Inheritance

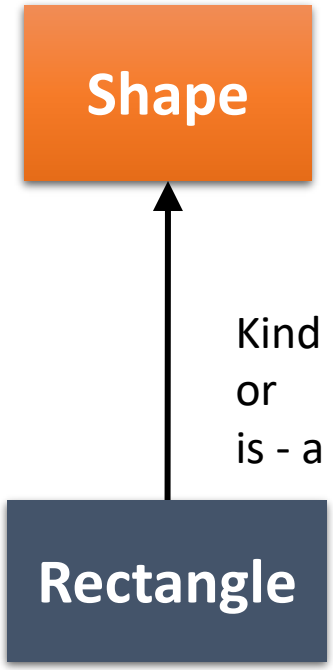
Object Oriented Programming

Inheritance

- **Inheritance:**
  - Inheritance is **to extend the functionality of an existing class**
  - Inheritance represents **is – a** relationship between the Child and its Parent.
  - In Java language all classes have a top parent class that is called Object

- **Using inheritance comes with a set of benefits:**
  - Reduces the amount of code needed to be written by developers.
  - Makes the code easy to maintain and update.
  - Helps in building more reasonable class hierarchy and simulating to the real world.

Inheritance



In Java

```
public class Shape {
    private int dim;

    public int getDim() {
        return dim;
    }

    public void setDim(int dim) {
        this.dim = dim;
    }
}
```

```
public class Rectangle extends Shape{
    private int width;

    public int getWidth() {
        return width;
    }
    public void setWidth(int w) {
        width = w;
    }
}
```

Inheritance

Inheritance

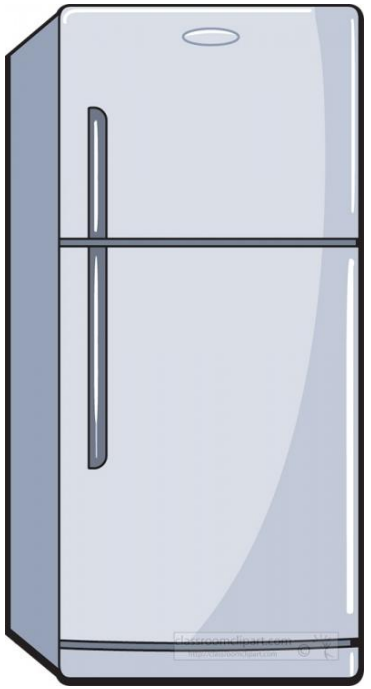
Based on the inheritance relationship the child can use both parent's methods and the methods defined within itself.

```
class Main{
    public static void main(String[] args) {
        Rectangle rectangle = new Rectangle();
        rectangle.setDim(10);
        rectangle.setWidth(20);
        System.out.println("Dim value is"+ rectangle.getDim());
    }
}
```

Used a method from parent class Shape

Inheritance

Imagine designing a system that is having a House, Plane and Fridge each of them contains a door. Can we make a parent class that has a door and let them inherit such class?



Inheritance

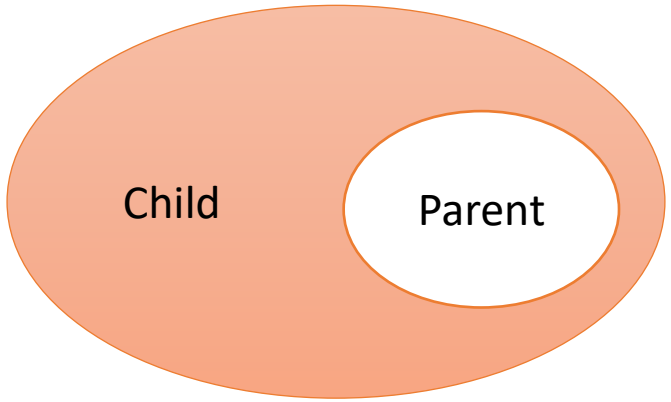
Inheritance decision is based on the availability of two conditions:

