Abstract Classes

Abstract classes and methods, concrete classes, instantiation, restrictions, interface implementation

Abstract methods and classes

The purpose of an **abstract class** is to hold common data and methods and to declare behaviors that extending classes must actually define. It is not meant to be instantiated.

- An **abstract method** in a class is declared but not defined. It must be declared with the keyword **abstract**.
- A class that has an abstract method is an **abstract class**. It must be declared with the keyword **abstract**.
- A class that is not abstract is **concrete**.

Example:

```
public abstract class Product {
//…
        public abstract String instructions();
```

The Product abstract class

```
public abstract class Product {
    private String barCode;
    private double price;

   public Product(String barCode) {
        this.barCode = barCode;
    }

    public abstract String instructions();

    public String getBarCode() { return barCode;  }
    public double getPrice() { return price; }
    public void setPrice(double price) {
      this.price = price;
   }
}
```

Abstract rules

1. A class that contains abstract methods must be declared abstract.
2. Abstract classes cannot be instantiated. (But they do have constructors.)
3. A class that does not contain abstract methods can also be declared abstract. (That makes it impossible to create instances of the class.)
4. Every concrete class that extends an abstract class must define all abstract methods of the parent.

Concrete class example: Shampoo

```
public class Shampoo extends Product {
  public Shampoo(String barCode) {
     super(barCode); // Call abstract superclass constructor
  }

  @Override
  public String instructions() {
     return "Wet hair and apply 1 tbsp shampoo. \n"
                + " Lather and rinse.";
  }
}
```

Notes:

1. Constructor calls superclass constructor. If there is no such explicit call, there is an implicit call to the superclass null constructor.
2. The abstract method is defined here. This class is concrete.

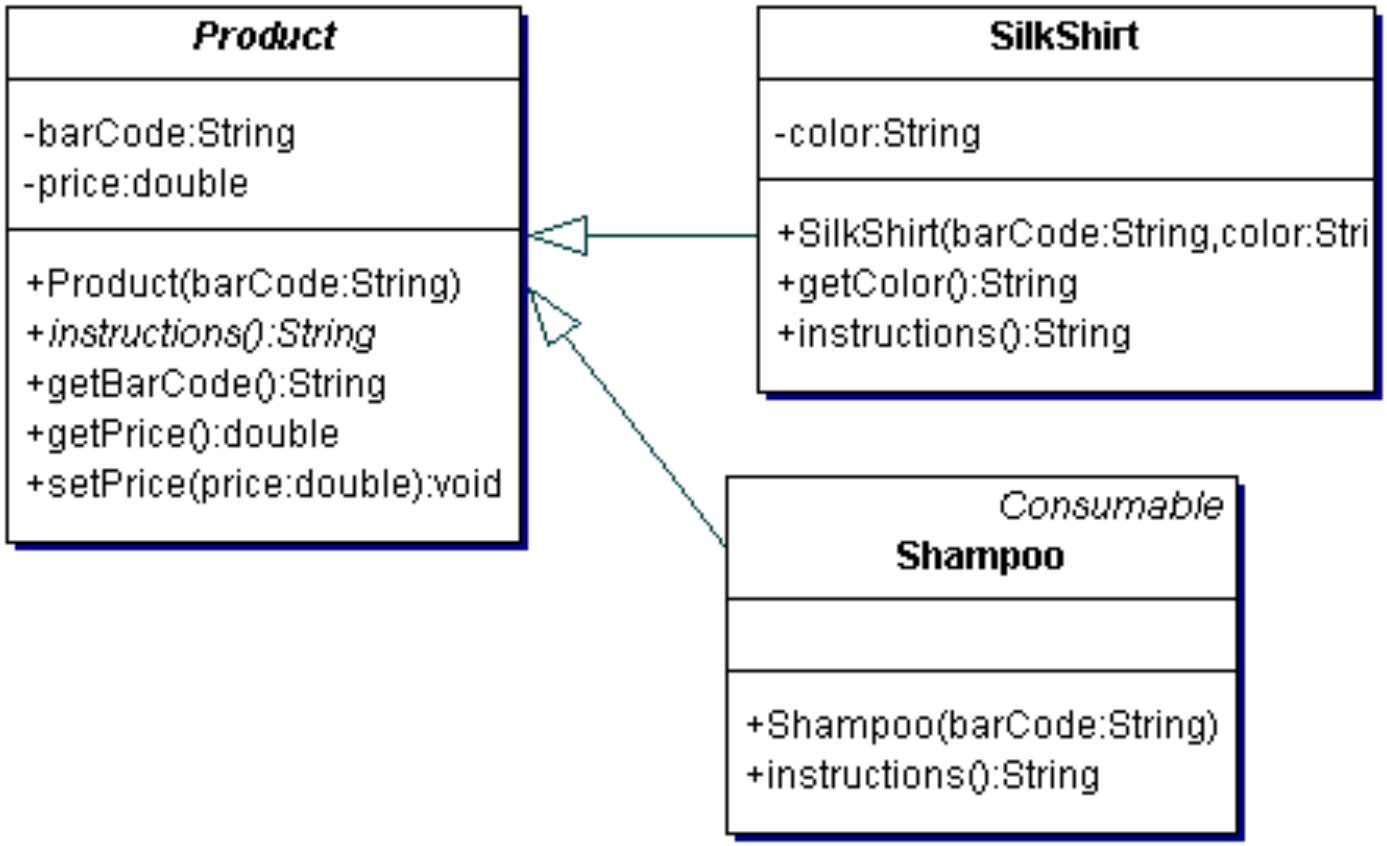
Concrete class example: SilkShirt

```
public class SilkShirt extends Product {
    public SilkShirt(String barCode, String color) {
        super(barCode);  //Calls abstract superclass constructor
        this.color = color;
    }

    @Override
    public String instructions() {
        return "Dry clean only.";
    }

    public String getColor(){ return color; }
}
```

