Software program

- a collection of data variables and instructions on how to modify those variables.
- Translation from a high-level programming language to machine code.
  - Compiler. (Software tool)
    - machine code is saved into an executable file which can be executed later.
  - Interpreter
    - Software that reads in the program one statement at a time, interprets it, then executes it.
  - Java programs are executed in 2 ways.
    - compiled into bytecode, then during execution the bytecode is interpreted and compiled on-the-fly by JVM into machine code.
    - Java program can also be interpreted by the Java interpreter.
- Java is a statically typed language - once compile-time type of a variable is declared, it cannot be changed

- type system: a set of rules that govern how types can interact with each other.
- Java compiler can help check for type errors.

OOP

- int initialises to 0
- Reference Types initialises to null
- Encapsulation
  - exposes just the right method interface for others to use
  - maintains the abstraction barrier.
  - concept of keeping all data and functions operating on the data related to a composite data type together
- keyword 'new' creates an object of a given class
- '.' notation to access an object's fields and methods.
- Information Hiding
  - private, public modifier. — private ⇒ cannot access even if child class.
  - protects the abstraction barrier from being broken. — client vs implementer
  - protection is enforced by compiler at compile time.
- keyword 'this' - reference variable that refer back to itself.
  - distinguishes between 2 variables of the same name.
- Tell, Don't Ask.
  - accessor (getter), mutator (setters)
  - if we provide a setter and getter for every private field ⇒ breaking encapsulation.
  - Right approach: Implement a method within the class that does whatever we want the class to do.
    In such a way that when the implementer decides to change the representation of the object and remove the direct access to the field, the client does not need to change anything.
- To associate a method or field (global values) with a class in java, use 'static'
- 'final' to indicate the value of the field will not change.

- we should tell an object what task to perform as a client, rather than asking for the data and doing the computation ourselves.

super()
- calls no args constructor if not specified in child class
  - if no args constructor exist.

Class fields/methods.
- associates with a class, not an instance of a class.

Class fields are useful for storing pre-computed values / configuration parameters associated with a class rather than individual objects.

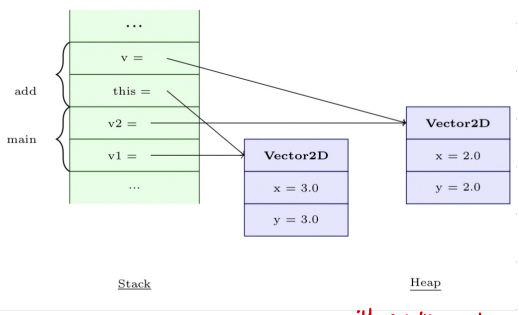
Hard Stuff.

Composition.

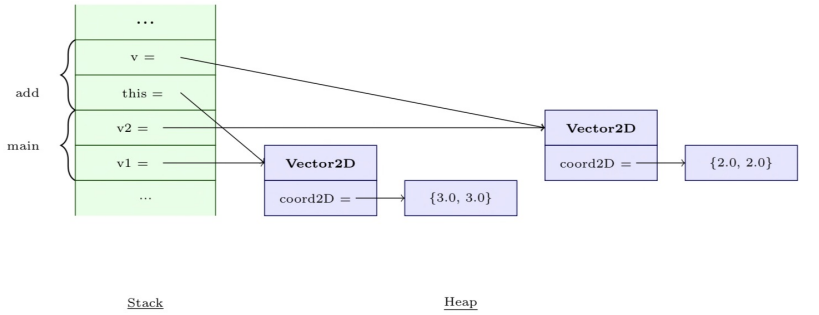
- models a HAS-A relationship between 2 entities
  - eg. circle has a point as the center.
- Sharing references (Aliasing)
  - common source of bugs
  - avoid → even though computational resources is not optimised as there are tons of objects.
  - use immutability.

Heap and Stack

- typically JVM partitions the memory into several regions:
  1. Method area → storing code for the methods.
  2. Metaspace → storing meta information about classes.
  3. Heap → storing dynamically allocated objects
  4. Stack → local variables and call frames
- Heap is the region in memory where all objects are allocated in and stored. → memory stays as long as there is a reference to it.
- Stack is the region where all variables (incl. primitives and object references) are allocated in and stored. → memory deallocated when a method returns
- Java call by value for primitive types, call by reference for objects



with primitive val.



with objects (eg. coords)

Inheritance

- subtyping → If S <: T, then any piece of code written for T must also works for S.
- using extends
- super calls the constructor for superclass.
- model for IS-A relationship.

Use composition (compose a new class) for HAS-A relationship

Use inheritance for a IS-A relationship. Make sure inheritance preserves the meaning of subtyping

Run-Time type

e.g. ColouredCircle <: Circle

Circle c = new ColouredCircle(p, 0, blue)

Circle is the compile-time type of c.

c is now referencing an object of subtype ColouredCircle

Since the assignment happen during run-time,

∴ runtime type of c is ColouredCircle.

Method Overriding

- able to alter the behaviour of an existing class.
- Method signature: method name + number, type and order of its parameters.
- Method descriptor: method signature + return type.
- when a subclass defines an instance method with the same method descriptor as an instance method in the parent class,
  instance method in the subclass overrides the instance method in the parent class

Method Overloading

- When we have 2 or more methods in the same class, with the same name
  but differing method signature.
- change a method's parameter number, type, order to overload it.
- says nothing about the name of the arguments
- possible to overload the constructor and static class methods.

Polymorphism

- related to method overriding.
- which method to invoke is decided during runtime, depending on the runtime type of the Object.
  - dynamic binding.

```
@Override
public boolean equals(Object obj) {
  if (obj instanceof Circle) {
    Circle circle = (Circle) obj;
    return (circle.c.equals(this.c) && circle.r == this.r);
  }
  return false;
}
```

type casting.