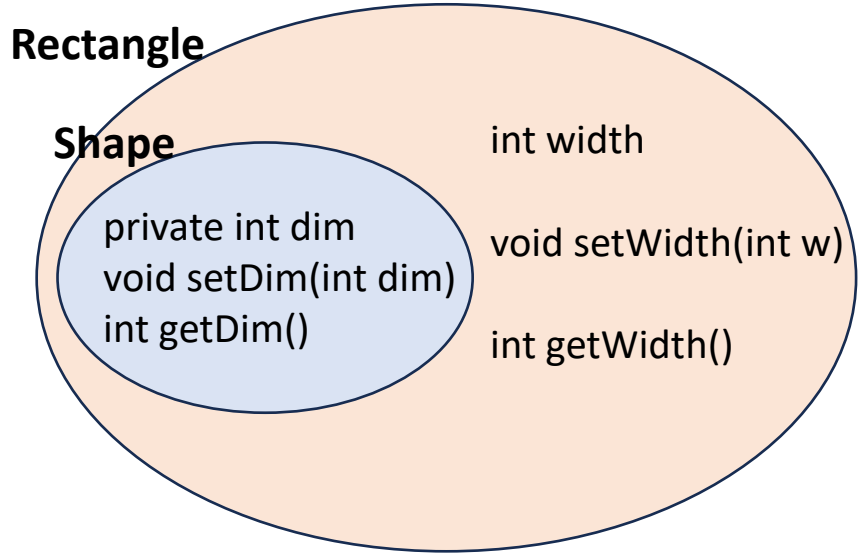
1. **is – a** relationship between the child and its parent
2. Hierarchy among the siblings.
   (Siblings have a lot of commonalities and a little bit different from each other)

Then, for the previous example we can't have House, Plane and Fridge as siblings. And thus, inheritance here is not the right thing to be done.

Inheritance

- In the Shape example,
  we want to be able to access the members of the parent class easily within the Child class without violating the encapsulation rules.





Inheritance

- We can mark our attributes and methods as **protected** to indicate that they are accessible from within the child classes but not from the outside world.

```
public class Shape {
    protected int dim;

    public int getDim() {
        return dim;
    }

    public void setDim(int dim) {
        this.dim = dim;
    }
}
```


Now the **dim** attribute is accessible inside the Shape class or any sub class of it, but not accessible in any other classes outside that hierarchy

Constructor Redirection

When creating an object from a child,
the parent object first get created then the child object starts to be created.

By default, a child will create an instance of the parent
using the default constructor by default.

If there's no default constructor then, we **must** determine a specific
parameterized constructor ourselves using the **super** keyword
and this **must** happen in the first line of code inside the Child's constructor.

Using Default Constructor

```java
public class Shape {
    private int dim;

    public Shape(){
        System.out.println("0 - Param constructor in Shape");
    }

    public int getDim() {...3 lines }

    public void setDim(int dim) {...3 lines }

    public int calcualteArea(){return 0;}
}
```

```java
public class Rectangle extends Shape{
    private int width;

    public Rectangle(){
        System.out.println("0 - Param Rectangle Constructor");
    }

    public int getWidth() {...3 lines }

    public void setWidth(int w) {...3 lines }

    public int calcualteArea() {...3 lines }
}
```

```java
class Main{
    public static void main(String[] args) {
        Rectangle rectangle = new Rectangle();
        rectangle.setDim(10);
        rectangle.setWidth(20);
        System.out.println("Dim value is "+ rectangle.getDim());
    }
}
```

```
run:
0 - Param constructor in Shape
0 - Param Rectangle Constructor
Dim value is 10
BUILD SUCCESSFUL (total time: 0 seconds)
```

