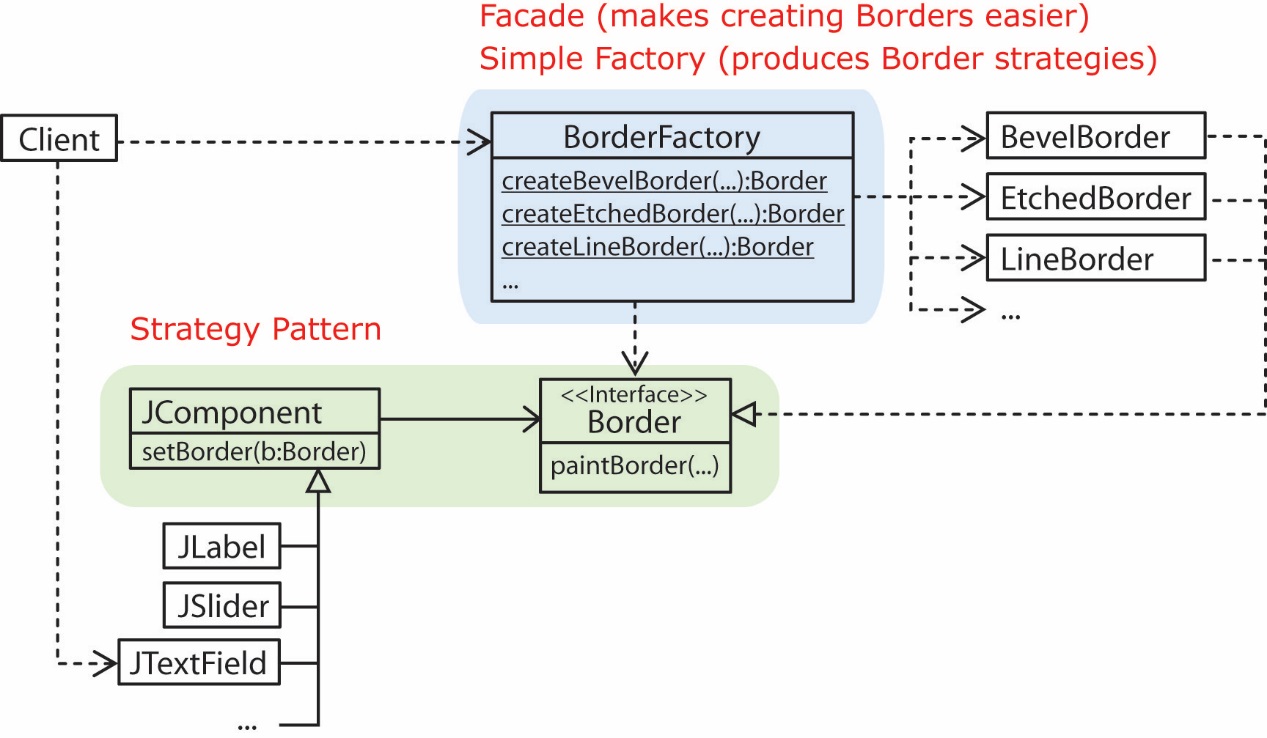
**Chapter 4 – Factory Method Pattern Notes**

**Simple Factory**

1. The *simple factory* is not really a pattern, it is just something you should do based on basic design principles. It simply says that when object creation takes more than a few lines we should wrap up that creation in its own class. Many times the simple factory has static methods for creating objects, although it doesn’t have to

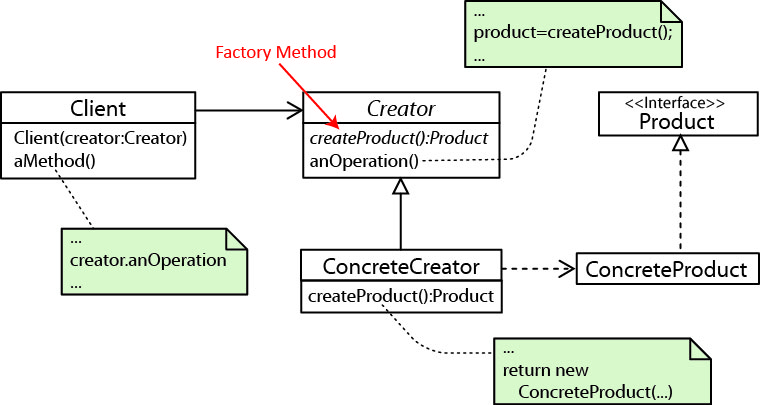
|  |  |
| --- | --- |
| E:\Data-Classes\CS 4322 - Software Engineering 2\Notes\04a-Factory Method\b1.jpg | E:\Data-Classes\CS 4322 - Software Engineering 2\Notes\04a-Factory Method\b2.jpg |
| E:\Data-Classes\CS 4322 - Software Engineering 2\Notes\04a-Factory Method\b3.jpg |  |

