CS61B Notes

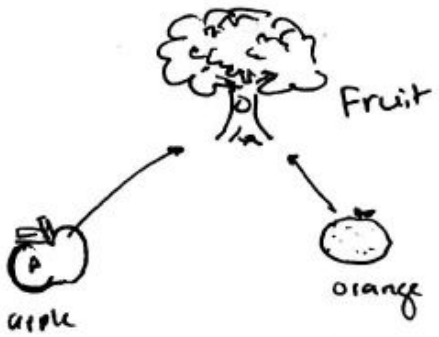
These notes will cover inheritance, interfaces, and typing (including static vs. dynamic and casting).

Inheritance: Extends

~~Usually its not enough to just define objects in a language because they~~ We want a way to represent RELATIONS between objects in a programming language. This is done in Java through INHERITANCE. For example:

One would say an orange and apple are

FRUITS!



Intuitively, if we say an apple and orange are fruits then whatever attributes and actions a fruit has, the oranges and apples should have them as well!

1) Super/Subclass and Extend.

The first way Java does inheritance is through sub/super classing. We can define Java classes ~~with respect~~ that

Inherit & ~~[illegible]~~ variables (attributes) and methods (actions) from other classes. Consider the fruit class.

```
public ~~Fruit~~ class Fruit {
    int weight ;
    String name ;
    public Fruit ( int w , String n ) {
        weight = w;
        name = n
    }
    public void grow ( ) {
        weight += 100 ; // does a fat fruit .
    }
    public void squish ( ) {
        System . out . println ( " splat! " );
    }
}
```

Now since an Orange is a Fruit, we can define an Orange class that inherits all methods and variables from Fruit.

```
public class Orange extends Fruit {
    String location ;
    // next page .
```

← The extends keyword tells Java Orange is a SUBCLASS of Fruit

```
public Orange (int w, String n, String l) {
    super(win);
    if (l.equals("Florida") || l.equals("California")) {
        location = l;
    } else {
        System.out.println("No");
        location = "California";
    }
}
...
```

calls the super class's (Fruit) constructor

Oranges can only be from California or Florida. Everything else is fake!

The above is code for Orange. Notice a few things.

- we didn't redefine any instance variables or methods. because we don't need to! Orange already inherits weight, mem grow and squish.
- ~~we can access all superclass stuff~~ By extending Orange from Fruit, we say Orange is a SUBCLASS of Fruit. Equivalently Fruit is the SUPERCLASS of Orange.
- we can access the constructor of the superclass by calling `super(...)` in the constructor. If this line is in the constructor it MUST be first in the constructor. Otherwise it is automatically inserted by the compiler.
    - Ergo any subclass's constructor ~~must~~ always makes a call to super!

Just like ~~this~~ we can access the super classes constructor, we can also access the super classes methods, and variables using super.

super.weight ⟶ Fruit's weight
super.squish() ⟶ Fruit's squish.

[2] Overriding

Okay but when does it make sense to call Fruit's squish! Or any super class's methods for that matter... When overriding.

Consider that perhaps Orange's behavior for squish is different (because it doesn't just splat, it has a peel ~~[illegible]~~ the peel must ~~[illegible]~~ split first!) We can redefine squish as follows.

```
// inside Orange class ...
@Override  <-- include for style.
public void squish() {   <-- peel splits first
    System.out.println("Split!");
    super.squish();   <-- then you get a splat by calling Fruit's squish.
}
```

A few comments.

- People usually include @Override as a piece of style (it makes code more readible by telling people squish overrides Fruit's squish). This is called a Javac directive.
- This is called <u>overriding a method</u>. Namely redefining a method inherited by the subclass.

[3] Field Hiding.

