Fall 2005

Course Development for a College Java Programming Class

Nathan Dodge
Regis University

Regis University
School for Professional Studies Graduate Programs
Final Project/Thesis

Disclaimer

REGIS UNIVERSITY

SCHOOL FOR PROFESSIONAL STUDIES

MASTER OF SCIENCE
IN
COMPUTER INFORMATION TECHNOLOGY

Course Development For a College Java Programming Class

PROFESSIONAL PROJECT PAPER

Nathan Dodge

October 29, 2005

Abstract

This project documents the development of college-level curriculum for an Object Oriented Programming with Java course. The curriculum includes a set of lessons that students work through interactively. The lessons teach the fundamentals of object orientation. A goal of the project is to have students work with the same problem example throughout the entire set of lessons. Most texts on object orientation use several abstract examples which are used in a chapter or two of the text and are often not fully implemented. Each lesson of the project's curriculum presents an iteration of an evolving shape drawing application. Each lesson walks the student through the design and development of a new fully implemented version. The shape drawing application's highly visual nature allows the students to relate the technical concepts to concepts they already understand.

Table Of Contents

1. Introduction

1.1. Overview

The deliverable of this project is curriculum consisting of a set of lessons on object technology. The goal of the lessons is to illustrate the use of object oriented concepts through the design and development of a functional application. The lessons involve the development of a class framework which deals with the application of drawing shapes. The anticipated audience of these lessons is a student taking a programming course in Java. The lessons are fully narrated and readable in a standalone setting and can therefore serve as material for either a distance or classroom-based course. It is expected that these lessons will supplement a textbook which covers Java and object concepts, as these lessons do not cover all aspects of Java syntax. These lessons are intended to illustrate and reinforce the major object-oriented concepts and demonstrate their implementation in Java in a real application, not necessarily to fully define every subtle nuance of Java's treatment of the topics. These lessons touch on many object oriented concepts. The lessons provide a framework for discussing objects, classes, encapsulation, inheritance, polymorphism, composition, association, responsibility and object persistence. The concepts are introduced and discussed as they are encountered in the development of the shape drawing application. The shape drawing application provides a problem context for discussing object oriented principles and illustrating object oriented programming approaches. The use of an evolving

example and it's modification, extension and refactoring illustrate how object oriented programming (OOP) concepts can solve typical coding problems and improve code reusability and code maintenance.

1.2. UML Model

A diagram of the static class relationships is shown below in Figure 1.

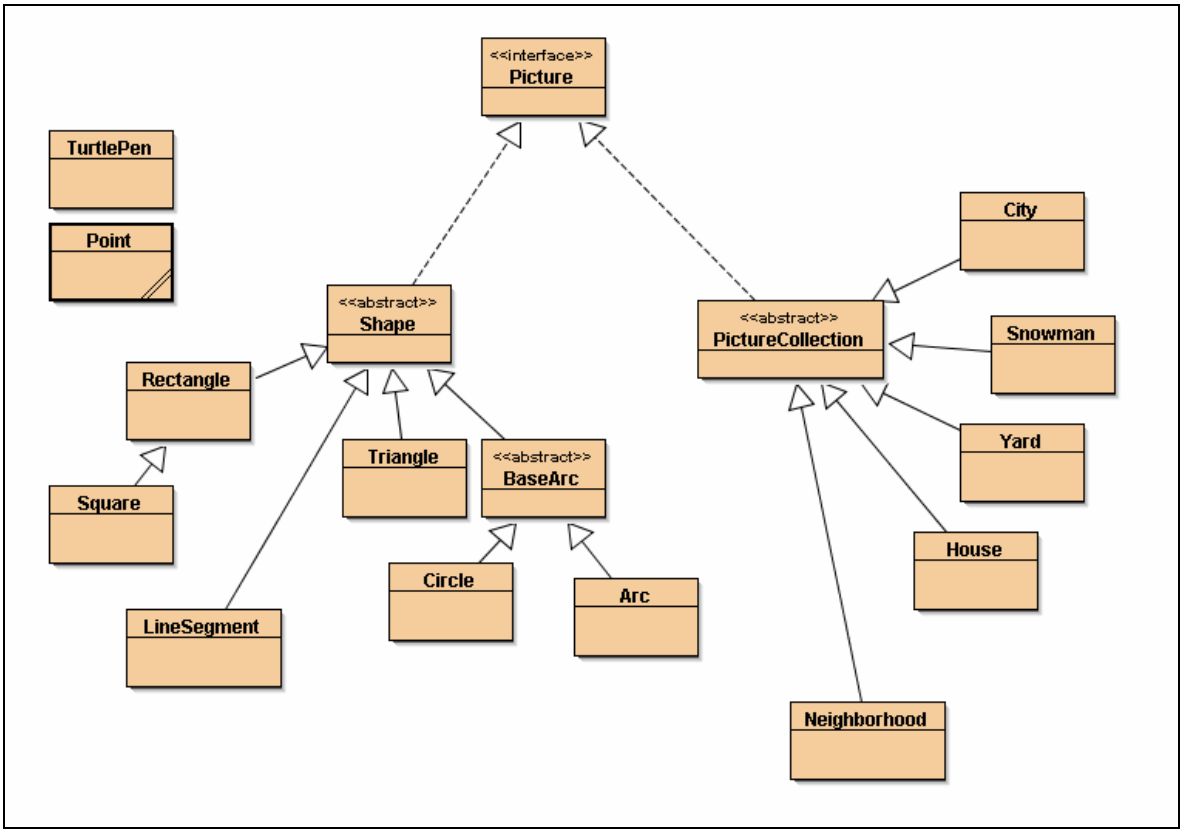


Figure 1

1.3. Need for the Project

Most texts on object orientation do not illustrate presented concepts with a fully implemented, realistic application. If examples are provided they are used briefly. A goal of the project is to have students work with the same problem example throughout the entire set of lessons. Each lesson presents an iteration of the evolving application and walks the student through the design and development of a new fully implemented version.

The lessons instruct the student to perform interactive steps using the BlueJ Java development environment. BlueJ is advantageous in that students can create and interact with live objects without writing any code. The BlueJ environment is fully documented in a tutorial provided with the software. These lessons instruct the user to work through the tutorial to become familiar with basic BlueJ operation.

Shape drawing was intentionally chosen as the problem context. The problem domain is intuitive, yet it has enough complexity to provide for opportunities for realistic solutions. Shape drawing facilitates visualization of object concepts. The project as a whole has a strong visual aspect. A highly visual application such as shape drawing as well as the use of visual techniques such as UML modeling and direct object interaction in BlueJ provide many graphic illustrations for the student.

1.4. Scope and Limitations

The lessons focus on object oriented programming concepts and therefore do not provide detailed coverage of analysis techniques such as use cases.

In a commercial shape-drawing application the end user would likely work through a front-end graphical interface that allowed them to select shapes and pictures from a palette and to drag and move and directly manipulate the shape objects. These lessons do not involve a graphical interface for drawing shapes. The code created in these lessons is the "model" layer, and a graphical interface layer, once created, would communicate with the model layer and create objects in response to user mouse and keyboard events.

2. Research

2.1. Textbook Research

A critical research task was to decide on the approach of the curriculum so that it didn't duplicate textbooks or material that are readily available on the market. Many textbooks exist that provide coverage of Java or object technology. Many texts try to provide full language coverage of Java. Other texts focus on object analysis or UML. After reviewing dozens of texts the following combination of factors was not found in any textbook:

- Project-driven approach, using an ongoing, evolving, working example
- Iterative approach, where concepts are introduced as they become applicable in the context of the problem at hand
- Emphasis on the object oriented thought process rather than on a specific tool
- Emphasis on object concepts in general rather than side language features such as graphical user interfaces or applets.
- Objects-first approach, rather than focusing on language nuances such as Java's main() method
- Use of examples that don't rely on awkward or complicated means of acquiring input
- UML coverage along with working Java examples

The above criteria was deemed desirable and critical and since a text could not be found that offered every feature it was decided to write the lessons in a manner that would offer these features.

2.2. Textbook Supplement

As mentioned in the introduction, the lessons do not attempt to provide full coverage and description of basic Java syntax. The lessons rely on a supplemental textbook to provide syntactical discussion for students. The text that was found to be the best match for the lessons was *Objects First With BlueJ* by Barnes and Kolling (Prentice Hall, 2004). The Objects First text provides an introduction to Java and it also matches on these desired philosophical approaches:

- Objects-first approach
- Iterative approach
- No full language coverage. Java syntax and features are covered to the extent needed for tasks
- Use of the BlueJ development environment (more on BlueJ below)

The Objects First text was not suitable on its own because of its lack of depth of coverage of inheritance and polymorphism. Inheritance and polymorphism are covered more towards the end of the text than would be desired. Although the

Objects First text has a number of small projects it does not have a significant project that evolves throughout the text.

2.3. Use of BlueJ

The instructional approach used in the lessons relies heavily on the use of the BlueJ Java development environment. BlueJ is a Java editor that was developed for computer education by the University of Southern Denmark, Deakin University and the University of Kent. BlueJ offers several advantages for instruction including:

- A free license
- A simple user interface, in contrast to the distracting interfaces of more complicated, commercial editors
- An integrated and automatic UML diagramming feature
- The ability to directly create and interact with objects

The ability to do direct experimentation with objects offers several advantages. Students can experiment and test their objects without having to write test driver classes. It is not necessary to code a main() method to create an object, which avoids the need to discuss complicated topics such as static and arrays until a more appropriate time. In other environments the cryptic signature of main `( public static void main(String args[]) )` exposes unnecessary technical detail and jargon to students new to Java. Methods can be run and

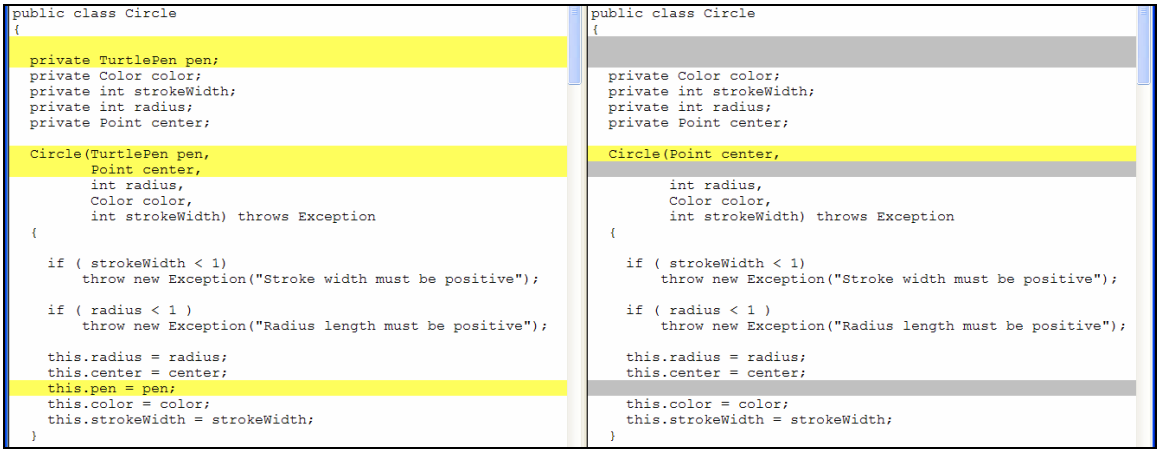
inputs supplied and return values examined without the need for writing complicated graphical user interfaces or low-level keyboard input code. Students can create objects interactively and invoke methods on the objects, as well as inspect objects to research their internal state.

2.4. Research of Other Tools

Other tools were needed for producing the lesson content. The lessons evolve over several iterations and some classes undergo refactoring from lesson to lesson. In order to illustrate to students implementation changes performed on classes between lessons a file compare tool was needed, preferably one that could show old and new files side by side with color-highlighted differences. After researching several tools a freeware file compare tool called WinMerge was selected (winmerge.sourceforge.net). WinMerge was used to produce file compares such as the one shown in Figure 2.

Another tool that was needed was a UML diagramming tool. Although BlueJ can create UML symbols it can only produce a small subset of UML diagrams. A tool was needed that could produce standard UML diagrams. Candidate tools that were researched were Rational Rose, Microsoft Visio and several freeware and shareware editors. After analyzing the requirements needed for a UML editor, it was realized that the main use would be to create UML diagrams for the lesson content. Students would need to view diagrams but they would not need to edit them. Advanced features such as those in Rational Rose were not needed, and therefore Rational Rose was ruled out due to its cost. While researching UML

tools the instructor learned of a way to obtain a free academic license for Visio and it proved to be an adequate tool for making UML diagrams that could be inserted into the lesson content. An example of a UML diagram created with Visio is shown in Figure 3.



```
public class Circle
{

  private TurtlePen pen;
  private Color color;
  private int strokeWidth;
  private int radius;
  private Point center;

  Circle(TurtlePen pen,
         Point center,
         int radius,
         Color color,
         int strokeWidth) throws Exception
  {

    if ( strokeWidth < 1)
        throw new Exception("Stroke width must be positive");

    if ( radius < 1 )
        throw new Exception("Radius length must be positive");

    this.radius = radius;
    this.center = center;
    this.pen = pen;
    this.color = color;
    this.strokeWidth = strokeWidth;
  }
```

```
public class Circle
{

  private Color color;
  private int strokeWidth;
  private int radius;
  private Point center;

  Circle(Point center,

         int radius,
         Color color,
         int strokeWidth) throws Exception
  {

    if ( strokeWidth < 1)
        throw new Exception("Stroke width must be positive");

    if ( radius < 1 )
        throw new Exception("Radius length must be positive");

    this.radius = radius;
    this.center = center;

    this.color = color;
    this.strokeWidth = strokeWidth;
  }
```

Figure 2. WinMerge File Compare

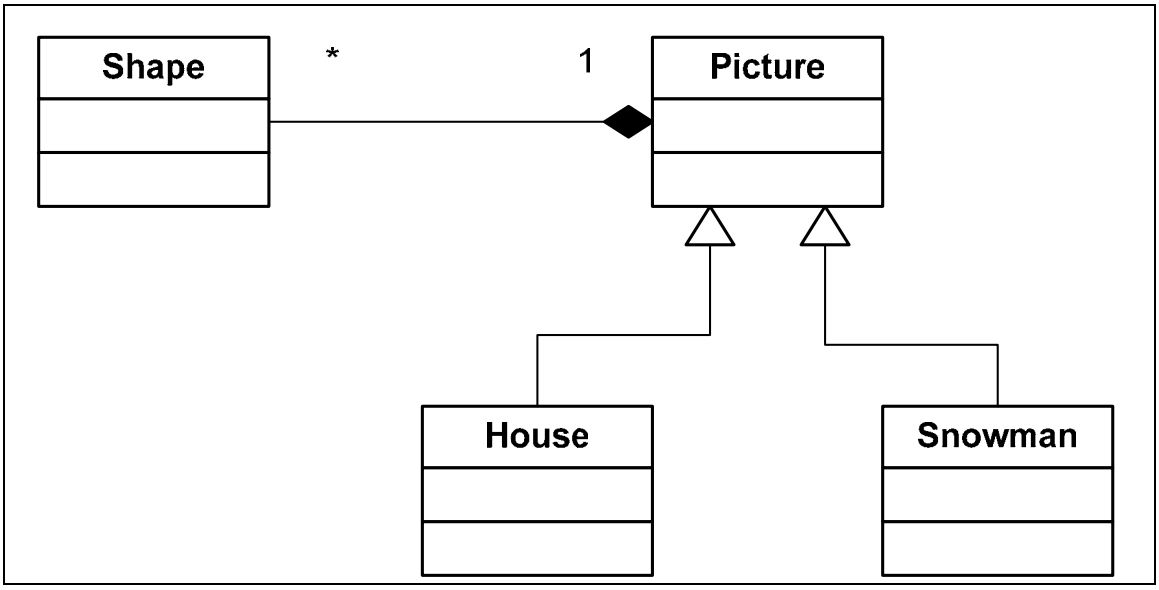


Figure 3. UML Class Diagram Created With Visio.

A key element of polymorphism is the concept of a heterogeneous collection. Students need to realize that a collection processed polymorphically contains objects of different types. The instructor was aware of visualization tools that could automatically generate graphs of data structures. Research was done to see if such a tool could create a graph of a heterogeneous Java collection. A tool called the Lightweight Java Visualizer (www.cs.auckland.ac.nz/~j-hamer/LJV.html) was discovered which could automatically create graphs of a Java ArrayList object. A graph produced from the Lightweight Java Visualizer is shown in Figure 4.

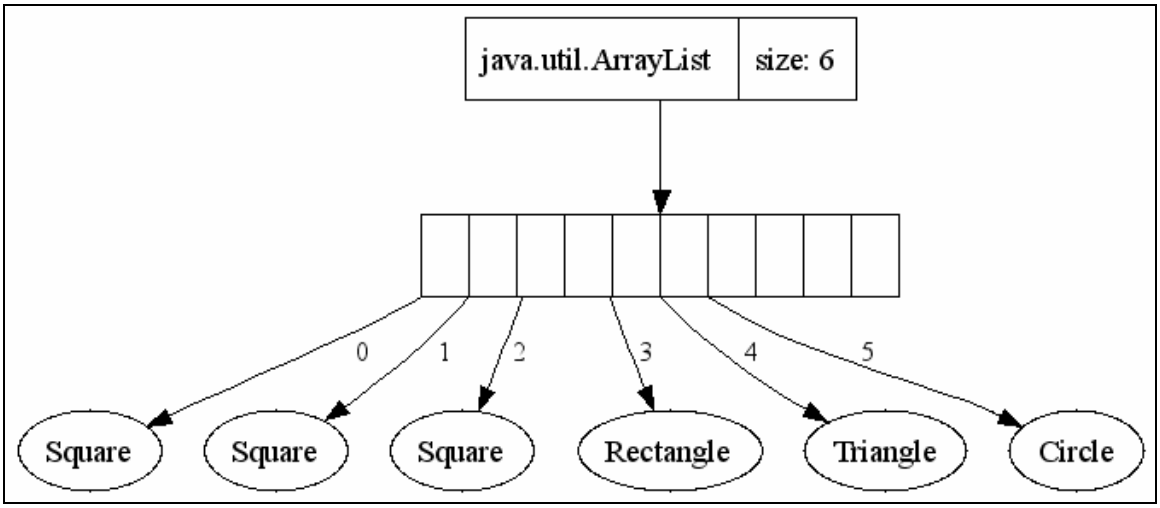


Figure 4. Graph produced by the Lightweight Java Visualizer.

2.5. Version Control

As development proceeded on the project many files were produced and many different versions of the classes had to be managed. Research was done to see if a version control tool could help to manage the large volumes of files. Many version control tools exist which rely on a server. A server-based tool was not

desired since the instructor did not have sufficient hardware of administration experience to host a server and it was not desirable to be dependent on an institution's server since the instructor often works as an adjunct professor and may not have access rights to any particular institution's server over time. What was desired was a version control product that offered a standalone version which could be ran locally on a single machine. Superversion was such a product (www.superversion.org). Jax magazine recommended Superversion as an excellent tool for single developers:

> "If you are working on a one-person project, and time is of essence, then Superversion could be just that handy version control tool that you are looking for."
>
> Jax magazine, 2004.

3. Methodology

Different methodologies were required for this project due to the different deliverables that were produced. The deliverables of this project were curriculum and its included software examples and demonstrations. Since this project's main deliverable was curriculum, curriculum-development methodologies were followed throughout the project. However, each student lesson involved a set of software examples and each piece of software followed its own development lifecycle. The curriculum development encapsulated the software development required for the lessons. The overall methodology was a blend of curriculum and software lifecycle phases.

The major project phases were:

1. Identify curriculum objectives
2. Identify use cases of lesson example
3. Identify classes used in lesson example
4. Map objectives to topics from lesson example
5. Plan and sequence lessons
6. Develop lesson example classes
7. Write lesson documents
8. Test lesson documents

Each project phase is discussed in more detail below.