1. The *BorderFactory* is a simple factory found in the Java API which has static methods for creating various borders. It also serves as a Façade to the Borders. One advantage to using the *BorderFactory* is that it provides shared instances. In other words, if two components each require a *BevelBorder*, then by using the *BorderFactory* only one instance of *BevelBorder* is created. Two other simple factories are: *BasicIconFactory*, *ColorChooserComponentFactory*.



**Factory Method Pattern**

1. The factory method pattern is simply an abstraction of the simple factory which allows new products to be added without modifying the factory. By doing so, it adheres to the design principle of programming to an interface, not an implementation.
2. The Factory Method pattern defines an interface for creating an object, but lets subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses. The intent of the Factory Method pattern is to relieve a client from having to know which class to instantiate.



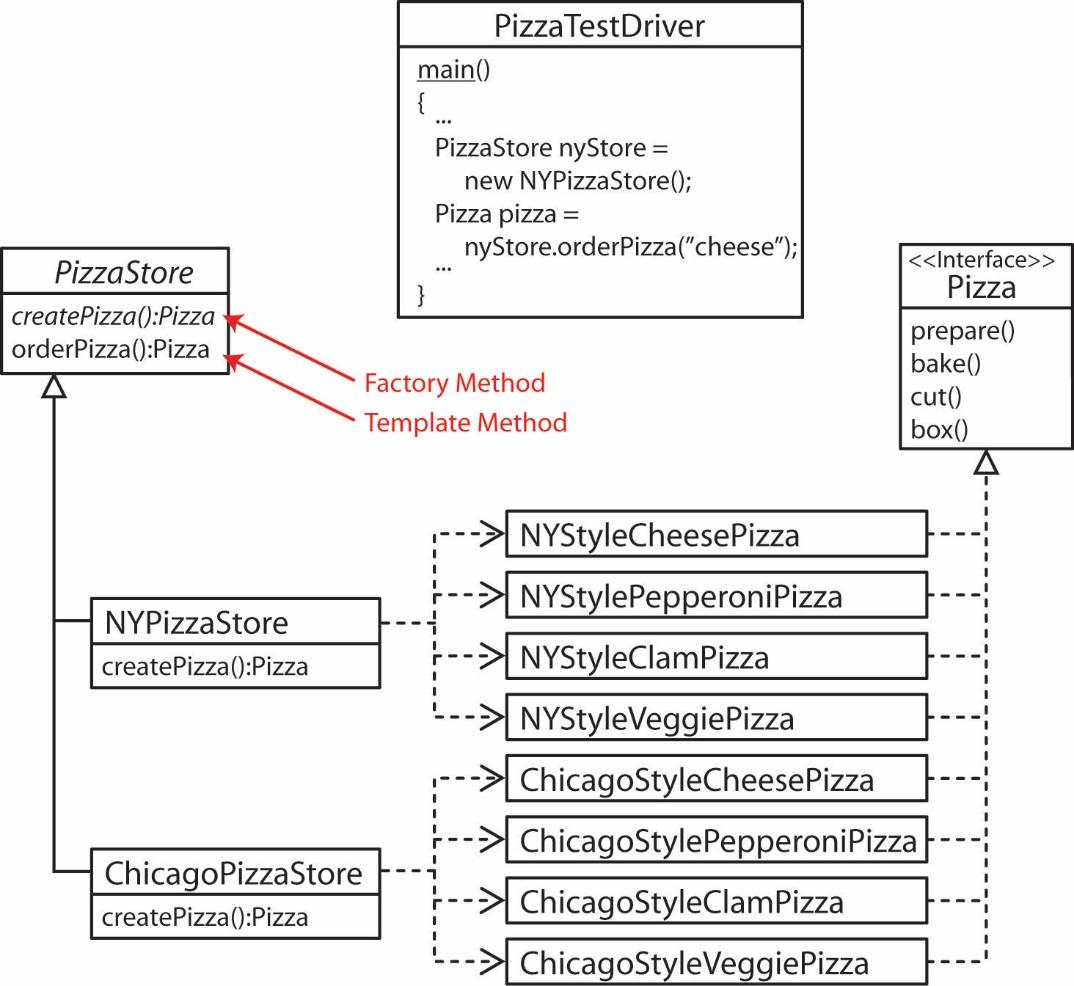
Usually, as shown in the UML above (and the example from the text), the Client calls a concrete method (*anOperation*) that that utilizes a factory method (*createProduct*) to create and product and manipulate it.

