

OBJECT-ORIENTED PROGRAMMING Using JAVA

INHERITANCE AND POLYMORPHISM

QUAN THAI HA

HUS

OCTOBER 14, 2024



1 Inheritance

- Inheritance and Visibility
- Inheritance and Constructors
- Java.lang.Object
- Casting

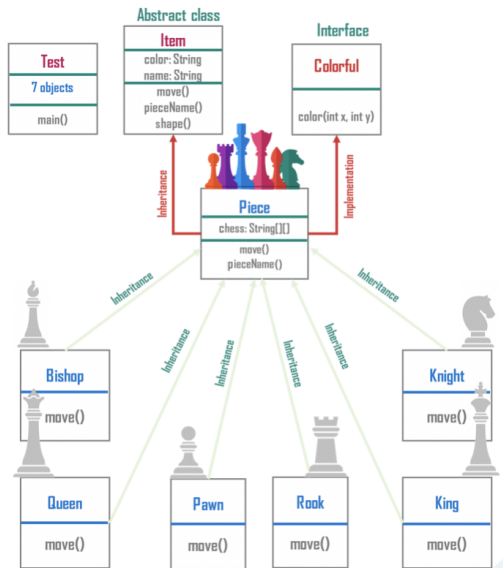
2 Polymorphism

3 References

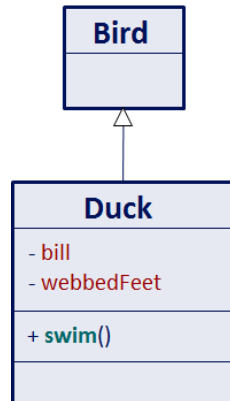
- They all have a shape, name, color, move.
- Can they share the same code?



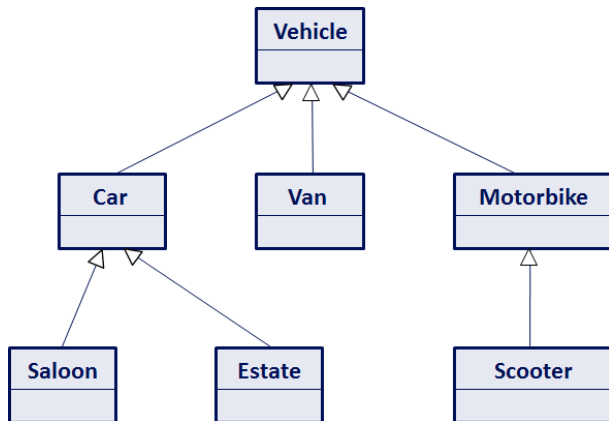
- If we have several descriptions with some commonality between these descriptions, we can group the descriptions and their commonality using inheritance to provide a compact representation of these descriptions.
- The object-oriented programming approach allows us to group the commonalities and create classes that can describe their differences from other classes.
- Frequently, a class is merely a modification of another class. Inheritance allows minimal repetition of the same code.



- Humans use the concept of inheritance in categorising objects and descriptions. For example, you may have answered the question - "What is a duck?", with "a bird that swims", or even more accurately, "a bird that swims, with webbed feet, and a bill instead of a beak". So we could say that a **Duck** is a **Bird** that swims.
- The figure illustrates the inheritance relationship between a **Duck** and a **Bird**. In effect we can say that a **Duck** IS-A special type of **Bird**.
- A new design created by changing an existing design.

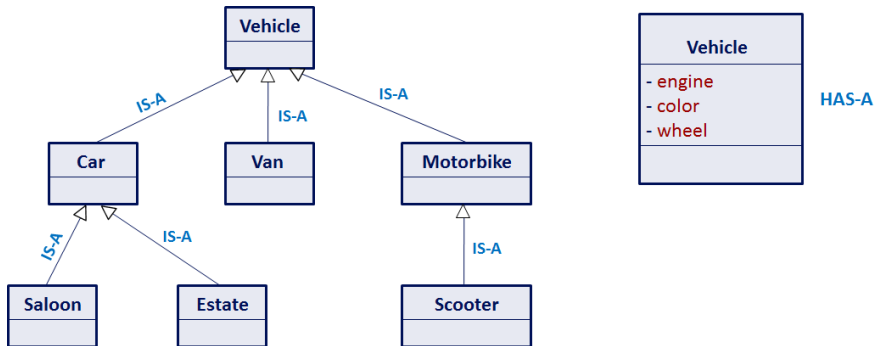


- For example, if we were to be given an unstructured group of descriptions such as **Car**, **Saloon**, **Estate**, **Van**, **Vehicle**, **Motorbike** and **Scooter**, and asked to organise these descriptions by their differences.
- You might say that a **Saloon** car **IS-A** **Car** but has a long boot, whereas an **Estate** Car **IS-A** **Car** with a very large boot, etc.



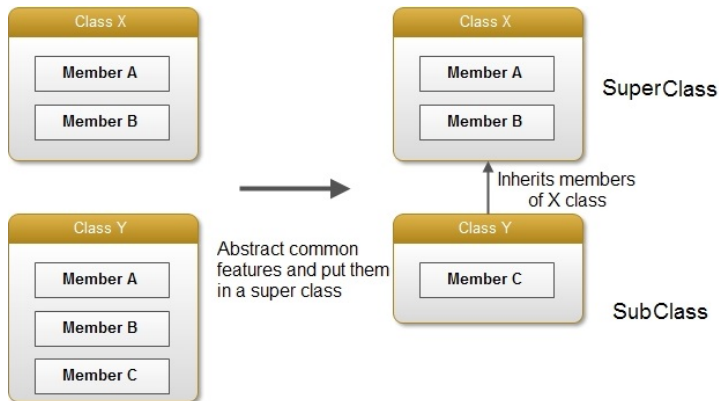
An example of organizing objects using inheritance.

- Using inheritance, we can **extend the functionalities of a class** in another class.
- Using inheritance, we can **define class in hierarchical order**.
- One way to determine that you have organised your classes correctly is to check them using the "IS-A" and "HAS-A" relationship checks. It is easy to confuse objects within a class and children of classes when you first begin programming with an OOP methodology.

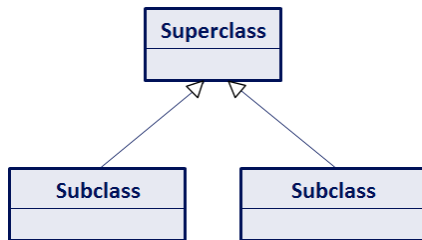


- The **IS-A** relationship describes the inheritance.
- We can say, "A **Car IS-A Vehicle**" and "A **Saloon Car IS-A Car**", so all relationships are correct.
- The **HAS-A** relationship describes the composition (or aggregation) of a class. So we can say "An **Vehicle HAS-A a Engine**", or "An **Engine, Color and Wheels IS-A-PART-OF a Vehicle**". This is the case even though an **Engine** is also a class! where there could be many different descriptions of an **Engine - Petrol, Diesel**, etc.
- So, using inheritance the programmer can:
 - ▶ Inherit a behaviour and add further specialised behaviour - for example, a **Car IS-A Vehicle** with the addition of four **Wheel** objects, **Seats**, etc.
 - ▶ Inherit a behaviour and replace it - for example, the **Saloon Car** class will inherit from **Car** and provide a new "boot" implementation.
 - ▶ Cut down on the amount of code that needs to be written and debugged - for example, in this case only the differences are detailed, a **Saloon Car** is essentially identical to the **Car**, with only the differences requiring description.

