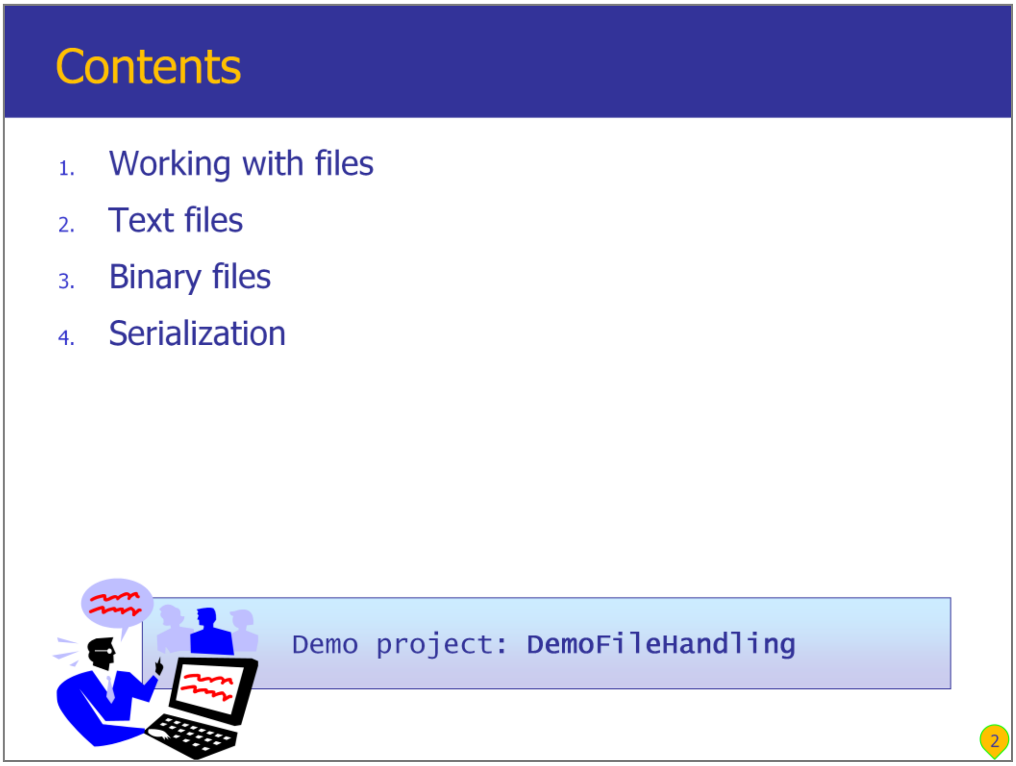
File Handling

olsen
software ltd

Java has extensive support for file handling. There are many classes and interfaces available, mostly defined in the `java.io` package. This chapter gives an overview of what's available, and then explores some of the most commonly used IO features.



Contents

1. Working with files
2. Text files
3. Binary files
4. Serialization

Demo project: `DemoFileHandling`

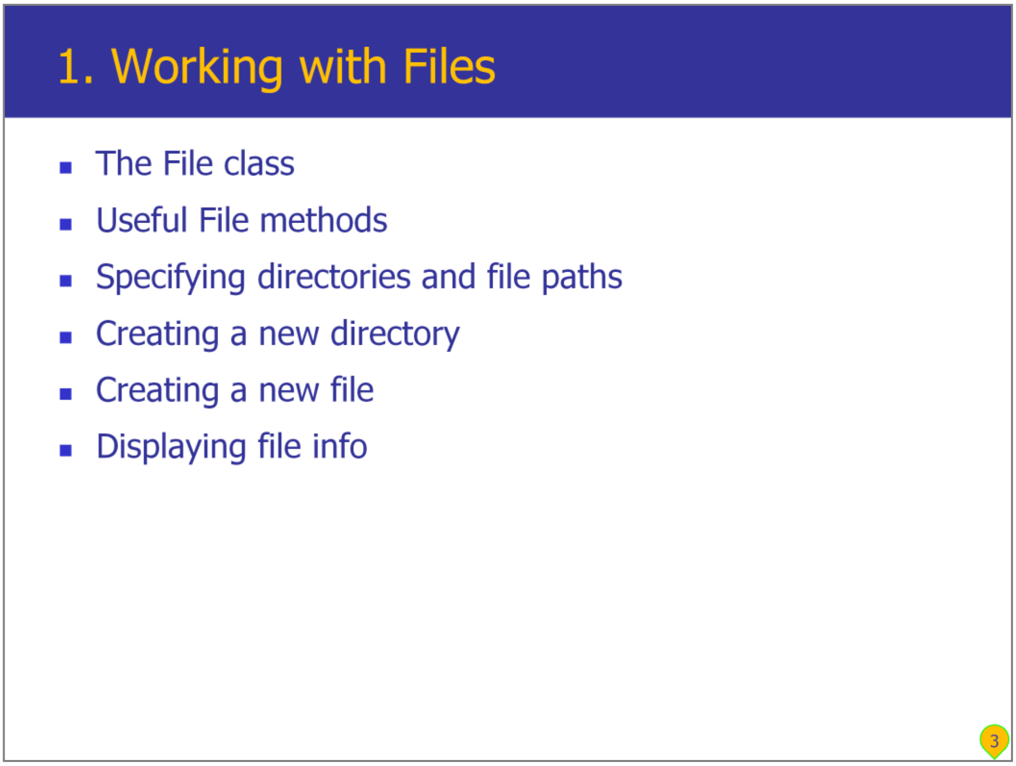
2

Section 1 introduces you to the File I/O API in Java, focussing on the `File` class in particular. You can use this class to perform common file and directory management tasks, such as determining if a file exists, creating a file, enumerating the files in a directory, etc.

Sections 2 and 3 describe how to read and write content in text files and binary files, respectively.

Section 4 introduces the concept of serialization, whereby you can convert an object into a flat byte stream (and then back again). This is useful if you want to pass objects over a network, for example.

The demos for chapter are located in the `DemoFileHandling` project.

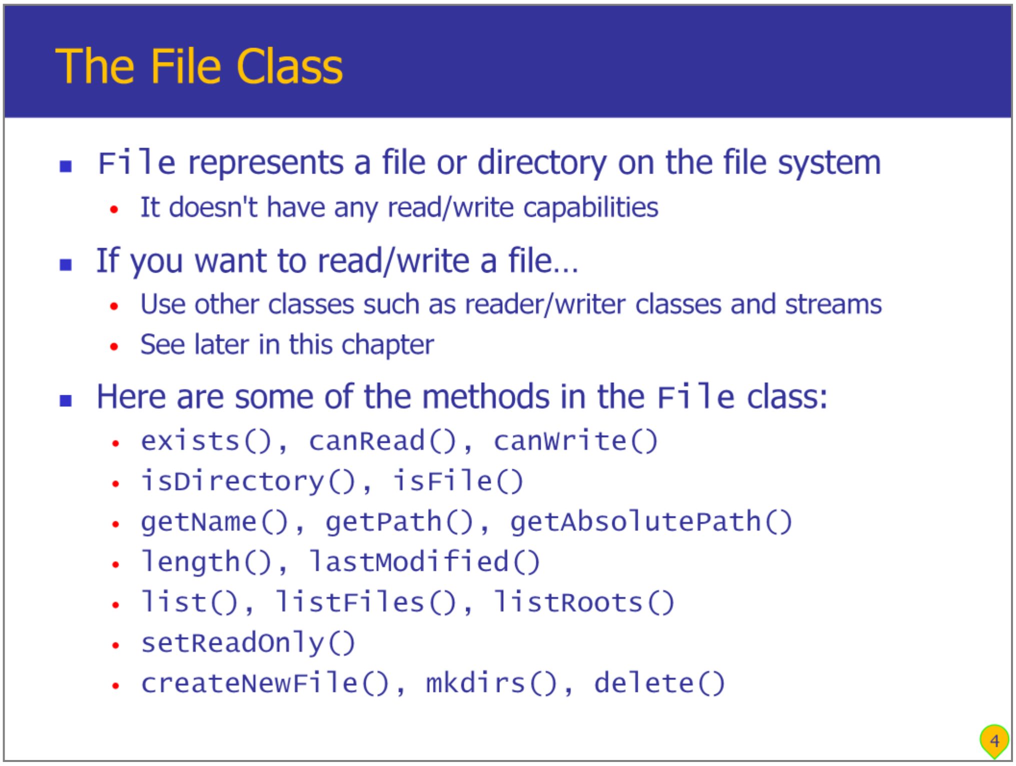


1. Working with Files

- The File class
- Useful File methods
- Specifying directories and file paths
- Creating a new directory
- Creating a new file
- Displaying file info

Java provides a rich set of classes and interfaces for manipulating files and data. These classes and interfaces are located in the `java.io` package. Many exceptions can occur during file handling; all of these IO exception classes inherit from `java.io.IOException`.

We begin by seeing how to perform whole-file and whole-directory operations using the `File` class. Note that there is no `Directory` class.



The File Class

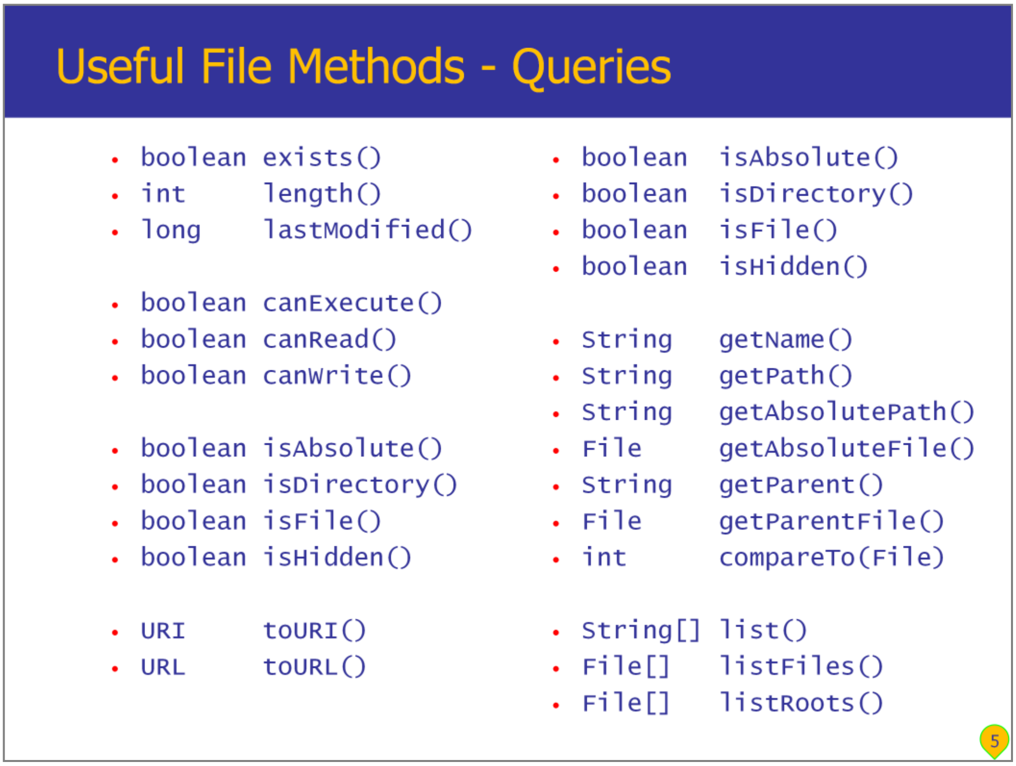
- `File` represents a file or directory on the file system
  - It doesn't have any read/write capabilities
- If you want to read/write a file…
  - Use other classes such as reader/writer classes and streams
  - See later in this chapter
- Here are some of the methods in the `File` class:
  - `exists(), canRead(), canWrite()`
  - `isDirectory(), isFile()`
  - `getName(), getPath(), getAbsolutePath()`
  - `length(), lastModified()`
  - `list(), listFiles(), listRoots()`
  - `setReadOnly()`
  - `createNewFile(), mkdirs(), delete()`

You can use the `File` class to perform operations on a file or a directory as a whole. The slide above lists some of the common methods in the `File` class.

