CS2030S Final Examination Helpsheet (MY)

0. Introduction to Java

1. 8 primitive types:
    a. byte (signed 1B, default 0)<:short (signed 2B, default 0)<:int (signed 4B, default 0)<:long (signed 8B, default 0L)<:float (4B IEEE-754 single-precision≈7dp, default 0.0f)<:double (8B IEEE-754 double-precision≈15dp, default 0.0d)
    b. char (unsigned 2B=UTF-16 word, default 'u\0000')<:int
    c. boolean (1 bit, default false) unrelated to other primitives
2. java.lang primitive wrappers all implement Serializable and Comparable<T> interfaces: Byte, Short, Integer, Long, Float, Double, Character, Boolean. Like all objects, default=null. All extend java.lang.Number barring Boolean. Like all generics, they're invariant (e.g. Byte, Short lack relation, double d=4 widening ☑, Double d=4 ☒). They're immutable for thread-safety.
    a. Autoboxing: primitive → wrapper class. Unboxing: wrapper class → primitive. Both are implicit. Their constructors are deprecated, so call <wrapper_class>::valueOf(value) instead.
    b. Pros of wrappers: serializable, comparable, extends Object (more OOP). Cons of wrappers: full-fledged object is less memory-efficient performance-wise.
    c. Class / Interface fields (i.e. have access modifiers) aren't implicitly-initialised vs local variables.
3. java.util.Arrays is a covariant fixed-size dynamically-allocated container object like C malloc. Use .length for array size, .length() for string size, .size for object memory size.
4. Operators' descending precedence [Right-to-left-associativity flagged]: brackets ≡ indexing ≡ dot op > postfix** > unary ±** ≡ logical NOT (!)** ≡ unary bitwise (~)** ≡ prefix ** > new ≡ typecast > binary multiplicative ops > binary additive ops ≡ string concat (+) > SLS, SRS (<<, >>) ≡ ULS (<<<) > comparison ops ≡ instanceof > equality ≡ inequality > bitwise AND (&) > bitwise XOR (^) > bitwise OR (|) > logical AND (&&) > logical OR (||) > ternary op (pred?a:b)** > assignment ops** > lambda op (→)**
5. Java uses C-style 3-expr (initialisation, condition, advancement) loop in general and foreach loop for java.util.Collection & arrays. Syntax: for (<type> item:<iterable>):<block>
    a. Labelled and un-labelled break exist. Labelled eg:

```
int[][] b={{1,2,4},{8,3,6,9}}; int key=9,i=-1,j=-1; // Intiialise counters outside for larger sc
ope
linearSearch: // Label name is a variable reference
    for (i=0;i<b.length;i++){
        for (j=0;j<b[i].length;j++){
            if (b[i][j]==key){break linearSearch;}}}
if (i==-1 && j==-1) System.out.println("Absent"); // Curly braces optional for single statement.
else System.out.printf("Key at b[%d][%d]\n",i,j);
```

1. Introduction to Java's OOP

1. Java is pass-by-value only, not pass-by-reference.
    a. Pass-by-value: Caller and callee have 2 independent variables. Pass-by-reference: caller and callee share same variable. Java variables are object references ("pointers") so Java is still pass-by-value as variable's value is reference itself, not the referred object in memory.

```
Dog myDog = new Dog("Roger");
foo(myDog); // Referred object now has name "Max". myDog still reference same object outside met
hod scope.
void foo(Dog dog) {dog.setName("Max");dog=new Dog("onlyInMethodScope");dog.setName("onlyInMethod
Scope2");}
```

2. Java is statically-typed.
    a. Compiler program: source code → standalone machine code in executable file once-and-for-all
    b. Interpreter program: source code → intermediate form / object code and directly execute it without saves
    c. Java's approaches: 1. javac (Java in .java file → bytecode in .class file), then JVM (bytecode → machine code), 2. direct interpretation via jshell REPL (aka. read-eval-print loop)
    d. Tombstone (T) diagram: Illustrate transformation from source language (T's left) to target language (T's right) via implementation language (T's bottom, often ASM like x86-64, ARM-64 ISA).

