Day 2

What is an Object?

- A ‘thingy’ eg car, person, building, book
- Has
  - Properties/attributes
  - Behaviour
  - Events

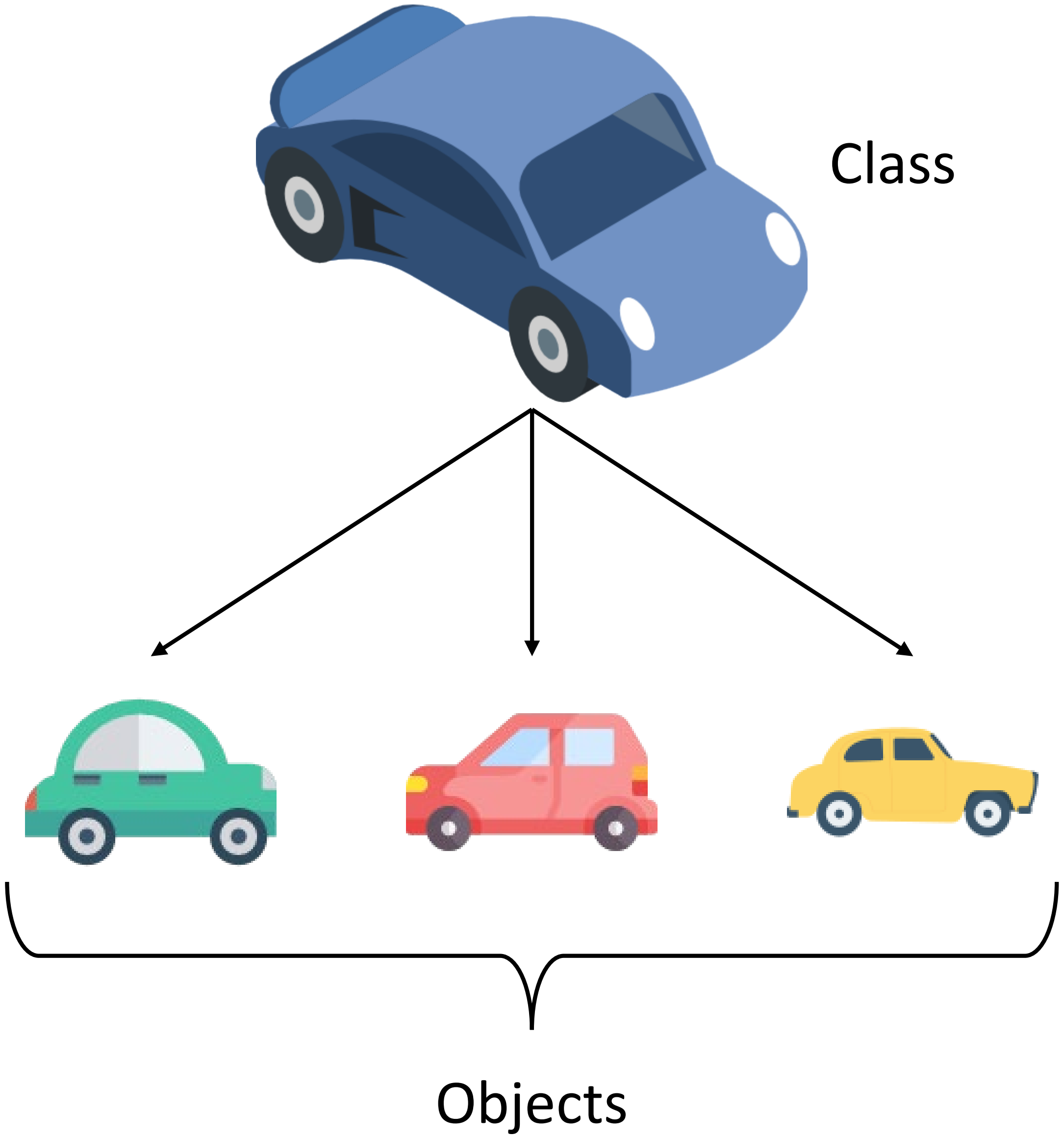
Properties
Colour: red
Make: Jeep
Engine size: 3L

