



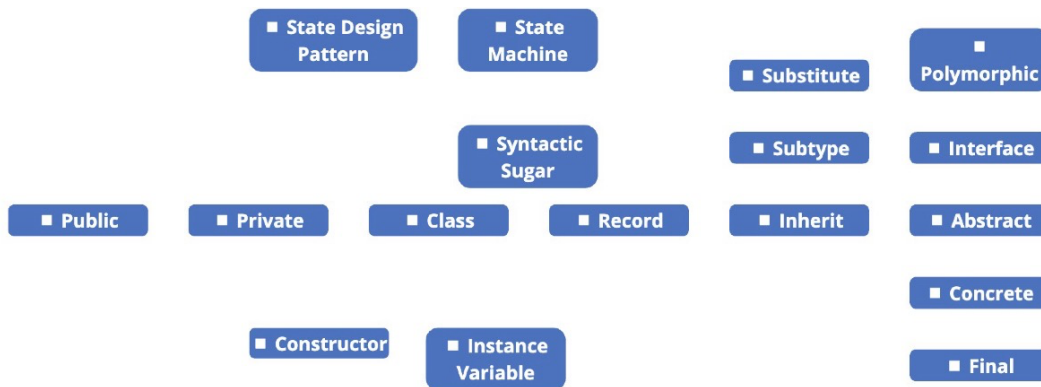
Inheritance

Student name:	TA signature:
---------------	---------------

Photo by Liane Metzler on Unsplash



Concepts Check off understood concepts, connect related concepts, label connections



Make sure you can **explain** each concept and each connection, you can provide **examples**, and you can **identify** them in a given piece of code.

Names Circle the methods, underline the types

There are no new names this week



Constructors

A **constructor** contains code that is executed to initialize a newly created object. This means, it will initialize all the instance variables of the object.

When you write `new Xxx()`, the constructor of class `Xxx` is called. It can do whatever it wants, but usually it will set each instance variable of the newly created object to its initial value.

A constructor is **similar** to a method, but there are some **differences**.

Complete the following table (check the appropriate boxes):

	Constructor	Method
Can freely be named?	<input type="checkbox"/> yes / <input type="checkbox"/> no	<input type="checkbox"/> yes / <input type="checkbox"/> no
Can have parameters?	<input type="checkbox"/> yes / <input type="checkbox"/> no	<input type="checkbox"/> yes / <input type="checkbox"/> no
Has return type?	<input type="checkbox"/> yes / <input type="checkbox"/> no	<input type="checkbox"/> yes / <input type="checkbox"/> no
Can be overloaded?	<input type="checkbox"/> yes / <input type="checkbox"/> no	<input type="checkbox"/> yes / <input type="checkbox"/> no
Can contain statements?	<input type="checkbox"/> yes / <input type="checkbox"/> no	<input type="checkbox"/> yes / <input type="checkbox"/> no

The name of the constructor is the name of the class.

In some way, a constructor is similar to a **class (i.e., static) method**, because the constructor call does not involve an object (`new Xxx()`, not `o.xxx()`).

But in some other way, a constructor is similar to an **instance method**: once it starts running, there will be an object, accessible via the **this** local variable.

```
public class Point {
    ...
    public Point() { // here is a constructor -- note: no return type!
        this.x = 10; // initialize the x instance variable
        this.y = 3; // initialize the y instance variable
    }

    public int getX() {
        return this.x;
    }
}
```

Instance Variables

In the above code the constructor initialized the `x` and `y` **instance variables**. But in Java, variables have to be declared before we can play with them! Where and how were those variables declared?? We hid that behind the `...`! Here's the secret:

```
public class Point {
    private final int x; // declare the x instance variable
    private final int y; // declare the y instance variable
    ...
}
```



Desugaring Records into Plain Old Classes

So far, when we wanted to create objects with specific fields, we defined **record classes**. However, we also can use plain old **classes**.

