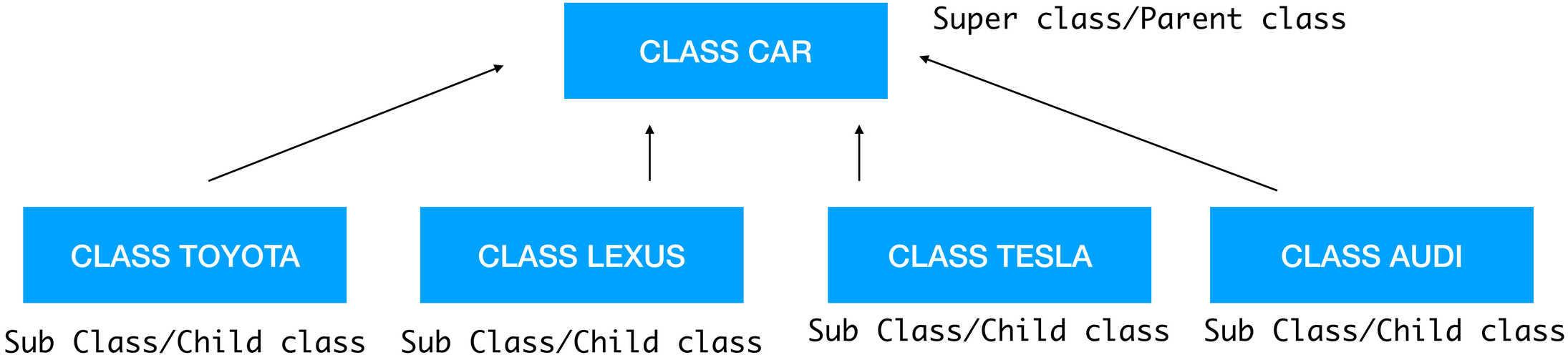


Car IS-A Object

Toyota IS-A Car

Lexus IS-A Car

Tesla IS-A Car

Audi IS-A Car

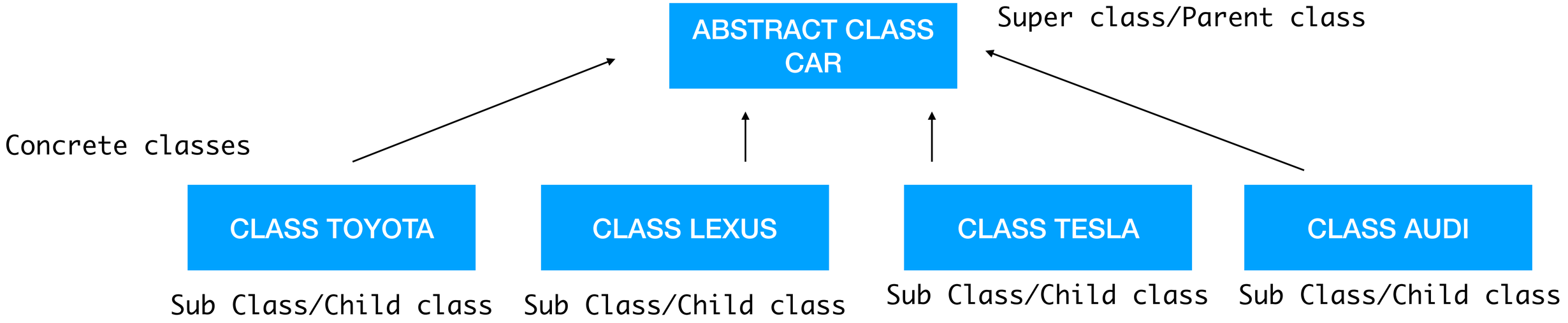
INSTANTIATION OPTIONS

REGULAR, REFERENCE/DATA TYPE AND OBJECT TYPE ARE SAME

```
Car car = new Car();
Toyota toy = new Toyota();
Lexus le = new Lexus();
Tesla te = new Tesla();
Audi au = new Audi();
```

POLYMORPHISM: REFERENCE/DATA TYPE IS PARENT CLASS AND OBJECT TYPE IS CHILD CLASS TYPE

```
Object car = new Car();
Car toy = new Toyota();
Car le = new Lexus();
Car te = new Tesla();
Car au = new Audi();
Audi a = new Car();//ERROR
Audi a = new Toyota();//ERROR
```



