

Chapter 11 Inheritance & Polymorphism

What is Inheritance?

การสืบทอด



คน

คุณสมบัติ

ชื่อ

อายุ



คุณครู

คุณสมบัติ

ชื่อ

อายุ

วิชาสอน



คุณหมอ

คุณสมบัติ

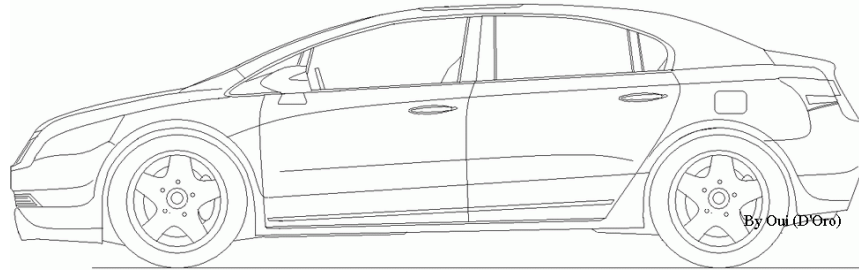
ชื่อ

อายุ

หมอเฉพาะทาง

What is Inheritance?

4 ล้อ
ขนาดรถยนต์



น้ำมัน



ไฟฟ้า



พลังงานไฮโดรเจน

What is Inheritance?



คุณสมบัติ
น้ำหนัก

คุณสมบัติ
น้ำหนัก
จำนวนชั่วโมง



คุณสมบัติ
น้ำหนัก
จำนวนเกียร์
รุ่น

Inheritance: Superclass, Subclass

More general properties
คุณสมบัติร่วม



คุณสมบัติ
น้ำหนัก



คุณสมบัติ
จำนวนชั่วโมง



คุณสมบัติ
จำนวนเกียร์
รุ่น

More specific properties : คุณสมบัติเฉพาะ

Superclass & Subclass

- * More generalized class is called “Superclass” คุณสมบัติร่วม
 - * Base class, Parent class
- * More specialized class is called “Subclass” คุณสมบัติเฉพาะ
 - * Extended class, Derived class, Child class

ถ้าไม่มี Inheritance

จักรยาน

น้ำหนัก



จักรยานไฟฟ้า

น้ำหนัก

จำนวนชั่วโมง



เขียน code ซ้ำ
ไม่สามารถ reuse ได้

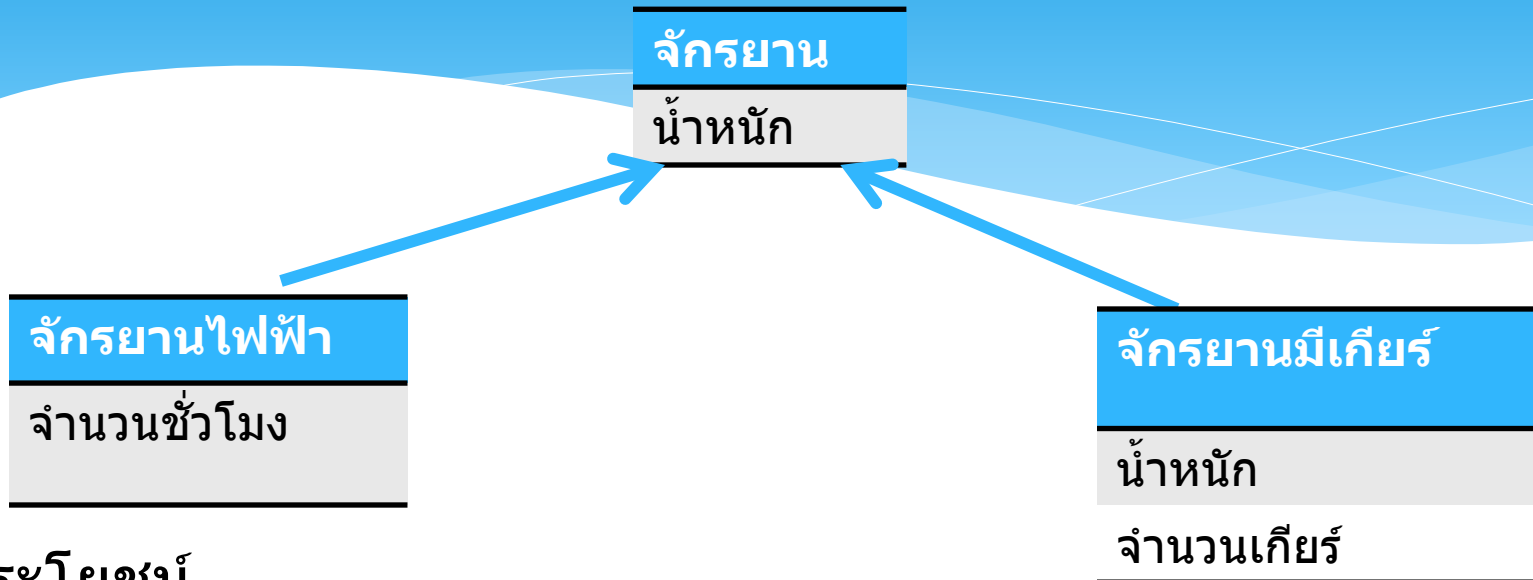
จักรยานมีเกียร์

น้ำหนัก

จำนวนเกียร์



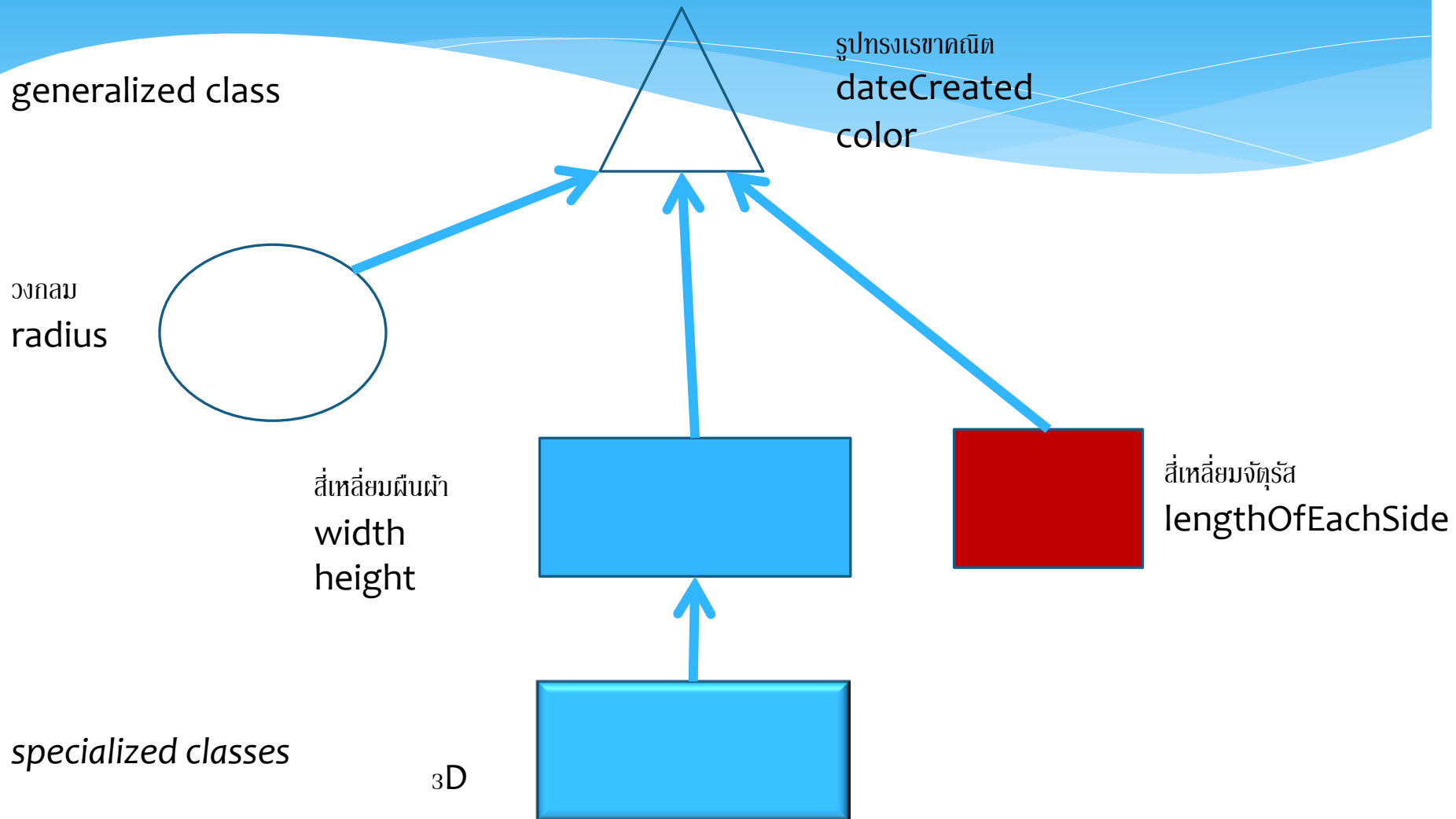
ถ้ามี Inheritance



ประโยชน์

1. ทำให้การออกแบบ Class มีประสิทธิภาพมากยิ่งขึ้น
 - ไม่ต้องเขียน code ซ้ำ
 - คลาสสามารถนำกลับมาใช้ได้อีก (reuse)
 - จัดกลุ่มคลาส
2. Subclass สามารถเรียกใช้งานคำสั่งต่าง ๆ จาก Superclass ได้เลยโดยไม่ต้องสร้าง object

ตัวอย่าง Inheritance : hierarchy



How to create Inheritance?