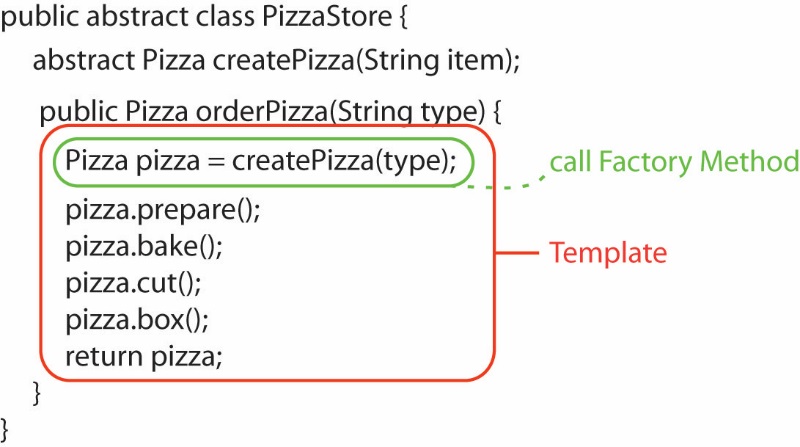
**Client c = new Client( new ConcreteCreator );**

**c.anOperation();**

In other cases, the Client calls the factory method directly in order to obtain a product.

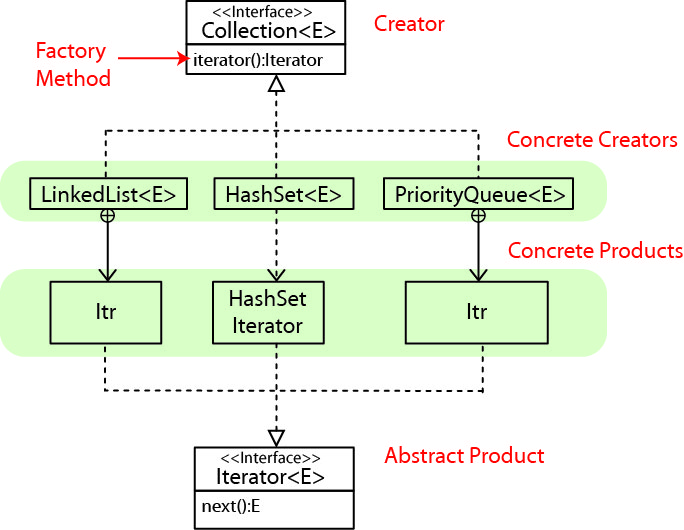
1. The example from the text shows that the driver calls the concrete *orderPizza* (template) method which creates a pizza with the factory method, *createPizza*, then builds the pizza, and returns it.





**Factory Method in Java**

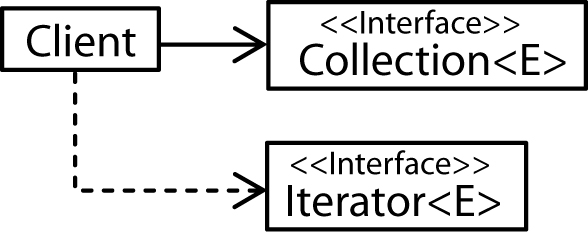
1. The *iterator* method specified in the Collection interface is a Factory Method. The *Collection* interface specifies that any class that wants to be a *Collection* must specify an *iterator* method that returns an instance that implements the *Iterator* interface.



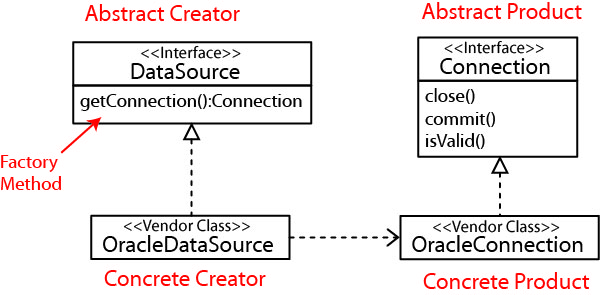
See the code in ArrayList class, lines 773, 780:

<http://www.docjar.com/html/api/java/util/ArrayList.java.html>

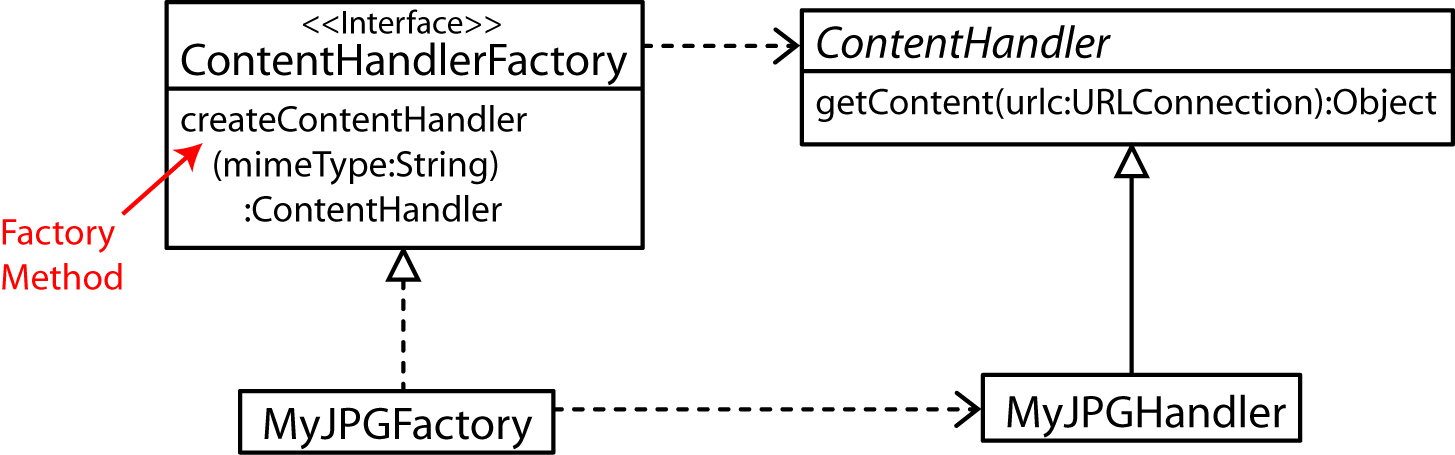
Clients of course can use the *Iterator* directly, but also the JVM utilizes an iterator when a client uses the enhanced for loop (for-each loop).



1. Java provides a *DataSource* interface that provides a specification for a Factory Method, *getConnection* for use in accessing data bases. Vendors supply classes that implement the *DataSource* interface, thus an implementation for *getConnection* which returns a *Connection* object for that particular Vendor’s product. Thus, clients can be coded against the Connection interface and not have to even be aware of the concrete connection class.



1. *ContentHandler’s* are used to get the content from the *URLConnection*. The Java interface *ContentHandlerFacotry* has a factory method, *createContentHandler* which returns a *ContentHandler* (abstract product) for a specified MIME type (*e.g. gif, text,* etc). The *ContentHandler* has a method which provides a *getContent* method which returns the content of a *URLConnection*.



1. The *TableCellRenderer* is a factory interface with a factory method: *getTableCellRedererComponent* used for drawing a cell in a *JTable*. The *DefaultTableCellRenderer* class implements this interface and is used for simple JTables. An [Example](http://www.java2s.com/Code/Java/Swing-JFC/Thisprogramdemonstratescellrenderingandeditinginatable.htm).

