Chapter 1 - Welcome To Java

Basic commands

`javac Zoo.java`
`java Zoo`

Compiling

Compiling with wildcards

- `javac packaged/*.java`
- wildcard expansion only works for depth=1, you can't just run `javac *.java`
- classpath - the preferred way of setting classpath is with `-cp` flag

javac command

- defn: javac - read source files and compiles them into class files
- `-cp <classpath>` -
- `-classpath <classpath>` - Location of classes
- `--class-path <classpath>` -
- `-d <dir>` - Directory to place generated .class files

java command

- defn: java - command to start a Java application
- `-cp <classpath>` -
- `-classpath <classpath>` - Location of .class files that are already compiled
- `--class-path <classpath>` -

jar command

- java archive file
- similar to a zip file of mainly .class files
- `-c / --create` - creates a new JAR file
- `-v / --verbose` - Prints details of creation process
- `-f / --file <fileName>` - JAR filename
- `-C <directory>` - Directory containing files to be used to create the JAR

single-file source-code programs

- new variant: In Java 11, it's also possible to run a program with just `java Zoo.java` without compiling first
- only works for one-file programs
- single-file programs can only use core-package imports

Importing

Wildcard import

- `import java.util.*;`
- wildcard imports don't import child packages, fields, or methods. Just imports classes
- depth = 1
- invalid : `import java.nio.*.*;` (only one wildcard allowed)
- java.lang is automatically imported

Order of class files

- P.I.C.
- package declaration
- imports
- class declaration
- NOTE: Only one top level public class/interface/enum allowed

---

Chapter 2 - Java Building Blocks

Constructors

- The name of the constructors must match the name of the class (case-sensitive)
- No return type
- Tricky exam questions: reusing class name as a method name but including a return type. Compiles fine.
- `(new Date()).Date()`

Initializer Blocks

- defn: code block - code between braces
- defn: instance initializer - floating code blocks at the member level.
- instance and static initializer blocks

Order of initialization

- See chapter 8. Initializer blocks run before constructors always.

Data Types

Primitive Types

- Java has 8 built in data types, called primitives.
- All Java objects are just complex collection of primitive data types

Primitive sizes

- (not required for exam, just relative size)

- boolean - (depends on VM)(small)
- byte - 8-bit integral value - [-128 to 127]
- short - 16-bit integral value - [-32_768 to 32,767]
- int - 32-bit integral value - [-2_147_483_648 to 2_147_483_647]
- long - 64-bit integral value - [-9_223_372_036_854_775_808 to 9_223_372_036_854_775_807]
- float - 32-bit floating-point value - [6 to 7 decimal digits]
- double - 64-bit floating-point value - [15 decimal digits]
- char - 16-bit Unicode value - [0 to 65_535]
- ASCII values range from 0 to 255, but char ranges from \u0000 (0) to \uffff (65,535)
- special relationship exists between `char` and `short` since they are both 16-bit integral values. `short` is signed but `char` is unsigned. No such thing as a negative char.
- each numeric type uses twice as many bits as the smaller similar type
- All numeric types are "signed"(include negative/"parity") in Java
- Java stored floating-point values in scientific notation. `a x 10**b` up to # of decimal precision. Not accurate.
- Java allocates memory based on primitive type. (int -> 32 bits). primitive references aren't the same size (but obj are)

Literals

- defn: literal - when a number is present in the code
- literals without a suffix are assumed to be int if number, double if decimal.
- `long max = 3_123_456_789` won't compile because it exceeds int size. Adding `L` suffix fixes
- `byte b = 200;` won't compile because it exceeds byte size.
- literal is int, but then down casted to left side without an explicit case if it fits
- Three bases
    1. Octal - prefix = `0` , example = `017`
    2. Hexadecimal - prefix = `0x` or `0X` , example = `0xFF` (NOTE: Hex is case-insensitive, [0-9, a-f/A-F])
    3. Binary - prefix = `0b` or `0B` , example = `0b1011100`

Underscores

- Underscores are allowed anywhere except:
    1. Beginning/End of literal
    2. Before/After decimal
    3. Before/After prefix (`0b`)

Reference Type

- defn: reference type - type that refers to an object
- references do not hold the value of the object
- references "point" to an object by storing the memory address where the object is located.
- defn: pointer - an address in memory
- An object in memory can only be accessed via a reference, not directly
- references can be assigned null, meaning they do not currently refer to an object
- primitive types can't be assigned null

Variables

- defn: variable - name for a piece of memory that stores data

- defn: variable declaration - stating the variable type
- defn: variable initialization - giving a variable a value using assignment operator (=)
- variables are declared to be a certain type `int e;`
- variables are initialized to a certain value `e = 10;`
- variables can be declared and initialized in one line `int e = 10;`
- three types of variables: 1. local 2. instance 3. static

Identifiers

- defn: identifier - name of a variable, method, class, interface, or package
- identifier charset = `[0-9, A-Z, a-z, _, $]`
- Identifier rules:
  1. Can't start with number
  2. Can't be equal to `_`
  3. Can't be a Java reserved word (case-sensitive)

Multi-declaration (Declaring Multiple Variables)

- Declaring multiple variables in one statement
- `int i, j, k = 10` // i j k all declared, but only k initialized
- Rules:
  1. Only one type allowed (duplicate/redundant types also not allowed!)
  2. Comma separated, semicolon end.
  3. Some values can be initialized, not required
  4. var not allowed

Local Variables

- defn: local variable - a variable defined within a method/constructor/initializer_block
- local variables do not have a default value and don't have to be initialized ever
- local variables must be initialized before used
- "x might not have been initialized" compiler error thrown if used before initialized
- compiler is smart enough to know if initialized in "branching" (if, if else, else)

Method/Constructor Parameters

- special type of local variables that have been pre-initialized

Instance/Class Variables

- defn: instance variable - aka field, is a value defined within a specific instance of an object.
- defn: class variable - static field, is a value defined on the class level and shared among all instances.
- instance and class variables do not have to be initialized
- instance and class variables are initialized to default values (except when final!)
- final instance/class variables must be initialized before constructor finishes or else compiler error
- default values = {boolean -> false, byte+short+int+long -> 0, float+double -> 0.0, char -> `\u0000`, obj -> null}

Var

- defn: var = local variable type inference

- Golden Rule: var can only be used when the compiler can infer the type from the context.
- `var e = 0;` var infers raw literals as int or double
- Rules for Var
    1. var can't be used as instance/static fields, method/constructor params
    2. var must be declared and initialized in the same statement
    3. var type can't change (but value can)
    4. var can't be initialized to null (without a cast), but can be reassigned to null.
    5. var not allowed in multi-declaration
    6. var is not a reserved word, but it is a reserved type!(can't declare class/interface/enum with name = var)
    7. var in lambda parameter list only if all parameters are var type (no mixing)

Scope

- local variables can never have a scope larger than the method they are defined in.

Garbage Collection

- defn: heap - aka free store - represents a large pool of unused memory allocated to Java
- defn: garbage collection - the process of automatically freeing memory on the heap by deleting objects that are no longer reachable.
- eligible for garbage collection != garbage collected. JVM does what it wants
- `System.gc()` merely suggests that the JVM kicks off garbage collection. No guarantee
- Just before `OutOfMemoryError` is thrown, JVM will attempt to garbage collect.
- GarbageCollection is the process of deleting unused objects, NOT REFERENCES.
- Reference vs Object itself

Chapter 3 - Operators

- defn: java operator - special symbol that can be applied to a set of operands and that returns a result
- defn: operand - the value or variable the operator is being applied to.

Types of Operators

- unary
- binary
- ternary
- arithmetic
- logical
- java operators are sometimes right-to-left(e.g. assignment), but mostly left-to-right evaluated.

Operator Precedence

- defn: operator precedence - order in which operators are evaluated
- assignment operator has the lowest order of precedence
- parentheses override the order of precedence

- ORDER
  0. UMARE LTA
     1. Unary (in order: ++a, a++, (-,!,~,+,(cast)))
     2. Multiplication/division/mod (*, / , %)
     3. Add/subtract (+,-)
     4. Relation (<,>,<=,>=, instanceof)
     5. Equality (==,!=)
     6. Logical Operators (in order: (&,^,!) , (&&,||)) // regular before short circuit
     7. Ternary (a ? b : c)
     8. Assignment (=,+=,-=,*=)

Unary Operators

- one operand
- pre increment (returns a+1 and stores a+1)
- post increment (returns a and stores a+1)
- increment and decrement operators will overflow, but stay the same type(126,127,-128,...)
- logical compliment (!) not to be confused with number negation (-)

Binary Operators

- two operands
- Arithmetic operators (also include unary operators)
- balanced parentheses
- defn : modulus - aka remainder operator - remainder left after dividing value.
- Integer division edge case: integer division returns the floor value of the division.

Numeric Promotion

-
- Rules:
  1. If two different sizes, smaller data type is widened
  2. bye,short,int automatically promoted to int when used in binary arithmetic operators.
  3. Promotions are retained after evaluation

Assigning Values

- defn: assignment operator - binary operator that modifies, or assigns, a value to a variable
- java will automatically promote from smaller to larger data types(widening)
- java requires cast operator to convert larger to smaller data type (narrowing)
- defn: casting: unary operator where one data type is explicitly interpreted as another data type.
- Unary means you have to be extra careful with parentheses around casting
- The assignment operator returns the value assigned to the left variable.

Overflow and Underflow

- defn: overflow - when a number is so large that it will no longer fit, so the system "wraps around" to the lowest negative value.
- defn: underflow - when a number is so small that it no longer fits, wraps back to positive MAX
- 2147483647 + 1 == -2147483648 // true

Compound Assignment Operators

- int a += b;
- equivalent to int a = (int)(a + b);
- compound assignment operator implicitly casts the right side back to left side, even if overflow.

Assignment Operator Return value

- The result of an assignment operator is an expression in and of itself.
- Java assignment operator is python walrus operator
- The result assigned to the left operand is also returned (a = 10 also returns 10)
- Exam Trick: boolean questions `(healthy = false)` // both assignment and returning the value false;

Comparing Values

- equality operator
- "reference equality operator"
- Three types of equality operators:
    1. Comparing two number or character primitives. Upcasted for comparison. `5 == 5.0` // true
    2. Comparing two boolean
    3. Comparing two object references (reference equality)
- All other uses of equality operator throw an exception
- null == null // true, same "location"
- reference equality doesn't have to be the same type ("bob".equals(bossObj) compiles but is false))
- reference equality has an `instanceof` check which immediately returns false if the objects are different types

Relation Operators

- defn: compare two expressions and return a boolean value
- <, >, <=, >=, instanceof
- less than and greater than (or equal) only applies to numeric values.
- less than and greater than follow same Numeric Promotion (to int!) rules

instanceof Operator

- `a instanceof b`
- defn: instanceof - return true if the reference that "a" points to is an instance of a class, subclass, or class that implements a particular interface "b"
- best practice to check instanceof before casting to a narrower type
- instanceof throws compiler error if attempting to compare incompatible types (A !<= B and B !<= A)
- instanceof struggles with interface instanceof. If class is final, instanceof can catch bad interface type comparisons, else not.
- `null instanceof Object` is always false
- `anything instanceof Object` is always true
- `anything instanceof null` // compiler error
- instanceof supports arrays
    - `new int[]{1} instanceof int[]` // true
    - `new Integer[]{1} instanceof Number[]` // true
    - `new Boss[]{1} instanceof Employee[]` // true

- instanceof can't be used with primitives

Logical Operators

- bitwise operators not on the exam
- ^ is XOR operator (a && !b || !a && b) (different values for both)
- short circuit operators
  - right operand may not evaluate if the left side is deterministic
- "underperformed side affect" because of short circuiting
- short circuit is good at preventing NPE `a != null && a.length() > 0`

Ternary

- `a ? b : c`
- just a condensed form of if/else.
- conditional ? expression_1 : expression_2
- `int e = a ? b : c`
- if used with assignment operator, both `b` and `c` must evaluate to covariant subtype of `e`, even if always true `int animal = (stripes < 9) ? 3 : "Horse"` // does not compile
- if used alone, no requirement on type `System.out.println((stripes < 9) ? 3 : "Horse")` // fine
- ternary can also have "underperformed side affect". false condition is never evaluated if true is true.

Chapter 4 - Making Decisions (Control Flow)

- defn: statement - a complete unit of execution, terminated with a semicolon
- defn: control flow statements - statements to break up the flow of executing by using decision making
- defn: block - zero or more java statements between balanced braces
- the target of a decision-making statement can be either a block or a single statement. (excluding try)
- Exam Trick: Using indentation to fool single statement blocks
- `goto` is a reserved keyword, but unused in java

if statement

- parentheses always required
- if statements must use conditionals(evaluate to boolean). `if (1)` compiler error, no truthy-falsy

switch statement

- defn: switch statement - decision-making structure where a single value is evaluated and branched to matching case.
- if no default option is available and no case matches, nothing happens
- parenthesis required around switch expression
- `case 15:` // colon, not semicolon after case
- braces required around entire switch
- `switch (month) {}` // valid switch statement. case statements are optional

- `switch () {}` // compiler error, "expression expected"
- allowed switch values:
  - byte/Byte
  - short/Short
  - int/Integer
  - char/Character
  - String
  - Enum
  - var (if the type resolves to one of above)
- boolean/long/float/double notable not allowed. Either too few options or too many
- `break;` or //fall through
- default doesn't need to be the last branch
- a case can fall through into a default below (and a default can fall through into a case below it)
- Exam Trick: missing break statements causing fall-through.
- !!!! case statements must be compile-time constants
- defn: compile-time constant : constants and trivial combinations of constants (+,concat, etc.)
- Constant expressions of type String are always "interned".
- switch statements support numeric promotion (upcast and downcast to fit switch type)

While loops

- parentheses required around conditional
- `while(conditional) {}`
- `do {} while (conditional);`
- semicolon required at the end of a do-while

For loops

- for loop questions consume a lot of time on the exam
- `for (initialization; conditional; updateStatement) {}`
- parentheses required
- semicolon separated
- variables are only in scope inside the for loop
- forward and backward loops important for the exam
- `for(;;)` // valid
- Multi-term for loops
  - `for (int x = 0, y = 10; x < 20 && y >= 0, x++, y--) {}`
  - x,y,... must be the same type in the initialization
- You can't declare an already declared variable
- var is supported in initialization `for (var i = 0;;) {}`

For each

- `for (datatype instance : collection) {}`
- colon separated
- parentheses required
- Either array, or must implement `Iterable<T>`
- Map, String, StringBuilder not supported.
- var is supported `for (var s : args) {}`

- You can't modify Collection while looping, `ConcurrentModificationException`

Branching Statements

- defn: label - optional pointer to the head of a statement
- labels can be added almost anywhere, but almost always before loops
- `OUTER_LOOP:`
- labels follow same rules as identifiers. Usually SCREAMING_SNAKE case
- `break;,break OUTER_LOOP;,continue;,continue OUTER_LOOP;,return;,return val;`
- any code directly after break,continue,return is "Unreachable" and will throw compiler error
- dead code != unreachable code

Chapter 5 - Core Java APIs

- String, StringBuilder, arrays [], Arrays, ArrayList, Math

Strings

String Concatenation

- `String::concat` == `+` == `s += "a"`

3 Concatenation Rules

1. If both operands are numeric, `+` means numeric addition
2. If either operand is a String, `+` means String concatenation
3. The expression is evaluated left to right
   - Example: `3 + 5 + "c";` // "8c"

String Methods

- int length()
- char charAt(int index)
- int indexOf(int ch) // note: "int ch" char promoted to int
- int indexOf(int ch, int fromIndex)
- int indexOf(String str)
- int indexOf(String str, int fromIndex) // offset "fromIndex"
- String substring(int beginIndex)
- String substring(int beginIndex, int endIndex)
- String toLowerCase()
- String toUpperCase()
- boolean equals(Object obj)
- boolean equalsIgnoreCase(String str)
- boolean startsWith(String prefix)
- boolean endsWith(String suffix)

- String replace(char oldChar, char newChar) - replaces all occurrences (multiple occurrences)
- String replace(CharSequence target, CharSequence replacement)
- boolean contains(CharSequence charSeq)
- String strip() - trim() plus unicodes.
- String stripLeading()
- String stripTrailing()
- String trim() - simple version of strip
- String intern() - returns the reference from the string pool (or otherwise place it there and return reference)

StringBuilder

Constructors

- new StringBuilder()
- new StringBuilder("animal");
- new StringBuilder(10); // capacity

StringBuilder Methods

- "chaining StringBuilder methods" because most methods returns a reference to the original object. Builder pattern.
- char charAt()
- int indexOf()
- int length()
- String substring()
- StringBuilder append(String str) - appends and return a reference to the current StringBuilder
- StringBuilder insert(int offset, String str) - adds characters to the StringBuilder at the requested index
- StringBuilder delete(int startIndex, int endIndex)
- StringBuilder deleteCharAt(int index)
- StringBuilder replace(int startIndex, int endIndex, String newString)
- StringBuilder reverse()
- String toString()

Equality Operator == vs .equals()

- reference equality (==) - same pointer
- `.equals()` is inherited from Object class. Some classes override the method(e.g. String) but others do not (array, StringBuilder)

String Pool

- `"a" != new String("a")` // false
- `"a" == new String("a").intern()` // true
- Constant expressions of type String are always "interned".
- `"a" + "b"` is interned automatically since it's a constant expression

Java Arrays

Java Arrays methods

- arr1.equals(arr2) // BAD!, reference equality/same reference
- Arrays.equals(arr1, arr2)
- Arrays.deepEquals(arr1, arr2)
- Arrays.sort(myArr)
- Arrays.sort(myArr, `Comparator<E>`)
- Arrays.binarySearch(myArr, val)
    - if .contains(), return index; else, return -index -1;
    - Array must be sorted before binarySearching
- Arrays.compare(arr1, arr2)
    - compare is effectively compareTo(). positive, zero when equal, negative.
- Arrays.mismatch(arr1, arr2)
    - return first index that is different , else (if equal) return -1;

ArrayList

- Generics vs raw type

Creating an ArrayList

- `new ArrayList<>();`
- `new ArrayList<>(10);` // initial capacity
- `new ArrayList<>(list2);` // Construct from Collection

ArrayList Methods

- boolean add(E element)
- void add(int index, E element) - effectively an insert statement. ArrayList doesn't define a .insert() method
- boolean remove(Object obj)
- E remove(int index)
- E set(int index, E newElement)
- boolean isEmpty()
- int size()
- void clear() - discard all elements of the ArrayList
- boolean contains(E element)
- boolean equals(E element) - ArrayList implements a custom .equals() method. Same elements, same order.

