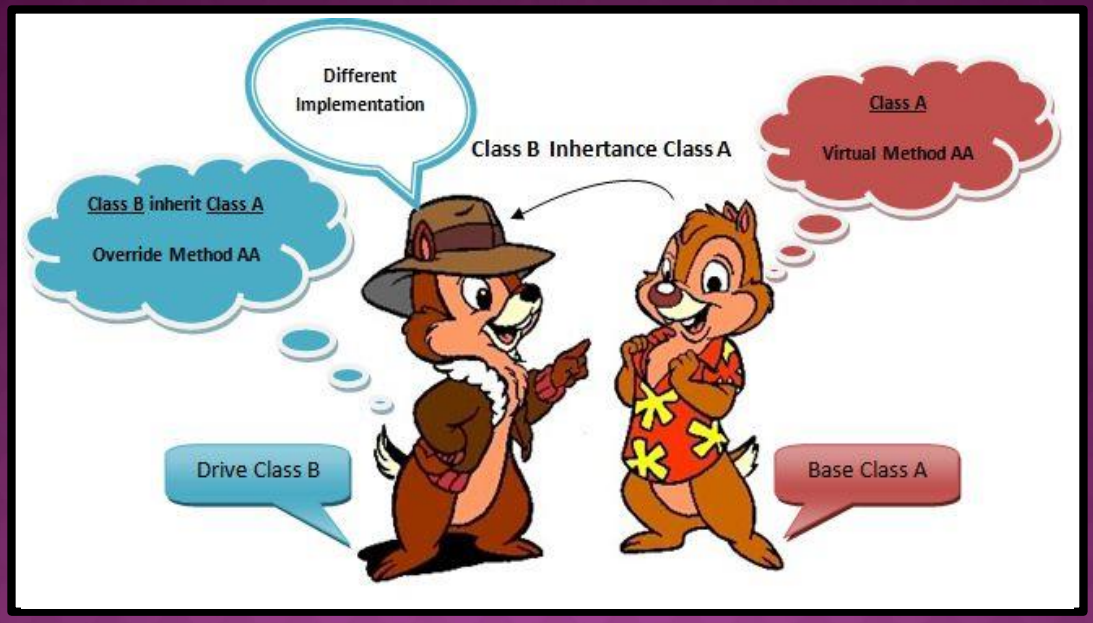


Object Oriented Thinking

Minal Maniar

OBJECT ORIENTED THINKING

- Write a program that prompts the user to enter a weight in pounds and height in inches and displays the BMI.

  1 pound = **0.45359237** kg

  1 inch = **0.0254** m

- The procedural paradigm focuses on designing methods and the object-oriented paradigm couples data and methods together into objects. Software design using the object-oriented paradigm focuses on objects and operations on objects.
- The object oriented approach combines the power of the procedural paradigm with an added dimension that integrates data with operations into objects.