- The idea of inheritance is simple but powerful. When you want to create a new class and there is already a class that includes some of the code that you want, you can derive your new class from the existing class. In doing this, you can reuse the fields and methods of the existing class without having to write (and debug!) them yourself.



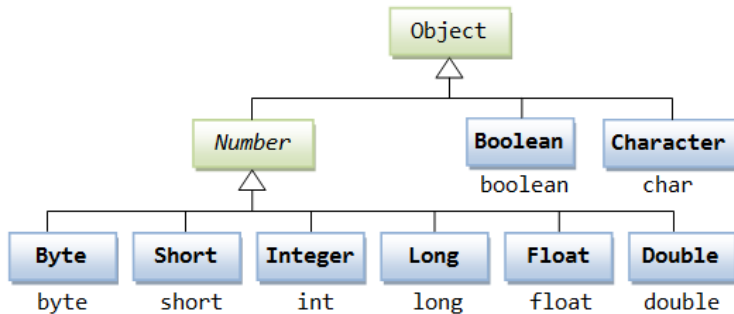
- Inheritance is a mechanism by which one object acquire some or all properties of another class.
- The class that inherits the properties is known as the **subclass** (also **derived class**, **extended class**, or **child class**). The class from which the properties are inherited is known as the **superclass** (also a **base class** or a **parent class**).
- The inheriting class **contains all the attributes and methods of the class it inherited from**.
- The inheriting class can **define additional attributes and methods**.
- The inheriting class can **override the definition of existing methods** by providing its own implementation.



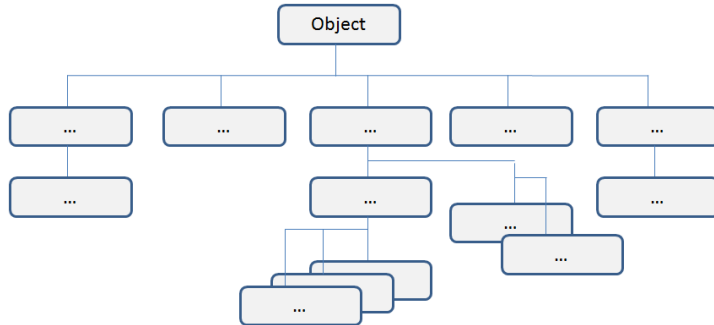
Illustrates the relationship between a superclass and a subclass. A subclass inherits from a superclass.

- Class one above
 - ▶ Parent class, Base class
- Class one below
 - ▶ Child class, Derived class
- Class one or more above
 - ▶ Superclass, Ancestor class, Base class
- Class one or more below
 - ▶ Subclass, Descendent class

- Designing number system in Java.



- Excepting **Object**, which has no superclass, every class has **one and only one direct superclass** (single inheritance). In the absence of any other explicit superclass, every class is implicitly a subclass of **Object**.
- Classes can be derived from classes that are derived from classes that are derived from classes, and so on, and ultimately derived from the topmost class, **Object**. Such a class is said to be descended from all the classes in the inheritance chain stretching back to **Object**.





```
1 public class Employee {  
    // The "protected" keyword allows subclass to access the attributes directly.  
3     protected String name;  
    protected int salary;  
5  
    // Constructors  
7     public Employee(String name, int startingSalary) {  
        this.name = name;  
9        salary = startingSalary;  
    }  
11  
    public void setSalary(int newSalary) {  
13        salary = newSalary;  
    }  
15  
    public void increaseSalary(int increment) {  
17        salary += increment;  
    }  
19  
    public void printInfo() {  
21        System.out.print("Employee:" + name)  
        System.out.print("Salary:" + salary);  
23    }  
}
```



```
public class Programmer extends Employee { // The "extends" keyword indicates inheritance.
2 // The Programmer subclass adds one field
  private Set<String> languages;
4
  public Programmer(String name, int startingSalary, Set<String> languages) {
6    super(name, startingSalary);
    this.languages = languages;
8  }

10 // The Programmer subclass adds one method
  public void addLanguage(String language) {
12    languages.add(language);
  }
14 }

16 // Creating an instance of Programmer
  Programmer programmer = new Programmer("Developer", 100, "Assembly");
18
  // Call method that inherited from superclass Employee
20 programmer.increaseSalary(10);

22 // Call method that added to subclass Programmer
  programmer.addLanguage("Java");
```



```
1 public class Programmer extends Employee {  
    private Set<String> languages;  
3  
    public Programmer(String name, int startingSalary, Set<String> languages) {  
5        super(name, startingSalary);  
        this.languages = languages;  
7    }  
9  
    public void addLanguage(String language) {  
        languages.add(language);  
11    }  
13    @Override  
    public void printInfo() {  
15        System.out.print("Employee:" + name);  
        System.out.print("Salary:" + salary);  
17        System.out.print(languages.toString());  
    }  
19 }
```




```
1 public class Programmer extends Employee {  
    private Set<String> languages;  
3  
    public Programmer(String name, int startingSalary, Set<String> languages) {  
6        super(name, startingSalary);  
        this.languages = languages;  
7    }  
9  
    public void addLanguage(String language) {  
11        languages.add(language);  
13  
    public void languagesInfo() {  
        System.out.print(languages.toString());  
15    }  
17  
    @Override  
    public void printInfo() {  
19        super.printInfo();  
        languagesInfo();  
21    }  
}
```

- Sometimes we need to modify the inherited method
 - ▶ to change/extend the functionality.
 - ▶ As you already know, this is called **method overriding**.
- For example, in the **Programmer** class:
 - ▶ The **printInfo()** method inherited from **Employee** should be modified to include the information of programming languages.
- To override an inherited method:
 - ▶ Simply recode the method in the subclass using the same **method header** and return type (the same **prototype**).
 - ▶ Method header refers to the name and parameters type of the method (also known as **method signature**).

1

super can be used to refer immediate parent class instance variable.



2

super can be used to invoke immediate parent class method.



3

super () can be used to invoke immediate parent class constructor.





```
public class Programmer extends Employee {  
2   private Set<String> languages;  
  
4   public Programmer(String name, int startingSalary, Set<String> languages) {  
    // It must be the first statement to call superclass' constructor.  
6     super(name, startingSalary);  
    this.languages = languages;  
8   }  
  
10  public void languagesInfo () {  
    System.out.print(languages.toString());  
12  }  
  
14  @Override  
    public void printInfo () {  
16      super.printInfo();  
        languagesInfo();  
18  }  
}
```

■ Inherits

- ▶ attributes
 - name, salary
- ▶ methods
 - setSalary(), increaseSalary()

