CS2030S cheatsheet (1)

jovyntls

TYPES

- **S is a subtype of T,** `S <: T` if a piece of code written for variables of type `T` can also be safely used on variables of type `S`.
  - *widening* conversion ⇒ a type `S` can be put into a variable of type `T` if `S <: T`.
  - *narrowing* conversion ⇒ requires typecasting
- **reflexive** - `T <: T`
- **transitive** - if `S <: T` and `T <: U`, then `S <: U`
- `S instanceof T` returns true if `S <: T`

primitive types

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

Liskov substitution principle

- if `S <: T`, then
  - any property of `T` should also be a property of `S`. (includes fields, methods)
  - an object of type `T` can be replaced by an object of type `S` without changing some desirable property of the program.
- **VIOLATION:** subclass changes the behaviour of the superclass - <specified> property no longer holds.
  - places in the program where the superclass is used cannot be replaced by the subclass

💡 `this` - reference variable that refers to the instance

RUN-TIME vs COMPILE-TIME TYPES

```
Circle c = new ColouredCircle(p, 0, red);
// ColouredCircle <: Circle
```

- compile-time type: `Circle` ; run-time type: `ColouredCircle`

OOP PRINCIPLES

encapsulation

- composite data types
- abstraction barrier - hide information & implementation
- `private` attributes, `public` methods

inheritance

- "is-a" relationship → `extends` (subtyping)
- vs "has-a" relationship → use composition

tell, don't ask

- don't make assumptions the implementation
- a class should be agnostic of another class

polymorphism

- **dynamic binding** → method invoked is determined at runtime

method overriding (dynamic polymorphism)

- same method signature
  - method name, type/number/order of arguments
  - method descriptor = method signature + return type (`@Override`)

method overloading (static polymorphism)

- same method name, diff parameter types/number of parameters

ABSTRACT CLASSES

```
abstract class Shape { ... }  // cannot have fields
```

- cannot be instantiated
- a concrete class cannot have abstract methods
  - as long as one method is abstract, the whole class is abstract
- an abstract class can have concrete and/or abstract methods

INTERFACE ("can-do")

```
interface getAreable {
  // methods are public and abstract by default
  double getArea();
}
```

- concrete classes implementing the interface have to implement the body of ALL the methods
- if class `C` implements interface `I`, then `C <: I`.
- a class can extend multiple interfaces

```
class C implements A, B { ... }
```

- an interface can extend multiple interfaces

```
interface I extends A, B { ... }
```

  - an interface cannot implement other interfaces (abstract!!)
- may have default implementation - `default`

CS2030S cheatsheet (2)

jovyntls

WRAPPER CLASS

- immutable, may be less efficient

```
Integer i = new Integer(2);
int j = i.intValue();
```

(un)boxing

```
int i = 1;      // i is an int
Integer j = i;  // j is an Integer
int k = j;      // k is an int
```

MODIFIERS

- `private` → only within the class
- `default` → only within the package
- `protected` → only within the package or outside the package through the child class
- `public` → everywhere
- `final` variable → can only be assigned once
- `final` class → cannot be inherited from
- `final` method → cannot be overridden

CASTING

```
// Circle <: Shape <: GetAreable
GetAreable findLargest(GetAreable[] array) { ... }
GetAreable ga = findLargest(circles);       // ok
Circle c1 = findLargest(circles);           // error
Circle c2 = (Circle) findLargest(circles);  // ok
```

- only cast when you can prove that it is safe

variance

Let $C(T)$ be a complex type based on type T. The complex type C is:

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

(Java array is covariant - $S <: T \implies S[] <: T[]$)

```
public protected private abstract default static final
transient volatile synchronized native strictfp
```

EXCEPTIONS

```
try {
  new Circle(new Point(1, 1), 0);
  // everything afterwards is skipped
  System.out.println("This will never reach");
} catch (IllegalException e) {
  // runs if there is an exception
} finally {
  // always runs
}
```

- exception will be passed up the call stack until it is caught
- after exception is caught: everything else proceeds normally

```
class MyException extends IllegalArgumentException {
  MyException(String msg) { super(msg); }
}
```

