Java Certification Key Facts

For Exam 1Z0-830
Java SE 21 Developer

UNDERSTANDING > KNOWLEDGE

DANCING CLOUD SERVICES, LLC

The Java 21 Exam Objective Groups

- Handling Date, Time, Text, Numeric and Boolean Values
- Controlling Program Flow
- Using Object-Oriented Concepts in Java
- Handling Exceptions
- Working with Arrays and Collections
- Working with Streams and Lambda expressions
- Packaging and deploying Java code
- Managing Concurrent Code Execution
- Using Java I/O API
- Implementing Localization
- Assume the following
- "Candidates are also expected to"

Handling Date, Time, Text, Numeric and Boolean Values

- Use primitives and wrapper classes. Evaluate arithmetic and boolean expressions, using the Math API and by applying precedence rules, type conversions, and casting.
- Manipulate text, including text blocks, using String and StringBuilder classes.
- Manipulate date, time, duration, period, instant and time-zone objects including daylight saving time using Date-Time API.

Use primitives and wrapper classes [...] type conversions, and casting.

- Primitive types are `boolean`, `byte`, `short`, `char`, `int`, `long`, `float`, and `double`
- Numeric literals have an implicit type
  - Just numbers/underscores -> `int`
  - Followed by L or l -> `long`
  - Followed by F or f -> `float`
  - Followed by D or d -> `double`
  - With decimal point or exponent notation -> `double`
- Numeric literals can have underscores between any two digits
  - Literals starting with `0x` / `0X` are base 16, those starting `0b` / `0B` are binary
  - Literals starting with `0` and having only digits and underscores are in base 8
- Widening promotions are provided automatically, narrowing ones are generally prohibited unless cast
  - Cast by prefixing an expression with the target type, e.g.: `int x = (int)getDoubleValue();`
  - Initialization of small integral primitive types is permitted from *constant expressions* of type int that "fit"
- Arithmetic is performed in the larger of two operand types, but at least `int`; `char` + `char` produces an `int`
- Wrappers exist for primitives: `Boolean`, `Byte`, `Short`, `Character`, `Integer`, `Long`, `Float`, and `Double`
  - These provide a home for utility methods including parsing / formatting and more
- Conversions between wrappers and their primitives can be automatic, but *never change fundamental type*

Evaluate arithmetic [...] expressions, [...] applying precedence rules

- Numeric operators are, in precedence order:
  - postfix `++`, `--` These expressions mutate an l-value, and have the original value of that l-value
  - prefix `++`, `--` These expressions mutate an l-value, and have the final value of the that l-value
  - unary `+`, unary `-`, bitwise negation `~`
  - multiplication `*`, division `/`, remainder `%` (this is not "mod")
  - addition `+`, subtraction `-`
  - shift `<<`, `>>`, `>>>`
  - comparison `<`, `>`, `<=`, `>=`
  - equality `==`, `!=`
  - bit manipulation (treat boolean as one bit): and `&`, exclusive or `^`, or `|` (precedence left to right)
  - assignment operators `=`, `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `^=`, `|=`, `<<=`, `>>=`, `>>>=`
    - Assignment operators are *expressions* that also perform an assignment, and take the value that was assigned

Evaluate [...] boolean expressions [...].

- If a `boolean` type is required, only a `boolean` or `Boolean` is acceptable, Java does not have "truthy/falsy"
- Operators `&`, `^`, `|`, accept boolean or bitwise (numeric) operands
- Operators `!`, `&&`, `||` accept only boolean operands
- `&&`, `||` are short circuit operators — if the first argument defines the result, the second is not evaluated
  - `true ||` anything -> `true`
  - `false &&` anything -> `false`
- Integer numbers in Java are two's-complement binary
  - Negative numbers have their most significant bit set, and -1 will be "all one bits" (`0xFFFFFFFF`, or `0b1111_1111_1111_1111_1111_1111_1111_1111`)

UNDERSTANDING > KNOWLEDGE

DANCING CLOUD SERVICES, LLC

[...] using the Math API [...]

- Class `java.lang.Math` provides a host of utility functions
- Arithmetic utilities, powers, roots, exponentiation, ceiling, floor, variants of arithmetic that don't silently over/under-flow,
- Trigonometric functions
- and more...

Manipulate text [...]

- Text is generally 16 bit UTF characters, the exam does not (seem to) concern itself with "code points"
- The most basic form of text is the `String`
- String has a literal format using double quote marks: `"Hello"`
- Single characters are represented as char type, which is a 16 bit unsigned numeric value
- char literals use single quotes: `'X'`
- Single vs double quotes are *not* interchangeable
- Escape sequences exist for some non-keyboard characters `'\n'` – newline `'\t'` tab, these can be used in string or char literals

[...] including text blocks [...]

String literals have a multi-line form known as a text block, these:

- are introduced by three sequential double-quotes followed by a newline (only whitespace newline may follow the triple-quote, and the newline does not form part of the result)
- contain embedded newlines where the original text has newlines
- are not "raw", they still process escape sequences
- leading spaces common across all lines are removed (to allow for better formatting)
- the `indent(int x)` method of `String` allows inserting x spaces at the start of every line (it also normalizes line endings)
- trailing whitespace on any one line is removed
- line endings are normalized to `'\n'` (code 10)

[...] using String and StringBuilder classes.

- `String` objects are immutable
  - To "change" a string, a new one must be created
  - The reference may be reassigned to refer to the new object if desired
- `StringBuilder` is a mutable type
  - has no literal form, and is *not assignment compatible* with `String`
- `String` and `StringBuilder` have a common parent interface `CharSequence`
- Any value can be concatenated with a `String` using the `+` operator
  - the second operand will be converted to text
  - if a reference type, its `public String toString()` method will be used
  - if no type-specific `toString` is found that of a parent, or ultimately `java.lang.Object`, will be used by polymorphism
  - if a primitive is converted, a simple/natural conversion will occur to a decimal, perhaps "scientific" form
- Comparison of `String` can be performed reliably using:
  - `public boolean equals(Object other)` Note the argument is Object, not String
  - `public int compareTo(String other)` Gives lexical order
  - `public int compareToIgnoreCase(String other)`
  - `==` tests two references for equivalence
- `StringBuilder` *does not implement* a useful equals method
  - but *does* provide `compareTo`

[...] using String and StringBuilder classes.

- `String` literals are subject to constant pooling
  - Any two identical string literals in source code will share the same object in memory
  - This applies *even for separately compiled code* as both the compiler and the class loading process look for duplicates
- Example:

```
String s1 = "Hello";
String s2 = "Hello";
String s3 = "He";
String s4 = s3 + "llo";
System.out.println("s1 == s2? " + (s1 == s2));
System.out.println("s1 == s4? " + (s1 == s4));
```

- Produces:

```
s1 == s2? true
s1 == s4? false
```

Handling Date, Time, Text, Numeric and Boolean Values

The `java.time` api:

- defines exclusively *immutable data types* (except for the exceptions), results of operations must be assigned
- provides types for local date, time, and date-time (no timezone information) and zoned-date-time
- accessor methods based on the data type, e.g. `getDayOfMonth`
- "mutator" methods (`withXxx` rather than `set`) based on type, e.g. `withDayOfMonth` these create new objects
- generalized access methods which throw exceptions if requesting fields not relevant to the type, e.g. `localDate.with(ChronoField.HOUR_OF_DAY, 12)`
- supports comparisons and ordering
- supports equality for *same type*, but `ZonedDateTime` objects representing the same instant in different timezones are not considered equal
- `Instant` represents a moment in the history of the universe, not a human calendar date/time
- `Duration` represents an offset in increments of nanoseconds (separate fields for seconds and nanoseconds as long)
- `Period` represents an offset in years, months, and days, but *cannot be converted to duration* (since 1 month might be 28, 29, 30, or 31 days, similar concern with leap years).
    - Add or subtract these from a starting date or datetime

Controlling Program Flow

- Create program flow control constructs including if/else, switch statements and expressions, loops, and break and continue statements.

[...] flow control [...] break and continue

- Any statement can be labeled, but labels are generally only useful with loops and switch constructs
- A label is a valid identifier followed by a colon that precedes a statement
  - the statement will be a block for this to be useful
  - the scope of the label is the statement (block), therefore labels can be reused, but not nested
- If a `break` statement is issued that refers to an in-scope label (which implies the `break` is in a labelled block!) then control is transferred to the first statement after that block
- A `continue` statement can only be issued from inside a labelled loop, and in that case:
  - control is transferred to the continuation behavior of the loop
  - if the `continue` was executed from inside a nested loop, or a switch, it breaks out of that nested structure
- Use of `break` or `continue` can reduce readability—take care in exam questions not to get confused.

[...] if/else [...] statements

The `if/else` structure provides simple flow control based on a boolean condition

- if requires a parenthesized `boolean` or `Boolean` test expression
- `if` and `else` take a single subordinate clause each
  - a block allows for multiple subordinate statements and is usual for readability
- `else` binds to the closest `if`
  - *indentation is not significant* to syntax
  - read very carefully if blocks are not used
- `if` can take a compile-time constant expression
  - As a special case, Java permits unreachable code in conditions to provide conditional compilation

[...] switch statements and expressions [...]

- There are several permutations of switch construct:
  - colon and arrow forms
  - pattern and non-pattern forms
  - statement and expression forms
- A switch has the general form

```
switch (switch-expression) {
     case elements…
     [default element]
}
```

UNDERSTANDING > KNOWLEDGE

DANCING CLOUD SERVICES, LLC

[...] switch statements and expressions [...]

- In all switch constructs:
  - A *switch expression* must be in parentheses
  - A *switch block* follows, which contains `case` elements and at most one `default` element
  - Cases may be either all arrow form, or all colon form, but these forms must not be mixed
  - At most one case target, or default, will be executed
  - No two case constants can be the same, neither `case null` nor `default` can be duplicated
  - Constant types for cases must be assignable to the switch expression type
    - For constants that are primitives, boxed values, and String, the switch expression must also be one of those
  - Each case presents either one or more constants, or a pattern
  - Unreachable cases based on constants are permitted, providing conditional compilation
    - But impossible *types* and *dominated patterns* are not permitted
- In the expression form:
  - A switch takes on the expression form when used in a syntax location that requires an expression, e.g. right side of an assignment, parameter list of a method call, or after return
  - The expression form must produce a value, or throw an exception for all possible switch-expression values, this often requires the use of `default`, but this can be moderated by switching on sealed type hierarchies
  - The expression type is resolved in the same way as for ternary expressions; approximately the nearest common parent type, combined with the aggregate of all common interfaces

[...] switch statements and expressions [...]

- Colon form
  - Cases fall through unless `break` is used
  - `break` is mandatory before a pattern
  - Alternation can be represented using multiple `case` *`xxx`*`:` in sequence without `break`
  - Alternation can be represented using comma separated constants
  - `default` picks up situations that don't match any case and makes the condition set complete
  - `default` does not have to be the last item in the series of cases
- Arrow form
  - Alternation *requires* a comma separated list of constants
  - Cases do not fall through
  - The right side of the arrow allows an expression, or throw followed by a semicolon, or a block
- Non-pattern forms
  - The switch expression must be one of `byte`, `short`, `char`, `int`, `Byte`, `Short`, `Character`, `Integer`, `String`, or an enum type
  - If the switch expression is an enum type the case constants can be enum values, without their class name
  - Alternation can be represented using a comma separated list of constants
  - incomplete condition sets are permitted

[...] switch statements and expressions [...]

- Pattern forms
  - The switch expression must be a reference type
  - If any pattern is used, all cases must be patterns, enum values, null or default
    - Except in the unique case of switching on a String type, in which case patterns may also include String literals
  - Patterns declare a capturing variable that is initialized if the basic match succeeds
  - If a basic match succeeds, a guard clause with "when" may further refine the match with a boolean expression
  - A guard may refer to in-scope variables, but local variables must be effectively final
  - A guard expression must not be a constant expression with the value false
  - Unguarded patterns "dominate" guarded ones if the are of a type that subsumes the guarded type
  - Dominating patterns must be listed after those they dominate
  - Equivalent *unguarded* patterns cannot coexist
  - `case null` is a pattern but does not dominate other patterns
  - `case null, default` is valid (the order matters)
  - `default` and `case null, default` dominate everything and must come last if either is used
  - `case Object obj ->` is equivalent to `default`

[...] switch statements and expressions [...]

- Patterns (but not guards) follow the form specified for `instanceof`, discussed fully later, this is an example that extracts values from a record:

```
record Customer(String name, int credit) {}
// …
Object obj = new Customer("Inaya", 10_000);
switch (obj) {
  case Customer(String name, int cl) when cl > 5_000 ->
    System.out.println("Welcome " + name);
  default -> System.out.println("what do you want?");
}
```

- Note that the destructuring calls the accessor method to get the component values, so if the returned value differs from the initializing value, switch will provide the returned value

[...] loops [...] – the while loop

- Java provides four explicit loop structures:
  - `while`
  - `do` / `while`
  - c-style `for`
  - enhanced `for`

- The most fundamental loop is the `while` loop
- This is controlled by a `boolean`/`Boolean` expression—Remember, Java does not have "truthy/falsy"
- The control expression must be in parentheses
- A "single subordinate statement" follows, but this is usually a block
- The block normally repeats for as long as the control expression evaluates true
  - `break`, `continue`, `return`, or a thrown exception can modify this
- The loop body will execute zero times if the control expression is immediately false
- If the control expression is a constant expression with the value false, a compiler error is issued for the unreachable body

[...] loops [...] – the do/while loop

- The `do while` loop test is placed in parentheses after the while keyword immediately after the block
- A semicolon is required after the closing parenthesis of the test
- The loop body executes at least once
- Iteration is controlled by a boolean expression

[…] loops […] – the c-style for loop

- The c-style for loop combines three common elements of a classic while loop

  `for` (<initialization> `;` <test> `;` <pre-restart>) *subordinate statement/block*

- The test part works exactly like a `while` loop
  - If the test is omitted, it behaves as a constant true value
- The initialization part may be either:
  - Declaration / initialization of one or more variables of the same base type
  - Zero or more comma separated *expression statements* (aka *statement expressions*)
- The pre-restart part is zero or more, comma separated, expression statements
- Variables declared in the initialization part have scope from the point of declaration to the end of the subordinate statement/block

Expression statements are:
- Assignment
- Increment / Decrement (++/−−)
- Method Invocation
- Object Creation (new)

[...] loops [...] – the enhanced for loop

- The enhanced `for` loop iterates over:
  - An array
  - An `Iterable`
- A local variable is declared and this is used to hold the "current" element of the iteration
  - The scope is local to the loop body
  - The variable is not implicitly final but might be effectively final
  - The variable may be explicitly marked `final`, even though it might appear to be overwritten with each iteration
- Order of iteration is governed by the target of the iteration
  - An array iterates in subscript order from zero up
  - An Iterable such as a `List` might iterate in a programmer-specified order, a `HashSet` will not

Using Object-Oriented Concepts in Java

- Declare and instantiate Java objects including nested class objects, and explain the object life-cycle including creation, reassigning references, and garbage collection.
- Create classes and records, and define and use instance and static fields and methods, constructors, and instance and static initializers.
- Implement overloaded methods, including var-arg methods.
- Understand variable scopes, apply encapsulation, and create immutable objects. Use local variable type inference.
- Implement inheritance, including abstract and sealed types as well as record classes. Override methods, including that of the Object class. Implement polymorphism and differentiate between object type and reference type. Perform reference type casting, identify object types using the instanceof operator, and pattern matching with the instanceof operator and the switch construct.
- Create and use interfaces, identify functional interfaces, and utilize private, static, and default interface methods.
- Create and use enum types with fields, methods, and constructors.