ArrayList --> Array

- `list.toArray()` // PROBLEM: returns Object[]
- `list.toArray(String[0])` // Safe: returns String[]. Simply a copy, no linking

Array --> ArrayList

- `Arrays.asList(arr1)` // PROBLEM: Fixed size. Linked to the original array.
- `new ArrayList<>(Arrays.asList(arr1))` // Safe: passing List as argument
- `List.of(arr1)` // PROBLEM: Immutable
- `(ArrayList) Arrays.stream(nums).boxed().collect(Collectors.toList());` // int[]

- `Arrays.stream(nums).boxed().collect(Collectors.toCollection(ArrayList::new));`

Collections Utility Class

- `Collections.sort(coll)`
- `Collections.binarySearch(coll, ele)`
- `Collections.min(coll)`
- `Collections.max(coll)`

Set

- defn: Unordered collection that does not allow duplicate elements
- HashSet vs TreeSet

Map

Map Methods

- V get(Object key)
- V getOrDefault(Object key, V other)
- V put(K key, V value)
- V remove(Object key)
- boolean containsKey(Object key)
- boolean containsValue(Object value)
- `Set<K>` keySet()
- `Collection<V>` values()

Math API

- double min(double a, double b)
- float min(float a, float b)
- int min(int a, int b)
- long min(long a, long b)
- double max(double a, double b)
- float max(float a, float b)
- int max(int a, int b)
- long max(long a, long b)
- long round(double num) - traditional round (.5 ? up : down)
- int round(float num)
- double pow(double number, double exponent)
- double random() - random double 0<= x < 1

Chapter 6 - Lambdas and Functional Interfaces

- Doesn't cover method references, see chapter 14

Lambda expressions

- aka lambdas
- defn: Lambda expression - let you express instances of single-method classes more compactly.
- Lambdas are like a method that you can pass as if it were a variable.
- Lambdas are an example of Deferred execution - code is specified now, but will run later.
- Lambdas work with interfaces that have only one abstract method(functional interfaces).

Syntax

- Simplest form `a -> a.canHop()`
- lambda param types can be omitted
- lambda parameter types are inferred from the context(functional interface params).
- lambda return type(s) are inferred from the context(functional interface return type).
- rules:
    - Parentheses can be omitted only if there is a single parameter AND no type
    - Braces can be omitted when body has only a single statement (same with if/else)
    - return is omitted IFF braces are omitted IFF semicolon is omitted
    - lambda parameter can be var, but if one var, all params must be var.
- Just like methods, when returning void, return is optional.
- edge case: () -> {} is valid syntax.

Functional Interface

- defn: Functional Interface - Interface with only 1 abstract method
- @FunctionalInterface annotation is similar type safety as @Override. Optional, but best practice

Four Fundamental Functional interfaces

- java.util.function
- Predicate - T -> boolean
    - `.test()`
- Consumer T -> {}
    - `.accept()`
- Supplier {} -> T
    - `.get()`
- Comparator TxT -> int
    - `.compare()`
    - `(a,b) -> a.compareTo(b)` is a Lambda that results a list in ascending order {1,2,3}

Lambda Rules

- defn: effectively final - values does not change after it is set. Can be marked final and still compile.
- effectively final means no reassignment. Effectively final != immutable. Effectively final variables can be altered.
- Variables can appear in three places in lambdas: 1. Params, 2. Local variables defined inside Lambda body, 3. Variables already defined outside of lambda body.

1. param variables
    - var can be used as type for lambda param type
    - param can be defined final inside of lambda parentheses
2. local variables defined inside lambda body
    - can't redeclare variables already defined in outer scope
3. Variables already defined outside of lambda body
    - lambda can access instance variables, method parameters, or even other local variables
    - method params/local variables must be effectively final or final
    - instance/static variables don't have to be effectively final or final
    - lambda parameters don't have to be effectively final or final

- instance/static variables don't have to be effectively final because when the temporary java class file is created for the Lambda, the instance/static variables are copied over.

Example

- `List.removeIf(Predicate<T>)`
- `Collections.sort(Comparator<T>)`
- `Collection.forEach(Consumer<T>)`

Chapter 7 - Methods and Encapsulation

Methods

- defn: method declaration - specifics all the info need to call the method. Everything.
- defn: method signature - method name + ordered parameter list
- access modifier vs optional specifier

Access Modifiers

- public
- protected
- default
- private

Optional Specifiers

- static
- abstract
- final
- synchronized
- native (not on exam)
- strictfp (not on exam)
- transient

Order of method declaration

- return type HAS TO appear right before method name
- formal type parameter (Generics) HAS TO appear right before return type `<T> T`
- [access_mods + optional_specifiers] + type parameter + return_type + methodName(params) throws ExceptionList {}
- order of access modifiers and optional specifiers doesn't matter, but bad practice
  - `final public void run(){}` this is valid

Return type

- return type is always required
- if void, return statement is optional in body. `return;`
- All branches of logic have to return a value. Compiler error if else.

Method Names

- Method names are "identifiers" as well.
- Method Names follow the same rules as identifiers.

Parameter list

- comma separated
- No multi-declaration
- No var

Exception list

- comma separated
- only one Throws
- Don't have to be exclusive at all (Intellij grays it out as unnecessary)
- You can `throws` Checked or unChecked, but throws unchecked is pointless
- `throws IOException, FileNotFoundException, TimeoutException, IllegalStateException`

Varargs

- must be last parameter
- only one vararg allowed (since it must be last)
- varargs can be passed as array (new int[]{1,2,3}) or as values (1, 2, 3) (e.g. `Stream.of(V...)`)
- vararg param can be left out entirely. sending two params instead of 3. Java will send empty array
- null can be passed as vararg value

Access modifiers

- public vs protected vs default vs private
- access modifiers for class determine where the class reference can be used
- access modifiers for methods determine where the method can be used, regardless of the reference.
- protected edge cases:
  - Even if a child class inherits, if you reference a protected method from a parent reference, no access allowed.

Static keyword

- (Static Context) A static member cannot call an instance member (without referencing an instance of the class)
- Tricksy: The compiler checks the type of the reference and uses that instead of the object.
- instance.staticMethod() - instance of an object to call a static member.
- ((Koala) null).staticMethod() - compiles fine.

Misc modifier/specifier

- final != immutable
- final instance/static fields must always be initialized by the time the constructor finishes.
- final instance/static fields must be set "exactly once" (not more, not less)
- final local variables don't need to be initialized if never used.
- In general, avoid static&instance initializers, just use constructors
- defn: static import - used to import static members (as opposed to classes)
- `import static java.util.Arrays.asList;`

Passing Data among Methods

- defn pass-by-value - a copy of the variable is made and the method receives the copy.

Overloaded

- defn: Overloaded - methods have the same name but different signature
- Overloaded methods have very few rules. Same method name, different parameters.
- Edge case:
  - Java treats varargs as an array, so the signature is the same
  - void fly(int[] e)
  - void fly(int... e) // compiler error
- Edge case:
  - void fly(int e)
  - void fly(Integer e) // fine, promotion/boxing wont ever be called.
- Edge case:
  - `void walk(List<String> strings) {}`
  - `void walk(List<Integers> integs) {}` // Type erasure with generics make these two the same signature

Method resolution

1. Exact
2. Widening
3. Autoboxing
4. Varargs

- NOTE: No combinations, only one conversion

Chapter 8 - Class Design

Inheritance

- Defn: Inheritance - the process by which a subclass automatically includes any public or protected members of the parent
- Inheritance is transitive
- private members are never inherited.
- Java supports single inheritance, not multiple inheritance(diamond problem)
- Java supports multiple-levels of inheritance (class chains)
- Cosmic SuperClass - All classes inherit from Object

Class modifiers

- Two access modifiers available for Top-Level types (classes, interface, enum)
    - public
    - default
- Two optional specifiers for Top Level types
    - final
    - abstract
    - strictfp(not on the exam)
    - native (not on the exam)

this

- defn: this - refers to the current instance of the class.
- this() calls another constructor of the same class
- this() must be the first statement in the constructor (excluding comments)
- by effect, there can only be one this() call per constructor
- this() can't cause an infinite recursion. Compiler error.
- `this` can't be accessed from a static context

this vs this()

- this refers to the current instance of a class
- this() refers to a constructor of the current class

super

- super is a reference to the parent class instance(not class!)
- Only members defined at the parent level are accessible via super
- `super` can't be accessed from a static context
- you can't call super to refer to an interface that this class implements.

super()

- calls parent class constructor
- must be first line of constructor if used
- super() always refers to the most direct parent (one level up)

super vs super()

- super is used to reference parent class members

- super() calls parent constructor

Constructors

- Defn: Constructor - a special method with the same name as the class but has no return type.
- constructors can be private
- no var params
- constructor overloading - multiple constructors with different signatures.
- The first line of every constructor is either
    1. explicit call to this()
    2. explicit call to super()
    3. implicit call to super() // compiler enhancement

Default no-arg Constructor

- `public Rabbit() {}`
- Every class has a constructor. If no constructor is coded, default no-arg constructor is generated at compile time.
- default construct - aka default no-arg constructors
- default no arg constructors only created if no other constructors exist
- if Parent class has no no-arg constructor, all child classes must have constructors. Compiler can't create default no-arg since super() isn't defined.

Compiler Enhancement

- Java compiler automatically inserts a call to the no arg super() if you don't call this() or super() yourself
- creates default no-arg constructor if you don't create any constructors
- If a parent does not define a no-arg constructor but does define 1 arg constructor, child class can't use default no-arg constructor since it would implicitly call super() which doesn't exist.

Constructors and final Fields

- local final variables don't have to be assigned. Not defaulted either.
- final instance/static variables MUST be assigned a value exactly once. Not defaulted.
- After initializer and constructors finish, final instance/static fields must have been assigned a value.

Order of initialization

- NOTE: A class may never be loaded if it is not used
- Class is initialized at most once, right before it is used.
- Initializing a class is different than initializing an instance
- Order:
    1. Recursively initialize parent class
    2. Static variable and static initializers in order
    3. Instance variables and instance initializers in order
    4. Constructor in top-down order (resolving stack naturally)
- Edge case:
    - The class containing main is initialized before the main method is executed just like any other method

Overriding

- Defn : Overriding - when a subclass declares a new implementation for an inherited method with the same signature.
- this and super allow you to select between the current and parent versions of a method.
- rules:
    1. same signature
    2. access modifier same or less restrictive
    3. same or more specific checked exception
    4. covariant return type
- covariant = same or subtype
- covariant iff you can assign the type from smaller to larger type
- Generic overriding
    - signature must be exactly the same, including the generic type
    - covariant return type
- You can't override a private method since private methods aren't inherited. It's just a new method.

Hiding Methods

- defn: hidden method = when a child class defines a static method with the same name and signature as an inherited static method defined in a parent class.
- Same 4 override rules apply.
- new rule: child static iff parent static

Hiding Variables

- defn: hidden variable = when a child class defines a variable with the same name as an inherited variable.
- No such thing as variable overriding
- Two distinct copies of the variable are created.
- hiding a variable replaces the member only if a child reference is used
- The type of the reference determines which variable is used.

Polymorphism

- Defn: Polymorphism - A Java object may be accessed using a reference that is same or supertype (without a case)
- a Java object may be accessed using a reference of
    - same type
    - super class
    - interface
- Polymorphic references don't create additional objects. Same object, just different references.
- Only the methods and variables available to the reference type are callable, regardless of the underlying object
- The type of the object determines which properties EXIST
- The type of the reference determines which properties are accessible.
- Overriding a method replaces the parent method on all reference variables
- hiding a method or variable replaces the member ONLY if a child reference type is used.

Hiding vs Polymorphism

- static method hiding is like the opposite of polymorphism
    - parent static -> child static = hiding
    - parent static -> child instance = compiler error

- parent instance -> child static = compiler error
- parent instance -> child instance = overriding

Variables

- Variables can't be overridden
- Variables can be hidden though
- Child classes can hide parent instance variables with child static variables
- Child classes can hide parent static variables with child instance variables
- defn: Hidden Variable - Child class defined variables with the same name as inherited variable.
- When hiding, two distinct copies of the variable exist
- The type of the reference variable determines which variable is used

Explicit vs Implicit Cast

- `Employee frank = new Boss(); // Implicit Cast`
- `Boss jill = (Boss) empObject; // Explicit Cast`
- If the current reference is a subtype, you do not need a cast operator.
- defn: cast conversion - implicit cast to a subtype
- Rules
  1. Casting a reference from a child to a parent doesn't require an explicit cast
  2. Casting a reference from a parent to a child does require an explicit cast
  3. The compiler disallows casts to an unrelated type
  4. The compiler can't check for interface casts at compile-time (unless the class is final)

Polymorphism vs hiding

- Polymorphism states that when you override a method, you replace all calls to it, even those defined in the parent class
- unlike method overriding, method hiding is very sensitive to the reference type and location where the member is being used.
- changing the reference type may grant access to new members, but the rule of polymorphism states that it doesn't change the logic of existing members

---

Chapter 9 - Advanced Class Design

Abstract Classes

- defn: Abstract Class - a class that cannot be instantiated (and may have abstract methods).
- Abstract classes can include variables, methods, inner classes, and even constructors.
- defn: Abstract Method - a method that does not have a body.
- implementing an abstract method is a form of @Override
- Abstract classes are not required to have any abstract methods
- if abstract method exists, then class must be Abstract

- abstract keyword is mandatory when (not implicit) (empty body iff abstract keyword) (abstract keyword is implicit for interfaces, but not for abstract classes)
- defn: Concrete Class - a class that is non-abstract
- The first concrete subclass that extends an abstract class is required to implement all inherited abstract methods
- abstract classes can extend concrete classes and vice versa and abstract classes.

Rules

- Abstract class constructors can call abstract methods
- Abstract methods can't provide an implementation
- Abstract classes can't be final
- Abstract methods can't be final
- Abstract methods can't be private
- Abstract methods can't be static
- Abstract methods must still follow @Override rules
- (NOTE: private + final is fine)

Interfaces

- defn: Interfaces: an interface is an abstract data type that declares a list of abstract methods that any implementing class must implement.
- Interfaces can also include variables, but they must be constants. (public static final)
- abstract methods are implicitly public
- variables are implicitly public
- variables are implicitly static
- variables are implicitly final
- interfaces are implicitly abstract
- interfaces can't be final since implicitly abstract
- interfaces can extend multiple other interfaces
- interface methods w/o a body are implicitly abstract and final
- Interfaces extend, not implement, other interfaces
- Interfaces can extend multiple other interfaces at once
- Interfaces can be thought of as being like specialized abstract classes.

Implicit modifiers

- implicit means the compiler will automatically add the modifier
- Okay to add implicit modifiers manually instead of letting compiler do it automatically
- Conflicting modifiers will throw compiler error

default for interfaces

- Since methods are implicitly public, if omit the access modifier, it will be made public (not default!)
- lack of access modifier for classes = default
- lack of access modifier for interfaces = public

Interfaces vs Abstract Classes

- Interfaces have implicit modifiers, Abstract Classes do not
- extends vs implements

- "abstract" keyword is required for Abstract Classes and Abstract methods in Abstract classes
- "abstract" keyword is optional for Interfaces and Interface methods(implicit)
- Abstract classes isn't forced into public implicit modifier for all methods

Name Collisions with Interfaces

- If class implements two interfaces with
    - same signature -> fine. compiler understands
    - different signature -> fine. This is just overloading
    - same signature & different return type -> Depends. Must follow @Override rules. Covariant else compiler error

Casting Interfaces

- Casting to interface is fine.
- Compiler struggles with validating casts at compile time.
- Compiler allows bad interface casts, but ClassCastException thrown a run-time
- The compiler does not allow a cast from an interface reference to an object reference if the object type is final and does not implement the interface.
- The compiler can check for unrelated interfaces if the reference is a class that is marked final.
- instanceof
    - The compiler will only report an unrelated type error for an instanceof operation with an interface on the right side if the reference on the left side is a final class that does not inherit the interface.

Inner Classes

- inner aka nested class aka member inner class
- defn: A class defined at the member level of another class
- private inner interfaces are possible
- inner classes can have any access modifier(public/protected/default/private)
- No static members in inner classes

Chapter 10 - Exceptions

Exceptions

- Three types of exceptions: Checked, Unchecked, Errors
- Exceptions are objects
- Exceptions can be stored in variables
- Don't forget the "new" in the `throw new Exception();` statement
- errors should not be recovered from (e.g. OutOfMemoryError)
- Throwable is the parent of all Errors&Exceptions

Checked Exception

- defn: Checked Exception: an exception that must be handled or declared

- All Checked Exceptions inherit Exception but not RuntimeException.
- An exception that "might reasonably be expected to recover from" - Oracle
- "Handle or Declare" rule
    - Checked exceptions must be Handled (try-catch) or declared (throws in method declaration)

Unchecked Exception

- runtime != Runtime
- Unchecked a.k.a. runtime exception, but notably also includes Errors subclasses, not just RuntimeException subclasses.
- defn: Unchecked Exception - an exception that doesn't need to be handled or declared
- It is optional to handle/declare. You can throw & throws & catch runtime Exceptions
- Adding `throws IllegalStateException;` or any runtime exception to a method declaration doesn't do anything. "documentation"
- Technically errors are a subset of Unchecked Exceptions
- declaring runtime exceptions is redundant

Errors

- Defn: a form of unchecked exception that extends the Error class
- They are thrown by the JVM
- should not be handled, declared, or thrown
- E.g. StackOverflowError OutOfMemoryError
- ExceptionInInitializerError- thrown when a static initializer throws an exception and doesn't handle it
- NoClassDefFoundError - thrown when a class that the code uses is available at compile time but not run time (could have been deleted)

Try statements

- curly braces are always required, even for single statements in a try-statement
- either (catch or finally) blocks required. try statement can't be alone (unlike try-with-resource)
- only one catch block can execute
- unreachable exception catch
- catch has to come before finally if both appear
- declaring/"throws" an unused exception isn't considered unreachable code.
- catch blocks that catch unrelated Checked exceptions are unreachable code and do not compile
- catch blocks for unchecked exceptions are always "reachable", even if they have nothing to do with the code.
- "swallowing"/suppressing/ignoring exceptions is bad practice. At least log something.

