Programming in Java

Lecture 11: Inheritance(2)

Mahesh Kumar
Assistant Professor (Adhoc)

Department of Computer Science
Acharya Narendra Dev College
University of Delhi

Course webpage
[http://www.mkbhandari.com/mkwiki]

Outline

1. Method Overriding
2. Dynamic Method Dispatch
3. Using Abstract Classes
4. Using Final with Inheritance
5. The Object Class

Method Overriding

- In a class hierarchy, 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 its 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.

Method Overriding

```
// 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( ) {
		System.out.println("k: " + k);
	}
}
```

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

This program displays the following output :

```
k: 3
```

1 When show( ) is invoked on an object of type B, the version of show( ) defined within B is used.

2 That is, the version of show( ) inside B overrides the version declared in A.

Q How to access the superclass version of an overridden method?

Method Overriding

- If you wish to access the superclass version of an overridden method, you can do so by using **super**.

```
// To access Superclass version of show( )

class B extends A {
        int k;
        B(int a, int b, int c) {
                super(a, b);
                k = c;
        }
        void show( ) {
                super.show( );    // this calls A's show( )
                System.out.println("k: " + k);
        }
}
```

This program displays the following output :

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

1 Here, super.show( ) calls the superclass version of show( ).

2 Method overriding occurs only when the **names** and the **type signatures** of the two methods are **identical.**

Q What if **names** and the **type signatures** of the two methods are **non-identical?**

Method Overriding

```
// Methods with differing type signatures are overloaded – not overridden.
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;
        }
        void show(String msg) {  // overload show( )
                System.out.println(msg + k);
        }
}
```

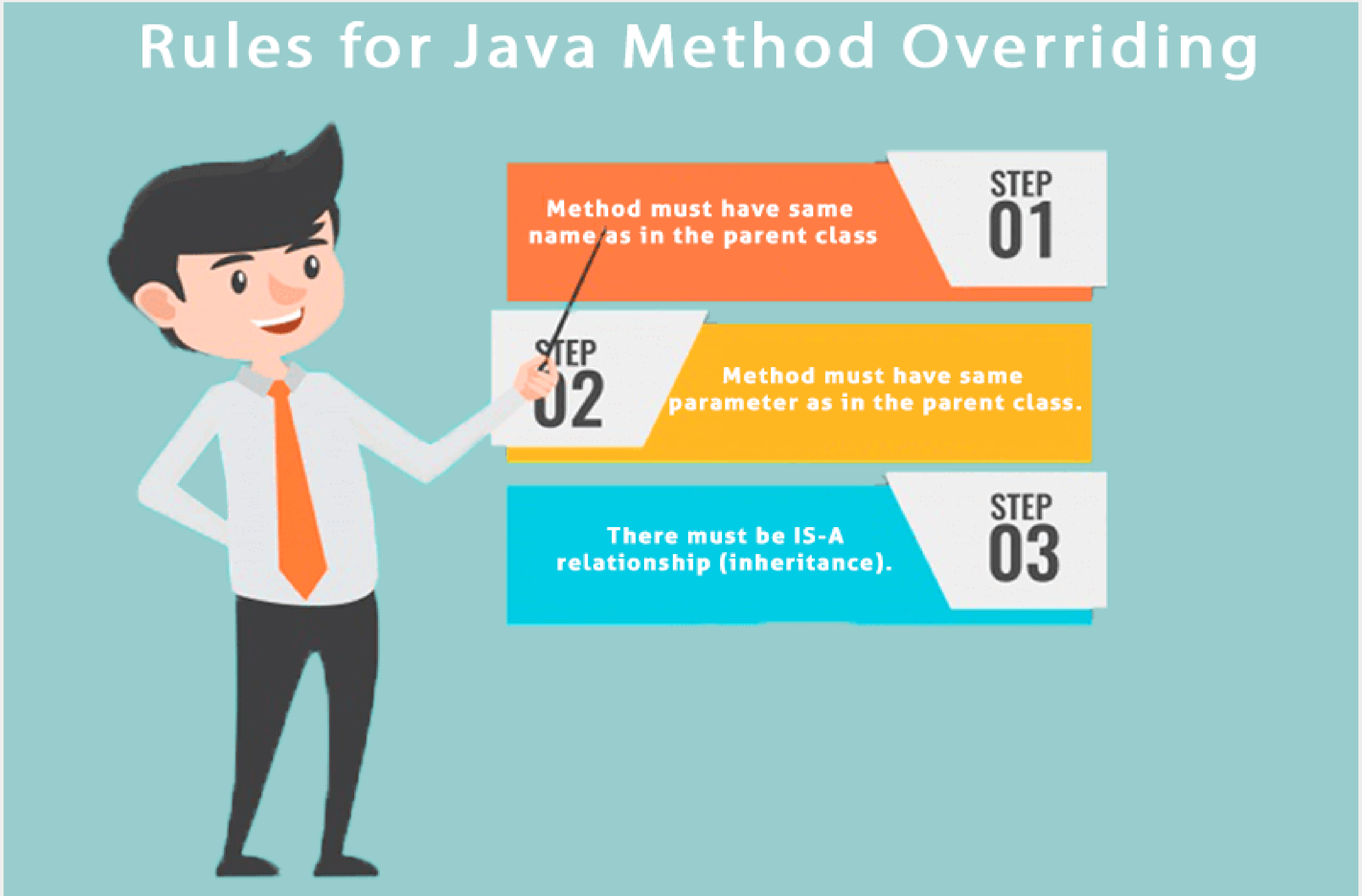
```
class Override {
        public static void main(String args[ ]) {
                B subOb = new B(1, 2, 3);
                subOb.show("This is k: "); // this calls show( ) in B
                subOb.show( );     // this calls show() in A
        }
}
```