`throw` exceptions

```
import java.lang.IllegalArgumentException
public Circle(Point c, double r) throws IllegalArgumentException {
    if (r < 0) {
      throw new IllegalArgumentException("radius cannot be negative."
    }
    // anything from here will not run if r<0
}
```

- `throw` causes method to immediately return

💡 overriding a method that throws exception E_0: must thow exception E_1 such that $E_1 <: E_0$ (LSP)

CS2030S cheatsheet (3)

jovyntls

GENERICS

- allow classes/methods (that use reference types) to be defined without resorting to using the Object type.
  - ensures **type safety** → binds a generic type to a specific type at compile time
  - ✓ errors will be at compile time instead of runtime
- generics are **invariant** in Java

generic class

```
class Pair<S extends Comparable<S>, T>
      implements Comparable<Pair<S, T>> {...}
class DictEntry<T> extends Pair<String,T> {...}
```

generic method

```
public static <T> boolean contains(T[] arr, T obj) {...}
// to call a generic method:
A.<String>contains(strArray, "hello");
```

- type parameter `<T>` is declared *before* the return type
- bounded type parameter: `public <T extends Comparable<T>> T foo(T t) { ... }`

note

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

TYPE ERASURE

type erasure

- at compile time, type parameters are replaced by `Object` or the bounds (e.g. `T extends Shape` is replaced by `Shape`)

```
Integer i = new Pair<String,Integer>("x", 4).foo();  // before
Integer i = (Integer) new Pair("x", 4).foo();        // after
```

suppress warnings

- `@SuppressWarnings` can only apply to declaration

```
@SuppressWarnings("unchecked")
T[] a = (T[]) new Object[size];
this.array = a;
```

WILDCARDS

upper-bounded: `? extends`

- *covariant* - if `S <: T`, then `A<? extends S>` <: `A<? extends T>`

lower-bounded: `? super`

- *contravariant* - if `S <: T`, then `A<? super T>` <: `A<? super S>`

unbounded: `?`

- `Array<?>` is the supertype of all generic `Array<T>`

PECS PRINCIPLE

- PE → if you need to produce T values, use `List<? extends T>`
- CS → if you need to consume T values, use `List<? super T>`
- if both producer & consumer → use wildcard `<?>`

RAW TYPES

- a generic type used without type arguments
- only acceptable as an operand of `instanceof`

TYPE INFERENCE

- ensures **type safety** → compiler can ensure that `List<myObj>` holds objects of type `myObj` at compile time instead of runtime
- `<? super Integer>` ⇒ inferred as `Object`
- `<? extends Integer>` ⇒ inferred as `Integer`

diamond operator: `<>`

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

- only for instantiating a generic type - not as a type
- generic methods: type inference is automatic
  - `A.contains()` not `A.<>contains()`

constraints for type inference

- target typing → the type of the expression (e.g. `Shape`)
- type parameter bounds → `<T extends GetAreable>`
- parameter bounds → `Array<Circle>` <: `Array<? extends T>`, so `T :> Circle`

```
public static <T extends GetAreable> T findLargest(Array<? extends T> arr
Shape o = A.findLargest(new Array<Circle>(0));
```

CS2030S cheatsheet (4)

jovyntls

IMMUTABILITY

- immutable class → an instance cannot have any *visible changes* outside its abstraction barrier
- **advantage**:
  - save space - share all references until instance needs to be modified (which will create a new copy)
  - enable safe sharing of (dependency on) internals
  - enable safe concurrent execution
- to update something `final`: explicitly reassign

```
final class Circle {
  final private Point c;
  final private double r;
   :
  public Circle moveTo(double x, double y) {
    return new Circle(c.moveTo(x, y), r);
  }
}
```

  - immutable class declared `final` - prevent overriding & inheritance

VARARGS

- pass a variable number of arguments of the same type
  - will be passed to the method as an array of items
  - `public void of(T... items) {}` ⇒ items will be `T[]`
  - `@SafeVarargs` annotation if `T` is a generic type
    - final or static methods/constructors

