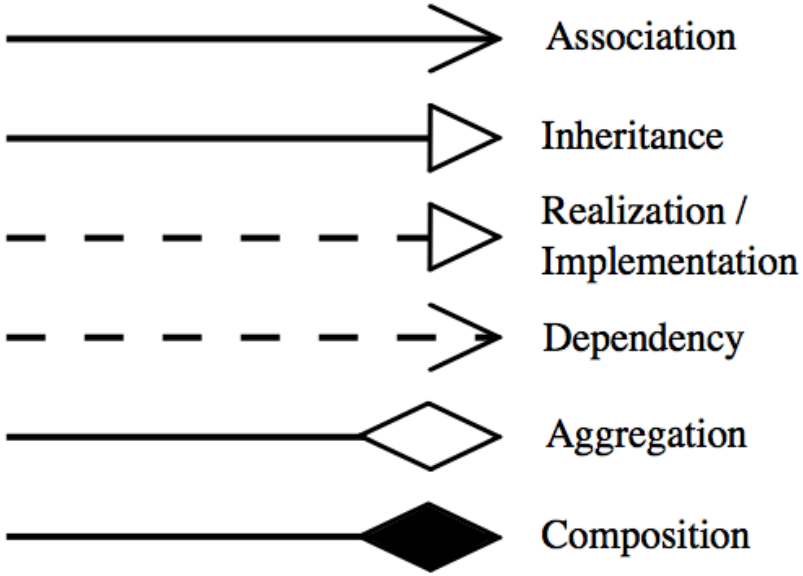
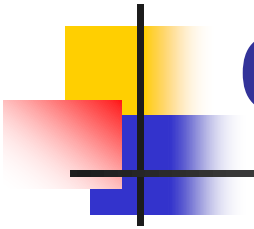
04. OO Relationships

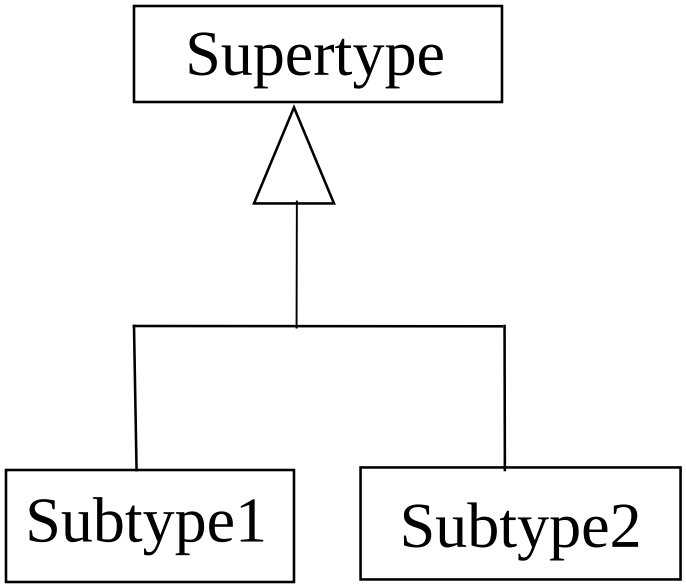




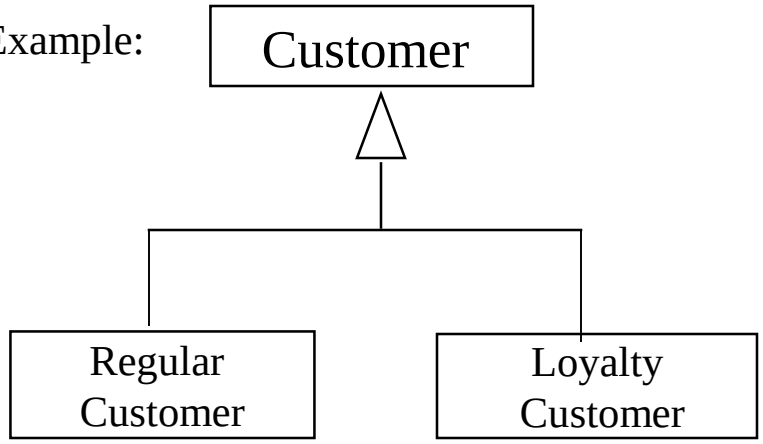
OO Relationships

- There are three kinds of Relationships
  - Generalizations (parent-child relationship)
  - Associations (student enrolls in course)
  - Dependencies
- Associations can be further classified as
  - Aggregation
  - Composition