3.1. Identifying Lesson Objectives

Following a methodology is not only important for designing software, it is also important for designing curriculum. Negative experiences have been suffered by the instructor in the past when teaching courses due to the failure of consistently applying a systematic process. For this course development, research was done to discover an established course development process. Such a method was found in the text Instructors and Their Jobs (Miller et al). Miller describes a method whereby educational content is discovered by trying to identify performance objectives. Performance objectives document what a learner should be able to perform after completing the course. Most objectives can be classified as either "knowing" (cognitive) or "doing" (psychomotor). Often a certain amount of fundamental cognitive skills must be known by the learner before applied tasks can be performed. The outline of course content is the list of performance objectives, properly categorized. Once a list of performance objectives has been developed, Miller suggests that the instructor can then divide the list further by dividing course content into units. Units allow the instructor to focus the content on a set of related performance objectives.

The global objectives of the course were identified as having the student be knowledgeable in the following fundamental object oriented concepts:

- Classes
- Inheritance
- Polymorphism

- Class responsibility and collaboration

These broad objectives were also identified as the units of the course. More detailed objectives were identified, grouped by the broader units, as documented below.

Students knowledgeable about classes should be able to:

- Invoke methods on object instances
- Identify the major parts of a class definition, such as fields, constructors and methods
- Discuss a class as a concept, as a specification and as an implementation
- Create multiple instances of the same class
- Distinguish between the interface and implementation of a class
- Identify the operations that should be allowed for objects of a class
- Make classes that are responsible for themselves
- Code accessor methods which query an object's state
- Code mutator methods which change an object's state
- Appreciate intuitive class interfaces
- Identify the components of a UML class symbol
- Hide implementation detail from the user of a class

Students knowledgeable about class collaboration should be able to:

- Identify associations on a UML diagram
- Identify an aggregation relationship on a UML diagram

- Identify a composition relationship on a UML diagram
- Describe the role of a class as a service provider
- Use objects of one class in the definition of another class
- Make decisions about how much responsibility a class should have
- Decide which class in a system should contain a given feature

Students knowledgeable about inheritance and polymorphism should be able to:

- Place behavior common to a group of classes into a new base class
- Inherit methods and fields from a base class into a derived class
- Make a specialized version of a class
- Override a base class method in a derived class
- Distinguish between abstract and concrete classes
- Identify an inheritance relationship on a UML diagram
- Appreciate the power of polymorphism
- Write source code that results in dynamic runtime behavior
- Process objects polymorphically
- Use a base class variable to hold subclass object instances
- Compare polymorphic and non-polymorphic code
- Use a Java interface to define a set of behaviors

3.2. Identifying Use Cases for Lesson Example

An instructional goal was to use the shape drawing example throughout the lessons to illustrate object oriented concepts. The instructor was already familiar with the basic idea of using shapes to illustrate object orientation, from having seen the example presented in other courses and textbooks and also from having used the example as a conceptual illustration when teaching courses in Systems Analysis and Design. Although the basic problem context was already understood it was deemed beneficial to list a set of use cases. Bahrami, in *Object Oriented System Development* describes use cases as defining "What users will be doing with the system. Use cases capture the goals of the users and responsibility of the system". Before considering any desired use, it was beneficial to consider if there were any constraints that the system should have. It was necessary to eventually implement the features of this application so analysis of anticipated use was confined to basic usage. The instructor was aware of different drawing frameworks and what they had in common was the ability to draw a line in a particular color in a particular width. This common line drawing functionality was used as a baseline and was used to constrain the list of potential features. After taking the system constraints into account the basic uses of the shape application were determined and are listed below:

- Draw a shape
- Erase a shape
- Size a shape
- Move a shape

- Change a shape's color
- Change a shape's line width
- Group shapes into a picture
- Draw a picture
- Erase a picture
- Move a picture
- Group pictures to form new pictures
- Save a picture

3.3. Identifying Classes for the Lesson Example

Once system use had been defined attention could be turned to identifying the objects and classes present in the problem domain. Bahrami describes a process for object analysis which includes these main steps:

- Identify classes
- Identify relationships
- Identify attributes
- Identify methods
- Iterate and refine

A first pass at identifying classes was accomplished by using a noun phrase approach. Examining the list of use cases for noun phrases identified the candidate classes Shape and Picture. "Shape" was used in the use cases as a category name of a broad set of objects. Determining a list of shape types

yielded more candidate classes, such as Circle, Rectangle, LineSegment, Triangle and Square. Other noun phrases found in the use cases, either directly or implied, included color and location (where to move to).

After the classes were identified class relationships were determined. The basic relationships that were determined were:

- A general Shape class would hold behavior common to all shapes
- A Circle is a kind of Shape
- A Triangle is a kind of Shape
- A Rectangle is a kind of Shape
- A LineSegment is a kind of Shape
- A Square is a kind of Rectangle
- Pictures contain numerous Shape objects
- Pictures can contain other Pictures

The following attributes were determined:

- Pictures and shapes have a location
- Shapes have a color
- Shapes are drawn in a particular line width
- Circles have a radius
- Rectangles have a height and a width
- Line segments have an ending point
- Triangles have three points in total

Since color and location are non-primitives classes were introduced to represent them. Location was changed to Point since the word "point" is commonly used to define a location in a two dimensional plane.

The following methods were determined:

- Shapes and pictures can be drawn
- Shapes and pictures can be erased
- Shapes and pictures can be moved
- Shapes and pictures can have their color changed
- Shapes and pictures can have their line width changed
- Circles can have their radius size changed
- Rectangles can have their height or width changed

The first pass of classes, relationships, attributes and methods were documented in a UML diagram, which is shown in Figure 5.

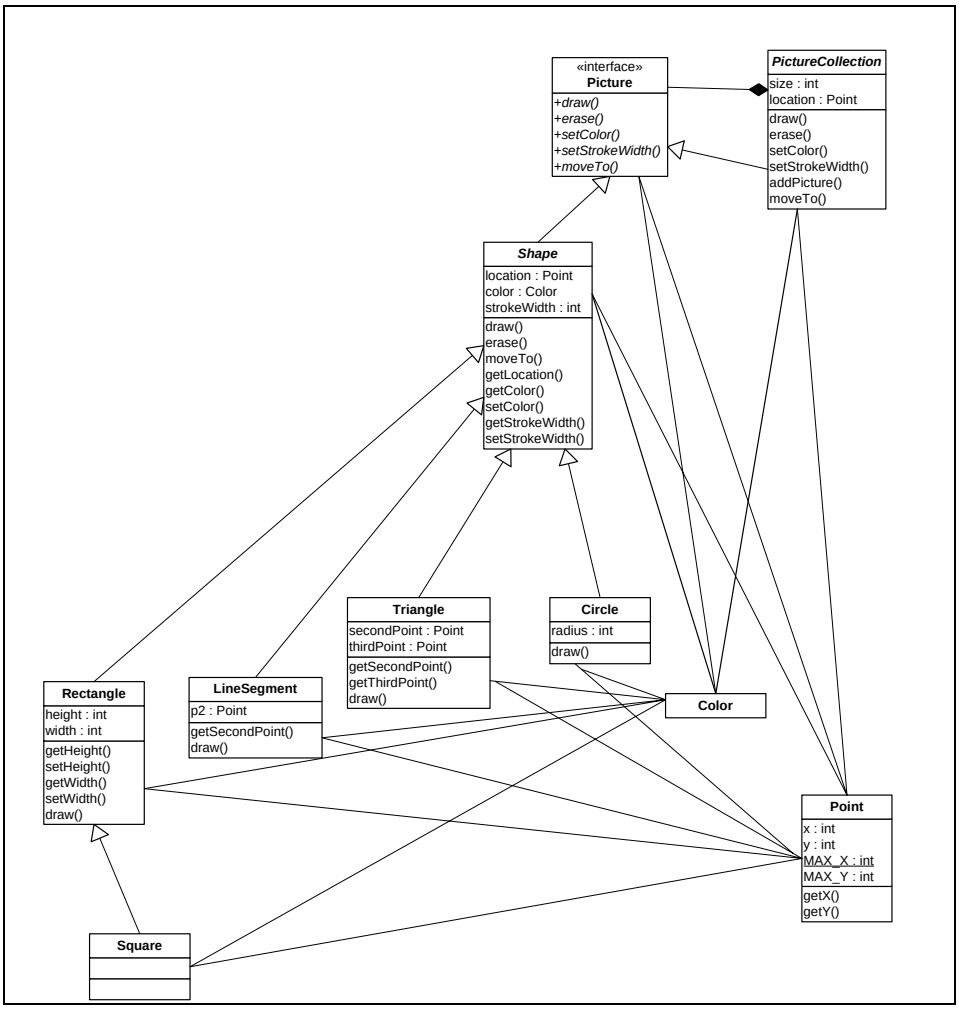


Figure 5 Initial UML diagram

3.4. Mapping Objectives to Topics from Lesson Example

At this point in the project development what had been produced was a list of objectives for learner competencies and an identified example project that had undergone initial analysis. The primary goal of the project was to develop lessons that would meet the performance objectives. What needed to be done next was to plan lessons by mapping objectives to discussions related to the problem example of the shape application. This process of mapping was used to refine and fill out the shape example, because it was anticipated that the initial

analysis might not contain enough substance and context to provide for examples for every objective. As objectives were encountered that did not have an immediate applicability to the problem example, analysis was done to identify how the shape example could be refined to illustrate the objective. The results of the mapping are shown in the following table.

Objective	How this objective will be accomplished through the shape example
Discuss a class as a concept, as a specification and as an implementation	Discuss every class in this manner. As each class is introduced, first explain the general concept, then describe what the public interface to the class should be, and finally, as a last step discuss the implementation.
Invoke methods on object instances	A means of drawing is needed. A turtle-graphics type pen object is a very simple, well-defined object with observable behavior that is a good choice for being a first object to use. Students will use the pen object and call its methods to draw onto the screen. Students will, of course, invoke methods on many other objects as well, such as drawing, erasing and moving shapes.
Identify the major parts of a class definition, such as fields, constructors and methods	Present these concepts using the Circle, Triangle and Rectangle classes. Discuss that each shape type has a unique set of fields. Each shape type needs unique code for drawing the shape.
Create multiple instances of the same class	Have the student create various Circle, Triangle and Rectangle objects with different characteristics.

Objective	How this objective will be accomplished through the shape example
Distinguish between the interface and implementation of a class	Do this for every class, by discussing class as concept, specification and implementation. Especially distinguish by showing multiple ways to store field values for a Rectangle.
Identify the operations that should be allowed for objects of a class	Do this for every class. For the Point class discuss utility methods such as equals() and toString().
Make classes that are responsible for themselves	Do this for every class. Discuss disallowing invalid parameters such as a negative size, two duplicate points for a LineSegment, duplicate points on a Triangle or all three points on a line for a Triangle.
Code accessor methods which query an object's state	Do this for every class. Initially discuss in detail for the Circle, Triangle and Rectangle classes.
Code mutator methods which change an object's state	Do this every class. Initially discuss in detail for the Circle, Triangle and Rectangle classes. For the Point class discuss whether or not to allow mutation.
Appreciate intuitive class interfaces	First create shape classes with the user supplying the familiar pen object, then redesign by proposing that end users don't need to know about the pen.
Identify the components of a UML class symbol	Discuss UML class symbols when discussing the fields and methods of the Circle, Triangle and Rectangle classes.

Objective	How this objective will be accomplished through the shape example
Hide implementation detail from the user of a class	First create shape classes with the user supplying the familiar pen, then redesign by proposing that end users don't need to know about the pen.
Identify associations on a UML diagram	Discuss association relationships as new classes are introduced (show UML diagrams with the new classes included with relationships to existing classes shown).
Identify an aggregation relationship on a UML diagram	First introduce the Picture class as aggregating Shapes, and then introduce the PictureCollection class as an aggregate of any kind of Picture. Present UML diagrams showing the aggregate relationships.
Describe the role of a class as a service provider	Do this for every class.
Use objects of one class in the definition of another class	The shape and Picture classes are defined in terms of a Point and a Color.
Make decisions about how much responsibility a class should have	Do this for every class.
Decide which class in a system should contain a given feature	Discuss which classes should be aware of the exact drawing mechanism that is used. TurtleGraphics uses a coordinate system where the center of the screen is the origin. Discuss approaches where the lower left is treated as the origin and discuss which classes in the system know about the center-based origin and which know about the corner-based origin.

Objective	How this objective will be accomplished through the shape example
Place behavior common to a group of classes into a new base class	Introduce the abstract Shape class which contains behavior common to all shape types. Create an Arc class that can have behavior common to Circles and other curved shapes.
Inherit methods and fields from a base class into a derived class	Discuss how Circle, Triangle, Rectangle and other shape classes inherit fields and methods from Shape.
Make a specialized version of a class	Discuss Square as a specialized Rectangle
Override a base class method in a derived class	Discuss that Shape's move method is not sufficient for either Triangle or LineSegment (but that it is sufficient for Circle and Rectangle).
Distinguish between abstract and concrete classes	Discuss that Shape is abstract and therefore objects of exactly type Shape cannot be created, in contrast to a concrete class like Rectangle which can have instances of it created.
Identify an inheritance relationship on a UML diagram	Show UML models with inheritance symbols for Shape and its subclasses. Also show that Shape and PictureCollection are specializations of Picture.
Appreciate the power of polymorphism	Emphasize that PictureCollection does not need to know exactly what kind of Picture objects it is processing.
Process objects polymorphically	Code PictureCollection's operations such as draw() as acting on Pictures, without the need for PictureCollection to know exactly what kind of Picture it is drawing.

Objective	How this objective will be accomplished through the shape example
Use a base class variable to hold subclass object instances	When processing all Pictures in a PictureCollection, hold the object in a variable of the abstract type Picture, even though the object will be of a more specific type such as a Shape or a PictureCollection.
Compare polymorphic and non-polymorphic code	Compare the polymorphic version of the Picture operations to a non-polymorphic version which tests for shape type.
Use a Java interface to define a set of behaviors	Define a Picture generically as something that can be drawn, erased, .etc.

3.5. Planning and Sequencing Lessons

The next task was to break the shape example into lessons of roughly uniform length. Each lesson was designed to present a set of new topics and use classes from the shape drawing context as illustrations and implementations of the concepts. Summaries of the lessons with lesson-specific objectives are shown below. Nine lessons were identified for the course.

Lesson	Title
1	Using an Existing Class
2	First Shape Classes – Circle, Triangle and Rectangle
3	The Point Class
4	Revisiting Circle, Triangle and Rectangle
5	Further Polish
6	Inheritance
7	Polymorphism
8	The Composite Pattern
9	Object Persistence

3.5.1. Lesson One – Using an Existing Class

In the first lesson the student is instructed to install and configure the BlueJ development environment. The student then works with an existing class. The concepts of class and instance are discussed. The student is introduced to the TurtlePen class, which is the means of drawing onto the screen. Three views of a class are presented – a class as concept, specification and implementation. This is the treatment of classes given by Martin Fowler in *UML Distilled* (Addison Wesley Longman 2003). Students learn how to create objects and call methods interactively within BlueJ and also programmatically with Java code statements.

After completing lesson one the student will be able to:

- Install and configure the BlueJ Java development environment

- Create an object on the BlueJ object workbench
- Interactively invoke methods on object instances
- Invoke methods through lines of code

3.5.2. Lesson Two – First Shape Classes

In the second lesson, students get shown how to build a class. The student is shown how they can represent common shapes such as circles, rectangles and triangles as classes. The notion of a class as a concept, a specification and an implementation is stressed and each shape class is considered from these three perspectives. Initial implementations of the shape classes are studied. Keeping interface separate from implementation is stressed and two different implementations of a rectangle are contrasted, while keeping the interfaces the same.

After completing lesson two the student will be able to:

- Identify the major parts of a class definition, such as fields, constructors and methods
- Discuss a class as a concept, as a specification and as an implementation
- Create multiple instances of the same class
- Distinguish between the interface and implementation of a class

3.5.3. Lesson Three – The Point Class

In the third lesson the importance of having classes model the real world is stressed. It is pointed out that the natural, intuitive way of describing locations of shapes is via a singular reference, such as a circle's center point or a rectangle's corner point. Triangle's existing interface is examined, which has six integers in its constructor for specifying the three points of the triangle. The desire to express information in a more natural way is highlighted, and a Point class is introduced which allows a more natural, conversational way of describing the locations of shape objects. The shape classes are adjusted to use Point objects rather than integers, providing an opportunity to discuss class relationships in general. Class functionality and responsibility are also discussed.

After completing lesson three the student will be able to:

- Describe the role of a class as a service provider
- Use objects of one class in the definition of another class
- Make decisions about how much responsibility a class should have

3.5.4. Lesson Four – Revisiting Circle, Triangle and Rectangle

In the fourth lesson the shape class implementations are discussed and examined more completely, with more shape operations such as erasing, moving and changing color defined. UML class symbols are discussed. Classes are designed so that they are responsible for ensuring that objects are always in a valid state.

After completing lesson four the student will be able to:

- Identify the components of a UML class symbol

- Identify associations on a UML diagram
- Identify the operations that should be allowed for objects of a class
- Make classes that are responsible for themselves
- Code accessor methods which query an object's state
- Code mutator methods which change an object's state

3.5.5. Lesson Five – Further Polish

In the fifth lesson nuances of our system are addressed. Solutions to these nuances are discussed and it is decided which classes should have responsibility for certain behavior. A closer look is given to the TurtlePen class, and it is discussed how TurtlePen acts as a façade in front of a more complicated StandardPen class. It is pointed out that users do not necessarily need to know about TurtlePen when creating shape objects. A singleton design pattern is discussed which is used to ensure that one and only one pen object is created, and that the pen is created only when it is needed. The importance of hiding implementation detail from our clients is stressed.

After completing lesson five the student will be able to:

- Appreciate intuitive class interfaces
- Hide implementation detail from the user of a class
- Explain the singleton design pattern
- Explain the façade design pattern
- Decide which class in a system should contain a given feature

3.5.6. Lesson Six – Inheritance

In the sixth lesson inheritance is introduced. A Square class is presented as a specialized Rectangle. A circle is discussed as being a special case of a more general Arc class. It is pointed out that conversationally the term "shape" has been used many times already in the lessons. The abstract class Shape is developed and discussed. Shape serves as a class which can contain the fields and methods common to all shapes. Inheritance solves the problem of redundant code that had existed in the shape classes.

After completing lesson six the student will be able to:

- Place behavior common to a group of classes into a new base class
- Inherit methods and fields from a base class into a derived class
- Make a specialized version of a class
- Describe the three basic class relationships
- Distinguish between abstract and concrete methods

3.5.7. Lesson Seven – Polymorphism

The seventh lesson focuses on polymorphism. A desire is expressed to be able to manipulate a picture which is made up of a combination of shapes. The picture should be able to be drawn, erased and moved as a singular unit. A Picture class is presented which contains a heterogeneous collection of shape objects. To implement the operations on a Picture, polymorphism is used. Essentially, to draw a picture the code cycles through the collection of shapes and asks each shape to draw itself. From Picture's perspective, what is being

drawn is referred to as a generic Shape object. At run-time the appropriate draw method is called depending on the actual type of shape retrieved.

After completing lesson seven the student will be able to:

- Appreciate the power of polymorphism
- Write source code that results in dynamic runtime behavior
- Process objects polymorphically
- Use a base class variable to hold subclass object instances
- Compare polymorphic and non-polymorphic code

3.5.8. Lesson Eight – The Composite Pattern

In the eighth lesson it is proposed to be able to create a picture which is made up of other pictures. The essence of "what is meant by a picture" is discussed. Picture is a term is that is getting increasingly abstract. Java interfaces are discussed as a way to define common behavior. Picture is reworked to be an interface rather than a class. A PictureCollection class is introduced which can contain either Pictures or other PictureCollections. PictureCollection as well as the Shape class both implement the Picture interface, allowing all items to be treated uniformly, whether they are a single, simple shape or a complex drawing made up of many sub drawings and pictures. After demonstrating the solution it is explained that the solution is an example of the general composite design pattern.

