Topic 2.0: Objects

Learning Goals (Week 2):

- **Write simple to moderately complex classes.**
- **Constructors, instance variables and instance methods, the this keyword**
- Class variables and class methods
- **Compile & run a Java code with multiple files in the same directory.**
- **Use instances of user-defined classes**
- **User-defined classes in other user-defined class and in main methods**
- Explain how and why the concept of encapsulation is useful
- How Encapsulation is achieved via accessors / mutators.
- Understand object references and use them appropriately in code,
- Deep versus shallow object copies.

OOP

- Stands for **O**bject **O**riented **P**rogramming
- It is a programming model based on the construction and use of **objects**:
  - i.e. a program using a set of objects interacting together, a bit like the real world
- Many languages inherently support OOP
- We're going to see the basic concepts of objects and how they work in **Java**
  - Second year courses exist which explore this concept and more complex topics further

- **NOTE 1:** a **class** is not just a container in which we put a main method

```
public class MyProgram {
     public static void main (String[] args) {
          System.out.println("Hello!");
     }
}
```

Class means type

- When you create a class, you're actually defining a new type of data.
- Like any other data type, your new type has:
  - a **name** (the class name, should be CapitalCaseNotCamelCaseLikeAVariable
  - some kinds of **information** we want to store in it
    - actually, **a collection of information** called **instance variables** (& class variables, in a bit)
  - some kinds of behaviours/actions we want the data type to be able to perform
    - actually, **a collection of information** called **instance methods** (& class methods, in a bit)
- **Objects** of your new data types are often used to represent real life objects, similar to other data types, just more complex in nature.
  - e.g. int variables like `int myAge; int yourAge;` stored values for my age and your age
  - now:
  - e.g. with a user-defined `Tree` class, we can make Object variables like `Tree oak; Tree elm;` which stores values about elm and oak trees.

Defining new data type

- We’ve done it already!
  - A class is defined in a .java file, with the same name as the class

Example file: **Person.java**

```
public class Person{
    public String name;
    public int age;
}
```

This is just the class definition,
i.e a blueprint / model /template for your new type .

it does not create anything in memory, anything you can “use” directly

When you actually create an instance of this class, you get what we call an object.

Defining new data type

Example file: Person.java

```
public class Person{
    public String name;
    public int age;
}
```

These are the instance variables,

i.e. the object's data.

- each instance will have its own specific set of values for the instance variables.

Similar to the data "inside" an array

We'll talk about the "public" later.

Defining new data type

Example file: Person.java

```
public class Person{
    public String name;
    public int age;
}
```

We normally assign values to the instance variables in the constructor.

We'll see that in a bit

Defining new data type

- We've done it already!
  - A class is defined in a .java file, with the same name as the class

Example file: **Person.java**

```
public class Person{
    public String name;
    public int age;
}
```

- When we compile this file we get a class file (just like our Main class file)
  - Person.class

```
public class Main {
    public static void main(String[] args) {
        Person morrigan;
        Person lamia;
    }
}
```

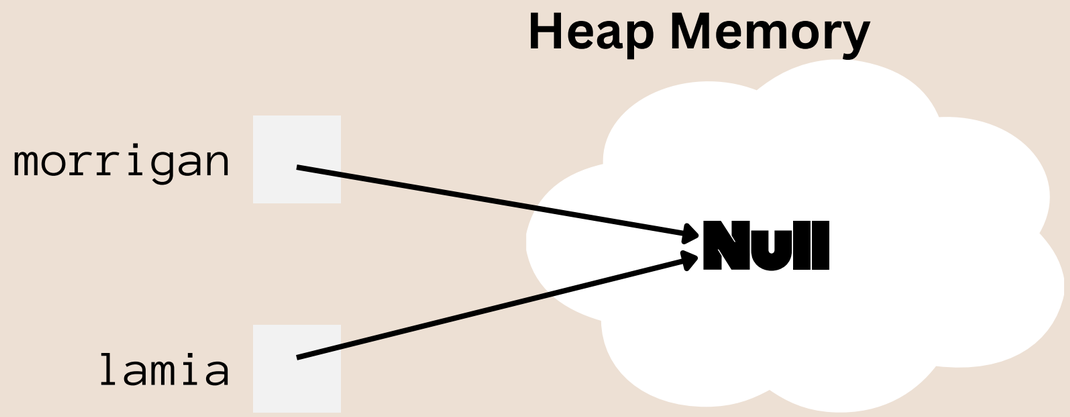
These are **DECLARATIONS.**
They are not assigned values.
They are **Null**