ใช้ extends

```
class Cycle {  
    int weight;  
}
```

```
class Bicycle extends Cycle{  
    String brand;  
}
```

Cycle

weight



Bicycle

brand



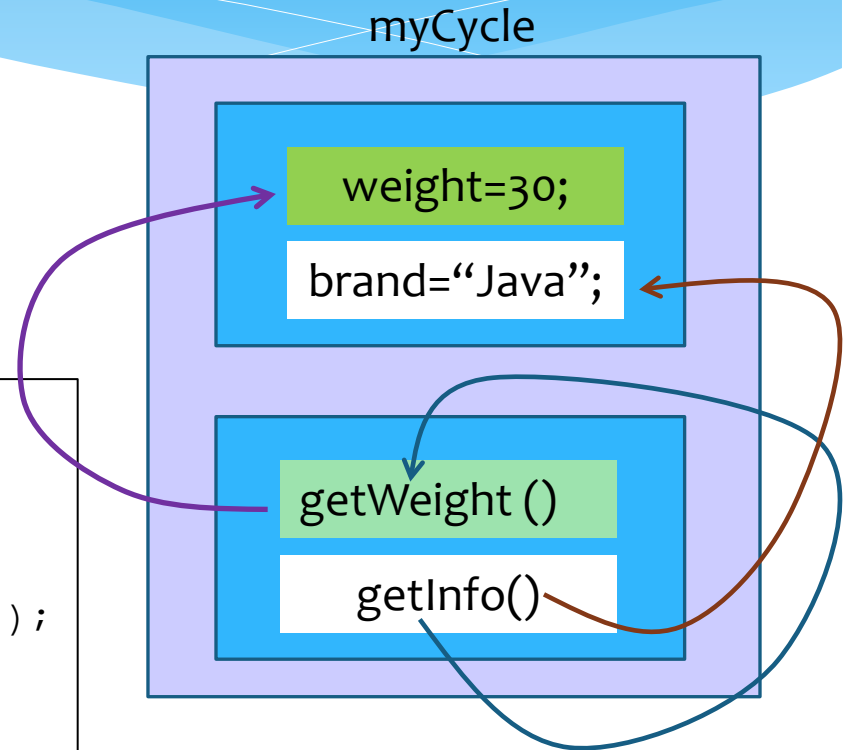
How to create Inheritance?

```
class Cycle {  
    private int weight =30;  
    public int getWeight(){  
        return weight;  
    }  
}
```

```
class Bicycle extends Cycle{  
    String brand = "Java";  
    public void getInfo(){  
  
system.out.println(brand+"", "getWeight()");  
    }  
}
```

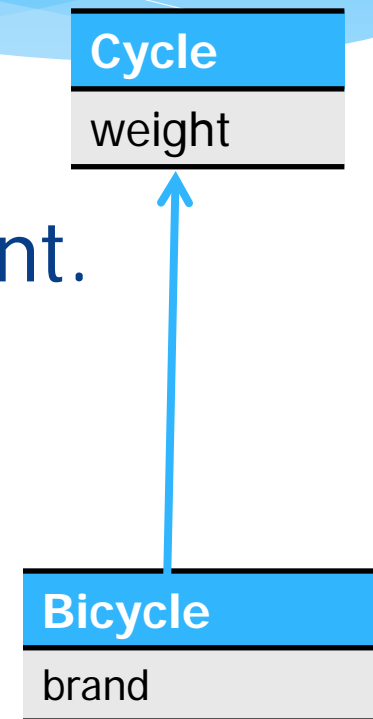
Subclass ไม่สามารถเข้าถึง
ส่วน variables และ methods
ของ superclass ที่ประกาศไว้
เป็น private

```
Bicycle myCycle = new Bicycle();  
myCycle.getInfo();
```



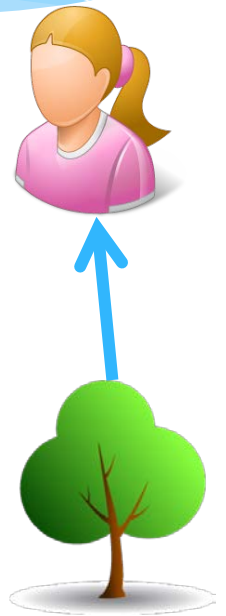
Inheritance: Is-a relationship

- * Bicycle is a Cycle.
- * Circle is a Geometry.
- * Undergraduate Student is a Student.
- * Car is a Vehicle.
- * House is a Building.
- * **But Building is not a House**



Is-a relationship

- * Square is a Rectangle. →side
- * Tree is a Person. →height
- * A Java class may inherit directly from only one superclass, known as *single inheritance*



Inheritance: extends

```
public class B{  
    private int num;  
    public B(int newNum) { num = newNum;}  
    public int getNum(){ return num;}  
}
```

สร้าง object
และวาดรูปคล้าย
slide 11

```
public class A extends B{  
    private double count;  
    public A(int newCount) { count = newCount;}  
    public int getCount(){ return count;}  
    public void getInfo(){ sout(count + ",num" + getNum());}  
}
```

*sout= System.out.println()

อะไรบ้างที่สืบทอดจาก Superclass สู่ Subclass

- *Constructor
- *Data field
- *Method


ไม่สืบทอด Constructor

จะถูกเรียกจาก Subclass แบบ implicit

หรือ แบบ explicit

```
public class A{  
    public A() { }  
}
```

```
public class Test{  
    public static void main(String[] args) {  
        B obj = new B(55);  
    }  
}
```



```
public class B extends A{  
    public B(double val) { }  
}
```

- ถ้าไม่มี implement การเรียก constructor ของ Superclass ที่ Subclass จะทำให้ คำสั่งsuper(); ถูกใส่เข้ามาให้อัตโนมัติที่ subclass
- ยกอีกตัวอย่างที่ไม่มีการประกาศ constructor ไว้ที่ใด

Constructor ถูกเรียกจาก Subclass แบบ explicit (first statement)

```
public class A{  
    private int num;  
    public A(int newNum) {  
        num = newNum; }  
}
```

```
public class Test{  
    public static void main(String[] args) {  
        B obj = new B(55);  
    }  
}
```

```
public class B extends A{  
    public B(double val) {  
        super(val);  
    }  
}
```

เมื่อใดสร้าง object จะมีผล
ทำให้ Constructor ของ
superclass ต้องถูกเรียกเสมอ

CAUTION

การเรียกใช้ constructor ของ superclass มีกฎดังนี้

1. ใช้คำสั่ง `super[(args)]` เพื่อเรียกใช้ constructor ของ superclass เสมอ ห้ามเรียกโดยใช้ชื่อคลาส
2. ถ้ามีคำสั่ง `super[(args)]` ต้องไว้เป็นบรรทัดแรก

Constructor Chaining

Constructing an instance of a class invokes all the superclasses' constructors along the inheritance chain. This is known as *constructor chaining*.

```
public class Faculty extends Employee {
    public static void main(String[] args) {
        new Faculty();
    }

    public Faculty() {
        System.out.println("(4) Faculty's no-arg constructor is invoked");
    }
}

class Employee extends Person {
    public Employee() {
        this("(2) Invoke Employee's overloaded constructor");
        System.out.println("(3) Employee's no-arg constructor is invoked");
    }

    public Employee(String s) {
        System.out.println(s);
    }
}

class Person {
    public Person() {
        System.out.println("(1) Person's no-arg constructor is invoked");
    }
}
```

นศ. วาด UML
ใส่คำสั่ง `super()`;



