*7. Java I/O fundamentals*

*Working with class java.io.File*

* File is an abstract representation of a path to a file or a directory.
* You can use an object of class File to create a new file or directory, delete it, or inquire about or modify its attributes.
* A File instance might not be necessarily associated with an actual file or directory.
* File’s method isDirectory() returns true if the path it refers to is a directory.
* File’s method isFile() returns true if the path it refers to is a file.
* For a directory, File’s method list() returns an array of subdirectories and files.
* You can create a File instance that represents a non-existing file on your file system. And you can even invoke methods like isFile() without getting an exception.
* The objects of class File are immutable; the pathname represented by a File object can’t be changed.
* Methods createNewFile(), mkdir(), and mkdirs() can be used to create new files or directories.

*Using byte stream I/O*

* Class java.io.InputStream is an abstract base class for all the input streams.
* Class InputStream defines multiple overloaded versions of method read(), which can be used to read a single byte of data as int, or multiple bytes into a byte array.
* Method read() returns the next byte of data, or -1 if the end of the stream is reached. It doesn’t throw an EOFException.
* Method close() is another important method of class InputStream. Calling close() on a stream releases the system resources associated with it.
* Class java.io.OutputStream is an abstract class. It’s the base class for all the output streams in Java.
* The most important method of OutputStream class is write(), which can be used to write a single byte of data or multiple bytes from a byte array to a data destination.
* Class OutputStream also defines methods write(), flush(), and close(). So these are valid methods that can be called on any objects of classes that extend class OutputStream.
* All the classes that include OutputStream in their name—FileOutputStream, ObjectOutputStream, BufferedOutputStream, and DataOutputStream—extend abstract class OutputStream, directly or indirectly.
* To read and write raw bytes from and to a file, use FileInputStream and FileOutputStream.
* FileInputStream is instantiated by passing it a File instance or string value. It can’t be instantiated by passing it another InputStream.
* Instantiation of FileOutputStream creates a stream to write to a file specified either as a File instance or a string value. You can also pass a boolean value specifying whether to append to the existing file contents.
* Copying a file’s content might not copy its attributes. To copy a file, it’s advisable to use methods such as copy() from class java.nio.file.Files.
* I/O operations that require reading and writing of a single byte from and to a file are a costly affair. To optimize the operation, you can use a byte array.
* Unlike read(), read(byte[]) doesn’t return the read bytes. It returns the **count of bytes** read, or -1 if no more data can be read. The actual data is **read in the byte array** that’s passed to it as a method parameter.
* Method write(int) in class OutputStream writes a byte to the underlying output stream. If you write an int value by using this method, only the 8 low-order bits are written to the output stream; the rest are ignored.
* To buffer data with byte streams, you need classes BufferedInputStream and BufferedOutputStream.
* You can instantiate a BufferedInputStream by passing it an InputStream instance.
* A BufferedOutputStream can be instantiated by passing it an OutputStream instance.
* You can specify a buffer size or use the default size for both BufferedInputStream and BufferedOutputStream.
* To instantiate BufferedInputStream, you must pass it an object of InputStream. To instantiate BufferedOutputStream, you must pass it an object of OutputStream.
* You can use FileInputStream and FileOutputStream to read and write only byte data from and to an underlying file. These classes (FileInputStream and FileOutputStream) don’t define methods to work with any other specific primitive data types or objects.
* Data input and output streams let you read and write primitive values and strings from and to an underlying I/O stream in a machine-independent way. Data written with DataOutputStream can be read by DataInputStream.
* If a mismatch occurs in the type of data written by DataOutputStream and the type of data read by DataInputStream, you might not get a runtime exception. Because data streams read and write bytes, the read operation constructs the requested data from the available bytes, though incorrectly.
* An ObjectOutputStream can write primitive values and objects to an OutputStream, which can be read by an ObjectInputStream.
* To write objects to a file, their classes should implement java.io.Serializable, or the code will throw a java.io.NotSerializableException.
* If a class implements the Serializable interface, but its base class doesn’t, the class’s instance can be serialized.
* A class whose object fields don’t implement the Serializable interface can’t be serialized even though the class itself implements the Serializable interface. An attempt to serialize such object fields will throw a runtime exception.
* Retrieve the data (primitive and objects) in the order it was written using object streams, or it might throw a runtime exception.
* When you write objects to a file using ObjectOutputStream, its transient or static variables aren’t written to the file.

*Using character I/O with readers and writers*

* Reader and Writer are abstract base classes for reading and writing Unicode compliant character data.
* Classes Reader and Writer handle 16-bit Unicode well, which isn’t supported by the byte-oriented InputStream and OutputStream classes.
* Abstract class Reader defines overloaded read() methods to read character data from an underlying data stream.
* Class Reader implements Closeable (and its parent interface AutoCloseable). So Reader objects can be declared as resources with a try-with-resources statement.
* Compare the overloaded read() methods of class InputStream with the read() methods of class Reader. The read() methods of InputStream accept an array of byte as their method parameter, and the read() methods of Reader accept an array of char as their method parameter.
* Abstract class Writer defines overloaded write() methods to write character data to an underlying data source.
* With the overloaded write() methods of class Writer, you can write a single character or multiple characters stored in char arrays or string to a data source.
* FileReader and FileWriter are convenience classes for reading and writing character data from files.
* You can instantiate a FileReader by passing it the name of a file as a string value or as a File instance.
* You can instantiate a FileWriter by passing it the name of a file as a string value or as a File instance. You also have the option of specifying whether you want to override the existing content of a file or append new content to it by passing a boolean value to the constructor.
* To buffer data with character streams, you need classes BufferedReader and BufferedWriter.
* You can instantiate a BufferedReader by passing it a Reader instance.
* You can instantiate a BufferedWriter by passing it a Writer instance.
* You can also specify a buffer size or use the default size for both Buffered- Reader and BufferedWriter.
* Class PrintWriter can be used to print (write) formatted representations of objects to a file. This class implements all the print() methods found in class PrintStream.
* This essentially means that you can use all the overloaded print() methods that you’ve been using (via the class variable System.out) to write data to a file, a PrintWriter instance.

