Module 12

- Review of Previous Week
- Classes as types
- Constructors
  - `this`
  - `super`
- Casting
  - `instanceof`
- Polymorphism
- Additional resources

Review of Previous Week

- Access modifiers
- Final keyword
- Inheritance
- Overriding

Classes as types

Constructors

Review: what is a constructor?

Constructors are special methods that match the name (casing matters!) of the class and have no return type. The constructor method run only once, and that is when you "instanciate" the class.

Here's an example

```
class HumanBeing {
  public HumanBeing() {
    System.out.println("A new HumanBeing is being
      created!");
  }
}

public class Main {
  public static void main(String[] args) {
    // Notice how I never explicitly call the class
    // constructor. Classes are instanciated using
    // the `new` keyword. When I do this, one of the
    // `HumanBeing` class constructors runs. I
    // should see "A new HumanBeing is being
    // created!" printed to the screen since that
    // code runs in the constructor for the HumanBeing
    // class.
    HumanBeing me = new HumanBeing();
  }
}
```

Remember

In order for a constructor to be valid, it must:

1. Have the exact name as the class. Case matters.
2. It cannot have a return type.

Question

Who can tell me (1) if the classes below have valid constructors and (2) if they are not, why they are not valid.

```
public class Cat {
  public cat() {
    System.out.println("meow");
  }
}

public class Dog {
  public void Dog() {
    System.out.println("woff");
  }
}
```

Answer

```java
public class Cat {
  // The method below is not a constructor because the
  // class is named `Cat` this method is named `cat`.
  // One has an upper case C and the other has a lower
  // case c, and so they are different.
  public cat() {
    System.out.println("meow");
  }
}

public class Dog {
  // The method below is not a constructor because it
  // has a return type, `void` in thie case. Constructor
  // cannot have return types, and so this is not a
  // constructor.
  public void Dog() {
    System.out.println("woff");
  }
}
```

Can I have multiple constructors?

Yes! But why would I want that? Well, let's imagine a scenario where we might allow a user of our code (or us using our own code) to create a class and instanciate it with different sets of data? If we find we're in this situation, we can create multiple constructors in the same class with the help of a feature called "overloading."

New Terminology!

Overloading: Overloading allows me to create the same method in a class, but give each of those methods different parameters and different types for those parameters.

Here's an example

```
class HumanBeing {
  protected String name;
  protected int age;

  public HumanBeing() {
    this.name = "Unknown Person";
    this.age = 0;
  }

  public HumanBeing(String name) {
    this.name = name;
    this.age = 0;
  }

  public HumanBeing(String name, int age) {
    this.name = name;
    this.age = age;
  }
}
```

The previous class allows me to create a new `HumanBeing` in the following ways:

- `HumanBeing me = new HumanBeing();`
- `HumanBeing me = new HumanBeing("Marcos");`
- `HumanBeing me = new HumanBeing("Marcos", 72);`

A constructor will be picked depending on what parameters I use and what their types are.

Question

Who can tell me what constructor runs when `me1` is instanciated?

What about for `me2`? Why?

```
public class Main {
  public static void main(String[] args) {
    HumanBeing me1 = new HumanBeing();
    HumanBeing me2 = new HumanBeing("Marcos"); } }

class HumanBeing {
  protected String name;
  protected int age;

  public HumanBeing() {
    this.name = "Unknown Person";
    this.age = 0; }

  public HumanBeing(String name) {
    this.name = name;
    this.age = 0; } }
```

`this`

The `this` keyword seems to come up a lot in Java, doesn't it? We've already seen it being used for when we want to access instance fields or call instance methods, but as we'll see, it also has another use case, and that it to call other constructors methods.

