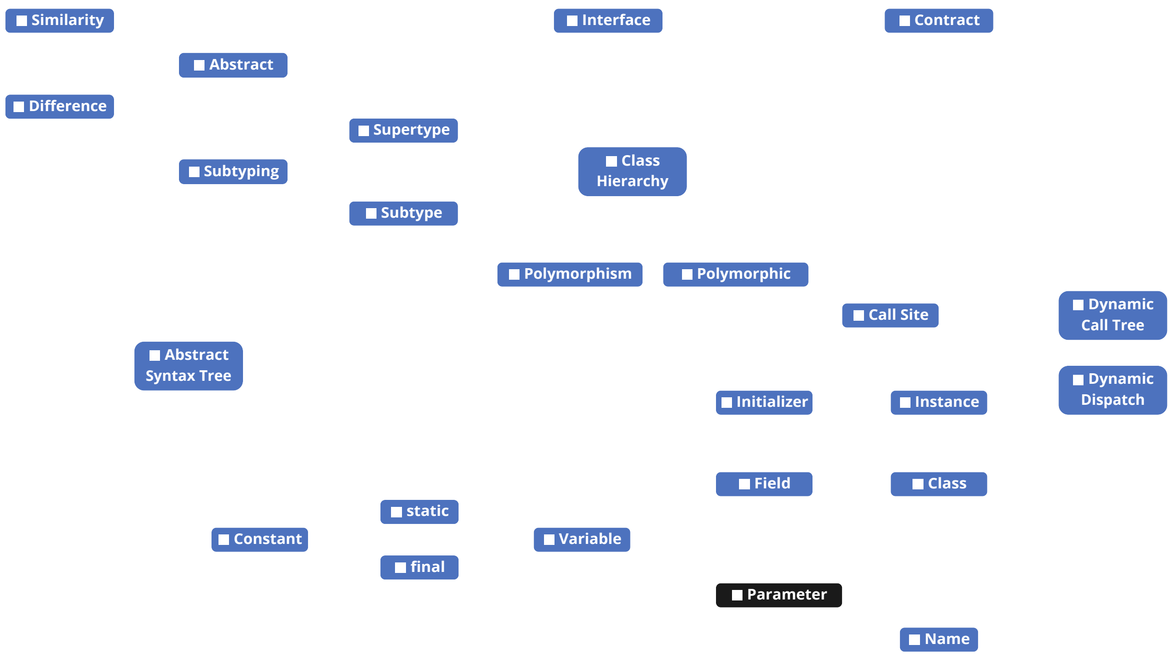
Interfaces, Subtyping, Polymorphism

Student name:	TA signature:

Based on photo by David Troeger on Unsplash

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

Add the following three concepts to the map: Local, Monomorphic, and Method. Connect them and connect everything else as well.



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

Interfaces, Subtyping, & Polymorphism

Let's model geometric shapes: squares and circles. Let's create a **record class** for each kind of shape. A `Square` is defined by its `side` length, a `Circle` by its `radius`.

```
public record Square(double side) { }
public record Circle(double radius) { }
```

We want to compute the area and the perimeter. **Complete the code** below:

```
public record Square(double side) {
  public double area() {
    return this.side() * this.side();
  }
  public double perimeter() {
    return
  }
}
public record Circle(double radius) {
  public double area() {
    return this.radius() * this.radius() * Math.PI;
  }



}
```

Describe the similarities and differences between the two record classes:

Now, given a `Square` or a `Circle`, we want to describe their geometric measures. **Complete the code** below:

```
public static String info(Square square) {
  return square.area() + ", " + square.perimeter();
}
public static String info(Circle circle) {
  return
}
```

Describe the **similarities** and **differences** between the two `info` methods:

We already know that we can abstract by turning differences into **parameters**. Can we rewrite the code so one single `info` method can work for both types of shapes? Maybe by introducing a **type parameter**? **Try by completing the code** below:

```
public static <T> String info(T shape) {
  return
}
```

What's the problem?

We have a challenge: if we use a type parameter (`T`) instead of the concrete type (`Square`, `Circle`), when we want to implement the method body, we do not know the type of the shape, and so we do not know **which methods** it offers. We just know `shape` is of type `T`, and `T` could be *any* type. Parametric types are not enough.

This is where subtyping comes in! It allows us to focus on the **similarities** between some types. We can introduce a supertype of a bunch of similar types. In Java, a good way to do that is by declaring an interface. An interface specifies the common "interface" (a set of methods) of some types.

Remembering the similarities between `Square` and `Circle`, the common methods are `area` and `perimeter`. We can specify the interface like this:

```
public interface Shape {
  public double area();
  public double perimeter();
}
```

It's **important to pick a good name**, so that we can say "a `Square` is an XXX" and "a `Circle` is an XXX". We also want to be able to say "an XXX has an `area`" and "an XXX has a `perimeter`". Picking the name `Shape` sounds reasonable.