*Working with the console*

* Class java.io.Console defines methods to access the character-based console device associated with the current JVM.
* You may or may not be able to access the console associated with a JVM, depending on the underlying platform and how the JVM was started.
* If you invoke a JVM from the command line without redirecting the standard input and output streams, you’ll be able to access its console, which will typically be connected to the keyboard and display from which the virtual machine was launched.
* You may not be able to access the console associated with a JVM if it started automatically as a result of the execution of some other program.
* You will not get access to the console when using IDEs like Eclipse.
* You can access an object of class Console by calling System.console().
* If no console device is available, System.console() returns null. A null value signals that either the program was launched in a non-interactive environment or perhaps the underlying operating system doesn’t support the console operations.
* You can’t create an object of Console yourself. Class Console doesn’t define a public constructor.

*8. Java file I/O (NIO.2)*

***Path objects***

* Objects of the Path interface are used to represent the path of files or directories in a file system.
* Because a Path object might not be tied to a real file or directory on a system, it can refer to a nonexistent file or directory.
* Apart from referring to a file or a directory, a Path object can also refer to a symbolic link. A symbolic link is a special file that refers to another file.
* When you read data from or write data to a symbolic link, you read from or write to its underlying target file. But if you delete a symbolic link, the target file isn’t deleted.
* A Path can never be equal to a Path associated with another file system, even if they include exactly the same values.
* You can create Path objects by using Paths.get() or FileSystems.getDefault().getPath().
* You can convert a File instance to a Path object by calling toPath() on the File instance.
* Behind the scenes, both Paths.get() and File.toPath() call FileSystems.getDefault().getPath().
* Most of the Path methods perform syntactic operations. They manipulate the paths to a file or directory without accessing the file systems. They’re logical operations on paths in memory.
* Methods getName(), getNameCount(), and subpath() don’t use the root directory of a path. Method getRoot() returns the root of an absolute path and null for relative paths.
* The Path methods that accept positions throw an IllegalArgumentException at runtime for invalid positions. For example, getName() and subpath() throw an IllegalArgumentException if you pass invalid path positions to them.
* You can compare paths lexicographically using method compareTo(Path).
* To check whether a path starts or ends with another path, you can use startsWith(String), startsWith(Path), endsWith(String), and endsWith(Path).
* Methods startsWith() and endsWith() are overloaded—startsWith(String), startsWith(Path), endsWith(String), and endsWith(Path). So if you pass null to these methods, you’ll get a compiler error.
* The method name to retrieve the absolute path from a Path object is toAbsolutePath() and not getAbsolutePath().
* You can remove redundant path values by calling method normalize() on Path.
* Path is immutable and calling normalize() on a Path object doesn’t change its value.
* Method normalize() doesn’t check the actual file system to verify if the file (or directory) the resulting path is referring to actually exists.
* If a Path object includes redundancies like . or .., calling information retrieval methods like subpath() or getName() will include these redundancies in the returned values.
* The overloaded methods resolve(String) and resolve(Path) are used to join a relative path to another path. If you pass an absolute path as a parameter, this method returns the absolute path.
* To retrieve the path to a file in the same directory, say, to create its copy or to rename it, you can use the overloaded methods resolveSibling(String) and resolveSibling(Path).
* Method resolveSibling() resolves a given path against a path’s parent. If the given path is an absolute path, this method returns the absolute path. If you pass it an empty path, it returns the parent of the path.
* Methods resolve() and resolveSibling() don’t check the actual file system to verify if the file (or directory) the resulting path is referring to actually exists.
* To construct a path between two Path objects, use method relativize(). It can be used to construct a path between two relative or absolute Path objects.
* You can’t create a path from a relative path to an absolute path and vice versa using method relativize(). If you do so, you’ll get a runtime exception (IllegalArgumentException).
* Method relativize() doesn’t check the actual file system to verify if the file (or directory) the resulting path is referring to actually exists.