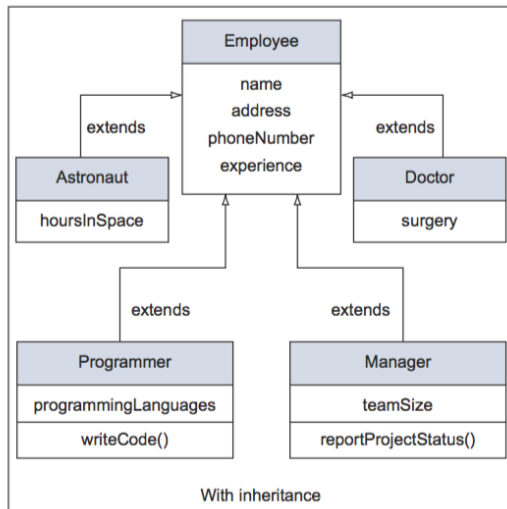
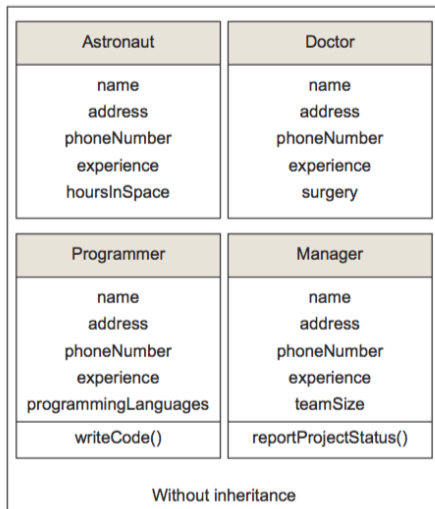
■ Adds

- ▶ attributes
 - languages
- ▶ methods
 - addLanguage(), languagesInfo()

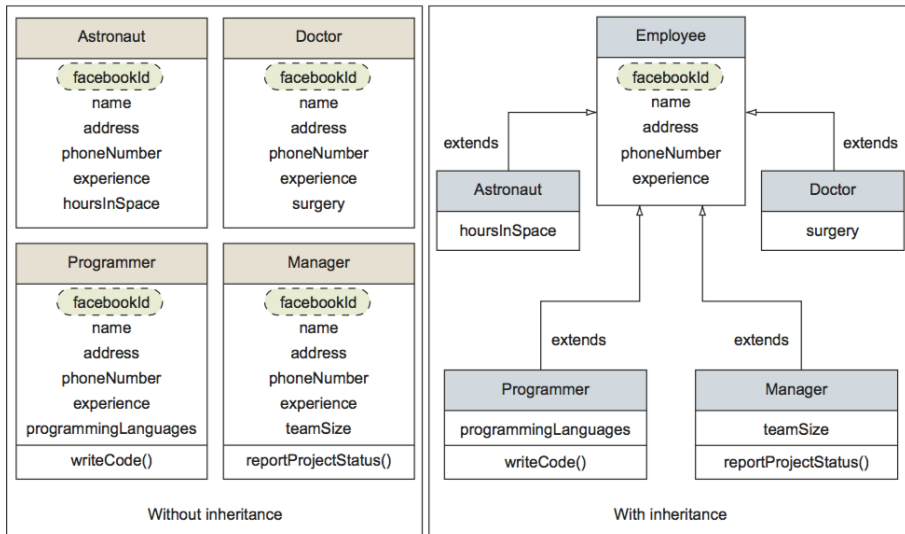
■ Modifies (overrides)

- ▶ methods
 - printInfo()

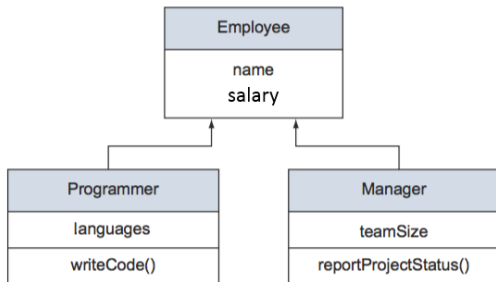
■ Less code redundancy:



■ Ease of Modification:



- **Compatible data types:** Inheritance enables an **IS-A** relationship. In an **IS-A** relationship, an object of a derived class also can be treated as an object of its base class.



```
1 public class Employee {
2     protected String name;
3     protected int salary;
4 }
5
6 public class Programmer extends Employee {
7     private Set<String> languages;
8 }
9
10 public class Manager extends Employee {
11     private int teamSize;
12 }
13
```


- Whenever a superclass object is expected, a subclass object is acceptable as substitution!
 - ▶ The reverse is NOT true (Eg: A Programmer is an Employee; but an Employee may not be a Programmer).
- Hence, all existing functions that works with the superclass objects will work on subclass objects with no modification!



```
1 public class HumanResources {  
    private List<Employee> employees;  
3    // Very difficult to manage if we use:  
    // private List<Programmer> programmers;  
5    // private List<Manager> managers;  
  
7    public HumanResources() { ... }  
  
9    private void addEmployee(Employee employee) { ... }  
  
11   private void sendInvitation(Employee employee) { ... }  
  
13   public void inviteProgrammer(Programmer programmer) {  
        // Superclass object is expected, a subclass  
15        // object is acceptable as substitution.  
        addEmployee(programmer);  
17        sendInvitation(programmer);  
    }  
19  
    public void inviteManager(Manager manager) {  
21        addEmployee(manager);  
        sendInvitation(manager);  
23    }  
}
```

1 Inheritance

- Inheritance and Visibility
- Inheritance and Constructors
- Java.lang.Object
- Casting

2 Polymorphism

3 References

	Method in the same class	Method of another class in the same package	Method of subclass	Method of another public class in the outside world
<i>private</i>	✓			
<i>package</i>	✓	✓		
<i>protected</i>	✓	✓	✓	
<i>public</i>	✓	✓	✓	✓



```
public class Car {  
2   private String licensePlate;  
   private boolean isOn;  
4  
   public void turnOn() {  
6       ...  
   }  
8  
   public void turnOff() {  
10      ...  
   }  
12 }  
  
14 public class SelfDrivingCar extends Car {  
   void print() {  
16       System.out.println(licensePlate);    // Do not work! Not visible!  
       System.out.println(getLicensePlate()); // Work!  
18   }  
}
```



```
1 public class Car {  
    String licensePlate; // default access modifier  
3     boolean isOn;  
  
5     public void turnOn() {  
        ...  
7     }  
  
9     public void turnOff() {  
        ...  
11    }  
    }  
13  
    public class SelfDrivingCar extends Car {  
15        void print() {  
            // Works if Car and SelfDrivingCar are located within the same package.  
17            System.out.println(licensePlate);  
            }  
19    }
```



```
1 public class Car {  
    protected string licensePlate;  
3    protected boolean isOn;  
  
5    public void turnOn() {  
        ...  
7    }  
  
9    public void turnOff() {  
        ...  
11   }  
}  
13  
class SelfDrivingCar extends Car {  
15     void print() {  
        // Works anyway!  
17         System.out.println(licensePlate);  
        }  
19 }
```

1 Inheritance

- Inheritance and Visibility
- Inheritance and Constructors
- Java.lang.Object
- Casting