Note that the `File` class doesn't have any methods for reading or writing the contents of a file. To do that, you must use a stream class or a reader/writer class, as discussed later in the chapter.

For full information about the `File` class, consult the JavaDoc documentation:

- http://docs.oracle.com/javase/7/docs/api/java/io/File.html



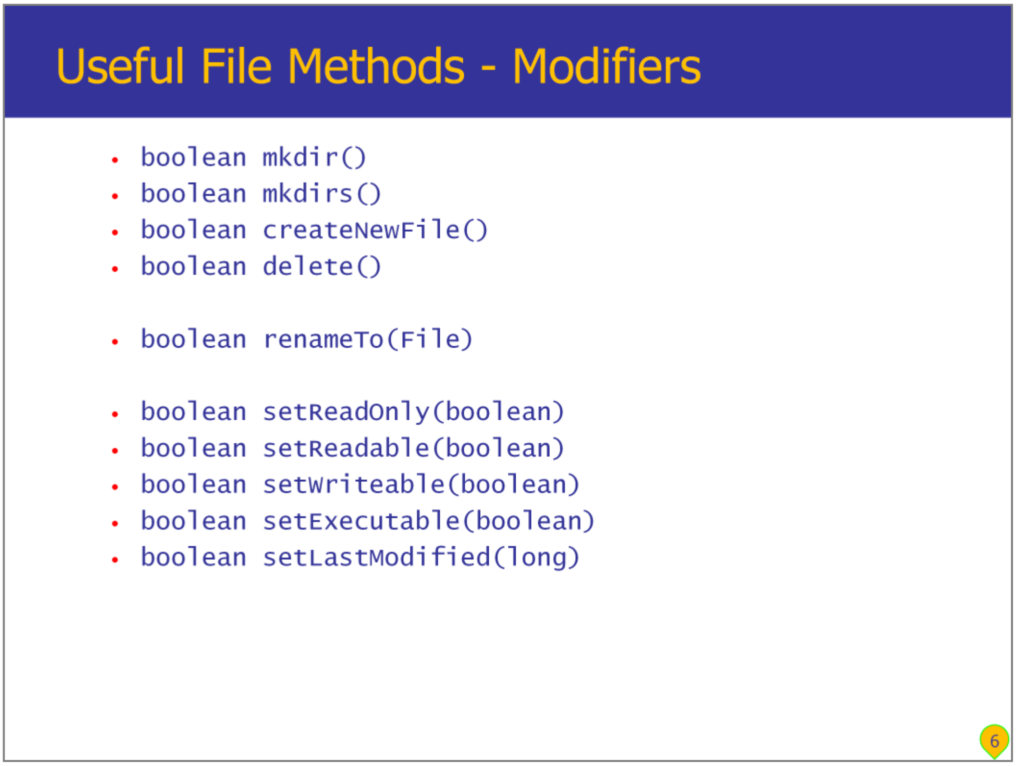
Useful File Methods - Queries

```
• boolean exists()
• int     length()
• long    lastModified()

• boolean canExecute()
• boolean canRead()
• boolean canWrite()

• boolean isAbsolute()
• boolean isDirectory()
• boolean isFile()
• boolean isHidden()

• URI     toURI()
• URL     toURL()
```

```
• boolean  isAbsolute()
• boolean  isDirectory()
• boolean  isFile()
• boolean  isHidden()

• String   getName()
• String   getPath()
• String   getAbsolutePath()
• File     getAbsoluteFile()
• String   getParent()
• File     getParentFile()
• int      compareTo(File)

• String[] list()
• File[]   listFiles()
• File[]   listRoots()
```

5

This slide lists some additional useful methods in the `File` class, for querying information about the file and directory attributes. As you can see, the `File` class provides all the capabilities you could ever need!



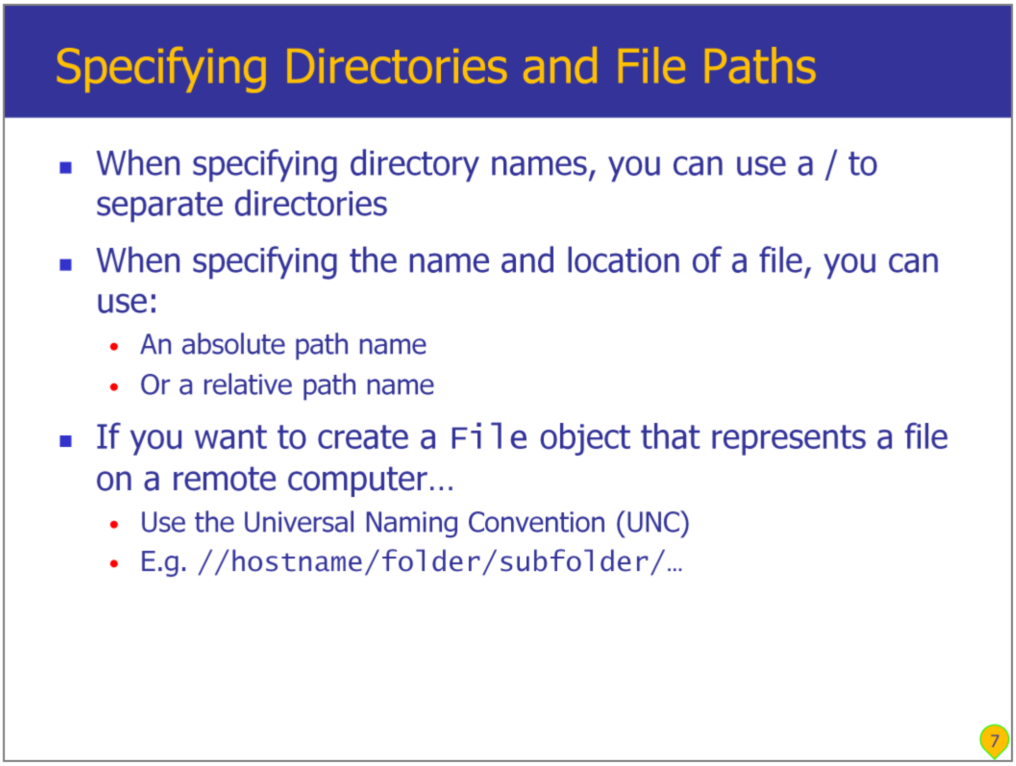
Useful File Methods - Modifiers

- `boolean mkdir()`
- `boolean mkdirs()`
- `boolean createNewFile()`
- `boolean delete()`

- `boolean renameTo(File)`

- `boolean setReadOnly(boolean)`
- `boolean setReadable(boolean)`
- `boolean setWriteable(boolean)`
- `boolean setExecutable(boolean)`
- `boolean setLastModified(long)`

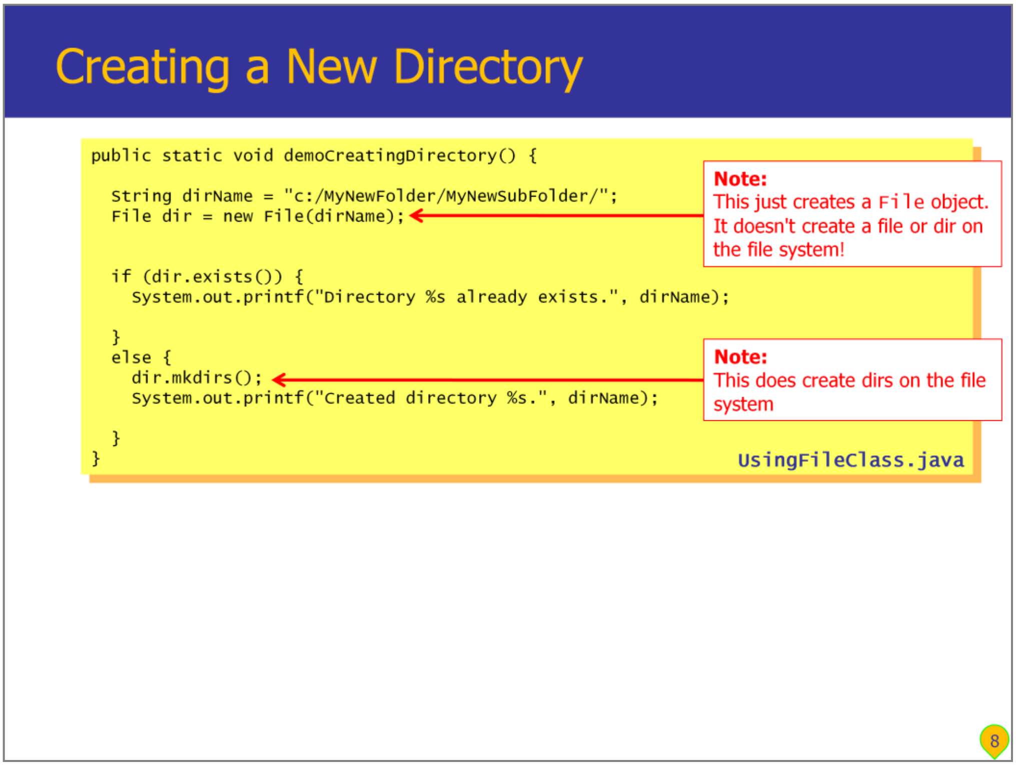
The File class provides methods that allow you to modify attributes on a file or directory, create a new file or directory, rename a file or directory, delete a file or directory, and so on.



Specifying Directories and File Paths

- When specifying directory names, you can use a / to separate directories
- When specifying the name and location of a file, you can use:
  - An absolute path name
  - Or a relative path name
- If you want to create a `File` object that represents a file on a remote computer…
  - Use the Universal Naming Convention (UNC)
  - E.g. `//hostname/folder/subfolder/…`

Java uses a forward slash / as the directory specifier, which is consistent with Unix and also allowable in Windows. You can also use UNC file names to represent remote files and directories on a network drive.