***Class Files***

* Class java.nio.file.Files defines static methods for manipulating files and directories.
* Method createFile() atomically checks for the existence of the file specified by the method parameter path and creates it if it doesn’t exist.
* Method createFile() fails and throws an exception if the file already exists, a directory with the same name exists, its parent directory doesn’t exist due to an I/O error, or the specified file attributes can’t be set.
* Method createDirectory() creates the specified directory (not the parent directory) on the file system. It also atomically checks for the existence of the specified directory and creates it if it doesn’t exist.
* Method createDirectory() throws an exception if a file or directory exists with the same name, its parent directory doesn’t exist due to an I/O error, or the specified directory attributes can’t be set.
* Method createDirectories() creates a directory, creating all nonexistent parent directories.
* If the target directory already exists, createDirectories() doesn’t throw any runtime exception. It throws an exception if the specified dir exists but isn’t a directory, an I/O occurs, or the specified directory attributes can’t be set.
* Specifying file or directory attributes is optional with methods createFile(), createDirectory(), and createDirectories(). All these methods declare to throw an IOException, which is a checked exception.
* You can check for the existence of a file or directory referred by a Path object using methods exists() and notExists().
* Method notExists() isn’t a complement of method exists(). It returns true if a target doesn’t exist or false if its existence can’t be determined. If these methods can’t determine the existence of a file, both of them will return false.
* Class Files’s overloaded method copy() enables you to read from InputStream and write to a Path object, read from a Path object and write to OutputStream, and read from and write to Path objects.
* Files.copy() can copy only files, not directories. If the source is a directory, then in the target an empty directory is created (without copying the entries in the directory). This method returns a long or Path value, not a boolean value.
* If you use a relative path to the target file, the file is created relative to your Java class file (.class) and not relative to the source file (passed as a parameter to method Files.copy()).
* To move files or directories programmatically, you can use Files.move(), which moves or renames a file to a target file.
* You can only move empty directories using method Files.move(). You can rename a nonempty directory by using Files.move(). But you can’t move a file or directory to a non-existing directory.
* To delete a directory or a file referred to by a Path object, you can use methods delete(Path) or deleteIfExists(Path).
* If you try to delete a directory that isn’t empty, methods delete(Path) and deleteIfExists(Path) will throw a DirectoryNotEmptyException.
* If you try to delete a nonexistent file or directory using method delete(), it will throw a NoSuchFileException. But method deleteIfExists() won’t throw an exception if the file or directory at the specified path doesn’t exist—rather, it will return false.
* Methods delete() and deleteIfExists() can be used to delete files and (nonempty) directories.

***Files and directory attributes***

* Class Files defines static methods to access individual attributes of a file or directory referred by a Path.
* You can access the individual attributes of a file or directory by using method Files.getAttribute(), passing to it the name of the attribute as a string value. To modify the attributes of an existing file or directory, you can use Files.setAttribute().
* You can access a group of file attributes by calling Files.getFileAttributeView() or Files.readAttributes().
* The BasicFileAttributes interface defines methods to access the basic attributes that should be supported by all the file systems.
* The BasicFileAttributeView interface can be used to modify the basic attributes.
* The DosFileAttributes interface extends BasicFileAttributes and defines methods to access attributes specific to Windows files and directories.
* The DosFileAttributeView interface defines methods to modify the DOS file attributes.
* The PosixFileAttributes interface also extends BasicFileAttributes and defines methods to access attributes related to the POSIX family of standards, like Linux or UNIX.
* The PosixFileAttributeView interface defines methods to modify attributes related to the POSIX family.
* Available only for Windows OS, the AclFileAttributeView interface supports access and updates of a file’s ACL.
* The FileOwnerAttributeView interface supports access and updates to the owner of a file or directory. It is supported by all systems that support the concept of file owners.
* The UserDefinedFileAttributeView interface supports the addition, modification, and deletion of user-defined metadata.
* The BasicFileAttributes, DosFileAttributes, and PosixFileAttributes interfaces define methods to access attributes. They don’t define methods to modify (or set) the attributes.
* The BasicFileAttributeView, DosFileAttributeView, PosixFileAttributeView, AclFileAttributeView, FileOwnerAttributeView, and UserDefinedFileAttributeView interfaces can be used to update attribute values.
* If a file system doesn’t support an attribute view, Files.getFileAttributeView() returns null. If a file system doesn’t support an attribute set, File.readAttributes() will throw a runtime exception.
* If an underlying system doesn’t support all the basic timestamps—that is, creationTime, lastAccessTime, and lastModifiedTime it might return system specific information.
* Methods Files.setAttribute() and Files.getAttribute() throw an IllegalArgumentException or UnsupportedOperationException if you pass them an invalid or unsupported attribute.
* The DosFileAttributes interface makes the following attributes available:
* archive
* hidden
* readonly
* system
* The DOS attributes are available on a Windows system only. Trying to access them on other systems will throw a runtime exception.
* When you read all DOS attributes using method Files.readAttributes(), you also read the basic attributes.
* The POSIX attributes are as follows:
* group
* owner
* permissions
* The POSIX attributes are available on the POSIX family of standards, such as UNIX and LINUX. Trying to access them on other systems will throw a runtime exception.
* To read or update the owner of a file or directory you can use the AclFileAttributeView, FileOwnerAttributeView, and PosixFileAttributeView interfaces.
* The UserDefinedAttributeView interface can be used to add, delete, access, and modify additional user-defined attributes to or from a file or directory. It defines methods delete(String), list(), read(String, ByteBuffer), size(String), and write(String, ByteBuffer) to, respectively, delete, list, read, get an attribute’s size, and write attribute values.

***Recursively access a directory tree***

* Class Files defines overloaded method walkFileTree() to walk recursively through the specified path. To define the traversal behavior, this method accepts an object of the FileVisitor interface.
* You can use the FileVisitor, a generic interface, to define the code that you want to execute during the traversal of a directory structure. When you traverse a directory structure, you can define what to do before or after you visit a directory, when you visit a file, or when access to a file is denied.
* Method postVisitDirectory() is invoked for a directory after entries in the directory and all of their descendants have been visited.
* Method preVisitDirectory() is invoked for a directory before entries in the directory are visited.
* Method visitFile() is invoked for a file in a directory.
* Method visitFileFailed() is invoked for a file that couldn’t be visited.
* Methods preVisitDirectory() and visitFile() are passed BasicFileAttributes of the path that they operate on. You can use these methods to query file or directory attributes.
* Class SimpleFileVisitor is a simple visitor of files with default behavior to visit all files and to rethrow I/O errors. It implements the FileVisitor interface.
* You can initiate traversal of a directory by calling the overloaded method walkFileTree() from class Files.
* The DirectoryStream interface can be used to iterate over all the files and directories in a directory. You can use an Iterator or for-each construct to iterate over a directory. The order in which the directory contents are iterated is unpredictable.
* If you pass Path to a file (and not a directory) to Files.newDirectoryStream(), it will throw a runtime exception. The order of iteration of files and directories in a specified directory using DirectoryStream is unpredictable.