This program displays the following output :

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

1. The version of show( ) in B takes a string parameter. This makes its type signature different from the one in A, which takes no parameters.
2. Therefore, no overriding (or name hiding) takes place – so show( ) is overloaded here.

Method Overriding - Summary



Rules for Java Method Overriding

STEP 01

Method must have same name as in the parent class

STEP 02

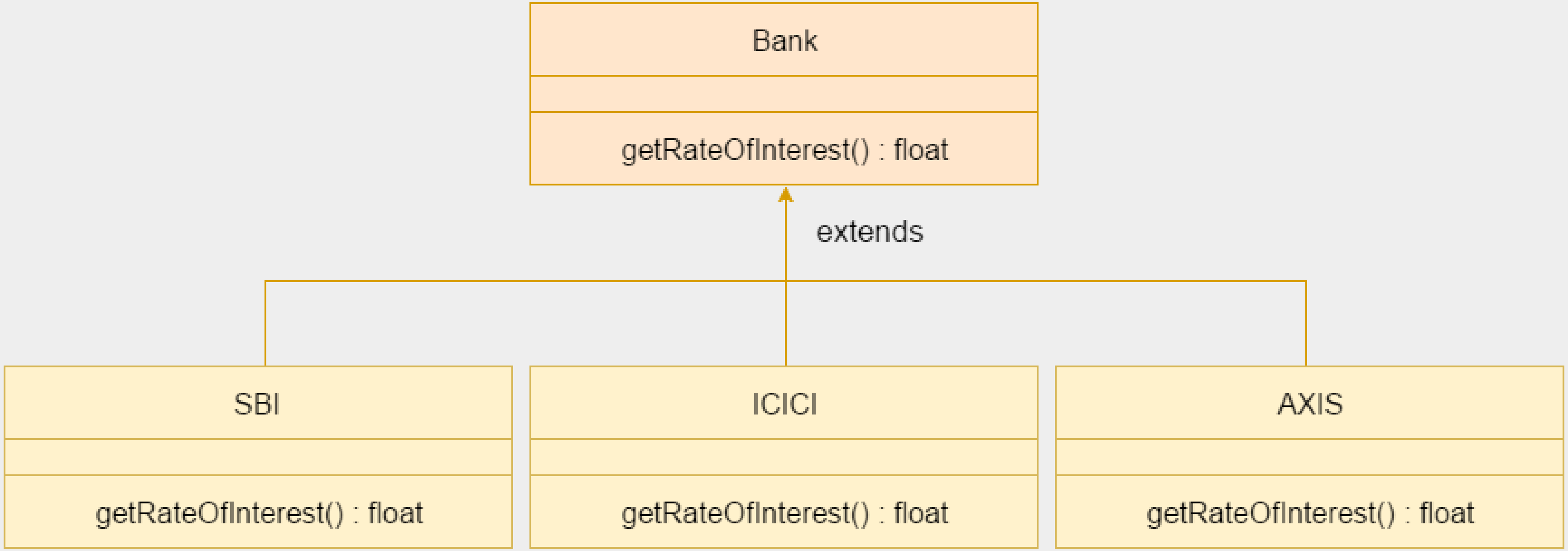
Method must have same parameter as in the parent class.