Chaining catch blocks

- at most one catch block will run
- each catch block much be "reachable"
- more specific exceptions first
- the exception variable is only in scope for that specific catch block
- Unrelated checked exceptions (not thrown by any of the statements) are unreachable and thus compiler error

Multi-catch block

- `} catch( ArrayIndexOutOfBoundsException | NumberFormatException e) {`
- allows multiple exception types to be caught by the same catch block
- only one variable (usually `e` or `ex`)
- pipe separated `|`
- No redundant catches regardless of order.
- Must be absolutely exclusive, no subtypes, regardless of order
- you can chain multi-catch and "single-catch" blocks together

finally block

- finally block always executes, regardless of Exceptions or returns
- catch is optional if finally block exists. One or the other or both
- finally can't come before catch
- if finally block has a return statement, this return statement will override all others. Finally always executes even if another block calls return. Finally "interrupts" the other return statements.
- finally blocks can also themselves be interrupted by exceptions
- finally blocks always execute, but aren't guaranteed to finish
- System.exit() is the only thing that will stop the finally block from executing. Stops everything, hard stop.

try-with-resource

- aka automatic resource management
- compiler replaces (try-with-resources) -> (try + implicit finally) block under the hood.
- compiler implicitly creates a finally block to close resource
- try-with-resource can have a finally block. It runs after implicit finally block
- when multiple resources are opened, they are closed in REVERSE ORDER.
- `try(in = null; out = null;){`
- similar to for loop syntax
- semicolons separated. Last semicolon is optional
- No multi-declaration.
- catch is optional, finally is optional
- try-with-resources is only try that can omit both optional blocks
- try-with-resources still responsible for checked exceptions (e.g. IOExceptions)
- Resources must implement AutoCloseable
- var allowed since a resource is a local variable
- resource is only in scope with try and implicit finally, not catch or explicit finally
- (NEW) resources can be declared before try-with-resource if the resource is final/effectively final. See Chapter 16.

Overriding methods with exceptions

- Override rule 3 : "An overridden method is not allowed to add new checked exceptions. Only same, fewer, or more specific"
- Put another way, an overridden method must throw the same, fewer, or covariant checked exceptions.
- Overridden method can throws new or covariant UncheckedExceptions at will
- Overridden method can declare fewer exceptions. (similar to unnecessary throws clause)
- If Interface/Abstract abstract method throws `Exception` (e.g. Callable), all implementations are free to throw any Exception

Output

- `System.out.println(e)` - "java.lang.RuntimeException: cannot hop"
- `System.out.println(e.getMessage())` - "cannot hop"
- `e.printStackTrace()` = "java.lang.RuntimeException: cannot hop /n at Handling.hop(Handling.java:15) /n ..."
- `e.getMessage()` returns the message from the Exception Constructor
- `System.out.println(e)` or `e.toString()` returns just the first line of the stacktrace (exception type and message)

Chapter 11 - Modules

- JPMS - Java Platform Module System - introduced in Java 9
- Defn: Module - group of one or more packages(plus static resources) plus a module-info.Java file.
- Module dependencies where one module relies on code in another
- module-info.java is required to be inside all modules.
- module names and package names are closely related. Generally, packages are "module_name" + "other_stuff"

Benefits of Modules

- Better access control - Allows targeted "public" access. Fifth level of access control. Expose packages to only specific packages.
- Clearer dependency management - Forcing users to specify which packages are dependent on others
- Smaller Java distributions - Smaller JDKs which only use some packages.
- Improved performance - Java only needs to look at subset of package at class loading time.
- Unique package enforcement - same packages, different versions.

Creating module-info

- module-info must be in the root directory of your module.
- top level type in the module-info is the `module` keyword
- same naming rules for package names.
- access modifier not allowed for top-level type of "module" in module-info.java

Compiling Modules

- Compiling modules uses the same `javac` command but with different arguments as the classpath arguments.

```
javac
--module-path mods
-d feeding
feeding/zoo/animal/feeding/*.java feeding/module-info.java
```

- replacing --class-path with --module-path
- "--module-path" indicates the location of any custom module files. Optional if no dependencies.

- "-d" option specifies the directory to place the class files
- the last two paths in the command is a list of the .java files to compile
- New option for `javac` command `--module-path` short form is `-p`

Running Modules

- `java --module-path feeding --module zoo.animal.feeding/zoo.animal.feeding.Task`
- `java --module-path mods --module book.module/com.sybex.OCP`
- Location of modules after --module-path
- module name after --module
- Forward slash between module name and class name `module.name/fully.qualified.class.Name`
- New option for `java` command `--module` short form is `-m`
    - takes argument of module containing class which contains the main method
- New option for `java` command `--module-path` short form is `-p`

Packaging Module

- jar'ing modules is identical to jar'ing packages.
- Same arguments as before (-cvf -C)
- `jar -cvf mods/zoo.animal.feeding.jar -C feeding/ .`

Keywords

- defn: Module Directives - "statements" that appear in module-info.java files (e.g. exports, requires, etc.)
- Directives can appear in any order in the module-info file
- module "keywords" are only keywords inside a module-info.java file. In other files, you are free to use these names elsewhere (e.g. provides, open, exports, etc.)

Exports

- defn: Exports - used to indicate that a module intends for those packages to be used by Java outside the module.
- Exporting makes public types (and protected) types visible. Not private or package-private. Same access rules after exported.
- Without an export, the module is only available internally
- `exports zoo.animal.feeding;`
- Qualified export: `exports zoo.animal.talks to zoo.staff;`

Requires

- defn: Requires - a module is needed.
- `requires zoo.animal.feeding;`
- `requires transitive zoo.animal.feeding`
- `requires static zoo.animal.feeding`
- requires transitive = any module that requires this module will also transitively depend on the modules this module marks as "requires transitive"
- requires transitive is like "requires" + "extra behavior"
- Compiler error if `require x` and `requires transitive x` in the same module-info.
- all modules implicitly require `java.base`

provides, uses, and opens

- provides keyword specifies that a class provides an implementation of a service
- uses keyword specifies that a module is relying on a service

New java commands

- `java --describe-module` or `-d`
- `java --list-modules`
- `java --show-module-resolution`

New jar Commands

- `jar --describe-module` // similar to java describe module

jdeps

- jdeps gives you information about dependencies within a module.
- tells you what dependencies are actually used rather than simply declared
- `jdeps -s`/ `jdeps -summary`
- specific module path

jmod

- only for working with JMOD files
- Don't have to learn the syntax

Second Half of Book

Chapter 12 - Java Fundamentals

Final Modifier

- final can be applied to variables, methods and classes
- final local variables can be declared and never initialized (unlike final instance/static variables)
- final != immutable. final objects can still be modified with method calls
- final classes cannot be extended

Enum

- defn: enumeration - top level type fixed set of constants
- enum values are considered constants and should be in SCREAMING_SNAKE_CASE
- list of enums values is required to be declared first
- list of enums are comma separated with a semicolon after the last enum.
- semicolon is optional if only enum constants, but if enum contains fields/constructors/methods, semicolon is required
- enums can be compared using == or .equals() because there is only one created by the JVM

- .values() method provided. returns an array of all of the values, e.g. returns `Season[]`
- .valueOf() method provided. Factory method turning String -> Enum
- .valueOf() must match the enum value exactly, case sensitive. runtime exception IllegalArgumentException thrown.
- enums can't be extended
- enums can implement interfaces however
- enum is a type, not a primitive

```
public enum Season {
    WINTER, SPRING, SUMMER, FALL
}
```

Enum switch statements

- enums can be used in switch statements.
- The case statement doesn't compile if you use the redundant `Season.FALL` instead of just `FALL`

Enum constructors, fields, and methods

- instance variables of enum can be mutable, but bad practice. Usually fields should be final
- All enum constructors are implicitly private.
- This makes sense since you can't extend an enum and you can't call constructors of enums.
- The first time the enum class is loaded, Java constructs all of the enums.

Enum methods

- enums can have simple methods that apply to all enum values
- enums can have abstract methods that are overridden by each enum
- "Creating a bunch of tiny subclasses"

```
public enum Season {
    WINTER {
        public String getHours() { return "10am-3pm"; }
    },
    SUMMER {
        public String getHours() { return "9am-7pm"; }
    },
    SPRING,FALL {
        public String getHours() { return "9am-5pm"; }
    },
    public abstract getHours();
}
```

Nested Classes

- defn: Nested class - a class that is defined within another class.
- Four types of nested classes:
    - Inner Class - A non-static type defined at the member level
    - Static nested class - A static type defined at the member level
    - Local class - A class defined within a method body
    - Anonymous class - A special case of a local class that does not have a name
- Nested classes help enforce encapsulation better. Although, they make the code more tightly coupled.
- interfaces and enums can be declared as both inner classes and static nested classes, but not as local or anonymous.
- It's possible to have inner classes within inner classes (bad practice)

Inner classes

- defn: Inner Class - A non-static type defined at the member level
- aka member inner class
- An inner object is associated with a specific outer object.
- properties
    - any access modifier (just like any member)
    - can extend any class and implement any interface
    - can be abstract or final
    - cannot declare static methods
    - all static fields must be static final
    - can access members of the outer class, including private members
- An inner class declaration looks like a stand-alone class declaration.
- Since an inner class is not static, it has to be used with an instance of the outer class.
- Multiple class files are created at compile time . Outer.class , Outer$Inner.class . ($ syntax not required on exam)
- Inner classes can have the same variable names as outer classes, complicating scope.
- static members of the outer class cannot instantiate the inner class without first instantiating the outer. The inner instance has to be associated with an instance of the outer.

Instantiating inner classes

- A method of the outer class can instantiate the inner class
- An inner object must be associated with a specific outer object.
- We can't just call `new Inner()` because Java won't know which instance of Outer it is associated with.

```
Outer outer = new Outer();
Inner inner = outer.new Inner()  // "new Inner()" is being used like a method name
```

Scope

- Inner classes can have the same variable names as outer classes, complicating scope.
- pre-qualifying "this" with a class reference. `Outer.this.x`
- `Outer.this.x` // inner vs `this.x` // depends on where

Static Nested Class

- defn: Static nested class - A static type defined at the member level
- Unlike inner classes, static nested classes can be instantiated without an instance of the enclosing class.
- Unlike inner classes, it can't access instance variables or methods from the outer class directly (but indirectly, by creating a dummy instance, yes)
- properties:
  - creates a namespace (like a new package `outer.inner.add`)
  - can be made private
  - Outer class can refer to the fields and methods of the static nested class
- You can instantiate a static nested class using `Outer.Inner inner = new Outer.Inner()`
- You can import static nested classes using regular imports or static imports
  - import bird.Outer.Inner;
  - import static bird.Outer.Inner;

Local Class

- Local class - A class defined within a method body
- Like local variables, local class declaration does not exist until the method is invoked
- Like local variables, local classes go out of scope when the method returns
- You can return instances of the local class as the return type of the method
- Local classes can be declared inside methods, constructors, and initializers
- properties:
  - No access modifiers
  - can't be static
  - can't declare static methods
  - static fields must be final
  - Full access to outer class fields and methods(*when defined in an instance method, otherwise static context)
  - Can access local variables iff final or effectively final
  - Only allowed to extend or implement one ??? property??? research
- a new .class file is generated for local classes.
- Local classes share the same rules as lambdas because lambdas are effectively local classes

Anonymous Class

- defn: Anonymous class - A special case of a local class that does not have a name
- declared and instantiated in the same one statement
- uses the new keyword, looks like calling constructor but with extra braces
- a type name
- braces

- ends with semicolon ***
- Since Anonymous classes are local classes, they can only access local variables iff final or effectively final
- Anonymous classes are required to extend an existing class or implement an existing interface
- Best used when you have a short implementation that will not be used anywhere else (e.g. sorting)
- You can "extend" abstract classes and interfaces with anonymous classes. The anonymous class provides the implementation of any abstract methods. The reference type will be the abstract class or interface you are implementing.
- Anonymous class is just an unnamed local class.
- Anonymous classes can be defined where they are needed, even if that is an argument to another method (e.g. PQ sort)
- You can define anonymous classes outside a method body. Tricky example:

```
public class Gorilla {
    interface Climb {}                    // nested interface
    Climb climbing = new Climb() {};     // import to remember semi-colon
    // braces makes this an anonymous class implementing the Climb interface
}
```

Interfaces

Interface members

- 6 types of interface members
    1. Constant Variables (public static final)
    2. Abstract method (public abstract)
    3. Default method (public default)
    4. static method (public)
    5. private method (private)
    6. private static (private static)

default method

- defn: default method - a default method in an interface is a method with a method body.
- "default implementation"
- default methods are treated as part of the instance, can't be used like static methods
- default methods must have the modifier "default". Not implicitly added
- May be overridden by a class implementing the interface, may not.
- not to be confused with the default keyword in switch statements
- not to be confused with the package-private access modifier that added if no access modifier

Rules for default interface methods

1. Only in interfaces
2. Must be marked "default" and have body

3. Implicitly public
4. Can't be abstract, final, or static
5. May be overridden by a class that implements the interface
6. If a class inherits two default methods with the same signature from different interfaces, then the class must override the method

Overridden Default Methods

- You can access overridden default methods by calling `Myinterface.super.defaultMethod()`

Static interface methods

- defined explicitly with the static keyword
- behave almost identically to static class methods
- Static interface methods are not inherited (this solves the diamond problem)
- called just like static class methods but with Interface name first, `MyInteface.staticMethod()`
- Since static methods are not inherited, there are no collisions with same method signatures from different Interfaces.
- Rules

1. must be marked "static"
2. must include a body
3. Either public or private
4. cannot be abstract or final
5. Static methods are not inherited and cannot be accessed in a class implementing the interface without a reference to the interface name.

Private Interface methods

- Not inherited
- mainly used to reduce code duplication inside of interface methods
- private vs private static
- private methods can't be referenced from static interface methods (static context)
- private interface methods behave a lot like instance methods within a class.
- Rules
  - Must be marked "private"
  - Must include a method body
  - Can only be called by default and private(non-static) methods within the interface.

Private Static Interface Methods

- mainly used to reduce code duplication
- Rules:

1. must be marked "private" and "static"
2. Must include a method body
3. May only be called by other methods within the interface.

Functional Programming

(bunch of repeated stuff from chapter 6)

- Functional Interfaces are used as the basis for lambda expressions
- Any functional interface can be implemented as a lambda expression
- defn: functional interface: - an interface that contains a single abstract method (SAM rule).
- defn: lambda expression: a block of code that gets passed around, sort of like an anonymous class that defines one method
- Weird edge case: if a functional interface includes an abstract method with the same signature as Object.toString or Object.equals() or Object.hashCode(), the interface is not considered a functional interface.
- behind the scenes, anonymous classes are used for lambda expressions (hence they share the final/effectively final rule)

Chapter 13 - Annotations

- Annotations are all about metadata.
- The purpose of an annotation is to assign metadata attributes to classes, methods, variables, etc.
- @ syntax
- can contain attribute/value pairs called "elements"
- annotations function a lot like interfaces
- annotations establish relationships that make it easier to manage data about our application.
- annotations ascribes custom information on the declaration where it is defined.
- annotations are optional metadata and by themselves don't do anything.
- new keyword is never used to create an annotation, just an "annotation literal"(`@Exercise(hoursPerDay=2)`)

Creating Custom Annotations

- @interface annotation to declare an annotation.
- Usually declared in their own file as a top-level type, but can be defined inside a class like inner classes.
- `public @interface Exercise{}`
- defn: Marker annotation - annotation that does not contain any elements, "zero arg"
- Parentheses are optional if no elements(marker annotations), but required for non marker annotations
- Annotations can be declared above or inline
- if an annotation is declared on a line by itself, then it applies to the next non-annotation type.
- Annotations are case sensitive
- Some annotations are lowercase, but PascalCase preferred

Elements

- defn: annotation element - "annotation param" - an attribute that stores values about the particular usage of an annotation.
- `int hoursPerDay();`
- When defined in the annotation class, it looks like an abstract method although it's an "element"
- parentheses required in "annotation literal" if annotation element exist
- When more than one element value, comma separated.

- syntax: `elementName=value` (similar to a Map)
- order of each element does not matter
- compiler checks for incompatible types
- Valid Element types = {primitives, String, "Clazz"(not a class), enum, another annotation, or an array of one of these types}
- Element types can't be {Primitive Wrapper classes, String[][] 2d arrays, Object, etc.}

```
public @interface Exercise {
    int hoursPerDay();
}
@Exercise(hoursPerDay=3) public class Cheetah {}
```

Optional Elements

- When declaring an annotation, any element without a default value is considered required.
- `int startHour() default 6;`
- Similar to case statement values, the default value of an annotation must be a non-null constant expression (compile time constant)

```
public @interface Exercise {
    int hoursPerDay();
    int startHour() default 6;
}
@Exercise(startHour=5, hoursPerDay=3) public class Cheetah {}
@Exercise(hoursPerDay=0) public class Sloth {}
```

Element Modifiers

- annotation elements are implicitly abstract and public (like abstract interface methods)
- since abstract, can't be final
- since public, can't be protected or private.

Constant variables

- Annotation can include constant variables
- can be accessed by other classes without creating the annotation. `@Test(timeout=TEST_TIMEOUT)`
- annotation variables are implicitly public, static, final (just like interface variables)
- constant variables are not considered annotation elements
- Marker annotations can contain constants

Annotation locations

- @Target annotation limits the types the annotation can be applied to.

- @Target takes a param of `ElementType[]`
- Annotations can be used to any Java declaration, including:
  - Cast Expressions (interesting!)
  - Classes, interfaces, enums, and modules
  - Variables (local, instance, static)
  - Methods and constructors
  - Method, constructor, and lambda parameters
  - Other annotations

value() element (Keyless Element/Argument)

- shorthand notation for one argument annotation where the one argument is unambiguous
- `@Injured("Broken Tail")`
- annotation with a value without the elementName/"key"
- Rules
  - annotation declaration must include `value()` method with exact name(with either a default value or required)
  - No required elements (except optionally `value()`)
  - Only one param, the value() param, can be passed to usage("annotation literal")
- you can optionally specify the key name, namely `value="foot"`
- `Injured("Legs")`
- `Injured(value="Legs")`
- Typically, the value() of an annotation should be related to the name of the annotation

Passing an array of values

- shorthand exists for singleton arrays
- If array type, and singleton array, braces are optional (e.g. testng `@Test(groups={"Regression"})`)
- Compiler provides the missing braces for you
- If multiple elements, compiler error (type checking)
- `@Music(genres={"Classical"})` === `@Music(genres="Classical")`
- NOTE `@Music(genres={})` is valid, `genres=null` or `genres=` are invalid
- Doesn't work for List or Collection since they are not supported element types
- value() shorthand can be combined with singleton array shorthand
- `@Music(value={"Swing"})` === `@Music("Swing")`

Built-in annotations

- java.lang.annotation package is not imported automatically.
- @Target - limits the types the annotation can be applied to. Takes an array of ElementType enum value
  - `@Target({ElementType.METHOD, ElementType.CONSTRUCTOR})`
  - Default behavior doesn't allow lambda expression or generic declarations to be annotated. Must be overridden with @Target
  - `ElementType.TYPE_USE` can be used anywhere there is a Java type (90% of everywhere)
- @Retention - Annotations may be discarded by the compiler or at runtime. @Retention overrides the retention policy
  - `@Retention(RetentionPolicy.RUNTIME)`
- @Documented - Marker annotation that tells generated javadoc to include annotation information on Java Types.

