CS2030S 21/22 Notes by Justin Peng

1. Variable & Type

Statically Typed (Java):
- Need to declare every variable in the program and specify its type
- A variable can only hold values of the same type as the variable type
- Once a variable is assigned a type, its type cannot be changed

Dynamically Typed:
- Same variable can hold values of different types
- Checking if the right type is used during execution of the program
- Type is associated with the values, and the type of the variable changes depending on the value it holds

Compile-Time Type:
- Type that a variable is assigned with upon declaration
- Compiler checks if compile-time type matches

Strong Typing (Java):
- Enforces strict rules in its type system, to ensure type safety
- Catch type errors during compile time rather than leaving it to runtime

Weak Typing:
- More permissive in terms of typing checking

Primitive Types:
- Numeric values: *byte, short, int, long, char, float, double*
- Boolean values: *true, false*
- Never share values

Reference Types:
- Stores only the reference to the value
- Can share the same value
- Change to one reference affects others with same value

Method signature:
Method name + number, type, & order of params
Method descriptor:
Method signature + return type.

2. OOP Principles

Abstraction (1):
- **Abstraction barrier** separates roles: (i) implementer provides function implementation, (ii) client uses function to perform task
- e.g., class (composite data type w/ fields + methods)

Encapsulation (2):
- Keep all data & functions operating on the data related to a composite data type together within an abstraction barrier

Information Hiding:
- Enforces the abstraction barrier
- Make all fields & necessary methods **<u>private</u>**
- Constructor method: same name as class, no return type

Tell, Don't Ask:
- **Accessors & Mutators (get & set):** Only when **<u>necessary</u>**, else breaks encapsulation
- Implement methods for class to process its own data
- Don't ask the class for data to process outside (class-agnostic)

Composition:
- Models the *HAS-A* relationship
- **Sharing References (aka Aliasing):** modifying reference type changes all objects that it composes — avoid sharing references

Inheritance (3):
- Models the *IS-A* relationship
- Uses **<u>extends</u>** keyword
- Creates subtypes

Polymorphism (4):
- Dynamic Binding: Method invoked is decided during run-time, depending on the run-time type of object
- Write succinct code that is future proof
- **Dynamic polymorphism (Overriding):** same method descriptor (parent vs child)
- **Static polymorphism (Overloading):** same name, different signature (within class)

3. Heap & Stack

Heap:

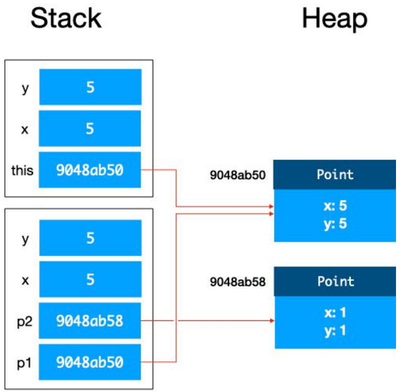
- Dynamically allocated objects
- Memory remains as long as it's being referenced

Stack:

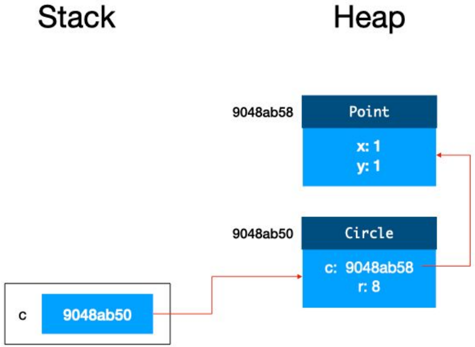
- Local variables (primitive & reference types)
- Stack frames on method call (this, method args, local vars, etc.)
- Nested method calls: the stack frames get stacked on top
- Memory deallocated when method returns

```java
class Point {
  private double x;
  private double y;

  public Point(double x, double y) {
    this.x = x;
    this.y = y;
  }

  public void move(double x, double y) {
    this.x = x;
    this.y = y;
  }
}
```

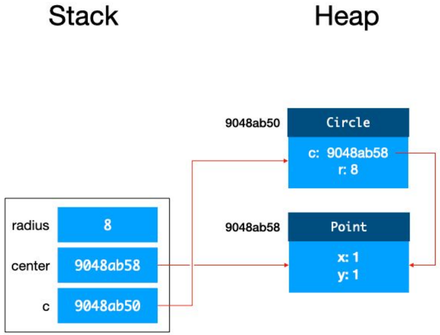
```java
Point p1 = new Point(0, 0);
Point p2 = new Point(1, 1);
double x = 5;
double y = 5;
p1.move(x, y);
```




```java
Circle c;
c = new Circle(new Point(1, 1), 8);
```