Complete the code on the right, so the class corresponds to the record class given on the left:

```
public record Pos(int x, int y) {  
}
```

```
public class Pos {  
  
    private final int x;  
    private final int y;  
  
    public Pos(int x, int y) {  
  
    }  
  
    public int x() {  
  
    }  
  
    public int y() {  
  
    }  
  
    public boolean equals(Object o) {  
        // let's skip this for now  
    }  
  
    public int hashCode() {  
        // let's skip this for now  
    }  
  
    public String toString() {  
  
    }  
  
}
```

All the code on the right is generated automatically for you by the Java compiler when you declare a record class.

Draw a memory diagram (just the heap) of the object created in both cases:

```
new Pos(1, 2)
```

```
new Pos(1, 2)
```

A record class is simply a convenient, shorter form to declare a traditional Java class! We call such shorter ways to write code **syntactic sugar**. Java records are syntactic sugar. They are sweet!



Information Hiding: public vs. private

When you write a class, you can decide, for each field and each method, whether it should be accessible from code outside the class (`public`), or whether it should only be accessible from code inside the class (`private`).

The following code shows a **good** design, where the `time` instance variable of class `TimeStamp` is **private**, and the behaviors one might want to do with a `TimeStamp` are provided through **public methods**:

```

public class TimeStamp {
    private final int time;
    public boolean before(TimeStamp other) {...}
}

public record TimeInterval(TimeStamp begin, TimeStamp end) {
    public boolean before(TimeInterval other) {
        return this.end.before(other.start); // no dependency on time
        // return this.end.time < other.start.time; // not possible (time private)
    }
}

```

Here is a **bad** design, where the field is **public**, and the code of the `TimeInterval` class plays with the `time` field directly:

```

public class TimeStamp {
    public final int time;
}

public record TimeInterval(TimeStamp begin, TimeStamp end) {
    public boolean before(TimeInterval other) {
        return this.end.time < other.start.time; // bad: dependency on time
    }
}

```

This bad design has some negative consequences, for example, it prevents the `TimeStamp` code from being refactored into a class that stores time as a `BigDecimal` (which might be necessary if we needed to support arbitrarily long intervals of time in the future).

As a guideline, all instance fields should be private. And methods that are only of “internal use” should be private as well.

Which of the following claims are correct?

<input type="checkbox"/>	private fields/methods are not visible outside the class
<input type="checkbox"/>	private instance fields/methods are not visible outside the object
<input type="checkbox"/>	class <code>C</code> { private int <code>f</code> ; public int <code>m(C o)</code> { return <code>o.f</code> ; } }
	The access of <code>o</code> 's field <code>f</code> is possible



Inheritance of Implementation

A class can implement an **interface** *I*, and then must* implement the methods prescribed by *I*. A class can also extend another **class** *c*, and then must* implement the methods prescribed by *c*.

In both cases, the class is a **subtype** of the other thing (of interface *I*, of class *c*).

However, there is a difference! When a class extends another class, it also **inherits** the *implementation* from that class. If the superclass defines a *field*, the subclass inherits it. If the superclass implements a *method*, the subclass inherits that method, including the implementation.

Draw the memory diagrams for both cases:

<pre>public interface Super { public String sing(); }</pre>	<pre>public class Super { private final String song; public Super() {song = "lalala";} public String sing() { return song; } }</pre>
<pre>public class Sub implements Super { private final String song; public Sub() {song = "lalala";} public String sing() { return song; } private final int volume = 10; public int getVolume() { return volume; } }</pre>	<pre>public class Sub extends Super { private final int volume = 10; public int getVolume() { return volume; } }</pre>
<pre>Super p = new Sub(); String s = p.sing(); Sub b = new Sub(); int v = b.getVolume();</pre>	<pre>Super p = new Sub(); String s = p.sing(); Sub b = new Sub(); int v = b.getVolume();</pre>
Draw stack frame and heap:	Draw stack frame and heap:

What are the fields that go into the objects of type Sub?



Abstract Methods, Abstract Classes

What is the difference between the methods we up to now defined in **interfaces** versus those we defined in **classes**?