Product class hierarchy



Shampoo and SilkShirt are both concrete classes.

Declaring variables of abstract type

```
public class ProductTester {
   private Product myProduct;  // Abstract type declaration

   public ProductTester(int kind) {
     if (kind == 0)
        myProduct = new Shampoo("0");
     else
        myProduct = new SilkShirt("1", “Green”);
   }

   public void printInstructions() {
     System.out.println(myProduct.instructions());
   }
}
```

… continued from previous page

```
// …

ProductTester productX = new ProductTester(0);
ProductTester productY = new ProductTester(1);

productX.printInstructions();
productY.printInstructions();
```

The output

```
Wet hair and apply 1 tbsp shampoo.
    Lather and rinse.
Dry clean only.
```

Casting with abstract classes

```
Product p = new SilkShirt("1", "GREEN");
// System.out.println(p.getColor());  ← Illegal

System.out.println( ((SilkShirt) p).getColor());
```

The output:

```
GREEN
```

Changing a design mistake

Shampoo is a consumed product, unlike SilkShirt. So we could declare an abstract base class, CanBeConsumed.

```
public abstract class CanBeConsumed {
   public abstract void beConsumed(double amount);
   public abstract double getAmountLeft();
}
```

But then we have a problem since a class cannot have two different parents. This code is illegal:

```
public class Shampoo extends Product, CanBeConsumed //NO
```

So use a Consumable interface instead

```
public interface Consumable {
    void consume(double amount);
    double amountLeft();
}
```

And now you can declare Shampoo this way:

```
public class Shampoo extends Product implements Consumable {
```

If Shampoo were to implement an additional interface, say Another, then it would be declared thus:

```
public class Shampoo extends Product
                    implements Consumable, Another {
```

Shampoo definition

```
public class Shampoo extends Product implements Consumable {

private double amount;
    public Shampoo(String barCode) {
        super(barCode);
        amount = 10.5;
    }
    public String instructions() {
        return "Wet hair and apply 1 tbsp shampoo.\n"
                + "Lather and rinse.";
    }

    public void consume(double amount){
        if (amount <= this.amount)
          this.amount -= amount;
    }
    public double amountLeft(){ return amount; }
}
```