NESTED CLASSES

- can access fields and mehthods of container class (incl. `private`)
- static nested class → associated with the containing class
  - can ONLY access static fields/methods of containing class
- inner class (non-static nested class) → associated with an instance
  - can access ALL fields/methods of containing class

qualified `this`

- differentiate between `this` of inner class and container class

```
class A {
  private int x;

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

ANONYMOUS CLASS

- format: `new Constructor(arguments) { body }`
  - or `new (className implements someInterface)(arguments) { body }`
- cannot implement more than one interface
- cannot extend a class and implement an interface at the same time
- same rules as local classes for variable access

LOCAL CLASS

- class defined within a method
- can access: class and instance variables from the enclosing class (use qualified `this`) + local variables of enclosing method
  - can only access variables declared `final` or *effectively* final
  - effectively final → variable does not change after initialisation

> 💡 WILL NOT COMPILE if the variables are NOT effectively final!!

variable capture

> 💡 when a method returns, all local variables of the method are removed from the stack

- instance of a local class makes a copy of local variables inside itself

CS2030S cheatsheet (5)

jovyntls

FUNCTIONS

- function → mapping from a domain to a codomain ($f : X \to Y$)
- referential transparency → if $f(x) = y$, any y can be substituted with $f(x)$

pure function

✓ no side effects (× cannot print/write to file/change value of arguments/throw exceptions/change other variables)

✓ every input mapped to an output in the codomain

(`null` is not within the codomain)

✓ deterministic; not dependent on external variables

- must return a value (cannot be `void`)

> 💡 immutable class → methods are pure functions

functions as first-class citizens

```
Transformer<Integer, Integer> square = new Transformer<>() {
  @Override
  public Integer transform(Integer x) {
    return x * x;
  }
};
```

- `@FunctionalInterface` annotation (only one abstract method)

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

LAMBDA FUNCTIONS

```
Transformer<Integer, Integer> incr = x -> x + 1;
Comparator<String> cmp = (s1, s2) -> s1.length() - s2.length();
```

METHOD REFERENCE

- static method in a class (`ClassName:staticMethodName`)
- instance method of a class/interface (`instanceName::methodName`)
- instance method of an object of a particular type (`Type::methodName`)
- constructor of a class (`ClassName::new`)

```
Transformer<T, U> foo = A::foo;
// (x, y) -> x.foo(y)      A is a type, foo is an instance of that type
// (x, y) -> A.foo(x, y)   A is an instance, foo is an instance method
```

- **at compile time:** Java searches for the matching method - performs type inference to find the method that matches the method reference.

> 💡 multiple matches/ambiguous match ⇒ compilation error

CURRIED FUNCTIONS

- translate a general *n*-ary functions to *n* unary functions
- stores the data from the environment where it is defined
  - closure → a construct that stores a function together with the enclosing environment

```
Transformer<Integer, Transformer<Integer, Integer>> add = x -> y -> (x + y);

=> add.transform(1) // gives a Lambda
=> add.transform(1).transform(2)  // returns 3
=> increment.transform(3)  // returns 4
```

LAZY EVALUATION

delayed computation with lambda functions

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

@FunctionalInterface
interface Task { void run(); }
```

```
i = 4;
Task print = () -> System.out.println(i);
Producer<String> toStr = () -> Integer.toString(i);
```

memoization

- use `Lazy<T>`

```
class Lazy<T> {
  T value;
  boolean evaluated;
  Producer<T> producer;

  public Lazy(Producer<T> producer) {
    evaluated = false;
    value = null;
    this.producer = producer;
  }

  public T get() {
    if (!evaluated) {
      value = producer.produce();
      evaluated = true;
    }
    return value;
  }
}
```

CS2030S cheatsheet (6)

jovyntls

MONAD

- contains a value + side information
  - `of` method to initialize the value and side information
  - `flatMap` method to update the value & side information

monad laws

1. left identity law
   - `Monad.of(x).flatMap(y -> f(y))` is equivalent to `f(x)`
2. right identity law
   - `monad.flatMap(y -> Monad.of(y))` is equivalent to `monad`
3. associative law
   - `monad.flatMap(x -> f(x)).flatMap(x -> g(x))` is equivalent to `monad.flatMap(x -> f(x).flatMap(x -> g(x)))`
   - aka same result regardless of how it's composed

