# Lecture - Inheritance and Access modifiers

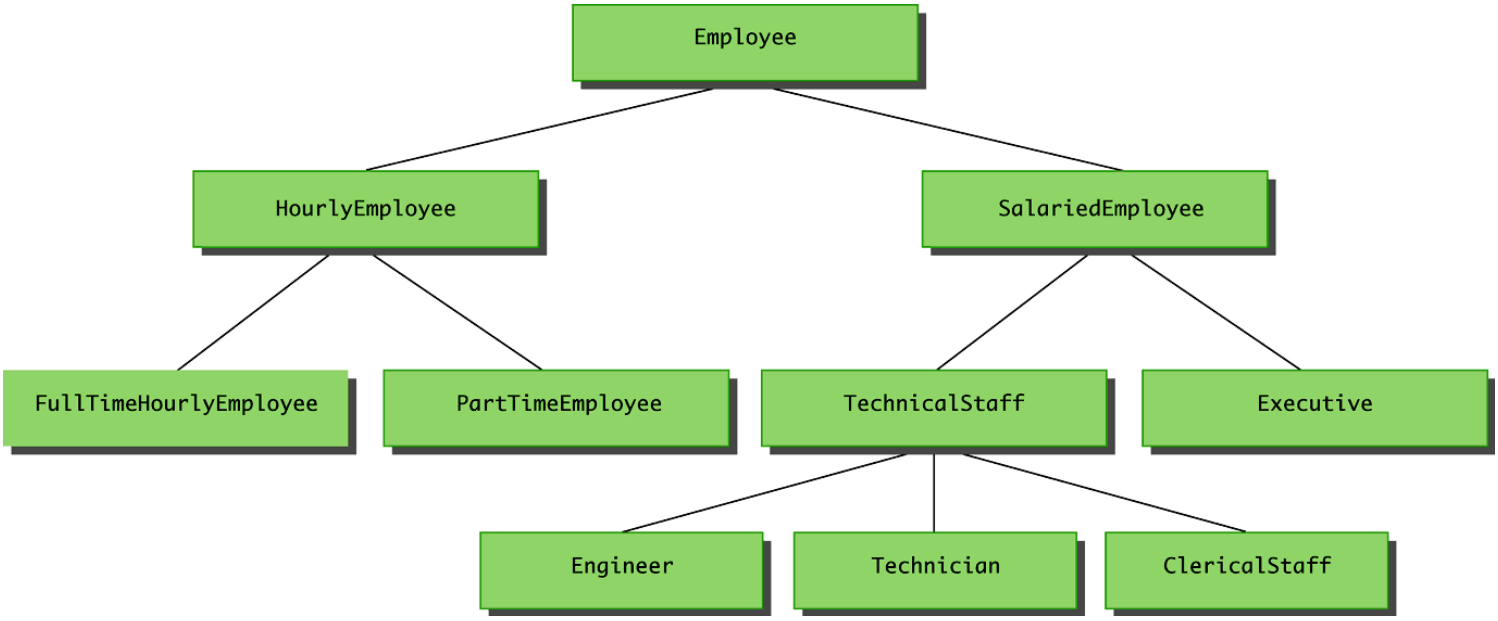
## Inheritance

*Inheritance* is one of the main techniques of object-oriented programming (OOP).

Using this technique, a very general form of a class is first defined and compiled, and then more specialized versions of the class are defined by adding instance variables and methods.

The specialized classes are said to *inherit* the methods and instance variables of the general class.

The more specialized versions (subclasses) can be further specialized, leading to a class hierarchy, like this:




Inheritance is the process by which a new class is created from another class.

- The new class is called a ***derived/child/sub*** class
- The original class is called the ***base/parent/super*** class

A derived class automatically has all the instance variables and methods that the base class has (except those with `private` or `package` scope), and it can have additional methods and/or instance variables as well.

A big advantage of Inheritance is is that it allows code to be reused, without having to copy it into the definitions of the derived classes.

## Derived classes

When designing certain classes, there is often a natural hierarchy for grouping them.

- In a record-keeping program for the employees of a company, there are *hourly employees* and *salaried employees*
- Hourly employees can be divided into *full time* and *part time* workers
- Salaried employees can be divided into those on *technical staff*, and those on the *executive staff*

All employees share certain characteristics in common:

- All employees have a name and a hire date
- The methods for setting and changing names and hire dates would be the same for all employees

Some employees have specialized characteristics

- Hourly employees are paid an hourly wage
- Salaried employees are paid a fixed wage

The methods for calculating wages for these two different groups would be different.

## Example: Employee class

We will now define a class called `Employee` that includes all employees.