Let's put this to use

Let's start with this class and let's update it to use `this` instead.

```
class HumanBeing {
  protected String name;
  protected int age;

  public HumanBeing() {
    this.name = "Unknown Person";
    this.age = 0;
  }

  public HumanBeing(String name, int age) {
    this.name = name;
    this.age = age;
  }
}
```

Example using `this`

Here we have the same class, but some of the constructors changed and now all they do is call `this` as if it were a method. The first constructor does a little bit of work and then passes on the work to the other constructor. Notice how the second constructor did not change. This is because we still need code that will do something!

```
class HumanBeing {
  protected String name;
  protected int age;

  public HumanBeing() { this("Unknown Person", 0); }

  public HumanBeing(String name, int age) {
    this.name = name;
    this.age = age; } }
```

If I instanciate the previous class like so `HumanBeing me = new HumanBeing();` what will happen is that Java will start by using the first constructor, and all the first constructor does is run this code `this("Unknown Person", 0);`, which calls the second constructor, so both constructors will end up running.

Rules for using `this` as a constructor

There are two rules for using `this` as a way to call another constructor:

1. The call to `this` must be the first code that runs in a constructor. I cannot do something then run `this()`. Instead I have to run `this()` then do other things in the constructor.
2. `this` must call a constructor that exists!

Question

Who can tell me (1) what is wrong with the constructors below and (2) how I can fix them.

```
class Cat {
  protected String name;
  protected String breed;

  public Cat() {
    this(1, 2, 3, 4, 5, 6, 7);
  }

  public Cat(String name) {
    System.out.println("Creating a new Cat");
    this(name, "Unknown Breed");
  }

  public Cat(String name, String Breed) {
    this.name = name;
    this.breed = breed;
  }
}
```

Answer

```java
public Cat() {
  // This is calling an invalid constructor. There are
  // no constructors that take seven integers as
  // arguments.
  this(1, 2, 3, 4, 5, 6, 7);
}

public Cat(String name) {
  // This is running code (`System.out.println`) before
  // the call to `this`, which is not allowed.
  System.out.println("Creating a new Cat");
  this(name, "Unknown Breed");
}

public Cat(String name, String Breed) {
  // This is ok.
  this.name = name;
  this.breed = breed;
}
```

Question

Who can tell me (1) what constructor(s) runs when I instanciate `me1` and (2) why? What about for when In instanciate `me2` ?

```
public class Main {
  public static void main(String[] args) {
    HumanBeing me1 = new HumanBeing();
    HumanBeing me2 = new HumanBeing("Marcos", 92); } }

class HumanBeing {
  protected String name; protected int age;

  public HumanBeing() {
    this("Unknown Person", 0);
    System.out.println("Running Constructor #1"); }

  public HumanBeing(String name, int age) {
    this.name = name;
    this.age = age;
    System.out.println("Running Constructor #2"); } }
```

Answer

Instanciating `me1` triggers the first constructor, which then triggers the second one, so we see:

```
Running Constructor #2
Running Constructor #1
```

Instanciating `me2` triggers the second constructor which only sets properties, so we see:

```
Running Constructor #2
```

Question

What are the rules for using `this` as a way to call other constructors?

Answer

1. The call to `this` must be the first code that runs in a constructor. I cannot do something then run `this()`. Instead I have to run `this()` then do other things in the constructor.
2. `this` must call a constructor that exists!

`super`

The `super` method works just like the `this` method except it calls constructors of the parent class?

```
class Animal {
  protected String species;
  Animal(String species) { this.species = species; } }

class Human extends Animal {
  protected String favoriteColor;
  Human(String fc) {
    super("Homo sapiens");      // Calls the constructor
    this.favoriteColor = fc; } }// in the Animal class

class Student extends Human {
  protected String subject;

  Student(String subject) {
    super("Unknown");           // Calls the constructor
    this.subject = subject; } } // in the Human class
```

Timeout, let's review some terminology

When talking about class hierarchy, terms like Base, Parent, and Child will come up when talking about classes. Using the example below,

- `Animal` is the "Base" class since it doesn't extend anything, it is also the "Parent" of `Human` and of `Student`, since `Human` extends `Parent` and `Student` extends `Human`,
- `Human` is the "Parent" of `Student` and the "Child" of `Animal`
- `Student` is the "Child" of both `Human` and `Animal`

```
class Animal { /* ... */ }

class Human extends Animal { /* ... */ }

class Student extends Human { /* ... */ }
```

