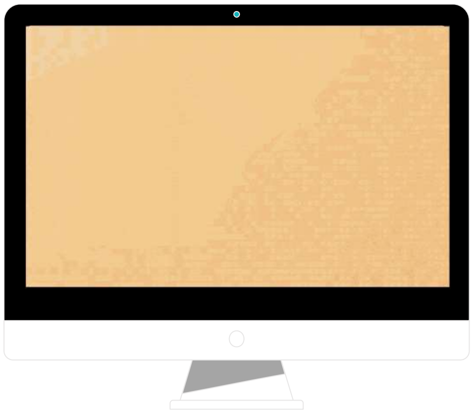
Recall: Java Fundamentals

1. What collections of data did we cover last week and what is the difference between them?
2. What was the best way of iterating over a collection of data?
3. How do we take an input from the user?
4. What can you tell me about the 'return' keyword and how it affects a method construction?



1

10. Object Oriented Programming

What is Object Oriented Programming?

Understanding public static void main(String[] args)

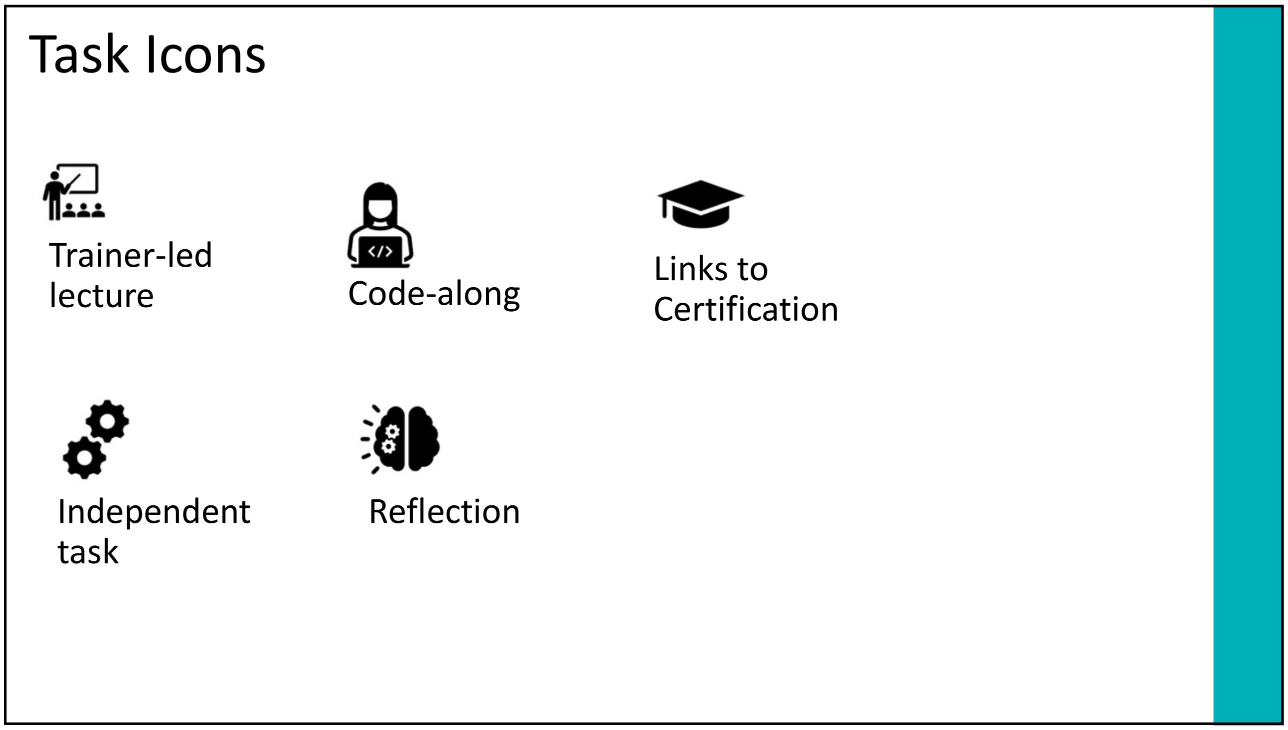
Introduction to OOP Concepts

01
Front-End
Development

02
Back-End
Development

03
BCS
Certification

2



Task Icons

Trainer-led lecture

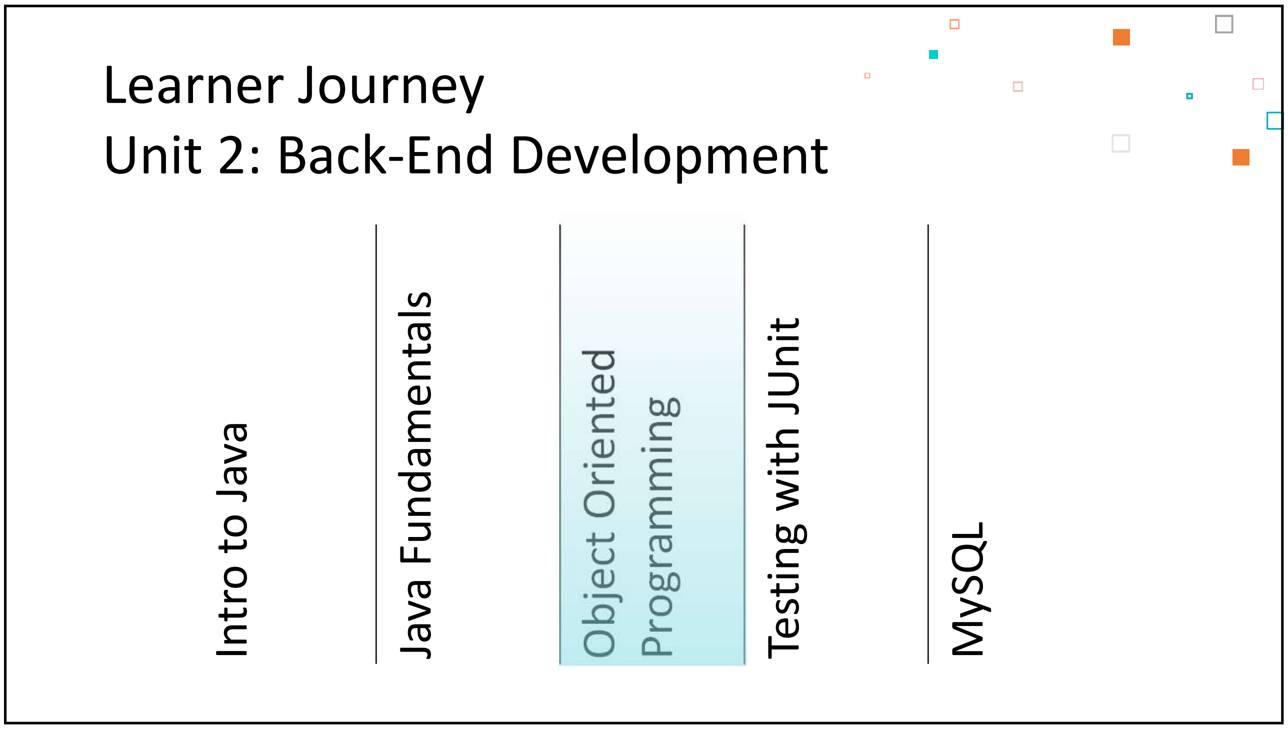
Code-along

Links to Certification

Independent task

Reflection

3



Learner Journey
Unit 2: Back-End Development

Intro to Java

Java Fundamentals

Object Oriented Programming

Testing with JUnit

MySQL

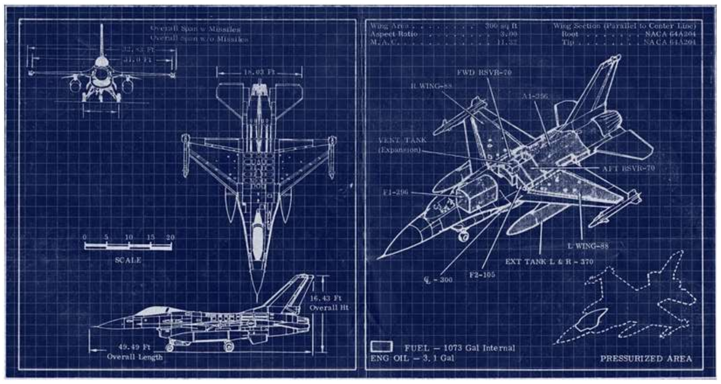
4

What is Object Oriented Programming (OOP)?

5

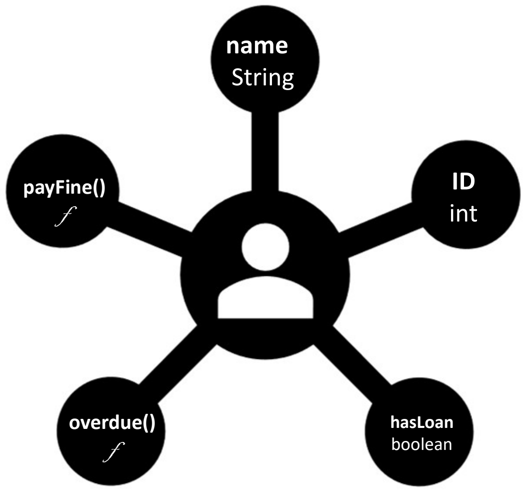
Classes are Blueprints

- This class then becomes a blueprint, allowing us to create as many objects as we need without having to write laborious amounts of code!



6

Example: Library User



Name: Joe
ID: 1
hasLoan: yes
-
overdue(): no
payFine(): no

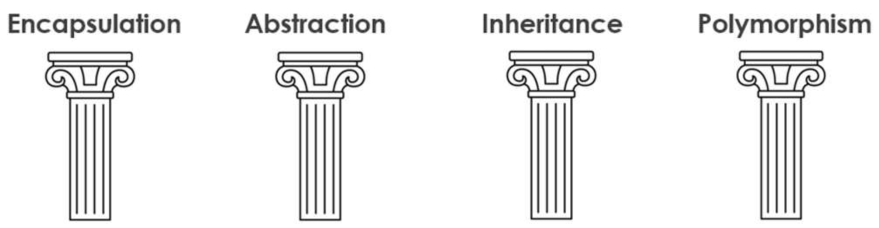
Name: Priya
ID: 2
hasLoan: yes
-
overdue(): yes
payFine(): no

Name: Lia
ID: 3
hasLoan: no
-
overdue(): no
payFine(): no

7

The Four Pillars

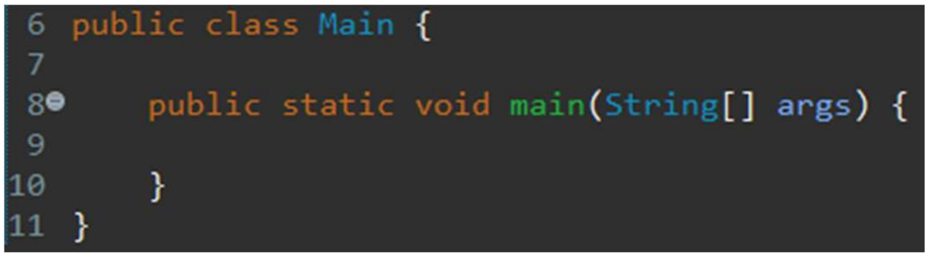
- Within Object Oriented Programming there are concepts that are fairly consistent between languages.
- These concepts are what define OOP.



8

Understanding Main and public static void main(String[] args)

- Before we jump into the concepts of OOP in detail, understanding the core of our program and what each word in a blank class means *for* our program is important.
- By now you should have seen this code several times – but what does it mean?