This class can be used to define classes for hourly employees and salaried employees.

In turn, the `HourlyEmployee` class can be used to define a `PartTimeHourlyEmployee` class, and so on.

But how do we tell java that one class is to be derived from another?

The phrase `extends BaseClass` is added to the class definition.

```
class Employee {
    int employeeNumber;
}

class HourlyEmmployee extends Employee {
    double hourlyRate;
}
```

cannot inherit private property.

## Inherited members

use protected not private in variable.
public . protected. non-modifier can be inherited
but private can not be inherited.

The derived class inherits all the public methods, all the instance variables, and all the static variables from the base class.  The derived class can (and usually will) add more instance variables, static variables, and/or methods.

> We will talk about protected and package access modifiers later

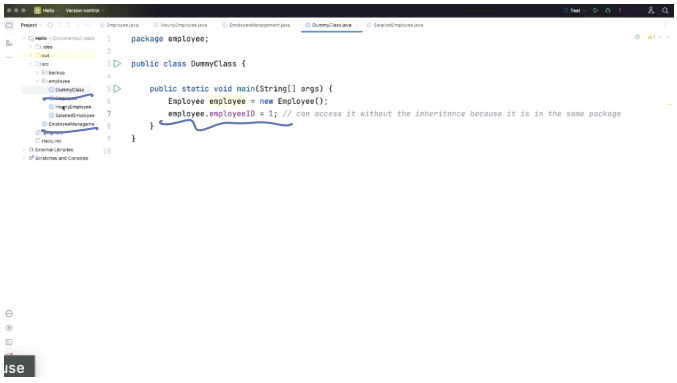
## Ancestor and descendant classes

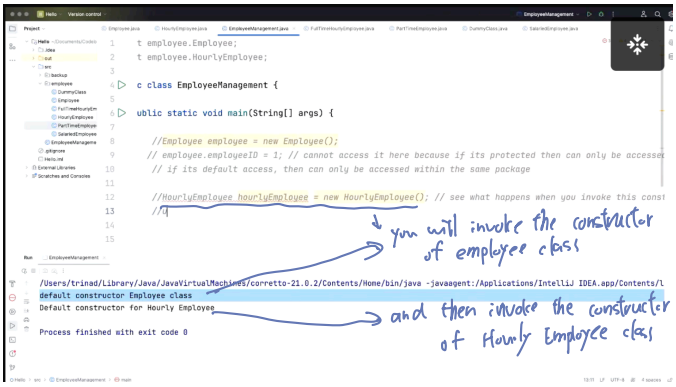
A base class is often called the ***parent*** *class*, and a derived class is then called a ***child*** *class*.

These relationships are often extended such that a class that is a parent of a parent of . . . of another class is called an ***ancestor*** *class*. If class `A` is an ancestor of class `B`, then class `B` can be called a ***descendent*** *of class* `A`.

In the same package, we can use variable, which without access modifier and don't need inheritance to do that.

But in the different package, we cannot do this.





If we use abstract method

↓

this must in the abstract class

abstract cannot be used to creat an object.

```
public class SalariedEmployee extends Employee {

    // Class SalariedEmployee must be declared abstract
    // or implement the abstract method calculateSalary
```

polymorphism.

# Overriding a method definition

One of the most powerful features of classes is the ability to override methods.

Although a derived class inherits methods from the base class, it can change or *override* an inherited method to change its behaviour.

In order to override a method definition, a new definition of the method is simply placed in the class definition, just like any other method that is added to the derived class.

(To make it clearer to the reader, it is good to put `@Override` before the new definition.)

```
class Language {
    void sayHello () {
        System.out.println("Hello, World!");
    }

    void sayHelloHello () {
        sayHello();
        sayHello();
        System.out.println();
    }

    static public void main (String[] args) {
        new Language   ().sayHelloHello();
        new Chinese().sayHelloHello();

        Language zh = new Chinese();
        zh.sayHelloHello();
    }
}

class Chinese extends Language {
    @Override
    void sayHello () {
        System.out.println("你好、世界!");
    }
}
```

Overriding a method is much more powerful than just defining a new method.  It can also change the behaviour of other methods that haven't been overridden.  This is called *polymorphism*, and will be discussed more later in this lesson.

In the above example, the function sayHelloHello called the function sayHello.  When the derived class overrode sayHello, it changed the behaviour of sayHelloHello, because the overridden form was called instead.

This allows the base class to describe a general procedure for doing things, and the derived classes to specify details that are different for different cases.

## Pitfall: Overriding vs Overloading

Do not confuse over*riding* a method in a derived class with over*loading* a method name.