STEP 03

There must be IS-A relationship (inheritance).

[Source: (3)]

Method Overriding - Summary


Bank
getRateOfInterest() : float
extends
SBI
getRateOfInterest() : float
ICICI
getRateOfInterest() : float
AXIS
getRateOfInterest() : float

[Source: (3)]

Method Overriding - Summary

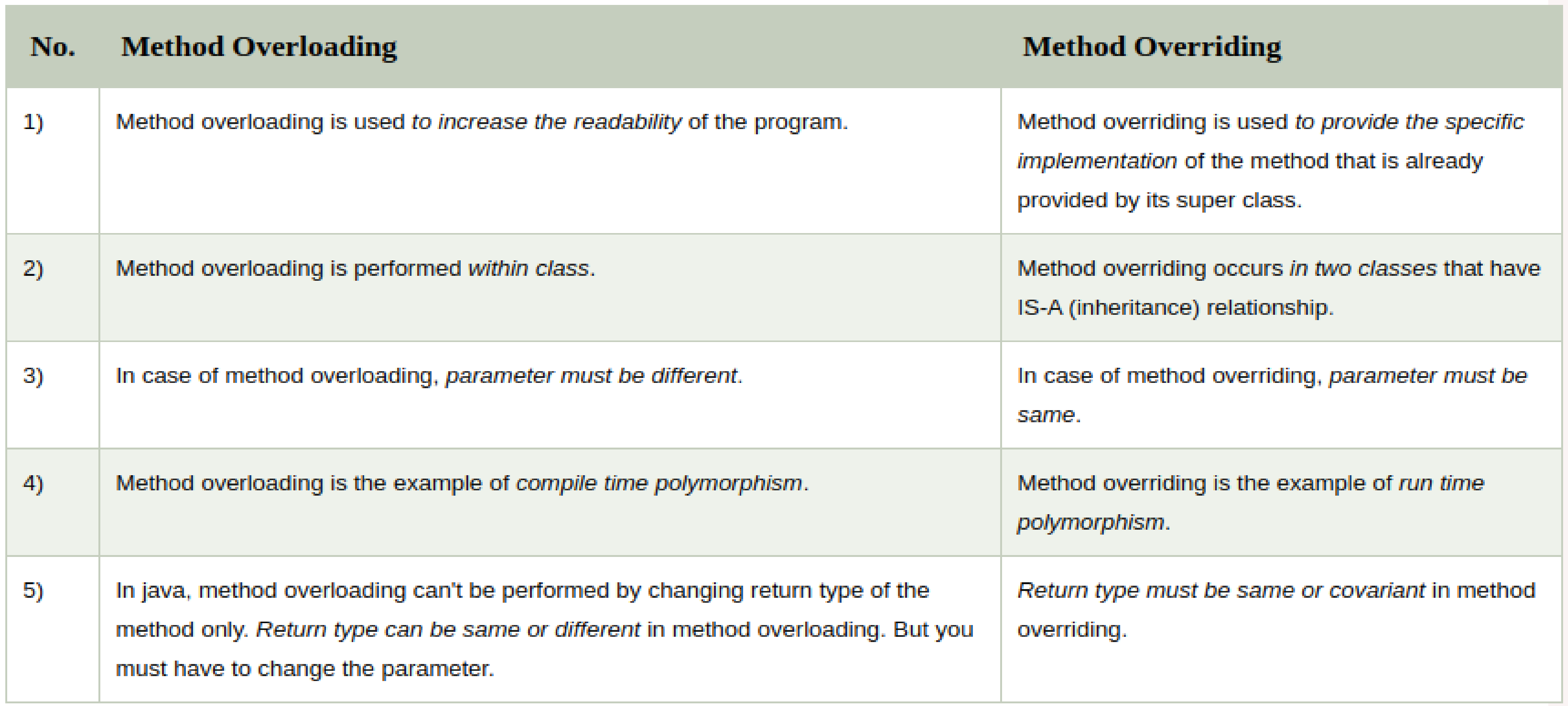
```
//Java Program to demonstrate the real scenario of Java Method Overriding
//where three classes are overriding the method of a parent class.
class Bank{
        int getRateOfInterest( ){
                return 0;
        }
}
class SBI extends Bank{
                int getRateOfInterest( ){
                return 8;
        }
}
class ICICI extends Bank{
        int getRateOfInterest( ){
                return 7;
        }
}
class AXIS extends Bank{
        int getRateOfInterest( ){
                return 9;
        }
}
```

```
//Test class to create objects and call the methods
class Test{
        public static void main(String args[ ]){
                SBI s=new SBI();
                ICICI i=new ICICI();
                AXIS a=new AXIS();
                System.out.println("SBI Rate of Interest: “
                                                        +s.getRateOfInterest( ));
                System.out.println("ICICI Rate of Interest: "
                                                        +i.getRateOfInterest( ));
                System.out.println("AXIS Rate of Interest: "
                                                        +a.getRateOfInterest( ));

        }
}
```

```
Output:
SBI Rate of Interest: 8
ICICI Rate of Interest: 7
AXIS Rate of Interest: 9
```