***Using PathMatcher***

* You can match your file or directory names against a regex or glob pattern by using PathMatcher.
* A glob pattern supports a simpler form of pattern matching than the regex. It supports fewer special constructs.
* In glob, \* matches zero or more characters. In regex, .\* matches zero or more characters.
* To match a Path object with a pattern, you should create an object of java .nio.file.PathMatcher. PathMatcher is an interface with just one method: matches(). It returns true if a given path matches this matcher’s pattern.
* You can create a PathMatcher by calling FileSystem.getPathMatcher() and passing it the pattern to be matched.

***Watch a directory for changes***

* WatchService enables you to watch a directory for changes like addition, modification, or deletion of contents of a directory.
* The first step to watch a directory for changes is to create a WatchService object.
* A WatchService object watches a directory for the following events:
* StandardWatchEventKinds.ENTRY\_CREATE—This event occurs when a new file or directory is created, moved, or renamed in the directory being watched.
* StandardWatchEventKinds.ENTRY\_DELETE—This event occurs when an existing file or directory is deleted, moved, or renamed in the directory being watched.
* StandardWatchEventKinds.ENTRY\_MODIFY—This event is platform-dependent. It usually occurs when contents of an existing file are modified. It can also occur if the attributes of a file or directory (in the directory being watched) are modified.
* StandardWatchEventKinds.OVERFLOW—This indicates that an event has been lost.
* You can register multiple directories to be watched with the same WatchService object by using method register() of Path.
* You can watch a directory for changes. If you try to register a file for changes, you’ll get a runtime exception (NotDirectoryException). Registering a directory for any event (create, modify, or delete) doesn’t implicitly register its subdirectories.
* The WatchKey object is a token that represents the registration of a Watchable object with a WatchService. A WatchKey object is created when you register your directory to be watched for create, modify, or delete events with a WatchService.
* A WatchKey can be in multiple states:
* *Ready*—A WatchKey is initially created with a ready state.
* *Signaled*—When an event is detected, the WatchKey is signaled and queued. It can be retrieved using method WatchService’s poll() or take().
* *Cancelled*—Calling method cancel() on a WatchKey or closing the WatchService cancels a WatchKey.
* *Valid*—A WatchKey in a ready or signaled state is in a valid state.
* A WatchService queues the registered events when they occur. The registered consumers can retrieve the queued WatchKeys and process the corresponding events.
* The WatchService interface defines method take() and overloaded method poll() to retrieve the queued WatchKeys. Once a key is processed, the consumer invokes the key’s method reset() so that it can be signaled and required for further events.
* Method take() of the WatchService interface retrieves and removes the next WatchKey, waiting if none is yet present.
* Method poll() of the WatchService interface retrieves and removes the next WatchKey, or null if none is present (no waiting). You can also use its overloaded method poll(long timeout, TimeUnit unit) to specify the waiting time if none is present.
* For each retrieved WatchKey, you can call the WatchKey’s method pollEvents() to retrieve and remove all pending events for the key.
* Method pollEvents() returns a list of the events (WatchEvent) that were retrieved. Because you can register multiple paths and events with the same WatchService object, you can query the WatchEvent to determine the source of the event and its type, and process the event as required.

*9. Building database applications with JDBC*

***Introduction***

* JDBC is part of the core Java API; you don’t need to download it separately to use it in your Java applications.
* By using the standard classes and interfaces from the JDBC API, Java classes can connect to a database, execute SQL, and process the results.
* JDBC is a standard specification any Java class can use JDBC to connect with a database using a JDBC driver.
* Contrary to the JDBC API, JDBC drivers are external class bundles not included in the JDK.
* JDBC has been around since 1997 and was added to JSE 1.1 as JDBC 1.0. Since its first version, multiple enhancements have been made to JDBC. Java 7 ships with JDBC 4.1.
* JDBC classes are defined in Java packages java.sql and javax.sql.
* You can access tabular data stored on relational databases, flat files, and Excel spreadsheets using the JDBC API.
* JDBC provides vendor-neutral access to the common database features. So if you don’t use proprietary features of a database, you can easily change the database that you connect to. The JDBC API includes an abstraction of a database connection, statements, and result sets.
* With the JDBC API, you have a choice of connecting to a local or remote data store either directly or through an application server.
* JDBC drivers are the implementation of the lower-level JDBC driver API as defined in the JDBC specifications.
* Depending on whether the drivers are implemented using only Java code, native code, or partial Java, they’re categorized as type 4, pure Java JDBC driver; type 3, pure Java driver for database middleware; type 2, native API, partial Java driver; and type 1, JDBC–ODBC bridge.

***Interfaces that make up the JDBC API core***

* The interfaces that make up the core of the JDBC API are java.sql.Driver, java.sql.Connection, java.sql.Statememt, java.sql.ResultSet, and javax.sql.RowSet.
* Every JDBC driver implementation *must* implement the interface Driver.
* Every driver must implement a minimum set of interfaces defined in the JDBC API to conform to the JDBC specifications.
* Connection represents a connection session with the specified database. It’s used to create SQL statements, execute them against the database, start and commit transactions, and retrieve other details.
* The interface Statement is used to create and execute static SQL statements and retrieve their results.
* Interfaces PreparedStatement and CallableStatement extend the Statement interface. They represent precompiled statements.
* PreparedStatement can be used to execute static or dynamic SQL statements.
* CallableStatement is used to execute stored database procedures.
* A ResultSet is retrieved as a result of executing a SQL SELECT statement against a database. It represents a table of data.