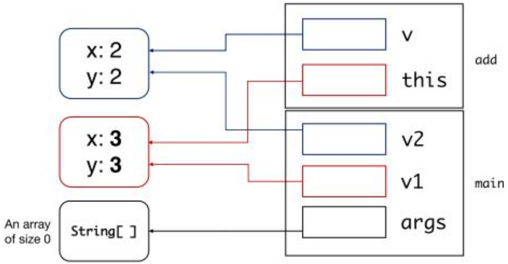



```java
Circle c;
Point center;
double radius;
radius = 8;
center = new Point(1, 1);
c = new Circle(center, radius);
```




```java
class Vector2D {
  private double x;
  private double y;

  Vector2D(double x, double y) {
    this.x = x;
    this.y = y;
  }

  void add(Vector2D v) {
    this.x += v.x;
    this.y += v.y;
    // line A
  }
}

class Main {
  public static void main(String[] args) {
    Vector2D v1 = new Vector2D(1, 1);
    Vector2D v2 = new Vector2D(2, 2);
    v1.add(v2);
  }
}
```

Solution:



Common mistakes include:

- Forgetting the `args` is a method parameter to main so should be allocated on stack. Java's convention is that `args` points to an empty array, but we are fine with `args` pointing to `null` too.
- Did not update the value of `v1` to (3, 3).
- Give the wrong order of stack frame.

4. Method Invocation

Compile Time (Method Descriptor):

1. Determine compile-time type of target (declared)
2. Search for all methods that can be correctly invoked on the given argument
3. Choose most specific one and store method descriptor

Run Time (Only for Instance Methods):

1. Retrieve method descriptor
2. Determine run-time type of target
3. Search for target type's method matching descriptor
4. Search upward in class hierarchy

Class Methods (static):

- Don't support dynamic binding
- Compile Time step is final ➔ invoke that most specific method

```java
Circle c = new ColoredCircle(p, 0, blue);
```

- Compile-time type of `c`: `Circle`
- Run-time type of `c`: `ColoredCircle`

5. Liskov Substitution Principle (LSP)

- If $S <: T$, then S must have all properties/functions of T
  - Subclass S shouldn't break expectations set by superclass T
- If $S <: T$, then T instances should be substitutable with S
  - Subclass S should pass all test cases of superclass T
- Use `final` keyword to prevent class inheritance or method overriding

```java
Integer[] intArray = new Integer[2] {
  new Integer(10), new Integer(20)
};
Object[] objArray;
objArray = intArray;
objArray[0] = "Hello!"; // <- compiles!
```

6. Special Classes

All Classes:

- Can only extend 1 (abstract/concrete) class
- Can inherit multiple interfaces

Concrete Class:

- No abstract methods (must override all abstract methods from parent)

Abstract Class:

- At least 1 abstract method (cannot be implemented, no method body)
- Can have concrete methods
- Cannot be instantiated

Interface:

- Completely abstract class (no concrete methods)
- Usually ends with the -able suffix

Wrapper Class:

- Reference type for primitive type
- All primitive wrapper class objects are immutable
- Auto-boxing/unboxing performs type conversion between primitive type and its wrapper class

7. Types

Casting:

- Forced narrowing type conversion requires explicit typecasting
- Ask the compiler to trust that the object is an instance of the subclass
- Must be careful (make sure it is safe)

Variance:

- Complex types: nested data structures composed of primitive types
- Let $C(T)$ be complex type of T (e.g., $Array<T>$):
  - **Covariant:** $S <: T$ ➔ $C(S) <: C(T)$
  - **Contravariant:** $S <: T$ ➔ $C(T) <: C(S)$
  - **Invariant:** neither covariant nor contravariant
- Java Arrays are covariant ($Integer <: Object$ ➔ $Integer[] <: Object[]$)
  - May cause run-time errors uncaught during compile-time

8. Exceptions

Unchecked Exceptions:

- *IllegalArgumentException, NullPointerException, ClassCastException*
- Not explicitly caught or thrown
- Subclasses of *RuntimeException*
- Indicate something wrong with the program and cause run-time errors

Checked Exceptions:

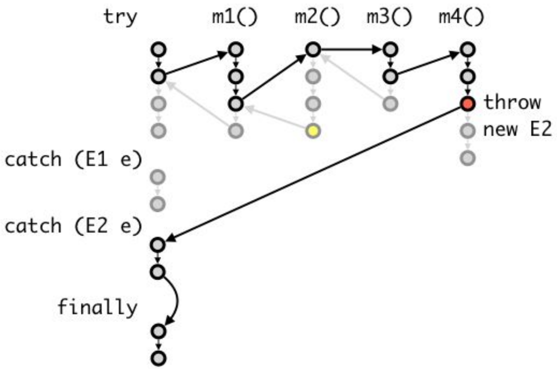
- *FileNotFoundException,*
- Must anticipate and handle them

Throwing Exceptions:

- Declare that class can throw an exception, with `throws` keyword
- Create a new *XXXException* object and `throw` it to the caller

Exception Propagation:

- Unchecked exception: propagate down the stack until caught or display error message to user
- Checked exception: must be caught somewhere else compile error



Creating Exceptions:

```java
class IllegalCircleException extends IllegalArgumentException {
  Point center;
  IllegalCircleException(String message) {
    super(message);
  }
  IllegalCircleException(Point c, String message) {
    super(message);
    this.center = c;
  }
  @Override
    public String toString() {
      return "The circle centered at " + this.center + " cannot be created:"
    }
}
```

Overriding Method that Throws Exceptions:

- Overriding method must throw only the same, none, or a more specific checked exception, than the overridden method (LSP)
- Caller of the overridden method cannot expect any new checked exception other than indicated by `throws`

Good Practices:

- Catch Exceptions to clean up resources (*finally* block)
- Don't catch all Exceptions as e
- Don't exit program because of Exception (prevents cleanup)
- Don't break abstraction barrier (handle Exceptions ASAP)
- Don't use Exception as control flow mechanism

9. Generics

Purpose:

- Allow classes/methods to use generic type (undefined) until compiled
- Ensures type safety: bind generic type to specific type at compile time
- Errors will occur at compile time instead of runtime
- Generics are invariant

Terminology (e.g., *Array<T>*)

- **Generic type *(Array)*:** takes other types as type parameters
- **Type arguments *(T)*:** passed into <>, can be non-generic type/ generic type/another type parameter that has been declared
- **Parameterized type:** instantiated generic type

Usage:

```
class Pair<S extends Comparable<S>,T> implements Comparable<Pair<S,T>> {
class DictEntry<T> extends Pair<String,T> {
public static <T> boolean contains(T[] array, T obj) {
A.<String>contains(strArray, 123); // type mismatch error
```

Bounded Type Parameters:

- Put a constraint on generic type
- *<S extends T>, <S super T>*

```
B implements Comparable<B> { ... }
A extends B { ... }
A <: B <: Comparable<B>
Comparable<A> INVARIANT Comparable<B>
Comparable<A> <: Comparable<? extends B>
```

10. Type Erasure

Code Sharing Approach:

- After type checking, erase type parameters/arguments on compilation
- Erasure of generic type ➔ raw type (Array<String> ➔ Array)
- Instead of creating a new type for every instantiation
- Only one representation of the generic type in the generated code, representing all instantiated generic types, regardless of type arguments
- Ensures casting will not lead to *ClassCastException* during run-time.

Example:

- **Unbounded:** Replaced with *Object*.
- **Bounded:** Replaced by the bounds (e.g., *T extends Integer* ➔ *Integer*)

```
Integer i = new Pair<String,Integer>("hello", 4).getSecond();
```

```
Integer i = (Integer) new Pair("hello", 4).getSecond();
```

Generics & Arrays:

- DO NOT MIX
- Type erasure removes generic types ➔ cannot guarantee correct types in Array

```
// create a new array of pairs
Pair<String,Integer>[] pairArray = new Pair<String,Integer>[2];
Not legal!! Can't create array of generics
// pass around the array of pairs as an array of object
Object[] objArray = pairArray;

// put a pair into the array -- no ArrayStoreException!
objArray[0] = new Pair<Double,Boolean>(3.14, true);
```

Terminology:

- **Heap Pollution:** Variable of a parameterized type refers to an object that is not of that parameterized type
- **Reifiable type:** Full type information is available during run-time (Array reifiable, generics not reifiable)

11. Unchecked Warnings

Collection of Generics:

- Can use *ArrayList* to store generic types
- *ArrayList* is generic type ➔ invariant ➔ no alias ➔ no heap pollution

Unchecked Warning:

- Message from the compiler that it has done what it can, and because of type erasures, there could be a run-time error that it cannot prevent