[...] instantiate Java objects [...] including nested class objects [...]

- Normal object creation (instantiation) is triggered by invoking the `new` keyword, followed by the name of a concrete class, and a parameter list.
  - This might be hidden inside a method (e.g. a factory)
- Instance inner classes
  - In addition to mutual access privileges, instance inner classes have an implicit reference to an instance of the enclosing class
  - This is passed, implicitly, to the constructor—default constructors for instance inner classes have an implicit argument that carries this; the implicit argument may be made explicit to allow for annotations if desired
  - Instance inner classes cannot be instantiated unless an outer instance is provided as the invocation target for the new operation. This may be the implicit `this` of the outer type, or an explicit outer object reference, it cannot be null.
  - The outer reference may be referred to directly from within the nested object as `OuterType.this`
- Static nested classes
  - Have no special requirements for instantiation, but have a name that includes, implicitly or explicitly, the name of the outer class
  - If the context is inside the outer class, this may be elided, otherwise the class name must be given longhand e.g.:

```
new Outer.Inner();
```

[...] object life-cycle including creation [...]

- If the class is already fully loaded and initialized, the process of instantiation follows a number of steps:
  - Allocation and zeroing of memory for all the members of the object (including all the parent type members)
  - The instance and the explicit parameters are passed to the matching constructor (constructors permit overloading)
  - The empty instance is received as the implicit `this` in the constructor
  - The constructor starts with either delegation to another overloaded constructor of this class (a call to `this(...)`) or delegation to a parent constructor (a call to `super(...)`) If no code is explicit, `super()` is implied
  - Execution reaches the constructor of `Object`, and upon completion, this returns to a more specific constructor. This "returning" process ensures that all elements of an object that has multiple levels of parent class are initialized from the parent class elements to the child class elements—this is consistent with the perspective that parent classes are foundations for the child class.
  - All paths through all overloaded constructors (using `this(...)`) must eventually reach a `super(...)` call or the class fails to compile. This implies that cycles in `this(...)` invocations cause the class to fail to compile.
  - Immediately upon return from `super(...)` the class undergoes instance initialization, and then control returns to the body of this constructor.
  - On completion of this constructor, control either returns to a delegating constructor (a `this(...)` or `super(...)` call returns) or the `new` invocation is completed and the object is returned to the original invoking code

[...] object life-cycle including creation [...]

- After return from `super(...)` and before the execution of the rest of a constructor body instance initialization occurs
- All instance fields that have initializing assignments are initialized and all instance initializer blocks are executed. These are executed in order from top to bottom of the class source code.

[...] reassigning references, and garbage collection.

- Variables of object type (anything except one of the 8 primitives) are references (a limited form of pointer) to an object, not the data itself
- This means two variables can refer to the same object (aliasing)
- Unless marked final, a reference can be reassigned to refer to another object
  - A consequence of this is that marking a variable final does not make the object immutable, it simply prevents reassignment of the variable to refer to another object. If, but only if, the object is also immutable then we have a constant value, but not that a "constant expression" is a very specific term and is not implied by a final reference to an immutable object
- If, but only if, an object is *not* reachable, then it is eligible for garbage collection
  - but eligibility does not imply that GC will necessarily occur at any given moment, or even before the program quits
- Reachability is determined starting from all reference variables in all the live stack frames of all live user threads, plus all static fields of all loaded classes, and the constant pool (which maintains objects representing String literals)
  - All these references are followed, transitively, and any object that can be found this way is reachable, any other objects are unreachable

Create classes [...]

- Classes (and all types) are declared in a source file which starts with a package declaration
  - The types declared in the source file are all members of that package
- Classes can be:
  - Top level in a source file
  - A static class inside another type (known as a nested class)
  - An instance class inside another type
  - A local class in a method
  - An anonymous class (static or instance by context)
- Top level classes:
  - Can be public or unlabelled. Unlabelled is package access
  - A public class must be in a source file with a name that matches the class name, so only one public class can be in any one file
  - If *no* constructor appears in the source code, a default constructor will be supplied by the compiler. This will have the same accessibility as the class, takes zero arguments, and only calls `super()`

Create classes [...]

- Member types
  - The primary effect of nesting types inside other types is that all these types—the enclosing type, all the enclosed types–share full access to all private elements of all the others
  - Static member types are really only related by this access privilege
- Static nested types
  - Nested interfaces (including annotations), enums, and records, are automatically static
  - Anything nested in an interface is also implicitly static
  - Nested records and enums are always static
- Nested, inner, and local, types are declared inside a class, or method, context
  - Nested and inner types may be public, default access, protected, or private
  - Objects of private types may be used outside their accessible scope as implementations of non-private interfaces or parent types
  - Local types are entirely local, but again, might be usable outside their defining method by means of an interface or parent type

[...] and records [...]

Records:

- Define classes with some special limitations mostly intended to help the record behave as immutable (or as nearly so as Java can easily support)
- Are implicitly `final` and prohibited from using the `extends` keyword
  - The parent type is always `java.lang.Record`
- Can implement interfaces, and form leaf nodes in sealed type hierarchies
- Benefit from automatic code generation that simplify common scenarios
- Gets `toString`, `equals`, and `hashCode` methods based on the components
- Can have static fields, and static initializers
- Cannot declare instance fields nor instance initializers
- Have "components" which are declared as part of the "canonical constructor" declaration, which occurs on the first line:

  ```
  record Customer(String name, int creditLimit) {}
  ```
- Each component causes creation of a `private final` field of the same name and an accessor method
  - No mutator is created

[...] and records [...]

- An accessor method created for a record:
  - Does not have the word "get" in its name—the name is the same as the component (and the underlying final field)
  - Takes no arguments
  - Is public
  - Does not throw checked exceptions
  - Has a return type matching that of the component, except in the case of vararg components, in which case it's an array with that base type
- The user can explicitly code an accessor provided that the code conforms to the rules above
  - Such a method can be annotated `@Override`
  - The user provided code prevents the automatic generation, so there is no super-dot version of the accessor method
- A record can have both instance and static methods
- A record's members can have any accessibility, although protected will be pointless since the type is final

[...] and records [...]

Initialization of records

- In the absence of any construction code written by the user, the *canonical* constructor will be created by the compiler
  - This has a formal parameter list that matches the component declaration on the record's name
  - The body of this constructor simply initializes the private final fields with the same names as the components using the component values
- The user is permitted to write the canonical constructor in the source code if desired, but in this case:
  - The formal parameter list must *exactly match* the component list (including the identifiers used as their names!)
  - The private final fields must be assigned exactly once, regardless of any conditional paths through the code
- Alternatively the user may create a *compact* constructor which:
  - Takes no formal parameter list in the source code, but the component list is implicitly the formal parameter list for this
  - Must not initialize the private final fields that represent the components
  - Is used by the compiler to create the actual canonical constructor by appending code that performs assignments from the formal parameters to the private final fields that represent the components
- Additional, overloaded, constructors may be created, but they must all start with delegation to another constructor with `this(...)`, and the delegation must reliably end at the canonical constructor

[...] instance and static fields [...]

- A class declaration can contain elements associated with the class as a whole, which are labelled `static`, or associated with a specific instance of the class, which are not labelled and are called "instance" elements
- These elements can be other types (nested types), fields, and methods
- Instance elements require the context of a particular instance to use, this can come from an explicit object reference prefix, or an implicit one
- The implicit reference for an instance method must be `this`, which refers to the context object on which the method was invoked
- The implicit reference for a static method will be the class in which the method is defined
    - This means that for a static method to refer to an instance feature (e.g. a field), an explicit prefix object must be provided
    - Note that there is *no prohibition* against static methods accessing instance fields, the only restriction is that there is no `this` context object, so an explicit object is required

[...] methods [...]

- A method's declaration consistently has several elements:
  - Indication whether the method is static or instance
  - Accessibility
  - A return type
  - The (base) name
  - A formal parameter list with explicit parameter types
  - An indication of any checked exceptions that might arise from the method
  - The method body
- The method's full identity comprises package name, containing type name, base method name, and the type sequence of the formal parameter list (excluding generic parameters of those types)
- The method must complete either by returning a value assignment compatible to the specified return type (unless the return type is void), or throwing an exception, for all paths through that body

[...] methods [...]

- To invoke a method generally requires the target method to be unambiguously identified at compile time. This might be achieved using the "full name", (package, class, method base name and the types of the actual parameters), but is more normally by some shortened approach
  - Selection among overriding methods is made at runtime
- An instance method is invoked with a *prefix object* (a.k.a. the *invocation target*), the method name, and the actual parameter list.
  - The prefix object must be of the type that contains, or inherits, the target method
  - If the invocation is made in the scope of an instance method that was itself invoked on the intended prefix object, then the prefix `this` may be used either explicitly, or more implicitly. If no suitable `this` object is in scope, compilation fails.
- A static method is identified entirely by the compile time package, type, base method name, and argument type sequence, however:
  - If the invocation is made from the same package, or an imported package, the package can be omitted
  - If the invocation is made from the same type the type can be omitted
  - If the method is declared in a single type and invocation is made from a subtype, then the type can be omitted
  - If the method is a `default` method, and is also declared in another interface applicable to the current type, but not implemented in the current type, then the declaring interface's name must be used as a prefix

[...] constructors [...]

- The form of a constructor is broadly similar to an instance method but:
  - Requires that the name exactly match the class name
  - No return type is specified—this is critical, it's valid to create a method with the same name as the class, and anything that has a return type in the declaration will be a method
  - Receives an implicit object of the class's type with the formal parameter name `this`—note that for a constructor it is not permitted to make this receiver parameter explicit
  - The body of the constructor can include return statements, but the statement must not be followed by any expression (the constructor mutates the `this` parameter, it does not return anything)
  - Any exception that might be thrown by the superclass constructor is possible for a child type constructor
- The first statement of a constructor might be a delegating call to another constructor in the same class, using the form `this(...)` or to a parent class constructor using the form `super(...)`
  - *At most one* of these delegating statements can be used
  - If no delegation is explicit, then `super()` with no arguments is implied, which mandates that the parent class has a zero argument constructor
  - Actual parameters to `this` and `super` calls must not make reference to the object references `this` or `super` explicitly or implicitly
  - No code can precede any `this(...)` or `super(...)` delegation call

[...] constructors [...]

- If a class has no coded constructor, the compiler will create the default constructor (except for record types)
- This:
  - Has the same accessibility as the class (except for enums, which are always private constructors)
  - Takes zero arguments (aside from the implicit this of the class type)
  - Delegates to `super()` with no arguments

[...] instance and static initializers.

- A class has two phases of initialization code in addition to constructors, these are called static and instance initializers
- Static initialization code consists of any static initializers and initialization code that assigns values to static fields
- This code runs in the source code order from top to bottom of the class declaration after the class is loaded into the JVM, usually before any real use of the class is executed
- A static initializer is the keyword static followed immediately by a block, located at the top level of nesting inside a class: `class X { static { <code> } }`
- Instance initialization code consists of any instance initializers and initialization code that assigns values to instance fields
- This code runs in source code order from top to bottom of the class after the delegation to a superclass constructor returns.
- An instance initializer is an unadorned block located at the top level of nesting inside a class: `class X { { <code> } }`

Implement overloaded methods [...]

- Because methods' identities include the type sequence of the formal parameter list, it's permissible to have multiple methods in the same type that are otherwise identical
  - These are called overloaded methods
  - Remember that generic types are not considered in this; that is, doStuff(List<String> ls) and doStuff(List<Date> ld) are not valid overloads–this is an effect of "type erasure"
- Overloading of constructors is similarly permitted
- The compiler identifies the correct target method for an invocation at the point of invocation based on the actual parameters supplied
- Type promotion can occur during selection, and this can lead to ambiguities, which might cause compilation failure

Implement overloaded methods [...]

- Argument promotion/conversion during overloaded method resolution occurs in multiple distinct steps, with the possibility of compilation failure at each
- An exact parameter type match always wins
- Widening conversion of a single argument can take the nearest promotion of several
  - an int would promote to a long rather than to a float and no ambiguity is seen
- Widening conversion of one or another argument is ambiguous and causes immediate compilation failure
- If widening conversions are not found, boxing conversions will be tried next
- Boxing conversions *never change the basic numeric type*; that is, an `int` will only box to `Integer`
- However, boxing conversions *can also widen the type*, that is; `Integer` to `Number` (but *not* to `Long`)
- Ambiguous boxing conversions will cause immediate compilation failure
- If no boxing conversion is found, a variable argument list may be tried
- Ambiguity again causes compilation failure

[...] including var-arg methods.

- A method can accept a variable length argument list
  - As the last formal parameter (implying only one varargs parameter is possible)
  - This is represented with three dots between the base type and the formal parameter name, e.g.:
    ```
    void doStuff(LocalDate ld, String ... s) {}
    ```
  - The data arrives as an array of the base type
- Such a method may be invoked:
  - With no actual parameter, which equates to an array of zero length
  - With a comma separated list of elements of a type promotable/boxable to the base type, which becomes an array of the base type containing the promoted/boxed elements
  - With an array of the base type, which is passed unchanged
- Passing an array directly allows the calling code to mutate the elements after the call, and to see mutations that occur in the method (whether this is wise or not!)

Understand variable scopes [...]

- Local variables are generally in scope from the point of declaration to the end of the immediately enclosing curly braces
- Formal parameters (local variables declared in parentheses that are syntactically followed by a subordinate statement or block, such as in methods, for loops, try, and catch structures) have a scope that starts at declaration and ends with the end of the immediately following subordinate statement or block
- Fields are generally in scope throughout the type declaration that contains them, but must be qualified by an instance or type reference depending on whether they are instance or static fields
    - This might be the implicit this, or implicitly the enclosing type name, or a parent type name

[...] apply encapsulation [...]