- equals takes in a parameter of CT-type Object.
  checks RT-type of obj if its a subtype of Circle (using instanceof)
  type cast assures the RT-type of obj is Circle
- Java is strongly typed and very strict about type conversions.
  - allows type casting from T → S if S<:T. (narrowing type conversion)
    - needs the type casting to be explicit and validation during run-time. (bad casting leads to runtime error).
  - (widening type conversion) is always allowed

Method Invocation

- Dynamic Binding → only applies to instance method invocations

Step 1: Compile Time

- Java determines the method descriptor of the method invoked, using the compile-time type of the target.
- multiple method that could be correctly invoked?
  - choose most specific one.

    method M is more specific than N if the arguments to M can be passed to N without compilation errors.
- stores method descriptor in the generated bytes

  (do not incl. info about class that implements this method.)
  - determined during RT.

Step 2: Run Time

- when a method is invoked, the method descriptor from CT is retrieved. ← CT type of the argument.

  Then RT - type of target is determined. → R.

  Java then looks for an accessible method with the matching descriptor in R

  If no such method is found → search will continue up the class hierarchy.
- first method implementation with a matching descriptor found will be the one executed.

Class methods (static) does not support dynamic binding.
- method to invoked is resolved during CT.
- step 1 will be taken, but step 2 will be 'just executing code found in step 1.

LSP

- Let $\phi(x)$ be a property provable about objects x of type T.

  Then $\phi(y)$ should be true for objects y of type S where $S <: T$.
- A subclass should not break the expectations set by the superclass.

  If a class B is substitutable for a parent class A then it should be able to pass all test cases of the parent class A.
- final class → bans a class from being inherited.

  final method → bans from being overriden. → critical for the correctness of the class.
- In another words, subclass must have the same implementation, but more detailed than the superclass.

Wrapper Class

- wraps around primitive types
- they are reference types, created with 'new'
- instances are stored on the heap.
- immutable - once created, cannot change.
- auto-boxing / unboxing.
- cost of allocating memory for the object and collecting garbage.
  - less efficient

Abstract Class

- a very general class that should not / could not be instantiated.
- one or more of its instance methods cannot be implemented without further details. → keyword 'abstract'
- reverse: Concrete class (no abstract method)

Interface

- is a type declared with interface
- interface can extends multiple interfaces,

  class can only extend one class, implements multiple interfaces
  - If class C implements an interface I, then C<:I → a type can have multiple supertypes

- fields in interface are static and final.

  but not for abstract class

Types

Casting

- Ask compiler to treat the object returned.
- Need to be sure about the run-time type.
- Leads to fragile code → depends on RT type, compiler won't catch the error.

Variance of types

- How the subtype relationship between complex types relates to the subtype relationship between components.
- Covariant if $S <: T \rightarrow C(S) <: C(T)$

  Contravariant if $S <: T \rightarrow C(T) <: C(S)$

  invariant if neither.
- Java Array is Covariant

  `S <: T`

  `S[] <: T[]`.
  - prone to run-time errors (even without typecasting)
  - eg.

```
1  Integer[] intArray = new Integer[2] {
2    new Integer(10), new Integer(20)
3  };
4  Object[] objArray;
5  objArray = intArray; → allowed since array is covariant.
6  objArray[0] = "Hello!"; // <- compiles!
```

Compiler thx Object[] refers to Integer[]

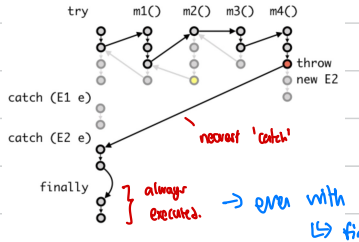
Exception

- Throwing Exceptions. (using try-catch-finally)
  - keyword 'throw' 'new'
  - Checked vs unchecked exception.

"Checked" for exceptions at Compile time.
- deal w at CT.
  - try-catch block
  - throws
- no control. (even with perfect code) eg. FileNotFound Exception.
  - must be handled else won't compile
- programmer's error eg. NullPointer Exception
  - not explicitly caught/thrown.
  - version of RuntimeException.
    - can do throw new RuntimeException();
- no compilation error
  - should use try-catch block.

  - Parsing the error:

    propagate automatically down the stack until either caught or throws an error.
  - Control Flow of Exceptions



→ even with return in the try block.
↳ finally's return (if have) will override.

  - Can create own exception by 'inheriting' from existing ones.
    - for additional useful information to the exception handler.
  - overriding method that Throw Exceptions.
    - must follow LSP → throw only the same, or a more specific checked exception than the overriden method.
    - The caller of the overriden method cannot expect any new checked exception beyond what has already been 'promised' in the method specification.

— Good Exceptions

- Catch exception to clean up (using finally)
  - if exception not handled, code that deallocates the resource is not called.
- Do not catch all vanilla Exception.
  - unable to figure out if something went wrong
- Do not exit program when caught exception.
  - prevent calling function from cleaning up the resource.
- Do not break abstraction barrier.
  - handle the implementation-specific exceptions within the abstraction barrier.
- Do not use it as a control flow mechanism
  - meant to handle unexpected errors not logic of program.
  - less efficient, might not be correct as exception may be triggered by something else in the program.

— Error class

- program should terminate as there is no way to recover from these errors.
- eg. StackOverflowError.
- no need to create/handle such error.

— Error and Exception are children of Throwable.

Generics Types (invariant) → Complex types

- takes other types as type parameters.
- use "< and >" to declare generic types.
- Cannot mix up the types
- To use: pass in type arguments — non-generic / generic / another type parameter that has been declared.
- once instantiated → parameterised type.
- only reference types can be used as type arguments.

To use Java generics effectively, you must consider the following restrictions:

- Cannot Instantiate Generic Types with Primitive Types
- Cannot Create Instances of Type Parameters
- Cannot Declare Static Fields Whose Types are Type Parameters
- Cannot Use Casts or `instanceof` With Parameterized Types
- Cannot Create Arrays of Parameterized Types
- Cannot Create, Catch, or Throw Objects of Parameterized Types
- Cannot Overload a Method Where the Formal Parameter Types of Each Overload Erase to the Same Raw Type

Static need to have <T> instead or not?

Generic Methods

- type parameter T is declared within < >, added before the return type of the method.
- T is then scoped within the whole method.
- eg. A.<String> convert(strArray, 123);
- normally static methods.

- either GenericExample.<String>shout("John")
- or shout("John")

Bounded Type Parameters.

- use extends
- T must be a subtype of S.
- normally declared at the introduction of the generic type.