Behaviour
Start
Stop
Turn left indicator
Reverse

Events
Low fuel
Car alarm

Class and Objects

- Class is a template for objects
- Objects are create from class
  - Created from the template
- Instantiation is the process of creating an object from a class
  - Objects are called instance of the class
- Object inherits all the properties, behaviour and events from the class
  - Attributes may be different between objects even though instantiated from the same class



Defining a Class

```
public class Car {
    private String color;
    private boolean started = false;

    public Car() { }

    public String getColor() { return this.color; }
    public void setColor(String color) { this.color = color; }
    public boolean isStarted() { return this.started; }

    public void start() {
        // Start the car
        this.started = true;
    }
    public void stop() {
        // Stop the car
        this.started = false;
    }
}
```

Class name

'Variables' for this class called members / attributes

Constructor is a special function with the same name as the class. Use to initialize the object when it is instantiated

Functions to read and update the members; called getters and setters or property assessors

`this` refers to the object

Functions for defining the class' behaviour; called methods

Instantiating an Object

```
public class Main {
    public static void main(String[] args) {
        Car fredCar = new Car();
        Car barneyCar = new Car();

        fredCar.setColor(“red”);
        barneyCar.setColor(“blue”);
        barneyCar.start();

        System.out.printf(“Has Fred started his car? %b\n”
            , fredCar.started);
        System.out.printf(“Has Barney started his car? %b\n”
            , barneyCar.isStarted());

    }
}
```

Instantiate a new `Car` object by calling the constructor with `new`

Set the color property for the respective car

Call a public method on the object

❌ Accessing started with result in compile time error

Accessing the Object

- Use the dot (.) to access public attributes and methods from an object
  - `fredCar.start()` - invoke the method start()
  - `fred.color` - get the value of the instance member, if it is accessible
- Constructor can only be used when instantiating an object
  - Use with the `new` keyword
  - Constructor has the same name as the class and has no return value

Constructor

- A special function/method that is called when an object is instantiated
- Special syntax
  - Should be the name of the class
  - Should not have return type
- Constructor may have parameters
  - A constructor without any parameter is called the default constructor
- Constructors are optional
  - JVM will provide a default constructor if it is not defined

Example - Constructor

```
pubic class Car {
   public Car() { }
   ...
}
```

Car class with a constructor. The constructor do not have any parameters

```
pubic class Car {
   ...
}
```

This is the same as above. Since no constructor is defined, the JVM will add a default constructor

```
pubic class Car {
   private String color;
   public Car(String color) {
      this.color = color;
   }
   ...
}
```

Constructor with one or more parameters. Used to initialize the object when it is intantiated

Encapsulation

- Hide implementation detail
  - eg. member name used to store the car's color
- Provide a higher level logical way of access/manipulating
  - eg. getter/setter methods
- Will not affect code when the implementation changes

