**Inheritance**

* What inheritance is
* How to inherit a class from another class
* The difference between early binding and late binding
* What method overriding is and how to override methods
* What field hiding and method hiding are and how to use them in your code
* What abstract classes are and where to use them
* How to declare final classes and methods
* The difference between “is-a,” “has-a,” and “part-of” relationships

**What is Inheritance?**

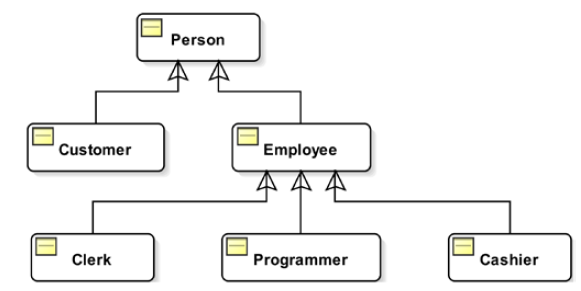
* It lets you use abstraction in a new way
* inheritance lets you define a new abstraction by extending an existing abstraction…

supertype, a superclass, parent class, or a base class,

subtype, a subclass, a child class, or a derived class

* supertype to define another subtype and so on…
* code reuse mechanism…

Inheritance allows you to use varying degrees of abstraction at different levels of hierarchy ..?

****

Substitutivity…?

can always be substituted with its subtype..!...?

**-** supertype and its subtype represent an “is-a” relationship

Emoloyee emp = new Programmer();

Emoloyee emp = new Clerk();

Is a Programmer an employee?.. Yes.

**-** subtype may extend the functionality of its supertype by adding more functionality…

Why do we need Inheritance?

* avoids copying the same code at multiple places, thus facilitating code reuse.. How ?

General Syntax

<<class modifiers>>class <<SubclassName>> extends <<SuperclassName>> {

// Code for the Subclass goes here

}

Example:

// #1 – Use the simple name of P in the extends clause and use an import statement.

package pkg2;

import pkg1.P;

public class Q extends P {

// Code for class Q goes here

}

// #2 – Use the fully qualified name of P. No need to use an import statement.

package pkg2;

public class Q extends pkg1.P {

// Code for class Q goes here

}

Java.lang.Object Class is the Default Superclass…

// #1 – "extends Object" is implicitly added for class P

public class P {

// Code for class P goes here

}

// #2 – "extends Object" is explicitly added for class P

public class P extends Object {

// Code for class P goes here

}

As Employee class is implicitly a subclass of the Object class, it can use these methods…

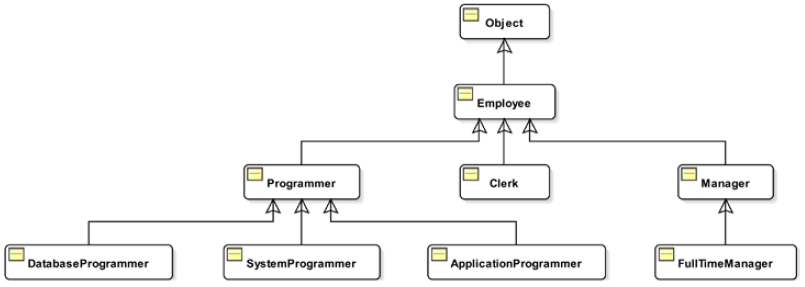
Employee emp = new Employee();

int hc = emp.hashCode();

String str = emp.toString();

**Inheritance hierarchy**

*A subclass can have its own subclasses, which in turn can have their own subclasses….*



*Java allows single inheritance for a class…*

What Is Inherited by a Subclass?

*“*a subclass *inheriting* something from its superclass” &  
 “a subclass *using* something from its superclass.” … ?

* A subclass inherits non-private members of its superclass
* constructors and initializers are not inherited..

Examaple:

class A extends java.lang.Object {   
//private member m1; // *declared member*  
//non-private member m2; //*declared member*

// members of class A are m1, m2, and all **inherited members** from the Object class  
}

class B extends A {   
// private member m3; // *declared member*  
// non-private member m4; // *declared member*  
 // all non-private members of class A.