- can have multiple bounds, class must be specified first, only one class, can extends multiple interfaces.

↳ use &.

Type Erasure. (for generics)

- Java takes a code sharing approach, instead of creating a new type for every instantiation, it chooses to erase the type parameters and type arguments during compilation. (after type checking)
- There is only one representation of the generic type in the generated code, representing all the instantiated generic types, regardless of the type arguments
- For backwards compatibility.

```
class Pair<S,T> {
  private S first;
  private T second;

  public Pair(S first, T second) {
      this.first = first;
      this.second = second;
  }

  S getFirst() {
      return this.first;
  }

  T getSecond() {
      return this.second;
  }
}
```

→

```
class Pair {
  private Object first;
  private Object second;

  public Pair(Object first, Object second) {
      this.first = first;
      this.second = second;
  }

  Object getFirst() {
      return this.first;
  }

  Object getSecond() {
      return this.second;
  }
}
```

If type is bounded, then it is replace by the bound instead.

Where a generic type is instantiated and used, the code

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

is transformed into

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

↳ ensures casting will not lead to ClassCastException during run-time.

- Generics and Arrays Can't Mix.

```
// create a new array of pairs
Pair<String,Integer>[] pairArray = new Pair<String,Integer>[2];

// 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);
```

↳ type checking is done during RT.

Heap pollution.

→

```
// create a new array of pairs
Pair[] pairArray = new Pair[2];

// 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(3.14, true);
```

due to type erasure, no information about the type argument.

- Causes heap pollution (variable of a parameterized type refers to an object that is not of parameterized type)
→ ClassCastException

```
// getting back a string?  -- now we get ClassCastException
String str = pairArray[0].getFirst();
```

- Java array is reifiable (a type where full type info is available during run-time)
  ↳ can check when we store into the array and maintain the type of the array ⇒ throws an ArrayStoreException if mismatch.
- Java generic is not reifiable due to type erasure.

Unchecked Warnings

- A message from the compiler that it has done what it can, because of type erasures → a runtime error that it cannot prevent.
- compiler can't guarantee the code is safe anymore.
- @SuppressWarning
  - use in the most limited scope to avoid unintentionally suppressing warnings that are valid concerns from the compiler.
  - only if we are sure it will not cause a type error later.
  - must always add a note to explain why a warning can be safely suppressed
  - cannot apply to assignment but only to declaration.

Raw types.

- generic type used without type arguments
- what code will look like after type erasure
- missing raw with parameterised type will lead to an error.
- Never used in code.
  - unless as an operand of the `instanceof` operator (since instanceof checks for RT type and type arguments have been erased).

Wildcards

- Upper-bounded (covariance)
  - If $S <: T$
  - then $A<? \text{ extends } S> <: A<? \text{ extends } T>$ ('anything below' T & T.)
  - note: if $K <: S$ then $A<K> <: A<? \text{ extends } S>$.
  - also $A<S> <: A<? \text{ extends } S>$
- Lower-bounded (contravariance)
  - If $S <: T$
  - then $A<? \text{ super } T> <: A<? \text{ super } S>$ ('anything above' S & S.)
  - also $A<S> <: A<? \text{ super } S>$.
- PECS

  Producer Extends; Consumer Super
  - Depends on the role of the variable
    - If the variable is a producer that returns T → declared as ? extends T.
    - If the variable is a consumer that accepts T → declared as ? super T.
- Unbounded
  - eg. $Array<T> <: Array<?>$ for every T.
  - Object is the supertype of all reference types
    - We want to write a method that takes in a reference type, but we want the method to be flexible enough → make method to accept a parameter of type Object.
  - Useful when want to write a method that takes in an array of some specific type, and the method to be flexible enough to take in an array of any type.

Generally do not use wildcards as return types.

- Array<?> is an array of objects of some specific but unknown type.
- Array<Object> is an array of Object instances, with type checking by the compiler.
- Array is an array of Object instances, without type checking.

- Using ?, we can remove exceptions for raw types
  1. a instanceof A<?> — unknown/erased type parameter
  2. new Comparable<?>[10].

  ↓ reifiable since no type info is lost.

It's important to note that `List<Object>` and `List<?>` are not the same. You can insert an `Object`, or any subtype of `Object`, into a `List<Object>`. But you can only insert `null` into a `List<?>`

Type Inference

- Java will look among the matching type that would lead to successful type checks, and pick the most specific one.

- eg.

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

- find all possibilities of types, get the most specific one.

eg.

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

We have a few more constraints to check:

- Due to target typing, the returning type of `T` must be a subtype of `Shape` (including `Shape`)
- Due to the bound of the type parameter, `T` must be a subtype of `GetAreable` (including `GetAreable`)
- `Array<Circle>` must be a subtype of `Array<? extends T>`, so `T` must be a supertype of `Circle` (including `Circle`)

Intersecting all these possibilities, only two possibilities emerge: `Shape` and `Circle`. The most specific one is `Circle`, so the call above is equivalent to:

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

Part 2: Functional Programming.

@SafeVarargs
⇒ prevent unchecked warnings since arrays and generics do not mix well.

Java Tips
varargs
of (T... items)
Java syntax for a variable number of arguments of the same type T.
Syntactic sugar for passing in an array of items to a method.

Previously on how to deal with software complexity:

1. encapsulation by hiding complexity behind abstraction barriers.
2. strong type system ⇒ adheres to subtyping substitution principle.
3. applying the abstraction principles ⇒ revising code written as functions, classes and generic types.

Now: avoid changing the code altogether.

Immutability

- Classes can be made immutable
- instances of immutable class cannot have any visible changes outside its abstraction barrier.
- every call of the instance's method must behave the same way throughout the lifetime of the instance.
- keyword: "final"
  - declare immutable classes as final to disallow inheritance
  - once an instance is created, remains unchanged outside the abstraction barrier.
  - copy-on-write semantic ⇒ avoid aliasing bugs without creating excessive copies of objects.
    ⇒ immutable classes allows us to share all the references until we need to modify the instance, in which case we make a copy.
- Advantages:
  - removes the risk of potential bugs when we use composition and aliasing.
  - Ease of understanding: do not have to trace through all the methods to make sure they don't modify the result.
  - Enabling safe sharing of objects: reducing the need to create multiple copies of the same object ⇒ creates and caches a single copy and returns the same copy when required.
  - Enables safe sharing of internals: internal fields are guaranteed not to mutate.
  - Enabling safe concurrent execution: ensure that regardless how the code interleaves, our object remains unchanged.

