// returns 1970-01-11;`

M. Romdhani, 2020

5



1Z0-809 Chapter 4

Using the LocalTime class

- **The java.time.LocalTime class is similar to LocalDate except that LocalTime represents time without dates or time zones.**
  - The time is in the ISO-8601 calendar system format: HH:MM:SS.millisecond.
  - Here is an example that uses LocalTime:

```
LocalTime currTime = LocalTime.now();
System.out.println("Current time is: " + currTime);
```

    - When we executed it, it printed the following: Current time is: **12:23:05.072**

- **Important methods in LocalTime Class**

Method	Short Description	Example Code

`LocalTime now(Clock clock)`
`LocalTime now(ZoneId zone)`
	Returns a `LocalTime` object with the current time using the passed `clock` or `zone` argument	`LocalTime.now(Clock.systemDefaultZone())`
		`// returns current time as 18:30:35.744`
		
		`LocalDate.now(ZoneId.of("Asia/Tokyo");`
		`// returns current time as 22:00:35.193`

`LocalTime ofSecondOfDay(long daySeconds)`	Returns the `LocalTime` from `daySeconds` passed as the argument (note that a 24-hour day has 86,400 seconds)	`LocalTime.ofSecondOfDay(66620);`
		`// returns 18:30:20 because`
		`// 66620 seconds have elapsed`

`LocalTime parse(CharSequence timeString)`	Returns the `LocalTime` from the `dateString` passed as the argument	`LocalTime.parse("18:30:05");`
		`// returns a LocalTime object`
		`// corresponding to the given String`
		`// hence it prints: 18:30:05`

M. Romdhani, 2020

6

6

Using the LocalDateTime Class

- **The class java.time.LocalDateTime represents both date and time without time zones.**
  - You can think of LocalDateTime as a logical combination of the LocalTime and LocalDate classes. The date and time formats use the ISO-8601 calendar system: YYYY-MM-DD HH:MM:SS.millisecond.
  - Here is a simple example that prints today's date and the current time:

```
LocalDateTime currDateTime = LocalDateTime.now();
System.out.println("Today's date and current time is: " +
   currDateTime);
```

    - This prints : Today's date and current time is: **2019-03-26T21:04:36.376**

      In this output, note that the character **T** stands for time, and it separates the date and time components.

- **Similar to LocalDate and LocalTimethe methods, LocalDateTime has methods such as now(), of(), and parse(). Again, similar, this class also provides methods to add or subtract years, months, days, hours, minutes, seconds, and nanoseconds.**

Instant, Period, Duration Classes

- **The instant values began on January 1, 1970, at 00:00:00 hours (known as the Unix epoch).**
  - The Instant class internally uses a long variable that holds the number of seconds since the start of the Unix epoch
- **The java.time.Period class is used to measure an amount of time in terms of years, months, and days.**
  - Assume that you have bought some expensive medicine and want to use it before it expires. Here is how you can find out when it will expire:

