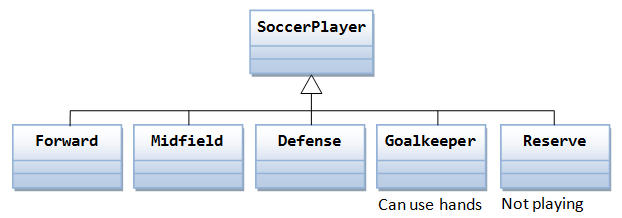
# **Advanced Programming (Java)**

# **Inheritance**

# **Week 5 – Reading, Daily Work and Practice Material**

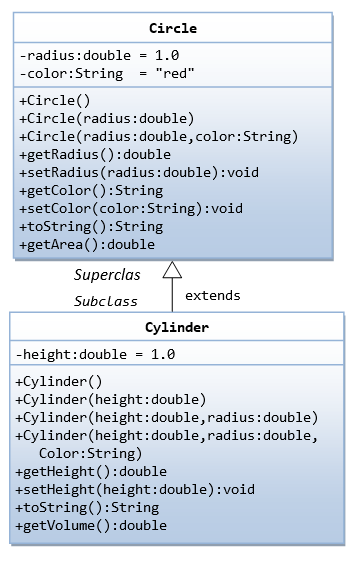
### Inheritance

In OOP, we often organize classes in hierarchy to avoid duplication and reduce redundancy. The classes in the lower hierarchy inherit all the variables (static attributes) and methods (dynamic behaviors) from the higher hierarchies. A class in the lower hierarchy is called a subclass (or derived, child, extended class). A class in the upper hierarchy is called a superclass (or base, parent class). By pulling out all the common variables and methods into the superclasses, and leave the specialized variables and methods in the subclasses, redundancy can be greatly reduced or eliminated as these common variables and methods do not need to be repeated in all the subclasses. For example,



A subclass inherits all the variables and methods from its superclasses, including its immediate parent as well as all the ancestors. It is important to note that a subclass is not a "subset" of a superclass. In contrast, subclass is a "superset" of a superclass. It is because a subclass inherits all the variables and methods of the superclass; in addition, it extends the superclass by providing more variables and methods. In Java, you define a subclass using the keyword "extends", e.g.,

#### 1: The Circle and Cylinder Classes



In this example, we derive a subclass called Cylinder from the superclass Circle, which we have created in the previous chapter. It is important to note that we reuse the class Circle. Reusability is one of the most important properties of OOP. (Why reinvent the wheels?) The class Cylinder inherits all the member variables (radius and color) and methods (getRadius(), getArea(), among others) from its superclass Circle. It further defines a variable called height, two public methods - getHeight() and getVolume() and its own constructors, as shown:

##### Circle.java (Re-produced)

public class **Circle** {

// private instance variables

private double radius;

private String color;

// Constructors

public Circle() {

this.radius = 1.0;

this.color = "red";

}

public Circle(double radius) {

this.radius = radius;

this.color = "red";

}

public Circle(double radius, String color) {

this.radius = radius;

this.color = color;

}

// Getters and Setters

public double getRadius() {

return this.radius;

}

public String getColor() {

return this.color;

}

public void setRadius(double radius) {

this.radius = radius;

}

public void setColor(String color) {

this.color = color;

}

// Describle itself

public String toString() {

return "Circle[radius=" + radius + ",color=" + color + "]";

}

// Return the area of this Circle

public double getArea() {

return radius \* radius \* Math.PI;

}

}

##### Cylinder.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43 | /\*  \* A Cylinder is a Circle plus a height.  \*/  public class **Cylinder extends Circle** {  // private instance variable  private double height;    // Constructors  public Cylinder() {  super(); // invoke superclass' constructor Circle()  this.height = 1.0;  }  public Cylinder(double height) {  super(); // invoke superclass' constructor Circle()  this.height = height;  }  public Cylinder(double height, double radius) {  super(radius); // invoke superclass' constructor Circle(radius)  this.height = height;  }  public Cylinder(double height, double radius, String color) {  super(radius, color); // invoke superclass' constructor Circle(radius, color)  this.height = height;  }    // Getter and Setter  public double getHeight() {  return this.height;  }  public void setHeight(double height) {  this.height = height;  }  // Return the volume of this Cylinder  public double getVolume() {  return getArea()\*height; // Use Circle's getArea()  }  // Describle itself  public String toString() {  return "This is a Cylinder"; // to be refined later  }  } |

