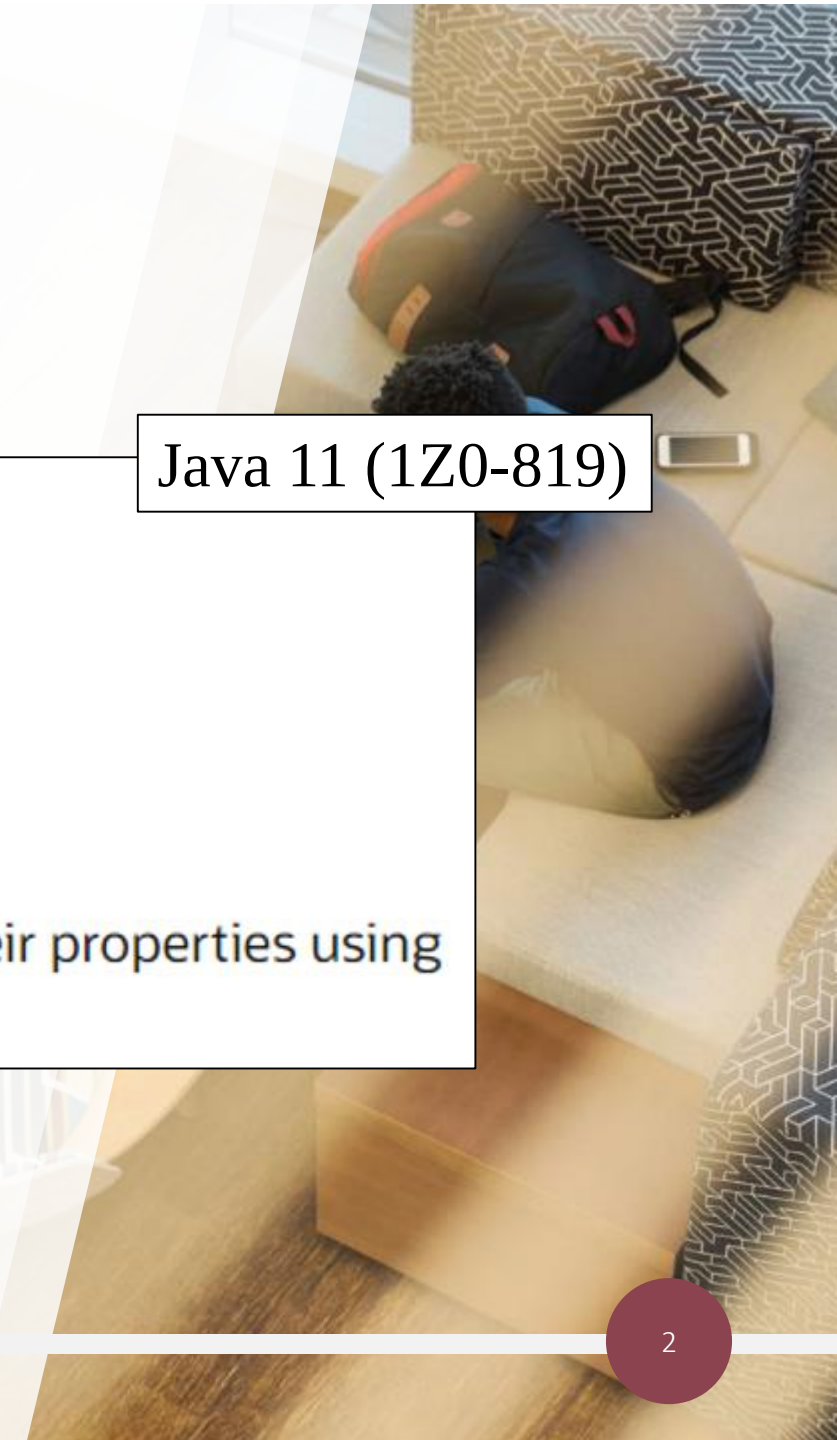
IO and NIO.2

IO and NIO.2



Java 11 (1Z0-819)

Use Java I/O API

- ✓ Read and write console and file data using I/O Streams
- ✓ Serialize and de-serialize Java objects
- ✓ Construct, traverse, create, read, and write Path objects and their properties using java.nio.file API

IO

- This first section relates to the *java.io* package and is called “Java I/O Fundamentals”.

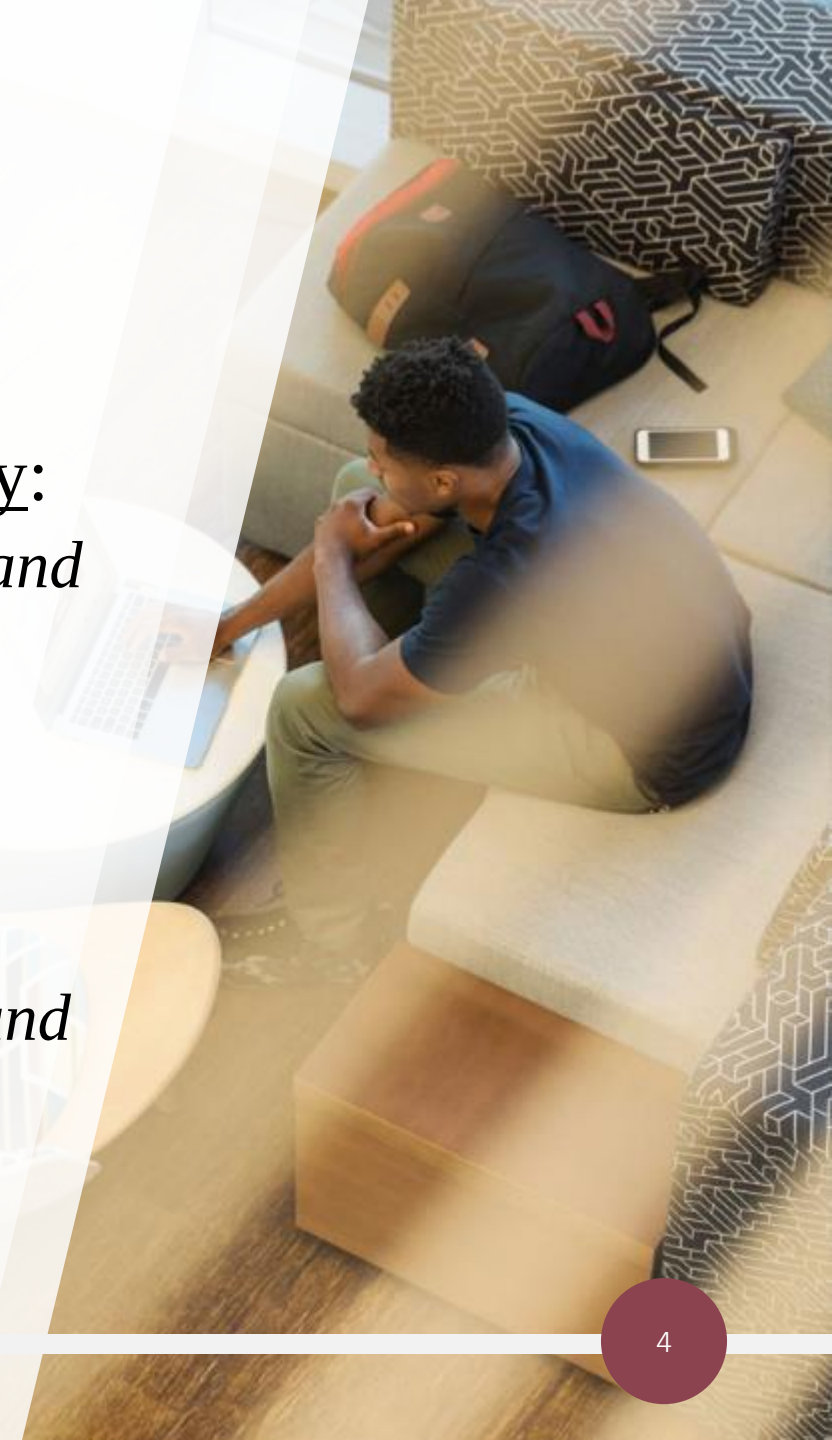
- The second section relates to the *java.nio.** packages and is called “NIO.2”. NIO.2 relates to the key features introduced in Java 7 residing in two packages :
  - *java.nio.file*
  - *java.nio.attribute*