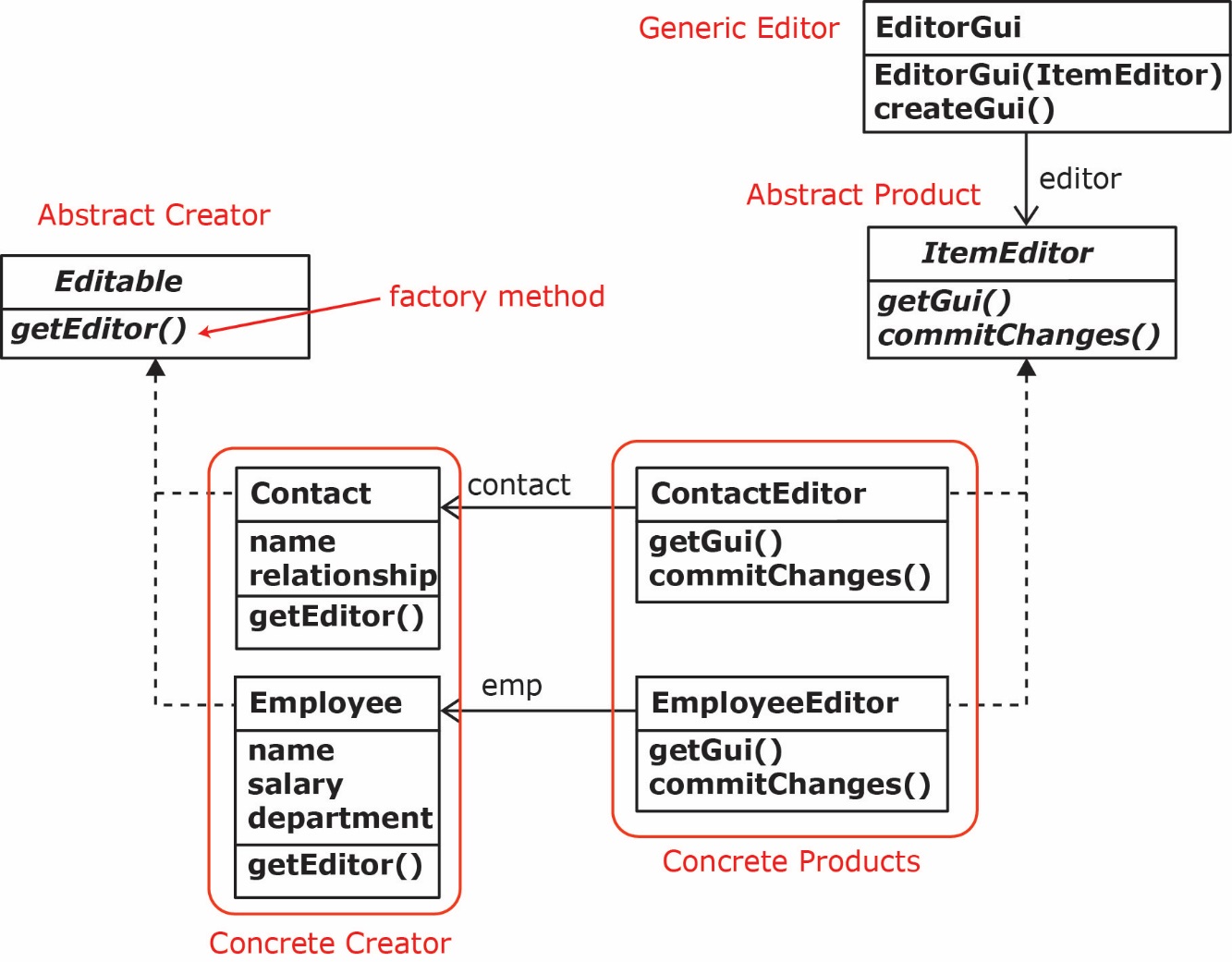
**Generic Editor Example**

1. Consider the Generic Editor that illustrates the Factory Method pattern as found in:

<http://www.java2s.com/Code/Java/Design-Pattern/FactoryMethodPatterninJava.htm>

|  |  |  |
| --- | --- | --- |
| One concrete factory produces a “Contact” editor: |  | Another concrete factory produces an “Employee” editor: |
|  |  |  |

1. Consider the design:



**Homework**

Suppose we have an abstract Shape class with subclasses: Triangle and Circle. We need a class with a method to draw a list of shapes. Drawing simply consists of looping through a list of shapes and displaying them on the console (toString). This method should accept a string that contains shape data. Sometimes we will use this class and method to parse the string to find all the triangles and display them. And, we can keep feeding different strings and displaying the triangles. Other times we need a different instance to display the circles. The string can contain any text and embedded within are triangles in the format: "triangle side1Len side2Len side3Len" and circles: "circle radius". For example, the string might look like this:

"When you are lost, a triangle 3.0 2.0 4.4 is a good thing to find. Often, though a circle 3.3 appears out of nowhere. Later you might find another triangle or a circle 5.0. Later you might drop a triangle 5.6 6.7 7.8 into a bucket."

This example had 2 triangles and 2 circles. Keep in mind that there could be any number. Also, that when parsing the string sometimes you are only interested in triangles. In another situation you are only interested in circles.

1. Use the factory method pattern to model this situation with a class diagram.
2. Write the code for all classes. Code for Shape, Triangle, and Circle provided below. Also supplied is code for doing the parsing. I supply all this code in case you want test out your design.
3. Write some code to show how this works.