    e. Static typing associates type to variable (∵ javac checks compile-time type); dynamic typing associates type to value (∵ runtime type can vary across codelines).
3. Java is strongly-typed.
    a. Explicit narrowing conversion (typecasting, conservative i.e. direct relation exists or **typecast to any interface(s) even if it doesn't implement them**), implicit widening conversion (coercion)
    b. Typecasting is discouraged where possible **runtime type mismatch** throwing unchecked ClassCastException may be possible (∴ CS2030S needs javadoc explanation). ClassCastException involves incompatible types that pass 2 conservative criteria above (e.g.: Object o = "H"; Long i = (Long) o; bypasses javac vs String s = "H"; Long i = (Long) s; compiler error).
    c. 3 typing properties: Reflexivity, Anti-symmetry (S<:T∧T<:S⇒S=T), Transitivity
4. Java is explicitly-typed with limited type-inference.
5. Java paradigms: OOP (dynamic, class-based), functional (∵ functional interface, λ expr, java.util.stream.Stream, immutability)
    a. 5 OOP pillars: Composition, inheritance, abstraction, encapsulation, polymorphism (CS2030S groups first 2 pillars actually)
        i. Composition: HAS-A relation; Inheritance: IS-A relation (**hierarchises classes, enabling code reuse**); Subtyping: IS-TYPE-OF relation.
            1. Java bans multiple (abstract/concrete) class inheritance to avoid diamond problem and allows multiple interface inheritance. Interface can't extend any abstract/concrete class.
            2. 2 methods are override-equivalent iff subclass's method's signature ⊆ direct superclass's method's signature. No 2 methods in a class / interface can be override-equivalent statically. E.g.: class Point{int x,y; abstract void move(int dx, int dy); void move(int dx, int dy){x+=dx; y+=dy;}}
            3. Class inherits interface's abstract and default methods so it inherits from but doesn't inherit the interface.
        ii. Abstraction barrier (caller function vs callee function defining method): Divide coder to implementer vs client, thus **freeing client fussing over low-level implementations (info-hiding)**.
        iii. Encapsulation: **Bundle objects' data (fields) & behaviour (methods)**, exemplified by class & interface. Class is extensible code template (blueprint) initialising data & behaviour of objects.
        iv. Polymorphism: **Dynamic dispatch** (write **generic** succinct code based only on target's **CTT**, actual behaviour based on target's RTT), **future-proof code by Liskov Substitution Principle**.
            1. obj instanceof class_or_interface checks if RTT of obj<:class_or_interface. CTT & RTT identification: <compile_time_type> c = new <runtime_type>();
            2. Object::equals(Object) defaults to referential equality but is more practically overridden to check functional equality. == operator checks referential equality via address values. Without explicit object instantiation via new keyword, same object is referenced (e.g. Strings). obj::equals(null) can be overridden to return true but breaks contract.
            3. Polymorphism isn't technically about overloading. Overriding∈Inheritance.
            4. LSP: S<:T⇒∀x(T(x)⇒S(x)) i.e. "what holds for T-objects holds for S-objects". Counter-e.g.: Square<:Rectangle in Rectangle::setWidth (∴ Implement both from Shape interface)
    b. 3 utilities of methods: Compartmentalise (isolate) computation and its effects [**modularisation**], **encapsulate** for readability and code update independent of clients, and allow **code reuse**.
        i. 4 components of method signature: **method name, param type, param count, param order**. Excluded: *return type, returned object count*.
        ii. **Overriding**: subclass & superclass methods with same return type, method name, param type, param count & param order. Excluded: param name, returned obj count. Opt @Override.
        iii. **Overloading**: same class's methods with same method name but ≥1 distinct among return type, param type, param count, param order & returned obj count. Excluded: param name
        iv. abstract method is a method declaration lacking a body (i.e. only signature exists).
            1. static method can't be abstract by designer choice as it's resolved statically so not overridable (∵ overriding is dynamic), but abstract method is a contract to be overridden.
            2. final method can't be abstract by designer choice as it's non-overridable by definition but abstract method is a contract to be overridden.
    c. Scope (vs lifetime) is a compile-time concept regarded as visibility / accessibility in CS2030S. 4 access (visibility) modifiers: private, package-private, protected, public. Package is a hierarchical grouping of related classes / interfaces (folder-file analogy).