// member m1 is declared private in class A, so it is not inherited by class B  
}

Note: non-private members of class A trickle down to class B, which in

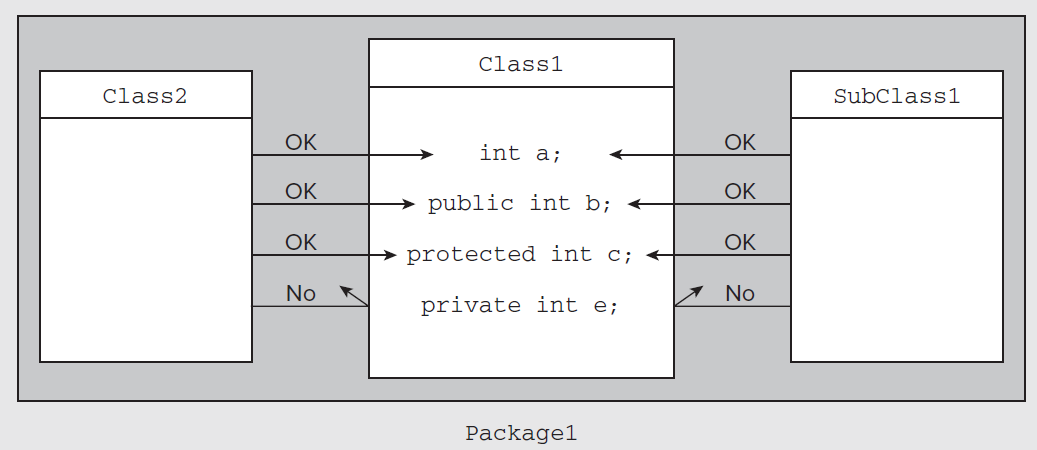
turn trickle down to class C…

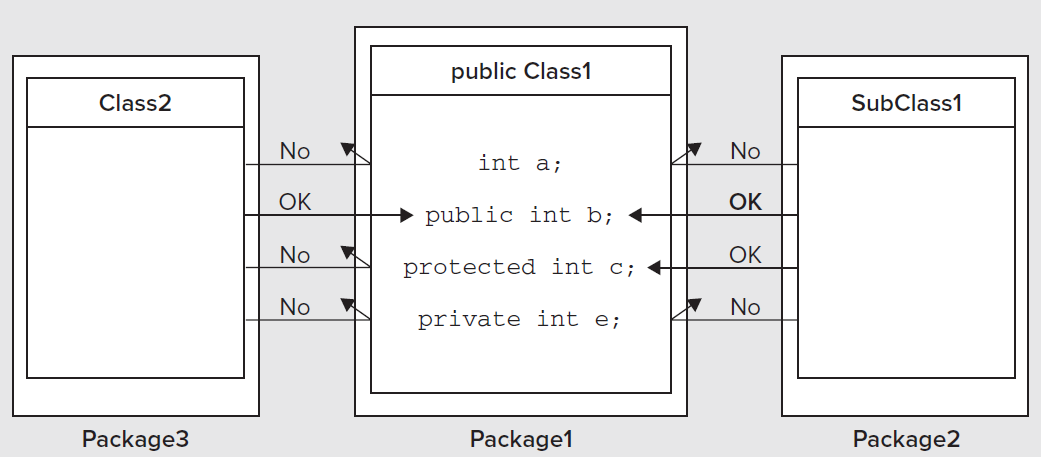
private, public, protected, and package-level.

Access level modifier of a class member determines two things:

* Who can access (or use) that class member directly
* Whether a subclass inherits that class member or not

Access modifiers are also used with non-members (e.g. constructors) of class ..!





is-a” relationship using inheritance,

* Whatever behaviour a client code expects to be present in a class will also be present in the class’s subclass

if client code works with a class, it will also work with the class’s subclass

Example:

Compiler, finds that the declared type of the emp variable is Employee… & it makes sure that the Employee class has setName() and getName() methods…

Employee emp;

emp = new Employee();

emp.setName("Richard Castillo");

String name = emp.getName();

A subclass guarantees at least the same behavior (methods) as its superclass…

Employee emp;

emp = new Manager(); // A Manager object assigned to an Employee variable

emp.setName("Richard Castillo");

compile-time type of “new Manager()” expression is the Manager type

String name = emp.getName();

compile-time type (or declared type) of the emp variable is Employee type

Note: an object of the Manager class “is-a” object of the Employee class (upcasting)

Examples of Upcasting: (a subclass “is-a” object of the superclass, too)

Object obj;

Employee emp;

Manager mgr;

PartTimeManager ptm;

// An employee is always an object

obj = emp;

// A manager is always an employee

emp = mgr;

// A part-time manager is always a manager

mgr = ptm;

// A part-time manager is always an employee

emp = ptm;

// A part-time manager is always an object

obj = ptm;

Upcasting is a very powerful feature of inheritance

* Upcasting lets you write polymorphic code..
* lets you code in terms of a superclass… that will always work with all subclasses…
* It lets you write generic code without worrying about a specific type..