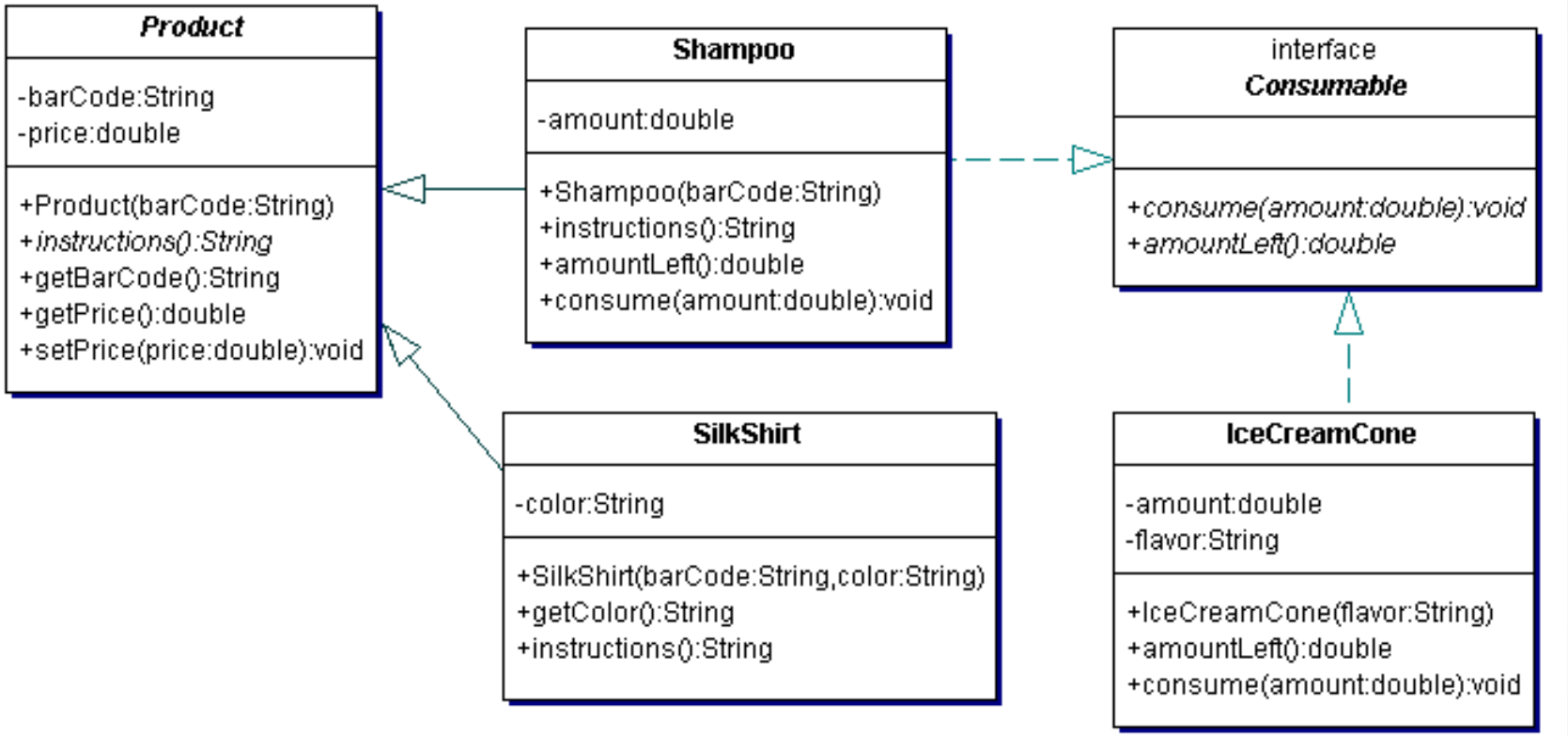
IceCreamCone definition

Consumable can be implemented classes outside the Product hierarchy.

```
public class IceCreamCone implements Consumable {
    private double amount;
    private String flavor;

    public IceCreamCone(String flavor) {
        this.flavor = flavor;
        amount = 4;  // ounces
    }
    public void consume(double amount){
        this.amount -= amount;
        if (this.amount < 0)
          this.amount = 0;
    }
    public double amountLeft(){ return amount; }
}
```

UML Diagram


Product
-barCode:String
-price:double
+Product(barCode:String)
+instructions():String
+getBarCode():String
+getPrice():double
+setPrice(price:double):void
Shampoo
-amount:double
+Shampoo(barCode:String)
+instructions():String
+amountLeft():double
+consume(amount:double):void
interface
Consumable
+consume(amount:double):void
+amountLeft():double
SilkShirt
-color:String
+SilkShirt(barCode:String,color:String)
+getColor():String
+instructions():String
IceCreamCone
-amount:double
-flavor:String
+IceCreamCone(flavor:String)
+amountLeft():double
+consume(amount:double):void

Implementing an interface without defining some methods

- Any class that implements an interface but doesn't define all public methods declared in the interface must be abstract.
- The undefined methods must be declared abstract.

```
public abstract class X implements Consumable {
   public abstract void consume(double amount);
   public abstract double amountLeft();
     // ….  Other code
}
```

When are abstract classes useful?

- To group a related collection of classes.
- To define a general base type that is not meant to be instantiated (requires more information to be instantiated).
- Provide common attributes and behaviors.
- Force concrete child classes to spell out behaviors that should be customized to each child.