Binding and Types

`byte` <: `short` <: `int` <: `long` <: `float` <: `double`
`char` <: `int`

Right to left is narrowing (requires explicit type casting).
Left to right is widening (always ok).

Narrowing type conversion has problems with generics due to type erasure

- Type inference is done only during compilation time
  - picks the most specific type if there is a lower bound
  - if there is only a upper bound, pick the upper bound (because can't choose a lower bound arbitrarily)
- Type erasure is done only during compilation time
- Type checking is done partially during runtime because value is not always available at compile time
- Dynamic binding happens during both runtime and compile time
- Type casting happens during compile time but is checked during runtime
- Accessibility checks done during both runtime and compile time

```
Integer i = 0;
Object o = (Number) i;

// i is of run time type Integer
// i is of compile time type Integer
// o is of run time type Integer (because pointing to the object I)
// o is of compile time type Object
```

Stack and Heap

JVM will partition memory into several regions.

method area for storing the code for the methods
metaspace for storing meta information about classes, like static fields
heap for storing dynamically allocated objects
stack for storing local variables (including primitive types and object references) and call frames

Heap and stack are common to all execution environments.

Instance fields are stored on heap.
Local variables and parameters goes on stack frame.
Lambda (and anonymous classes) are local variables on the stack that point to the class on the heap.

OOP Principles

- Encapsulation
  - Groups *primitive types* together
- Abstraction
  - Information/data hiding (client & implementer)
- Inheritance -> not used to avoid code duplication
  - Classes can only inherit from 1 parent class
  - Classes can implement multiple interfaces
  - `ArrayList extends List` => `ArrayList <: List`
  - `List<Integer> list = new ArrayList<>();`
    - Compile-time type of `list` is `List`.
    - Run-time type of `list` is `ArrayList`.
- Polymorphism

- Polymorphism
    - One thing many forms.
- Composition
    - composing complex types

Wrapper Classes

Slower than primitives.
Immutable.

`Double pi = 3.14;` (auto boxing)

```
int i = 10000;
Integer x = i;
Integer y = i;

x == y; // may be true or false (depends on if x and y are cached)
```

Final Keyword

Variable → cannot be re-assigned (can be initialised only once)
Method → cannot be overridden by child classes
Class → cannot be extended (inherited)

Dynamic Binding

Only for instance methods

Most *specific* method will be called. M is more specific than N if arguments to M can be passed to N without compilation errors.
Compile-time return type is used to make sure that the required method exists. (decide method descriptor with compile type)
Run-time type of target (*a*.method()) and compile-time type of arguments are used to determine method. Searched up the class hierarchy. (step 2)

`instanceof` matches run-time type.

```
interface I1 {
  void bar(X x);
  void bar(I1 i);
}
interface I2 {
  void bar(Y y);
}
class X implements I1 {}
class Y extends X implements I1, I2 {}

bar(X x) in X prints 1
bar(I1 i) in X prints 2
bar(X x) in Y prints 3
bar(I1 i) in Y prints 4
bar(Y y) in Y prints 5

X x = new X();
X xy = new Y();
Y y = new Y();
I1 i = new Y();

i.bar(y);  // 3
x.bar(y);  // 1
y.bar(xy); // 3
x.bar(x);  // 1
```

```
i is of compile time type I1 thus only has access to methods in I1 but has run time type of Y, thus uses those methods
in Y. Since y's compile time type is Y, and Y is subtype of X and I1, but priority given to class.
```

Liskov Substitution Principle (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 $S <: T$".
Any code written for T would still work if T is substituted for S.

Abstract Classes

Cannot be instantiated.
Should contain at least 1 abstract method.

Interface

An interface models what an entity can do, with a name usually ending with the -able suffix.

All methods are `public abstract` by default.
Interfaces can have static fields (as constants).

An interface can extend from one or more interfaces, but *not* from a class.

Method Overloading

Overload by defining another method of the *same* name but *different* method signature.
Method descriptor = return type + method signature.
Overloading requires changing the order, number and/or type of arguments. (not name)

Method signature

- method name
- number of arguments
- type of arguments
- order of arguments

Cannot override private/static methods

Exceptions

`try` `catch` `finally` keywords.
`throws` `throw`
`throw new SomeException("Optional Message");`

Unchecked Exceptions

iff *subclass* of `RuntimeException`
Something wrong with the program and might cause run-time errors.