[Source: (3)]

Method Overriding vs Overloading



No.	Method Overloading	Method Overriding
1)	Method overloading is used *to increase the readability* of the program.	Method overriding is used *to provide the specific implementation* of the method that is already provided by its super class.
2)	Method overloading is performed *within class*.	Method overriding occurs *in two classes* that have IS-A (inheritance) relationship.
3)	In case of method overloading, *parameter must be different*.	In case of method overriding, *parameter must be same*.
4)	Method overloading is the example of *compile time polymorphism*.	Method overriding is the example of *run time polymorphism*.
5)	In java, method overloading can't be performed by changing return type of the method only. *Return type can be same or different* in method overloading. But you must have to change the parameter.	*Return type must be same or covariant* in method overriding.

[Source: (3)]

Method Overriding vs Overloading

```
//Method Overloading example
class OverloadingExample{

        static int add(int a,int b){
                return a+b;
        }

        static int add(int a,int b,int c){
                return a+b+c;
        }
}
```

```
//Method Overriding example
class Animal{

        void eat( ){
                System.out.println("eating...");
        }
}

class Dog extends Animal{

        void eat( ){
                System.out.println("eating bread...");
        }
}
```

[Source: (3)]

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.
- As aready discussed, **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.

Dynamic Method Dispatch

- 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.
- Dynamic method dispatch is important because this is how Java implements run-time polymorphism.
- 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.
- Therefore, if a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different versions of the method are executed.

Dynamic Method Dispatch

```
// Dynamic Method Dispatch
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
        }
}
```

Dynamic Method Dispatch

The output from the program is shown here:

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

NOTE: the version of callme( ) executed is determined by the type of object being referred to at the time of the call.

```
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
        }
}
```

Why Overridden Methods?

- The overridden methods allow Java to support run-time polymorphism.
- Polymorphism is essential to object-oriented programming for one reason: it allows a general class to specify methods that will be common to all of its derivatives, while allowing subclasses to define the specific implementation of some or all of those methods.
- Overridden methods are another way that Java implements the **"one interface, multiple methods"** aspect of polymorphism.
- Part of the key to successfully applying polymorphism is understanding that the superclasses and subclasses form a hierarchy which moves from lesser to greater specialization.
- Used correctly, the superclass provides all elements that a subclass can use directly.
- It also defines those methods that the derived class must implement on its own.

Why Overridden Methods?

- This allows the subclass the flexibility to define its own methods, yet still enforces a consistent interface.

- Overridden methods are another way that Java implements the “one interface, multiple methods” aspect of polymorphism.

- Thus, by combining inheritance with overridden methods, a superclass can define the general form of the methods that will be used by all of its subclasses.

- **Dynamic, run-time polymorphism** is one of the most powerful mechanisms that object-oriented design brings to bear on **code reuse and robustness.**

- The ability of existing code libraries to call methods on instances of new classes without recompiling while maintaining a clean abstract interface is a profoundly powerful tool.

