Agenda

- Access Modifiers
- What is and Why Inheritance?
- How to derive a sub-class?
- Object class
- Constructor calling chain
- “super” keyword
- Overriding methods
- Hiding methods
- Final class and final methods
- Polymorphism

Access Modifiers

Static :

Static Is a access modifier.

Signature:

Variable - Static int b;

Method - static void meth(int x)

- When a member is declared as Static, it can be accessed before any objects of its class are created and without reference to any object.
- Static can be applied to Inner classes, Variables and Methods.
- Local variables can’t be declared as static.
- A static method can access only static Variables, and they can’t refer to this or super in any way.
- Static methods can’t be abstract.
- A static method may be called without creating any instance of the class.
- Only one instance of static variable will exist any amount of class instances.

Example:

```
class InitTest {
                InitTest (int x) {
                      System.out.println("1-arg const");
                }
                InitTest () {
                       System.out.println("no-arg const");
                }
               static {
                       System.out.println("1st static init");
               }

               {
                    System.out.println("1st instance init");
               }
               {
                    System.out.println("2nd instance init");
                }
               static {
                      System.out.println("2nd static init");
               }
               public static void main(String [] args) {
                         new InitTest ();
                         new InitTest (7);
               }
}
```

Output:

1st static init

2nd static init

1st instance init

2nd instance init

no-arg const

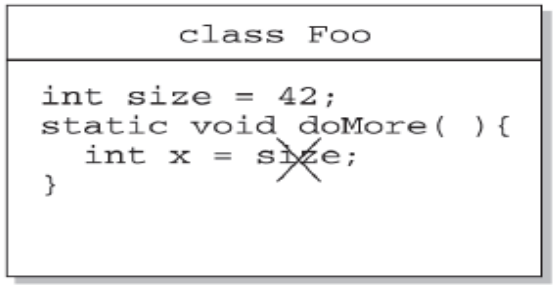
1st instance init

2nd instance init

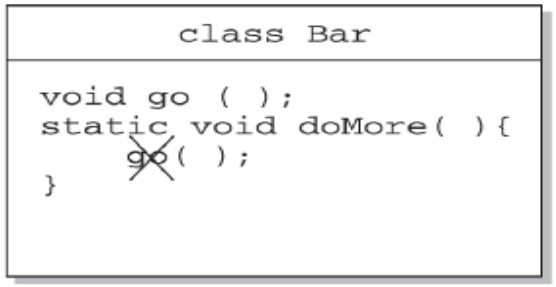
1-arg const

Note:

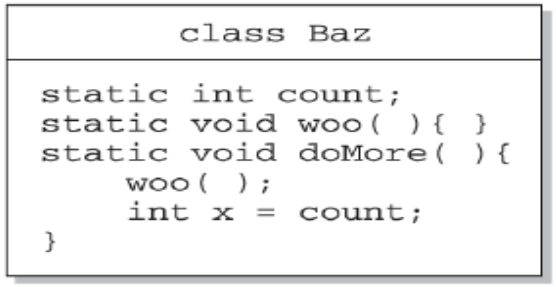
- Init blocks execute in the order they appear.
- Static init blocks run once, when the class is first loaded.
- Instance init blocks run every time a class instance is created.
- Instance init blocks run after the constructor's call to super().
- Instance init blocks are often used as a place to put code that all the constructors in a class should share. That way, the code doesn't have to be duplicated across constructors.



static method cannot access an instance (non-static) variable



static method cannot access a non-static method

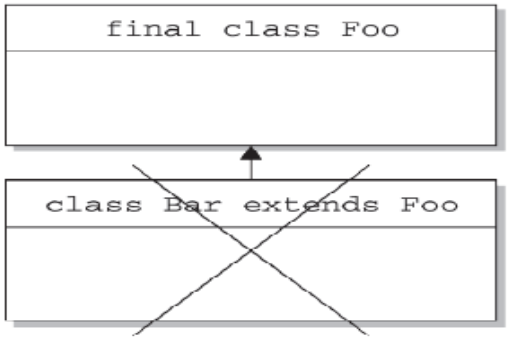


static method *can* access a static method or variable

final

- final is a access modifier
- All the Variables, methods and classes can be declared as Final.
- Classes declared as final class can’t be sub classed.
- Method ‘s declared as final can’t be over ridden.
- If a Variable is declared as final, the value contained in the Variable can’t be changed.
- Static final variable must be assigned in to a value in static initialized block.