FUNCTOR

- has two methods `of` and `map`
  - does not carry side information

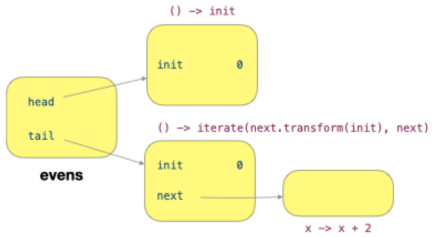
functor laws

1. preserving identity
   - `functor.map(x -> x)` is the same as `functor`
2. preserving composition
   - `functor.map(x -> f(x)).map(x -> g(x))`

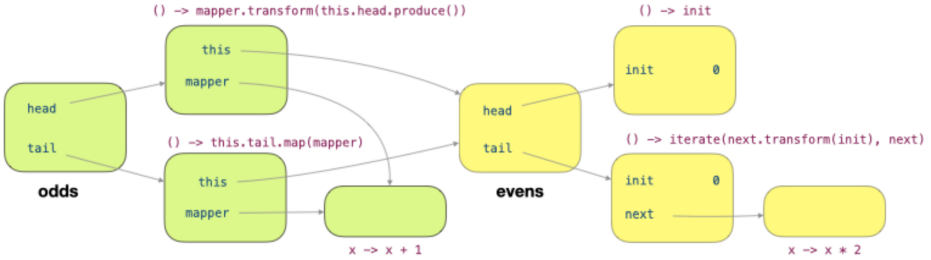
     is the same as `functor.map(x -> g(f(x))`

TYPES

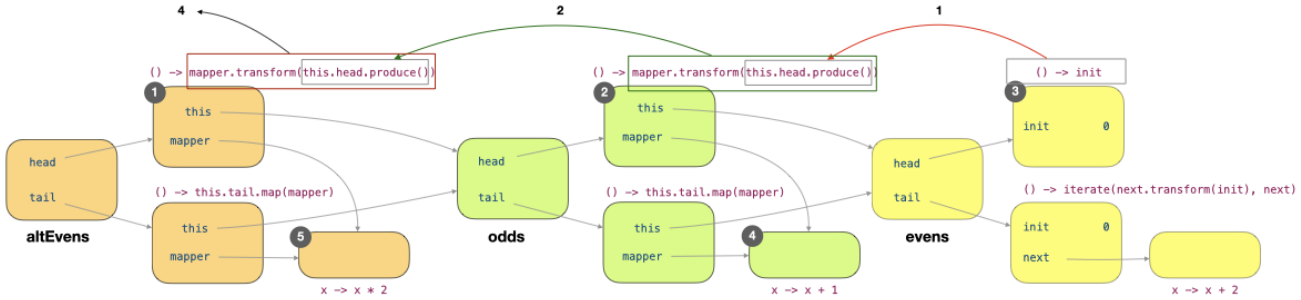
```
InfiniteList<Integer> evens = InfiniteList.iterate(0, x -> x + 2);
```




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




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



CS2030S cheatsheet (7)

jovyntls

STREAMS

- consumed only once ⇒ `IllegalStateException` if consumed again

PARALLEL STREAMS

- add `.parallel()` before the terminator or use `parallelStream()`
- CANNOT be:
  - stateful → result depends on any state that may change during execution of the stream
  - interfering with stream data → one of the stream operations modifies the source of the stream during execution of the terminal operation (throws `ConcurrentModificationException`)
  - side effects → use `collect()` for ArrayList to avoid
- associativity → an operation is parallelisable by reducing each substream and combining them with a `combiner` if:
  - `combiner.apply(identity, i)` is equal to `i`
  - `combiner` and `accumulator` are associative (order of application does not matter)
  - `combiner` and `accumulator` are compatible (types)

ordered vs unordered source

- ordered → created from `iterate` / `of` / ordered collections (`List`)
- unordered → created from `generate` / unordered collections (`Set`)
- `distinct` and `sorted` preserve order (aka stable)
  - only for FINITE streams!
- use `.unordered()` to make parallel operations more efficient
  - no need to coordinate between streams to maintain order

THREADS (`java.lang.Thread`)