When a method is overridden, the new method definition given in the derived class has the same "signature".  That is, the *exact same number and types of parameters* as in the base class.

When a method in a derived class has a different signature from the method in the base class, that is *overloading*.

Note that when the derived class over*load*s the original method, it still inherits the original method from the base class as well.

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

class Child extends Main {
    public static void main (String[] args) {
        // This overrides the main class
    }

    public static void main (int i) {
        // This overloads main
    }
}
```

## Multiple types and permissions

An object of a derived class has multiple types.

It has the type of the derived class, but it also has the type of the base class -- and all ancestor classes.

In particular, it can be assigned to a variable of any of its ancestor classes.

```
class Main {
    public static void main (String[] args) {
        Main var = new Child ();
    }
}

class Child extends Main {
}
```

That is the motivation for *covariant return types* introduced in Java 5.0.

If a method returns a primitive type (int, double, ...) or an array, then the overriding method must return the same type.

However, if the return type is a class type, then the returned type may be changed to that of any descendant class of the return type.

> **Advanced**: The reason for it to be a descendant is so that code written for the original function will still receive an object of a type it can handle.

An example of covariant return types is:

```
public class BaseClass {
    public Employee getSomeone (int someKey) {}
}

class DerivedClass extends BaseClass {
    public HourlyEmployee getSomeone(int SomeKey) {}
}
```

## Changing the access Permission of an Overridden Method

The access permission of an overridden method can be changed from, say, protected in the base class to public (or some other more permissive access) in the derived class.

However, the access permission of an overridden method must be *at least as accessible* as method in the base class.  Again, this is so that any code written for the base class can still be used with the

derived class.

Given the following method header in a base case:

```
protected void doSomething()
```

The following method header is valid in a derived class:

```
public void doSomething()
```

However, the opposite is not valid

Given the following method header in a base case:

```
public void doSomething()
```

The following method header is not valid in a derived class:

```
private void doSomething()        // illegal
```

inherited

Can not change final method.

or final class

## Preventing overriding and inheritance

If the modifier `final` is placed before the definition of a method, then that method may not be redefined in a derived class.

If the modifier `final` is placed before the definition of a class, then that class may not be used as a base class to derive other classes.

This gives rise to two errors in the following code.

```
class Main {
    final static public void main (String[] args) {
    }
}

final class Major extends Main {
    static public void main (String[] args) {
    }
}

class Minor extends Major {

}
```

C++ programmers, note that in Java, everything is `virtual` by default, and `final` acts to say "this is not `virtual`".

# Constructors

## The `super` constructor

A derived class uses a constructor from the base class to initialize all the data inherited from the base class.

In order to invoke a constructor from the base class, it uses a special syntax.  The parent's class (and in particular, the parent class's constructor) is called "`super`":

```
public derivedClass(int p1, int p2, double p3){
    super(p1, p2);        // call parent's constructor
    instanceVariable = p3;
}
```

In the above example, `super(p1, p2);` is a call to the base class constructor.

There are some rules that apply to the `super` constructor:

- A call to the base class constructor can never use the name of the base class, but uses the keyword
  `super` instead
- A call to `super` must always be the first action taken in a constructor definition
- An instance variable cannot be used as an argument to `super`.

If a derived class constructor does not include an invocation of `super`, then the no-argument constructor of the base class will automatically be invoked.  This can result in an error if the base class has not defined a no-argument constructor.

Since the inherited instance variables should be initialized, and the base class constructor is designed to do that, then an explicit call to `super` should always be used.

## Using super to access an overridden method

**Within** the definition of a method of a derived class, the base class version of an overridden method of the base class can still be invoked.  Simply preface the method name with `super` and a dot

```
public String toString(){
    return (super.toString() + "$" + wageRate);
}
```

However, using an object of the derived class **outside** of its class definition, there is **no way** to invoke the base class version of an overridden method.

If you think of `super` as being the name of the parent class, then this syntax corresponds to the syntax for calling `static` methods. Similarly, using `super` as a constructor corresponds to the fact that a constructor is normally called by the class name.

It is only valid to use super to invoke a method from a direct parent. Repeating super will not invoke a method from some other ancestor class.

For example, if the `Employee` class were derived from the class `Person`, and the `HourlyEmployee` class were derived form the class `Employee`, it would not be possible to invoke the `toString` method of the `Person` class within a method of the `HourlyEmployee` class