INSTANTIATION OPTIONS

Car IS-A Object

Toyota IS-A Car

Lexus IS-A Car

Tesla IS-A Car

Audi IS-A Car

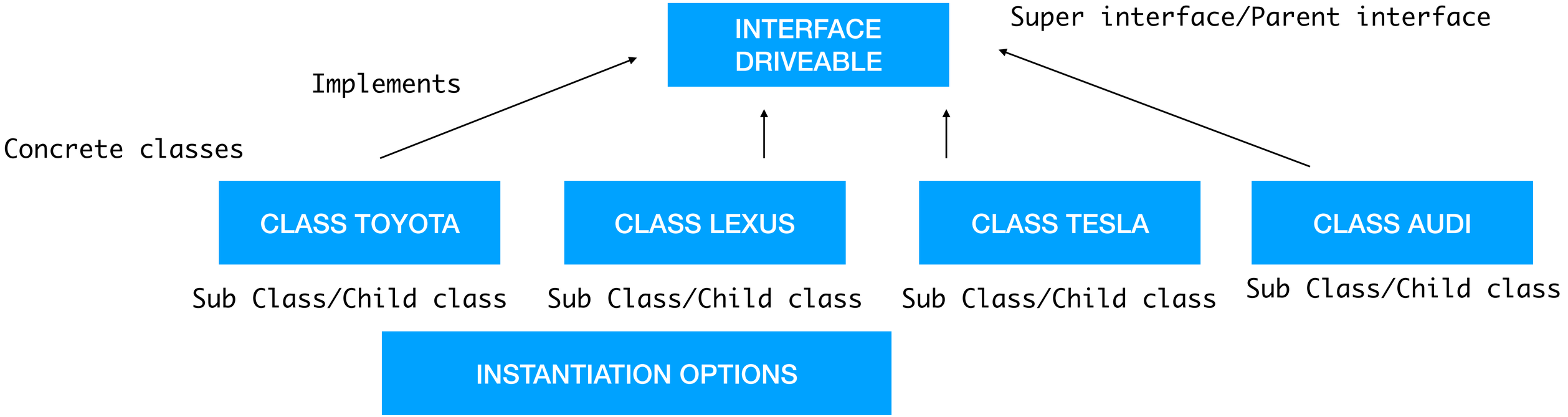
REGULAR, REFERENCE/DATA TYPE AND OBJECT TYPE ARE SAME

```
Car car = new Car();//ERROR CANNOT INST
Toyota toy = new Toyota();
Lexus le = new Lexus();
Tesla te = new Tesla();
Audi au = new Audi();
```

POLYMORPHISM: REFERENCE/DATA TYPE IS PARENT CLASS AND OBJECT TYPE IS CHILD CLASS TYPE

```
Object car = new Car();//ERROR
Car toyo = new Toyota();
Car le = new Lexus();
Car te = new Tesla();
Car au = new Audi();
```

Abstract class can be reference/ data type for a variable. But u cant create object from it



