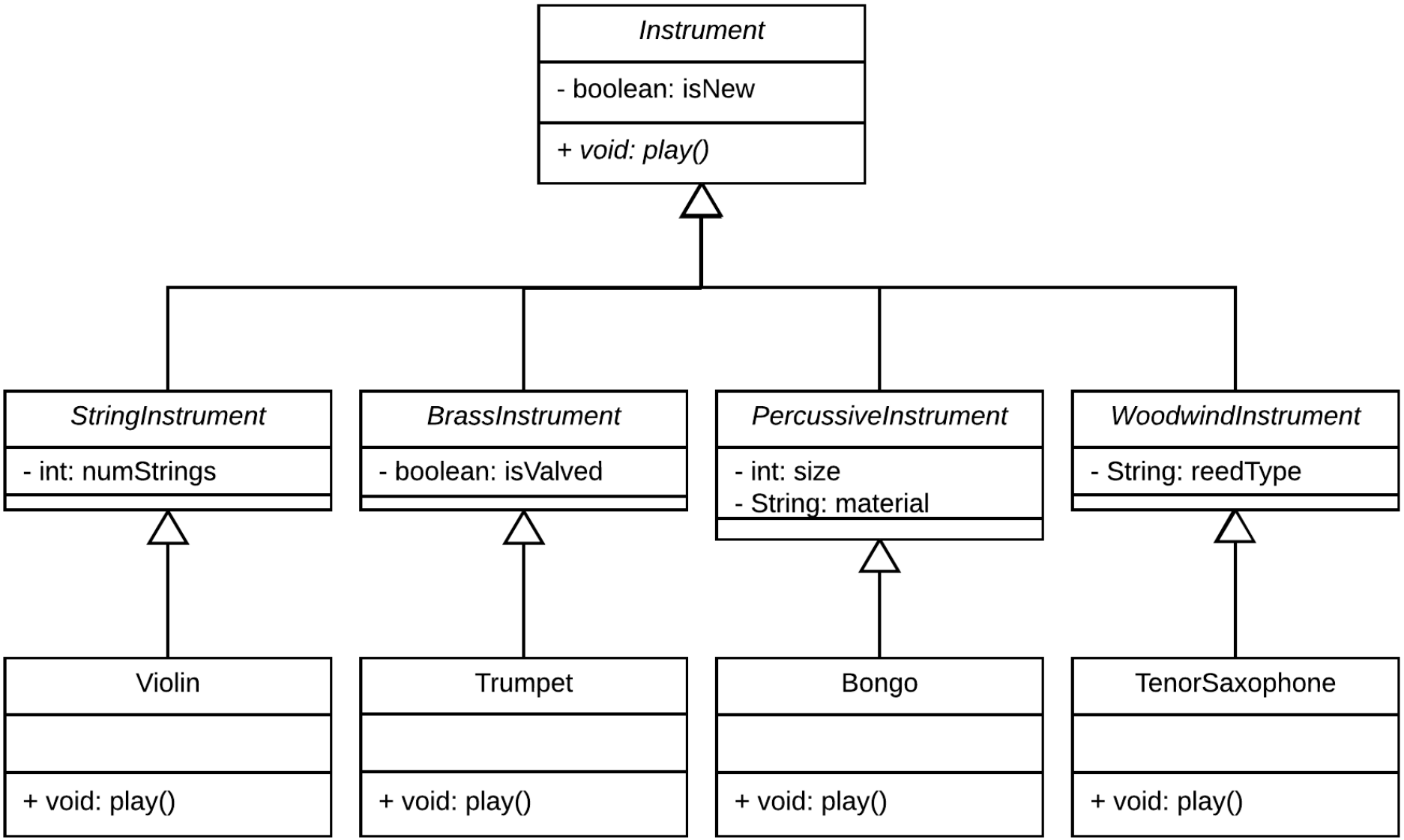
Assess Yourself

As a music store owner, you want to create an inventory so that your less than knowledgeable customers can better understand how to select a musical instrument.

Instruments can be new or used, and produce their own unique sound. Stringed instruments like the cello and harp have some number of strings; brass instruments like the trumpet and tuba can be valve or slide controlled; percussive instruments like a cymbal are defined by the material they're made of and their size; woodwind instruments like the flute and saxophone have a particular reed type.

Design classes, with attributes and methods, that appropriately represent this scenario.