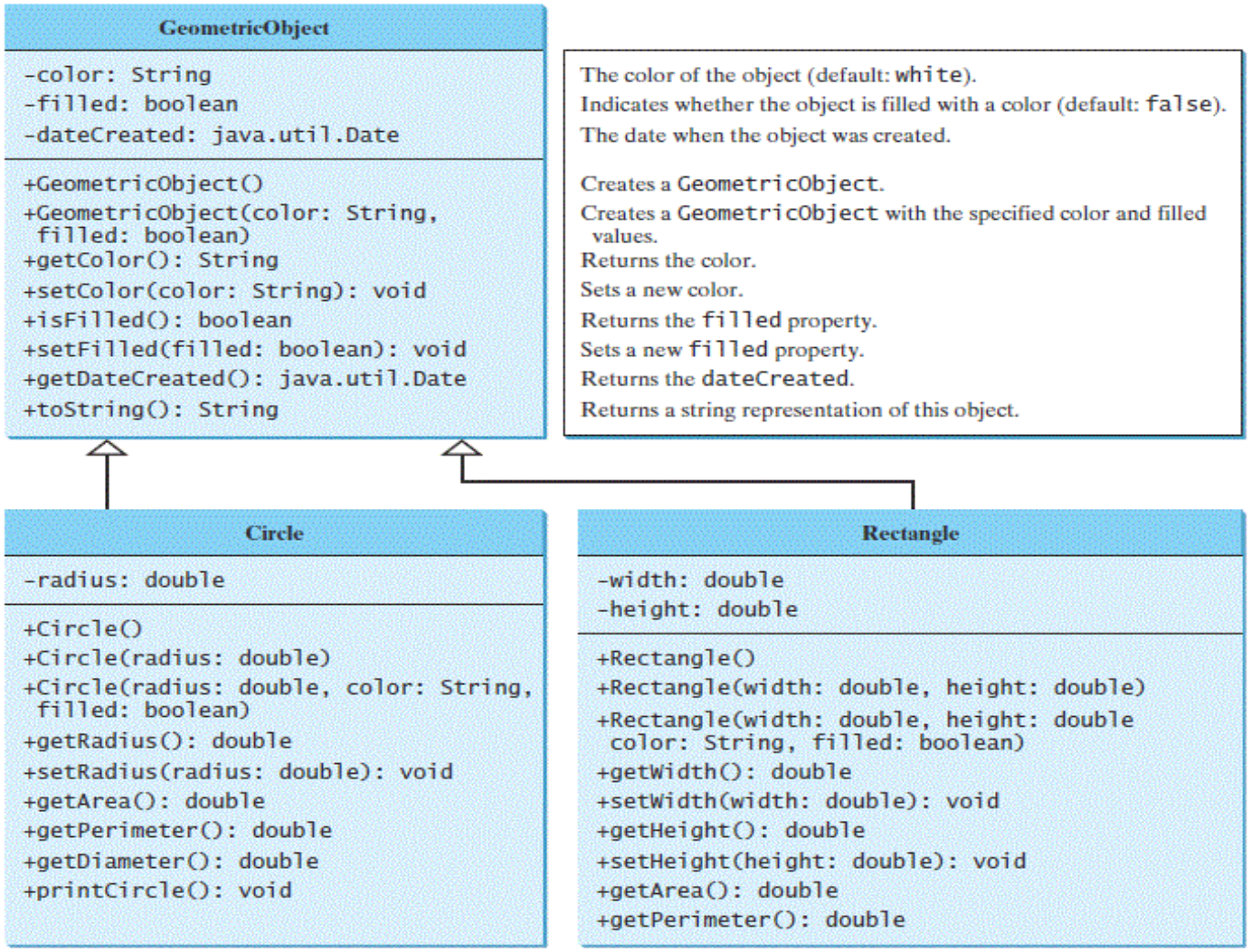
OBJECT ORIENTED

```java
public class ComputeAndInterpretBMI {

    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("Enter weight in pounds: ");
        double weight = input.nextDouble();

        System.out.print("Enter height in inches: ");
        double height = input.nextDouble();

        final double KILOGRAMS_PER_POUND = 0.45359237; // Constant
        final double METERS_PER_INCH = 0.0254; // Constant

        double weightInKilograms = weight * KILOGRAMS_PER_POUND;
        double heightInMeters = height * METERS_PER_INCH;
        double bmi = weightInKilograms
                / (heightInMeters * heightInMeters);

        System.out.println("BMI is " + bmi);
        if (bmi < 18.5) {
            System.out.println("Underweight");
        } else if (bmi < 25) {
            System.out.println("Normal");
        } else if (bmi < 30) {
            System.out.println("Overweight");
        } else {
            System.out.println("Obese");
        }
    }
}
```

INHERITANCE

- Object-oriented programming allows you to define new classes from existing classes. This is called inheritance.
- Inheritance enables you to define a general class (i.e., a superclass) and later extend it to more specialized classes (i.e., subclasses).