- Encapsulation restricts access to data, enforcing interaction via methods
  - Those methods can check and enforce consistency rules on the data, avoiding, or at least announcing, semantic errors
- Encapsulation may be enforced using a variety of access controls. From least accessible to most, these are:
  - Method local — only visible inside the method that contains it
  - `private` — only accessible from inside the enclosing top-level curly braces surrounding it (note, nested types and the enclosing type all share *full access* to `private` elements declared in any of those related types)
  - package access (often, but not always, the effect achieved by omitting explicit access control on a type element)
  - `protected` — accessible anywhere within the same package and additionally by code in a subtype using a reference of that sub-type
  - `public` in a package *not exported* from a module — accessible anywhere within any type within that module
  - `public` in a package that is exported from a module — accessible anywhere within any type within that module or any module that "reads" (usually by means of the `requires` directive) that module
  - `public` in a JVM that is not running the module system — anywhere in that JVM

[...] apply encapsulation [...]

Effective encapsulation might use combinations of several techniques, but the most likely approach is:

- Mark all mutable fields as `private`
- Ensure that all constructors / factories can only produce objects in valid states
- Ensure that all mutation operations result in valid states
- Ensure that operation that updates state based on values provided from the "outside" (e.g. as function arguments) does not store that state unless either the type provided is a primitive, immutable (e.g. `String`, `LocalDateTime`), or a copy is taken
  - In particular, mutable collections, arrays in general, and variable length argument lists might be mutated by the caller after their use
- Ensure that access methods do not return references to mutable internal state; instead return only references to immutable data, an unmodifiable proxy (`Collections.unmodifiableList`, etc.), or a (deep) copy of that data
- Additionally, marking fields that should never change value as `final` can be helpful
  - Note; a `final` reference does make a target object immutable, it only prevents reassignment of the reference

[...] create immutable objects [...]

- Immutability is implemented very similarly to strong encapsulation, with the additions that *no mutation methods are provided*
- Fields in immutable objects should generally be marked `final`
- Prefer immutable types/implementations for members, e.g. `List.of`, `List.copyOf` etc.
- Immutable types might provide "withXxx" methods rather than "setXxx" methods; by convention these create a new object that represents the effect of a change on the original object (and, of course, the original object is unchanged by the operation)
    - Note that in truly immutable objects, member data can safely be shared with the newly created object

[...] Use local variable type inference.

- If a local variable (*not a field*) is initialized in the same statement that it is declared, the type may be inferred from the initializing expression if the declaration uses `var` in place of the type. E.g.:
  ```
  var count = 10;
  ```
  declares `count` to be of type `int`, and initial value 10.
  - Recall that a literal composed only of digits must be an `int` type
- The word var is a *contextual keyword* or pseudo-type; it has special meaning only when used in a place that syntax expects a type name. Elsewhere it's a legitimate identifier. E.g.:
  ```
  var var = "var";
  ```
  is valid, and declares a variable called `var`, with type `String`

[...] Use local variable type inference.

- When using `var` for type inference of a local variable:
  - The type is fixed at the declaration — this does not create "dynamic typing" in the manner of JavaScript
  - The declaration *and initialization* must be in a single statement
  - The initializing value must not be `null`
  - The word var stands for the *entire type*
    e.g. `var [] ia = new int[]{};` is not permitted, but `var ia = new int[]{};` is OK
  - The initializing value must have a fully known type; it cannot infer type from the left side, this is particularly relevant with array and lambda declarations
  - When used with an array, the type of the array must be made explicit
    e.g. `var [] ia = {1,2,3};` is not permitted, but `var ia = new int[]{1,2,3};` is OK
  - If a lambda is to be assigned, the type must be explicit:

```
var adder = (IntBinaryOperator)((a, b) -> a + b); // OK, if rather pointless!
var adder = (a, b) -> a + b; // FAILS
```

  - Only a single variable may be declared in one statement

[...] Use local variable type inference.

- Type inference of formal parameters is valid if the context fully defines the type, e.g.:
  - Variables declared in c-style or enhanced-for constructs
  - Resources declared in a try-with-resources construct
  - Often, but not always, lambda parameters
- But will not be valid in other situations:
  - Formal parameters to regular methods (i.e. non-lambdas)
  - The catch section of any try construct
- The use of `var` can provide a *non-denotable type* which can be the aggregate of a class type along with interfaces. e.g. given:

  ```
  var b = true ? "X" : 99;
  ```

  `b` has type `Object & Serializable & Comparable` (actually a bit more than that)
  - Note that 99 is autoboxed to Integer

UNDERSTANDING > KNOWLEDGE

DANCING CLOUD SERVICES, LLC

Implement inheritance [...]

- Inheritance describes one type acquiring features of another type by a simple syntax feature
- All classes (except `java.lang.Object`) inherit from *one* other class
- A class can inherit from any number of interfaces
- An interface can inherit from other interfaces
- Inheritance can give a type a field, a method, an abstract method, or the obligation to implement an abstract method
- Abstract methods constitute a guarantee that any object that claims to be of the type that declares the abstract method, must have implemented that method
    - Only concrete classes can be instantiated
    - Therefore, a concrete class is prohibited from having any abstract methods, so any abstract methods in the inheritance hierarchy become an obligation to implement that method
- When a type inherits from another type, it is also said to be "polymorphic", meaning it can be treated as an its own type, or as the type that was inherited. This is sometimes referred to as an "is-a" relationship
    - In particular, the object can be assigned to a reference of the inherited type, including being passed as an argument or return value of that type
    - All the methods and features of the inherited type are available on the inheriting type

Implement inheritance [...]

- Inheritance of a class can be prevented by marking the class `final`
- Enum and record types are generally implicitly final and may not be marked as such
  - An enum will be final unless it contains subtypes, but such subtypes can only be defined as members of the enum. In this example, `E1` is not final. If the value `X` had not been declared as a subtype, `E1` would have been final

```
enum E1 {
  X {}, // this is a subtype
  Y     // this is type E1
}
```

- Sealed type hierarchies also control / restrict inheritance

[...] including abstract [...]

- A class may be labeled `abstract`, an interface is implicitly abstract but the label is permitted
- An abstract type cannot be instantiated directly—a subtype will be, by polymorphism, of that type
- An abstract method has a signature but a semicolon in place of the body
- Abstract types are permitted to contain abstract methods (concrete classes are not)
  - In an abstract class, an abstract method must be labeled `abstract`
  - In an interface, an abstract method is implicitly so, and need not be labeled as such
- An abstract class cannot be marked `final`—such a class cannot be used except by subclassing, so marking it `final` would render it useless

[...] sealed types [...]

- Sealed type hierarchies control inheritance so that programmers (and the compiler) can know the types that might exist that will be assignment compatible with a given parent type
  - When behaviors need to be type specific unexpected types might introduce bugs—this feature can control that risk
- The root of a sealed type hierarchy is marked `sealed` and in the general case carries a `permits` clause that enumerates all the valid subtypes
  - The `permits` clause may be omitted if all the subtypes are in the same source file
  - Any type that is sealed *must* have subtypes
- All types in a sealed type hierarchy must be one of:
  - A record, an enum, or a final class
  - A sealed type
  - Marked `non-sealed`, or be a subtype of a non-sealed type
- If running under the module system, all types in a sealed hierarchy must be in the *same module*
- If not running under the module system, all types in a sealed hierarchy must be in the *same package*

[...] record classes [...]

- Addressed earlier

[...] Override methods [...]

- When a sub-type defines a method with the same name and argument type sequence (including generic types) as a method already declared in a parent type, we say this method *overrides* the parent's method
  - The override occurs because of the match of name and argument type sequence with the parent method
  - A simple typo can easy create an overloading method by mistake—to avoid this risk we can annotate the child method `@Override` — this does not cause the method to be an override, but raises an error in case of such an typo
- An overriding method must conform to the syntax requirements of "The Liskov Substitution Principle". In Java this requires:
  - The overriding method must not be less accessible than the method being overridden—Note that overridable interface methods are all public (whether marked as such or not). Therefore any implementation of an interface method must be marked `public`
  - The overriding method returns a "suitable" type. For primitive returns, this must be *identical*. For reference types, this must be *assignment compatible* with the type returned by the method being overridden
  - The overriding method must not declare throwing of any checked exceptions that would not be valid for the overridden method. It doesn't matter if the exceptions are thrown or not.

[...] Override methods [...]

- Overriding is only possible with *non-private*, *non-final*, *instance*, *methods*.
  - Note that no such behavior exists where fields with common names exist in parent and child types
- Code in a subtype can invoke the previous class implementation of an overridden method using the `super.xxx()` form
  - It is not possible to climb to a specific implementation higher up the class hierarchy
- Code that *directly* inherits a default method from an interface can delegate to that method using the form `IntTypeName.super.xxx()` (where `IntTypeName` is the immediate parent interface name)

[...] including that of the Object class [...]

- Several methods of the class java.lang.Object are commonly overloaded. Take care to note the signatures:
  - `public boolean equals(Object obj)`
  - `public int hashCode()`
  - `public String toString()`
  - `protected Object clone() throws CloneNotSupportedException`
- Notes:
  - The `equals` method takes an *`Object`* as its formal parameter, *not the current class type*, which this would be an *overload*, not an *override*, and would fail in use
  - The `clone` method is protected. It can therefore be overridden to be public, making it generally available for a user-defined type. It's also allowed to throw an exception if the type or the object does not want to be cloned. Note that the `Object.clone()` method does not throw any exception.

[...] including that of the Object class [...]

Semantics of valid `equals` and `hashCode` methods

- The notion of "equality" in objects is a bit more complex than the regular mathematical meaning, and might be better considered as "equivalence". However, the `equals` and `hashCode` methods are used in the collections API to determine, for example, whether a particular collection contains an object that is equivalent to one presented in the inquiry.
- Two objects should be considered equal/equivalent if the field values they contain are equivalent–but the programmer gets to decide which fields are relevant in this context.
- Two objects that are equal/equivalent *must* return the same hashCode value
- Two objects that are equal/equivalent might return the same hashCode value, but a good implementation will minimise the likelihood of such a "collision"

[...] including that of the Object class [...]

Things can get rather ugly with inheritance; if we test two objects, `a` and `b`, where `b` is of of a subtype of `a`, then the test `a.equals(b)`, uses the code defined in the class of `a`, but if we test `b.equals(a)` we use the code defined in the class of `b`. If the two methods provide different answers, we have a non-reflexive equality test, which is clearly bogus

- Avoid inheritance, marking the base class `final`
- Avoid overriding the `equals` method, marking the method `final`
- Code `equals` methods such that objects do not report equal unless they are exactly the *same type*, rather than merely assignment compatible
  - For this, use a test of the form `a.getClass() == b.getClass()` rather than `b instanceof ClassOfA`
  - Strictly, this approach "breaks" the Liskov substitution principle for these types

[...] Implement polymorphism and differentiate between object type and reference type [...]

- Polymorphism has several meanings in computer science. In this context we're concerned with the idea that an object may be used in situations that require other types if the required type is part of this object's class or interface hierarchy
- A key feature is that objects can respond to method invocations with different implementation code where a method is overridden, or is a different implementation of an interface method
  - This is the central behavior of overriding methods
- A reference, whether a variable, expression of reference type, method argument, or method return value, has a type that is known to the compiler.
  - This type determines the methods that the compiler will allow us to invoke
- The object that is referred to at runtime might have a different type (provided that the object also conforms to the reference type somewhere in its hierarchy)
  - For overridden methods (*but only overridden methods*), the behavior exhibited at runtime will be that associated with the actual type of the object, not that of the type of the reference
  - Recall the restriction on overriding; *it only occurs for non-private, non-final, instance, methods*.

[...] Perform reference type casting [...]

- If a reference expression of a generalized (super) type actually refers to an object of a specialized (sub) type, we can create a reference expression of that specialized type that refers to the same object using a *cast*
  - This allows access to features of the subtype that do not exist in the supertype, and access to behaviors that are not overridden
- If `aSuper` is a reference variable of type `Sup`, and type `Sub` is a subtype of `Sup`, this is the cast expression: `(Sub)aSuper`
  - If `aSuper` did not in fact refer to an object of type `Sub`, a `ClassCastException` is thrown
- The compiler rejects cast constructions that are logically impossible
- However, the compiler does not reject casts involving interfaces and non-final classes, since it might be possible to introduce new classes later that implement interfaces in a way that makes the proposal possible

[...] identify object types using the instanceof operator [...]

- Prior to casting, it's wise to ensure the cast will succeed. The instanceof operator allows determination of the runtime type of an object referred to by an expression, like this:

```
Parent p = new Child();
boolean isAChild = p instanceof Child;
```

  - Note that an instanceof expression has a boolean type
- If the compiler can prove that the expression will *always* be false (even in the face of new types added to the existing hierarchy) then the expression will fail to compile
  - Again, combinations of interface types and non-final classes will always be accepted
- In normal use, a test of this kind typically precedes a cast:

```
Parent p = new Child();
if (p instanceof Child) {
  Child aChild = (Child)p;
  // do something with aChild…
}
```

[...] pattern matching with [...] instanceof [...]

- Newer Java versions provide a single construction that performs both test and (conditionally) assignment to a new reference variable. The scope of the new variable is tightly controlled by the compiler to those parts of the code where it is *definitely assigned*

```
if (p instanceof Child aChild) {
  // aChild in scope
} else {
  // aChild not in scope
}
```

- The instanceof structure is still a boolean expression, so chaining with `&&` and `||` works (and the scope rules work correctly). E.g.:

```
Object obj = "Hello";
if (obj instanceof CharSequence cs && cs.length() > 2) {
  out.println("more than two characters: " + cs);
} else {
  out.println("very short text, or not text");
}
```

UNDERSTANDING > KNOWLEDGE

DANCING CLOUD SERVICES, LLC

[…] pattern matching with […] instanceof […]

- Pattern matching on record types can assign multiple variables to extract the component values instead of a single reference to the record object itself

```
record Customer(String name, int credit) {}
// …
Object obj = new Customer("Inaya", 10_000);
if (obj instanceof Customer(String name, int cl) && cl > 1000) {
  System.out.println("Welcome " + name);
}
```

- Nested patterns can be extracted:

```
Object obj = new Rectangle(new Point(10, 10), new Point(25, 5), "RED");
if (obj instanceof Rectangle(Point(int tlx, int tly), Point(int brx, int bry),
    String color)) {
  out.println(color + " rectangle w: " + (brx - tlx) + " h: " + (tly - bry));
}
```

[...] pattern matching with [...] instanceof [...]

