Technical Note No. 105
Understanding Java Inheritance

Published: May 2, 2008. Last reviewed on January 4, 2020
By Daryl Close, Professor of Computer Science and Philosophy, Heidelberg University, Tiffin, Ohio

Summary: This note explains the concept of Java inheritance.

Java inheritance must be stated very precisely. First, only *members* of a class can be inherited. Second, whether a member is in fact inherited depends on the circumstances of that specific member. These details are defined in *The Java Language Specification*[1] (JLS).

Let's start with the concept of a member of a class. The JLS states that members of a class type are "fields and methods and nested classes and interfaces" (*JLS13* 207). So, we know that constructors are not inherited because they're not members of the class.

However, not even all fields and methods in the parent class are inherited by the child. A child class inherits all *accessible* members of the parent class, viz., interfaces, fields and methods of the parent class. Note the emphasis on the term "accessible." Consequently, private fields are *not* inherited, as is clearly stated by the creators of the Java language:

> A class inherits from its direct superclass and direct superinterfaces all the nonprivate fields of the superclass and superinterfaces that are both accessible to code in the class and not hidden by a declaration in the class.
>
> A private field of a superclass might be accessible to a subclass - for example, if both classes are members of the same class. Nevertheless, a private field is never inherited by a subclass. (*JLS13* 8.3, 231)

Again, in the JLS we read

> Members of a class that are declared `private` are not inherited by subclasses of that class. Only members of a class that are declared `protected` or `public` are inherited by subclasses declared in a package other than the one in which the class is declared. . . . Constructors, static initializers, and instance initializers are not members and therefore are not inherited. (*JLS13* 8.2, 225)

This is a source of much confusion in the Java textbook market where authors regularly misstate Java inheritance rules, e.g., as "all instance fields and methods," or "all fields and methods." Here are some examples:

- When one class inherits from another, it gets all the methods and fields from that parent class.[2]
- [t]he derived class inherits all the public methods, all the public and private instance variables, and all the public and private static variables from the base class . . .[3]
- When you extend a class, the new class inherits the superclass's members—though the private superclass members are *hidden* in the new class.[4]

This lack of precision is confusing to the Java student on several levels. First, in good data encapsulation design, the most common accessibility modifier for instance fields is `private`. The confusion

---

[1]Gosling, James, Bill Joy, Guy Steele, and Gilad Bracha. (1996) 2019. *The Java Language Specification SE 13 Edition* (hereafter "*JLS13*" with section and page numbers). This edition and previous editions of the JLS are available in HTML and PDF formats at http://docs.oracle.com/javase/specs/#21831. The JLS language cited in this technical note remains unchanged throughout the various editions of the JLS. For example, see Gosling, James, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. 2014. *The Java Language Specification Java SE 8 Edition.* Upper Saddle River, N. J.: Addison-Wesley.

[2]Guzdial, Mark, and Barbara Ericson. 2007. *Introduction to Computing & Programming with Java: A Multimedia Approach*. Upper Saddle River, N. J.: Pearson Education, Inc., 348.

[3]Savitch, Walter. 2016. *Absolute Java*. 6th ed. Boston: Pearson Education, Inc., 435.

[4]Deitel, Paul, and Harvey Deitel. 2012. *Java: How to Program*. 9th ed. Upper Saddle River, NJ: Pearson Prentice-Hall, 386.

here is compounded by the fact that inheritance brings about memory allocation for the parent's private instance fields as well as for those that have some level of visibility outside of the class. If a child class wants to access the private fields of its parent class, it must do so like everybody else, viz., through getter and setter methods—if any—in the parent class.[5] Inheritance is about access and polymorphism, not memory allocation, per se. In fact, some Java experts argue that static methods are not inherited because they are not overridable—even though nothing in the JLS suggests that nonprivate static methods are not inherited.

The second area of confusion is the phenomenon of the hiding of fields (both static and instance fields) and hiding of static methods. Hiding interrupts inheritance even if a field or a static method is not private because it makes the hidden member in the parent inaccessible. The JLS provides the following example of instance field hiding:

```
class Point {
   int x = 2;
}
class Test extends Point {
   double x = 4.7;
   void printBoth() {
      System.out.println(x + " " + super.x);
   }
   public static void main(String[] args) {
      Test sample = new Test();
      sample.printBoth();
      System.out.println(sample.x + " " + (Point)sample).x);
   }
}
```

> . . . because the declaration of x in class Test hides the definition of x in class Point, . . . class Test does not inherit the field x from its superclass Point. It must be noted, however, that while the field x of class Point is not inherited by class Test, it is nevertheless implemented by instances of class Test. In other words, every instance of class Test contains two fields, one of type int and one of type double. Both fields bear the name x, but within the declaration of class Test, the simple name x always refers to the field declared within class Test. (*JLS13* 8.3.3.2, 236)

A third area of confusion for students involves constructors. Although constructors aren't inherited, inheritance affects the parent class constructors. The key to understanding constructors in a child class is knowing what happens when we create an object of a child class type. Object creation via the `new` operator causes the child class constructor to be called as usual. This in turn causes the parent class default constructor to be called. The reason this second, implicit call to the parent class constructor occurs is because the child class needs to initialize any parent class instance variables that have implemented.

Normally, we don't want this second step to occur behind the scenes. Instead, we want to have our child class constructor call the parent class constructor *explicitly*. This is done with `super()` or `super([parameter list])`. This invocation of a parent class constructor must occur in the first line of the child class default constructor. The invocation is chained to each ancestor class all the way to the `Object` class. This is called constructor chaining.

Finally, in Java, a child class may inherit from just one parent class. Multiple inheritance of classes is not permitted in Java. A class may implement multiple interfaces, however.

[5]Among leading CS1 Java textbooks, Liang is among the most careful: "A subclass inherits accessible data fields and methods from its superclass . . ." See Liang, Y. Daniel. 2013. *Introduction to Java Programming, Comprehensive Version*. 9th ed. Upper Saddle River, NJ: Pearson Prentice-Hall, 408.