2 Polymorphism

3 References

- Since each subclass “contains” an instance of the parent class, the latter **must be initialized**.
- Java compiler automatically calls the **default constructor (no params!)** of the parent class. The call is inserted as the **first statement** of each child constructor.
- If parent class disabled default constructor (by defining others) **parent constructor must be called explicitly!**



```
1 public class Car {  
    protected String licensePlate;  
3    protected boolean isOn;  
  
5    // Default constructor enabled!  
    }  
7  
    public class SelfDrivingCar extends Car {  
9        private boolean isSelfDriving;  
  
11       // Default constructor enabled!  
        }  
13  
  
15 Car sdCar = new SelfDrivingCar(); // Works!
```



```
1 public class Car {  
    protected boolean isOn;  
3    protected String license;  
  
5    // Default constructor enabled!  
    }  
7  
    public class SelfDrivingCar extends Car {  
9        private boolean isSelfDriving;  
  
11       // Default constructor disabled!  
        public SelfDrivingCar() {  
13            // Automatic call to parent default constructor here!  
        }  
15    }  
  
17 Car sdCar = new SelfDrivingCar(); // Works!
```



```
1 public class Car {  
    protected String licensePlate;  
3    protected boolean isOn;  
  
5    // Default constructor disabled!  
    public Car(String licensePlate, boolean isOn) {  
6        this.licencePlate = licensePlate;  
7        this.isOn = isOn;  
9    }  
}  
11  
12 public class SelfDrivingCar extends Car {  
13     private boolean isSelfDriving;  
  
15     // Default constructor disabled!  
    public SelfDrivingCar() {  
17         // Automatic call to parent default constructor here!  
18     }  
19 }  
  
21  
    Car sdCar = new SelfDrivingCar(); // Not working!
```



```
public class Car {
2   protected String licensePlate;
   protected boolean isOn;
4
   public Car(String licensePlate, boolean isOn) { // Default constructor disabled!
6       this.licence = license;
       this.isOn = isOn;
8   }
}
10
public class SelfDrivingCar extends Car {
12   private boolean isSelfDriving;
14
   // Default constructor disabled!
   public SelfDrivingCar(boolean isOn, String license, boolean isSelfDriving) {
16       super(licencePlate, isOn);
       this.isSelfDriving = isSelfDriving;
18   }
}
20
22 Car sdCar = new SelfDrivingCar(); // Works!
```

- **this** is a reference to the current object.
- **super** is a reference to the parent class.
- **super()** calls the default constructor of parent class.
- **super(params)** calls other constructors of parent class.
 - ▶ **Must be the first statement** in child constructors.

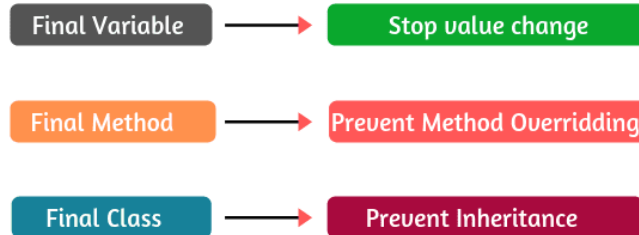
- Execution of constructors proceeds top-down along the inheritance hierarchy.
- In this way, when a method of the child class is executed (constructor included), the superclass is completely initialized already.



```
1 public class Car {  
2     public Car() {  
3         super();  
4         System.out.println("Car");  
5     }  
6 }  
  
8 public class SelfDrivingCar extends Car {  
9     public SelfDrivingCar() {  
10        super();  
11        System.out.println("SelfDrivingCar");  
12    }  
13 }  
14  
15 // Which output?  
16 SelfDrivingCar sdCar = new SelfDrivingCar();
```

- Sometimes, we want to prevent inheritance by another class (eg: to prevent a subclass from corrupting the behaviour of its superclass): use the **final** keyword on class.
- Sometimes, we want a class to be inheritable, but want to prevent some of its methods to be overridden by its subclass: use the **final** keyword on the particular method.

Java **Final** Keyword



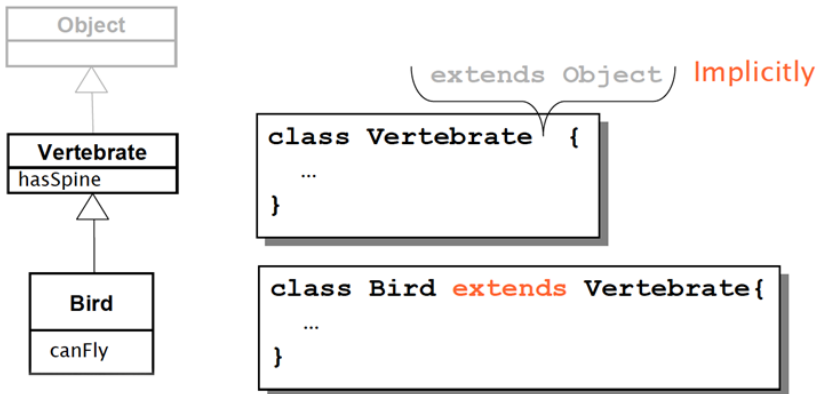
1 Inheritance

- Inheritance and Visibility
- Inheritance and Constructors
- `Java.lang.Object`
- Casting

2 Polymorphism

3 References

- `java.lang.Object`
- All classes are subtypes of Object



- All objects can be seen as **Object** instances.
- **Object** defines **basic services**, which are useful for all classes. They are often overridden in sub-classes. For example:
 - ▶ **toString()**: returns a string representation
 - ▶ **equals(Object o)**: tests content equality
 - ▶ **clone()**: returns a shallow copy of the object
- **Object.toString()**:



```

1 public String toString () {
    return getClass ().getName ()+"@"+Integer.toHexString (hashCode () );
3 }
    
```

- Every class in Java is a child of the **Object** class either directly or indirectly. And since the **Object** class contains a `toString()` method, we can call `toString()` on any instance and get its string representation.
- Whenever we print an object reference, `println()` invokes the `toString()` method internally.
- If we don't define a `toString()` method in our class, then `Object.toString()` is invoked.



```
public class Car {  
2   private boolean isOn;  
   private String brand;  
  
4  
   public Car(String brand) {  
6       this.isOn = false;  
       this.brand = brand;  
8   }  
  
10  public static void main(String[] args) {  
    Car car = new Car("Audi");  
12    System.out.println(car);  
    // Car@31f1e5  
14  }  
}
```



```
public class Car {  
2   private boolean isOn;  
   private String brand;  
4  
   public Car(String brand) {  
6       this.isOn = false;  
       this.brand = brand;  
8   }  
  
10  @Override  
   public String toString() {  
12      return "Car[" + "isOn=" + isOn +  
          ", brand='" + brand + '\'' + " ]";  
14  }  
}  
16  
Car car = new Car("Audi");  
18 System.out.println(car); // Car[isOn=false , brand='Audi ']
```

- We can use `==` operators for reference comparison (address comparison) and `.equals()` method for content comparison.
- In simple words, `==` checks if both objects point to the same memory location whereas `equals()` evaluates to the comparison of values in the objects.
- If a class does not override the equals method, then by default, it uses the `equals(Object o)` implementation of the closest parent.



```
public class Car {  
2   private boolean isOn;  
   private String brand;  
  
4  
   public Car(String brand) {  
6       this.isOn = false;  
       this.brand = brand;  
8   }  
   ...  
10 }  
  
12 String s1 = new String("BMW");  
   String s2 = new String("BMW");  
14 Car c1 = new Car("BMW");  
   Car c2 = new Car("BMW");  
16 System.out.println(s1 == s2);           // false  
   System.out.println(s1.equals(s2));      // true  
18 System.out.println(c1 == c2);           // false  
   System.out.println(c1.equals(c2));      // false
```

- You must override `hashCode()` in every class that overrides `equals()`.
- Failure to do so will result in a violation of the general contract for `Object.hashCode()`, which will prevent your class from functioning properly in conjunction with all hash-based collections, including `HashMap`, `HashSet`, and `HashTable`. (J. Bloch)



```
1 public class Car {  
    private boolean isOn;  
3    private String licence;  
  
5    public Car(boolean isOn, String licence) {  
        this.isOn = isOn;  
7        this.licence = licence;  
    }  
}
```



```
// If not overridden, equals() behaves like the == operator
2  @Override
   public boolean equals(Object obj) {
4      if (this == obj)
          return true;

6      if (obj == null || getClass() != obj.getClass())
8          return false;

10     Car car = (Car) obj;
    return (isOn == car.isOn) && Objects.equals(licence, car.licence);
12 }

14 // hashCode() must be overridden as well
   @Override
16 public int hashCode() {
    return Objects.hash(isOn, licence);
18 }
}
```

- Whenever `hashCode()` is invoked on the same object more than once during an execution of a Java application, the `hashCode()` method must consistently return the same integer, provided no information used in equals comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.
- If two objects are equal according to the `equals(Object)` method, then calling the `hashCode()` method on each of the two objects must produce the same integer result.
- It is not required that if two objects are unequal according to the `equals(java.lang.Object)` method, then calling the `hashCode()` method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hashtables.