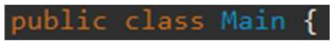


```
public class Main {

    public static void main(String[] args) {

    }
}
```

9

Breaking it Down

- At the top of our program, we are presented with the following code:



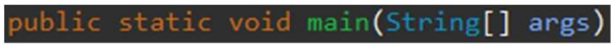
```
public class Main {
```

- This code tells us a few things:
  - The code within the file is publicly **accessible by other classes**
  - The file that contains the code is itself a class
  - The name of the class is called ‘Main’

10

Breaking it Down – main method

- The next section of code is somewhat more complicated and is what we call the 'main method':



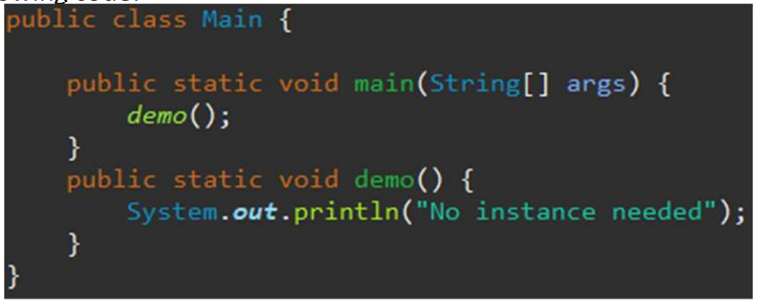
```java
public static void main(String[] args)
```

- Just like the line of code previous, this method is also publicly accessible.
- 'static' means that this method can be executed (called) without creating an object of that class.

11

static

- Consider the following code:



```java
public class Main {

    public static void main(String[] args) {
        demo();
    }
    public static void demo() {
        System.out.println("No instance needed");
    }
}
```

- We can clearly see from our interpreter that no errors are shown here. Executing this code will run the 'main' method (as we have seen before), which will then call the 'demo()' method.
- We have not created an instance of demo but have been able to execute it.

12

Task 1: Methods in the Main Class

- Create a new project called "IntroToOOP" and make a "Main" class with "public static void main(String[] args)" ticked.
- Create a method **outside** of the Main method:

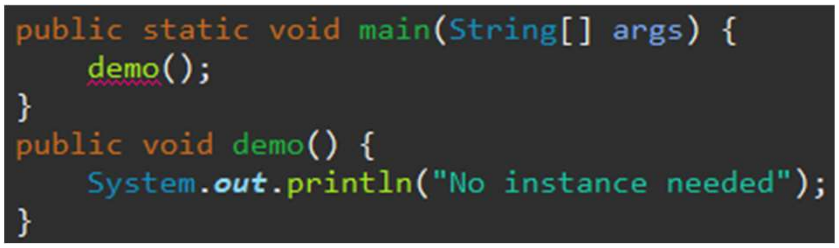
```
public static void printToScreen() {
    System.out.println("Hello World");
}
```

- Have this method print a message to the console.
- Once you have seen it work, remove the word "void" from the "printToScreen()" method. What are the results?