- Generic type information is largely absent at runtime, specifically there is no information in, for example, `List<String>` and `List<LocalDate>` indicating the content type of the list
- Consequently, there is no information available at runtime that could allow instanceof to test for the generic type of such a list
- However, if there is sufficient information at compile time, then an instanceof expression can refer to generic types

[...] pattern matching with [...] instanceof [...]

- This succeeds, but is less helpful than it might be because the content type of the list is unknown, requiring a cast:

```
Object data = List.of("A", "B");
if (data instanceof List ls) {
  String st = (String)ls.get(0);
  System.out.println("First element is " + st);
}
```

- This fails, because the type of the list cannot be tested at runtime:

```
if (data instanceof List<String> ls) { // compilation fails
  String st = ls.get(0);
  System.out.println("First element is " + st);
}
```

[…] pattern matching with […] instanceof […]

- However, this succeeds because if data actually refers to a List, it's already established in the declaration of of the variable that the generic content type must be String, hence runtime testing of that is not required:

```
Collection<String> data = List.of("A", "B");
if (data instanceof List<String> ls) { // This is now OK
  String st = ls.get(0);
  System.out.println("First element is " + st);
}
```

- Although not particularly useful, these two forms also work (and the first has been permitted since Java 5!)

```
Object data = List.of("A", "B");
if (data instanceof List<?>) { … // historically, and still, valid
if (data instanceof List<?> ls) { …
```

[...] switch construct.

- Addressed earlier

Create and use interfaces [...]

- Interfaces allow us to define an abstract type (one that cannot be instantiated directly) in terms of `public abstract`, `public default`, `public static`, `private static`, `private static`, and `private` instance methods
  - No intermediate accessibilities are possible with interfaces
- Interfaces can also declare values, but these are implicitly public, static, and final, and cannot be otherwise

- Abstract methods
  - define the syntax required to interact with a capability without any specification of how that capability might be implemented
  - are instance methods
  - have no body, just a signature followed by a semicolon
  - in interfaces, are public and abstract by default, and need not carry either keyword

[...] identify functional interfaces [...]

- A functional interface is any interface that declares *exactly one* abstract method
  - This includes any methods "inherited" from parent interfaces
  - Methods of the `Object` class that have been redefined as abstract are excluded from this count
- Such an interface *may* optionally, be annotated with `@FunctionalInterface`
  - This annotation provides error checking—if an interface is so annotated, but has zero, or more than one, abstract method, the annotation raises a compilation error
  - The annotation is not involved in granting the status of functional interface
- Generics can allow any single interface method to describe variations of argument and/or return types
- However, the expressivity of generics has significant restrictions specifically:
  - Generics are inefficient in handling primitives—autoboxing/unboxing takes CPU power and memory
  - Generics can describe variations in types, but not the arity (number of formal parameters) of a method
  - Generics cannot unify methods that return values with methods that have void returns
- Because of these limitations, many different interfaces might be needed, so the core package `java.util.function`defines 43 functional interfaces, most are variations on a small number of themes.

[...] identify functional interfaces [...]

- The main categories of interface in the core APIs are:
  - `Consumer` — declares a function that receives a parameter but returns void
  - `Supplier` — declares a function that takes zero arguments but returns some data
  - `Function` — declares a function that takes an argument and returns a result
  - `Predicate` — declares a method that takes an argument and returns a primitive boolean
- Variations are provided for two-argument functions (excepting supplier), these usually use the prefix `Bi`
- Variations are provided for functions that accept a primitive argument, these use one of the prefixes `Int`, `Long`, `Double`, according to the type `int`, `long`, or `double`, of the primitive argument
- Variations are provided for functions that return a primitive result, these are generally one of the prefixes `ToInt`, `ToLong`, `ToDouble`
  - The variations of supplier do not use the word `To`, instead they are simply `IntSupplier` etc. because there's no argument, only a return, so there's no ambiguity
- Special cases are provided for one and two argument functions that take argument(s) and return types that are identical, these are called `UnaryOperator` and `BinaryOperator`, with primitive variations in the form `IntBinaryOperator`
- Special cases for consumers that take one object and one primitive are of the form `ObjIntConsumer`

[...] identify functional interfaces [...]

- Many functional interfaces (and other interfaces) provide default instance, and static, utility methods, some of these are factories for implementations of the interface, some others create compound actions
  - `Consumer`: `andThen`
  - `Function`: `identity`, `andThen`, `compose`
  - `Predicate`: `and`, `or`, `not`, `negate`, `isEqual`
  - `Comparator`: `comparing`, `naturalOrder`, `nullsFirst`, `nullsLast`, `reverseOrder`
  - `Map.Entry` (not a functional interface): `comparingByKey`, `comparingByValue`

[...] utilize private, static, and default interface methods.

- Private instance methods in interfaces
  - are marked `private`
  - can be static or instance
  - just like private methods in classes, these factor out reusable behavior that can be invoked from other methods inside the interface (including any nested types)
  - have a first formal parameter of the interface type called `this`, which is usually implicit (this is exactly like instance methods in classes)
  - instance private methods require an invocation target of the interface type; in an interface, only instance private methods and default methods have an implicit `this`, but explicit instances can be used for the invocation target
- Static methods in interfaces
  - can be public or private
  - if not marked with an accessibility, they will be public by default
  - just like static methods in classes, they provide methods that are grouped with the "concept" represented by the interface, but not specifically associated with any given instance

[...] utilize private, static, and default interface methods.

- Default methods
  - Provide methods that will likely be replaced in classes that implement this interface, but for maintenance reasons must have a fallback for existing classes that do not provide the method and cannot conveniently be updated
  - Applicable methods in classes will always supersede any default method, regardless of their relative position in the type hierarchy
  - If two equivalent methods are provided by interfaces, and any of them is a default method, then any class implementing both interfaces must define the method explicitly or compilation will fail (due to ambiguity)
  - Such a method may delegate to the default method of an immediate parent interface using the qualified super syntax:

```
interface I1 { default void doStuff() {} }
interface I2 { void doStuff(); } // 2 doStuff() methods are override equivalent
class C1 implements I1, I2 { // both interfaces are implemented
  // the doStuff() method must be explicit here
  // and can delegate to to the I1 version explicitly with this syntax
  public void doStuff() { I1.super.doStuff(); }
}
```

Create and use enum types with fields, methods [...]

- Enum types explicitly define all their instances in the source code, so are useful for representing things like days of the week (avoiding the type-ambiguity of using constant int or string values)
- Enum types are full-fledged classes; they can have instance and static fields, instance and static methods, and nested types.
- The declared instances are named, in a comma-separated list, immediately after the opening curly brace of the type's body. A semicolon follows, and then fields, methods, etc. can be declared.
  - If nothing other than instances are to be declared, it's possible, but unwise, to drop the semicolon that ends the list of instances
- Example:

```
enum DayOfWeek {
  MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY;
  private static String[] enFrancais = {
      "lundi", "mardi", "mercredi", "jeudi", "vendredi", "samedi", "dimanche"
  };
  public boolean isWeekend() { return this == SATURDAY || this == SUNDAY; }
  public String frenchName() { return enFrancais[this.ordinal()]; }
}
```

Create and use enum types with fields, methods [...]

Enum types have several built-in methods, for an enum type `TheEnum` these are:

- `public static TheEnum valueOf(String)` — takes a String and finds the existing enum of the type that has that name. Throws an IllegalArgumentException if no value matches that text
- `public static TheEnum[] values()` — returns an array of all the enum's values, in the order originally declared
- `public int ordinal()` — returns the zero-based index position of this enum's constant in the declaration list
- `public final String name()` — returns the exact text of the enum constant's declared name
- `public int compareTo(TheEnum other)` — returns an int with a sign indicating the relative position in the sequence of constant declarations
- `public String toString()` — by default, the same as `name()`, but can be overridden
- `equals`, `hashCode` — standard methods of `Object`, `equals` is the same as `==`, `hashCode` returns a different value for each constant

Create and use enum types with fields, methods [...]

- Enum types are "almost" final—they *can* have subtypes, but only declared as inner types within the enum
- If no subtypes are declared, the enum is actually final
- No enum can use the keyword `extends`, nor be the target of an `extends` keyword
- Subtype example—`ITEM2` is of an anonymous subtype of `Parent`, and overrides the `toString` method:

```
enum Parent {
  ITEM1,
  ITEM2 {
    @Override public String toString() { return "I'm " + super.toString(); }
  };
}
```

UNDERSTANDING > KNOWLEDGE

DANCING CLOUD SERVICES, LLC

[...] and constructors.

- Enums can have explicit, and overloaded, constructors
- All enum constructors must be private (and are private by default, so need not be marked as such)
- Constructors with arguments are called explicitly from the instance with the actual parameter list appended to the instance name
- Example—two constructors, one with zero arguments, one takes a single boolean:

```
enum DayOfWeek {
  MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,SATURDAY(true), SUNDAY(true);
  private boolean isWeekend = false;
  DayOfWeek() {} // used my MONDAY-FRIDAY
  DayOfWeek(boolean wEnd) { isWeekend = wEnd; } // used by SATURDAY & SUNDAY
}
```

- If no constructor is coded explicitly, a default that takes no arguments will be created, but if any other constructor is created, this will be removed and if needed must be re-coded explicitly

Handling Exceptions

- Handle exceptions using try/catch/finally, try-with-resources, and multi-catch blocks, including custom exceptions.

Handle exceptions using try/catch/finally [...]

- Any exception that is thrown causes control to jump to the end of the nearest surrounding try block, jumping out of methods and up the call stack if none is found in the current method
- If an exception reaches the top of the call stack for its thread, that thread is killed and the exception trace printed
  - [This behavior can be modified if the thread is part of a ThreadGroup that has an uncaught exception handler registered]
  - If this was the last non-daemon user-thread in the JVM, the JVM will shut down
- When the exception finds a try block the system looks for an associated catch block that can handle that type of exception
  - This determination is made on an assignment-compatibility basis, so `catch (IOException ioe) {}` might be used to handle a `FileNotFoundException` , which is a subtype of `IOException`
  - The order of catch blocks for class-hierarchy related exception types must be from most specific exceptions to most general, or compilation fails
  - If no suitable catch block is associated with this try block, control again moves out looking for try blocks (they can be nested in a single method) and up out of methods to their callers, looking for another potential try block
  - If a suitable catch block is found, the exception is considered "handled", at this point, the catch block executes, any associated finally block executes, and execution continues after the finally block
  - An exception might arise in a catch or finally block, and the result is to restart the search for an enclosing try

Handle exceptions using try/catch/finally [...]

- If a try block starts executing, the finally block will definitely start executing regardless of whether an exception arose and was unhandled, arose and was handled by a catch block, or no exception arose
  - The order will be execute the try block until/unless an exception arises
  - If an exception arose and a matching catch is found, then execute the catch
  - Execute the finally block
  - If no exception arose, or if one arose but a matching catch was found, continue normally after the finally block
  - If an exception arose, but no matching catch was found, jump out of the method to the caller (where the decision process repeats)
- If, however, an exception arises during execution of the finally, then the finally might not complete normally
- If an exception arises in a catch block, that exception will supercede the original, but the finally block will still execute in its turn.

Handle exceptions using try/catch/finally [...]

- If a checked exception could escape from a method, the method must declare the possibility with a throws clause in the method signature line. If multiple exceptions are possibilities these should be documented in a comma-separated list E.g.:

```
void mightBreak() throws FileNotFoundException, SQLException { }
```

- It is not mandatory to enumerate each exception type, it's permitted instead to enumerate a parent type that generalizes some or all of the exceptions. More than one generalized type may be listed and all possible checked exceptions must be covered by the generalizations listed
- Checked exceptions usually represent *recoverable problems*, and a healthy program should normally recover from these. Program bugs and catastrophic events are usually not checked, as recovery is impossible, unsafe, or at best severely challenging.

[...] multi-catch blocks [...]

Sometimes, more than one exception requires identical handling, this can be approached three ways:

- Provide multiple catch blocks, duplicating the code
  - Duplication is bad
- Provide a single catch block naming the nearest common parent exception
  - This is the correct approach if one of the exceptions is the common parent (e.g. `IOException` and `FileNotFoundException`)
  - This also catches every child of the common parent if all the exceptions are not in a single hierarchy
- Use a multi-catch, naming the individual exceptions, separated by a vertical bar / pipe character

  `catch (IOException | SQLException ex) {}`
  - Note that this structure prohibits listing exceptions that have a parent/child relationship
- The type of `ex` will be the nearest common ancestor of the listed exceptions, so only features common to all listed exceptions will be useable
  - But if specific features are needed, this is not "identical handling"
- If `ex` is re-thrown, only the listed exceptions need be declared in the throws clause of the method

[...] try-with-resources [...]

The original intent of finally was to allow "cleanup" of resources regardless of whether an exception arose or not, typically, this means closing of IO connections. The structure is hard to use well and try-with-resources updates it

- In this structure we declare resources that should be closed and the compiler generates code to close them
- Resources must implement the `java.lang.AutoCloseable`interface
  - The interface declares a single abstract method `void close() throws Exception`
  - User provided classes may implement this in which case they will work correctly in the try-with-resources system
- Resource references must be final or effectively final
- Resources are closed in reverse order of their appearance in the resources block
- The scope of references declared in the resources block is point of declaration to the end of the try body
  - These behave as "formal parameters"
  - Resources may refer to previous resources to build chains/pipelines
- Resources may be opened before the resources block and simply mentioned in the resources
  - These will be closed, but the variable will remain in scope; Note that the variable must be effectively final

[...] try-with-resources [...]

- Example:

```
try (var inFile = Files.newBufferedReader(Path.of(inName));
     var outFile = Files.newBufferedWriter(Path.of(outName));) {
    char [] buffer = new char[8];
    int count;
    while ((count = inFile.read(buffer)) >= 0) {
      outFile.write(buffer, 0, count);
    }
  } catch (IOException ioe) {
   System.out.println("something broke");
}
```

Resource declarations are separated, and optionally terminated with semicolons

Resource declarations may use `var`, or explicit types, but resources must be `AutoCloseable`. *Declarations can simply mention a variable already in scope.*
All these will be closed in reverse order of mention when try, and any activated catch block, have completed. This is *before* any explicitly coded finally block

An optional `finally` may be included, after all the catch blocks, but this is not relevant for closing resources (which will typically be out of scope anyway at that point)

[...] try-with-resources [...]