Using Parameterized Constructor

```java
public class Shape {
    private int dim;

    public Shape(int d){
        dim = d;
        System.out.println("1 - Param constructor in Shape");
    }

    public int getDim() {...3 lines }

    public void setDim(int dim) {...3 lines }

    public int calcualteArea(){return 0;}
}
```

```java
public class Rectangle extends Shape{
    private int width;

    public Rectangle(){
        System.out.println("0 - Param Rectangle Constructor");
    }

    public int getWidth() {...3 lines }

    public void setWidth(int w) {...3 lines }

    public int calcualteArea() {...3 lines }
}
```

Error message that appears in the Rectangle class.

```
constructor Shape in class Shape cannot be applied to given types;
  required: int
  found: no arguments
  reason: actual and formal argument lists differ in length
```

Constructor Redirection

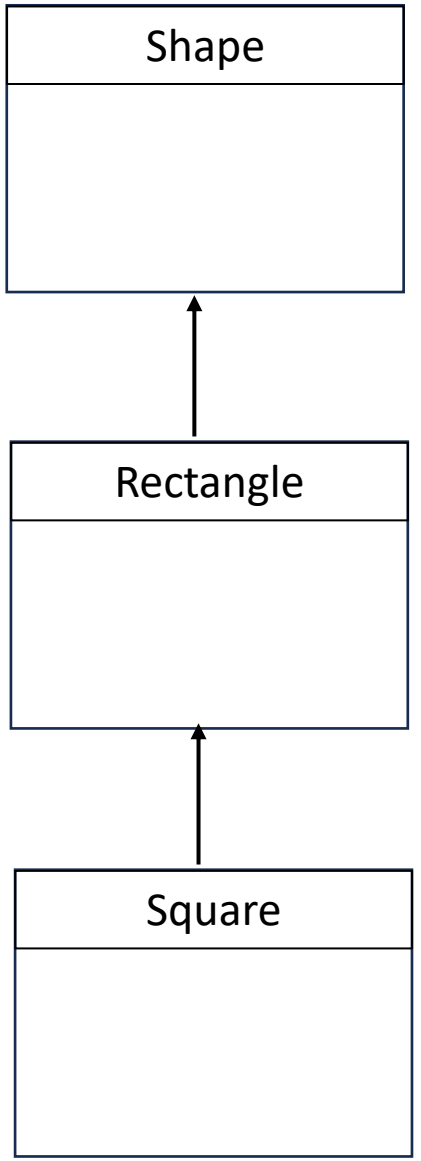
```java
public class Shape {
    private int dim;

    public Shape(int d){
        dim = d;
        System.out.println("1 - Param constructor in Shape");
    }

    public int getDim() {...3 lines }

    public void setDim(int dim) {...3 lines }

    public int calcualteArea(){return 0;}
}
```

```java
public class Rectangle extends Shape{
    private int width;

    public Rectangle(){
        super(10);
        System.out.println("0 - Param Rectangle Constructor");
    }

    public int getWidth() {...3 lines }

    public void setWidth(int w) {...3 lines }

    public int calcualteArea() {...3 lines }
}
```

Adding the **super** keyword solved the previous problem and helped us creating an object from the Shape (Parent) class as it specified the required constructor and passed the parameters.