But what about for variables. Well in the same manner, I can sort of override a variable by redefining it in my subclass like so...

```
public class Orange extends Fruit {
    int weight; // now I've redefined weight
    String location;
    ...
}
```

DON'T DO THIS! This is terrible practice because intuitively this makes no sense AND it makes execution weird... Consider since Orange inherits Fruit's weight, ~~then~~ then as Orange is a Fruit, the weight of the orange whether or not it's considered an orange or a fruit should be the same. But it's not! Suppose our constructor in Orange was defined as...

```
public class Orange extends Fruit {
    int weight;  ——→ Sad!
    String location;
    public Orange (int w, String n, String l) {
        super(w, n);
        weight = 9001;  ——→ Oh no...
        ... // everything else the same.
    }
    ...
}
```

now if I write the code...

```
Orange o = new Orange(61, "Tim", "florida");
System.out.println(o.weight);

Fruit f = (Fruit) o;
System.out.println(f.weight);
```

The first prints [9001] but the second prints [61] because super(w,n) sets weight in the Fruit class (since Orange is a fruit) to weight = w = 61. But the line weight = 9001 sets weight in the Orange class to 9001. Strange...

The trick to understanding this is to keep in mind what the static type is... we'll come back to this later when we talk about Types!

[4] Variable Access.

~~This is the~~ Now we've always said to keep your variables private because it's good practice. But what is private?

A variable that is private can only be accessed directly from within the class definition. This means the following:

Consider if we made weight private in Fruit.

```
public class Fruit {
    private int ~~fruit~~ weight;   oh no...
    ...
}
```

now in our Orange class, we write...

```
public class Orange extends Fruit {

    String location;

    public Orange (int w, String n, String l) {
        super(w,n);
        weight = wtl;
        ... // the rest the same
    }
```

→ what happens?

Super(w,n) will work because the superconstructor is called from wl in the definition of Fruit but weight = w will not ~~because~~ ~~work~~ UNLESS Orange is defined as an inner class wl in Fruit just like how node was in IntList.

The way to get around this is to make weight [protected] inside Fruit that is to write

private weight; ⟶ protected weight;

which means weight is accessible to the class and all subclasses

[5] Constructors.

This note is particularly pedantic but ~~[illegible]~~ interesting nonetheless. So notice how in Fruit we have a constructor wl args and same in Orange. If in my Orange constructor, I write...

```
public Orange (int w, String n, String l) {
    super();  ——→ does that work?
    ...
}
```

calling super() does not work because in Fruit I do not have a constructor that takes in no arguments (however defined one). However, if we removed the constructor we defined in Fruit then it would work.

This is because of default constructors. If we define a class w/o a constructor (say Blah.java) we can still construct a Blah object by calling new Blah() because the compiler will automatically provide a DEFAULT CONSTRUCTOR that ~~sets all variable~~ takes no arguments and sets all instance variables to its default values (Objects → null and primitives → 0);.

Thus because we defined a constructor in Fruit, it will not have a no-args constructor and super() will fail.

Interfaces

Now if we consider an orange, we notice that it's kinda like a ball in that we can roll it or squish it. So that means we should just make Ball a class and then have Orange extend that right?

Wrong! Because if Ball defines squish() and ~~Thing~~ Fruit defines squish(). which squish() does Orange use? To get around this Java defines interfaces and enforces class to only be able to extend one ~~thing~~ class while implementing many interfaces.

Now an interface is simply a list of methods (and sometimes implementations if you consider default methods but let's ignore them for now). The syntax is as follows...

```
public interface Ball {
    void roll();
    void bounce();
}
```

→ a list of method headers.

To have a class use an interface, the class must implement it. Suppose Orange implements Ball...

```
public class Orange extends Fruit implements Ball {
    String location;
    public Orange(int w, String n, String l) {
        ...
    }
    ...
    @Override
    public void roll() {
        System.out.println("Im rolling");
    }
    @Override
    public void bounce() {
        System.out.println("boing");
    }
}
```

↳ tells compiler Orange implements Ball

→ To implement an interface means the class MUST implement the interface's methods.

