OCP Java (OCPJP) 8 Exam Quick Reference Card

Keep it handy for your exam/interview (easy reference)

Hari Kiran | Ganesh S G

Cracking OCPJP 8 exam made easy!

Making note of summary in each exam topic for your quick reference!! Here it is…..

What is this reference card all about?

If you are preparing to appear for Oracle Certified Professional Java SE 8 Programmer (OCPJP 8) certification exam, this a reference card (sort of long cheat sheet) meant to help you. You may want to print this reference card for your easy and quick reference when you prepare for your exam.

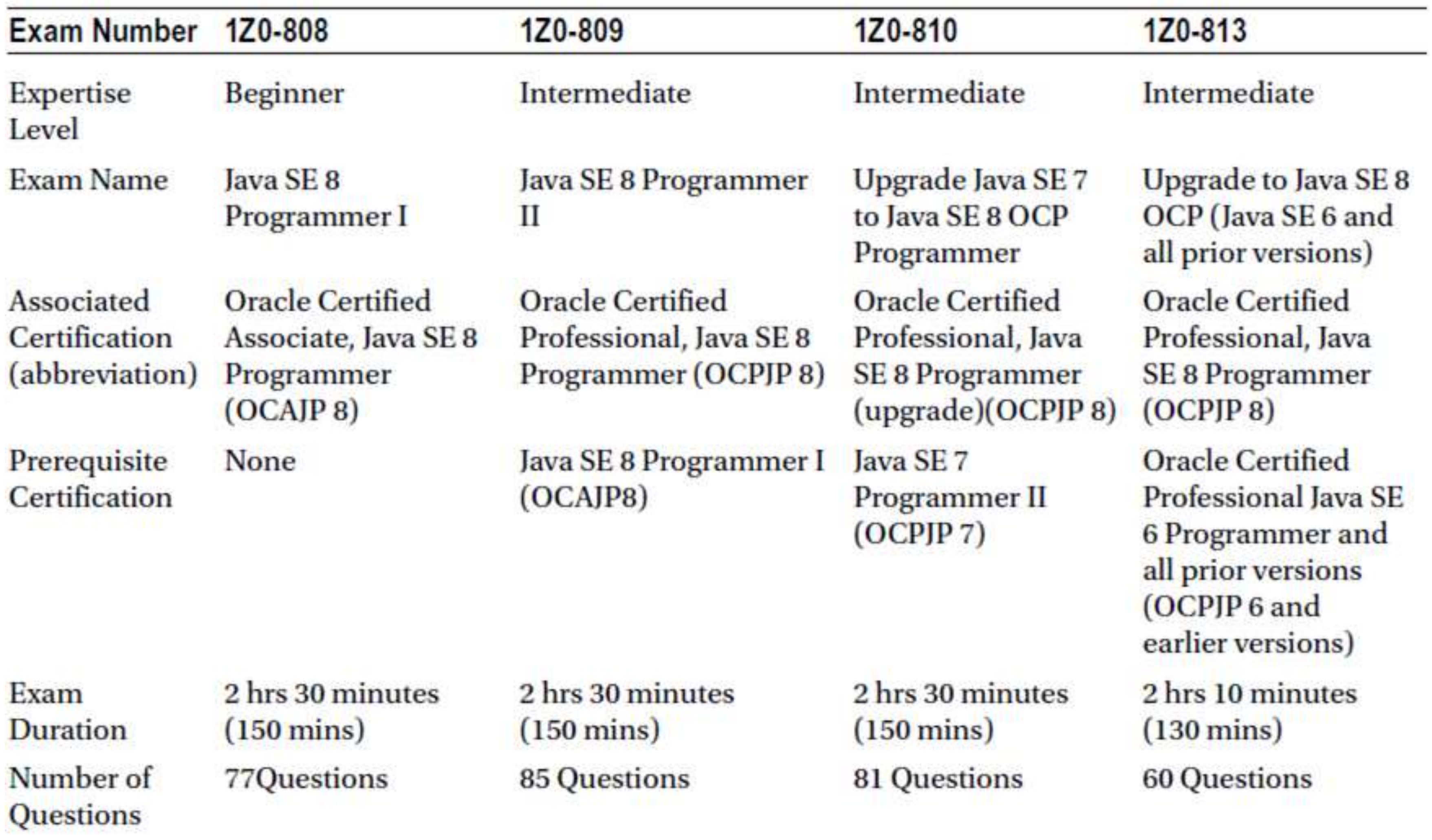
Why did we create this reference card?

OCPJP 8 covers broad range of topics with reasonable depth of understanding required by the candidate to crack the exam. As authors of the first book on OCPJP 8 exam preparation, we found through our experience that revising key concepts from exam topics will significantly helps a candidate to crack the exam, and so, we prepared this!

What is covered in this reference card?

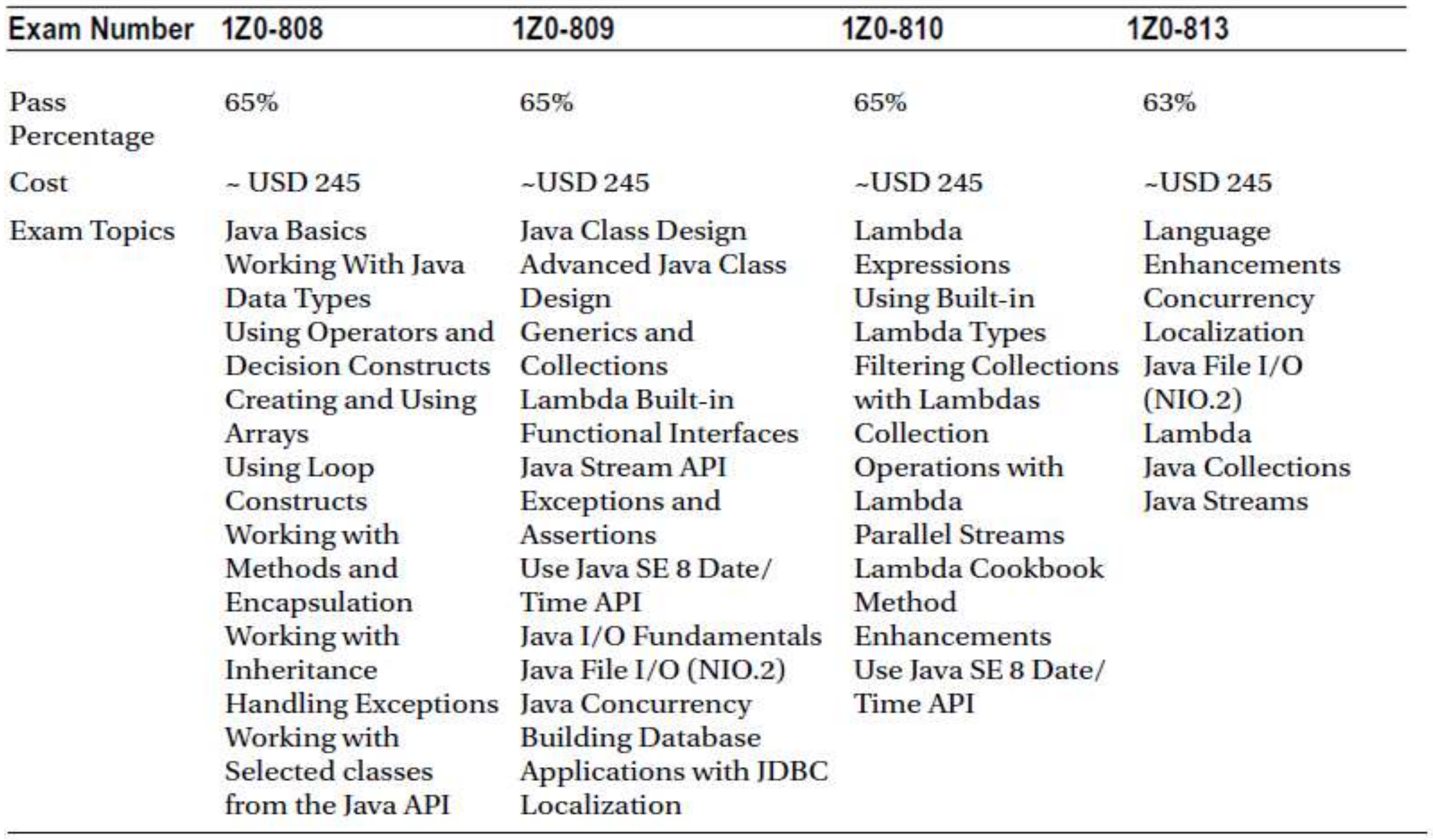
The first few pages of this reference card provides overall information on the OCPJP exam. The main meat is in the summary for each exam topic, organized by the exam sub-topics. You'll also find tables and class hierarchies which will help you remember key points and crack the exam with ease.

Comparison of the Oracle Exams Leading to OCAJP8 and OCPJP8 Certification



Exam Number	1Z0-808	1Z0-809	1Z0-810	1Z0-813
Expertise Level	Beginner	Intermediate	Intermediate	Intermediate
Exam Name	Java SE 8 Programmer I	Java SE 8 Programmer II	Upgrade Java SE 7 to Java SE 8 OCP Programmer	Upgrade to Java SE 8 OCP (Java SE 6 and all prior versions)
Associated Certification (abbreviation)	Oracle Certified Associate, Java SE 8 Programmer (OCAJP 8)	Oracle Certified Professional, Java SE 8 Programmer (OCPJP 8)	Oracle Certified Professional, Java SE 8 Programmer (upgrade)(OCPJP 8)	Oracle Certified Professional, Java SE 8 Programmer (OCPJP 8)
Prerequisite Certification	None	Java SE 8 Programmer I (OCAJP8)	Java SE 7 Programmer II (OCPJP 7)	Oracle Certified Professional Java SE 6 Programmer and all prior versions (OCPJP 6 and earlier versions)
Exam Duration	2 hrs 30 minutes (150 mins)	2 hrs 30 minutes (150 mins)	2 hrs 30 minutes (150 mins)	2 hrs 10 minutes (130 mins)
Number of Questions	77Questions	85 Questions	81 Questions	60 Questions

Comparison of the Oracle Exams Leading to OCAJP8 and OCPJP8 Certification



Exam Number	1Z0-808	1Z0-809	1Z0-810	1Z0-813
Pass Percentage	65%	65%	65%	63%
Cost	~ USD 245	~USD 245	~USD 245	~USD 245
Exam Topics	Java Basics Working With Java Data Types Using Operators and Decision Constructs Creating and Using Arrays Using Loop Constructs Working with Methods and Encapsulation Working with Inheritance Handling Exceptions Working with Selected classes from the Java API	Java Class Design Advanced Java Class Design Generics and Collections Lambda Built-in Functional Interfaces Java Stream API Exceptions and Assertions Use Java SE 8 Date/ Time API Java I/O Fundamentals Java File I/O (NIO.2) Java Concurrency Building Database Applications with JDBC Localization	Lambda Expressions Using Built-in Lambda Types Filtering Collections with Lambdas Collection Operations with Lambda Parallel Streams Lambda Cookbook Method Enhancements Use Java SE 8 Date/ Time API	Language Enhancements Concurrency Localization Java File I/O (NIO.2) Lambda Java Collections Java Streams

https://ocpjava.wordpress.com

Lets get started….