An interface declares the methods, but it **does *not* implement** them. An interface is a contract. You can say "if I have a `Shape`, I can call its `area` method".

Now we can rewrite our `info` method. **Complete the code** below:

```
public static String info(Shape shape) {
  return
}
```

However, we are not yet done. If we wanted to call the `info` method, our call would **not compile**:

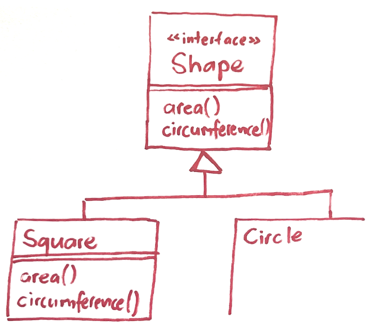
```
info(new Circle(100))
```

Explain what kind of compiler error this code triggers, and why:

The types `Circle` and `Shape` are not connected. We want to connect the two types. What can we say about the relationship we want between them?

- ☐ a Circle is a Shape
- ☐ type Circle is a subtype of type Shape
- ☐ class Circle implements the Shape interface

Complete the class diagram. It shows the interface, its methods, the two classes, their methods, and the subtype relationship. Subtypes point to their supertype with a wide, triangular arrow.



Now we're ready to put everything together. We connect the subtype to its supertype by adding `implements SuperType`, as follows:

```
public interface Shape {
  public double area();
  public double perimeter();
}

public record Square(double side) implements Shape {
  public double area() {
    return this.side() * this.side();
  }
  public double perimeter() {
    return 4 * this.side();
  }
}
```

With this information, the compiler, when it compiles class `Square`, checks that `Square` indeed implements an `area` method that has no parameters and returns a `double`, and a `perimeter` method as promised in the `Shape` interface.

If we do the same for `Circle`, then we can use one info method to work with circles and squares (and other possible subtypes of `Shape`!):

```
public static String info(Shape shape) {
  return shape.area() + ", " + shape.perimeter();
}
public static String demo() {
  return info(new Circle(100)) + " and " + info(new Square(20));
}
```

Did you realize what just happened??? Something pretty crazy: polymorphism!

Draw the **dynamic call tree** for the call to `demo()`:

How **many** possible targets does the `shape.area()` call site in method `info` have?

Draw the expression trees of the expression in the `return`-statement of `info`:

Polymorphic: Dynamic Dispatch, Monomorphic: Static Dispatch

The two highlighted call sites inside the info method above were two examples of polymorphic call sites—call sites with **multiple** possible call target ("poly"). When such a call site executes, at runtime, Java needs to decide **which** of the possible methods to call. It does that based on the object on which the method is called:

- When `shape` refers to a `Circle` object, `shape.area()` calls `Circle.area`.
- When `shape` refers to a `Square` object, `shape.area()` calls `Square.area`.

This decision-making process is called "dispatching". What happens at a polymorphic call site is a dynamic dispatch – the decision is made **at runtime**, **each time** the call site executes (and thus, dynamically). Polymorphic method calls are also known as "virtual method calls".

A different situation arises for **static method calls**, like `Colors.rgb(0, 64, 128)`. There is **no** object involved. `Colors` is a class. And `rgb` is a static method in that class. This is a static call site, also called a monomorphic call site—a call site with only one possible call target ("mono"). Java dispatches the call statically (determine the call target once and for all at compile-time).

Complete the table below:

Call Site	Dispatch	Bytecode	Call	Call Targets
monomorphic	static	invokestatic	`Colors.rgb(1, 2, 3)`	
polymorphic	dynamic	invokevirtual	`shape.area()`	

An Object-Oriented Boolean Type

Java provides the built-in type `boolean`, and a wrapper class named `Boolean`. Let's use our new understanding of subtyping and polymorphism to model the idea of a Boolean in a nice, object-oriented way, using a class hierarchy (supertype and subtypes):

```
public interface Bool {
  public Bool not();
  public Bool and(Bool other);
  public Bool or(Bool other);
}
```

Now let's implement the behavior of the three methods. What does `not` do? Well, **it depends** on whether we call `not` on `TRUE` or on `FALSE`. On a `TRUE` object, `not` will return a `FALSE` object. But on a `FALSE` object, `not` will return a `TRUE` object.

We can implement this with a conditional (like we did previously). But now that we have polymorphism, we can use **dynamic dispatch instead of conditionals**!

Complete class `True`, a subtype of `Bool`:

```
public record True() implements Bool {
  public Bool not() { return
  public Bool and(Bool other) { return
  public Bool or(Bool other) { return
}
```

Now implement class `False`:

```
public record
```

Given these classes, what happens when you execute this expression?

```
new True().not().not()
```

First, we allocate an object of type `True`. Then we call `not` on that object, which returns an object of type `False`. Then we call `not` on that object, which returns an object of type `True`. So, this expression is equivalent to `!!true` in primitive Java.