Java IO Fundamentals

- Java reads/writes characters or bytes.

- Classes with “Stream” in their name read/writes binary:
  - *FileInputStream*, *FileOutputStream*, *ObjectInputStream and ObjectOutputStream*.
  - used for images or executable files

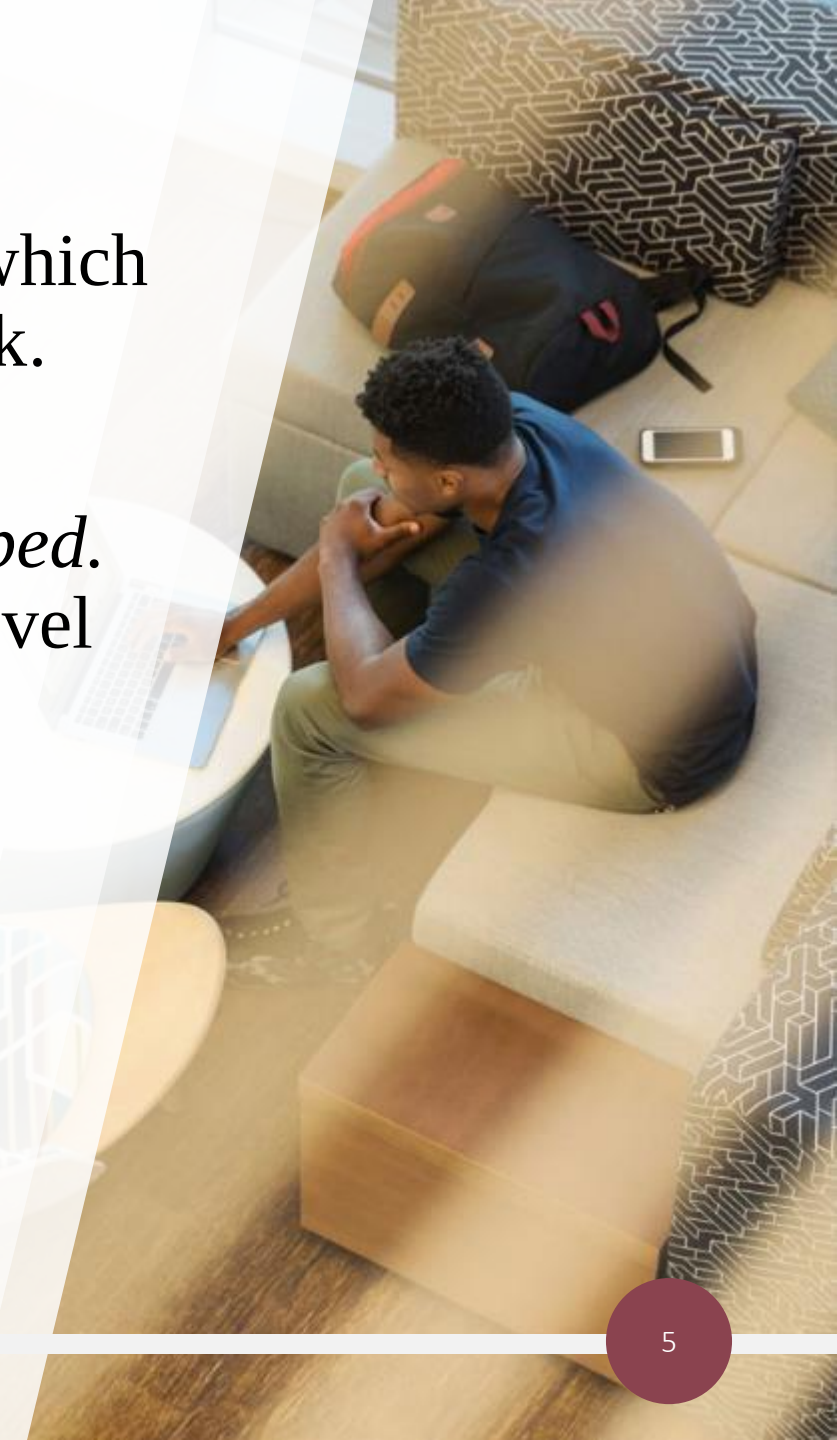
- *Reader*s and *Writer*s are used to read/write text.
  - *FileReader, BufferedReader, FileWriter, BufferedWriter and PrintWriter*.
  - used for text files i.e. character or string data



Java IO Fundamentals

- *File* is the class that enables you create objects from which you can then create actual physical files on the hard disk.

- Many of the previous classes are intended to be *wrapped*. This enables low-level classes to get access to higher-level functionality. This is for efficiency e.g. buffering.



IO

Abstract classes	Low-level classes	High-level classes (for efficiency)	High-level classes (other)	Example
InputStream	FileInputStream (reads one byte at a time)	BufferedInputStream	ObjectInputStream	new ObjectInputStream(new FileInputStream("f.dat"));
OutputStream	FileOutputStream	BufferedOutputStream	ObjectOutputStream PrintStream	new ObjectOutputStream(new FileOutputStream("f.dat"));
Reader	FileReader	BufferedReader	InputStreamReader (bridge between byte streams and character streams)	new BufferedReader(new FileReader("in.txt")); new BufferedReader(new InputStreamReader(System.in));
Writer	FileWriter	BufferedWriter	PrintWriter OutputStreamWriter (bridge between character streams and byte streams)	new BufferedWriter(new FileWriter("in.txt")); new BufferedWriter(new OutputStreamWriter (System.out));

- ReadingWritingExamples.java

Console class

- The *Console* class is specifically designed for interacting with the user.

- *Console* is a singleton where you use a factory method to create the one and only instance. The constructors are all private; thus, you never “new” it.