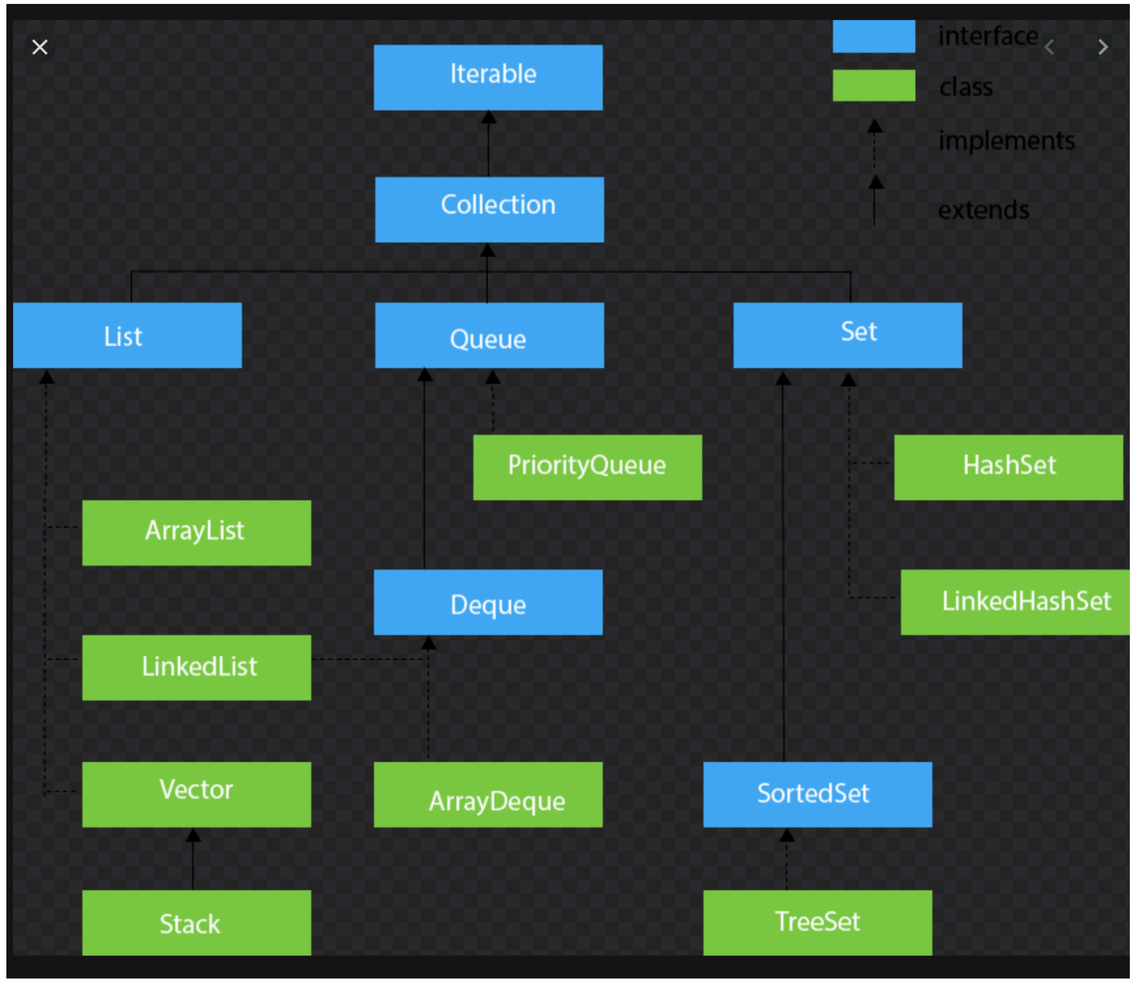
REGULAR, REFERENCE/DATA TYPE AND OBJECT TYPE ARE SAME

```
Driveable dr = new Driveable();//ERROR CANNOT INST
Toyota toy = new Toyota();
Lexus le = new Lexus();
Tesla te = new Tesla();
Audi au = new Audi();
```

POLYMORPHISM: REFERENCE/DATA TYPE IS PARENT INTERFACE AND OBJECT TYPE IS CHILD CLASS TYPE

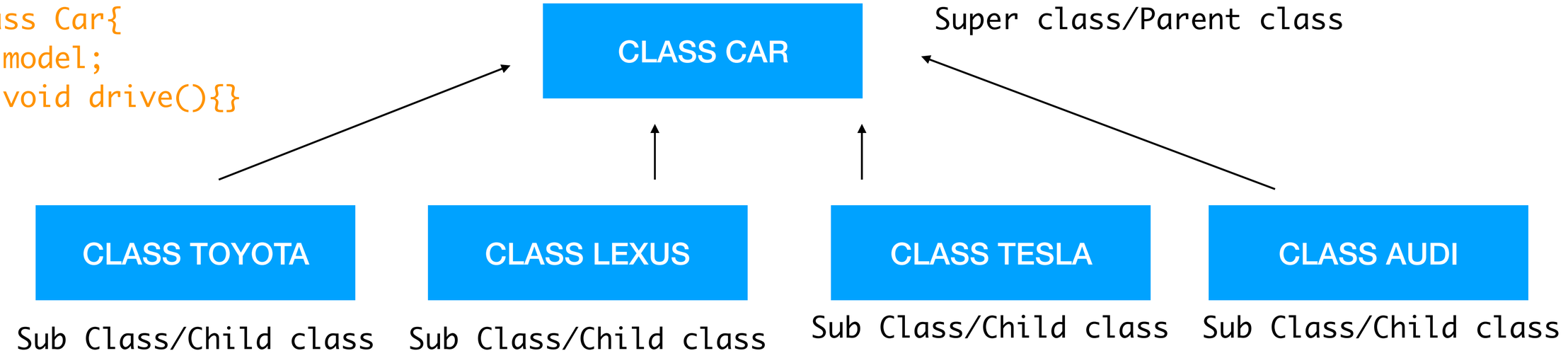
```
Object car = new Driveable();//ERROR
Driveable toyo = new Toyota();
Driveable le = new Lexus();
Driveable te = new Tesla();
Driveable au = new Audi();
Interface can be reference/
data type for a variable. But u cant
create object from it
```