@SuppressWarning:

- Only if you are sure that the type casting is safe
- Cannot apply to an assignment, but only to declaration of new var
- e.g., `array = (T[]) new Object[size];`
- Array private ➔ insert into array only through the Array::set ➔ only accept type T ➔ only retrieve type T ➔ *Object[]* to *T[]* is safe
- Must use in the most limited scope ➔ avoid unintentionally suppressing warnings that are valid concerns from the compiler
- Only if it will not cause a type error later
- Must add a note (as a comment) explaining why safe to suppress

Raw Types:

- Generic type used without type arguments
- <u>**NEVER USE**</u> raw types
- Do not ignore or suppress raw type warning
- Example:

```
Array<String> a = new Array<String>(4);
populateArray(a);
String s = a.get(0);
```

```
void populateArray(Array a) {
    a.set(0, 1234);
}
```

Raw Type

12. Wildcards

```
public static <S, T extends S> boolean contains(Array<T> array, S obj) { .. }
```

Upper-Bounded (<*? extends S*>):

- For any type *S*, *A<S> <: A<? extends S>*
  - *Array<Circle> <: Array<? extends Circle>*
- *S <: T* ➔ *A<? extends S> <: A<? extends T>* (covariance)
  - *Circle <: Shape* ➔ *Array<? extends Circle> <: Array<? extends Shape>* ➔ *Array<Circle> <: Array<? extends Shape>*

Lower-Bounded (<*? super S*>):

- For any type *S*, *A<S> <: A<? super S>*
  - *Array<Shape> <: Array<? super Shape>*
- *S <: T* ➔ *A<? super T> <: A<? super S>* (contravariance)
  - *Circle <: Shape* ➔ *Array<? super Shape> <: Array<? super Circle>* ➔ *Array<Shape> <: Array<? super Circle>*

PECS:

- Producer Extends
  - Producer that returns a variable of type *T* ➔ *<? extends T>*
- Consumer Super
  - Consumer that accepts a variable of type *T* ➔ *<? super T>*

Unbounded (<*?*>):

- Take in any type, same as *<? extends Object>*
- *Array<?>* is the supertype of every parameterized type of *Array<T>*
- Pros: Invariant, can cast `Array<?> a1 = new Array<String>(0);`
- Cons: Can only retrieve *Object*, can only set *null* for type safety
- *Array<?>*: objects of some specific, but unknown type;
- *Array<Object>*: Object instances, with type checking by compiler
- *Array*: Object instances, without type checking (Raw Type)
- Unlike *Comparable<String>*, *Comparable<?>* is reifiable
  - Unknown type *<?>* ➔ no type information lost during erasure

```
new Comparable<?>[10];
```

Acceptable

13. Type Inference

Diamond Operator:

- Infer instantiated generic type based on compile-time type of variable

```
Pair<String,Integer> p = new Pair<>();
Pair<String,Integer> p = new Pair<String,Integer>();
```

Type Inferencing:

- Search for all matching types that would lead to successful type checks ➔ pick the most specific ones

```
public static <S> boolean contains(Array<? extends S> array, S obj) {
A.contains(circleArray, shape);
A.<Shape>contains(circleArray, shape);
```

1. *S obj* is *Shape* ➔ ***S <: <? super Shape>*** (if widening type conversion)
2. *Array<? extends S> array* is *Array<Circle>* ➔ Array<Circle> <: Array<? extends S> ➔ *Circle <: <? extends S>* ➔ ***S <: <? super Circle>***
3. Most specific: *S* is *Shape*

```
public static <T extends GetAreable> T findLargest(Array<? extends T> array) {
  double maxArea = 0;
  T maxObj = null;
  for (int i = 0; i < array.getLength(); i++) {
    T curr = array.get(i);
    double area = curr.getArea();
    if (area > maxArea) {
      maxArea = area;
      maxObj = curr;
    }
  }
  return maxObj;
}
```

Target Typing:

- Type inferencing involving the type of the expression

```
Shape o = A.findLargest(new Array<Circle>(0));
```

- Target typing (*Shape o*) ➔ returning type ***T <: <? extends Shape>***
- Type parameter bound ➔ ***T <: <? extends GetAreable>***
- Argument *array*: *Array<Circle> <: Array<? extends T>* ➔ ***T <: <? super Circle>***
- Most specific: **T** is **Circle**

```
Shape o = A.<Circle>findLargest(new Array<Circle>(0));
```