Java Class Design

Certification Objectives
Implement encapsulation
Implement inheritance including visibility modifiers and composition
Implement polymorphism
Override hashCode, equals, and toString methods from Object class
Create and use singleton classes and immutable classes
Develop code that uses static keyword on initialize blocks, variables, methods, and classes

Implement encapsulation

• *Encapsulation* : Combining data and the functions operating on it as a single unit

• You cannot access the *private* methods of the base class in the derived class.

• You can access the *protected* method either from a class in the same package (just like package private or default) as well as from a derived class.

• You can also access a method with a *default access modifier* if it is in the same package.

• You can access *public* methods of a class from any other class.

Implement inheritance including visibility modifiers and composition

• *Inheritance* : Creating hierarchical relationships between related classes. Inheritance is also called an *"IS-A“ relationship* .

• You use the super keyword to call base class methods.

• Inheritance implies IS-A and composition implies HAS-A relationship.

• Favour composition over inheritance.

Java Class Design

Access modifiers/ accessibility	Within the same class	Subclass inside the package	Subclass outside the package	Other class inside the package	Other class outside the package
Public	Yes	Yes	Yes	Yes	Yes
Private	Yes	No	No	No	No
Protected	Yes	Yes	Yes	Yes	No
Default	Yes	Yes	No	Yes	No

Implement polymorphism

• *Polymorphism* : Interpreting the same message (i.e., method call) with different meanings depending on the context.

• Resolving a method call based on the dynamic type of the object is referred to as *runtime polymorphism* .

• Overloading is an example of *static polymorphism* (*early binding*) while overriding is an example of *dynamic polymorphism* (*late binding*).

• *Method overloading* : Creating methods with same name but different types and/or numbers of parameters.

• You can have *overloaded constructors* . You can call a constructor of the same class in another constructor using the this keyword.

• *Overload resolution* is the process by which the compiler looks to resolve a call when overloaded definitions of a method are available.

• In *overriding* , the name of the method, number of arguments, types of arguments, and return type should match exactly.

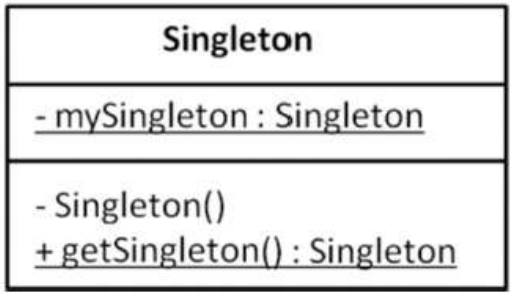
• In *covariant return types* , you can provide the derived class of the return type in the overriding method.

Java Class Design

Create and use singleton classes and immutable classes

• A singleton ensures that only one object of its class is created.

• Making sure that an intended singleton implementation is indeed singleton is a nontrivial task, especially in a multi-threaded environment.

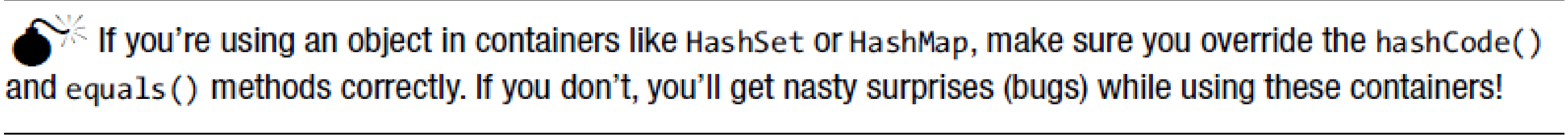


• Once an immutable object is created and initialized, it cannot be modified.

• Immutable objects are safer to use than mutable objects; further, immutable objects are thread safe; further, immutable objects that have same state can save space by sharing the state internally.

• To define an immutable class, make it final. Make all its fields private and final. Provide only accessor methods (i.e., getter methods) but don't provide mutator methods. For fields that are mutable reference types, or methods that need to mutate the state, create a deep copy of the object if needed.

Java Class Design


If you're using an object in containers like `HashSet` or `HashMap`, make sure you override the `hashCode()` and `equals()` methods correctly. If you don't, you'll get nasty surprises (bugs) while using these containers!

Override hashCode, equals, and toString methods from Object class

• You can override clone() , equals(), hashCode() , toString() and finalize() methods in your classes. Since getClass() , notify() , notifyAll() , and the overloaded versions of wait() method are declared final , you cannot override these methods.

• If you're using an object in containers like HashSet or HashMap , make sure you override the hashCode() and equals() methods correctly. For instance, ensure that the hashCode() method returns the same hash value for two objects if the equals() method returns true for them.

Develop code that uses static keyword on initialize blocks, variables, methods, and classes

• There are two types of member variables: class variables and instance variables. All variables that require an instance (object) of the class to access them are known as *instance variables* . All variables that are shared among all instances and are associated with a class rather than an object are referred to as *class variables* (declared using the static keyword).

• All static members do not require an instance to call/access them. You can directly call/access them using the class name.

• A static member can call/access only a static member of the same class.

Advanced Class Design

Certification Objectives
Develop code that uses abstract classes and methods
Develop code that uses final keyword
Create inner classes including static inner class, local class, nested class, and anonymous inner class
Use enumerated types including methods and constructors in an enum type
Develop code that declares, implements, and/or extends interfaces and use the atOverride annotation
Create and use Lambda expressions

Develop code that uses abstract classes and methods

• An abstraction specifying functionality supported without disclosing finer level details.

• You cannot create instances of an abstract class.

• Abstract classes enable runtime polymorphism, and runtime polymorphism in turn enables loose coupling.

Develop code that uses a final keyword

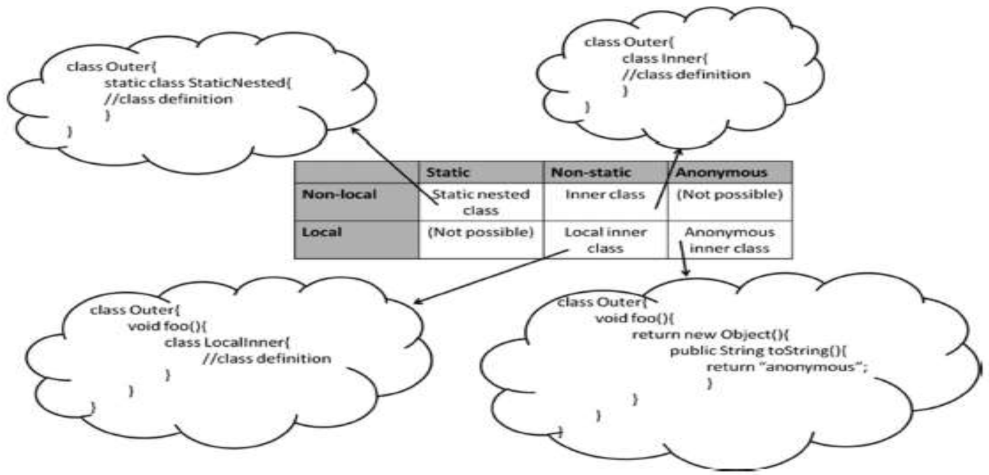
• A final class is a non-inheritable class (i.e., you cannot inherit from a final class).

• A final method is a non-overridable method (i.e., subclasses cannot override a final method).

• All methods of a final class are implicitly final (i.e., non-overridable).

• A final variable can be assigned only once.

Advanced Class Design



```
class Outer{
     static class StaticNested{
     //class definition
     }
}
```

```
class Outer{
     class Inner{
     //class definition
     }
}
```

	Static	Non-static	Anonymous
Non-local	Static nested class	Inner class	(Not possible)
Local	(Not possible)	Local inner class	Anonymous inner class

```
class Outer{
     void foo(){
          class LocalInner{
               //class definition
          }
     }
}
```

```
class Outer{
     void foo(){
          return new Object(){
               public String toString(){
                    return "anonymous";
               }
          }
     }
}
```

Create inner classes including static inner class, local class, nested class, and anonymous inner Class

• Java supports four types of nested classes: static nested classes, inner classes, local inner classes, and anonymous inner classes.

• Static nested classes may have static members, whereas the other flavors of nested classes can't.

• Static nested classes and inner classes can access members of an outer class (even private members). However, static nested classes can access only static members of outer classes.

• Local classes (both local inner classes and anonymous inner classes) can access all variables declared in the outer scope (whether a method, constructor, or a statement block).

Advanced Class Design

Use enumerated types including methods, and constructors in an enum type

• Enums are a typesafe way to achieve restricted input from users.

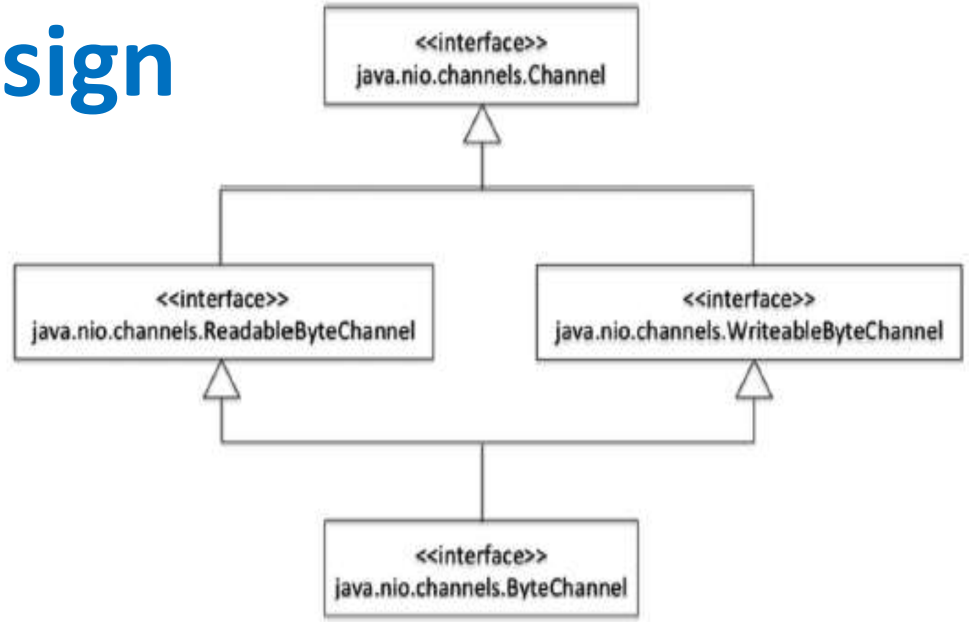
• You cannot use new with enums, even inside the enum definition.

• Enum classes are by default final classes.

• All enum classes are implicitly derived from java.lang.Enum .