Trace Execution

```
public class Faculty extends Employee {  
    public static void main(String[] args) {  
        new Faculty();  
    }  
  
    public Faculty() {  
        System.out.println("(4) Faculty's no-arg constructor is invoked");  
    }  
}  
  
class Employee extends Person {  
    public Employee() {  
        this("(2) Invoke Employee's overloaded constructor");  
        System.out.println("(3) Employee's no-arg constructor is invoked");  
    }  
  
    public Employee(String s) {  
        System.out.println(s);  
    }  
}  
  
class Person {  
    public Person() {  
        System.out.println("(1) Person's no-arg constructor is invoked");  
    }  
}
```

1. Start from the
main method



Trace Execution

```
public class Faculty extends Employee {  
    public static void main(String[] args) {  
        new Faculty();  
    }  
  
    public Faculty() {  
        System.out.println("(4) Faculty's no-arg constructor is invoked");  
    }  
}  
  
class Employee extends Person {  
    public Employee() {  
        this("(2) Invoke Employee's overloaded constructor");  
        System.out.println("(3) Employee's no-arg constructor is invoked");  
    }  
  
    public Employee(String s) {  
        System.out.println(s);  
    }  
}  
  
class Person {  
    public Person() {  
        System.out.println("(1) Person's no-arg constructor is invoked");  
    }  
}
```

2. Invoke Faculty
constructor



Trace Execution

```
public class Faculty extends Employee {  
    public static void main(String[] args) {  
        new Faculty();  
    }  
}
```

```
    public Faculty() {  
        System.out.println("(4) Faculty's no-arg constructor is invoked");  
    }  
}
```

3. Invoke Employee's no-arg constructor

```
class Employee extends Person {  
    public Employee() {  
        this("(2) Invoke Employee's overloaded constructor");  
        System.out.println("(3) Employee's no-arg constructor is invoked");  
    }  
}
```

```
    public Employee(String s) {  
        System.out.println(s);  
    }  
}
```

```
class Person {  
    public Person() {  
        System.out.println("(1) Person's no-arg constructor is invoked");  
    }  
}
```



Trace Execution

```
public class Faculty extends Employee {  
    public static void main(String[] args) {  
        new Faculty();  
    }  
}
```

```
    public Faculty() {  
        System.out.println("(4) Faculty's no-arg constructor is invoked");  
    }  
}
```

4. Invoke Employee(String)
constructor

```
class Employee extends Person {  
    public Employee() {  
        this("(2) Invoke Employee's overloaded constructor");  
        System.out.println("(3) Employee's no-arg constructor is invoked");  
    }  
}
```

```
    public Employee(String s) {  
        System.out.println(s);  
    }  
}
```

```
class Person {  
    public Person() {  
        System.out.println("(1) Person's no-arg constructor is invoked");  
    }  
}
```




Trace Execution

```
public class Faculty extends Employee {  
    public static void main(String[] args) {  
        new Faculty();  
    }  
  
    public Faculty() {  
        System.out.println("(4) Faculty's no-arg constructor is invoked");  
    }  
}  
  
class Employee extends Person {  
    public Employee() {  
        this("(2) Invoke Employee's overloaded constructor");  
        System.out.println("(3) Employee's no-arg constructor is invoked");  
    }  
  
    public Employee(String s) {  
        System.out.println(s);  
    }  
}  
  
class Person {  
    public Person() {  
        System.out.println("(1) Person's no-arg constructor is invoked");  
    }  
}
```

5. Invoke Person() constructor

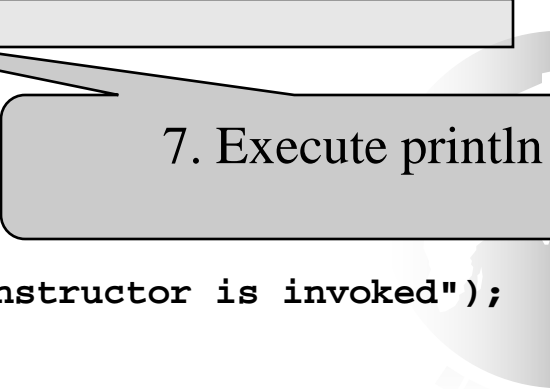
Trace Execution

```
public class Faculty extends Employee {  
    public static void main(String[] args) {  
        new Faculty();  
    }  
  
    public Faculty() {  
        System.out.println("(4) Faculty's no-arg constructor is invoked");  
    }  
}  
  
class Employee extends Person {  
    public Employee() {  
        this("(2) Invoke Employee's overloaded constructor");  
        System.out.println("(3) Employee's no-arg constructor is invoked");  
    }  
  
    public Employee(String s) {  
        System.out.println(s);  
    }  
}  
  
class Person {  
    public Person() {  
        System.out.println("(1) Person's no-arg constructor is invoked");  
    }  
}
```



Trace Execution

```
public class Faculty extends Employee {  
    public static void main(String[] args) {  
        new Faculty();  
    }  
  
    public Faculty() {  
        System.out.println("(4) Faculty's no-arg constructor is invoked");  
    }  
}  
  
class Employee extends Person {  
    public Employee() {  
        this("(2) Invoke Employee's overloaded constructor");  
        System.out.println("(3) Employee's no-arg constructor is invoked");  
    }  
  
    public Employee(String s) {  
        System.out.println(s);  
    }  
}  
  
class Person {  
    public Person() {  
        System.out.println("(1) Person's no-arg constructor is invoked");  
    }  
}
```



Trace Execution

```
public class Faculty extends Employee {  
    public static void main(String[] args) {  
        new Faculty();  
    }  
  
    public Faculty() {  
        System.out.println("(4) Faculty's no-arg constructor is invoked");  
    }  
}  
  
class Employee extends Person {  
    public Employee() {  
        this("(2) Invoke Employee's overloaded constructor");  
        System.out.println("(3) Employee's no-arg constructor is invoked");  
    }  
  
    public Employee(String s) {  
        System.out.println(s);  
    }  
}  
  
class Person {  
    public Person() {  
        System.out.println("(1) Person's no-arg constructor is invoked");  
    }  
}
```



8. Execute println

Trace Execution

```
public class Faculty extends Employee {  
    public static void main(String[] args) {  
        new Faculty();  
    }  
  
    public Faculty() {  
        System.out.println("(4) Faculty's no-arg constructor is invoked");  
    }  
}  
  
class Employee extends Person {  
    public Employee() {  
        this("(2) Invoke Employee's overloaded constructor");  
        System.out.println("(3) Employee's no-arg constructor is invoked");  
    }  
  
    public Employee(String s) {  
        System.out.println(s);  
    }  
}  
  
class Person {  
    public Person() {  
        System.out.println("(1) Person's no-arg constructor is invoked");  
    }  
}
```

9. Execute println



อะไรบ้างที่สืบทอดจาก Superclass สู่ Subclass

- * Constructor

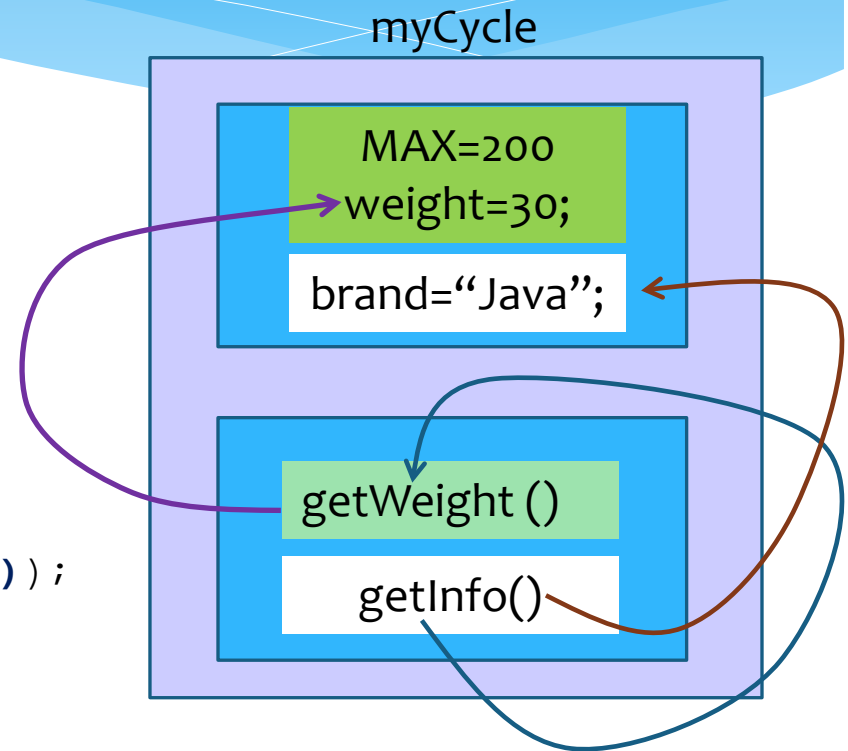
- * Data field

- * Method

A subclass inherits
accessible
data fields and methods
(ทั้ง static and instance)
from its superclass and
may also add new data
fields and methods.