Assess Yourself



SWEN20003
Object Oriented Software Development

Polymorphism and Interfaces

Semester 1, 2019

The Road So Far

- OOP and Java Foundations
- Classes and Objects
- Abstraction
  - Inheritance
  - Abstract Classes
  - Polymorphism
- Software Tools

Lecture Objectives

After this lecture you will be able to:

- Describe the purpose and use of an **interface**
- Describe what it means for a class to **use** an interface
- Describe when it is appropriate to use inheritance vs. interfaces
- Use interfaces and inheritance to achieve powerful abstractions
- Make any class "sortable"

In-class code found here

Interfaces

Interfaces

Interfaces are **vague**, **distant** relatives of abstract classes:

- Defines an "abstract" entity that can't be instantiated
- Can only contain **constants** and **abstract methods**
- Defines **behaviours**/**actions** that are common across a number of classes

Keyword

Interface: Declares a set of constants and/or methods that define the **behaviour** of an object.

Can Do

- All interfaces represent a "**Can do**" relationship
- Classes that implement an interface *can do* all the actions defined by the interface
- Interface names are generally called `<...>able`, and relate to an action
- For example, classes that implement the `<Drivable>` interface can all be driven, because they implement the `drive()` method

Defining Interfaces

```
public interface Printable {

    int MAXIMUM_PIXEL_DENSITY = 1000;

    void print();

}
```

- Methods **never** have any code
- All methods are *implied* to be `abstract`
- All attributes are *implied* to be `static final`
- All methods and attributes are *implied* to be `public`

Interfaces

Keyword

interface: Defines an *interface*, rather than a class.

Keyword

implements: Declares that a class implements all the functionality expected by an interface.

Implementing Interfaces

```
public class Image implements Printable {
    public void print() {
        <block of code to execute>
    }
}
```

```
public class Spreadsheet implements Printable {
    public void print() {
        <block of code to execute>
    }
}
```

- Concrete classes that *implement* an interface **must** implement **all** methods it defines
- Classes that don't implement all methods must be **abstract**

Default Methods

Classes can be "forced" to have an implementation of a method, that can then be overridden.

```
public interface Printable {
    default void print() {
        System.out.println(this.toString());
    }
}
```

Keyword

default: Indicates a *standard* implementation of a method, that can be overridden if the behaviour doesn't match what is expected of the implementing class.

Assess Yourself

A person can wear many items of clothing and apparel, but each item can go on a different part of a person's body.

Implement a possible interface for this scenario, as well as one or more implementations of the interface's method(s), such that a hypothetical `Person` object can "wear clothes".

Bonus: Why are we using an interface for this?

Assess Yourself

```
public interface Wearable {
    public void wear();
}
```

```
public class Clothing implements Wearable {

    private String bodyPart;

    public Clothing(String bodyPart) {
        this.bodyPart = bodyPart;
    }

    public void wear() {
        System.out.format("%s is worn on your %s.\n",
            this.getClass().getName(), this.bodyPart);
    }
}
```

Assess Yourself

```
public class Seatbelt implements Wearable {

    private Car car;

    private boolean isWorn = false;

    public Seatbelt(Car car) {
        this.car = car;
    }

    public void wear() {
        this.isWorn = true;
        this.car.setCanDrive(true);
    }
}
```

Even though `Clothing` and `Seatbelt` can both be "worn", there is no logical relationship between them; they should not be represented using inheritance!

Extending Interfaces

```
public interface Digitisable extends Printable {
    public void digitise();
}
```

- Interfaces can be extended just like classes
- Forms the same "Is a" relationship
- Used to add additional, specific behaviour

Sorting

What is sorting?

- Arranging things in **an** order

What can we sort?

- Any piece of data

How do we sort?

```
Arrays.sort(arrayOfThings);
```

But... How? How does Java know how to arrange `Robots`? Or `Dogs`?

String

How does Java sort an array of `Strings`? Why?

```
String[] strings = new String[]{"dragon",
                    "Jon Snow", "Game of Thrones"};

System.out.println(Arrays.toString(strings));
Arrays.sort(strings);
System.out.println(Arrays.toString(strings));
```

```
[dragon, Jon Snow, Game of Thrones]
[Game of Thrones, Jon Snow, dragon]
```

String

Class String

java.lang.Object
    java.lang.String

All Implemented Interfaces:

`Serializable, CharSequence, Comparable<String>`

Comparable Interface

A class that implements `Comparable<ClassName>`

- Can (unsurprisingly) be compared with objects of the same class
- Must implement `public int compareTo(<ClassName> object)`
- Can therefore be **sorted** automatically

The general use of `<ClassName>` will be explained in a later lecture, stay tuned

compareTo

How does it work?