***Connecting to a database***

* The first step to make a Java class communicate with a database is to establish a connection between them.
* Class java.sql.DriverManager talks with the JDBC driver to return a Connection object, which can be used by your Java classes for all further communication with the database.
* For the exam, it’s important to note the difference between a JDBC driver (lowercase d) and a Driver (uppercase D). A JDBC driver is a set of classes provided by the database vendor or a third party, usually in a .jar or .zip file, to support the JDBC API. A Driver class is an implementation of the interface java.sql.Driver.
* To connect with a data source, you need class DriverManager only once.
* Manual loading of drivers is required for JDBC API version 3.0 and before.
* JDBC drivers should be manually loaded by calling Class.forName(), passing it the name of the Driver class.
* If Class.forName() can’t load the JDBC driver, it throws a ClassNotFoundException (a checked exception).
* For JDBC API 4.0 and later, JDBC drivers can be automatically loaded and registered by class DriverManager.
* When a class is loaded in memory, its static initializer block executes. According to the JDBC specifications, a driver must register itself with the DriverManager.
* The JVM loads class DriverManager when you call any of its methods.
* Class DriverManager manages all the instances of JDBC driver implementations registered with a system.
* When you invoke method getConnection(), class DriverManager finds the appropriate drivers from its set of registered drivers, establishes a connection with a database, and returns the Connection object.
* There are three overloaded versions of method getConnection().
* You can connect to a database by including the username and password as part of the JDBC connect URL string.
* It’s common to use Properties to specify the login credentials for the JDBC URL in method getConnection().
* The property names for specifying the username and password to establish a connection are “user” and “password”. An attempt to use any other key to specify and use username and password will throw a SQLException.
* If your application can’t load a JDBC driver or connect to a database due to invalid login credentials, it will throw an exception.

***CRUD (create, retrieve, update, and delete) operations***

* To create a Statement object, call method createStatement() on a Connection object.
* To define and execute a static SQL statement for a Statement object, call executeQuery() or executeUpdate() on Statement.
* Method executeQuery() returns ResultSet.
* Method executeUpdate() returns an int value specifying the number of affected rows.
* Method executeUpdate() is used to execute SQL queries to insert new rows in a table, and update and delete existing rows. It’s also used to execute DDL queries like the creation, modification, and deletion of database objects like tables. If you use method executeQuery() for any of these operations, you’ll get a SQLException at runtime.
* The SQL statements that you include in your code are defined as string values. This essentially means that if any of the SQL statement is invalid, no compilation errors are thrown.
* If a SQL SELECT returns no rows, ResultSet doesn’t refer to a null value. It refers to an initialized ResultSet object with zero rows.
* Connection, Statement, and ResultSet objects should be closed by calling their method close() either implicitly or explicitly.
* If you create Connection, Statement, and ResultSet objects using a try-with resources statement, it will auto-close them.

***JDBC transactions***

* A transaction is a logical set of SQL statements. Either all or none of the statements must execute from a transaction.
* To initiate a transaction, set the default database auto-commit mode to false.
* If the auto-commit mode of a connection is set to true, calling any of the transaction methods like commit() or rollback() will throw a SQLException.
* Method executeUpdate() returns a count of the rows that are or would be affected in the database for row insertions, modifications, and deletion. The value is returned even if the statement isn’t committed to a database.
* By using a savepoint, you can exercise finer control over the work done by a set of SQL statements in a transaction.

***RowSet objects***

* You can use RowSet objects as JavaBeans components, which can be created and configured at design time.
* You can configure a RowSet object by setting its properties, connecting to a JDBC data source, executing a SQL statement, and getting the results.
* The interface javax.sql.RowSet extends the interface java.sql.ResultSet.
* You can register listeners with a RowSet object so that when an event occurs on a RowSet object (like any modification to its value), the registered listeners can be notified.
* RowSet objects can be *connected* or *disconnected*. A connected RowSet object, like JdbcRowSet, maintains a connection with its data source throughout its life. On the other hand, a disconnected RowSet object, like CachedRowSet, establishes a connection with the data source, gets the values, and then disconnects itself.
* The interface javax.sql.rowset.RowSetFactory defines the implementation of a factory that can be used to obtain different types of RowSet implementations.
* Class javax.sql.rowset.RowSetProvider defines factory methods to get a RowSetFactory implementation. The RowSetFactory can then be used to create objects of different types of RowSet implementations. The Java API defines a default implementation of RowSetFactory.
* If the execute method isn’t successful on a RowSet object, you can only call execute and close methods on it. The rest of the methods will throw an exception.