**Code to do Triangle parsing**

// Does not handle case where data ends with a punctuation mark. For example:

// s="...triangle 1.0 2.0 3.0. Green eggs and ham." will fail to detect the triangle.

String s = "When you are lost, a triangle 3.0 2.0 4.4 is a good thing to find. " +

"Often, though a circle 3.3 appears out of nowhere. " +

"Later you might find another triangle or a circle 5.0 . " +

"Later you might drop a triangle 5.6 6.7 7.8 into a bucket.";

List<Shape> shapes = **new** ArrayList<>();

String[] tokens = s.split(" ");

**int** i=0;

**while**(i<tokens.length) {

**if**(tokens[i++].equals("triangle")) {

**double**[] sides = getSides(tokens,i);

**if**(sides!=**null**) {

shapes.add(**new** Triangle(sides[0],sides[1],sides[2]));

}

i+=3;

}

}

// Helper methods

**private** **boolean** has3MoreTokens(String[] tokens, **int** i) {

**return** (i+2)<tokens.length;

}

**private** **boolean** areNext3TokensDoubles(String[] tokens, **int** i) {

**for**(**int** j=i; j<i+3; j++) {

**try** {

Double.*parseDouble*(tokens[j]);

}

**catch**(NumberFormatException e) {

**return** **false**;

}

}

**return** **true**;

}

**public** **double**[] getSides(String[] tokens, **int** i) {

**if**(!has3MoreTokens(tokens, i)) **return** **null**;

**if**(!areNext3TokensDoubles(tokens,i)) **return** **null**;

**double**[] sides = **new** **double**[3];

**for**(**int** j=0; j<3; j++) {

sides[j] = Double.*parseDouble*(tokens[i++]);

}

**return** sides;

}

**Code to do Circle parsing**

// Does not handle case where data ends with a punctuation mark. For example:

// s="...circle 3.3. Green eggs and ham." will fail to detect the circle.

String s = "When you are lost, a triangle 3.0 2.0 4.4 is a good thing to find. " +

"Often, though a circle 3.3 appears out of nowhere. " +

"Later you might find another triangle or a circle 5.0 . " +

"Later you might drop a triangle 5.6 6.7 7.8 into a bucket.";

List<Shape> shapes = **new** ArrayList<>();

String[] tokens = s.split(" ");

**int** i=0;

**while**(i<tokens.length) {

**if**(tokens[i++].equals("circle")) {

Double radius = getRadius(tokens,i);

**if**(radius!=**null**) {

shapes.add(**new** Circle(radius));

}

i++;

}

}

// Helper method

**public** Double getRadius(String[] tokens, **int** i) {

**if**(i>=tokens.length) **return** **null**;

Double radius = **null**;

**try** {

radius = Double.*parseDouble*(tokens[i]);

}

**catch**(NumberFormatException e) {

}

**return** radius;

}

**Shape, Triangle, Circle classes**

**public** **abstract** **class** Shape {

**private** String color;

**protected** Shape() {

**this**.color = "Purple";

}

**protected** Shape(String color) {

**this**.color = color;

}

**public** String getColor() {

**return** color;

}

**public** **void** setColor(String color) {

**this**.color = color;

}

**public** **abstract** **double** getArea();

}

**public** **class** Triangle **extends** Shape {

**private** **double** side1, side2, side3;

**public** Triangle(**double** side1, **double** side2, **double** side3) {

**this**.side1 = side1;

**this**.side2 = side2;

**this**.side3 = side3;

}

**public** **double** getSide1() {

**return** side1;

}

**public** **double** getSide2() {

**return** side2;

}

**public** **double** getSide3() {

**return** side3;

}

**public** **double** getArea() {

**double** s = (side1 + side2 + side3) / 2;

**return** Math.*sqrt*(s \* (s - side1) \* (s - side2) \* (s - side3));

}

@Override

**public** String toString() {

**return** "Triangle: side1 = " + side1 + ", side2 = " + side2 +

", side3 = " + side3;

}

}

**public** **class** Circle **extends** Shape {

**private** **double** radius;

**public** Circle(**double** radius) {

**this**.radius = radius;

}

**public** **double** getRadius() {

**return** radius;

}

**public** **void** setRadius(**double** radius) {

**this**.radius = radius;

}

@Override /\*\* Return area \*/

**public** **double** getArea() {

**return** radius \* radius \* Math.***PI***;

}

@Override

**public** String toString() {

**return** "Circle: radius = " + radius;

}

}