        i. private: class; default: class, package, subclass of same package; protected: class, package, subclass of same package, subclass of distinct packages; public: all including world
        ii. Top-level class / interface defaults to package-private and can't be private/protected. Class fields / methods default to package-private. Interface fields / methods default to public.
        iii. Access control is based on class, not instance. E.g.:

```
class A {private int x;}
public class B {
    private int y; // y is a field, not a local variable.
    void foo(A a, B b) {
        b.y=2; // access is permissible within same class B on which b's data and behaviour r
ely. Calling a.x=0; would be illegal.
        this.y=1;}}// access is permissible outside constructors.
```

            1. Non-access modifiers: Interfaces' fields default to static final. Interfaces' methods default to abstract. Interfaces' class / nested interface default to static.
            2. abstract class is a contract declaring fields & methods and *intended* to be subclassed. It may counter-intuitively have concrete methods and initialised fields and no abstract members.
            3. Interface bundles (encapsulates) strictly public abstract methods and strictly public static final fields unlike abstract class.
    d. Tell, Don't Ask (TDA) principle: Client should tell object what task to perform vs asking for object's data then performing the task. Getter (accessor) & setter (mutator) proliferation (i.e. per class / interface member) is an OO anti-pattern.
    e. Class modifier: public protected private abstract default static final transient volatile synchronized native strictfp. final class bans inheritance, final method is non-overridable, final field is non-reassignable (transitively-mutable like Python tuple only), abstract class is non-instantiable, static field / method belongs to the class, so it's accessible without instance and a unique copy exists across all instances.
    f. Aliasing (reference-sharing) allows code reuse and avoids similar objects' proliferation, but harms thread-safety (side-effect propagated to all reference to same object). Solution: immutability

2. Heap, Stack & Meta-space for Runtime Illustration

1. Heap stores dynamically-allocated objects. Each heap's object has 3 components: **class name** (heading), **instance field(s) & corresponding values** (may be primitive or further point to heap's other objects), **captured (∵ effectively-final) variables & corresponding values** (may be primitive or further point to heap's other objects). Simply put, store in heap iff it follows new keyword i.e. object.
    a. Optionally specify hexadecimal memory address per object to the right of class name heading, if the address is even specified.
2. Stack stores **LIFO** (enqueue and dequeue bottom, newer frames stacked "on-top" older ones) **created-and-yet-terminated call frames** (effectively (static & non-static) **method bodies** invariably including main ∵ entry-point for program's execution, bottommost frame). Each call frame has 2 components: **param(s)** (may be primitive or point to object(s) in heap), **local variable(s)** (may be primitive or point to object(s) in heap).
    a. main has args param. Each non-static (instance) method implicitly has qualified <enclosing_method>.this local variable which must be depicted.
3. Meta-space stores **static field(s**) & corresponding value (may be primitive or point to objects in heap). Meta-space and heap lack guaranteed insertion/deletion order vs stack.

