

Encapsulation

The principle of encapsulation is that all of an object's data is contained and hidden in the object and access to it restricted to methods of that class.

- Code outside of the object cannot (or at least should not) directly access object fields.
- All access to an object's fields takes place through its methods.

Encapsulation allows you to:

- Change the way in which the data is actually stored without affecting the code that interacts with the object
- Validate requested changes to data
- Ensure the consistency of data — for example preventing related fields from being changed independently, which could leave the object in an inconsistent or invalid state
- Protect the data from unauthorized access
- Perform actions such as notifications in response to data access and modification

More generally, encapsulation is a technique for isolating change. By hiding internal data structures and processes and publishing only well-defined methods for accessing your objects.

Access Modifiers: Enforcing Encapsulation

- Access modifiers are Java keywords you include in a declaration to control access.
- You can apply access modifiers to:
 - Instance and static fields
 - Instance and static methods
 - Constructors
 - Classes
 - Interfaces (discussed later in this module)
- Two access modifiers provided by Java are:

- `private` (visible only within the same class)
- `public` (visible everywhere)

Note

There are two additional access levels that we'll discuss later.

Accessors (Getters) and Mutators (Setters)

- A common model for designing data access is the use of accessor and mutator methods.
- A mutator — also known as a **setter** — changes some property of an object.
 - By convention, mutators are usually named `setPropertyname`.
- An accessor — also known as a **getter** — returns some property of an object.
 - By convention, accessors are usually named `getPropertyname`.
 - One exception is that accessors that return a boolean value are commonly named `isPropertyName`.
- Accessors and mutators often are declared `public` and used to access the property outside the object.
 - Using accessors and mutators from code within the object also can be beneficial for side-effects such as validation, notification, etc.
 - You can omit implementing a mutator — or mark it `private` — to implement immutable (unchangeable) object properties.

Example: Access modifiers, Accessors, and Mutators

```
public class Employee {
    private String name;
    public void setName(String name) {
        if (name != null && name.length() > 0) {
            this.name = name;
        }
    }
    public String getName() {
        return this.name;
    }
}
```

Now, to set the name on an employee in `EmployeeTest.main()` (i.e., from the outside), you must call:

```
empOne.setName("Tom");
```

as opposed to:

```
empOne.name = "Tom"; // won't compile, name is hidden externally
```