INSTANTIATION OPTIONS FOR ARRAYLIST




```
Normal instantiation:
ArrayList<Integer> list = new ArrayList<>();

ERROR instantiation:
List<Interger> nums = new List<>(); NOO, ERROR!!!

Polymorphic instantiation:

List<Integer> nums = new ArrayList<>();
Collection<Integer> nums = new ArrayList<>();
Iterable<Integer> nums = new ArrayList<>();

Other possible options for a list:
List<Double> prices = new LinkedList<>();
List<Double> prices = new Vector<>();
List<Double> prices = new Stack<>();
```

```
public class Car{
    String model;
    public void drive(){}
}
```




```
public class Toyota extends Car{
    double price;
    public void start(){}
}
```

```
Car car = new Car();
car.model="BMW 530i";
car.drive();
car.start();ERROR
```

```
Toyota corolla = new Toyota();
corolla.model="Corolla SE";
corolla.price = 19345.99;
corolla.drive();
corolla.start();
```

```
public class Lexus extends Car{
    double price;
    public void start(){}
}
```

ONLY Reference type variables and methods can be accessed, when using polymorphism!

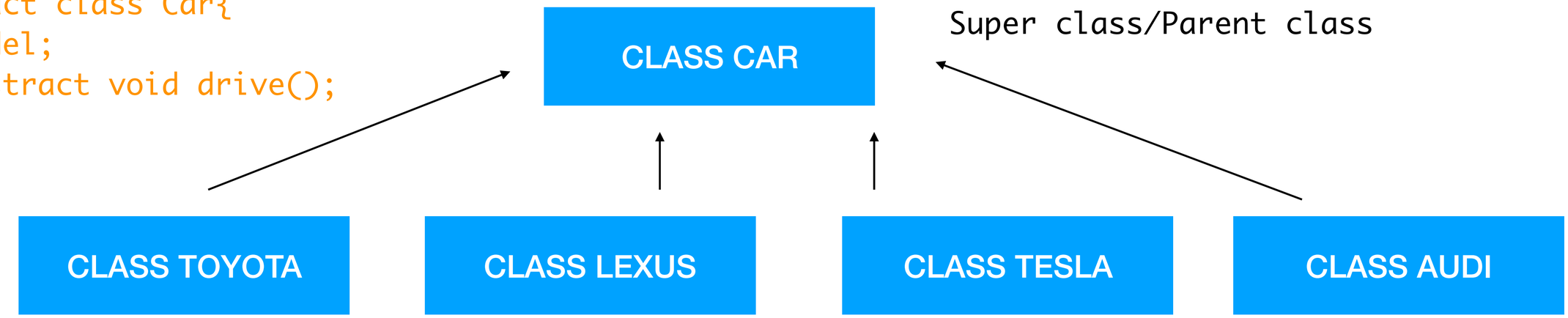
```
Car car = new Lexus();
car.model = "LFA";
car.drive();
car.price = 375000.99;ERROR
car.start(); ERROR
```

```
public class Tesla extends Car{
    double price;
    public void start(){}
}
```

Reference type decides what is accessible.

```
Car tesla  = new Tesla();
tesla.model = "Model X";
tesla.drive();
tesla.price = 375000.99;ERROR
tesla.start(); ERROR
```

```
public abstract class Car{
    String model;
    public abstract void drive();
}
```




```
public class Toyota extends Car{
    double price;
    public void start(){}
    public void drive(){}
}
```

```
public class Lexus extends Car{
    double price;
    public void start(){}
    public void drive(){}
}
```

```
public class Tesla extends Car{
    double price;
    public void start(){}
    public void drive(){}
}
```

```
Car car = new Car();
car.model=“BMW 530i”;
car.drive();
car.start();ERROR
```

ONLY Reference type variables and methods can be accessed, when using polymorphism!

