CS2030S MT AY25/26
holy cheatsheet

1. Types

S is a subtype of T, i.e. S <: T if a piece of code written for variables of type T can be safely used on variables of type S. The subtyping relationship in general must satisfy two properties:

1. **Reflexive**: For any type S, we have S <: S
2. **Transitive**: If S <: T and T <: U, then S <: U

Primitive Types

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

Reference Types

There are only two kinds of types in Java. Apart from primitive types, all other types (objects) are **reference types**.
If an Object is declared but **not** initialized, e.g. `Circle c1;`, `c1` will be initialized to `null`, meaning "this variable does not refer to any object."

Liskov Substitution Principle

If S <: T, then

- Any property of T should also be a property of S.
- An object of type T can be replaced by an object of type S without changing some desirable property of the program.

VIOLATION if subclass changes the behaviour of superclass - some property no longer holds.

Preventing Inheritance and Overriding

`final` keyword can be used

1. in class declaration to prevent class from being inherited.
2. in method declaration to prevent method from being overridden.
3. in field declaration to prevent re-assignment.

Casting

Only type cast explicitly if we can prove that it is safe.

Variance

Let C(T) be a complex type based on type T. Then 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 above.

Java array is covariant. (S <: T implies S[] <: T[])

2. OOP Principles

Encapsulation

- **Encapsulation** = bundling data (fields) with operation (methods) that manipulate that data to maintain an **abstraction barrier** between implementation and usage.
- Realised in Java via classes/objects.
- **Goal**: class is responsible for keeping its own state consistent; clients ineteract via methods (not internals)

Information Hiding

- **Information hiding** enforces the abstraction barrier in practice: clients should only use the public interface and not rely on representation.
- Directly accessing fields may leak representation assumptions; changing representation then breaks client code.
- With hidden fields, **constructors** become th safe way to create valid objects.

Tell, Don't Ask

- Tell objects what to do instead of asking for internal data and doing logic in client.
- Getters and setters (accessors/mutators) are common but **can weaken encapsulation, increase coupling, and leak implementation details** when used indiscriminately.

Inheritance

- Use inheritance to model a IS-A relationship; use composition for a HAS-A relationship.
- Inheritance must preserve the meaning of subtyping or designs can become weird or brittle.

Method Overriding

Overriding: subclass defines an instance method with same method descriptor (signature + return type) as parent's method.

- Requires **identical method signature** for polymorphism to work correctly.
- Uses **dynamic binding**.
- Use `@Override` to get compiler to check for overriding.

Method Overloading

Overloading: Declaring a method with the same name but different parameter list (types, order, arity).

- Resolved entirely at compile time unlike overriding.
- Chaning only parameter names or only return type $\neq$ overloading.
- Selected overloaded method to run depends on **argument CTT**.

Polymorphism

Polymorphism: runtime method selection based on runtime type (dynamic binding)

Dynamic Binding - a 2-Step Process:

During Compile Time

CTT(target) used to determine method descriptor of method invoked. If CTT(target) = C, compiler searches for all methods in C (including inherited methods) that can be invoked on given argument and given return type. **Most specific method** is chosen.
If Java fails to determine a single most specific method, compilation error thrown.

During Run Time

Method descriptor from Step 1 (e.g. `boolean equals(Object)`) retrieve. RTT(target) = R is determined. Java then looks for first accessible method with matching descriptor in R, followed by parent class etc. until root `Object`.

Class Methods

Dynamic Binding applies to instance methods but **not to class methods**. Class method to invoke is resolved statically and fixed at compile time. RTT(target) ignored.

3. Abstract Class & Interface

Abstract Classes

A class that is so general that it cannot be instantiated. Useful if possibly one or more of its instance methods cannot be implemented without further details.
Abstract class can contain multiple fields and multiple methods (incl. class methods). Class with *at least one* abstract instance method must be declared abstract, but abstract class *may have no* abstract method.

```
abstract class Shape {
    private int numOfAxesOfSymmetry;
    public boolean isSymmetric() {
        return numOfAxesOfSymmetry > 0;
    }
    abstract public double getArea();
    //any concrete subclass must @Override to implement
}
```

Interface

- **Interface** models behaviour/capability.
- Methods are `public abstract` by default.
- class can extend only 1 superclass but implement multiple interfaces; interfaces can extend other interfaces.

```
interface I {...}
class A {...}
class B {...} implements I

I i; A a; B b;
i = b; //compiles: B <: I
b = i; //does not compile:  I </: B
i = a; //does not compile: A </: I

(I) a;
//compiles though A </: I, as subtype of A could implement I.
```

4. Wrapper Class

Auto-boxing & Unboxing

```
Integer i = 4; //auto-boxing
int j = i; //unboxing
```

- All primitive wrapper class objects are immutable, so every time a wrapper class object is updated, a new object is created, leading to performance issues.
- Always use `equals` method to compare wrapper class objects.

