# CS2030S

Finals Cheatsheet for 23/24 S2
zaidan s.

## Types

**Subtypes** $S <: T$ or $T >: S$ ($S$ is subtype of $T$)

1. **Reflexive** $S <: S$
2. **Transitive** $S <: T \land T <: U \implies S <: U$
3. **Antisymmetry** $S <: T \land T <: S \implies S = T$

**Subtyping Relationship**

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

**Widening and Narrowing Type Conversion**

**Widening** Variable of type $T$ can hold a value of type $S$ if $S <: T$
**Narrowing** Variable of type $S$ cannot hold a value of type $T$ if $S <: T$

**Variance of Types**

**Covariance** $S <: T \implies C(S) <: C(T)$
**Contravariance** $S <: T \implies C(T) <: C(S)$
**Invariant** Neither

## OOP Principles

**Encapsulation**

**Information Hiding** Private instance fields

**Abstraction**

Do not show actual implementation of methods.

**Inheritance**

Ability to reuse code of existing super classes.
Models **is-a** relationship.

**Polymorphism** Using same method signatures in subclasses to determine behaviour for specific subclasses.

**Tell Don't Ask** The client should not be doing computation on the object's behalf.

## Method Signature and Descriptor

**Method Signature** method name, number of parameters, type of each parameter and order of parameters
**Method Descriptor** method signature + return type

**Liskov Substitution Principle**
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 $S <: T$

*Replacing `super` with child should not change result.*

### Overriding & Overloading

**Overriding**

**Same method signature**: method name, number of parameters, type of parameters and order

Must use `@Override` annotation
Return type must be a subtype of the overridden method's return type.

**Overloading**

**Same method name** in the same class
**Different method signature** (cannot be different return type with same method signature)

## Interfaces and Abstract Classes

| **Interface** | **Abstract** |
|---|---|
| Can implement multiple | Can only extend one |
| Only abstract methods | Abstract & non-abstract |
| Can type-cast regardless of subtype | Cannot |

## Wrapper Classes

**Autoboxing** Convert primitive to wrapper
**Unboxing** Convert wrapper to primitive

**Allowed** Unboxing & Widening
**Not Allowed** Autoboxing & Widening

## Dynamic Binding

1. Determine compile-time type of target
2. Check all accessible methods (including inherited ones)
3. Most specific one callable
4. Determine run-time type of target
5. Determine method called.

| CTT | Methods Callable | Most Specific | RTT |
|---|---|---|---|
| | | | |

## Wildcards

**PECS** Producer Extends, Consumer Super
**Upper-Bounded Wildcards**
`A < ? extends S >` **Covariant**.
**Lower-Bounded Wildcards**
`A < ? super S >` **Contravariant**.

## Raw Type

`A<?>` Complex type of a specific but unknown type.
`A<Object>` Complex type of Object instances with type checking.
`A` Complex type of Object instances without type checking.
Raw types **throw unchecked warnings.**

## Type Inference

**Constraints**

1. Target Type
2. Argument Type
3. Type Parameter Bound

## Functional Interface

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

## Exceptions

**Checked Exceptions**

Exception programmer has **no control** over
All checked exceptions E <: Exception
If a checked exception, must put `throws` keyword

**Unchecked Exceptions**

Exception programmer has **control** over

All unchecked exceptions <: RuntimeException
Does not require `throws` keyword

**Exceptions can only compile** if:
The order of `catch` must be so that all exceptions are accessible (for checked exceptions)

## Immutability

**Advantages**

1. Ease of understanding
2. Enabling safe sharing of objects
3. Enabling safe sharing of internals
4. Enabling safe concurrent execution

## Nested Classes

**Inner class** access to outer class
**Static nested** inner class (only static vars)
**Local class** variable capture, effectively final
**Anonymous class** a local class without a name

## Monads and Functors

**Monads**

Left-identity law:
```
Monad.of(x).flatMap(y -> f(y)) = f(x)
```

Right-identity law:
```
monad.flatMap(y -> Monad.of(y)) = monad
```

Associative law:

```
monad.flatMap(x -> f(x))
    .flatMap(x -> g(x)) =
monad.flatMap(x -> f(x)
    .flatMap(x -> g(x)))
```

**Functors**

Identity
```
functor.map(x -> x) = functor
```

Composition:

```
functor.map(x -> f(x)).map(x -> g(x))
= functor.map(x -> g(f(x)))
```

## Streams

```
allMatch(Predicate<? super T> predicate)
anyMatch(Predicate<? super T> predicate)
noneMatch(Predicate<? super T> predicate)
takeWhile(Predicate<? super T> predicate)

// Bounded
count()
distinct()
reduce(BinaryOperator<T> accumulator)
toArray()

// Mutation
flatMap(Function<? super T,
    ? extends Stream<? extends R>> mapper)
map(Function<? super T,? extends R> mapper)
filter(Predicate<? super T> predicate)

// Parallel
parallel()
sequential()
unordered()
```

When using `reduce`,
1. `combiner.apply(identity, i) = i`
2. Combiner and accumulator must be associative
3. `accumulator.apply(u,t) = combiner.apply(u, accumulator.apply(identity,t))`

**Parallel**

**Concurrency** Divides computation into subtasks
**Parallelism** Subtasks are running at same time

**Asynchronous**

**CompletableFuture**

```
// instantiation
runAsync(Runnable runnable)
supplyAsync(Supplier<U> supplier)
completedFuture(U value)

// mapping
thenApply(Function<? super T,
    ? extends U> fn)
// map

thenCompose(Function<? super T,
    ? extends CompletionStage<U>> fn)
// flatMap

thenCombine(CompletionStage<? extends U> other,
    BiFunction<? super T, ? super U,
    ? extends V> fn)
// combine
```

## Fork-Join-Pull

**Order**

```
f1.fork();
f2.fork();
f2.join();
f1.join();
```

**Fork** adds task to the **front** of the queue
**Join** blocks computation until completed
**Task Stealing** idle threads will take from the **back** of the queue