3. Method Invocation (Static & Dynamic Binding)

1. Static (early) binding [before type-erasure]: javac resolves most CTT-specific method's descriptor (return type & method signature), type-erases it to remove generics, then stores it in bytecode.
    a. If a class's methods have same specificity post-type-erasure, they're override-equivalent and raise compiler error. E.g. if 2nd program's A's T doesn't extend Comparable, code won't compile.
2. Dynamic (late) binding [runtime]: JVM decodes static binding's descriptor, determines target object of method invocation's RTT (∵ account for runtime overriding), then **determine from most specific (possibly-overridden) method upward matching RTT to CTT**. E.g.:

```
class A {void foo(A a) {System.out.println("Class A, Param A");}} // CTT method descriptors stored
in generated bytecode: foo(A) in A
class B extends A { // CTT method descriptors stored in generated bytecode: foo(A) in B, foo(B)
```

```
    @Override void foo(A a) {System.out.println("Class B, Param A");}
    void foo(B b) {System.out.println("Class B, Param B");}}
class C extends B {void foo(C c) {System.out.println("Class C, Param C");}} // CTT method descripto
rs stored in generated bytecode: foo(A) in B, foo(B), foo(C)
class D extends C {@Override void foo(B b) {System.out.println("Class D, Param B");}} // CTT method
descriptors stored in generated bytecode: foo(A) in B, foo(B) in D, foo(C)
public class Main { // In a.foo(d), target object of method invocation is a, whose CTT is decoded,
then RTT is determined type D, then D's CTT is compared upwards to match A's CTT
    public static final void main(String...args) {
        A a = new D(); // CTT A, RTT D
        B b = new D(); C c = new D(); D d = new D();
        a.foo(d); b.foo(d); c.foo(d); d.foo(d);}} // stdout: Class B, Param A\nClass D, Param B\nCl
ass C, Param C\nClass C, Param C
```

```
class A<T extends Comparable<T>> { // CTT method descriptors stored in generated bytecode: fun(Comp
arable) in A, fun(Object) in A [NO-LONGER fun(T), fun(S) post type-erasure!!!]
    public void fun(T t) {System.out.println("1");}
    public <S> void fun(S s) {System.out.println("2");}}
class B extends A<Double> { // CTT method descriptors stored in generated bytecode: fun(Comparable)
in A, fun(Integer) in B, fun(Object) in B
    public void fun(Integer i) {System.out.println("3");}
    public <S> void fun(S s) {System.out.println("4");}} // @Override optionally unused.
public class Main { // In a.fun(2), target object of method invocation is a, whose CTT is decoded,
then RTT is determined equal, then A's CTT is compared upward to match A's CTT
    public static final void main(String...args) {
        A a = new A<Integer>(); B b = new B();
        a.fun(2); a.fun(2f); a.fun(new B()); b.fun(2); b.fun(2f); b.fun(new B());}} // stdout: 1 1
2 3 1 4
```

4. Type variance

1. Complex Type C(S): Nested data structure composed of primitive datatype S. E.g..: java.util.Collection & arrays.
2. Covariance: S<:T⇒C(S)<:C(T). E.g. array, upper-bounded wildcard (className<S><:className<? extends S><:className<? extends T>)
    a. Arrays' covariance violates LSP that causes heap pollution. E.g. Shape[] s = new Circle[10]; shapes[0] = new Square(); compiles but accessing it causes UB. ∴ Java isn't type-safe here.
3. Contravariance: S<:T⇒C(S):>C(T). E.g. lower-bounded wildcard (className<T><:className<? super T><:className<? super S>)
4. Invariance: S<:T⇒ No subtyping / supertyping relation between C(S) and C(T). E.g. generics without wildcard.

5. Imperative Programming

1. java.lang.Exception-handling (try-catch-finally, Java lacks else vs Python, optional finally block always executed on normal/exceptional completion before exit, throw error)
    a. Java ban exception subtype being thrown after a supertype with compilation error : exception <subtype_exception> has already been caught.
    b. Each optional catch block defines its own scope so variable names are reusable. Avoid risky "Pokémon exception handling" (catch (Exception e)) or silent exit in catch (System.exit(0);). Beware some exceptions may break abstraction barrier (e.g. SQLException exposes presence of SQL database). Do not misuse try-catch as if-else.
    c. Checked (compile-time) exception is recoverable from and must be explicitly handled by caller via throws keyword. E.g.: FileNotFoundException, IOException, InterruptedException.
        i. Passing the buck is the avoidance of explicit catch for (more-often checked) exception by deferring its handling to a higher level (enclosing method / class) via throws.
    d. Unchecked (runtime) exception = java.lang,RuntimeException is a programming error best unhandled to spot them early, else can be handled via throws in defensive programming.
    e. If overridden method throws exception, overriding method must throws same or more specific exception by LSP. As LSP is unenforceable, violating this still compiles.
    f. java.lang.Error<:Throwable like Exception but exposes system resource deficit and isn't handled.
2. Conditional flow

