Object Oriented Programming

LESSON 04

Delegation and Inheritance (P1)

Outline

1. What is Delegation?
2. Aggregation/Composition
3. Inheritance
4. Generalization
5. Specification

Overview

In this chapter, you are going to learn about

- Know in details about delegation and code re-use
- Know Aggregation and Composition
- Know inheritance
- Know Generalization
- Know Specification

Learning content

1. What is Delegation?
   - Re-use
   - Delegation concept
   - Delegation example
2. Aggregation/Composition
   - Aggregation
   - Composition
   - Aggregation and Composition in UML
3. Inheritance
   - Introduction Example
   - Creating Inheritance
   - Using instances of inherited class
4. Generalization
   - Class hierarchy
   - More General, wider use
   - Example
5. Specification
   - Class hierarchy
   - More Specific, deeper details
   - Example

Pre-Test

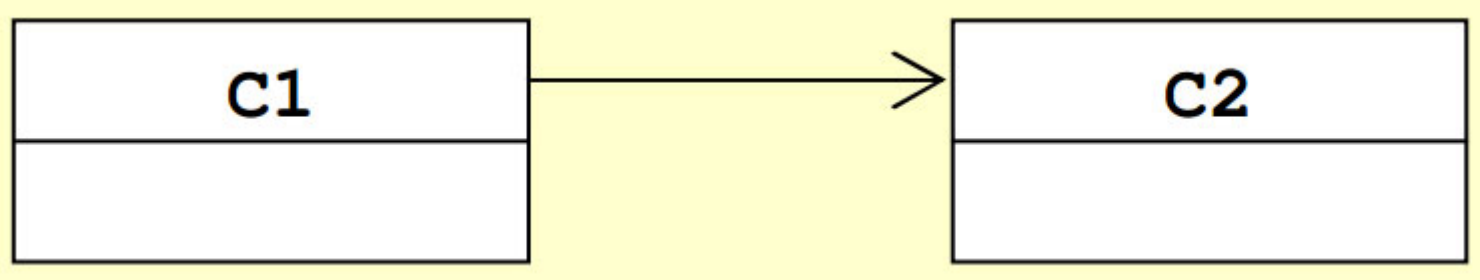
Question	Possible answers	Correct Answer	Question Feedback
1. Which one of the action bellow, that can't be done by yourself?	a. Reading book b. Hair style cutting c. Cooking	b. Hair style cutting	You can read the book by yourself; or cooking by yourself
2. Which of the following is not in the same category?	a. Elephant b. Eggplant c. Cow	b. Eggplant	Eggplant is plant and the other 2 are animals
3. Which of the following that have similar characteristics?	a. Whale b. Cat fish c. Cat	a, b	Cat lives on dry land, but cat fish and whale are living in water.

1.1. Re-use

- How to use a class as a building block to design other classes?
- In object concept, we defined associations (relations) to express reuse between class.
- UML defined typologies for all possible associations between classes. In this introductio n we will focus on two forms of association
  - An object can call another object: Delegation
  - An object can be created from “mold” of another object: inheritance