```
super.super.toString() // ILLEGAL!
```

As explained at [https://www.geeksforgeeks.org/accessing-grandparents-member-in-java-using-super], using super within an overriding method effectively bypasses the behaviour of the class you are defining, which is up to you. However, you shouldn't be allowed to bypass the behaviour of the class you are derived from. For example, the implementer of that class should be able to derive it from a different base class.

## The `this` constructor

Within the definition of a constructor for a class, `this` can be used as a name for invoking another constructor in the same class. The same restrictions on how to use a call to `super` apply to the `this` constructor.

If it is necessary to include a call to both `super` and `this`, the call using `this` must be made first, and then the constructor that is called must call `super` as its first action.

**Exercise**: Fix the following code

```
class Main {
    int value;
    Main () {
        value = 1;
    }

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

class Child extends Main {
    int another;
    Child (int i) {
        another = i;
    }

    Child () {
        super();
```

```
        this(1);
    }
}
```

Often, a no-argument constructor uses this to invoke an explicit-value constructor.

- No-argument constructor (invokes explicit-value constructor using this and default arguments):

```
public ClassName(){
    this(argument1, argument2);
}
```

- Explicit-value constructor (receives default values):

```
public ClassName(type1 param1, type2 param2){
. . .
}
```

For example,

```
public HourlyEmployee(){
    this("No name", new Date(), 0, 0);
}
```

will cause the following constructor to be invoked:

```
public HourlyEmployee(String theName, Date theDate, double theWageRate, double theHours)
```

not yet finished read this part

# Enhanced StringTokenizer class

Thanks to inheritance, most of the standard Java library classes can be enhanced by defining a derived class with additional methods.

For example, the StringTokenizer class enables all the tokens in a string (for example, all of the runs of non-whitespace characters) to be generated one time. However, sometimes it would be nice to be able to cycle through the tokens a second or third time.

This can be made possible by creating a derived class. For example, EnhancedStringTokenizer can inherit the useful behavior of StringTokenizer. It inherits the countTokens method unchanged.

The new behavior can be modeled by adding new methods, and/or overriding existing methods. In the following, a new method, tokensSoFar, is added, and an existing method, nextToken, is overriden.

## Task 1

Once you have read and understood the code, write a test file to parse a string, such as this sentence.

## Task 2

Write a class derived from EnhancedStringTokenizer, called WordNonWord, which can return an array of the words (strings of characters for which isLetter() is true) in the string, an array of the non-words (including things like "work," with a mix of punctuation and words).

# Access and inheritance

## Private Instance Variables cannot be accessed by name from the Base Class

An instance variable that is private in a base class is not accessible by name in the definition of a method in any other class, not even in a method definition of a derived class.

For example, an object of the HourlyEmployee class cannot access the private instance variable hireDate by name, even though it is inherited from the Employee base class.

Instead, a private instance variable of the base class can only be accessed by the public accessor and mutator methods defined in that class.

An object of the HourlyEmployee class can use the getHireDate or setHireDate methods to access hireDate

If private instance variables of a class were accessible in method definitions of a derived class, then anytime someone wanted to access a private instance variable, they would only need to create a derived class, and access it in a method of that class. This would allow private instance variables to be changed by mistake or in inappropriate ways (for example, by not using the base type's accessor and mutator methods only).

## Private Methods Are Effectively Not Inherited

The private methods of the base class are like private variables in terms of not being directly available.

However, a private method is completely unavailable, unless invoked indirectly. This is possible only if an object of a derived class invokes a public method of the base class that happens to invoke the private
method

This should not be a problem because private methods should just be used as helping methods. If a method is not just a helping method, then it should be public (or package access), not private.

## Protected and Package access

If a method or instance variable is modified by `protected` (rather than `public` or `private`), then it can be accessed by name

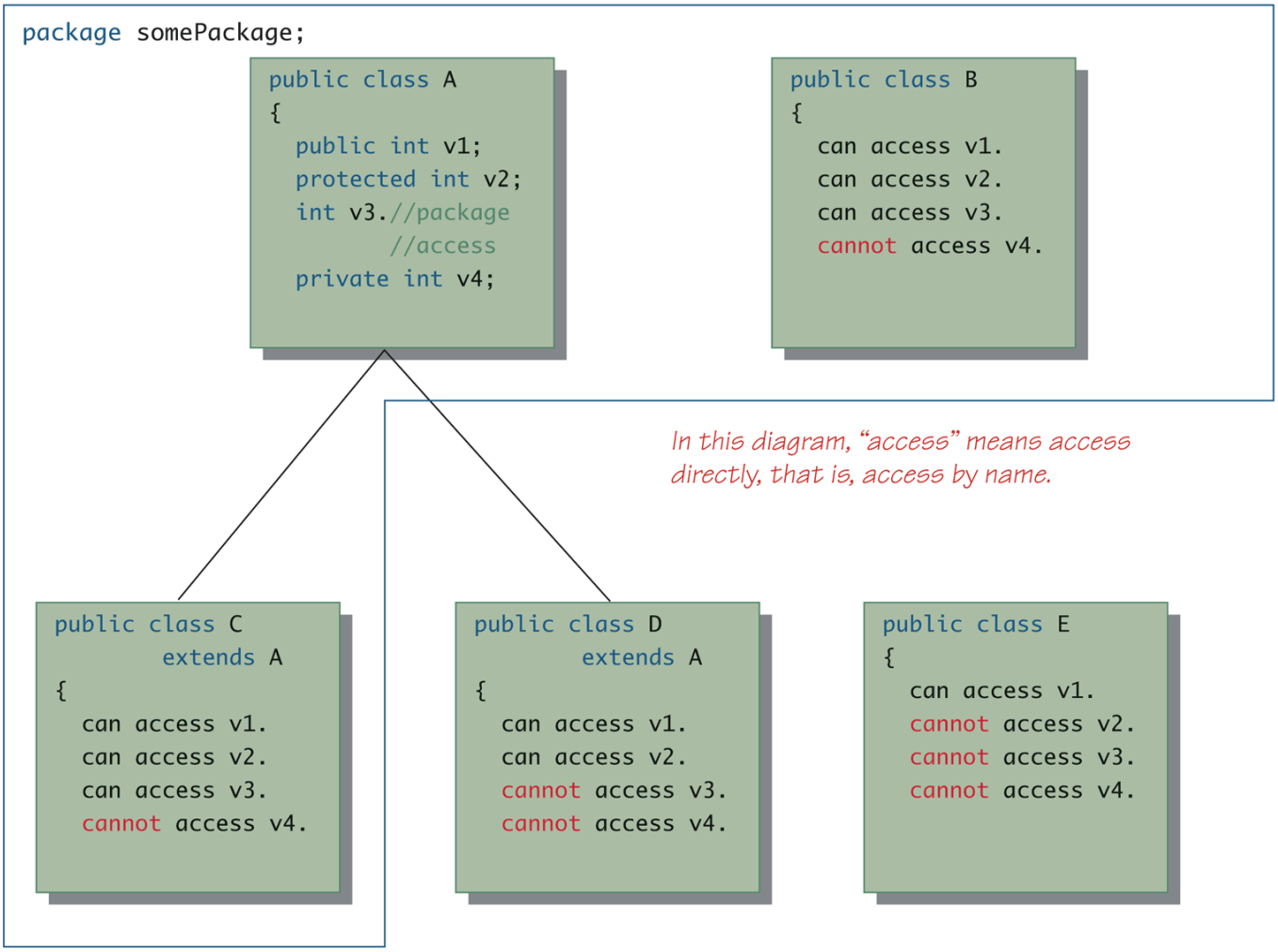
- Inside its own class definition
- Inside any class derived from it
- In the definition of any class in the same package

The `protected` modifier provides very *weak* protection compared to the `private` modifier. It allows direct access to any programmer who defines a suitable derived class. Therefore, instance variables should normally not be marked protected.

An instance variable or method definition that is not preceded with a modifier has *package* access. Package access is also known as default or friendly access.

Instance variables or methods having package access can be accessed by name inside the definition of any class in the same package. However, neither can be accessed outside the package