Applying Method Overriding

```
// Using run-time polymorphism (a more practical example)
class Figure {
        double dim1;
        double dim2;
        Figure(double a, double b) {
                dim1 = a;
                dim2 = b;
        }
        double area( ) {
                System.out.println("Area for Figure is undefined.");
                return 0;
        }
}
class Rectangle extends Figure {
        Rectangle(double a, double b) {
                super(a, b);
        }
        double area( ) {  // override area for rectangle
                System.out.println("Inside Area for Rectangle.");
                return dim1 * dim2;
        }
}
```

Applying Method Overriding

```
class Triangle extends Figure {
        Triangle(double a, double b) {
                super(a, b);
        }
        double area( ) {    // override area for right triangle
                System.out.println("Inside Area for Triangle.");
                return dim1 * dim2 / 2;
        }
}
class FindAreas {
        public static void main(String args[ ]) {
                Figure f = new Figure(10, 10);
                Rectangle r = new Rectangle(9, 5);
                Triangle t = new Triangle(10, 8);
                Figure figref;
                figref = r;
                System.out.println("Area is " + figref.area());
                figref = t;
                System.out.println("Area is " + figref.area());
                Figref = f;
                System.out.println("Area is " + figref.area());
        }
}
```

The output from the program is shown here:

```
Inside Area for Rectangle.
Area is 45
Inside Area for Triangle.
Area is 40
Area for Figure is undefined.
Area is 0
```

1. Through the dual mechanisms of **inheritance and run-time polymorphism**, it is possible to define one consistent interface that is used by several different, yet related, types of objects.
2. In this case, if an object is derived from Figure, then its area can be obtained by calling area( ).
3. The interface to this operation is the same no matter what type of figure is being used.

Using Abstract Classess

- It is used to achieve abstraction which is one of the pillar of Object Oriented Programming(OOP).
- Abstraction is a process of hiding the implementation details and showing only functionality to the user. Abstraction lets you focus on what the object does instead of how it does it.
- A class which is declared as abstract is known as an abstract class. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.
- To declare a class abstract, use this general form :

```
abstract class class-name{
        //body of class
}
```