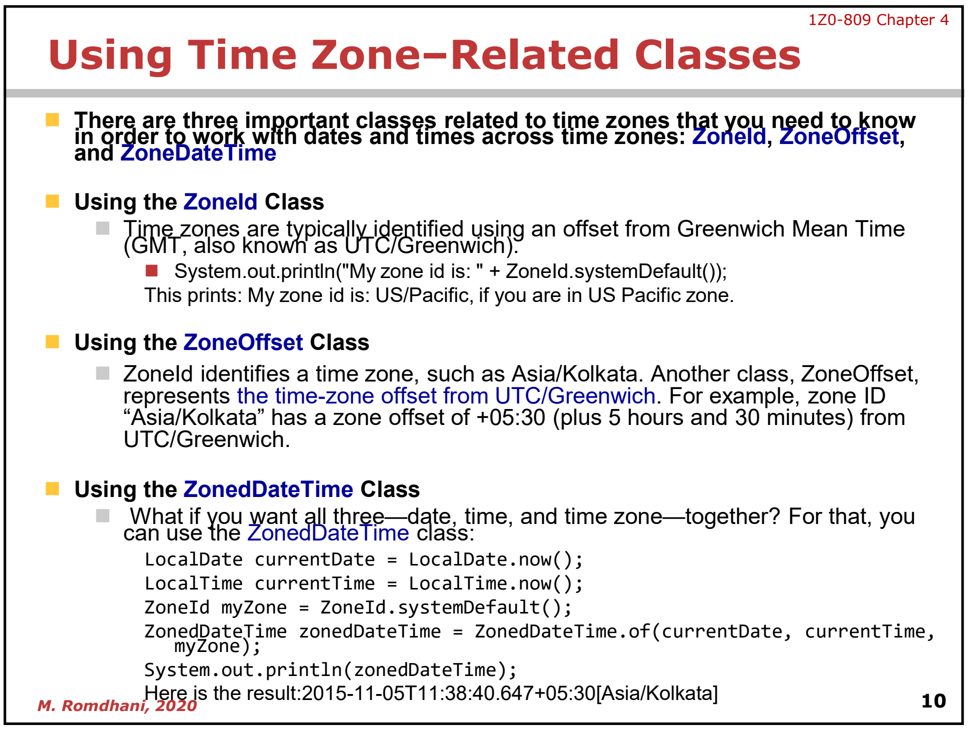
```
LocalDate manufacturingDate = LocalDate.of(2016, Month.JANUARY, 1);
LocalDate expiryDate = LocalDate.of(2018, Month.JULY, 18);

Period expiry = Period.between(manufacturingDate, expiryDate);
System.out.printf("Medicine will expire in: %d years, %d months, and %d days
   (%s)\n“, expiry.getYears(), expiry.getMonths(), expiry.getDays(), expiry)
```

- **Duration is the time equivalent of Period. The Duration class represents time in terms of hours, minutes, seconds, and so on.**
  - It is suitable for measuring machine time or when working with Instance objects. Similar to the Instance class, the Duration class stores the seconds component as a long value and nanoseconds using an int value.

Dealing with Time Zones and Daylight Savings

9



1Z0-809 Chapter 4

Using Time Zone–Related Classes

- **There are three important classes related to time zones that you need to know in order to work with dates and times across time zones: ZoneId, ZoneOffset, and ZoneDateTime**
- **Using the ZoneId Class**
  - Time zones are typically identified using an offset from Greenwich Mean Time (GMT, also known as UTC/Greenwich).
    - System.out.println("My zone id is: " + ZoneId.systemDefault());

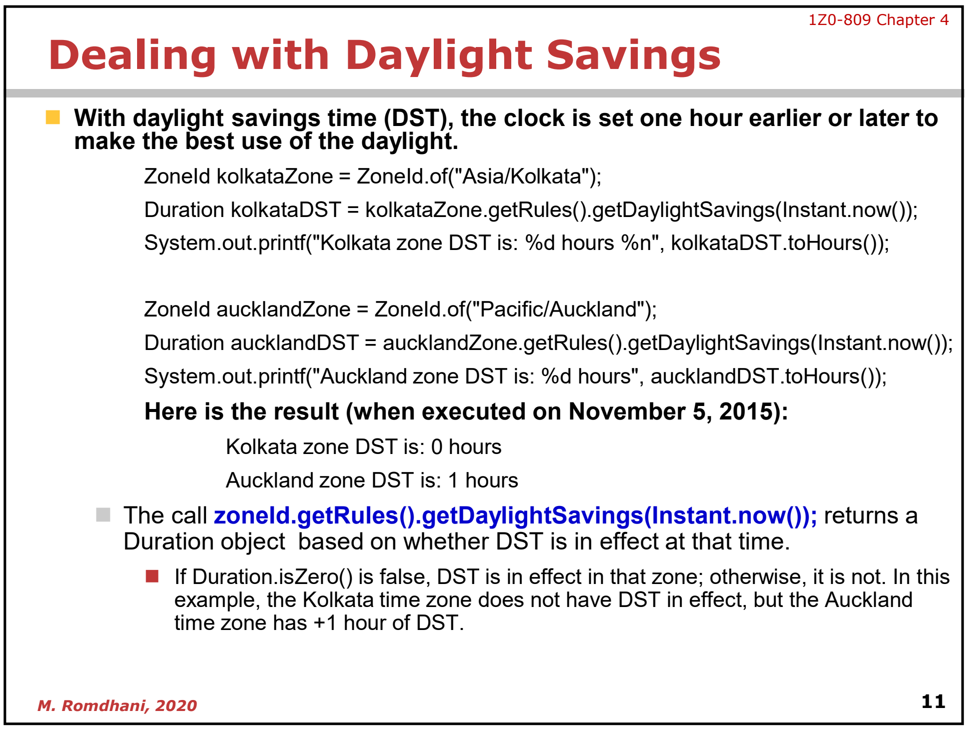
    This prints: My zone id is: US/Pacific, if you are in US Pacific zone.
- **Using the ZoneOffset Class**
  - ZoneId identifies a time zone, such as Asia/Kolkata. Another class, ZoneOffset, represents the time-zone offset from UTC/Greenwich. For example, zone ID “Asia/Kolkata” has a zone offset of +05:30 (plus 5 hours and 30 minutes) from UTC/Greenwich.
- **Using the ZonedDateTime Class**
  - What if you want all three—date, time, and time zone—together? For that, you can use the ZonedDateTime class:

```
LocalDate currentDate = LocalDate.now();
LocalTime currentTime = LocalTime.now();
ZoneId myZone = ZoneId.systemDefault();
ZonedDateTime zonedDateTime = ZonedDateTime.of(currentDate, currentTime,
    myZone);
System.out.println(zonedDateTime);
```