a. switch statement: control-flow imperative internally-implemented as hash tables so faster than if-else for big case count, assignable from SE12. Java falls-through like C (SE12 introduces new case label without fall-through) with mandatory default case (vs C). Syntax:

```
switch (<val>){ // switch value must be of type byte, char, short, int (or their wrappers), e
num, or String, else better to just use if-else
    case <key1>:<block_1_with_fallthrough_and_optional_break>;
    case <key2>->(single_expr2>; // no fall-through
    case <key3>->{} // no fall-through with statement block
    ... // Arbitrary case count
    default: <block_with_redundant_break>;}
```

6. Generics & Type Erasure + Inference

1. Generic method declaration: public static <S> T methodName(? super S input,..., ? extends T output) {}. Generic class declaration: class className<T1,...,Tn) {}.
2. Generic type must be non-primitive. **Parameterised type** is an instantiation of generic type with actual type arguments. **Raw type** is a non-generic type with no diamonds nor type arguments (∵ turn-off generic) that is non-type-safe vs type with **unbound wildcard**. Eg: void append(List<?> l){l.add(new Object()); doesn’t compile though l is type-erased as List<Object> from javac’s conservativity (∴ modify to List<? extends Object> instead), but raw type List doesn’t check.
   - a. ∴ @SuppressWarnings(”rawtype”) mustn’t be used in CS2030S except instanceof runtime (∵ type-erasure in-effect) checks. Comment to prove contextual type-safety immediately beneath.
   - b. Unbound wildcard type is a non-parameterised (unknown) type via wildcard character (?). 2 scenarios: method implemented using functionality given in Object class, method doesn’t rely on type param of enclosing generic class. It’s the supertype of all parameterised type: ∀S(className<S><:className<?> ∧ className<S><:className<Object>).
     - i. Unbound wildcard type crucially differs from parameterised type containing Object in that it has an unknown type: Conservative javac accepts **only** null elements in className<?> but all elements in className<Object>.
   - c. className<> without ? is NOT an unbound wildcard type; it’s just an optional syntactic sugar from Java 7 where parameterised type is given in same/previous codeline so javac can infer it. E.g. List<String> = new ArrayList<>();.
3. Generics restrictions:
   - a. Reifiable type: type info is required to be fully-available at runtime. E.g.: primitive, non-generics, raw type, array, invocations of unbound wildcard. ∴ Generic array declaration compiles (e.g. T[] arr;) but not generic array instantiation (e.g. new T[2];) owing to heap pollution woe by designer choice. Alternative @SuppressWarnings(”unchecked”) has 3 CS2030S requirements:
     - i. Apply it to most limited scope possible. If used in method / constructor, specify in its body immediately after opening curly brace. Use with extreme discretion.
     - ii. Always comment immediately beneath to prove contextual type-safety (∵ its existence makes Java potentially not type-safe here).
     - iii. It can apply to local variables. E.g. @SuppressWarnings(”unchecked”) List<String> l = (List<String>) new ArrayList<String>(); but commenting still applies.

```
public Seq(int size) { // Constructor here.
    @SuppressWarnings("unchecked") // ALL Java annotations lack semicolons.
    T[] temp = (T[]) new Object[size]; // Separated into 2 lines as @SuppressWarnings("unchecke
d"); applies to declaration only, NOT assignment.
    this.array = temp;}
```

   1. No direct instantiation, catch, or throw of objects of parameterised type of abstract classes / interfaces (concrete classes work).
   2. Static generic fields / methods must explicitly spell out type params (e.g. static <S> S s;) as they’re resolvable at compile-time post-type-erasure.
   3. Overloading where formal param types type-erase to same raw type generates override-equivalent methods in same class, which can’t compile (E.g. void f(List<String> l){} & void f(List<Long> l){}).
4. “Producer extends Consumer super” (PECS) is w.r.t. variable referencing DS in the program, not the human programmer (reverse angle). **No multiple bounds** even if they’re interfaces.
5. 4 type-inference rules: type1<:T<:type2 ⇒ T inferred as type1; type1<:T⇒ T inferred as type1; T<:type2⇒ T inferred as type2; if no type fulfills all constraints, compiler error is raised.