Creating a New Directory

```
public static void democreatingDirectory() {

  String dirName = "c:/MyNewFolder/MyNewSubFolder/";
  File dir = new File(dirName);


  if (dir.exists()) {
    System.out.printf("Directory %s already exists.", dirName);

  }
  else {
    dir.mkdirs();
    System.out.printf("Created directory %s.", dirName);

  }
}
```

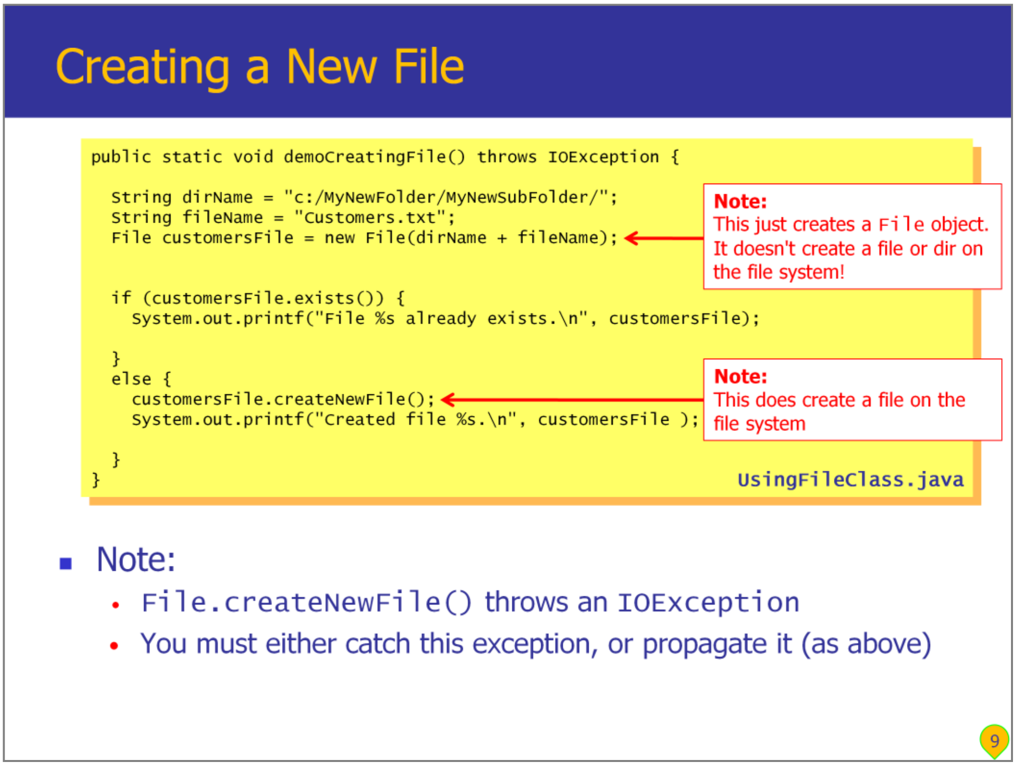
Note:
This just creates a `File` object. It doesn't create a file or dir on the file system!

Note:
This does create dirs on the file system

UsingFileClass.java

This slide shows how to use the `File` class to work with directories. Note that we can use a `File` class to represent a directory or a file; there isn't a separate `Directory` class.

The code first checks if the specified directory already exists. If it doesn't, the code proceeds to create the directory (plus all intermediate directories) by using the `mkdirs()` method.



Creating a New File

```
public static void demoCreatingFile() throws IOException {

  String dirName = "c:/MyNewFolder/MyNewSubFolder/";
  String fileName = "Customers.txt";
  File customersFile = new File(dirName + fileName);

  if (customersFile.exists()) {
    System.out.printf("File %s already exists.\n", customersFile);

  }
  else {
    customersFile.createNewFile();
    System.out.printf("Created file %s.\n", customersFile );

  }
}
```

Note: This just creates a `File` object. It doesn't create a file or dir on the file system!

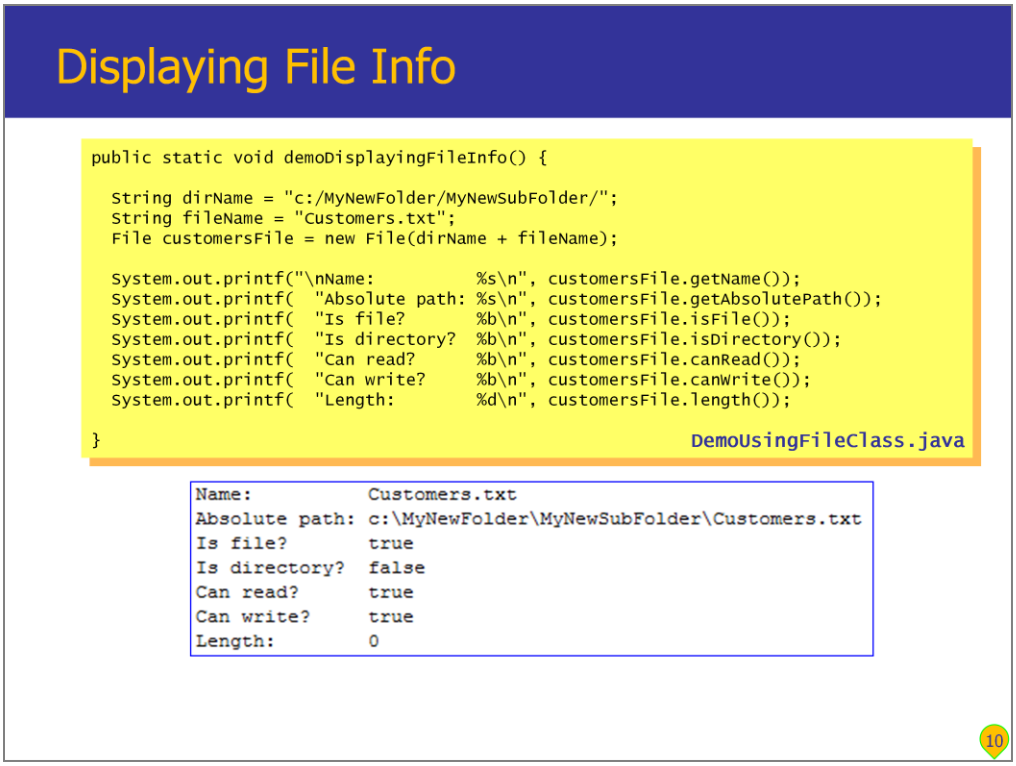
Note: This does create a file on the file system

UsingFileClass.java

- Note:
  - `File.createNewFile()` throws an `IOException`
  - You must either catch this exception, or propagate it (as above)

In a similar vein, this example shows how to use the `File` class to create a file.

Note that the `createNewFile()` method can throw an `IOException`, so you must either catch this exception yourself, or decorate your method with a `throws` clause to indicate to the calling method that it must handle the exception.



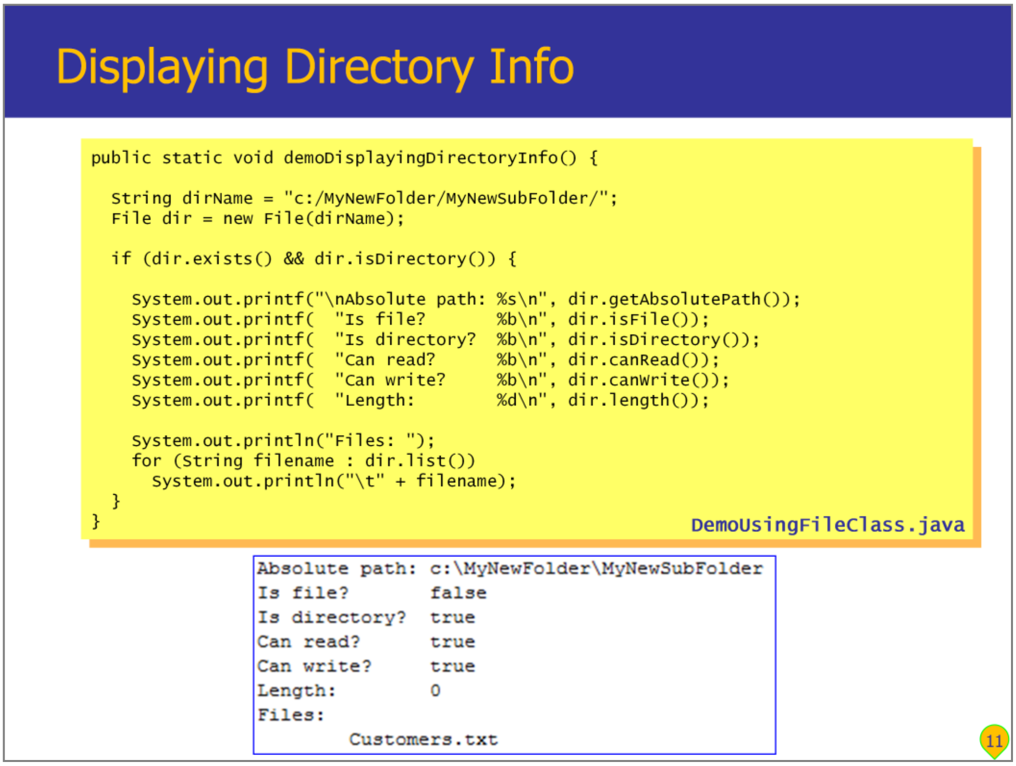
Displaying File Info

```
public static void demoDisplayingFileInfo() {

  String dirName = "c:/MyNewFolder/MyNewSubFolder/";
  String fileName = "Customers.txt";
  File customersFile = new File(dirName + fileName);

  System.out.printf("\nName:          %s\n", customersFile.getName());
  System.out.printf(  "Absolute path: %s\n", customersFile.getAbsolutePath());
  System.out.printf(  "Is file?       %b\n", customersFile.isFile());
  System.out.printf(  "Is directory?  %b\n", customersFile.isDirectory());
  System.out.printf(  "Can read?      %b\n", customersFile.canRead());
  System.out.printf(  "Can write?     %b\n", customersFile.canWrite());
  System.out.printf(  "Length:        %d\n", customersFile.length());

}
```

DemoUsingFileClass.java

```
Name:          Customers.txt
Absolute path: c:\MyNewFolder\MyNewSubFolder\Customers.txt
Is file?       true
Is directory?  false
Can read?      true
Can write?     true
Length:        0
```

In this example, we show how to use the `File` class to obtain information about an existing file on the file system. You can use these methods to interrogate the system for information about a file, before you proceed to perform operations upon the file.



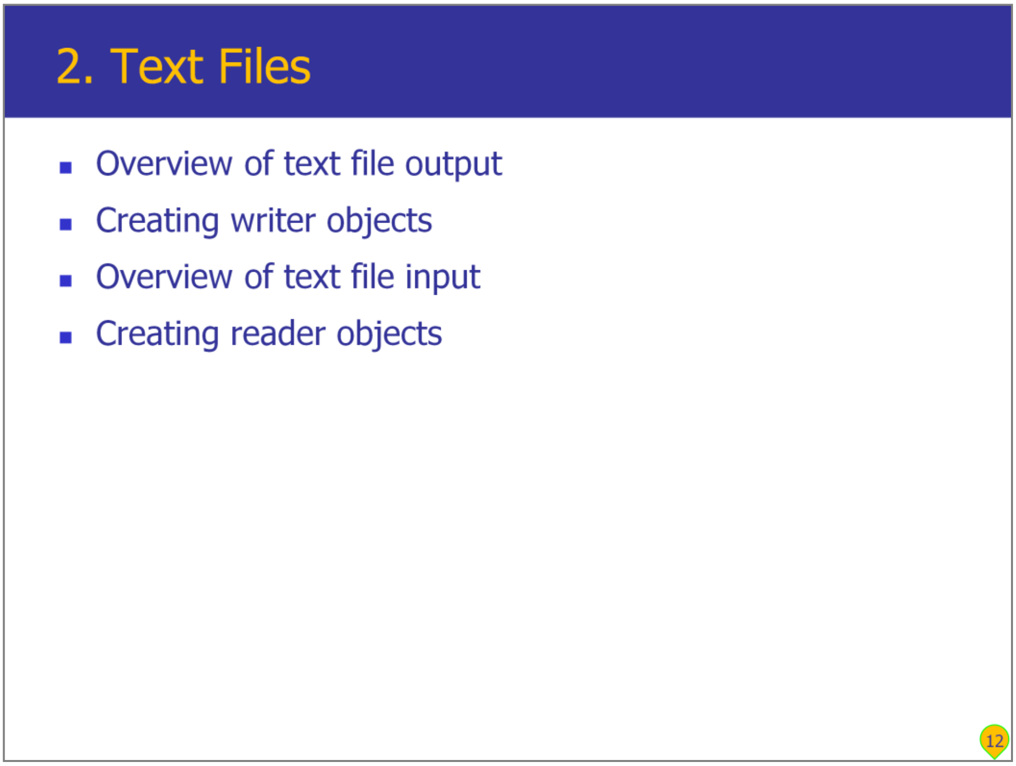
Displaying Directory Info

```
public static void demoDisplayingDirectoryInfo() {

  String dirName = "c:/MyNewFolder/MyNewSubFolder/";
  File dir = new File(dirName);

  if (dir.exists() && dir.isDirectory()) {

    System.out.printf("\nAbsolute path: %s\n", dir.getAbsolutePath());
    System.out.printf(  "Is file?       %b\n", dir.isFile());
    System.out.printf(  "Is directory?  %b\n", dir.isDirectory());
    System.out.printf(  "Can read?      %b\n", dir.canRead());
    System.out.printf(  "Can write?     %b\n", dir.canWrite());
    System.out.printf(  "Length:        %d\n", dir.length());

    System.out.println("Files: ");
    for (String filename : dir.list())
      System.out.println("\t" + filename);
  }
}
```

DemoUsingFileClass.java

```
Absolute path: c:\MyNewFolder\MyNewSubFolder
Is file?       false
Is directory?  true
Can read?      true
Can write?     true
Length:        0
Files:
        Customers.txt
```

This example is similar to the previous slide, except that it obtains information about a directory rather than a file. Notice that most of this code is the same as on the previous slide, i.e. we're using the `File` class again.