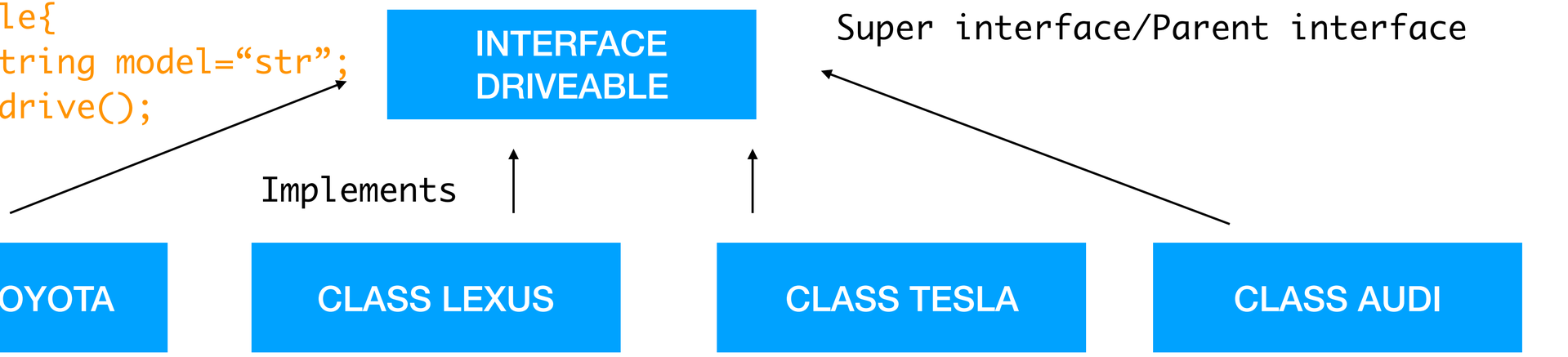
Reference type decides what is accessible.

```
Toyota corolla = new Toyota();
corolla.model=“Corolla SE”;
corolla.price = 19345.99;
corolla.drive();
corolla.start();
```

```
Car car = new Lexus();
car.model = “LFA”;
car.drive();
car.price = 375000.99;ERROR
car.start(); ERROR
```

```
Car tesla  = new Tesla();
tesla.model = “Model X”;
tesla.drive();
tesla.price = 375000.99;ERROR
tesla.start(); ERROR
```

```
public interface Driveable{
    public static final String model="str";
    public abstract void drive();
}
```




```
public class Toyota implements Driveable{
    double price;
    public void start(){}
    public void drive(){}
}
```

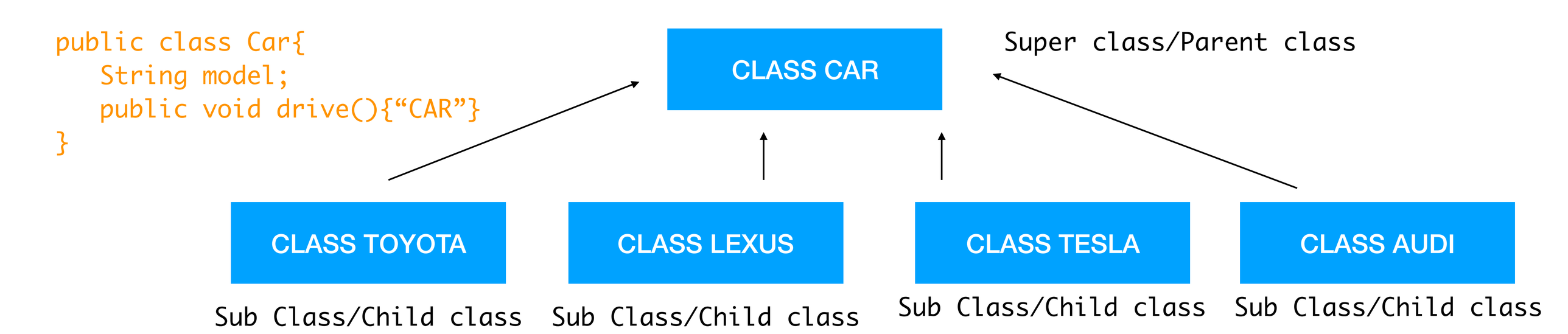
```
public class Tesla implements Driveable{
    double price;
    public void start(){}
    public void drive(){}
}
```

ONLY Reference type variables and methods can be accessed, when using polymorphism!