- Constructors are a good way for generating objects which are copies of other objects.
- If proper constructors are not available, the method `clone()` can be used instead.
- `clone()` returns an **Object** reference, thus casting is necessary to use the cloned object.



```
1 Point p1 = new Point(2,3);  
  Point p2 = new Point(p1);  
3 p2.setLocation(4,4);  
  System.out.println(p1);    // 2, 3  
5 System.out.println(p2);    // 4, 4  
  
7 Point p1 = new Point(2, 3);  
  Point p2 = (Point) p1.clone();  
9 p2.setLocation(4, 4);  
  System.out.println(p1);    // 2, 3  
11 System.out.println(p2);    // 4, 4
```

- When JVM is called for cloning, it does the following things:
 - ▶ If the class has only primitive data type members then a completely new copy of the object will be created and the reference to the new object copy will be returned.
 - ▶ If the class contains members of any class type then **only the object references to those members are copied and hence the member references in both the original object as well as the cloned object refer to the same object (this is called shallow copy).**



```
1 ArrayList<Point> l1 = new ArrayList<>();  
  l1.add(new Point(1, 1));  
3 l1.add(new Point(2, 2));  
  l1.add(new Point(3, 3));  
5  
  ArrayList<Point> l2 = (ArrayList) l1.clone();  
7 l2.get(0).setLocation(9, 9);  
9 System.out.println(l1);  
  // [java.awt.Point[x=9,y=9], java.awt.Point[x=2,y=2], java.awt.Point[x=3,y=3]]  
11 System.out.println(l2);  
  // [java.awt.Point[x=9,y=9], java.awt.Point[x=2,y=2], java.awt.Point[x=3,y=3]]
```

1 Inheritance

- Inheritance and Visibility
- Inheritance and Constructors
- Java.lang.Object
- Casting

2 Polymorphism

3 References

- Java is a **statically typed** language.
- Both references and objects have a type.

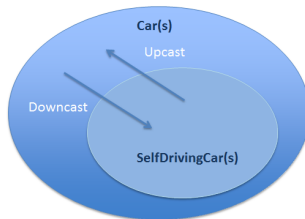


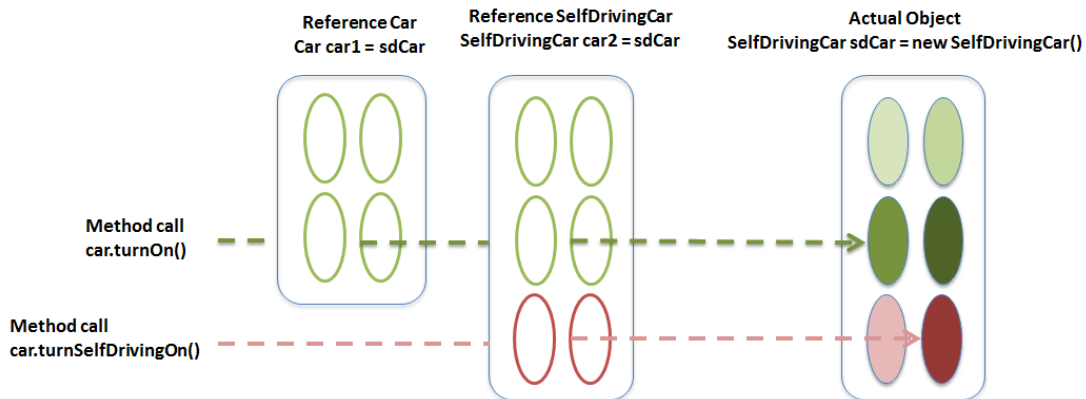
```
float f;  
2 f = 4.7;    // OK!  
  f = "hello!";    // Compile-time error  
4  
Car car;  
6 car = new Car();    // OK!  
  car = new String();    // Compile-time error
```

- Specialization defines a subtyping relationship (IS-A).
- All **SelfDrivingCar(s)** are **Car(s)**. Not all **Car(s)** are **SelfDrivingCar(s)**.
- **Upcasting** and **Downcasting** refer to the possibility of changing the reference type of a given object.
- **Upcasting** consists in using references associated with a more generic type (less methods), while **Downcasting** with more specific type (more methods).



```
1 class Car {};  
  class SelfDrivingCar extends Car {};  
3  
  Car c1 = new Car(); // OK!  
5 SelfDrivingCar c2 = new SelfDrivingCar();  
  // OK!  
7  
  // But also SelfDrivingCar is-a Car  
9 Car c3 = new SelfDrivingCar();
```





- Upcasting refers to the use of a reference which is more general (closer to Object type) than the actual type of the object itself.
- Note that, **references** and **objects** are **separate concepts**. Object referenced by 'car' continues to be of **SelfDrivingCar** type! Only the reference (i.e., the external interface) changes!
- It is dependable: It is always true that an **SelfDrivingCar** **IS-A** **Car** too.
- It is automatic: **Car car = new SelfDrivingCar();**



```
class Car {};  
2 class SelfDrivingCar extends Car {};  
  Car car = new SelfDrivingCar();
```



```
1 public class Car {  
    protected String licensePlate;  
3    protected boolean isOn;  
  
5    public void turnOn() {  
        ...  
7    }  
  
9    public void turnOff() {  
        ...  
11   }  
12 }  
13  
14 public class SelfDrivingCar extends Car {  
15     private boolean isSelfDriving;  
  
17     public void turnSelfDrivingOn() {  
        ...  
19     }  
  
21     public void turnSelfDrivingOff() {  
        ...  
23     }  
24 }
```



```
1 SelfDrivingCar sdCar = new SelfDrivingCar();  
sdCar.turnSelfDrivingOn(); // OK!  
3  
4 // Upcast  
5 Car car = sdCar;  
  
7 // Compile time error!  
8 // turnSelfDrivingOn method is hidden from Car reference.  
9 // Car reference is not allowed  
10 // to call turnSelfDrivingOn() method  
11 car.turnSelfDrivingOn();
```


- Each class is either directly or indirectly a subclass of `Object`.
- It is always possible to upcast any object to `Object` type (the price to pay is **losing access to all methods more specific than those defined in `Object`**).



```
1 AnyClass any = new AnyClass();  
  Object obj = (Object) any;  
3  
  Object obj = (Object) new Car();  
5 obj.turnOn() // compile-time error
```