Object obj = new AnyJavaClass(); // Upcasting

**instanceof Operator**

* Determines that whether a reference variable has a reference to an object of a class or a subclass of the class at runtime.

<<Class Reference Variable>>instanceof <<Class Name or Interface>>

Example:

*Use the instanceof operator before downcasting to check….*

Manager mgr = new Manager();

Employee emp = mgr;

if (emp instanceof Manager) {

// The following downcast will always succeed

mgr = (Manager)emp;

}

else {

// emp is not a Manager type

}

Manager mgr = null;

if (mgr instanceof Clerk) { // A compile-time error

}

Note: mgr can hold a reference of Manager type or its descendant type….. Clerk type is not in the same inheritance-chain…

**The equals() method of java.lang.Object:**

* It returns true if the argument and the object on which this method is called are considered equal. Otherwise, it returns false.

**Early Binding**

* the decision about which method code and field will be accessed is made by the compiler at compile time…
* decides which method from which class will be executed
* compiler decides which field from which class will be accessed

Early binding is used for the following types of methods and fields.

All types of fields: static and non-static

* Static methods
* Non-static final methods

Note: *method or a field is accessed based on the declared type (or compile-time type) of the variable*

**Late Binding**

* All non-static, non-final methods follows the rules of late binding.
* The decision as to which version of the method is called is made at runtime

*version of the method that will be called depends on the runtime type of the object …….. not on its compile-time type*

late binding.

JVM decides which setName(String s) method should be called

JVM continues its search for a method definition up to the Object class.

*ex:* Employee emp = new Manager();

emp.setName("John Jacobs");

compiler detects that the setName(String s) method in the Employee class is an instance method, which is not final.

* Note: - For an instance method call, the compiler does not perform binding. It will leave this work for runtime

**Method Overriding**

* Redefining an instance method in a class, which is inherited from the superclass,

*Ex:*

* *It is like class B telling class A, “Thanks for being my superclass and letting me inherit your print() method.*
* I am going to redefine it my way, without affecting your print() method in any way.
* You can keep using your print() method.

public class A {

print() method in class B overrides the print() method of class A

public void print() {

System.out.println("A");

}

}

public class B extends A {

public void print() {

System.out.println("B"); }

}

What method does class C inherit: A.print() or B.print(), or both?

It inherits the print() method from class B.

public class C extends B {

// Inherits B.print()

}

public class D extends C {

// Inherits B.print() through C

}

Class E overrides the print() method of class B, which it inherited from class D

public class E extends D {

public void print() {

System.out.println("E");

}

}

public class F extends E {

// Inherits E.print() through E

}

Let’s consider the following definitions of two classes S and T:

public class S {

public void print() {

Does the print() method in class T override the print() method in its superclass S? The answer is no.

Because..

This is called method overloading.

System.out.println("S");

}

}

public class T extends S {

public void print(String msg) {

System.out.println(msg);

}

}

Class T will now have two print() methods….!... ?

Method Overriding Rules..?

Method Overriding Rule #1

* method must be an instance method.

Method Overriding Rule #2

* overriding method must have the same name as the overridden method.

Method Overriding Rule #3

* overriding method must have the same number of parameters of the same type in the same order as the overridden method.

Method Overriding Rule #4

* the return type of the overriding and the overridden methods must be the same (Java 5, has changed for return types of reference data types.)

example:

public class P {

public Employee getEmp() {

getEmp() method of class R also overrides the getEmp() method of class P even though its return type Manager is different from the return type of the overridden method, which is Employee.

// Code goes here

}

}

public class Q extends P {

public Employee getEmp() {

// code goes here

}

}

public class R extends P {

public Manager getEmp() {

// code goes here

}

}

Method Overriding Rule #5

* access level of the overriding method must be at least the same or more relaxed than that of the overridden method

Method Overriding Rule #6

* overriding method cannot add a new exception to the list of exceptions in the overridden method.