OO Relationships: **Generalization**



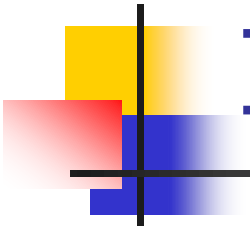
Example:



-Inheritance is a required feature of object orientation

-Generalization expresses a parent/child relationship among related classes.

-Used for abstracting details in several layers

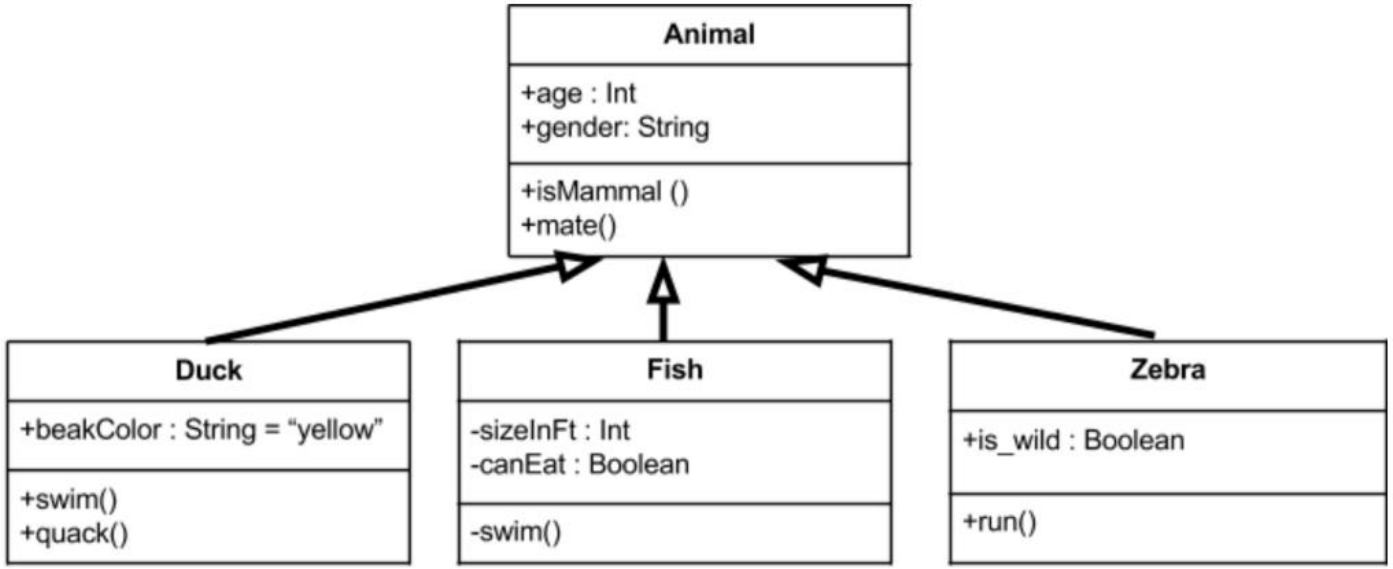


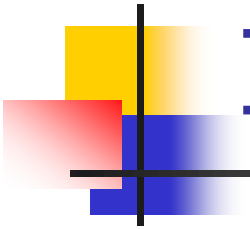
Inheritance

- Allows the creation of hierarchical classification.
- Using inheritance, we can create a general class that defines traits common to a set of related items.
- A class that is inherited is called a *'superclass'*.
- The class that does the inheriting is called a *'subclass'*.