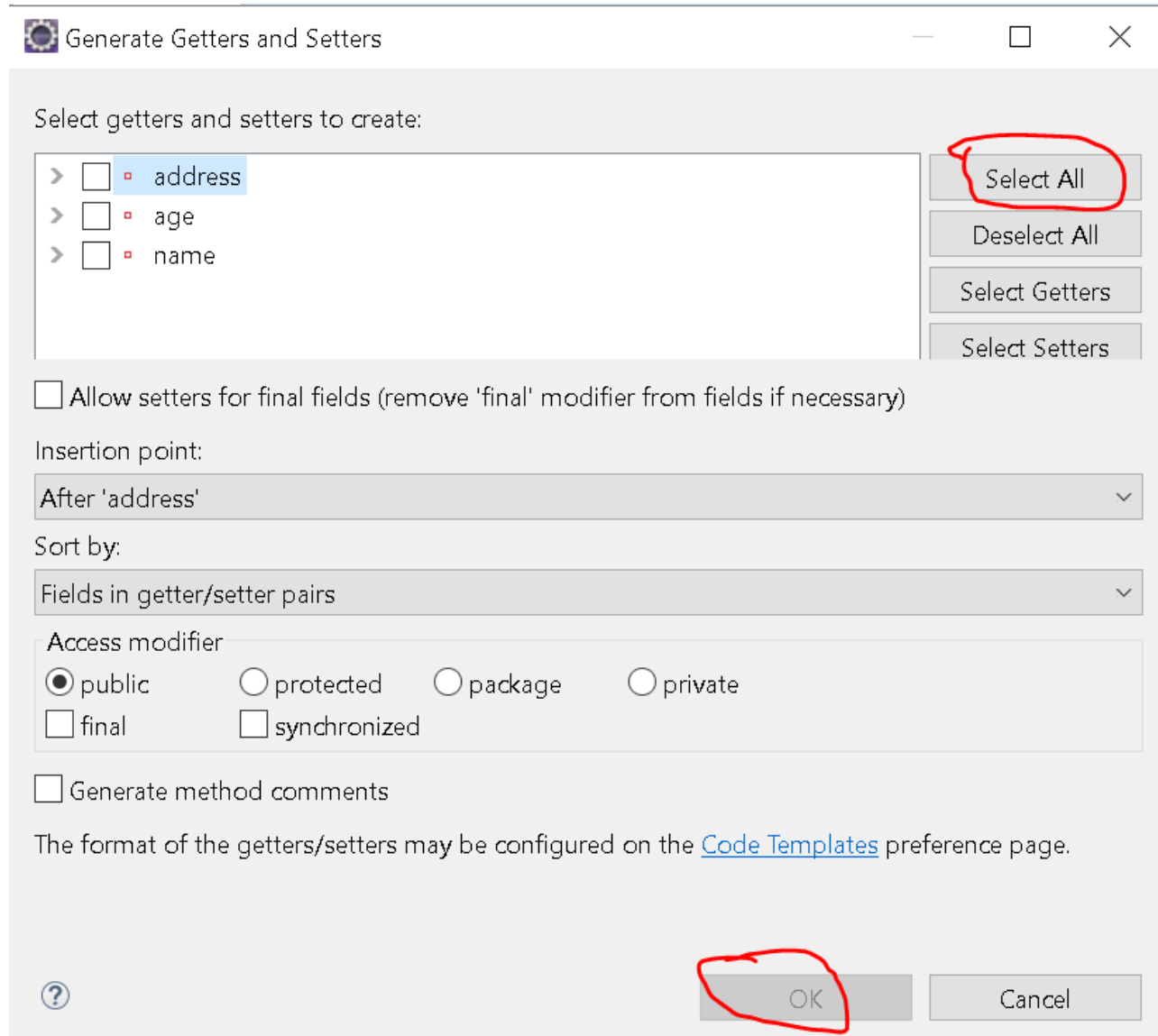
Use IDE to generate Getters and Setters

After finish coding the private properties, you can use Eclipse IDE to generate your Getters and Setters for you. For example, you've hand written the following code:

Example:

```
public class Employee {  
    private String name;  
    private int age;  
    private String address;  
}
```

Now in Eclipse, you right click to trigger the context menu, then hover on “**Source**” then select “**Generate Getters and Setters**”, you'll see:



Generate Getters and Setters

Select getters and setters to create:

- ☒ address
- ☐ age
- ☐ name

☐ Allow setters for final fields (remove 'final' modifier from fields if necessary)

Insertion point:

After 'address'

Sort by:

Fields in getter/setter pairs

Access modifier

☒ public ☐ protected ☐ package ☐ private

☐ final ☐ synchronized

☐ Generate method comments

The format of the getters/setters may be configured on the [Code Templates](#) preference page.

Click “Select All” then click “OK”, Eclipse will generate three getters and three setters for you.

Code After the operation:

```
public class Employee {  
    private String name;  
    private int age;  
    private String address;  
    public String getName() {  
        return name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
    public int getAge() {  
        return age;  
    }  
    public void setAge(int age) {
```

```
        this.age = age;
    }
    public String getAddress() {
        return address;
    }
    public void setAddress(String address) {
        this.address = address;
    }
}
```

Note the “*this*” keyword:

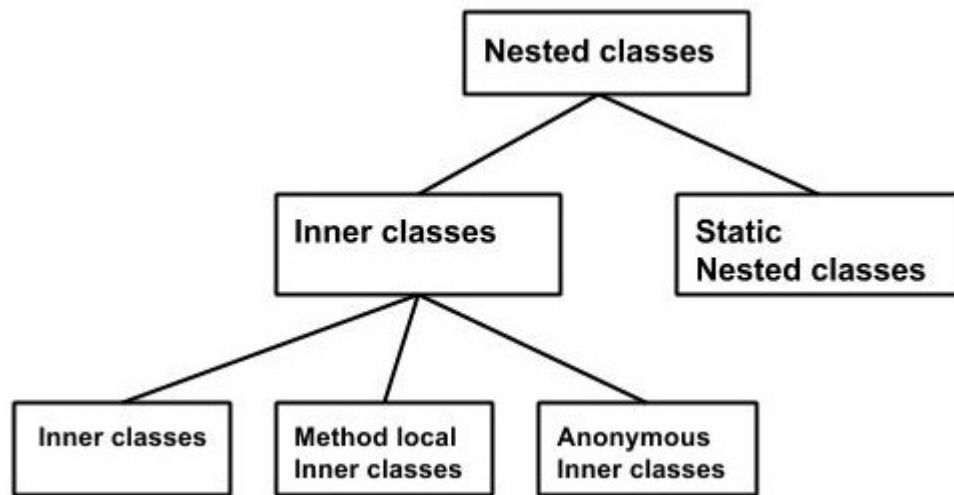
this is a keyword in Java. It can be used inside the Method or constructor of Class. It works as a reference to the current Object whose Method or constructor is being invoked. The ***this*** keyword can be used to refer to any member of the current object from within an instance Method or a constructor. ***this*** keyword can be very useful in the handling of *Variable Hiding*: as in the example above, we can create one instance variable & one local variable with the same name. In this scenario the local variable will hide the instance variable.