Inheritance: data fields, methods

```
class Cycle {  
    private int weight =30;  
    static final int MAX=200;  
    public int getWeight(){  
        return weight;  
    }  
}  
  
class Bicycle extends Cycle{  
    String brand = "Java";  
    public void getInfo(){  
        system.out.println(brand+" , "getWeight());  
        system.out.println(MAX)  
    }  
}
```



Subclass ไม่สามารถเข้าถึง ส่วน
variables และ methods ของ
superclass ที่ประกาศไว้เป็น private

```
Bicycle myCycle = new Bicycle();  
myCycle.getInfo();
```

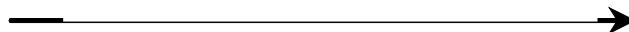
The protected Modifier

- * `protected` สามารถวางไว้ที่ data หรือ เมธอด

ถ้าส่วน data หรือเมธอด อยู่ใน super class ประเภท `public` จะทำให้ subclass ที่อยู่ใน package เดียวกัน หรือต่าง package สามารถเข้าถึง ส่วน data หรือ เมธอด ประเภท `protected` นี้ได้

- * `private`, `default`, `protected`, `public`

Visibility increases



`private`, `none` (if no modifier is used), `protected`, `public`

Visibility Modifiers

package p1;

```
public class C1 {  
    public int x;  
    protected int y;  
    int z;  
    private int u;  
  
    protected void m() {  
    }  
}
```

```
public class C2 {  
    C1 o = new C1();  
    can access o.x;  
    can access o.y;  
    can access o.z;  
    cannot access o.u;  
  
    can invoke o.m();  
}
```

```
public class C3  
    extends C1 {  
    can access x;  
    can access y;  
    can access z;  
    cannot access u;  
  
    can invoke m();  
}
```

package p2;

```
public class C4  
    extends C1 {  
    can access x;  
    can access y;  
    cannot access z;  
    cannot access u;  
  
    can invoke m();  
}
```

```
public class C5 {  
    C1 o = new C1();  
    can access o.x;  
    cannot access o.y;  
    cannot access o.z;  
    cannot access o.u;  
  
    cannot invoke o.m();  
}
```


Accessibility Summary

Modifier on members in a class	Accessed from the same class	Accessed from the same package	Accessed from a subclass	Accessed from a different package
public	✓	✓	✓	✓
protected	✓	✓	✓	–
default	✓	✓	–	–
private	✓	–	–	–

Defining a Subclass

A subclass inherits from a superclass. You can also:

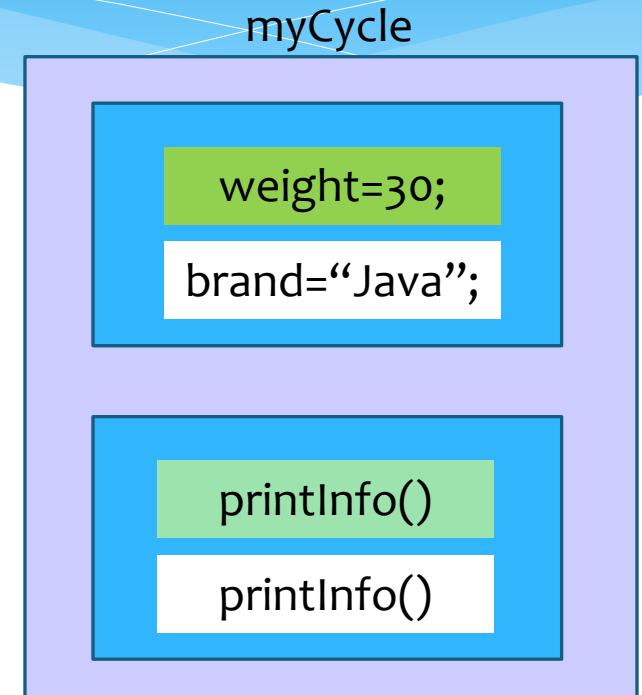
- ➡ Add new properties
- ➡ Add new methods
- ➡ Override the methods of the superclass (ได้เฉพาะ instance method)
 - ➡ to modify the implementation of a method defined in the superclass

Overriding method

```
class Cycle {  
    int weight = 30;  
    public void printInfo(){  
        sout(weight);  
    }  
}
```

```
class Bicycle extends Cycle{  
    String brand = "Java";  
    public void printInfo() {  
        sout(brand);  
    }  
}
```

```
Bicycle myCycle = new Bicycle();  
myCycle.printInfo();
```



Overriding method

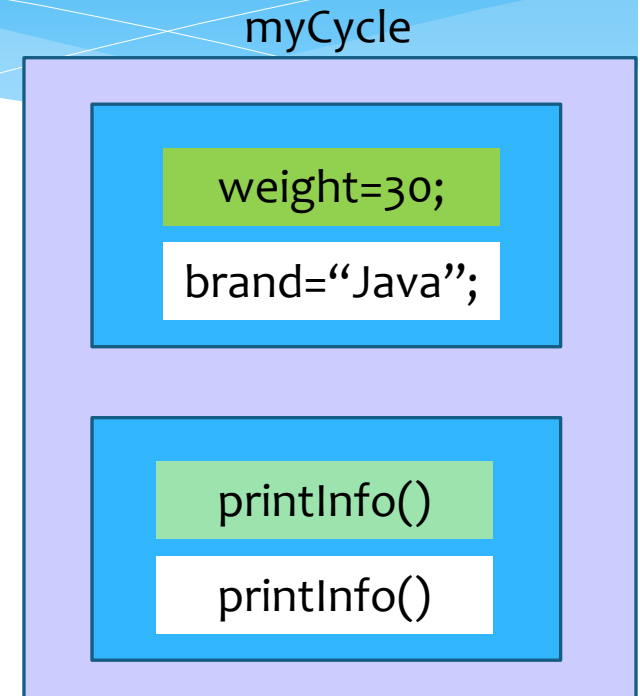
- * อยู่ในความสัมพันธ์แบบ Inheritance
- * เป็นเมธอดที่มีประกาศไว้แล้วใน Superclass โดย Subclass ได้นำเมธอดนั้นมาเขียนซ้ำเพื่อปรับปรุงการทำงานของเมธอดให้เป็นไปตามวัตถุประสงค์การทำงานของ Subclass เอง

Overriding method: ต้องการเรียก method ของ Superclass → `super.methodName([args]);`

```
class Cycle {  
    int weight = 30;  
    public void printInfo(){  
        sout(weight);  
    }  
}
```

```
class Bicycle extends Cycle{  
    String brand = "Java";  
    public void printInfo(){  
        sout(super.printInfo()+" , "+brand);  
    }  
}
```

```
Bicycle myCycle = new Bicycle();  
myCycle.printInfo();
```



Java.lang.Object

- * ทุกคลาสในภาษาจาวา มีบรรพบุรุษเป็นคลาส Object
- * ในคลาสนี้ มีเมธอดมากพอสมควรไว้ให้เรานำมาใช้งานเช่น
 - * toString(), equals()

```
public class Circle {  
    ...  
}
```

Equivalent

```
public class Circle extends Object {  
    ...  
}
```

เมธอด toString(): Default

- * The toString() method returns a string representation of the object.
- * **Default** implementation returns
 - * a string consisting of a class name of which the object is an instance, and
 - * the at sign (@), and a number representing this object.

```
Loan loan = new Loan();
```

```
System.out.println(loan.toString()); //Loan@15037e5
```

ควรจะ override เมธอด toString เพื่อแสดงค่าคุณสมบัติของ object ณ เวลา นั้นๆ

Overriding Methods: toString() จากคลาส Object

```
public class Geometry {  
    private String color = "white";  
    private boolean filled;  
    private java.util.Date dateCreated;  
    public String toString() { return "created on " + dateCreated + "\n" + color + " and filled: " + filled; }  
}
```

เรียกเมธอดของ Super class ด้วยคำสั่ง `super.methodName([args]);`

```
public class Circle extends Geometry {  
    private double radius;  
  
    /** Override the toString method defined in Geometry */  
    public String toString() {  
        return super.toString() + "\n" + radius is " + radius;  
    }  
}
```


The equals Method

The `equals()` method compares the **contents of two objects**. The default implementation of the equals method in the `Object` class is as follows:

```
public boolean equals(Object obj) {  
    return this == obj;  
}
```

For example, the equals method is overridden in the **Circle** class.

```
public boolean equals(Object o) {  
    if (o instanceof Circle) {  
        return radius == ((Circle o).radius;  
    }  
    else  
        return false;  
}
```