Creating variables of our new type

```
public class Main {
    public static void main(String[] args) {
        Person morrigan;
        Person lamia;
    }
}
```

These are **DECLARATIONS.**
They are not assigned values.
They are **Null**

- All variables of an object type contain a reference, not the object itself: just like we have seen for arrays (and Strings too), because they are objects as well



A note on references

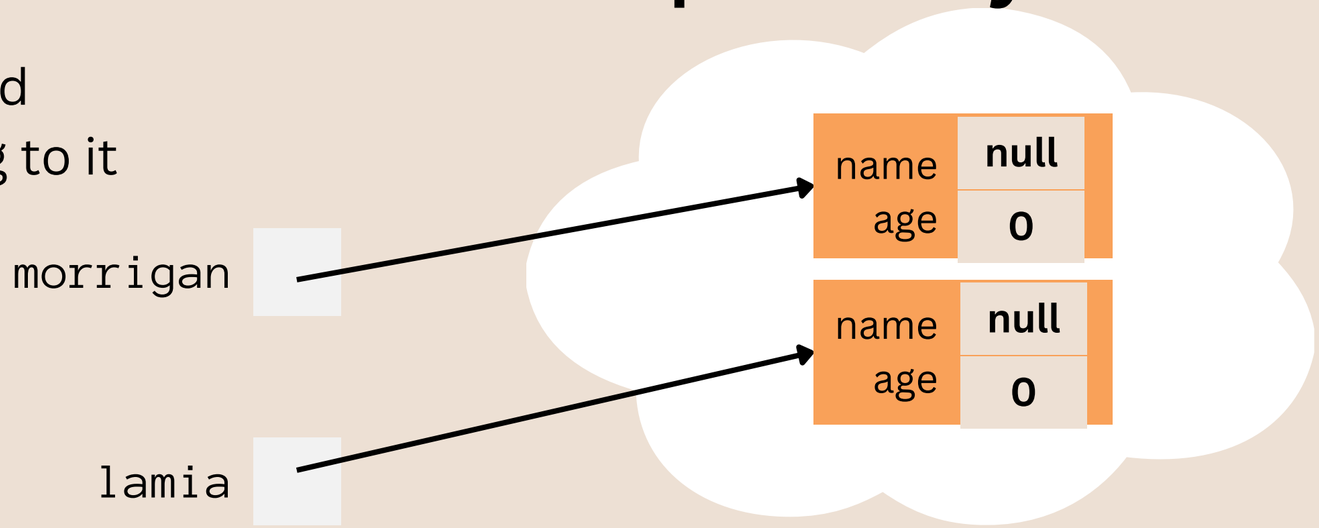
- Consider references a bit like a library book call number
- Each book has it’s own call number
- If two people have the same call number, you would be looking for the same book
- If someone scribbles all over the book with a call number 1.7.341 and someone else goes to check out the same book, they will see the scribbles
- Two copies of the same book can have their own unique call numbers, two individual, separate copies of the same book.