Nested Class

- It is a class defined within another containing class.
- used to group logically relevant classes together.
- normally would have no use outside of the container class.
- encapsulates information within a container class ⇒ when implementation of container class becomes too complex.
- can access fields and methods of the container class, incl. those declared as "private" ⇒ itself is a field of the container class.
- introduced only if it belongs to the same encapsulation as the container class ⇒ else leak implementation details to the nested class.
- static / non-static
  - ⇓ associated with the containing class (not an instance)
    ⇓ can only access static fields and methods of containing class
    ⇓ "static nested class"
  - access all fields and methods of the containing class
    ⇓ "inner class"
- qualified 'this' reference ⇒ prefixed with the enclosing class name
  ⇒ differentiate btw this for inner class and this of enclosing class.

Local Class

- A class declared within a function ⇒ like a local variable.
- It is scoped within the method just like a local variable.
- It has access to variables of the enclosing class through qualified 'this' reference.
- It can access the local variables of the enclosing method.

Variable Capture

- When a method returns, all local variable of the methods are removed from the stack.
  But, an instance of the class might still exist
- Even though a local class can access the local variables in the enclosing method, the local class makes a copy of local variables inside itself.
  ⇒ A local class captures the local variables.

Eg.
```
interface C {
    void g();
}
class A {
    int x = 1;
    C f() {
        int y = 1;
        class B implements C {
            void g() {
                x = y;
            }
        }
        B b = new B();
        return b;
    }
}
```

```
A a = new A();
C b = a.f();
b.g(); ⇒ What is the value of y?
```

Effectively final

- Java only allows a local class to access variables that are explicitly declared final or implicitly (effectively) final.
- A implicitly final variable does not change after initialisation.
- Prevents the code from not compiling due to similar mistakes

Anonymous class

- A class where it is instantiated in a single statement.
- Does not require a class name.
- Has format: `new X (arguments) { body }`
  - X: A class/interface that anonymous class extends/implements ⇒ Cannot extend other class and implement an interface at the same time.
    - Cannot be empty
  - arguments: arguments that you want to pass into the constructor of the anonymous class. If implementing an interface, no constructor but use '()'.
  - body: body of the class. Cannot have a constructor for an anonymous class

- Are just like local class ⇒ captures the variables of the enclosing scope as well.
  ⇒ Same rules to variable access as local classes.

Side Effect-Free Programming

Functions

- A mapping from a set of inputs (domain) X to a set of output values (codomain) Y.
- $f: X \to Y$.
- Every input domain must be map to exactly one output but multiple inputs can map to same output.
- Not all values in the codomain need to be mapped.
- Applying f(x) does not change the value of x, or any other information ⇒ no side effects.
- Referential Transparency: Let f(x) = a, then for all f(x), we can replace it with a. (same conversely)
  ⇒ resulting formulas are still equivalent.
- Functions as first-class citizens: assign functions to a variable, pass it as parameters, return a function from another function.

Pure Functions

- Given an input, the function computes and returns an output.
- Does nothing else. ⇒ Does not print to screen, write to filer, throw exceptions, change other variables, modify the values of the arguments.
  ⇒ Does not cause any side-effects.
- Must be deterministic ⇒ given the same input, function must produce the same output, every single time.
  ⇒ ensure referential transparency.
- Eg.

```
void sortNames(List<String> name){
    Comparator<String> cmp = new Comparator<String>(){
        public int compare(String s1, String s2){
            return s1.length() - s2.length();
        }
    };
    Names.sort(comp);
}
```

→ Comparison function implemented as a method in an anonymous class that implements an interface.
An instance of this anonymous class or a function.
⇒ can pass it around, return it from a function, assign it to a variable, just like any other reference type.

Lambda Expressions

- An interface with one method ⇒ functional interface.

```
eg. @FunctionalInterface
    interface Transformer<T,R>{
        R transform(T t);
    }
```

- advantage: no ambiguity about which method is being overriden by an implementing subclass.

```
eg. Transformer<Integer, Integer> square = x -> x*x;
```

- Keyword: '->'
- use () if no parameter.

Curried Functions

- We can build functions that takes in multiple arguments.

```
eg. Transformer<Integer, Transformer<Integer, Integer>> add = x -> y -> x+y;
    Call add.transform(1).transform(1)
```

takes in an Integer object
returns an unary function over Integer.
It is a higher-order function that takes in a single argument and returns another function.

- The technique that translates a general n-ary function to a sequence of n unary functions ⇒ currying
- After currying ⇒ a sequence of curried functions.
- We can evaluate diff. arguments at a different time
- Partially apply a function first ⇒ useful if one of the arguments is not used often/ expensive

partially save the results as a function and continue applying later.

Method References

```
eg. Transformer<Point, Double> dist = p -> origin.distanceTo(p);
    Transformer<Point, Double> dist = origin::distanceTo;
                                      ^ method reference.
```

- Use to refer to a static method in a class
  instance method of a class or interface
  constructor of a class.
- eg. A::foo ⇒ either (x, y) -> x.foo(y) or (x, y) -> A.foo(x, y)
  - When compiling, Java would search for matching method, perform type inference to find the method that matches the given method reference.
  - A compilation error will be thrown if there are multiple matches or if there is ambiguity in which method matches.

Lambda as Closure

- A lambda exp. also stores the data from the environment where it is defined.
- i.e. variable capture
- Function + enclosing environment ⇒ closure.
- Makes code cleaner, fewer param, duplicate code.
- Can separate logic to do diff. tasks in a diff. part of program easier.

dynamically create functions as needed, save them, and invoke them later.

Box and Maybe

Lambda as a cross-barrier state manipulator.

- every class has an abstraction barrier between the client and the implementer.
- internal states of the class are heavily protected and hidden.
- the implementer selectively provides a set of methods to access and manipulate the internal state of instances.
  - ⇒ allows the implementer to control what client can do to the internal states.
  - ⇒ good if want to build abstractions over specific entities such as shapes / data structures such as stacks.
  - ⇒ not flexible enough to build general abstraction.