Multi-level Inheritance

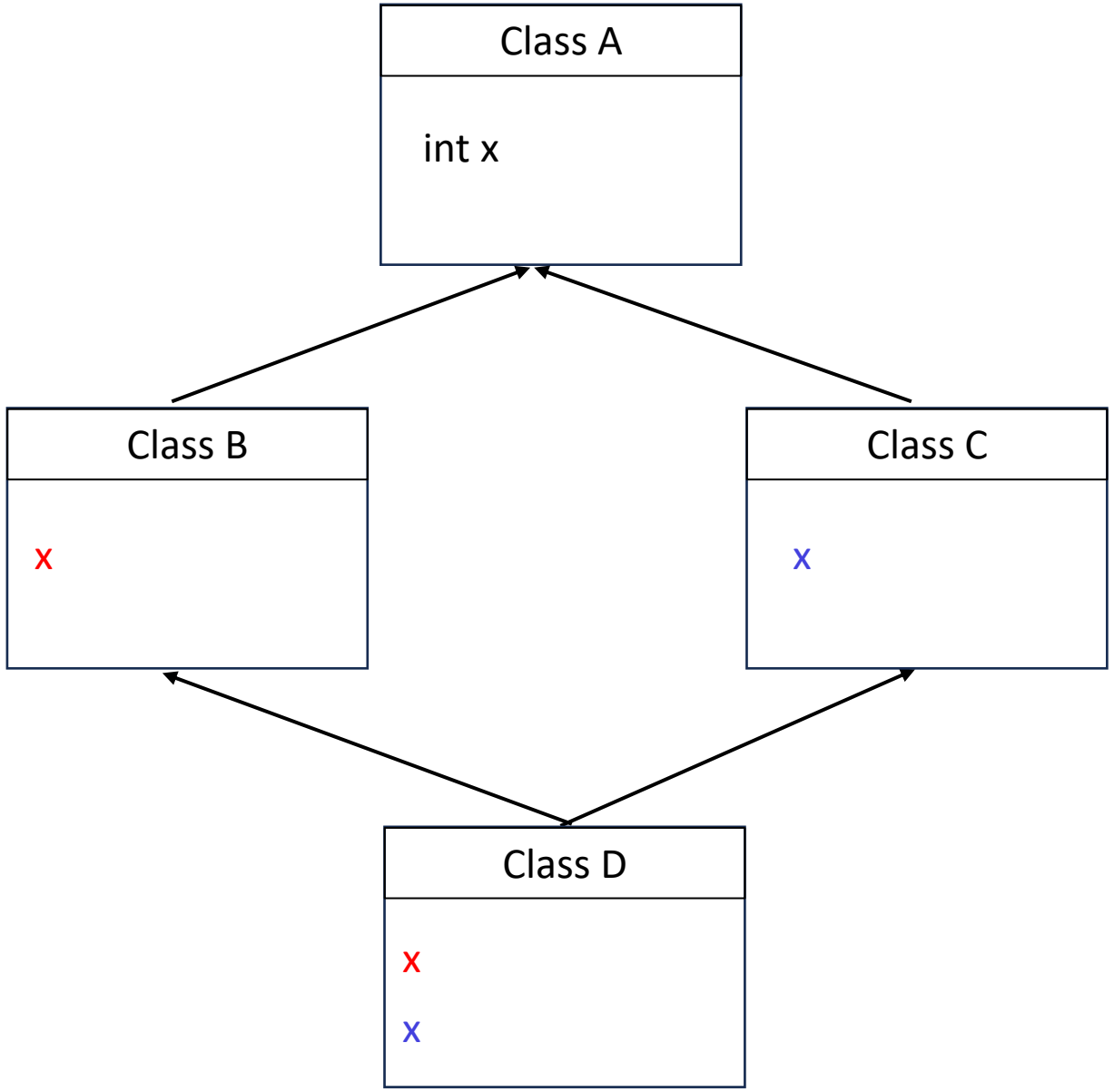
```
class Shape{
    public Shape(){
        System.out.println("0 Parameters Constructor - Shape ");
    }
}
class Rectangle extends Shape{
    public Rectangle(){
        System.out.println("0 Parameters Constructor - Rectangle ");
    }
}
class Square extends Rectangle{
    public Square(){
        System.out.println("0 Parameters Constructor - Square ");
    }
}
```



Multiple Inheritance

- This is what we call

  The Diamond Problem



Polymorphism

Object Oriented Programming

Polymorphism

- Polymorphism has Three Applications:
    1. Method Overloading
    2. Method Overriding
    3. Reference Variable Casting

Polymorphism

1- Method Overloading

- The first polymorphism concept that can be seen inside one class or more than one class with inheritance relationship is the overloading.
- Overloading means reusing a method name, but with different arguments.
- Overloaded methods:
  - Must have different argument lists (Type, Order, or Count)
  - May have different return types, if argument lists are also different
- Note: changing the method return type **is not** considered overloading and will result in compilation error