6. All non-statically-inferrable generic type T without conflict is widened to Object but risks heap pollution in variadic methods (e.g. main(String...args){}). ∴ Add @SafeVarargs annotation.

7. Nested Class

1. Inner (non-static nested) class accesses enclosing class’s other members vice versa (∵ enclosing class can instantiate private inner class within). Variable capture applies to local variables so it isn’t applicable for inner class. Java offers qualified this (enclosingClass.this.x) vs innerClass.this.
   - a. Inner & local class’s static members must be constant (aka. primitive / explicitly-declared final) to be statically-resolvable as inner & local class cannot be created independently from instance of enclosing class but static members can’t be runtime-dependent.
2. Static (non-inner) nested class is behaviourally top-level class nested for packaging (hierarchisation / encapsulation) so it doesn’t exhibit variable capture.
3. Local inner class is declared in a block, often method body. It can access enclosing class’s other members vice vers and capture effectively-final (non-reassigned, reference types have “loophole” in the form of in-place mutation) local variables & params in enclosing method.
   - a. Block can’t be interface body as interface members including its nested class is implicitly static so such a class is static nested (non-inner).
4. Anonymous class is declared as λ-expr in a statement strictly. Syntax: parentType obj=new parentConstructor(args){//classBody};. Anonymous local inner class exhibits variable-capture (λ-expr in method body). Its class body allows 4 features: field declaration, field initialisation, local class, added method. Exclude: reassignment, incre / decre, method call, class instantiation, inheritance.

```
public class Outer{ // static nested interface↓
  interface A{int add();int addi(int i);}
  Add addTwo = new A { // Anonymous interface
    int g = 0; // Field initialisation can
    void newMethod(); // New method can
    abstract class B(); //local class cannot
    g=1; g++; // Reassignment / Increment cannot
    System.out.print(""); // method call cannot
    Byte b = new Byte();}}//instantiation cannot
```

8. Functional Programming

1. 4 implementation principles for Immutable class: sealed class A permits B{} (∴ inheritable, i.e. non-overridable methods) class, non-final fields are still re-assignable so it isn’t sufficient); final fields; private fields + no public setter; instantiate new object clones independent of initial object reference (deep-copy in constructor / unit).
   - a. 3 utilities of immutability: ease of understanding, safe sharing (pass-by-value of references directly) of objects and internals alike, safe multithreading for concurrency.
2. Pure function is side-effect-free and referentially-transparent.
   - a. Side effect: a function’s state modification beyong primary I/O. CS2030S side-effect list: stdout, file write (☒file read), checked & unchecked exception (e.g. division may throw unchecked ArithmeticException), (inplace & out-of-place) modification / reassignment of other variables, no output / input.
   - b. Referential transparency: identical return values for identical inputs by substitution with statics, non-locals and mutable references. ∴ It ⇒ determinism, which strengthens to iff for functions, but fails for general expr, e.g. rand method call with setSeed).
3. Functional Interface: not-declared sealed interface with exactly 1 abstract method barring Object’s methods, allowing optional @FunctionalInterface. It’s unambiguously-reducible to λ-expr (∵ 1 overridden method, 0 constructor) so it’s a 1st-class citizen.
   - a. 1st-class citizens fulfill 5 criteria: passable as function args, returnable in functions, L.H.S. of assignment, incorporable to all DS, comparable for equality.
   - b. Lambda (closure) is syntactic sugar for 1st-class anonymous inner class: (lambdaOperator)→{lambdaBody}. It allows currying. E.g.: Transformer<Integer, Transform<Integer, Integer>> curriedAdd = x → y → x+y;.
4. Monad has unit (static of, often private constructor for better instantiation control) and bind (flatMap).
   - a. Left Identity Law: Monad.of(x).flatMap(x→f(x))≡f(x)
   - b. Right Identity Law: monad.flatMap(x→Monad.of(x))≡monad
   - c. Associativity Law: monad.flatMap(x→f(x)).flatMap(x→g(x))≡monad.flatMap(x→f(x).flatMap(x→g(x)))