- Defines a method allowing us to **order** objects
- Compares **exactly two** objects, A and B
- B can be a *subclass* of A, as long as they are both `Comparable`
- Returns a negative integer, zero, or a positive integer if object A (`this`) is "less than", "equal to", or "greater than" object B (the argument)

```
public int compareTo(String string) {
    return this.length() - string.length();
}
```

Worked Example

Assess Yourself

Write a class called `RandomNumber` that implements the `Comparable` interface.

Each `RandomNumber` object should have a single instance variable called `number`, that is given a random integer when the object is instantiated.

When `RandomNumbers` are sorted, they should appear in **ascending** order, according to the value of `number`.

Comparable Interface Example

```
import java.util.Random;
import java.util.Arrays;

public class RandomNumber implements Comparable<RandomNumber> {

    private static Random random = new Random();

    public final int number;

    public RandomNumber() {
        this.number = random.nextInt(100);
    }

    public int compareTo(RandomNumber randomNumber) {
        return this.number - randomNumber.number;
    }

    public String toString() {
        return Integer.toString(this.number);
    }

}
```

Comparable Interface Example

```
public static void main(String args[]) {
    RandomNumber randomNumbers[] = new RandomNumber[10];

    for (int i = 0; i < randomNumbers.length; i++) {
        randomNumbers[i] = new RandomNumber();
    }

    System.out.println(Arrays.toString(randomNumbers));

    Arrays.sort(randomNumbers);

    System.out.println(Arrays.toString(randomNumbers));
}
```

```
[51, 90, 65, 50, 75, 67, 42, 72, 65, 49]
[42, 49, 50, 51, 65, 65, 67, 72, 75, 90]
```

Next Level Abstraction

```
public class Spreadsheet extends Document implements Printable,
            Colourable, Filterable, Comparable<Spreadsheet> {
    public void print() {
        <block of code to execute>
    }
}
```

- Classes can only *inherit* one class, but can *implement* multiple interfaces
- Inheritance and interfaces work together to build very powerful abstractions that make creating solutions much easier

Multiple Inheritance

- You: "Oh, so we *can* do multiple inheritance in Java!"
- Me: "No, you can't."
- You: "But you just said classes can implement multiple interfaces?"
- Me: "I did. They are not the same thing."
- You: "But..."
- Me: "Totally. Different. Things. Stop talking now."

Inheritance is for generalising **shared properties** between **similar classes**; "is a".
Interfaces are for generalising **shared behaviour** between (potentially) **dissimilar classes**; "can do".

Polymorphism

Inheritance

```
Robot robot = new WingedRobot(...);
```

Interfaces

```
Comparable<Robot> comparable = new Robot(...);
```

Subtype polymorphism applies to interfaces!

Assess Yourself

Interface or Inheritance?

- All `Dogs` can bark.

Both?

Needs more context...

Assess Yourself

Interface or Inheritance?

- All `Animals`, including `Dogs` and `Cats` can make noise.

Inheritance

Assess Yourself

Interface or Inheritance?

- All `Animals` and `Vehicles` can make noise.

Interface

Assess Yourself

Interface or Inheritance?

- All classes can be compared with themselves.

Interface

Assess Yourself

Interface or Inheritance?

- All `Characters` in a game can talk to the `Player`.

Inheritance

Assess Yourself

Interface or Inheritance?

- Some `GameObjects` can move, some can talk, some can be opened, and some can attack.

Interface

Interface or Inheritance?

Inheritance:

- Represents *passing shared information* from a *parent* to a *child*
- Fundamentally an "Is a" relationship; a *child* **is a** *parent*, plus more; hierarchical relationship
- All `Dogs` are `Animals`