- The close method of AutoCloseable is permitted to throw checked exceptions, and any method can throw unchecked exceptions
  - This means that it's possible for exceptions to arise while closing resources, whether or not the main program flow completed normally
  - In the JVM there can only be a single active exception for any single thread—this was a problem often missed, resulting in "lost" exceptions with simple try/catch/finally
- The try-with-resources mechanism does not "lose" exceptions, instead:
  - The first exception to arise (whether it arises in the try block during business logic processing, or in the closure process) becomes the "main" exception—this is the exception that is thrown to the caller
  - All subsequent exceptions are added to an array of "suppressed exceptions" that is attached to the main exception
  - The suppressed exceptions can be accessed (usually in the caller, or while printing stack traces) using the method defined on the `Throwable` class: `public final Throwable[] getSuppressed()`
  - Suppressed exceptions are added in the order they arise—recall that resources are closed in *reverse order* of mention in the resources block

[...] custom exceptions

- Programmers can add custom exceptions to describe problems
- These will usually be subtypes of `java.lang.Exception`, or `java.lang.RuntimeException`
  - Subtypes of `java.lang.Exception` are checked exceptions
  - Subtypes of `java.lang.RuntimeException` are unchecked
- User exception types should delegate constructors to their parents to allow message, cause, and suppressed exceptions to be recorded and handled properly

Working with Arrays and Collections

- Create arrays, List, Set, Map and Deque collections, and add, remove, update, retrieve and sort their elements.

Create arrays [...]

- Arrays are objects, variables of array type are references
- Arrays have a strongly, statically defined, element type that is also known and checked at runtime
- Arrays support two literal syntaxes, depending on the usage context
- In any case, new, the element type, empty square brackets, and a comma separated list of elements in curly braces:

```
Object obj = new String[] {"Hello", "Goodbye", null};
```

- If the context *unambiguously* defines (notably, this *does not include* actual parameters and return values of methods) the base type of the array this can be shortened:

```
String[] obj = {"Hello", "Goodbye", null};
```

- If a literal form is *not* required (that is, if initializing values will not be specified in the initial creation) a third form is provided, in which the number of elements must be provided (the size is specified with an expression; a constant is not necessary):

```
String[] obj = new String[3];
```

[...] arrays [...] add, remove, update, retrieve [...]

- The size of an array is *fixed* at initial creation
- The size of an array can be queried with the `length` read-only field
- If an array is not big enough for an operation, a new array must be allocated and the contents copied. This usually requires reassigning the reference(s) to refer to the new array.
- Copying contiguous elements in an array should be done with the `System.arraycopy`method
- Subscripting is used for read or write access:

```
int[] numbers = {0, 1, 2, 3}; String [] names = new String[10];
numbers[2] = 10; // assign / update the element
int nameIndex = 3;
String n = names[nameIndex]; // read / retrieve the element
```

- Indexes are `int` expressions, in the range 0 … length - 1
- Arrays are always contiguous, not sparse
- Access outside the valid range raises a `java.lang.ArrayIndexOutOfBoundsException`
- Array elements cannot be "added" or "removed", only overwritten

Create [...] List, Set, Map and Deque collections [...]

- `List`, `Set`, `Map`, `Deque` are interfaces, and have several implementing classes in the core APIs and are often created by instantiating these types
- *Lists* provide array-like storage: elements are stored in an order controlled by the programmer, and can be read from those positions. The list can be iterated, and can (usually) grow or shrink dynamically
- *Sets* provide storage with fast searching for the presence or absence of an object. The (generally) use an internally controlled order that facilitates the fast search, rather than maintaining the order specified by the program's operations. A set will contain at most one of any given object.
- *Maps* provide key-value pair storage, and can be searched quickly by key. The keys are unique (no duplicates, similar to a set)
- *Deques* are "double-ended queues". They have similarities with lists in that items are (usually) stored in an order that's controlled by the program, and they can grow and shrink, but items can be added to either "end" of the structure, and similarly be removed from either end. Their use case tends to be rather esoteric, the classic example being task management in operating systems.

Create [...] List, Set, Map and Deque collections [...]

- `List`, `Set`, and `Map` provide convenient methods for creating "unmodifiable" structures of these types. The creating is done using static factory methods on the interfaces, and the implementing classes of these unmodifiable structures are anonymous.
  - These data structures are referred to as "unmodifiable" rather than immutable, since it's just the structure that cannot be changed. If the structure contains references to mutable elements (e.g. a `List` that contains `StringBuilder` elements) there is nothing the structure can do to prevent mutation of the element directly.
- Each interface provides two methods `of` and `copyOf`
  - The `of` method creates a new unmodifiable structure using the argument elements
  - The `copyOf` method returns an unmodifiable structure containing the same elements (by reference, so again, see the caveat about mutable elements) as the original structure. If the original structure was modifiable, the returned object will be a newly created unmodifiable structure. If the original structure was already an unmodifiable type, then that object is returned unchanged.

Create [...] List, Set, Map and Deque collections [...]

- `Arrays.asList` places a view around the storage of an array. This allows interaction using the methods of list, but the storage is of fixed size. Elements can be `set`, but not removed or added. Changes made in the underlying array will be visible through the list view, and vice versa.
- `Collections.unmodifiable`*`Xxx`* (`List`, `Set`, etc.) creates a view that rejects any changes, but the underlying structure might still be modified, and in that case, the change will be reflected.
- Note that `Deque` does not provide unmodifiable features, since it is intended to be a dynamic queue-like structure

[...] collections [...] add, remove, update, retrieve [...]

- `List`, `Set`, and `Deque` implement the `Collection` interface, which allows them to be iterated directly via the `Iterator` interface (a super-interface of `Collection`) and by the enhanced-`for` loop.
- `Map` is *not* a `Collection`
- A map's keys can be extracted as a `Set` using the method `keySet()`
- A map's values can be extracted as a `Collection` (duplicates are possible), using the method `values()`
- A map's key-value pairs can be extracted as a `Set` of `Map.Entry` objects
  - `Map.Entry` is a nested interface that defines two access methods `getKey()` and `getValue()`
  - `Map.Entry` defines a mutator method for the value `setValue(...)` — this will throw an exception if the underlying map is unmodifiable
  - A `Map.Entry` is a *view* on the underlying data. That is, changes made through the view will be visible in the original data structure
- Iteration of a map's contents can be performed on the key-set, the values, or the set of entries obtained from the above methods

[...] List [...] add, remove, update, retrieve and sort [...]

- `List` methods: `add, addAll, addFirst, addLast, clear, contains, containsAll, equals, get, getFirst, getLast, hashCode, indexOf, isEmpty, lastIndexOf, remove, removeAll, removeFirst, removeLast, retainAll, reversed, set, size, sort, subList, toArray`
- Sort:
  - in place (mutates the original) using `sort(Comparator)`
  - via a stream using `sorted` which does not mutate the original data structure
- Iterate using: `listIterator, iterator`
- Functional type methods: `forEach, parallelStream, removeIf, replaceAll, spliterator, stream`
- Note that `equals` and `hashCode` are expected to work across *different List types*, comparing elements at the same position for equivalence using their `equals` methods

[…] Set […] add, remove, update, retrieve and sort […]

- `Set` methods: `add, addAll, clear, contains, containsAll, equals, hashCode, isEmpty, remove, removeAll, retainAll, size, toArray`
- Set order is normally defined internally, so sorting is not relevant
- Iterate using: `iterator`
- Functional methods: `forEach, parallelStream, removeIf, spliterator, stream`
- Note that `equals` and `hashCode` are expected to work across *different Set types*, comparing elements for equivalence using their `equals` methods

[…] Map […] add, remove, update, retrieve and sort […]

- `Map` methods: `clear, containsKey, containsValue, entrySet, equals, get, getOrDefault, hashCode, isEmpty, keySet, put, putAll, putIfAbsent, remove, replace, size, values`
- Map order of keys is normally defined internally, so sorting is not relevant
- Iteration is not directly supported, but the key set, values, and entry set may be obtained, and each of these interfaces supports iteration directly
- Functional methods: `compute, computeIfAbsent, computeIfPresent, forEach, merge, replaceAll,`
- Note that `equals` and `hashCode` are expected to work across *different Map types*, comparing key-value pairs for equivalence using the `equals` methods of each part

Working with Streams and Lambda expressions

- Use Java object and primitive Streams, including lambda expressions implementing functional interfaces, to create, filter, transform, process, and sort data.
- Perform decomposition, concatenation, and reduction, and grouping and partitioning on sequential and parallel streams.

Use Java object and primitive Streams [...]

- Streams provide for functional style processing of potentially very large data sets
  - This is achieved by processing one element at a time in each thread that's working on the stream
  - Provides a simple mechanism to allow multiple threads (and thereby CPU cores) to operate on the data set simultaneously, with the goal of increasing throughput, while avoiding the usual complications of threading
  - Data is processed in a "lazy" or "pull" fashion, such that data move through the process pipeline on request from the terminal operation, rather than being pushed from the source
- Three primary methods operate on data in the stream, these are `filter`, `map`, and `flatMap`
- Various methods are provided for aggregating a "final result" from the data reaching the end of the stream
  - Including 3 `reduce` methods, 2 `collect` methods, and a number of `Collector` types
- Used as intended in a functional style, the processing elements:
  - Operate on each data element "in isolation" and do not access any variable data other than the local data of the functions that are invoked at each step
  - Do not mutate any data–instead steps create new data items to reflect changes and pass down the stream pipeline
  - These rules are what allow easy threading—the usual difficulties in threading arise from the use of shared data that is mutated
- Java's generics mechanism handles primitives by boxing and unboxing, which is less efficient. To mitigate this, three variations of stream dedicated to `int`, `long`, and `double` primitive data are provided, along with conversion tools among these and the generic reference-type stream

Use [...] Streams [...] filter, transform, process [...]

- The `filter` method is used to drop items from the data flow
- This takes a `Predicate` of the data type (possibly a primitive), if the test returns true, the data item progresses down the stream, otherwise it is abandoned
- This operation has a one-to-zero-or-one effect; it cannot change the data only conditionally drop it

- The `map` method takes a `Function` argument. Each data item reaching the `map` is passed into the function, and the returned value from that function is passed down stream
  - Variants that convert among the primitive and reference stream types are provided, e.g.: `mapToInt`, `mapToObj`
- This method has a one-to-one effect, but the downstream data may be of any type, or the same type with a potentially different value

Use [...] Streams [...] filter, transform, process [...]

- The `flatMap` method takes a `Function<A, Stream<R>>` argument. Each data item reaching the `flatMap` is passed into the function which must return a Stream of some kind (primitive streams are possible too). All of the data (which could be zero items) in the returned stream are then passed down stream
  - Variants that convert among the primitive and reference stream types are provided, e.g.: `flatMapToInt`, `flatMapToObj`
- This method has a one-to-many (including zero) effect, and the downstream data may be of any type, or the same type with a potentially different value
- This method is broadly parallel with the concept of a *monad* found in many functional programming languages and category theory (however, it does not conform to the rigorous mathematical definition)

Use [...] Streams [...] filter, transform, process [...]

- The `forEach` method takes a `Consumer` argument. Each data item reaching the `forEach` is passed into the consumer which does something with the result (e.g. printing it out, or storing it in a database)
  - The consumer returns void, and so does the `forEach` method
- This method is one example of a terminal method
  - Terminal methods *do not* return a Stream object, so no more chaining is possible
  - More generally, terminal methods produce a "final result", often returning a value, although forEach is a void method

- Example, take three strings, convert them to upper case, convert them to individual letters, filter out the vowels, print all the resulting letters, note that char is a 16 bit unsigned number, and the smallest number type that has efficient processing (avoiding boxing) is int:

```
int [] vowels = {'A', 'E', 'I', 'O', 'U'};
Stream.of("hello", "there", "and", "welcome")
  .map(s -> s.toUpperCase())
  .flatMapToInt(s -> s.chars()) // chars produces an IntStream
  .filter(c -> Arrays.binarySearch(vowels, c) < 0)
  .forEach(c -> System.out.println((char)c));
```

Use [...] Streams [...] filter, transform, process [...]

- We have mentioned these stream instance methods:
  `filter, flatMap, flatMapToDouble, flatMapToInt, flatMapToLong, forEach, map, mapToDouble, mapToInt, mapToLong`
- There are many more:
  `close, dropWhile, isParallel, iterator, limit, mapMulti, mapMultiToDouble, mapMultiToInt, mapMultiToLong onClose, skip, spliterator, takeWhile`
- And more terminal methods:
  `allMatch, anyMatch, count, findAny, findFirst, forEachOrdered, max, min, noneMatch, toArray, toList`
- Yet more that we will address
  `collect, distinct, parallel, peek, reduce, sequential, sorted, unordered`
- Primitive streams also provide some additional methods:
  `average, boxed, mapToObj, sum, summaryStatistics`

[...] lambda expressions [...]

- Lambda expressions are *object expressions* of an anonymous type that implements a functional interface
- The syntax allows the programmer to provide an implementation for the single abstract method declared by the functional interface
- The syntax is designed for brevity and to focus on the behavior of the function's implementation with minimal "syntactic scaffolding"
    - So, no class body is needed, and the method body is minimized
    - Instead a formal parameter list (often simplified) is provided and an (often simplified) method body

[...] lambda expressions [...]

- Traditionally, a class can implement `Predicate<String>` to test if a `String` is longer than 3 characters:

```
class LongString implements Predicate<String> {
  @Override public boolean test(String s) { return s.length() > 3; }
}
```

- This could then be instantiated and used in a `filter`:

```
Stream.of("Hello", "I", "The")
  .filter(new LongString())
  .forEach(System.out::println);
```

- Alternatively, the lambda syntax allows the highlighted part of the class above to be modified and embedded directly, the arrow is key to the lambda syntax, and an instance of the now-anonymous class is automatically created:

```
.filter((String s) -> { return s.length() > 3;})
```

- Notice the formal parameter list, and the method body are unchanged, but the method and class declaration parts are elided
  - Further simplifications are permitted...

[...] lambda expressions [...]

- A lambda expression can only be used where the context makes it unambiguous what type the lambda must conform to, and that type is a functional interface. Hence *this is not permitted*:

  ```
  Object longStringPred = (String s) -> {return s.length() > 3;}; // NO CONTEXT
  ```

  But this is fine:

  ```
  Predicate<String> longStringPred = (String s) -> {return s.length() > 3;}; // OK
  ```

  - Notice that the highlighted lambda part is identical—but without context indicating this should be a Predicate<String> the compiler rejects the code
- Context usually comes from an assignment of the lambda to something. The target of the assignment can be:
  - A variable—which does not have to be simultaneously declared
  - A method's actual argument
  - A method's return value
- It's also possible to define the context of a lambda expression using a cast:

  ```
  Object lSP = (Predicate<String>)((String s) -> {return s.length() > 3;});
  ```

[...] lambda expressions [...]