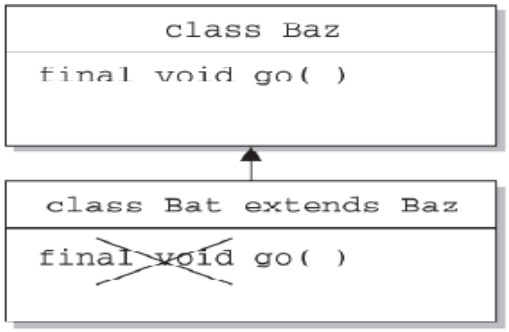
final class



final *class* cannot be subclassed

---

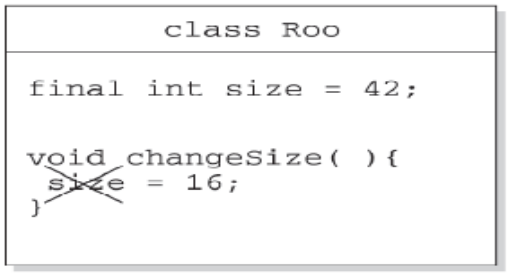
final method



final *method* cannot be overridden by a subclass

---

final variable



final *variable* cannot be assigned a new value, once the initial method is made (the initial assignment of a value must happen before the constructor completes).

transient :

- transient is a access modifier
- Transient Variable is not persisten.
- Transient variables may not be final or static, But the complies allows the declaration and no compile time error is generated.

- Transient can be applied only to class level variables.
- Local variables can’t be declared as transient.
- During serialization, Object’s transient variables are not serialized.

Volatile :

- volatile is a access modifier
- Transient and volatile can not come together.
- Volatile is used in multi-processor environments.
- Volatile applies to only variables.
- Volatile can applied to static variables.
- Volatile can not be applied to final variables.

native :

- native is a access modifier
- Native applies to only to methods.
- Native can be applied to static methods also.
- Native methods can not be abstract.
- Native methods can throw exceptions.
- Native method is like an abstract method.
- The implementation of the abstract class and native method exist some where else, other than the class in which the method is declared.

Synchronized :

- synchronized is a access modifier
- Synchronize keyword is used to control the access to critical code in multi-threaded programming.
- Synchronized keyword can be applied to methods or parts of the methods only.

What is Inheritance?

Inheritance is the concept of a child class (sub class) automatically inheriting the variables and methods defined in its parent class (super class).

A primary feature of object-oriented programming along with encapsulation and polymorphism.

Why Inheritance?

Benefits of Inheritance in OOP : Reusability

1. Once a behavior (method) is defined in a super class, that behavior is automatically inherited by all subclasses

- Thus, you write a method only once and it can be used by all subclasses.

2. Once a set of properties (fields) are defined in a super class, the same set of properties are inherited by all Subclasses.

- A class and its children share common set of properties.

3. A subclass only needs to implement the differences between itself and the parent.

How to derive a sub-class?

- To derive a child class, we use the extends keyword.
- Suppose we have a parent class called Person.

```
public class Person {
protected String name;
protected String address;
/**
* Default constructor
*/
public Person(){
   System.out.println(“Inside Person:Constructor”);
   name = ""; address = "";
}
. . . .
}
```

- Now, we want to create another class named Student
- Since a student is also a person, we decide to just extend the class Person, so that we can inherit all the properties and methods of the existing class Person.
- To do this, we write,

```
public class Student extends Person {
      public Student(){
        System.out.println(“Inside Student:Constructor”);
      }
    . . . .
}
```

What You Can Do in a Sub-class

- A subclass inherits all of the “public” and “protected” members (fields or methods) of its parent, no matter what package the subclass is in.

- If the subclass is in the same package as its parent, it also inherits the package-private members (fields or methods) of the parent

What You Can Do in a Sub-class Regarding Fields

- The inherited fields can be used directly, just like any other fields.
- You can declare new fields in the subclass that are not in the super class
- You can declare a field in the subclass with the same name as the one in the super class, thus hiding it (not recommended).
- A subclass does not inherit the private members of its parent class. However, if the super class has public or protected methods for accessing its private fields, these can also be used by the subclass.

What You Can Do in a Sub-class Regarding Methods

- The inherited methods can be used directly as they are.
- You can write a new instance method in the subclass that has the same signature as the one in the super class, thus overriding it.

- You can write a new static method in the subclass that has the same signature as the one in the super class, thus hiding it.