1- Method Overloading

Polymorphism

```
public void draw(String s) {

}
public void draw(int i) {

}
public void draw(double f) {

}
public void draw(int i, double f) {

}
public int draw(int i, double f){
    return 5;
}
```

Compilation error

Polymorphism

2- Method Overriding

- Second type of polymorphism is method overriding.
- Overriding is redeclaring a method inside a child class typically with the same signature as it was declared in its parent class.
- Overriding **can only happen inside different classes** inside the same class hierarchy structure.

```
public class Shape {
    private int dim;

    public int getDim() {...3 lines }

    public void setDim(int dim) {...3 lines }

    public int calculateArea(){
        return 0;
    }
}
```

```
public class Rectangle extends Shape{
    private int width;

    public int getWidth() {...3 lines }

    public void setWidth(int w) {...3 lines }

    public int calcualteArea() {
        return width* getDim();
    }
}
```

Polymorphism

2- Method Overriding

- Now if we try to call the **`calculateArea( )`** method inside the Rectangle class the newly overridden method inside the Rectangle class will respond.
- If we want to call the parent version, we can use the **super** reference.
- Overloading and overriding are used to redefine the method implementation with preserving the method name.
- This helps the class users to remember only a little about my class, and in the other hand the class will behave differently according to the implemented method.

Polymorphism

2- Method Overriding

- Inside the child class if you want to refer | call the parent's version of the method not the child's overridden version,
  then you must call the method by using the **super** reference.
- **`super`** refers to the current instance of the parent.

```
public class Shape {
    private int dim;
    public Shape(){...3 lines }
    public int getDim() {...3 lines }
    public void setDim(int dim) {...3 lines }
    public int calcualteArea(){
        System.out.println("Parent's method version");
        return 0;
    }
}
```

```
public class Rectangle extends Shape{
    private int width;
    public Rectangle(){...3 lines }
    public int getWidth() {...3 lines }
    public void setWidth(int w) {...3 lines }
    public int calcualteArea() {
        super.calcualteArea();
        System.out.println("Child's version");
        return width* getDim();
    }
}
```

Polymorphism

3- Reference Variable Casting

- Polymorphism allows that one reference type can point to different objects as long as they are in the correct class hierarchy.

```
class Main{

    public static void main(String[] args) {
        Shape rectangle = new Rectangle();
        rectangle.calculateArea();
    }

}
```

- And when calling the overridden method **calculateArea( )** it will call the correct object method of the Rectangle class.

Polymorphism

3- Reference Variable Casting

When making these lines of code there are two phases happen to make the code work.

```
Shape rectangle = new Rectangle();
rectangle.calculateArea();
```