After completing lesson eight the student will be able to:

- Describe the Composite design pattern

- Use a Java interface to define a set of behaviors

3.5.9. Lesson Nine – Object Persistence

In the ninth lesson it is shown how classes can take on added responsibility. The desire to save created pictures is presented, and intelligence is added to each class in order to show how each can know how to save its state data as XML. The Java-XML integration is accomplished by making use of a third party JDOM class library.

After completing lesson nine the student will be able to:

- Work with the JDOM class library
- Store an object's state as XML data

3.6. Developing Lesson Example Classes

A specification was written for every class, showing the methods needed for the public interface. These specifications were included and explained to the students as part of the lesson documents. An example specification is shown below.

The Point Class	
Method	**Notes**
new Point (int x, int y)	Create a point with the given x and y values
int x()	Return the x value of a point
int y()	Return the y value of a point
boolean equals()	Compare a point to another point
String toString()	Return the point information as a formatted String

After specifications were written each of the following tasks were completed for each of the lessons:

- Implement classes used in the lesson
- Test classes used in the lesson
- Write descriptions and explanations of code listings used in the lesson text
- Implement student assignment solution
- Test student assignment solution
- Proofread lesson text
- Test student activities

When testing the classes, it was often advantageous to write a test class, since testing interactively using BlueJ can be very tedious. These test classes were kept in a project separate from the student files, to avoid cluttering the projects that the students would be using. Also, student assignment solutions were placed into a separate project. Care was needed when developing projects to be used by students. Projects needed to be set up with the student activities undone, and then the student activities needed to be performed and tested without disturbing the student version to make sure the directions would result in a code solution that met the intended design.

A UML class diagram of the completed system is shown below

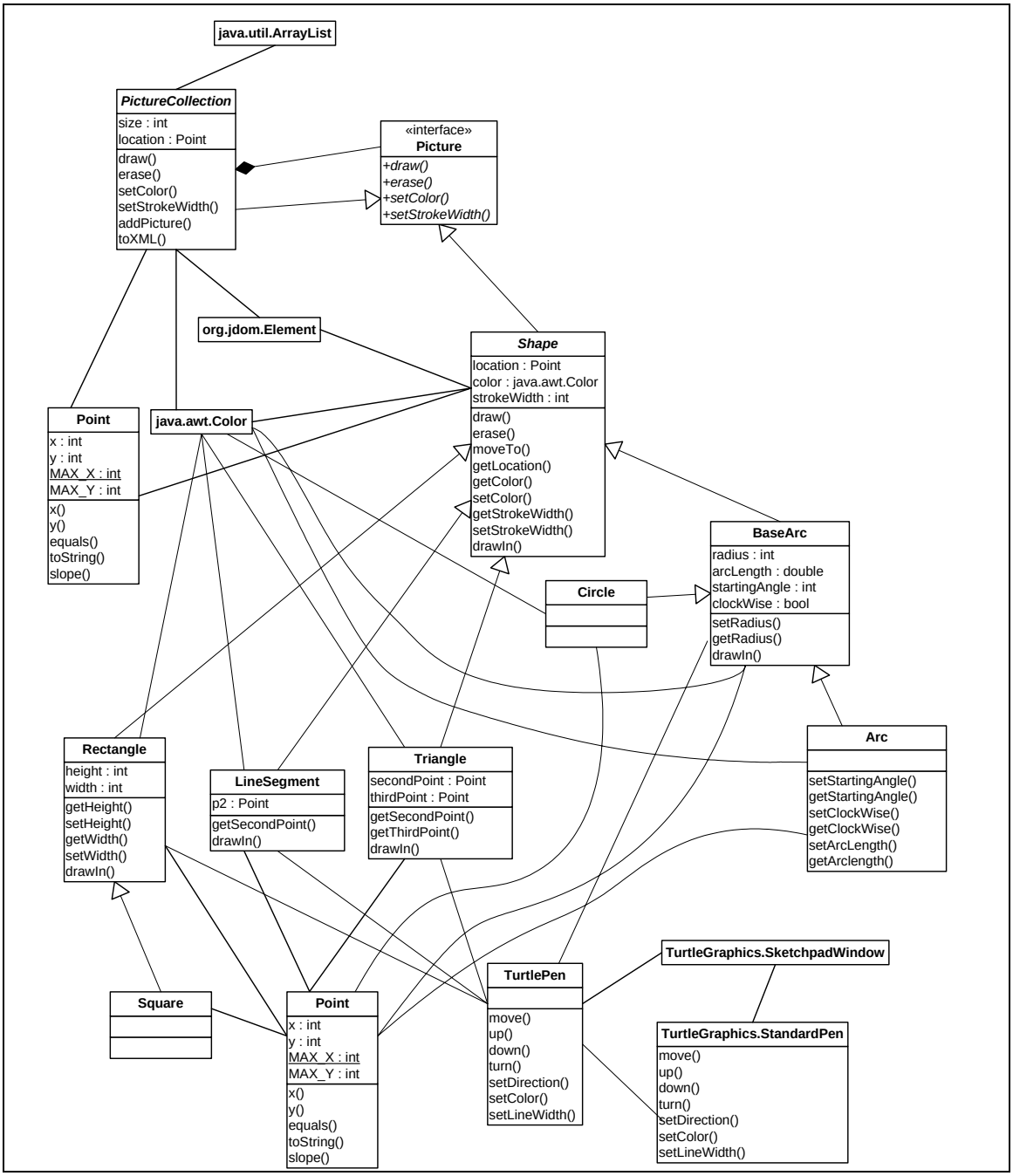


Figure 6 Static Class Structure of Classes Used in the Lessons.

3.7. Writing Lesson Documents

It was decided that it would be advantageous both for student reading and for lesson development to have each lesson follow a consistent format. A template document was produced to be used as a starting point for each lesson document. Standards were developed that were followed for each lesson document. Step by step instructions are consistently shown in a shaded background so that they are distinguishable from general discussion text. Code listings are consistently shown in a fixed Courier font. UML class diagrams are shown to depict important class relationships. Screen shots are used to show student tasks in the BlueJ environment or to show pictures of output that the students should be able to reproduce in their step by step tasks.

Each lesson includes:

- Description of files needed
- Deliverables
- Objectives
- Introduction discussion
- Main content, which contains:
    - Concept discussions
    - Step by step instructions for students
    - UML diagrams of classes, where appropriate
    - Discussion of classes as concept, specification and implementation
    - Code listings with discussion
- Assignment directions

3.8. Testing of Lesson Documents

As a final test, all lessons were reread and all student activity steps were performed. These steps were performed on a machine that did not initially have BlueJ, Java or any files installed. The lessons were given a final proofreading. The lessons were then tested starting with the first lesson and working through every lesson in the order that the students would encounter the activities.

3.9. Backup of Materials

As project development proceeded, many documents were produced, including word processing documents, UML diagrams and Java source code. To protect the investment of time put into these documents, consistent backup of materials was performed. Backing up documents to a different media was critical to protect against media failure (such as a hard drive crash). Having backups on different media would do no good if the media were in the same physical location and a physical disaster such as a fire occurred, so backup to different locations was important. Redundant backups to multiple media was considered important in case one backup media itself failed. The specific backup process utilized was as follows:

- Condense all documents and code into compressed folders
- Copy compressed folders onto a removable flash drive
- Place flash drive in separate location from development computer
- Copy compressed folders onto virtual Internet drive (for location redundancy)

It was important to follow this backup procedure on a timely basis. Backups were generally performed every week or after a substantial amount of material was produced.

3.10. Best Practices for Students

Just as a solid methodology was used to develop the course curriculum and materials, a process was also developed for using best practices while conducting the class. Several guideline documents were developed which communicate to the student how to submit (and resubmit) solutions in a manner that facilitates an efficient software review, testing and tracking process which models industry practices as closely as possible. The student was reminded that their activities span many lifecycle phases, especially the ending phases. Students are involved with the packaging and distribution of software deliverables and such activities should be treated with care. Students were informed that the instructor will perform two main activities when critiquing their assignments:

1. Peer Code Review**.** Student program code will be browsed and reviewed and feedback will be given on style and coding practices. Students are reminded that peer code reviews are commonplace in industry.
2. "Black-Box" Testing**.** Student programs will be ran and tested for correctness, robustness and usability. "Black Box" means that the tester is not concerned with how the program was made (i.e. the tester does not care what is "in" the box), but rather that the program runs correctly and efficiently from an end-user's point of view.

Students are able to submit revisions of assignments additional times as opportunities to correct items and get additional feedback. The encouragement of resubmission and the communication of feedback models industry interaction between software testers and software developers. In industry, software cannot be left in a dysfunctional state, so students should strive to perfect their solutions rather than simply submitting solutions once and moving on. Coding style is important and students were given a guideline for proper use of capitalization, indentation and code formatting.

4. Project History

Many events in the past few years had an impact on the final nature of the project. I developed proficiency in object oriented concepts by coding for several years in industry using C++ and Java. I first taught object oriented concepts when teaching a systems analysis course, which included a small unit on object orientation. I taught object orientation in further detail when teaching several C++ and Java courses at three different Minneapolis area colleges. My initial proposal for Regis was to develop a set of advanced enterprise Java courses, but the exact nature of these courses was a constantly moving target and the school I was to develop them for closed its doors before the courses were offered. Eventually I decided to focus my proposal on developing a set of object oriented design lessons that would be generic enough to use as a supplement to many different courses on Java or object oriented programming.

A detailed timeline documenting significant events is presented below.

- From 1992 until 2001 I worked in industry mostly doing development with object oriented C++ and Java.
- I began teaching courses in 1999 and used the basic idea of the shape example to illustrate object oriented concepts. The ideas were purely conceptual and were not implemented in a programming language.
- In 2001 I began taking Regis master's classes, pursuing a concentration in Java and OO technologies. I completed most of my coursework in 2001.

After completing my last non-thesis class, I decided to take a little time off from school and tackle my thesis after a break.

- Also in 2001 I had been teaching part time at NEI College of Technology in Minneapolis, teaching primarily Visual Basic courses in a traditional MIS program. (I taught Java and C++ courses at another college, but on an occasional basis.)
- In 2001, NEI was preparing for a new degree, which seemed very unique and aggressive in it's offerings – giving students background in client-side and server side technologies as well as general e-business. The thought was to have a strong Java aspect to the curriculum, especially on the server-side portion. This program was called the "Web Program". The program was scheduled to start in Jan of 2002.
- Due to a lot of politics at the school, there was no headway in sharing any core courses between the MIS program and the Web Program. The two programs remained separate from one another.
- In late 2001 I was approached about being hired full time to work as an instructor in the web program, but they did not have the funds to bring me on full time until the students began taking the classes they anticipated me developing. My courses would be taken by students during their later quarters in the program (with first offerings starting in late 2002). There was a time during 2001 and 2002 when my involvement was anticipated but not formalized. I continued to teach part time in the established MIS program

- In early 2002 I decided on proposing that my development of curriculum for the NEI College Web Program would be my idea for my Regis Professional Project.
- In the spring of 2002 I enrolled in Regis MSC696A, but I withdrew – wanting to wait until the definition of the later courses in the web program was more solid.
- I re-enrolled in MSC696A in the summer of 2002. I wrote a proposal which argued about the uniqueness of the NEI Web Program degree and about my role as curriculum developer and instructor. The exact courses that I would be involved with was still not entirely known at that point, so I could only speak about them at a high level. I knew that the curriculum was going to involve server side technologies and Java. Candidate topics were Object Orientation, Websphere, JSP, Servlets and J2EE. My Regis proposal was accepted at the end of the MSC696A term.
- In September of 2002, I came on board as a full-time faculty member of the Web Program department.
- In the NEI Web Program there were two main faculty members (other than myself) involved with the degree development. The department head (Dr. Bill Warner) was involved with the client-side portion of the program, and these were the courses that were offered first.
- The other faculty member, Bob Nell, had concerns about the sequencing of courses in the web program. Bob was in charge of the server side courses. I shared the same concerns as Bob. The main concern was that

students were first being shown "fancy" front-end tools like DreamWeaver, FrontPage, PhotoShop and Flash but were not being instructed at all about programming fundamentals and other fundamental, critical-thinking skills. Concerns were expressed to the department head but they were brushed aside.

- The first two courses that I was involved with in the web program were an HTML / JavaScript class, and a course in Java / OO. Before these two courses, students had no programming background whatsoever.
- I tried to do the best that I could with the situation, even though it seemed futile. In two classes had to teach students HTML and JavaScript and Java and OO, in the hope of preparing them for courses such as JSP and J2EE and Websphere.
- The HTML and JavaScript course actually had two other main objectives – to teach students Cascading Style Sheets and Dynamic HTML. This curriculum was far too aggressive and there were many fires being put out in trying to make the best out of this course. The JavaScript portion of this course was the first spot in the degree where students were introduced to programming.
- The first run of the Java / OO course was similarly disastrous -- the objectives established by the department head were far too aggressive and students simply did not have enough fundamental prerequisite skills. Basic Java syntax and simple programming examples and a survey of sorts of OO concepts was what was able to be covered. This "survey" of OO

concepts material was the beginnings of the curriculum presented in this paper.

- It became very apparent that the curriculum was completely flawed regarding offering server-side Java courses such as JSP or J2EE. As a stop-gap measure, other technologies were put in place for the first run of some of the server-side and middle-ware courses. Other instructors taught students some tools such as Coldfusion.
- With all of the flurry of change and putting out of fires in this first year of the Web Program, and with the scaling back and nearly eliminating the use of Java in the degree – I wasn't quite sure what to do regarding my Regis project, so I delayed it.
- Later in 2002 the department head of the Web Program announced that he would be leaving his position. The Dean of the college hired Bob Nell, the server-side expert, to be the new department head, and also made him department head of a combined department which now included the old MIS program (the MIS program had seen very steadily declines in enrollment).
- Over the next few months, steady progress was made in redesigning and merging the curriculum for the two degrees. A common "IT Core" was developed and then, in later courses, students would focus on a concentration of either web development or traditional development.

- What remained in the new curriculum related to my Regis MSC696 proposal (related to Java) was a single Java / OO course. There were no longer any server-side Java courses remaining.
- Many things were in flux at NEI during an interim period where we had both day and evening students and both MIS and Web students caught mid-way in the transitions of the curriculum. The Java / OO course was taken for a few times by a mix of Web and MIS students (as electives by some). In an attempt to accommodate different audiences, the course contained a "survey" of different components – programming logic and syntax in Java, comparing and contrasting syntax styles with Java and JavaScript, and then, as a last portion an exposure to OO concepts in Java. The course at this time went well. I decided to propose that my Regis professional project would now be to take the material that was hastily developed for the Java / OO course and more formally develop it.
- In 2003, just when things were settling down, an announcement was made that my school would be merging with Dunwoody College Of Technology, another technical college in Minneapolis. Some adjunct staff and a couple full time faculty members at NEI were let go, and the remaining faculty members were given a heavy work load and had difficult job assignments (preparing courses they hadn't taught before) during the interim period before the merger was complete. Job security was an issue for everyone. My Regis professional project again was put on the back burner.

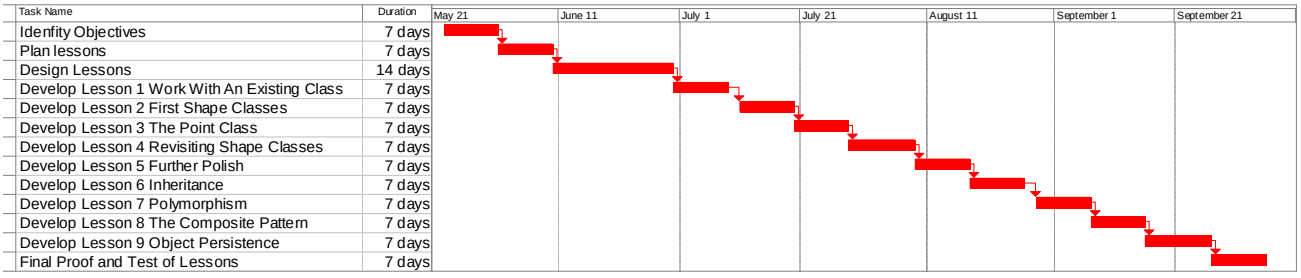
- In early 2004 the merger was completed and courses were now being offered at Dunwoody's campus. I accepted a position to teach at Dunwoody.
- I wanted to push through my Regis project before I got too involved with a new, large pending project – the merging of NEI 's curriculum with that of Dunwoody. And in the meantime, both Dunwoody and NEI curriculum tracks were being offered until the existing students in each program completed their tenures.
- I enrolled in Regis MSC696B in March 2004 and had a couple hundred pages of material in my paper– formalizing examples from what I had used in the Java course over the last year or so. Due to the hectic nature of the course development there was a lot of work to do to formalize the material (with defined objectives and lesson plans, .etc).
- I had been under the false assumption that my developed curriculum was the bulk of the Regis paper (I now know otherwise), and I had thought that I needed to get the majority of my curriculum documents polished and prettied up and neatly bundled into a formal document – this is what the effort on my Regis project was for quite some time.
- As the March 2004 Regis term approached, I was not satisfied with the organization of the material, so I decided to (once again) delay my entry into 696B, with the hope of targeting a May start.
- I sent my paper to Ted Faurer (my advisor) for review in early May 2004. I greatly regret not involving Ted more before then, but the exact course I

was actually developing had been in flux for a long time, and the hectic ness of the department direction changes and curriculum mergers and school mergers made it so that I did not know if what I had been proposing was even going to be offered at my school. Before I gave Ted something to look at, I wanted to have completed something more substantial.

- Mr. Faurer made some good commentary on my project. He questioned why the JavaScript portion was included – and had some questions regarding whether my material was supposed to be a textbook, be online material, or material that would supplement a textbook. Also, a lot of the fundamental Java logic and fundamental OO examples I had developed would be, he said, adequately covered in a standard Java textbook. I did have an extended OO example that had some merit, but it needed more explanation and documentation.
- Mr. Faurer and I had some good conversation going back and forth, and we came up with an idea that my extended OO example could be more fully annotated and have other OOD artifacts included. This extended example would become, essentially, the "course", and the JavaScript and Java logic portions could be removed. I began to make adjustments to my technical content to meet this end.
- In May and June of 2004 there were many meetings at Dunwoody to outline the new curriculum that would replace the separate NEI and Dunwoody programs.

- Due to a heavy workload in this new department I again delayed my thesis paper for the rest of 2004.
- In January of 2005 I received notice that I was being laid off from Dunwoody College of Technology due to low student enrollment.
- In February of 2005 I accepted an offer from the College of Saint Scholastica in Duluth, MN to develop an Object Oriented Design course for their Masters in Computer Information Systems program. The course objectives for this course were very similar to the objectives of the Java OO content of the courses that I had developed for both NEI and Dunwoody over the past few years.
- I decided to develop the set of OO lessons into a format that would be usable in many different course situations. I developed lessons that were fully annotated that could be read and completed offline. The problem context would also be suitable for an instructor to lecture on and demonstrate in a classroom.
- From May to September of 2005 I finalized the development of the lessons. A detailed timeline for 2005 is shown below.

4.1. Project Design and Development Timeline



5. Conclusion

5.1. Future Developments

Further development is very likely on this project. The shape example can be greatly expanded and can be used to illustrate many other object technology concepts. It is desired to include much more upfront material for the student regarding use cases and object oriented analysis, to truly make the example one that starts from project inception and proceeds through system completion. As the student material expands to include more content on other lifecycle phases much opportunity exists to illustrate many other UML diagram types, such as state diagrams and sequence diagrams.