Reference type decides what is accessible.

```
Toyota corolla = new Toyota();
corolla.model="Corolla SE"; ERROR.
Final variable cannot be changed
corolla.price = 19345.99;
corolla.drive();
corolla.start();
```

```
Driveable corolla = new Toyota();   System.out.println(Driveable.model);
corolla.drive();
ERROR:
corolla.price = 19345.99; ERRROR
corolla.start(); ERROR
```




```
public class Toyota extends Car{
   double price;
   public void start(){}
   public void drive(){“TOYOTA”}
}
```

```
public class Lexus extends Car{
   double price;
   public void start(){}
   public void drive(){“LEXUS”}
}
```

```
public class Tesla extends Car{
   double price;
   public void start(){}
}
```

```
Car car = new Car();
car.model=“BMW 530i”;
car.drive(); “CAR”
car.start();//ERROR
```

```
Toyota corolla = new Toyota();
corolla.model=“Corolla SE”;
corolla.price = 19345.99;
corolla.drive(); “TOYOTA”
corolla.start();
```

ONLY Reference type variables and methods can be accessed, when using polymorphism! Overridden version from sub class will be called

```
Car car = new Lexus();
car.model = “LFA”;
car.drive(); “LEXUS”
car.price = 375000.99;ERROR
car.start(); ERROR
```

Reference type decides what is accessible.

```
Car tesla = new Tesla();
tesla.model = “Model X”;
tesla.drive(); “CAR”
tesla.price = 375000.99;ERROR
tesla.start(); ERROR
```

```
public class Shape{
   public String type;
   public Shape(){
      type = “shape”;
   }
   public void draw(){
     Print “shape: *********”
   }
}
```