Develop code that declares, implements, and/or extends interfaces and use the atOverride annotation

• An interface can have three kinds of methods: abstract methods, default methods, and static methods.



<<interface>> java.nio.channels.Channel
<<interface>> java.nio.channels.ReadableByteChannel
<<interface>> java.nio.channels.WriteableByteChannel
<<interface>> java.nio.channels.ByteChannel

• The "diamond problem" occurs when a derived type inherits two method definitions in the base types that have the same signature.

• If two super interfaces have the same method name and one of them has a definition, the compiler will issue an error; this conflict has to be resolved manually.

• If a base class and a base interface define methods with the same signature, the method definition in the class is used and the interface definition is ignored.

Advanced Class Design

One way to think about lambdas is "anonymous function" or "unnamed function": they are functions without a name and are not associated with any class. Specifically, they are NOT static or instance members of the class in which they are defined. If you use `this` keyword inside a lambda function, it refers to the object in the scope in which the lambda is defined.

Develop code that declares, implements, and/or extends interfaces and use the atOverride annotation

• A functional interface consists of exactly one abstract method but can contain any number of default or static methods.

• A declaration of a functional interface results in a "functional interface type" that can be used with lambda expressions.

• For a functional interface, declaring methods from Object class in an interface does not count as an abstract method.

Create and use Lambda expressions

• In a lambda expression, the left side of the -> provides the parameters; the right side, the body. The arrow operator ("->") helps in concise expressions of lambda functions.

• You can create a reference to a functional interface and assign a lambda expression to it. If you invoke the abstract method from that interface, it will call the assigned lambda expression.

• Compiler can perform type inferences of lambda parameters if omitted. When declared, parameters can have modifiers such as final .

• Variables accessed by a lambda function are considered to be *effectively final* .

Generics and Collections

Certification Objectives
Create and use a generic class
Create and use ArrayList, TreeSet, TreeMap, and ArrayDeque objects
Use java.util.Comparator and java.lang.Comparable interfaces
Collections Streams and Filters
Iterate using forEach methods of Streams and List
Describe Stream interface and Stream pipeline
Filter a collection by using lambda expressions
Use method references with Streams

Create and use a generic class

• Generics will ensure that any attempts to add elements of types other than the specified type(s) will be caught at compile time itself. Hence, generics offer generic implementation with type safety.

• Java 7 introduced *diamond* syntax where the type parameters (after new operator and class name) can be omitted. The compiler will infer the types from the type declaration.

• Generics are not covariant. That is, subtyping doesn't work with generics; you cannot assign a derived generic type parameter to a base type parameter.

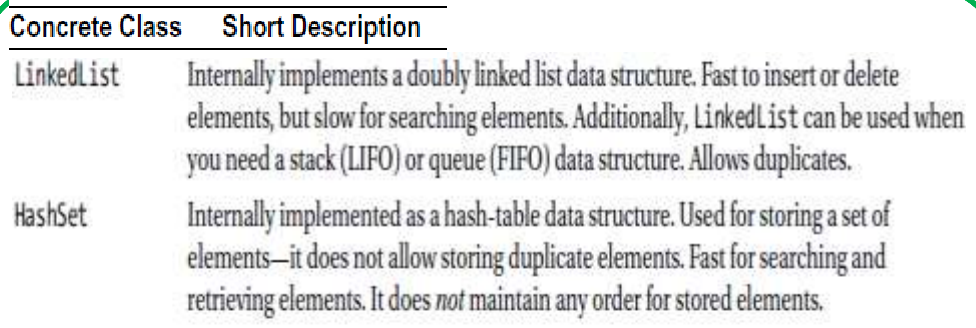
• Avoid mixing raw types with generic types. In other cases, make sure of the type safety manually.

• The <?> specifies an unknown type in generics and

Create and use ArrayList, TreeSet, TreeMap, and ArrayDeque objects

• The terms Collection, Collections, and collection are different. Collection —java.util.Collection<E> —is the root interface in the collection hierarchy. Collections —java.util.Collections —is a utility class that contains only static methods. The general term *collection(s)* refers to containers like map, stack, and queue.

Generics and Collections



Concrete Class	Short Description
LinkedList	Internally implements a doubly linked list data structure. Fast to insert or delete elements, but slow for searching elements. Additionally, LinkedList can be used when you need a stack (LIFO) or queue (FIFO) data structure. Allows duplicates.
HashSet	Internally implemented as a hash-table data structure. Used for storing a set of elements—it does not allow storing duplicate elements. Fast for searching and retrieving elements. It does *not* maintain any order for stored elements.

Create and use a generic class

•Remember that you cannot add or remove elements to the List returned by the Arrays.asList() method. But, you can make changes to the elements in the returned List , and the changes made to that List are reflected back in the array.

• A HashSet is for quickly inserting and retrieving elements; it does *not* maintain any sorting order for the elements it holds. A TreeSet stores the elements in a sorted order (and it implements the SortedSet interface).

• A HashMap uses a hash table data structure internally. In HashMap, searching (or looking up elements) is a fast operation. However, HashMap neither remembers the order in which you inserted elements nor keeps elements in any sorted order. Unlike HashMap, TreeMap keeps the elements in sorted order (i.e., sorted by its keys). So, searching or inserting is somewhat slower than the HashMap.

• Deque (Doubly ended queue) is a data structure that allows you to insert and remove elements from both ends. There are three concrete implementations of the Deque interface: LinkedList, ArrayDeque, and LinkedBlockingDeque .

• The difference between an ArrayList and ArrayDeque is that you can add an element anywhere in an array list using an index; however, you can add an element only either at the front or end of the array deque.

Generics and Collections

Most classes have a natural order for comparing objects, so use `Comparable` interface in those cases. If you want to compare the objects other than the natural order or if there is no natural ordering present for your class type, then use the `Comparator` interface.

Use java.util.Comparator and java.lang.Comparable interfaces

• Implement the Comparable interface for your classes where a natural order is possible. If you want to compare the objects other than the natural order or if there is no natural ordering present for your class type, then create separate classes implementing the Comparator interface. Also, if you have multiple alternative ways to decide the order, then go for the Comparator interface.).

Collections Streams and Filters

• The new stream API is provided in the java.util.stream package introduced in Java 8. The main type in this package is Stream<T> interface, which is the stream of object references. IntStream , LongStream , and DoubleStream are streams for primitive types int , long , and double types respectively.

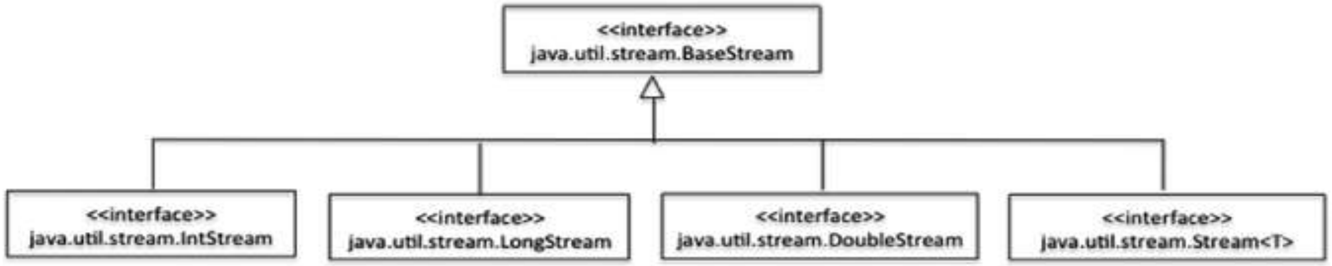
• A stream is a sequence of elements. In Java 8, the Collection interface has been added with the methods stream() and parallelStream() from which you can respectively get sequential and parallel streams.

Iterate using forEach methods of Streams and List

• In Java 8, we are moving from external iteration to internal iteration. It is a *major* change with Java 8 approach of functional programming.

• The interfaces Stream and Iterable define forEach() method. The forEach() method supports internal iteration.

Generics and Collections


<<interface>> java.util.stream.BaseStream
<<interface>> java.util.stream.IntStream
<<interface>> java.util.stream.LongStream
<<interface>> java.util.stream.DoubleStream
<<interface>> java.util.stream.Stream<T>

Describe Stream interface and Stream pipeline

• Stream operations can be "chained" together to form a pipeline known as "stream pipeline".

• A stream pipeline has a beginning, middle, and an end: *source* (that creates a stream), *intermediate operations* (that consist of optional operations that can be chained together), and *terminal operations* (that produce a result).

• The terminal operation can produce a result, accumulate the stream elements, or just perform an action.

• You can use a stream only once. Any attempt at reusing the stream (for example, by calling intermediate or terminal operations) will result in throwing an IllegalStateException .

Filter a collection by using lambda expressions

• The filter() method in the Stream interface is used for removing the elements that do not match the given condition.

Use method references with Streams

• When lambda expressions just route the given parameters, you can use method references instead.

• Since method references serve as a way to route the parameters, it is often convenient (as it results in more concise code) to use them than their equivalent lambda expressions.

Lambda Built-in Functional Interfaces

Certification Objectives
Use the built-in interfaces included in the java.util.function package such as Predicate, Consumer, Function, and Supplier
Develop code that uses primitive versions of functional interfaces
Develop code that uses binary versions of functional interfaces
Develop code that uses the UnaryOperator interface

Use the built-in interfaces included in the java.util.function package such as Predicate, Consumer, Function, and Supplier

• Built-in functional interfaces Predicate , Consumer , Function , and Supplier differ mainly based on the signature of the abstract method they declare.

• A Predicate tests the given condition and returns true or false ; hence it has an abstract method named " test " that takes a parameter of generic type T and returns boolean type.

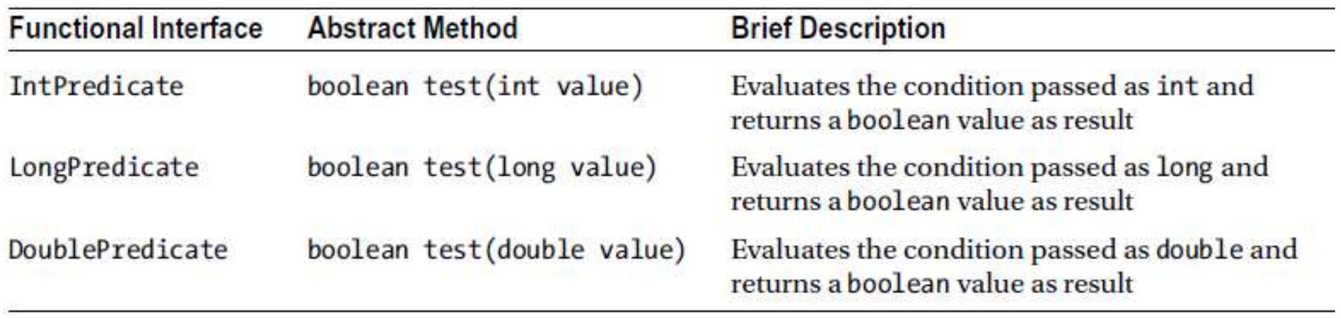
• A Consumer "consumes" an object and returns nothing; hence it has an abstract method named " accept " that takes an argument of generic type T and has return type void .