`fredCar.getColor()` 👍

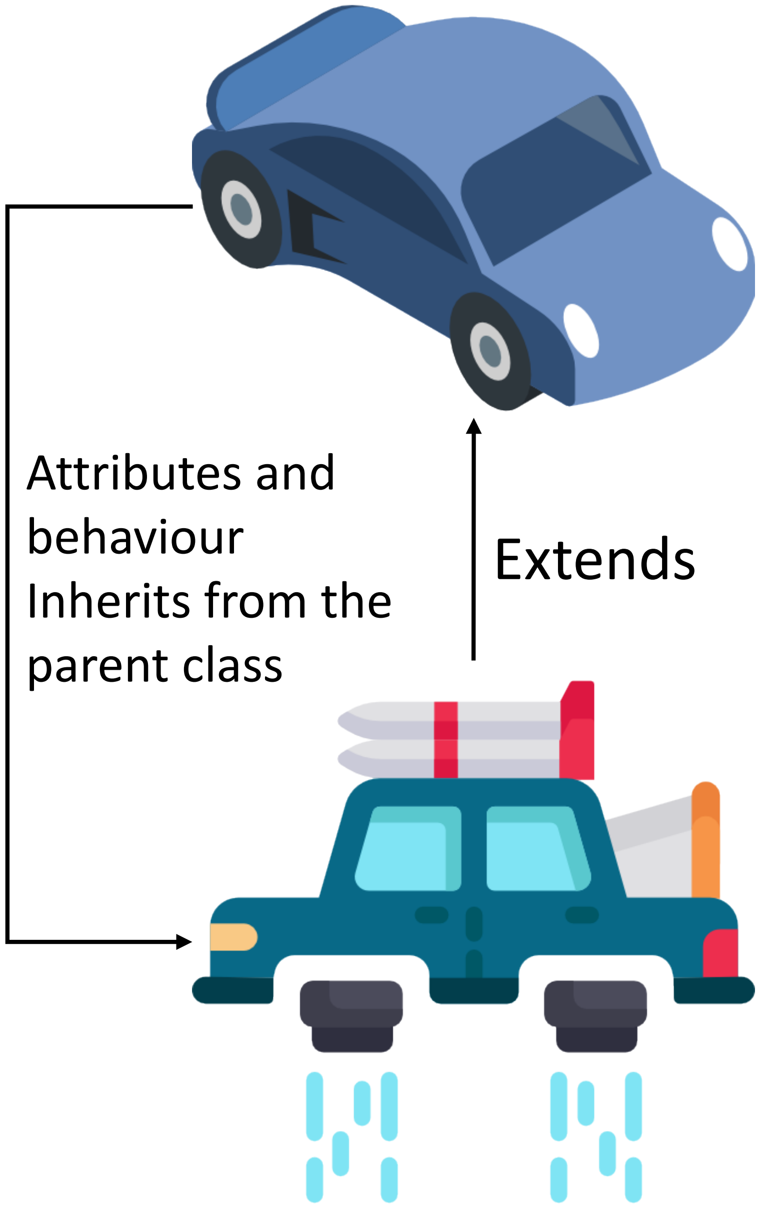
Will not affect the user if we change color member name to `colour` or the type from `String` to `java.awt.Color`. Will only impact the Car class

`fredCar.color` 👎

Changing the member name or the type will impact every class that references `color` member

Derived Classes

- A class that is created from an existing class
  - Also known as extending the class or inherit from the class
- Derive class inherits all the parent class’ properties and methods
- Derive class can only extend from one parent class
  - Single not multiple inheritance



Parent class

Attributes and behaviour Inherits from the parent class

Extends

`FlyingCar` is derived from **`Car`**

Derived Classes

- Derived classes can have their own properties and behaviours (methods)
  - Derived classes can be thought of as a specialized version of the parent class
- Override when the derive class has a method with the same signature as the parent
  - The derived class’ method will be used instead

Example - Derived Class

```
public class FlyingCar extends Car {

    private int altitude = 0

    public FlyingCar() { }
    public FlyingCar(String color) {
        this.setColor(color);
    }

    @Override()
    public String getColor() {
        return “Matte %s”.format(super.getColor());
    }

    public void climb(int height) {
        this.altitude += height;
    }
}
```

extends keyword to indicate that `FlyingCar` class is derived from `Car`

Additional attribute

Overload constructor to provide different instantiation options

Call `setColor()` to set the color attribute. Since `setColor()` is not overrode, use the ‘original’ implementation

Overriding the `getColor()` to provide new behaviour @Annotation to signal intention to override. Compiler will generate error if signature is incorrect

`super` to reference parent class `getColor()`

New behaviour in derived class

Working with Derived Classes

```
Car car = new FlyingCar(“red”);
```

Can assign a derived class instance to its parent class variable

```
FlyingCar car = new Car();
```

Derived class variable cannot reference parent class instances

```
if (car instanceof FlyingCar) {
   FlyingCar fly = (FlyingCar)car;
   fly.climb(10);
}
```