`@SuppressWarnings("unchecked")` (no semicolon)

Do not require to be handled.

Checked Exceptions

Exceptions that a programmer has no control over. For example, `FileNotFoundException`.
Must be handled otherwise will not compile (must catch or throws). i.e. programmer has to foresee these exceptions.

Errors

`OutOfMemoryError` or `StackOverflowError`
Don't need to handle.

A method overriding a method that throws an exception, does not need to throw an exception (or handle it).

Variance

- *covariant* if $S <: T \rightarrow C(S) <: C(T)$
- *contravariant* if $S <: T \rightarrow C(T) <: C(S)$
- *invariant* if neither covariant nor contravariant

Arrays are covariant.
Generics are invariant.

`a.compareTo(b)` ⇒ `a - b`

Type Erasure

Generics (complex type) enable type safety by allowing the developer to not resort to using `Object`.

Generics are erased during compilation.

Unbounded types are replace with `Object`.
Bounded types are replaced with its bound.

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

class A<T, S extends GetAreable> {
  T t;
  S s;
}

// transformed to
Integer i = (Integer) new Pair("hello", 4).getSecond();

class A {
  Object t;
  GetAreable s;
}
```

Arrays and generics do not mix (causes heap pollution).

Arrays are *reifiable* type -- a type where full type information is available during run-time BUT generics are not.

Raw Types

DON'T USE → use `<?>`
Generic type used without type arguments.
Compiler cannot do any type checking.

Wildcards

PECS
Producer Extends, Consumer Super.
If both producer and consumer, just T

Upper Bounded Wildcard

Covariant

`? extends T`
Matches classes that are child classes of T.
`A<S> <: A<? extends S>`

Lower Bounded Wildcard

Contravariant

`? super T`
Matches classes that are super classes of T.
`A<S> <: A<? super S>`

Unbounded Wildcard

`Array<?>` will accept *any* type of Array<>. "Object" of the generic world.

For any type *S*:
`A<S> <: A<?>`
`A<? super S> <: A<?>`
`A<? extends S> <: A<?>`
`A<Integer> <: A<? extends Number>`
`A<Integer> <: A<? extends Object>`

`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.

`a instanceof A<?>`

```
public static <T extends GetAreable> T findLargest(Array<? extends T> array);
```

Immutability

Making class *immutable* by making fields *final*.

- Ease of understanding
- Enabling safe sharing of objects
  - Example: ORIGIN Point, EMPTY_BOX Box
  - No problem of aliasing (reference types may share the same reference values)
  - use factory method instead of exposing constructor
- Enabling safe sharing of internals
- Enabling safe concurrent execution
- can be used to represent same object at different stages
- can "revert" objects to a previous state (Undo-able)

```
// @SafeVarargs ise used here because compiler would throw unchecked warning as generics and arrays do not mix well
@SafeVarargs
public static <T> ImmutableArray<T> of (T... items) {
  return new ImmutableArray(items, 0, items.length - 1);
}
```

Nested Classes

- Act like fields.
- Used to group logically relevant classes together.
- Tightly coupled with the container class and can declare as private if has no use outside of container class
- Can access private fields of container class (thus should only be nested class if same encapsulation as container class)

Nested class: Class inside a class
Local class: Class inside a method

Nested Non-Static Class

aka Inner Class

- Cannot have static fields because they belong to an instance
- UNLESS these static fields are constants known at compile time like int literals, string literals

Nested Static Class

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

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

  // the definition of static nested classes are stored in the metaspace
  static class C {
    void bar() {
      x = 1; // accessing x (instance field) from A is not OK since C is static
      y = 1; // accessing y (static field) is OK
    }
  }
}
```

Local and Anonymous Classes and Lambda

Classes *inside a method*. Scoped only inside the method.
Local classes cannot be declared public, protected, private.

```
// Comparator is a common use case for local classes
void sortNames(List<String> names) {

  class NameComparator implements Comparator<String> {
    public int compare(String s1, String s2) {
      return s1.length() - s2.length();
    }
  }
  names.sort(new NameComparator());
}

// anonymous class
names.sort(new Comparator<String>() {
  @Override
  public int compare(String s1, String s2) {
    return s1.length() - s2.length();
 }
});

// lambda
(x, y) -> x + y;
i -> i + 1;
Function<Integer, Integer> f = x -> {
  System.out.println("Test");
  return x;
};
f.apply();
```

Variable Capture

Local variables (including primitives) referenced from an inner class must be *final* or *effectively final* (only assigned once even though not declared final).
Note: this constraint *does not* apply to static variables!