• A Function "operates" on the argument and returns a result; hence it has an abstract method named "apply" that takes an argument of generic type T and has generic return type R .

• A Supplier "supplies" takes nothing but returns something; hence it has an abstract method named "get" that takes no arguments and returns a generic type T .

• The forEach() method defined in Iterable (implemented by collection classes) method accepts a Consumer<T> .

Lambda Built-in Functional Interfaces



Functional Interface	Abstract Method	Brief Description
`IntPredicate`	`boolean test(int value)`	Evaluates the condition passed as `int` and returns a `boolean` value as result
`LongPredicate`	`boolean test(long value)`	Evaluates the condition passed as `long` and returns a `boolean` value as result
`DoublePredicate`	`boolean test(double value)`	Evaluates the condition passed as `double` and returns a `boolean` value as result

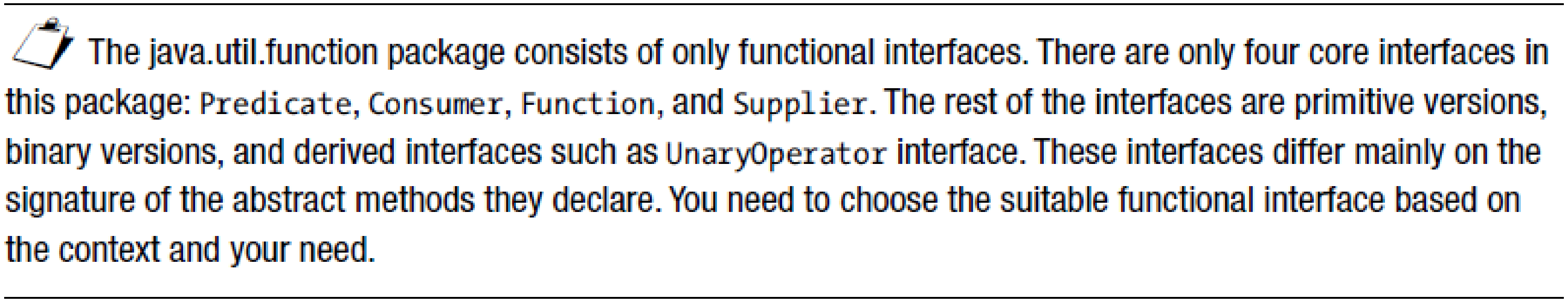
Develop code that uses primitive versions of functional interfaces

• The built-in interfaces Predicate , Consumer , Function , and Supplier operate on reference type objects. For primitive types, there are specializations available for int, long, and double types for these functional interfaces.

• When the Stream interface is used with primitive types, it results in unnecessary boxing and unboxing of the primitive types to their wrapper types. This results in slower code as well as wastes memory because the unnecessary wrapper objects get created. Hence, whenever possible, prefer using the primitive type specializations of the functional interfaces Predicate, Consumer, Function, and Supplier .

• The primitive versions of the functional interfaces Predicate , Consumer , Function , and Supplier are available only for int, long and double types (and boolean type in addition to these three types for Supplier). You have to use implicit conversions to relevant int version when you need to use char, byte, or short types; similarly, you can use the version for double type when you need to use float .

Lambda Built-in Functional Interfaces


The java.util.function package consists of only functional interfaces. There are only four core interfaces in this package: `Predicate`, `Consumer`, `Function`, and `Supplier`. The rest of the interfaces are primitive versions, binary versions, and derived interfaces such as `UnaryOperator` interface. These interfaces differ mainly on the signature of the abstract methods they declare. You need to choose the suitable functional interface based on the context and your need.

Develop code that uses binary versions of functional interfaces

• The functional interfaces BiPredicate, BiConsumer, and BiFunction are binary versions of Predicate , Consumer, and Function respectively. There is no binary equivalent for Supplier since it does not take any arguments. The prefix “Bi” indicates the version that takes “two” arguments.

Develop code that uses the UnaryOperator interface

• UnaryOperator is a functional interface and it extends Function interface.

• The primitive type specializations of UnaryOperator are IntUnaryOperator, LongUnaryOperator, and DoubleUnaryOperator for int, long, and double types respectively.

Java Stream API

Certification Objectives
Develop code to extract data from an object using peek() and map() methods including primitive versions of the map() method
Search for data by using search methods of the Stream classes including findFirst, findAny, anyMatch, allMatch, noneMatch
Develop code that uses the Optional class
Develop code that uses Stream data methods and calculation methods
Sort a collection using Stream API
Save results to a collection using the collect method and group/partition data using the Collectors class
Use flatMap() methods in the Stream API

Develop code to extract data from an object using peek() and map() methods including primitive versions of the map() method

• The peek() method is useful for debugging: it helps us understand how the elements are transformed in the pipeline.

• You can transform (or just extract) elements in a stream using map() method

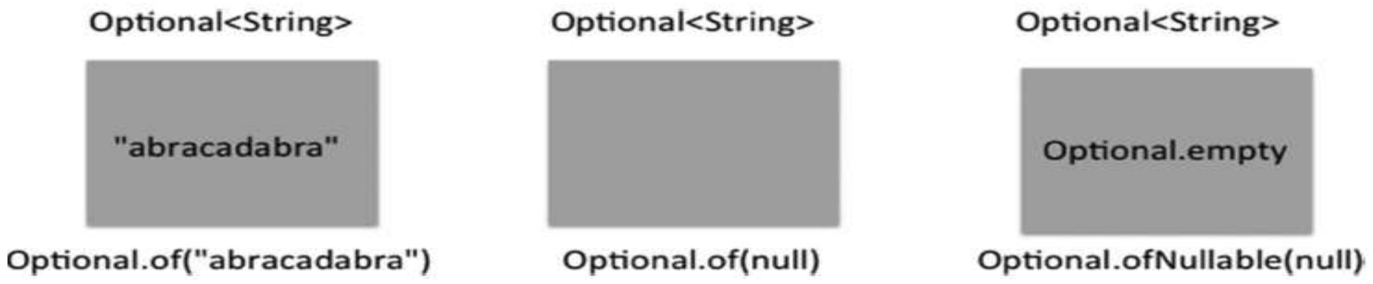
Search for data by using search methods of the Stream classes including findFirst, findAny, anyMatch, allMatch, noneMatch

• You can match for a given predicate in a stream using the allMatch(), noneMatch(), and anyMatch() methods. Unlike the anyMatch() method that returns false when the stream is empty, the allMatch() and noneMatch() methods return true if the stream is empty.

• You can look for elements in a stream using the findFirst() and findAny() methods. The findAny() method is faster to use than the findFirst() method in case of parallel streams.

• The “match” and “find” methods “short-circuit”: the evaluation stops once the result is found and the rest of the stream is not evaluated.

Java Stream API



Develop code that uses the Optional class

• When there are no entries in a stream and operations such as max are called, then instead of returning null or throwing an exception, the (better) approach taken in Java 8 is to return Optional values.

• Primitive type versions of Optional<T> for int, long, and double are OptionalInteger, OptionalLong, and OptionalDouble respectively.

Develop code that uses Stream data methods and calculation methods

• The Stream<T> interface has data and calculation methods count(), min() and max() ; you need to pass a Comparator object as the parameter when invoking these min() and max() methods.

• The primitive type versions of Stream interface have the following data and calculation methods: count(), sum(), average(), min(), and max() .

• The summaryStatistics() method in IntStream, LongStream, and DoubleStream have methods for calculating count, sum, average, minimum, and maximum values of elements in the stream.

Java Stream API

The `flatMap()` method operates on elements just like `map()` method. However, `flatMap()` flattens the streams that result from mapping each of its elements into one flat stream.

Sort a collection using Stream API

• One way to sort a collection is to get a stream from the collection and call sorted() method on that stream. The sorted() method sorts the elements in the stream in natural order (it requires that the stream elements implements the Comparable interface).

• When you want to sort elements in the stream other than the natural order, you can pass a Comparator object to the sorted() method.

• The Comparator interface has been enhanced with many useful static or default methods in Java 8 such as thenComparing() and reversed() methods

Save results to a collection using the collect method and group/partition data using the Collectors class

• The collect() method of the Collectors class has methods that support the task of collecting elements to a collection.

• The Collectors class provides methods such as toList(), toSet(), toMap(), and toCollection() to create a collection from a stream.

• You can group the elements in a stream using the Collectors.groupingBy() method and pass the criteria for grouping (given as a Function) as the argument.

Use flatMap() method of the Stream API

• The flatMap() method in Stream flattens the streams that result from mapping each element into one flat stream.

Exceptions and Assertions

Certification Objectives
Use try-catch and throw statements
Use catch, multi-catch, and finally clauses
Use Autoclose resources with a try-with-resources statement
Create custom exceptions and Auto-closeable resources
Test invariants by using assertions

Use try-catch and throw statements

• When an exception is thrown from a try block, the JVM looks for a matching catch handler from the list of catch handlers in the method call-chain. If no matching handler is found, that unhandled exception will result in crashing the application.

• While providing multiple exception handlers (stacked catch handlers), specific exception handlers should be provided before general exception handlers.

• You can programmatically access the stack trace using the methods such as printStackTrace() and getStackTrace(), which can be called on any exception object.

Use catch, multi-catch, and finally clauses

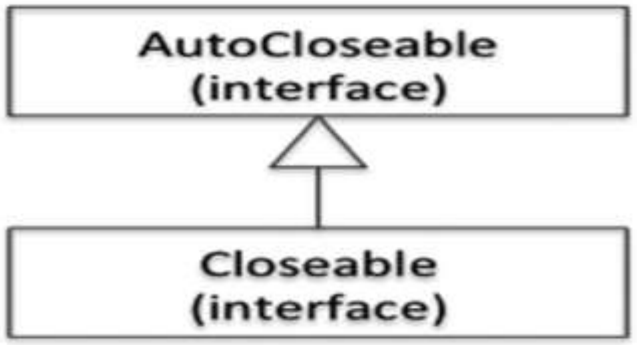
• A try block can have multiple catch handlers. If the cause of two or more exceptions is similar, and the handling code is also similar, you can consider combining the handlers and make it into a multi-catch block.

• A catch block should either handle the exception or rethrow it. To *hide* or *swallow* an exception by catching an exception and doing nothing is really a bad practice.

• You can wrap one exception and throw it as another exception. These two exceptions become *chained exceptions* .

• The code inside a finally block will be executed irrespective of whether a try block has successfully executed or resulted in an exception.