Many new features have been considered for the application, each an opportunity to discuss other design patterns. A few of the identified potential features are listed below:

- Create a graphical user interface layer which a user could use to create and manipulate a drawing with direct interaction. The user interface layer would communicate with the model layer developed in this project, implementing the classic model-view-controller pattern.
- Allow for scaling of shapes. Scaling of shapes was not done completely in this project, partly due to an incomplete design of how to scale shapes such as LineSegments and Triangles and how to treat the scaling of all shape and picture types uniformly, so that a Picture made up of various elements could be scaled with code better designed than that in the

createPictures() methods of the PictureCollection subclasses. It would be desirable to come up with a better design to “chain” shapes together in a picture in a different way. A potential design is one where each picture element “knows” what it is “attached” to. Such a design would result in more elegant solutions for picture moving and sizing.

- Animation. Drawing and erasing with TurtlePen performs extremely slow. With a faster drawing mechanism it would be easy to draw, erase and move in a repetitive fashion in order to show primitive cartoon animation. Primitive animation was developed in a proof-of-concept setting but the drawing and erasing did not occur fast enough for the animation to be obvious to the student, so development was postponed.
- Implement filled in shapes. The TurtlePen drawing mechanism would not be a good choice for implementing filled-in shapes, but the class library could be redesigned (with the redesign discussed as a lesson for the student) so that the classes use Java 2D drawing rather than the TurtlePen. The TurtlePen would remain in the current lessons but design patterns could be demonstrated in later lessons by showing how knowledge of the pen could be minimized and that the pen could ultimately be replaced altogether. Design patterns such as Bridge and Adapter could be illustrated showing how to bridge the current lesson's classes with those of the Java 2D package.
- Sort shapes according to their relative size by using a custom Comparator object. Shapes could be sorted by location on the screen or by surface

area. The intent of such a feature would be to illustrate the flexibility of the Java language, showing that even a notion of comparison can be customized.

As well as implementing new features it would also be desirable to incorporate and leverage the use of several new tools into the project and curriculum development. Version control became an issue throughout the project, and although Superversion was introduced it was not used as religiously as it should have been. Better use of labeling and project markers would make version control more successful. A very interesting visualization tool called Jeliot was discovered but there was not enough time to incorporate it into the project. With Jeliot students can step through code and watch live animations of code statement processing and movement of data. Jeliot would be an excellent tool to use to illustrate to students such concepts as parameter passing and calls to superclass constructors when a subclass object is created. Another tool which would be good to leverage is the unit testing capability of BlueJ. BlueJ has an integrated testing facility which is a subset of JUnit. It is likely that BlueJ's testing features would prove beneficial for creating and managing test programs.

5.2. Lessons Learned

Several lessons were learned on this project. From a masters' student perspective, what was learned the hard way was a wisdom of not postponing a thesis project. In hindsight, taking a break after completing non-thesis classes was a mistake. Another lesson learned as a student was to ask for help and get feedback from advisors earlier and more often.

What was learned mostly was an appreciation of methodology. The shape example had been partially developed before this project, developed almost through intuition rather than with a thought-out methodology. Much time was spent trying to salvage existing work. Early on work in parts was done almost backwards – trying to reverse engineer and discover the methodology that had been used to get part way. In hindsight what should have been done earlier on was to start at the very beginning and essentially start over. So a major lesson learned was the pain of trying to develop something without having first done a firm definition of the project. Once methodologies were consistently used progress on the project greatly increased with much less pain.

Another lesson learned was to be wary of shifting projects that are hard to define. Much energy was expended trying to tailor course development and an academic project on what was a constantly moving target. Courses that were first proposed to be developed ended up never being offered by the college they were being developed for. Concern arose that factors such as these would invalidate

the project from Regis' point of view. Much time was spent waiting, on numerous occasions until a stable environment emerged. What was originally proposed for the project – the courses, the degree they were in and even the school itself disappeared before the project could be completed. A solution that finally came to light was to design curriculum that would be flexible and general enough to supplement courses at a variety of institutions. A hard lesson learned was the lesson of building upon facets that can be controlled, rather than being dependent on factors that cannot be controlled.

A very important lesson that was discovered is how difficult it is to develop good curriculum, especially curriculum dealing with software development. Curriculum development and software development are each complex endeavors and combining them together was often a challenge. What was discovered is how much room for improvement there is in knowing curriculum development methodologies, and how valuable and appreciated these methodologies have become.

6. References

Bahrami, Ali., (1999). *Object Oriented Systems Development.* Irwin McGraw Hill.

Barnes, David J., Kölling, Michael (2003). *Objects First With Java.* Prentice Hall.

BlueJ (2003). BlueJ — The Interactive Java Environment. [online] Available: www.bluej.org (1 July 2005).

Fowler, Martin. (2003). *UML Distilled.* Addison Wesley Longman.

Hamer, John. The Lightweight Java Visualizer (LJV) [online]. Available: www.cs.auckland.ac.nz/~j-hamer/LJV.html (1 September 2005).

Lambert, Kenneth, Osborne, Martin. (2002). *Java Basics*. Addison Wesley.

Miller, R.W., Miller M.F., (2002). *Instructors And Their Jobs.* American Technical Publishers.

Moreno, A., Myller, N., Sutinen, E., Ben-Ari, M (5/25/2004) . Visualizing Programs with Jeliot 3. *Proceedings of the International Working Conference on Advanced Visual Interfaces AVI 2004* [online]. Available: http://cs.joensuu.fi/jeliot/files/avi04.pdf (20 October 2005).

Staff (9/7/2004). Superversion: Version Control for Gourmets. In *Jax Magazine* [online]. Available: http://www.jaxmagazine.com/itr/news/psecom,id,16443,nodeid,146.html (10 September 2005).

7. Appendices

7.1. Lesson Document: Introduction

Overview

The deliverable of this project is curriculum consisting of a set of lessons on object technology. The goal of the lessons is to illustrate the use of object oriented concepts through the design and development of a functional application. The lessons involve the development of a class library which deals with the application of drawing shapes. The anticipated audience of these lessons is a student taking a programming course in Java. The lessons are fully narrated and readable in a standalone setting and can therefore serve as material for either a distance or classroom-based course. It is expected that these lessons will supplement a textbook which covers Java and object concepts, as these lessons do not cover all aspects of Java syntax. These lessons are intended to illustrate and reinforce the major object-oriented concepts and demonstrate their implementation in Java in a real application, not necessarily to fully define every subtle nuance of Java's treatment of the topics. These lessons touch on many object oriented concepts. The lessons provide a framework for discussing objects, classes, encapsulation, inheritance, polymorphism, composition, association, responsibility and object persistence. The concepts are introduced and discussed as they are encountered in the development of the shape drawing application. The shape drawing application provides a problem context for discussing object oriented principles and illustrating object oriented programming approaches. The use of an evolving

example and it's modification, extension and refactoring illustrate how object oriented programming (OOP) concepts can solve typical coding problems and improve code reusability and code maintenance.

UML Model

A diagram of the static class relationships is shown below in Figure 1.

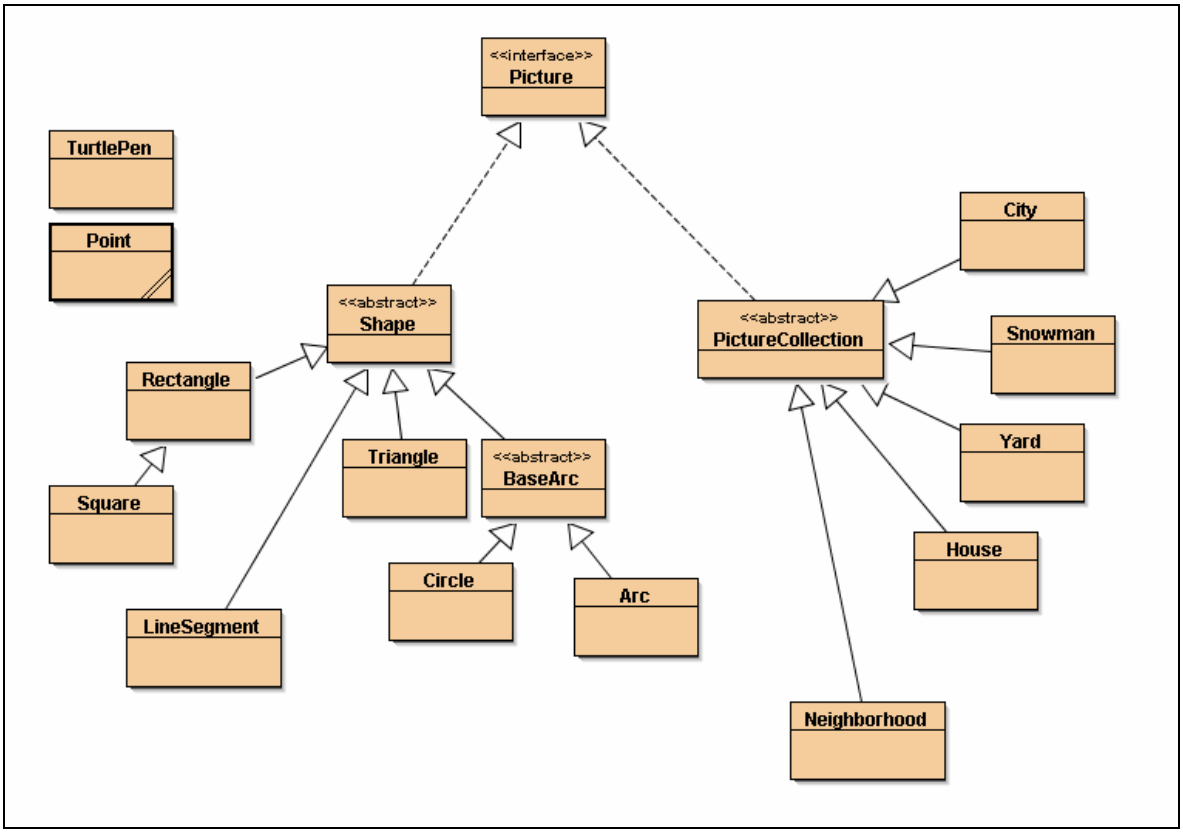


Figure 1

Need for the Project

Most texts on object orientation do not illustrate presented concepts with a fully implemented, realistic application. If examples are provided they are used briefly. A goal of the project is to have students work with the same problem example throughout the entire set of lessons. Each lesson presents an iteration of the evolving application and walks the student through the design and development of a new fully implemented version.

The lessons instruct the student to perform interactive steps using the BlueJ Java development environment. BlueJ is advantageous in that students can create and interact with live objects without writing any code. The BlueJ environment is fully documented in a tutorial provided with the software. These lessons instruct the user to work through the tutorial to become familiar with basic BlueJ operation.

Shape drawing was intentionally chosen as the problem context. The problem domain is intuitive, yet it has enough complexity to provide for opportunities for realistic solutions. Shape drawing facilitates visualization of object concepts. The project as a whole has a strong visual aspect. A highly visual application such as shape drawing as well as the use of visual techniques such as UML modeling and direct object interaction in BlueJ provide many graphic illustrations for the student.

Scope and Limitations

The lessons focus on object oriented programming concepts and therefore do not provide detailed coverage of analysis techniques such as use cases.

In a commercial shape-drawing application the end user would likely work through a front-end graphical interface that allowed them to select shapes and pictures from a palette and to drag and move and directly manipulate the shape objects. These lessons do not involve a graphical interface for drawing shapes. The code we create in these lessons is the "model" layer, and a graphical interface layer, once created, would communicate with our model layer and create objects in response to user mouse and keyboard events.

Lesson Summaries

Lesson	Title
1	Using an Existing Class
2	First Shape Classes – Circle, Triangle and Rectangle
3	The Point Class
4	Revisiting Circle, Triangle and Rectangle
5	Further Polish
6	Inheritance
7	Polymorphism
8	The Composite Pattern
9	Object Persistence

Lesson One – Using an Existing Class

In the first lesson the student is instructed to install and configure the BlueJ development environment. The student then works with an existing class. The concepts of class and instance are discussed. The student is introduced to the TurtlePen class, which is the means of drawing onto the screen. Three views of a class are presented – a class as concept, specification and implementation. This is the treatment of classes given by Martin Fowler in *UML Distilled* (Addison Wesley Longman 2003). Students learn how to create objects and call methods interactively within BlueJ and also programmatically with Java code statements.

After completing lesson one the student will be able to:

- Install and configure the BlueJ Java development environment
- Create an object on the BlueJ object workbench
- Interactively invoke methods on object instances
- Invoke methods through lines of code

Lesson Two – First Shape Classes

In the second lesson, students get shown how to build a class. The student is shown how they can represent common shapes such as circles, rectangles and triangles as classes. The notion of a class as a concept, a specification and an implementation is stressed and each shape class is considered from these three perspectives. Initial implementations of the shape classes are studied. Keeping interface separate from implementation is stressed and two different

implementations of a rectangle are contrasted, while keeping the interfaces the same.

After completing lesson two the student will be able to:

- Identify the major parts of a class definition, such as fields, constructors and methods
- Discuss a class as a concept, as a specification and as an implementation
- Create multiple instances of the same class
- Distinguish between the interface and implementation of a class

Lesson Three – The Point Class

In the third lesson the importance of having classes model the real world is stressed. It is pointed out that the natural, intuitive way of describing locations of shapes is via a singular reference, such as a circle's center point or a rectangle's corner point. Triangle's existing interface is examined, which has six integers in its constructor for specifying the three points of the triangle. The desire to express information in a more natural way is highlighted, and a Point class is introduced which allows a more natural, conversational way of describing the locations of shape objects. The shape classes are adjusted to use Point objects rather than integers, providing an opportunity to discuss class relationships in general. Class functionality and responsibility are also discussed.

After completing lesson three the student will be able to:

- Describe the role of a class as a service provider

- Use objects of one class in the definition of another class
- Make decisions about how much responsibility a class should have

Lesson Four – Revisiting Circle, Triangle and Rectangle

In the fourth lesson the shape class implementations are discussed and examined more completely, with more shape operations such as erasing, moving and changing color defined. UML class symbols are discussed. Classes are designed so that they are responsible for ensuring that objects are always in a valid state.

After completing lesson four the student will be able to:

- Identify the components of a UML class symbol
- Identify associations on a UML diagram
- Identify the operations that should be allowed for objects of a class
- Make classes that are responsible for themselves
- Code accessor methods which query an object's state
- Code mutator methods which change an object's state

Lesson Five – Further Polish

In the fifth lesson nuances of our system are addressed. Solutions to these nuances are discussed and it is decided which classes should have responsibility for certain behavior. A closer look is given to the TurtlePen class, and it is discussed how TurtlePen acts as a façade in front of a more complicated StandardPen class. It is pointed out that users do not necessarily need to know

about TurtlePen when creating shape objects. A singleton design pattern is discussed which is used to ensure that one and only one pen object is created, and that the pen is created only when it is needed. The importance of hiding implementation detail from our clients is stressed.

After completing lesson five the student will be able to:

- Appreciate intuitive class interfaces
- Hide implementation detail from the user of a class
- Explain the singleton design pattern
- Explain the façade design pattern
- Decide which class in a system should contain a given feature

Lesson Six – Inheritance

In the sixth lesson inheritance is introduced. A Square class is presented as a specialized Rectangle. A circle is discussed as being a special case of a more general Arc class. It is pointed out that conversationally the term "shape" has been used many times already in the lessons. The abstract class Shape is developed and discussed. Shape serves as a class which can contain the fields and methods common to all shapes. Inheritance solves the problem of redundant code that had existed in the shape classes.

After completing lesson six the student will be able to:

- Place behavior common to a group of classes into a new base class
- Inherit methods and fields from a base class into a derived class
- Make a specialized version of a class

- Describe the three basic class relationships
- Distinguish between abstract and concrete methods

Lesson Seven – Polymorphism

The seventh lesson focuses on polymorphism. A desire is expressed to be able to manipulate a picture which is made up of a combination of shapes. The picture should be able to be drawn, erased and moved as a singular unit. A Picture class is presented which contains a heterogeneous collection of shape objects. To implement the operations on a Picture, polymorphism is used. Essentially, to draw a picture the code cycles through the collection of shapes and asks each shape to draw itself. From Picture's perspective, what is being drawn is referred to as a generic Shape object. At run-time the appropriate draw method is called depending on the actual type of shape retrieved.

After completing lesson seven the student will be able to:

- Appreciate the power of polymorphism
- Write source code that results in dynamic runtime behavior
- Process objects polymorphically
- Use a base class variable to hold subclass object instances
- Compare polymorphic and non-polymorphic code

Lesson Eight – The Composite Pattern

In the eighth lesson it is proposed to be able to create a picture which is made up of other pictures. The essence of "what is meant by a picture" is discussed. Picture is a term is that is getting increasingly abstract. Java interfaces are

discussed as a way to define common behavior. Picture is reworked to be an interface rather than a class. A PictureCollection class is introduced which can contain either Pictures or other PictureCollections. PictureCollection as well as the Shape class both implement the Picture interface, allowing all items to be treated uniformly, whether they are a single, simple shape or a complex drawing made up of many sub drawings and pictures. After demonstrating the solution it is explained that the solution is an example of the general composite design pattern.

After completing lesson eight the student will be able to:

- Describe the Composite design pattern
- Use a Java interface to define a set of behaviors

Lesson Nine – Object Persistence

In the ninth lesson it is shown how classes can take on added responsibility. The desire to save created pictures is presented, and intelligence is added to each class in order to show how each can know how to save its state data as XML. The Java-XML integration is accomplished by making use of a third party JDOM class library.

After completing lesson nine the student will be able to:

- Work with the JDOM class library
- Store an object's state as XML data

7.2. Lesson Document: Lesson 1 Using an Existing Class

Files Needed

BreezySwing.jar, TurtleGraphics.jar, jdom.jar, v1_WorkWithTurtlePen project folder.

Deliverables

Assignment hand-in. See assignment section at the end of the lesson.

Objectives

By the end of this lesson you will be able to:

- install and configure the BlueJ Java development environment.
- create an object on the BlueJ object workbench.
- interactively invoke methods on object instances.
- invoke methods through lines of code.

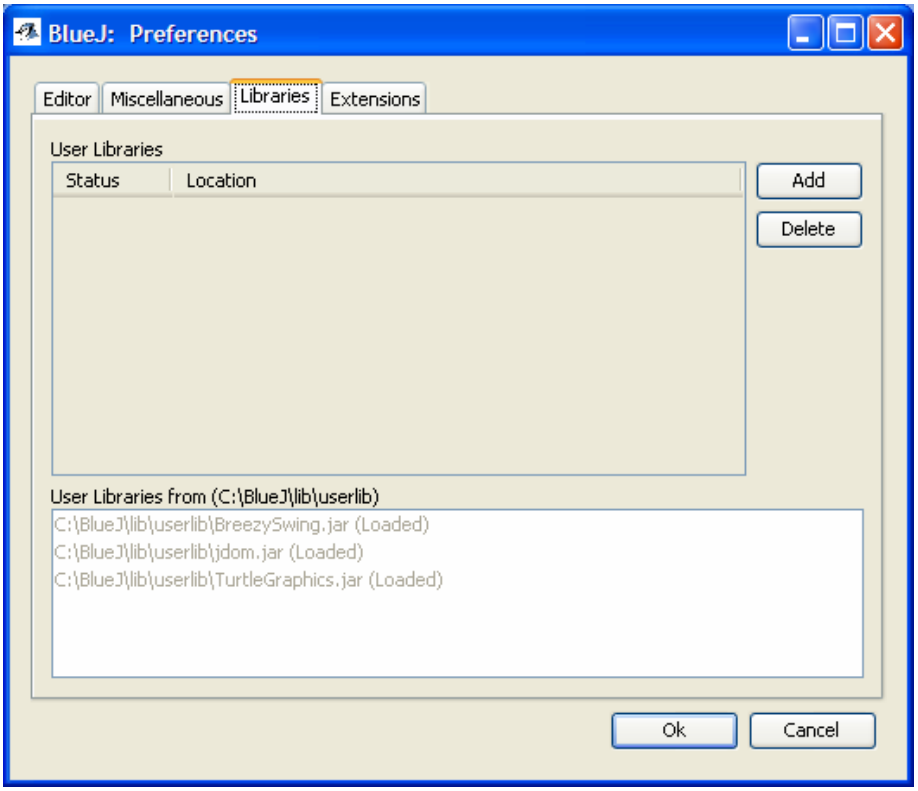
BlueJ Installation

- If you have not done so already, install the latest version of the Java JDK.
- Download and install the latest version of BlueJ from www.bluej.org .
- To become familiar with BlueJ's basic operations, work through the BlueJ tutorial, which can be found off of the BlueJ help menu, and also at the web site www.bluej.org/tutorial/tutorial.pdf .