Usually, when a method returns, all local variables of the method are removed from the stack. BUT an instance of a local class can still exist. Thus, local classes make a copy of (captures) local variables inside itself (even if unused as a failsafe). Hence, why these variables cannot be changed.

Lambda Closure
Value of `origin` is captured by the lambda expression `dist`, and hence has to be final or effectively final.

```
Point origin = new Point(0,0);
Transformer<Point, Double> dist = p -> origin.distanceTo(p);
Transformer<Point, Double> dist = origin::distanceTo;
// above 2 are equivalent
```

Pure Functions

- Deterministic
- No side effects → once stack frame is done, nothing except final result should be left
  - **print** to screen
  - **write** to file
  - throw **exception**
    - be careful of division by zero
    - FileNotFoundException
  - **change** other variables
  - **modifying** fields of reference type arguments passed to it
- Referential transparency (no internal state) → always same output
  - violated by using instance fields that are not final
- Memoization only makes sense when function is pure and deterministic

`@FunctionalInterface` annotation
An interface in Java with **only one abstract method** is called a functional interface.

Lambda expressions allow you (as the client) to "customise" some implementation into the Implementor's methods (including modifying Implementor's internal states).

Streams

`import java.util.stream.*;`

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

Streams can only be operated once **once**. Attempting to iterate through a stream more than once throws a `IllegalStateException`.
Java also has `IntStream`, `LongStream`, ... that contains primitive values instead of wrapper classes

`Stream.iterate(0, x -> x + 1)` generates an infinite list of sorted non-negative numbers.
`range(x, y)` x is inclusive, y is exclusive
`rangeClosed(x, y)` both x and y are inclusive
`List::stream()` constructs a stream based on the values in a `List`.

- Terminal Operations
  - `forEach`, `reduce`, `count()` (returns `long`) etc
- Intermediate Stream Operations → lazy
  - `map`, `filter`, `flatMap` etc
- Stateful and Bounded Operations (need to maintain some internal state)
  - `sorted` (can pass in Comparator), `distinct`
- Truncation of Infinite Streams
  - `limit`, `takeWhile` (might still be infinite if predicate always true)
- Peeking with Consumer
  - `peek`
- Reducing
  - `reduce(init, (acc, x) -> ...)`
  - `reduce(T identitiy, BinaryOperator<T> accumulator)`
  - `reduce(U identity, BiFunction<U,? super T,U> accumulator, BinaryOperator<U> combiner)`
- Element Matching
  - `noneMatch`, `allMatch`, `anyMatch` (terminal)

Functor and Monad

Need to explain *why* the laws hold.
For flatMap, be careful of side information stored in both original Monad and Monad returned from the mapper (both should be retained, in the correct order).

Monad

These are examples of monads: `Maybe<T>`, `Lazy<T>`, `Loggable<T>`, `Stream<T>`, `CompletableFuture`
Must have `of` and `flatMap` methods.

Identity Law

Factory method **should not** do anything extra to the value and side information.

Left identity law says:
`Monad.of(x).flatMap(y -> f(y))` must be equivalent to `f(x)`

Right identity law says:
`monad.flatMap(x -> Monad.of(x))` must be equivalent to `monad`

Associative Law

`monad.flatMap(x -> f(x)).flatMap(y -> g(y))` must be the same as `monad.flatMap(x -> f(x).flatMap(y -> g(y)))`
(NOTE: The location of the closing brackets)

Functor

Functors is similar to monad but only ensures that lambdas can be applied sequentially to the value, without care about side information.
Must have `map` method.