##### A Test Drive for the Cylinder Class (TestCylinder.java)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | /\*  \* A test driver for the Cylinder class.  \*/  public class **TestCylinder** {  public static void main(String[] args) {  Cylinder cy1 = new Cylinder();  System.out.println("Radius is " + cy1.getRadius()  + " Height is " + cy1.getHeight()  + " Color is " + cy1.getColor()  + " Base area is " + cy1.getArea()  + " Volume is " + cy1.getVolume());    Cylinder cy2 = new Cylinder(5.0, 2.0);  System.out.println("Radius is " + cy2.getRadius()  + " Height is " + cy2.getHeight()  + " Color is " + cy2.getColor()  + " Base area is " + cy2.getArea()  + " Volume is " + cy2.getVolume());  }  } |

Keep the "Cylinder.java" and "TestCylinder.java" in the same directory as "Circle.class" (because we are reusing the class Circle). Compile and run the program. The expected output is as follows:

Radius is 1.0 Height is 1.0 Color is red Base area is 3.141592653589793 Volume is 3.141592653589793

Radius is 5.0 Height is 2.0 Color is red Base area is 78.53981633974483 Volume is 157.07963267948966

#### Method Overriding & Variable Hiding

A subclass inherits all the member variables and methods from its superclasses (the immediate parent and all its ancestors). It can use the inherited methods and variables as they are. It may also override an inherited method by providing its own version, or hide an inherited variable by defining a variable of the same name.

For example, the inherited method getArea() in a Cylinder object computes the base area of the cylinder. Suppose that we decide to override the getArea() to compute the surface area of the cylinder in the subclass Cylinder. Below are the changes:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | public class **Cylinder extends Circle** {  ......  // Override the getArea() method inherited from superclass Circle  @Override  public double getArea() {  return 2\*Math.PI\*getRadius()\*height + 2\*super.getArea();  }  // Need to change the getVolume() as well  public double getVolume() {  return super.getArea()\*height; // use superclass' getArea()  }  // Override the inherited toString()  @Override  public String toString() {  return "Cylinder[" + super.toString() + ",height=" + height + "]";  }  } |

If getArea() is called from a Circle object, it computes the area of the circle. If getArea() is called from a Cylinder object, it computes the surface area of the cylinder using the overridden implementation. Note that you have to use public accessor method getRadius() to retrieve the radius of the Circle, because radius is declared private and therefore not accessible to other classes, including the subclass Cylinder.

But if you override the getArea() in the Cylinder, the getVolume() (=getArea()\*height) no longer works. It is because the overridden getArea() will be used in Cylinder, which does not compute the base area. You can fix this problem by using super.getArea() to use the superclass' version of getArea(). Note that super.getArea() can only be issued from the subclass definition, but no from an instance created, e.g. c1.super.getArea(), as it break the information hiding and encapsulation principle.

#### Annotation @Override (JDK 1.5)

The "@Override" is known as annotation (introduced in JDK 1.5), which asks compiler to check whether there is such a method in the superclass to be overridden. This helps greatly if you misspell the name of the method to be overridden. For example, suppose that you wish to override method toString() in a subclass. If @Override is not used and toString() is misspelled as TOString(), it will be treated as a new method in the subclass, instead of overriding the superclass. If @Override is used, the compiler will signal an error.

@Override annotation is optional, but certainly nice to have.

Annotations are not programming constructs. They have no effect on the program output. It is only used by the compiler, discarded after compilation, and not used by the runtime.

#### Keyword "super"

Recall that inside a class definition, you can use the keyword this to refer to this instance. Similarly, the keyword super refers to the superclass, which could be the immediate parent or its ancestor.

The keyword super allows the subclass to access superclass' methods and variables within the subclass' definition. For example, super() and super(argumentList) can be used invoke the superclass’ constructor. If the subclass overrides a method inherited from its superclass, says getArea(), you can use super.getArea() to invoke the superclass' version within the subclass definition. Similarly, if your subclass hides one of the superclass' variable, you can use super.variableName to refer to the hidden variable within the subclass definition.

#### More on Constructors

Recall that the subclass inherits all the variables and methods from its superclasses. Nonetheless, the subclass does not inherit the constructors of its superclasses. Each class in Java defines its own constructors.