BlueJ Configuration

In order to work through the remainder of the lessons, you will need to configure BlueJ so that it can find some class libraries that we are going to use.

- Ensure that you have received a copy of the files **BreezySwing.jar**, **jdom.jar** and **TurtleGraphics.jar** from the instructor.
- Copy these files to the folder **C:\BlueJ\lib\userlib** .
- Start or restart BlueJ, and then click on the **Tools, Preferences** menu and then click on the **Libraries** tab.
- Ensure that the screen appears as below, showing the libraries as loaded:



The TurtlePen Class

The TurtlePen class will be used as the basis for developing many other classes. The TurtlePen class will provide for us the ability to draw onto the screen. As we discuss classes during the lessons we will consider them from the three perspectives suggested by Fowler in *UML Distilled* (Addison Wesley Longman 2003):

- a class as **concept**
- a class as **specification**
- a class as **implementation**

TurtlePen as Concept

A TurtlePen is a pen used to draw in a drawing window. The TurtlePen is very similar in concept to the educational programming language called **LOGO** which uses the notion of **Turtle Graphics.** In turtle graphics, commands are issued to a "turtle", instructing the turtle to move around and to raise and lower his tail. The tail in the down position leaves a visible trail as the turtle moves around. TurtlePen is a simplification of this concept. We give instructions to a pen, rather than a turtle. We can lift the pen up, put it down and move it around. If the pen is down it will draw on the drawing surface as it is moved. The pen can be told to move to a particular spot (absolute move) or it can be told to move a certain distance in the direction it is pointing (relative move). The pen remembers its location on the drawing surface and it remembers the direction it is pointing. The color of the pen can be changed as well as the thickness of the line that is drawn.

TurtlePen as Specification

The methods that a TurtlePen object will respond to are summarized in the table below:

Method	Description
up()	Lift the pen up off of the drawing surface.
down()	Place the pen down onto the drawing surface.
setDirection(int)	Point the pen in an absolute direction. The direction is interpreted as an angle, where 90 is considered north, 180 is west, 270 is south and 0 or 360 is considered east.
turn(double)	Point the pen in a new direction, relative to its current direction. A positive parameter would indicate counter-clockwise rotation. A negative parameter would indicate clockwise rotation.
move(double)	Move the specified distance, in the current direction. If the pen is in the down position before the move, the move operation will result in a line being drawn.
move(int,int)	Move from the current position to an absolute position, specified as an x, y location. If the pen is in the down position before the move, the move operation will result in a line being drawn.
setColor(Color)	Change the color of the pen to a new color. The default color is blue. Colors are specified using a java.awt.Color constant, such as Color.red .
setWidth(int)	Change the width of the stroke that the pen draws in. The default width is 2 pixels.

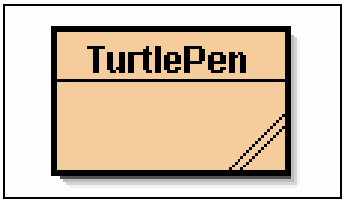
TurtlePen as Implementation

We will not examine the implementation of TurtlePen in detail in this lesson (We will investigate the internals in future lessons). TurtlePen is an adaptation (done with permission) of the StandardPen class which can be found in Lambert and Osborne's *Java Basics* text (Course Technology, 2002).

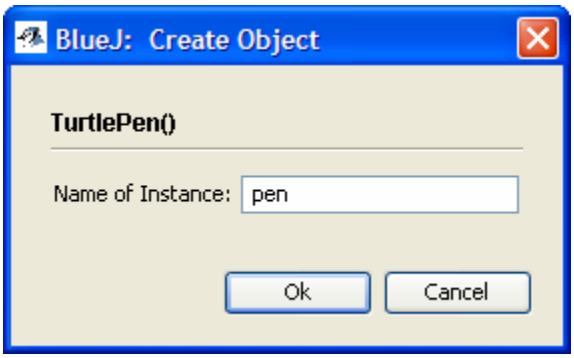
Experimentation With TurtlePen

- Open the v1_WorkWithTurtlePen project in BlueJ.

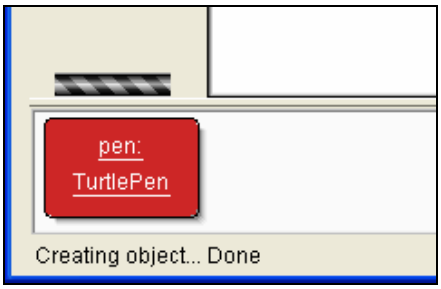
In the project window you should see a rectangle symbol with **TurtlePen** in the top part of the box. The rectangle symbol is the class symbol in UML. Our next task will be to create an object instance of type TurtlePen (in other words to create a pen object).



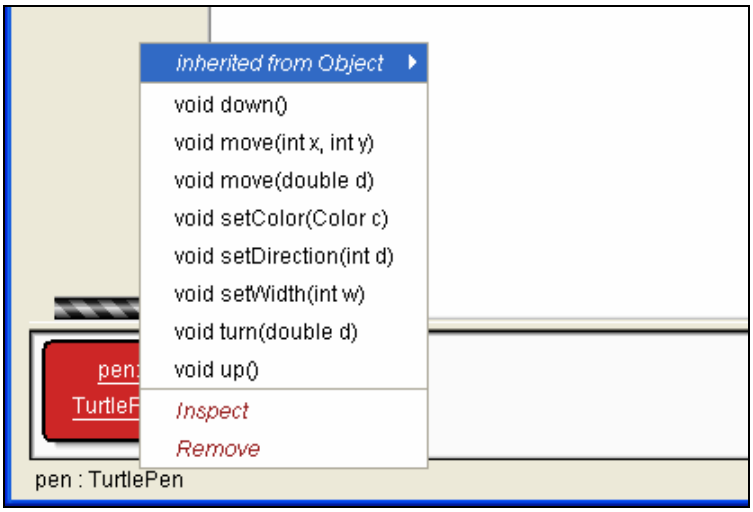
- Right click on the TurtlePen symbol and then select **new TurtlePen()** from the pop-up menu. In the dialog window that comes up give the object a more meaningful name (such as “pen”).



A reminder: object instances are made from a class. A class is the type specification that describes a category of objects. A class is a static definition. It is by making an instance of a particular class that we create something live that we can interact with. You should see a new item in the bottom portion of BlueJ (the Object Bench). The rounded rectangle is the created object instance of type TurtlePen. We will interact with this object by calling methods and observing its behavior.



- Right click on the object in the object bench. You should see a list of methods.

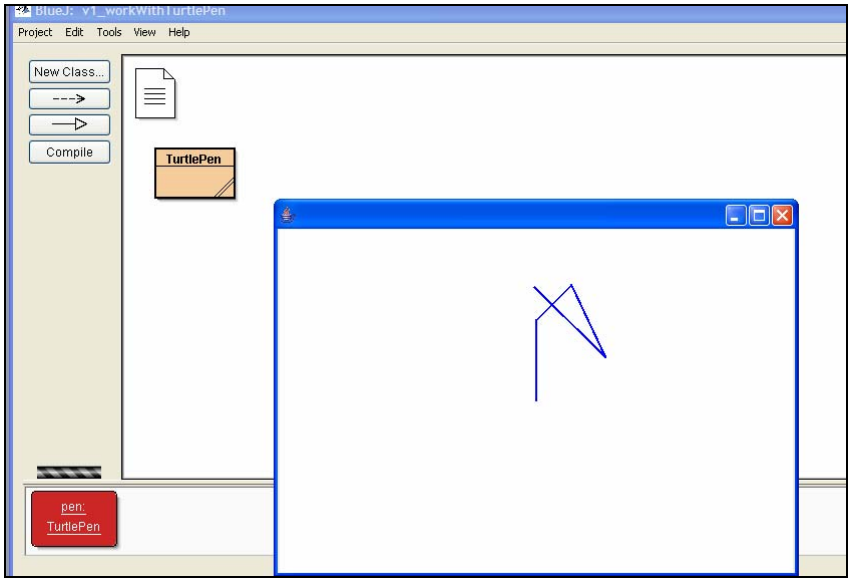


This list of methods is the **public interface** as defined by the class. It defines the operations that we can perform on the object. This list of methods for our pen object is the same list as the list of methods described in the specification section above for TurtlePen. In order for the pen to draw we need to ensure that the pen is in the **down** position. We do this by calling the down() method.

- Right click on the object in the object bench again, if necessary, to get the method list displayed. Then click on **void down()** . We have just invoked the down method on our object.

To make the pen draw we need to move the pen. We can either move to an absolute position or we can move a certain distance relative to the current position of the pen. The drawing occurs in a separate window. You will likely need to resize this window Note that the location of the origin (0, 0) is the center of the screen. The annoyances of the small initial window size and the unnatural placement of the origin will be remedied in future lessons.

- Right click on the pen object and then experiment by calling the move methods. Experiment with other methods as well, such as turning, setting direction, and changing the pen color and width. Remember that the actual drawing occurs in the secondary window, not in the main BlueJ window. If you want to move the pen to a new location without drawing then you will need to call the **up** method, move the pen using a move method and then put the pen down again when you are ready to draw.



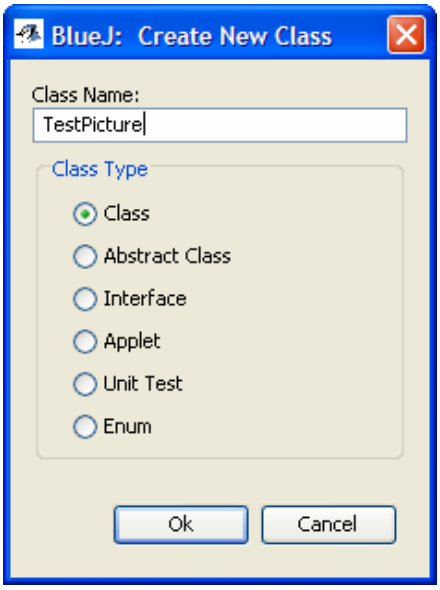
What we have done thus far is the following:

- Opened a project.
- Created an object instance by doing a **new** operation on the class.
- Called methods on the object instance.

Interacting With Objects By Writing Code

BlueJ's object bench is an excellent tool for creating objects and working with them interactively. However, suppose we want to use the pen to draw a picture. Any non-trivial picture would involve a long series of pen commands. If we make a mistake we would either have to start over or draw back over lines with a white pen. An easier way would be to use the text editor in BlueJ and type in method calls and save these commands so they can be ran again or edited. We will do this next.

- In the BlueJ main window, right click on an empty space and then select **New Class…** from the popup menu. In the dialog that comes up, type in a name such as TestPicture and then click Ok.



- You should see a rectangle symbol named TestPicture in the main BlueJ window. Double click the rectangle. The code window should open.

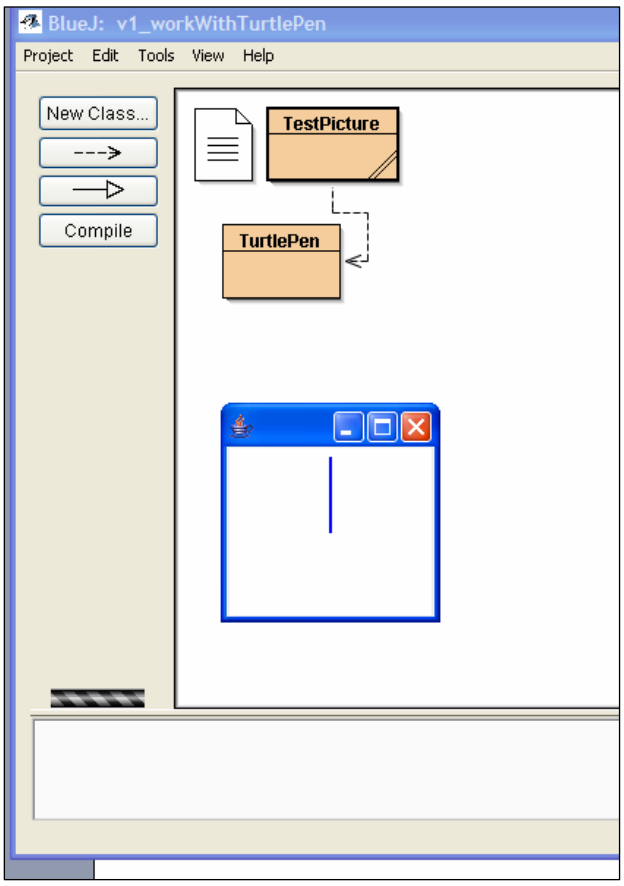
- Highlight and delete the code that you see. In its place enter the code shown below:

```
public class TestPicture
{
  public static void draw()
  {
     TurtlePen pen = new TurtlePen();
     pen.down();
     pen.move(50);
  }
}
```

- Click the **Compile** button and ensure that there are no errors. Close the code editor window.

- Right click on the TestPicture class and then click on **void draw**().

The drawing window should open and you should see a single line drawn. What we have done is we have created a small program which has code which creates a pen object and then calls two methods of the pen. The method calls are all done through code rather than working with an object interactively (note that there are no objects on the object bench).

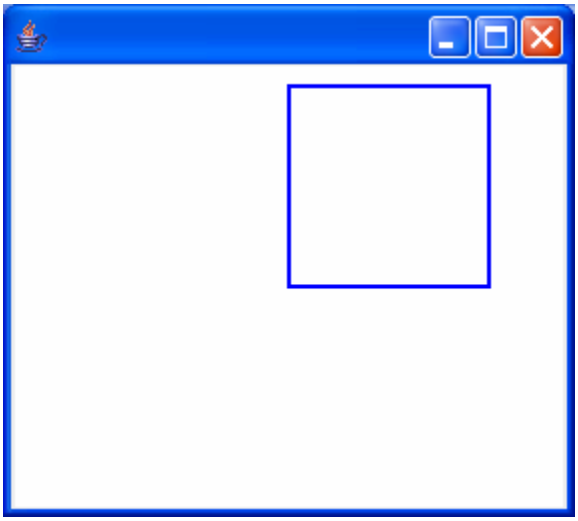


We can use this class and it's draw method as a template for making drawings. We can type in additional method calls in order to instruct the pen to do more movements.

For example, the following sequence of pen commands is one way to draw a square with a side length of 100 pixels.

```
TurtlePen pen = new TurtlePen();
pen.down();
pen.setDirection(90);  // point north
pen.move(100);  // draw right side
pen.turn(-90);  // point east
pen.move(100);  // draw top
pen.turn(-90);  // turn south
pen.move(100);  // draw left side
pen.turn(-90);  // turn west
pen.move(100);  // draw bottom
```

Note that the use of this static draw method is a "quick and easy" way to group method calls into a program structure that can be easily called from BlueJ. Designing pure classes involves much more work and involves somewhat different coding techniques. We will design pure classes in upcoming lessons.



Summary

In this lesson we were introduced to the TurtlePen class. Although we did not concern ourselves with how the class was built, we did get some experience understanding the behavior of a TurtlePen object. In upcoming lessons we will use TurtlePen objects as one of the building blocks of other classes that we create.

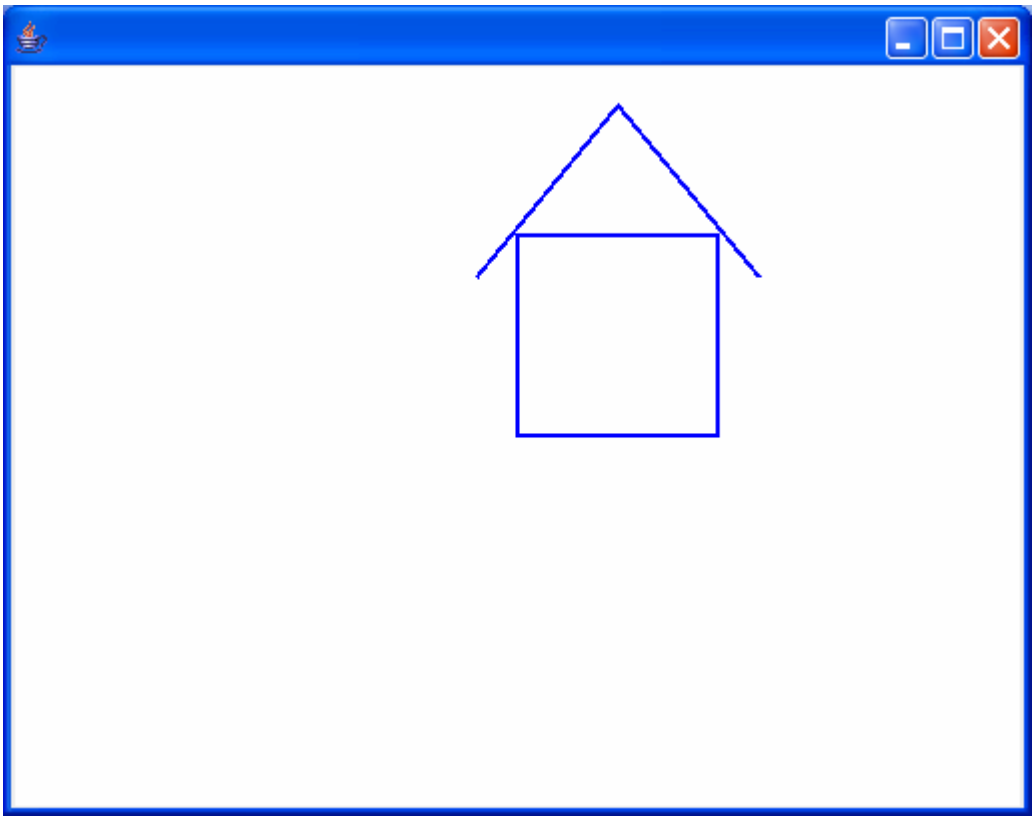
Assignment: Draw a Picture of a House

Using the last set of lesson steps as a guide, create a program which results in a simple picture of an “A” frame house being drawn, such as the one shown below.

Draw a sketch first and determine the x,y locations of significant points of the drawing.

A pure mathematical approach could be used to determine exact angles and side lengths, or a trial-and-error approach could be used.

Hand in the .java file that contains your pen method calls.



7.3. Lesson Document: Lesson 2 First Shape Classes

Files Needed

v2_firstShapeClasses project files.

Deliverables

Assignment hand-in. See assignment section at the end of the lesson.

Objectives

By the end of this lesson you will be able to:

- identify the major parts of a class definition, such as fields, constructors and methods.
- discuss a class as a concept, as a specification and as an implementation.
- create multiple instances of the same class.
- distinguish between the interface and implementation of a class.

Introduction

At the end of the last lesson, we had seen how to write code which consisted of a sequence of method calls on a TurtlePen object. We can group TurtlePen method calls together in order to create a drawing of some sort. We saw an example where we drew a square. We could group other sequences of commands together in order to create other shapes. For example, we could draw a close approximation of a circle by using a repetition or loop construct. We could draw a tiny line segment, then turn the pen slightly and draw another line segment and continue in this fashion until we had come full-circle back to where we had started. The code fragment below will draw a circle of radius 80 centered at the point 100,100.

```
TurtlePen pen = new TurtlePen();
pen.up();
pen.move(100, 100 ); // move to center
pen.move(80); // move to top
pen.turn(90); pen.move(0.8);
pen.down();

// Draw the circle.  The circle is drawn by creating
// 120 short line segments and turning the
// pen slightly ( 3 degrees ) after each segment is drawn
for (int i = 1; i <= 120; i++)
{
   pen.move(2.0 * Math.PI * 80 / 120.0 );
   pen.turn(3);
}
```

Experimentation

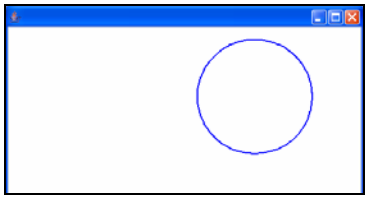
- Open the v2_firstShapeClasses project. (Ignore the Circle, Rectangle and Triangle classes for now. We will discuss them shortly.).