Draw the expression tree and annotate each node with its type for:
`new False().not().and(new True()).or(new False().not()).not()`

This may look a bit crazy, and you would not use this code instead of the equivalent code with primitive booleans. However, this is how fully object-oriented languages like **Smalltalk** or **Self**, where ***everything* is an object**, work!

Why? Because the above class hierarchy allows us to eliminate conditionals completely! No conditional expressions, no if-statements. Just objects! And turtles.

How could this possibly work? We will get all the way in a subsequent workbook.

Now **summarize our design with a class diagram** for `Bool`, `True`, and `False` (include all methods):

Defining and Using Constants: Final Static Fields

We have given names to all kinds of things:

- **packages**
- **classes** (normal classes, record classes, interfaces)
- **methods** (instance methods and class methods)
- **type parameters** (often single-letter names like `S` or `T`)
- **method parameters**
- **record components**

We also already heard about something else: a field. We have **not** yet *explicitly* declared fields, but we have done so *implicitly*. Like there are instance methods and class methods, there also are instance fields and class fields.

When we specify a…	…this generates…
record component	

Let's first declare our own class fields (static fields). Specifically, let's declare final static fields (also known as constants).

In the past we sometimes wrote pure **nullary methods** like `fontName` or `fontColor`:

```
public class Demo {
  public static String fontName() {
    return "Helvetica";
  }
  public static Color fontColor() {
    return rgb(200, 165, 73);
  }
  public static Graphic label(String content, double size) {
    return text(content, fontName(), size, fontColor());
  }
}
```

Whenever we call these methods, they do a little bit of work and then return the exact same value. **Every. Single. Time.** Creating such methods was a nice way to give a name to such a value. This way, whenever we needed that value in our code, we could call the nicely named method. And if we ever wanted to reconfigure our program (e.g., to change the font color), we could simply edit that one method, and all the rest of the program would make use of the new value.

Defining such methods is ok, but **there is a simpler way**. We can define **constants**:

```
public class Demo {
  public static final String FONT_NAME = "Helvetica";
  public static final Color FONT_COLOR = rgb(200, 165, 73);
  public static Graphic label(String content, double size) {
    return text(content, FONT_NAME, size, FONT_COLOR);
  }
}
```

What are advantages of using constants instead of pure nullary methods?

Constants are usually defined as `public` (accessible from everywhere), `static` (class field, there's only one, not one in every instance), and `final` (value cannot be changed once it is initialized).

You already used constants before:

Name	Declaration – you did not (need to) see those
`Colors.RED`	`public static final Color RED = rgb(255, 0, 0);`
`Points.CENTER`	`public static final Point CENTER = new Point(…);`
`Math.PI`	`public static final double PI = 3.141592653589793;`

Refactor the following code to use constants:

```
public class Example {
  public static double width() {
    return 200;
  }
  public static double height() {
    return 100;
  }
  public static Color backgroundColor() {
    return rgb(200, 100, 0);
  }
  public static Graphic background() {
    return rectangle(width(), height(), backgroundColor());
  }
}
```

```
public class Example {

  public static








}
```

When initializing a constant, you provide its type, its name, an equals sign, and an initializer (an **expression** that computes its value).

☐ The initializer expression can use other constants

Do the following classes compile? If not, what is the problem?

`public class Forward {`
`  public static final int A = B;`
`  public static final int B = 1;`
`}`
	`public class Cyclic {`
	`  public static final int A = B;`
	`  public static final int B = A;`
	`}`

☐ compiles ☐ does not compile	☐ compiles ☐ does not compile

Static field declarations are **evaluated in order, from top to bottom**. If an initializer of a constant uses another constant that has not been declared yet, the compiler reports an error.

Another Example of Subtyping

Let's assume we want to compute carbon emissions. We model travel as trips. There are different types of trips: train and car. For any given trip we can determine its carbon emissions (in g CO_2).

```
public interface Trip {
  public double carbonEmissions();
}
public record CarTrip(double km, double lPerKm) ______________ {
  public double carbonEmissions() {
    return this.km() * this.lPerKm() * 2700; // wild guess of 2700 g/lt
  }
}
public record TrainTrip(double km, boolean busyTrain) ______________ {
  public double carbonEmissions() {
    return this.km() * (this.busyTrain() ? 8 : 80); // wild guess of g/km
  }
}
```

Implement the following method to compute the total emissions of the given trips:

```
public static double totalEmissions(Sequence<Trip> trips) {
  return



}
```

Now we want to use our method as follows:

`public static double demo() {`
`  return totalEmissions(of(`
`    new CarTrip(2, 0.06),`
`    new TrainTrip(220, false),`
`    new TrainTrip(15, true)`
`  ));`
`)`
	If you try to compile this, **what error do you get?**

Fix the error by completing the code above. What's the result of invoking `demo` after fixing the code?