- You can declare new methods in the subclass that are not in the super class.

Object Class

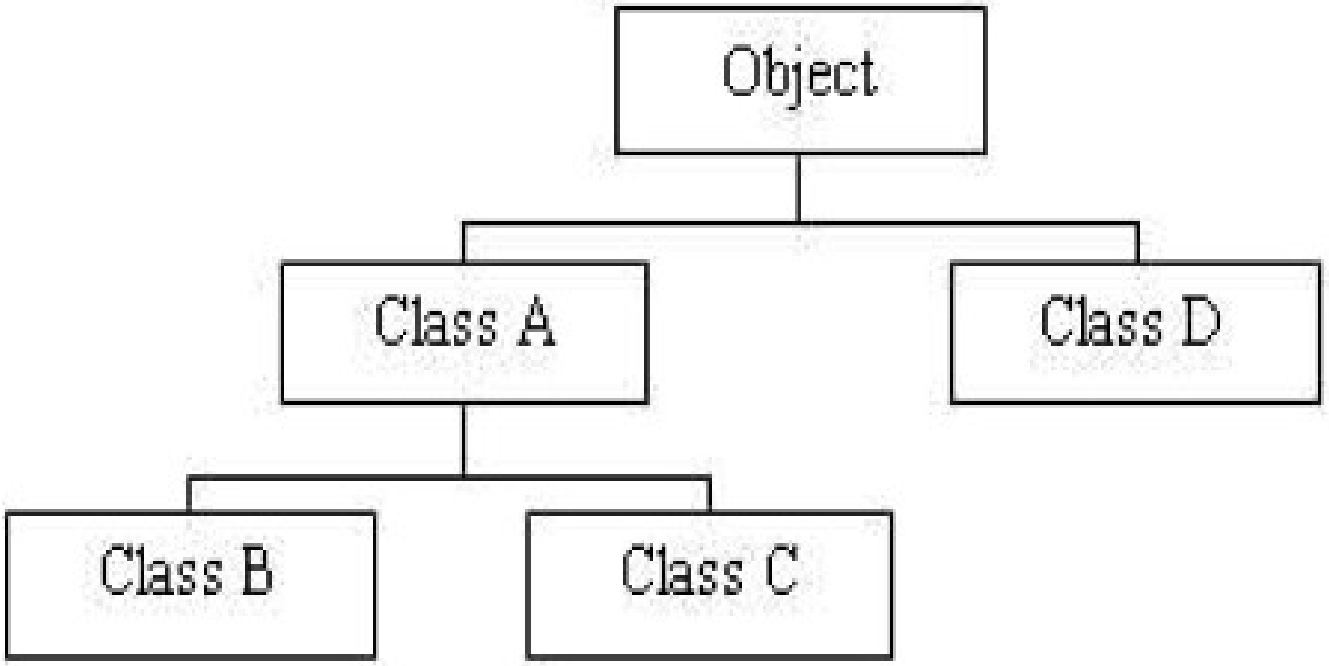
- Object class is mother of all classes
  – In Java language, all classes are sub-classed (extended) from the Object super class
  – Object class is the only class that does not have a parent class
- Defines and implements behavior common to all classes including the ones that you write.

The methods in Object class as follows.

Void finalize( )	String toString( )	Boolean equals( )
Object clone( )	final void notify( )	Int hashCode( )
final void notify( )	Final Class getClass( )	final void wait( )

Class Hierarchy

A sample class hierarchy.



Super class & Sub class

- Super class (Parent class)
  - Any class above a specific class in the class hierarchy.
- Sub class (Child class)
  - Any class below a specific class in the class hierarchy.

Constructor Calling Chain

How Constructor method of a Super class gets called

- A subclass constructor invokes the constructor of the super class implicitly
  - When a **Student** object, a subclass (child class), is instantiated, the default constructor of its super class (parent class), **Person** class, is invoked implicitly before sub-class's constructor method is invoked
- A subclass constructor can invoke the constructor of the super explicitly by using the “super” keyword
  - The constructor of the Student class can explicitly invoke the constructor of the Person class using “super” keyword
  - Used when passing parameters to the constructor of the super class

```java
package com.altisource.examples;

public class Person {

	public Person(){

		System.out.println("Inside
		    Person:Constructor");
	}
}
```

```java
package com.altisource.examples;

public class Student extends Person{

	public Student(){

		System.out.println("Inside
		   Student:Constructor");

	}

}
```