The methods in interfaces are **abstract**. So far, all the methods we defined in classes were **concrete**.

Differences between abstract and concrete methods: **Answer** each cell (YES/NO):

	Abstract methods	Concrete methods
Have a header (name, parameters, return type)		
Have a body		
Have an implementation		

When defining an interface, one does not need to use the `abstract` modifier when declaring methods. Normal interface methods are abstract by definition.

```
public interface Turtle {
    public Turtle move();
    public abstract Turtle turnLeft();
}
```

Both methods (`move` and `turnLeft`) in the above `Turtle` interface are abstract.

Even classes can declare abstract methods. `Thing` is a class that defines a concrete method `getName` and an abstract method (see, no body!) `describe`.

```
public abstract class Thing {
    public abstract String describe();
    public String getName() { return "something"; }
}
```

If a class declares any abstract method, the class itself **must** be abstract. What does that mean? If a type is abstract, we cannot instantiate it. We cannot create instances of the abstract class `Thing`. Why? Because, what would happen if we created a `Thing` object and called the abstract method `describe` on it?

```
new Thing().describe() // what should this do???
```

For the same reason, we cannot instantiate an interface:

```
new Turtle().turnLeft() // what should this do???
```

There is no code for method `turnLeft` (and no code for method `move`). This is no problem, because `Turtle` is an interface, and thus is very abstract, and we cannot instantiate it, and thus we cannot end up with an object for which we don't have the implementation of a method we can call on it.



When extending a class, the subclass doesn't just **inherit** the **requirements** imposed by **abstract** instance methods, but it also inherits the **implementations** of **concrete** instance methods. And it inherits the instance **fields**.

Draw the class diagram of this class hierarchy (no need to draw Object):

```
public interface Exp {
    public int eval();
}
public record Lit(int val) implements Exp {
    public int eval() { return val; }
}
public abstract class BinOp implements Exp {
    private Exp left;
    private Exp right;
    public BinOp(Exp left, Exp right) {
        this.left = left; this.right = right;
    }
    public abstract int compute(int l, int r);
    public int eval() {
        return compute(
            left.eval(), right.eval()
        );
    }
}
public final class Add extends BinOp {
    public Add(Exp left, Exp right) {
        super(left, right);
    }
    public int compute(int l, int r) {
        return l + r;
    }
}
public final class Sub extends BinOp {
    public Sub(Exp left, Exp right) {
        super(left, right);
    }
    public int compute(int l, int r) {
        return l - r;
    }
}
```

Which of the following claims are correct?

<input type="checkbox"/>	new Exp() creates a new object of type Exp
<input type="checkbox"/>	new Lit(1) creates a new object of type Lit
<input type="checkbox"/>	new BinOp(new Lit(1), new Lit(2)) creates a new object of type BinOp
<input type="checkbox"/>	Exp is a supertype of Add
<input type="checkbox"/>	Sub is a subtype of Add
<input type="checkbox"/>	Sub cannot be subtyped
<input type="checkbox"/>	Exp is a subtype of Object
<input type="checkbox"/>	We can substitute a Sub for a BinOp



Draw a memory diagram (heap) of `new Add(new Lit(1), new Lit(2))`

Final Methods, Final Classes

While **abstract methods** have to be eventually implemented in subclasses, **final methods** CANNOT be implemented (overridden) in subtypes.

```
public class BinOp {  
    ...  
    public final int eval() {  
        return compute(left.eval(), right.eval());  
    }  
}
```

In the above `BinOp` class, the method `eval` is `final`. This means that subclasses (like, e.g., `class Add extends BinOp`) cannot possibly override the `eval` method. The `eval` method is cast in stone. We can be 100% sure that it will work the way it is implemented in class `BinOp` for all subclasses of `BinOp`.