Run the code in your IDE and write the result here:

Defining and Using Local "Constants" – Final Local Variables

If a method contains a common subexpression (a "code clone"), you want to extract it. Here is a bad implementation of a function to produce a pair of eyes:

```
public static Graphic eyes(double diameter) {
    return beside(
        overlay(
            Toolbelt.circle(diameter * 0.5, BLACK),
            Toolbelt.circle(diameter, WHITE)
        ),
        overlay(
            Toolbelt.circle(diameter * 0.5, BLACK),
            Toolbelt.circle(diameter, WHITE)
        )
    );
}
```

This code is correct but horrible! There is a lot of **redundant source code**. We know we should extract common subexpressions into separate methods, as follows:

```
public static Graphic eye(double diameter) {
    return overlay(
        Toolbelt.circle(diameter * 0.5, BLACK),
        Toolbelt.circle(diameter, WHITE)
    );
}

public static Graphic eyes(double diameter) {
    return beside(
        eye(diameter),
        eye(diameter)
    );
}
```

This is much cleaner! However, as we can see when we draw the dynamic call tree of a call like `eyes(100)`, there is a lot of **redundant computation** going on (we call `eye` twice, with the exact same argument). Wouldn't it be nice if we could compute one eye, and then reuse the same eye twice, instead of computing two eyes?

Here is a way to make that happen:

```
public static Graphic eyes(double diameter) {
    final Graphic theEye = eye(diameter);
    return beside(
        theEye,
        theEye
    );
}
```

We defined a final local variable, named `theEye`, with type `Graphic`, and initialized it to the result of computing an eye with the given diameter. Then we use `theEye`, twice. We compute once. And we reuse the result as many times as we want!

☐ We could use a "global" constant (a static final field) instead

Composite Data Structures

Let's build our own **little language** for arithmetic expressions like `1 + 3 / 2`.

In programming languages we model programs as **trees**. We call them AST – abstract syntax tree. You already know expression trees. Expression trees are ASTs of expressions. A program consists of more than just expressions, and ASTs model the complete program (including parts like "public class X" or "package game;").

In our little language we only need to deal with expressions, and only with very simple ones. We support integer literals, addition, subtraction, multiplication, and division.

We model all the nodes of the expression tree as objects of some type `Node`. But there are different subtypes of node: one type of node to represent additions, another type to represent subtractions, ... But all subtypes of node have some common feature: given a node object, you can ask it for its **value**. You can **evaluate** it.

Let's specify the common feature in an interface. Here is an interface that specifies what ***any*** type of expression tree node must be able to do:

```
public interface Node {
  public int evaluate();
}
```

An interface is a special kind of class (like a **record**). It cannot be instantiated: we cannot create an object of type `Node`. An interface specifies a contract by listing a bunch of methods, without implementing them. We can then create other classes that "implement" that contract, by implementing all the listed methods.

Let's create a subtype of `Node` to represent literal values, like 1, 2, or 3:

```
public record Lit(int value) implements Node {
  public int evaluate() {
    return this.value();
  }
}
```

Let's create another subtype of `Node`, which represents an addition. This should be a record class with two components: the two child `Node`s. **Implement** its `evaluate` method (it should call `evaluate` on each child `Node`, and then add the results):

```
public record Add(Node left, Node right) implements Node {




}
```

The `Sub`, `Mul`, and `Div` classes are similar. What is **similar** and what is **different** between the four classes?

Let's use the above `Node` class hierarchy to represent arithmetic expressions:

Arithmetic Expression	Code to Construct an AST

`1 + 3 / 2`	`new Add(`
	`  new Lit(1),`
	`  new Div(new Lit(3), new Lit(2))`
	`)`

`1 - 2 - 3`	
`(1 - 2) - 3`	
`1 - (2 - 3)`	

Now that we can compose an AST of an expression, let's use it to **compute the value** of the expression (i.e., to evaluate the expression).

```
public class Demo {
  public static int compute() {
    return new Add(
      new Lit(1),
      new Div(new Lit(3), new Lit(2))
    ).evaluate();
  }
}
```

To understand how the evaluation proceeds, let's draw a dynamic call tree of `Demo.compute()`.

Constructor calls: show class & constructor name, e.g. `MyClass.MyClass(…)`
Static method calls: show class & method name, e.g., `MyClass.m(…)`
Instance method calls: show value/object & method name, e.g., `"A".charAt(…)`
For each call, draw the **argument values/objects for all parameters (if any).**
Above each call tree node, draw the value it returns.