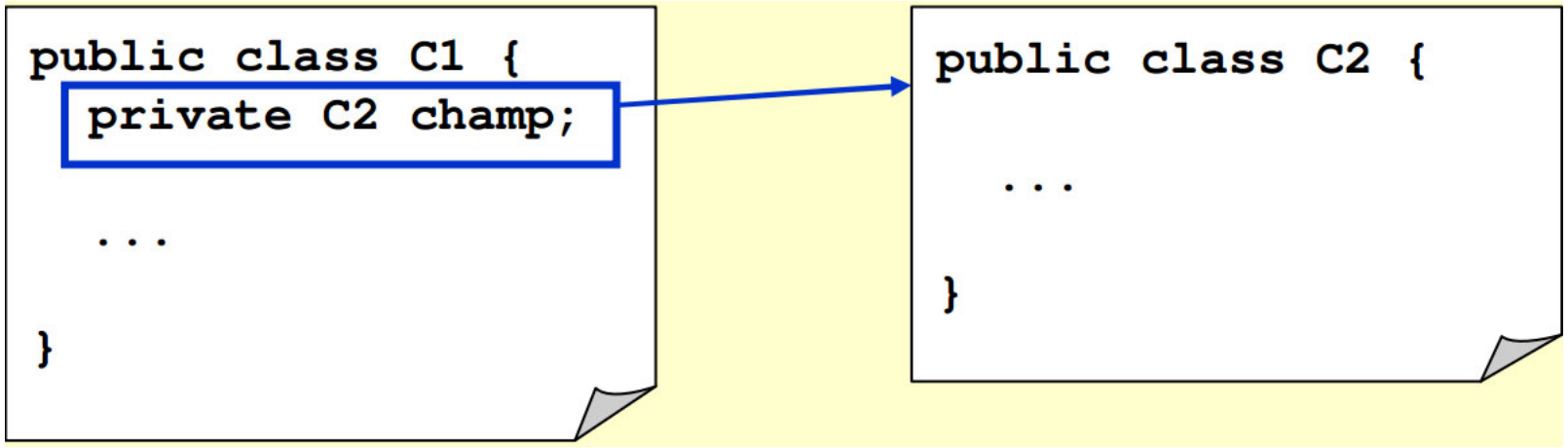
1.2. Delegation concept

- An object instance o1 of class C1 uses the services of an object instance o2 of class C2 (o1 delegates part of its activity to o2)
- Class C1 uses the services in Class C2
  - C1 is the client class
  - C2 is the waitress class



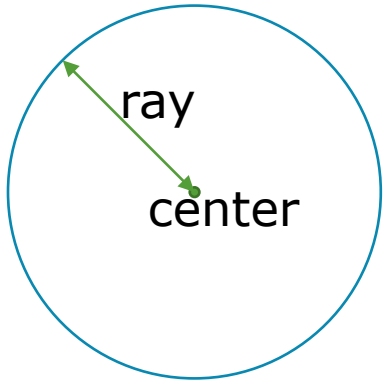
1.2. Delegation concept

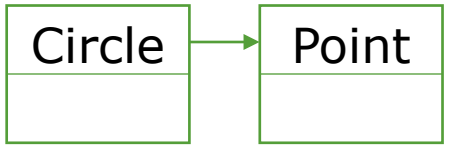
- The client class (C1) contains a reference of type of server class (C2)



1.3. Delegation Example

- Example class Cercle
  - ray: double
  - center: double x and double y or Point

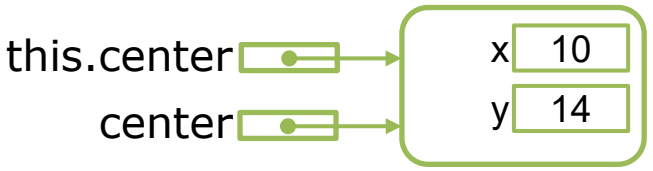





```
public class Circle {
  /**
   * center of circle
   */
  private Point center;
  /**
   * ray of circle
   */
  private double r;

  public void translate(double dx, double dy) {
    center.translate(dx, dy);
  }
}
```

2.1. Aggregation

- Action of an object of a class uses another object of other class
  - The point, that represents center of circle, exists autonomy (lifecycle independent)
  - It can be shared (at the same time it can be linked to multiple instances of other classes)