   - d. All monads are functors but converse fails. map implementation of flatMap:

```
public <U> Monad<U> flatMap(Transformer<? super T, ? extends Monad<? extends U>> f) {return new
Monad<U>(f.transform(this.item).get());}
public <U> Monad<U> map(Transformer<? super T, ? extends U> f) {return this.flatMap(this.item->M
onad.of(f.transform(this.item)));}
```

5. Functor has unit and tool (map) i.e. simpler abstraction of monad without flattening.
   - a. Right Identity Property: functor.map(x→x)≡functor
     - i. “Left Identity”: Functor.of(x).map(x→f(x)) may not equate to Functor.of(f(x)). E.g. Maybe::of(null) has type None; Maybe::of(f(null)) may have type Some.
   - b. Composition Property: functor.map(x→f(x)).map(x→g(x))≡functor.map(x→g(f(x)))

9. Lazy Evaluation

1. Lazy evaluation (call-by-need) is non-strict (delay computing till value is explicitly fetched via get()) and uses sharing (memoization to avoid repetition). Eager Java’s λ-expr simulates lazy initialisation.
   - a. 3 utilities: abstraction of control flow, allow possibly-infinite (e.g. Fibonacci) DS for easier algos, rapid prototyping of partly-defined-buggy DS.
2. java.util.stream.Stream (Simulate Infinite Collection) allows single-consumption (stricter than Python iterators), else unchecked IllegalStateException. Eg: for(int i=0;i<2;i++){stream.forEach(System.out::println);} completes exceptionally.
   - a. Stateful ops rely on state (e.g. order, value) of previously-processed elements (E.g.: distinct(), sorted(), limit(long), skip(long)). Bounded ops will produce finite streams (E.g.: limit(long), skip(long)). Sorting op is lazily-called on infinite stream so sorted() is unbounded; some ops are both
     - i. Stateful op has negative side-effects (synchronisation overhead) on parallelism; bounded op has positive side-effect (smaller stream optimisation).
   - b. Other methods: generate, iterate, allMatch(Predicate), anyMatch(Predicate), noneMatch(Predicate), count(), findAny(), findFirst(), peek(Consumer), filter(Predicate), collect(Collector). collect maps elements then stores them in an eager mutable container (List, Set, Map) so it differs from reduce.
   - c. 3 method signatures of stream::reduce (left-fold)

```
Optional<T> reduce(BinaryOperator<T> accumulator) { // Signature 1
    boolean isStreamEmpty = true; T output = null;
    for (T element : this) {
        if (isStreamEmpty) {isStringEmpty = true; output = element;}
        else {output = accumulator.apply(output,e);}
    return isStreamEmpty ? Optional.empty() : Optional.<T>of(output);}
T reduce(T identity, BinaryOperator<T> accumulator) {
    T output = identity;
    for (T element : this) {output = accumulator.apply(output, element);}
    return output;}
<U> reduce(U identity, BiFunction<U, ? super T, U> accumulator, BinaryOperator<U> combiner) // o
ptionally allow parallelism.
```

   1. 3 sufficient conditions for side-effect-free in signature 3:
      - a. Identity is an identity for accumulator and combiner. CS2030S stresses latter: combiner.apply(identity,t)≡t.
      - b. Individually-associative accumulator and combiner.
      - c. Compatible accumulator and combiner: combiner.apply(u,accumulator.apply(identity,t))≡accumulator.apply(u,t).

