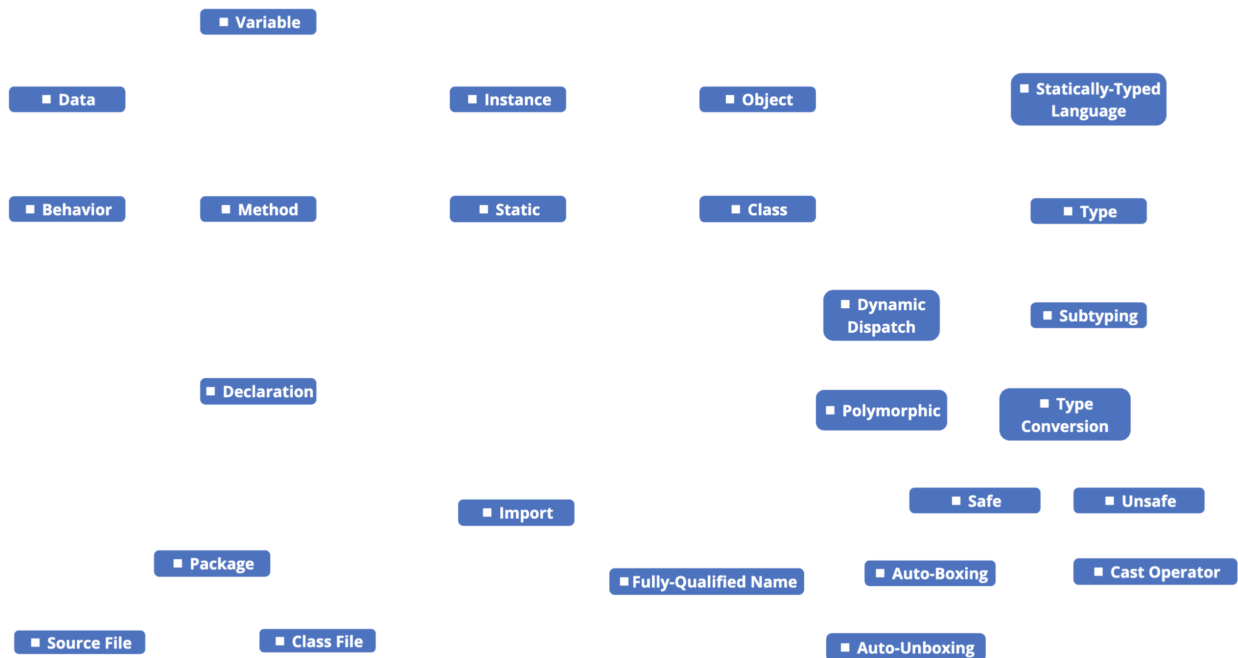
Instance Fields, Methods, Type Casts

Student name:	TA signature:

Object = Data + Behavior

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

Add the following three concepts to the map: Field, Call, and Widening. 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

Java Classes & Packages, Source Files, Class Files, Directories

In Java each class usually is stored in a separate file. For example, a class named `Pacman` is stored in a file with the name **Pacman.java**. This is the **source file**, the file containing the source code. Once you compile the class (by clicking some button in the IDE, or by running **javac Pacman.java** on the command line), the Java compiler creates a file containing the compiled bytecode of your class (e.g., file **Pacman.class**). This is the **class file**.

Complete the table:

Class Name	Source File Name	Class File Name
Tile		
	WorldState.java	

Packages

In Java, classes can be organized into **packages**. So far, the classes we wrote in workbooks were in the "unnamed package", the classes we wrote in labs were in package `lab`, and the JTamaro classes we used were in package `jtamaro.en`. A package in the Java language corresponds to a **directory** in the file system. Given that we have a package `jtamaro.en`, there must be a corresponding directory. Indeed, there is a directory **jtamaro/en**, hidden away somewhere.

To put a class into a package, we have to:

- Save the source/class file in the corresponding directory
- Put a **package declaration** such as `package lab;` at the start of the source file

To refer to class `Pacman` that is inside a package `game`, we can use the **fully-qualified class name**, `game.Pacman`. There will be a file **Pacman.java** inside the directory **game**.