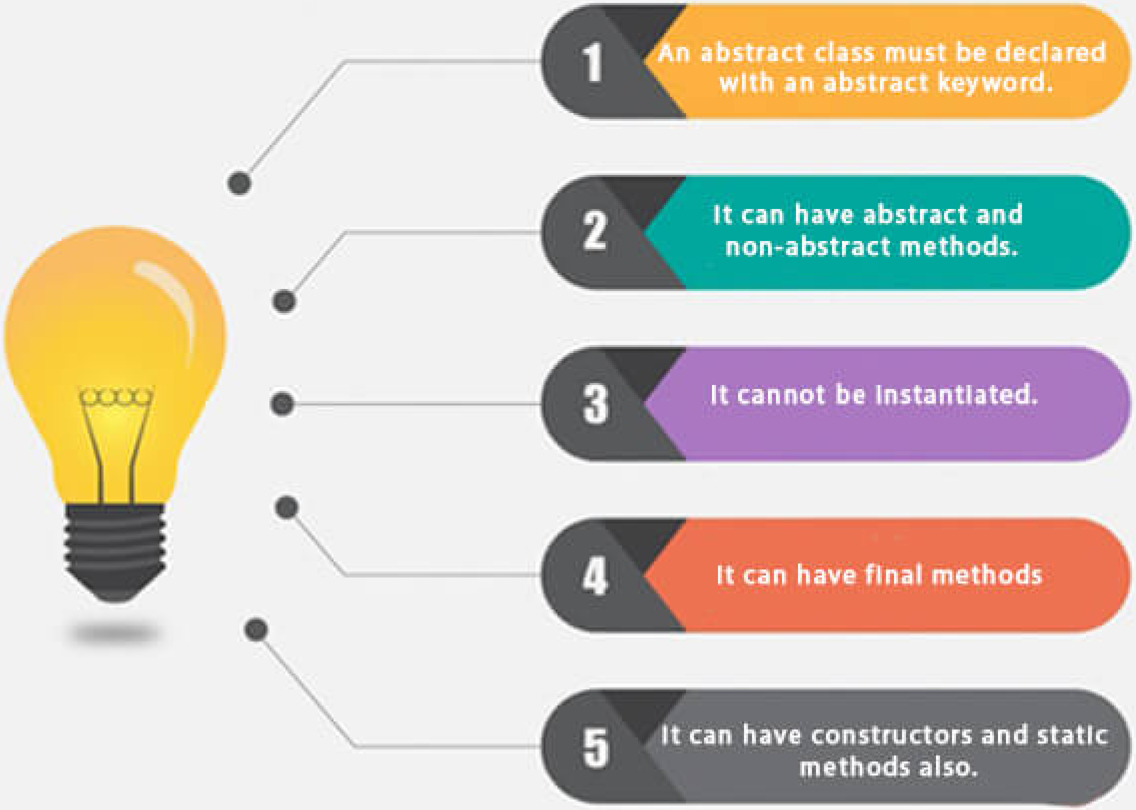
Using Abstract Classess

- A method which is declared as abstract and does not have implementation is known as an abstract method.
- You can require that 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.
- To declare an abstract method, use this general form:

```
abstract type name(parameter-list);  // no method body is present.
```

Using Abstract Classess

Rules for Java Abstract class



1. An abstract class must be declared with an abstract keyword.
2. It can have abstract and non-abstract methods.
3. It cannot be instantiated.
4. It can have final methods
5. It can have constructors and static methods also.

[Source: (3)]

Using Abstract Classess

```
// A Simple demonstration of 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( ) { // must override*
                System.out.println("B's implementation of callme.");
        }
}
class AbstractDemo {
        public static void main(String args[ ]) {
                B b = new B( );
                b.callme( );
                b.callmetoo( );
        }
}
```

* ohterwise **Compile Time** error will occur

1. Notice that no objects of class A are declared in the program

2. class A implements a concrete method (non-abstract) called callmetoo( )

3. Abstract classes can include as much implementation as they see fit.

4. Abstract classes can not be instantiated, but can create object references, because Java's approach to run-time polymorphism isimplemented through the use of superclass references.

Using Abstract Classess

```
// Improving th Figure class shown earlier
// Using abstract methods and classes
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);
        }
        // Must override area( )*
        double area( ) {
                System.out.println("Inside Area for Rectangle.");
                return dim1 * dim2;
        }
}
```

```
class Triangle extends Figure {
        Triangle(double a, double b) {
                super(a, b);
        }
        // Must override area( )*
        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( ));
        }
}
```

* ohterwise **Compile Time** error will occur

Using final with Inheritance

- The keyword final has three uses:

  1. Create the equivalent of a named constant (already discussed).

  2. Using final to Prevent Overriding

     To disallow a method from being overridden, specify final as a modifier at the start of its declaration.

     Methods declared as final cannot be overridden.

  3. Using final to Prevent Inheritance

     To prevent a class from being inherited, precede the class declaration with final.

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

- Can we declare declare a class as both abstract and final ?

Using final to Prevent Overridding

```
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!");
        }
}
```