Not only can we make a method `final` to protect it from being overridden in a subclass, but we can even make the entire class `final`, to protect it from being subclassed at all:

```
public final class Password { ... }
```

Nobody can create a subclass of `Password`. If you expect a `Password`, you are 100% certain you're getting an instance of the `Password` class, and not an instance of some "hacked" subclass of `Password`. The Java library contains several `final` classes, amongst them class `String`. It's not possible to extend class `String`. A variable of type `String` always refers to an object of exactly the type `String`.

If you want to be cautious, you will make all methods `final` and all classes `final`, unless you have a clear reason for not doing so.

Note: `final` for methods/classes is **different** from immutability (`final` variables). `final` methods cannot be overridden. `final` variables cannot change their value.



Interface, Abstract Class, Concrete Class, Final Class

Java provides various degrees of abstraction for classes:

Kind of Type	Can have abstract instance methods	Can have concrete instance methods	Can have instance fields	Can be instantiated	Can be subtyped
Interface					
Abstract Class					
Concrete Class					
Final Class					

The declarations of `abstract`, `concrete`, and `final` classes look almost the same:

```
public abstract class MyAbstractClass extends ... implements ... { ... }
public class MyConcreteClass extends ... implements ... { ... }
public final class MyFinalClass extends ... implements ... { ... }
```

Interfaces are at the top of class hierarchies. An interface can be subtyped by another interface (`interface Sub extends Super {...}`). Below interfaces come abstract classes. Below that usually come the concrete classes, which can be instantiated because they have no abstract methods. The `final` classes must be at the bottom of the class hierarchy, they cannot have further subtypes.

The Root of the Class Hierarchy – `java.lang.Object`

In Java every reference type is a subtype of class `java.lang.Object`. You don't specify this. It's implicit. (Strangely, even an interface is a subtype of `Object`.)

As a consequence of every class extending `java.lang.Object`, every class inherits the methods implemented in `Object`, amongst them:

```
public String toString()
```

Because of this, you can get a string representation of any object. And because `Object.toString` is not `final`, subclasses can override the `toString` method to provide their own, more appropriate string representations. What strings do the following expressions produce?

<code>new Object().toString()</code>	
<code>new String().toString()</code>	
<code>"ABC".toString()</code>	
<code>new Lit(1).toString()</code>	
<code>new Add(new Lit(2), new Lit(3)).toString()</code>	

Another method in `java.lang.Object` allows comparing this object to any other object. Records implement `equals` by comparing whether all their fields are equal.

```
public boolean equals(Object other)
```

Subtyping & Polymorphism: The Liskov Substitution Principle



We have seen **subtyping**: for example, we saw that a `Cons` is a (specific kind of) `Seq`.

```
public interface Seq {
    public abstract boolean isEmpty();
}
public record Cons(String first, Seq rest) implements Seq {
    public boolean isEmpty() { return false; }
}
```

This means that whenever we expect a `Seq`, we are happy if we receive a `Cons`:

```
public static int length(Seq s) { ... } // length expects a Seq
length(new Cons(...)); // length is happy with a Cons
```

We can **substitute** a `Cons` for a `Seq`. We can do this, because the definition of our `Cons` guarantees that it will have **all** the methods that `Seq` promises. Our `length` method, that expects a `Seq`, will be happy with the given `Cons`, because the given `Cons` will support everything the `length` method expects to be able to call.

This **substitutability** applies to every kind of variable (not just a parameter):

```
Seq s; // s is expected to point to a Seq
s = new Cons(...); // s points to a Cons object
```

Barbara Liskov (who got an honorary PhD from USI in 2011) received the **Turing Award** for her contributions to informatics, including for the idea now known as the **Liskov Substitution Principle**:

$$S \leq T \rightarrow \forall x:T. \phi(x) \rightarrow \forall y:S. \phi(y)$$

If `S` is a subtype of `T` (e.g., `Cons` is a subtype of `Seq`), then if some property ϕ holds for values of type `T` (`Seq`) that property must also hold for values of type `S` (`Cons`).