13

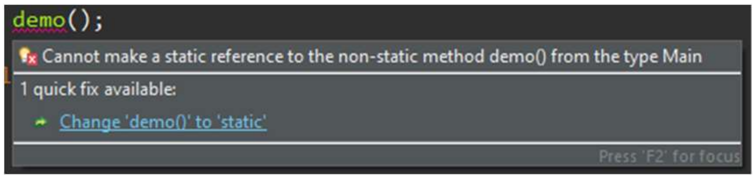
static

- Now consider the code below:



```
public static void main(String[] args) {
    demo();
}
public void demo() {
    System.out.println("No instance needed");
}
```

- The only change here is the **removal of the word static** from the 'demo' method. We can clearly see the interpreter shows that 'demo' now gives the following error:

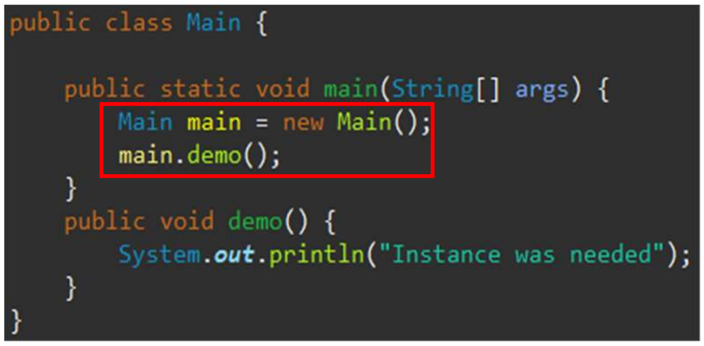


```
demo();
Cannot make a static reference to the non-static method demo() from the type Main
1 quick fix available:
Change 'demo()' to 'static'
Press 'F2' for focus
```

14

static (cont…)

- What we need to do to access the demo method without the ‘static’ keyword is **instantiate** a new **instance** of the class ‘Main’ (not the method).
- From this new object called ‘main’ (in yellow), we are then able to access the demo method.

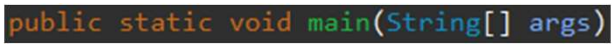


```
public class Main {

    public static void main(String[] args) {
        Main main = new Main();
        main.demo();
    }
    public void demo() {
        System.out.println("Instance was needed");
    }
}
```

15

Breaking it Down

- Back to our original line of code:



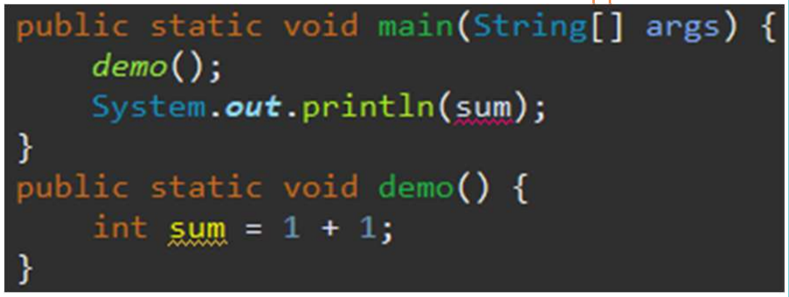
```
public static void main(String[] args)
```

- The ‘void’ keyword simply means that our method does not ‘return’ any information to where the method was called from.

16

void

- Consider the following code:



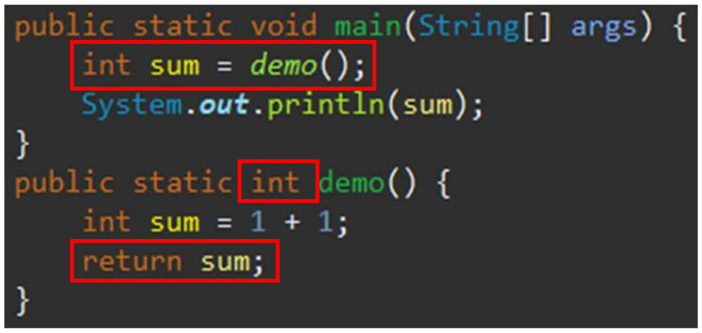
```
public static void main(String[] args) {
    demo();
    System.out.println(sum);
}
public static void demo() {
    int sum = 1 + 1;
}
```

- As you can see, ‘sum’ is not accessible outside of the demo method. This is because ‘sum’ is a member of the demo method, and can only be accessed *within* the demo method.

17

void (cont.)

- Access to members of another method will be covered later within **Encapsulation**, but for now, we will cover how to get that value from the method using a return type and the ‘return’ keyword:



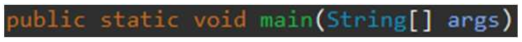
```
public static void main(String[] args) {
    int sum = demo();
    System.out.println(sum);
}
public static int demo() {
    int sum = 1 + 1;
    return sum;
}
```

- Where the word ‘void’ once was is now replaced with the word ‘int’. This is because we are now **returning** an integer to the method that called it, whilst also assigning that returned value to an integer that is a **member of main**.

18

Breaking it Down (cont.)

- Finally, let us take a look at the final key words within this line of code, 'String[] args':



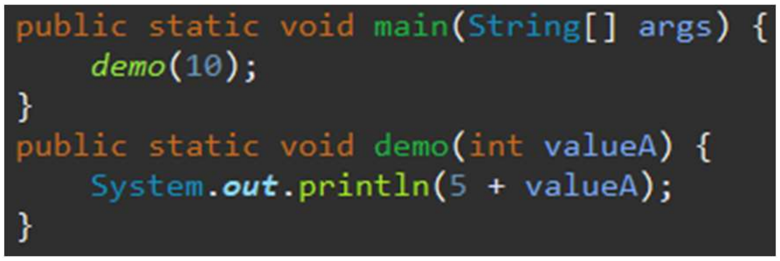
```
public static void main(String[] args)
```

- Each method that is created must be presented with parenthesis:
  - myGame()
  - move()
  - main()
- These parenthesis, when empty, indicate that no data will be **passed to the method**.

19

args

- Consider the following code, we have changed the demo method to include the text 'int valueA':



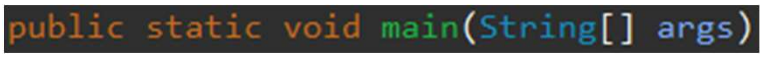
```
public static void main(String[] args) {
    demo(10);
}
public static void demo(int valueA) {
    System.out.println(5 + valueA);
}
```

- You may be able to notice from the syntax colouring that the word 'int' is a variable type, and 'valueA' is the name of that variable.
- Within the 'main' method, we have passed in the value of '10' to that method.

20

args (cont.)

- Once we understand that a method can take values as input parameters, the following code becomes clearer:



```
public static void main(String[] args)
```

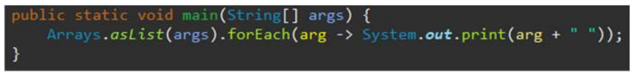
- What is happening here is that the main method is capable of being supplied with an **array** of **String** values that will be stored in a variable called **args**.

