abstract class

→ like a type.

↳ Interface

→ way to describe what can a class do.

→ eg. Comparable <T>

→ simple.

→ keyword: interface.

→ put in abstract methods.

eg. ~~public abstract~~ double getArea();

(all know its abstract.)

→ any class implementing interface how to implement its method.

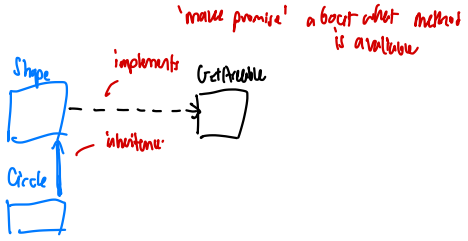
→ abstract something the class can do.

→ keyword 2: "implements"

→ if an abstract class don't need to have the abstract method. If using interface all.

→ can implement multiple interfaces (using commas).

→ can extend and implement together.



→ interface is a type.

→ C <: I = C implements I. (C is a subtype)

→ ∴ can use interface as general type.

→ extend at most 1 class.

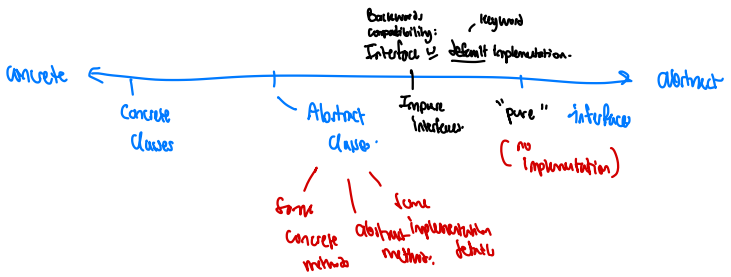
Class → implements multiple interfaces.

Interface → can inherit from multiple interfaces (not advisable).

↳ can become too complex.

Concrete vs Abstract classes.

Classes that can be instantiated.



```
abstract class A {
    private int count;
    public int getCount() {
        return count;
    }
    abstract boolean something();
}
```

— Not possible to instantiate

VS

```
interface B {
    boolean something();
    int doSomethingElse();
}
```

— have to implement these.

eg. api

- Contracts / methods which exist in the class.

Object is not a supertype of a primitive.

but `Object o = 1` works!

→ Java api: Wrapper Classes

→ Wrapper around primitives.

→ rather than using int, can use Integer

→ using wrapper class can be polymorphised to Object.

→ Integer <: Number <: Object.

→ final class, can't be inherited from.

→ extends Number, implements Comparable.

→ How to use Integer Class?

a lot of stuff

```
double d = 1.0;
Double d2 = new Double(1.1);
d2.doubleValue() → 1.1 (return value).
```

`Double rt = 1.0;` (now can directly set it)

↓ autoboxing ⇒ automatically box it up.

`double pt = rt;` → unboxing. Calling rt.value()

} Java compiler makes code more readable.

`Integer i = new Integer(4);`

`i = i + 1;` (immutable)

creating objects in heap

∴ huge overhead when using wrapper classes.

Stick to normal primitive unless need reference types.

Linux time Command to measure time

Command: eg. time java Main.

Wrapper class vs primitive

- Can use primitive type in any method that is written for reference type
- faster.
- less overhead

Type Getfrobble only specifies 1 method.

↳ just because something implements it doesn't mean it have its methods.

↳ Type casting.

Circle c = (Circle) g;

- runtime type
- compile time type
- gets rid of compiler protection.

→ to type cast
- be very sure
- narrowing type conversion is dangerous.

Variance of types — describes how complex types behave

- Covariant.
- Contravariant.
- Invariant.

eg. C(T) is a complex type based on T. — made up of component of some other type

S <: T → C(S) <: C(T)

T <: S → C(S) <: C(T)

neither.

eg. array.

Circle <: Object

Circle[] <: Object[]

if covariant

* Java array is covariant

```
Object[] o;
Integer[] i;
o = i    possible to point o to i.
i = o    otherwise is impossible
: Integer[] <: Object[].
Integer[] ints = new Integer[] {1, 2, 3};
Object[] objs;
objs = ints; // covariant
objs[0] = "Oh no!"; // possible. ⇒ String <: Object.
```

⇓

runtime error (diff. to debug, don't want people to see it).

ArrayStoreException (compiler can't check for this).

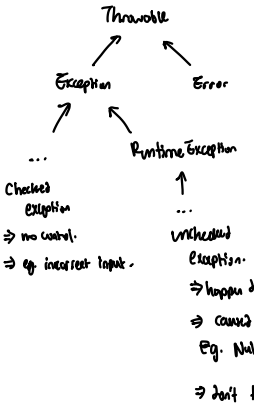
compile time errors are preferable.

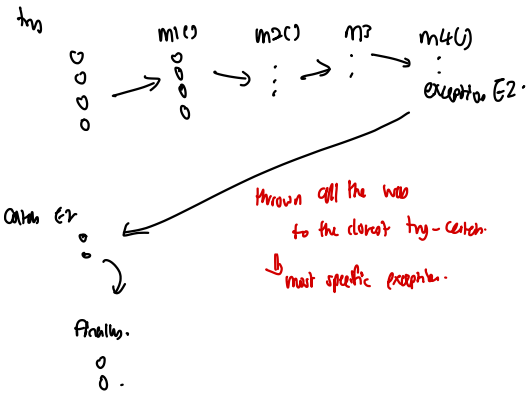
Error.

What can go wrong?
- incorrect inputs
- no inputs / corrupted inputs.

Java mechanism to handle errors.
- Exception
  - Classes, can be instantiated
  - try, catch, finally.
    - try: to write
    - catch: handle exception. / can have multiple
    - finally: clean up code, even if no exception are thrown. eg. close the file.
- Exception hierarchy.

Exception





Best practice?

- don't want to use exception as control flow.
- eg. `if (something != null){ something.task(); } else { somethingElse.task(); }` ⇒ `try { something.task(); } catch (NullPointerException e) { somethingElse.task(); }`

Might not be something which is null, might be some other method that throws this.

Should not use for decision making.

Also using Exception — may end up blocking some client from how hidden exception.

Also overreacting — Shouldn't completely shut down program, Use this as a way to recover from an error.