Here is the result:2015-11-05T11:38:40.647+05:30[Asia/Kolkata]

M. Romdhani, 2020 **10**

10



Dealing with Daylight Savings

- **With daylight savings time (DST), the clock is set one hour earlier or later to make the best use of the daylight.**

```
ZoneId kolkataZone = ZoneId.of("Asia/Kolkata");
Duration kolkataDST = kolkataZone.getRules().getDaylightSavings(Instant.now());
System.out.printf("Kolkata zone DST is: %d hours %n", kolkataDST.toHours());

ZoneId aucklandZone = ZoneId.of("Pacific/Auckland");
Duration aucklandDST = aucklandZone.getRules().getDaylightSavings(Instant.now());
System.out.printf("Auckland zone DST is: %d hours", aucklandDST.toHours());
```

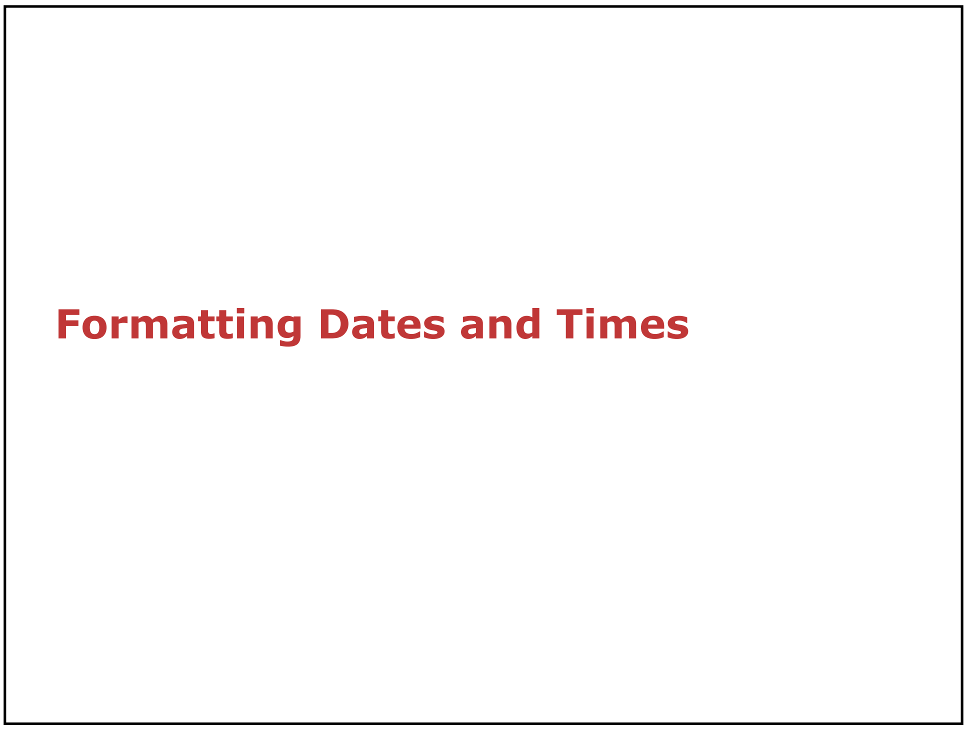
Here is the result (when executed on November 5, 2015):

```
Kolkata zone DST is: 0 hours
Auckland zone DST is: 1 hours
```

  - The call **zoneId.getRules().getDaylightSavings(Instant.now());** returns a Duration object based on whether DST is in effect at that time.
    - If Duration.isZero() is false, DST is in effect in that zone; otherwise, it is not. In this example, the Kolkata time zone does not have DST in effect, but the Auckland time zone has +1 hour of DST.