- @Inherited - Marker annotation that forces subclasses to inherit the annotation information from parent class. This annotation is applied to other annotations, not to classes.

Duplicate annotations (@Repeatable)

- Without the @Repeatable annotation, an annotation can be applied only once
- Generally, you use repeatable annotations when you want to apply the same annotation with different values
- @Repeatable - used when you want to specify an annotation more than once on a type
- requires creating two annotations, "containing annotation" and "repeatable annotation"
- To declare a @Repeatable annotation , you must define a containing annotation type value
- A containing annotation type is a separate annotation that defines a `value()` array element. The type of this array is the type of the Annotation you want to repeat
- By convention, the name of the annotation is often the plural form of the repeatable annotation (Risk and Risks)
- Rules
  - Repeatable annotation must be declared with @Repeatable and contain a value that refers to the containing type annotation
  - containing type annotation must include an element named value(), which is a primitive array of the repeatable annotation type.

```
public @interface Risks {
    Risk[] value();
}
@Repeatable(Risks.class)
public @interface Risk {
    String danger();
    int level() default 1;
}
```

Javadoc vs java annotation

- @param, @return, @author are all javadoc annotations included in the javadoc(/** */)
- javadoc annotations are usually all lowercase
- Java annotations usually start with an uppercase
- @deprecated (javadoc) vs @Deprecated (java)

Common Annotations

- @Override - optional marker annotation to indicate a method is overriding another.
- incorrect usage of @Override will throw compiler error
- @FunctionalInterface - optional marker annotation to state a Interface is a FunctionalInterface
- incorrect usage of @FunctionalInterface will throw compiler error. (more than one default method)
- @Deprecated - notates that a Java type is scheduled for deprecation
- optional arguments: `String since()` and `boolean forRemoval()`
- not to be confused with `@deprecated` which is a javadoc annotation. Best practice to use both.
- @SuppressWarnings - Tells the compiler to stop spamming logs with warnings. Apply to class, method, or type
- `@SuppressWarnings({"deprecation", "unchecked"})`

- Common suppressed warnings: deprecated methods, unchecked raw types (e.g. List)
- @SafeVarargs - Market annotation that indicates that a method does not perform any potential unsafe operations on its varargs parameter. Can only be applied to constructors and methods that can't be overridden (e.g. private, static, final)
- Example of unsafe operations: Generic array casting varargs, heap pollution,

---

Chapter 14 - Generics and Collections

- Review of Chapter 5 as well as more in depth

Method References

- `System.out::println`
- `::` operator tells Java to call the println method later.
- `::` is like a lambda because it is used for deferred execution with functional interfaces
- A method reference can look the same but behave differently based on the surrounding context.
- If a method is overloaded, both lambdas and method references infer which version of a method to use from the context.
- If no methods or multiple methods match from context, compiler error thrown (ambiguous type error)
- Java knows to pass the parameter into the method.
- You can pretend the compiler turns your method references into lambdas for you
- Four formats:
    1. Static methods
    2. Instance methods on a particular instance
    3. Instance methods on a parameter to be determined at runtime
    4. Constructors

Calling Static Methods

- Collections::sort
- ClassName::staticMethodName

Calling Instance Methods on a Particular Object

- s::startsWith
- variableName::methodName

Calling Instance Methods on a parameter

- String::isEmpty;
- ClassName::instanceMethodName

Constructors References

- Constructor reference is a special type of method reference that uses new instead of a method name.
- It is common for a constructor reference to use a Supplier reference

- ArrayList::new
- ClassName::new
- `Functional<Integer, List<String>> methodRef = ArrayList::new` // Instantiate ArrayList with capacity integer param.
- Inferring the type of the constructor from the context

Wrapper Classes (review)

- defn: autoboxing - compiler automatically converts primitive to wrapper
- defn: unboxing - compiler automatically converts wrapper to primitive
- `valueOf()` vs `parseInt()`
- wrappers can be null, but attempting to unbox a null reference throws NPE
- autounboxing isn't a word. It's autoboxing and unboxing.

Diamond Operator

- defn: diamond operator - shorthand notation that allows you to omit the generic type from the right side of a statement when the type can be inferred from the left side
- Can only be used on the right side of an assignment operator. Fails to compile altogether, not just raw type.
- `<>` looks like a diamond
- `List<String> = new ArrayList<>();`
- Var edge cases:
  - `var list = new ArrayList<Integer>();` // `list instanceof ArrayList<Integer>`
  - `var list = new ArrayList<>();` // `list instanceof ArrayList<Object>`
  - var can be used with generics, but the type defaults to Object if not specified.

Lists, Sets, Maps, and Queues

- defn: collection - a group of objects contained in a single Object.
- The Java Collections Framework is a set of classes in java.util for storing collections.
- Four main Interfaces in Java Collections Framework:
  - List - ordered collection of elements accessible by index
  - Set - unordered collection that does't allow duplicates
  - Map - collection that maps keys->values with no duplicate keys allowed
  - Queue - collection that orders elements in a specific order for processing.
- List, Queue, Set instanceof Collection, but not Map
- Map is considered part of the Java Collection Framework but not a sub-interface of Collection.

Common Collections Methods

- Many of the common collections methods are convenience methods that could be implemented in other ways.
- .add() - inerts a new element into the collection and returns whether it was successful
- adding to a list always returns true, but adding to a set may return false.
- remove() - removes a single matching value and returns true if the element was found and removed.
- notably only removes a single value, not all values.

- remove() can also be called with an index. If index is too larger, `IndexOutOfBoundsException`
- NOTE: ConcurrenceModificationException if iterating and removing
- .isEmpty() and .size()
- .clear() - discard all elements of the collection. void return type.
- .contains() - returns true if a certain value is in the collection.
- .contains() method calls .equals() on each element in the collection
- .removeIf() - `Predicate<? super E>` that removes an element if condition is true.
- .forEach() - `Consumer<? super T>` does something with each value and doesn't return anything, void.
- (ArrayList only) .replaceAll() - `UnaryOperator<E> operator` replace each value with the output (e.g. x -> x*2)

List Interface

- ordered (can access by index), can contain duplicates
- Unlike an array, many List implementations can change in size after they are declared
- ArrayList is a resizable array. Constant time lookup by index. Adding or removing an element is slow.
- ArrayList is good for reading when you know index. Bad for writing large amounts of data.
- LinkedList is a special list because it implements both List and Queue.
- LinkedList allows constant time add/removing from the beginning and end of the LinkedList.
- LinkedList is bad at index. Linear time to access element by index.

Factory Methods of Lists

- Arrays.asList -> returns fixed size list backed by an array. You can replace, but not delete or add.
- List.of -> returns an immutable list. You can't add, delete, or replace.
- List.copyOf -> returns an immutable copy. You can't add, delete, or replace.
- ArrayList::new -> Solution. Constructor with one argument that takes the other Collection and uses it as a template.

Extra list methods

- add(index, element)
- get(index)
- remove(index)
- replaceAll(UnaryOperator) - nums.replaceAll(x -> x*2)
- set(index, element)
- .iterator()

Set Interface

- HashSet stores elements in a hash table.
- The keys are a hash and the values are an Object. .hashCode()
- adding and checking if contains element are both constant time
- TreeSet stores its elements in order
- TreeSet uses a heap
- TreeSet is much slower at adding and checking if contains element.
- TreeSet does not allow null values

Set Methods

- .add() returns false if element already exists in set
- Set.of() - Immutable set
- Set.copyOf() - Immutable clone
- (other Collection methods)

Queue Interface

- Queues are used when order is important
- Unless stated otherwise, a queue is FIFO (first-in, first-out)
- Some queue implementations change this to use LIFO(Last-in, first-out)
- A stack is LIFO. A stack still implements the Queue interface, but LIFO vs FIFO.
- Stack is older code. Use LinkedList instead
- LinkedList is a double ended queue that is also a List
- defn: double-ended queue - you can insert and remove elements from both the front and the back of the queue
- LinkedList is not as efficient as "pure" queues.

Queue Methods

- add() - Adds an element to the back of the queue and returns true or throws and exception
- element() - Returns next element or throws an exception if empty queue
- offer() - Adds an element to the back of the queue and returns true if successful
- remove() - Removes and returns next element or throws exception if empty
- poll() - Removes and returns next element or returns null if empty
- peek() - Returns(but don't remove) the next element or returns null if empty
- Basically two sets of methods
  - One set throws an exception when something goes wrong --- (add, remove, element)
  - The other set returns null --- (offer, poll, peek)
- offer/poll/peek are more common.

Map Interface

- Map has a static interface inside the Map class called Entry
- Map.Entry provides two methods Map.Entry::getKey and Map.Entry::getValue
- Map.of -
- Map.copyOf -
- Map.ofEntries(Map.entry...)
- HashMap stores the keys in a hash table. (using .hashCode() method). Adding and getting elements both are constant time
- TreeMap stores the keys in a order using a heap. Adding and getting elements takes longer time
- TreeMap does not allow null values

Map Methods

- boolean .containsKey - returns true if key in keySet
- boolean .containsValue - returns true if value in values
- Set .entrySet - returns keyValue pairs as a `Set<Map.Entry<K,V>>`
- Set .keySet - returns keys as a `Set<K>`

- Collection .values - returns values as a `Collection<V>`
- int .size - returns the number of key/value pairs (.entrySet().size())
- boolean .isEmpty - returns true if the size == 0
- V .get - Returns the value mapped by key OR NULL
- V .getOrDefault - Returns the value mapped by key or DEFAULT value provided
- V .put - Adds or replaces key/value pair. Returns previous value or null.
- V .putIfAbsent - sets a value but skips if the value is already set to a non-null value.
- V .remove - Removes and returns value mapped to key. Returns null if key doesn't exist
- void .clear - Removes all keys and values from the map
- V .replace - map.replace(2,10) // replace(k, v)
- void .replaceAll - map.replaceAll((k,v) -> (k, v*2))
- void .forEach
- V .merge

Sorting Data

- numbers < ALPHA < alpha - (ASCII table order)
- Collections.sort() returns void, in place sorting
- Comparable - Interface that defines .compareTo()
- If you class implements Comparable, it can be used in these data structures that allow sorting.
- Comparator - class that is used to specify that you want to use a different order than the object provides by default
- "natural order" vs "custom order"

Comparable

- java.lang (automatically imported)
- Comparable Interface has only one method, `int compareTo(T o)`
- Comparable is an Functional interface since the only method is an abstract method

```
public class Duck implements Comparable<Duck> {
    public int compareTo(Duck d) {
        return name.compareTo(d.name);
    }
}
Collections.sort(ducks)
```

- Any object can be comparable.
- Three rules for compareTo()
    1. Zero when equal
    2. Negative when object is smaller than the parameter in compareTo
    3. Positive when object is greater than the parameter in compareTo

- .compareTo() and .equals need to be consistent on when two objects are equal else collections may misbehave.
- If Comparable is implemented with raw type, then `compareTo(Object obj)` method takes Object.
- If Comparable is implemented with raw type, then cast is needed in the implementation of compareTo method
- Be careful of null comparisons: myObj.compareTo(null)

Comparator

- java.util.Comparator (different package than Comparable, not automatically imported)
- We can define a different way than the existing way by creating a Comparator
- Comparator is a functional interface since there is only one abstract method.
- `Comparator<Duck> byWeight = Comparator.comparing(Duck::getWeight);`
- `Comparator::comparing` is a static factory interface method that creates a Comparator given a lambda or method reference.

Comparable vs Comparator

- They are both functional interfaces
- The point of Comparable is to implement it inside the object being compared
- The point of Comparator is to be passed as a lambda for custom sorting on demand
- Different packages (java.lang vs java.util)
- Different method names : Comparable::compareTo vs Comparator::compare
- Comparable::compareTo takes 1 argument, Comparator::compare takes 2 arguments

Comparing Multiple Fields (/tie breakers)

- `.thenComparing()` for tie breakers
- `Comparator<Squirrel> c = Comparator.comparing(Squirrel::getSpecies).thenComparingInt(Squirrel::getWeight)`
- `var c = Comparator.comparing(Squirrel::getSpecies).reversed()`

Comparator methods

- static methods for building a Comparator
- comparing()
- comparingDouble
- comparingInt
- comparingLong
- naturalOrder
- reverseOrder
- reversed
- thenComparing
- thenComparingDouble
- thenComparingInt
- thenComparingLong

Sorting and Searching

- Collections.sort method uses the compareTo() method to sort
- Collections.sort() expects the collection passed as an argument to implement the Comparable interface

- `Collections.sort(Comparable<T>)` - if collection implements the Comparable interface
- `Collections.sort(Collection<T>, Comparator<T>)` - if not, then pass a Comparator lambda/method reference
- Compiler will throw error if collection passed that doesn't implement Comparable and no second argument.
- .sort() and .binarySearch() methods allow you to pass in a Comparator object when you don't want to use the natural order.
- three param binary search `Collections.binarySearch(names, "Hoppy", myComparator)` //binary search in descending order.

Generics

- Generics(Parameterized types) vs Raw Types
- Raw Types are a nightmare to deal with. Error prone.
- Generics are rarely used in custom classes you write, but they are everywhere in Java core packages, such as Java Collections Framework.
- Reifiable types aren't on the exam

Generic Classes

- defn: Generic class is a class that declares a formal type parameter in angle brackets.
- The generic type T is available anywhere within the Crate class.
- When you instantiate the class, you tell the compiler what T should be for each particular instance
- `Crate<Integer>` vs `Crate<String>` vs `Crate<Date>`

```
public class Crate<T> {
    private T contents;
    public T emptyCrate() {
        return contents;
    }
    public void packCrate(T contents) {
        this.contents = contents
    }
}
```

Naming Conventions for Generics

- A type parameter can be named anything.
- The convention is to use single uppercase letters to make it obvious that they aren't real class names
  - E for element
  - K for map key
  - V for map value
  - N for a number
  - T for a generic data type
  - S,U,V,... for multiple generic types

Type Erasure

- Generic types help enforce rules at compile time, but not a runtime
- At runtime, the compiler replaces all references to `T` with `Object`
- This is called Type Erasure
- Type Erasure allows your code to be compatible with older versions of Java that don't have generics.
- The compiler adds the relevant casts from `Object` -> `T`

Generic Interfaces

- Just like classes, interfaces can declare a formal type parameter

```
public interface Shippable<T> {
    void ship(T t)
}
```

- If a class implements a generic interface:
  1. The class specifies the generic type in the Class declaration `class ShippableRobotCrate implements Shippable<Robot> {}`
  2. The class and the interface share a generic type `class ShippableAbstractCrate<U> implements Shippable<U> {}`
  3. The class uses raw types

Raw Types

- This is the old way of writing code before generics.
- Generates compiler warnings (which can be suppressed with `@SuppressWarning("unchecked")`)

Generic Methods

- Methods also allow formal type parameters
- Method and class formal parameter types are entirely independent if both declare a formal type parameter, even if the same letter/name `<T>`.
- If same letter without `<>`, it's not a formal type parameter being declared.
- This is often useful for static methods since they aren't part of an instance.
- Before the return type, we declare the formal parameter of `<T>`
- Unless a method is obtaining the generic formal type parameter from the class/interface, it is specified immediately before the return type of the method.

```
public class Handler {
    public static <T> void prepare (T t) {
        System.out.println("preparing " + t);
    }
    public static <T> T doNothing(T t) {return t;}
}
```

- You can specify the type explicitly on method calls with this weird syntax. Otherwise, compile will try to infer.
- `Box.<String>ship("Package")`

Bounded Generic Types

- defn: bounded parameter type - a generic type that specifies a bound for the generic.
- defn: wildcard generic type - an unknown generic type represented with a `?`
- Bounded Generics are only allowed on the left side of an assignment operator
- You can use generic wildcards in three ways:
    1. Unbounded `?`
    2. Wildcard with an upper bound `? extends type`
    3. Wildcard with a lower bound `? super type`

Unbounded Wildcards

- An unbounded wildcard,`?`, represents any data type.
- Unbounded generics are immutable
- A compiler error thrown if attempting to add an item to a list with an unbounded or upper-bounded wildcard.
- `List<String>` cannot be assigned to `List<Object>`
- `List<String>` can be assigned to `List<?>` however
- Review casting for generics
- `var` and `List<?>` are not the same thing. Both return type Object when calling `get()` though

Upper-Bounded Wildcards

- `List<? extends Number> list = new ArrayList<Integer>();`
- A compiler error thrown if attempting to add an item to a list with an unbounded or upper-bounded wildcard.
- The upper-bound wildcard says that any class that extends Number or Number itself can be used as the formal parameter type
- Upper bounds are like anonymous classes in that they use extends regardless of whether we are working with a class or an interface

Lower-Bounded Wildcards

- `List<? super String>`
- We are telling Java that the list will be a list of String objects or a list of objects that are a superclass of String. Either way, it is safe to add a String to the list.
- Lower-bounded objects are mutable (unlike Upper-bounded or wildcard boundless)

Chapter 15 - Functional Programming

- Java i/o streams and the Streams API are nothing alike, even though they use the same name.
- Recall functional interfaces have exactly one abstract method

Built-in Functional interfaces

- UnaryOperator and BinaryOperator are special cases of Function. T,T -> T
- UnaryOperator extends Function, just with different formal type parameters
- The generic declarations on UnaryOperator force the functionality on Function.
- Predicate returns boolean, not Boolean. If Boolean returned, think Function not Predicate.

Convenience Methods on Functional Interfaces

- All of these help modify and combine functional interfaces
- Must be the same functional interface type
- Consumer :::: .andThen()
- Function :::: .andThen() , .compose(),
- PREDICATE :::: .and(), .or(), .negate()
- remember f o g , f.compose(g), is equivalent to f(g(x)).