In the body of a constructor, you can use super(args) to invoke a constructor of its immediate superclass. Note that super(args), if it is used, must be the first statement in the subclass' constructor. If it is not used in the constructor, Java compiler automatically insert a super() statement to invoke the no-arg constructor of its immediate superclass. This follows the fact that the parent must be born before the child can be born. You need to properly construct the superclasses before you can construct the subclass.

#### Default no-arg Constructor

If no constructor is defined in a class, Java compiler automatically create a no-argument (no-arg) constructor, that simply issues a super() call, as follows:

// If no constructor is defined in a class, compiler inserts this no-arg constructor

public ClassName () {

super(); // call the superclass' no-arg constructor

}

Take note that:

* The default no-arg constructor will not be automatically generated, if one (or more) constructor was defined. In other words, you need to define no-arg constructor explicitly if other constructors were defined.
* If the immediate superclass does not have the default constructor (it defines some constructors but does not define a no-arg constructor), you will get a compilation error in doing a super() call. Note that Java compiler inserts a super() as the first statement in a constructor if there is no super(args).

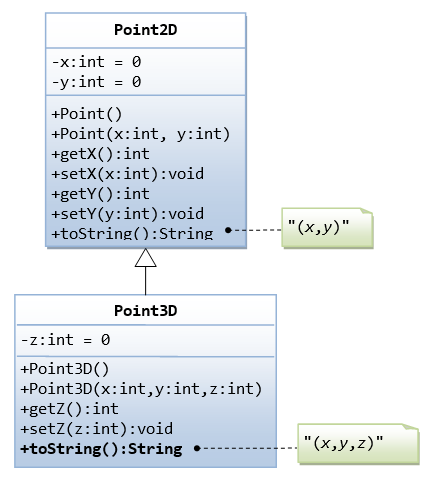
#### Single Inheritance

Java does not support multiple inheritance (C++ does). Multiple inheritance permits a subclass to have more than one direct superclasses. This has a serious drawback if the superclasses have conflicting implementation for the same method. In Java, each subclass can have one and only one direct superclass, i.e., single inheritance. On the other hand, a superclass can have many subclasses.

#### Common Root Class - java.lang.Object

Java adopts a so-called common-root approach. All Java classes are derived from a common root class called java.lang.Object. This Object class defines and implements the common behaviors that are required of all the Java objects running under the JRE. These common behaviors enable the implementation of features such as multi-threading and garbage collector.

#### 2: The Point2D and Point3D Classes



##### The Superclass Point2D.java

/\*

\* The Point2D class models a 2D point at (x, y).

\*/

public class Point2D {

// Private instance variables

private int x, y;

// Constructors

public Point2D() { // default constructor

this.x = 0;

this.y = 0;

}

public Point2D(int x, int y) {

this.x = x;

this.y = y;

}

// Getters and Setters

public int getX() {

return this.x;

}

public void setX(int x) {

this.x = x;

}

public int getY() {

return this.y;

}

public void setY(int y) {

this.y = y;

}

// Return "(x,y)"

public String toString() {

return "(" + this.x + "," + this.y + ")";

}

}

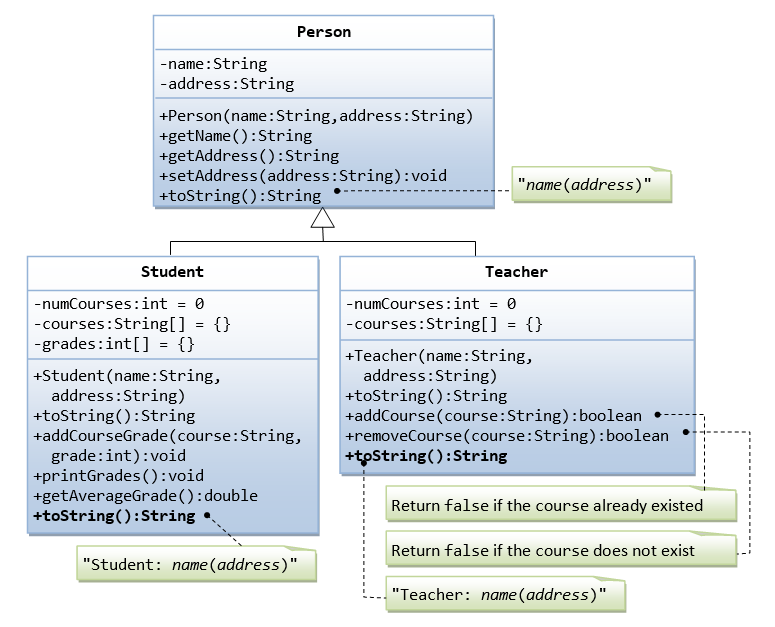
##### The Subclass Point3D.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | /\*  \* The Point3D class models a 3D point at (x, y,z),  \* which is a subclass of Point2D.  \*/  public class Point3D extends Point2D {  // Private instance variables  private int z;  // Constructors  public Point3D() { // default constructor  super(); // x = y = 0  this.z = 0;  }  public Point3D(int x, int y, int z) {  super(x, y);  this.z = z;  }  // Getters and Setters  public int getZ() {  return this.z;  }  public void setZ(int z) {  this.z = z;  }  // Return "(x,y,z)"  @Override  public String toString() {  return "(" + super.getX() + "," + super.getY() + "," + this.z + ")";  }  } |