Use `instanceof` operator to check if a parent class is referencing a derived class instance

Cast allows us to treat car as a `FlyingCar` instance. Casting will fail if car is not an actual instance of `FlyingCar`

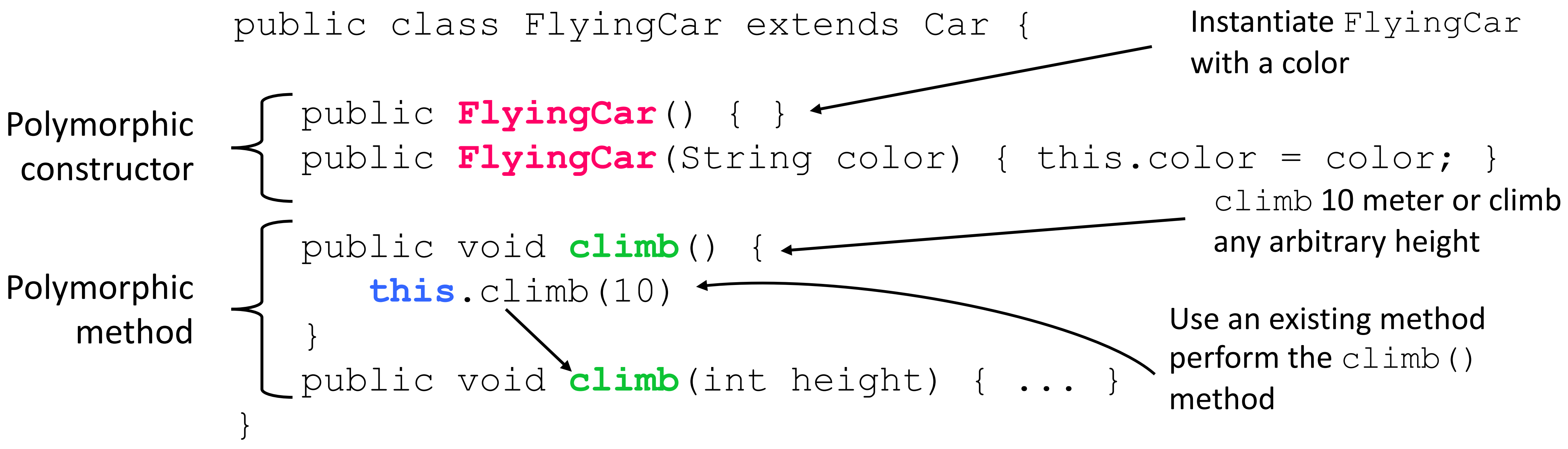
Example

What happens if we swap these 2 conditions?

```
public class Main {
   public static void whatIsMyType(Object obj) {
      if (obj instanceof FlyingCar)
         System.out.println(“This is a FlyingCar”);
      else if (obj instaoceof Car)
         System.out.println(“This is a Car”);
      else
         System.out.printf(“This is a %s class\n”
            , obj.getClass().getName());
   }

   public static void main(String[] args) {
      whatIsMyType(new Car());
      whatIsMyType(new FlyingCar());
      whatIsMyType(new JFrame());
   }
}
```

Overloading

- Ability to take many different forms, also known as polymorphism
- Ability for a method/function to have different parameters
    - Same method can be used in many different ways



```
public class FlyingCar extends Car {

    public FlyingCar() { }
    public FlyingCar(String color) { this.color = color; }

    public void climb() {
        this.climb(10)
    }
    public void climb(int height) { ... }
}
```

Polymorphic constructor

Polymorphic method

Instantiate `FlyingCar` with a color

`climb` 10 meter or climb any arbitrary height

Use an existing method perform the `climb()` method

Overriding

- Feature that allows a derived class to provide a more specific or different behaviour a method in the parent’s class
  - The override method must have the same method signature as that of the parent
  - As a safety measure, use the `@Override` annotation to declare your intention to override a method

```
public class FlyingCar extends Car {
    ...

    @Override()
    public String getColor() {
        return “Matte %s”.format(super.getColor());
    }
}
```