- Using the technique you learned in the last lesson, add a new class to the project and create a static draw method, like the code below:

```
public class TestPicture
{
  public static void draw()
  {
  }
}
```

- In the draw method, enter in the circle-drawing code which we discussed above.

- Test this code by running the draw method. You should see a circle.



Creating Classes

The approach of using a static method is not the best approach to use. We want to quickly turn towards the thought of designing a genuine Java class. Classes represent concepts. Classes are used to define categories of behaviors. Classes define what all objects in that class can do. We have used one class, TurtlePen, to draw onto the screen. It would be nice if we could create other classes to represent common shapes such as circles, triangles and rectangles. Then we could "ask for" a circle, and the circle would know how to draw itself. We then would not need to repeat the detailed steps every time and we, as a user of a circle would not need to concern ourselves with the details of exactly how the circle is drawn.

Classes as Concepts

Before we create a class we should take care that we understand fully the inherent underlying concept behind the class. We will first build classes for the following shapes: circles, rectangles and triangles.

We should be asking ourselves:

What is a circle ? What is a rectangle ? What is a triangle ?

The American Heritage dictionary gives us the following definitions:

circle	*A plane curve everywhere equidistant from a given fixed point, the center.*
rectangle	*A four-sided plane figure with four right angles.*
triangle	*The plane figure formed by connecting three points not in a straight line by straight line segments; a three-sided polygon.*

Circles, triangles or rectangles that we might draw can be drawn in different colors, in different sizes, in different locations.

If we create a class that represents the generic notion of a shape such as a circle, then we should be able to create a circle object with whatever sort of characteristics we want the circle to have. If we have a triangle class then from it we should be able to make any sort of triangle, not just specific types such as right triangles (if we only wanted right triangles perhaps we could create a class that would only be for right triangles, but the name of the class should be appropriately called something like RightTriangle).

Classes as Specifications

When writing a class, we should anticipate what users of this class will need. The user of the class must first ask the class to create an object that has the desired characteristics. The user would then call methods to interact with the object.

We must consider how it is that the user will ask for an object. (Note that "user" means "user of the class" which is not likely an end user directly, but more likely another programmer.) How would a user ask for a circle ? This in some ways is like a waiter asking a question such as "How would you like your eggs?" or "How would you like your steak?". We are, essentially, asking the user to answer the question "How would you like your circle?".

So how might a user ask for or describe the circle that they want ? The user could specify:

- the circle's size (radius length)
- the circle's color
- the circle's location
- the thickness of the line that is used to draw the circle.

A reminder: we are starting off these lessons with a simple pen object that can be moved around to draw lines. At present we do not have an easy ability to create filled-in shapes. We can, however, use the pen object to draw an outline shape of a circle.

Similarly for rectangles the user could specify:

- the size (as expressed by a height and a width)
- the color
- the location
- the thickness of the line that is used to draw the rectangle

For a circle, the "location" is intuitively the center of the circle. But what about for a rectangle ? It would be possible to consider the center of the rectangle as the location, but perhaps it may be more intuitive to consider one of the corners as the location. These decisions must be thought out carefully and the final choice should be whatever is most conceptual for the user.

How would a user specify characteristics of a triangle ? Note that there may be more than one way for our user to give us the necessary information. Often a class should support multiple means of requesting an object, if there is more than one intuitive way that would be convenient for the user. Our user could define a triangle in terms of angles and the length of a base and hypotenuse. However, there is another way that might be easier. A triangle can be defined by it's three points. The three points would determine the triangle's location and size. Like the other shapes, the user could specify the color and the line width of the triangle.

Once created, what would a user *do* with a circle or rectangle or triangle ? These desired actions will likely become the set of public methods that we provide as an interface for the class.

Shapes can be:

- drawn
- moved
- sized
- erased
- changed to a different color
- changed to have a different line thickness

In this lesson we will focus on the creation of the object and on the drawing of the object. We will address other operations in later lessons.

Classes as Implementations

In this section we will look at implementations of the three shape classes we have considered above.

At an implementation level, classes define data and methods. The class defines a set of fields which collectively store the state of an object. Each object instance has its own values stored for each of the fields. The class defines methods which act on the data in the object. Some methods are mutator methods because they change the state of the object. Other methods are called accessor methods because they do not change state – rather they are used to interrogate the object.

Examining Implementations

- In BlueJ, open up the Circle source code file. Notice the private fields that are defined at the top (also shown below):

```
private TurtlePen pen;
private Color color;
private int strokeWidth;
private int radius;
private int x; // center x
private int y; // center y
```

There is a field to store every necessary part of a circle's state. We have fields to store the location, size, color and line thickness of the circle.

Notice also that we store as a field an object of type TurtlePen. The circle relies on the pen to do the drawing.

- Open up the code window for Rectangle. The fields for Rectangle are shown below:

```
private TurtlePen pen;
private Color color;
private int strokeWidth;
private int x; // lower left x
private int y; // lower left y
private int width;
private int height;
```

Some of the fields, such as the pen, color and stroke width are the same as for Circle. Some fields, however, are specific to Rectangles. A Rectangle's location is stored as the x and y location of the lower left corner of the rectangle. While a circle's size can be stored in one field (the radius length) the dimensions of the rectangle are stored in two fields (a height and a width).

- Open up the code for Triangle. Its fields are shown below.

```
  private TurtlePen pen;
  private Color color;
  private int strokeWidth;
  private int x1;
  private int y1;
  private int x2;
  private int y2;
  private int x3;
  private int y3;
```

Like the other classes, we store a pen for drawing and we also store the color and line thickness. However, the location and size of the triangle is stored as a set of three points. Note that when using primitives we resort to storing three pairs of integers, which can get cumbersome. We will remedy this in a future lesson when we introduce the notion of a Point class which will represent a location on the drawing.

- Look at each of the three source files and compare the code in the draw methods. Each draw method has specific instructions for drawing the shape that the class represents.

The draw method for Circle contains instructions to move the pen as we described at the top of this lesson (short line segments drawn in a loop). Rectangle's draw method moves and turns the pen in order to draw the four sides of the rectangle. Triangle's draw method is fairly simple in that we draw line segments between the three points of the triangle.

The constructors for the three classes are fairly trivial. The constructors provide a way for the user of a class to supply initial values for the characteristics of the object. In these three classes the code for the constructors copies the values from the parameters into the fields. The code for the constructors is shown below.

Circle's constructor

```
  Circle(TurtlePen pen,
         int x, // center point, x
         int y, // center point, y
         int radius,
         Color color,
         int strokeWidth)
  {
    this.radius = radius;
    this.x = x;
    this.y = y;
    this.pen = pen;
    this.color = color;
    this.strokeWidth = strokeWidth;
  }
```

Rectangle's constructor

```
  Rectangle(TurtlePen pen,
            int x,
            int y,
            int width,
            int height,
            Color color,
            int strokeWidth)
  {
    this.x = x;
    this.y = y;
    this.width = width;
    this.height = height;
    this.pen = pen;
    this.color = color;
    this.strokeWidth = strokeWidth;
  }
```

Triangle's constructor

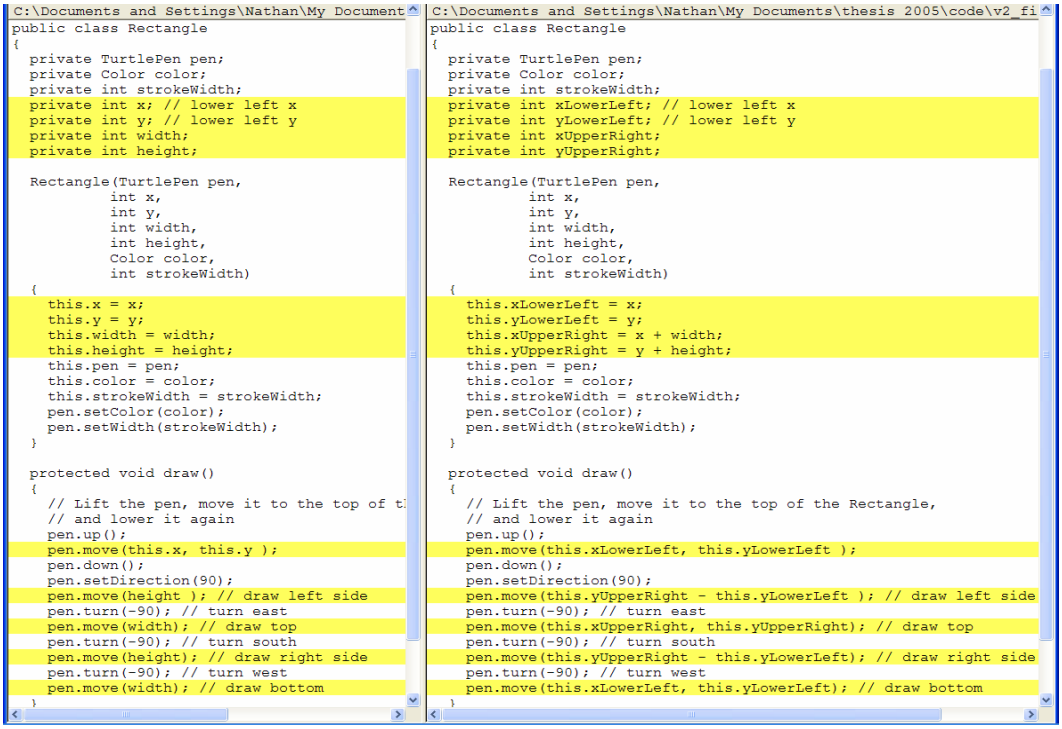
```
  Triangle(TurtlePen pen,
           int x1, int y1,
           int x2, int y2,
           int x3, int y3,
           Color color,
           int strokeWidth)
  {
    this.x1 = x1;
    this.y1 = y1;
    this.x2 = x2;
    this.y2 = y2;
    this.x3 = x3;
    this.y3 = y3;
    this.pen = pen;
    this.color = color;
    this.strokeWidth = strokeWidth;
  }
```

Interface vs. Implementation

It is important to note that we structure classes so that the interface to the class (what the user sees and is able to directly work with) is public, and the implementation is private. This allows us to change the implementation without impacting the user. As long as we keep the interface the same we can rework or rewire the internals of the class and the users will not have to change their code. Having private data is called **data hiding**. Hiding the implementation from the user so that they can't see "inside" the class is part of what is called **encapsulation**. Encapsulation also includes the concept of a class “encapsulating” or including both attributes and behavior. To demonstrate these concepts let us consider the rectangle class.

Currently we store the lower left corner location and the height and width. An alternative approach would be to store the lower left and upper right corner locations and then calculate the height and the width as needed.

The file compare below shows the first version of Rectangle on the left and the revised version on the right. Highlighted lines indicate a difference between the files. Our user gives us the lower left location and the height and the width in each version. However, in the revised version we calculate and store the upper right corner location whereas in the first version we store the height and width directly. In practice we would want to carefully analyze which implementation was better. Often there are tradeoffs between size and speed. Java developers sometimes offer different versions of a class where each version has the same interface but a different implementation. Java collection classes fit this description.



C:\Documents and Settings\Nathan\My Document

```
public class Rectangle
{
  private TurtlePen pen;
  private Color color;
  private int strokeWidth;
  private int x; // lower left x
  private int y; // lower left y
  private int width;
  private int height;

  Rectangle(TurtlePen pen,
            int x,
            int y,
            int width,
            int height,
            Color color,
            int strokeWidth)
  {
    this.x = x;
    this.y = y;
    this.width = width;
    this.height = height;
    this.pen = pen;
    this.color = color;
    this.strokeWidth = strokeWidth;
    pen.setColor(color);
    pen.setWidth(strokeWidth);
  }

  protected void draw()
  {
    // Lift the pen, move it to the top of t
    // and lower it again
    pen.up();
    pen.move(this.x, this.y );
    pen.down();
    pen.setDirection(90);
    pen.move(height ); // draw left side
    pen.turn(-90); // turn east
    pen.move(width); // draw top
    pen.turn(-90); // turn south
    pen.move(height); // draw right side
    pen.turn(-90); // turn west
    pen.move(width); // draw bottom
  }
```

C:\Documents and Settings\Nathan\My Documents\thesis 2005\code\v2_fi

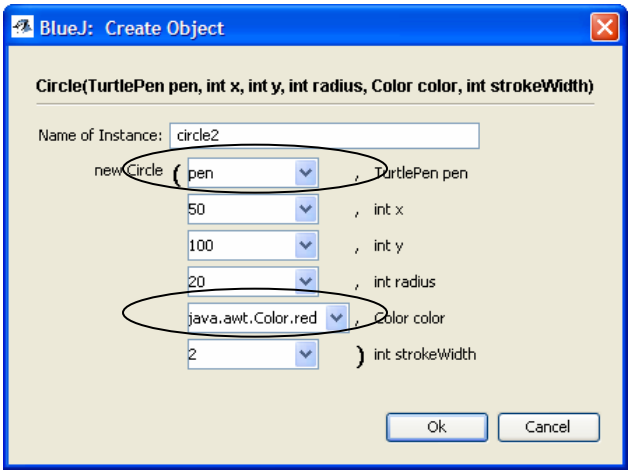
```
public class Rectangle
{
  private TurtlePen pen;
  private Color color;
  private int strokeWidth;
  private int xLowerLeft; // lower left x
  private int yLowerLeft; // lower left y
  private int xUpperRight;
  private int yUpperRight;

  Rectangle(TurtlePen pen,
            int x,
            int y,
            int width,
            int height,
            Color color,
            int strokeWidth)
  {
    this.xLowerLeft = x;
    this.yLowerLeft = y;
    this.xUpperRight = x + width;
    this.yUpperRight = y + height;
    this.pen = pen;
    this.color = color;
    this.strokeWidth = strokeWidth;
    pen.setColor(color);
    pen.setWidth(strokeWidth);
  }

  protected void draw()
  {
    // Lift the pen, move it to the top of the Rectangle,
    // and lower it again
    pen.up();
    pen.move(this.xLowerLeft, this.yLowerLeft );
    pen.down();
    pen.setDirection(90);
    pen.move(this.yUpperRight - this.yLowerLeft ); // draw left side
    pen.turn(-90); // turn east
    pen.move(this.xUpperRight, this.yUpperRight); // draw top
    pen.turn(-90); // turn south
    pen.move(this.yUpperRight - this.yLowerLeft); // draw right side
    pen.turn(-90); // turn west
    pen.move(this.xLowerLeft, this.yLowerLeft); // draw bottom
  }
```

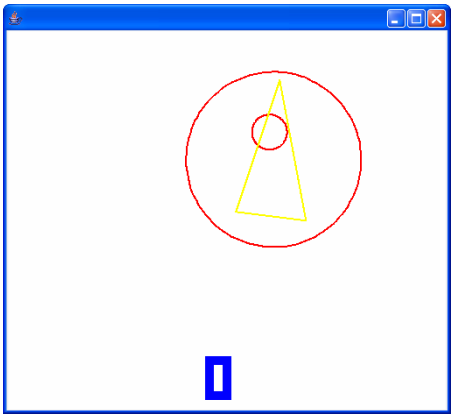
Creating Objects

- Create some objects on the object bench and then draw them.
- Create different shapes with different characteristics.

Note that you will need to create a TurtlePen object first and then, when creating a shape object you can either type in the name of the pen object or you can click on the pen object in the object bench when the cursor is blinking in the “TurtlePen pen” field box (the box circled below).

Also note that when specifying a color you will need to enter the fully qualified class name, including the package name (java.awt.Color.red for example).





Assignment: Create a Line Segment

Using the Circle, Triangle and Rectangle classes as a guide, create a new class which represents a line segment. This class should allow users to create line segment objects where a line segment is a part of a line. A line segment should have two endpoints. Recall that a line extends indefinitely and has infinite length.

Line segments should be able to be drawn in different colors and in different widths.

Add a new class to the v2_firstShapeClasses project.

Be sure to test your class by creating and drawing objects.

Hand in your project.

7.4. Lesson Document: Lesson 3 Point Class

Files Needed
v3_pointClass project files.

Deliverables
Assignment hand-in. See assignment section at the end of the lesson.

Objectives
By the end of this lesson you will be able to:

- describe the role of a class as a service provider
- use objects of one class in the definition of another class
- make decisions about how much responsibility a class should have

Introduction

In the last lesson we looked at some first implementations of classes. When designing the Circle, Rectangle and Triangle classes we had to consider questions such as:

- What *is* a triangle?
- What do rectangles *do* ? (Or, rather What can be *done* to rectangles ?)
- What do circles *have* ?
- How do you *ask* for a triangle ?
- How do you *describe* a rectangle ?
- What is a triangle *made up of* ?

Classes describe, at a specification level, what can be done to objects of that class. We will consider more fully the public interface of our shape classes in the next lesson. Currently we allow users of the class to create the shape objects and to then draw them. In the next lesson we will consider and implement other operations that can be performed on shapes.

Classes also describe, at an implementation level, what objects “look like” in memory. When implementing a class we must decide what fields values will be used to make up the object’s state. Users of the class often send along initial values for many if not all of these fields when they invoke a **new** operation on the class. Constructors serve as a place to put initialization code which will ensure that the objects get put into a valid initial state.

In our last lesson we described circles in part in terms of the circle’s **location**, which was the center of the circle. Similarly for rectangles the location was defined as the lower left corner of the rectangle. Other fields for circles and rectangles stored the size information. For triangles the three points determined both the size and location of the shape.

Note the use of singular nouns such as “location, point, center and corner” in the paragraph above. This is the natural way to discuss the notion of location. We refer to the “center of the circle”, or the “corner of the rectangle”. This is the conceptual way in which we speak. Note the difference though when we look at the implementations from the last lesson. The center of a circle is stored as two integers – an **x** and a **y** value. Certainly we need to know at some point the x and y values in order to know the precise location. However, in conversational speech we would normally refer to circles as “having a center point”, rather than “having a center point x value and a center point y value”.

A goal in object oriented programming is to have the code be readable at a fairly high level, with classes introduced to represent concepts present in the problem domain. Consider again the triangle. We can define a triangle in terms of three points. The previous sentence was the conceptual, natural description. In implementation detail we could discuss that we actually store six integers – three sets of x and y value pairs representing the three points. It would be nice to not have this difference in detail apparent every time we need to refer to the notion of a point. In other words, there are times when conceptually we would like to think of a “point” as a singular entity but in implementation detail we always have to “carry around” a couple of integers. The interface to the triangle constructor is the most obvious example we have of how the non-natural, implementation-detail approach gets cumbersome. The user of our Triangle class has to supply a list of six integers which represent the three points, taking care to list the six integers in the correct order.

Point Class As Concept

In an attempt to have our software model include concepts apparent in the problem domain, we will introduce the notion of a Point class and then adapt our earlier shape classes to use the Point class rather than pairs of integers. A Point object simply represents a location on the drawing. It "wraps" an x and a y value into a single unit. Once we have this notion of a point we can then pass along a Point object whenever we need to describe another object in terms of its location. Circles then will be defined as having a size, a color and a location, where the location is specified as a Point object. Rectangles will have a height, a width, a color and a Point location object. Triangles will be defined, both conceptually and in implementation in terms of three Point location objects. It may seem like overkill to introduce a Point class to replace a pair of integers. However, having a class which represents a real-world concept and having a class which can take on appropriate responsibility for tasks related to that concept creates an opportunity for robust and intuitive solutions.