- Where the types of all the formal parameters are known from the context, they can all be left out, or they can all be replaced with `var`
  - This is *all or nothing*—all explicit types, or all `var`, or all without any type information
  - The use of var simply provides a place to attach an annotation to a formal parameter
- Where exactly one formal parameter exists, and no type, nor var, is used, the parentheses can be omitted:

```
Predicate<String> longStringPred = s -> {return s.length() > 3;};
```

- Where the body of the method is only a return statement, this can be simplified further:

```
Predicate<String> longStringPred = s -> s.length() > 3 ;
```

- A lambda can create a closure around method local data provided that variable is final, or effectively final.
- This is permitted:

```
int[] c = {0}; Consumer<String> cs = s -> System.out.println(s + ++c[0]));
```

- But this is not:

```
int c = 0; Consumer<String> cs = s -> System.out.println(s + ++c));
```

UNDERSTANDING > KNOWLEDGE

DANCING CLOUD SERVICES, LLC

[...] lambda expressions [...]

- Some lambdas have a "shorthand" form (not always shorter!) called a *method reference*
- These forms can only describe a lambda that takes its parameter(s) and returns an method invocation expression directly
    - They have no formal parameter list for the method invocation
    - The actual parameters to the method must *exactly match* the formal parameter list that would have been required for the "longhand" lambda—*this applies to the type, the order, and the values*
    - The syntax uses a double colon, e.g. `System.out::println` — this is the entirety of a method reference, no formal parameters, arrow, block, nor anything else can be present
    - There must exist a method on the target class or object that fits the expectations of the context

[...] lambda expressions [...]

- Method references can be used in a variety of situations, note the number of arguments is not a constraint on using a method reference, except:
  - The argument count, order and type must match what's expected for the lambda's context
  - In the case of target instance method invocation, at least one argument is required, since there has to be a target object to invoke the method on
- Most of the time these things can be read without ambiguity—they do what you think they will do

Static method	`(a,b) -> Integer.sum(a,b)`	`Integer::sum`
Explicit instance method	`a -> "Root".concat(a)`	`"Root"::concat`
Constructor invocation	`y -> new ArrayList<>(y)`	`ArrayList::new`
Array construction	`s -> new String[s]`	`String[]::new`
target instance method	`s -> s.length()`	`String::length`

[...] sort data [...]

- Methods of the Stream classes allow for sorting the data, these are called `sorted(...)`
- Variations exist to allow the natural order of objects, or specification of a Comparator
- Primitive streams support sorting in natural order only
- Sorting, inevitably, *jeopardizes scalability*, since the entire data set must be stored in memory before any data can continue downstream

- Duplicates can be removed from a stream using the method `distinct()`
- This is also available on both object and primitive streams
- This must store one of each unique data item during the stream processing, so it *jeopardizes scalability*
- Distinction for object data is determined using the data's own equals method—if this is not correctly implemented, the results will not be as intended

Perform decomposition, concatenation [...]

- Streams may be created from many sources, not just from `Collection` objects
- Create them from:
  - Provided data in a variable length argument list: `Stream.of(...)`
  - A single data item that might be null (implying the stream should be empty): `Stream.ofNullable(...)`
  - A builder that accepts individual elements with multiple calls: `Stream.builder().add( … ).add( … ).build()`
  - Concatenation of other streams: `Stream.concat(...)`
  - An empty stream: `Stream.empty()`
  - Generated by calling a Supplier repeatedly: `Stream.generate(supplier)`
  - Iterated from a starting value and then by calling a unary operator repeatedly: `Stream.iterate(initial, operator)`

Perform decomposition, concatenation [...]

- Streams can run in a single thread—*sequential mode*—or in multiple threads—*parallel mode*
  - In parallel mode, the system launches multiple threads and shares out the data across those threads so each thread performs a subset of the total work. The splitting up of the data is sometimes called *sharding*, or *decomposition*
  - This will often (but not always) improve throughput and reduce the time to completion
  - Enable parallel mode using the `parallelStream` method of a `Collection`, or `parallel` method of stream types
- Parallel mode can be turned off using the `sequential` method
- Parallel vs sequential mode is a decision made while the pipeline is being constructed, when the stream processes data *the entire operation is either sequential or parallel*

Perform decomposition, concatenation [...]

- By default parallel mode *maintains ordering*
  - This means the *terminal operation receives items in order*
  - So reduce operations need not be commutative (but must be associative)
  - Ordering can be disabled using the `unordered` method
- Some data sources, and some terminal operations, are intrinsically unordered, in which case order will not be maintained. E.g.:
  - Using a `HashSet` as a data source
  - Using `forEach`, rather than `forEachOrdered` as the terminal operation
- When a stream runs in parallel, it's particularly important that operations executed by stream methods (map, etc.) adhere to thread-safety rules:
  - Do not mutate data (only create new data to represent results)
  - Do not interact with (read or mutate) shared data
  - Do not make assumptions about the order in which individual items will be processed by the pipeline (their order will be restored at the terminal operation if the stream is running in ordered mode)

[...] reduction [...]

- Producing a final answer from a stream must be done incrementally to avoid having to load all the values into memory at once. This is done using a process referred to as "reduction"
- A simple example is addition to find the sum of all values, we start with the "sum so far", which is zero, and then repeated call an operation to produce an updated "sum so far" from the previous value and the new data item. E.g.:
  Sum so far = 0, next data item = 1, new Sum so far = 0 + 1 -> 1
  Sum so far = 1, next data item = 4, new Sum so far = 1 + 4 -> 5
  Sum so far = 5, next data item = 2, new Sum so far = 5 + 2 -> 7
  Sum so far = 7, next data item = -3, new Sum so far = 7 + (-3) -> 4
  When the data runs out, our "sum so far" is our final answer
- This is generalized to a starting value of the data type, and a `BinaryOperator` of the data type
- If the stream is empty, the result is the initial value
- Example, sum of numbers 1,2,3,4. Note initial value 0, and operation to add two values:

```
int sum = IntStream.of(1,2,3,4).reduce(0, (a, b) -> a + b);
```

UNDERSTANDING > KNOWLEDGE

DANCING CLOUD SERVICES, LLC

[...] reduction [...]

- A variation of reduce is possible without a starting value. This takes the first value as the starting value, but cannot produce a result at all if no data is in the stream
- The possibility of *no result* is represented using an `Optional` type, this is like a stream in that it provides methods like `filter`, `map`, etc. but it contains either zero, or one, element.
  - Primitive variants for `int`, `long`, and `double` are provided too
  - Instead of a `forEach`, optional types provide a method called `ifPresent`
  - It can help avoid null pointers and null pointer exceptions
- Example. Notice there's no initial value, the `reduce` operation produces an `OptionalInt`, and that object has an `ifPresent` method:

```
OptionalInt sum = IntStream.of(1,2,3,4).reduce((a, b) -> a + b);
sum.ifPresent(i -> System.out.println(i));
```

[...] reduction [...] on sequential and parallel streams [...]

- If the data stream were sharded into sub-groups, and each sub-group were passed for processing to a different thread (hopefully using a different CPU core), we might get the work done in less elapsed time. However, to get the right answer, several conditions must be met, some of these conditions relate to concurrent processing, others to the logic.
- Conditions related to concurrent processing
  - Each item must be processed independently of the others with no reference to "outside" data—so long as we are not sharing any mutating state between threads in our processing we can avoid the usual concurrency issues. Note that this supposes that the processing infrastructure is also built so as to handle this correctly, and to shard the data across the threads in a thread safe way.
  - When performing the reduce operation the result must eventually reach a single thread, and we assume that the movement of data across threads is performed correctly by the infrastructure
- Each thread will create a "result so far" from the sub-group of data that it processes. These must be merged into a final answer for the entire data set.
  - The associative binary operator can be used to achieve this, provided the infrastructure moves the results across threads correctly (which it does, and is not our problem!)
  - Unless the binary operator is *commutative*, in addition to associative, the infrastructure must somehow maintain the overall ordering (which it does by default). Maintaining ordering can be very expensive in terms of memory.

[...] reduction [...] on sequential and parallel streams [...]

- Conditions related to logic
  - The order of operations must be maintained, unless it's know that the operation is commutative (this assumption is not made by the stream api, and the order of operations is maintained even in parallel mode)
  - The grouping of operations must not matter, that is, the operation must be *associative*, this is necessary since the single thread model might have started with (((a + b) + c) + d), but the parallel form might perform (a + b) + (c + d)
  - If we have a "starting value", this must not make any difference to the result of the calculation no matter how many times it's included in the calculations. Clearly we can add 0 to a sum any number of times without altering the result.

Side note: If we have a ***data type T***, an ***associative binary operator on type T***, and a ***value of type T*** that can be incorporated in the calculation without changing the result, mathematics refers to this as a ***monoid***. The value is called an ***identity***.
Don't confuse monoid with monad; monad is the name for a thing with a flatmap behavior!

[...] reduction [...] on sequential and parallel streams [...]

- A restriction of the reduce operations seen so far is that they produce a result of the same type as the stream data
  - This can be handled by using the map method to transform individual items into "result of a single operation", and then reducing the partial result types
- The object stream api (but not the primitive variants) provides an alternative reduction approach that allows data items to be merged directly with a "result so far" of a different type
- For this operation to work, and as before, we need an initial "result so far" representing the result of zero data, and an operation that creates a new "result so far" from the existing result and the data item
  - This operation will not be a `BinaryOperator<StreamDataType>` but instead a **`BiFunction<ResultType, StreamDataType, ResultType>`**
- However, if this is run in the parallel mode, we would be left with multiple "result so far" objects, and these must be merged. This is done by a third formal parameter to this type of reduce, which is a `BinaryOperator<ResultType>`
  - This operation is not invoked unless the stream is running in parallel mode, but *cannot be null even in sequential mode*

[...] reduction [...] on sequential and parallel streams [...]

- Reduce operations create a new "result so far" at every new data item, this sometimes reduces scalability due to large amounts of allocation/initialization and perhaps garbage collection. To mitigate this, another variation is provided which mutates a "result so far" object, rather than making a new one every time
- For this to be thread-safe in parallel mode, each thread will be given its own result object. Because that object is not shared, the thread can mutate it safely.
- When all data of any given shard has been processed, the result object will need to be merged with some other completed result object. This requires a new combining approach.
- Because of these requirements, the three formal parameters are:
  - `Supplier<ResultType>` — this allows the infrastructure to create new result objects when it needs to
  - `BiConsumer<ResultType, StreamType>` — this is used to update the result with the next stream data item
  - `BiConsumer<ResultType, ResultType>` — this is used to update one result with the contents of another result
- Although the primitive streams do not provide the three argument form of reduce, they do provide the three argument collect described here

[...] grouping and partitioning [...]

- The reduce and collect operations can be used to create arbitrary business-logic-specific results, but the APIs provide some pre-packaged utilities for commonly used final results. These are provided in several parts:
  - A single argument `collect` operation that takes an object implementing the interface `Collector`.
  - A `Collector` (roughly speaking) packages the three features of the three-argument `collect` operation just discussed, plus some refinements.
  - The `Collector` interface can be implemented by programmers who want to provide packaged utilities for their own business domain
  - A class called `Collectors` provides factory methods for common use cases

[...] grouping and partitioning [...]

- A key behavior of the provided collectors is building a table (represented as a `Map`) from the stream data
  - [These operations are closely equivalent to the GROUP BY operation of SQL databases]
- Two variants are provided, one builds a map that has two keys, true, and false. Stream data, possibly further manipulated, is used to build the values associated with each of these keys.
- Example. Suppose we have a stream containing student data stored in records like this:

```
record Student(String name, int courses) {}
```

and we have a list of students, like this:

```
List<Student> students = List.of(
    new Student("Ishan", 3),
    new Student("Inaya", 4),
    new Student("Ayo", 1),
    new Student("Siobhan", 2)
);
```

We want to group those students who take more than two courses together, and those who take two or fewer together.

[...] grouping and partitioning [...]

- This does the work:

```
var table = students.stream()
  .collect(Collectors.partitioningBy(s -> s.courses() > 2));
```

- Collectors.partitioning by creates a Collector object that builds a Map
- Each student that passes the test (takes more than two courses) will be added to a List<Student> in the value part of a Map entry, against the key true, those that do not pass the test are added to a list stored against a key of false
- The resulting map looks like this:

```
{
false=[Student[name=Ayo, courses=1], Student[name=Siobhan, courses=2]],
true=[Student[name=Ishan, courses=3], Student[name=Inaya, courses=4]]
}
```

[...] grouping and partitioning [...]

- Partitioning always produces a table with two keys, true and false. To build a table with flexible keys, the `Collectors.groupingBy` features can be used.
- In the simplest form a single argument is provided to the factory, and this is used to create the key for each stream data item, again, the items are added to a list stored as the value against that key
- It's also possible to add "downstream processing" so that instead of creating a simple list with the original stream data items, post processing (in a stream-like fashion) can be applied to the data, and other final data types can be created.
- Example, we want to create a table with congratulatory messages for our students based on the number of courses they take. We will consider three or more courses "vigorous", two courses will be "enthusiastic", and a single course is "showing promise". We will need to create the text for the key, and then instead of adding the students to a list, extract just their names.

[...] grouping and partitioning [...]

- This function takes a student and gives the categorizing word used for grouping:

```
Function<Student, String> message = student -> {
  int count = student.courses;
  if (count >= 3) return "vigorous";|
  if (count == 2) return "enthusiastic";
  if (count == 1) return "showing promise";
  return "oh dear!";
};
```

- This performs the grouping and conversion:

```
var table = students.stream()
    .collect(Collectors.groupingBy(message,
        Collectors.mapping(s -> s.name, // first downstream collector
            Collectors.joining(", ")))); // second downstream collector
```

- This generates the messages from the table:

```
table.forEach((k, v) -> System.out.println("Students " + v + " are " + k));
```

[...] reduction [...]

- Stream processing is generally lazy, data is pulled by the terminal method
- Some terminal methods can give a result before all data has been seen
  - `allMatch`
  - `anyMatch`
  - `count`
  - `findAny`
  - `findFirst`
  - `noneMatch`

Packaging and Deploying Java Code

- Define modules and expose module content, including that by reflection, and declare module dependencies, define services, providers, and consumers.
- Compile Java code, create modular and non-modular jars, runtime images, and implement migration to modules using unnamed and automatic modules.

Define modules [...]

