# Inheritance

By Rahul Barve

# Objectives

- Understanding Containment
- Understanding Inheritance
- Working with `super`
- Method Overriding
- Abstract Classes
- Introducing `final`

# Containment

By Rahul Barve

# Containment

- When an object is composed of another object or several objects then that object model in known as Containment.
- It represents HAS-A relationship.

# Containment

- A Customer has Address, where `Customer` is an object that holds another object: `Address`.
- E.g.

```
class Address {
    String street, bldg, city;
    int pincode;
    …
}
```

# Containment

```
class Customer {
        String customerId, name, email;
        Address permanentAddress;
        ….
}
```

# Containment

- In containment, the main object has a dependency on the contained object.
- The dependency is either Strong or Weak.

# Containment

- Hence containment is of two types:
  - Composition (Strong Dependency)
    - `Car` has `Engine`
  - Aggregation (Weak Dependency)
    - `Car` has `MusicSystem`

# Inheritance

By Rahul Barve

# Inheritance

- Inheritance is used to reuse the same structure to adopt different situations.
- It represents IS-A relationship.

# Inheritance

- Every CricketPlayer is a Player, where `CricketPlayer` is a class which inherits `Player` class.
- E.g.

```
class Player {
    ….
}
class CricketPlayer extends Player {
    …
}
```

# Inheritance

- Java supports 3 types of inheritances:
  - Simple
  - Multilevel
  - Hierarchical
- Inheritance in Java is always `public`.

# Inheritance

- Constructors in inheritance are invoked from bottom to top and executed from top to bottom.
- In case of parameterized constructors, this is done with the help of `super` keyword.

# Inheritance

- `super` always refers to an immediate super class.
- Allows to pass the control to the parent class constructor from the child class constructor.

# Inheritance

- It must be the first statement in the child class constructor.
- If not used, parent class uses its no-argument constructor.

# Method Overriding

By Rahul Barve

# Method Overriding

- When a subclass defines a method with same name, same signature with identical return type (in case of object references only) as that of the method in the super class, then that method is known as overridden method.

# Method Overriding

- Overriding is directly related to sub-classing.
- Overriding is required to modify the behavior of a super class's method to suit the new requirement.

# Method Overriding

- Consider the hierarchy:
  - `Employee` ← `Manager` ← `SalesManager` and method `getSalary()` has been overridden.

# Method Overriding

- `Employee a = new Employee();`

  `a.getSalary()` will call Employee's getSalary();
- `Manager b = new Manager();`

  `b.getSalary()` will call Manager's getSalary();
- `SalesManager c = new SalesManager();`

  `c.getSalary()` will call SalesManager's getSalary();

# Method Overriding

- Dynamic data type governs method selection.
- Every reference has 2 types – static and dynamic.

# Method Overriding

- E.g. `Employee emp = new Manager();`
- In the above statement, the static type of `emp` is `Employee` whereas its dynamic type is `Manager`.
- Hence, `emp.getSalary()` will invoke Manager's `getSalary()`.

# Overloading Vs. Overriding

- Static Polymorphism.
- Method Overloading happens in the same class.
- Signatures of overloaded methods must be different.
- Return types of the overloaded methods may be same or different.

- Dynamic Polymorphism.
- Method Overriding happens between inherited classes.
- Signatures of overridden methods must be same.
- Return types of the overridden methods must be same for primitives and may be identical for objects.

# `super()` Revisited

By Rahul Barve

# `super()` Revisited

- Can be used to invoke super class method from subclass overridden method.

```
class Derived extends Base{
    public void myMethod(){
        super.myMethod();
        //Invokes myMethod from Base
    }
}
```