***Precompiled statements***

* Interfaces java.sql.PreparedStatement and java.sql.CallableStatement extend the java.sql.Statement interface.
* The objects of interfaces PreparedStatement and CallableStatement represent precompiled SQL statements.
* Precompiled statements execute faster than their non-compiled counterparts.
* Another major advantage offered by PreparedStatement and CallableStatement is their ability to include placeholders in SQL statements using ?. You can assign values to these placeholders by calling one of the appropriate setDataType(parameterIndex, value) on these objects.
* Unlike Statement, you must specify the relevant SQL statement when you create an object of PreparedStatement.
* PreparedStatement defines three methods to execute its SQL statement: execute(), executeQuery(), and executeUpdate(). Method execute() can execute any type of SQL statement and returns a boolean value. Method executeQuery() executes a SQL SELECT statement and returns a ResultSet. Method executeUpdate() executes DDL statements and table INSERT, UPDATE, and DELETE SQL statements. It returns 0 for DDL statements and the number of rows affected for SQL INSERT, UPDATE, and DELETE statements.
* You can execute the database-stored procedures from your Java applications by using CallableStatement in the JDBC API.
* If a database-stored procedure doesn’t accept any parameter, it’s acceptable to drop the () following the procedure name in a call to execute it using CallableStatement.
* A database stored procedure can accept multiple parameters: an IN (input) parameter to send values to the procedure, and an OUT (output) parameter to return values from the procedure.
* You can define parameters of type IN, OUT, and INOUT with a database-stored procedure using CallableStatement.

*10. Threads*

***Create and use threads***

* All nontrivial Java applications are multithreaded.
* Multiple threads can be supported by an underlying system by using multiple hardware processors, by time-slicing a single processor, or by time-slicing multiple processors.
* Implementation of Java threads is JVM-specific.
* Though related, a Thread instance and a thread of execution aren’t the same. A Thread instance is a Java object.
* The main thread is named main by the JVM. Don’t confuse it with the method main().
* Class Thread and interface Runnable can be used to create and start a new thread of execution.
* To create your own thread objects using class Thread, you must extend it and override its method run().
* When you call start() on a Thread instance, it creates a new thread of execution.
* When a new thread of execution starts, it will execute the code defined in the thread instance’s method run(). Method start() will trigger the creation of a new thread of execution, allocating resources to it.
* Because you can’t be sure of the order of execution of threads by an underlying OS, multithreaded code might output different results when executed on the same or a different system.
* When you create a thread class by extending class Thread, you lose the flexibility of inheriting any other class.
* When you implement the Runnable interface, you must implement its method run().
* If your class implements the Runnable interface, then you should pass its instance to the constructor of class Thread.
* The Thread constructor accepts a Runnable object. A Thread instance stores a reference to a Runnable object and uses it when you start its execution (by calling start()).
* Because class Thread implements the Runnable interface, you can instantiate a thread by passing it another Thread instance.
* Each thread is created with a priority. Its range varies from 1 to 10, with 1 being the lowest priority and 10 the highest priority. By default, a thread creates another thread with the same priority as its own.
* You can’t guarantee that a thread with a higher priority will always execute before a thread with a lower priority.

***Thread lifecycle***

* You can control the transition of a thread from one state to another by calling its methods.
* The exact time of thread state transition is controlled by a thread scheduler, which is bound to vary across platforms.
* A thread can exist in multiple states: NEW, RUNNABLE, WAIT, TIMED\_WAITING, BLOCKED, or TERMINATED.
* A thread that hasn’t yet started is in the NEW state.
* Calling start() on a new thread instance implicitly calls its method run(), which transitions its state from NEW to READY.
* A thread in the RUNNABLE state is all set to be executed. It’s just waiting to be chosen by the thread scheduler so that it gets the processor time.
* As a programmer, you can’t control or determine when a particular thread transitions from the READY state to the RUNNING state, and when it actually gets to execute.
* The states READY and RUNNING are together referred to as the RUNNABLE state.
* A thread in the RUNNABLE state is executing in the JVM, but it may be waiting for other resources from the OS, such as a processor.
* A thread that’s blocked waiting for a monitor lock is in the BLOCKED state.
* A thread that’s waiting for another thread to perform an action for up to a specified waiting time is in the TIMED\_WAITING state.
* A RUNNING thread enters the TIMED\_WAITING state when you call sleep(int), join(int), or wait(int) on it.
* A thread that’s waiting indefinitely for another thread to perform a particular action is in the WAITING state.
* When you call wait() on a RUNNING thread, it transitions to the WAITING state. It can change back to the READY state when notify() or notifyAll() is called.
* A RUNNING thread might enter the BLOCKED state when it’s waiting for other system resources like network connections or to acquire an object lock to execute a synchronized method or code block. Depending on whether the thread is able to acquire the monitor lock or resources, it returns back to the READY state.
* With the successful completion of run(), a thread is in the TERMINATED state.
* A thread might transition from any state to the TERMINATED state due to an exception.

***Methods of class Thread***

* Calling start() on a Thread instance creates a new thread of execution, which executes run().
* You can call start() on a thread that’s in the NEW state. Calling start() from any other thread state will throw an IllegalThreadStateException.
* Calling run() on a Thread instance doesn’t start a new thread of execution. The run() continues to execute in the same thread.
* A thread might pause its execution due to the calling of an explicit method or when its time slice with the processor expires.
* Method yield() makes the currently executing thread pause its execution and give up its current use of the processor. But it only acts as a hint to the scheduler. The scheduler might also ignore it.
* Method yield()is static. It can be called from any method, and it doesn’t throw any exceptions.
* Method yield() can be placed literally anywhere in your code—not only in method run().
* Method sleep() is guaranteed to cause the currently executing thread to temporarily give up its execution for at least the specified number of milliseconds (and nanoseconds).
* Unless interrupted, the currently executing thread will sleep for at least the specified duration. It might not start its execution immediately after the specified time elapses.
* Method sleep() is Thread’s static method and it makes the currently executing thread give up its execution. Because all code is executed by some thread, placement of sleep() will determine which Thread instance will give up its execution.
* A thread that’s suspended due to a call to sleep() doesn’t lose ownership of any monitors.
* If thread A calls join() on a Thread instance B, A will wait for B to complete its execution before A can proceed to its own completion.
* Method join() guarantees that the calling thread won’t execute its remaining code until the thread on which it calls join() completes.
* A thread can pause its execution and wait on an object, a queue in this case, by calling wait(), until another thread calls notify() or notifyAll() on the same object.
* Methods wait(), notify(), and notifyAll() can be called on all Java objects, because they’re defined in class Object and not class Thread.
* A thread completes its execution when its method run() completes.