M. Romdhani, 2020 **11**

11



Formatting Dates and Times

12

The DateTimeFormatter class

- **The DateTimeFormatter class provides many predefined constants for formatting date and time values.**
  - Here is a list of a few such predefined formatters (with sample output values):
    - **ISO_DATE (2015-11-05)**
    - **ISO_TIME (11:25:47.624)**
    - **RFC_1123_DATE_TIME (Thu, 5 Nov 2015 11:27:22 +0530)**
    - **ISO_ZONED_DATE_TIME (2015-11-05T11:30:33.49+05:30[Asia/Kolkata])**

- **Here is a simple example that uses the predefined ISO_TIME of type DateTimeFormatter:**

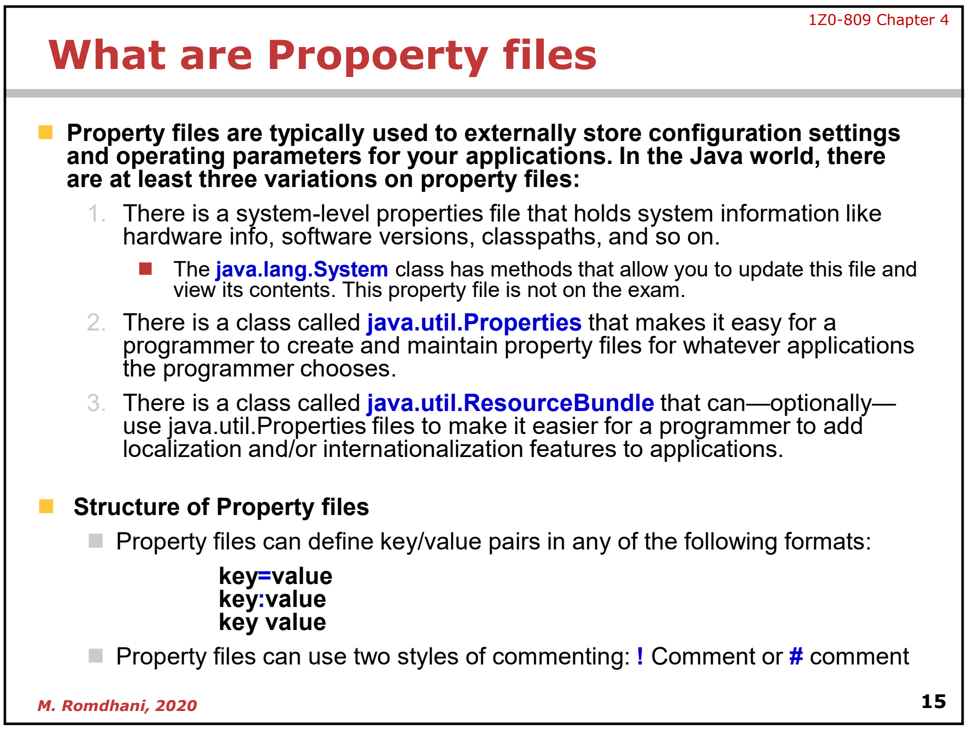
```
LocalTime wakeupTime = LocalTime.of(6, 0, 0);
System.out.println("Wake up time: " +
   DateTimeFormatter.ISO_TIME.format(wakeupTime));
```

  - This printed the following: Wake up time: 06:00:00
  - If you want to use a custom format instead of any of the predefined formats? To do so, you can use the ofPattern() method in the DateTimeFormatter class:

```
DateTimeFormatter customFormat = DateTimeFormatter.ofPattern("dd
   MMM yyyy");
System.out.println(customFormat.format(LocalDate.of(2016,
   Month.JANUARY, 01)));
```