##### A Test Driver for Point2D and Point3D Classes (TestPoint2DPoint3D.java)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | /\*  \* A test driver for the Point2D and Point3D classes  \*/  public class TestPoint2DPoint3D {  public static void main(String[] args) {  /\* Test Point2D \*/  // Test constructors and toString()  Point2D p2a = new Point2D(1, 2);  System.out.println(p2a); // toString()  Point2D p2b = new Point2D(); // default constructor  System.out.println(p2b);  // Test Setters and Getters  p2a.setX(3); // Test setters  p2a.setY(4);  System.out.println(p2a); // toString()  System.out.println("x is: " + p2a.getX());  System.out.println("x is: " + p2a.getY());  /\* Test Point3D \*/  // Test constructors and toString()  Point3D p3a = new Point3D(11, 12, 13);  System.out.println(p3a); // toString()  Point2D p3b = new Point3D(); // default constructor  System.out.println(p3b);  // Test Setters and Getters  p3a.setX(21); // in superclass  p3a.setY(22); // in superclass  p3a.setZ(23); // in this class  System.out.println(p3a); // toString()  System.out.println("x is: " + p3a.getX()); // in superclass  System.out.println("y is: " + p3a.getY()); // in superclass  System.out.println("z is: " + p3a.getZ()); // in this class  }  } |

#### 3: Superclass Person and its Subclasses



Suppose that we are required to model students and teachers in our application. We can define a superclass called Person to store common properties such as name and address, and subclasses Studentand Teacher for their specific properties. For students, we need to maintain the courses taken and their respective grades; add a course with grade, print all courses taken and the average grade. Assume that a student takes no more than 30 courses for the entire program. For teachers, we need to maintain the courses taught currently, and able to add or remove a course taught. Assume that a teacher teaches not more than 5 courses concurrently.

We design the classes as follows.

##### The Superclass Person.java

/\*

\* Superclass Person has name and address.

\*/

public class **Person** {

// private instance variables

private String name, address;

// Constructor

public Person(String name, String address) {

this.name = name;

this.address = address;

}

// Getters and Setters

public String getName() {

return name;

}

public String getAddress() {

return address;

}

public void setAddress(String address) {

this.address = address;

}

// Describle itself

public String toString() {

return name + "(" + address + ")";

}

}

##### The Subclass Student.java

/\*

\* The Student class, subclass of Person.

\*/

public class **Student extends Person** {

// private instance variables

private int numCourses; // number of courses taken so far

private String[] courses; // course codes

private int[] grades; // grade for the corresponding course codes

private static final int MAX\_COURSES = 30; // maximum number of courses

// Constructor

public Student(String name, String address) {

super(name, address);

numCourses = 0;

courses = new String[MAX\_COURSES];

grades = new int[MAX\_COURSES];

}

// Describe itself

@Override

public String toString() {

return "Student: " + super.toString();

}

// Add a course and its grade - No validation in this method

public void addCourseGrade(String course, int grade) {

courses[numCourses] = course;

grades[numCourses] = grade;

++numCourses;

}

// Print all courses taken and their grade

public void printGrades() {

System.out.print(this);

for (int i = 0; i < numCourses; ++i) {

System.out.print(" " + courses[i] + ":" + grades[i]);

}

System.out.println();

}

// Compute the average grade

public double getAverageGrade() {

int sum = 0;

for (int i = 0; i < numCourses; i++ ) {

sum += grades[i];

}

return (double)sum/numCourses;

}

}

##### The Subclass Teacher.java

/\*

\* The Teacher class, subclass of Person.