***Protect shared data***

* Interleaving of multiple threads that manipulate shared data using multiple steps leads to thread interference.
* A simple statement like incrementing a variable value might involve multiple steps like loading of the variable value from memory to registers (working space), incrementing the value, and reloading the new value in memory.
* When multiple threads execute this seemingly atomic statement, they might interleave, resulting in incorrect variable values.
* Making your applications thread safe means securing your shared data so that it stores correct data, even when it’s accessed by multiple threads.
* Thread safety isn’t about safe threads—it’s about safeguarding your shared data that might be accessible to multiple threads.
* A thread-safe class stores correct data without requiring calling classes to guard it.
* You can lock objects by defining synchronized methods and synchronized statements.
* Synchronized methods are defined by prefixing the definition of a method with the keyword synchronized. You can define both instance and static methods as synchronized methods.
* For non-static synchronized methods, a thread locks the monitor of the object on which the synchronized method is called. To execute static synchronized methods, a thread locks the monitor associated with the Class object of its class.
* A thread releases the lock on an object monitor after it exits a synchronized method, whether due to successful completion or due to an exception.
* To execute synchronized statements, a thread must acquire a lock on an object monitor. For instance methods an implicit lock is acquired on the object on which it’s called. For synchronized statements, you can specify an object to acquire a lock on.
* To execute synchronized statements, a lock must be acquired before the execution of the statements.
* Multiple threads can concurrently execute methods with synchronized statements if they acquire a lock on monitors of separate objects.
* A thread releases the lock on the object monitor once it exits the synchronized statement block due to successful completion or an exception.
* Immutable objects like an instance of class String and the wrapper classes (like Boolean, Long, Integer, etc.) are thread safe because their contents can’t be modified.
* You can define an immutable class by limiting access to its attributes within the class and not defining any methods to modify its state.
* Once initialized, an immutable instance doesn’t allow modification to its value.
* You can use volatile variables to synchronize access to data.
* When a thread reads from or writes to a variable (both primitive and reference variables) marked with the keyword volatile, it accesses it from the main memory, as opposed to storing its copy in the thread’s cache memory. This prevents multiple threads from storing a local copy of shared values that might not be consistent across threads.

***Identify and fix code in a multithreaded environment***

* Threading issues arise when multiple threads work with shared data and when they’re dependent on other threads.
* Local variables, method parameters, and exception handler parameters are always safe in a multithreaded application.
* Class and instance variables might not always be safe in a multithreaded application.
* Methods wait(), notify(), and notifyAll() can be used for inter thread notification.
* To call wait() or notify() a thread must own the object’s monitor lock. So calls to these methods should be placed within synchronized methods or blocks or else an IllegalMonitorStateException will be thrown by the JVM.
* All overloaded versions of wait() throw a checked InterruptedException. Methods notify() and notifyAll() don’t throw an InterruptedException.
* Unlike Thread’s method join(), which waits for another thread to complete its execution, methods wait() and notify() don’t require a thread to complete their execution.
* Multiple threads might deadlock when they have acquired a lock on objects and are waiting to acquire locks on additional objects that are owned by other waiting threads.
* All threads are assigned a priority, either implicitly or explicitly. Usually threads with a higher priority are preferred to execute by the thread scheduler. But this preference might leave threads with a lower priority starved to be scheduled.
* A thread can also starve to be scheduled when it’s waiting to acquire a lock on an object monitor that has been acquired by another thread that usually takes long to execute and is invoked frequently.
* Threads in a livelock aren’t blocked; they’re responding to each other, but they aren’t able to move to completion.
* With threads, there’s little that can be guaranteed. The Java language uses a happens-before relationship, which is when one task is guaranteed to happen before another in a multithreading environment.
* The execution of start() happens-before any action in a thread is started.
* When code is defined in a sequence, step 1 happens-before step 2.
* Unlocking of an object monitor happens-before any other thread acquires a lock on it.
* A write to a volatile field happens-before every subsequent read of that field.
* All actions in a thread happens-before any other thread returns from a join on that thread.

*11. Concurrency*

***Concurrent collection classes***

* BlockingQueue defines a first-in-first-out data structure that blocks or times out when you attempt to add to a full queue or retrieve from an empty queue.
* ConcurrentMap is a sub interface of java.util.Map that defines useful atomic operations.
* ConcurrentMap operations remove or replace a key-value pair only if the key is present, or add a key-value pair only if the key is absent. Making these operations atomic helps avoid synchronization.
* The standard general-purpose implementation of ConcurrentMap is ConcurrentHashMap, which is a concurrent analog of HashMap.
* ConcurrentNavigableMap is a sub interface of ConcurrentMap that supports approximate matches.
* The standard general-purpose implementation of ConcurrentNavigableMap is ConcurrentSkipListMap, which is a concurrent analog of TreeMap.
* A concurrent collection helps avoid memory consistency errors by defining a happens-before relationship between an operation that adds an object to the collection and subsequent operations that access or remove that object.