Suppose we have:

```
class Box <T>{
    private T item;
}
```

- We want to keep item hidden and we want certain rules and maintain some semantics about the use of the item. ⇒ no setters or getters, so client may not break our rules.
- Provide methods that accept a lambda expression, apply the lambda expression on the item, and return the new box with the new value.

```
public <U> Box<U> map (Transformer <? super T, ? extends U> transformer) {
        if (!isPresent()){
            return empty();
        }
        return Box.ofNullable (transformer.transform (this.item));
}
```

⇒ takes in a lambda expression and allows us to arbitrarily apply a function to the item.

```
public Box<T> filter(BooleanCondition <? super T> condition) {
        if (!isPresent() || !(condition.test(this.item)) {
            return empty();
        }
        return this;
}
```

⇒ allows us to perform an arbitrary check on the property of the item.

- Method such as these, which accepts function as a parameter, allows the client to manipulate the data behind the abstraction barrier without knowing the internals of the object.
- Treating expression as "manipulators" that we can pass in behind the abstraction barrier and modify the internals arbitrarily for us, while the container or the box tries to maintain the semantics for us.

Maybe

- It is an option type that is a wrapper around a value that is either there or is null.
- Allows us to write code without worrying about the possibility that our value is missing
- When we call map on a missing value, nothing happen.
- Internalises all the checks for 'null' on the client's behalf
- Ensures that if 'null' represents a missing value, then the semantics of the missing value is preserved throughout the chain of 'map' and 'filter' operations.
- Within its implementation, Maybe<T> do the right thing when the value is missing to prevent us from encountering "NullPointerException".
- There is a check for null when needed, internally, within Maybe<T>.
- This internalisation removes the burden of checking for null on the programmer and remove the possibility of run-time crashes due to missing null checks.

Lazy Evaluation ⇒ "procrastinating our computation until we really need the data".

```
@FunctionalInterface
interface Producer<T> {
        T produce();
}
```

```
Producer<String> toStr = () -> Integer.toString(1);
```

- nothing is executed by declaring the lambda expression
- Lambda expression allows us to delay the execution of code, saving them until we need it later.
- Enables lazy evaluation.
    - ⇒ build up a sequence of complex computations, without actually executing them, until we need to.
    - ⇒ expressions are evaluated on demand when needed.

Memoization ⇒ "not to repeat ourselves"

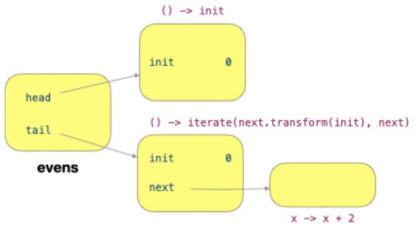
- If we have computed the value before, we can cached (memoize) the value, keep it somewhere, so that we don't need to compute again.
- Useful if the function is PURE.
    - ⇒ regardless of how many times we invoke the function, it always returns the same value, and invoking it has no side effects on the execution of the program.

Infinite List

- A list with possibly infinite number of elements.
- Instead of storing head and tail of the list, we can think of infinite list as consisting of 2 functions, the first generates a head, the second generates the tail.

Let's look at what gets created on the heap when we run

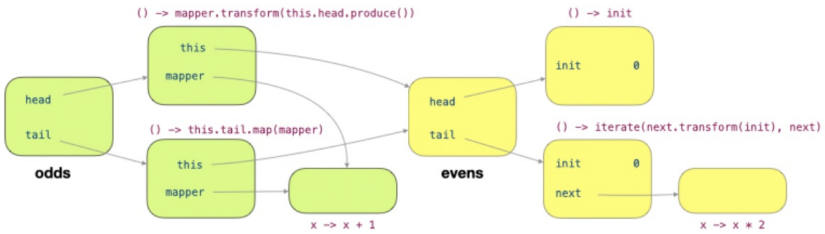
```
1  InfiniteList<Integer> evens = InfiniteList.iterate(0, x -> x + 2); // 0, 2, 4, 6, ...
```



The figure above shows the objects created. `evens` is an instance of `InfiniteList`, with two fields, `head` and `tail`, each pointing to an instance of `Producer<T>`. The two instances of `Producer<T>` capture the variable `init`. The `tail` additionally captures the variable `next`, which itself is an instance of `Transformer<T,T>`.

Next, let's look at what gets created on the heap when we run

```
1  InfiniteList<Integer> odds = evens.map(x -> x + 1); // 1, 3, 5, ...
```

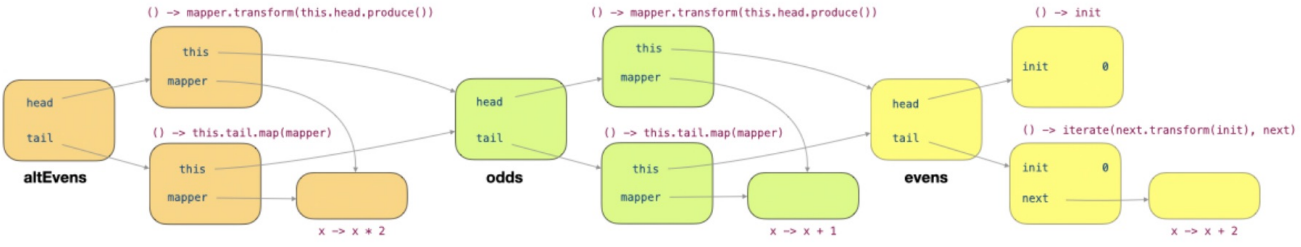


The figure above shows the objects added. `odds` is an instance of `InfiniteList`, with two fields, `head` and `tail`, each pointing to an instance of `Producer<T>`. The two instances of `Producer<T>` capture the local variable `this` and `mapper` of the method `map`. `mapper` refers to an instance of `Transformer<T, T>`. Since the method `map` of `evens` is called, the `this` reference refers to the object `evens`.