Now notice a few things, a class that implements an interface MUST implement the methods in the interface or be declared abstract. Second ~~an interface~~ a class may implement as many interfaces as it wants. Hence we can have another interface

```
public interface Edible {
    void eat();
}
```

And Orange may implement this as well...

```
public class Orange extends Fruit implements Ball, Eatable {
    ...
    @Override
    public void eat() {
        ...
    }
}
```

Why is this what we want? Well remember, an Orange may be a Fruit and a Ball. In general, an object can be many things and so really what we wanted to do was to find a way to represent an object inheriting from multiple other objects (this is called MULTIPLE INHERITANCE)

Extending from multiple classes wouldn't work because you could have conflicting method definitions.

But extending from one class then implementing from multiple interfaces works very well because think about what actually happens. When we implement an interface in an object, we are forced to implement all methods declared in that interface. This means the object ~~takes on the behavior~~ basically inherited the behavior any object implementing that interface would have.

Further because interfaces ~~don't~~ are just a list of methods (again NO LONGER true because of default methods) we won't

have conflicts in implementation. [illegible].

[illegible] How are interfaces used in Java?

1 An example

Suppose we want to define a generic method to sort a list. Well in Java there are many different list like objects.

ArrayList

LinkedList

Vector

CopyOnWriteArrayList

...

just to name a few

They all implement a list in different ways. Now immediately we could write a sort method for each type of list ... but that sucks. Don't do that. Instead, we're going to notice that each of these objects implement Java's List interface!

In Java, we can use List as a type. Meaning, we can do this...

```
List<Integer> memes = new ArrayList<Integer>();
```

Now memes as an object will only allow you to call methods defined in the List interface. This is because the static type of memes is List<Integer>. (We'll talk

about static type later). Now we can write a method like so!

```
public static void sort ( List<Integer> list ) {
    ...
    //do things
}
```

And it will sort all lists!

[2] Default Methods.

We said that interfaces are only a list of methods welp, I lied

We can define default methods in interfaces that provide a default implementation for methods in an interface. For example

```
public interface Ball {
    void roll() ;
    default void bounce() {
        System.out.println("bang");
    }
}
```

default keyword declares this blockcas a default method!

Now in Orange class we need only implement ~~bounce()~~ roll() since bounce() is already inherited and implemented!

```
public class Orange extends Fruit implements Ball {
    String location;
```

```
public Orange(int w, String n, String l) {
    ...
}
@Override
public void squish() {
    ...
}
```

```
@Override
public void roll() {
    ...
}
```

→ still required because roll is not implemented in Ball

```
// no more methods!
}
```

To make it even weirder, we can use other methods ~~if they~~ declared in the interface and even declare objects THAT IMPLEMENTS THE INTERFACE! Basically ~~how~~ you can write it like a normal method...

```
public interface Ball {
    void roll();
    default void bounce() {
        roll();
    }
}
```

→ calls roll which must be implemented in the implements class.

```
Orange o = new Orange(7, "what", "the heck");
o.~~bounce~~ roll();
```

what about inheritance conflicts because WE HAVE THOSE NOW.

well there are two cases.

- A ~~subclass implements~~ superclass implements a default method. For example Fruit implements bounce() Here, the super class's bounce() is inherited by Orange.
- Two interfaces implement a default method, here you ~~get a compiler error~~ have two FURTHER cases....
  - if the superclass implements the method, you're fine
  - if not, then you get a compiler error.

Basically default methods are used when ~~a subclass~~ an implementing class or its superclass hasn't implemented the method.

~~Secondly, what I need to make of a perfectly good solution~~

because

Typing

There is one part we've not talked about regarding inheritance (sort of). What methods and variables are actually called when during execution. To setup this discussion we'll talk about Java's type system.

So in Java, each variable has types and it must be declared when we create it. For example.

```
int a = 10;

Orange o = new Orange(17, "James", "Florida")
```

For primitives they kinda only have one type, the type that they're declared with.

```
int a = 37;  ⟶ type is int.
```

However reference types (namely objects) have two associated types: static and dynamic.