14. Immutability

Concepts:

- Make all fields & uninheritable classes `final`
- Make all modifying methods return new instance

Advantages:

- Ease of understanding
- Safe sharing of objects
  - Create & cache common instance: origin/none/empty/etc.
  - Private constructor, public static factory method
- Safe sharing of internals
  - Fields of different instance point to same DS, no copying
- Safe Concurrent Execution
  - Instances remain unchanged regardless of execution order

15. Nested Classes

Intuition:

- Tightly coupled with container class (no use outside)
- Encapsulate complex information of container class in a field (helper)
- Can access private fields & methods of container class
- Declared private if possible ➔ keep within abstraction barrier
- Static nested class: only access static fields/methods
- Non-static nested class (inner class): access all fields and methods

```
class A {
  private int x;
  static int y;

  class B {
    void foo() {
      x = 1; // accessing x from A is OK
      y = 1; // accessing y from A is OK
    }
  }

  static class C {
    void bar() {
      x = 1; // accessing x from A is not OK since C is static
      y = 1; // accessing y is OK
```

- Qualified `this`: access container instance

```
class A {
  private int x;

  class B {
    void foo() {
      A.this.x = 1; // ok
```

Local Class:

- Class declared within `{}` or method
- Keep the definition of the class and its usage closer together
- Access fields & methods of the container class through qualified `this`
- Access local variables of the enclosing method

Variable Capture:

- When a method returns, all local variables removed from stack
- Instance of local class may still exist if there is a reference to it
- Local class instance contains copy of *effectively final* local variables
  - Effectively final: explicitly declared `final` or not reassigned

Anonymous Class

- Unnamed class declared and instantiated in a single statement
- Format: `new X(args){body}`
- Anonymous class extends class X/implements interface X (only 1)

```
class A {
  int x = 1;

  void f() {
    int y = 1;

    class B {
      void g() {
        x = y; // accessing x and y is OK.
      }
    }

    new B().g();
  }
}
```

```
interface C {
  void g();
}

class A {
  int x = 1;

  C f() {
    int y = 1;

    class B implements C {
      void g() {
        x = y; // accessing x and y is OK.
      }
    }

    B b = new B();
    return b;
  }
}
```

b captures y

```
A a = new A();
C b = a.f();
b.g();
```

16. Functions

Pure Functions

- Computes and returns an output (no side-effects)
- Same input ➔ same output every time (referential transparency)

First-Class Citizen

- Functions as instances of classes ➔ pass around, return, assign
- Basically acts like a variable

Lambda Expression

- Functional Interface: interface w/ only 1 abstract method (e.g., Comparator, Transformer), annotated with @FunctionalInterface
- No ambiguity about which method is being overridden ➔ can simplify to: `Transformer<Integer, Integer> incr = x -> x + 1;`
- Method Reference: double-colon notation refers to:
    - Static method in class (`Box::of`)
    - Instance method of class/interface (`x::compareTo`)
    - Constructor of class (`Box::new`)
- Curried Functions: `Transformer<Integer, Transformer<Integer, Integer>> add = x -> y -> (x + y);`
    - `add.transform(1).transform(1); // 1 + 1`
    - n-ary function ➔ sequence of unary functions
    - Delayed evaluation (store arguments in function)
- Lambda as Closure: variable capture of env (e.g., instance of class)
    - Point origin is captured

```
Point origin = new Point(0, 0);
Transformer<Point, Double> dist = origin::distanceTo;
```

18. Lazy Evaluation

- Using FIs like Producer ➔ Don't evaluate unless needed
- Memoisation: Don't repeat evaluations, store values

17. Box<T>/Maybe<T>

Lambda as a Cross-Barrier State Manipulator

- Keep contents of the Box hidden (no getter/setter)
- Methods accept lambdas to apply on the item (maintain abstraction)

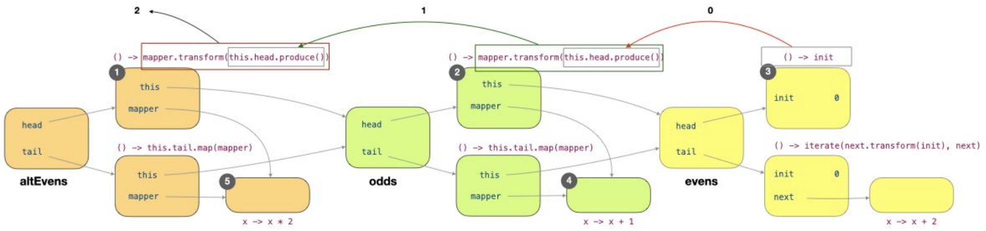
Maybe

- Can hold null values ➔ keep functions pure (that may return null)
- Internalise null checks ➔ prevent NullPointerExceptions