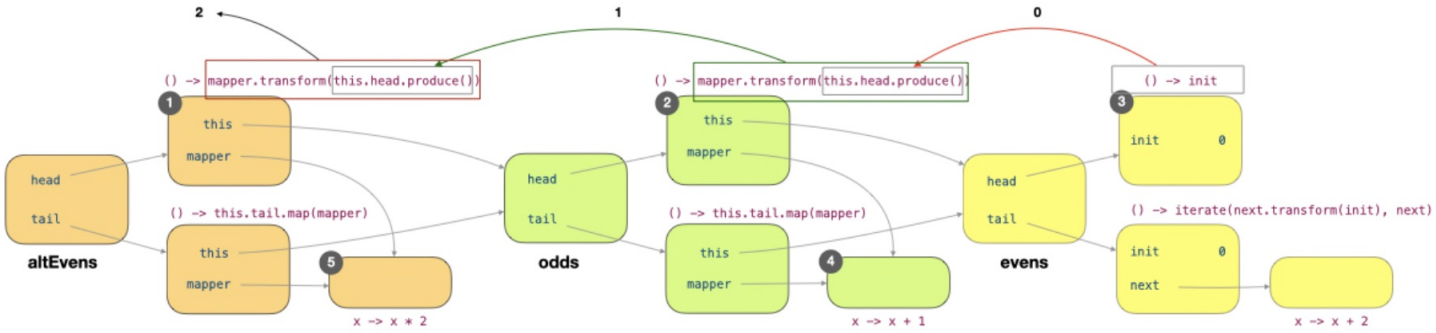
After calling

```
1  InfiniteList<Integer> altEvens = odds.map(x -> x * 2); // 2, 6, 10, ..
```

We have the following objects set up.



Let's now trace through what happens when we call `altEvens.head()`. This method leads to the call `this.head().produce()`, where `this` refers to `altEvens`. The call to `produce` invoked `mapper.transform(this.head.produce())` of the producer labelled 1 in the figure below. This leads to `this.head.produce()` of this producer being called. Within this producer, `this` refers to `odds`, and so `this.head.produce()` invoked `mapper.transform(this.head.produce())` of the producer labelled 2. Now, `this` refers to `evens`, and `this.head.produce()` causes the producer `() -> 0` (labelled 3) to produce 0.



The execution now returns to the invocation of `mapper.transform(this.head.produce())` and call `mapper.transform(0)` (labelled 4). This returns the value 1, which we pass into the `mapper.transform(1)` (labelled 5). The `mapper` is `x -> x * 2` so we have the result 2, which we return from `altEvens.head()`.

Streams

CS2030S	java.util.function
`BooleanCondition<T>::test`	`Predicate<T>::test`
`Producer<T>::produce`	`Supplier<T>::get`
`Transformer<T,R>::transform`	`Function<T,R>::apply`
`Transformer<T,T>::transform`	`UnaryOp<T>::apply`
`Combiner<S,T,R>::combine`	`BiFunction<S,T,R>::apply`

monad and functor — not monad or functor

CS2030S	Java version
`Maybe<T>`	`java.util.Optional<T>`
`Lazy<T>`	N/A
`InfiniteList<T>`	`java.util.stream.Stream<T>`

Building a Stream

- static of method `Stream.of(1,2,3);`
- generate & iterate method Similar to infiniteList.
- convert an array to stream using `arr.stream()`
- convert a List instance (or any Collection instance) `list.stream()`

Terminal Operations

- An operation on the stream that triggers the evaluation of the stream.
- Chain a series of intermediate operation together, ending with a terminal operation.
- The forEach method is a terminal operation that takes in a stream and applies a lambda expression to each element.
  - does not return any value (void)
  - Consumer<T> functional interface.

Intermediate Stream Operations

- An intermediate operation on a stream returns another Stream.
  - java provides 'map', 'filter', 'flatMap', and other intermediate operations.
  - lazy and do not cause the stream to be evaluated.
- FlatMap
  - behaves similarly
  - takes in a lambda expression that transforms every element in the stream into another stream.
  - the resulting stream is flattened and concatenated together.
- Stateful and Bounded Operation

  Stateful = need to keep track of some state to operate

  eg. sorted and distinct.

  - return a stream where the elements are sorted.
  - without arguments, it sorts by natural order
  - defined by Comparable<T> interface.

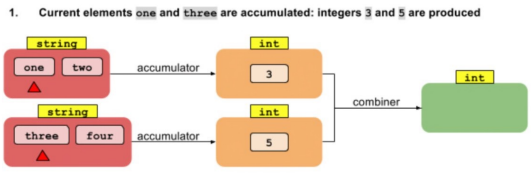
  return a stream with only distinct elements in the stream.

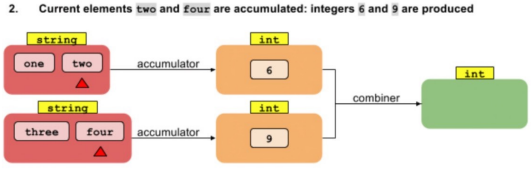
  bounded operation ⇒ can only be called on finite streams.
- Truncating an Infinite List
  - converts an infinite stream to finite stream.
    - limit ⇒ takes in int n, returns a stream w/ first n elements of the stream.
    - takeWhile ⇒ takes in a predicate, returns a stream containing the elements of the stream, until the predicate becomes false.

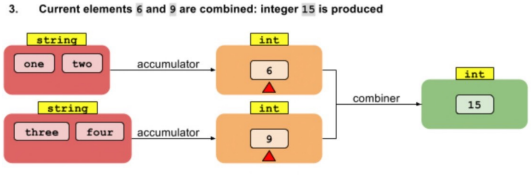
      The resulting stream might still be infinite if predicate never becomes false.
- Peeking with a Consumer
  - peek takes in a Consumer, allowing us to apply a lambda on a "fork" of the stream.
  - eg. `Stream.iterate(0, x->x+1).peek(System.out::println).takeWhile(x->x<5).forEach(x->{});`

    note: not terminal!

    must pair with a terminal operation.
- Reducing a Stream. (Terminal)
  - applies a lambda repeatedly on the elements of the Stream to reduce it to a single value.
  - takes in an identity value and an accumulating function $(x, y) \to x+y$ and returns the reduced value.
  - 2 other versions ⇒ one has null as identity.
    - ⇒ the other supports a different return type than the type of the element in the stream.
    - ⇒ requires a combiner to switch types.