- Downcasting refers to the use of a reference which is more specific (closer to the type of the object itself) than current reference.
- MUST be explicit because it's a risky operation, no automatic conversion provided by the compiler (it's up to you!).



```
1 Car car = new SelfDrivingCar();  
  
3 // Downcast  
  SelfDrivingCar sdCar = (SelfDrivingCar) car;
```



```
1 public class Car {
2     protected String licensePlate;
3     protected boolean isOn;
4
5     public void turnOn() {
6         ...
7     }
8
9     public void turnOff() {
10        ...
11    }
12 }
13
14 public class SelfDrivingCar extends Car {
15     private boolean isSelfDriving;
16
17     public void turnSelfDrivingOn() {
18         ...
19     }
20
21     public void turnSelfDrivingOff() {
22         ...
23     }
24 }
```



```
1 Car car = new SelfDrivingCar();
2
3 // Compile time error!
4 car.turnSelfDrivingOn();
5
6 // Downcast
7 SelfDrivingCar sdCar = (SelfDrivingCar) car;
8
9 // Accidentally OK!
10 // The object referenced by car was actually
11 // of class SelfDrivingCar
12 sdCar.turnSelfDrivingOn();
```



```
public class Car {
2   protected String licensePlate;
   protected boolean isOn;
4
   public void turnOn() {
6       ...
   }
8
   public void turnOff() {
10      ...
   }
12 }

14 public class SelfDrivingCar extends Car {
   private boolean isSelfDriving;
16
   public void turnSelfDrivingOn() {
18       ...
   }
20
   public void turnSelfDrivingOff() {
22       ...
   }
24 }
```



```
1 Car car = new Car();
3 // Compile time error!
  car.turnSelfDrivingOn()
5
  // Downcast
7 SelfDrivingCar sdCar = (SelfDrivingCar) car;
9 // Run-time error!
  sdCar.turnSelfDrivingOn();
```

- Compilers aid developers in writing working code. Runtime errors cannot be identified by compilers! Developers must be careful!
- Use the instanceof operator



```
1 Car car = new SelfDrivingCar();  
  if (car instanceof SelfDrivingCar) {  
3     SelfDrivingCar sdCar = (SelfDrivingCar) car;  
     sdCar.turnSelfDrivingOn();  
5 }  
  
7 // Since Java 14  
  if (car instanceof SelfDrivingCar sdCar){  
9     sdCar.turnSelfDrivingOn();  
  }
```

1 Inheritance

2 Polymorphism

- Types of Polymorphism
- Polymorphism Explained

3 References

- The derivation of the word Polymorphism is from two different Greek Words - Poly and morphs. "Poly" means numerous, and "Morphs" means forms. So, polymorphism means innumerable forms. Polymorphism, therefore, is one of the most significant features of Object-Oriented Programming.
- Polymorphism is the ability of an object to take on different forms. In Java, polymorphism refers to the ability of a class to provide different implementations of a method, depending on the type of object that is passed to the method.
- Polymorphism in Java is the task that performs a single action in different ways.
- Inheritance is a powerful feature in Java. Java Inheritance lets one class acquire the properties and attributes of another class. Polymorphism in Java allows us to use these inherited properties to perform different tasks. Thus, allowing us to achieve the same action in many different ways.



```
public class Shape {  
2   public void draw() {  
    System.out.println("Drawing a shape");  
4   }  
}  
6  
public class Circle extends Shape {  
8   @Override  
    public void draw() {  
10    System.out.println("Drawing a circle");  
    }  
12 }  
  
14 public class Square extends Shape {  
    @Override  
16    public void draw() {  
        System.out.println("Drawing a square");  
18    }  
}
```




```
1 public class TestPolymorphism {  
    public static void main(String[] args) {  
3         Shape s1 = new Circle();  
        Shape s2 = new Square();  
5  
        s1.draw(); // Output: "Drawing a circle"  
7        s2.draw(); // Output: "Drawing a square"  
    }  
9 }
```

Command window



```
1 Drawing a circle  
Drawing a square
```

1 Inheritance

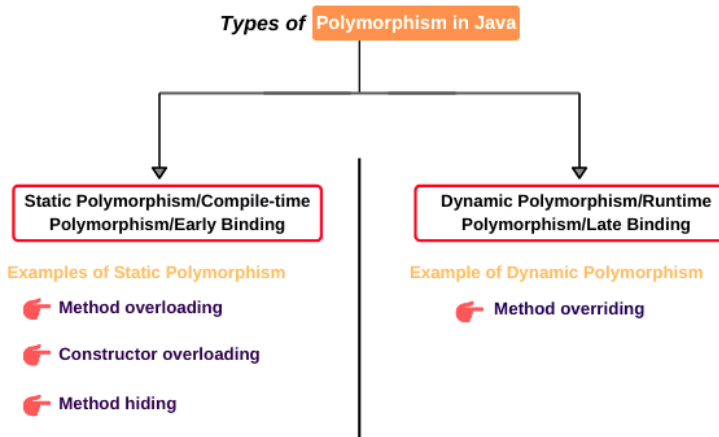
2 Polymorphism

- Types of Polymorphism
- Polymorphism Explained

3 References

- You can perform Polymorphism in Java via two different methods:

- ▶ Method Overloading
- ▶ Method Overriding



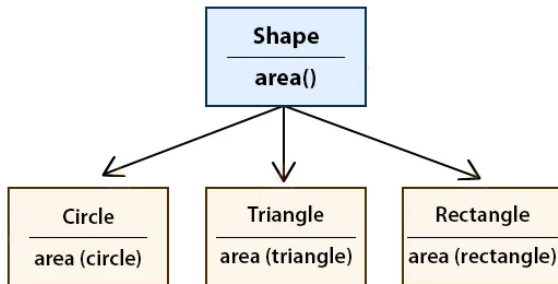
- Compile-time polymorphism is also called method overloading. It allows users to define multiple methods with the same name but different parameters. It is beneficial because it helps write cleaner and more efficient code.
- It is a form of program that enables multiple classes to implement similar functions but with different behaviors. It has the following attributes:
 - ▶ **Static Binding** - It relies heavily on static binding, meaning all the information necessary for making a call to a function must be known before compiling the code so that it can be solved during compilation.
 - ▶ **Function Overloading** - It is useful in function overloading which lets programmers create multiple versions of the same function with varying parameters while still keeping code concise—this makes it easier for developers when writing programs since they don't have duplicate codes.
 - ▶ **Generic Programming** - It provides another way to leverage compile-time polymorphism by allowing generic block codes that work in many contexts without needing modification each time. This further simplifies coding tasks by giving more flexibility when manipulating multiple values together at once instead of having separate instructions for every combination possible.

- **Method Overloading** is the process that can create multiple methods of the same name in the same class, and all the methods work in different ways. **Method Overloading** occurs when there is more than one method of the same name in the class.



```
public class Calculator {  
2   public static int add(int x, int y) {  
       return x + y;  
4   }  
  
6   public static float add(float x, float y) {  
       return x + y;  
8   }  
  
10  public static double add(double x, double y) {  
       return x + y;  
12  }  
}
```

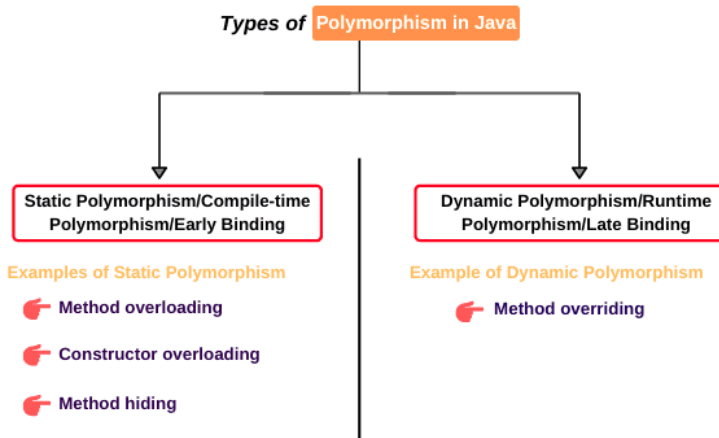
- **Method Overriding** is done when a child or a subclass has a method with the same name, parameters, and return type as the parent or the superclass; then that function overrides the function in the superclass.