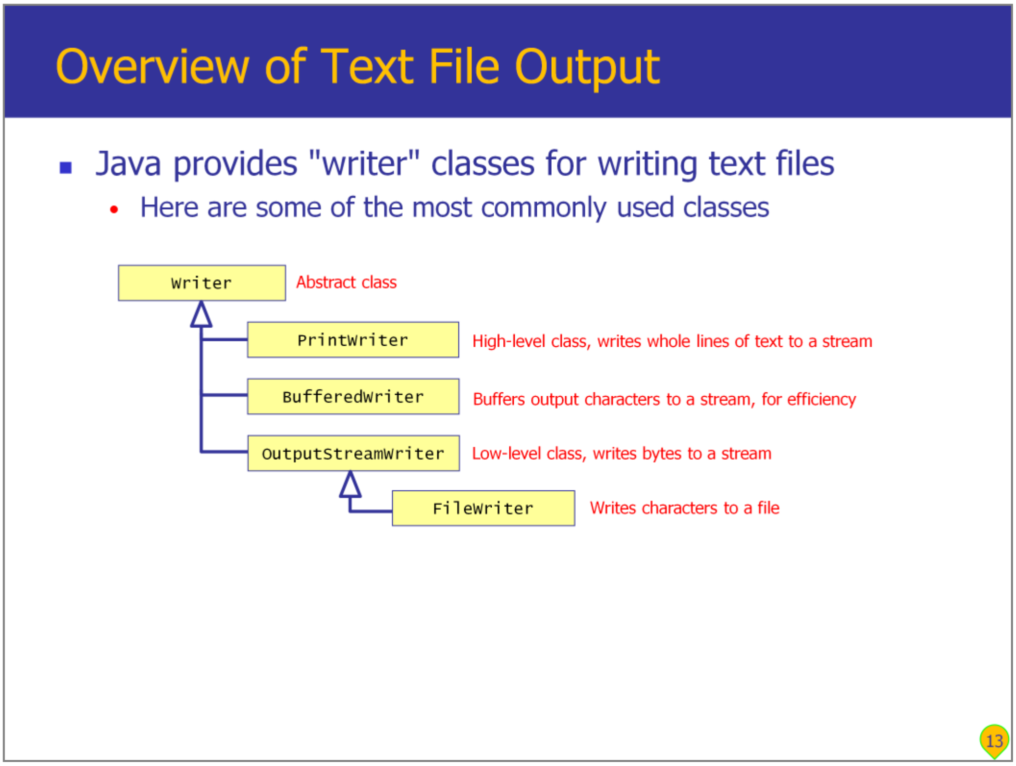
2. Text Files

- Overview of text file output
- Creating writer objects
- Overview of text file input
- Creating reader objects

12

If you want to read or write the content of a text file, you must create a reader or writer object. The reader or writer object provides methods that allow you to get at the content of the file; this is in stark contrast to the `File` class discussed in the previous section, which just lets you manipulate the file entity on the file system without accessing its content.

Note: if you want to read or write the content of a binary file, you must use a stream object. We discuss stream objects in the next section.



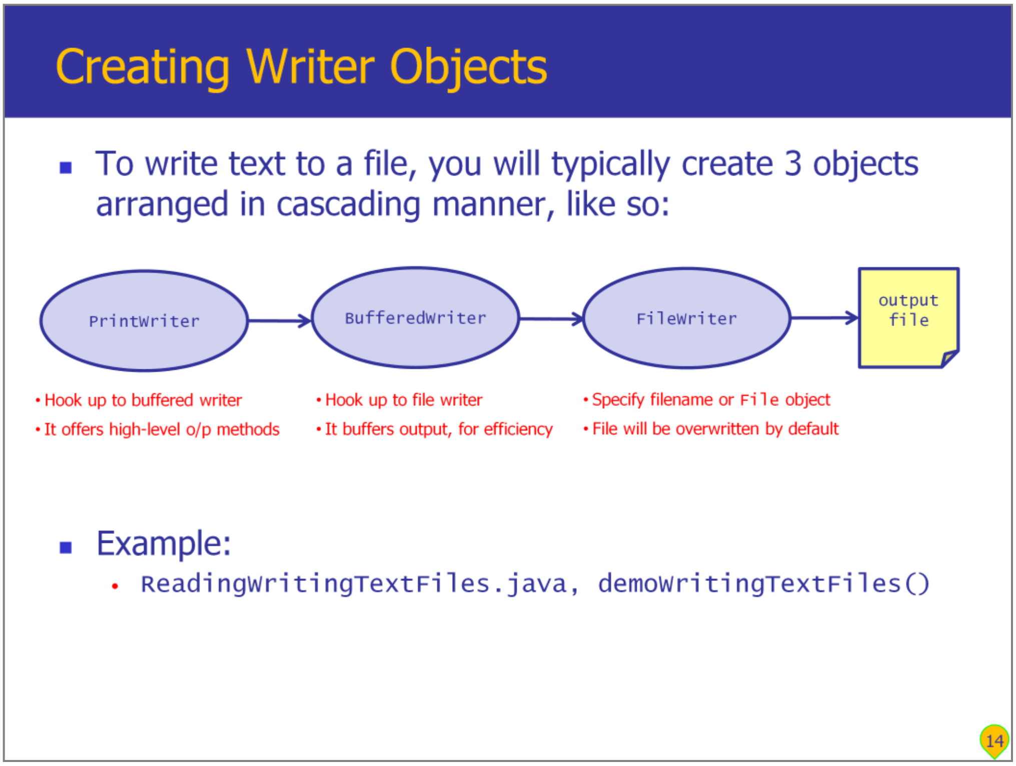
Overview of Text File Output

- Java provides "writer" classes for writing text files
  - Here are some of the most commonly used classes

Writer
Abstract class
PrintWriter
High-level class, writes whole lines of text to a stream
BufferedWriter
Buffers output characters to a stream, for efficiency
OutputStreamWriter
Low-level class, writes bytes to a stream
FileWriter
Writes characters to a file

The `java.io` package contains various writer classes that allow you to write text content to a file. These classes are arranged in a hierarchy as shown in the slide above; the `Writer` base class defines basic capabilities, and then each subclass adds additional capabilities.

The following slide shows how to use these classes in a typical scenario.



Creating Writer Objects

- To write text to a file, you will typically create 3 objects arranged in cascading manner, like so:

PrintWriter → BufferedWriter → FileWriter → output file

- Hook up to buffered writer
- It offers high-level o/p methods

- Hook up to file writer
- It buffers output, for efficiency

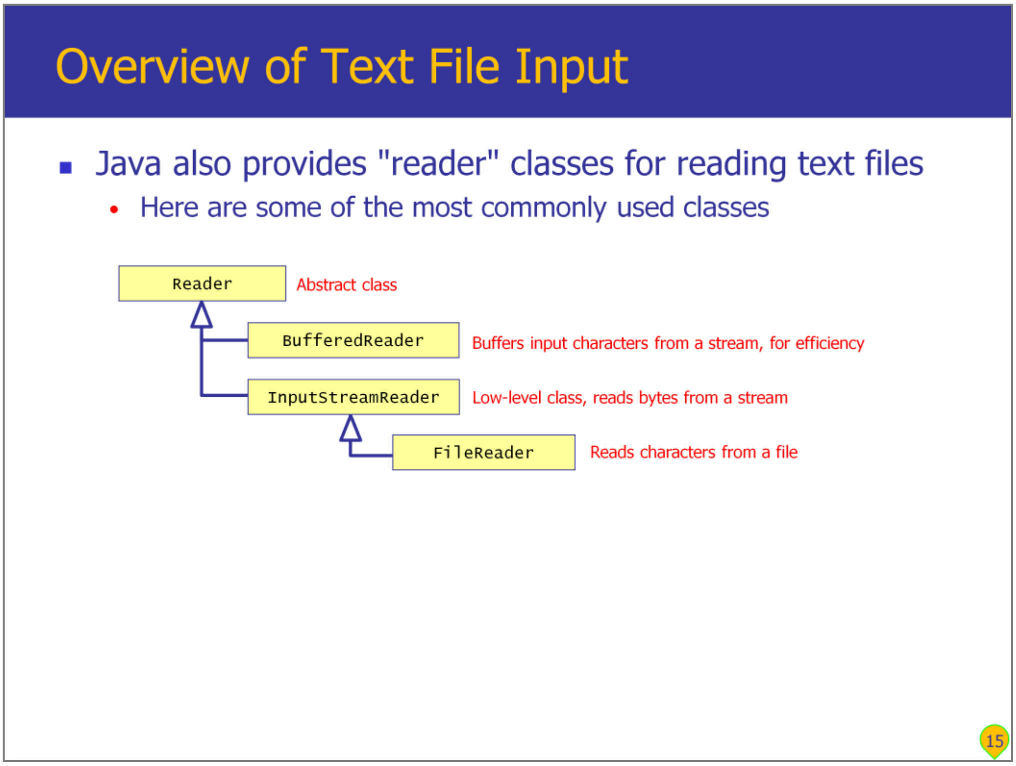
- Specify filename or `File` object
- File will be overwritten by default

- Example:
  - `ReadingWritingTextFiles.java, demoWritingTextFiles()`

The writer classes are a good example of the Decorator pattern. According to this pattern, you can create a series of objects that hook up to each other; each object adds another layer of capability, over-and-above what's on offer from the existing object in the chain.

Here's a description of what's going on in the slide, starting from the right-hand-side of the diagram:

- The first object we create is a `FileWriter` object, which will allow us to write to an output file. The `FileWriter` class is very primitive; it only provides methods to write strings and character arrays.
- Next up, we create a `BufferedWriter` object and initialize it with a reference to the existing `FileWriter` object. `BufferedWriter` has all the capabilities of `FileWriter`, but buffers content in memory for efficiency. This results in fewer disk I/O hits.
- Finally, we create a `PrintWriter` object and initialize it with a reference to the existing `BufferedWriter` object. `PrintWriter` offers a lot of extra methods that perform a much higher-level API for writing data to a file. For example, it has heavily overloaded `print()` and `println()` methods to print different types of value to the file.