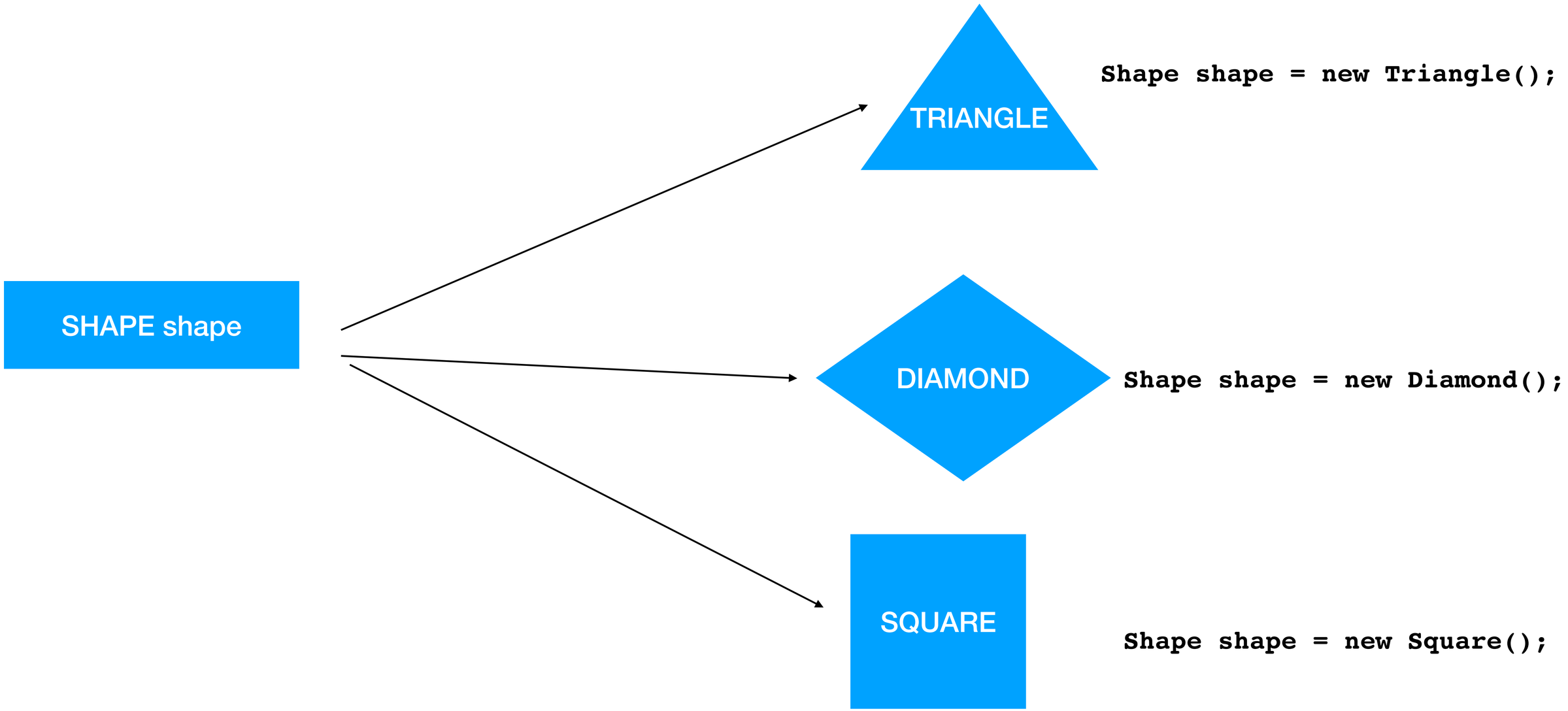
CLASS SHAPE

SQUARE

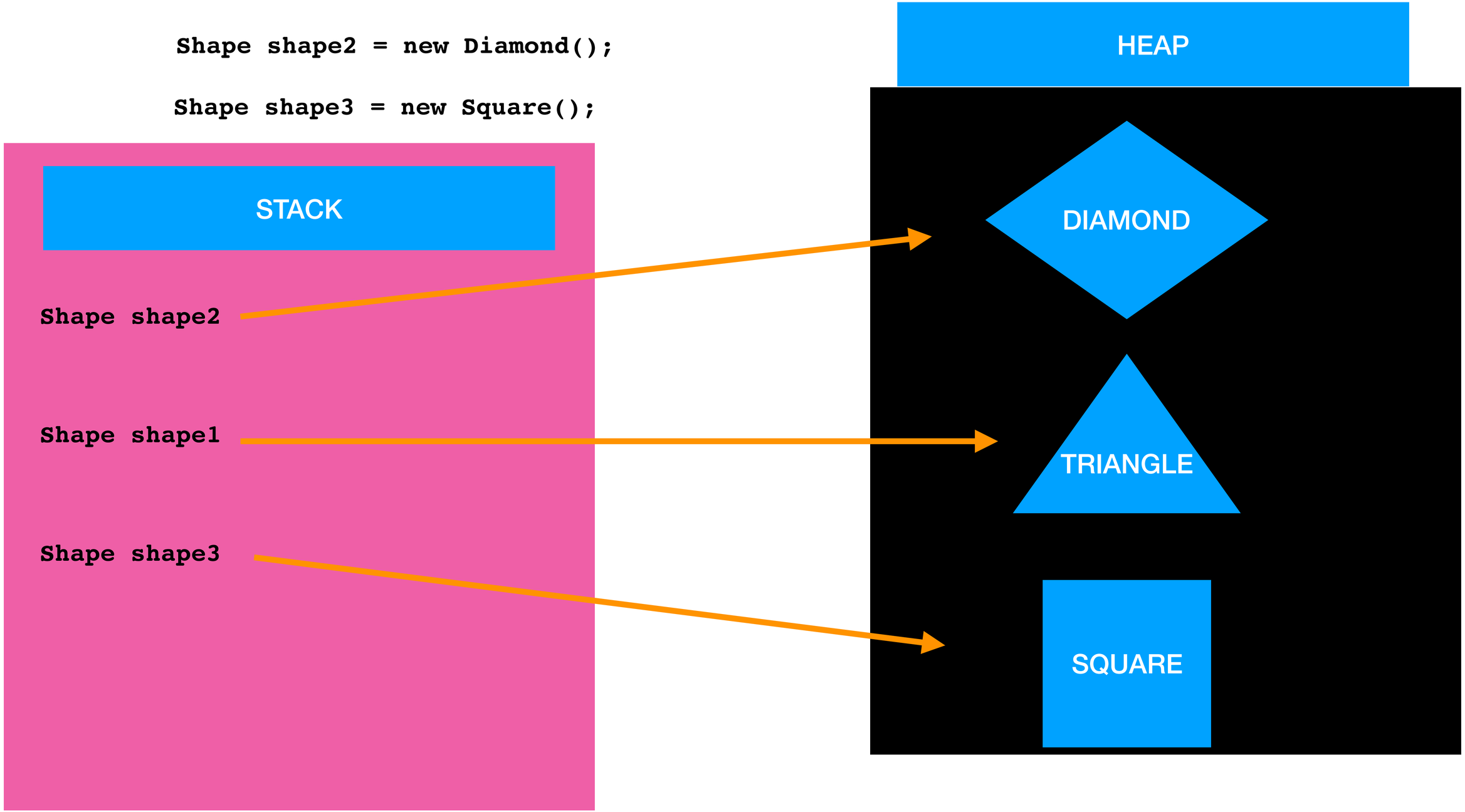
TRIANGLE

DIAMOND

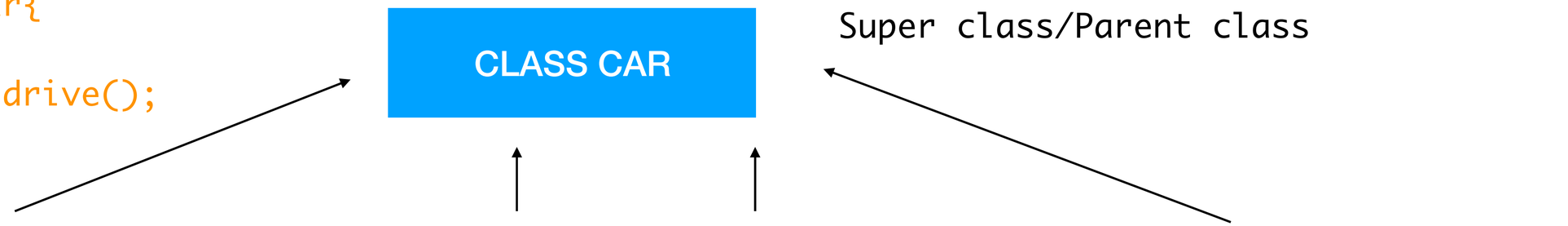
```
public class Square extends Shape{
  public Square(){
    type = “square”;
  }
  @Override
  public void draw(){
     Print”square: * * * * * * “

  }
}
```


SHAPE shape
TRIANGLE
DIAMOND
SQUARE
Shape shape = new Triangle();
Shape shape = new Diamond();
Shape shape = new Square();

```
Shape shape1 = new Triangle();

Shape shape2 = new Diamond();

Shape shape3 = new Square();
```



```
public abstract class Car{
    String model;
    public abstract void drive();
}
```



Sub Class/Child class

```
public class Toyota extends Car{
    double price;
    public void start(){}
    public void drive(){}
}
```

Sub Class/Child class

```
public class Lexus extends Car{
    double price;
    public void start(){}
    public void drive(){}
}
```

Sub Class/Child class

```
public class Tesla extends Car{
    double price;
    public void start(){}
    public void drive(){}
}
```

```
Car car = new Car();
car.model=“BMW 530i”;
car.drive();
car.start();ERROR
```

ONLY Reference type variables and methods can be accessed, when using polymorphism!

Reference type decides what is accessible.

```
Toyota corolla = new Toyota();
corolla.model=“Corolla SE”;
corolla.price = 19345.99;
corolla.drive();
corolla.start();
```

```
Car car = new Lexus();
car.model = “LFA”;
car.drive();
((Lexus)car).price = 375000.99;
((Lexus)car).start();
```

DOWNCASTING TO LEXUS

```
Car tesla  = new Tesla();