Optional

- Optional objects are used to express "not applicable" or "we don't know"
- Optional is created using a factory method

Optional methods

- Factory methods
  - .of() - creates an optional of `<T>` type
  - .ofNullable() - factory method to return Optional.empty() if value==null, or Optional.of(value) else.
  - .empty() - creates an empty Optional
- Methods
  - .isPresent() - returns true if Optional is not empty
  - .ifPresent() - Consumer
  - .get() on an optional without checking is empty could throw `NoSuchElementException`
  - .orElse() - maxOpt.orElse(0);
  - .orElseGet() - Supplier
  - .orElseThrow() - zero param or one param with `Supplier<Exception>`
  - .getAsInt() vs .get() only available for OptionalInt/OptionalDouble/OptionalLong, not Optional

Streams

- defn: stream - a sequence of data of elements from a source that supports aggregate operations.
- defn: stream pipeline - operations that run on a stream to produce a result
- defn: stream operations - operations (intermediate or terminal) that act on the elements of a stream.
- stream does not implement `Iterable<T>` and thus can't be used in for-each loops
- finite streams have a limit, infinite streams do not
- lazy evaluation - delays execution until necessary
- Three parts of stream pipeline:
  1. Source - where the stream comes from
  2. Intermediate operations - transforms the stream and returns another stream

3. Terminal operations - Ends the stream and returns a result

- There is no limit to the number of intermediate operations
- Intermediate operations do not run until the terminal operation runs
- Any intermediate operation after a terminal operation will throw SymbolError
- Special streams exist for primitives
- You can't reuse the same stream

Stream Sources

- finite stream constructors
    - Stream.empty()
    - Stream.of(T...)
    - coll.stream()
    - coll.parallelStream()
    - Arrays.stream()
- infinite stream constructors
    - Stream.generate(Math::random);
    - Stream.iterate(1, n -> n + 2);
    - Stream.iterate (identity, Predicate, UnaryOperator) - (1, n < 100, n -> n + 2);
- you can limit infinite streams with .limit()
- `Stream.<String>builder()` - returns a builder for a stream.cf

Terminal Operations

- Reductions are special type of terminal operations where all of the contents are combined into a single primitive or Object
- .count() - Reduction operator. return long number of elements.
- .min() and .max() - Reduction operator. Takes a Comparator and returns `Optional<T>`.
    - Required Comparator param, no "natural order" (unlike .sorted())
- .findAny() and .findFirst() - Terminal but not reduction. returns Optional. "short circuit" not all elements are seen.
- .allMatch(), anyMatch(), noneMatch() - Takes a `Predicate<T>` and returns boolean
- .forEach() - Takes a `Consumer<T>` and returns void.
    - collection.forEach() is different than myStream.forEach()
- .reduce() - combines a stream into a single object
    - Three different signatures
        1. `T reduce(T identity, BinaryOperator<T> accumulator)`
        2. `Optional<T> reduce(BinaryOperator<T> accumulator)`
        3. `<U> U reduce(U identity, BiFunction<U, ? super T,U> accumulator, BinaryOperator<U> combiner)`
    - identity is the initial value of the reduction (e.g. "", or new ArrayList)
    - accumulator combines the current result with current value
    - combiner combines any intermediate totals.
    - `Stream.of("A", "B", "C").reduce("", String::concat);` // "ABC"
- .collect() - mutable reduction
    - Use a prebuilt Collector in `Collectors` class or create one yourself
    - More efficient than a regular reduction because we reuse the same mutable object while accumulating
    - Two different signatures

    1. `<R> R collect(Supplier<R> supplier, BiConsumer<R, ? super T> accumulator, BiConsumer<R,R> combiner)`
    2. `<R,A> R collect(Collector<? super T,A,R> collector)`
  - `stream.collect(StringBuilder::new, StringBuilder::append, StringBuilder::append)`
  - supplier creates the object that will store the results. Similar to identity for .reduce()
  - accumulator BiConsumer that adds one more element
  - combiner BiConsumer merges two intermediate/subsection results together
  - `stream.collect(TreeSet::new, TreeSet::add, TreeSet::addAll)`
  - `stream.collect(Collectors.toCollection(TreeSet::new)`
  - `stream.collect(Collectors.toList()`
  - `stream.collect(Collectors.toSet()`

Intermediate Operations

- returns another stream
- .filter() - Takes a Predicate and returns only the elements where predicate is true
- .distinct() - No params and returns a stream without duplicates. (calls .equals() under the hood)
- .limit() - returns a stream with a smaller source
- .skip() - skips the first n elements
- .map() - maps elements from old stream to a new stream
  - `<R> Stream<R> map(Function<? super T, ? extends R> mapper)`
  - can return a stream of a different type than the original Stream (e.g. .map(String::length))
- .flatMap() - returns a stream of the elements of the
  - `<R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>> mapper)`
  - Helpful for combining a stream of lists.
  - `Stream<ArrayList<Integer>> -> Stream<Integer>`
- .sorted() - returns a stream with the elements sorted
  - uses natural ordering unless a comparator is provided
  - `stream.sorted(String::compareTo)`
- .peek() - apply a Consumer without altering or terminating the Stream
  - `Stream<T> peek(Consumer<? super T> action)`
  - Not intended to change results. If you want to change, use a map

Primitive Streams

- Convert `Stream<Integer>` -> IntStream with `stream.mapToInt(x -> x)`
- Primitive streams have many of the same intermediate and terminal methods as Stream
- Primitive streams also include specialized methods for working with numeric data.

Creating Primitive Streams

- All primitives don't have their own Streams class. There are only three.
- Three Types of primitive streams
  1. IntStream - used for byte,short,int, and char
  2. LongStream - used for long
  3. DoubleStream - used for float and double
- same factory methods, .of(), .empty(), .generate(), .iterate()

Primitive Stream methods

- .boxed() returns a `Stream<T>` where T is the boxed version of prim in PrimStream
- .min(), .max(), .average(), .sum
- .range() , .rangedClosed()
  - `IntStream.range(1,11)` // 1-10, `[a,b)`
  - `IntStream.rangeClosed(1,10)` // 1-10, `[a,b]`
- .summaryStatistics()
- flatMapToInt, flatMapToDouble, flatMapToLong

Mapping Streams

- `Stream<T>` -> IntStream
  - stream.mapToInt()
- mapToInt(), mapToDouble(), mapToLong(), mapToObj()
- IntStream -> `Stream<T>`
  - iStream.mapToObj()
- The parameters to these mappers are actually custom interfaces
  - `ToDoubleFunction<T>`
  - `ToIntFunction<T>`
  - `LongToDoubleFunction`
- charts on page 711 and 712

Primitive Options

- primitive streams return OptionalDouble, instead of `Optional<T>`
- `Optional<Double>` holds wrapper Doubles
- OptionalDouble holds primitive doubles
- `OptionalDouble opt = stream.average()`
- sum() notably returns int/double/long, not OptionalInt/OptionalDouble/OptionalLong

Summarizing Statistics

- `IntSummaryStatistics stats = iStream.summaryStatistics()`
  - `iStream.summaryStatistics().getMax()`
  - `iStream.summaryStatistics().getMin()`
  - `iStream.summaryStatistics().getSum()`
  - `iStream.summaryStatistics().getAverage()`
  - `iStream.summaryStatistics().getCount()`

Special Functional Interfaces for Primitives

- `BooleanSupplier.getBoolean() returns boolean`
- `IntConsumer.accept() void`
- `DoublePredicate.test() returns boolean`
- `LongFunction<R>.apply() returns <R>`
- `IntUnaryOperator.applyAsInt() returns int`
- `DoubleBinaryOperator.applyAsDouble() returns double`
- `ToIntFunction<T> returns int`

- `ToDoubleBiFunction<T,U> returns double`
- `LongToDoubleFunction returns double`
- `ObjIntConsumer returns void`

Advanced Streams

Collection backed Streams

- coll.stream() creates a stream BACKED by a collection. Altering the underlying collection also changes the stream
- Recall streams can only be traversed once. Altering the backed collection after doesn't do anything. Only before.

Collecting Results

- Collector is an interface
- Collectors is a helper class
- Collectors interface has static factory methods for Collector objects.
- Collect is passed as a parameter to the `stream.collect()` method
- Collectors.averagingDouble(), averagingInt, averagingLong - calculates the average and returns Double
- .counting() - counts the number of elements, returns Long
- .averagingDouble()
  - `stream.collect(Collectors.averagingInt(String::length))`
- groupingBy
- joining
  - `Stream.of("lions", "tigers", "bears").collect(Collectors.joining(", "));` // lions, tigers, bears
  - .joining delimiter
- maxBy, minBy
- mapping
- partitioningBy
- summarizingDouble
- summingInt
- toList
- toSet
- toCollection
  - `.collect(Collectors.toCollection(TreeSet::new))`
- toMap

Collecting into Maps

- There are many different ways to collect into maps, some being complicated
- Simple:
  - `stream.collect(Collectors.toMap(Function::identity, String::length))`
- When creating a map, you need to specify two functions. One for key creation, the other for value creation.
- `IllegalStateException: Duplicate key 42`
- collector will freak out if duplicate keys are created.
- groupingBy() and partitioningBy() appear on the exam a lot
- .groupingBy()

  - return type = Map
  - `.collect(Collectors.groupingBy(String::length))`
  - groupingBy tells collect() to group all of the elements of the stream into a map
  - The function determines the keys in the map.
  - Each value is a List of entries
  - does not allow null keys
  - downstream collector - A second collector that does something special with the values
  - `stream.collect(Collectors.groupingBy(String::length, Collectors.toSet()))` // two Collectors references!
- .partitionBy()
  - Partitioning is a special case of grouping
  - Partitioning groups values into key=True and key=False
  - `Map<Boolean, List<String>> myMap = stream.collect(Collectors.partitioningBy(s -> s.length() <= 5))`
  - passing a Predicate
  - We can change the collection that holds the values. Defaulted to List, but we can make it a set, etc.
  - `Map<Boolean, Set<String>> myMap = stream.collect(Collectors.partitioningBy(s -> s.length() <= 5), Collectors.toSet())`
  -

OCC MAP

- Creating an occurence map

```
 Map<Integer, Long> occMap = stream.collect(Collectors.groupingBy(String::length,
Collectors.counting()))
 Map<Integer, Long> occMap = stream.collect(Collectors.groupingBy(Function::identity,
Collectors.counting()))
        Map<Integer, Integer> occMap  = Arrays.stream(nums).boxed().collect(
                Collectors.groupingBy(
                        x -> x,
                        Collectors.summingInt(x -> 1)
                )
        );
```


Chapter 16 - Exceptions, Assertions, and Localization

EXAM UPDATE

- Asserts removed from the exam

Reviewing Exceptions

- var is allowed in try-with-resource
- Throwable is the parent of all Exceptions
- NumberFormatException < IllegalArgumentException
- FileNotFoundException < IOException
- NotSerializableException < IOException

Creating Custom Exceptions

- `class CannotSwimException extends Exception {}`
- "Checked" vs "unchecked" is inherited since being a subclass is transitive.
- Three most common Exception constructors
  - No argument constructor created manually
  - One arg constructor (Exception) that wraps the "cause" exception
  - One arg constructor (String) that passes a message

Stack Traces

- defn: shows the exception along with the method calls it took to get there
- e.printStackTrace()

try-with-resource

- aka automatic resource management
- resources must implement AutoCloseable interface (.close() method)
- implicit finally
- semicolon separated
- NEW: possible to use resources declared prior to the try-with-resource if final/effectively_final
- NEW: just use the resource name instead of declaration

```
final var bookReader = new FileReader("A");
try(bookReader) {
    //
}
```

- `e.getSuppressed()`
- Suppressed exceptions - when multiple exceptions are thrown, all but the first are suppressed
- exceptions in implicit finally can be suppressed if the try statement throws a fatal exception

---

Assertions

- removed from exam

Working with Dates & Times

- The new Java 11 Exam reduced the amount of date/time classes that we need to know
- Still need to know how to format dates, just not the classes themselves.

Creating Dates & Times

- Before Java 8: `Date` and `SimpleDateFormat`
- New Java 8: `LocalDate` and `DateTimeFormatter`
- LocalDate - day,month,year - `2020-10-14`
- LocalTime - time of day - `12:45:20.000`
- LocalDateTime - day and time w/o TZ - `2020-10-14T12:45:20.000`
- ZonedDateTime - Date and time with TZ - `2020-10-14T12:45:20.000-04:00[America/New_York]`
- All four of the classes have static factory methods
  - .now()
  - .of()
    - `LocalDate.of(2020, 10, 20);`
    - You can omit params and defaults will be provided
    - `LocalTime.of(6, 15)` // no seconds
    - `LocalDateTime.of(LocalDate.now(), LocalTime.now())` // passing LocalDate and LocalTime as params to LDT
- All constructors are private for LocalDate. Factory methods are the only way.
- Months are 1-indexed (october = 10). Alternatively. `Month.OCTOBER` enum was created

Formatting Dates and Times

- .getDayOfWeek() // TUESDAY
- .getMonth // OCTOBER
- .getYear // 2020
- .getDayOfYear() // 294
- DateTimeFormatter class.
- Prebuilt formats:
  - `ld.format(DateTimeFormatter.ISO_LOCAL_DATE)`
- custom formats:
  - `DateTimeFormatter.ofPattern("MMMM dd, yyyy 'at' hh:mm");`
- DateTimeFormatter cant be used with the wrong type (using time for date or date for time)
- Format symbols: `y,d,M,h,m,s` all obvious. `a` = "am/pm", `z` = Time Zone name, `Z` = Time Zone offset (-0400)
- `formatterObject.format(ldtObject) vs ldtObject.format(formatterObject)` equivalent. Same thing, different caller.
- You can escape the formatter syntax using `''` single quotes. `'at'`
  - Using single quotes `"MMMM dd 'Party''s at'"` // double single quotes next to each other. using single quote to escape single quote

---

i18n and l10n

- defn: i18n - internationalization (i18n) is the process of designing and developing software or products that can be adapted to different languages and cultures
- defn: l10n - localization (l10n) is the process of adapting a product or content for a specific locale or market
- l10n includes (country, language) pair and formatting dates and numbers in the correct locale
- Builder pattern = creational design pattern where object is first created, then properties are set with methods
- Factory pattern = creational design pattern where a factory class is used to provide instances of an object instead of relying on constructors.

Locale class

- java.util
- `Locale locale = Locale.getDefault()` //en_US
- Locale identifier:
  - en_US
  - lowercase language code
    - en
  - Uppercase country code:
    - US
- Constants
  - Locale.GERMAN // de
  - Locale.GERMANY // de_DE
- Constructors
  - new Locale("fr") // fr
  - new Locale("hi", "IN") // hi_IN
- Builder Pattern (not factory!) `Locale usLoc = new Locale.Builder().setLanguage("en").setRegion("US").build();`
- Set locale
  - `Locale.setDefault(otherLocale)`

Localizing Numbers

- java.text
- formatting turns (Object/primitives) into (Strings)
- NumberFormat
- NumberFormat factory methods
  - .getInstance() - general-purpose formatter
  - .getNumberInstance() - same as .getInstance()
  - .getCurrencyInstance() - For formatting monetary amounts
  - .getPercentageInstance() - For formatting percentages
  - .getIntegerInstance() - Rounds decimal values before displaying
  - all above methods are overloaded for (Locale otherLocale) param.
- .format()
  - `nformatter.format(3_200_000)` // 3,200,200
  - `nformatter.getCurrencyInstance.format(48d)` // $48.00
- .parse()

  - returns Number object
  - throws checked exception `ParseException`
  - `nformatter.parse("40,45")` // 40.45 in en_US
  - `(Double) nformatter.getCurrencyInstance.parse("$92,807.99")` // $92807.99 cast Number -> Double
- Custom Number Formatter
  - new DecimalFormat("###,###,###.0")
  - `#` = Omit the position if no digit
  - `0` = Put 0 in position if no digit exists

Localizing Dates

- - DateTimeFormatter.ofLocalizedDate(dateStyle)
  - DateTimeFormatter.ofLocalizedTime(timeStyle)
  - DateTimeFormatter.ofLocalizedDateTime(dateTimeStyle)
  - DateTimeFormatter.ofLocalizedDateTime(dateStyle, timeStyle)
- .withLocale()
  - `dtf.format(dateTimeObject).withLocale(otherLocale)`
- Locale.Category enum specifies
- You can set Locale to display and format in different formats.
- Locale.DISPLAY in one locale but Locale.FORMAT in another

Resource Bundles

- defn: resource bundle - contains the locale-specific objects to be used by a program
- Similar to a map with keys and values
- commonly stored in a .properties file
- .properties files are in key=value format
- Zoo_en
- Zoo_fr
  - var rb = ResourceBundle.getBundle("Zoo", locale)
  - rb.getString("ZOO_TITLE_CASE")
  - rb.keySet()
- .getBundle("name")
- .getBundle("name", locale)
- Bundles are a hierarchy that override the values above if present
- Bundle resolution
  1. Exact _fr_FR
  2. Just request language _fr
  3. default _en_US
  4. default just language _en
  5. no locale (Zoo.properties)
- If resource not in any bundle, then `MissingResourceException` thrown at runtime

Formatting Messages

- `MessageFormat.format("Hello, {0} and {1}", "Tammy", "Henry")`
- Different than `String.format()` ??? Research

Properties Class

- Properties class functions like a `HashMap<String, String>`
- new Properties()
- props.setProperty("name", "San Diego Zoo")
- .getProperty("camel") // null
- props.get() is map.get(). Shouldn't be used. use .getProperty()
- props.getProperty() is map.getOrDefault()
- props.getProperty("missingKey", "defaultValue")

Chapter 17 - Modular Applications

- module-info.java file.
- classpath vs Module path

Reviewing Module Directives

- `exports <package>` - Allows all modules to access the package
- `exports <package> to <module>` - Allows a specific module to access the package
- `requires <module>` - Indicates module is dependent on another module
- `requires transitive <module>` - Indicates the module and all downstream modules are dependent on module
- `uses <interface>` - Indicates that a module uses a service
- `provides <interface> with <class>` - Indicates that a module provides an implementation of a service

Comparing Types of Modules

- Three types of modules:
    1. Named modules
    2. Unnamed modules
    3. Automatic modules

Named Module

- defn: named module - a module containing a module-info file.
- Named modules appear on the module path rather than the classpath.
- A named module has the name defined inside the module-info file.