Example:

public class G {

public void m1() throws CheckedException1, CheckedException2 {

// Code goes here

}

}

public class H extends G {

public void m1() throws CheckedException1, CheckedException2, CheckedException3 {

// Code goes here

} }

**Accessing Overridden Method**

* to access the overridden method from a subclass. A subclass can use the keyword super as a qualifier to call the overridden method of the superclass.

**Method Overloading**

* Having more than one method with the same name in the same class.
* Overloaded methods must have different number of parameters, different types of parameters, or both.
* return type, access level and throws clause of a method play no role in making it an overloaded method

Example:

public class OME1 {

public void m1(int a) {

// Code goes here

The m1() method of the OME1 class is an example of an overloaded method.

}

public void m1(int a, int b) {

// Code goes here

}

public int m1(String a) {

// Code goes here

}

public int m1(String a, int b) throws CheckedException1 {

// Code goes here

}

}

// Won't compile

public class OME2 {

OME2 class would not compile because it has a duplicate declaration for the m2() method.

public void m2(int p1) {

// Code goes here

}

public void m2(int p2) {

// Code goes here

}

}

The order of the parameters may play a role in making a method overloaded…

Example:

public class OME3 {

Inside OME3 class, both methods have one parameter of type int and another of type double. However, they are in a different order.

public void m3(int p1, double p2) {

// Code goes here

}

public void m3(double p1, int p2) {

// Code goes here

} }

**Note:** Method overloading , is another kind of polymorphism where the same method name has different meanings.

* overloading is bound at compile time as opposed to method overriding that is bound at runtime.

**Note:** For an overloaded method call, the compiler chooses the most specific method. If it does not find an exact match, it will try to look for a more generic version by converting the actual parameter type to a more generic type using the rules of automatic type widening.

**Inheritance and Constructors**

An Object: state & behaviour

* Each object of a class maintains its own state.

Note: memory is allocated for all instance variables declared in the class and all instance variables declared in its ancestors at all levels

Example:

public class U {

private int id;

protected String name;

}

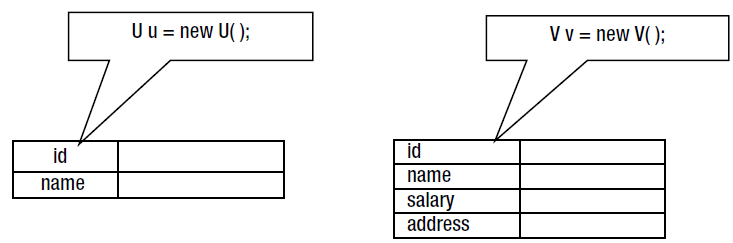
public class V extends U {

protected double salary;

protected String address;

}

*Memory allocation for an object includes all instance variable of the class and all its ancestors*



Let’s get to the Constructors…

* Constructors are not inherited by subclasses…

*Note: To initialize the instance variables of ancestor classes, the constructors of ancestor classes must be called*

How..! … ?

compiler injects the call to the immediate ancestor’s no-args constructor as the first statement in every constructor you add..

Note: super is used in many contexts. It also refers to the immediate ancestor of a class

examples of calling constructor of a superclass:

// Call no-args constructor of superclass

super();

// Call superclass constructor with a String argument

super("Hello");

// Call superclass constructor with two double arguments

super(10.5, 89.2);

A few more rules about using the constructors of the superclass from the subclass.

// X.java

package com.jdojo.inheritance.pkg1;

public class X {

// X() has package-level access

X() {

}

}

generates a compiler error as follows: Error(7): X() is not public in com.jdojo.inheritance.pkg1.X; cannot be accessed from outside Package..!...why?

Because, no-args constructor of class X has a package-level access

// Y.java

package com.jdojo.inheritance.pkg2;

import com.jdojo.inheritance.pkg1.X;

public class Y extends X {

public Y() {

}

}

How do you fix class Y?