Object Instantiation

- To create an instance (an actual object), the basic syntax is:

```
public class Main {
    public static void main(String[] args) {
        Person morrigan = new Person();     // this calls the constructor
        Person lamia = new Person();
    }
}
```

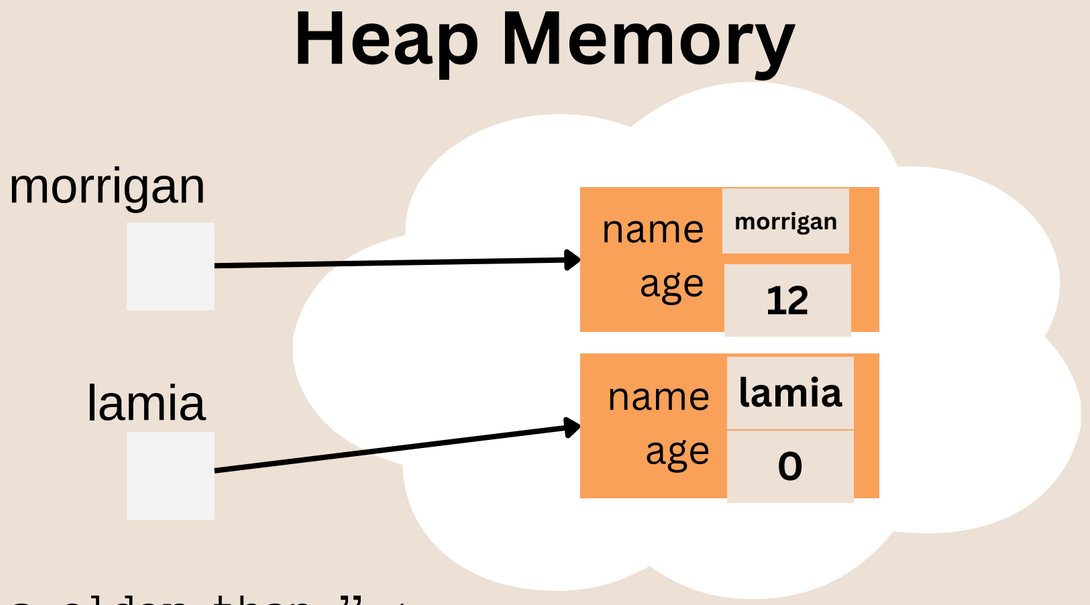
- This creates the object in memory, and returns a reference (address) pointing to it



Using Object Instances

- Given a reference to an object (e.g. morrigan or lamia below), you can access the instance variables using the syntax below, assuming that they are “public” (but normally they shouldn’t be… more on that soon)

```
public class Main {
    public static void main(String[] args) {
        Person morrigan = new Person();
        Person lamia = new Person();
        morrigan.name = “Morrigan”;
        lamia.name = “lamia”;
        morrigan.age = 12;
        if(morrigan.age > lamia.age) {
          System.out.println(morrigan.name + “ is older than ” +
              lamia.name);
        }
    }
}
```



Instance Methods

- Instance methods are methods that can only be used on instances (instantiated objects)
- These methods have access to the instance variables of that specific instance

- They can be defined similarly to the methods we previously introduced, but for instance methods you must
  - omit the static keyword
  - add the public keyword (in most cases... more on that soon)

Instance Methods

```
public class Person {
     public String name;
     public int age;

     //Instance methods below:
     public void haveBirthday() { age++; }

     public int getNumLettersInName() {
         return name.length();
     }
}
```

- age & name are ther **instance** variables for a specified object

Instance Methods

```
public class Person {
     public String name;
     public int age;

     //Instance methods below:
     public void haveBirthday() { age++; }

     public int getNumLettersInName() {
         return name.length();
     }
}
```

- age & name are ther **instance** variables for a specified object

```
public class Main {
    public static void main(String[] args) {
         Person morrigan = new Person();

         morrigan.name = “Morrigan”;
         morrigan.age = 12;


         morrigan.haveBirthday();
         // age has now been increased to 13

         System.out.println(morrigan.getNumLettersInName()); // 8
    }
}
```

Instance Methods

```
public class Main {
    public static void main(String[] args) {
        Person morrigan = new Person();

        morrigan.name = “Morrigan”;
        morrigan.age = 12;

        int letters = morrigan.getNumLettersInName());
    }
}
```