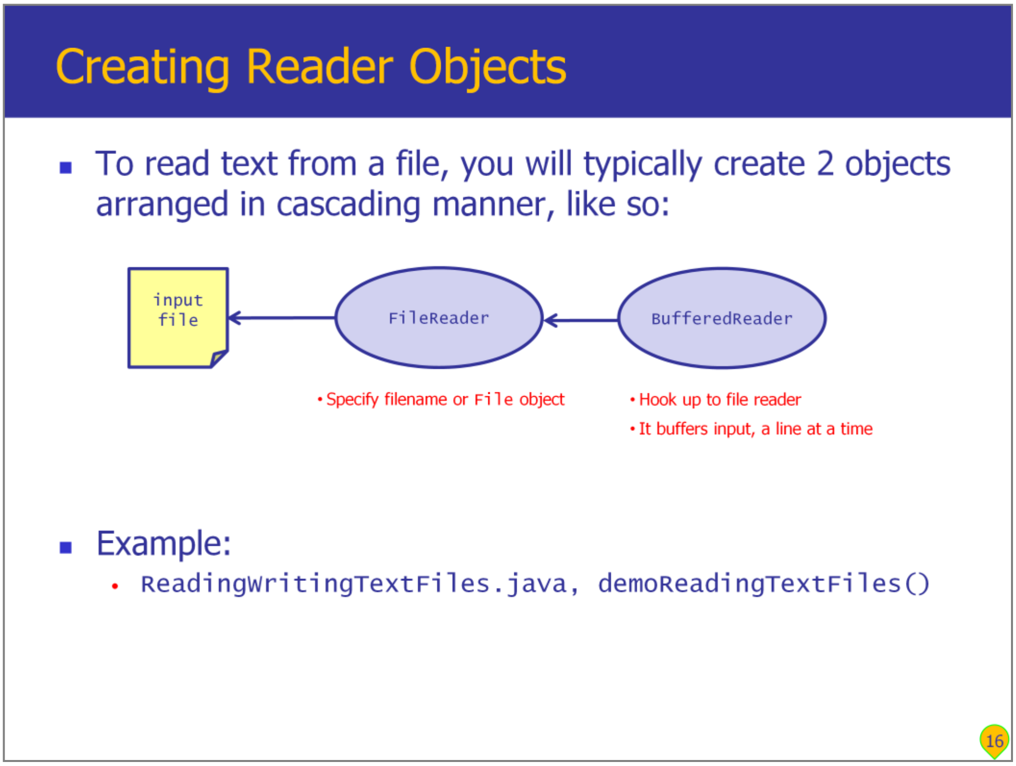
Overview of Text File Input

- Java also provides "reader" classes for reading text files
  - Here are some of the most commonly used classes

Reader
Abstract class
BufferedReader
Buffers input characters from a stream, for efficiency
InputStreamReader
Low-level class, reads bytes from a stream
FileReader
Reads characters from a file

As you would expect, the `java.io` package also contains various reader classes that allow you to read text content from a file. These classes are arranged in a hierarchy as shown in the slide above; the `Reader` base class defines basic capabilities, and then each subclass adds additional capabilities.

The following slide shows how to use these classes in a typical scenario.



Creating Reader Objects

- To read text from a file, you will typically create 2 objects arranged in cascading manner, like so:

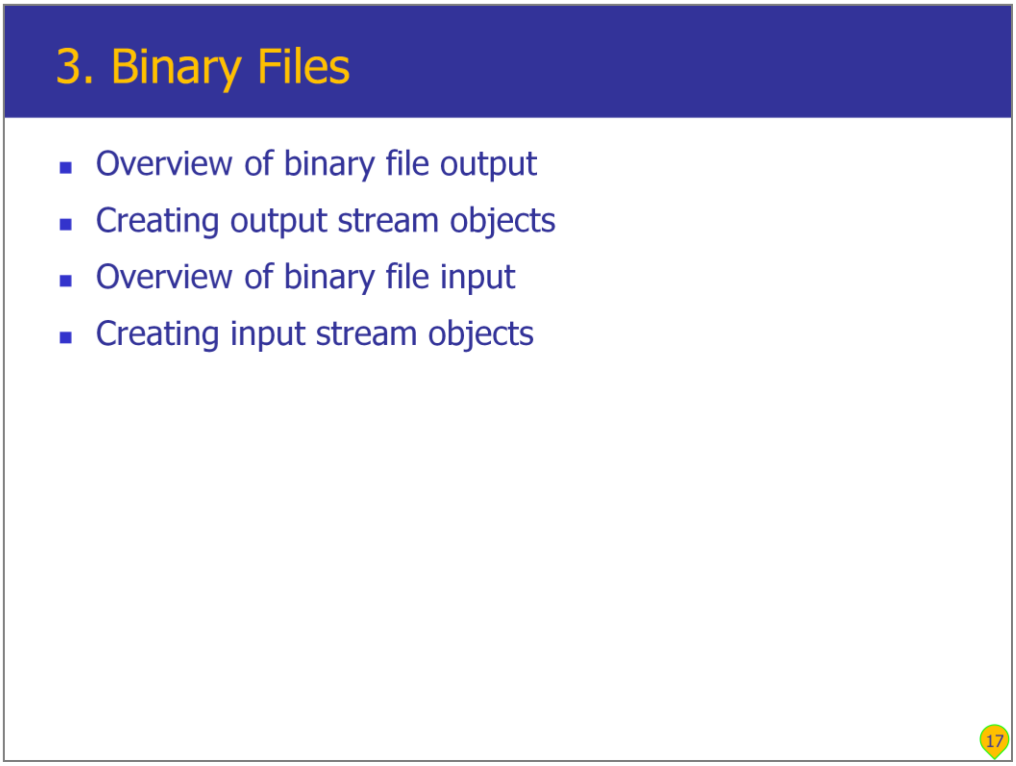
input file
FileReader
BufferedReader

- Specify filename or `File` object
- Hook up to file reader
- It buffers input, a line at a time

- Example:
  - `ReadingWritingTextFiles.java, demoReadingTextFiles()`

The reader classes are another good example of the Decorator pattern. Here's a description of what's going on in the slide, starting from the left-hand-side of the diagram this time:

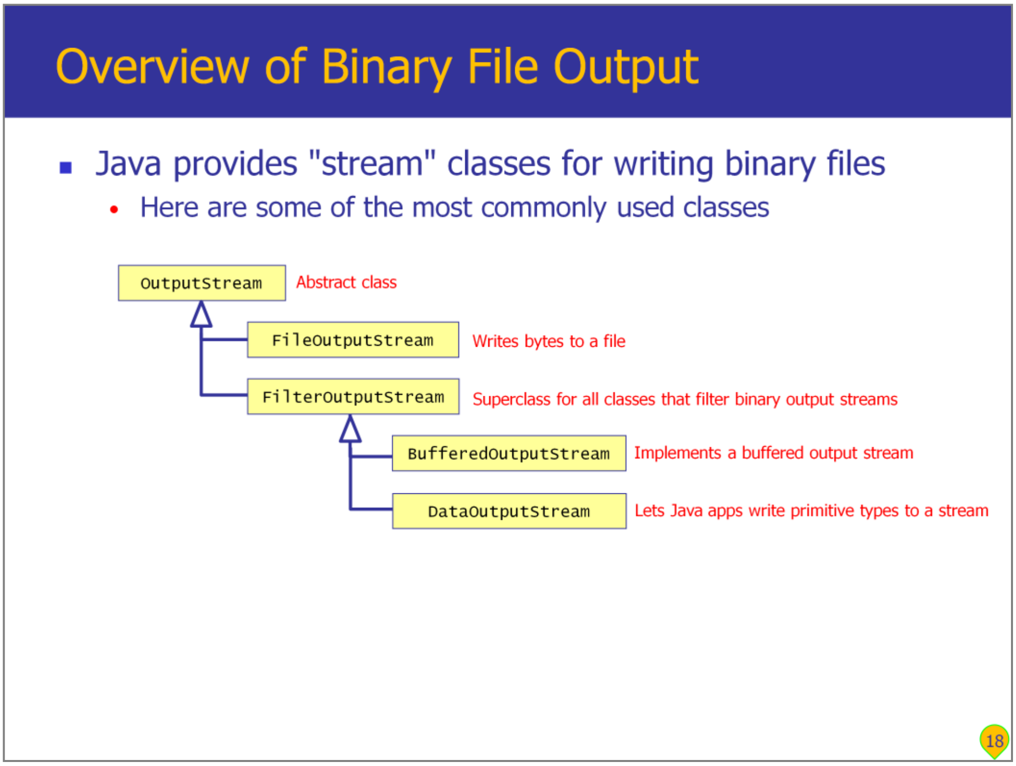
- The first object we create is a `FileReader` object, which will allow us to read from an existing file. The `FileReader` class is very primitive; it only provides methods to read characters.
- We then create a `BufferedReader` object and initialize it with a reference to the existing `FileReader` object. `BufferedReader` has all the capabilities of `FileReader`, but reads a whole line of text into memory at a time. The class has a `readLine()` method, which makes it easier to process a text file. The class also has methods that allow you to mark a position in a file, and then reset the current position in the file to the mark position.



3. Binary Files

- Overview of binary file output
- Creating output stream objects
- Overview of binary file input
- Creating input stream objects

This section shows how to read and write binary content in a file, by using stream classes in the `java.io` package.



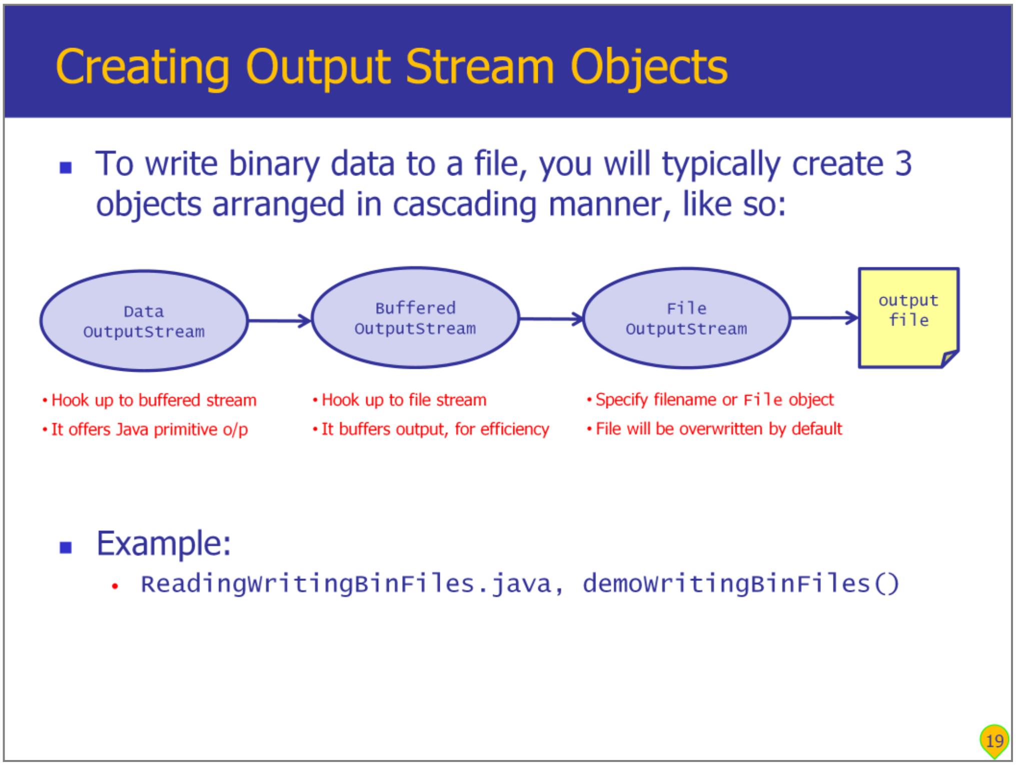
Overview of Binary File Output

- Java provides "stream" classes for writing binary files
  - Here are some of the most commonly used classes

OutputStream
Abstract class
FileOutputStream
Writes bytes to a file
FilterOutputStream
Superclass for all classes that filter binary output streams
BufferedOutputStream
Implements a buffered output stream
DataOutputStream
Lets Java apps write primitive types to a stream
18

The `java.io` package contains various "output stream" classes that allow you to write binary content to a file. These classes are arranged in a hierarchy as shown in the slide above; the `OutputStream` base class defines basic capabilities, and then each subclass adds additional capabilities.