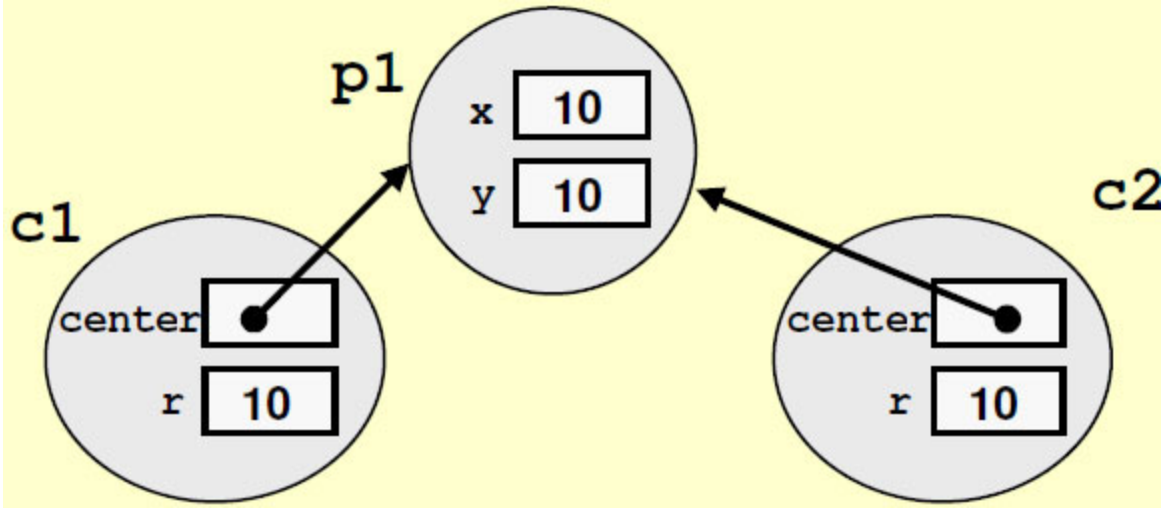



```java
public class Circle {
  /**
   * center of circle
   */
  private Point center;
  /**
   * ray of circle
   */
  private double r;
  public Circle(Point center, double r) {
    this.center = center;
    this.r = r;
  }
  public void translate(double dx, double dy) {
    center.translate(dx, dy);
  }
}
```

2.1. Aggregation

- With the content constructor above, the object center of the circle is shared, like in example below:

```
Point p1 = new Point(10,10);
Circle c1 = new Circle(p1,10);
Circle c2 = new Circle(p1,20);
```



2.1. Aggregation

- It (p1) can be used outside of circle (c1and c2) that p1 the center of these circles (Pay attention to side effect of it!)

```
Point p1 = new Point(10,10);
Circle c1 = new Circle(p1,10);
Circle c2 = new Circle(p1,20);
…
p1.rotate(90);
c2.translate(10,10);
```

  - When p1 is rotated 90^{o}, effected c1 and c2 to be rotated too.
  - When c2 translated dx=10 and dy=10, effected c1 to be translated too.
- In Aggregation, a waiter/waitress can serve multiple clients (p1 serves c1 and c2).

2.2. Composition

- The point that is the center of the circle is not shared (at the same time, an instance of Point inside of an object of class Circle, is linked to only one Circle)

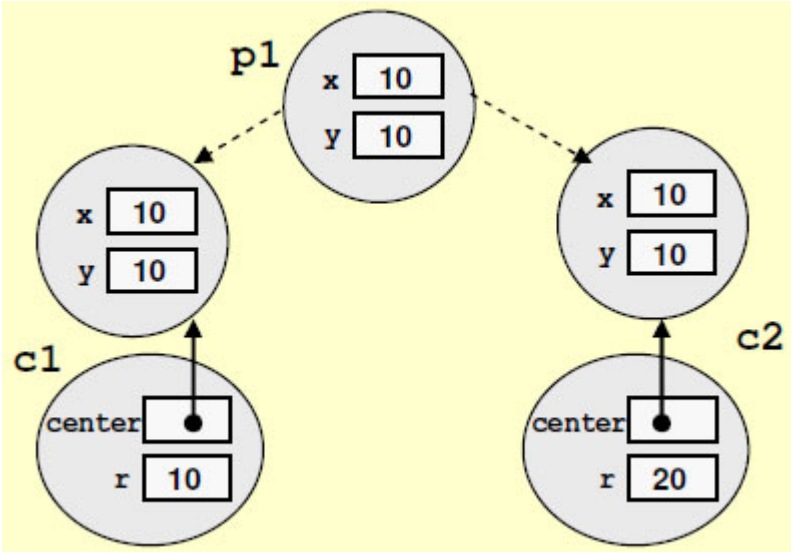
```
public class Circle {
  /**
   * center of circle
   */
  private Point center;
  /**
   * ray of circle
   */
  private double r;

  public Circle(Point center, double r){
    this.center = new Point(center);
    this.r = r;
  }
  public void translate(double dx, double dy) {
    center.translate(dx, dy);
  }
}
```