- Modules provide additional layers of encapsulation by:
  - Reducing the accessibility of public features to less than "anywhere in the JVM"
  - Creating a meaningful enforcement of the package-level access
  - Changing the default behavior of reflection so that it cannot break the encapsulation of private features unless the programmer explicitly permits this
- Modules also provide a means of organizing groups of code, for example individual libraries, and explicitly documenting the dependencies among them
  - Those dependencies are also prohibited from circular relationships
- The module system simplifies distribution of finished programs by allowing the necessary parts of the JVM, it's libraries, and supporting libraries, to be packaged with the code into a single unit
- The module system can speed up program launch by indicating to the JVM exactly which modules might be needed, rather than forcing the search to check through potentially every jar and classfile found on the classpath
- The module system formalizes a mechanism for distributing, finding, and using "services"

Define modules [...]

- A module is defined in source code by the presence of a `module-info.java` file in the root directory of the package hierarchy
- The contents of the `module-info.java` file are broadly made up of the word `module` followed by a module name and a block containing any number of:
  - exports directives
  - requires directives
  - opens directives
  - uses directives
  - provides directives
- The module name follows the same basic form as a package name (identifiers concatenated with dots, all lowercase by convention)
  - Unlike packages, the directory or archive file that contains the module (either as source or binary) will be a single directory with a dotted name, rather than a series of sub-directories

[...] declare module dependencies [...]

- If a module (the "user") is built in a way that needs features of another module (the "provider") then:
  - The user must actively express the need for the provider so as to be granted access
  - The provider must be available to the JVM when the program starts
- A corollary of the second point is that in general, the JVM knows it can ignore modules for which no need has been expressed
  - This point has some subtleties, particularly with respect to services, and when migrating a project
- A user module expresses the need for a particular provider with a `requires` directive
- A `requires` directive has *both of the effects* mentioned above:
  - The terminology used for being granted access is that the user module *reads* the provider module—reads is implicit in a requires directive, but the relationship can be created in other ways
  - A required module must be available to the JVM at launch time, or the JVM will refuse to start
- Example:

```
module my.module {
  requires java.sql; // core modules (java.base is implicitly required)
  requires your.support.module;
}
```

[...] expose module content [...] reflection

- An `exports` directive is followed by a package name, and indicates that the specified package may be made visible to other modules if they "read" it
  - An optional list of other modules may be provided after the word `to`, in which case only the named modules are permitted to read this module
  - This list is permitted to include non-existent modules—this can simplify handling unfinished projects
- Example: `module my.module { exports com.mystuff.usefulpackage; }`
- An `opens` directive is followed by a package name, and indicates that code in other modules can use the reflection mechanism to investigate and interact with the specified package
  - An optional list of other modules may be provided after the word `to`, in which case reflection is limited to those modules
  - This list may include non-existent modules
- Example: `module my.module { opens com.mystuff.reflectable to my.other.module; }`
- If an entire module is to be open to reflection, the module declaration can carry the prefix open
  - In this case, no opens directive is permitted
- Example: `open module my.module {}`

[...] expose [...] services, providers, and consumers

- A `provides` directive is followed by the type of a service generalization (usually, but *not necessarily, an interface*). It indicates that an implementation of that service type is available from this module
- The text `provides <service type>`is followed by `with <implementing class> ;`
  - The generalization must be available in this module, but *might* be from another module
  - The generalization must also be available to potential users of the service
  - The implementing class must be in the module, and is not expected to be in an exported package—users of the service do not expect to know the specific type that provides the service
- Example:

```
module my.module {
  exports com.myservices;
  provides com.myservices.ServiceInterface with com.myimpls.MyServiceImpl;
}
```

[...] services, [...] consumers

- A `uses` directive is followed by the type of service generalization that is desired
  - The module using a service will normally `require` the module that exposes the service interface type
  - The service user code is written in terms of this interface type
  - The module using a service does not have to `require` the module that contains the *implementation* of the service
- Example:

```
module your.module {
  requires my.module;
  uses com.myservices.ServiceInterface;
}
```

Compile Java code [...]

- Compilation of Java code requires knowledge of:
  - Where to put the output code
  - Where to find existing compiled modules needed to support the compilation
  - Where to find sources for modules
  - What code to compile
- Specify the output location using `-d <directory>`
- Specify how to find existing pre-compiled modules using `--module-path <modules-directories>`
  - Multiple directories are separated using semicolon on Windows, and colon on Linux/Unix and Mac/BSD
  - A shorthand for this is `-p`
- Source directories are found based on a pattern, of which part of the pattern should match the module name. Specify a pattern using `--module-source-path <source-path-pattern>`
- Example, suppose we specify `--module-source-path "./projectsrc/*/src/main/java"` then the compiler would look for source for module `my.module` in the directory `./projectsrc/my.module/src/main/java`
- Specify the module(s) to compile using `--module <module-list>`
  - A shorthand for this is `-m`
  - The module list must not contain spaces, but can enumerate more than one module separated by commas

[...] create modular and non-modular jars [...]

- Module jar files are created essentially the same way as traditional jar files, by placing the entire directory structure containing the binary (.class) files and supporting resources into a Java Archive (jar) file
- The root directory of the jar file should be the root of the package naming that is contained within
- If a `module-info.class` file (which is compiled from the `module-info.java` file) is found in the root, then this is a modular jar, otherwise it is non-modular

[...] create modular and non-modular jars [...]

- If a non-modular jar file is placed on the module-path rather than the classpath, then it becomes an *automatic module*
  - If the jarfile has an attribute "Automatic-Module-Name" in its main manifest, then that's the name used
  - Otherwise, the name of the automatic module is derived from the name of the jar file, essentially anything that looks like version information is dropped, as is the .jar extension — full details are in the API documentation for the `java.lang.module.ModuleFinder.of` method
- If a jarfile is loaded from the classpath, the contents become part of the *unnamed module*
- Explicit modules (those with module-info.class files) can only access other named modules, and must declare that they `requires` them
  - Automatic modules are named modules
- Automatic modules:
  - *read* all other modules (important since they can't declare dependencies)
  - *cannot force loading* of other modules (because they can't declare `requires`) we might need to use the runtime directive ––add-modules to force loading
  - implicitly export all packages (again, they can't export anything explicitly)
  - can interact with the unnamed module (thereby providing a bridge between full modules and legacy libraries)

[...] migration to modules using unnamed and automatic modules

- Migration of projects to the JPMS will involve partially modular software with explicitly named modules, automatic modules, and jarfiles loaded into the unnamed module
- Migration can be done "top down" or "bottom up"
  - Top down implies modularizing the code containing the main method first, and adding automatic modules to bridge to the unnamed / classpath code that make up the rest of the system
  - Bottom up implies modularizing the lowest level code first, and forcing the system to load those modules using the `--add-modules` directive
- For more detail and worked examples on all of JMPS https://learning.oreilly.com/course/up-and-running/9780137475216/

[...] runtime images [...]

Command-line tools

- jdeps — analyse class and package level dependencies both on modules and on non-modular jars
- jlink — build a custom runtime image for a specific platform, although that need not be the host platform (although that's not trivially easy) jlink creates files in a format called jimage
- jpackage — generates installable packages for modular and non-modular Java applications, this tool uses jlink in its work

Managing Concurrent Code Execution

- Create both platform and virtual threads. Use both Runnable and Callable objects, manage the thread lifecycle, and use different Executor services and concurrent API to run tasks.
- Develop thread-safe code, using locking mechanisms and concurrent API.
- Process Java collections concurrently and utilize parallel streams.

[...] Use both Runnable and Callable objects [...]

- Two interfaces are intimately connected with the threading system, `java.lang.Runnable` and `java.util.concurrent.Callable`
  - Both are functional interfaces
- `Runnable` declares **`void run();`**
  - No arguments, no checked exceptions, naturally public and abstract
- `Callable` is generic in a single type parameter, and declares `T call() throws Exception;`
- The threading system can launch a new thread that executes the body of the method
  - The method can call other methods
  - The method has access to any instance variables in the object that contains it, and to any data that is captured by closure from the surrounding method
- `Callable` is particularly well suited to use with executors, since the value it returns is passed back to the `Future` (job "handle") that's created when such a task is submitted

[...] platform and virtual threads [...]

- Platform threads attach directly to a thread in the underlying operating system
  - This tends to make them relatively heavyweight objects and some operating systems can only support quite small numbers before running into scalability issues
  - Platform threads are a good choice for CPU intensive work that does not block for IO
- Virtual threads are managed by the JVM such that multiple virtual threads can share (time-domain multiplexing) a single platform thread
  - They are less resource intensive and can be created in much greater numbers
  - They're a good choice for tasks that are generally IO bound
  - By default, they do not have a name (this saves a bit of memory), however, a name can be set explicitly
  - They are always daemon threads—attempting to set them non-daemon throws an exception
  - They have a fixed (normal) priority—attempting to set a different priority is ignored

Create both platform and virtual threads [...]

- Platform threads can be created simply by constructing a new `Thread` object and passing a `Runnable` into the constructor.
  - It's fairly common to pass in a String to give the thread a specific name
  - Other arguments exist for more esoteric situations
- The `Thread` class has a builder capability too. Builders can be used to create *either* platform or virtual threads (a builder is created for a type of thread)
- The interfaces `OfPlatform` and `OfVirtual` have a common ancestor interface `Thread.Builder` which declares the common features of both types
  - Because virtual threads cannot be daemon, and setting priority is not effective (and some other differences), the specific builder features differ by the target thread type.
- To obtain a builder, execute one of these forms:

```
Thread.Builder.OfPlatform tbp = Thread.ofPlatform(); // builds platform threads
Thread.Builder.OfVirtual tbv = Thread.ofVirtual();   // builds virtual threads
```

Create both platform and virtual threads [...]

- To create a platform thread using a builder:

```
Thread.Builder.OfPlatform tb = Thread.ofPlatform();
tb.name("MyThread", 0); // creates names with sequential id numbers
Thread t1 = tb.unstarted(() -> {
  System.out.println("Thread running!");
});
t1.setDaemon(true); // not available for virtual thread
t1.setPriority(7); // not available for virtual thread
t1.start();
```

- The method `unstarted` has a sibling method `started` which creates the thread already started
- The sibling builder for virtual threads lacks daemon and priority control and is created like this:

```
Thread.Builder.OfVirtual tb = Thread.ofVirtual();
```

- The name method used in this example would create sequentially numbered threads if called to create multiples.
  - Names are generally not appropriate to virtual threads, but the interface supports them

[...] Use both Runnable and Callable objects [...] use different Executor services

- Threads can be created directly to work on a `Runnable` object, but `Callable` objects are launched by `ExecutorService` objects
- Executor service types are predefined with a variety of behaviors:
  - A fixed size pool of threads
  - A dynamically variable pool of threads
  - A scheduling ability, to start tasks after a delay, or periodically
- ExecutorService provides methods `execute(Runnable)`, and three variations of `submit(...)` these typically add the task (whether `Runnable` or `Callable`) to a queue, and the task will be picked up by an available thread in due course, and then executed.
- The execute method returns void
- The submit methods return a Future which allows subsequent interaction with the task:
  - `get()` methods return the value returned by the task, waiting (perhaps with a time limit) for it to complete if necessary
  - `cancel()` attempts to cancel the task, removing it from the queue if not already started, sending an interrupt if it has
  - `isCancelled()` determines if the task was cancelled prior to completing
  - `isDone()` determines if the task has completed normally
- `Callable` and `Future` are generic, indicating the return type of the `call` and `get` methods

[...] manage the thread lifecycle [...]

- Threads have a number of states visible at the Java level (platform threads might have different states in their underlying implementation, but this would be hidden)
  - NEW — A thread that has not yet started is in this state.
  - RUNNABLE — A thread executing in the Java virtual machine is in this state.
  - BLOCKED — A thread that is blocked waiting for a monitor lock is in this state.
  - WAITING — A thread that is waiting indefinitely for another thread to perform a particular action is in this state.
  - TIMED_WAITING — A thread that is waiting for another thread to perform an action for up to a specified waiting time is in this state.
  - TERMINATED — A thread that has exited is in this state.
- Thread state can be read (but not altered) by the `getState()` `Thread` instance method
- Thread states changes are mostly predictable from the meaning of the state

[...] manage the thread lifecycle [...]

- Threads may also be *interrupted* by a call to the `interrupt()` method of the `Thread` instance, this sets a condition in the thread, but doesn't change its state
- A threads interrupted condition remains set until the thread "notices" by one of these means:
  - an `InterruptedException` is thrown and caught
  - the thread tests its interrupt condition with the static `Thread.interrupted()` method
- The `Thread` instance method `isInterrupted()` does not reset the interrupted condition

Develop thread-safe code [...] concurrent API

Thread safety depends on three key constraints:

- **Timing** — avoid reading data that has not yet been written, and avoid overwriting data that is still in use elsewhere
  - Reads or writes that do not have reliable timing controls are referred to as *race conditions*
- **Transactional correctness** — avoid interacting with data while other threads could possibly be mutating it
  - Simply reading data can be a problem is that data is being concurrently modified
  - This problem is not unique to structured data, a read-modify-write cycle is also unsafe
- **Visibility** — counterintuitively, writing data to a variable in one thread is not sufficient to ensure that the update will be seen in another thread
  - Further, if one thread makes two writes in order it's possible for another thread to observe, one, the other, or neither update. It also might see the second update first
  - Visibility is a key purpose of the happens-before relationship
- Visibility is best achieved using the `java.util.concurrent`package
  - In general, if a "write" is performed by a concurrent utility, and a subsequent "read" from another thread uses a concurrent utility, then visibility will be handled correctly

[...] locking mechanisms [...]

- The `java.util.concurrent`package provides several utilities that can be used to create mutual exclusion and timing effects
- A `Lock` might best be considered like a source of magically-connected lockable doors
  - A call to `myLock.lock()` represents such a door
  - Initially, a call to `myLock.lock()` allows the requesting thread to proceed, but will simultaneously lock any other doors associated with this lock and ensure this door is locked behind this thread
  - This thread can proceed unhindered
  - If another thread comes along and calls the `lock()` method on the same lock object (even if the reference variable is different, and it's a different piece of code) then that thread will be frozen in its tracks (BLOCKED state)
  - Later, suppose the original thread calls `myLock.unlock()` , the second thread is now permitted to pass the door, which is immediately re-locked behind it
- A lock also provides `tryLock` methods (one supports a timeout). These methods:
  - Attempt to place the lock (waiting for the timeout if not available, giving up immediately if no timeout)
  - If successful, return true, otherwise return false

[...] locking mechanisms [...]