Note that package access is more restricted than `protected`. Package access gives more control to the programmer defining the classes. Whoever controls the package directory (or folder) controls the package access.


package somePackage;
public class A
{
public int v1;
protected int v2;
int v3.//package
//access
private int v4;
public class B
{
can access v1.
can access v2.
can access v3.
cannot access v4.
In this diagram, "access" means access directly, that is, access by name.
public class C
extends A
{
can access v1.
can access v2.
can access v3.
cannot access v4.
public class D
extends A
{
can access v1.
can access v2.
cannot access v3.
cannot access v4.
public class E
{
can access v1.
cannot access v2.
cannot access v3.
cannot access v4.
A line from one class to another means the lower class is a derived class of the higher class.
If the instance variables are replaced by methods, the same access rules apply.

# Pitfalls and Tips

## Forgetting the default package

When considering package access, do not forget the default package.  All classes in the current directory (not belonging to some other package) belong to an unnamed package called the *default package*.

If a class in the current directory is not in any other package, then it is in the default package.  If an instance variable or method has package access, it can be accessed by name in the definition of any other class in the default package.

## A restriction on protected access

If a class B is derived from class A, and class A has a protected instance variable n, but the classes A and B are in different packages, then the following is true:

- A method in class B can access n by name (n is inherited from class A)
- A method in class B can create a local object of itself, which can access n by name (again, n is inherited from class A)

However, if a method in class B creates an object of class A, it can not access n by name.  A class knows about its own inherited variables and methods. However, it cannot directly access any instance variable or method of an ancestor class *unless they are public*.  Therefore, B can access n whenever it is used as an instance variable of B, but B cannot access n when it is used as an instance variable of A.

This is true if A and B are *not* in the same package. If they were in the same package there would be no problem, because `protected` access implies package access

## Tip: Static variables are inherited

Static variables in a base class are inherited by any of its derived classes.

The modifiers `public`, `private`, and `protected`, and package access have the same meaning for static variables as they do for instance variables

## "Is a" versus "Has a"

There are many ways in which objects can be related.  Two of these are called the "is a" hierarchy and the "has a" hierarchy.

A derived class demonstrates an "*is a*" relationship between it and its base class

- Forming an "is a" relationship is one way to make a more complex class out of a simpler class
- For example, an `HourlyEmployee` "*is an*" `Employee`
- `HourlyEmployee` is a more complex class compared to the more general `Employee` class

Another way to make a more complex class out of a simpler class is through a "has a" relationship
– This type of relationship, called *composition*, occurs when a class *contains* an instance variable of a class
type
– The `Employee` class contains an instance variable, `hireDate`, of the class `Date`, so therefore, an `Employee` "*has a*" `Date`

Both kinds of relationships are commonly used to create complex classes, often within the same class.  Since `HourlyEmployee` is a derived class of `Employee`, and contains an instance variable of class Date, then `HourlyEmployee` "is an" `Employee` and "has a" `Date`.

**Exercise**:  Create a class hierarchy such that a `Person` "has a" `WirelessDevice`, a `Phone` "is a" `WirelessDevice`, and a `Notebook` "is a" `WirelessDevice`.  Create a person who has a phone and a person who has a notebook.

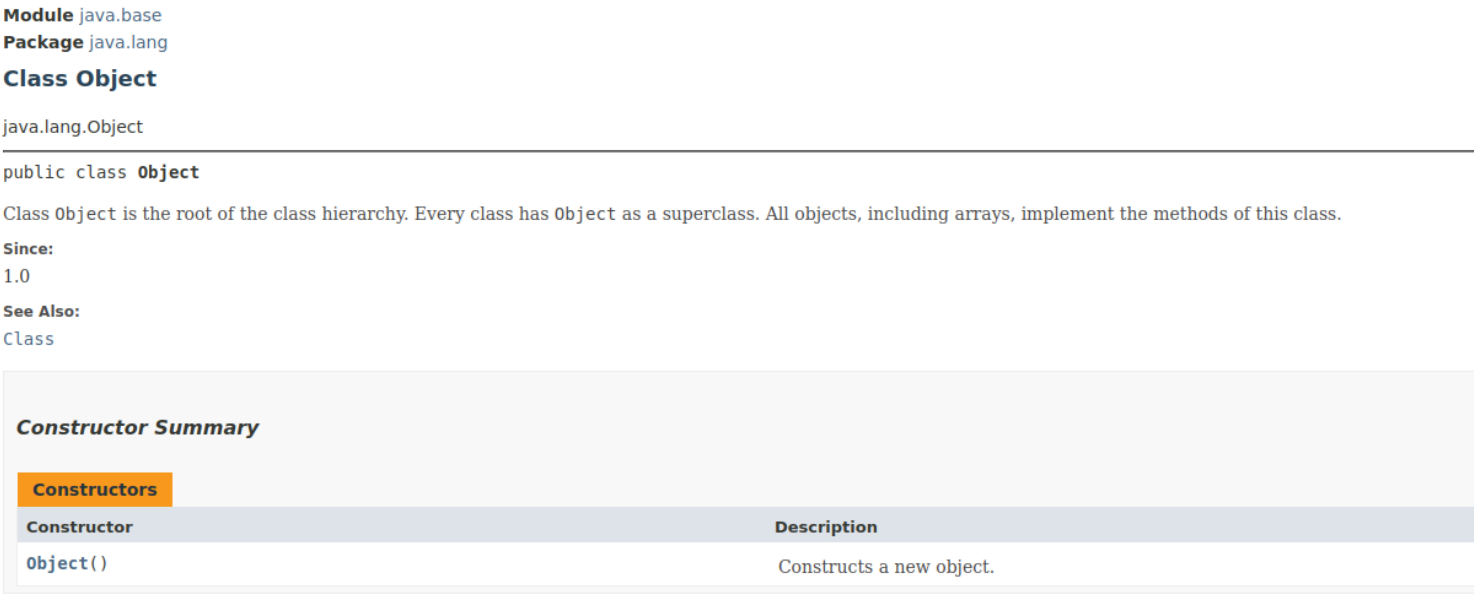
```
class Main {
    public static void main (String[] args) {
        Person p = new Person();
    }
}