(Static type): The type of the object at compile time

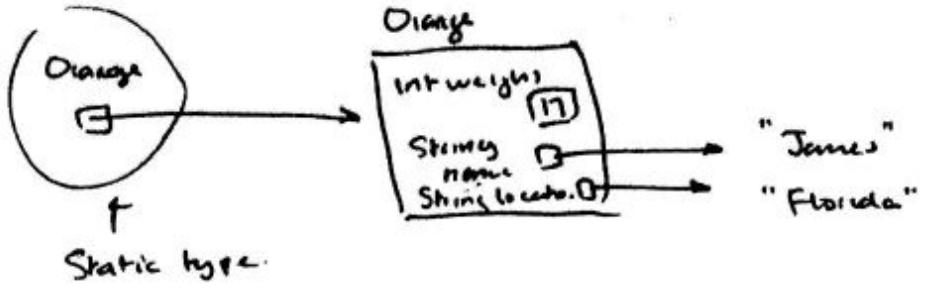
(dynamic type): The type of the object at runtime.

But what is the type of the object at compile time? Its the declared type!

```
Orange o = new Orange(17, "James", "Florida");
```

↑ this thing.

If we draw out the box and pointer diagram.



~~Thus~~ Because the static type is the type at compile time, at compile time, the compile ONLY KNOWS the static (declared) type. ~~From~~ ~~Attributes of the final~~

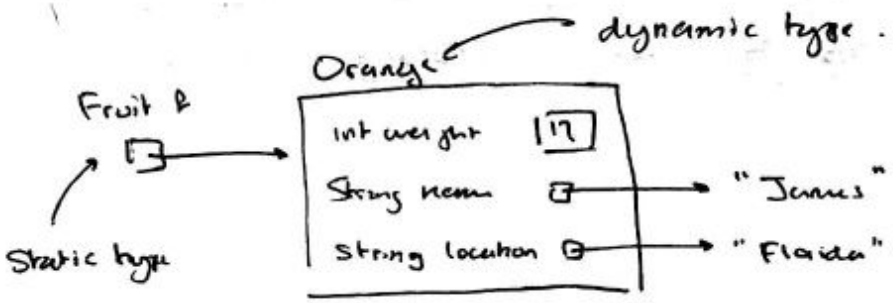
The dynamic type is the type of the object at runtime. This is the type the object was instantiated with. e.g.

```
Orange o = new Orange(17, "James", "Florida");
               ↑this.
```

Why is this distinction important? Because we can also do this.

```
Fruit f = new Orange(17, "James", "Florida");
```

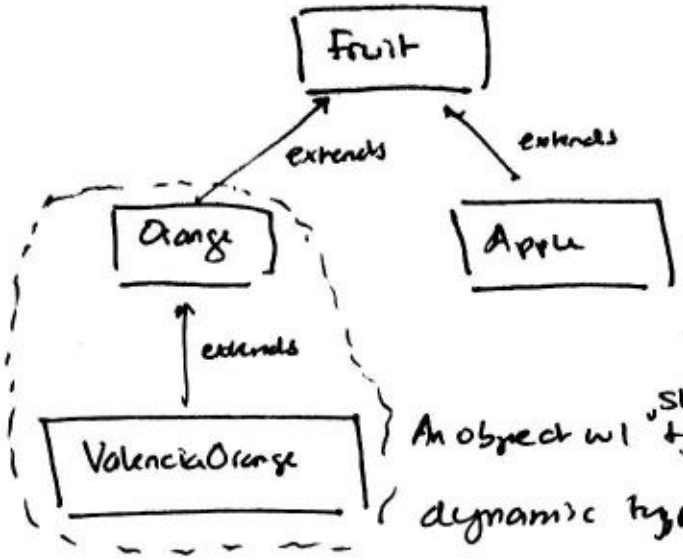
now the static type is Fruit, the dynamic type is Orange



How does this fit into inheritance?

In general the static type overbounds the dynamic type

meaning the dynamic type must be either the static type or a subclass of the static type!



~~Fruit can be object~~

An object w/ "static type Orange can have dynamic type Orange or ValenciaOrange but CANNOT be ~~a~~ Fruit.