Point Class As Specification

When considering the public interface of the Point class we need to consider what our users will need to do with Point objects. Users of the Point class will need to:

- create a point and specify the x and the y values
- get the x value from a point
- get the y value from a point

Should the user be able to change the x or the y values of a point ? This is a philosophical question. Changing the x or the y value of a point in essence creates a new point, a new location. Should the interface of our class include an ability to change the x and y values of a Point object, or should the user be required to create a new Point instance if they need a new location ? Will users frequently wish to create points that move around ? The phrase "new location" seems to imply creation, so for this version of the Point class we will consider that Point objects are immutable locations with the coordinates defined at construction. We will not allow users to change a Point object. If a new location is needed then a new Point object will need to be created.

Are there any other anticipated uses of the Point class ? Consider the line segment shape. A line segment can be defined in terms of two points. But what if when creating a line segment duplicate points are supplied ? Would we then have a true line segment ? The line segment class should have the responsibility to ensure that the two points defining the shape are not identical. What does it mean for two points to be identical ? Two point

objects represent the same location if both their x and y values are the same. Certainly clients of Point such as the line segment could perform such a test with code similar to below, assuming the existence of **x** and **y** methods in Point:

```
if ( pointA.x() == pointB.x() && pointA.y() == pointB.y() )
   // then the two points are equal
```

This code is manageable, but is it necessary for all clients of Point to have to write this logical test every time they want to compare two points ? Consider the triangle class, which define triangles in terms of three points. Certainly all three points must be unique in order to have a legal triangle object.

Assuming the three points contained in triangle are defined as p1, p2 and p3, the validation code to check for duplicate points could look something like this:

```
if ( ( p1.x() == p2.x() && p1.y() == p2.y() ) ||
     ( p1.x() == p3.x() && p1.y() == p3.y() ) ||
     ( p3.x() == p2.x() && p3.y() == p2.y() )

        // we've found two points that are identical
```

Taking the consideration a step further, what about irregular shapes which are defined in terms of their vertices ? Some shapes could have several points. **If** statements such as those shown above would quickly become too cumbersome.

A solution would be for Point classes to define an equality operation. The Point class could supply a method which would compare the Point object with another Point object and return true if the two locations were equal.

Assuming that the method is called **equals** then the validation code in line segment would look like:

```
if ( pointA.equals(pointB) )
  // points are identical
```

The validation code in Triangle would change to:

```
if ( p1.equals(p2) || p1.equals(p3) || p3.equals(p2) )
        // we've found two points that are identical
```

Note: Other object-oriented languages such as C++ which support operator overloading can make the code even more condense since the behavior of the equality operator itself (==) can be customized. C++ code for the triangle duplicate point check could look like:

```
if ( p1==p2 || p1==p3 || p3==p2 ) // identical points
```

The use of an equals method is actually standard practice in Java. Java collection classes will look for an equals method and they will call it when necessary, such as for ensuring uniqueness in collections that don't allow duplicates. The equals method is defined in Java's top-level Object class, and any class in Java can define its own notion of equality by providing an equals method. The exact coding mechanism that makes the equals method work will be discussed when we consider the concept of inheritance in future lessons.

The theme of this discussion has been that it is advantageous to put behavior into a class if it makes coding more convenient for users of the class. Are there any other services that the Point class could provide ? How would points be printed if we wanted to print location or shape information somewhere ? A conventional way to display point information is to list the x and y values in parentheses, separated by a comma. For example, the point with x value 10 and y value 30 would be shown as:

(10, 30)

If we wanted a Point object printed or displayed in this manner then we could piece a String together with code such as:

```
String formattedPoint = "( " + somePoint.x() + ", " + somePoint.y() + " )" ;
```

We could require clients of Point to do this string concatenation code every time or we could provide a **toString** method in Point that clients could simply call if they wanted to display the point information in a formatted way.

To display a Point object as a formatted string clients would simply code:

```
somePoint.toString()
```

Like the equals method, the presence of a toString method in a class is recognized by the compiler and the compiler can recognize situations where the toString method should be called implicitly, whenever an object should be treated or represented as a String.

The interface to the Point class is summarized in the table below:

Method	Notes
new Point (int x, int y)	Create a point with the given x and y values
int x()	Return the x value of a point
int y()	Return the y value of a point
boolean equals()	Compare a point to another point
String toString()	Return the point information as a formatted String

Before we look at the implementation of the Point class, it would be good (as it is good to do for any class) to confirm our class interface by considering in more detail client code. Circle, Triangle and Rectangle will all be clients of Point, since each shape will be defined in part by its location. Shown below are the pertinent parts of each shape class that needed changing to utilize Point. The old code is on the left and the new versions using Point are shown on the right. Notice that each class now accepts Point objects as parameters to the constructors. Clients supply Point objects to define the locations of the shapes. The classes also maintain this information in fields of data type Point. When drawing the shapes, the classes call the x() and the y() methods in order to pass the coordinate information to the TurtlePen class.

- Examine these code comparisons and also open up and study the full source files of Circle, Triangle and Rectangle in the v3_pointClass project.

Circle class adapted to use Point.

```
public class Circle
{

  private TurtlePen pen;
  private Color color;
  private int strokeWidth;
  private int radius;
  private int x; // center x
  private int y; // center y

  Circle(TurtlePen pen,
         int x, // center point, x
         int y, // center point, y
         int radius,
         Color color,
         int strokeWidth)
  {
    this.radius = radius;
    this.x = x;
    this.y = y;
    this.pen = pen;
    this.color = color;
    this.strokeWidth = strokeWidth;
    pen.setColor(color);
    pen.setWidth(strokeWidth);
  }
```

```
public class Circle
{

  private TurtlePen pen;
  private Color color;
  private int strokeWidth;
  private int radius;
  private Point center;


  Circle(TurtlePen pen,
         Point center,

         int radius,
         Color color,
         int strokeWidth)
  {
    this.radius = radius;
    this.center = center;

    this.pen = pen;
    this.color = color;
    this.strokeWidth = strokeWidth;
    pen.setColor(color);
    pen.setWidth(strokeWidth);
  }
```

```
  protected void draw()
  {
    // Lift the pen, move it to the top of the circle,
    // and lower it again
    pen.up();
    pen.move(this.x, this.y );
    pen.move(radius);
    pen.turn(90); pen.move(0.8);
    pen.down();

    // Draw the circle
    // The circle is drawn by creating
    // 120 short line segments and turning the
    // pen slightly ( 3 degrees ) after
    // each segment is drawn
    for (int i = 1; i <= 120; i++)
    {
       pen.move(2.0 * Math.PI * radius / 120.0 );
       pen.turn(3);
    }
  }
```

```
  protected void draw()
  {
    // Lift the pen, move it to the top of the circle,
    // and lower it again
    pen.up();
    pen.move(this.center.x(), this.center.y() );
    pen.move(radius);
    pen.turn(90); pen.move(0.8);
    pen.down();

    // Draw the circle
    // The circle is drawn by creating
    // 120 short line segments and turning the
    // pen slightly ( 3 degrees ) after
    // each segment is drawn
    for (int i = 1; i <= 120; i++)
    {
       pen.move(2.0 * Math.PI * radius / 120.0 );
       pen.turn(3);
    }
  }
```

Rectangle class adapted to use Point.

```
public class Rectangle
{
  private TurtlePen pen;
  private Color color;
  private int strokeWidth;
  private int x; // lower left x
  private int y; // lower left y
  private int width;
  private int height;

  Rectangle(TurtlePen pen,
            int x,
            int y,
            int width,
            int height,
            Color color,
            int strokeWidth)
  {
    this.x = x;
    this.y = y;
    this.width = width;
    this.height = height;
    this.pen = pen;
    this.color = color;
    this.strokeWidth = strokeWidth;
    pen.setColor(color);
    pen.setWidth(strokeWidth);
  }
```

```
public class Rectangle
{
  private TurtlePen pen;
  private Color color;
  private int strokeWidth;
  private Point lowerLeft;

  private int width;
  private int height;

  Rectangle(TurtlePen pen,
            Point lowerLeft,

            int width,
            int height,
            Color color,
            int strokeWidth)
  {
    this.lowerLeft = lowerLeft;

    this.width = width;
    this.height = height;
    this.pen = pen;
    this.color = color;
    this.strokeWidth = strokeWidth;
    pen.setColor(color);
    pen.setWidth(strokeWidth);
  }
```

```
  protected void draw()
  {
    pen.up();
    pen.move(this.x, this.y );
    pen.down();
    pen.setDirection(90);
    pen.move(height ); // draw left side
    pen.turn(-90); // turn east
    pen.move(width); // draw top
    pen.turn(-90); // turn south
    pen.move(height); // draw right side
    pen.turn(-90); // turn west
    pen.move(width); // draw bottom
  }
```

```
  protected void draw()
  {
    pen.up();
    pen.move(lowerLeft.x(), lowerLeft.y() );
    pen.down();
    pen.setDirection(90);
    pen.move(height ); // draw left side
    pen.turn(-90); // turn east
    pen.move(width); // draw top
    pen.turn(-90); // turn south
    pen.move(height); // draw right side
    pen.turn(-90); // turn west
    pen.move(width); // draw bottom
  }
```

Triangle class adapted to use Point

```
public class Triangle
{

  private TurtlePen pen;
  private Color color;
  private int strokeWidth;
  private int x1;
  private int y1;
  private int x2;
  private int y2;
  private int x3;
  private int y3;

  // constructor -- initializes the object
  Triangle(TurtlePen pen,
           int x1,
           int y1,
           int x2,
           int y2,
           int x3,
           int y3,
           Color color,
           int strokeWidth)
  {
    this.x1 = x1;
    this.y1 = y1;
    this.x2 = x2;
    this.y2 = y2;
    this.x3 = x3;
    this.y3 = y3;
    this.pen = pen;
    this.color = color;
    this.strokeWidth = strokeWidth;
    pen.setColor(color);
    pen.setWidth(strokeWidth);
  }
```

```
public class Triangle
{

  private TurtlePen pen;
  private Color color;
  private int strokeWidth;
  private Point p1;
  private Point p2;
  private Point p3;


  // constructor -- initializes the object
  Triangle(TurtlePen pen,
           Point p1,
           Point p2,
           Point p3,


           Color color,
           int strokeWidth)
  {
    this.p1 = p1;
    this.p2 = p2;
    this.p3 = p3;


    this.pen = pen;
    this.color = color;
    this.strokeWidth = strokeWidth;
    pen.setColor(color);
    pen.setWidth(strokeWidth);
  }
```

```
  protected void draw()
  {
    // Lift the pen, move it to the top of the Triangle
    // and lower it again
    pen.up();
    pen.move(this.x1, this.y1 );
    pen.down();
    pen.move(this.x2, this.y2 );
    pen.move(this.x3, this.y3 );
    pen.move(this.x1, this.y1 );
  }
```

```
  protected void draw()
  {
    // Lift the pen, move it to the top of the Triangl
    // and lower it again
    pen.up();
    pen.move(this.p1.x(), this.p1.y() );
    pen.down();
    pen.move(this.p2.x(), this.p2.y() );
    pen.move(this.p3.x(), this.p3.y() );
    pen.move(this.p1.x(), this.p1.x() );
  }
```

Point Class As Implementation

We will now briefly discuss each of the sections of the Point class implementation.

- Read the discussion below and also open and study the full source file for the Point class in the v3_pointClass project.

The fields for the Point class are two integers – one integer for the x, or horizontal value and one integer for the y, or vertical value.

```
  private int x;
  private int y;
```

The Point class supplies two constructors. One constructor, which takes no arguments, initializes the point to be the origin (0, 0). The other constructor initializes a point to the passed in x and y values.

```
   public Point()
   {
      x = y = 0;
   }

   public Point( int x, int y )
   {
       this.x = x;
       this.y = y;
   }
```

Accessor, or "getter" methods are supplied to "get" or "query" the x and y values stored in the object.

```
   public int x() { return x; }
   public int y() { return y; }
```

The toString method returns the location information as a formatted string.

```
   public String toString()
   {
     return "(" + x() + ", " + y() + ")";
   }
```

The equals method compares the acted on Point object to another Point object and returns true if the two objects represent the same location.

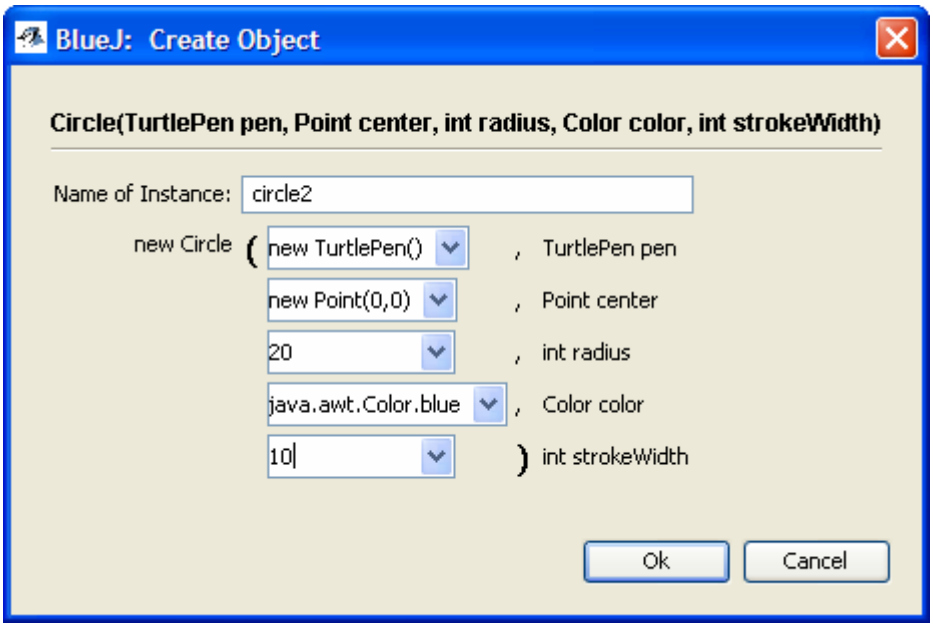
```
    public boolean equals(Point otherPoint)
    {
        if ( this.x() == otherPoint.x() &&
             this.y() == otherPoint.y() )
                return true;
        else
                return false;
    }
```

Using the Point Class

As discussed earlier in this lesson, when a Circle, Triangle or Rectangle is created one or more Point objects must be passed at construction time to define the location of the shape. We will create some shape objects now, both interactively in BlueJ and programmatically, in a Java test program.

- In BlueJ, with the v3_pointClass project open, create a Circle object.
- When the Create Object dialog appears, fill in the field information as shown below (you may name your circle instance whatever you would like).

The dialog below shows how to create objects such as the TurtlePen and the center point of the circle "on the fly" when the circle is created, rather than first creating the pen and point objects on the object bench. The expressions **new TurtlePen()** and **new Point(0,0)** create the pen and point objects, respectively.



- Create a rectangle object. Notice that, like Circle, one point object is needed to define the location of the shape.

- Create a triangle object. You will want to determine the points of the triangle ahead of time, and then create three Point objects as you are creating the triangle.

Typing in expressions like "new Point(20,80)" in BlueJ may seem more involved than is
necessary and may seem like more work than typing in 20 and 80 separately (like we did before the introduction of the Point class).

Creating objects interactively in BlueJ is not the typical way that objects are created in production use of Java. Objects are, of course, normally created by Java program code. We will now examine a Java program which creates Point objects programmatically. The program should illustrate how the use of the Point class adds organization to our program.

The program draws a simple picture of a house. In order to draw the picture, a sketch was made on paper and then critical points on the sketch were identified. Code was written to create Point objects corresponding to the points identified on the sketch. The Point objects were given meaningful names. The rest of the drawing was created by making various shape objects which referenced the point objects. Sections of the code are shown and discussed below.

- Right click on the **House** class in BlueJ.

- Click on the **draw()** method in order to show the picture.

- Open up the House source code and study the code and also read the commentary below.

Point objects are made which represent key locations on the drawing.

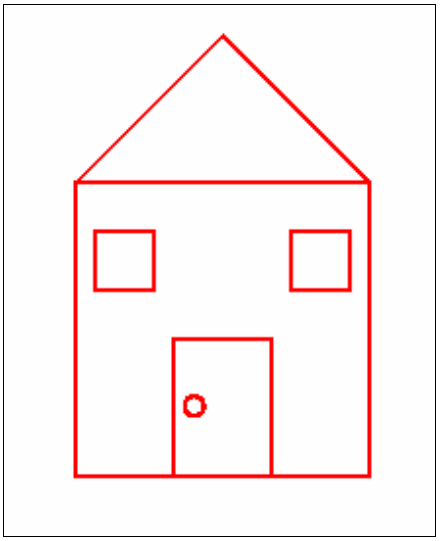
```
Point houseCorner = new Point(0,0);
Point leftWindowCorner = new Point(10,95);
Point rightWindowCorner = new Point(110,95);
Point roofLeft = new Point(0,150);
Point roofRight = new Point(150,150);
Point roofTop = new Point(75,225);
Point doorknobCenter = new Point(60,35);
Point doorCorner = new Point(50,0);
```

Various shape objects are created, referencing the appropriate Point objects.

```
Rectangle house = new Rectangle(pen,houseCorner,150,150,red,2);
Rectangle leftWin = new Rectangle(pen,leftWindowCorner,30,30,red,2);
Rectangle rightWin = new Rectangle(pen,rightWindowCorner,30,30,red,2);
Rectangle door = new Rectangle(pen,doorCorner,50,70,red,2);
Triangle roof = new Triangle(pen,roofLeft,roofTop,roofRight,red,2);
Circle doorknob = new Circle(pen,doorknobCenter,5,red,2);
```

The picture is drawn by drawing each of the shape objects.

```
house.draw();
leftWin.draw();
rightWin.draw();
door.draw();
roof.draw();
doorknob.draw();
```



Assignment

Using the House class as a guide, create a picture of a cat's face. The drawing can be simple, but it should include:

- Eyes
- Ears
- A nose
- A mouth
- Whiskers

Add your version of LineSegment (from the last lesson) to the v3_pointClass project (click on Edit, Add Class from File) so that you can include LineSegment objects in your drawing.

Create a sketch and figure out the shapes you will need and where the shapes would appear on your picture.

7.5. Lesson Document: Lesson 4 Revisiting Circle, Triangle and Rectangle

Files Needed

v4_secondShapeClasses project files.

Deliverables

Assignment hand-in. See assignment section at the end of the lesson.

Objectives

By the end of this lesson you will be able to:

- Identify the components of a UML class symbol
- Identify associations on a UML diagram
- Identify the operations that should be allowed for objects of a class
- Make classes that are responsible for themselves
- Code accessor methods which query an object's state
- Code mutator methods which change an object's state

Introduction

In this lesson we will revisit our Circle, Triangle and Rectangle classes and give more consideration to their public interfaces. Currently, what we can do with these objects is to create them and to draw them. What other operations would be logical for users to perform on these shapes ? The user should be able to:

- erase a shape
- move a shape
- change a shape's size
- change a shape's color
- change the width of the line used to draw the shape (the stroke width)

The erase, color change and line width change operations would make sense to be performed on any of the three shapes. Moving a circle or rectangle would be straightforward since both shapes are located at a specific point. Moving to a new location would involve specifying the new point. Moving a triangle would

require a little more definition because a triangle has three points and it has not been identified which of the points would be used as the basis for a move. What we can do is to consider the first point the user gives us the reference point for the triangle. Changing the size of a circle would involve changing the radius of the circle. Changing the size of a rectangle would involve either changing the height or changing the width. Changing the size of a triangle would be hard to define without advanced math topics so triangle sizing will not be addressed in this lesson.

Operations such as the ones discussed above change the state of the objects. Such methods mutate, or change the object and are therefore called **mutator** methods. There are times when a client of a shape does not need to change the shape but does need to query the object for information. These methods are called **accessor** methods. We should be able to ask a shape object where it is, what size it is, what color it is and what its line thickness is. Like the mutator methods, some accessor methods would be included in all three shape classes and some would be specific to a certain shape type.

