CS2030 Reference

# Java language

## Overloading

* Exceptions and return types are not part of function signature

## Overriding

* Cannot throw checked exceptions that are not stated in (or subclassed from) overridden method
* Cannot reduce access (protected/private method cannot override public method)
* Can increase access (public method can override protected method)
* private, static, or final methods cannot be overridden

## Enums

* Enums are actually Java classes with syntactic sugar
* Enums are final; they cannot be inherited from
* public final class EventType extends Enum<EventType>
where Enum<E extends Enum<E>>

## Numeric Types

* byte 🡪 short 🡪 int 🡪 long 🡪 float 🡪 double
* char 🡪 int 🡪 long 🡪 float 🡪 double
* Widening conversions may be implicit
* Narrowing conversions need explicit casts
* Boxing and unboxing may be implicit
* Boxed type can be converted (implicitly or explicitly, as per primitive type rules) to any primitive type
* Unboxed type cannot be converted to other boxed types, even explicitly
* Boxed type cannot be converted another boxed type, even explicitly
* All boxed numeric types are final
* Any number of “\_” can appear between digits
* Small Integers are interned by the Integer.valueOf(int) factory method and will have the same object reference

## Booleans

* boolean
* Does not convert to/from any other primitive type, even explicitly

## Strings

* Strings are immutable reference types which default to null
* String s1 = "test"; String s2 = "test"; will be the same object reference because of automatic string internment of literal strings and string-valued constant expressions

## Range-based For Loops

* for(x : collection) or for(T x : collection)
* Works with any collection that implements Iterable<U> where U is implicit-convertible to T
* Works even if T is a primitive type, as long as implicit-convertibility holds

## Memory Model

* this reference is always placed on the stack when calling a non-static method

## Generics

* Type inference for instantiation of generic classes – checks both assignee type and constructor parameters to determine inferred type
	+ List<Integer> x = new ArrayList<>() – okay, inferred as Integer
	+ List<? extends Integer> x = new ArrayList<>() – good, inferred as Integer
	+ List<? extends Object> x = new ArrayList<>() – good, inferred as Object
	+ List<? super Integer> x = new ArrayList<>() – good, inferred as Object
	+ List<Object> x = new ArrayList<>().subList(0,1) – good, inferred as Object (but throws IndexOutOfBoundsException)
	+ List<Integer> x = new ArrayList<>().subList(0,1) – bad, inferred as Object and List<Object> cannot be implicitly converted to List<Integer>
	+ List<Integer> x = new ArrayList<>(new ArrayList<Integer>()) – good, inferred as Integer
	+ List<Object> x = new ArrayList<>(new ArrayList<Integer>()) – good, inferred as Object
	+ List<Integer> x = new ArrayList<>(new ArrayList<Object>()) – bad, cannot infer type argument (i.e. no type will give a valid expression)
	+ Object x = new ArrayList<>().get(0) – good, inferred as Object (but throws IndexOutOfBoundsException)
	+ Integer x = new ArrayList<>().get(0) – bad, inferred as Object which cannot be implicitly converted to Integer

## Exceptions

* Use try/catch/finally
* Only Throwables can be thrown
* Exception extends Throwable
* RuntimeException extends Exception
* All Exceptions that are not RuntimeExceptions must be explicitly declared in methods that might throw them
* More specific exceptions must be caught before more generic exceptions – catching a subclass of an already-caught exception is a compile error
* return statement in finally block will override return statement in try block (with a compiler warning that “finally clause cannot complete normally”)
* return statement in finally block will prevent exceptions from being propagated (with a compiler warning that “finally clause cannot complete normally”)
* Common Throwables:
	+ java.lang.Error (unchecked)
		- AssertionError
		- ExceptionInInitializerError
		- StackOverflowError
		- NoClassDefFoundError
	+ java.lang.Exception (checked)
		- ExecutionException
		- IOException
			* FileNotFoundException
			* FileSystemException
		- InterruptedException
	+ java.lang.RuntimeException (unchecked)
		- CancellationException
		- ClassCastException
		- IllegalArgumentException
		- IllegalStateException
		- IndexOutOfBoundsException
		- NoSuchElementException
		- NullPointerException
		- NumberFormatException

## Polymorphism

* An interface extends another interface
* A class extends another class
* A class implements an interface
* Can implicitly cast type A to type B ⇔ this casting is valid for all dynamic types storable in A
Can explicitly cast type A to type B ⇔ there exists a dynamic type storable in A such that this casting is valid (final keyword is taken into consideration)
	+ Casting upwards (superclass) – implicit cast works
	+ Casting downwards (subclass) – explicit cast only
	+ Casting sidewards from class A to class B – cannot (compile error)
	+ Casting sidewards from interface A to final class B or vice versa – cannot (compile error)
	+ Casting sidewards from interface A to non-final class or interface B or vice versa – explicit cast works because there might be a class that extends/implements both A and B
* Liskov Substitution Principle: "Let ϕ(x) be a property provable about objects x of type T. Then ϕ(y) should be true for objects y of type S where S is a subtype of T." This means that if S is a subclass of T, then an object of type T can be replaced by an object of type S without changing the desirable property of the program.