- Although we can do this to any method, it is especially important when executing a Java program from a command line interface.

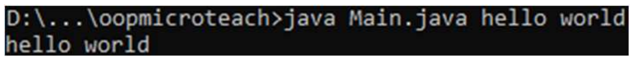
21

args (cont…)

- Executing the program with the following code, from the command line interface, will iterate through all arguments and print them to the screen using a lambda:



```
public static void main(String[] args) {
    Arrays.asList(args).forEach(arg -> System.out.print(arg + " "));
}
```



```
D:\...\oopmicroteach>java Main.java hello world
hello world
```

22

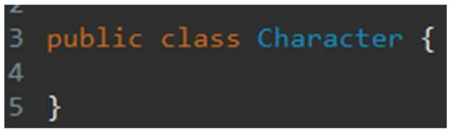
Creating a Custom Class

- To begin learning the basics of OOP, let us begin by making a new class called ‘Character’.
- This class will serve as a starting point for our OOP concepts and will house information about character objects we want to use within, for example, a game that we are developing.
- What kind of information could we store about a character in our game?

23

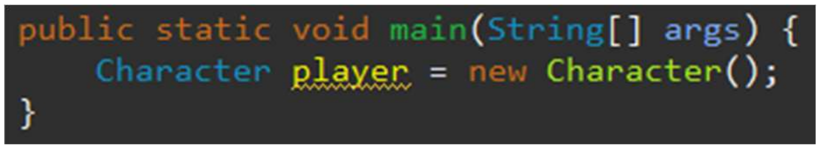
Character Class

- Creating the new Character class will present you with your standard blank class:



```
public class Character {

}
```

- With this class, we can then create a Character object in our Main class:



```
public static void main(String[] args) {
    Character player = new Character();
}
```

24

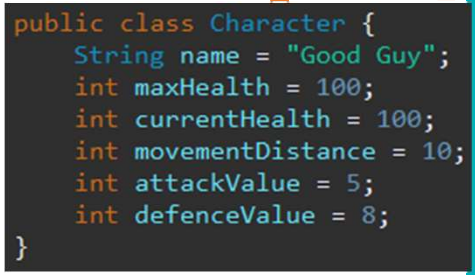
Character Class

- The Character object that has been named ‘player’, unfortunately, cannot do anything right now, and has no values for the criteria specified earlier:
  - Name
  - Max health
  - Current health
  - Attack value
  - Defence value
  - Movement distance

25

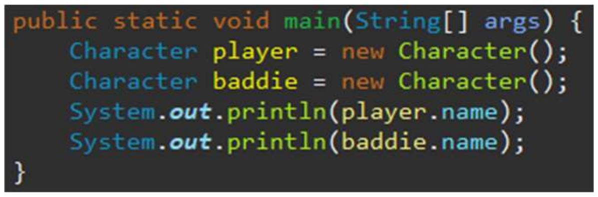
Character Class (cont…)

- We can create these values within the class directly:

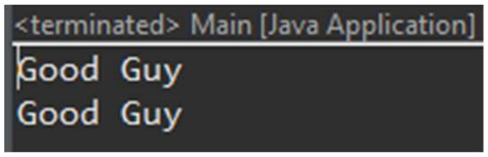


```java
public class Character {
    String name = "Good Guy";
    int maxHealth = 100;
    int currentHealth = 100;
    int movementDistance = 10;
    int attackValue = 5;
    int defenceValue = 8;
}
```

- The problem here is that when we use this Character class to make a “Bad Guy” character, it will share all the same characteristics of the “Good Guy” – this makes for a terrible game!



```java
public static void main(String[] args) {
    Character player = new Character();
    Character baddie = new Character();
    System.out.println(player.name);
    System.out.println(baddie.name);
}
```

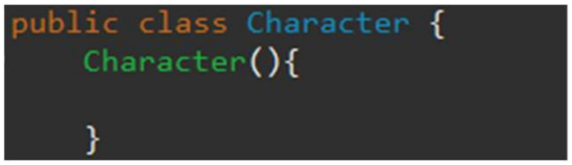


```
<terminated> Main [Java Application]
Good Guy
Good Guy
```

26

Constructors

- To overcome this issue, we need to create a custom constructor. This is how we are going to pass information to the class, which will then be initialised with the values presented.



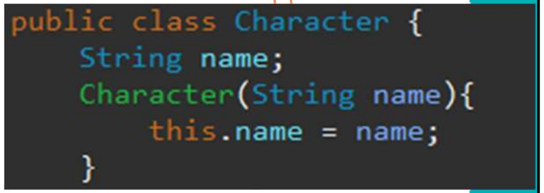
```
public class Character {
    Character(){

    }
```

- The new constructor method here **must** be called the same name as the class. As it stands, we are not expecting any information in this constructor as the parenthesis are empty!

27

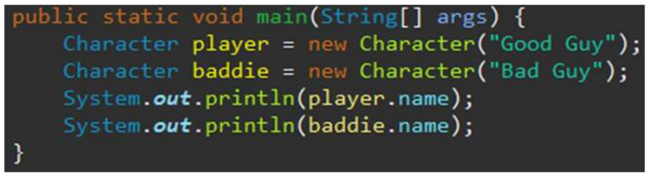
Constructors (cont…)

- Lets start with just adding a name to this constructor:

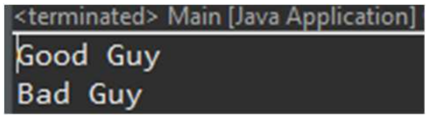


```
public class Character {
    String name;
    Character(String name){
        this.name = name;
    }
```

- We have removed the ' = "Good Guy"' part of the code from the variable instantiation and instead are asking for the name when we call the Character constructor:



```
public static void main(String[] args) {
    Character player = new Character("Good Guy");
    Character baddie = new Character("Bad Guy");
    System.out.println(player.name);
    System.out.println(baddie.name);
}
```

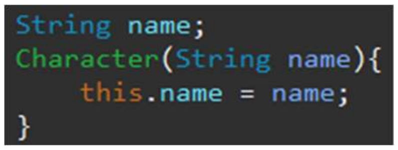


```
<terminated> Main [Java Application]
Good Guy
Bad Guy
```

28

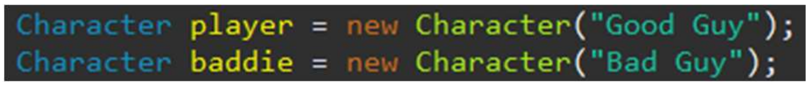
Constructors, ‘this’ and ‘new’

- You may have notice the line ‘this.name = name’ in the last code snippet:



```java
String name;
Character(String name){
    this.name = name;
}
```

- The ‘this’ keyword is very important for OOP. Although it can be tricky to grasp, what it means is the ‘name’ that is being passed to the constructor is the one that is associated with the variable that called the constructor from main:



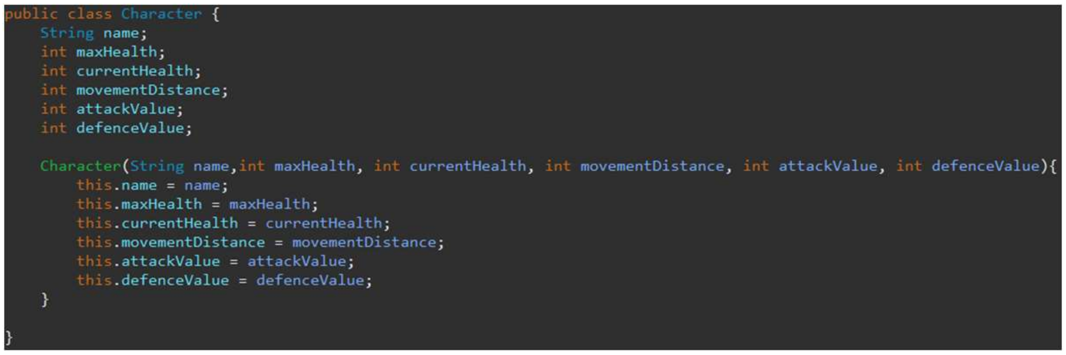
```java
Character player = new Character("Good Guy");
Character baddie = new Character("Bad Guy");
```

- It is only possible to associate the name with the variable that is being created because we create a **new** instance of that object!

29

Character Constructor Complete

- The following code includes all values as part of the constructor:

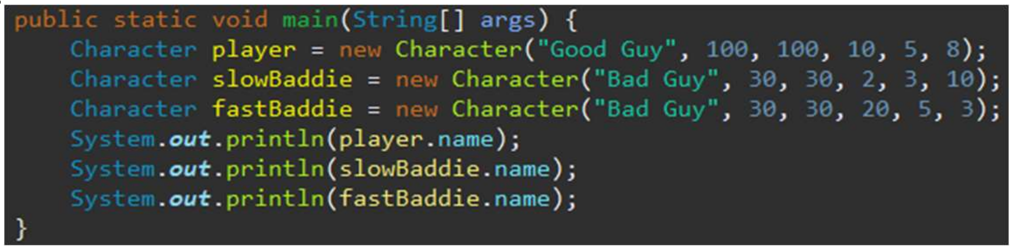


```java
public class Character {
    String name;
    int maxHealth;
    int currentHealth;
    int movementDistance;
    int attackValue;
    int defenceValue;

    Character(String name,int maxHealth, int currentHealth, int movementDistance, int attackValue, int defenceValue){
        this.name = name;
        this.maxHealth = maxHealth;
        this.currentHealth = currentHealth;
        this.movementDistance = movementDistance;
        this.attackValue = attackValue;
        this.defenceValue = defenceValue;
    }

}
```

30

Unlimited Characters

- This now means that we can create as many characters with **different** attributes as we want:



```java
public static void main(String[] args) {
    Character player = new Character("Good Guy", 100, 100, 10, 5, 8);
    Character slowBaddie = new Character("Bad Guy", 30, 30, 2, 3, 10);
    Character fastBaddie = new Character("Bad Guy", 30, 30, 20, 5, 3);
    System.out.println(player.name);
    System.out.println(slowBaddie.name);
    System.out.println(fastBaddie.name);
}
```

Stretch & Challenge

- Research Constructor Overloading

31

Task 2: Create a Custom Class

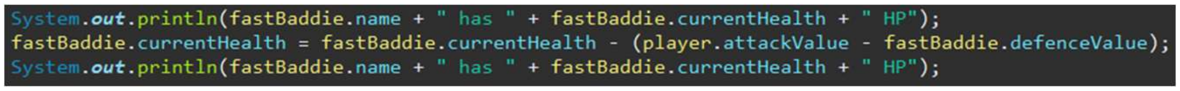
- Create a custom Character class that will allow you to make many characters.
- Your class can have whatever attributes you would like. Good attributes could be:
  - Name, Attack strength, Defence, Current and Max Health, Speed, Hair/Eye colour, Height, Scars, Clothing, Description, Background, etc.
- Make a constructor for your Character class and create multiple characters within Main.

Stretch Challenge: Create a method in your Character class that will print all of the information stored in a Character object.

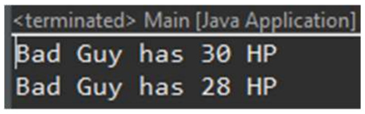
32

Getting and Setting Object Attributes

- Now that we have a player and a couple of enemies to work with, being able to manipulate those attributes is as simple as changing them directly:



```
System.out.println(fastBaddie.name + " has " + fastBaddie.currentHealth + " HP");
fastBaddie.currentHealth = fastBaddie.currentHealth - (player.attackValue - fastBaddie.defenceValue);
System.out.println(fastBaddie.name + " has " + fastBaddie.currentHealth + " HP");
```



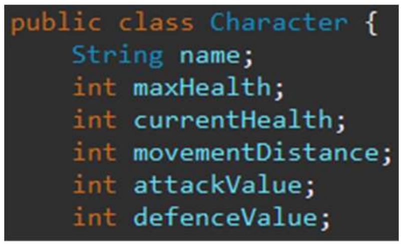
```
<terminated> Main [Java Application]
Bad Guy has 30 HP
Bad Guy has 28 HP
```

- Although this might seem like a good thing, it is in fact considered highly bad practice to access and manipulate an object attribute directly!

33

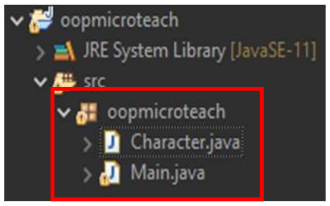
Why Direct Manipulation is Bad

- Lets take a look at how we have created our variables inside our Character class:



```
public class Character {
    String name;
    int maxHealth;
    int currentHealth;
    int movementDistance;
    int attackValue;
    int defenceValue;
```

- These class members are, by default, visible by any class within **the entire package**:



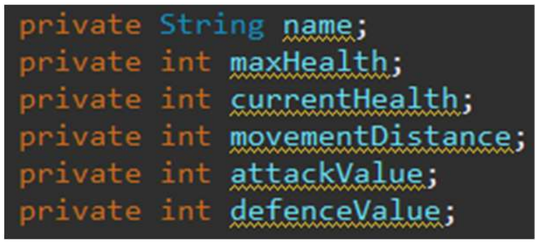
```
oopmicroteach
  > JRE System Library [JavaSE-11]
  src
    oopmicroteach
      > Character.java
      > Main.java
```

34

Why Direct Manipulation is Bad

- Although we only have Character and Main in our package at present, these packages will eventually consist of many more classes, and having visibility from every class is a security flaw!

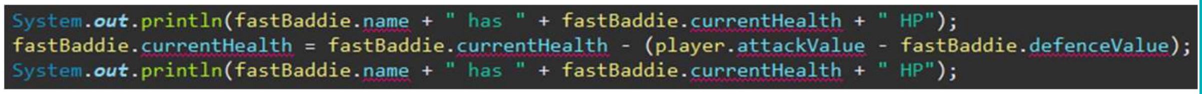
- To avoid our class members being visible to everything within the package, we change the access type to **private**:



```
private String name;
private int maxHealth;
private int currentHealth;
private int movementDistance;
private int attackValue;
private int defenceValue;
```

Why Direct Manipulation is Bad

- By changing our attributes to **private**, we have removed the ability to access those members directly!