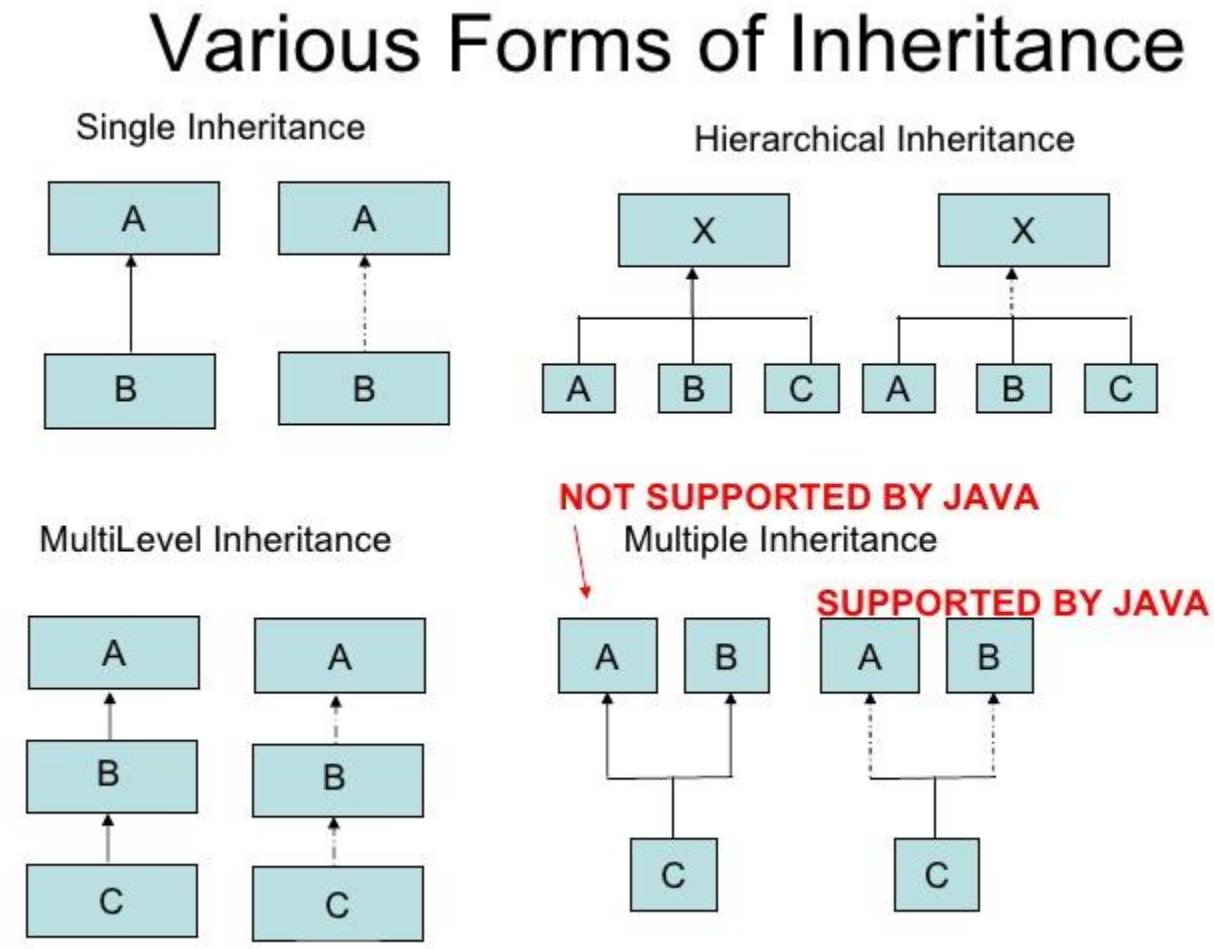
INHERITANCE



INHERITANCE

- Contrary to the conventional interpretation, a subclass is not a subset of its superclass.
- In fact, a subclass usually contains more information and methods than its superclass.
- Private data fields in a superclass are not accessible outside the class. Therefore, they cannot be used directly in a subclass. They can, however, be accessed/mutated through public accessors/mutators if defined in the superclass.

INHERITANCE



USE OF SUPER

- super can be used to refer immediate parent class instance variable.
- super can be used to invoke immediate parent class method.
- super() can be used to invoke immediate parent class constructor.