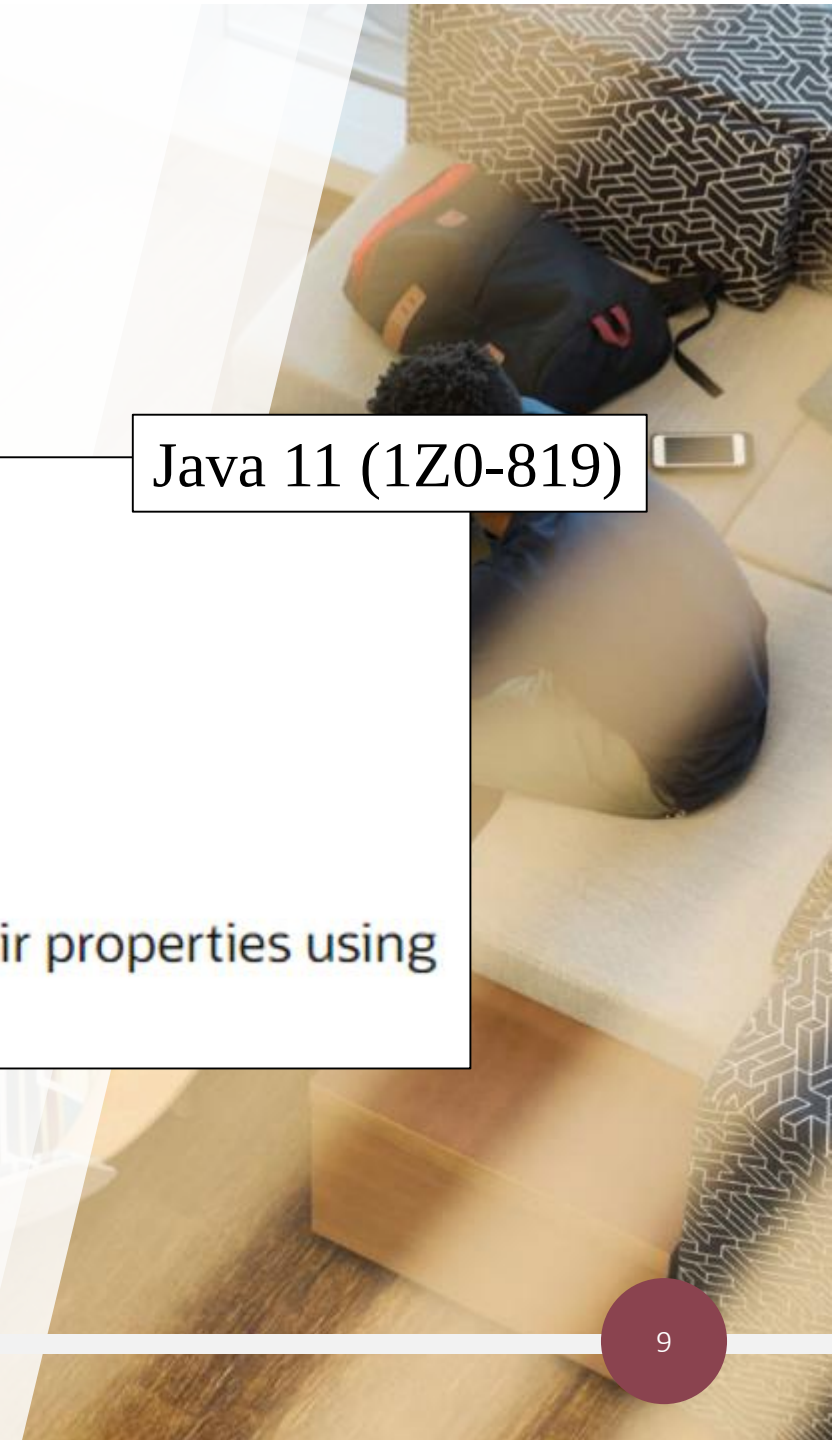
- *ConsoleTest.java*

IO and NIO.2

Java 11 (1Z0-819)

Use Java I/O API

- ✓ Read and write console and file data using I/O Streams
- ✓ Serialize and de-serialize Java objects
- → Construct, traverse, create, read, and write Path objects and their properties using java.nio.file API



NIO.2

- NIO.2 is a more powerful API than the legacy *java.io.File* class.

- The *Path* interface is central to NIO.2 (*Path* objects are immutable) :
  - *Path* - this interface replaces *File* as the representation of a file or directory; it is a lot more powerful than *File*
  - *Paths* - contains static methods to create *Path* objects (these objects implement the *Path* interface)
  - *Files* - contains static methods for working with *Path* objects i.e. methods that take *Path* objects as parameters