1) Casting

In Java you can coerce an object to be a specific type during compile time through casting. So why is this useful? First consider that during compile time an assignment parses "compilation" the type

Casting

In java you can coerce the type of an object to be something else

for example, the following would not ~~work~~. COMPILE!

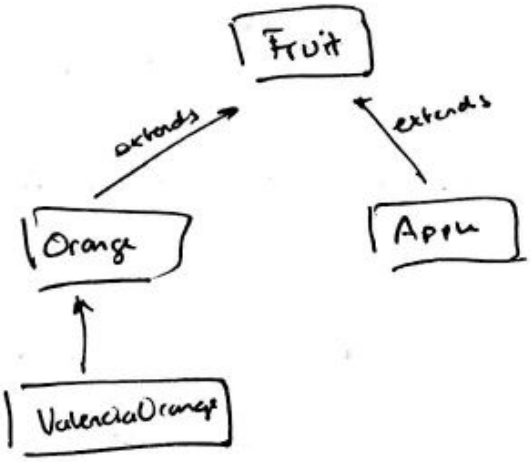
```
Fruit f = new Orange();
Orange o = f;
```

This because an Orange cannot be be set to a Fruit (remember the static type is only known at compile time). However, if we do this

```
Fruit f = new Orange();
Orange o = (Orange) f;
```

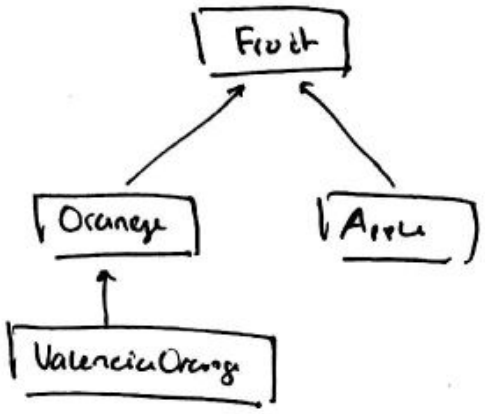
we'll pass compile time because (Orange) tells the compiler hey trust me f is an Orange. Then during run time, when we figure out f's dynamic type is Orange, everything is ok!

So what can we actually cast... Consider the inheritance tree for Fruits.



When checking if a cast works, we need to check during compile time and during runtime because it can error at both times!

- During compile time: you can cast up and down a branch



For example: these codes pass compile time.

```
Fruit f = new Fruit(...);
Orange o = (Orange) f;
ValenciaOrange v = (ValenciaOrange) o;
```

But you cannot cast across branches

```
Orange o = new Orange();
Apple a = (Apple) o;
```

} throws compiler error!

Because an Orange can never be an Apple (no matter how much it wants to be).

- During runtime: now the cast only works if its the same type as the dynamic type or a superclass of the dynamic type.

The first block from the previous page does not work!

```
Fruit f = new Fruit(...);
Orange o = (Orange) f;
```

because the dynamic type of f is Fruit ~~which is not a~~ and Orange is NOT a superclass of Fruit. But this works

```
Fruit f = new ValenciaOrange(...);
Orange o = (Orange) f;
```

Since Orange is a superclass of ValenciaOrange.

[2] What method is called.

Java method lookup is kind of confusing! But here are some rules to get you through it.

① An object has accers to the methods of its static type. AND access to methods that its dynamic type has overriden.
↑ important!

② Method lookup for parameters (arguments passed in) is done by static type.

example.

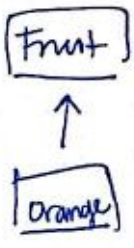
```
Fruit f = new Orange(...);
Fruit o = new Orange();
Orange r = new Orange(...);
o.method(f)
```

↑ look at Fruit methods + methods overriden by Orange.

Get: Orange::method(fruit). So (B) is called.

```
o.method(r);
```

(C) does not override anything! compilation error



Fruit
(A) method(Fruit)

Orange
@override
(B) method(Fruit)

(C) method(Orange)
↑ not overriden!