Exceptions and Assertions



Closeabe Interface extends AutoCloseable Interface

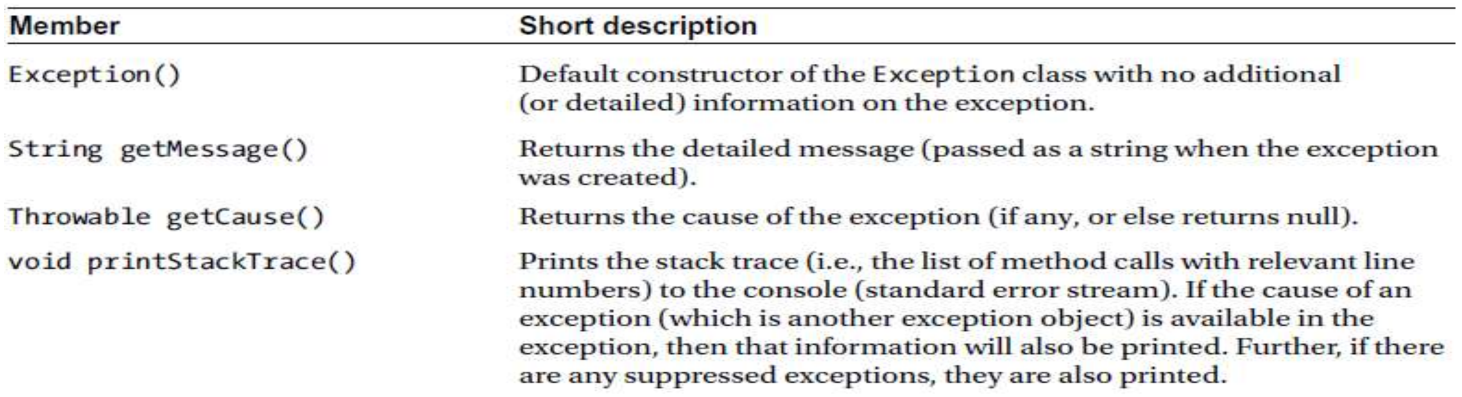
Use autoclose resources with a try-with-resources statement

• Forgetting to release resources by explicitly calling the close() method is a common mistake. You can use a try-with-resources statement to simplify your code and auto-close resources.

• You can auto-close multiple resources within a try-with-resources statement. These resources need to be separated by semicolons in the try-with-resources statement header.

• If a try block throws an exception, and a finally block also throws exception(s), then the exceptions thrown in the finally block will be added as suppressed exceptions to the exception that gets thrown out of the try block to the caller.

Exceptions and Assertions



Member	Short description
`Exception()`	Default constructor of the Exception class with no additional (or detailed) information on the exception.
`String getMessage()`	Returns the detailed message (passed as a string when the exception was created).
`Throwable getCause()`	Returns the cause of the exception (if any, or else returns null).
`void printStackTrace()`	Prints the stack trace (i.e., the list of method calls with relevant line numbers) to the console (standard error stream). If the cause of an exception (which is another exception object) is available in the exception, then that information will also be printed. Further, if there are any suppressed exceptions, they are also printed.

Important Methods and Constructors of the Exception Class

Create custom exceptions and auto-closeable resources

- It is recommended that you derive custom exceptions from either the Exception or RuntimeException class.

- A method's throws clause is part of the contract that its overriding methods in derived classes should obey.

- An overriding method can provide the same throw clause as the base method's throws clause or a more specific throws clause than the base method's throws clause.

- The overriding method cannot provide a more general throws clause or declare to throw additional checked exceptions when compared to the base method's throws clause.

- For a resource to be usable in a try-with-resources statement, the class of that resource must implement the java.lang.AutoCloseable interface and define the close() method.

Exceptions and Assertions

Command-Line Argument	Short Description
`-ea`	Enables assertions by default (except system classes).
`-ea:<class name>`	Enables assertions for the given class name.
`-ea:<package name>...`	Enables assertions in all the members of the given package <package name>.
`-ea:...`	Enable assertions in the given unnamed package.

Test invariants by using assertions

• Assertions are condition checks in the program and should be used for explicitly checking the assumptions you make while writing programs.

• The assert statement is of two forms: one that takes a Boolean argument and one that takes an additional string argument.

• If the Boolean condition given in the assert argument fails (i.e., evaluates to false), the program will terminate after throwing an AssertionError . It is not advisable to catch and recover from when an AssertionError is thrown by the program.

• By default, assertions are disabled at runtime. You can use the command-line arguments of –ea (for enabling asserts) and –da (for disabling asserts) and their variants when you invoke the JVM.

Using the Java SE 8 Date/Time API

Certification Objectives
Create and manage date-based and time-based events including a combination of date and time into a single object using LocalDate, LocalTime, LocalDateTime, Instant, Period, and Duration
Work with dates and times across timezones and manage changes resulting from daylight savings including format date and times values
Define and create and manage date-based and time-based events using Instant, Period, Duration, and TemporalUnit

Create and manage date-based and time-based events including a combination of date and time into a single object using LocalDate, LocalTime, LocalDateTime, Instant, Period, and Duration

• The Java 8 date and time API uses ISO 8601 as the default calendar format.

• The java.time.LocalDate class represents a date without time or time zones; the java.time.LocalTime class represents time without dates and time zones; the java.time.LocalDateTime class represents both date and time without time zones.

• The java.time.Instant class represents a Unix timestamp.

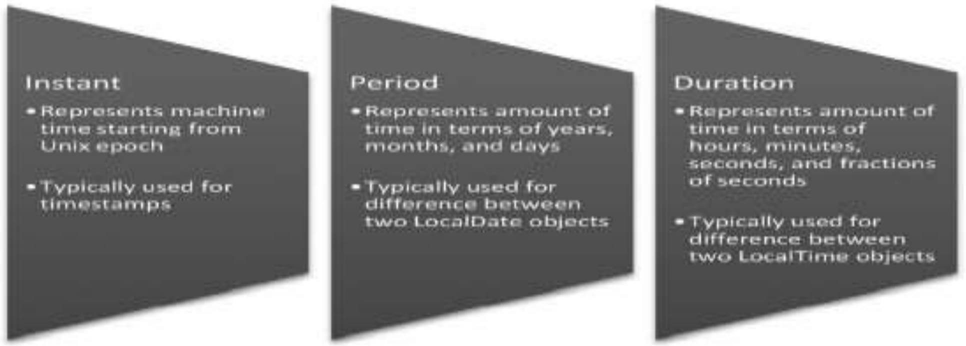
• The java.time.Period is used to measure the amount of time in terms of years, months, and days.

• The java.time.Duration class represents time in terms of hours, minutes, seconds, and fraction of seconds.

Work with dates and times across timezones and manage changes resulting from daylight savings including Format date and times values

• ZoneId identifies a time zone; ZoneOffset represents time zone offset from UTC/Greenwich.

Using the Java SE 8 Date/Time API


Instant
• Represents machine time starting from Unix epoch
• Typically used for timestamps
Period
• Represents amount of time in terms of years, months, and days
• Typically used for difference between two LocalDate objects
Duration
• Represents amount of time in terms of hours, minutes, seconds, and fractions of seconds
• Typically used for difference between two LocalTime objects

• ZoneId identifies a time zone; ZoneOffset represents time zone offset from UTC/Greenwich.

• ZonedDateTime provides support for all three aspects: date, time, and time zone.

• You have to account for daylight savings time (DST) when working with different time zones.

• The java.time.format.DateTimeFormatter class provides support for reading or printing date and time values in different formats.

• The DateTimeFormatter class provides predefined constants (such as ISO_DATE and ISO_TIME) for formatting date and time values.

• You encode the format of the date or time using case-sensitive letters to form a date or time pattern string with the DateTimeFormatter class.

Define and create and manage date-based and time-based events using Instant, Period, Duration, and TemporalUnit

• The enumeration java.time.temporal.ChronoUnit implements the java.time.temporal.TemporalUnit interface.

• Both TemporalUnit and ChronoUnit deal with time unit values such as seconds, minutes, and hours and date values such as days, months, and years.

Java I/O Fundamentals

Certification Objectives
Read and write data from the console
Use BufferedReader, BufferedWriter, File, FileReader, FileWriter, FileInputStream, FileOutputStream, ObjectOutputStream, ObjectInputStream, and PrintWriter in the java.io package

Read and write data from the console

• The public static fields in , out , and err in java.lang.System class respectively represent the standard input, output and error streams. System.in is of type java.io.InputStream and System.out and System.err are of type java.io.PrintStream .

• You can redirect standard streams by calling the methods System.setIn , System.setOut and System.setError .

• You can obtain a reference to the console using the System.console() method; if the JVM is not associated with any console, this method will fail and return null.

• Many methods are provided in Console-support formatted I/O. You can use the printf() and format() methods available in the Console class to print formatted text; the overloaded readLine() and readPassword() methods take format strings as arguments.

• The template of format specifiers is: %[flags][width][.precision]datatype_specifierEach format specifier starts with the % sign, followed by flags, width, and precision information, and ending with a data type specifier. In the format string, the flags, width, and precision information are optional but the % sign and data type specifier are mandatory.

• Use the readPassword() method for reading secure strings such as passwords. It is recommended to use Array ’s fill() method to “empty” the password read into the character array (to avoid malicious access to the typed passwords).

Java I/O Fundamentals

Use BufferedReader, BufferedWriter, File, FileReader, FileWriter, FileInputStream, FileOutputStream, ObjectOutputStream, ObjectInputStream, and PrintWriter in the java.io package

• The java.io package has classes supporting both character streams and byte streams.

• You can use character streams for text-based I/O. Byte streams are used for databased I/O.

• Character streams for reading and writing are called *readers* and *writers* respectively (represented by the abstract classes of Reader and Writer).

• Byte streams for reading and writing are called *input streams* and *output streams* respectively (represented by the abstract classes of InputStream and OutputStream).

• You should only use character streams for processing text files (or human-readable files), and byte streams for data files. If you try using one type of stream instead of another, your program won't work as you would expect; even if it works by chance, you'll get nasty bugs. So don't mix up streams, and use the right stream for a given task at hand.

• For both byte and character streams, you can use buffering. The buffer classes are provided as wrapper classes for the underlying streams. Using buffering will speed up the I/O when performing bulk I/O operations.

• For processing data with primitive data types and strings, you can use data streams.

• You can use object streams (classes ObjectInputStream and ObjectOutputStream) for reading and writing objects in memory to files and vice versa.

Java File I/O (NIO.2)

Certification Objectives
Use Path interface to operate on file and directory paths
Use Files class to check, read, delete, copy, move, manage metadata of a file or directory
Use Stream API with NIO.2

Use Path interface to operate on file and directory paths

• A Path object is a programming abstraction to represent the path of a file/directory.

• You can get an instance of Path using the get() method of the Paths class.

• Path provides two methods to compare Path objects: equals() and compareTo(). Even if two Path objects point to the same file/directory, the equals() method is not guaranteed to return true.

Use Files class to check, read, delete, copy, move, manage metadata of a file or directory

• You can check the existence of a file using the exists() method of the Files class.