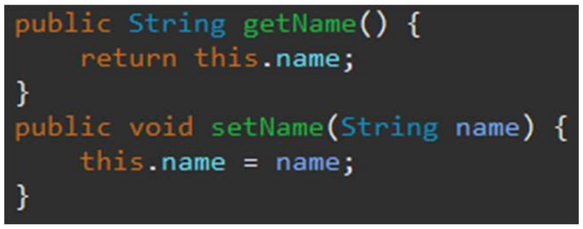
```
System.out.println(fastBaddie.name + " has " + fastBaddie.currentHealth + " HP");
fastBaddie.currentHealth = fastBaddie.currentHealth - (player.attackValue - fastBaddie.defenceValue);
System.out.println(fastBaddie.name + " has " + fastBaddie.currentHealth + " HP");
```

- This means we need to introduce a new way of accessing those members **indirectly** whilst still maintaining control over those members.

- We do this with Getters and Setters.

Getters and Setters

- As the name implies, getters and setters will either get the information, or set the information, for the attribute in question.
- We do this by creating special **public** methods within the Character class literally called ‘getName’, ‘setName’, ‘getCurrentHP’,… etc:



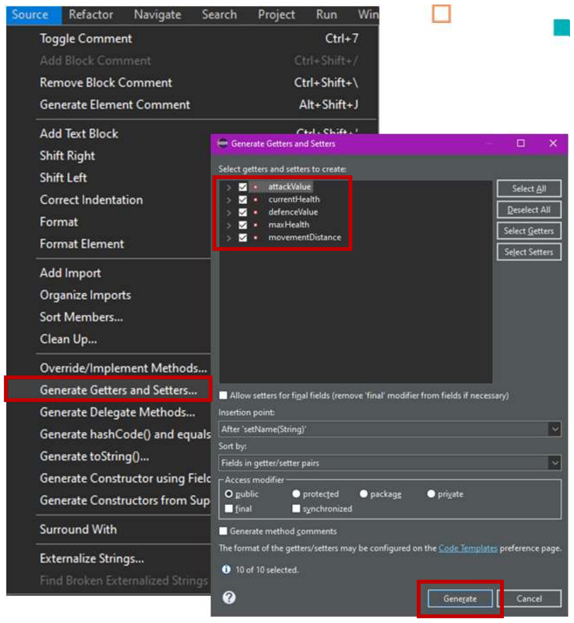
```
public String getName() {
    return this.name;
}
public void setName(String name) {
    this.name = name;
}
```

- We are using familiar code above; return, void, this, etc.

37

Getters and Setters

- To save time typing all these getters and setters out, your IDE will have an option to automatically generate the getters and setters for you!


Source
Refactor
Navigate
Search
Project
Run
Win
Toggle Comment Ctrl+7
Add Block Comment Ctrl+Shift+/
Remove Block Comment Ctrl+Shift+\
Generate Element Comment Alt+Shift+J
Add Text Block
Shift Right
Shift Left
Correct Indentation
Format
Format Element
Add Import
Organize Imports
Sort Members...
Clean Up...
Override/Implement Methods...
Generate Getters and Setters...
Generate Delegate Methods...
Generate hashCode() and equals
Generate toString()...
Generate Constructor using Fielc
Generate Constructors from Sup
Surround With
Externalize Strings...
Find Broken Externalized Strings
Generate Getters and Setters
Select All
Deselect All
Select Getters
Select Setters
Generate
Cancel

Select getters and setters to create:

- [✓] attackValue
- [✓] currentHealth
- [✓] defenceValue
- [✓] maxHealth
- [✓] movementDistance

☐ Allow setters for final fields (remove 'final' modifier from fields if necessary)

Insertion point: After 'setName(String)'

Sort by: Fields in getter/setter pairs

Access modifier: [✓ public] protected package private
☐ final ☐ synchronized

☐ Generate method comments

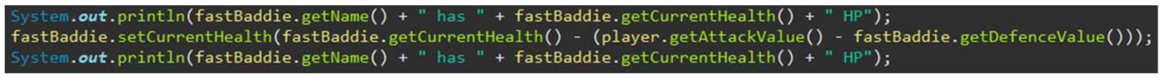
The format of the getters/setters may be configured on the Code Templates preference page.

10 of 10 selected.

38

Modifying Main

- Modifying the Main class to utilise these new getters and setters should now look like:



```
System.out.println(fastBaddie.getName() + " has " + fastBaddie.getCurrentHealth() + " HP");
fastBaddie.setCurrentHealth(fastBaddie.getCurrentHealth() - (player.getAttackValue() - fastBaddie.getDefenceValue()));
System.out.println(fastBaddie.getName() + " has " + fastBaddie.getCurrentHealth() + " HP");
```

39

Encapsulation

- What we have achieved by creating getters and setters is **Encapsulation** – from the word capsule!

- We have wrapped our Character class attributes in publicly accessible getters and setters, ensuring that the information is no longer visible to any class that has not instantiated an object of the Character class.

40

Task 3: Updating our class with Getters/Setters

- Update your Character class to include Getters and Setters.
- There should be a getter and a setter for each attribute you want the user to be able to update.
- If you have used attributes like Name and Eye Colour, you may not want to make only Setters for these!

Stretch Challenge: Update your print method to use getters instead of direct access.

41

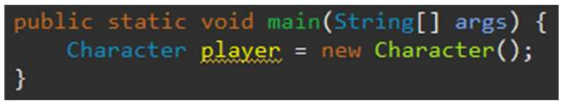
Polymorphism

- The term polymorphism can be broken down into two parts:
  - Poly – which means "many"
  - Morph – which means to be able to change
- This is a Greek word that means "many-shaped".
- We use polymorphism in programming to allow the developer to access different types of construction of a class instance.
- These are in the form of constructors.

42

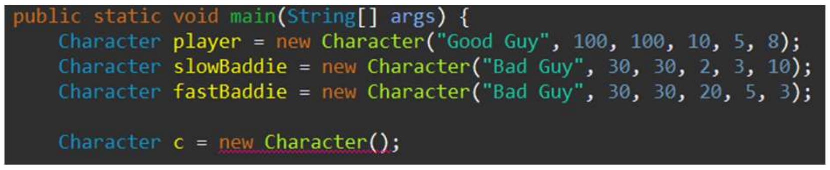
Default Constructor Not Working

- Previously, before making a constructor in our Character class, we were able to make a character with just the **default constructor**.



```
public static void main(String[] args) {
    Character player = new Character();
}
```

- Now that we have made **our own constructor**, the default one no longer exist. If we want to be able to make generic characters with no traits, we must create a constructor with no traits.