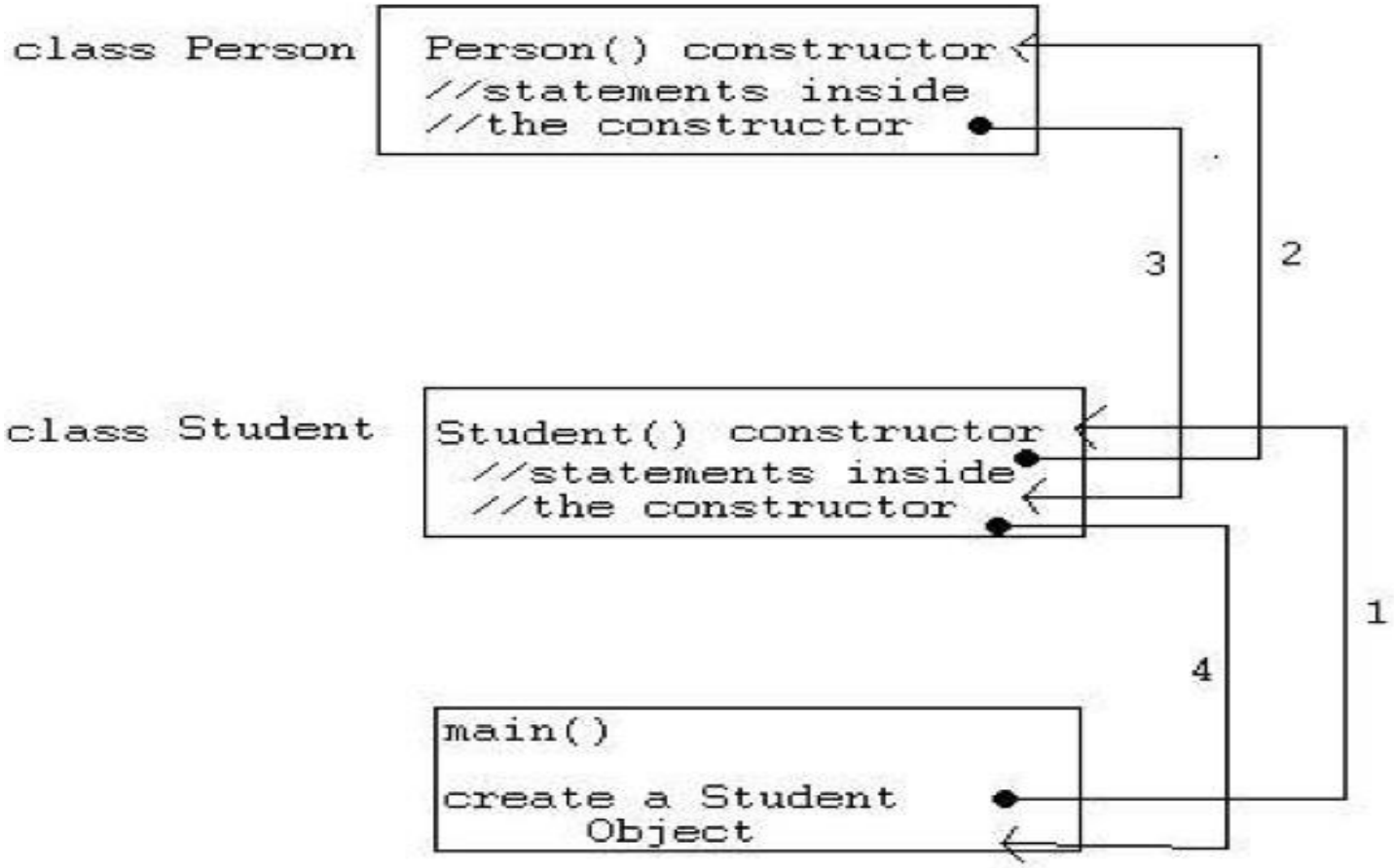
```java
package com.altisource.examples;
public class TestConstructorCall {

	public static void main(String[] args) {
		Student student = new Student();
	}

}
```

The output of the program is,

Inside Person:Constructor

Inside Student:Constructor

The program flow is shown below.



super keyword:

- A subclass can also explicitly call a constructor of its immediate super class.
- This is done by using the super constructor call.
- A super constructor call in the constructor of a subclass will result in the execution of relevant constructor from the super class, based on the arguments passed.

For example, given our previous example classes

Person and Student, we show an example of a super constructor call.