Created another copy of Point

2.2. Composition

- The life cycle of Point and Circle are linked: if the circle is destroyed, the center Point is also be destroyed. Like in example below:

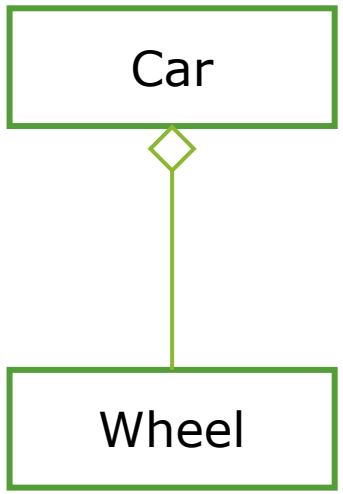
```
Point p1 = new Point(10,10);
Circle c1 = new Circle(p1,10);
Circle c2 = new Circle(p1,20);
…
p1.rotate(90);
c2.translate(10,10);
```



2.3. Aggregation and Composition in UML

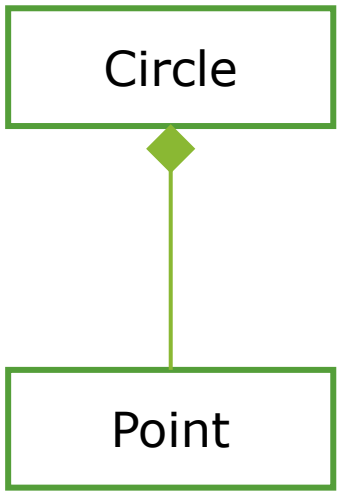
- UML distinguishes the 2 semantics by defining 2 types of relations:

Aggregation



The aggregated element (Wheel) has an existence autonomy outside aggregator (Car)

Composition



Strong Aggregation

At the same time, an instance of component (Point) can be linked to only one aggregator (Circle), and the component has life cycle depend on aggregator.

3.1. Introduction Example

- Problem:
  - An application needs services that only a part of it is different from an already defined class (we don't really need its source code)
  - Do not re-write the code

A Point:
- Has a position
- Can be moved
- Can calculate distance from origin
- …

Application needs to manipulate on points (like all functions in class Point) but in addition, it can be drawn on screen.

PointGraphic = Point
+ color
+ operation draw

  - In OOP: we use inheritance
    - Define a new class from an existing class

3.2. Creating Inheritance

- Class PointGraphic inherits from class Point

PointGraphic.java

```
public class PointGraphic extends Point {
  …
  Color color;
  public void draw(Graphics g){
    g.setColor(color);
    g.fillRect((int)x - 1,
               (int)y - 1, 3, 3);
  }
}
```

PointGraphic inherits from Point
It contains variables and methods defined in class Point