- In simpler terms, if the subclass provides its definition to a method already present in the superclass; then that function in the base class is said to be overridden.
- Overriding is done by using a reference variable of the superclass. The method to be called is determined based on the object which is being referred to by the reference variable. This is also known as upcasting.

■ Also, Polymorphism in Java can be classified into two types, i.e:

- ▶ Static/Compile-Time Polymorphism
- ▶ Dynamic/Runtime Polymorphism



- **Compile-Time Polymorphism** in Java is also known as **Static Polymorphism**. Furthermore, the call to the method is resolved at compile-time. Compile-time polymorphism is achieved through **Method Overloading**. This type of polymorphism can also be achieved through **Operator Overloading**. However, Java does not support **Operator Overloading**.
- **Method Overloading** is when a class has multiple methods with the same name, but the number, types, and order of parameters are different. Java allows the user freedom to use the same name for various functions as long as it can distinguish between them by the type and number of parameters.

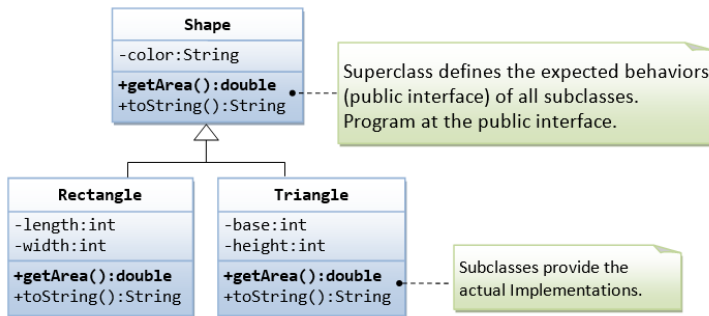
- **Runtime Polymorphism** in Java is also popularly known as **Dynamic Binding** or **Dynamic Method Dispatch**. In this process, the call to an overridden method is resolved dynamically at runtime rather than at compile-time. You can achieve runtime polymorphism via **Method Overriding**.
- Runtime polymorphism is a type of program that allows objects to respond differently according to their class type while still executing common parent behavior.
 - ▶ **Late Binding** - This form relies heavily on techniques such as virtual functions and late binding since bindings cannot be determined until runtime due to a lack of information available beforehand.
 - ▶ **Object Instantiation** - With this program, any changes made only take effect after object instantiation. This allows subclasses more control over how they behave compared with compile-time solutions which require all definitions upfront before running code.
 - ▶ **Speed** - Characteristics-wise run-time polymorphism can be quite powerful yet at times it can slow down performance depending on what's being done due to its reliance on processes happening affected. However, if used sparingly it should not cause drastic drops in speed or affect other components negatively.

- When using collections of objects belonging to a hierarchy of classes, **the actual methods which are called are known only at runtime.**
- The same method call (methods with the same signature) might have different results (**polymorphism**) depending on the actual class of the object.



```
1 Car[] garage = new Car[4];  
  
3 garage[0] = new Car();  
  garage[1] = new SelfDrivingCar();  
5 garage[2] = new SelfDrivingCar();  
  garage[3] = new Car();  
  
7  
  for(Car car : garage) {  
9     car.turnOn();  
  }  
  
11  
  /*  
13  * Which method is actually called  
  * is not knowable at compile time!  
15  */
```

- Suppose that our program uses many kinds of shapes, such as triangle, rectangle and so on.
- We should design a superclass called **Shape**, which defines the public interfaces (or behaviors) of all the shapes.
- For example, we would like all the shapes to have a method called **.getArea()**, which returns the area of that particular shape.





```
1 public class Shape {  
    // Private member variable  
3     private String color;  
  
5     // Constructs a Shape instance with the given color  
    public Shape (String color) {  
6         this.color = color;  
7     }  
  
9     // Returns a self-descriptive string  
11    @Override  
    public String toString() {  
13        return "Shape[color=" + color + "];"  
14    }  
  
15    // All shapes must provide a method called getArea()  
17    public double getArea() {  
        System.err.println("Shape unknown! Cannot compute area!");  
19        return 0;  
20    }  
21 }
```



```
1 public class Rectangle extends Shape {
2     private int length;
3     private int width;
4
5     // Constructs a Rectangle instance with the given color, length and width
6     public Rectangle(String color, int length, int width) {
7         super(color);
8         this.length = length;
9         this.width = width;
10    }
11
12    // Returns a self-descriptive string
13    @Override
14    public String toString() {
15        return "Rectangle[length=" + length + ",width=" + width + ", " + super.toString() + "];";
16    }
17
18    // Override the inherited getArea() to provide the proper implementation for rectangle
19    @Override
20    public double getArea() {
21        return length*width;
22    }
23 }
```



```
1 public class Triangle extends Shape {
    private int base;
3    private int height;

5    // Constructs a Triangle instance with the given color , base and height
    public Triangle(String color , int base , int height) {
6        super(color);
7        this.base = base;
8        this.height = height;
9    }

11   // Returns a self-descriptive string
12   @Override
13   public String toString() {
14       return "Triangle[base=" + base + ",height=" + height + ", " + super.toString() + "];"
15   }

17   // Override the inherited getArea() to provide the proper implementation for triangle
18   @Override
19   public double getArea() {
20       return 0.5*base*height;
21   }
22 }
23 }
```



```
1  /**
   * A test driver for Shape and its subclasses
   */
3  public class TestShape {
4      public static void main(String[] args) {
5          Shape s1 = new Rectangle("red", 4, 5); // Upcast
6          System.out.println(s1); // Run Rectangle's toString()
7          // Rectangle[length=4,width=5,Shape[color=red]]
8          System.out.println("Area is " + s1.getArea()); // Run Rectangle's getArea()
9          // Area is 20.0
10
11         Shape s2 = new Triangle("blue", 4, 5); // Upcast
12         System.out.println(s2); // Run Triangle's toString()
13         // Triangle[base=4,height=5,Shape[color=blue]]
14         System.out.println("Area is " + s2.getArea()); // Run Triangle's getArea()
15         // Area is 10.0
16     }
17 }
```

1 Inheritance

2 Polymorphism

- Types of Polymorphism
- Polymorphism Explained

3 References

- A **virtual function** is a special type of member function that, when called, resolves to the most-derived version of the function for the actual type of the object being referenced or pointed to.
- To implement virtual functions, C++ implementations typically use a form of **late binding** known as the **virtual table**. The virtual table is a lookup table of functions used to resolve function calls in a **dynamic/late binding** manner. The virtual table sometimes goes by other names, such as **vtable**, **virtual function table**, **virtual method table**, or **dispatch table**.
- Every class that uses **virtual functions** (or is derived from a class that uses virtual functions) is given its own virtual table. This table is simply a static array that the compiler sets up at compile time. A virtual table contains one entry for each virtual function that can be called by objects of the class. Each entry in this table is simply a function pointer that points to the most-derived function accessible by that class.
- The compiler also adds a **hidden pointer** that is a member of the base class, which we will call ***__vptr**. ***__vptr** is set (automatically) when a class object is created so that it points to the virtual table for that class. Unlike the ***this** pointer, which is actually a function parameter used by the compiler to resolve self-references, ***__vptr** is a real pointer member. Consequently, it makes each class object allocated bigger by the size of one pointer. It also means that ***__vptr** is inherited by derived classes, which is important.



```
class GrandParent
2 {
    public:
4     virtual void function1();
    virtual void function2();
6 };

8 class Parent : public GrandParent
{
10     public:
    void function1() override;
12 };

14 class Child : public Parent
{
16     public:
    void function2() override;
18 };
```