19. InfiniteList<T>

Intuition

- Lazily-evaluated, 2 fields: head producer & tail producer
- Methods: generate(Producer) & iterate(T, Transformer)
- `iterate(0,x➔x+2).map(x➔x+1).map(x➔x*2)`



20. Streams

- Build: Stream.of/generate/iterate(), Arrays/List::stream
- Terminate: s.forEach(*function*)
- Intermediate:
    - Stateful & bounded: keep track of states & call on finite stream (e.g., sorted, distinct)
    - Truncate: infinite ➔ finite (e.g., limit, takeWhile)
    - peek(*function*): apply lambda (Consumer) on "fork" of stream
    - reduce(*bifunction*), element matching: none/all/anyMatch(*predicate*)
- Consumed once: cannot iterate multiple times ➔ IllegalStateException

21. Loggable<T>

- 2 fields: value & log string
- flatMap(*Transformer<T,Loggable>*): let lambda update value & log

22. Monad

- Contains a value + side information (e.g., log, isPresent, isEvaluated)

Laws

1. Left Identity Law: Monad.of(x).flatMap(x ➔ f(x)) == f(x)
2. Right Identity Law: monad.flatMap(x ➔ Monad.of(x)) == monad
3. Associative Law: monad.flatMap(x ➔ f(x)).flatMap(x ➔ g(x)) == monad.flatMap(x ➔ f(x).flatMap(y ➔ g(y)))

Functors

- Contains value (no side info), lambdas can be applied sequentially

1. Identity Law: functor.map(x ➔ x) == functor
2. Composition Law: functor.map(x ➔ f(x)).map(x ➔ g(x)) == functor.map(x -> g(f(x))

23. Parallel Streams

Parallel & Concurrent Programming

- Concurrency: multithreading on single-core processor (illusion)
  - Separate unrelated tasks into threads, write threads separately
  - Improves processor utilisation (switch threads while waiting)
- Parallelism: multi-core/>1 processor (real)

Parallel Streams

- Add .parallel() (last call w/ sequential() wins) or call parallelStream()
- Breaks stream into subsequences and runs in parallel ➔ tasks that complete first will be operated on first ➔ may be different order
- To reorder, use forEachOrdered() instead of forEach (less efficient)
- Embarrassingly Parallel: little effort needed to divide the tasks (no communication or dependency between the processes)
- Stream ops must not interfere w/ stream data, most of the time stateless, minimum side effects ➔ parallel execution output correct
- ConcurrentModificationException: modifying stream during execution
- Stateless: doesn't depend on a state that might change during exec
- Side-effects: e.g., forEach modifies ArrayList (non-thread-safe DS) ➔ use .collect(Collectors.toList) or thread-safe DS (CopyOnWriteAL)
- Associativity: parallel reduce ➔ combiner c, accumulator a
  - c.apply(identity, i) == i
  - Associative: order of applying must not matter
  - Compatible: c.apply(u, a.apply(identity, t)) == a.apply(u, t)
- Thread creation overhead ➔ not always improve performance
- Ordered streams maintain order ➔ expensive to parallelise findFirst, limit, skip, etc. ➔ if order not important, add .unordered()

24. Threads

- Synchronous Programming: method blocks until return ➔ not efficient
- Thread: single flow of execution in a program, runs asynchronously
- new Thread(*Runnable (void run()*), start() returns immediately
- Thread.currentThread().getName(); "main": autocreated on main() call
- Thread.sleep(ms): pause; isAlive(): still running?

25. Asynchronous Programming (CompletableFuture<T>)

- Monad side info: whether value it promises is ready/task is completed
- Create: completedFuture, runAsync(*Runnable*), supplyAsync(*Supplier*), allOf (when all CF done), anyOf (when any CF done)
- Chain: thenApply (map), thenCompose (flatMap), thenCombine (cmb)
  - Async versions: thenApplyAsync etc. (runs in order, diff thread)
- Result (synchronous): get() or join() ➔ ensures program finishes
  - get() throws InterruptedExc. & ExecutionExc. ➔ handle
- CompletionExc.: contains an exception ➔ handle(*BiFunction(value, exception, return value)*) ➔ either value/exception is null