* *create a constructor for the class X, which has a public or protected access, so it can be accessed from class Y.*

Sometimes, the access level of the constructors of a class cannot be accessed at all.

public class NoSubclassingAllowed {

A private constructor cannot be accessed from anywhere including subclasses

This concludes that the NoSubclassingAllowed class cannot be inherited by any other classes

private NoSubclassingAllowed() {

}

// Other code goes here

}

// Won't compile.

public class LetUsTryInVain extends NoSubclassingAllowed {

}

**Field Hiding**

* field declaration (static or non-static) in a class hides the inherited field with the same name in its superclass
* Field hiding occurs solely based on the field name

Example:

public class G {

protected int x = 200;

The field declarations x, y, and z in class H hide the inherited fields x, y, and z in class G

Simple names of fields x, y, and z in class H refer to the hiding fields, not inherited fields

You use the simple name x in class H, it refers to the field x declared in class H, not in class G

protected String y = "Hello";

protected double z = 10.5;

}

public class H extends G {

protected int x = 400; // Hides x in class G

protected String y = "Bye"; // Hides y in class G

protected String z = "OK"; // Hides z in class G

}

If you want to refer to the field x in class G from inside class H,…

……………………….you need to use the keyword super, for example, super.x.

**public** **void** print() {

/\*

\* FHidingSub2 class has four fields, two inherited (num and name) and two declared (num and name).

\*/

System.***out***.println("num: " + num);

System.***out***.println("name: " + name);

/\* If you want to refer to the inherited fields from the superclass, you need to

qualify the field names with the keyword super. F

\*/

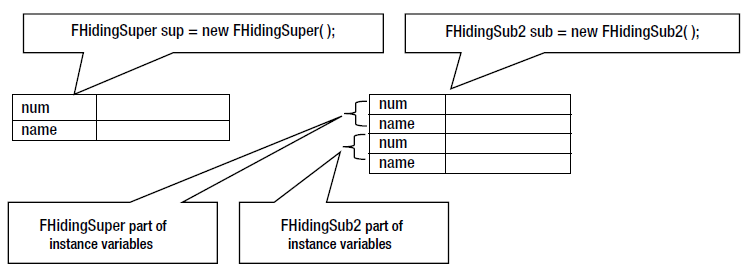
System.***out***.println("Values from the inherited properties");

System.***out***.println("super.num: " + **super**.num);

System.***out***.println("super.name: " + **super**.name);

}

*Recall that when an object is created, the Java runtime allocates memory for all instance variables in the class of the object and all of its ancestors*



**Disabling Inheritance**

* You can disable subclassing for a class by declaring the class final.
* A final class cannot be subclassed.

Example:

public final class Security {

// Code goes here

}

// Won't compile. Cannot inherit from Security

public final class CrackedSecurity extends Security {

// Code goes here

}

You can also declare a method as final.

* A final method cannot be overridden or hidden by a subclass.

Example:

public class A {

public final void m1() {

// Code goes here

}

public void m2() {

// Code goes here

}

}

public class B extends A {

// Cannot override A.m1() here because it is final in class A

// OK to override m2() because it is not final in class A

public void m2() {

// Code goes here

} }

Why would you declare a class or a method as final?

OR

Why would you want to prevent subclassing of a class or overriding/hiding of a method?

* reasons for doing this are security, correctness, and performance.

**Abstract Classes and Methods**

* create a class just to represent a concept rather than to represent object
* Java lets you create a class whose objects cannot be created (abstract class.)

just to represent an idea, which is common to objects of other classes

ex: public abstract class Shape {

to represent a concept rather than to represent objects

// No code for now

}

you cannot create its object..

Shape s; // OK

new Shape(); // An error. Cannot create a Shape object

*Note: By declaring a class abstract, you indicate that the class has some incomplete method definitions (behaviors) for its objects…*

*// Your Shape class does not know how to draw a shape until you mention a specific shape… but one thing is sure that you should be able to draw a shape no matter what kind of shape it is.*

public abstract class Shape {

public Shape() {

}

public abstract void draw(); // is a good candidate to be declared as an abstract method

}

Example:

An Abstract Shape Class with One Instance Variable, Two Constructors, and One Abstract Method

// Shape.java

package com.jdojo.inheritance;

public abstract class Shape {

private String name;

public Shape() {

this.name = "Unknown shape";

}

public Shape(String name) {

this.name = name;

}

public String getName() {

return this.name;

}

public void setName(String name) {

this.name = name;

}

// Abstract methods

public abstract void draw();

public abstract double getArea();

public abstract double getPerimeter();

}