- Element Matching
  - terminal operations for testing if the element pass a given predicate
  - noneMatch - returns true if none of the element passes the predicate
  - allMatch - returns true if every element passes the given predicate.
  - anyMatch - returns true if at least one element passes the given predicate.

- Consumed Once
  - Limitation of stream: can only be operated on once.
  - Cannot iterate through a stream multiple times.
  - doing so will throw "IllegalStateException"
  - will have to recreate the stream if we want to operate on the stream more than once.

Specialised Streams on Primitives: IntStream, LongStream, DoubleStream.

With a Stream, we no longer have to write loops, mutable states and counters.

- code becomes more declarative ⇒ concern about what we want at each step, much less how to do it.

* Not all loops can be translated to stream elegantly ⇒ i.e. triple nested loops.

Loggable

- There are many possible operations on int, we do not want to add a method fooWithLog for every function foo.
- One way to make Loggable general is to abstract out the int operation and provide that as a lambda expression to Loggable.

eg.
```
Loggable map (Transformer<Integer, Integer> transformer) {
    return new Loggable(transformer.transform(this.value), this.log);
}
```
⇒ `Loggable.of(4).map(x -> incr(x)).map(x -> abs(x))`

- but map allows us to only apply the function to the value.
- it is not sufficient to tell us what message we want to add to log.

⇒ use flatMap.

```
Loggable flatMap(Transformer<Integer, Loggable> transformer) {
    Loggable l = transformer.transform(this.value);
    return new Loggable(l.value, l.log + this.log);
}
```

Monad

Loggable<T> follows Maybe<T>, Lazy<T> and InfiniteList<T>:

- an of method to initialise the value and side information
- have a flatMap method to update the value and side information.
- different classes have different side info that is initialised, stored, and updated when we use of and flatMap.
  - Maybe<T> stores side info of whether the value is there or not.
  - Lazy<T> stores side info of whether the value is evaluated or not.
  - Loggable<T> stores side info of a log describing the operations done on the value.
- they are Monads. — i.e. of method shouldn't do anything extra to the value and side info. It should simply wrap the value into the Monad.
  flatMap Method should'nt do anything extra to the value and the side info. It should simply apply the given lambda expression to the value.
- they are also functors

3 laws of Monads:

Identity laws — Let 'Monad' be a type that is a monad and 'monad' be an instance of it.

1. Left identity law

   Monad.of(x).flatMap(x → f(x)) = f(x).

2. Right identity law

   monad.flatMap(x → Monad.of(x)) = monad.

i.e. Monad::of should have no side-effects.

Associative Law

- regardless of how we group the calls to flatMap, their behaviour must be the same.

monad.flatMap(x → f(x)).flatMap(x → g(x)) = monad.flatMap(x → f(x).flatMap(y → g(y)))

Functors

- Common abstraction in functional style programming.
- A simpler construction than a monad in that it answers only lambdas can be applied sequentially to the value, without worrying about side info.
- can think of a functor as an operation that supports "map".

2 laws:

1. preserving identity

   functor.map(x → x) = functor

2. preserving composition

   functor.map(x → f(x)).map(x → g(x)) = functor.map(x → g(f(x)))

Parallel

Concurrency

- Dividing the computation into subtasks: Threads.
- Allows us to separate unrelated tasks into threads, and write each thread separately.
- Improves utilisation of processor - i.e. if I/O is in one thread, and UI rendering is in another, then when the processor is waiting for IIO to complete, it can switch to the rendering thread to make sure that the slow IIO does not affect responsiveness of UI.

Parallelism

- Multiple subtasks are running at the same time
  - A processor that is capable of running multiple instructions at the same time
  - Multiple cores/processor to handle dispatched instructions so that they are executed at the same time.

All parallel programs are concurrent, but not all concurrent programs are parallel.

Parallel Stream

- Stream allows parallel operations on the elements in one single line of code.

```
eg. IntStream.range(2_030_000, 2_040_000)
        .filter(x -> isPrime(x))
        .parallel()
        .forEach(System.out::println)
```

- Stream breaks down the numbers into subsequences, and run filter and forEach for each subsequence in parallel. Since there is no coordination among the parallel tasks on the order of printing ⇒ whichever parallel task that completes first will output the result to the screen first ⇒ causing the sequence to be reordered.
  ⇒ use `forEachOrdered`
- Embarrassingly parallel

```
eg. IntStream.range(2_030_000, 2_040_000)
        .filter(x -> isPrime(x))
        .parallel()
        .count()
```

  - code produces the same output regardless of being parallelised, as the task is stateless and does not produce any side-effects. Each element is also processed individually without depending on other elements.
  - The only communication needed for each of the parallel tasks is to combine the result of count() from the subtasks into the final count.

- `parallel()`
  - adding this to the chain of calls in a stream enables parallel processing of the stream.
  - It is a lazy operation ⇒ merely marks the stream to be processed in parallel.
  - can invert the call to parallel() anywhere in the chain.
  - also can call `parallelStream()` on creating a stream.

To ensure that the output of parallel execution is correct, the stream operations must not interfere with the stream data, and most of the time must be stateless.

Side-effects should be kept to minimum.

Interference

- One of the stream operations modifies the source of the stream during the execution of the terminal operation.

```
eg. list.stream()
        .peek( name -> {
            if(name.equals("Hon")){
                list.add("Chemie");
            }                          ↘ would cause ConcurrentModificationException
        })
        .forEach(i -> {});
```

Stateful vs Stateless

↳ The result depends on any state that might change during the execution of the stream.

```
eg. Stream.generate(scanner::nextInt)
        .map(i -> i + scanner.nextInt())
        .forEach(System.out::println)
```

- generate and map are stateful since they depend on the state of the standard input.
- parallelising this may lead to incorrect output.
- additional work needs to be done to ensure that state updates are visible to all parallel subtasks.

Side Effects

- can lead to incorrect results in parallel execution.

```
eg. list.parallelStream()
        .filter(x->isPrime(x))
        .forEach(x-> result.add(x))
```

- The forEach lambda generates a side effect -> it modifies the result.
- ArrayList is a non-thread-safe data structure - if 2 threads manipulate it at the same time, an incorrect result may result.
- resolve using .collect method:

```
list.parallelStream()
    .filter(x->isPrime(x))
    .collect(Collectors.toList())
```

- or use a thread-safe data structure i.e. CopyOnWriteArrayList.