- You use an instance method by applying it as an operation to one particular object instance
- The instance methods are called “on” a specific instance, and they will be able to access/change the instance variables (e.g. name/age) of this specific instance

A note on running multi-file programs

- Each class is in it’s own file (based on the class name):
  Person.java
  Main.java
- These files **must** be in the same folder directory on your computer
- When compiled they become:
  Person.class
  Main.class
- Class files
- We can now run the Main class main method
- As long as the files are in the same directory, the code can find it easily

‘Under the hood’

- E.g. send the haveBirthday() message to the morrigan object:

```
morrigan.haveBirthday();
```

- This method calls acts as if this was happening (**WARNING**: NOT the way to do this in Java):

```
haveBirthday(morrigan);

//in class definition:
 public void haveBirthday(Person this) { this.age++; }
```

- **AGAIN: NOT** what is happening, but sort of similar. *this* is a keyword in Java, we will see it later.

Scope

- In an instance method, if you have a local variable (e.g. a parameter) with the same name as an instance variable; the local one will be used (shadows the instance variable)
  - i.e. Java sees the “closer” variable first

```
public class Person {
	public String name;
	public int age;

	public void giveAName(String name) { // name shadows name
		name = “Hashan”; // sees the local name before the global name
	}
}
```

The *this* keyword

- You can use the keyword **this** to represent the current **instance** of the object (the instance on which the method was called)

```
public class Person {
     public String name;
     public int age;

     public void giveAName(String name) { // name shadows name
          this.name = “Hashan”;
          // this ensures the instance name use, not the local one
     }
}
```

- ***this*** can be used to specify that you want to access the instance variable for the object that the message was sent to

The *this* keyword: another example: compilation error

```
public class Person {
     public String name;
     public int age;

     public void setAge(int age) {
          age = age;
          // this will not work, age is ambiguous
          // Java doesn't know which you want
     }
}
```

The *this* keyword: another example: fixed

```
public class Person {
     public String name;
     public int age;

     public void setAge(int age) {
          this.age = age;
          // this keyword disambiguates between the age variables
     }
}
```

toString() method

- How do we print the information in our Person object? **Okay this works**

```
public class Main {
    public static void main(String[] args) {
        Person morrigan = new Person();

        morrigan.name = “Morrigan”;
        morrigan.age = 12;

        System.out.println(morrigan.name + “ is ” + morrigan.age + “ years old.”);
    }
}
```

- ...But what if we have 10 Person objects? Do we need to write that same print line each time?
  - What happened if we want to change it later....10 changes....

toString() method

- How do we print the information in our Person object?

```
public class Main {
    public static void main(String[] args) {
        Person morrigan = new Person();

        morrigan.name = “Morrigan”;
        morrigan.age = 12;


        System.out.println(morrigan); // what does this print?
    }
}
```

toString() method

- This method is automatically called by Java to get a String representation of your object (called automatically when System.out.println() is needed with your object)

```
public class Person {
     public String name;
     public int age;

     public String toString() {
          // what goes in here?
     }
}
```

toString() method

- toString() is a very useful instance method, which you should always try to supply using this signature:

```
public class Main {
    public static void main(String[] args) {
        Person morrigan = new Person();

        morrigan.name = “Morrigan”;
        morrigan.age = 12;



        System.out.println(morrigan); // will call your toString() method now
    }
}
```

toString() method

- When you define your toString() method for your object, **you** control how your object will be displayed as a String

```
public class Person {
     public String name;
     public int age;

     public String toString() {
          return name  “ is ” + age + “ years old.”;
     }
}
```

toString() method

- Once toString() is defined, you can now get readable results from, for example:

```
public class Main {
    public static void main(String[] args) {
        Person morrigan = new Person();
        Person hamish = new Person();
        hamish.name = “Hamish”;
        morrigan.name = “Morrigan”;
        morrigan.age = 12;

        System.out.println(morrigan); // will call your toString() method now
        System.out.println(hamish);
    }

}
```

One more note on toString()

- Note that there is always a toString() method à if you don’t supply one, Java uses a default one
  - That was the garbage we saw it spit out before.
  - The default toString() prints the Data Type and Memory Address (reference) of the Object…not very useful for us

Pause & Practice!