5. Exceptions

```
try {
    // do something
} catch (an exception parameter) {
    // handle exception
} catch (another exception parameter) {
    // can have more catch blocks
} finally { //optional
    // is executed regardless of whether an exception occurs
}
```

Throwing Exceptions

```
class Circle {
    ...
    public Circle(Point c, double r)
                throws IllegalArgumentException {
        if (r < 0) {
            throw new IllegalArgumentException("r > n");
        }
        this.c = c; this.r = r;
    }
}
```

Checked vs Unchecked Exceptions

An **unchecked exception** is an exception caused by programmer's errors e.g. `IllegalArgumentException`, `NullPointerException`, `ClassCastException`.
Unchecked Exceptions are subclasses of `RuntimeException`.
A **checked exception** is an exception that programmer has no control over and should be anticipated and handled.

Overriding Method that throws Exceptions

When overriding a method that throws a checked exception, the overriding method must throw only the same or more specific checked exception than the overriden method. (LSP)

The Error class

Errors are for situations where the program should terminate as generally no way to recover. `Exception` and `Error` <: `Throwable`.

6. Generics

Allow classes/methods to be defined without resorting to using the `Object` type.
Generics are invariant in Java.

Type Erasure

At compile time, type parameters are replaced by `Object` or the bounds (e.g. `? extends Number` is replaced by `Number`)
Given some class `Pair`,

```
class Pair<S, T> {
    private S first;        // -> private Object first
    private T second;       // -> private Object second
    ...
    public S getFirst() {   // -> public Object getFirst() {
        return this.first;  //       return this.first;
    }                       //    }
    ...
}

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

Implications of Type Erasure

Overloading based on Type Arguments

Given some class A, after type erasure we have...

```
class A{
    void foo(Pair<String, String> p) {...}
    // -> void foo(Pair p) {...}

    void foo(Pair<Integer, Integer> p) {...}
    // -> void foo(Pair p) {...}
}
```

Both methods have the same signature and code will not compile.

Using Type Parameters in Static Contexts

It is necessary for a generic class method to declare its own type parameters.

```
class D<T> {
    public static <T> T foo(T x) {...}
}
```

Generics & Arrays

Generic array *declaration* is fine but generic array *instantiation* is not.

```
new Pair<S,T>[2]; //illegal
new T[2]; //illegal
T[] array; //allowed
```

Bridge Methods

Bridge Methods are compiler-generated synthetic methods inserted after type erasure to preserve polymorphism for generics.
Bridge erased method signature expected by the superclass/interface to the more specific method in the subclass.
After erasure, if we have

```
class Box {
    void get(Object o) {...}
}
class StringBox extends Box {
    void get(String s) {...}
}
```

The compiler requires StringBox to implement a `void get(Object)` method, hence it adds the following bridge method within `StringBox`

```
void get(Object s) {
    this.get((String) s);
}
```

Unchecked Warnings

We cannot instantiate a generic type directly (e.g. `new T[]`), hence we can get around this restriction as follows:

```
this.array = (T[]) new Object[size];
```

This yields an **unchecked warning**, a message from the compiler that it has done what it can abe because of type erasures, there could be an unpreventable runtime error.
`@SuppressWarnings("unchecked")` cannot apply to an assignment but only to a declaration.
Using raw types can also lead to errors. Raw types only acceptable as an operand of `instanceof`

Wildcards

- **Upper-Bounded:** `? extends T`
- covariant (if S <: T, then `A<? extends S>` <: `A<? extends T>`
- **Lower-Bounded:** `? super T`
- 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 need to produce T values, use `List<? extends T>`
CS - if need to consume T values, use `List<? super T>`
if need both, use ?.

Type Inference

Diamond Operator

only for instantiating a generic type - not as a type

```
Pair<String, Integer> p = new Pair<>();
//Pair<> is inferred to be CTT(p) = Pair<String, Integer>
```

Given some class

```
class A{
    public static <S> boolean
            contains(Seq<? extends S> seq, S obj) {...}
}

A.<Shape>contains(circleSeq, shape);
//equivalent to
A.contains(circleSeq, shape);
```

Rules for Type Inference

- `Type1 <: T <: Type2` then `T` is inferred as `Type1`
- `Type1 <: T` then `T` is inferred as `Type1`
- `T <: Type2`, then `T` is inferred as `Type2`