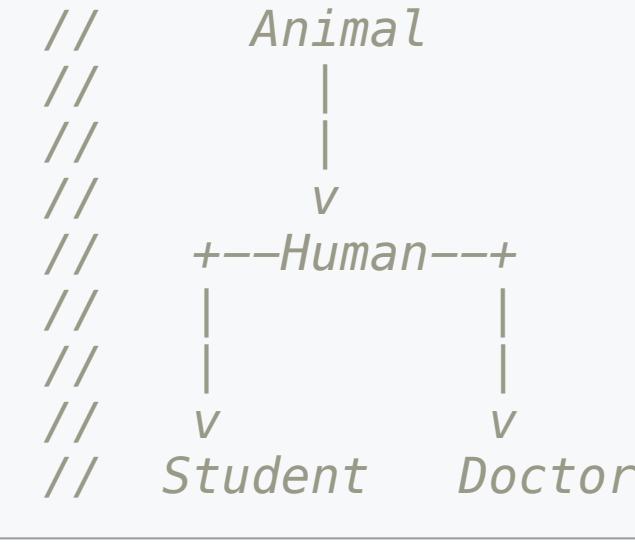
Try to imagine a root system or a family tree when thinking about class hierarchy, where base classes are at the top and they are followed by their child classes, which can also be parents to other classes.

```
class Animal {}

class Human extends Animal {}

class Student extends Human {}

class Doctor extends Human {}
```



Rules for using `super`

The rules for using `super` are the same as for `this`, which if you remember are the following:

1. The call to `this` must be the first code that runs in a constructor. I cannot do something then run `this()`. Instead I have to run `this()` then do other things in the constructor.
2. `this` must call a constructor that exists!

Notice how the first rule states we must call `this` first thing if we're calling it. If the same goes for `super`, this means that we can't have a constructor that calls both `super` and `this`, since they both want to go fist we have to pick one or the other.

Question

What are the rules for using `super`?

Answer

1. The call to `super` must be the first code that runs in a constructor. I cannot do something then run `super()`. Instead I have to run `super()` then do other things in the constructor.
2. `super` must call a constructor that exists!

Question

Are these constructors (1) valid? If not, (2) then why?

```
class Human extends Animal {
  protected String favoriteColor;

  Human() {
    super("Homo sapiens");
    this("Blue");
  }

  Human(String favoriteColor) {
    super("Homo sapiens");
    this.favoriteColor = favoriteColor;
  }
}
```

Answer

```
class Human extends Animal {
  protected String favoriteColor;

  Human() {
    // I cannot have a constructor run both `super` and
    // `this` because of the rules of `this`/`super`
    super("Homo sapiens");
    this("Blue");
  }

  Human(String favoriteColor) {
    // This is ok.
    super("Homo sapiens");
    this.favoriteColor = favoriteColor;
  }
}
```

Question

What is the difference between `this()` and `super()`?

Answer

`super()` calls a constructor in the parent class while `this()` calls a constructor in the current class.

A final note on constructors

Remember inheritance and how one of its benefits was that it allowed us to automatically call methods of a parent class? Well, this is not exactly the case with constructors as we are required to explicitly define constructors in child classes.

Using `super` to access parent class

Just like we can use `this` to access fields and methods or the current class, we can use `super` as an explicit way to access the parent class.

Casting

Let's talk about casting, why we need it and how to use it.

Polymorphism

Remember this from a little while ago? Do you also remember what classes were the parents and which ones were the children?

```
class Animal {}

class Human extends Animal {}

class Student extends Human {}

class Doctor extends Human {}
```

```
//      Animal
//        |
//        |
//        v
//    +--Human--+
//    |         |
//    |         |
//    v         v
//  Student   Doctor
```

Polymorphism (cont.)

- `Animal` is the "Base" class since it doesn't extend anything, it is also the "Parent" of `Human` and of `Student`, since `Human` extends `Parent` and `Student` extends `Human`,
- `Human` is the "Parent" of `Student` and the "Child" of `Animal`
- `Student` is the "Child" of both `Human` and `Animal`. Same goes for `Doctor`.