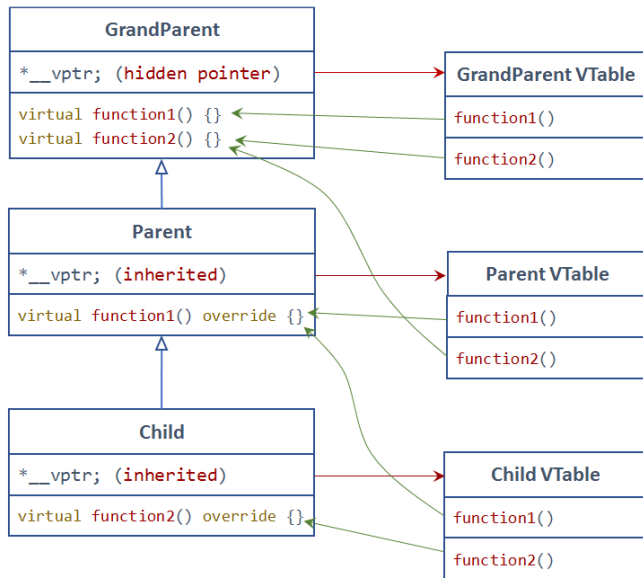
```
class GrandParent
2 {
    public:
4     VirtualTable *__vptr;

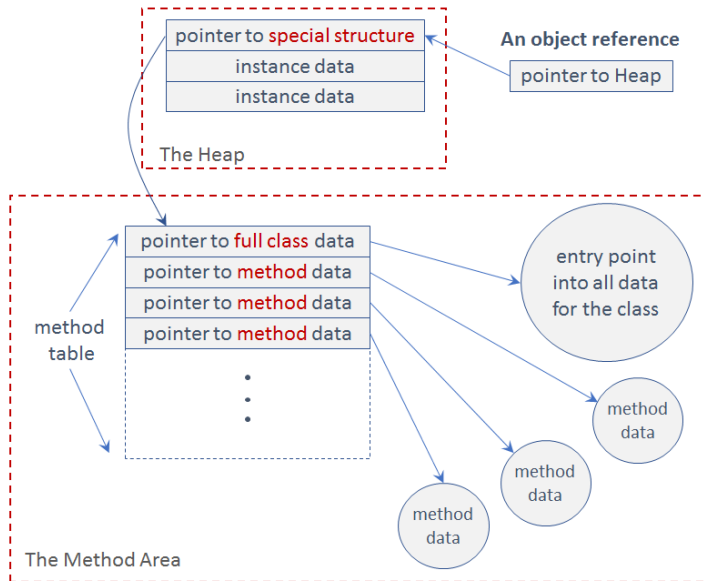
6     virtual void function1();
    virtual void function2();
8 };

10 class Parent : public GrandParent
    {
12     public:
        void function1() override;
14 };

16 class Child : public Parent
    {
18     public:
        void function2() override;
20 };
```










- When a class object is created, `*__vptr` is set to point to the virtual table for that class. So when an object of type `GrandParent` is created, `*__vptr` is set to point to the virtual table for `GrandParent`. When objects of type `Parent` or `Child` are constructed, `*__vptr` is set to point to the virtual table for `Parent` or `Child` respectively.
- Because there are only two virtual functions here, each virtual table will have two entries (one for `function1()` and one for `function2()`). When these virtual tables are filled out, each entry is filled out with the most-derived function an object of that class type can call.
- The **virtual table** for `GrandParent` objects is simple. An object of type `GrandParent` can only access the members of `GrandParent`. `GrandParent` has no access to `Parent` or `Child` functions. Consequently, the entry for `function1()` points to `GrandParent::function1()` and the entry for `function2()` points to `GrandParent::function2()`.
- An object of type `Parent` can access members of both `Parent` and `GrandParent`. However, `Parent` has overridden `function1()`, making `Parent::function1()` more derived than `GrandParent::function1()`. Consequently, the entry for `function1()` points to `Parent::function1()`. `Parent` hasn't overridden `function2()`, so the entry for `function2()` will point to `GrandParent::function2()`.
- The virtual table for `Child` is similar to `Parent`, except the entry for `function1()` points to `Parent::function1()`, and the entry for `function2()` points to `Child::function2()`.





- **Code Reusability:** Polymorphism provides the reuse of code, as methods with the same name can be used in different classes.
- **Flexibility and Dynamism:** Polymorphism allows for more flexible and dynamic code, where the behaviour of an object can change at runtime depending on the context in which it is being used.
- **Reduced Complexity:** It can reduce the complexity of code by allowing the use of the same method name for related functionality, making the code easier to read and maintain.
- **Simplified Coding:** Polymorphism simplifies coding by reducing the number of methods and constructors that need to be written.
- **Better Organization:** Polymorphism allows for better organization of code by grouping related functionality in one class.
- **Code Extensibility:** Polymorphism enables code extensibility, as new subclasses can be created to extend the functionality of the superclass without modifying the existing code.
- **Increased Efficiency:** Compile-time polymorphism can lead to more efficient coding. The compiler can choose the appropriate method to call at compile time, based on the number, types, and order of arguments passed to it.

- 1 Inheritance
- 2 Polymorphism
- 3 References**

-  ALLEN B. DOWNEY, CHRIS MAYFIELD, ***THINK JAVA***, (2016).
-  GRAHAM MITCHELL, ***LEARN JAVA THE HARD WAY***, 2ND EDITION, (2016).
-  CAY S. HORSTMANN, ***BIG JAVA - EARLY OBJECTS***, 7E-WILEY, (2019).
-  JAMES GOSLING, BILL JOY, GUY STEELE, GILAD BRACHA, ALEX BUCKLEY, ***THE JAVA LANGUAGE SPECIFICATION - JAVA SE 8 EDITION***, (2015).
-  MARTIN FOWLER, ***UML DISTILLED - A BRIEF GUIDE TO THE STANDARD OBJECT MODELING LANGUAGE***, (2004).
-  RICHARD WARBURTON, ***OBJECT-ORIENTED VS. FUNCTIONAL PROGRAMMING - BRIDGING THE DIVIDE BETWEEN OPPOSING PARADIGMS***, (2016).
-  NAFTALIN, MAURICE WADLER, PHILIP ***JAVA GENERICS AND COLLECTIONS***, O'REILLY MEDIA, (2009).
-  ERIC FREEMAN, ELISABETH ROBSON ***HEAD FIRST DESIGN PATTERNS - BUILDING EXTENSIBLE AND MAINTAINABLE OBJECT***, ORIENTED SOFTWARE-O'REILLY MEDIA, (2020).
-  ALEXANDER SHVETS ***DIVE INTO DESIGN PATTERNS***, (2019).

THANK YOU!