This principle is the basis of object-oriented programming, subtyping, and **polymorphic method calls**. It guarantees that if a variable `o` has type `Sprite`, and there are various subtypes of `Sprite` (e.g., `Pacman` and `Ghost`), and type `Sprite` has a method `step`, we can compile `o.step()`, and no matter whether `o` points to a `Pacman`, a `Ghost`, or an object of some other subtype of `Sprite`, the class of the object `o` points to will have an implementation of method `step`.

```
public interface Sprite { public abstract void step(); }
public class Pacman implements Sprite { public void step() {...} }
public class Ghost implements Sprite { public void step() {...} }

Sprite s = new Pacman();
s.step(); // which exact method will this call?
s = new Ghost();
s.step(); // which exact method will this call?
Seq<Sprite> sprites = of(new Pacman(), new Ghost(), new Ghost());
for (Sprite sprite : sprites) {
    sprite.step(); // which exact method will this call?
}
```



“State” Design Pattern

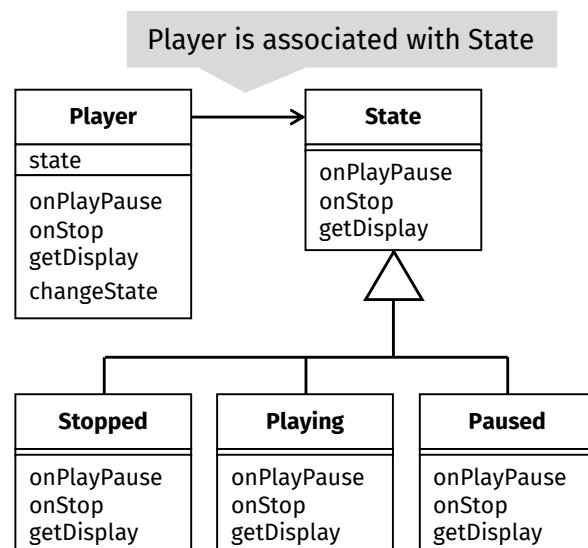
Let’s model a media player with a play/pause button  and a stop button . Pressing these buttons allows you to put your player into three different states: stopped, playing, and paused.

Draw a diagram of a “state machine”, with a circle for each possible state. For each circle, draw arrows going from that circle for each button that could be pressed. The arrows point to the state that button press would cause the player to go into.



State machines are common when modeling real-world processes and systems (traffic lights, vending machines, ...). Let’s model our media player in Java. We need two methods that can handle the two buttons, and a method that can produce the text to show on the player’s display:

```
public record Player(State state) {
    ...
    public Player onPlayPause() {
        // action depends on state
        return state.onPlayPause(this);
    }
    public Player onStop() {
        // action depends on state
        return state.onStop(this);
    }
    public String getDisplay() {
        // action depends on state
        return state.getDisplay();
    }
}
```



Our Player can be in one of three different states. Its behavior (what happens when you call a method) is different in each state. We *could* use a conditional in each of the methods, with a case for each possible state. But in OOP we prefer polymorphism over conditionals. This specific use of polymorphism is called the **State Design Pattern**. It involves a class hierarchy, with a subclass for each state.

```
public interface State {
    public Player onPlayPause(Player player);
    public Player onStop(Player player);
    public String getDisplay();
}
```

Now each state (each subclass of State) can handle the buttons differently. Here is the implementation of the playing state (the Playing subtype of State):



```
public record Playing() implements State {
    public Player onPlayPause(Player player) {
        return player.changeState(new Paused());
    }
    public Player onStop(Player player) {
        return player.changeState(new Stopped());
    }
    public String getDisplay() {
        return "Paused. Press ▶⏸ to pause, ⏹ to stop.";
    }
}
```

What does this code mean? When the media player is in the playing state, pressing the ▶⏸ button pauses it (transitions the player into the paused state), and pressing the ⏹ button stops it (transitions the player into the stopped state).

Write the code for the Paused and Stopped implementations of State. Look at the diagram of the state machine on the prior page:

```
public record Paused
}
public record Stopped
}
```

We left out a method from our Player class. That method is called to transition to a different state. Write that method:

```
public record Player(State state) {
    public Player changeState
```