- PathGeneral.java
- PathOperations.java

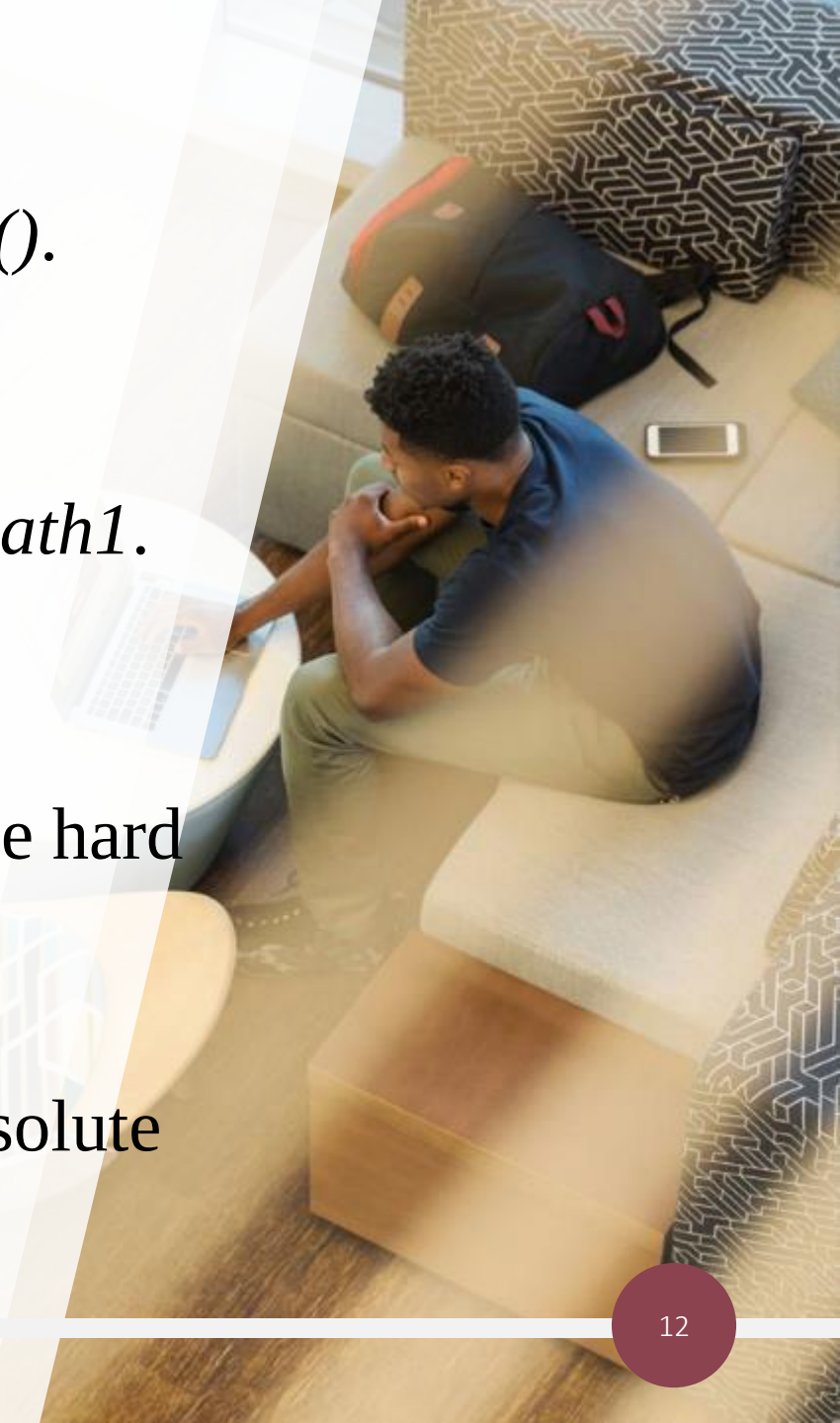
Resolving a Path

- To concatenate two Path objects together, use *resolve()*.
  - Resolve Concatenates i.e. Resolve to get Certified

- *path1.resolve(path2)* means concatenate *path2* onto *path1*.
  - ➢ strictly speaking, we are resolving *path2* within *path1*

- *resolve()* does not check whether the paths exist on the hard disk.

- If an absolute path is passed in, then that path (the absolute path) is simply returned.



- ResolveExamples.java

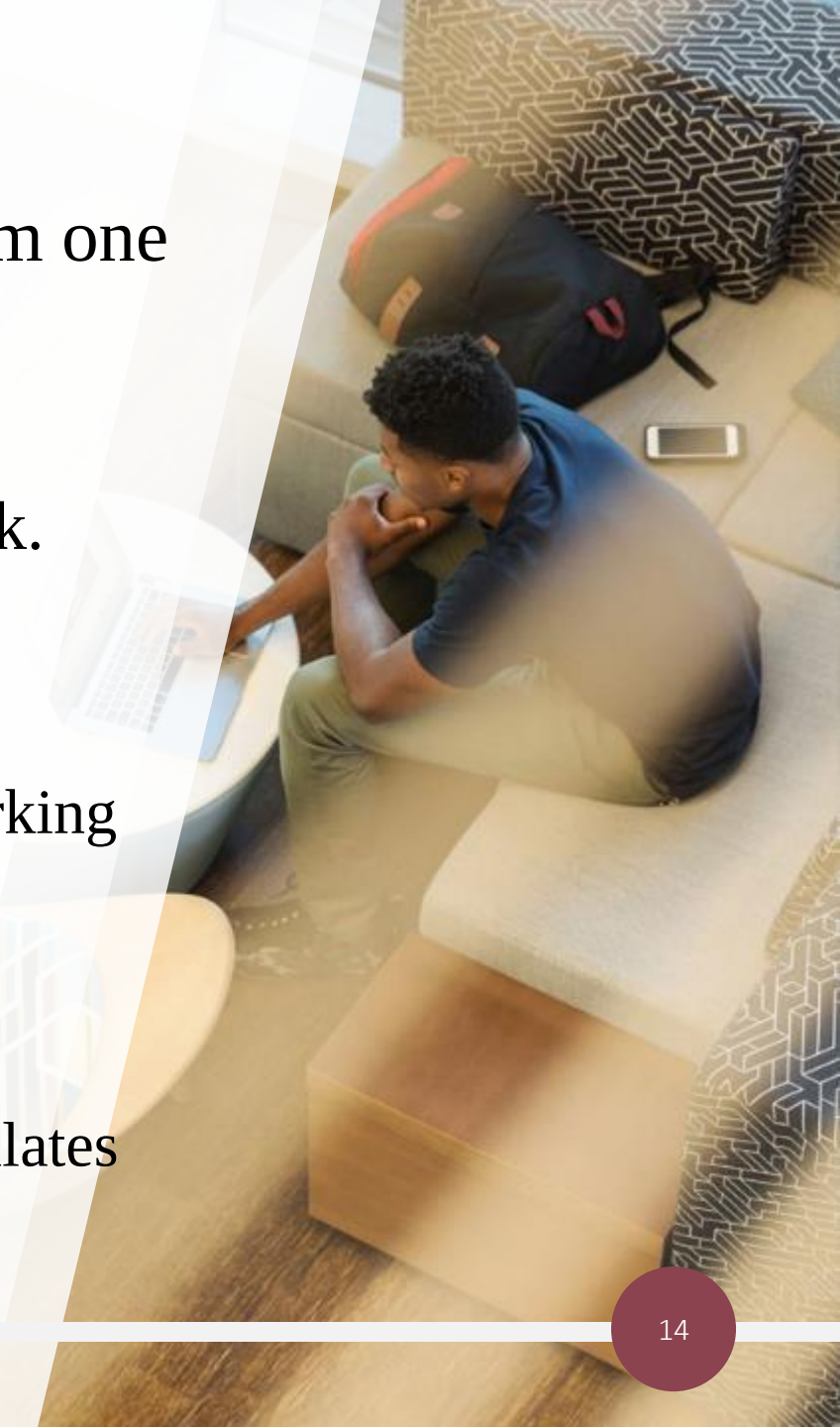
Relativizing a Path

- The *relativize()* method constructs a **relative** path from one *Path* to another.

- As with *resolve()*, the paths need not exist on the hard disk.

- If both *Path*'s are relative:
  - *relativize()* assumes both *Path*s are in the same current working directory.

- If both *Path*'s are absolute:
  - *relativize()* ignores the current working directory and calculates the relative path from one to the other



Relativizing a Path

- If the *Path*'s are mixed i.e. one is relative and the other absolute:
  - *relativize()* throws an exception.

- **Note that a file itself counts as one level** – this will be clearer in the examples.