\*/

public class **Teacher extends Person** {

// private instance variables

private int numCourses; // number of courses taught currently

private String[] courses; // course codes

private static final int MAX\_COURSES = 5; // maximum courses

// Constructor

public Teacher(String name, String address) {

super(name, address);

numCourses = 0;

courses = new String[MAX\_COURSES];

}

// Describe itself

@Override

public String toString() {

return "Teacher: " + super.toString();

}

// Return false if the course already existed

public boolean addCourse(String course) {

// Check if the course already in the course list

for (int i = 0; i < numCourses; i++) {

if (courses[i].equals(course)) return false;

}

courses[numCourses] = course;

numCourses++;

return true;

}

// Return false if the course cannot be found in the course list

public boolean removeCourse(String course) {

boolean found = false;

// Look for the course index

int courseIndex = -1; // need to initialize

for (int i = 0; i < numCourses; i++) {

if (courses[i].equals(course)) {

courseIndex = i;

found = true;

break;

}

}

if (found) {

// Remove the course and re-arrange for courses array

for (int i = courseIndex; i < numCourses-1; i++) {

courses[i] = courses[i+1];

}

numCourses--;

return true;

} else {

return false;

}

}

}

##### A Test Driver (TestPerson.java)

/\*

\* A test driver for Person and its subclasses.

\*/

public class **TestPerson** {

public static void main(String[] args) {

/\* Test Student class \*/

Student s1 = new Student("Tan Ah Teck", "1 Happy Ave");

s1.addCourseGrade("IM101", 97);

s1.addCourseGrade("IM102", 68);

s1.printGrades();

System.out.println("Average is " + s1.getAverageGrade());

/\* Test Teacher class \*/

Teacher t1 = new Teacher("Paul Tan", "8 sunset way");

System.out.println(t1);

String[] courses = {"IM101", "IM102", "IM101"};

for (String course: courses) {

if (t1.addCourse(course)) {

System.out.println(course + " added.");

} else {

System.out.println(course + " cannot be added.");

}

}

for (String course: courses) {

if (t1.removeCourse(course)) {

System.out.println(course + " removed.");

} else {

System.out.println(course + " cannot be removed.");

}

}

}

}

Tan Ah Teck(1 Happy Ave)

Tan Ah Teck

8 Sunrise Place

Student: Mohd Ali(8 Kg Java)

Mohd Ali

9 Kg Satu

Student: Mohd Ali(9 Kg Satu) IM101:97 IM102:68

Average is: 82.5

Teacher: Paul Tan(8 sunset way)

IM101 added.

IM102 added.

IM101 cannot be added.

IM101 removed.

IM102 removed.

IM101 cannot be removed.

# Daily Work – 9th February 2022

# (To be done during Class and lab time)

**Exercise 1**:

Consider a superclass **PurchaseItem** which models customer’s purchases. This class has:

* two private instance variables *name* (String) and *unitPrice* (double).
* One constructor to initialize the instance variables.
* A default constructor to initialize name to “no item”, and unitPrice to 0. use *this( )*
* A method *getPrice* that returns the unitPrice.
* Accessor and mutator methods.
* A *toString* method to return the name of the item followed by @ symbol, then the unitPrice.

Consider two subclasses **WeighedItem** and **CountedItem**. WeighedItem has an additional instance variable *weight* (double) in Kg while CountedItem has an additional variable *quantity* (int) both private.

- Write an appropriate constructor for each of the classes making use of the constructor of the superclass in defining those of the subclasses.

- Override *getPrice* method that returns the price of the purchasedItem based on its unit price and weight (WeighedItem), or quantity (CountedItem). Make use of getPrice of the superclass

- Override also *toString* method for each class making use of the toString method of the superclass in defining those of the subclasses. ***toString* should return something that can be printed on the receipt.**

**For example**

Banana @ 3.00 1.37Kg 4.11 SR (in case of WeighedItem class)

Pens @ 4.5 10 units 45 SR (in case of CountedItem class)

Write an application class where you construct objects from the two subclasses and print them on the screen.