กฎ overriding ที่ควรทราบ

- * ไม่สามารถ override เมธอดแบบ private ได้
 - * ถ้าทำ ถือว่าเป็นอิสระต่อกัน
- * ไม่สามารถ override เมธอดแบบ static ได้
 - * ถ้าทำ ใน subclass จะมองไม่เห็นเมธอดแบบ static ของ superclass
- * เมื่อมีการ override แล้วระดับการเข้าถึงของ subclass ห้ามต่ำกว่าของ superclass

Overriding vs. Overloading

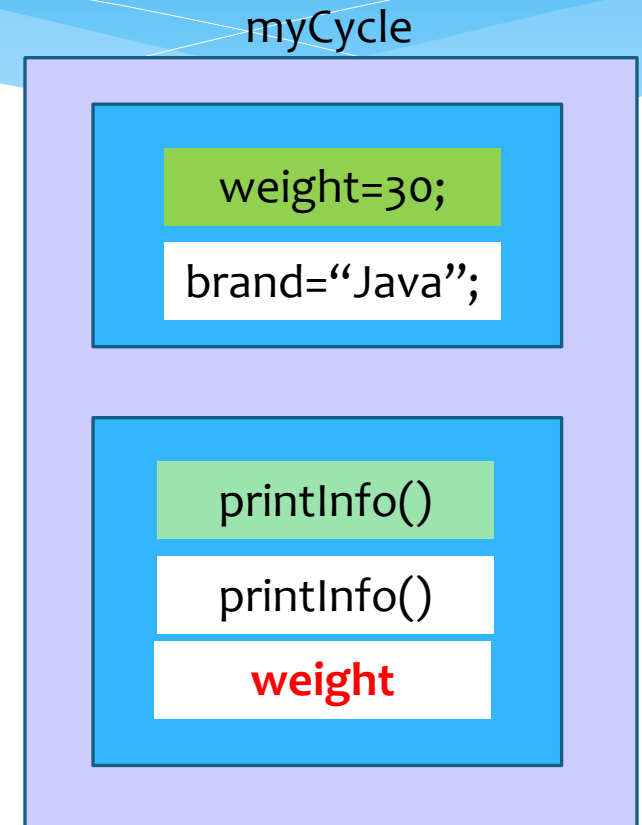
```
public class Test {  
    public static void main(String[] args) {  
        A a = new A();  
        a.p(10);  
        a.p(10.0);  
    }  
}  
  
class B {  
    public void p(double i) {  
        System.out.println(i * 2);  
    }  
}  
  
class A extends B {  
    // This method overrides the method in B  
    public void p(double i) {  
        System.out.println(i);  
    }  
}
```

```
public class Test {  
    public static void main(String[] args) {  
        A a = new A();  
        a.p(10);  
        a.p(10.0);  
    }  
}  
  
class B {  
    public void p(double i) {  
        System.out.println(i * 2);  
    }  
}  
  
class A extends B {  
    // This method overloads the method in B  
    public void p(int i) {  
        System.out.println(i);  
    }  
}
```

สิ่งที่ต้องระวัง: no compile error

```
class Cycle {  
    private int weight = 30;  
    public void printInfo(){  
        System.out.println(weight);  
    }  
}  
class Bicycle extends Cycle{  
    String brand = "Java";  
    int weight;  
    public void printInfo(){  
        System.out.println(brand+", "+ weight);  
    }  
}
```

```
Bicycle myCycle = new Bicycle();  
myCycle.printInfo();
```



The `final` Modifier

- * The `final` class cannot be extended:

```
final class Math {  
    ...  
}
```

- * The `final` variable is a constant:

```
final static double PI = 3.14159;
```

- * The `final` method cannot be overridden by its subclasses.

NOTE

- The **modifiers** are used on classes and class members (data and methods), except that the **final** modifier can also be used on **local variables** in a method.
- A final local variable is a **constant** inside a method.

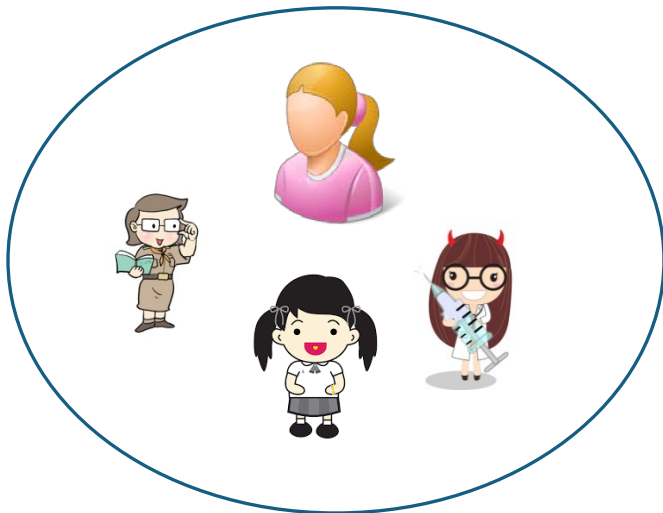
Polymorphism

- * เกิดในกรณีที่คลาสมีความสัมพันธ์ แบบ Inheritance
- * การมีหลายรูปแบบ
 - * สร้าง object ได้หลายรูปแบบ
 - * การเรียกใช้ overriding method ทำให้การตอบสนองการเรียกใช้เมธอดดังกล่าว ตอบสนองออกมาหลายรูปแบบ