***Locks***

* Lock and ReadWriteLock are interfaces.
* ReentrantLock and ReentrantReadWriteLock are concrete classes.
* Lock objects offer multiple advantages over the implicit locking of an object’s monitor. Unlike an implicit lock, a thread can use explicit lock objects to wait to acquire a lock until a time duration elapses.
* Lock objects also support interruptible lock waits, non block-structured locks, lock polling, and scalability benefits.
* Method lock() acquires a lock on a Lock object. If the lock isn’t available, it waits until the lock can be acquired.
* Method lock() is comparable to intrinsic locks because it waits until a lock can be acquired on a Lock object.
* Call unlock on a Lock object to release its lock when you no longer need it.
* If you don’t call unlock() on a Lock object after acquiring a lock on it, the code will still compile successfully.
* Method tryLock() tries to acquire a lock on a Lock object, and returns immediately a boolean value specifying whether it could obtain the lock or not.
* Watch out for the use of methods acquire(), acquireLock(), release(), and releaseLock() on the exam. None of these is a valid method. Because the terms *acquire* and *release* are used to discuss methods lock(), unlock(), tryLock(), and lockInterruptibly(), these terms might be used on the exam to confuse you.
* Methods lockInterruptibly() and tryLock(long time, TimeUnit unit) of the Lock interface enable you to specify a waiting timeout or to try and acquire a lock while being available for interruption.
* Extrinsic locks or the lock on Lock objects can be acquired across methods.
* An interface, ReadWriteLock maintains a pair of associated locks, one for read only operations and another for write-only operations.
* The ReadWriteLock interface doesn’t extend Lock or any other interface.
* The ReadWriteLock interface defines only two methods, readLock() and writeLock().
* A ReentrantReadWriteLock has a read and a write lock associated with it. You can access these locks (reference variables of type Lock) by calling its methods readLock() and writeLock().
* You can acquire read locks until a write lock has been acquired on a ReadWriteLock object.
* WriteLock is an exclusive lock; it can be acquired by only one thread when no read thread has been acquired.
* The java.util.concurrent package defines multiple classes that support atomic operations of read-compare/modify-write on single variables.
* Other commonly used classes defined in the java.util.concurrent.atomic package are AtomicInteger, AtomicBoolean, AtomicLong, AtomicIntegerArray, AtomicLongArray, and AtomicReference<V>.
* The java.util.concurrent.atomic package doesn’t define classes by the names AtomicShort, AtomicByte, AtomicFloat, or AtomicDouble. These invalid class names might be used on the exam.

***Executors***

* The Executor framework enables decoupling of task submission with task execution.
* By using this framework, you can create tasks using interfaces Runnable and Callable.
* The Runnable interface defines method run() and the Callable interface defines method call().
* Executor allows you to decouple task submission and its execution.
* The Executor interface defines only one method, void execute(Runnable).
* You can define your own execution policy by implementing Executor’s method execute().
* Comparing the Runnable and Callable interface, method run() of Runnable doesn’t return a value and can’t throw a checked exception. But method call() of Callable can return a value and throw a checked exception.
* If you don’t want Callable to return a value, you can create it using Callable<Void>, defining the return type of call() as Void and returning null from it.
* The ExecutorService interface extends the Executor interface and defines methods to manage progress and termination of tasks that are submitted to it.
* Implemented using ExecutorService, thread pools use a pool of worker threads to execute new tasks. If the pool runs out of worker threads, the submitted task waits for a worker thread to become available.
* Thread pools prevent spawning of new threads to execute each new submitted task, thereby avoiding overwhelming the scheduler.
* Class Executors defines utility and factory methods for interfaces Executor, ExecutorService, and ScheduledExecutorService.
* ScheduledExecutorService extends ExecutorService and supports future or periodic execution of tasks.
* Future represents the state of an asynchronous task and can be used to query its status or cancel it.
* ScheduledExecutorService can schedule Callable or Runnable tasks that execute just once, after a given delay.
* ScheduledExecutorService can schedule only Runnable tasks (not Callable tasks) that can execute multiple times, starting its first execution after an initial delay and subsequent execution after the specified period.

***Parallel fork/join framework***

* The parallel fork/join framework makes the best use of multicore processors.
* The fork/join framework extends the existing Java concurrency package, supporting hardware parallelism, a key feature of multicore systems.
* The fork/join framework isn’t intended to replace or compete with the existing concurrency classes from the java.util.concurrent package.
* The fork/join framework works by breaking down a larger task into smaller tasks recursively, processing each unit of the task, and then combining back the results.
* You can define a recursive task using either RecursiveTask or RecursiveAction. RecursiveAction is used to define tasks that don’t return a value. RecursiveTask returns a value.
* You can instantiate a fork/join pool by using ForkJoinPool and passing it the number of processors it should use. By default, ForkJoinPool uses all the available processors.
* Execution of RecursiveAction or RecursiveTask starts when you call invoke on ForkJoinPool, passing it a RecursiveAction or RecursiveTask object, which calls their compute().

ForkJoinPool pool = new ForkJoinPool();

pool.invoke(new CalcSum(intArray));

* Method compute() determines whether the task is small enough to be executed or if it needs to be divided into multiple tasks. If the task needs to be split, a new RecursiveAction or RecursiveTask object is created, calling fork on it. Calling join on these tasks returns their result.

if (currentSize <= UNIT\_SIZE) {

return computeSum();

}

else {

int center = currentSize/2;

int leftEnd = startPos + center;

CalcSum leftSum = new CalcSum(values, startPos, leftEnd);

**leftSum.fork();**

int rightStart = startPos + center+1;

CalcSum rightSum = new CalcSum(values, rightStart, endPos);

**return(rightSum.compute() + leftSum.join());**

}

*12. Localization*