The following slide shows how to use these classes in a typical scenario.



Creating Output Stream Objects

- To write binary data to a file, you will typically create 3 objects arranged in cascading manner, like so:

Data OutputStream → Buffered OutputStream → File OutputStream → output file

- Hook up to buffered stream
- It offers Java primitive o/p

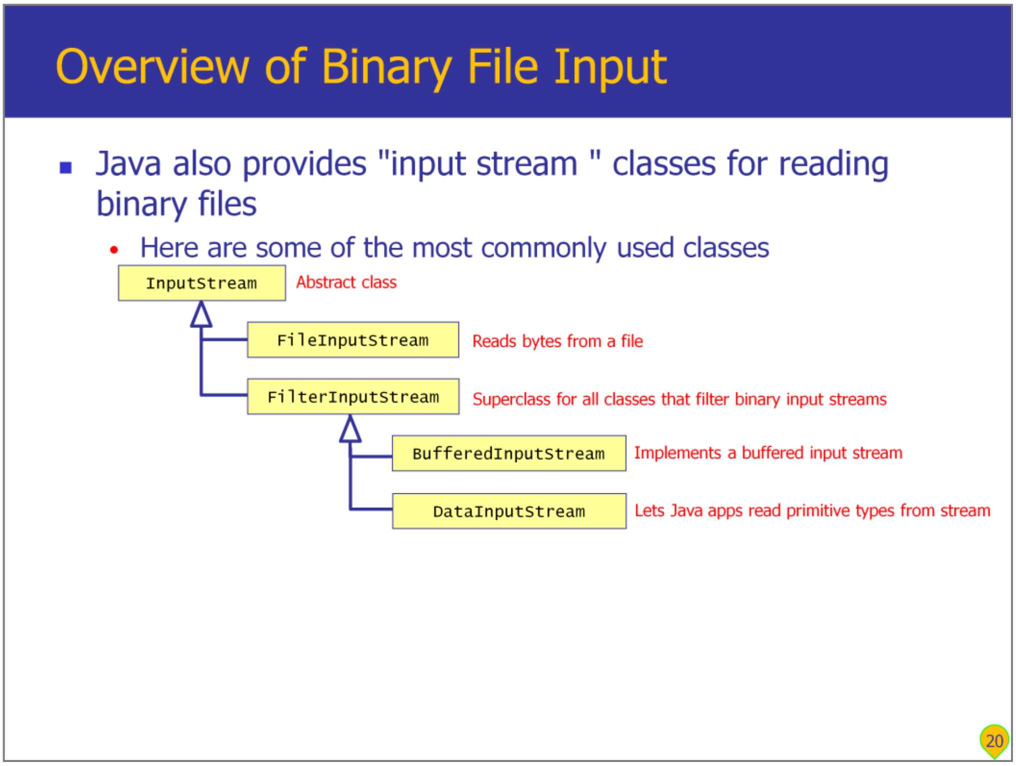
- Hook up to file stream
- It buffers output, for efficiency

- Specify filename or `File` object
- File will be overwritten by default

- Example:
  - `ReadingWritingBinFiles.java, demoWritingBinFiles()`

Here's a description of the objects involved in this diagram, starting from the right-hand-side of the diagram:

- The first object we create is a `FileOutputStream` object, which will allow us to write binary data to a file. We specify the file either by its filename or by a `File` object.
- Next up, we create a `BufferedOutputStream` object and initialize it with a reference to the existing `FileOutputStream` object. As you might expect, `BufferedOutputStream` has all the capabilities of `FileOutputStream`, but buffers content in memory for efficiency.
- Finally, we create a `DataOutputStream` object and initialize it with a reference to the `BufferedOutputStream` object. `DataOutputStream` offers a lot of extra methods that perform a much higher-level API for writing primitive types to a file. For example, it has methods such `writeBoolean()`, `writeByte()`, `writeShort()`, `writeInt()`, etc.



Overview of Binary File Input

- Java also provides "input stream " classes for reading binary files
  - Here are some of the most commonly used classes

`InputStream` Abstract class

`FileInputStream` Reads bytes from a file

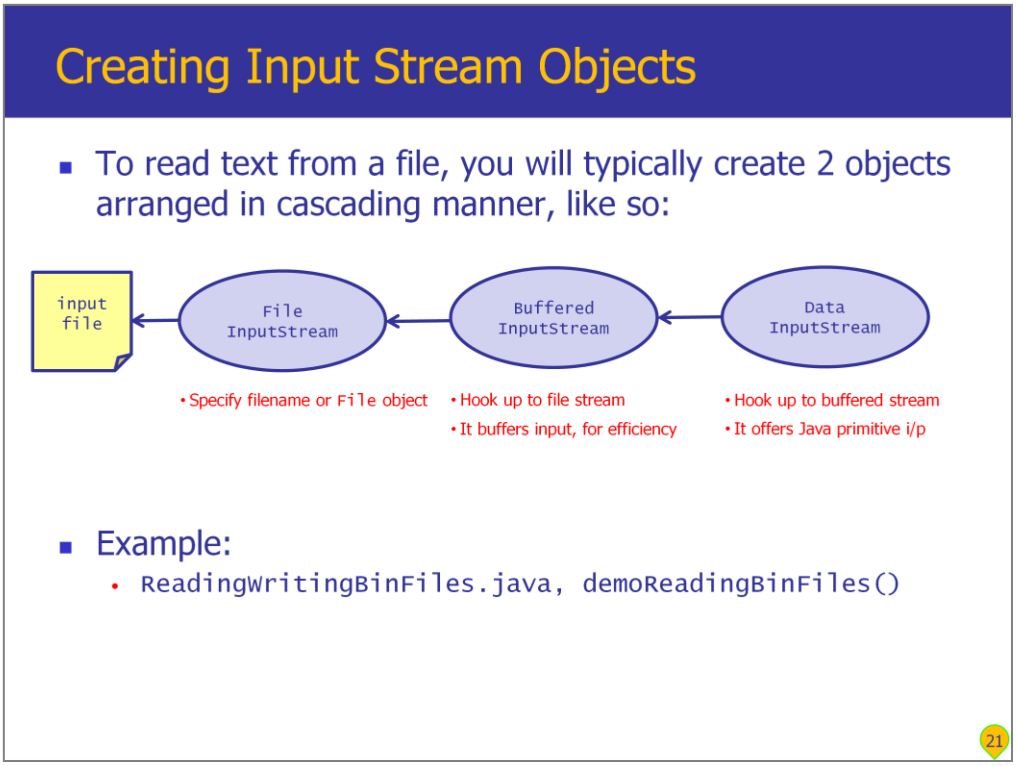
`FilterInputStream` Superclass for all classes that filter binary input streams

`BufferedInputStream` Implements a buffered input stream

`DataInputStream` Lets Java apps read primitive types from stream

The `java.io` package also contains various "input stream" classes that allow you to read binary content from a file. These classes are arranged in a hierarchy as shown in the slide above; the `InputStream` base class defines basic capabilities, and then each subclass adds additional capabilities.

The following slide shows how to use these classes in a typical scenario.



Creating Input Stream Objects

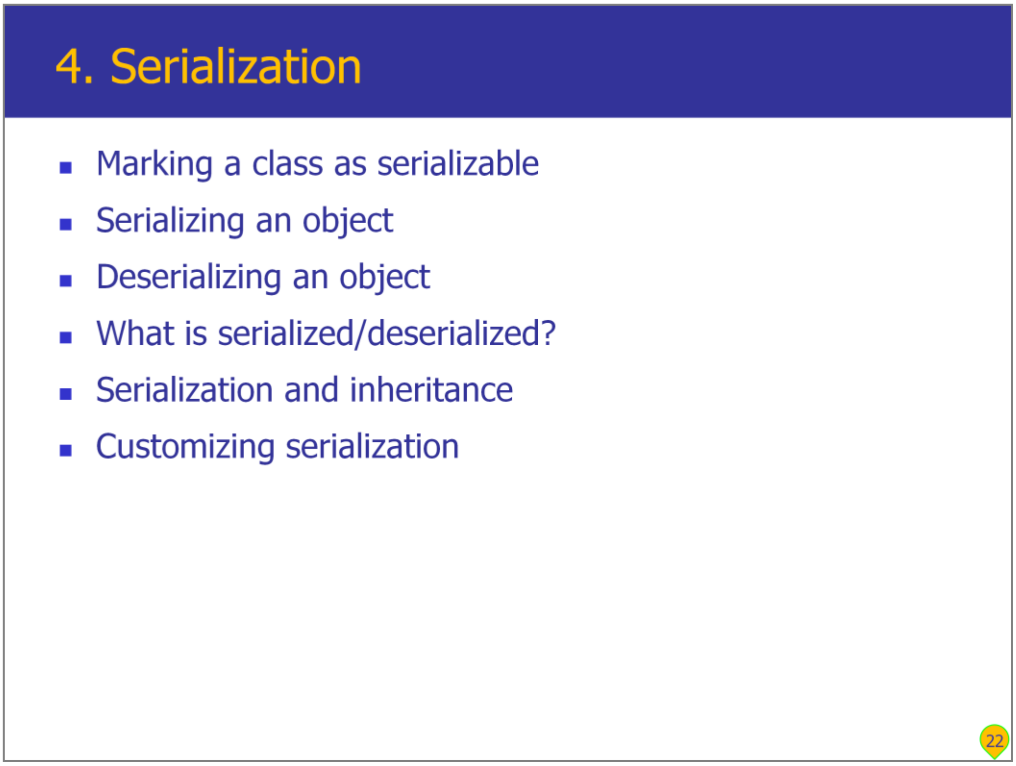
- To read text from a file, you will typically create 2 objects arranged in cascading manner, like so:

- Specify filename or `File` object
- Hook up to file stream
- It buffers input, for efficiency
- Hook up to buffered stream
- It offers Java primitive i/p

- Example:
  - `ReadingWritingBinFiles.java, demoReadingBinFiles()`

Here's a description of the objects involved in this diagram, starting from the left-hand-side of the diagram:

- The first object we create is a `FileInputStream` object, which will allow us to read binary data from a file. As before, we specify the file either by its filename or by a `File` object.
- Next up, we create a `BufferedInputStream` object and initialize it with a reference to the existing `FileInputStream` object. This gives us the performance benefit of buffering.
- Finally, we create a `DataInputStream` object and initialize it with a reference to the `BufferedInputStream` object. `DataInputStream` has methods for reading primitive types from a file. For example, it has methods such `readBoolean()`, `readByte()`, `readShort()`, `readInt()`, etc.