- Several caveats exist with this:
  - If many threads are trying to get through doors guarded by the same lock, one will be allowed to proceed when an unlock call is issued, but there is no guarantee of which
  - It's entirely the programmer's responsibility to ensure that the mutual exclusion effect this creates protects all the correct regions of code
  - It's entirely the programmer's responsibility to ensure that the lock is unlocked, and at at the right time/place, failure to unlock it will probably cause the program to stop, since the only thread that can get past a call to lock is the one that failed to release the lock
  - These locks are called "reentrant" which means if a single thread calls lock twice on the same lock object, it does not block the second time (contrast this with a semaphore) but must then call unlock twice to release the lock for others
  - It's highly recommended to use a structure like the following to ensure that unexpected exceptions don't cause locks to remain locked:

```
mylock.lock();
try { /* do transactionally sensitive stuff */ }
finally { mylock.unlock(); }
```

[...] locking mechanisms [...]

- If two threads need two locks, be very careful about locking order. Suppose thread A locks two locks L1 and L2 in that order, and thread B is coded to lock L2, then L1, it's possible to end up with thread A holding lock L1, waiting to obtain lock L2, while thread B holds lock L2 and is waiting for lock L1
- This situation is called a *deadlock* and is a nasty, and often hard to identify problem

[...] concurrent API

The java.util.concurrent API (and it's sub-packages) provides a number of feature categories

- **Executors** (and mostly ExecutorServices) — thread pool structures to execute our tasks
- **Queues** — Data written into one end of a queue and subsequently read out of the other creates a guarantee of visibility. Timing is easily supported in that an attempt to write to a full queue is made to wait (avoiding overwriting data) and an attempt to read an empty queue is made to wait (avoiding premature reading). However, since the queue passes a reference to a potentially mutable object, it's important that an object is not put into a queue until it's "stable" and then is never touched again by the writing thread
- **Synchronizers** — Various tools that can be configured to make one or more threads wait until a particular situation arises. It's the programmer's responsibility to ensure that the condition is correctly identified and notified, but the synchronizer will release threads on instruction and also create a happens-before relationship from the triggering thread to the others at the moment of release allowing for data visibility
- **Concurrent collections** — Thread-safe (and generally highly scalable) collections that behave properly in the face of concurrent use. Note that `java.util.Collection`collections might be corrupted by concurrent access but can be protected using `Collections.synchronizedXxx`wrapper factories

[...] concurrent API

- **Atomic data structures** — the `java.util.concurrent.atomic` package provides utilities that handle read-modify-write access to data, implementing the transactional integrity and visibility requirements (but not timing)
  - Some of these features also provide for very high scalability
- **Locks** — The `java.util.concurrent.locks`package provides the locks discussed earlier (and more)

Process Java collections concurrently and utilize parallel streams

- Parallel stream processing requires that the three rules (timing, transactions, and visibility) are not violated
- A pure functional programming approach ensures this:
  - Never mutate data, instead create new items derived from the old
  - Do not refer to non local data that is possibly being mutated
- Example problem, shared data being mutated by parallel threads:

```
long [] count = {0L};
long numSeen = ThreadLocalRandom.current().doubles(1_000_000_000_000L)
    .parallel()
    .peek(x -> count[0]++) // unsafe mutation!!
    .filter(x -> true) // needed to avoid shortcircuiting
    .count();
```

- In the example, if the call to parallel is commented out, `count[0]` and `numSeen` are both correct, but in parallel mode, the `count[0]` value is a fraction of the expected number

Using Java I/O API

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

Read and write console [...] using I/O streams

- Interaction with keyboard and screen in text mode can be done via standard in, out, and error channels, or via the "console"
- The standard io channels are available as `System.in`, `System.out`, and `System.err`
- The console is obtained using the `java.lang.System.console()` method
  - If a JVM does not have an associated console, this might return `null` — this seems to happen in IDEs
- System.in/out/err, and the console are all subject to redirection e.g. by command line operations
- System.in/out/err are "Stream" types, that is, they use 8 bit character encoding. Usage is typically simplified by wrapping these in `BufferedReader` (which provides a `readLine` method) and `PrintWriter`, which work with `char` data
- The console provides utilities for reading passwords, this feature disables echoing, and reads data into a `char` array rather than a `String`, so that the data can be erased from memory by writing over the actual storage

Read and write [...] file data using I/O streams

- File data is accessible using several classes:
- `FileInputStream`, `FileOutputStream`, [and `RandomAccessFile`, but this is not a "stream" in this sense]
  - 8 bit data, not necessarily textual
- `FileReader`, `FileWriter`
- Any of these types might be wrapped in a decorator, e.g. `BufferedReader`, `PrintWriter`
- Generally, identifying the target file will be done using a `Path`, but legacy modes allow using a `File` object (which describes file/pathname, not an actual file on disk) or a `String` directly
- Input might be wrapped in a `Scanner` for the convenience methods it offers

Serialize and de-serialize Java objects

- Serialization requires that a class implement the marker interface `java.io.Serializable`
- This interface has no methods (it's called a *marker interface* and predates annotations)
- When such an object is serialized, all *non-transient, instance* fields of this type are written to the output data
- When an object is deserialized:
  - Memory is allocated and zeroed
  - The zero argument constructor of the first non-serializable parent type is invoked—if no zero arg constructor is accessible deserialization fails, grand parents are initialized by delegation from the first non-serializable parent
  - No constructor invocation, nor instance initialization, occurs for the serializable types

- Serialization is performed by writing an object to an `ObjectOutputStream`

```
ObjectOutputStream oos = new ObjectOutputStream(fileOutput);
oos.writeObject(anObject);
```

- Deserialization is performed by reading through an `ObjectInputStream`and casting the result:

```
ObjectInputStream ois = new ObjectInputStream(fileInput);
Thing t1 = (Thing)ois.readObject();
```

Serialize and de-serialize Java objects

- If members are added or removed after serialization, deserialization will fail by default
  - This applies to fields or methods, whether static or instance, but not nested types
- However, if a `private static final long` field called `serialVersionUID` exists, and has the same value after the change, then deserialization will be permitted
  - Data for fields that have been removed in the newer version will simply be ignored
  - Added fields for which data is not in the serial form will be left at the default, zeroed, value
- If a class was created without the explicit serialVersionUID, and is about to be updated, the correct value can be calculated using the serialver command which actually outputs the exact field definition required to maintain compatibility:

```
$ serialver -classpath classes tryit.Thing
tryit.Thing:    private static final long serialVersionUID = -1054203193367308360L;
```

- If starting with the first version of a serializable type, `serialVersionUID` can simply start at 1, the fancy value is a hash of all the relevant elements and is calculated if no explicit value is found

Serialize and de-serialize Java objects

- Customization of serialization is possible in several ways, one is to add these methods which the system delegates to if they're present:

```
private void writeObject(java.io.ObjectOutputStream stream) throws IOException;
private void readObject(java.io.ObjectInputStream stream)
    throws IOException, ClassNotFoundException;
```

- The `writeObject` method is called to perform serialization, and the `readObject` method is called to perform deserialization. The infrastructure provides the streams to or from which data should be sent.
- The `ObjectXxxStream` types also provide methods to perform the default manner of serialization/deserialization, these can be used if desired, these are:

```
public void defaultWriteObject() throws IOException
public void defaultReadObject() throws IOException, ClassNotFoundException
```

- Additional methods in the `ObjectXxxStream` classes provide for writing data of other types, allowing additional, or replacement data to be handled.
- The format read by the read operation must be consistent with what was written
- These methods must be called from the `read`/`writeObject` methods of the object being serialized/deserialized or they will fail with exceptions

Serialize and de-serialize Java objects

- Example—This example adds three additional values representing the date at which the object was serialized, and prints the value out when deserializing it:

```
class S implements Serializable {
  String name;
  S(String n) {this.name = n;}
  private void writeObject(ObjectOutputStream oos) throws IOException {
    LocalDate ld = LocalDate.now();
    oos.writeInt(ld.getYear()); oos.writeInt(ld.getMonthValue());
    oos.defaultWriteObject();
  }
  private void readObject(ObjectInputStream ois)
      throws ClassNotFoundException, IOException {
    out.println(LocalDate.of(ois.readInt(), ois.readInt(), 1));
    ois.defaultReadObject();
  }
}
```

Construct [...] Path objects [...]

- Representation of path and filename is best done using the `java.nio.file.Path` class
  - Previously `java.io.File` and `String` have been used and interoperation is generally successful
- Obtain an instance of Path with the factory method `of` which takes a comma separated list of path elements (directories)
  - Class Paths provided factory methods called get which are superseded by Path
- `Path` provides methods: `endsWith`, `forEach`, `getFileName`, `getName`, `getNameCount`, `getParent`, `getRoot`, `isAbsolute`, `iterator`, `normalize`, `register`, `relativize`, `resolve`, `resolveSibling`, `spliterator`, `startsWith`, `subpath`, `toAbsolutePath`, `toFile`, `toRealPath`, `toUri`
  - and: `compareTo`, `equals`, `hashCode`, `toString`
  - and: `getFileSystem`

[...] traverse, create, read, and write Path objects and their properties using the java.nio.file API

- Manipulation of files can be done "traditional" ways with `Reader` and `Writer` types, but the class `java.nio.file.Files` has *many* utilities including methods for copying data, creating files and directories (including temporaries), checking attributes, handling data, traversing directory trees, etc.
- `readAllLines` -> `List<String>`
- `readString` -> `String`
- `lines` -> `Stream<String>`
- `list` -> `Stream<Path>`
- `find` -> `Stream<Path>` filtered
- `walk` -> `Stream<Path>`
- `newDirectoryStream` — `Iterable`, not `Stream`
- `exists`, `notExists`
- `isExecutable`, `isHidden`, `isReadable`, `isRegularFile`, `isSameFile`, `isSymbolicLink`, `isWritable`

Implementing Localization

- Implement localization using locales and resource bundles. Parse and format messages, dates, times, and numbers, including currency and percentage values.

Implement localization using locales and resource bundles [...]

- Java classloaders provide access to read-only "resources", allowing them to be distributed with the application binary, without regard to the distribution medium/mechanism/directory
- These are treated as being in packages, and should be alongside classfiles, whether those are in a directory tree, a jar file, a module, or loaded over some protocol like http
    - In builds, these files might need to be elsewhere—e.g. maven expects to find them in .../src/main/resources, and below that in a directory tree that mimics the package structure
- Assuming that a resource bundle file is in the directory com.myresources in the projects binaries, and is called `Labels.properties`, this code loads the bundle, and obtains the resource called name:

```
ResourceBundle bundle = ResourceBundle.getBundle("com.myresources.Labels");
String theName = bundle.getString("name");
```

- The bundle file `Labels.properties` contains:

```
name=My Wonderful Project
```

Implement localization using locales and resource bundles [...]

- If the resource bundle, or the individual resource, is not found, a MissingResourceException is thrown
- Resource bundles can be provided for specific locales
  - Locale can carry an extension representing language (lower case) then a regional variation/country (upper case)
  - Elements are separated with underscores,
- If the local is specific, e.g. pt-BR, the system searches for the best match, in this case, Brazilian-Portugese, Portuguese, and then the default file that carries no extension.
- Example files:

  `Labels.properties`

  `Labels_fr.properties` — French language

  `Labels_fr_CA.properties` — French language Canadian
- Local can be specified when searching for a resource bundle, or the system default will be used

[...] Parse and format messages, dates, times, and numbers, including currency and percentage values.

- Format messages with `format` and `printf` utilities in several classes, e.g.:

```
String phase = "Waxing Gibbous";
LocalDate ld = LocalDate.of(2025, Month.FEBRUARY, 7);
System.out.printf("On %TB %Te, %TY, the moon was %s\n", ld, ld, ld, phase);
```

- Documentation on format conversions is in the API for `java.util.Formatter`
- Other converters are provided too, e.g. `NumberFormat`, `DecimalFormat`, `DateFormat`
- `NumberFormat` can present a variety of formats, e.g. currencies, percentages, and are local sensitive:

```
NumberFormat nf = NumberFormat.getCurrencyInstance(Locale.GERMANY);
System.out.println(nf.format(123456)); // -> 123.456,00 €
```

- Wrappers can also parse—most parsing operations have pretty strict rules, failing with excess text before during or after the string. E.g. this fails with a `NumberFormatException`:

```
int cost = Integer.valueOf("1,234");
```

UNDERSTANDING > KNOWLEDGE

DANCING CLOUD SERVICES, LLC

Assume the following

- Missing package and import statements: If sample code do not include package or import statements, and the question does not explicitly refer to these missing statements, then assume that all sample code is in the same package, or import statements exist to support them.
- No file or directory path names for classes: If a question does not state the file names or directory locations of classes, then assume one of the following, whichever will enable the code to compile and run:
  - All classes are in one file
  - Each class is contained in a separate file, and all files are in one directory
- Unintended line breaks: Sample code might have unintended line breaks. If you see a line of code that looks like it has wrapped, and this creates a situation where the wrapping is significant (for example, a quoted String literal has wrapped), assume that the wrapping is an extension of the same line, and the line does not contain a hard carriage return that would cause a compilation failure.
- Code fragments: A code fragment is a small section of source code presented without its context. Assume that all necessary supporting code exists and that the supporting environment fully supports the correct compilation and execution of the code shown and its omitted environment.
- Descriptive comments: Take descriptive comments, such as "setter and getters go here," at face value. Assume that correct code exists, compiles, and runs successfully to create the described effect.

Candidates are also expected to:

- Understand the basics of Java Logging API.
- Use Annotations such as Override, FunctionalInterface, Deprecated, SuppressWarnings, and SafeVarargs.
- Use generics, including wildcards.

Resources

- Several courses on the O'Reilly Platform are relevant:
  Coming soon Java SE 21 Developer (1Z0-830)
  Java SE 17 Developer (1Z0-829) https://learning.oreilly.com/course/java-se-17/9780138194796/
  Up and Running with The Java Platform Module System (JPMS)
  https://learning.oreilly.com/videos/-/9780137475216/
  Functional Programming for Java LiveLessons https://learning.oreilly.com/videos/-/9780134778235/
  Modern Java Collections https://learning.oreilly.com/videos/-/9780134663524/
  Java Programming Essentials https://learning.oreilly.com/videos/-/9780137475438/
  Core Java Data Types https://learning.oreilly.com/videos/-/9780137368624/
- Simon's YouTube channel: https://www.youtube.com/@DancingCloudServices
- Contact Simon directly: +1 303 249 3613, simon@dancingcloudservices.com
- Connect with me on LinkedIn: https://www.linkedin.com/in/simonhgroberts/

- ***I'm planning a series of live classes in the 1 hour range, making deep dives into individual questions designed to simulate those in the exam, send me email, and/or connect with me and keep an eye out on LinkedIn if you might be interested***

UNDERSTANDING > KNOWLEDGE

DANCING CLOUD SERVICES, LLC