PointGraphic has defined a new attribute

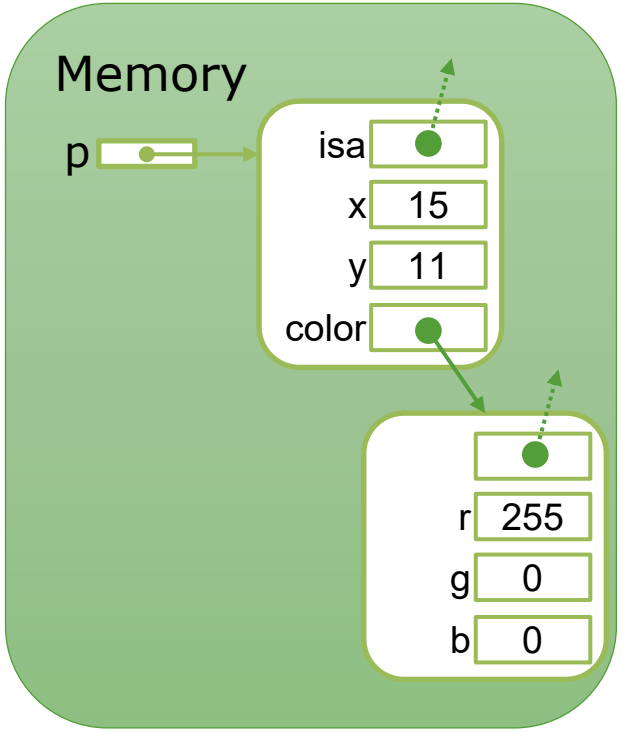
PointGraphic has defined a new attribute

Attribute that inherited from class Point

3.3. Using instances of inherited class

- An object instances of PointGraphic which contains attributes defined in PointGraphic and also attributes defined in Point (PointGraphic is also Point)
- An object instances of PointGraphic response to messages defined by methods described in class PointGraphic and also methods in class Point

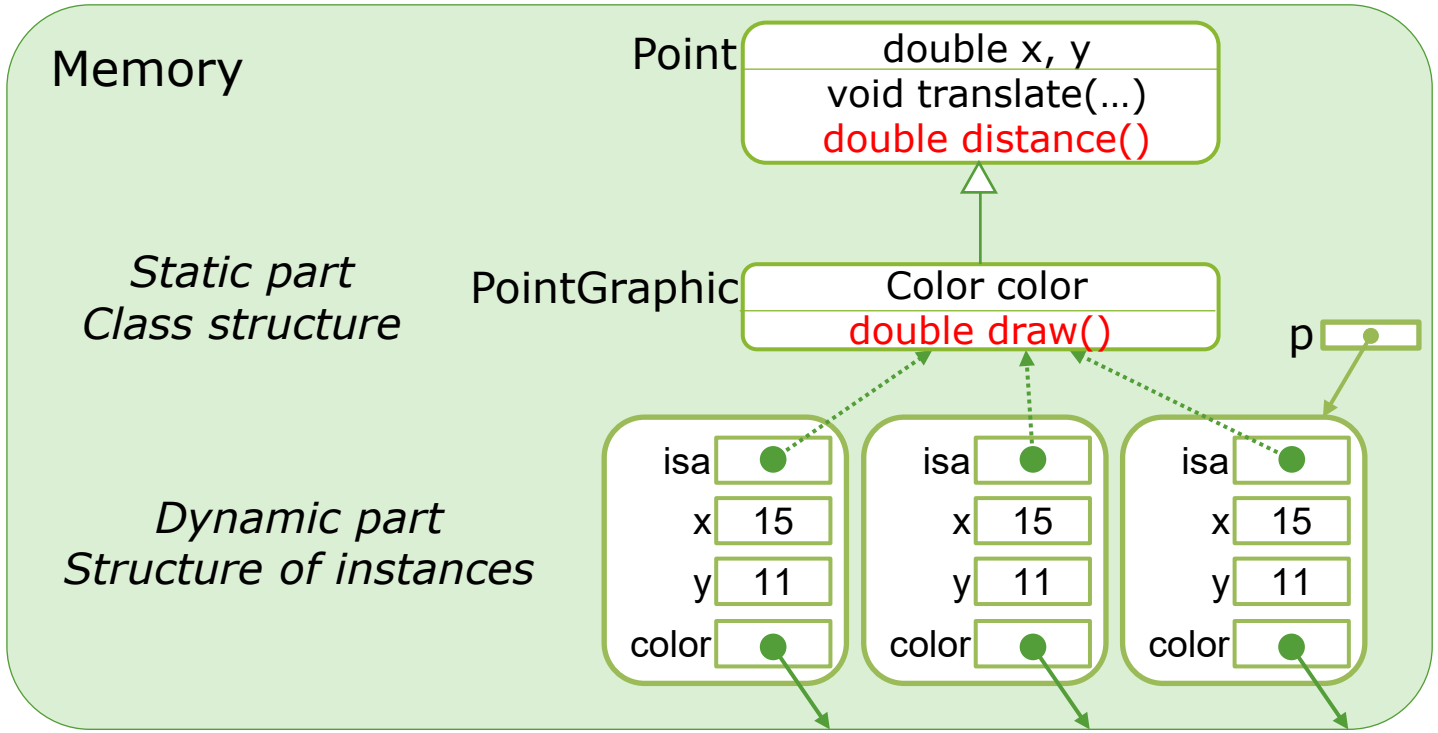
```
JFrame jframe = new JFrame();
jframe.setSize(200,200);
jframe.setVisible(true);
//jframe.setContentPane(new JPanel(null));
PointGraphic p = new PointGraphic(3, 5);
// using inherited instance variables
p.x = 15;
p.y = 11;
// using specific instance variable
p.color = new Color(255,0,0);
// using inherited methods
double dist = p.distance();
// using specific method
p.draw(jframe.getContentPane().getGraphics());
```