10. Asynchronous Programming

1. Concurrency interleaves modularised subtasks vs sequentialism. Parallelism ⊂ concurrency, wherein >1 modularised task is run simultaneously vs serialism, warranting >1 thread.
2. Asynchronicity is a contract for a thread to continue onto other ops while waiting for current task to execute to completion.
   - a. Thread pool is an assembly of threads (workers) executing tasks in parallel. It reduces thread creation+dismantling (deallocation) overheads as freed thread is reused for a next task. Types in java.util.concurrent.Executors:
     - i. newFixedThreadPool fixes existing thread count i.e termination of a thread prompts pool to create replacement, enqueue task if all threads are running. ∴ User can cap thread count below server capacity safely.

       1. Use: parallelise huge count of async tasks.
     - ii. newSingleThreadExecutor creates a degenerate 1-thread pool so task execution is effectively sequential.
     - iii. newCachedThreadPool creates new thread per extra task if existing threads are all running (∴ reuse if possible) and terminates thread idle for 1 min. ∴ All threads are temporal so pool is instantiated in cache. Use: parallelise many short-lived async tasks.
   - b. Future interface denotes a yet-completed async task’s result; it is a contract. java.util.concurrentCompletableFuture<T> is a Future that may be explicitly completed. 3 instantiation methods: static <U> completedFuture(U value); static CompletableFuture<Void> runAsync(Runnable r); static <U> CompletableFuture<U> supplyAsync(Supplier<U> s);
   - c. Runnable functional interface is optionally implemented by a task, oft-thread in CS2030S at least, to describe the task execution returning nothing. Builtin java.lang.Thread<:Runnable has default empty Thread::run body.
     - i. Callable functional interface analogously describes a task execution with a single generic return type.
   - d. Executor interface completes async tasks oft-in-parallel for more efficiency, & automatically via execute() for friendliness. It decouples task submission & execution: new Thread(new RunnableTask()).start() is redundant
     - i. ExecutorService<:Executor via startup, forced shutdown, await, and status peek functionalities. Its submit() can input Callable vs Executor inputting Runnable only.
     - ii. ForkJoinPool is a thread pool extending ExecutorService: it has work-stealing to further raise throughput. 3 task execution methods:
       1. this::execute(Runnable) arranges for the specified task to be async-executed by a pool’s thread, returning nothing.
          - a. ForkJoinTask::fork modularises a task to smaller subtasks (divide-and-conquer algo), returning this for simplification, letting underlying subtasks to complete.
          - b. Combination of underlying subtasks can only be explicit via ForkJoinTask::join or invokeAll, both being sync (∵ calling thread returns only if it completes task).
          - c. Beware join / invokeAll don’t avoid idle thread from stealing any other tasks from its tail while it runs its task
       2. this::invoke(ForkJoinTask) awaits completion of task and returns task’s generic result. It’s analogous to 1(b) where a calling thread awaits (completion &) combination of subtasks’ results while pool blocks others.
       3. this::submit(ForkJoinTask) mimics execute but returns a Future (∵ task isn’t necessarily of Runnable type).
          - a. ∴ Both start task execution async; submit has better exception-handling encapsulated by Future; execute applies to fire-and-forget execution.
   - e. java.util.concurrent provides a thread-safe implementation of deque, namely LinkedBlockingDeque, per thread (worker), though non-thread-safe alternative ArrayQueue can be opted.
     - i. ForkJoinTask::fork enqueues subtasks at thread’s deque’s head to be executed first (FIFO), pushing preceding tasks tailward.
     - ii. Idle thread steals the tail-most task from a busy thread to enqueue at its deque’s head.
3. 3 summative behaviours for ForkJoinTask::join:
   - a. If the task is *yet-executed* (∵ join internally tests if task has begun), its calling thread calls ForkJoinTask::compute to execute task itself.
   - b. If the task is executing but yet-completed, calling thread cannot return till task is complete (∵ join is sync) but it can steal work from other threads if possible.
   - c. If the task is complete, its result is returned (fetched if task was stolen).