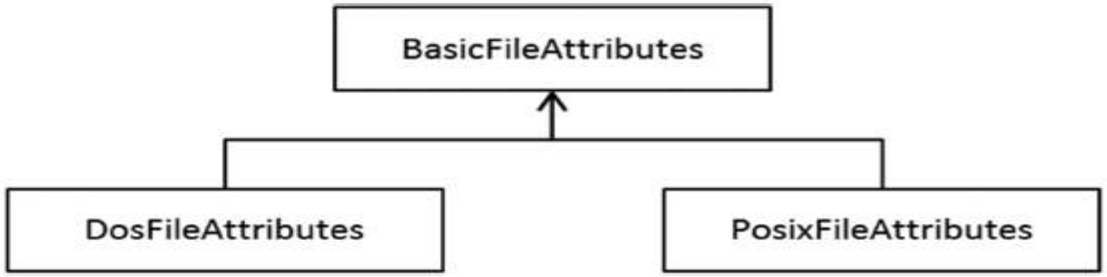
• The Files class provides the methods isReadable() , isWritable() , and isExecutable() to check the ability of the program to read, write, and execute programmatically, respectively.

• You can retrieve the attributes of a file using the getAttributes() method.

• You can use the readAttributes() method of the Files class to read file attributes in bulk.

• The copy() method can be used to copy a file from one location to another. Similarly, the move() method moves a file from one location to another.

Java File I/O (NIO.2)



The hierarchy of BasicFileAttributes

• While copying, all the directories (except the last one, if you are copying a directory) on the specified path must exist to avoid a NoSuchFileException .

• Use the delete() method to delete a file; use the deleteIfExists() method to delete a file only if it exists.

Use Stream API with NIO.2

• The Files.list() method returns a Stream<Path>. It does not recursively traverse the directories in the given Path .

• The Files.walk() method returns a Stream<Path> by recursively traversing the entries from the given Path; in one of its overloaded versions, you can also pass the maximum depth for such traversal and provide FileVisitOption.FOLLOW_LINKS as the third option.

• The Files.find() method returns a Stream<Path> by recursively traversing the entries from the given Path; it also takes the maximum depth to search, a BiPredicate , and an optional FileVisitOption as arguments.

• Files.lines() is a very convenient method to read the contents of a file. It returns a Stream<String>.

Java Concurrency

Certification Objectives
Create worker threads using Runnable, Callable, and use an ExecutorService to concurrently execute tasks
Identify potential threading problems among deadlock, starvation, livelock, and race conditions
Use synchronized keyword and java.util.concurrent.atomic package to control the order of thread execution
Use java.util.concurrent collections and classes including CyclicBarrier and CopyOnWriteArrayList
Use parallel Fork/Join Framework
Use parallel Streams including reduction, decomposition, merging processes, pipelines, and performance

Create worker threads using Runnable, Callable, and use an ExecutorService to concurrently execute tasks

• You can create classes that are capable of multi-threading by implementing the Runnable interface or by extending the Thread class.

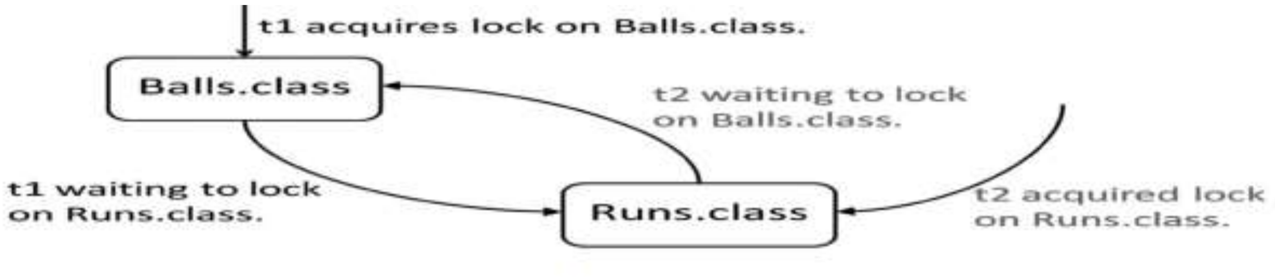
• Always implement the run() method. The default run() method in Thread does nothing.

• Call the start() method and not the run() method directly in code. (Leave it to the JVM to call the run() method.)

• The Callable interface represents a task that needs to be completed by a thread. Once the task completes, the call() method of a Callable implementation returns a value.

• The Executor hierarchy abstracts the lower-level details of multi-threaded programming and offers high-level user-friendly concurrency constructs.

Java Concurrency


t1 acquires lock on Balls.class.
Balls.class
t2 waiting to lock on Balls.class.
t1 waiting to lock on Runs.class.
Runs.class
t2 acquired lock on Runs.class.

Deadlock between threads t1 and t2

Identify potential threading problems among deadlock, starvation, livelock, and race conditions

• Concurrent reads and writes to resources may lead to the *race condition* problem.

• You must use thread synchronization (i.e., locks) to access shared values and avoid race conditions. Java provides thread synchronization features to provide protected access to shared resources—namely, synchronized blocks and synchronized methods.

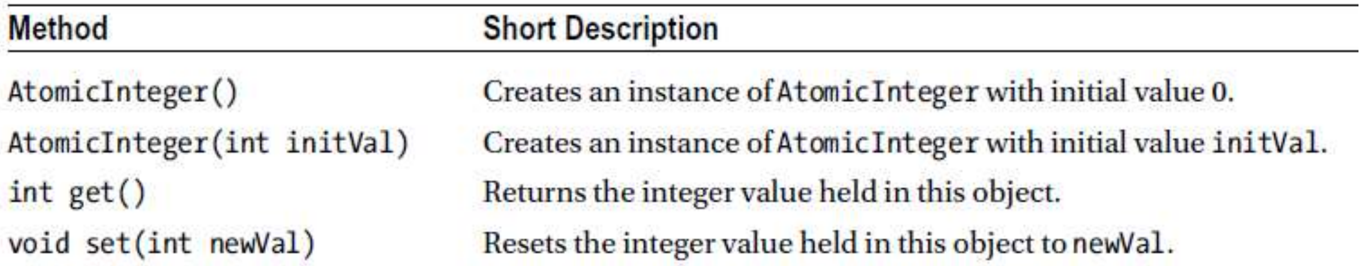
• Using locks can introduce problems such as deadlocks. When a deadlock happens, the process will *hang* and will never terminate.

• A deadlock typically happens when two threads acquire locks in opposite order. When one thread has acquired one lock and waits for another lock, another thread has acquired that other lock and waits for the first lock to be released. So, no progress is made and the program deadlocks.

• When a change done by a thread is repeatedly undone by another thread, both the threads are busy but the application as a whole does not make progress; this situation is known as a livelock.

• The situation in which low-priority threads “starve” for a long time trying to obtain the lock is known as lock starvation.

Java Concurrency



Method	Short Description
`AtomicInteger()`	Creates an instance of `AtomicInteger` with initial value 0.
`AtomicInteger(int initVal)`	Creates an instance of `AtomicInteger` with initial value `initVal`.
`int get()`	Returns the integer value held in this object.
`void set(int newVal)`	Resets the integer value held in this object to `newVal`.

Important Methods in the AtomicInteger Class

Use synchronized keyword and java.util.concurrent.atomic package to control the order of thread execution

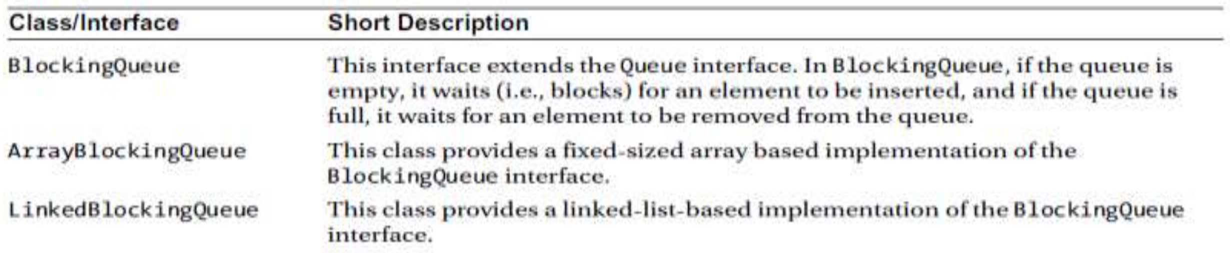
• In synchronized blocks, you use the synchronized keyword for a reference variable and follow it by a block of code. A thread has to acquire a lock on the synchronized variable to enter the block; when the execution of the block completes, the thread releases the lock.

• Java provides an efficient alternative in the form of atomic variables where one needs to acquire and release a lock just to carry out primitive operations on variables.

• A lock ensures that only one thread accesses a shared resource at a time.

• Performing locking and unlocking for performing operations on primitive types is inefficient. A better alternative is to use atomic variables provided in java.util.concurrent.atomic package including AtomicBoolean , AtomicInteger , AtomicIntegerArray , AtomicLong , AtomicLongArray , AtomicReference<V> , and AtomicReferenceArray<E>.

Java Concurrency



Class/Interface	Short Description
BlockingQueue	This interface extends the Queue interface. In BlockingQueue, if the queue is empty, it waits (i.e., blocks) for an element to be inserted, and if the queue is full, it waits for an element to be removed from the queue.
ArrayBlockingQueue	This class provides a fixed-sized array based implementation of the BlockingQueue interface.
LinkedBlockingQueue	This class provides a linked-list-based implementation of the BlockingQueue interface.

Some Concurrent Collection Classes/Interfaces in the java.util.concurrent Package

Use java.util.concurrent collections and classes including CyclicBarrier and CopyOnWriteArrayList

• Semaphore controls access to one or more shared resources.

• CountDownLatch allows threads to wait for a countdown to complete.

• Exchanger supports exchanging data between two threads.

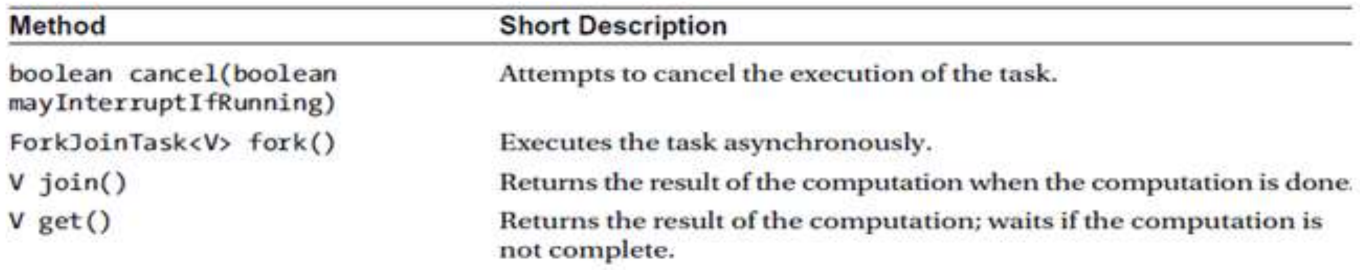
• Phaser is used to support a synchronization barrier.

• CyclicBarrier enables threads to wait at a predefined execution point.