Another way of saying this would be:

- A `Human` is an `Animal`
- A `Student` is a `Human`
- A `Student` is an `Animal` since a `Human` is an `Animal` and we know that a `Student` is a `Human`. Same goes for `Doctor`.

A Child class "is a" Parent class.

Polymorphism (cont.)

Remember how a child class "is a" Parent class? Polymorphism says that a method that a method that wants a Parent class can be given a Child class as well. Here's an example:

```
public class Main {
  public static void main(String[] args) {
    Student you = new Student();
    you.favoriteColor = "Blue";
    printHumanFavoriteColor(you); }

  public static void printFavColor(Human h) {
    System.out.println("Favorite color: " +
      h.favoriteColor); } }
```

Notice how the `printHumanFavoriteColor` method wants a `Human` but we give it a `Student` instead. This is allowed because a `Student` is a `Human` and so anything a `Human` can do, so can a `Student`.

Suppose we have these classes:

```
public class Animal {
  private String sound;
  Animal(String sound) {
    this.sound = sound;
  }
  public void makeSound() {
    System.out.println(this.sound);
  }
}
public class Dog extends Animal {
  public Dog() {
    super("bark");
  }
}
public class Cow extends Animal {
  public Cow() {
    super("moo");
  }
}
```

This means that a `Dog` will make the "bark" sound and a `Cow` will make the "moo" sound.

Benefits (1)

We can instantiate a member of a child class and then save it into a variable for a parent class!

```
public class Main {
  public static void main(String[] args) {
    // A Dog is a kind of Animal so this is OK.
    Animal spot = new Dog();
    spot.makeNoise(); // "bark"

    Animal bessie = new Cow();
    bessie.makeNoise(); // "moo"
  }
}
```

Benefits (2)

We can even do this with arrays and lists, which means we can store objects of *different child classes* in a single data structure!

```
public class Main {
  public static void main(String[] args) {
    Animal[] animals = {
      new Dog(),
      new Cow()
    };
    for (int i; i < animals.length; i++) {
      animals[i].makeSound();
    }
  }
}
```

What will be the output of this program?

`instanceof`

This lets us check if an object belongs to a class, or if it extends that class (either directly or indirectly).

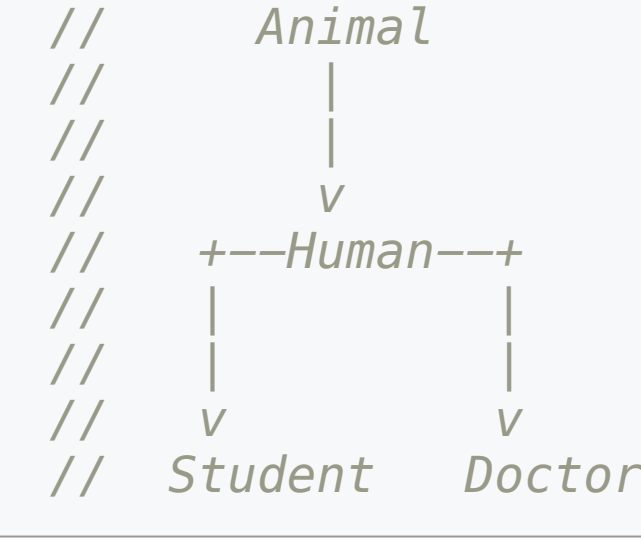
Example

```
class Animal {}

class Human extends Animal {}

class Student extends Human {}

class Doctor extends Human {}
```



```java
public class Main {
  public static void main(String[] args) {
    HumanBeing p = new Teacher("Andrew");
    if (p instanceof Teacher) {
      System.out.println("This person is a teacher!");
    }
  }
}
```

Additional resources

- *A Closer Look at Parameters* by Jim Wilson
  https://app.pluralsight.com/player?course=java-fundamentals-language&author=jim-wilson&name=java-fundamentals-language-m7&clip=0&mode=live
- *Class Inheritance* by Jim Wilson
  https://app.pluralsight.com/player?course=java-fundamentals-language&author=jim-wilson&name=java-fundamentals-language-m8&clip=0&mode=live