Method Overriding

Method Overriding

➢when a method in a subclass has *the same name and type signature* as a method in its superclass

➢then the method in the subclass is said to *override* the method in the superclass.

➢When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass.

➢The version of the method defined by the superclass will be hidden.

```java
// Method overriding.
class A {
int i, j;
A(int a, int b) {
i = a;
j = b;
}
// display i and j
void show() {
System.out.println("i and j: " + i + " " + j);
}
}
class B extends A {
int k;
B(int a, int b, int c) {
super(a, b);
k = c;
}
// display k – this overrides show() in A
void show() {
     super.show(); // this calls A's show()
System.out.println("k: " + k);
}
}
```

```java
class Override {
public static void main(String args[]) {
B subOb = new B(1, 2, 3);
subOb.show(); // this calls show() in B
}
}
```

Output :

```
i and j: 1 2
k: 3
```

Difference between Method Overloading &Method Overriding

```
 class A {
 int i, j;
 A(int a, int b) {
 i = a;
 j = b;
 }
 // display i and j
 void show( ) {
 System.out.println("i and j: " + i + " " + j);
 }
 }
// Create a subclass by extending class A.
class B extends A {
int k;
B(int a, int b, int c) {
super(a, b);
k = c;
=}
// overload show( )
void show(String msg) {
System.out.println(msg + k);
}
```

```
class Override {
public static void main(String args[]) {
B subOb = new B(1, 2, 3);

// this calls show( ) in B

subOb.show("This is k: ");

// this calls show( ) in A

subOb.show();
}
}
```

Output:

```
This is k: 3
i and j: 1 2
```

Dynamic Method Dispatch

Dynamic Method Dispatch

- Method overriding forms the basis for one of Java’s most powerful concepts: *dynamic method dispatch.*
- Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.
- Dynamic method dispatch is important because this is how Java implements run-time polymorphism.
- An important principle: a superclass reference variable can refer to a subclass object
- Java uses this fact to resolve calls to overridden methods at run time.
- When an overridden method is called through a superclass reference, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs.
- Thus, this determination is made at run time.
- When different types of objects are referred to, different versions of an overridden method will be called.
- In other words, *it is the type of the object being referred to* (not the type of the reference variable) that determines which version of an overridden method will be executed.

```
class A {
void callme( ) {
System.out.println("Inside A's callme
method");
}
}
class B extends A {
// override callme( )
void callme( ) {
System.out.println("Inside B's callme
method");
}
}
class C extends A {
// override callme( )
void callme( ) {
System.out.println("Inside C's callme
method");
}
}
```

```
class Dispatch {
public static void main(String args[]) {
A a = new A( ); // object of type A
B b = new B( ); // object of type B
C c = new C( ); // object of type C
A r; // obtain a reference of type A
r = a; // r refers to an A object
r.callme( ); // calls A's version of callme
r = b; // r refers to a B object
r.callme( ); // calls B's version of callme
r = c; // r refers to a C object
r.callme( ); // calls C's version of callme
}
}
```

Output:

```
Inside A’s callme method
Inside B’s callme method
Inside C’s callme method
```

Abstract Classes

Using Abstract Classes

To declare an abstract method, use this general form:

abstract *type method-name(parameter-list)*;

➢certain methods be overridden by subclasses by specifying the **abstract** type modifier.

➢These methods are sometimes referred to as *subclasser responsibility* because they have no implementation specified in the superclass.

➢Thus, a subclass must override them, it cannot simply use the version defined in the superclass.

➢Any class that contains one or more abstract methods must also be declared abstract.

➢There can be no objects of an abstract class. That is, an abstract class cannot be directly instantiated with the **new** operator.

➢cannot declare abstract constructors, or abstract static methods

➢Any subclass of an abstract class must either implement all of the abstract methods in the superclass, or be itself declared **abstract**.

```
abstract class A {
abstract void callme( );

// concrete methods are still allowed in abstract classes

void callmetoo( ) {
System.out.println("This is a concrete method.");
}
}

class B extends A {
void callme( ) {
System.out.println("B's implementation of callme.");
}
}

class AbstractDemo {
public static void main(String args[ ]) {
B b = new B( );
b.callme( );
b.callmetoo( );
}
}
```

```
abstract class Figure {
 double dim1;
 double dim2;
 Figure(double a, double b) {
 dim1 = a;
 dim2 = b;
 }
 // area is now an abstract method
 abstract double area( );
 }
 class Rectangle extends Figure {
 Rectangle(double a, double b) {
 super(a, b);
 }
 // override area for rectangle
 double area( ) {
 System.out.println("Inside Area for
 Rectangle.");
 return dim1 * dim2;
 }  }
 class Triangle extends Figure {
 Triangle(double a, double b) {
 super(a, b);
 }
```

```
// override area for right triangle
double area( ) {
System.out.println("Inside Area for Triangle.");
return dim1 * dim2 / 2;
}
}

class AbstractAreas {
public static void main(String args[ ]) {
// Figure f = new Figure(10, 10); // illegal now
Rectangle r = new Rectangle(9, 5);
Triangle t = new Triangle(10, 8);
Figure figref; // this is OK, no object is created
figref = r;
System.out.println("Area is " + figref.area( ));
figref = t;
System.out.println("Area is " + figref.area( ));
}
}
```

final

The keyword **final** has three uses.

- (1)First, it can be used to create the equivalent of a named constant.

  Eg:- final double radius=4.5;

- The other two uses of **final** apply to inheritance.

(2)Using final to Prevent Overriding - Methods declared as **final** cannot be overridden.

```
class A {
final void meth( ) {
System.out.println("This is a final method.");
}
}

class B extends A {
void meth( ) { // ERROR! Can't override.
System.out.println("Illegal!");
}
}
```

(3)Using final to Prevent Inheritance

- Declaring a class as **final** implicitly declares all of its methods as **final**, too

- it is illegal to declare a class as both **abstract** and **final** since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations.

```
final class A {
// ...
}
// The following class is illegal.
class B extends A { // ERROR! Can't subclass A
// ...
}
```

it is illegal for **B** to inherit **A** since **A** is declared as **final**.