  - Here is the result: 01 Jan 2016

Properties Files



1Z0-809 Chapter 4

What are Propoerty files

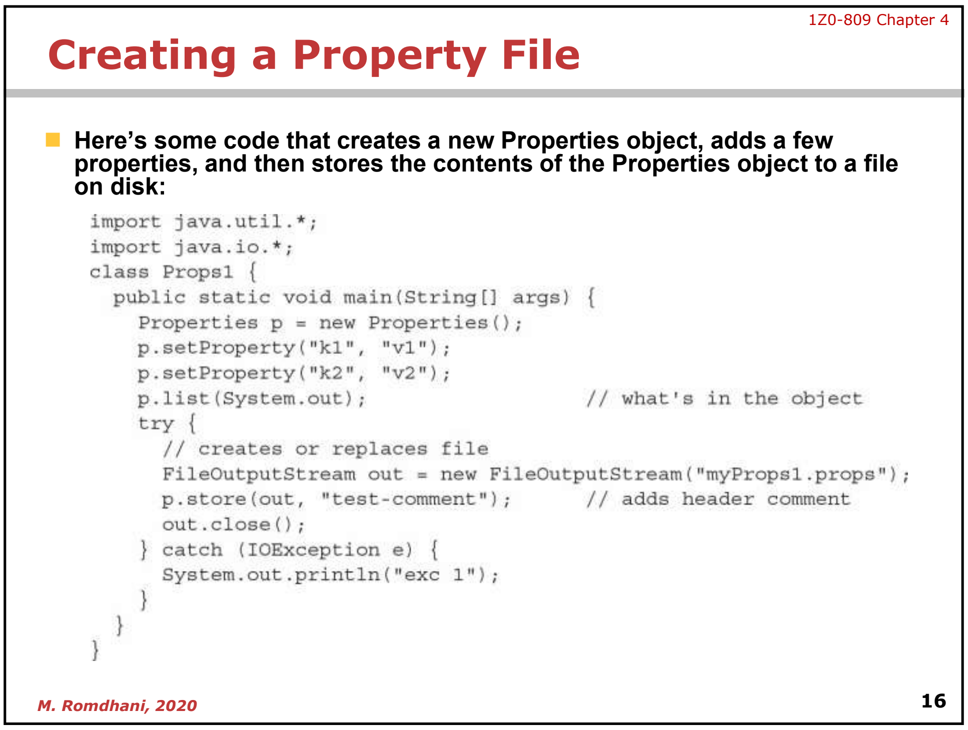
- **Property files are typically used to externally store configuration settings and operating parameters for your applications. In the Java world, there are at least three variations on property files:**
  1. There is a system-level properties file that holds system information like hardware info, software versions, classpaths, and so on.
     - The **java.lang.System** class has methods that allow you to update this file and view its contents. This property file is not on the exam.
  2. There is a class called **java.util.Properties** that makes it easy for a programmer to create and maintain property files for whatever applications the programmer chooses.
  3. There is a class called **java.util.ResourceBundle** that can—optionally—use java.util.Properties files to make it easier for a programmer to add localization and/or internationalization features to applications.

- **Structure of Property files**
  - Property files can define key/value pairs in any of the following formats:

    key=value
    key:value
    key value
  - Property files can use two styles of commenting: **!** Comment or **#** comment

M. Romdhani, 2020 15

15



1Z0-809 Chapter 4

Creating a Property File

- **Here's some code that creates a new Properties object, adds a few properties, and then stores the contents of the Properties object to a file on disk:**

```
import java.util.*;
import java.io.*;
class Props1 {
  public static void main(String[] args) {
    Properties p = new Properties();
    p.setProperty("k1", "v1");
    p.setProperty("k2", "v2");
    p.list(System.out);                  // what's in the object
    try {
      // creates or replaces file
      FileOutputStream out = new FileOutputStream("myProps1.props");
      p.store(out, "test-comment");      // adds header comment
      out.close();
    } catch (IOException e) {
      System.out.println("exc 1");
    }
  }
}
```