class Person {
}

class WirelessDevice {
}

class Phone {
}

class Notebook {

}
```

# class Object

In Java, every class is a descendent of the class `Object`.

- Every class has `Object` as its ancestor
- Every object of every class is of type `Object`, as well as being of the type of its own class

If a class is defined that is not explicitly a derived class of another class, it is still automatically a derived class of the class `Object`.



**Module** java.base
**Package** java.lang

**Class Object**

java.lang.Object

---

`public class Object`

Class `Object` is the root of the class hierarchy. Every class has `Object` as a superclass. All objects, including arrays, implement the methods of this class.

**Since:**
1.0

**See Also:**
`Class`

***Constructor Summary***

**Constructors**

| Constructor | Description |
|---|---|
| `Object()` | Constructs a new object. |

The class `Object` is in the package `java.lang` which is always imported automatically.

Having an `Object` class enables methods to be written with a parameter of type `Object`.
A parameter of type `Object` can be replaced by an object of any class whatsoever.
For example, some library methods accept an argument of type `Object` so they can be used with an argument that is an object of any class.

Go back to the methods of ArrayList and work out how Object is used there.

The class `Object` has some methods that every Java class inherits. For example, the `equals` and `toString` methods.

Every object inherits these methods from some ancestor class: either the class `Object` itself, or a class that itself inherited these methods (ultimately) from the class `Object`.

However, these inherited methods should be overridden with definitions more appropriate to a given class. Some Java library classes assume that every class has its own version of such methods.

## instanceof, getClass() and equals()

Every object inherits the same `getClass()` method from the `Object` class. This method is marked `final`, so it cannot be overridden.

An invocation of `getClass()` on an object returns a representation only of the class that was used with new to create the object. The results of any two such invocations can be compared with `==` or `!=` to determine whether or not they represent the exact same class

```
(object1.getClass() == object2.getClass())
```

The `instanceof` operator checks if an object is of the type given as its second argument

```
object instanceof ClassName
```

This will return `true` if `object` is of type `ClassName`, and otherwise return `false`.

Note that this means it will return `true` if `object` is the type of any descendant class of `ClassName`.

Although these are similar, they have quite different uses.

The most common use of `instanceof` is to find out whether a particular method can be used on an object. That is because a method will often take an argument that is a base class, and it doesn't know which descendant of that class the actual object is.

```
class Descendant {
    public static void main (String[] args) {
        Person a = new Employed ("John", "computer scientist");
        Person b = new Musician ("Jane", "flute");

        a.print ();
        b.print ();
    }
}

class Person {
    String name;

    Person (String n) {
        name = n;
    }

    void print () {
        System.out.print (name);
        if (this instanceof Employed) {
            // the cast (Employed) means
            // "treat  this  as type Employed, not just Person"
```

```
            // That allows us to use methods implemented by Employed,
            // but not by all objects of type Person.
            Employed e = (Employed)this;
            System.out.print(", job: " + e.getJob());
        }
        if (this instanceof Musician) {
            Musician m = (Musician)this;
            System.out.print(", instrument: " + m.getInstrument());
        }
        System.out.println("");
    }
}

class Employed extends Person {
    String job;
    Employed (String n, String j) {
        super (n);
        job = j;
    }

    String getJob () {
        return job;
    }
}

class Musician extends Person {
    String instrument;
    Musician (String n, String inst) {
        super (n);
        instrument = inst;
    }

    String getInstrument () {
        return instrument;
    }
}
```

We will soon see how to use "interfaces" to allow a person to be both a musician and employed (despite all the jokes about musicians never being employed...).

**Exercise**: What happens if you drop the "`if (this instanceof ...)`"?

**Exercise**: Derive a subclass `ExperiencedMusician` from `Musician` that has an instance variable representing the number of years the person has been playing the instrument. Override `getInstrument` to include the number of years. That makes it not just an accessor function but something that computes a new string. Check that objects of type `ExperiencedMusician` are also printed correctly -- with the instrument and number of years.

In contrast `getClass()` is used when you want to check the exact class of the object, not just that it is in a particular branch of the hierarchy.

## The right way to define `equals`

In most cases, `instanceof` is a better choice than `getClass()`, but there is one big exception: *testing for equality*.

The overridden version of equals must meet the following conditions

- The parameter otherObject of type Object must be type cast to the given class (e.g., Employee)
- However, the new method should only do this if otherObject really is an object of that class, and if otherObject is not equal to null
- Finally, it should compare each of the instance variables of both objects