- RelativizeExamples.java

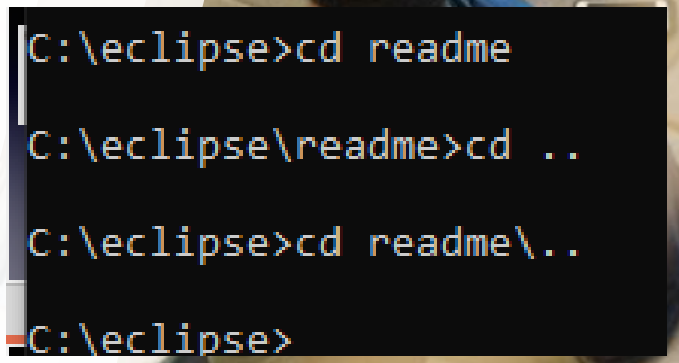
Normalizing a Path

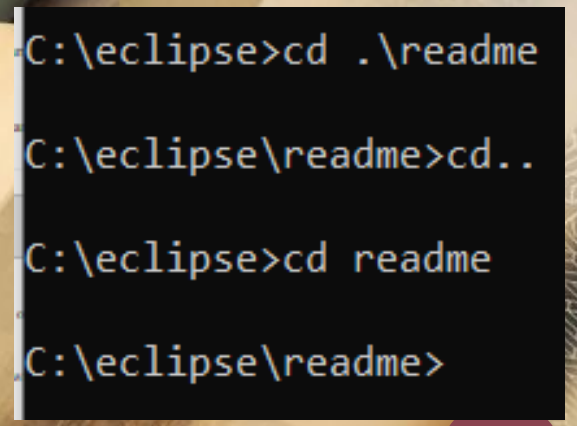
- The *normalize()* method removes unnecessary redundancies in a *Path*, so that it resembles what you would “normally” type in.

- .. refers to the parent directory and thus any directory followed by .. can be removed from the path.

- . refers to the current directory and thus any . followed by / (or \\) can be ignored.

- *normalize()* is useful when comparing different *Path*’s.





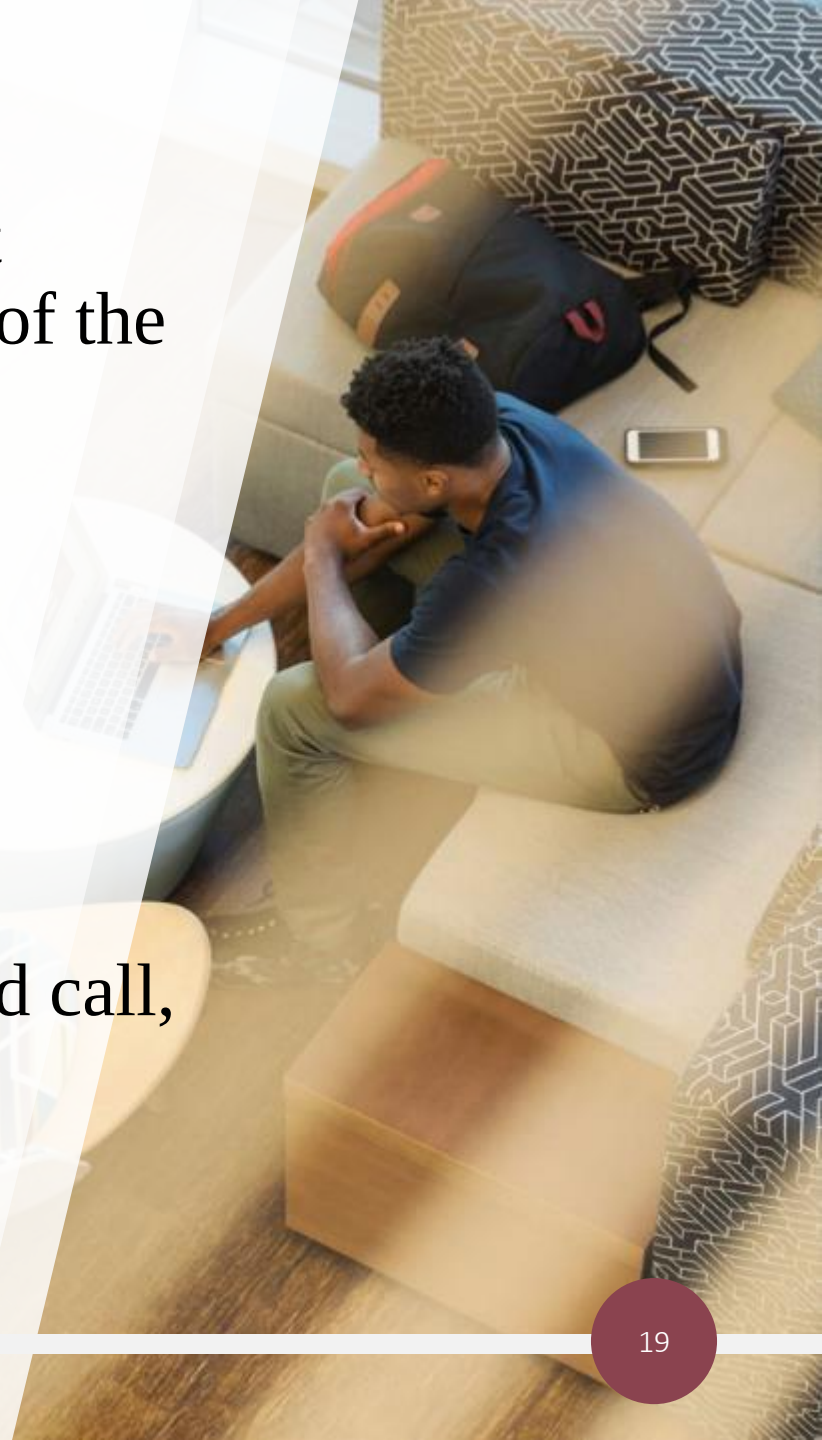
- NormalizeExamples.java

File Attributes

- File attributes (metadata) is data about a file that is not stored as part of the file contents. For example, the size of the file is an attribute.