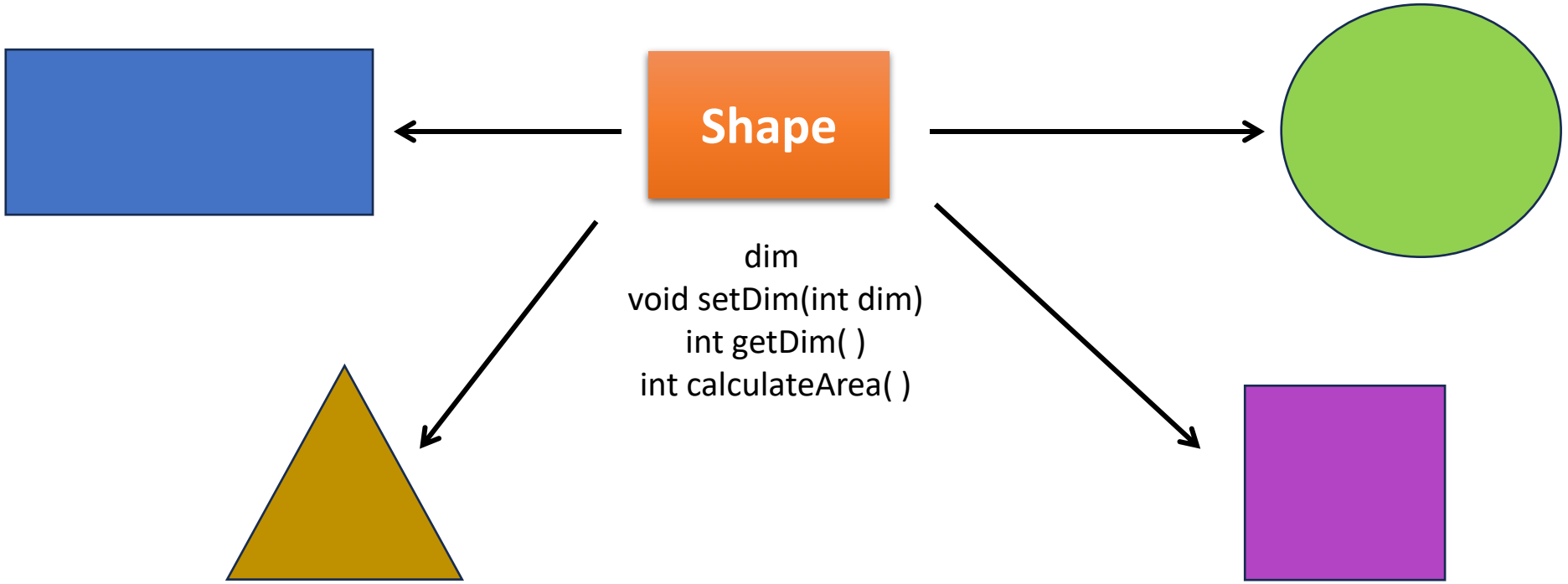
Phase1 (Compile Time Check):

1. The compiler will check whether Rectangle extends Shape?
2. Does Shape (Parent) class contain the called method calculateArea( )?

Phase2 (Run Time Check):

When the previous conditions are met, and the code compiles the method that'll work is the last version of calculateArea( ) in the real object's class (Rectangle)

Abstract Class



Abstract Class

- At this point, the class Shape is not well defined and implementing its methods at this point will be pointless.
- Also instantiating an object of class Shape is also pointless as the Shape class is only a concept class to indicate that any shape can have this properties, but it is only **abstract** definition.
- So, it's better to declare the class Shape as **abstract** class, rather than declaring a normal class.
- This provides a more professional and efficient class hierarchy building.

Abstract Class

- Abstract class is a class that may **contain zero or more abstract method(s)** and | or **zero or more concrete method(s)**.
- Abstract class is created by adding the abstract key word before the class name.

```
public abstract class Shape {
```

- At this point we cannot instantiate any objects from this class.

```
Shape s1 = new Shape();
```

Shape is abstract; cannot be instantiated

- Abstract method is a method without implementation body only the **method signature**.

```
public abstract int calcualteArea();
```

Abstract Class

- Abstract class can be inherited by other classes and its abstract methods overridden to provide an implementation to a case of this abstract class.
- Inheriting from abstract class without overriding its abstract methods makes your child class also abstract.
- Overriding the abstract method is a must or declare your child class as abstract too.
- Abstract class can have zero or more abstract methods, but abstract methods cannot exist in normal class.

Lab Assignment

- Create a Shape class which have a dim attribute and an abstract calculateArea( ), the shape will have three subclasses Rectangle, Triangle and Circle.
  Finally, create a Picture class which will have one method sumAreas( ) which accepts any three shapes as parameters and return the total sum of their areas.

- Create a Parent class that has two attributes: num1 & num2 and sum( )
  that calculates the total sum of num1 and num2.
  After that, create a Child class which inherits the Parent and has a num3 attribute.
  Your Child class should override the sum( ) to calculte the total sum of the three numbers.