- Create a class **VideoGameCharacter** in its own Java file. Your class should have a **String name** and **int hitPoints.** Create a **toString()** method which prints all the information nicely.
- Create a main class which makes **VideoGameCharacter** characters, give them names and hit point values and practice printing them using System.out.println.

Constructors

- A **constructor** is a special method that is used to **instantiate** (create an instance of) an object. “
  - think about it like allocating/reserving memory for an object (**not** setting it, just making space in memory for it)

- We normally use it to initialize the instance variables (variables of our data type, our Objects)

- We can also do any kind of special processing that needs to be done when the object is created (input, output, calculations, creation of other objects, etc.)
  -

Constructors

- A constructor is a special case of an instance method:
  - **no return type at all (not even void)**
  - it must have the exact same name as the class

- It’s run automatically when an object instance is created (by new <ClassName>(...))

```
public class Main {
    public static void main(String[] args) {
        Person morrigan = new Person();
    }
}
```

Defining a constructor

- Syntax:

```
public class Person {
     public String name;
     public int age;

     public Person(String name, int age) {
          //instructions to do during object instantiation
     }
}
```

- Let’s break down each part of this constructor signature
- Although it’s possible to have a private constructor, we normally make them public

Defining multiple constructors

- You can define multiple constructors, as long as they have different signatures (lists of parameters)

```
public class Person {
    public String name;
    public int age;
    public Person() {          // no parameters – “default”
        name = “Newborn”;
        age = 0;
    }
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

A note on constructor parameters

- You could also use different parameter names to just avoid the conflicts, so that ‘this.’ isn’t required

```
public class Person {
	public String name;
	public int age;

	public Person(String n, int a) { // not as descriptive but no this
		name = n;
		age = a;
	}
}
```

Constructing new Objects

- If any constructors are supplied, the correct parameters for one of them must be used when creating an instance:

```
Person Kehinde = new Person("Kehinde", 29); // 2nd one
Person newborn = new Person( ); // 1st one

// error this constructor does not exist!
Person otherNewborn = new Person(0);
```

- A note on instance variable values: When a constructor does not initialize the instance variables, they just keep their default value

The default Constructor

- A default constructor is **already** provided by Java in case no constructor is defined in a class

```
public NameOfTheClass() { } // by default, you do not need to make this
```

- This default constructor does not do anything except instantiating the object

- **Note** that this default constructor disappears as soon as one constructor is defined in the class (no matter the list of parameters it uses)

Default Constructor: Example

```
//in Person.java file:
public class Person {
 public String name;
 public int age;
}
```

```
//in Test.java file
public class Test{
   public static void main(String[] args) {
       Person p = new Person();
   }
}
```

- This compiles, no errors, the default constructor is present here and can instantiate the Person object

Default Constructor: Example

```
//in Person.java file:
public class Person {
   public String name;
   public int age;

    public Person(String n, int i) {
        //statements here
     }
}
```

```
//in Test.java file
public class Test{
   public static void main (String[] args) {
      Person p = new Person();
   }
}
```

- This does not compile anymore!
  - no default constructor!
- cannot find symbol constructor Person()

Pause & Practice

Create a Java class named Animal. Your class should have the following components:
Variables:
name (String): to store the name of the animal.
numLegs (int): to store the number of legs the animal has.

Constructor:
Create a constructor for the Animal class that takes in name and numLegs as parameters and initializes the respective class variables.

toString() Method:
Override the **toString()** method to return a string in the format: "Animal{name='[name]', numLegs=[numLegs]}", where [name] and [numLegs] are replaced with the actual values.
sit Method:

Create a method named **sit()** that doesn't return anything (void). When called, it should print a message: "[name] is sitting.", where [name] is the name of the animal.

A main method has been added in UMLearn to test this Animal.java class with