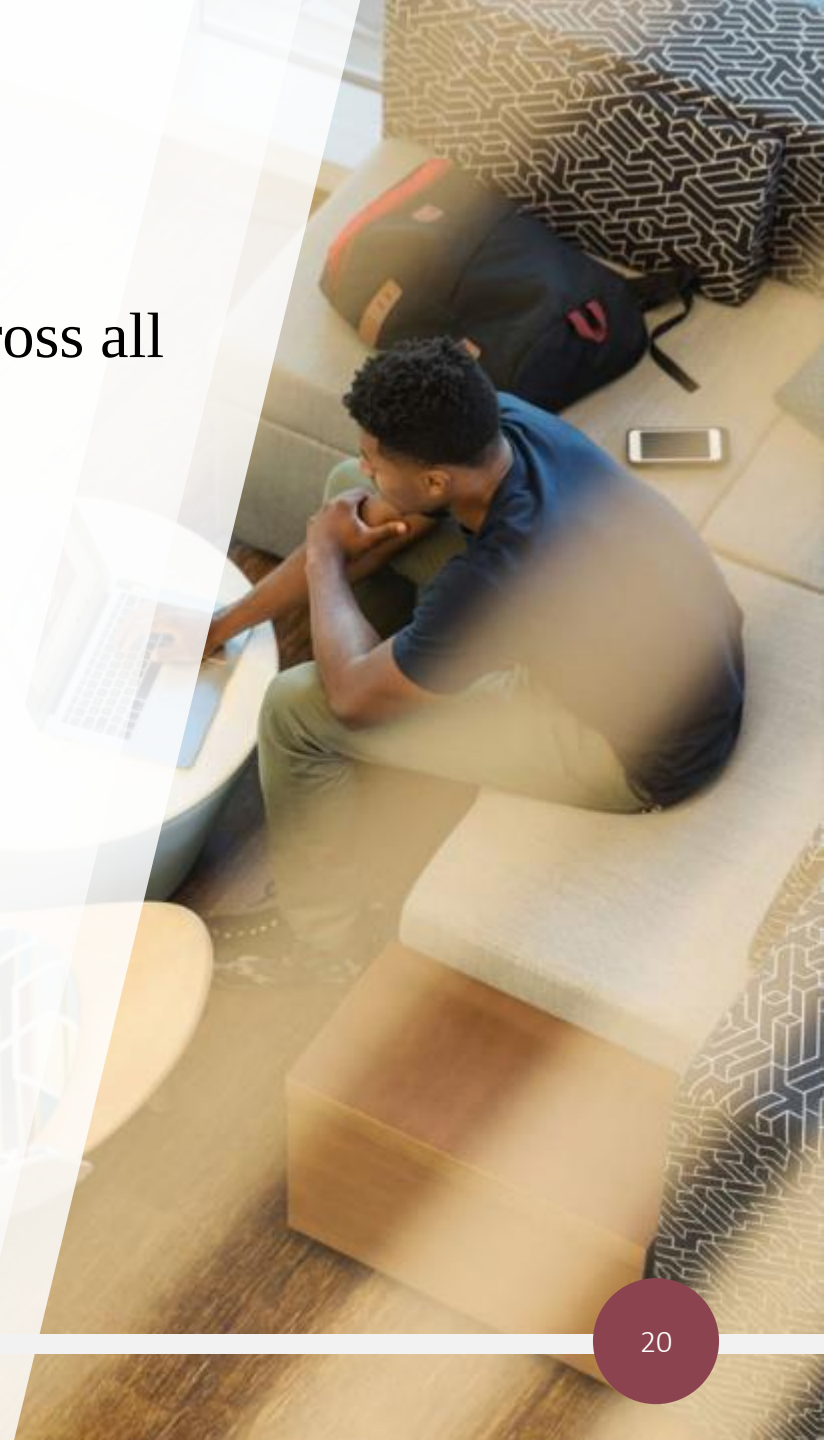
- The *Files* utility class contains several methods for determining file attributes.

- If you need to read several attributes then rather than making several method calls, one can, in a single method call, obtain a *view* (a group of related attributes).



File Attributes

- There are 3 views:
  - *BasicFileAttributeView* - basic set of attributes common across all file systems.
  - *DosFileAttributeView* - DOS/Windows systems.
  - *PosixFileAttributeView* – Unix/Linux/Mac systems.



- Attributes.java
- AttributesView.java

Traversing a Directory tree

- Given a directory, how do we list all the .java files in that directory tree i.e. include the sub-directories?

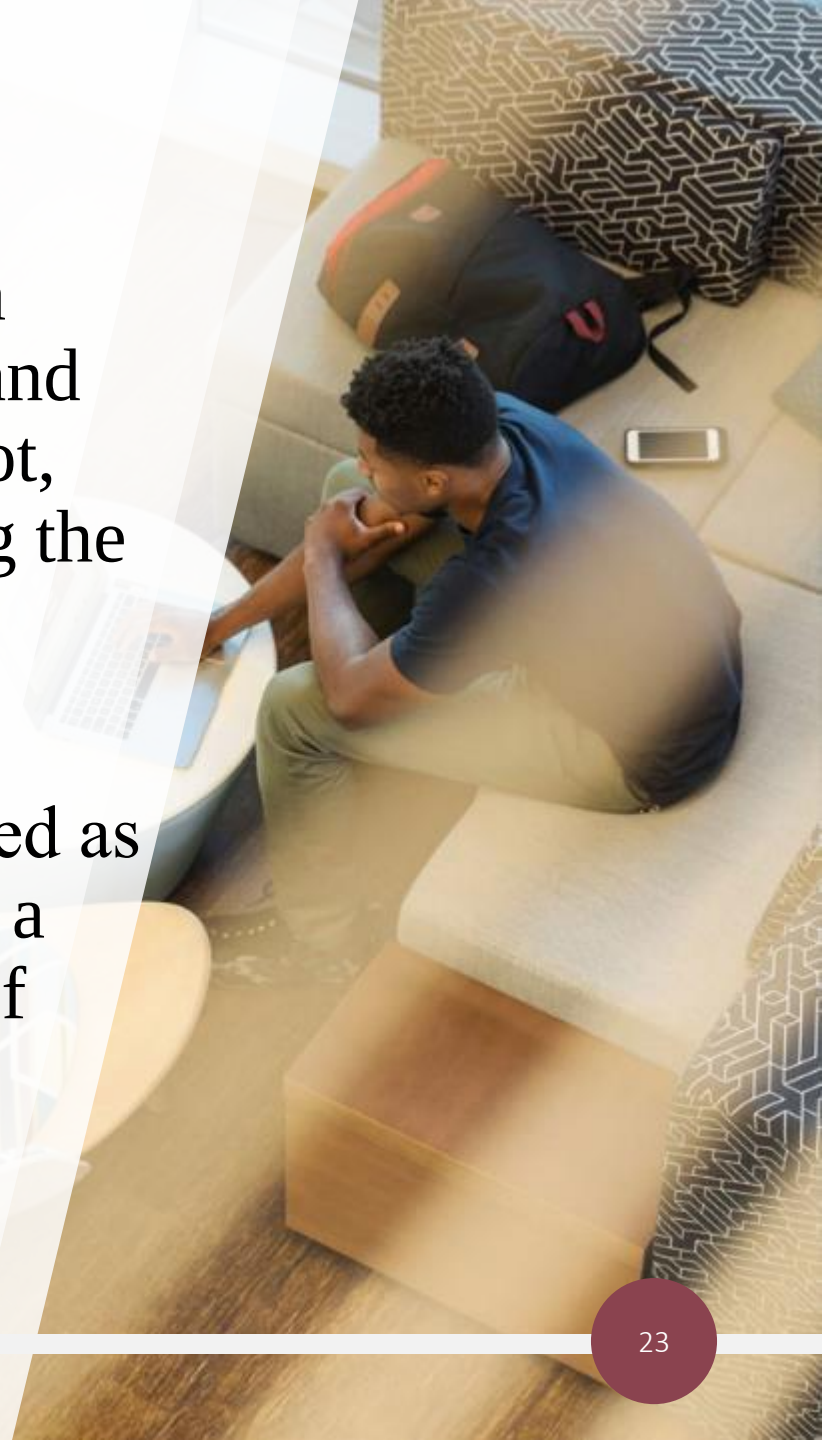
- Prior to the Stream API, this was accomplished with *DirectoryStream* and *FileVisitor*. Note: *DirectoryStream* is not a Stream class!

- The Stream API method is far superior and requires much less code.

Traversing a Directory tree

- The NIO.2 Streams API uses a *depth-first* search with a depth limit. In other words, searching begins at the root and traverses to the leaf and then goes back up toward the root, traversing fully down any path it may have skipped along the way.