3.3. Using instances of inherited class

- Message resolution

```
PointGraphic p;
…
p.draw(g);
p.distance();
```



4.1. Class hierarchy

- Terminology
  - Inheritance allows reuse characteristics of existed class M for expansion and define new class F that inherited from M.
  - All objects of class F contains all characteristics of class M plus others defined in F
    - Point is parent class and PointGraphic is child class
    - Class PointGraphic inherited from class Point
    - Class PointGraphic is a sub-class of class Point
    - Class Point is super-class of class PointGraphic
  - The inheritance relationship can be viewed as "generalization/specialization" relationship between a class (super-class) and multiple classes which are more specific (sub-class)

4.2. More General, wider use

- Generalization express a relation "is-a" between a class and its super-class (each instance of it, is also describe in general as its super-class)
- If we want to record all kinds of elephants in Cambodia, we may create a class named "Elephant"
- If we want to record all kinds of butterflies in Cambodia, we may create a class named "Butterfly"
- But, if we want to records all kinds of animals in Cambodia, we may create a class named "Animal"
  - Animal is super-class of Elephant
  - Animal is super-class of Butterfly
  - Animal is more general than Elephant and Butterfly (because it can record elephant and also other kind of animals)
  - Animal wider use for recording all kinds of animals

4.3. Example

- Example on Animal, Elephant, and Butterfly classes

```java
public class Animal {
  double weight;
  int eyes;
  String name;
  String[] nicknames;
  String type;
  int legs, ears;
  void sleep(long timeInMillis){
    …
  }
  void eat(Object anotherObject){
    …
  }
}
```

