Inheritance

Sources:

Horstmann 9.1-9.5

Summary:

Inheritance is a fundamental concept of **object-oriented programming (OOP).** It enables classes to inherit properties from a **parent** classes and provide them to **child** classes. This creates a hierarchy topped by the **Object** class, which provides basic methods such as **toString()** and **clone()**.

Inheritance creates **is-a** relationships between classes. For example, a Dog class can inherit from a Mammal class. This means that the Dog class *is* a Mammal. In this case, the Dog is the child of the Mammal. All non-private fields and methods of the parent class are passed down to the child class. The child class may have fields and methods of its own. For example, the Mammal class might have the *height* and *weight* fields. The Dog class can also have these fields, as they can be inherited from Mammal. It can also have fields unique to itself, such as a *breed* field.

In Java, the syntax for inheritance is as follows:

public class Dog extends Mammal

The **extends** keyword is used to specify what to inherit from. Note that Java does not support **multiple inheritance**- it is not possible to inherit from more than one parent. Of course, however, many different classes can inherit from the same parent. It is also possible to chain together many levels of inheritance. Dog inherits from Mammal, which inherits from Animal, and so forth.

When a child class is constructed, the parent's default no argument constructor is called first, unless the super(…) constructor is invoked with arguments. This call MUST be placed in the first line of the child's constructor or it WILL fail. The only exception is in the case when a default constructor is available- the compiler will automatically insert a super() call at the start of the child's constructor.

All methods and fields can be redefined. For example, both the parent and child may define a foo() function. They will behave as expected within the parent and child classes. Parents CANNOT access the methods of their children. A child CAN still access its parent's non-private functions, however, by using the **super** prefix. super.foo() will call the parent's foo() function, rather than the child's. Note that super.super.foo() is an invalid function call. Instead, give the parent a function that calls its own parent's function. By "chaining" such functions, a child can go up many levels of inheritance at once.

Example code:

```
class SuperParent
{
	public SuperParent()
	{
		System.out.println("Superparent constructed");
	}

	public void foo()
	{
		System.out.println("Success!");
	}
}

class Parent extends SuperParent
{
	public Parent(String input)
	{
		//The compiler automatically inserts a super() call here
		System.out.println("Parent constructed");
		System.out.println(input);
	}

	public void foo()
	{
		System.out.println("Chaining to superparent...");
		super.foo();
	}
}

class Child extends Parent
{
	public Child()
	{
		super("Parameter passed successfully.");
		System.out.println("Child constructed");
		this.foo();
	}

	public void foo()
	{
		System.out.println("Chaining to parent...");
		super.foo();
	}
}
```

Output upon constructing Child:

```
Superparent constructed
Parent constructed
Parameter passed successfully.
Child constructed
Chaining to parent...
Chaining to superparent...
Success!
```