Nested classes

Writing a class within another is allowed in Java. The class written within is called the ***nested class***, and the class that holds the inner class is called the outer class. It looks like:

```
class Outer_Demo {
    class Nested_Demo {
    }
}
```

Different kinds of nested classes can be categorized into the following groups:



Inner Classes

Inner classes are a **security mechanism** in Java. We know a class cannot be associated with the access modifier `private`, but if we have the class as a member of other class, then the inner class can be made `private`. And this is also used **to access the private members** of a class.

Inner classes are of three types depending on how and where you define them. They are –

- (normal) Inner Class
- Method-local Inner Class
- Anonymous Inner Class

(normal) Inner Class

Creating an inner class is quite simple. You just need to write a class within a class. Unlike a class, an inner class can be `private` and once you declare an inner class `private`, it cannot be accessed from an object outside the class.

Example:

```
//outer class: People
public class People {
    private String name = "Eddie Zhao"; // private member of the outer
class
    //inner class
    public class Student {
```

```
String ID = "20170808"; // member of the inner class
// a method of the inner class
public void stuInfo() {
    System.out.println("you can actually access a private member of
another class!:" + name);
    System.out.println("ID : " + ID);
}
}
// a tester
public static void main(String[] args) {
    People a = new People(); // create an object out of the outer class
    People.Student b = a.new Student(); // using the outer-class-object
to create an inner-class object
    b.stuInfo(); // now you can call this inner-class object's method
}
}
```

Output:

you can actually access a private member of another class!:Eddie Zhao
ID : 20170808

To summarize:

1. class Student is a member of class People. Thus any access modifier can be used upon Student.
2. Student is inside People, so it can access all the members of People, the outer class.

Method-local Inner Class

In Java, we can write a class within a method and this will be a local type. Like local variables, the scope of the inner class is restricted within the method.

A method-local inner class can be instantiated only within the method where the inner class is defined. The following program shows how to use a method-local inner class.

Example:

```
public class People {
    // class Student is inside peopleInfo method
    public void peopleInfo() {
        final String sex = "man";
        class Student {
```

```
String ID = "20151234";

    public void print() {
        System.out.println("sex variable from outside : " + sex);
        System.out.println("ID:" + ID);
    }
}
Student a = new Student();
a.print();
}

public static void main(String[] args) {
    People b = new People();
    b.peopleInfo();
}
}
```

Output:

```
sex variable from outside : man
ID:20151234
```

Anonymous Inner Class

An inner class declared without a class name is known as an anonymous inner class. In case of anonymous inner classes, we ***declare and instantiate them at the same time***. Generally, they are used whenever you need to override the method of a class or an interface.

Example (Outer.java):

```
public class Outer {

    public Inner getInner(final String name, String city) {
        return new Inner() {
            private String nameStr = name;

            public String getName() {
                return nameStr;
            }
        };
    }

    public static void main(String[] args) {
        Outer outer = new Outer();
```



```
        Inner inner = outer.getInner("Inner", "NewYork");
        System.out.println(inner.getName());
        System.exit(0);
    }
}

interface Inner {
    String getName();
}
```

Output:

Inner

Static Nested Class

A static inner class is a nested class which is a static member of the outer class. It can be accessed without instantiating the outer class, using other static members. Just like static members, a static nested class does not have access to the instance variables and methods of the outer class.

Example:

```
public class People {
    private String name = "Eddie Zhao"; // a private member of outer class
    // a static member
    static String ID = "510xxx199X0724XXXX";

    // Static Nested Class 'Student'
    public static class Student {
        String ID = "20151234";

        public void stuInfo() {
            System.out.println("name from outer class: " + (new
People().name));
            System.out.println("a static member of outer class ID: " +
People.ID);
            System.out.println("member of inner class ID: " + ID);
        }
    }

    public static void main(String[] args) {
```

```
        Student b = new Student(); // can be created directly without the
outer class, don't need to do this:
                                // People a = new People(); then
    People.Student b = a.new Student();
        b.stuInfo();
    }
}
```

Further reading

Java Encapsulation

Advantage of java inner classes

Why prefer non-static inner classes over static ones?

How are Anonymous (inner) classes used in Java?