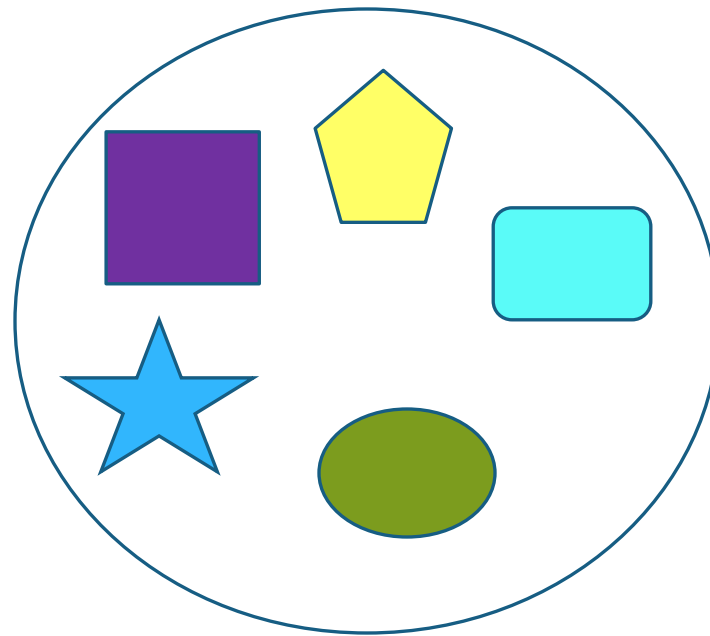
Polymorphism

สร้าง object ได้
หลายรูปแบบ

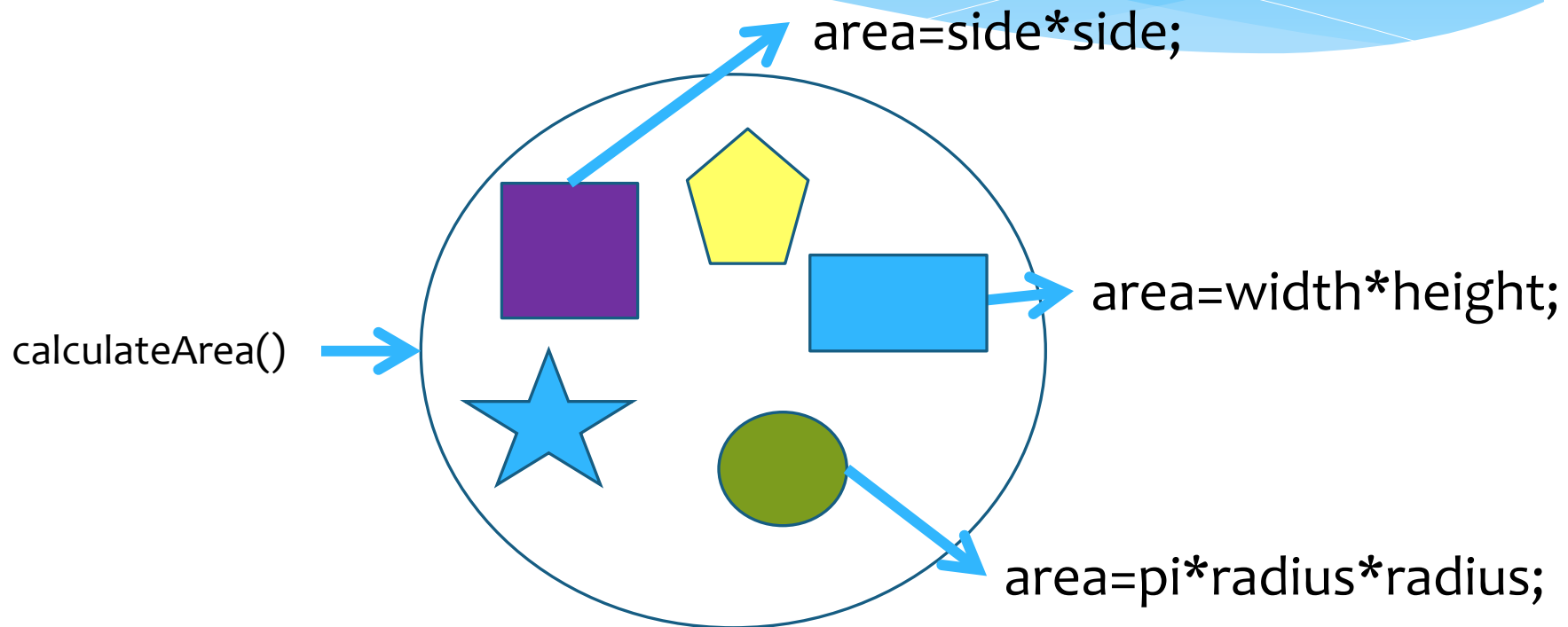
Person



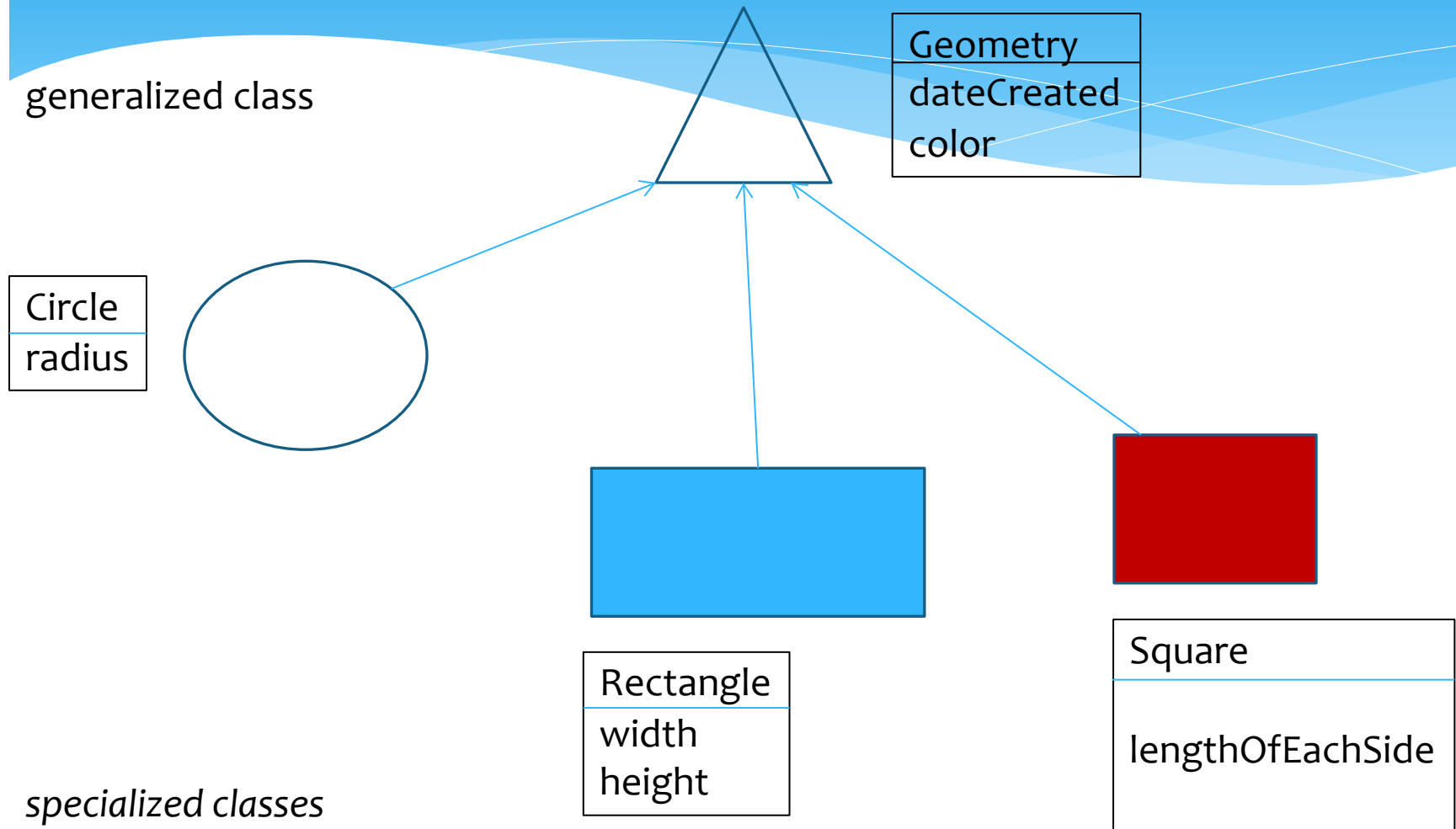
Geometry



Polymorphism: overriding method



สร้าง object ได้หลายรูปแบบ



สร้าง object ได้หลายรูปแบบ

```
Geometry obj1 = new Circle();  
Geometry obj2 = new Square();  
Geometry obj3 = new Rectangle();
```

```
Geometry objArray[] = {new Circle() , new Square(),  
                        new Rectangle()};
```

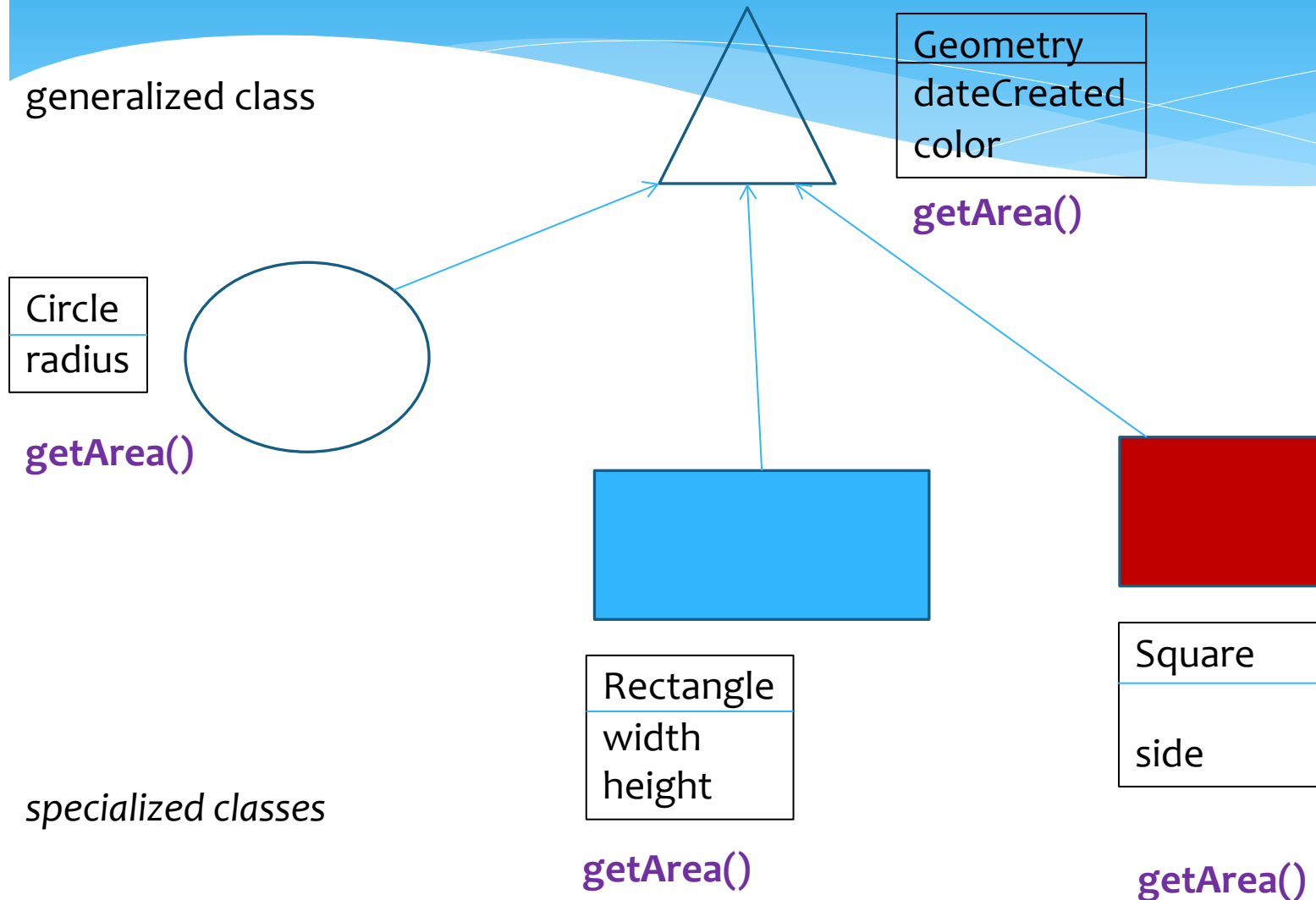
//เพราะ object ของ subclass ถือว่าเป็น object ของ superclass ด้วย

Circle เป็น subtype ของ Geometry
Geometry เป็น supertype ของ Circle

จำนวน object? Subset?