Automatic Module

- defn: automatic module - a module on the module path but does not contain a module-info file.
- Simply a regular JAR file that is placed on the module path and is treated as a module
- Java "automatically" determines the module name if `Automatic-Module-Name` field is not added MANIFEST.MF
- the code referencing an automatic module treats it as if there is a module-info file present.
- It automatically exports all packages

- It also determines the module name. Either MANIFEST.MF has the value stated, or JAR file name is used (after stripping .jar, version, and converting special characters to "." periods).

Unnamed Modules

- defn: unnamed module - module that appears on the classpath
- Like automatic modules, unnamed modules are regular JARs
- Unlike automatic modules, unnamed modules appear on the classpath (rather than the module path)
- unnamed modules are treated like old code and as second-class citizens to modules.
- An unnamed module does not USUALLY contain a module-info file. If it does, it is ignored
- Unnamed modules do not export any packages to named or automatic modules.
- Unnamed module can read from any JARs on the classpath or module path.
- Unnamed modules are similar to the way "java worked before modules".

Comparing Module Types

- Key Point: code on the classpath can access the module path, but code on the module path can't read from the class path
- Chart 17.3 on page 810

JDK Dependencies

Built-in Modules

- java.base is the most import module. It contains most of the packages
- java.base is available without using the requires directive
- other famous modules `java.logging, java.sql, java.xml, java.desktop`
- most core modules start with `java.` but jdk modules start with `jdk.` (e.g. `jdk.javadoc, jdk.charsets, jdk.jdpes`)

jdeps

- defn: jdeps - gives you information about dependencies
- jdeps can be used to assist in identifying dependencies and problems for projects not yet modularized.
- `jdeps zoo.dino.jar`
- jdeps against a JAR
- jdeps prints the modules that we would need to use the require directives on.
- jdeps also prints what packages are used and their corresponding modules
- summary mode
  - `jdeps -s` or `-summary`
  - if we run jdeps in summary mode, we only see the modules that we would need to use the require directive on.
- jdk internals
  - `jdeps --jdk-internals zoo.dino.jar`
  - `jdeps -jdkinternals zoo.dino.jar`
  - lists any classes that you are using that call an internal API

Migration Applications

- ordering modules
- bottom-up migration
- top-down migration
- how to split up an existing project

Determining the order

- Before you can migrate an application to use modules, you need to know how the libraries in the existing app are structured
- dependency graph
- BOTTOM = Projects that do not have any dependencies
- TOP = Projects with (the most) dependencies
- Arrow points FROM X -> Y. X depends on Y. x requires y. y exports packages.

```
chicken
 ↓
nest
 ↓
egg
```

Bottom-Up migration

- easiest migration approach
- works best when you have the power to convert any JAR files that aren't already modules.
- Steps:
    1. Pick the lowest-level project that has not yet been migrated
    2. Add a module-info.java (including exports and requires directives)
    3. Move this newly migrated named module from the classpath to the module path
    4. Ensure any projects that have not yet been migrated stay as unnamed modules on the class path (instead of automatic)
    5. Repeat with the next-lowest-level project
- in effect, we are moving (unnamed) -> (named)

Top-down migration

- most useful when you don't have control of every JAR file used by your app.
- Steps:
    1. Place all projects on the module path (making all modules automatic)
    2. Pick the highest-level project that has not yet been migrated to modules
    3. Add a module-info file to convert this module from automatic -> named. (remember exports and requires directives)
    4. Repeat with the next-lowest-level project
- in affect, converts everything to automatic

- then, adding module-info files one-by-one making each package (automatic) -> (named)

Cyclic dependencies

- defn: cyclic dependencies - aka circular dependency - when two things directly or indirectly depend on each other.
- if you run into cyclic dependency, just merge the two modules together.
- java will not allow you to compile modules with circular dependencies.
- Cyclic dependencies are allowed between packages, but not modules.

Creating a Service

- defn: service - composed of an interface, any classes that interface references, and a way of looking up implementations of the interface
- Implementations are not part of the service
- defn: service provider interface - "interface/abstract class" that specifies what behavior this service has.
- service provider "interface" can be either an interface or abstract class.
- defn: service locator - finds any classes that implement a service provider interface
- Java provides ServiceLocator class to be a service locator.
- `ServiceLocator<tour> loader = ServiceLoader.load(Tour.class)` where Tour is the interface being `uses`
- ServiceLoader was added to java.util in JDK6, prior to that the basic technology was used in the Service class.

Using a service as a Consumer

- defn: Consumer - aka client - is a module that obtains and uses a service
- defn: service provider - implementation of a service provider "interface"
- Recall it's possible to have multiple implementation classes or modules
- `provides zoo.tours.api.Tour with zoo.tours.agency.TourImpl`

Chapter 18 - Concurrency

- defn: multithreaded processing - execute multiple tasks at the same time.
- Concurrency API was introduced to avoid manually handling Threads

Objects/Interfaces to know

- Thread - Thread holds a task and is used to to execute. thread != Thread.
- Runnable - A task without a return type
- Callable - A task with a return type
- ExecutorService - (Task Manager) Object that manages Thread creation/deletion/recycle
- ScheduledExecutorService - Task Manager that can schedule tasks
- Future - Represents the "result" of a task

Threads

- defn: thread - smallest unit of execution that can be scheduled by the operating system
- defn: process - a group of associated threads that execute in the same shared environment
- defn: task - single unit of work performed by a thread
- Single threaded process vs multithreaded process
- A thread can complete multiple independent tasks, but only one task at a time
- If one thread updates the value of a static object, all threads immediately see the update

Thread Types

- All java applications are multithreaded, even hello world
- defn: system thread - thread created by the JVM and runs in the background of the application.
- Garbage collection is managed by a system thread
- Generally, system threads are invisible
- defn: user-defined thread - a thread created by the developer to accomplish a specific task

Thread Concurrency

- defn: concurrency - the property of executing multiple threads and processes at the same time
- Even single CPU PCs have multiple threads. In general, threads = cpu * 2
- defn: thread scheduler - determines which threads should be currently executing
- defn: round-robin schedule - each available thread receives an equal number of CPU cycles in circular order.
- defn: context switch - process of storing a thread's current state and later restoring the state of the thread to continue execution.
- There is often lost time associated with a context switch
- defn: thread priority - numeric value associated with a thread that is taken into consideration by the thread scheduler when determining which threads should currently be executing.
- In Java, thread priorities are specified as integer values

Runnable

- java.lang.Runnable
- Runnable is a functional interface with a zero argument + void run() method.

```
@FunctionalInterface public interface Runnable {
    void run();
}
```

- Runnable is commonly used to define the task or work a thread will execute.
- `Runnable r = () -> {}` is a valid lambda

Creating Threads

- java.lang.Thread
- First you define a thread with a task, then you start the thread with `Thread.start()`
- Java does not provide any guarantees about the order in which a thread will be processed once it is started.
- Exam Trick: order of thread execution is not often guaranteed.

- Defining Tasks:
    1. Calling Thread Constructor with a Runnable object or lambda
        - `new Thread(new PrintData()).start();`
    2. Creating a class that extends thread
        - `(new ReadInventoryThread()).start();`
- defn: asynchronous - main() method thread does not wait for the results before continuing
- defn: synchronous - the program waits for the thread to finish executing before moving to the next line
- `run()` vs `start()`
    - calling run() on a Thread or Runnable does not actually start a new thread
    - calling .run() manually triggers the task synchronously, calling .start() creates a thread that will call run
- In general, you should implement Runnable instead of extending Thread.

Polling with Sleep

- defn: Polling - process of intermittently checking data at some fixed interval.
- Thread.sleep method requests the current thread to rest for a specified number of milliseconds.
- `Thread.sleep(millis)`
- throws checked `InterruptedException`

Concurrency API

- Concurrency API was created to handle the complicated work of managing threads
- Includes ExecutorService interface, which defines services that create and manage threads for you

Single-Thread Executor

- ExecutorService is an interface
- Executors factory class to create instances of ExecutorService
- `Executors.newSingleThreadExecutor();` method

```
ExecutorService service = Executors.newSingleThreadExecutor();
service.execute(task1)
service.execute(task2)
service.execute(task3)
service.shutdown()
```

Shutting Down a Thread Executor

- Once finished using a thread executor, it is important that you call `.shutdown()`
- Failing to call shutdown will result in your application never terminating
- ExecutorService life cycle (`.isShutdown()` vs `.isTerminated()`)
- Shutting Down is a process
    1. Start rejecting all new tasks while existing tasks finish (isShutdown=true, isTerminated=false)
    2. All existing tasks finish (isShutdown=true, isTerminated=true)
- RejectedExecutionException thrown

- `.shutdown()` does not actually stop any tasks that have already been submitted to the thread executor
- `.shutdownNow()` will ATTEMPT to stop all running tasks. Not guaranteed
- `.shutDownNow()` returns `List<Runnable>` of tasks that were submitted but never started
- ExecutorService does not extend AutoCloseable, so you cannot use try-with-resource

Submitting Tasks

- ExecutorService methods to submit tasks
    1. `.execute()`
        - inherited from Executor
        - Takes a Runnable lambda expression or instance and completes the task asynchronously
        - result of the task is ignored since void return type of Runnable.
        - "fire and forget" method
    2. `.submit()`
        - asynchronously just like .execute()
        - returns a `Future` instance that can be used to determine whether the task is complete
        - can also be used to return a generic result object after the task has been completed
        - overloaded to accept either Runnable or Callable
    3. `.invokeAll()`
    4. `.invokeAny()`
- In general, submit > execute, even if you don't need the Future object

Waiting for Results

- Future instance can be used to determine the result
- Future is an interface
- Future methods
    - `.isDone()`
        - Returns true if the task was completed, threw an exception, or was cancelled
    - `.isCancelled()`
        - Returns true if the task was cancelled before it completed normally
    - `.cancel()`
        - Attempts to cancel execution of the task
        - returns true if it was successfully cancelled
        - returns false if it could not be cancelled or it completed
    - `.get()`
        - Retrieves the result of a task, waiting endlessly if it is not yet available
        - overloaded with two timeout params.
- Checked `TimeoutException` thrown if attempting to get result times out
- `.get()` always returns null when working with Runnable expressions because Runnable::run returns void in the FunctionalInterface
- TimeUnit enum : TimeUnit.DAYS, .HOURS, .MINUTES, .SECONDS, .MILLISECONDS, .MICROSECONDS, .NANOSECONDS

Callable

- java.util.concurrent.Callable
- Functional interface
- Similar to Runnable except it's default method, call(), returns a value

```
@FunctionalInterface public interface Callable<V> {
    V call() throws Exception;
}
```

- Callable is often preferred over Runnable since it allows more details.
- Runnable and Callable are often interchangeable
- ExecutorService includes an overloaded version of submit() that takes a Callable object

```
ExecutorService service = Executors.newSingleThreadExecutor();
Future<Integer> result = service.submit(() -> 30 + 11);
System.out.println(result.get());   // 41
service.shutdown();
```

Waiting for All Tasks to Finish

- `service.awaitTermination(1, TimeUnit.MINUTES);`
- `.awaitTermination()` returns once all tasks finish, waits until timeout, or InterruptedException thrown

Submitting Task Collections

- .invokeAll() and .invokeAny()
- Execute synchronously and take a Collection of tasks
- Will wait until the results are available before returning control to the enclosing program
- invokeAll()
  - executes all tasks in a collection
  - returns `List<Future>` corresponding to each task in the order they were received
- invokeAny()
  - executes a collection of tasks
  - returns the result of one of the tasks that successfully completes execution
  - cancels all unfinished tasks
  - "short circuit"
- ExecutorService also includes overloaded version of invokeAny and invokeAll with timeout values

Scheduling Tasks

- ScheduledExecutorService can be used to schedule tasks
- Sub-interface of ExecutorService
- Just like ExecutorService, instances are provided from factory methods in `Executors`
- `ScheduledExecutorService service = Executors.newSingleThreadedScheduledExecutor()`
- ScheduledExecutorService methods
  - schedule
    - Creates and executes a Callable task after the given delay

        - overloaded to accept Callable or Runnable
        - `service.schedule(task1, 10, TimeUnit.SECONDS)`
    - scheduleAtFixedRate
        - Creates and executes a Runnable task after the given initialDelay
        - Creates a new task every period value that passes
        - creates a new task regardless of whether the previous task finished
        - `service.scheduleAtFixedRate(command, 5, 1, TimeUnit.MINUTES);`
    - scheduleWithFixedDelay
        - Creates and executes a Runnable task after the given initial delay
        - Creates a new task after with the given delay between termination of one execution and the commencement of the next
        - useful when you don't know how long it will take to finish the task
- returns ScheduledFuture instance (nearly identical to Future, except it includes .getDelay() method that returns the remaining delay)

Pools

- defn: thread pool - group of pre-instantiated reusable threads that are available to perform on a set of arbitrary tasks
- Allows multi threading now
- Just like ExecutorService, instances of pools are provided from factory methods in `Executors`
- factory methods
    - newCachedThreadPool
        - return `ExecutorService`
        - Creates a unbounded size thread pool that creates new threads as needed
        - will reuse previously constructed thread when they are available
        - encouraged for executing many short-lived asynchronous tasks
        - strongly discouraged for long-lived processes.
    - newFixedThreadPool(int)
        - return `ExecutorService`
        - Creates a thread pool that reuses a fixed number of threads operating off a shared unbounded queue
        - calling `newFixedThreadPool(1)` is equivalent to `newSingleThreadExecutor()`
    - newScheduledThreadPool(int)
        - return `ScheduleExecutorService`
        - Creates a thread pool that can schedule commands to run after a given delay or to execute periodically.
        - identical to `newFixedThreadPool` except that it returns an instance of ScheduledExecutorService.
- Since they return `ExecutorService`/`ScheduleExecutorService` types, we can submit tasks the same way as above.
- In practice, choosing Thread Pool Size is done by the number of CPUs and background processes running other than Java.

Single Thread vs Pool

- While a single-thread executor will wait for a thread to become available before running the next task
- A pooled-thread executor can execute the next task concurrently.

- If the pool runs out of available threads, the task will be queued by the thread executor and wait to be completed.

Thread Safety

- defn: thread safe - property of an object that guarantees safe execution by multiple threads at the same time.
- defn: race condition - two tasks executing at the same time but the order is incompatible

Atomic Classes

- java.util.concurrent.atomic package
- Atomic classes for cookie-cutter thread-safe primitives.
- defn: Atomic - the property of an operation to be carried out a single unit of execution without any interference by another thread.
- AtomicBoolean, AtomicInteger, AtomicLong
- Atomic methods
  - get()
  - set()
  - getAndSet()
  - incrementAndGet()
  - getAndIncrement()
  - decrementAndGet()
  - getAndDecrement()

Synchronized blocks

- defn: monitor - aka lock - a structure that supports mutual exclusion
- defn: mutual exclusion - property that at most one thread is executing a particular segment of code at a given time.
- defn: synchronized block - a block with a crude built-in lock behavior
- Any Object can be used as a monitor with `synchronized`

```
SheepManager manager = new SheepManager();
synchronized(manager) {
    //
}
//
```

- `synchronized(obj)` to synchronize on a specific
- `synchronized(SheepManager.class)` to synchronize on a specific class
- While other another thread is executing the synchronized block, all threads that arrive there will wait.
- Once a block is finished, the lock is released.
- Synchronizing the task != synchronizing the creation of the threads
- Synchronized block + atomic variables is redundant
- Unlike Locks, we can't find out if a synchronized block is "locked" or not

Synchronized Methods

- Adding synchronized modifier to method declaration

```
private synchronized void doThing() {
    //
}
```

- instance methods or static methods

Lock Framework

- Conceptually similar to using `synchronized` keyword, but with a lot more helper methods
- You can "lock" only an object that implements the Lock interface
- Much safer than synchronized since we can add timeouts
- Many types of Locks, but we only need to know `ReentrantLock` for the exam.

ReentrantLock Interface

- ReentrantLock is the most common form of Lock object
    1. Create an instance of Lock
    2. each thread calls lock() before using the protected code
    3. each thread calls unlock() before exiting the protected code

```
Lock lock = new ReentrantLock();
try {
    lock.lock();
    // protected code
} finally {
    lock.unlock();
}
```

- try/finally not required, but strongly recommended to prevent lock from being locked forever after exception thrown.
- Attempting to unlock an unlocked lock will throw `IllegalMonitorStateException`

Lock methods

- lock()
    - Requests a lock and blocks until lock is acquired (infinite wait if locked)
- unlock()

- Releases a lock - throws exception if already unlocked
- tryLock()
    - Requests a lock and returns immediately.
    - Returns a boolean indicating whether the lock was successfully acquired
    - Does not wait if another thread already holds the lock. returns immediately
    - Overloaded with a timeout parameter to wait up to x time for lock to become available

---

CyclicBarrier

- defn: CyclicBarrier - Barrier that halts threads from continuing until a certain number of threads assemble at the barrier.
- `new CyclicBarrier(4);`
- If ThreadPoolSize < CyclicBarrierLimit, the barrier will never be reached and code will hang indefinitely
- "wait and stop" barrier.
- Number of threads waiting at a cyclic barrier

Concurrent Collections

- Collections framework also supplies collections with concurrency support out of the box.
- defn: memory consistency error - error when two threads have inconsistent views of what should be the same data.
- Conceptually, we want writes on one thread to be available to another thread.
- Java may throw `ConcurrentModificationException` if attempting to modify non-concurrent collections
- At any given moment, all threads should have the same consistent view of a collection

Classical collections

- Removing a key from a map while iterating through the keys.

```
Map<String, Integer> foodMap = new HashMap<>();
foodMap.put("penguin", 1);
foodMap.put("flamingo", 2);
for (String key: foodMap.keySet()) {
    foodMap.remove(key);                  // ConcurrentModificationException
}
```

- fixed using `foodMap = new ConcurrentHashMap<>();`

Concurrent Classes

- No point in using Concurrent classes if you are using an immutable or read-only object.
- Good practice to use a nonconcurrent reference to a concurrent object
- classes

- ConcurrentHashMap
- ConcurrentLinkedQueue
- ConcurrentSkipListMap
- ConcurrentSkipListSet
- CopyOnWriteArrayList
- CopyOnWriteArraySet
- LinkedBlockingQueue

- These objects are safe to pass to multiple threads

SkipList Collections

- SkipList means sorted (Tree)
- SkipList classes (ConcurrentSkipListSet and ConcurrentSkipListMap) are the concurrent versions of TreeSet and TreeMap
- They maintain their elements in natural order.