Complete the table:

Package Name	Unqualified Class Name	Fully-Qualified Class Name	Directory + Source File Name
-	World	World	World.java
lab	Editor	lab.Editor	lab/Editor.java
jtamaro.en	Graphic		

What is the package declaration at the top of JTamaro's **Graphic.java**?

Import

We have several options to **refer to a class located in a named package**:

- If we are located in the **same** package as the class, we can just use the unqualified class name (e.g., `Pacman`)
- We can use the fully-qualified class name (e.g., `game.Pacman`)
- **Preferred way:** We can **import** the class (e.g., `import game.Pacman;`) and then use the unqualified class name (e.g., `Pacman`)
- We can import all classes of the package (e.g., `import game.*;`) and then use the unqualified class name (e.g., `Pacman`)

Write an **import declaration** to import all classes in package `jtamaro.en`:

Review: Structure, Type, and Value of Expressions

```
public static boolean lt(double v1, double v2) {
  return v1 < v2;
}

public static String compare(int a, int b) {
  return a > b ? "greater than" : (lt(a, b) ? "less than" : "equal to");
}
```

Draw the expression tree for the highlighted expression, give the **types** for each node, and show the **values** for each node that is evaluated when the method is called with `compare(1, 1)`:

Draw the expression tree again, give the **types** for each node, and show the **values** for each node that is evaluated when the method is called with `compare(1, 0)`:

Which of the following claims are true?

- ❑ The **structure and node contents** of the expression tree is the same, no matter whether and in what state the method might get called.
- ❑ The **types** of each node in the expression tree are the same, no matter whether and in what state the method might get called.
- ❑ The **values** of each node in the expression tree are the same, no matter whether and in what state the method might be called.

In **statically-typed languages** like Java (or C, C#, C++, Scala, Haskell, ...) the types are determined statically, that means, when the code is compiled, **before** it is executed, **before** we know the values the parameters will have.

Type Conversion

Java provides different **type conversion** approaches. It **auto-boxes** primitive types into wrapper types, and it **auto-unboxes** wrapper types into primitive types. Moreover, it automatically **widens** "narrower" numerical types into "wider" types, e.g., it accepts an expression of type `int` in a hole of type `double`, so we can write expressions like `rectangle(100, 50.5, RED)`, even though `100` is of type `int`. Java also provides **methods** to **convert** from Strings to some other types (e.g., `Integer.parseInt`), or from other types to Strings (e.g., `Integer.toString`).

Besides the above ways to convert between types, Java also provides **cast operators** (which are part of the Java language). For example, the following expression casts the `double` value `3.14` to type `int`:

```
(int) 3.14
```

When casting a value to a different type, some information may get lost. In the example above, the expression evaluates to the `int` value `3`. The digits after the comma are lost.

The cast operators are unary operators. They are written before their operand.

Operator	Type Signature	Description
`(int)`	`double` ⟶ `int`	Cast double to int
	`char` ⟶ `int`	Cast char to int
`(double)`	`int` ⟶ `double`	Cast int to double
	`char` ⟶ `double`	Cast char to double
`(char)`	`int` ⟶ `char`	Cast int to char
	`double` ⟶ `char`	Cast double to char

What are the results of the expressions `(int)3.14*3.14` and `(double)(int)3.14`?

Draw two **expression trees**, each node annotated with its **type** and **value**:

Which of the following claims are true?

- ❏ In Java, you cannot cast from a `boolean` to other primitive types.
- ❏ In Java, you cannot cast to a `boolean` from other primitive types.

Type Safety

The purpose of statically-typed languages is to guarantee **type safety**: once the compiler type checked your program, no type incompatibilities can happen at runtime. You are **safe**.

But these languages may also provide ways to circumvent these guarantees.

One example is the cast operator in Java. Some cast operations are **unsafe**: they will lead to an error at runtime caused by types that are incompatible but were not rejected at compile time. This is why casts should be **avoided**. If you are tempted to introduce a cast, there is probably a better design that eschews it.

Review: Instance Fields and Instance Methods

When we use the term **field**, what we usually mean is **instance field**. An instance field is a field inside an **object** (instance).

To deconstruct a record, that is, to access its **instance fields**, we use its **instance methods**. An instance method works on an object and has access to the instance fields of that specific object.

We already saw this:

```
public static double dotProduct(Vector a, Vector b) {
  return a.x() * b.x() + a.y() * b.y();
}
```

We could read `a.x()` as "Hey, a! What's your x?" The expression `a` evaluates to the object on which method `x` will work.