Given the following code for Student,

```
public Student(){
    super( "SomeName", "SomeAddress" );
    System.out.println("Inside Student:Constructor");
}
```

Few things to remember when using the super constructor call:

- The super() call must occur as the first statement in a constructor
- The super() call can only be used in a constructor (not in ordinary methods)

Another use of super is to refer to members of the super class (just like the this reference).

- For example,

```
public Student() {
        super.name = “somename”;
        super.address = “some address”;
}
```

Overriding Methods

1. The overriding method has the same name, number and type of parameters, and return type as the method it overrides.
2. The access level can't be more restrictive than the overridden method's.
3. The access level CAN be less restrictive than that of the overridden method.
4. Instance methods can be overridden only if they are inherited by the subclass. A subclass within the same package as the instance's superclass can override any superclass method that is not marked private or final.
5. A subclass in a different package can override only those non-final methods marked public or protected (since protected methods are inherited by the subclass).
6. The overriding method CAN throw any unchecked (runtime) exception, regardless of whether the overridden method declares the exception.

7. The overriding method must NOT throw checked exceptions that are new or broader than those declared by the overridden method.
8. You cannot override a method marked final.
9. You cannot override a method marked static.

Example:

```
public class Animal {
 public void eat() {
   System.out.println(“Animal eat method”);
}
 public void printYourself() {
   System.out.println(“Printing Animal information..”)
 }
}
```

```
class Horse extends Animal {
public void printYourself() {
        super.printYourself(); // Invoke the superclass (Animal) code
                                // Then do Horse-specific print work here
        System.out.println("Horse information printing");
  }
}
Public class TestOverriding {
  public static void main(String[] args) {
        Horse horse = new Horse ();
        horse. printYourself();
  }
}
```

Hiding Methods:

- If a subclass defines a class method (static method) with the same signature as a class method in the super class, the method in the subclass “hides” the one in the super class.

Example: Coding of Hiding Static Method

```
public class Animal {
        public static void testClassMethod() {
            System.out.println("The class method in Animal.");
        }
}
  // The testClassMethod() of the child class hides the one of the
 // super class – it looks like overriding, doesn't it? But
 // there is difference. We will talk about in the following slide.

public class Cat extends Animal {
        public static void testClassMethod() {
            System.out.println("The class method in Cat.");
        }
}
```

```
class Animal {
  static void doStuff() {
    System.out.print("a ");
  }
}
class Dog extends Animal {
  static void dostuff() {          // it's a redefinition,
                                   // not an override
    System.out.print("d ");
  }
  public static void main(String [] args) {
    Animal [] a = {new Animal(), new Dog(), new Animal()};
    for(int x = 0; x < a.length; x++)
      a[x].doStuff();                // invoke the static method
  }
}
```

Running this code produces the output:

```
a a a
```

Overriding Method vs. Hiding Method

- Hiding a static method of a super class looks like overriding an instance method of a super class
- The difference comes during runtime
  - When you override an instance method, you get the benefit of run-time polymorphism
  - When you override an static method, there is no runt-time polymorphism

Dynamic dispatch:

- Dynamic dispatch is a mechanism by which a call to Overridden function is resolved at runtime rather than at Compile time , and this is how Java implements Run time Polymorphism.

Final Class & Final Method

Final Classes

– Classes that cannot be extended

– To declare final classes, we write,

```
public final ClassName{
   . . .
}
```

Example:

- Other examples of final classes are your wrapper classes and String class
  – You cannot create a subclass of String class

Final Methods

– Methods that cannot be overridden

– To declare final methods, we write,

public final [returnType] [methodName]([parameters]){

. . .

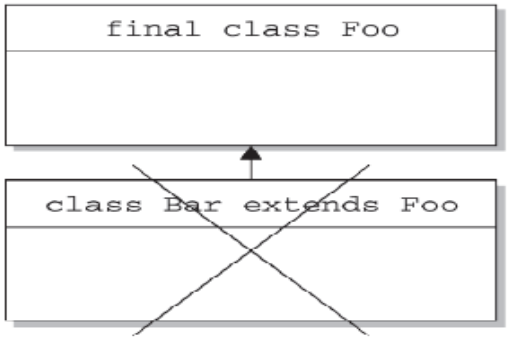
}

● Static methods are automatically final

final variables:

The final keyword applied to a variable makes it impossible to reinitialize a variable once it has been assigned a value.