## Nested Classes

* Can access private members of enclosing class
* Declare class as static to not associate with an instance of the enclosing class
* Can also be declared in a method of the enclosing class (“local class”), anonymous class, or lambda expression
	+ Can access all local variables that are effectively final
* Anonymous class declaration:

# Java library

## Hashing

* Arrays.hashCode(arr) will calculate a hash code for an array of anything (both reference and primitive types)

## Functional

* Function<T, R> 🡪 .apply(T)
	+ .andThen(…), .compose(…)
* Predicate<T> 🡪 .test(T)
* Supplier<T> 🡪 .get()
* Consumer<T> 🡪 .accept(T)
* Arity = number of arguments

## Streams

* Stream.of(varargs…)
* Arrays.stream(arr)
* .parallel(), .sequential(), .unordered()
* Parallel stream operations:
	+ Must not *interfere* with stream data (i.e. must not modify its own stream)
		- ConcurrentModificationException will be thrown otherwise
	+ Should preferably be stateless (i.e. result should not depend on things that might change while executing the stream)
	+ Should minimise side-effects (e.g. writing data to a list, especially one that is not thread-safe)
* reduce(identity, accumulator, combiner)
	+ combiner.apply(identity, i) == i, for any i in the stream
	+ combiner and accumulator must be *associative*
		- combiner(a, b) == combiner(b, a), for any a and b
		- accumulator(accumulator(i, a), b) == accumulator(accumulator(i, b), a), for any a and b
	+ combiner and accumulator must be *compatible*
		- combiner.apply(u, accumulator.apply(identity, t)) == accumulator.apply(u, t), for any t and u

## Collectors

* 
* .characteristics()
	+ CONCURRENT – accumulator function can be called on the same result container concurrently from multiple threads
	+ IDENTITY\_FINISH – the finisher function is the identity function and can be elided
	+ UNORDERED – collection operation does not preserve the encounter order of input elements, i.e. it does not matter which other the input elements are fed to the collector
* To make a custome collector, either write a class that implements Collector or use Collector.of(…)
* Combiner is necessary even for serial collectors – there is no guarantee that combiner() will not be called

# Functors & Monads

## Functors

* 
* Functor laws:
	+ functor.f(x -> x) == functor
	(applying identity function must not change object)
	+ functor.f(g.compose(h)) == functor.f(h).f(g)
	(applying composed function must be same as applying sequentially)

## Monads

* 
* Monad laws:
	+ Has Monad.of(x) that wraps object(s) into a monad
	+ Monad.of(x).f(func) == func(x)
	(left identity law)
	+ monad.f(x -> Monad.of(x)) == monad
	(right identity law)
	+ monad.f(g).f(h) == monad.f(x -> g(x).f(h))
	(associative law, f() should be associative)

# Parallel and Asynchronous Programming

## Concurrency

* When different tasks are executed out-of-order without affecting the final outcome, i.e. program is decomposed into parts that are not dependent on order of execution

## Parallelism

* Multiple tasks are truly running at the same time
	+ on multiple processors which can run at the same time
	+ on a single processor capable of running multiple instructions at the same time
* See **Stream**

## RecursiveTask

* extend RecursiveTask<T> and override T compute() to write the task and write constructor to accept task arguments
* .fork(), .join() – run on another thread
* .compute() – run on this thread
* ForkJoinPool.commonPool().invoke(task) to start the task
* Has some overhead in spawning tasks, so parallelization should only be used if speed-up outweighs the overhead

## Asynchronous programming

* Asynchronous method call allows execution to continue immediately after invoking the method, so the rest of the code can continue to execute in parallel with the (long-running) method

## CompletableFuture

* Is a functor (.thenApply()) and a monad (.thenCompose())
* .\_\_\_Async() – runs the supplied functions on another thread



* .whenComplete((result, exception) -> {…})
* .exceptionally(exception -> {…})
* .thenApply(result -> {…})

# Style

## Order of class modifiers

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