Preserving Identity

`functor.map(x -> x)` must be the same as `functor`

Preserving Composition

`functor.map(x -> f(x)).map(x -> g(x))` must be the same as `functor.map(x -> g(f(x)))`

Parallel and Concurrent Programming

Divide the computation into subtasks called *threads*.

Multi-thread programs are useful

1. allow programmers to separate unrelated tasks into threads, and write each thread separately
2. improves utilisation of the processor, e.g. I/O and UI is split up

Parallel **NOT** always faster than non-parallel due to overhead with creating threads.

Concurrency

Concurrency gives the *illusion* of subtasks running at the same time.

Parallelism

Parallel computing refers to when multiple subtasks are actually running at the same time.

Parallel Streams

`parallel` is a lazy intermediate operation.
`sequential` will force the stream to be processed sequentially instead.
Note that if multiple `parallel` and `sequential` are used on the last stream, the last one used will override.
`parallelStream()` instead of `stream()` also exists to be used on a collection.

What can be parallelised?

1. Stream operations must not **interfere** with the stream data
2. Most of the time must be stateless (like stdin)
3. Side-effects should be kept to the minimum

Interference will cause `ConcurrentModificationException` to be thrown.

`CopyOnWriteArrayList` and several other thread-safe data structures in `java.util.concurrent`.

Associativity while Reducing

`reduce` operation is parallelisable but have to abide by some rules:
`reduce(identity, accumulator)`
`reduce(identity, accumulator, combiner)`

Accumulator: First parameter is the accumulated value, second parameter is the current value of the stream.
Combiner: Used to combine the results of all the parallel sub-streams.

1. `combiner.apply(identity, i)` must be the same as `i`
2. `combiner` and `accumulator` must be associative

- Parameters for the combiner and accumulator must be able to be swapped.
- Order of reduction should not matter

3. `combiner` and `accumulator` must be compatible

- `combiner.apply(u, accumulator.apply(identity, t))` must be equal to `accumulator.apply(u, t)`

Multiplication is a valid operation that abides by these 3 rules.

Ordered vs Unordered Source

Streams created from `iterate` and ordered collections (List, arrays) are **ordered**
Streams created from `generate` and unordered collections (Set) are **unordered**

Stable Operations
Stable operations preserve the original ordering of the elements.

`distinct`, `sorted`

Parallel versions of `findFirst`, `limit`, `skip` can be expensive on an ordered stream, since it needs to coordinate between streams to maintain order.

Synchronous Programming

Execution of program is stalled until a method returns. The method is *blocking* the execution of the program.

Threads

`java.util.Thread` is a single flow of execution in the program

`new Thread(...)` takes in a `Runnable`, a functional interface with a method `void run()` `Thread::start()` then starts the execution of the program in *another* thread.
NOTE: Creating new threads causes overhead. Thus, should try and re-use threads as much as possible.

CompletableFuture

- `CF.completedFuture`
- Returned CF completes when given lambda expression finishes
  - `runAsync(Runnable runnable)`
  - `supplyAsync(Supplier<T> runnable)`
  - NOTE: Java program *may* terminate before the Runnable is completed!
- `CF.allOf(...)`, `CF.anyOf(...)` takes in variable number of CF instances
- `runAfterBoth(CF other, Runnable r)`, `runAfterEither(CF other, Runnable r)`
- Chaining CF
  - `thenApply` (map), `thenCompose` (flatMap), `thenCombine` (combine)
  - `thenRun(Runnable r)`, `thenAccept(Consumer c)`
- Getting result (blocks program execution)
  - `get()` throws `InterrupedException` (thread has been interrupted) and `ExecutionException` (errors/exceptions during execution)
  - `join()`, similar to `get()` but does not throw checked exceptions
- Handling exceptions
  - `handle(BiFunction<Value, Exception> f)` value will be null if got exception and vice versa
  - `exceptionally` only handles exceptions (no result)
  - `whenComplete` used for logging information

Thread Pool

Thread pool lets us reuse threads and therefore reduces the overhead of creating new threads.

`ForkJoinPool` is an implementation of a thread pool. *Forks* the problem into smaller problems and then *joins* them.
Fork and join **must** be used in tandem.