METHOD OVERRIDING

Method name is same in Super and sub class, signature is same

Return type

Method signature = Number of parameters + data type of parameters

```
class Animal
{
    public void eat()
    {
        System.out.println("Generic Animal eating");
    }
}

class Dog extends Animal
{
    public void eat()    //eat() method overriden by Dog class.
    {
        System.out.println("Dog eat meat");
    }
}
```

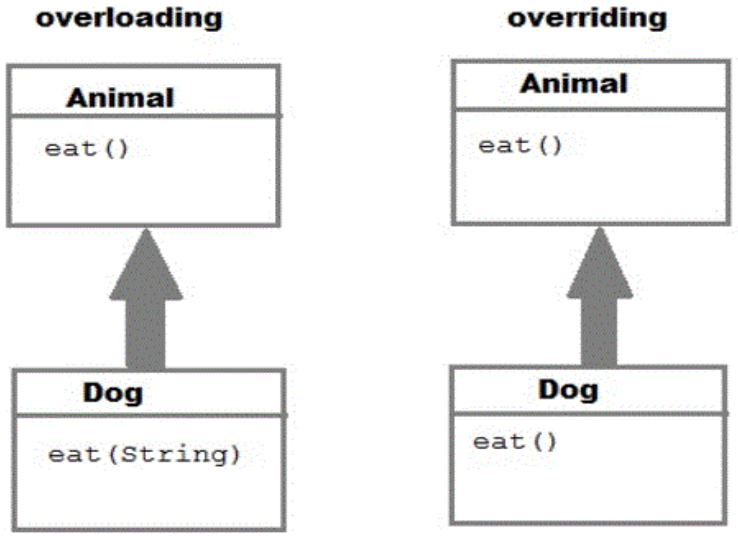
COVARIANT RETURN TYPE

Since Java 5, it is possible to override a method by changing its return type. If subclass override any method by changing the return type of super class method, then the return type of overridden method must be **subtype of return type** declared in original method inside the super class. This is the only way by which method can be overridden by changing its return type.

COVARIANT RETURN TYPE

```
class Animal {

    Animal myType() {
        return new Animal();
    }
}

class Dog extends Animal {

    Dog myType() //Legal override after Java5 onward
    {
        return new Dog();
    }
}

public class Override {

    public static void main(String[] args) {
        Dog d = new Dog();
        System.out.println(d.myType());
    }
}
```

OVERLOADING VS OVERRIDING

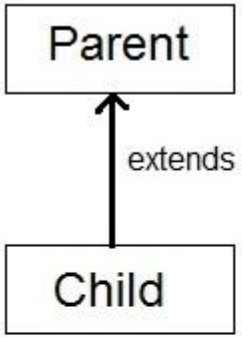


Method Overloading	Method Overriding
Parameter must be different and name must be same.	Both name and parameter must be same.
Compile time polymorphism.	Runtime polymorphism.
Increase readability of code.	Increase reusability of code.
Access specifier can be changed.	Access specifier most not be more restrictive than original method(can be less restrictive).

STATIC & DYNAMIC BINDING

- Static binding in Java occurs during compile time while dynamic binding occurs during runtime. Static binding uses type(Class) information for binding while dynamic binding uses instance of class(Object) to resolve calling of method at run-time.
- Overloaded methods are bonded using static binding while overridden methods are bonded using dynamic binding at runtime.

DYNAMIC METHOD DISPATCH OR VIRTUAL METHOD INVOCATION OR RUN TIME POLYMORPHISM




```
Parent p = new Parent( );

Child c = new Child( );

Parent p = new Child( );
```

Upcasting

```
Child c = new Parent( );
```

incompatible type

When Parent class reference variable refers to Child class object, it is known as Upcasting

EXAMPLE

```
class A {
    int x;
    public void printIt() {
        System.out.println("Method in class A...");
    }
}
class B extends A {
    public void printIt() {
        System.out.println("Method in class B....");
    }
}
public class Test {
    public static void main(String[] args) {
        A a = new B();
        a.printIt(); // prints "method in class B"
    }
}
```

METHOD HIDING

```
class C {
    public static void printStatic() {
        System.out.println("In C....");
    }
}

class D extends C {
    public static void printStatic() {
        System.out.println("In D....");
    }
}

class TestStatic {
    public static void main(String[] args) {
        C c = new D();
        c.printStatic(); // prints "In C...."
    }
}
```

Static methods cannot be overridden because, a static method is bounded with class where as instance method is bounded with object.