In OOP lingo we say that `a.x()` "invokes method `x` on object `a`". And method `x` will return the value of the given object's `x` field.

In general, to invoke an instance method, we write an expression producing an object, a dot, the name of the method, and the argument list in parenthesis:

```
some expression producing an object.method()
```

Watch out: Invoking an **instance method** looks syntactically very similar but is **fundamentally different** from invoking a **static method**. A static method does not work on a specific object. When we invoke a static method, we just write the name of the class in which the method is defined before the dot:

```
Graphics.emptyGraphic()
```

Until quite recently we did **not** use instance methods. All our methods were static methods, like `Graphics.emptyGraphic()`.

Instance Fields and Instance Methods of Record Classes

Any record has an instance method for each instance field. That method returns the value of the corresponding instance field. It has the same name as the field, has no parameters, and has a return type that is the same as the type of the field.

Complete this table of instance fields and instance methods:

Record (Class)	Instance fields	Instance methods
`Vector`	`x` `y`	
`Interval`	`start` `end`	
`Odds`		

Class Methods or Instance Methods

So far most of our methods were **static methods**. A synonym of "static methods" is **class methods**. We also saw a different kind of method: **instance methods**.

Here is an example of a class `Calculator` with a **class method** `add`:

```
public class Calculator {
  public static int add(int a, int b) {
    return a + b;
  }
}
```

And here is how we can call method `add`:

```
Calculator.add(1, 2)
```

Here is an example of a record class `Vector`, which automatically gets an **instance method** `x` (the Java compiler generates an instance method for each component of the record):

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

And here is how we can call method `x`:

```
new Vector(1, 2).x()
```

You can define class methods and instance methods in normal classes or in record classes. **Complete** the table:

<table>
<tr><td><pre>class C {
  public static int cm() {
    return 1;
  }
  public int im() {
    return 1;
  }
}</pre></td><td><pre>record R() {
  public static int cm() {
    return 1;
  }
  public int im() {
    return 1;
  }
}</pre></td></tr>
<tr><td><code>C.cm()</code></td><td></td></tr>
<tr><td><code>new C().im()</code></td><td></td></tr>
</table>

When you call a **class method**, there is ***no*** object involved. A class method is like a function in Racket BSL, or in Python. The only difference is that it is nested within a class, and you may need to put the class name and a dot in front of the method name to call it.

When you call an **instance method**, there must be an object involved. You invoke the method "on an object". It is like a request you make to the object.

Why would you ever want to create an instance method? You would do so when the object has some fields (instance variables). The instance method can access the fields of the object. When you use record classes, you can call the generated methods to access the fields corresponding to the components of the record.

OOP: Combining Data (Fields) and Behavior (Methods)

The probably most important aspect of object-oriented programming is that we can use objects to combine **data** and **behavior**. The data is stored in the **instance variables** of objects (e.g., the fields of records), and the behavior is provided by **methods** which are somehow related to these objects. The methods may be class methods or instance methods.

Let's implement the **exact same** functionality using class methods and using instance methods, to compare the two ways.

Let's model a class representing **left-closed intervals**, like [5,9) for 5, 6, 7, 8. It should provide functionality to represent an interval as a string, to compare two intervals for equality, and to compute the hull of two intervals. The **data** is stored in objects of class `Interval`. Those objects contain two instance variables: `start` and `end`. The **behavior** is implemented by three methods of class `Interval`.

Complete this solution with **class methods**:

```
public record Interval(int start, int end) {
  public static String asString(Interval i) {
    return "[" + String.valueOf(i.start()) +
           "," + String.valueOf(i.end()) + ")";
  }
  public static boolean equalTo(Interval a, Interval b) {
    return
  }
  public static Interval hull(Interval a, Interval b) {
    return new Interval(
      Math.min(a.start(), b.start()),
      Math.max(a.end(), b.end())
    );
  }
}
```

Complete this solution with **instance methods**:

```
public record Interval(int start, int end) {
  public String asString() {
    return "[" + String.valueOf(this.start()) +
           "," + String.valueOf(this.end()) + ")";
  }
  public boolean equalTo(Interval other) {
    return this.start() == other.start() && this.end() == other.end();
  }
  public Interval hull(Interval other) {
    return




  }
}
```

What is **different** between the two styles of implementation?