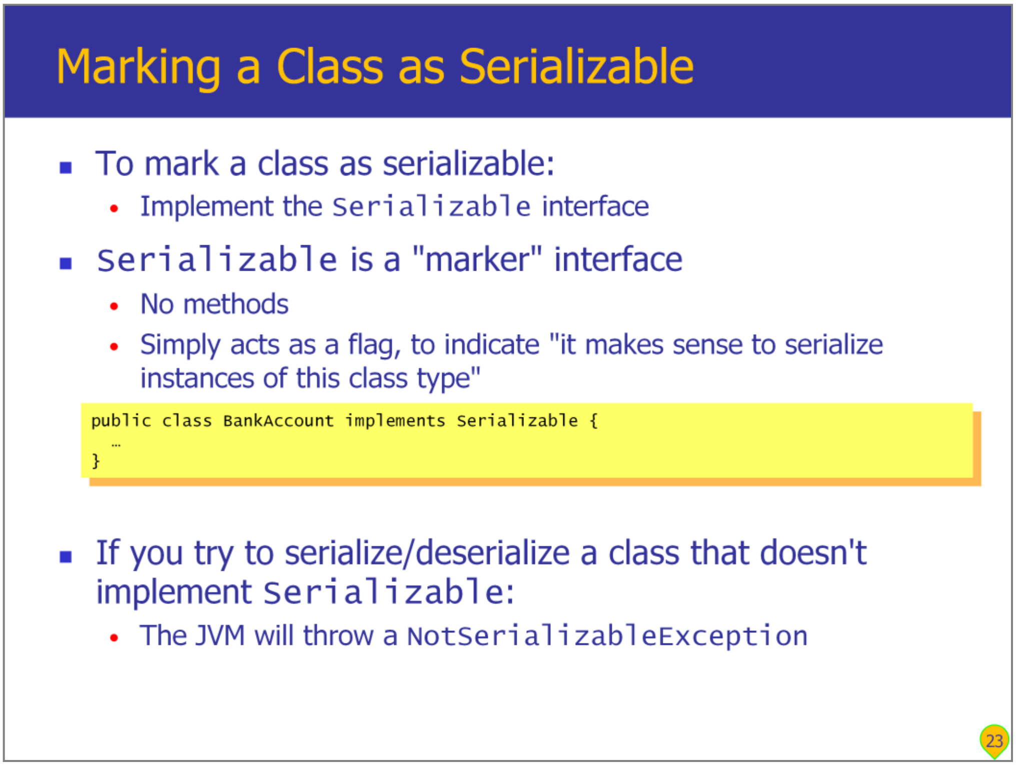


4. Serialization

- Marking a class as serializable
- Serializing an object
- Deserializing an object
- What is serialized/deserialized?
- Serialization and inheritance
- Customizing serialization

Java provides automatic support for object serialization and deserialization. Serialization is the process whereby an object is written out as a byte stream, and deserialization is the process whereby a byte stream is read back in again, to recreate a copy of the original object.

Serialization and deserialization are useful for transmitting objects over networks, between Java apps, and for saving and restoring objects in web applications.



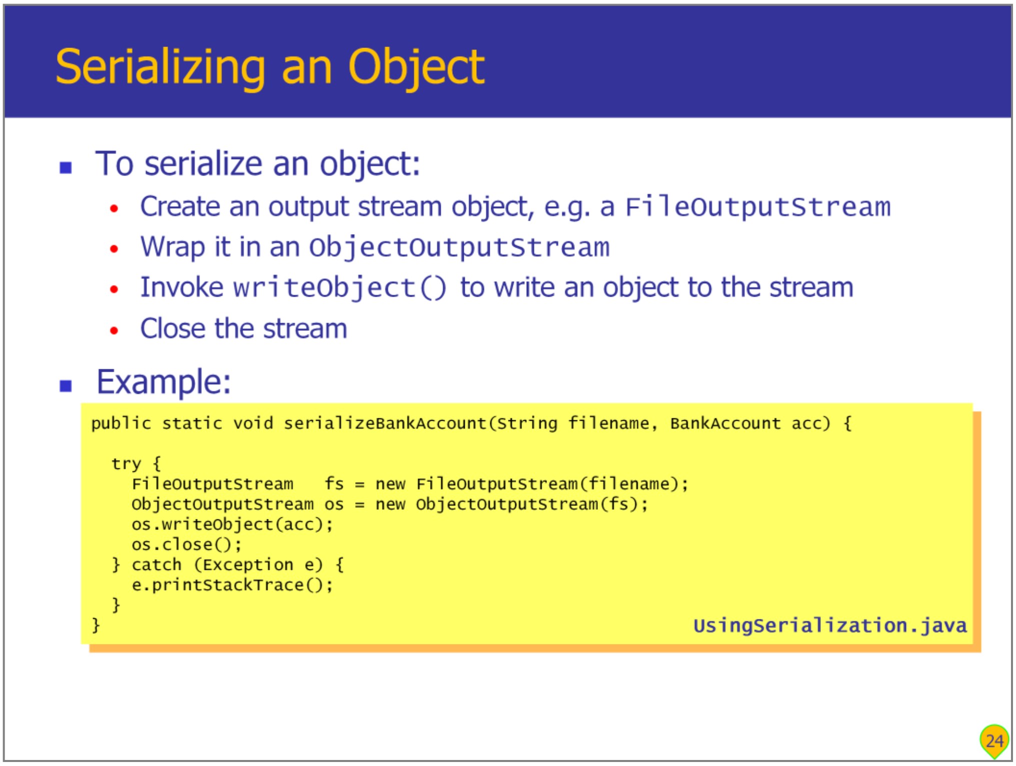
Marking a Class as Serializable

- To mark a class as serializable:
  - Implement the `Serializable` interface
- `Serializable` is a "marker" interface
  - No methods
  - Simply acts as a flag, to indicate "it makes sense to serialize instances of this class type"

```
public class BankAccount implements Serializable {
  …
}
```

- If you try to serialize/deserialize a class that doesn't implement `Serializable`:
  - The JVM will throw a `NotSerializableException`

To indicate that the JVM is allowed to serialize/deserialize instances of a class, the class must implement the `Serializable` interface. This is a marker interface; it has no members, so you don't need to implement any specific behaviour in your class.



Serializing an Object

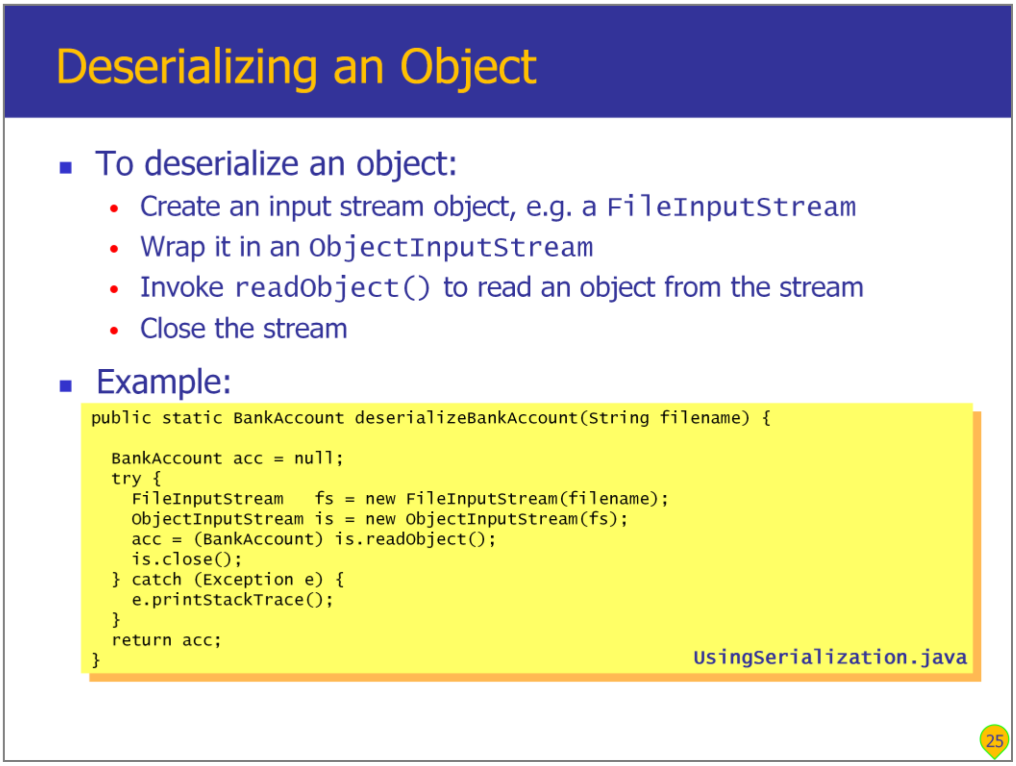
- To serialize an object:
  - Create an output stream object, e.g. a `FileOutputStream`
  - Wrap it in an `ObjectOutputStream`
  - Invoke `writeObject()` to write an object to the stream
  - Close the stream
- Example:

```
public static void serializeBankAccount(String filename, BankAccount acc) {

  try {
    FileOutputStream   fs = new FileOutputStream(filename);
    ObjectOutputStream os = new ObjectOutputStream(fs);
    os.writeObject(acc);
    os.close();
  } catch (Exception e) {
    e.printStackTrace();
  }
}
```

UsingSerialization.java

This slide shows how to serialize an object to a byte stream. In this example, the byte stream is actually written to a file, although this isn't strictly necessary.

Note the use of the `ObjectOutputStream` class, which has a `writeObject()` method to serialize the specified object. If the object in question doesn't implement the `Serializable` interface, a `NotSerializableException` will occur at this point.



Deserializing an Object

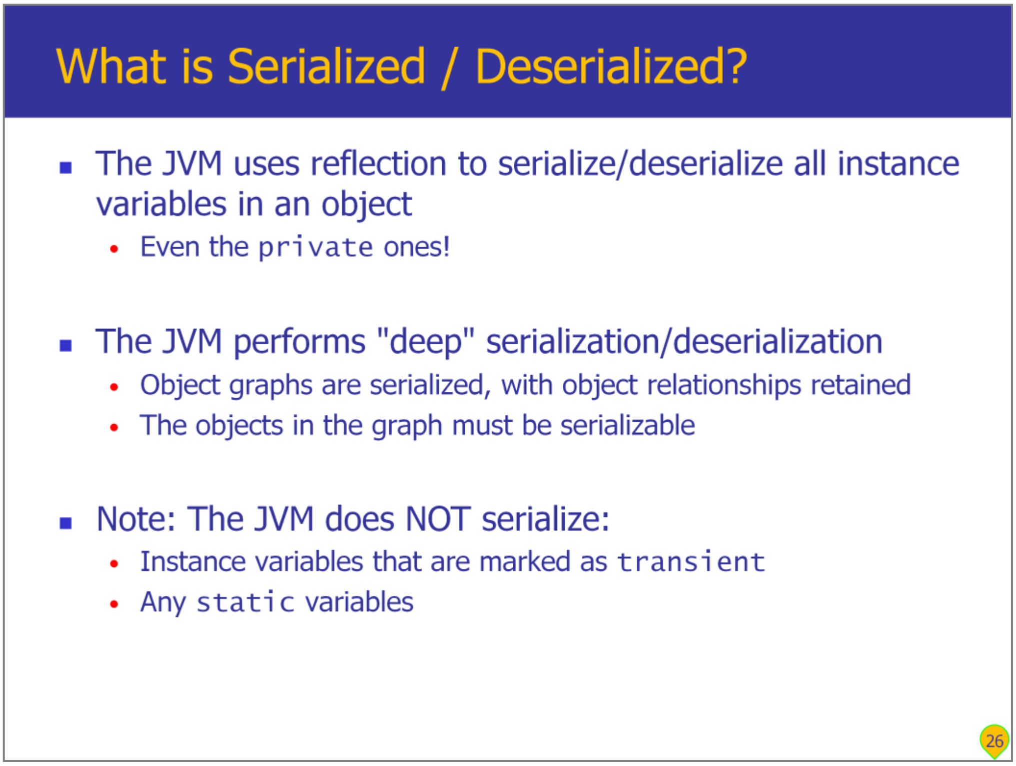
- To deserialize an object:
  - Create an input stream object, e.g. a `FileInputStream`
  - Wrap it in an `ObjectInputStream`
  - Invoke `readObject()` to read an object from the stream
  - Close the stream
- Example:

```
public static BankAccount deserializeBankAccount(String filename) {

  BankAccount acc = null;
  try {
    FileInputStream   fs = new FileInputStream(filename);
    ObjectInputStream is = new ObjectInputStream(fs);
    acc = (BankAccount) is.readObject();
    is.close();
  } catch (Exception e) {
    e.printStackTrace();
  }
  return acc;
}
```

`UsingSerialization.java`

This slide shows how to deserialize an object from a byte stream. In this example, the byte stream is retrieved from a file, although this isn't strictly necessary.