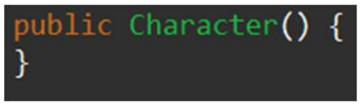


```
public static void main(String[] args) {
    Character player = new Character("Good Guy", 100, 100, 10, 5, 8);
    Character slowBaddie = new Character("Bad Guy", 30, 30, 2, 3, 10);
    Character fastBaddie = new Character("Bad Guy", 30, 30, 20, 5, 3);

    Character c = new Character();
```

43

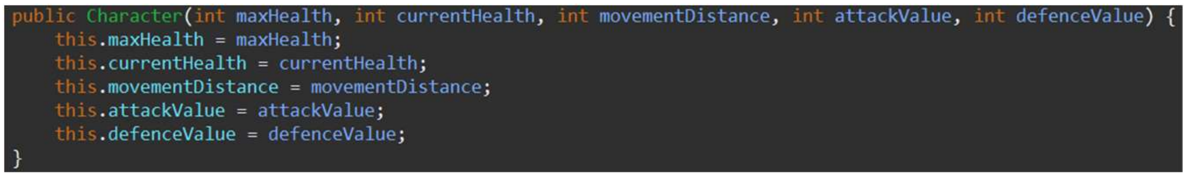
More Constructors

- With the previous issue, we can simply create a blank constructor inside the Character class.



```
public Character() {
}
```

- We can also create other Character class constructors, too, such as a Character without names.

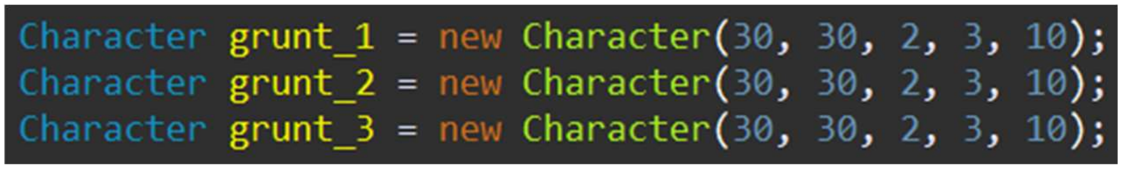


```
public Character(int maxHealth, int currentHealth, int movementDistance, int attackValue, int defenceValue) {
    this.maxHealth = maxHealth;
    this.currentHealth = currentHealth;
    this.movementDistance = movementDistance;
    this.attackValue = attackValue;
    this.defenceValue = defenceValue;
}
```

44

More Constructors

- The result will be “generic” characters that don’t have a name.

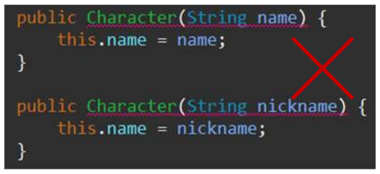


```java
Character grunt_1 = new Character(30, 30, 2, 3, 10);
Character grunt_2 = new Character(30, 30, 2, 3, 10);
Character grunt_3 = new Character(30, 30, 2, 3, 10);
```

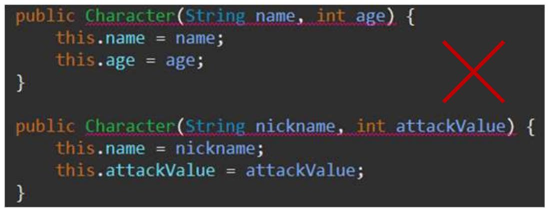
45

Multiple Constructors

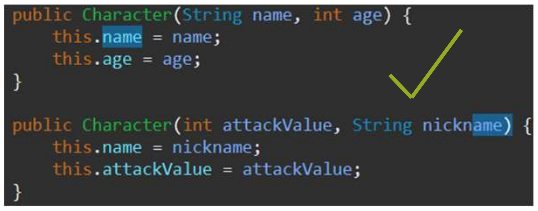
- We do have to be careful and consider how our multiple constructors need to be set up though. We are not able to create more than one constructor that takes in the same data types in the **same order!**



```java
public Character(String name) {
    this.name = name;
}

public Character(String nickname) {
    this.name = nickname;
}
```



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

public Character(String nickname, int attackValue) {
    this.name = nickname;
    this.attackValue = attackValue;
}
```



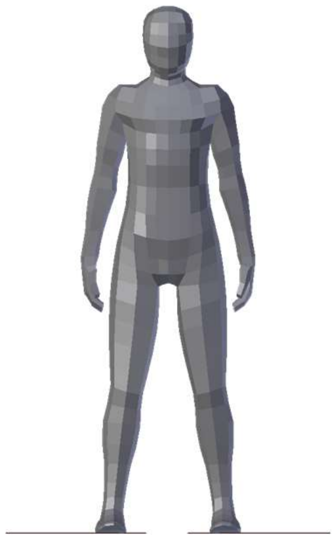
```java
public Character(String name, int age) {
    this.name = name;
    this.age = age;
}

public Character(int attackValue, String nickname) {
    this.name = nickname;
    this.attackValue = attackValue;
}
```

46

Task 4: Updating our class with Constructors

- Update your Character class to include another Constructor for a “generic character”.
- This constructor should have a **different number of values** to your original one.

Stretch Challenge: Create multiple constructors for different types of characters, NPCs, Player Characters, Enemies, etc



47

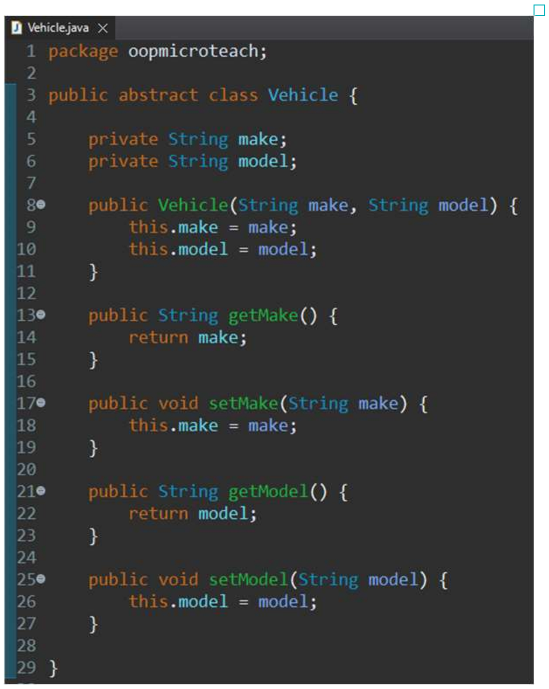
Abstraction and Inheritance

- These two concepts are closely linked.
- Abstraction allows us to create a class that **cannot be instantiated**. That is to say, we cannot make a copy of it with code such as:
  - Character c = new Character().
- This is because the **abstract class** is designed to be another kind of blueprint!

48

Abstract Class

- Consider the class to the right.
- It has all the same concepts as the Character class we were looking at earlier, except it has the **abstract** key word within the class definition.

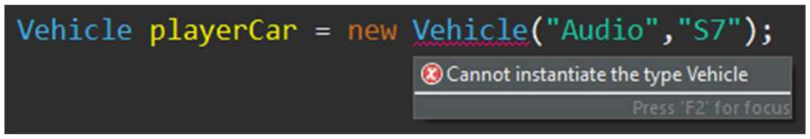


```java
package oopmicroteach;

public abstract class Vehicle {

    private String make;
    private String model;

    public Vehicle(String make, String model) {
        this.make = make;
        this.model = model;
    }

    public String getMake() {
        return make;
    }

    public void setMake(String make) {
        this.make = make;
    }

    public String getModel() {
        return model;
    }

    public void setModel(String model) {
        this.model = model;
    }

}
```

49

Abstract Class

- See how we're not able to create an instance of the class, even though we used the correct constructor.