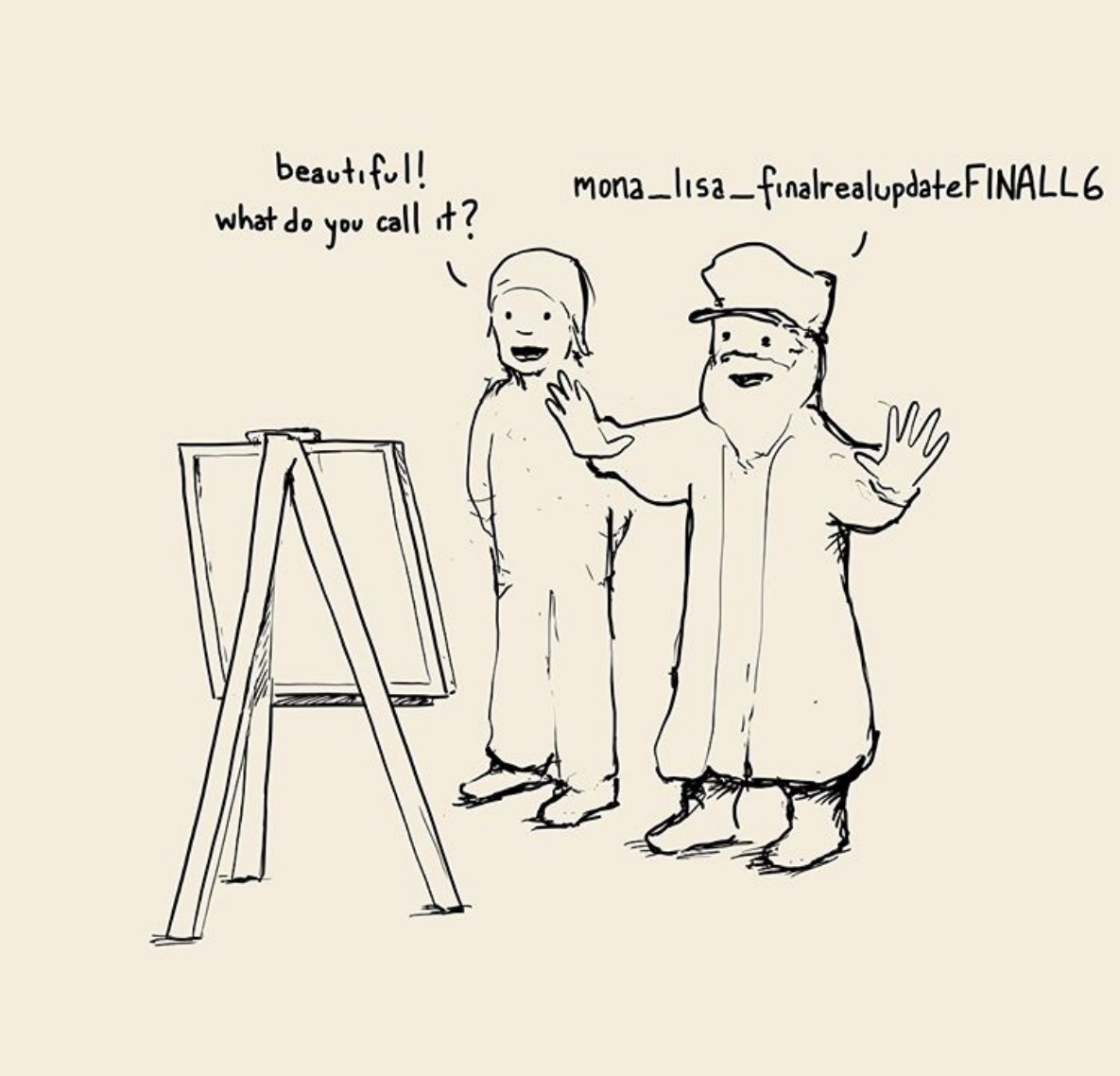
Interface:

- Represents the ability of a *class* to *perform an action*
- Fundamentally a "Can do" relationship; a *Comparable* object can be *compared* when sorting
- `Strings` can be compared and sorted

Meme Submissions

Courtesy of Miro

Meme Submissions



Courtesy of Miro

Metrics

A `Student` is specified by a first and last name, a student ID, and a list of subjects. When `Students` are sorted, they should appear in increasing student number order.

A `Subject` is specified by a name, subject code, and a list of students. When `Subjects` are sorted, they should appear in order of ascending subject code.

A `Course` is specified by a name, a course code, a list of (possible) subjects, and a list of students. When `Courses` are sorted, they should appear in order of ascending course code.

Implement appropriate `compareTo` methods for each class, and implement the `Enrollable` interface such that a `Student` can enrol in both a `Subject` and a `Course`.

Mid-Semester Test

Details

- Date: Tuesday, April 16th
- Time: 5:20pm-6:10pm
- Duration: 10 minutes reading, 40 minutes writing
- Content: Weeks 1-6 (lectures 1-12)

Venues

```
for (Student s: studentList) {
    if (s.getStudentNumber() < 914000) {
        s.setTestVenue("Kwong Lee Dow 122-125");
    } else {
        s.setTestVenue("Redmond Barry Rivette Theatre");
    }
}
```

Questions?

Sample Test