M. Romdhani, 2020 16

16

Reading a Property file

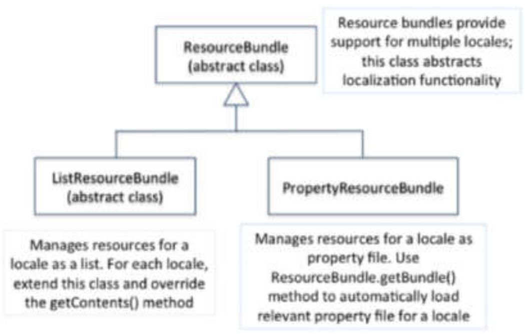
- **Now let's run a second program that opens up the file we just created, adds a new key/value pair, then saves the result to a second file on disk:**

```
import java.util.*;
import java.io.*;
class Props2 {
  public static void main(String[] args) {
    Properties p2 = new Properties();
    try {
      FileInputStream in = new FileInputStream("myProps1.props");
      p2.load(in);
      p2.list(System.out);
      p2.setProperty("newProp", "newData");
      p2.list(System.out);
      FileOutputStream out = new FileOutputStream("myProps2.props");
      p2.store(out, "myUpdate");
      in.close();
      out.close();
    } catch (IOException e) {
      System.out.println("exc 2");
    }
  }
}
```

Resource Bundles

What are ResourceBundles

- **In Java, resource bundles provide a solution to customize the application locale-specific needs.**
- **A resource bundle is a set of classes or property files that help define a set of keys and map those keys to locale specific values.**
  - The abstract class **java.util.ResourceBundle** provides an abstraction of resource bundles in Java. It has two derived classes: **java.util.PropertyResourceBundle** and **java.util.ListResourceBundle**


ResourceBundle (abstract class)
Resource bundles provide support for multiple locales; this class abstracts localization functionality
ListResourceBundle (abstract class)
PropertyResourceBundle
Manages resources for a locale as a list. For each locale, extend this class and override the getContents() method
Manages resources for a locale as property file. Use ResourceBundle.getBundle() method to automatically load relevant property file for a locale

Using PropertyResourceBundle

- **If you design your application with localization in mind using property files, you can add support for new locales to the application without changing anything in the code !**
- **Example : LocalizedHello**

```
D:\> type ResourceBundle.properties
Greeting=Hello

D:\> type ResourceBundle_ar.properties
Greeting=As-Salamu Alaykum

D:\> type ResourceBundle_it.properties
Greeting=Ciao
```

```
import java.util.Locale;
import java.util.ResourceBundle;

public class LocalizedHello {
    public static void main(String args[]) {
        Locale currentLocale = Locale.getDefault();
        ResourceBundle resBundle =
                ResourceBundle.getBundle("ResourceBundle", currentLocale);
        System.out.printf(resBundle.getString("Greeting"));
    }
}
```

  - Using the LocalizedHello

    D:\> java LocalizedHello, prints **Hello**

    D:\> java -Duser.language=it LocalizedHello, prints **Ciao**

    D:\> java -Duser.language=ar LocalizedHello, prints **As-Salamu Alaykum**

Using ListResourceBundle

- **Support for a new locale can be added using ListResourceBundle by extending it.**
  - While extending the ListResourceBundle, you need to override the abstract method getContents(); the signature of this method is:
    - **protected Object[][] getContents();**
- **ListResourceBundle example**

```
// default US English version
public class ResBundle extends ListResourceBundle {
    public Object[][] getContents() {
        return contents;
    }
    static final Object[][] contents = {
        { "MovieName", "Avatar" },
        { "GrossRevenue", (Long) 2782275172L }, // in US dollars
        { "Year", (Integer)2009 }
    };
}
```

  - Now, let's define a ResBundle for the Italian locale. You give the class the suffix "_it_IT

```
// Italian version
public class ResBundle_it_IT extends ListResourceBundle {
    public Object[][] getContents() {
        return contents;
    }
    static final Object[][] contents = {
        { "MovieName", "Che Bella Giornata" },
        { "GrossRevenue", (Long) 43000000L }, // in euros
        { "Year", (Integer)2011 }
    };
}
```

Using ListResourceBundle

- **Now let's see how to use our resource bundles.**
  - While extending the ListResourceBundle, you need to override the abstract method getContents(); the signature of this method is:

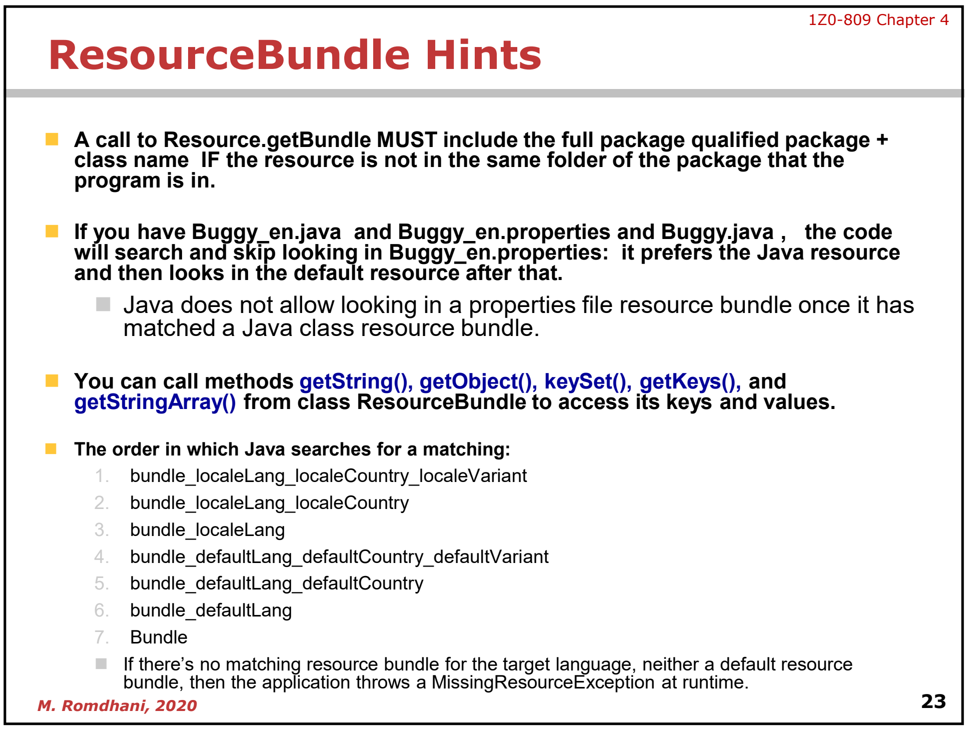
```
public class LocalizedBoxOfficeHits {
    public void printMovieDetails(ResourceBundle resBundle) {
        String movieName = resBundle.getString("MovieName");
        Long revenue = (Long)(resBundle.getObject("GrossRevenue"));
        Integer year = (Integer) resBundle.getObject("Year");

        System.out.println("Movie " + movieName + "(" + year + ")" + " grossed "
                + revenue );
    }
    public static void main(String args[]) {
        LocalizedBoxOfficeHits localizedHits = new LocalizedBoxOfficeHits();
        // print the largest box-office hit movie for default (US) locale
        Locale locale = Locale.getDefault();
        localizedHits.printMovieDetails(ResourceBundle.getBundle("ResBundle", locale));

        // print the largest box-office hit movie for Italian locale
        locale = new Locale("it", "IT", "");
        localizedHits.printMovieDetails(ResourceBundle.getBundle("ResBundle", locale));
    }
```

  - It prints the following:

```
Movie Avatar (2009) grossed 2782275172
Movie Che Bella Giornata (2011) grossed 43000000
```



ResourceBundle Hints

- **A call to Resource.getBundle MUST include the full package qualified package + class name IF the resource is not in the same folder of the package that the program is in.**
- **If you have Buggy_en.java and Buggy_en.properties and Buggy.java , the code will search and skip looking in Buggy_en.properties: it prefers the Java resource and then looks in the default resource after that.**
  - Java does not allow looking in a properties file resource bundle once it has matched a Java class resource bundle.
- **You can call methods getString(), getObject(), keySet(), getKeys(), and getStringArray() from class ResourceBundle to access its keys and values.**
- **The order in which Java searches for a matching:**
  1. bundle_localeLang_localeCountry_localeVariant
  2. bundle_localeLang_localeCountry
  3. bundle_localeLang
  4. bundle_defaultLang_defaultCountry_defaultVariant
  5. bundle_defaultLang_defaultCountry
  6. bundle_defaultLang
  7. Bundle
  - If there's no matching resource bundle for the target language, neither a default resource bundle, then the application throws a MissingResourceException at runtime.