Note the use of the `ObjectInputStream` class, which has a `readObject()` method to deserialize a byte stream to create a copy of the original object. The `readObject()` method's stated return type is `Object`, so we must cast this to the correct type of object that we expect to retrieve from the stream.



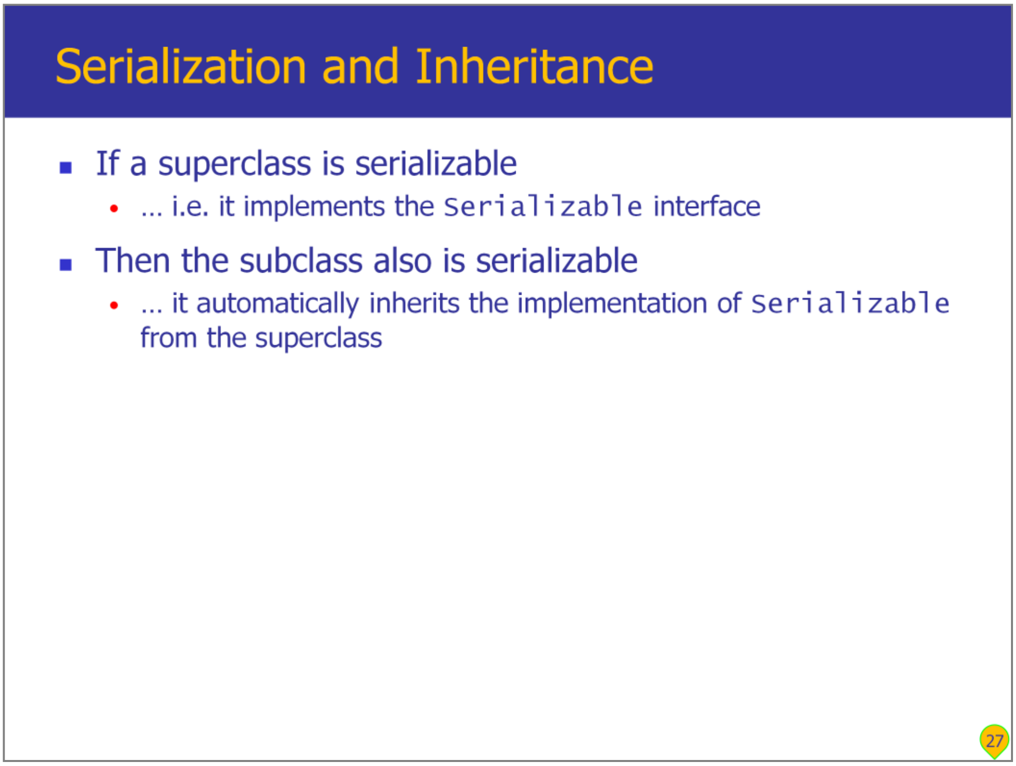
What is Serialized / Deserialized?

- The JVM uses reflection to serialize/deserialize all instance variables in an object
  - Even the `private` ones!
- The JVM performs "deep" serialization/deserialization
  - Object graphs are serialized, with object relationships retained
  - The objects in the graph must be serializable
- Note: The JVM does NOT serialize:
  - Instance variables that are marked as `transient`
  - Any `static` variables

Under the covers, the Java serialization mechanism serializes all the instance variables in an object, even those that are declared as `private`. Furthermore, the mechanism is robust enough to handle complex object graphs, where an object has references to other objects.

If there is a data member in your class that doesn't need to be serialized, you must declare the instance variable as `transient`. This indicates the variable holds temporary data that probably won’t have any relevance when the object is deserialized in the future.

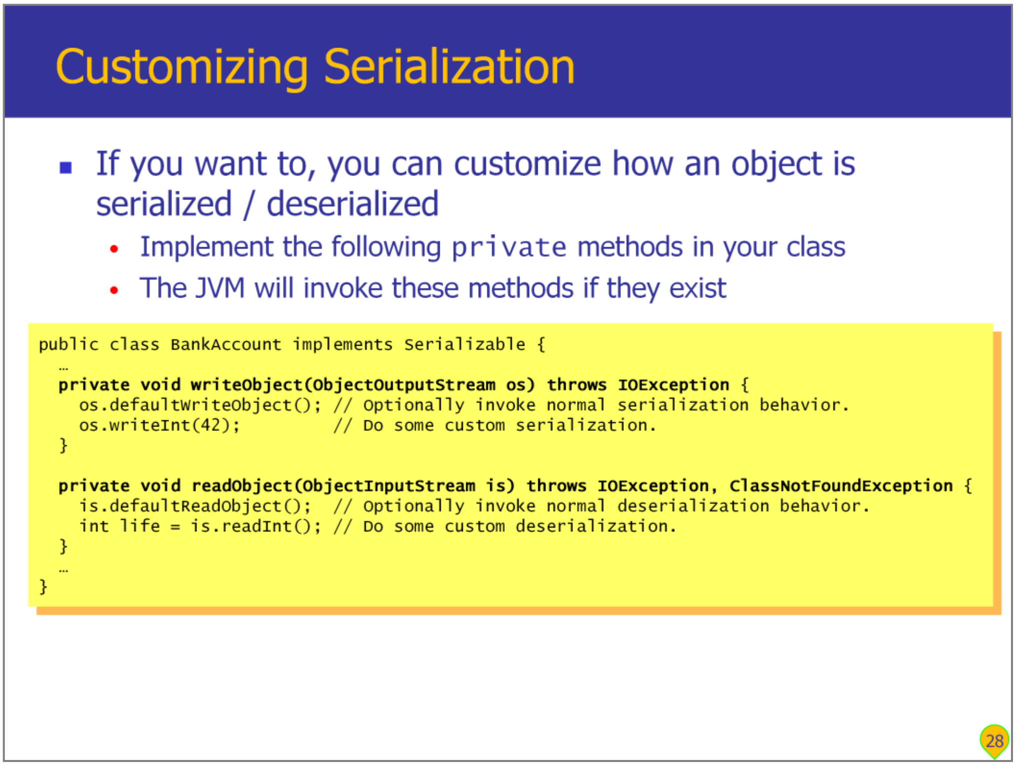
Also note that `static` members are not serialized.



Serialization and Inheritance

- If a superclass is serializable
  - … i.e. it implements the `Serializable` interface
- Then the subclass also is serializable
  - … it automatically inherits the implementation of `Serializable` from the superclass

This slide summarizes how serialization works if you have an inheritance hierarchy. Basically, if you declare that the superclass implements `Serializable`, then all the subclasses will also be serializable as well.



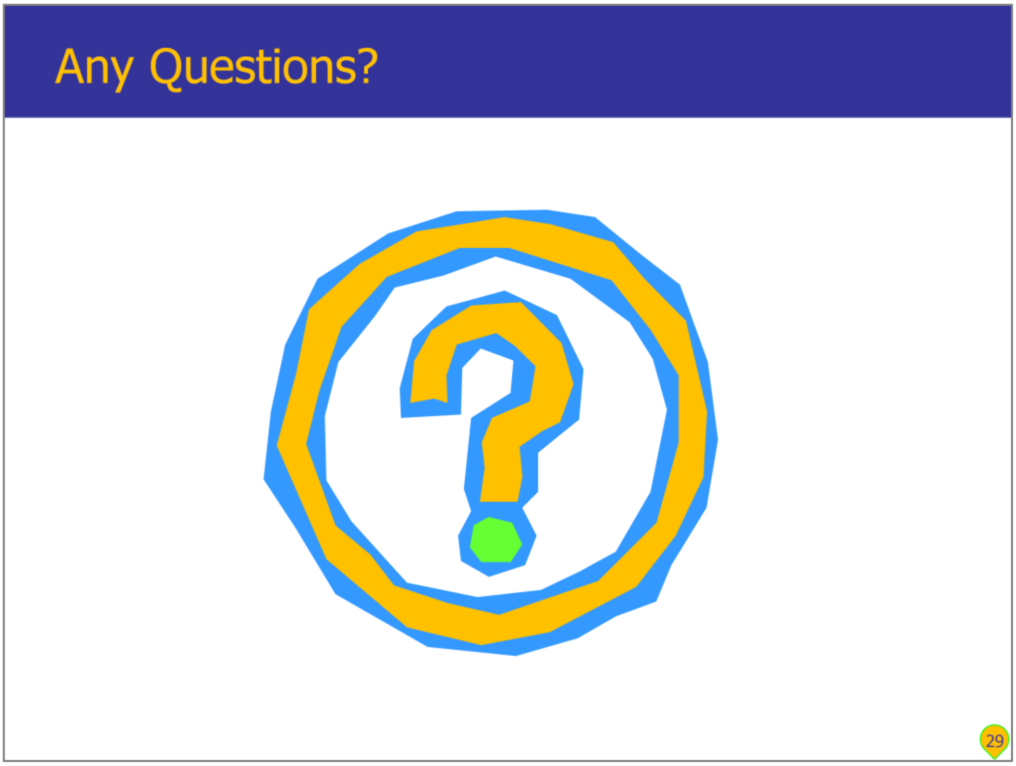
Customizing Serialization

- If you want to, you can customize how an object is serialized / deserialized
  - Implement the following `private` methods in your class
  - The JVM will invoke these methods if they exist

```
public class BankAccount implements Serializable {
  …
  private void writeObject(ObjectOutputStream os) throws IOException {
    os.defaultWriteObject(); // Optionally invoke normal serialization behavior.
    os.writeInt(42);         // Do some custom serialization.
  }

  private void readObject(ObjectInputStream is) throws IOException, ClassNotFoundException {
    is.defaultReadObject();  // Optionally invoke normal deserialization behavior.
    int life = is.readInt(); // Do some custom deserialization.
  }
  …
}
```

To finish the chapter, here's an advanced technique… You can customize the serialization and deserialization mechanism for a class. This is an optimization measure, if you have some inside knowledge of how to perform serialization and deserialization more efficiently than the built-in mechanism.

Note: You shouldn't need to do this very often. Only consider doing this if you're sure the built-in mechanism is too slow.



Any Questions?

29