- * เมื่อมีการสร้าง objects แล้ว objects เหล่านั้น ใครเป็น เจ้าของได้มากกว่ากัน ระหว่าง superclass และ subclass
- * Subclass เป็น subset ของ superclass หรือไม่

Dynamic binding



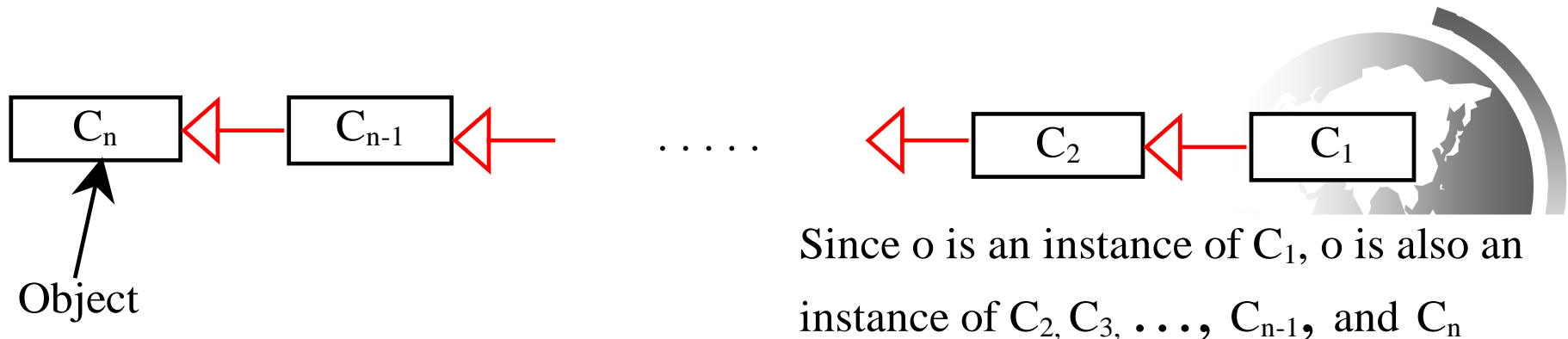
Dynamic binding

```
public class test {  
    public static void main(String[] args) {  
        Geometry objArray[] = {new Circle() , new Square(),  
                                new Rectangle()};  
  
        for (int i = 0; i < objArray.length; i++) {  
            System.out.println(objArray[i].getArea());  
        }  
    }  
}
```

ดังนั้นเมธอด **getArea()** ของแต่ละคลาสจะ **implement** การคำนวณพื้นที่ตามรูปแบบของตัวเอง

Dynamic Binding

Dynamic binding works as follows: Suppose an object o is an instance of classes C_1, C_2, \dots, C_{n-1} , and C_n , where C_1 is a subclass of C_2 , C_2 is a subclass of C_3 , ..., and C_{n-1} is a subclass of C_n . That is, C_n is the most general class, and C_1 is the most specific class. In Java, C_n is the Object class. If o invokes a method p , the JVM searches the implementation for the method p in C_1, C_2, \dots, C_{n-1} and C_n , in this order, until it is found. Once an implementation is found, the search stops and the first-found implementation is invoked.



Method Matching vs. Binding

Matching a method signature and binding a method implementation are two issues.

- The compiler finds a matching method according to parameter type, number of parameters, and order of the parameters at compilation time.

- A method may be implemented in several subclasses. The Java Virtual Machine dynamically binds the implementation of the method at runtime.

ยกตัวอย่างประกอบ



Polymorphism, Dynamic Binding

```
public class PolymorphismDemo {
    public static void main(String[] args) {
        m(new GraduateStudent());
        m(new Student());
        m(new Person());
        m(new Object());
    }

    public static void m(Object x) {
        System.out.println(x.toString());
    }
}

class GraduateStudent extends Student {
}

class Student extends Person {
    public String toString() {
        return "Student";
    }
}

class Person extends Object {
    public String toString() {
        return "Person";
    }
}
```

Method `m` takes a parameter of the `Object` type. You can invoke it with any object.

An object of a subtype can be used wherever its supertype value is required. This feature is known as *polymorphism*.

When the method `m(Object x)` is executed, the argument `x`'s `toString` method is invoked. `x` may be an instance of `GraduateStudent`, `Student`, `Person`, or `Object`. Classes `GraduateStudent`, `Student`, `Person`, and `Object` have their own implementation of the `toString` method. Which implementation is used will be determined dynamically by the Java Virtual Machine at runtime. This capability is known as *dynamic binding*.

Animation

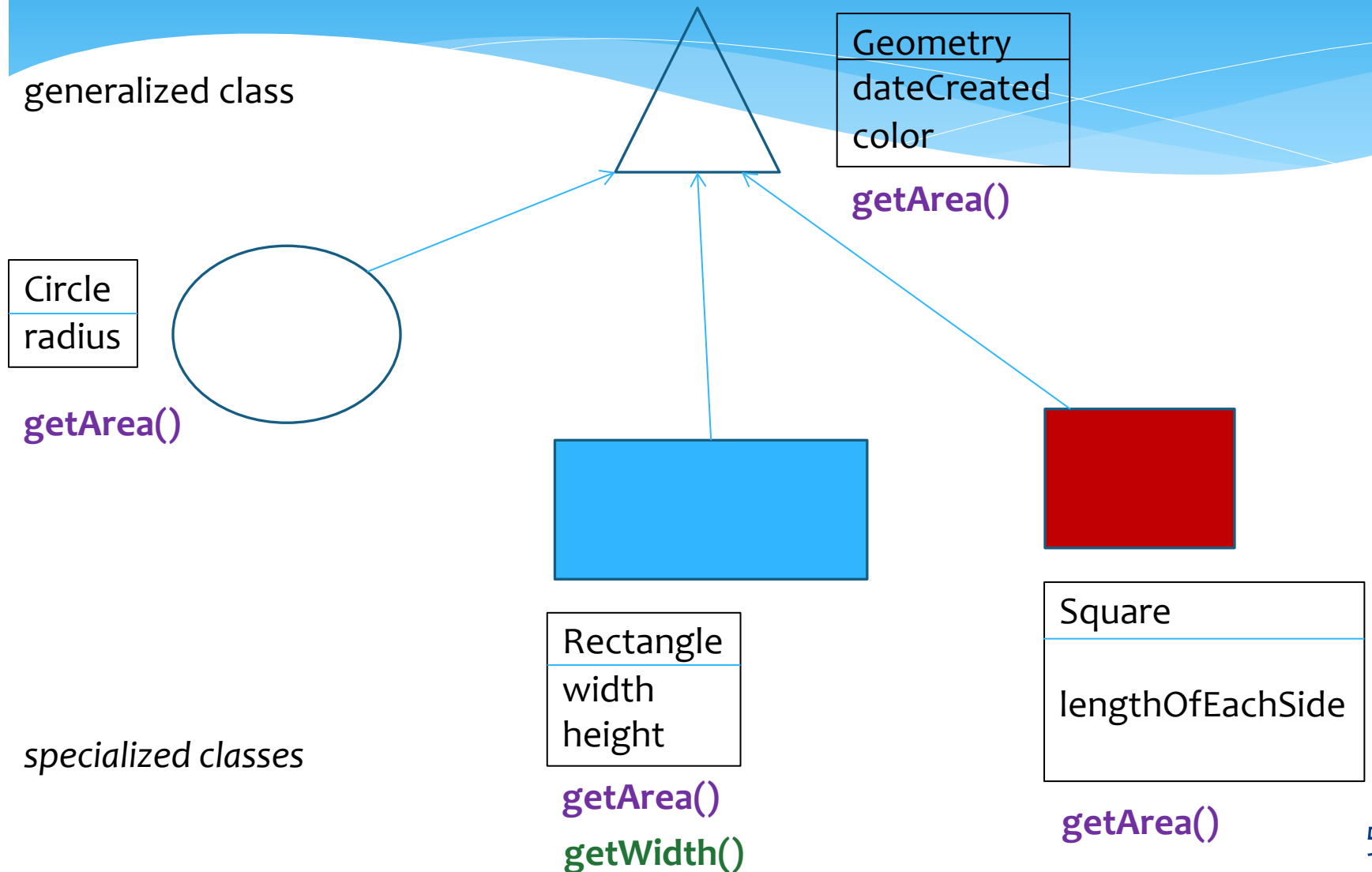
DynamicBindingDemo

Run

Generic programming

```
public class test {  
    public static void main(String[] args) {  
        m(new Square());  
        m(new Rectangle());  
        m(new Circle());  
  
    }  
    public static void m(Geometry x){  
        System.out.println(x.getArea());  
    }  
}
```