- As streams use lazy evaluation, *Path*'s are only evaluated as they are encountered i.e. the memory required to process a directory tree is low even if the tree includes thousands of records.

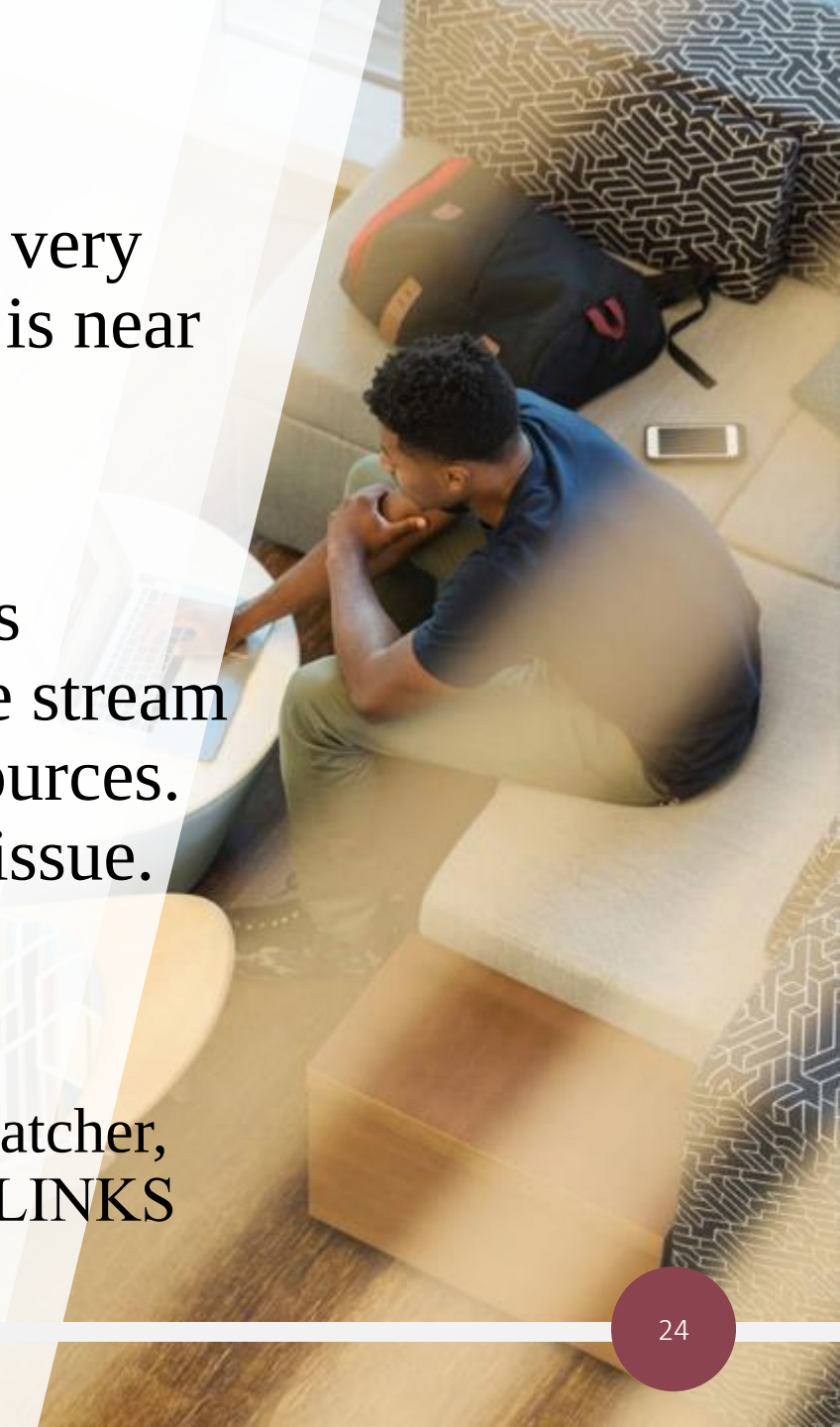


Traversing a Directory tree

- The *depth limit* is useful where the file system may he very large but we know the information we are searching for is near the root.

- Note that the NIO.2 stream based methods can result is resource leaks if not properly closed. This is because the stream terminal operations do **not** close the underlying file resources. Using *try-with-resources* is the cleanest solution to this issue.

- Stream<Path> find(Path start, int maxDepth,
  BiPredicate<Path, BasicFileAttributes> matcher,
  FileVisitOption… options) // FOLLOW_LINKS

