# This and Super

When writing code in the instance (or non-static) context of a class, the keyword **this** can be used to signify the instance of the class currently executing that code.

For example, the following code snippet defines a class Human that has a field age and a constructor that initializes the age field:

```
public class Human {
     public int age;
     ...
     public Human(int age) {
          this.age = age;
     }
}
```

The parameter age shadows the field age in the scope of the constructor. If instead we were in a context outside of Human, say in class World, and we had a reference to an instance of Human, we could simply access the field age using the `.' operator:

```
class World {
     ...
     void someMethod() {
          Human h = new Human(5);
          System.out.print(h.age); //prints 5
     }
}
```

However, inside the Human class context we do not have a handle on that reference. We know we are executing code for an instance, but how can we reference that instance? How can the object reference itself? Hence the keyword **this**.

The keyword **super** references the instance's superclass. Note that **super** and **this** can be used to access methods as well as fields. For example:

```
public class Human {
     private String name;
     ...
     public Human(String name) {
          this.name = name;
     }

     public String getName() {
          return String;
     }
}

...

public class Student extends Human {
     private String username;

     public Student(String name, String username) {
          super(name);
          this.username = username;
```

```
    }

    public String getName() {
        return username;
    }

    public String getRealName() {
        return super.getName();
    }
}

...

public class World {
    ...
    void someMethod() {
        Student alice = new Student("Alice", "abc");
        System.out.println(alice.getRealName()); // what gets printed?
```

In the above code snippet a Student is a Human with an extra field, username. Student overrides Human's getName method to return the username instead of name. However, the Human name can be accessed through another method, getRealName(), which calls the super's method, getName(). Thus, "Alice" is printed in World's someMethod(). If Human's name field were protected, then getRealName could return super.name, or simply name (since name has not been shadowed by a local variable or field in Student, the compiler knows to use the field in Human). The important point is that failure to use **super** will call the Student's getName() method, which would incorrectly print "abc".

Use **super** and **this** when you have to access fields and methods of the instance. However, when no shadowing occurs, for example in Student's getName() method, **this** is implicit and can be left off.

Finally, note the use of **super** in Student's constructor to call Human's constructor. The first line of a subclass's constructor must call the superclasses constructor. We wouldn't want the Student to have to initialize the Human's fields, so this makes sense.

Classes can contain multiple constructors, so use **this** to access a constructor of the same class.

```
public class Student extends Human {
    private String username;

    public Student(String name, String username) {
        super(name);
        this.username = username;
    }

    public Student(String name) {
        this(name, name);
    }

    public Student() {
        this("Anonymous");
    }
}
```

In the above code snippet, Student has three constructors: one that takes a name and username, one that takes a name, and one that takes nothing. Remember, when overloading methods, the methods have the

same name but different parameters.

Suppose we instantiate a student using the constructor that takes a single argument:

```
Student sting = new Student("Sting");
```

Execution first enters the single argument constructor, which calls the double argument constructor with "Sting" as both parameters. Thus, a Student is instantiated with name and username both equal to Sting.

Likewise, `Student anon = new Student();` constructs a student with "Anonymous" as both the username and name.