* **Compile Time** error will occur

Using final to Prevent Inheritance

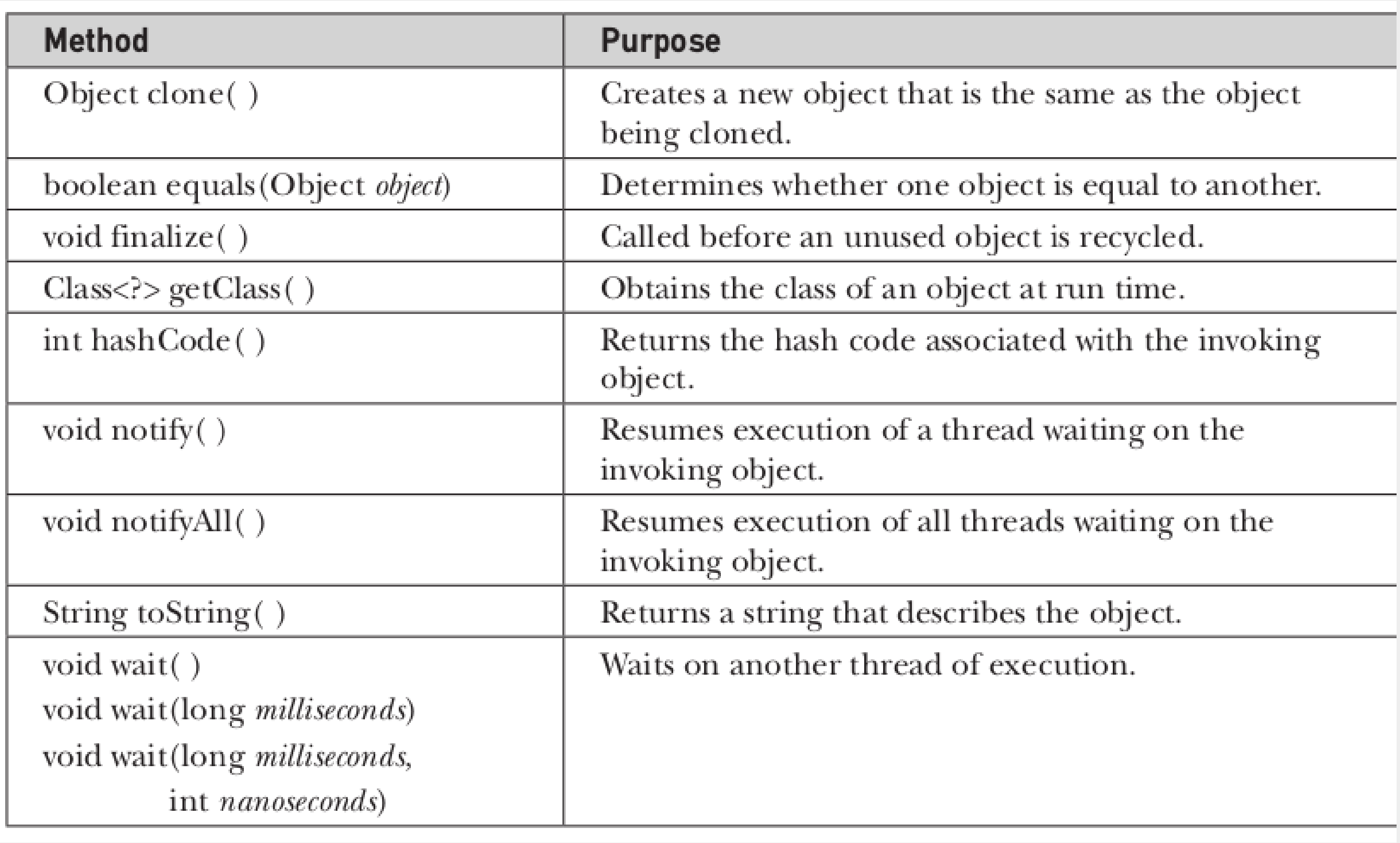
```
final class A {
        //...
}

// The following class is illegal.

class B extends A { // ERROR! Can't subclass A
        //...
}
```

NOTE: A **final class** can not have **abstract methods** and an **abstract class** can not be declared **final**.

The Object Class

- There is one special class, Object, defined by Java.
- All other classes are subclasses of Object. That is, Object is a superclass of all other classes.
- This means that a reference variable of type Object can refer to an object of any other class.
- Also, since arrays are implemented as classes, a variable of type Object can also refer to any array.
- Object defines some methods, which means that they are available in every object.

The Object Class



	Method	Purpose
	Object clone( )	Creates a new object that is the same as the object being cloned.
	boolean equals(Object *object*)	Determines whether one object is equal to another.
	void finalize( )	Called before an unused object is recycled.
*	Class<?> getClass( )	Obtains the class of an object at run time.
	int hashCode( )	Returns the hash code associated with the invoking object.
*	void notify( )	Resumes execution of a thread waiting on the invoking object.
*	void notifyAll( )	Resumes execution of all threads waiting on the invoking object.
	String toString( )	Returns a string that describes the object.
*	void wait( ) void wait(long *milliseconds*) void wait(long *milliseconds*, int *nanoseconds*)	Waits on another thread of execution.

References

R Reference for this topic

- [Book: Java: The Complete Reference, Ninth Edition: Herbert Schildt]
  https://www.amazon.in/Java-Complete-Reference-Herbert-Schildt/dp/0071808558

- [Web: GeeksforGeeks]
  https://www.geeksforgeeks.org/java/

- [Web: Java T Point tutorial]
  https://www.javatpoint.com/java-tutorial