```java
Vehicle playerCar = new Vehicle("Audio","S7");
```

Cannot instantiate the type Vehicle

Press 'F2' for focus

- This is because the abstract key word prevents us from doing this.

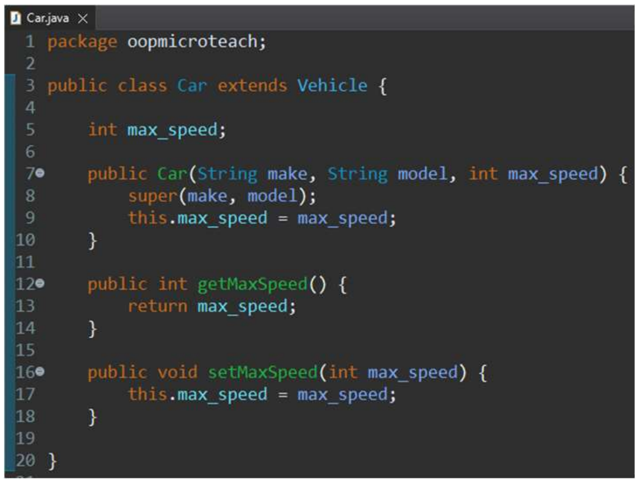
Stretch & Challenge

- Research more information on OOP abstraction

50

Inheritance

- The way we work with abstract classes is to inherit their properties into another class.
- The Car class to the right looks like any other class except for two concepts:
  - extends Vehicle
  - super()



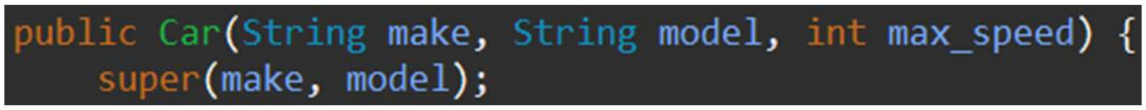
Car.java

```java
package oopmicroteach;

public class Car extends Vehicle {

    int max_speed;

    public Car(String make, String model, int max_speed) {
        super(make, model);
        this.max_speed = max_speed;
    }

    public int getMaxSpeed() {
        return max_speed;
    }

    public void setMaxSpeed(int max_speed) {
        this.max_speed = max_speed;
    }

}
```

51

extends and super()

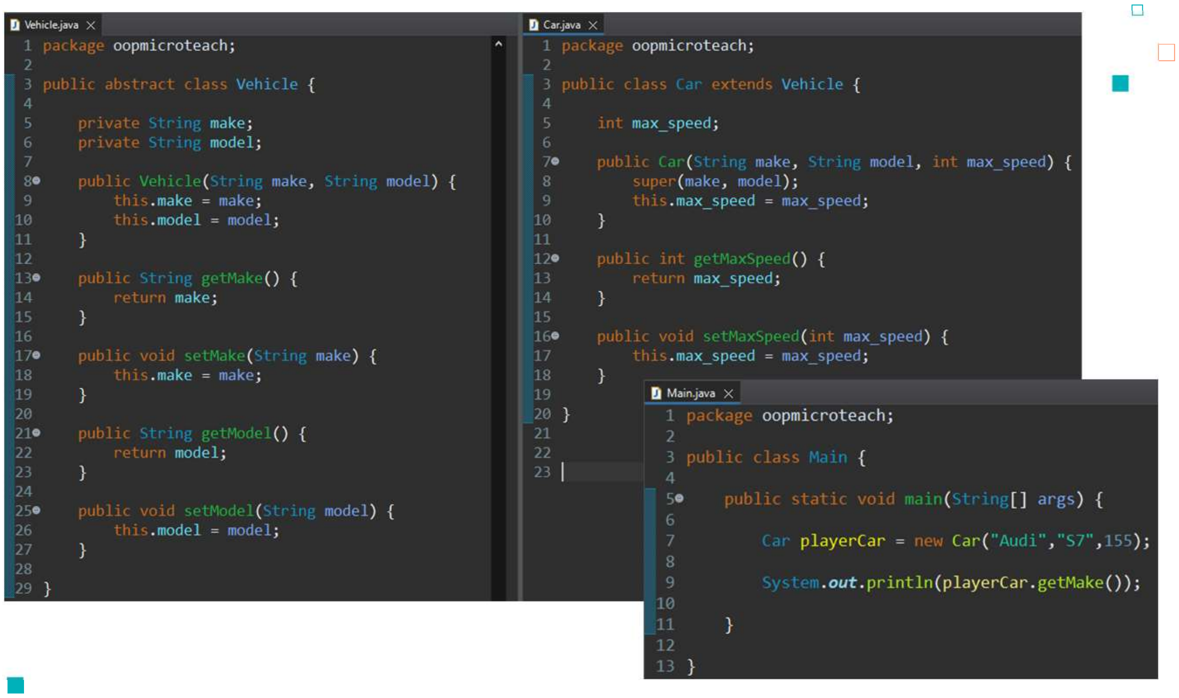
- Extending a class is telling Java that there is some functionality that needs to be taken from a **parent** class or the **super** class. The **child** class (in this case, car is the child, vehicle is the parent) must implement the 'make' and the 'model' from it's parent.
- We implement that within the constructor using super.



```java
public Car(String make, String model, int max_speed) {
    super(make, model);
```

- We still need to pass the make and model to the constructor, as is shown in the previous code, but instead of having to type in "this.make = make" and "this.model = model", we can just simple put "super(make, model)".

52



Vehicle.java

```java
package oopmicroteach;

public abstract class Vehicle {

    private String make;
    private String model;

    public Vehicle(String make, String model) {
        this.make = make;
        this.model = model;
    }

    public String getMake() {
        return make;
    }

    public void setMake(String make) {
        this.make = make;
    }

    public String getModel() {
        return model;
    }

    public void setModel(String model) {
        this.model = model;
    }

}
```

Car.java

```java
package oopmicroteach;

public class Car extends Vehicle {

    int max_speed;

    public Car(String make, String model, int max_speed) {
        super(make, model);
        this.max_speed = max_speed;
    }

    public int getMaxSpeed() {
        return max_speed;
    }

    public void setMaxSpeed(int max_speed) {
        this.max_speed = max_speed;
    }

}


```

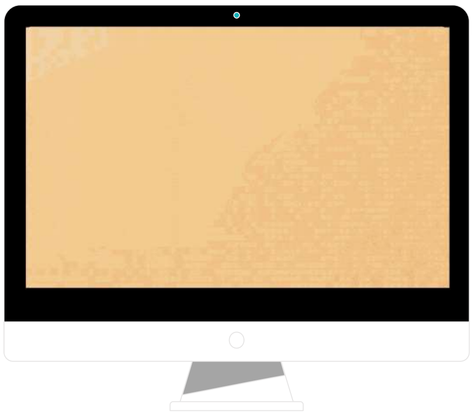
Main.java

```java
package oopmicroteach;

public class Main {

    public static void main(String[] args) {

        Car playerCar = new Car("Audi","S7",155);

        System.out.println(playerCar.getMake());

    }

}
```

53

Review

- What is the purpose of getters and setters and why is it the preferred way of accessing object information?



54