Overriding the default `getColor()` method from the parent class

Invoking the parent class’ `getColor()`

Overloading vs Overriding

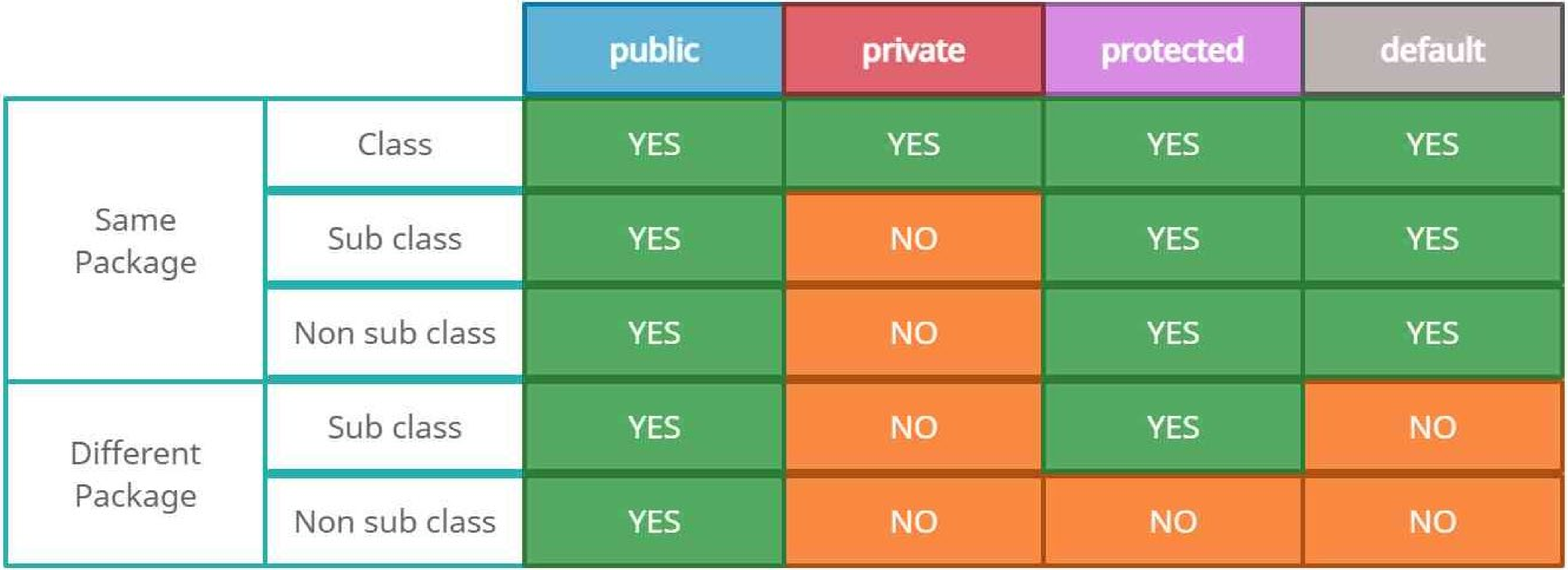
```
public class Car {

    public String getColor() { return this.color; }

}

public class FlyingCar extends Car {

    @Overload
    public String getColor() { ... }

    public void climb() { ... }
    public void climb(int height) { ... }
    public void climb(int height, int ceiling) { ... }
}
```

Override
same method signature

Overload
different method signature

Controlling Access to a Class

- Access modifiers controls who can access a member or method



		public	private	protected	default
Same Package	Class	YES	YES	YES	YES
	Sub class	YES	NO	YES	YES
	Non sub class	YES	NO	YES	YES
Different Package	Sub class	YES	NO	YES	NO
	Non sub class	YES	NO	NO	NO

Image from https://usemynotes.com/what-are-access-modifiers-in-java//

Interface

- All ‘startable’ objects have start and stop button
- Users of these objects do not need to know how the start process works
  - The implementation details
- Users expect a start and stop button on these object