Associativity

- The reduce operation is inherently parallelisable, as we can easily reduce each sub-stream and then use the combiner to combine the results.

```
eg. Stream.of(1,2,3,4).reduce(1, (x,y)->x*y, (x,y)->x*y);
```

- This allows us to run reduce in parallel, however, there are several rules that the identity, the accumulator, and the combiner must follow:
  - combiner.apply(identity, i) = i eg. i * 1 = i
  - combiner and accumulator must be associative - i.e. the order of applying must not matter. eg. (x*y)*z = x*(y*z)
  - combiner and accumulator must be compatible - i.e. accumulator.apply(identity, t) = accumulator.apply(u, t). eg. u*(1*t) = u*t.

Note: Parallelising a stream does not always improve performance. Creating a thread to run a task incurs some overhead, and the overhead of creating too much threads might outweigh the benefits of parallelisation.

Ordered vs Unordered Source.

- A stream may define an encounter order.
- Streams created from iterate, ordered collections (List/array), from of, are ordered.
- Streams created from generate or unordered collections (Sets) are unordered.
- Some stream operations respect the encounter order. (eg. distinct and sorted preserve the original order of elements ⇒ operation is stable).
- parallel version of findFirst, limit, skip can be expensive on an ordered stream, since it needs to coordinate between streams to maintain order.
- if we have an ordered stream and respecting the original order is not important, we can call unordered() as part of the chain command to make parallel operations much more efficient.

Asynchronous Programming.

- Synchronous Programming is when the execution of our program stalls, waiting for the method to complete its execution.
  Only after the method returns can the execution of our program continue.
  ⇒ method blocks until it returns.
  ⇒ not very efficient, esp. when there are frequent method calls that block for a long period (such as methods that involve expensive computations or reading from a remote server).

Threads

- A thread is a single flow of a execution in a program.
- The new Thread(..) is the constructor to create the Thread instance.
  The constructor takes a Runnable instance as an argument. A Runnable is a functional interface with a method run() that returns void.
- start() causes the given lambda expression to run. It does not return only after the given lambda expression completes its execution.
  asynchronous execution. ↳ applies to a Thread.
- The Operating System has a scheduler that decides which threads to run when, and on which core/processor.
  Might see different interleaving of executions everytime you run the same program.

Names

- Every thread has a name
- getName() → find out the name of a thread.
- Thread.currentThread() → get the reference of the current running thread.
- There is always a "main" thread.

Sleep

- Can cause current execution thread to pause execution immediately for a given period.
- After sleep time is over, the thread is ready to be chosen by the scheduler to run again.

Note: isAlive() → periodically checks if another thread is still running
The program exits only after all the threads created run to their completion.

Limitations of Thread.

- It still takes a fair amount of effort to write complex multi-threaded programs in Java.
- Consider the situation where you have a series of tasks that we wish to execute concurrently and we want to organize them.
  We also want to handle the exceptions gracefully – i.e. if one of the tasks encounters an exception, the other tasks not dependent on it should still be completed.
  - Implementing using Threads requires careful coordination.
  - There are no method within Thread that return a value, we need the threads to communicate through shared variables.
  - There are no mechanisms to specify the execution order and dependencies among them – which thread to start after another thread completes.
  - We have to consider the possibility of exceptions in each of our tasks.
  - Drawback of Thread: overhead in creating a Thread instances, we want to reuse the Thread instances to run multiple tasks.
    ↳ this is difficult to achieve.

Higher-Level Abstraction

- Focusing on specifying the tasks and their dependencies, without worrying about the details.
- Use a Monad to chain up the computations.
- CompletableFuture<T> is a monad that encapsulates a value that is either there or not there yet.

CompletableFuture

- Key property: whether the value it promises is ready.
  i.e. the tasks that it encapsulates have completed or not.
- Creating a CompletableFuture
  - completedFuture method ⇒ creating a task that is already completed and return us a value.
  - runAsync method (takes in a Runnable lambda expression) ⇒ returns CompletableFuture<Void>. The returned CompletableFuture instance completes when the lambda expression finishes.
  - supplyAsync method (takes in a Supplier<T>) ⇒ returns CompletableFuture<T>. The returned CompletableFuture instance completes when the given lambda expression finishes.
  - create a CompletableFuture that relies on other CompletableFuture instances.
    - use allOf / anyOf methods. ⇒ takes in a number of CompletableFuture instances.
      ⇒ A new CompletableFuture created with anyOf is completed when any one of the given CompletableFutures completes.
- Chaining CompletableFuture
  - thenApply ⇒ map.
  - thenCompose ⇒ flatMap.
  - thenCombine ⇒ combine.

  The methods run the given lambda expression in the same thread as the caller.
  There is also an asynchronous version (with ...Async behind) which may cause the given lambda expression to run in a different thread (Thus more concurrency).
- Getting The Result.
  - get() is a synchronous call, it blocks until the CompletableFuture completes.
  - To maximize concurrency, we should call get() as the last step in our code.
  - get() throws InterruptedException and ExecutionException ⇒ need to catch and handle.
    InterruptedException: thread has been interrupted. ExecutionException: errors during execution. Both checked.
  - Use join() as there is no checked exception thrown.

- Handling Exceptions
  - one of the advantage of using CompletableFuture<T> instead of Thread to handle concurrency is its ability to handle exception.
  - Methods. exceptionally, whenComplete, handle

- Since the computation is asynchronous and could run in a different thread, the question of which thread should catch and handle the exception arises.
- CompletableFuture<T> keeps things simpler by storing the exception and passing it down the chain of calls, until .join() is called.
- .join() might throw CompletionException and whoever calls join() will be responsible for handling this exception.
- The CompletionException contains information on the original exception.
- If we want to continue the chaining of tasks despite exception, we can use handle method. (Takes in a BiFunction)

  eg. cf.thenApply(x->x+1)
  .handle((t,e)-> (e==null) ? t : 0)
  .join();

Fork and Join

Not Tested :C