tesla.model = “Model X”;
tesla.drive();
((Tesla)tesla).price = 375000.99;
((Tesla)tesla).start();
```

DOWNCASTING TO TESLA

```
Square[] squareArr = new Square[2];
squareArr[0] = new Square();
squareArr[1] = new Square();
```


```
squareArr[0].draw();
squareArr[1].draw();
```

POLYMORPHISM WITH ARRAY

```
Shape[] shapeArr = new Shape[4];
shapeArr[0] = new Shape();
shapeArr[1] = new Square();
shapeArr[2] = new Triangle();
shapeArr[3] = new Diamond();
```

```
for(int i=0; i<shapeArr.length;i++){
   shapeArr[i].draw();
}

for(Shape eachShape : shapeArr){
     eachShape.draw();
}
```

```
List<Square> squareList = new ArrayList<>();
squareList.add(new Square());
squareList.add(new Square());
```


```
squareList.get(0).draw();
squareList.get(1).draw();
```

POLYMORPHISM WITH ARRAYLIST

```
List<Shape> shapeList = new ArrayList<>();
shapeList.add(new Shape());
shapeList.add(new Square());
shapeList.add(new Triangle());
shapeList.add(new Diamond());
```

```
for(int i=0; i<shapeList.size();i++){
   shapeList.get(i).draw();
}

for(Shape eachShape : shapeList){
     eachShape.draw();
}
```

READING OBJECT TYPE

```
Shape shape1 = new Triangle();

Shape shape2 = new Diamond();

Shape shape3 = new Square();
```

GETCLASS().GETSIMPLENAME()

```
System.out.println(shape1.getClass().getSimpleName());
Prints: Triangle

System.out.println(shape2.getClass().getSimpleName());
Prints: Diamond

System.out.println(shape3.getClass().getSimpleName());
Prints: Square
```

INSTANCEOF OPERATOR WITH IF

```
if(shape3 instanceof Square ){
  System.out.println("Square object"));
}
```