CopyOnWriteCollections

- CopyOnWriteArrayList and CopyOnWriteArraySet
- These classes copy all of their elements to a new underlying structure anytime an element is added/modified/removed.
- Modifying an individual element doesn't change the reference so a new structure is not allocated
- "snapshots"
- Any iterator established prior to a modification will not see the changes.
- CopyOnWrite classes can use a lot of memory since a new collection is allocated anytime collection is modified

Blocking Queues

- LinkedBlockingQueue implements BlockingQueue interface
- BlockingQueue is just like a regular Queue except that it includes methods that will wait TimeUnit to complete an action
  - offer
  - poll
- Waiting the specified time and returning false if the time elapses before space is available
- Checked InterruptedException must be handled or declared

Converting nonconcurrent collection to ConcurrentCollection

- Concurrency API includes methods for obtaining synchronized versions of existing noncurrent collection
- Synchronized collection != concurrent collection
- Unlike concurrent collections, the synchronized collections also throw ConcurrentModificationException
- Collections class
  - .synchronizedCollection
  - .synchronizedList
  - .synchronizedMap
  - .synchronizedSortedMap
  - .synchronizedNavigableMap
  - .synchronizedSet
  - .synchronizedSortedSet

    - .synchronizedNavigableSet

Threading Problems

Liveness

- defn: liveness - the ability of an application to be able to execute in a timely manner.
- Liveness problems are problems where the app becomes unresponsive or "stuck"
- Three types of liveness problems:
    1. Deadlock
    2. Starvation
    3. Livelock

Deadlock

- defn: deadlock - two or more threads are blocked forever waiting on each other.

Starvation

- defn: starvation - when a single thread is perpetually denied access to a shared resource or lock.
- The thread is still active, but is unable to complete its work.

Livelock

- defn: livelock - two or more threads are conceptually blocked forever, although they are each still active and trying to complete their task.
- Livelock is a special case of resource starvation.
- Livelock is often a result of two threads trying to resolve a deadlock.

Race Condition

- defn: race condition - undesirable result that occurs when two tasks, which should be completed sequentially, complete at the same time.
- Race conditions can lead to invalid data.
- Race conditions tend to appear in highly concurrent applications (like websites!)

Parallel Streams

- One of the most powerful features of the Stream API is built-in concurrency support
- defn: serial stream - stream in which results are ordered, with only one entry being processed at a time.
- defn: parallel stream - stream capable of processing results concurrently using multiple threads
- Parallel streams generally improve performance
- Parallel streams can change output though (e.g. findFirst)

Creating Parallel streams

- Two ways

1. From existing stream
   - calling `.parallel` on existing stream to convert it to one that support multithreaded processing.
   - .parallel is an intermediate operation.
   - `List.of(1,2).stream.parallel()`
2. From collection object
   - `.parallelStream()`
   - Collection interface includes a method .parallelStream()
   - `List.of(1,2).parallelStream()`

Parallel Decomposition

- defn: parallel decomposition - the process of taking a task, breaking it up into smaller pieces that can be performed concurrently, and then reassembling the results.
- Order is not guaranteed
- We can guarantee order with .forEachOrdered() on a parallel stream, at the tradeoff of some performance losses.

Parallel Reductions

- reduction operations on parallel streams are referred to as parallel reductions
- .findAny may result in unexpected behavior
- `List.of(1,2,3).parallelStream().findAny().get();` // could be any value 1 2 or 3
- Since there's overhead with parallelization, sometimes it's slower in practice than a serial stream
- .reduce()
  - parallel streams are why we needed a third reduce overload
  - We needed a combiner as well as an accumulator because , in parallel, the results are computed then combined
  - "Intermediate values" and then combining.
  - golden rule: make sure the accumulator and combiner work regardless of the order they are called in.
  - for parallel streams, the three arg .reduce() method is better than the one or two arg .reduce methods for performance reasons
- .collect()
  - Make sure to use a concurrent collection to combine the results or else `ConcurrentModificationException`
  - Failing to use the right collection will result in parallel stream performing like a serial stream
- defn: stateful lambda expression - lambda whose result depends on any state that might change during exception of the pipeline

---

Chapter 19 - I/O (old)

- java.io API to interact with files and streams.
- I/O streams are completely unrelated to Functional Programming streams

Fundamental Classes/Interfaces:

- File

- InputStream and OutputStream
- Reader and Writer
- FileInputStream and FileOutputStream
- BufferedInputStream and BufferedOutputStream
- ObjectInputStream and ObjectOutputStream
- BufferedReader and BufferedWriter
- FileReader and FileWriter
- PrintStream and {}
- InputStreamReader and OutputStreamWriter

File System Basics

- defn: a file - record within the storage device that holds data
- defn: directory - a location that can contain files as well as other directories
- We use many of the same classes to operate on directories as files
- defn: file system - system responsible for organizing files and directories
- defn: root directory - topmost directory in the file system
- defn: path - a String representation of a file or directory within a file system
- Absolute vs relative paths

Bytes

- Data is stored in a file system (and memory) as bits (0 or 1).
- defn: byte - set of 8 bits

File Class

- java.io.File can represent a file or directory
- java.io.File class is one of the most commonly used class in the java.io API
- java.io.File is used to read information about existing files and directories, ls, create/delete files&directories
- An instance of a File class represents the path to a particular file or directory
- java.io.File cannot read or write data within a file, but it can be passed as a reference to classes that can
- defn: absolute path - full path from the root directory
- defn: relative path - path from current working directory
- `java.io.File.separator == System.getProperty("file.separator")`
- Three constructors
  1. `public File( String pathname)`
  2. `public File( File parent, String pathname)`
  3. `public File(String parent, String pathname)`
- java.io.File only represents a path to a file. It is not connected to an actual file
- EXAM TRICK: `/data/zoo.txt` is not necessarily a file. directories can be misnamed with extensions to be confusing.

File methods

- Methods
  - boolean delete()
    - Deletes the file or directory and returns true only if successful
    - if directory, directory must be empty to be deleted

  - boolean exists() - returns true if a file exists
  - String getAbsolutePath - Returns the absolute path
  - String getName() - Returns the name of the file or directory
  - String getParent()
    - Returns the parent directory that the path is contained in
    - Returns null if root
  - boolean isDirectory() - Return true if a File object is a directory
  - boolean isFile() - Returns true if a File object is a file
  - long lastModified - Returns the number of milliseconds since the file was last modified
  - long length() - returns the number of BYTES in the file
  - File[] listFiles() - if directory, return list of files within directory
  - boolean mkdir()
  - boolean mkdirs() - Creates the directory named by this path and any intermediate parent directories needed
  - boolean renameTo(File destination)
    - Renames the file or directory to "destination"
    - Returns true only if successful

I/O Streams

- I/O = Input/Output

I/O Stream fundamentals

- defn: stream - list of data elements presented sequentially
- We don't know when a stream will end
- Normal vs Buffered streams
- Writing a file one byte at a time is time consuming. The round-trip from jvm to file system is expensive.
- BufferedOutputStream groups bytes together and makes less trips
- Although commonly used with file I/O, streams are more generally used to handle reading/writing from sequential data sources.
- Submitting data to a website with an output stream and reads the results from an input stream
- Streams allow reading/writing to files that are massive (100TB) because the entire file is not loaded into memory
- Only a small "window" of a stream is visible at a time

Byte vs Character Streams

- java.io API defines two sets of streams for reading/writing:
  1. byte streams
     - read/write binary data (0s and 1s)
     - have class names that end in `InputStream` or `OutputStream`
     - primarily used to work with binary data, such as image or executable file
  2. character streams
     - read/write text data
     - have class names that end in `Reader` or `Writer`
     - primarily used to work with text files.
     - convenience methods for working with text.

- e.g. FileInputStream vs FileReader
- Some streams don't have the word "Stream" in the name (e.g. FileInputReader). They are still I/O streams
- Byte streams can still work with text files since text is still binary.
- defn: character encoding - determines how characters are encoded and stored in bytes in a stream. (UTF-16)
- `Charset utf8Charset = Charset.forName("UTF-8");`

Input vs Output Stream

- Most input stream classes have corresponding output stream classes
- FileInputStream vs FileOutputStream
- Most Reader classes have corresponding Writer classes.
- NOTE: `PrintWriter` has no accompanying Reader class.
- NOTE: `PrintStream` has no accompanying InputStream class

Low-Level vs High-Level Streams

- defn: low-level stream - connects directly with the source of the data
- Low level streams process the raw data or resource and are accessed in a direct and unfiltered manner
- FileInputStream is a low level stream that reads file data one byte at a time
- defn: high-level stream - stream built on top of another stream using wrapping
- defn: wrapping - process by which an instance is passed to the constructor of another class and operations on the resulting instance are filtered and applied to the original instance
    - example: `new BufferedReader(new FileReader("zoo.txt"))`
- Streams can be wrapped multiple times
- Outer wrapper provides new logic and reuses underlying logic
- Buffered classes should almost always be used for performance reasons.

Stream Base Classes

- java.io library defines four abstract classes that are the parents of all stream classes
    1. InputStream - input byte streams
    2. OutputStream - output byte streams
    3. Reader - input character streams
    4. Writer - output character streams
- We can't instantiate these abstract classes
- Constructors of high-level streams often take a param of the base classes.
- EXAM TRICK: Passing the wrong base class as a parameter to the constructor of high level stream
- `new BufferedInputStream(new FileReader("zoo.txt"));` // DOES NOT COMPILE

I/O Stream Class Names

- Recapping the types of classes so far
- Properties
    - A class with "InputStream"/"OutputStream" in its name is used for reading/writing binary/byte data
    - A class with "Reader"/"Writer" in its name is used for reading/writing character or string data
    - Most, but not all, input classes have a corresponding output class
    - low-level streams connect directly with the source of the data
    - high-level streams are built on top of another stream using wrapping
    - A class with Buffered in its name reads/writes data in groups of bytes or characters

- With few exceptions, you only wrap a stream with another stream if they share the same abstract parent

---

Common I/O Stream Operations

Reading and Writing Data

- Both InputStream and Reader classes define `.read()` methods
- Both OutputStream and Writer classes define `.write()` methods
- .read()
    - read(int b) - read a byte at a time
    - read(byte[] b) - read multiple bytes
    - read(byte[] b, int offset, int length) - read bytes from an array from `b[offset] to b[offset + length]`
    - read(char[] c) - read multiple char
    - read(char[] c, int offset, int length) - read char from an array from `c[offset] to c[offset + length]`
- .write()
    - write(int b) - write a byte at a time
    - write(byte[] b) - write multiple bytes
    - write(byte[] b, int offset, int length) - write bytes from an array from `b[offset] to b[offset + length]`
    - write(char[] c) - write multiple char
    - write(char[] c, int offset, int length) - write char from an array from `c[offset] to c[offset + length]`
- `-1` indicates the end of a stream.
- `while ((b = inputStream.read()) != -1) {outputStream.write(b);}`
- Most I/O streams declare a checked IOException

Closing the stream

- All I/O streams implement Closeable
- All I/O streams include a method to release any resource within the stream
- `public void close() throws IOException`
- Resource leaks if not closed. Locked files if not closed.
- try-with-resource
- If passing streams as parameter, the method that created the stream should be the one closing it, not the inner method
- When closing a wrapped stream, calling `.close()` on the topmost object will cascade close all lower objects

Manipulating Input Streams

- All input streams classes include methods to manipulate the order in which data is read from a stream
- `.markSupported()`
    - returns true if method `.mark()` is supported\
    - good practice to call `markSupported()` before calling mark() or reset()
- `.mark()`
    - mark the current position in the stream
- `.reset()`
    - Attempts to reset the stream to the mark()'d position
- `.skip(long n)`

- skip the next x bytes/characters
    - basically reads data from the stream and discards the contents
- `.flush()`
    - flushes buffered data through the stream
    - requests that all accumulated data in memory be written immediately to disk
    - JVM is free to call flush whenever it wants.
    - helps reduce data loss if application crashes
    - close() also calls flush()

Working with I/O Stream Classes

- The I/O stream classes include hundreds of overloaded constructors and methods

Reading and Writing Binary Data

- Constructors
    - FileInputStream(File file)
    - FileInputStream(String name)
    - FileOutputStream(File file)
    - FileOutputStream(String name)
    - All 4 throw FileNotFoundException
- If you need to just append, FileOutputStream class includes overloaded constructor with a boolean append flag

```
try (var in = new FileInputStream(srcFile); var out = new FileOutputStream(destFile)) {
    int b;
    while((b = in.read()) != -1) out.write(b); // copy a file
}
```

Buffering Binary Data

- Recall the poor performance of using non-buffered streams
- As high level streams, BufferedInputStream and BufferedOutputStream that take other streams as input.
- Constructors
    - BufferedInputStream(InputStream in)
        - `new BufferedInputStream(new FileInputStream(srcFile))`
    - BufferedOutputStream(OutputStream out)
        - `new BufferedOutputStream(new FileOutputStream(destFile))`
    - NOTE: InputStream and OutputStream are abstract base classes and thus allowing many different params
- Choosing buffer size using powers of 2
- `out.write(buffer, 0, lengthRead)`
- `out.flush()`

Reading and Writing Character Data

- `FileReader` and `FileWriter` classes, along with their associated buffer classes, are among the most convenient I/O classes
- Built-in support for text data
- Constructors
    - FileReader(File file)
    - FileReader(String name)
    - FileWriter(File file)
    - FileWriter(String name)
    - All 4 throw checked FileNotFoundException

```
try (var reader = new FileReader(srcFile); var writer = new FileWriter(destFile)) {
    int b;
    while((b = reader.read()) != -1) writer.write(b); // copy a file
}
```

- FileWriter inherits a new overloaded method: `.write(String str)`

Buffering Character Data

- BufferedReader and BufferedWriter
- High-level "wrapped" buffered character streams
- constructors
    - BufferedReader(Reader in)
    - BufferedWriter(Writer out)
    - take existing Reader and Writer instances as params
- two new methods:
    - readLine()
        - returns a line as a String
        - strips out line break character
    - newLine - /n

```
try (var reader = new BufferedReader(new FileReader(srcFile));
    var writer = new BufferedWriter(new FileWriter(destFile))) {
        String s;
        while ((s = reader.readLine()) != null) {
            writer.write(s);
            writer.newLine();
        }
    }
```

- Reading and writing lines at a time instead of bytes
- Temporarily storing data in a String before writing.

- Checking for the end of the stream with a `null` value instead of `-1`

Serializing Data

- defn: serialization is the process of converting an in-memory object to a byte stream
- defn: deserialization - the process of converting from a byte stream to an object.
- To serialize an object using the I/O API, the object must implement the java.io.Serializable interface
- Serializable is a "marker interface"
- defn: marker interface - an interface that does not have any methods
- Any class can implement the Serializable interface
- The purpose of implementing Serializable is to inform any process attempting to serialize the object that you built support to make the object serializable.
- defn: transient - a field that is marked transient will not be saved to a stream when the class is serialized
- e.g. making password fields transient.
- Good practice to declare a static `serialVersionUID` variable in every class that implements Serializable
- Every time the class structure changes, `serialVersionUID` should be updated/incremented.
- How to make a class Serializable:
  1. Class must be marked Serializable by implementing Serializable
  2. Every instance member of the class is serializable, marked transient, or has a null value at the time of serialization

Storing data with ObjectOutputStream and ObjectInputStream

- ObjectInputStream class is used to deserialize an object from a stream
- ObjectOutputStream is used to serialize an object to a stream
- They are both high-level streams that operate on existing streams
- Constructors
  - `ObjectInputStream(InputStream in)`
  - `ObjectOutputStream(OutputStream out)`
  - both throw IOException
- Contain a lot of methods
- Two methods need to know for the exam:
  - `.readObject()`
    - throws IOException, ClassNotFoundException`
  - `.writeObject(Object obj)`
    - throws IOException

```


// write
try (var out = new ObjectOutputStream(new BufferedOutputStream(new
FileOutputStream(dataFile)))) {
    for (Gorilla gorilla : gorillas) out.writeObject(gorilla);
}

// read
try (var in = new ObjectInputStream(new BufferedInputStream(new
FileInputStream(dataFile)))) {
    while(true) {
        var object = in.readObject();
        if (object instanceof Gorilla) {
            gorillas.add((Gorilla) object);
        }
    }
} catch (EOFException e) {
    // file end reached
}
```

- Unlike with our readers, `-1` and `null` can't be used to denote the end of the file.
- EOFException when the end of the stream is reached

Understanding the Deserialization Creation Process

- When you deserialize an object, the constructor of the serialized class, along with any instance initializers, is not called when the object is created.
- Java will call the no-arg constructor of the first non-serializable parent class it can find in the class hierarchy, usually Object
- ??? revisit this section

Printing Data

- PrintStream and PrintWriter are high level output print streams classes
- Both `System.out` and `System.err` are PrintStream objects.
- Useful for writing text data to a stream.
- PrintStream and PrintWriter are the only I/O streams classes on the exam that do not have corresponding input stream classes
- PrintStream also doesn't follow the "Output in Class name" rule for OutputStreams
- Constructors:
  - `PrintStream(OutputStream out)`
  - `PrintStream(File file)` - convenience constructor that creates the inner stream for you
  - `PrintStream(String fileName)`
  - `PrintWriter(Writer out)`
  - `PrintWriter(File file)`
  - `PrintWriter(String fileName)`
  - `PrintWriter(OutputStream out)` - RARE EXCEPTION: mixing byte stream with character stream

PrintStream methods

- Besides the inherited .write() methods, the print stream classes include numerous methods for writing data.
- Unlike the majority of I/O streams we've covered, these methods do not throw any Checked Exceptions
- Methods:
  - print()
    - calls `myStream.write(String.valueOf(param))` under the hood
    - numerous overloaded methods for various primitives
  - println()
    - print() + line break
  - format()
    - format(String format, Object args...)

        - format(Locale loc, String format, Object args...)
        - equivalent to `printf()`
        - %s (any type)
        - %d (int or long)
        - %f (float or double)
        - %n (inserts line break)
        - using the wrong % causes IllegalFormatConversionException
        - `System.out.format("Hello %s", "Bob")`
        - using `%5.2f` to specify format of floating points. rounding/truncating ?

User input

- java.io API includes numerous classes for interacting with the user.

Printing data to the user

- Java includes two PrintStream instance for providing information to the user: System.out and System.err
- Syntax for calling System.err is the same as System.out
- System.err is sent to a different text stream than `System.out` and often sent to different log files
- IDEs often print System.err in red

Reading Input as a stream

- `System.in` returns an InputStream and is used to retrieve text input from the user
- `var reader = new BufferedReader(New InputStreamReader(System.in));`
- `reader.readLine()`

Closing System Streams

- Closing System.out or System.err is not recommended
- Static Streams created by the JVM and shared everywhere
- Don't wrap in try-with-resource
- IOException thrown when attempting to operate on a closed stream.