```
public boolean equals(Object otherObject)
{
    if(otherObject == null)
        return false;
    else if(getClass( ) != otherObject.getClass( ))
        return false;
    else {
        Employee otherEmployee = (Employee)otherObject;
        return (name.equals(otherEmployee.name) &&
            hireDate.equals(otherEmployee.hireDate));
    }
}
```

What would this look like using `instanceof`? For class Employee, we would have

```
. . . //excerpt from bad equals method
else if(!(OtherObject instanceof Employee))
    return false; . . .
```

and from class HourlyEmployee, we would have

```
. . . //excerpt from bad equals method
else if(!(OtherObject instanceof HourlyEmployee))
return false; . . .
```

Now consider the following:

```
Employee e = new Employee("Joe", new Date());
HourlyEmployee h = new HourlyEmployee("Joe", new Date(),8.5, 40);
boolean testH = e.equals(h);
boolean testE = h.equals(e);
```

- testH will be true, because h is an Employee with the same name and hire date as e
- testE will be false, because e is not an HourlyEmployee, and cannot be compared to h

Note that this problem would not occur if the getClass() method were used instead, as in the previous

equals method example

## Employee Example Code Snippet

Here is the complete code that shows inheritance-based relationship between classes. We have covered these examples in the live lecture.  Please run the code samples, copy it change it and play with it. Some comments have been added for your understanding.

```
class EmployeeSim{

    public static void main(String[] args){

        //creating objects for inheritance based classes
        //syntax : BaseClass variableName = new DerviedClassConstructor();
        Employee hEmp = new HourlyEmployee("Trina", 1, "2022-01-03", 40.5f, 10);
        Employee sEmp = new SalariedEmployee("John", 2, "2022-01-03", 5000f);

        System.out.println(hEmp.printName()); // finds the method in base class
        System.out.println(hEmp.printAnotherName()); // finds the method in base class
        System.out.println(hEmp.printMyDetails());
        System.out.println(hEmp.printMySalary());

        System.out.println(sEmp.printName());
        System.out.println(sEmp.printAnotherName()); // LateBinding: finds the overridden method in
        System.out.println(sEmp.printMyDetails());
        System.out.println(sEmp.printMySalary());

        //Employee emp = new Employee(); This is forbidden. Cannot create objects of abstract class

    }
}

//class is abstract because it has atleast one abstract method
abstract class Employee{

    String name;  // no access modifier means package access
    protected int employeeNumber;   // protected modifier also means derived classes and package ac
    protected String hireDate;

    public Employee(){}

    public Employee(String name, int empNumber, String hireDate){
        this.name = name;
        this.employeeNumber = empNumber;
        this.hireDate = hireDate;
    }

    // final in a method signature means this can be overridden
    protected final String printName(){
        return "Hello my name is " + this.name;
    }
```

```
    protected String printAnotherName(){
        return "Hello my name is " + this.name;
    }

    protected String printMyDetails(){
        return "Hello my name is " + this.name + ". I was hired on " + this.hireDate + ". My employ
         + ".";
    }

    //abstract method is not implemented but the definition is mentioned
     public abstract String printMySalary();
}


class HourlyEmployee extends Employee {

    private float hourlyRate;
    private int numberOfHoursWorked;

    public HourlyEmployee(String n, int en, String hd, float hr, int hours){
        super(n,en,hd);
        this.hourlyRate = hr;
        this.numberOfHoursWorked = hours;
    }

    public String printMyDetails(){
        return  super.printMyDetails() +  "I am a hourly employee. This is my name "+ super.name;
    }

    public String printMySalary(){
        return "I earned AUD " + this.hourlyRate * this.numberOfHoursWorked;
    }

}


class SalariedEmployee extends Employee{
    private float monthlySalary;

    public SalariedEmployee(String n, int en, String hd, float hr){
        super(n,en,hd); //super here calls the constructor of base file
        this.monthlySalary = hr;
    }

    // override allows you to override the method from the parent class
    @Override
    public String printAnotherName(){
        return "this is an overriden method.";
    }


    public String printMyDetails(){
        //accessing method from base/parent class using super.methodName
```

```
        return  super.printMyDetails() + "I am a salaried employee.";
    }

    public String printMySalary(){
        return "I earned AUD " + this.monthlySalary;
    }

}
```