Recursive Task

Abstract class `RecursiveTask<T>` supports `fork()` and `join()` and has an abstract method `compute()` which we will use to specify the computation.

`RecursiveAction` is a result-less version of Recursive Task.

```
class Summer extends RecursiveTask<Integer> {
    private static final int FORK_THRESHOLD = 2;
    private int low;
    private int high;
    private int[] array;
```

```java
    public Summer(int low, int high, int[] array) {
      this.low = low;
      this.high = high;
      this.array = array;
    }

    @Override
    protected Integer compute() {
      // stop splitting into subtask if array is already small.
      if (high - low < FORK_THRESHOLD) {
        int sum = 0;
        for (int i = low; i < high; i++) {
          sum += array[i];
        }
        return sum;
      }

      int middle = (low + high) / 2;
      Summer left = new Summer(low, middle, array);
      Summer right = new Summer(middle, high, array);
      // !!
      // many ways to do next 2 lines
      // but using 1.fork(), 2.compute(), 1.join() is the best way
      left.fork();
      return right.compute() + left.join();
    }
  }

    Summer task = new Summer(0, array.length, array);
    int sum = task.compute();
```

1.fork(), 2.compute(), 1.join() is the most optimal
compute() is likely faster than fork() then join() because it reduces the overhead of having to interact with the ForkJoinPool

ForkJoinPool

1. Each thread has a deque of tasks
2. When a thread is idle, it checks its queue of tasks.
    1. If queue is not empty, it picks up a task at the *head* of the queue to execute (invokes its `compute()` method)
3. Otherwise, it picks up a task from the *tail* of the queue of another thread to run. This is known as *work stealing*. Picking up from the tail to minimise conflicts.
4. When `fork()` is called, the caller adds itself to the *head* of the queue of the executing thread. This is similar to recursion stack.
5. When `join()` is called, several different cases
6. If the subtask to be joined hasn't been executed, its `compute()` method is called and the subtask is executed.
7. If the subtask to be joined has been completed (other thread stole it), then the result is read and done.
8. If the subtask to be joined has been stolen and is still being executed, current thread works on another task (in local queue or steals another task)

The threads are always looking for something to do and cooperate to maximise work done! Minimise the time each thread is spent idling.

Misc

`new A().foo()` is valid even when `foo` is a static method

Metaspace:

- static fields
- other meta information about classes

Stack frame:

- parameters
- local variable (including reference types → points to heap)
- lambda & anon. class (local var that point to class on heap)

Heap:

- reference types (instances)

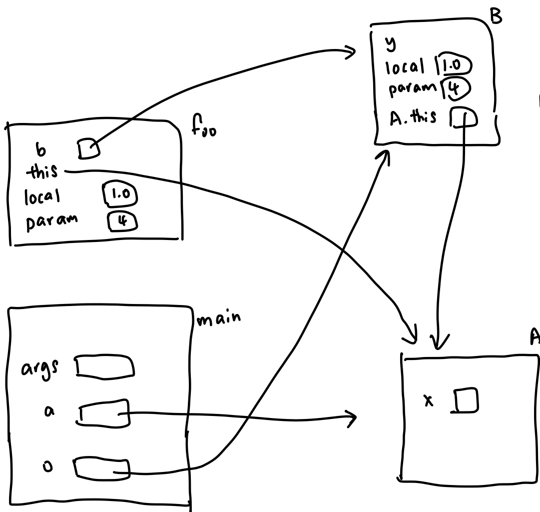
Variable capture:

- only captured if used by local class
- must be final/effectively final
- can capture reference types through pointer
  - cannot change pointer
  - but can change value in reference type

```
class A {
    int x;
    Object foo(int param) {
        double local = 1.0;
        class B {
            int y;

            @Override
            String toString() {
                return "" + y + local + param
                        + A.this.x;
            }
        }

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

    main(...) {
        A a = new A();
        Object o = a.foo(4);
        o.toString();
    }
```

can access stuff from foo & A (→ local + param)

qualified this (→ A.this.x)



if in
1. foo: capture variable
2. A: capture "A.this" (always captured ∵ no cost)

qualified "this" captured ⟺ it is a non-static nested class