- thread → a single flow of execution
- `new Thread` constructor: takes in a Runnable
- `Runnable` → functional interface with `run()` method (returns `void`)
- `.start()` → thread begins execution (returns immediately)
- `.isAlive()` → returns boolean representing if thread is alive
- `Thread.currentThread()` → returns reference of current running Thread (for name: `Thread.currentThread().getName()`)
- `Thread.sleep(ms)` → pauses execution of current thread

```
Stream.of(1, 2, 3, 4)
  .parallel()
  .reduce(0, (x, y) -> {
      System.out.println(Thread.currentThread().getName());
      return x + y;
  });
```

- prints:

```
main
ForkJoinPool.commonPool-worker-5
ForkJoinPool.commonPool-worker-5
ForkJoinPool.commonPool-worker-9
ForkJoinPool.commonPool-worker-3
ForkJoinPool.commonPool-worker-3
ForkJoinPool.commonPool-worker-3
```

PARALLEL PROGRAMMING

- concurrency → divides computation into subtasks called threads
  - separate unrelated tasks into threads; write each thread separately
  - improves utilization of the processor (can switch between threads)
- parallelism → multiple subtasks are truly running at the same time
  - parallelism $\subseteq$ concurrency

THREADS (cont.)

```
System.out.println(Thread.currentThread().getName());  // main

Thread myThread = new Thread(() -> {
  System.out.print(Thread.currentThread().getName());
  for (int i = 1; i < 100; i += 1) {
    System.out.print("_");
  }
})

myThread.start();

new Thread(() -> {
  System.out.print(Thread.currentThread().getName());
  for (int i = 2; i < 100; i += 1) {
    System.out.print("*");
  }
}).start();

while (myThread.isAlive()) {
  try {
    Thread.sleep(1000);
    System.out.print(".");
  } catch (InterruptedException e) {
    System.out.print("interrupted");
  }
}
```

CS2030S cheatsheet (8)

jovyntls

`CompletableFuture` MONAD

- `java.util.concurrent.CompletableFuture`
- to instantiate:
  - `.completedFuture(thing)`
  - `.runAsync(Runnable)` → returns a `CompletableFuture<Void>` which completes when the given lambda (Runnable) finishes
  - `.supplyAsync(Supplier<T>)` → returns a `CompletableFuture<T>` which completes when the given lambda (Supplier) finishes
- chaining in the same thread
  - `.thenApply(res -> f(res))` → map
  - `.thenCompose(res -> CF)` → flatMap
  - `.thenCombine(CF, (thisCF, givenCF) -> ...)` → combine
  - with `Async` → given lambda can run in a different thread
- getting result
  - `.get()` → returns result (synchronous - blocks until the CompletableFuture completes)
    - throws `InterruptedException` & `ExecutionException` - should catch and handle!
  - `.join()` → behaves like `get()` but no checked exception thrown
    - may throw `CompletionException` - whoever calls join will handle this
- handling exceptions (before join)
  - `.handle((result, exception) -> (exception == null) ? result : somethignElse)`

THREAD POOL

- comprises
  1. a collection of threads, each waiting for a task to execute
  2. a collection of tasks to be executed (usually queue)
- reuse existing threads → avoid overhead cost of creating new threads
- assume queue is thread safe
- once freed, a thread runs the next task (dequeued) from the queue
- `fork()` → caller (task) adds itself to the thread pool
- `join()` → blocks computation until completed

`ForkJoinPool`

- each thread has a queue of tasks
- when a thread it idle, it will `compute()` the task at the head of its queue
  - work stealing → if queue is empty, it will `compute()` a task at the tail of the queue of another thread
- `fork()` → caller (task) adds itself to the head of the queue of the executing thread
  - most recently forked task gets executed next
- `join()` → if the subtask to be joined:
  - **has not been executed** → `compute()` the subtask
  - **has been completed** (work stealing) → read the result and return
  - **has been stolen/is being executed by another thread** (work stealing) → current thread finds another task to work on (from local queue OR steal another task)

💡 should `join()` the most recently `fork()`-ed task first

functional interfaces

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

abstractions

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

- `Optional` violates left identity law & function composition preservation