Casting object



Casting Objects

- * Implicit casting

Geometry obj = new Rectangle();

- * บางครั้งต้องการใช้บริการเมธอดของ subclass โดยตรง ที่ไม่ใช่กลุ่ม overriding method

- * `obj.getWidth();` // ไม่ได้เพราะตัวแปรนี้มีชนิดข้อมูลเป็น Geometry

- * ต้อง Explicit casting

- * `Rectangle rObj = obj;` // แบบนี้ไม่ได้ compiler ไม่ฉลาดพอ

- * `Rectangle rObj = (Rectangle)obj;`

- * `rObj.getWidth();`

Casting Objects: ไม่สำเร็จเสมอไป ควรใช้คำสั่ง instanceof

- * Geometry obj = new Rectangle();
Circle rObj = (Circle)obj;
// casting ไม่ได้ runtime error ขึ้น java.lang.ClassCastException
- * ควรตรวจสอบก่อน explicit casting ด้วยคำสั่ง instanceof
Geometry obj = new Rectangle();
if (obj instanceof Circle) {
 Circle rObj = (Circle)obj; // กรณีได้ค่าเป็นเท็จ ไม่เข้าทำ
 หรือ ((Circle)obj).getRadius(); //ใช้งานครั้งเดียว
}

Example on the Impact of a Superclass without no-arg Constructor

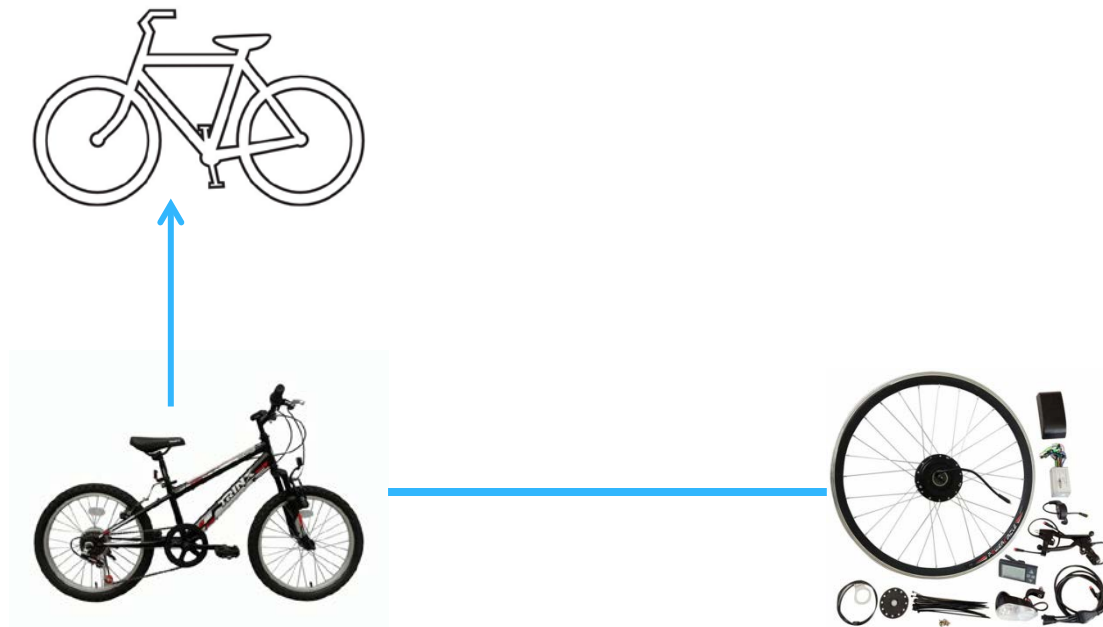
Find out the errors in the program:

```
public class Apple extends Fruit {  
}  
  
class Fruit {  
    public Fruit(String name) {  
        System.out.println("Fruit's constructor is invoked");  
    }  
}
```

Inheritance vs Composition

Is-a vs Has-a

แตกต่างกันอย่างไร



The ArrayList Class

You can create an array to store objects. But the array's size is fixed once the array is created. Java provides the ArrayList class that can be used to store an unlimited number of objects.

java.util.ArrayList<E>

```
+ArrayList()  
+add(o: E) : void  
+add(index: int, o: E) : void  
+clear(): void  
+contains(o: Object): boolean  
+get(index: int) : E  
+indexOf(o: Object) : int  
+isEmpty(): boolean  
+lastIndexOf(o: Object) : int  
+remove(o: Object): boolean  
+size(): int  
+remove(index: int) : boolean  
+set(index: int, o: E) : E
```

Creates an empty list.

Appends a new element *o* at the end of this list.

Adds a new element *o* at the specified index in this list.

Removes all the elements from this list.

Returns true if this list contains the element *o*.

Returns the element from this list at the specified index.

Returns the index of the first matching element in this list.

Returns true if this list contains no elements.

Returns the index of the last matching element in this list.

Removes the element *o* from this list.

Returns the number of elements in this list.

Removes the element at the specified index.

Sets the element at the specified index.

Generic Type

ArrayList is known as a **generic class with a generic type E**. You can **specify a concrete type to replace E** when creating an ArrayList. For example, the following statement creates an ArrayList and assigns its reference to variable cities. This ArrayList object can be used to store strings.

```
ArrayList<String> cities = new ArrayList<String>();
```

```
ArrayList<String> cities = new ArrayList<>();
```



TestArrayList

Run

Differences and Similarities between Arrays and ArrayList

Operation	Array	ArrayList
Creating an array/ArrayList	<code>String[] a = new String[10]</code>	<code>ArrayList<String> list = new ArrayList<>();</code>
Accessing an element	<code>a[index]</code>	<code>list.get(index);</code>
Updating an element	<code>a[index] = "London";</code>	<code>list.set(index, "London");</code>
Returning size	<code>a.length</code>	<code>list.size();</code>
Adding a new element		<code>list.add("London");</code>
Inserting a new element		<code>list.add(index, "London");</code>
Removing an element		<code>list.remove(index);</code>
Removing an element		<code>list.remove(Object);</code>
Removing all elements		<code>list.clear();</code>



DistinctNumbers

Run

Array Lists from/to Arrays

Creating an **ArrayList** from an array of objects:

```
String[] array = {"red", "green", "blue"};  
    ArrayList<String> list = new  
ArrayList<>(Arrays.asList(array));
```

Creating an **array of objects** from an **ArrayList**:

```
String[] array1 = new String[list.size()];  
list.toArray(array1);
```



max and min in an Array List

```
String[] array = {"red", "green", "blue"};
```

```
System.out.println(java.util.Collections.max(  
    new ArrayList<String>(Arrays.asList(array))));
```

```
String[] array = {"red", "green", "blue"};
```

```
System.out.println(java.util.Collections.min(  
    new ArrayList<String>(Arrays.asList(array))));
```



Shuffling an Array List

```
Integer[] array = {3, 5, 95, 4, 15, 34, 3, 6, 5};  
ArrayList<Integer> list = new  
    ArrayList<>(Arrays.asList(array));  
java.util.Collections.shuffle(list);  
System.out.println(list);
```



The MyStack Classes

A stack to hold objects.



MyStack	
-list: ArrayList	
+isEmpty(): boolean	
+getSize(): int	
+peek(): Object	
+pop(): Object	
+push(o: Object): void	
+search(o: Object): int	

A list to store elements.

Returns true if this stack is empty.

Returns the number of elements in this stack.

Returns the top element in this stack.

Returns and removes the top element in this stack.

Adds a new element to the top of this stack.

Returns the position of the first element in the stack from the top that matches the specified element.