Let's assume there was some mathematical operator, say ⊜, such that `a ⊜ b` means "hull of the two intervals a and b". In that case we could write the following to compute the hull of the two intervals [0,4) and [7,9)—the result would be [0,9):

```
[0,4) ⊜ [7,9)
```

Many programming languages, Java included, do not allow us to introduce arbitrary new operators (like ⊜). Many languages, Java included, do not even allow us to redefine *existing* operators (like +). So, we cannot introduce a ⊜ operator, and we cannot even redefine Java's + operator to compute the hull of two `Interval` objects. If we want to provide our own behavior, we have to define functions, procedures, or methods (with names like `hull`), not operators (with symbols like ⊜).

Let's write the expression `[0,4) ⊜ [7,9)` in the two styles (with class methods and with instance methods). **Draw** the corresponding expression trees:

With static methods	With instance methods

`Interval.hull(`
`  new Interval(0, 4),`
`  new Interval(7, 9)`
`)`
	`new Interval(0, 4)`
	`  .hull(new Interval(7, 9))`

In the left style, we call a **class method** `hull` and provide the two intervals as arguments. In the right style we call the **instance method** `hull` on the first `Interval` object, and we pass the second `Interval` object as an argument.

In both cases, the method `hull` has access to the two `Interval` objects. Check the implementation of the two methods on the previous page: The class method can access the two intervals via its two parameters, `a` and `b`. The instance method can access the first interval via the "magic" parameter `this` and the second interval using its parameter `other`.

The `this` variable in an instance method always refers to the object the method is invoked on: the value of the expression in front of the dot in the method call, e.g., the value of `x` in `x.hull(y)`, or the value of `new Interval(0, 4)` in `new Interval(0, 4).hull(y)`.

The value of `this` becomes clear when you draw the method invocation with an expression tree. It's the value coming into the hole in front of the dot:

Static method invocation	Instance method invocation
Interval.hull(□,□) a inside hull method b inside hull method	□.hull(□) will be known as this inside hull method will be known as other inside hull method

Now, let's write a Java method that we can use to test whether hull is **associative**:

```
(a ⊜ b) ⊜ c   =   a ⊜ (b ⊜ c)
```

Using **static methods** (use `equalTo` to compare):

```
public static boolean yes(Interval a, Interval b, Interval c) {

  return
}
```

Using **instance methods** (use `equalTo` to compare):

```
public static boolean yes(Interval a, Interval b, Interval c) {

  return
}
```

Using the static method style, binary operations, i.e., methods that take two arguments, look more symmetric, while using the instance method style, one of the two operands has to be chosen to be the object we ask to perform the operation (on it*self* and the given other object):

Static method invocation	Instance method invocation
`hull(a, b)`	`a.hull(b)`
`equalTo(a, b)`	`a.equalTo(b)`

This asymmetry for operations that work on multiple objects is a quirk of mainstream object-oriented programming languages (there are some experimental languages, like **Cecil**, that allow symmetric "multi-methods").

Instance method calls are more powerful than class method calls

We will soon learn about **subtyping**, and a powerful feature enabled by subtyping: **polymorphic method calls** (also known as **dynamic dispatch**). Dispatching means deciding **which** method to call. Dispatching can happen at **compile time** ("Come on! It's clear that at this call site we will always call the `hull` method in class `Interval`!") or at **runtime** ("Hmm. At this call site we might call the `hull` method in class `Interval`, or the `hull` method in class `FastInterval`. Let's wait until runtime, and decide each time we call which of those methods to actually execute.").

Dynamic dispatch is called "dynamic", because the method implementation that will actually execute will be looked up **at runtime** (not statically, at compile time). In Java, dynamic dispatch happens at all **instance method invocations**.

There is no dynamic dispatch at **static method invocations**. That's why they are called "static", because it is already 100% clear at compile time which method implementation is going to execute.

With dynamic dispatch, the strange, asymmetric, notation for instance method calls, `a.m(b)`, with the hole for the implicit "this" parameter being placed in front of the dot instead of in the parameter list, will start to make some sense. The "this" parameter is very special, because its value provides the information needed to decide which exact method should actually be called at runtime.