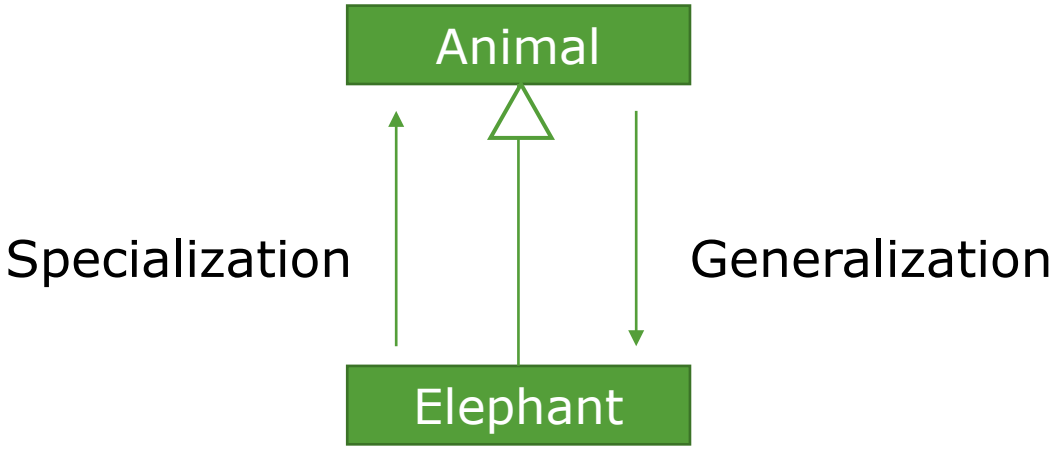
```java
public class Elephant extends Animal {
  int tails = 1;
  void run(int speed){ … }
  void walk(int speed){ … }
}
```

```java
import java.awt.Color;
public class Butterfly extends Animal {
  int wings = 2;
  Color color;
  double wingSize;
  void fly(int height){ … }
}
```

```java
public class AnimalRecord {
  public static void main(String[] args) {
    Animal animals[] = new Animal[3];
    animals[0] = new Butterfly();
    animals[1] = new Elephant();
    animals[2] = new Butterfly();
    // Do some display functions
  }
}
```

5.1. Class hierarchy

- Opposite to Generalization, the Specialization is more specific. The more specific, the more details into object that we want to represent in the real world.



- Specialization express a relation of “particularity” between a class and its sub-class (each instance of sub-class, is described in specific way)

5.2. More Specific, deeper details

- Utilization of inheritance:
  - In direction of “Specialization” for reusing code and incrementally modify the existing descriptions.
  - In direction of “Generalization” for abstraction by factoring the common properties to sub-classes.
- The sub-class will add new fields and methods for its specialty. The deeper of sub-class, the more special and narrow to type.

- Animal can represent animal all over the world including elephants, butterflies, and so on, but Elephant will represent only animal in kind of elephant

5.3. Example

- Example, list of Elephants:

```
public class ElephantRecord {
  public static void main(String[] args) {
    Elephant elephants[] = new Elephant[3];
    elephants[0] = new Elephant();
    elephants[1] = new Butterfly();
    elephants[2] = new Animal();
  }
}
```


  - In the list of elephants, we can not put Butterfly or Animal in elephant kind.
    - Simple explanation is that:
      - We say elephant run away, but we don’t say butterfly run away, we say butterfly fly away.
      - We say elephant has 4 legs and a tails but not for butterfly
  - We can’t add animal, because elephant has more characteristics than animal.

Test

Question	Possible Answers	Correct Answer
1) Fill in the gaps:	Delegation means an object of a class uses …… of another object of ……	
2) When an object own another object of another class, we call it:	a) Inheritance b) Association c) Aggregation d) Composition e) Class owner	
3) A child class that inherits from super class, will get:	a) Methods of super class b) Attributes of super class c) Static fields of super class d) Constructors of super class e) Constants of super class	

Practice

No.	Exercise	Solution
1.	Create Animal Family classes.	
2,	Create Employee hierarchy class	
3,	Implement class Point and Circle in Course	

Summarize

- Delegation is action of an object use services of other object of another class
- Aggregation is delegation that has shared reference of object of other class; and Composition is strong aggregation
- Inheritance is action of creating new class from existing class
- Generalization is used to categorize classes in the same or similar characteristics by providing common properties
- Specialization is used to add more detail information about specific case of super class

Reference

- "Object Oriented Programming" by Ph. Genoud – Université Joseph Fourier, 2006
- "Building Skills in Object-Oriented Design" by Steven F. Lott, 2009
- "Exercices en Java" 2nd edition by Groupe Eyrolles, 2006
- "Java Programming – Introductory" by Joyce Farrell, 1999
- "Java Examples in a Nutshell" 3rd Edition by David Flanagan, 2004
- "Programmer en Java" 5th edition by Groupe Eyrolles, 2008