**2. Shadowing, Overriding, Hiding**

Suppose you're reading some Java code, and you come across something like this:

class A {

int A = 37;

A() {

int A = 47;

A aref = new A()

{

int A = 57;

void A() {}

};

}

}

This usage is legal, but not necessarily desirable. In fact, it raises an interesting question about how the Java programming language specification treats conflicting names. There are several terms used to describe various cases: shadowing, overriding, and hiding

First an important point needs to be made: just because the Java programming language allows you to do something, it doesn't always mean that it's a desirable thing to do. For example, it's legal to say:

class A {

int A;

}

in a program, but you probably shouldn't because it's confusing. The best way to handle issues with conflicting names is to simply avoid them as far as possible. For example, you can avoid many problems if you follow a coding convention that specifies that the first letter of a type name (such as "class A") should be capitalized, while the first letter of a field name (such as "int A") should be lowercase.

Now let's look at an example of shadowing:

public class Shadow {

int a;

int b;

// parameters a/b shadow instance variables a/b

public Shadow(int a, int b) {

// set parameter equal to itself

a = a;

// set instance variable b equal to parameter b

this.b = b;

}

public static void main(String args[]) {

Shadow s = new Shadow(37, 47);

System.out.println("a = " + s.a);

System.out.println("b = " + s.b);

}

}

When your run Shadow, you should see:

a = 0

b = 47

One place shadowing comes up is when you have field names and parameter names that are the same, and you want to use the parameters to set the fields:

int a;

public void f(int a) {

a = a;

}

This doesn't work, because **the parameter "a" shadows the field "a"**, that is, the parameter name blocks access via a simple name to the field name. You can get around this problem by saying:

this.a = a;

which means "set field a to parameter a". Whether this style of usage is desirable or not depends on your particular biases; one point in its favor is that you don't have to invent parameter names like "a1" or "\_a".

The second example is one that illustrates **overriding:**

class B {

void f() {

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

}

}

public class Override extends B {

// instance method f overrides instance method B.f

void f() {

System.out.println("Override.f");

}

void g() {

// call Override.f

f();

// call B.f

super.f();

}

public static void main(String args[]) {

Override o = new Override();

o.g();

}//end of main()

}//end of class

When you run Override, you should see:

Override.f

B.f

In this example, the method Override.f overrides the method B.f. If you have an object of type Override, and call f, Override.f is called. However if you have an object of type B, B.f is called. This approach is a standard part of object-oriented programming. For example, java.lang.Object declares a hashCode method, but subclasses, such as String, provide an overriding version of the method. The overriding version is tailored to the particular type of data represented by the class. You can call the superclass method by using the notation:

super.f();

A third example is that of **hiding**

class A {

static void f() { // static method

System.out.println("A.f");

}

void g() { // instance method

System.out.println("A.g");

}

}

public class SubA extends A {

static void f() { // static methods are not overridden

System.out.println("Hide.f");

}

void g() {

System.out.println("Hide.g");

}

public static void main(String args[]) {

A aref = new SubA();

// call A.f()

aref.f();

// call SubA.g()

aref.g();

}

}

:

When you run Hide, you should see:

A.f

Hide.g

In this example, Hide.f hides A.f, and Hide.g overrides A.g. One way of seeing the difference between hiding and overriding is to note that overriding applies to regular instance methods; the actual method that is called is determined at run time based on the type of the actual object. This sort of dynamic lookup does not happen for static methods or for fields. For example, in this code:

class C {

int x = 37;

void f() {

System.out.println("C.f");

}

}

public class Lookup extends C{

int x = 47;

void f() {

System.out.println("Lookup.f");

}

public static void main(String args[]) {

C cref = new Lookup();

// call Lookup.f

cref.f();

// display C.x

System.out.println(cref.x);

}

}

the method reference through "cref" results in Lookup.f being called, but the field reference obtains C.x. Or to say it another way, the actual class of an object determines which instance method is called. But for fields and static methods, the type of the reference is used (here it's cref, of type C). When you run Lookup, you should see:

Lookup.f

37

**Exercise 2:**

Use the sample Java programs provided above and modify them to answer the following questions.

1. Can an instance method override a static method?
2. Can a static method override (hide) an instance method?
3. Can you override a final instance method?
4. Can you override an instance method and make it final?
5. Can you override an instance method and change its return type?
6. Can you hide a final static method ?
7. Can an instance field hide a static field?
8. Can a static field hide an instance field?
9. Can an instance method with public visibility override an instance method with default visibility?
10. Can an instance method with default visibility override an instance method with public visibility?
11. Can an instance method with protected visibility override an instance method with default visibility?
12. Can an instance method with default visibility override an instance method with protected visibility?
13. Based on the last four question, order the access visibility from the widest to the narrowest (weakest) and state the rule for overriding (instance methods) or hiding (static methods) ?