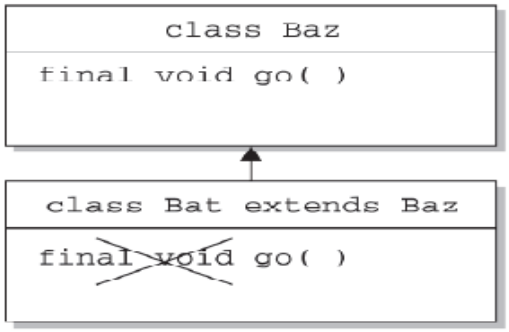
final class



final *class* cannot be subclassed

---

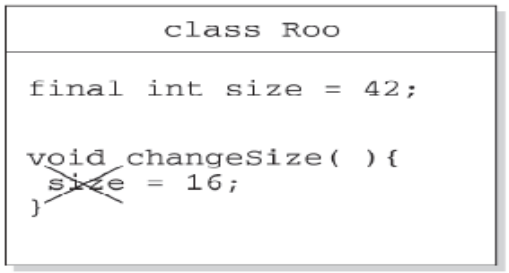
final method



final *method* cannot be overridden by a subclass

---

final variable



final *variable* cannot be assigned a new value, once the initial method is made (the initial assignment of a value must happen before the constructor completes).

Polymorphism:

- The ability to take more that one form.

Two kind of polymorphism.

1. Compile time Polymorphism
2. Run time Polymorphism

Compile time Polymorphism:

This can be achieved by Method overloading.

What is Method overloading?

- When a method in a class having the “same method name with different arguments (diff Parameters or Signatures) is said to be Method Overloading”.

Run time Polymorphism

This can be achieved by **Method Overriding**.

What is Method Overriding?

- When a method in a Class having “same method name with same arguments is said to be Method overriding”.
- “Providing a different implementation of a method in a subclass” of the class that originally defined the method.

Difference b/w Method overloading and Method Overriding?

1. In Over loading there is a relationship between the methods available in the same class, where as in Over riding there is relationship between the Super class method and Sub class method.

2. Overloading does not block the Inheritance from the Super class, Whereas in Overriding blocks Inheritance from the Super Class.

3. In Overloading separate methods share the same name, where as in Overriding Sub class method replaces the Super Class.

4. Overloading must have different method Signatures , Where as Overriding methods must have same Signatures.

Quiz:

```
public class Animal {
public void eat() {
        System.out.println("Generic Animal Eating
                Generically");
    }
}
public class Horse extends Animal {
    public void eat() {
        System.out.println("Horse eating hay ");
    }
    public void eat(String s) {
        System.out.println("Horse eating " + s);
    }
}
```

Method Invocation Code	Result
`Animal a = new Animal();` `a.eat();`	`Generic Animal Eating Generically`
`Horse h = new Horse();` `h.eat();`	`Horse eating hay`
`Animal ah = new Horse();` `ah.eat();`	`Horse eating hay` Polymorphism works—the actual object type (Horse), not the reference type (Animal), is used to determine which eat() is called.
`Horse he = new Horse();` `he.eat("Apples");`	`Horse eating Apples` The overloaded eat(String s) method is invoked.
`Animal a2 = new Animal();` `a2.eat("treats");`	Compiler error! Compiler sees that Animal class doesn't have an eat() method that takes a String.
`Animal ah2 = new Horse();` `ah2.eat("Carrots");`	Compiler error! Compiler *still* looks only at the reference, and sees that Animal doesn't have an eat() method that takes a String. Compiler doesn't care that the actual object might be a Horse at runtime.

```
public class Foo {
void doStuff() { }
}
class Bar extends Foo {
void doStuff(String s) { }
}
```

Don’t be fooled by a method that’s overloaded but not overridden by a subclass. It’s perfectly legal .

- The Bar class has two doStuff() methods: the no-arg version it inherits from Foo (and does not override), and the overloaded doStuff(String s) defined in the Bar class. Code with a reference to a Foo can invoke only the no-arg version, but code with a reference to a Bar can invoke either of the overloaded versions.

```
class Clidder {
        private final void flipper() {
                System.out.println("Clidder");
    }
}
public class Clidlet extends Clidder {
  public final void flipper() {
        System.out.println("Clidlet");
  }
 public static void main(String [] args) {
        new Clidlet().flipper();
 }
}
```

Output: Clidlet

- Although a final method cannot be overridden, in this case, the method is private, and therefore hidden.