• The java.util.concurrent package provides a number of classes that are threadsafe equivalents of the ones provided in the collections framework classes in the java.util package; for example, java.util.concurrent.ConcurrentHashMap is a concurrent equivalent to java.util.HashMap .

• CopyOnWriteArrayList is similar to ArrayList , but provides safe concurrent access. When you modify a CopyOnWriteArrayList, a fresh copy of the underlying array is created.

Java Concurrency



Method	Short Description
`boolean cancel(boolean mayInterruptIfRunning)`	Attempts to cancel the execution of the task.
`ForkJoinTask<V> fork()`	Executes the task asynchronously.
`V join()`	Returns the result of the computation when the computation is done.
`V get()`	Returns the result of the computation; waits if the computation is not complete.

Important Methods in the ForkJoinTask Class

Use Parallel Fork/Join Framework

• The Fork/Join framework is a portable means of executing a program with decent parallelism.

• The framework is an implementation of the ExecutorService interface and provides an easy-to-use concurrent platform in order to exploit multiple processors.

• This framework is very useful for modeling divide-and-conquer problems.

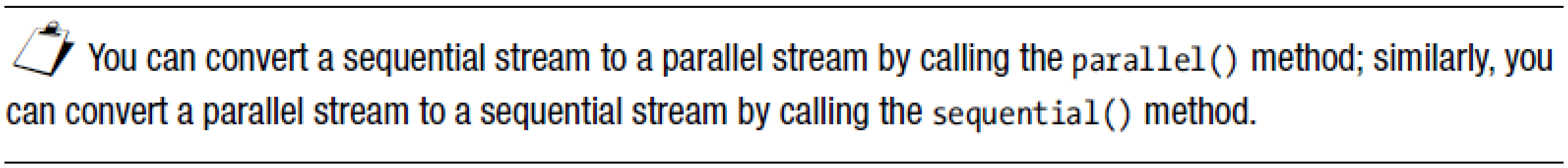
• The Fork/Join framework uses the work stealing algorithm: when a worker thread completes its work and is free, it takes (or “steals”) work from other threads that are still busy doing some work.

• The work-stealing technique results in decent load balancing thread management with minimal synchronization cost.

• ForkJoinPool is the most important class in the Fork/Join framework. It is a thread pool for running fork/join tasks—it executes an instance of ForkJoinTask. It executes tasks and manages their lifecycles.

• ForkJoinTask<V> is a lightweight thread-like entity representing a task that defines methods such as fork() and join().

Java Concurrency



You can convert a sequential stream to a parallel stream by calling the `parallel()` method; similarly, you can convert a parallel stream to a sequential stream by calling the `sequential()` method.

Use parallel Streams including reduction, decomposition, merging processes, pipelines, and Performance

• Parallel streams split the elements into multiple chunks, process each chunk with different threads, and (if necessary) combine the results from those threads to evaluate the final result.

• When you call the stream() method of the Collection class, you will get a sequential stream. When you call the parallelStream() method of the Collection class, you will get a parallel stream.

• Parallel streams internally use the fork/join framework. To use parallel streams correctly, the process steps should consist of stateless and independent tasks.

• You can convert a sequential stream to a parallel stream by calling the parallel() method; similarly, you can convert a parallel stream to a sequential stream by calling the sequential() method.

• You can check if the stream is sequential or parallel by calling the isParallel() method.

Building Database Applications with JDBC

Certification Objectives
Describe the interfaces that make up the core of the JDBC API including the Driver, Connection, Statement, and ResultSet interfaces and their relationship to provider implementations
Identify the components required to connect to a database using the DriverManager class including the JDBC URL
Submit queries and read results from the database including creating statements, returning result sets, iterating through the results, and properly closing result sets, statements, and connections

Identify the components required to connect to a database using the DriverManager class including the JDBC URL

• JDBC hides the heterogeneity of all the DBMSs and offers a single set of APIs to interact with all types of databases. The complexity of heterogeneous interactions is delegated to the JDBC driver manager and JDBC drivers.

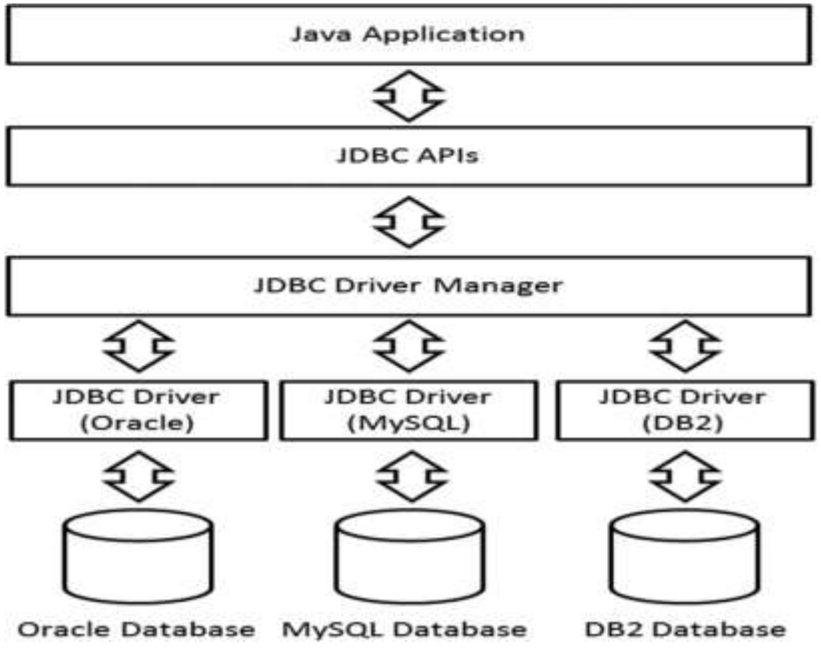
• The getConnection() method in the DriverManager class takes three arguments: a URL string, a username string, and a password string.

• The syntax of the URL (which needs to be specified to get the Connection object) is jdbc:<subprotocol>:<subname> .

• If the JDBC API is not able to locate the JDBC driver, it throws a SQLException. If jars for the drivers are available, they need to be included in the classpath to enable the JDBC API to locate the driver.

Building Database Applications with JDBC

Describe the interfaces that make up the core of the JDBC API including the Driver, Connection, Statement, and ResultSet interfaces and their relationship to provider implementations

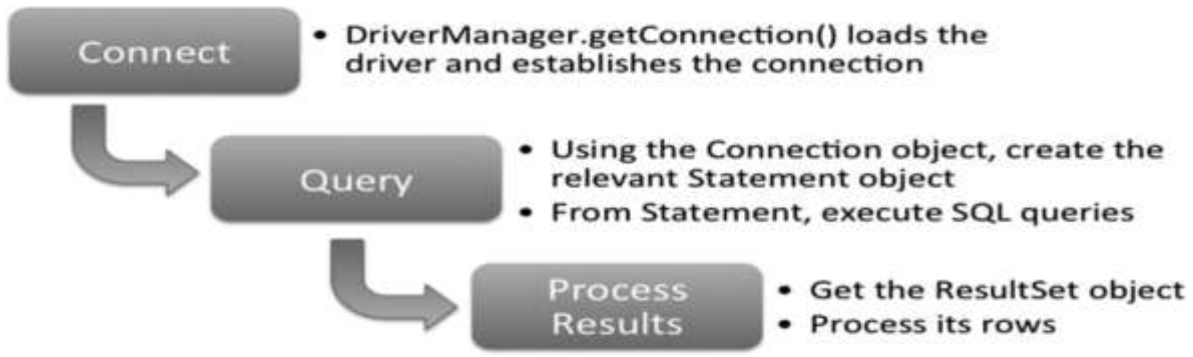


- The java.sql.Connection interface provides a channel through which the application and the database communicate.

- JDBC supports two classes for querying and updating: Statement and Resultset .
- A Statement is a SQL statement that can be used to communicate a SQL statement to the connected database and receive results from the database. There are three types of Statement s:
    - Statement : Sends a SQL statement to the database without any parameters
    - PreparedStatement : Represents a precompiled SQL statement that can be customized using IN parameters
    - CallableStatement : Executes stored procedures; can handle IN as well as OUT and INOUT parameters

- A resultset is a table with column heading and associated values requested by the query.

Building Database Applications with JDBC


Connect
DriverManager.getConnection() loads the driver and establishes the connection
Query
Using the Connection object, create the relevant Statement object
From Statement, execute SQL queries
Process Results
Get the ResultSet object
Process its rows

Submit queries and read results from the database including creating statements, returning result sets, iterating through the results, and properly closing result sets, statements, and connections

• A ResultSet object maintains a cursor pointing to the current row. Initially, the cursor is set to just before the first row; calling the next() method advances the cursor position by one row.

• The column index in the ResultSet object starts from 1 (*not* from 0).

• You need to call updateRow() after modifying the row contents in a resultset; otherwise, changes made to the ResultSet object are lost.

• You can use a try-with-resources statement to close resources (Connection , ResultSet , and Statement) automatically.

Localization

Certification Objectives
Read and set the locale by using the Locale object
Create and read a Properties file
Build a resource bundle for each locale and load a resource bundle in an application

Read and set the locale by using the Locale object

• A *locale* represents a language, culture, or country; the Locale class in Java provides an abstraction for this concept.

• Each locale can have three entries: the language, country, and variant. You can use standard codes available for language and country to form locale tags. There are no standard tags for variants; you can provide variant strings based on your need.

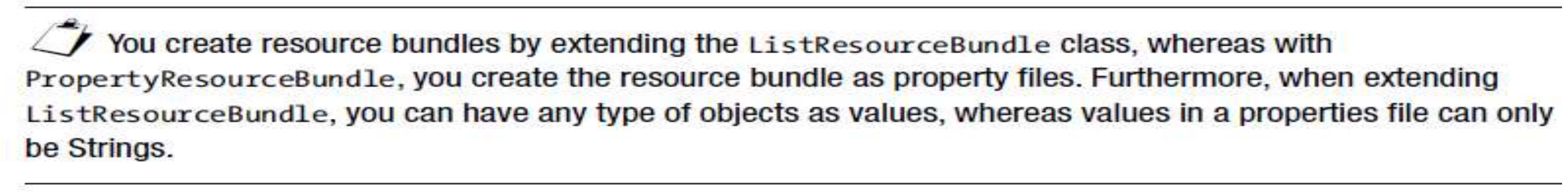
• The getter methods in the Locale class—such as getLanguage(), getCountry(), and getVariant() — return *codes*; whereas the similar methods of getDisplayCountry(), getDisplayLanguage(), and getDisplayVariant() return *names* .

• The getDefault() method in Locale returns the default locale set in the JVM. You can change this default locale to another locale by using the setDefault() method.

• There are many ways to create or get a Locale object corresponding to a locale:

- Use the constructor of the Locale class.
- Use the forLanguageTag(String languageTag) method in the Locale class.
- Build a Locale object by instantiating Locale.Builder and then call setLanguageTag() from that object.
- Use the predefined static final constants for locales in the Locale class.