Inheritance

- Indicates that child (subclass) is considered to be a specialized form of the parent (super class).
- For example consider the following:

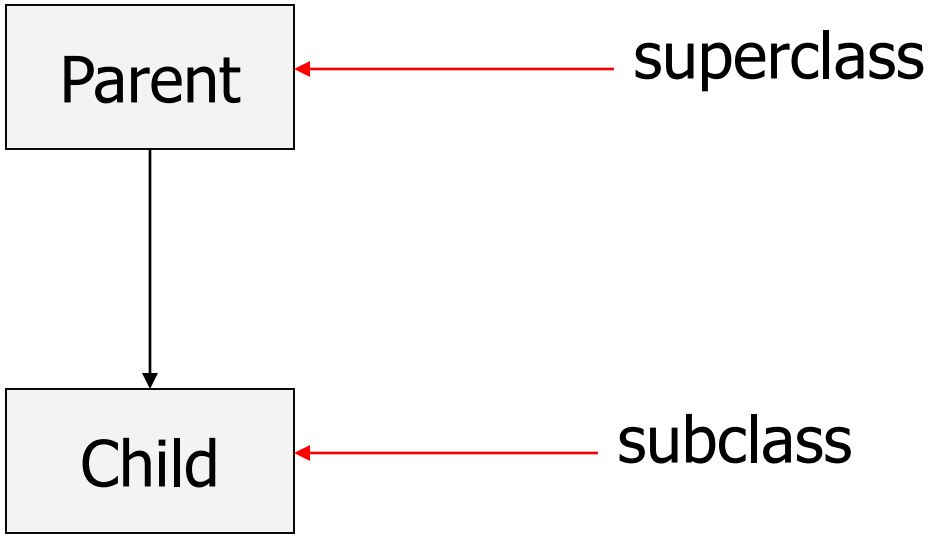




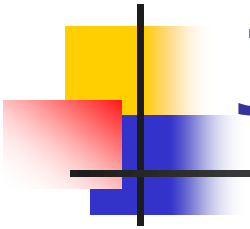
Inheritance

- Derive new classes from old classes
- Improve code re-use
- Easier to manage and understand complexity

Inheritance



- Java supports multilevel inheritance but not multiple inheritance.

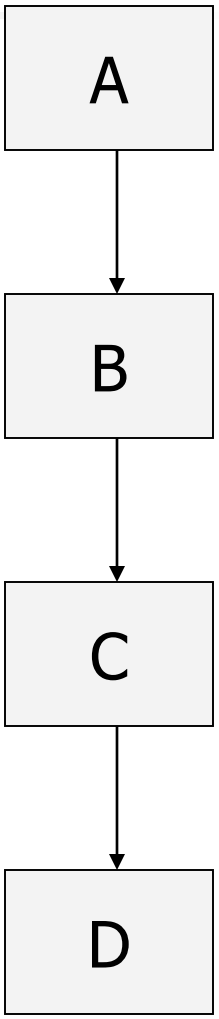


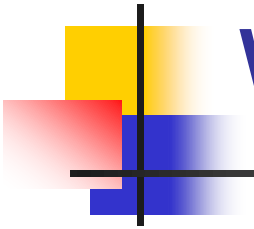
Java Implementation

```
public class Parent {
   . . .
}

public class Child extends Parent  {
   . . .
}
```

Multilevel Inheritance

```
class A {. . . }

class B extends A { . . . }
class C extends B { . . . }
class D extends C { . . . }
```





What is inherited?

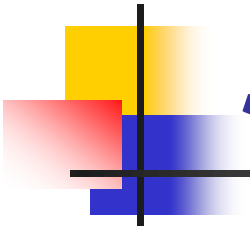
- All *public* data members and methods (except constructors) in the superclass are *inherited* by the subclass. It is as if their definitions are copied into the subclass's class definition.
- No members of the subclass are visible to the superclass.