Refer:
http://docs.oracle.com/javase/tutorial/java/landl/override.html

CASTING OBJECTS

```
class A{}

class B extends A{}

    A a=new B();
    a.methodSpecificToB(); // illegal
    B b=(B)a;
    b.methodSpecificToB(); //legal
```

CASTING OBJECTS EXAMPLE

```
class A {
    int x;
    public void printIt() {
        System.out.println("\nMethod in class A...");
    }
}
class B extends A {
    public void printIt() {
        System.out.println("\nMethod in class B....");
    }
    public void printAgain(){
        System.out.println("\nPrint again....");
    }
}
public class Test {
    public static void main(String[] args) {
        A ob = new A();
        ob.printIt();

        A a = new B();
        a.printIt(); // prints "method in class B"
        //a.printAgain() is not accessible

        B b = (B)new A(); // cast object
        b.printAgain();
        b.printIt();  //
    }
}
```

CASTING OBJECTS EXAMPLE

```
class Machine{

    public void start(){
        System.out.println("Machine Started");
    }
}

class Camera extends Machine{
    public void start(){

        System.out.println("Camera Started");
    }
    public void snap(){
        System.out.println("Photo taken");
    }
}

Machine machine1 = new Camera();
machine1.start();
```

Explanation is given in next slide..

CASTING OBJECTS EXAMPLE

- It will create an instance of Camera class with a reference of Machine class pointing to it.
- So, now the output will be "Camera Started" The variable is still a reference of Machine class.

- If you attempt machine1.snap(); the code will not compile

- The takeaway here is all Cameras are Machines since Camera is a subclass of Machine but all Machines are not Cameras. So you can create an object of subclass and point it to a super class reference but you cannot ask the super class reference to do all the functions of a subclass object(In our example machine1.snap() wont compile).

- The superclass reference has access to only the functions known to the superclass [In our example machine1.start()). You can not ask a machine reference to take a snap.]

INSTANCEOF

```
class Parent{ }

public class Child extends Parent
{
    public void check()     {
        System.out.println("Sucessfull Casting");
    }

    public static void show(Parent p)     {
       if(p instanceof Child)        {
           Child b1=(Child)p;
           b1.check();
       }
    }

    public static void main(String[] args)     {

      Parent p=new Child();
      Child.show(p);

      }
}
```

Used to check the type of an object at runtime, returns true or false.

INSTANCEOF

```
class Parent{}

class Child1 extends Parent{}

class Child2 extends Parent{}

class Test{
  public static void main(String[] args)  {
        Parent p =new Parent();
        Child1 c1 = new Child1();
        Child2 c2 = new Child2();

        System.out.println(c1 instanceof Parent);
        System.out.println(c2 instanceof Parent);
        System.out.println(p instanceof Child1);
        System.out.println(p instanceof Child2);

        p = c1;
        System.out.println(p instanceof Child1);
        System.out.println(p instanceof Child2);

        p = c2;
        System.out.println(p instanceof Child1);
        System.out.println(p instanceof Child2);

    }
}
```

CONSTRUCTOR CHAINING

```
class AA {
    AA() {
        System.out.println("Constructor of A running");
    }
}
class BB extends AA {
    BB() {
        //a call to super() is placed here autmatically
        System.out.println("Constructor of B running");
    }
}

public class ConstructorChaining {
    public static void main(String[] args) {
        new BB();
    }
}
```

```
output
======
Constructor of A running
Constructor of B running
```

SUMMARY

1) A class can extend only one class.
2) Every class is a sub class of Object directly or indirectly.

3) A **super** class reference can refer to a sub class object.

4) If reference is **super** class and object is of sub class then calling an instance method on it will result in execution of the method defined in sub class.

5) An overridden method has same signature and **return** type (or compatible with the **return** type) as that of **super** class method.

6) We can hide a static method defined in **super** class by defining a static method having same signature or **return** type (or compatible with the **return** type) as that of **super** class method.

7) When a sub class constructor runs the **super** class constructor also runs. This is called constructor chaining.

REFERENCES

- http://www.studytonight.com/java/dynamic-method-dispatch
- http://javapapers.com/core-java/covariant-return-type-in-java/
- http://www.vogella.com/tutorials/JavaIntroduction/article.html
- http://www.studytonight.com/java/dynamic-method-dispatch