Starts all these

Example

Add new method for every ‘startable’ object

```
public class Car {
    public void start() { ... }
}

public class WashingMachine {
    public void start() { ... }
}

public class K8SCluster {
    public void start() { ... }
}

public class Timer {
    public void start() { ... }
}

public class Book { }
```

```
public class Person {
    public void startCar(Car car) {
        car.start();
    }
    public void statWashingMachine(
            WashingMachine wm) {
        wm.start();
    }
    public void startK8SCluster(
            K8SCluster k8s) {
        k8s.start();
    }
    public void startTimer(Timer timer) {
        timer.start();
    }

    public void startBook(Book book) {
        book.start();
    }
}
```

Compile time error
There is no guarantee that a class has the `start()` method

Example

```
public interface Startable {
    public void start();
    public void stop();
}
```

Every class that implements an interface must provide implementation for all the methods in that interface - like a contract between the class and the interface

```
public class Car implements Startable {
    public void start() { ... }
    public void stop() { ... }
    ...
}
```

```
public class K8SCluster implements Startable {
    public void start() { ... }
    public void stop() { ... }
    ...
}
```

```
public class WashingMachine implements Startable {
    public void start() { ... }
    public void stop() { ... }
    ...
}
```

```
public class Timer implements Startable {
    public void start() { ... }
    public void stop() { ... }
    ...
}
```

Example

```
public class Person {
    public start(Startable startable) {
        startable.start();
    }
    ...
}
```

One method that will accept any `Startable` implementation as its parameter

```
Person fred = new Person();

Car car = new Car();
WashingMachine washingMachine = new WashingMachine();
K8SCluster k8sCluster = new K8SCluster();
Timer timer = new Timer();

fred.start(car);
fred.start(washingMachine);
fred.start(k8sCluster);
fred.start(time);
```

Implements the `Startable` interface

Example - Collection

Interface

Implementation

```
List<String> todos = new LinkedList<>();
Set<String> pokemon = new HashSet<>();
Map<String, Integer> inventory = new TreeMap<>();
```

```
public void printList(List<String> list) {
    for (String item: list)
        System.out.println(item);
}
```

Print any list of string

```
public void printList(LinkedList<String> list) {
    for (String item: list)
        System.out.println(item);
}
```

Only prints linked list of string

Difference Between Class and Interface

a. Can only extend a single class

b. Can have concrete methods

  - Classes with only method signature are called abstract class

c. Methods and members can be public, private or protected

d. Can be instantiated

a. Can implement multiple interfaces

b. Only contain method signature

  - Not concrete methods

c. Methods and members (static) can only have public access

d. Cannot be instantiated

Generics

- A language feature that allows you to define classes or interfaces with parametrized type
    - Allow you to state the type when instantiating the object
- Use `Object` as type when required to hold any instance
    - Object is the parent class of all classes
- Not type safe
    - What is the type of `getItem()`

```
public class Box {
    private Object item;
    public void setItem(Object item) {
        this.item = item;
    }
    public Object getItem() {
        return this.item;
    }
    public boolean isEmpty() {
        return Objects.isNull(this.item);
    }
}
```

Example - Generic Type

Placeholder for the actual type

```
public class Box<T> {
   private T item;
   public void setItem(T item) {
      this.item = item;
   }
   public T getItem() {
      return this.item;
   }
   public boolean isEmpty() {
      return Objects.isNull(this.item);
   }
}
```

Use the placeholder T
in all declarations

Example - Instantiating Generic Classes

Specify the type when instantiating the object

```
Box<String> boxOfStrings = new Box<String>();
Box<Cookie> boxOfCookies = new Box<>();
```

Can leave the type empty on the RHS - type inference

✓
```
boxOfStrings.setItem(“Hello Fred”);
```

```
boxOfStrings.setItem(123);
```
✗

```
Box<Object> box = new Box<>();
```

```
box.setItem(“hello, world”);
```
✗