Acquiring Input with Console

- java.io.Console specifically designed to handle user interactions
- Console class is a singleton class (all private constructors, one factory method call)
- Only one instance of Console and it is created by the JVM.
- `Console console = System.console();`
- methods
    - reader
        - returns Reader
        - Analogous to calling System.in
    - writer
        - returns PrintWriter
        - Analogous to calling System.out
    - format

        - convenience method for .writer().format()
        - does not support Locale overload like print stream classes
    - readLine
        - reads input until user presses the Enter key
    - readPassword
        - similar to readLine
        - returned as a char[] instead of a String

Chapter 20 - NIO.2

- NIO = Non-blocking Input/Output API
- allows us to do a lot more with files and directories than the original java.io API

Introducing NIO.2

- NIO.2 is a replacement for the legacy java.io.File class
- more intuitive, more feature-rich API for working with files and directories
- The NIO.1 API was never popular and is not on the exam.

Path

- java.nio.file.Path interface
- A path instance represents a hierarchical path on the storage system to a file or directory
- Path in NIO.2 is sort of a replacement for java.io.File
- Just like java.io.File, relative or absolute path
- Just like java.io.File, file or directory
- Unlike java.io.File, Path interface contains support for symbolic links.
- defn: symbolic link: special file within a file system that serves as a reference or pointer to another file or directory.
- Files (plural!) operates on Path instances, not File instances.

Creating Paths

- Since Path is an interface, we can't instantiate it directly.
    - Path.of(String first, String... more)
        - static factory method
        - varargs to pass additional path elements. If varargs, then path = varargs.joining("/")
- Paths class
    - Paths plural is a utility class
    - Paths.get(String first, String... more)
        - same vararg joining rules
- URI object
- defn: URI - Uniform resource identifier
    - URI constructor
        - new URI(String str)

  - Converting URI -> Path
    - Path.of(URI uri)
    - Paths.get(URI uri)
    - pathInstance.toURI()
- FileSystem abstract and FileSystems utility class
  - FileSystems.getDefault()
    - returns FileSystem
  - fileSystemInst.getPath(String first, String... more)
  - connecting to other file System
    - .getFileSystem(URI uri)
- Path from java.io.File
  - pathInstance.toFile()
    - returns File
  - fileInstance.toPath()
    - returns Path

Applying Path Symbols

- defn: path symbol - reserved series of characters that have special meaning with some file systems.
- "." - current directory
- ".." - parent of current directory

Providing Optional Arguments

- Many of the methods in this chapter include a varargs that takes an optional list of values
- Enum Type - LinkedOption, StandardCopyOption, StandardOpenOption, FileVisitOption
- Table page 978

Paths

- Path instances are immutable
- Many of the methods available in Path interface transform the path instance and return a new Path object (like String).

Path methods

- retrieving basic info
  - toString()
    - returns a string representation of the entire path
  - getNameCount()
    - number of directories/file in path (not including root directory `/` as part of the path)
  - getName(int index)
    - get the ith directory/file in the path
    - Doesn't include root. .getName(0) throws IllegalArgumentException
- Creating a new Path
  - .subpath(int beginIndex, int endIndex)
    - Returns a portion of a Path as a new Path instance

        - 0-indexed and does not include root. Index must be in range or else IllegalArgument
        - `[beginIndex, endIndex)` // exclusive
        - `Path.of("/a/b/c/d/e").subPath(0.3)` // "a/b/c"
- Accessing Path Elements
    - getFileName()
        - returns the Path element of the current file or directory
    - getParent()
        - returns the full path of the containing directory
        - returns null if called on root or at the top of a relative path
    - getRoot()
        - returns root element of file within the file system or null if the path is a relative path
- Checking Path Type
    - isAbsolute()
        - returns true if the path the object references is absolute. false if relative
    - toAbsolutePath()
        - converts relative Path object to an absolute Path object by joining it to the cwd. if already absolute, return this
- Joining Paths
    - resolve(Path otherPath)
    - resolve(String other)
    - concatenate paths
    - does not clean up path symbols
    - Intelligently mixes absolute paths. Does not just simply concatenate
- Deriving a path
    - relativize()
        - constructing the relative path from one Path to another, often using path symbols
        - Exactly what we do with Xpath with Selenium.
        - Can't mix relative and absolute paths, both must match, else IllegalArgumentException
- Cleaning up a path
    - normalize()
        - eliminate unnecessary redundancies in a path
        - Doesn't eliminate all path symbols, only redundant path symbols
- Retrieving the File System Path
    - toRealPath()
        - toRealPath(LinkOption... option)
        - Returns an absolute path from the current path object by eliminating redundant path symbols (../.)
        - Throws exception if the path does not exist

Operating on Files and Directories

- The Files helper class is capable of interacting with real files and directories within the system.
- Most of the methods take optional params and throw IOException if the path does not exist.
- Files class also replicates numerous methods in java.io.File
- Files operators on path objects, not File objects like the naming plural convention might suggest

Files Methods

- All of these methods declare IOException (except exists())
- Checking for Existence
  - .exists()
    - checks whether the file exists
- Testing Uniqueness
  - .isSameFile()
    - boolean isSameFile(Path path, Path path2)
    - return true if two paths are same file(or directory!)
    - follows symbolic links
- Making Directories
  - .createDirectory()
    - Path createDirectory(Path dir, FileAttribute<?>... attrs)
    - create a directory
    - throws exception if already exists or the paths leading up to the directory do not exist
  - .createDirectories()
    - Path createDirectories(Path dir, FileAttribute<?>... attrs)
    - creates a directory along with any nonexistent parent directories leading up to the path
- Copying Files
  - .copy()
    - `Path copy(Path source, Path target, CopyOptions... options)`
    - copy file or directory from one location to another
    - shallow copy
    - By Default, if target already, copy() with throw an exception. Else, StandardCopyOption.REPLACE_EXISTING.
    - `long copy(InputStream in, Path target, CopyOptions... options)`
      - reads the contents of a stream and writes the output to a file
    - `long copy(Path source, OutputStream out)`
      - reads the contents of a file and writes the output to a stream.
- defn: shallow copy - files and subdirectories with the directory are not copied
- defn: deep copy - the entire tree is copied, including all of its content and subdirectories
- Moving or Renaming Paths
  - .move()
    - Path move(Path source, Path target, CopyOption... options)
    - moving or renaming files/directories. Atomic move
- Deleting a File
  - .delete()
    - throws exception if path does not exist
  - .deleteIfExists()
    - Returns true if file exists and was deleted.
  - directories must be empty to be deleted
  - if the path is a symbolic link, only the symbol is deleted.
- Reading and Writing Data
  - newBufferedReader()
  - newBufferedWriter()
  - Two convenient methods
  - overloaded with option to supply Charset parameter
- Replacing a File

- .readAllLines()
  - `List<String> readAllLines(Path path)`
  - Returns a list of lines
  - If the file is huge, OutOfMemoryError

Managing File Attributes

- defn: file attributes - file/directory metadata
- size, visibility, etc.
- methods
  - .isDirectory()
  - .isSymbolicLink()
  - isRegularFile()
    - defn: regular file - file that can contain content, as opposed to a symbolic link, directory, resource
    - follows symbolic links
  - .isHidden()
  - .isReadable()
  - .isWritable()
  - .isExecutable()
  - .size()
    - size of the file in bytes
    - undefined for directories
  - .getLastModifiedTime()
    - returns a `FileTime` object (which represents a timestamp)
    - FileTime.toMillis() for epoch time

Improving Attribute Access

- Retrieving attributes one at a time is slow. NIO2 provides a way of getting all file attributes for directories.
- defn: view - group of related attributes for a particular file system type.
- readAttributes
  - `readAttributes(Path path, Class<A> type, LinkOptions... Options)`
  - getFileAttributeView()

Applying Functional Programming

- file operations with `Stream<T>` api methods
- Streams of Paths need to be closed

Files::list

- `Files.list()` is `ls` and is similar to java.io.File::listFiles , but different return type (`Stream<Path>` vs File[])
- `Stream<Path> list(Path dir)`
- List the contents of a directory

Traversing a Directory Tree

- defn: Traversing a directory, aka walking a directory tree, is the process over recursively iterating of descendants
- NIO.2 Streams API methods us depth-first searching (not BFS!) with a depth limit.
- two methods
  - .walk(Path start, FileVisitOption... options)
  - .walk(Path start, int maxDepth, FileVisitOption... options)
  - default maximum depth is `Integer.MAX_VALUE`. depth = 0 means just the cwd
- Symbolic links can cause tree traversal to have loop/cycle
- FileVisitOption.FOLLOW_LINKS defaults to false
- .find()
  - similar to .walk() , but takes a predicate to filter the data
- .lines()
  - similar to .readAllLines(), but returns a `Stream<String>`
  - solves the memory issues that .readAllLines has since streams are lazy evaluated

---

---

three or four technical achievements from you.

- Amazon

- Few engineers can brag about being a "10x engineer", and even fewer have the evidence to support it. While at Amazon, I delivered over 330 Pull Request("PRs", code submissions) in a 9 month period. In effect, I more than 10x'd the second top contributor of my team and more than doubled the entire output of the rest of my team combined. These accomplishments continue to pay off to this day by both reducing manual testing efforts before a release as well as catching bugs earlier in the development life-cycle.

- Amazon

- While working on Amazon Music's Android and iOS app, I led the manual testing effort for part of one of the largest visual overhauls in the app's history. Through both my coordination and implementation, we delivered a highly-stable and largely bug free update for tens of millions of users.

- Fitch

- While at Fitch Ratings, I single-handidly created a new automation test suite for a $500 million a year product. This new test suite leveraged new and exiting technologies to validate the data integrity of financial data files distributed via SFTP. Ultimately, we saw a 40% decrease in the number of bugs discovered in production in the 6 months after the new framework was dialed up.

Chapter 21 - JDBC

- defn: JDBC - Java Database Connectivity
- JDBC works by sending a SQL command to the database and then processing the response
- EXAM UPDATE: - CallableStatement is removed from the exam

Intro Relational Databases and SQL

- defn: database - an organized collection of data
- defn: relational database - database that is organized into tables(rows and columns)

- Persistence is not on the exam
- defn: SQL - Structured Query Language - a programming language used to interact with database records.
- defn: primary key - a unique way to reference each row
- basic SQL:
  - create - INSERT
  - read - SELECT
  - update - UPDATE
  - delete - DELETE
- SQL syntax is out of scope

JDBC Interfaces

- For the exam, we need to know five key interfaces of JDBC, java.sql
- With JDBC, the concrete classes come from the JDBC drivers provided by the creators of the databases
- The concrete classes implement the key interfaces, allowing code to work with different databases.
- 5 key interfaces
  1. Driver - establishes a connection to the database
  2. Connection - sends commands to a database
  3. PreparedStatement - Executes a SQL query
  4. CallableStatement - (REMOVED FROM EXAM)
  5. ResultSet - Reads results of a query

Connecting to a Database

- Connecting to a database requires a third party JAR downloaded.
- `jdbc:postgres://localhost:5432/zoo`
- `jdbc:mysql://localhost:3306`
- `jdbc:oracle:thin:@123.123.123.123:1521:zoo`
  1. Protocol - "jdbc" always
  2. Subprotocol - Product/Vendor name
  3. sub-name - database specific connection details
- sub-name often includes a URI/IP address
- Separated by colons
- Don't need to memorize the sub-name formats.

Getting a Database Connection

- Two main ways to get a Connection
  1. DriverManager
  2. (Not on the exam) DataSource
- DriverManager uses the factory pattern with a static method to return a Connection
- `DriverManager.getConnection("jdbc:postgres://localhost:5432/zoo")`
- `DriverManager.getConnection("jdbc:postgres://localhost:5432/zoo", "username7", "password123")`
- Overloaded to pass "username", "password" optionally
- Checked SQLException

PreparedStatement

- Statement interface (not in scope)
- Sub-interfaces PreparedStatement and CallableStatements
- Statement can be used directly, but PreparedStatement is better for : 1. Performance 2. Security 3. Readability 4. Future use
- PreparedStatements prevent SQL Injection, but Statements do not
- `try(conn.prepareStatement("SELECT * FROM a")) {}`
- PreparedStatement represents a SQL statement that you want to run using conn.
- PreparedStatement does not actually execute the query
- String param with the SQL query is mandatory parameter.
- `.executeQuery()` is used for SELECT, `.exceuteUpdate()` is used for DELETE, INSERT, UPDATE

```
try (PreparedStatement ps = conn.prepareStatement(insertSQLQuery)) {
    int numberOfChangedRows = ps.executeUpdate();
}
```

- Each distinct SQL statement needs its own `conn.prepareStatement()` factory method call

```
try (var ps = conn.prepareStatement(sqlSelectString) ; ResultSet rs = ps.executeQuery())
{
    //
}
```

- PreparedStatement methods:
  - `.executeQuery()`
    - SELECT only
    - returns ResultSet
    - `ResultSet rs = ps.executeQuery()`
  - `.executeUpdate()`
    - UPDATE, DELETE, INSERT
    - returns int (number of rows changed)
    - `int numberOfChangedRows = ps.executeUpdate();`
  - `.execute()`
    - any type (SELECT, UPDATE, DELETE, INSERT)
    - returns a boolean
    - returns true if the query would return a ResultSet instead of an integer
    - `boolean isResultSet = ps.execute()`

Parameters in PreparedStatements

- defn: bind variable - placeholder that lets you specify the actual values at runtime

- Bind variables are like parameters to PreparedStatements.

```
String sqlString = "INSERT INTO names VALUES(?, ?, ?)";
try (PreparedStatement ps = conn.prepareStatement(sqlString)) {
    ps.setInt(1, 1337);
    ps.setString(3, "frank");
    ps.setInt(2, 69420);
    ps.executeUpdate();
}
```

- Use .setInt(), .setString(), .setDouble(), .setObject(), setBoolean() to set the bind variables
- EXAM TRICK: bind variables are 1-indexed! JDBC starts counting columns with 1.
- Bind variables can be set out-of-order
- All bind variables must be set before the query is executed else SQLException
- You can reuse a PreparedStatement object in a for loop and resetting bind variables

ResultSet

- Recall calling .executeQuery() will return a ResultSet if the sqlString is a SELECT statement
- ResultSets have a cursor
- Always use loop or if statement for ResultSets when calling `rs.next()`
- defn: cursor - pointer to the current location in the data
- `while (rs.next()) {}`
- The cursor points to a current row and we retrieve that row's columns using `.getInt()` , `.getString()`, etc.
- .getString("middleName") or .getString(3) // 3 is the index of middleName
- Using index out of bounds or a column name that doesn't exist throws SQLException (exam trick: .getInt("count"))
- Column name is better because it provides context and immune to changes in Query ordering

```
try (var ps = conn.prepareStatement("SELECT id, name FROM name_table) ; ResultSet rs =
ps.executeQuery()) {
    while(rs.next()) {
        int id = rs.getInt(1);    // 1-indexed!
        String name = rs.getString("name")
    }
}
```

CallableStatements

- CallableStatement are removed from the exam

Closing Database Resources

- JDBC resources such as Connection are expensive to create.
- Not closing them creates a resource leak.
- JDBC resources need to closed in a specific order (reverse order of creation)
  1. ResultSet
  2. PreparedStatement
  3. Connection
- Closing a Connection also cascade closes PreparedStatements(which also cascade closes ResultSets)
- JDBC also automatically closes a ResultSet when you run another SQL statement using the same PreparedStatement

---

Chapter 22 - Security

- The exam only covers JAVA SE apps, not web apps

Designing a Secure Object

- techniques:
  1. Limiting Accessibility
     - Principle of least privilege
  2. Restricting Extensibility
     - Preventing classes from being inherited.
  3. Creating Immutable Objects
     - Marking classes as final, marking instance variables private, removing setters in favor of constructor.
     - Getters returning copies of instance fields instead of the original reference.
  4. Cloning Objects
     - Cloneable interface default (not abstract!) .clone() method
     - .clone() makes a shallow copy by default.
     - overriding .clone() with unique implementation

Injection and Input Validation

- defn: Injection - an attack where dangerous input runs in a program as part of a command.
- defn: exploit - an attack that takes advantage of weak security.
- SQL Injection:
  - PreparedStatement > string concatenation/formatting
  - PreparedStatements without bind variables is just as bad as concatenation
- Command Injection:
  - uses operating system commands to do something malicious
  - sending ".." as a file name
  - using whitelist/blacklist of allowed/denied values
  - whitelist is better than blacklist because you don't have to forsee all possible issues.

Confidential Information

- avoid putting confidential information in a `.toString()` method, stack traces, error messages, writing files.

- defn: dump file - a file containing values of everything in memory
- Storing passwords in char[] instead of String to avoid the String Pool
- null out values instead of waiting for garbage collection

Limiting File Access with Security Policy

- Security Policy explicitly gives certain read/write/execute permissions to each file.
- don't need to know how to write or run a policy.

Serializing and Deserializing Objects

- Specifying which fields to Serialize
- Don't serialize sensitive inofrmation with `transient` optional specifier
- Encrypting fields that you Serialize
- readResolve() - runs after the readObject() method and is capable of replacing the reference of the object returned by deserialization.
- writeReplace() - runs before writeObject() and allows us to replace the object that got serialized.

Constructing Sensitive Objects

- Preventing subclasses from changing the behavior
- Marking Methods final
- Marking Classes final
- Marking Constructors private

DDOS

- defn: Denial of Service attack - DoS attack - one or more requests with the intent of disrupting legitimate requests.
- sending very large requests or sending a lot of requests.
- defn Distributed denial of service - DDoS - DOS attack from many sources at once.
- checking the size of a file before reading it.
- defn: inclusion attack - when multiple files or components are embedded within a single file.
- "Zip bomb", "billion laughs attack"
- Overflowing numbers abuse
- overriding .hashCode() to flood a HashMap or HashSet
- Creating arrays of billions of elements

Extra Exam Topic

https://www.oracle.com/java/technologies/javase/seccodeguide.html

Oracle added the word privileged to the security objective and this results in the doPrivileged() method now being in scope. For these questions, you need to read Section 9 of the Oracle Secure Coding Guidelines for Java SE. Some people reported seeing doPrivileged() on the 1Z0-816 exam, though, so it’s possible Oracle updated the 1Z0-816 exam more recently with this topic.