Localization


You create resource bundles by extending the ListResourceBundle class, whereas with PropertyResourceBundle, you create the resource bundle as property files. Furthermore, when extending ListResourceBundle, you can have any type of objects as values, whereas values in a properties file can only be Strings.

Create and read a Properties file

• A resource bundle is a set of classes or property files that help define a set of keys and map those keys to locale-specific values.

• The class ResourceBundle has two derived classes: PropertyResourceBundle and ListResourceBundle . You can use ResourceBundle.getBundle() to get the bundle for a given locale.

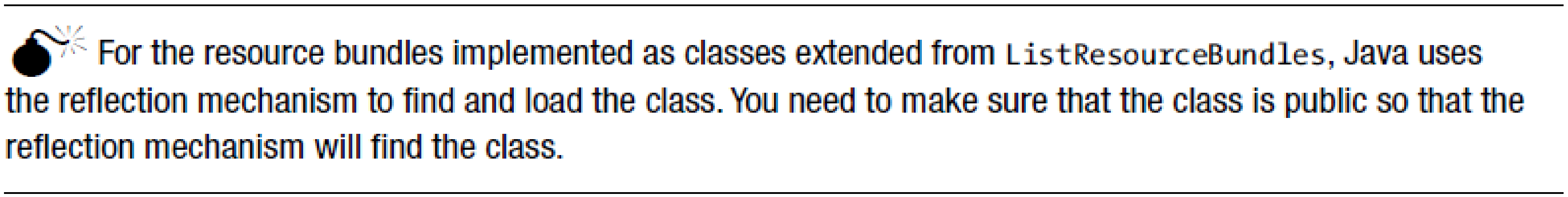
• The PropertyResourceBundle class provides support for multiple locales in the form of property files. For each locale, you specify the keys and values in a property file for that locale. You can use only String s as keys and values.

• To add support for a new locale, you can extend the ListResourceBundle class. In this derived class, you have to override the Object [][] getContents() method. The returned array must have the list of keys and values. The keys must be Strings, and values can be any objects.

• When passing the key string to the getObject() method to fetch the matching value in the resource bundle, make sure that the passed keys and the key in the resource bundle exactly match (the keyname is case sensitive). If they don't match, you'll get a MissingResourceException .

• The naming convention for a fully qualified resource bundle name is packagequalifier.bundlename + "_" + language + "_" + country + "_" + (variant + "_#" | "#") + script + "-" + extensions .

Localization


For the resource bundles implemented as classes extended from `ListResourceBundles`, Java uses the reflection mechanism to find and load the class. You need to make sure that the class is public so that the reflection mechanism will find the class.

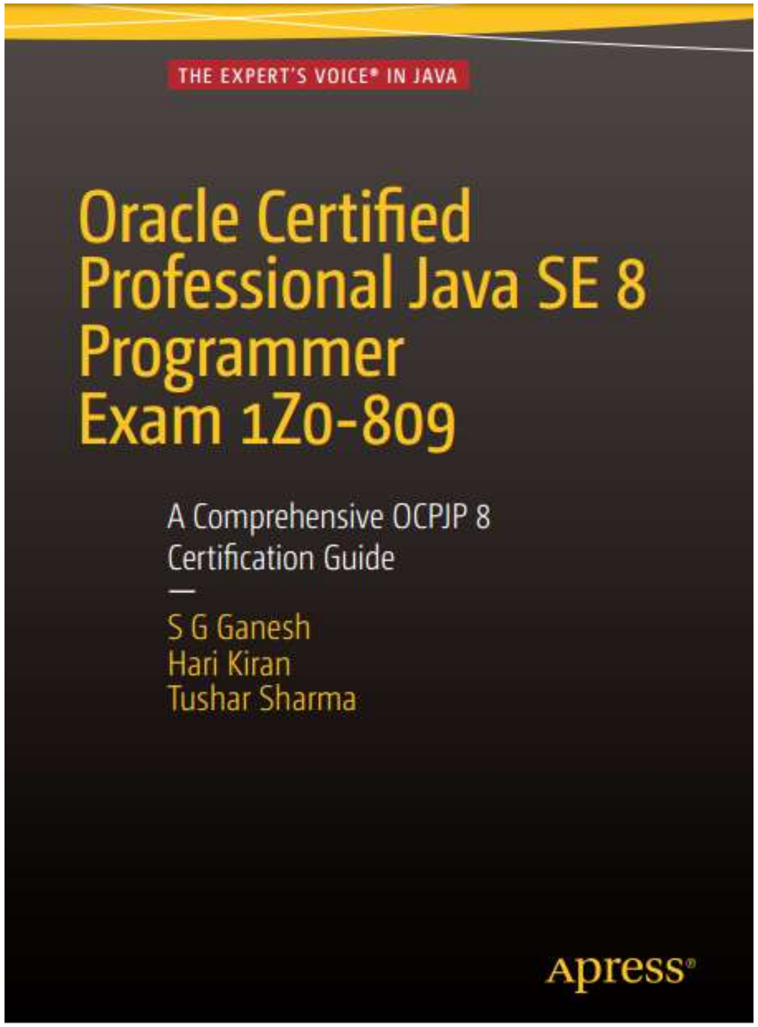
Build a resource bundle for each locale and load a resource bundle in an application

• The process of finding a matching resource bundle is same for classes extended from ListResourceBundles as for property files defined for PropertyResourceBundles.

• Here is the search sequence to look for a matching resource bundle. Search starts from Step 1. If at any step the search finds a match, the resource bundle is loaded. Otherwise, the search proceeds to the next step.

- **Step 1:** The search starts by looking for an exact match for the resource bundle with the full name.
- **Step 2:** The last component (the part separated by _) is dropped and the search is repeated with the resulting shorter name. *This process is repeated till the last locale modifier is left*.
- **Step 3:** The search is continued using the full name of the bundle for the default locale.
- **Step 4:** Search for the resource bundle with just the name of the bundle.
- **Step 5:** The search fails, throwing a MissingBundleException.

• The getBundle() method takes a ResourceBundle.Control object as an additional parameter. By extending this ResourceBundle.Control class and passing that object, you can control or customize the resource bundle searching and loading process.


THE EXPERT'S VOICE® IN JAVA

Oracle Certified Professional Java SE 8 Programmer Exam 1Z0-809

A Comprehensive OCPJP 8 Certification Guide

S G Ganesh
Hari Kiran
Tushar Sharma

Apress®

- Check out our latest book for OCPJP 8 exam preparation
- http://amzn.to/1NNtho2
- www.apress.com/9781484218358 (download source code here)
- https://ocpjava.wordpress.com (more ocpjp 8 resources here)

http://facebook.com/ocpjava

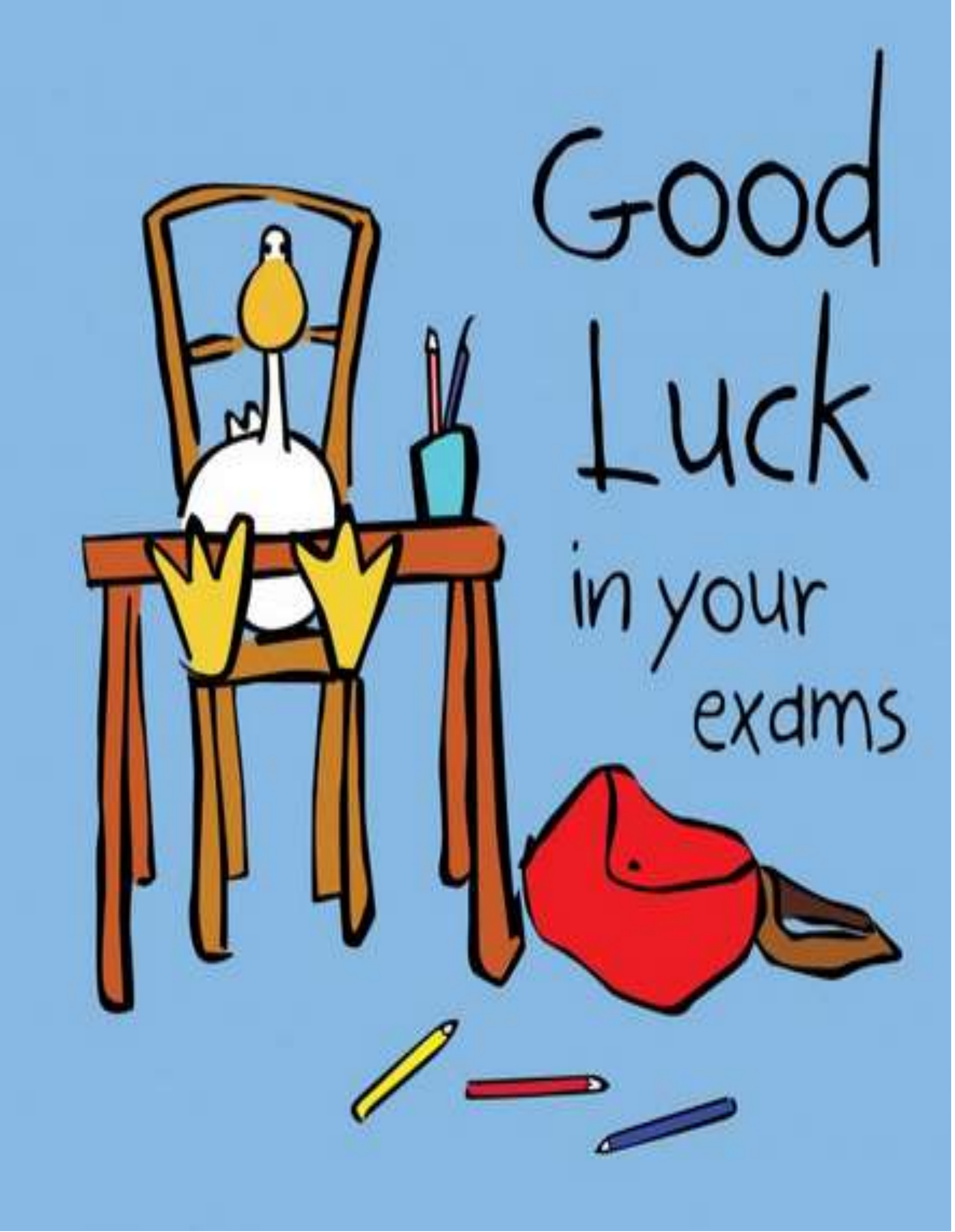


Image credits

https://getentrance-zealousinfomedia.netdna-ssl.com/wp-content/uploads/XAT-tips.png

http://www.the-platform.org.uk/wp-content/themes/Nuke/timthumb.php?src=http://www.the-platform.org.uk/wp-content/uploads/2013/05/exams1.gif&w=520&h=250&zc=1

https://nursessity.wordpress.com/2015/02/24/2-days-before-physical-health-assessment-validation/taking-test-clipart-ltkk4oata/

http://2.bp.blogspot.com/-nG2jD5Z2McY/UsjJhooLk8I/AAAAAAAAA7Q/-nNPi6SU_as/s1600/BouncingDuke-small.gif