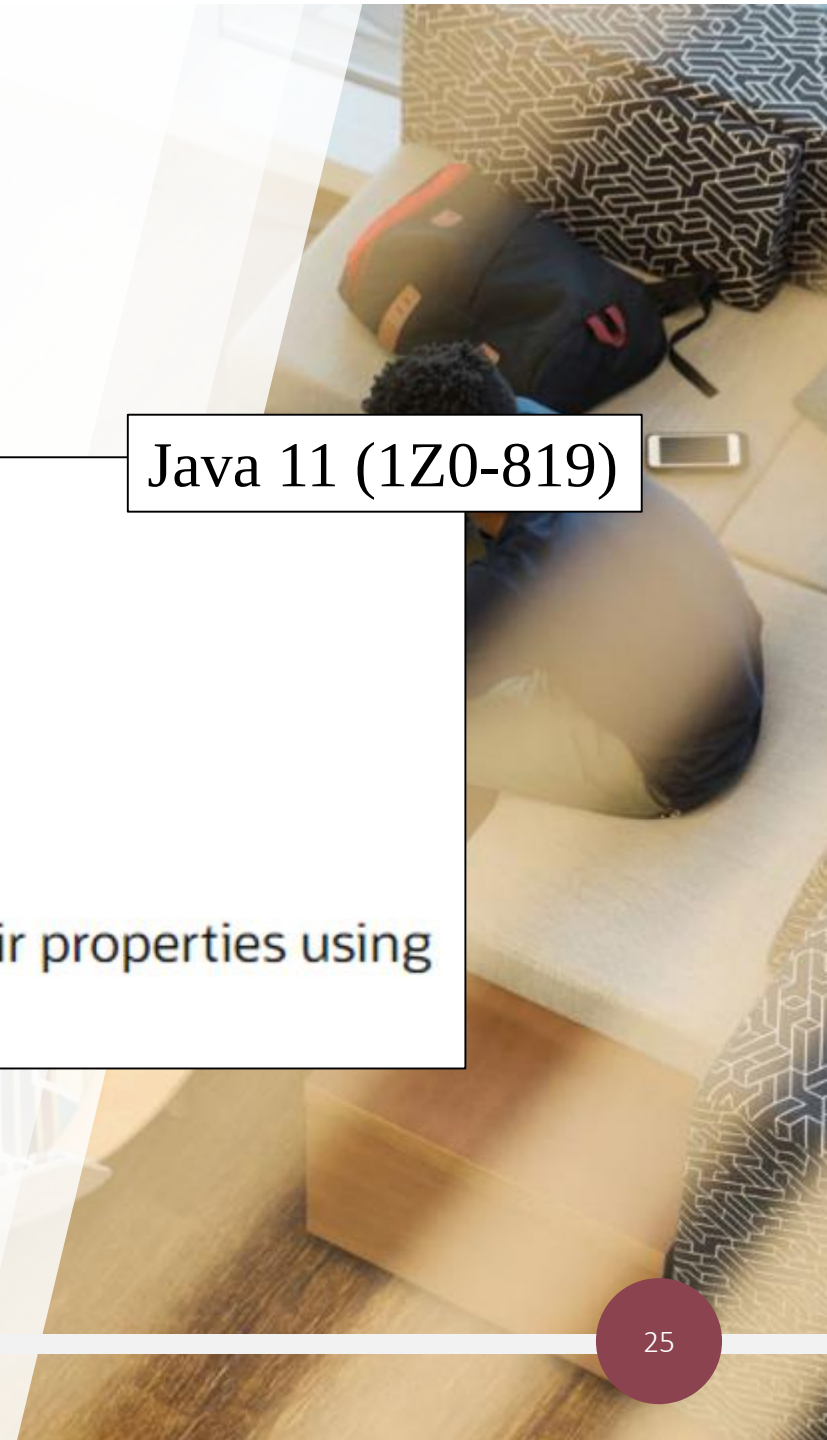


IO and NIO.2

Java 11 (1Z0-819)

Use Java I/O API

- ✓ Read and write console and file data using I/O Streams
- → Serialize and de-serialize Java objects
- ✓ Construct, traverse, create, read, and write Path objects and their properties using java.nio.file API



Serialisation

- Serialisation is the process of converting an in-memory Java object and saving it to a file (byte stream).

- De-serialisation is the opposite: it is the process of converting from a file to a Java object.

- To inform Java that your class is serialisable, you implement the marker interface *Serializable*. A “marker interface” is an interface that has no methods.

- Apply *Serializable* to data oriented classes as their state is not short-lived.

Serialisation

- As we are serialising objects, instance variables (and not *static* variables) are serialised.

- *private static final long serialVersionUID = 1L;*
  - although *static*, it is serialised as part of serialisation.
  - used to synchronise the class definition with the stored data
    e.g. what if a new field was inserted into the class definition and you were attempting to deserialise a file that was serialised with the old class definition i.e. without that new field? This results in an *InvalidClassException* during deserialisation.

Serialisation

- *private transient String password;*
  - ➢to prevent a field from being serialised, mark it as *transient*.
  - ➢when being deserialised, a *transient* field will revert to it's default Java value e.g. *null* for *String*, 0 for *int* etc…

- In summary, other than *serialVersionUID*, only non-*transient* instance members are serialised.

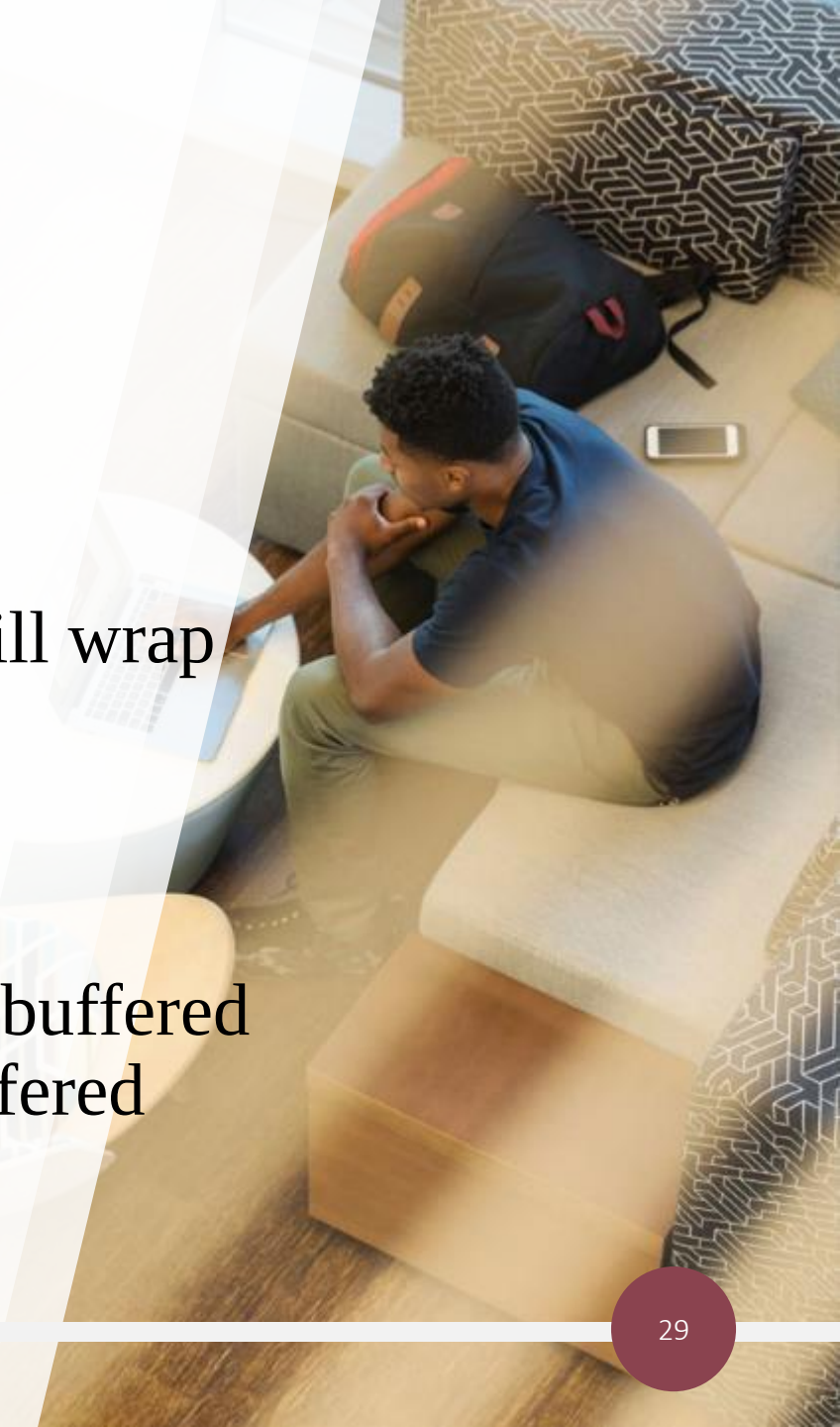
- Regarding your instance variables, if they are types, ensure that they are *Serializable* also. For example, if *Pen* is *Serializable* and has an instance variable of type *Nib* then if you are serialising *Pen*, *Nib* must also be *Serializable*.

Serialisation

- Serialisation requires the use of two classes:
  - *ObjectOutputStream*
  - *ObjectInputStream*

- These classes are considered high-level classes and will wrap lower level classes such as *FileOutputStream* and *FileInputStream*.

- Typically, we will start with a file stream, wrap it in a buffered stream to improve performance and then wrap the buffered stream with an object stream to access the serialisation/deserialisation methods.



- Serialisation.java