Inheritance Example

```
public class Person  {

    public String name;
    public int age;
    public Date birthDay;

    public Person(String name, int age, Date birthDay)  {
          this.name = name;
          this.age = age;
          this.birthDay = birthDay;
    }

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

Example - con't

```
public class Student extends Person  {
    public int studentID;
    public String dept;
    private float GPA;

    public Student(String name, int age, Date birthDay, int
                                studentID, String dept, float GPA)  {

        super(name, age, birthDay);
        …
        this.studentID = studentID;
        …
    }
}
```



super Keyword

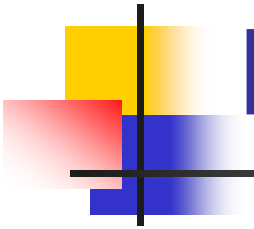
- *super* is used to refer to the members of the current object's superclass.
- Used for calling the superclass version of a method which the subclass has over-ridden.

```
public Roof getRoof()  {
      if ( convertibleRoofIsBroken )
            return super.getRoof();
      return new ConvertibleRoof();
}
```

super - con't

- Can be used in a constructor to access a super-class constructor.
- Must be first line in constructor if present
- Syntax: super(<constructor args>);

```
public class Student extends Person  {
    …
    public Student(String name, int age, Date birthday, int studentID)  {
            super(name, age, birthday);
            this.studentID = studentID;
    }
}
```



Method Overriding

- Redefine methods inherited from superclass to add or change functionality.
- That is, When a method in the subclass has the same name and type signature as a method in its superclass.
- Method overriding allows Java to support run-time polymorphism.

Method Overriding: Example

```
class A  {
    int i, j;

    A(int a, int b)  {
        i = a;
        j = b;
    }

    void display()  {
        System.out.println("i and
                    j: " +i+ " " +j);
    }
}
```

```
class B extends A  {
    int k;

    B(int a, int b, int c)  {
        super(a, b);
        k = c;
    }

    void display()  {
        // super.display();
        System.out.println("k: "+k);
    }
}
```

```
class Override  {
    public static void main(String
                                args[])  {
        B subOb = new B(1, 2, 3);
        subOb.display();
    }
}
```

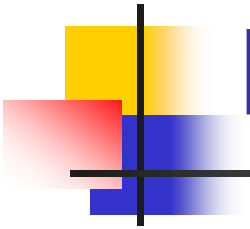
Run-time polymorphism - Example

```
class Figure  {
    double dim1;
    double dim2;

    Figure(double a, double b)  {
        dim1 = a;
        dim2 = b;
    }

    double area()  {
        System.out.println("Area
                            undefined");
        return 0;
    }
}
```

```
class Rectangle extends Figure  {

    Rectangle(double a, double b)  {
        super(a, b);
    }

    double area()  {
        System.out.println("Inside
                            rectangle");
        return dim1*dim2;
    }
}
```

```
class Triangle extends Figure  {

    Triangle(double a, double b)  {
            super(a, b);
    }

    double area()  {
            System.out.println("Inside triangle");
            return dim1*dim2/2;
    }
}
```

```
class AreaFinder  {

    public static void main(String args[])  {

            Figure f = new Figure(10, 10);
            Rectangle r = new Rectangle(9, 5);
            Triangle t = new Triangle(10, 8);

            Figure figRef;

            figRef = r;
            System.out.println("Area is " + figRef.area());

            figRef = t;
            System.out.println("Area is " + figRef.area());

            figRef = f;
            System.out.println("Area is " + figRef.area());
    }
}
```



Dynamic Method Dispatch

- Dynamic method dispatch is the mechanism by which a call to an overridden function is resolved at run time, rather than compile time. Dynamic method dispatch is important because this is how java implements run-time polymorphism.