A summary of the methods to be included in the Circle, Triangle and Rectangle classes is shown below:

Circle class interface

Method	Notes
void draw()	Draw the circle
void erase()	Erase the circle
void moveTo(Point)	Move the circle to a new location
void setRadius(int)	Change the length of the radius
void setColor(Color)	Change the color of the circle
void setStrokeWidth(int)	Change the width of the line used to draw the circle
Color getColor()	Query the current color
int getRadius()	Query the current radius length
int getStrokeWidth()	Query the current stroke width
Point getLocation()	Query the current location

Rectangle class interface

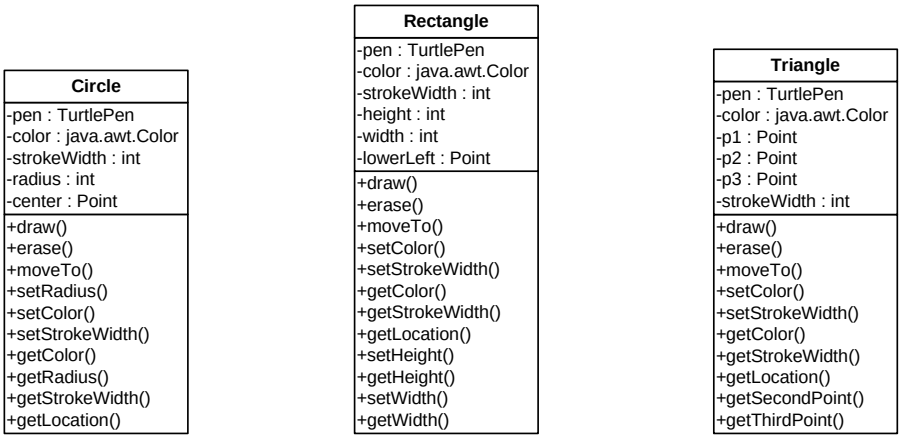
Method	Notes
void draw()	Draw the rectangle
void erase()	Erase the rectangle
void moveTo(Point)	Move the rectangle to a new location
void setWidth(int)	Change the width of the rectangle
void setHeight(int)	Change the height of the rectangle
void setColor(Color)	Change the color of the rectangle
void setStrokeWidth(int)	Change the width of the line used to draw the rectangle
Color getColor()	Query the current color
int getHeight()	Query the current height
int getWidth()	Query the current width
Point getLocation()	Query the current location
int getStrokeWidth()	Query the current stroke width

Triangle class interface

Method	Notes
void draw()	Draw the triangle
void erase()	Erase the triangle
void moveTo(Point)	Move the triangle to a new location. One point of the triangle must be identified as an anchor point.
void setColor(Color)	Change the color of the triangle
void setStrokeWidth(int)	Change the width of the line used to draw the triangle
Color getColor()	Query the current color
int getStrokeWidth()	Query the current stroke width
Point getLocation()	Query the anchor point of the triangle
Point getSecondPoint()	Query the second point of the triangle
Point getThirdPoint()	Query the third point of the triangle

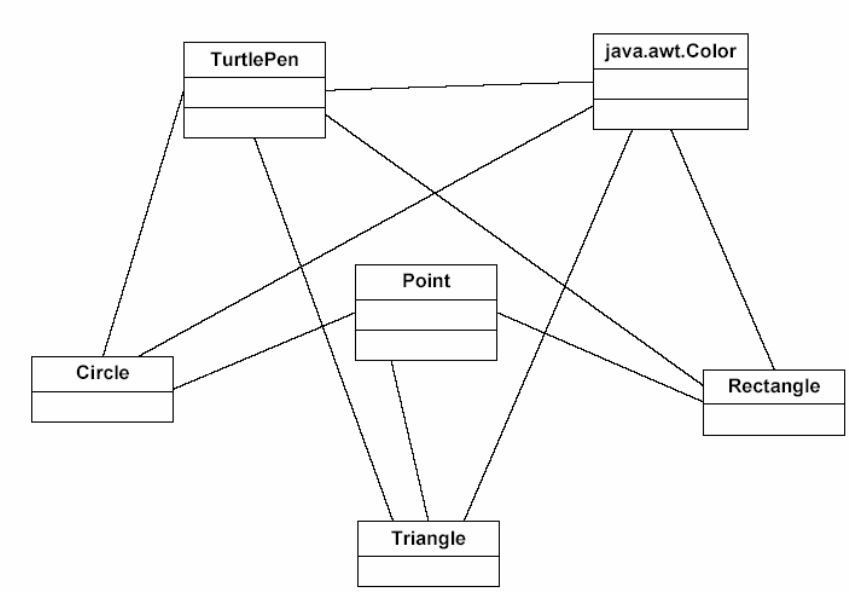
UML Class Diagrams

Below are UML class diagrams for the three classes. The class diagram is a rectangle with three compartments. The top compartment contains the class name. The middle compartment is used to list fields. The developer can show or hide detail on a class diagram depending on communication style and purpose. The third compartment lists the methods of the class. The plus sign indicates that the methods are public methods (in contrast to private methods which are not accessible to clients of the class).



Showing Class Relationships

We can show other detail in our diagrams, such as showing relationships between classes. Each shape object uses other classes, such as the TurtlePen, Point and Color classes. Lines between classes indicate that there is an association between the classes.



Implementations

We will first examine the implementations of methods that are common to all three shape classes.

Let's examine the accessor methods first. Accessor methods provide read access to a characteristic about an object. Often but not always, an accessor method returns a copy of the value of a field (An exception would be if a method calculated and returned a result). There are three accessor methods common to all three classes: getColor(), getStrokeWidth() and getLocation(). Each of these methods follows a similar pattern.

The getColor() and getStrokeWidth() methods are simple and are shown below.

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

  public int getStrokeWidth() { return strokeWidth; }
```

Although these methods are not doing anything other than returning the value of a field variable, the fact that our user must call the method (rather than access the field directly) means that we could put more complicated code in the method body at a later date. For example, currently in the Point class the x() and y() methods simply return a field value. In a later lesson we will revisit shape locations and we will do calculations and mapping of coordinates in order to work with the origin as the lower left of the screen rather than the center. We could replace the "return x;" or "return y;" lines of code in the accessor methods with calculations to peforming the mapping of coordinates. The user, however, would still code to the same interface as before.

The getLocation() method, which is in every shape, has a slightly different implementation in each class due to a naming difference for the Point field. In Circle, the location field is called **center**. In Rectangle it is called **lowerLeft** and in Triangle we are considering the first point given by the user to be the location, and the name of this field is **p1**. Implementations for these three methods are shown below.

```
 public Point getLocation() { return center; }    // in Circle

 public Point getLocation() { return lowerLeft; } // in Rectangle

 public Point getLocation() { return p1; }        // in Triangle
```

Note: By now you have likely noticed that the three shape classes have a lot in common. They have numerous methods and fields which are either identical or else very similar. In a future lesson we will look at techniques such as inheritance which we will use to factor out the commonality rather than have it repeated in multiple locations.

Each of the accessor operations has a corresponding "change" or "set" operation. However, the change operations alter the appearance of the shape, which would require that the shape be redrawn. Of course we cannot simply redraw the shape because the old shape would still be on the screen. We must first erase the shape and then redraw it with its new characteristics.

How might we go about implementing an erase operation ? Recall that our underlying method of drawing is the use of the TurtlePen, and that all we can really do with the TurtlePen is lift it up and put it down and move it around. We accomplished drawing by moving the pen while the pen was in the down position. If these are the only features that we have to work with then how might we "undo" a pen movement ? One possibility is to redo all of the movements with a pen with a color of white. Since the background of our drawing window is white, drawing over lines with a white pen should theoretically erase everything that we had drawn before.

In our current code for our shape classes we set the pen to be the shape color in the draw() method. We need to set the color each time because we are using one pen to draw every shape. However, if we set the color in the draw() method then how do we draw a shape in white in order to erase it ? We will need to break apart the methods somewhat. We need a way to draw the shape in more than one color. One way is to make our current draw() method a draw**In**() method, which accepts a color parameter. Then we can code a new draw() method which will draw the shape in the current color and we can create an erase() method which will draw the shape in white. Below are implementations of draw() and erase() that will work for all three shape classes. The old draw() methods are renamed to be drawIn() and they are also marked **private** since they are no longer intended to be part of the public interface (draw() and erase() are the public methods).

- Open up Triangle, Circle and Rectangle and study the draw(), drawIn() and erase() methods for each class.

```
public void draw()
{
    drawIn(this.color);
}

// a Shape is erased by temporarily setting the
// color to the background color ( white ),
// and then drawing in the background color
public void erase()
{
  drawIn(Color.white);
}
```

- In BlueJ, create a circle object and then draw it. Next, call the erase method. The circle should have disappeared. Call draw(). The circle should reappear.

Now that we have an ability to erase shapes, we can implement the methods which alter the appearance of the shape, such as changing color, location or line width. Each of these methods can be implemented using the following general pattern:

- erase the shape
- change the characteristic of the shape (color, location or line width)
- redraw the shape

The implementations are shown below.

```
  public void setStrokeWidth(int w)
  {
    erase();
    this.strokeWidth = w;
    draw();
  }

  public void setColor(Color color)
  {
    erase();
    this.color = color;
    draw();
  }
```

Just as the getLocation() method had to be coded slightly different due to field naming differences, so does the moveTo() method. Below are the implementations for Circle and for Rectangle:

```
  // For Circle
  public void moveTo(Point newLocation)
  {
    erase(); // erase ( at old location )
    center = newLocation; // change location
    draw();  // draw ( at new location )
  }
```

```
  // For Rectangle
  public void moveTo(Point newLocation)
  {
    erase(); // erase ( at old location )
    lowerLeft = newLocation; // change location
    draw();  // draw ( at new location )
  }
```

Moving a triangle is a bit more involved than simple changing the value of one point and redrawing. The second and third points have to be adjusted by the same relative amount that the first point was moved. The code for Triangle's moveTo is shown below:

```
  public void moveTo(Point newLocation)
  {
    erase(); // erase ( at old location )

    // Calculate the amount to move
    int xAmount = newLocation.x() - p1.x();
    int yAmount = newLocation.y() - p1.y();
    p1 = newLocation; // change location
    p2 = new Point(p2.x() + xAmount, p2.y() + yAmount);
    p3 = new Point(p3.x() + xAmount, p3.y() + yAmount);
    draw();  // draw ( at new location )
  }
```

The remaining method implementations to examine in Circle are a getter and a setter for the radius field:

```
  public int getRadius() { return radius; }

  public void setRadius(int radius)
  {
      erase();
      this.radius = radius;
      draw();
  }
```

For Rectangle, we need methods to set or query the height and width:

```
  public int getHeight() { return height; }
  public void setHeight(int height)
  {
      erase();
      this.height = height;
      draw();
  }
```

```
public int getWidth() { return width; }
public void setWidth(int width)
{
    erase();
    this.width = width;
    draw();
}
```

For Triangle we need accessor methods for querying the second and third points:

```
public Point getSecondPoint() { return p2; }

public Point getThirdPoint() { return p3; }
```

- In BlueJ create several shapes of different kinds. Call various methods to change the size, color, location or thickness of the shapes. Use the Object Inspector to observe how the method calls result in a change of state as the internal field values change.

Classes responsible for themselves

A well-designed class ensures that objects of that class are always in a consistent, reasonable state. Now that our shape classes have a richer, fuller interface we should turn our attention to considering if that interface is robust. Are there requests that the user could make that would result in an object being put in an invalid state ? Method calls such as draw() or erase() are not dangerous. Perhaps a user may call draw() if the object is already visible, or erase() if the object has already been erased, but these actions would not result in an invalid state. We should consider requests made by the user that would not be able to be carried out. Consider parameters to methods or constructors such as radius size, height and width. These parameter values should be positive. A zero or negative value would not make any sense. Earlier in our lessons we discussed that the Triangle class should ensure that the three points of the triangle are unique. We will implement this check below. Some methods deal with a Point or a Color object. Could perhaps the user supply an invalid Color or Point ? When considering responsibility it is always wise to keep perspective in mind. We are currently considering what the shape classes should do to act responsibly to requests. Just as these shape classes should be responsible, so should the Color and Point classes. The Color and Point classes should be responsible for ensuring that an instance of their class is a valid object. In other words, it should not be the responsibility of a shape class to ensure the validity of a color or a location. (Note: the Color class is provided as part of the Java API, so we will not look at its internal coding. The Point class is our creation and we

will revisit the internal robustness coding of Point in more detail in a future lesson). An integer, on the other hand, may be a valid integer (such as -300) but the value may not make sense in a particular context (such as the height of a rectangle).

We also must consider and decide on the approach to take to handle invalid requests. The nature of constructors poses a problem. Constructors cannot have any return values specified, so our options of communicating failure back to the object creator are limited. We have essentially two choices. We can either consistently implement two-step initialization or we can rely on exception handling.

Two-step initialization means that the body of the constructor is essentially empty. The user is expected to call another method, often conventionally named **initialize()** in order to complete the initialization of the object. Since initialize is a normal, non-constructor method it can have a return value which we can use to communicate failure or success. Two-step initialization requires strong discipline because the calling of the initialize method is not enforced by the compiler – it must be done faithfully by each developer.

Exception handling is another approach to handing invalid requests. If our object receives an invalid request, either in a constructor or a normal method then we can create and throw an exception object. The responsibility then is shifted back to the caller to catch and handle the exception. We will use the approach of exception handling in our examples.

The modified setHeight() method of Rectangle is shown below.

We have added a **throws** clause to the method header to communicate to the outside world that this method may potentially throw an Exception. As a first step in the method we check to ensure that the requested height is a positive value. If it is not then we throw an exception. (Note: The type of exception that we use here is a simple, “anonymous” exception. Other techniques are possible, such as using reusable, named exceptions).

```
  public void setHeight(int height) throws Exception
  {
    if ( height < 1 )
        throw new Exception("Height must be positive");
    erase();
    this.height = height;
    draw();
  }
```

- Create a Rectangle object. Call its setHeight() method and put in an illegal value for the parameter. What happens ? BlueJ should take you to the offending line of code. (Note that this behavior is BlueJ-specific. Other Java runtime environments will simply crash the program).

As mentioned earlier, the Triangle class needs to carefully examine the three points to ensure that the three points in fact do define a triangle. Not only must the three points be unique but they must also not all fall on the same line. In a valid triangle, there are three line segments formed – between the first and second points, the first and third points and the second and third points. If we calculate the slope of these three line segments and ensure that the slope is not the same for all three then we will have verified that the three points are not all in the same line.

The code added to the Triangle constructor is shown below.

```
    // if we find two points that are identical then throw an exception
    if ( p1.equals(p2) || p1.equals(p3) || p3.equals(p2) )
        throw new Exception("Triangle does not have three unique points");

    // ensure that the three points are not all on the same line
    if ( p1.slope(p2) == p2.slope(p3) &&
         p2.slope(p3) == p3.slope(p1))
          throw new Exception("Triangle points cannot all be on same line");
```

The code above has calls to a slope() method of the Point class. There are times when we have to add functionality to the system and we realize that we need to add a method to a class in order to put the new functionality in the class that is the most logical choice. In the Triangle class we needed to calculate the slope of three line segments. Rather than copy and paste the formula three times a choice was made to put a new method into Point which calculates the slope of a line formed between two Point objects. Once this behavior was added to Point, the Triangle class can use simply call the method to access the new behavior. The code for the slope method is below:

```
// In Point:
public double slope(Point p1, Point p2)
{
    double deltaX = this.x() - p2.x();
    double deltaY = this.y() - p2.y();
    return deltaY / deltaX;
}
```

- Try to create a Triangle object where the three points are all on the same line. You can use the points (0,0), (50,50) and (100,100). An exception should be thrown which prevents the creation of the invalid object.

- Open up the Rectangle, Circle and Triangle classes and scan down through them to see what other methods could potentially fail. Note that the exception-handling approach for constructors is essentially the same as for other methods.

Assignment

Add your latest version of the LineSegment class to the v4_secondShapeClasses project.

Review the additions and changes we made to Circle, Triangle and Rectangle and add any additions or changes to LineSegment that would be appropriate.

Be sure to test your changes.

7.6. Lesson Document: Lesson 5 Further Polish

Files Needed

v5_furtherPolish project files.

Deliverables

Assignment hand-in. See assignment section at the end of the lesson.

Objectives

By the end of this lesson you will be able to:

- Appreciate intuitive class interfaces
- Hide implementations from the user of a class
- Explain the Singleton design pattern
- Explain the Façade design pattern
- Decide which class in a system should contain a given feature

Introduction

In this lesson we continue to examine class responsibility and class behavior. Our system has a few annoyances and nuances that should be improved. We shall discuss and provide solutions for these issues.

Client Awareness of TurtlePen

We started out our series of lessons discussing the TurtlePen class, and although the TurtlePen is a critical class in our *implementation* (it is our means of drawing) we should consider whether or not it is a critical element in our *interface*. In other words, do our clients need to be aware that we use a TurtlePen object for drawing ? Currently, when a circle, triangle, rectangle or line segment shape is made the client has to supply a pen object. Should the client need to do this ? We should not require tasks of our clients that aren't strictly necessary. If we are able to do work for our clients then we should strive to put that responsibility in our class rather than require it of every client that wishes to use our services. The general set of services that our system is offering is that of drawing shapes. We can presume that our client is interested in basic information about shapes and not necessarily the exact mechanism that causes the shape to be drawn on the screen. In other words, there is no reason for clients to know about the TurtlePen.

The question really is "who needs to be aware of the pen" ? Currently, our drawIn() methods use the pen object to draw lines appropriate for a given shape type. Certainly the shape classes themselves need to be aware of the pen. But our user does not need to be aware of the pen. We should adjust our interface so that a pen is retrieved when needed without the user having the responsibility to create and pass a pen object.

Taking TurtlePen out of the interface is fairly simple. The TurtlePen field is removed and the pen parameter is taken out of the constructor. The drawIn() method is the only place where a pen object is needed so we create the pen inside of the method. The changes needed to Circle are shown below. The changes to the other shape classes are essentially the same.

```
public class Circle
{

  private TurtlePen pen;
  private Color color;
  private int strokeWidth;
  private int radius;
  private Point center;

  Circle(TurtlePen pen,
         Point center,
         int radius,
         Color color,
         int strokeWidth) throws Exception
  {

    if ( strokeWidth < 1)
        throw new Exception("Stroke width must be positive");

    if ( radius < 1 )
        throw new Exception("Radius length must be positive");

    this.radius = radius;
    this.center = center;
    this.pen = pen;
    this.color = color;
    this.strokeWidth = strokeWidth;
  }
```

```
public class Circle
{

  private Color color;
  private int strokeWidth;
  private int radius;
  private Point center;

  Circle(Point center,

         int radius,
         Color color,
         int strokeWidth) throws Exception
  {

    if ( strokeWidth < 1)
        throw new Exception("Stroke width must be positive");

    if ( radius < 1 )
        throw new Exception("Radius length must be positive");

    this.radius = radius;
    this.center = center;

    this.color = color;
    this.strokeWidth = strokeWidth;
  }
```

```
  private void drawIn(Color color)
  {
    // Lift the pen, move it to the top of the circle,
    // and lower it again

    pen.setColor(color);
    pen.setWidth(strokeWidth);
    pen.up();
    pen.move(this.center.x(), this.center.y() );
    pen.setDirection(90);
    pen.move(radius);
    pen.turn(90); pen.move(0.8);
    pen.down();

    // Draw the circle
    // The circle is drawn by creating
    // 120 short line segments and turning the
    // pen slightly ( 3 degrees ) after
    // each segment is drawn
    for (int i = 1; i <= 120; i++)
    {
       pen.move(2.0 * Math.PI * radius / 120.0 );
       pen.turn(3);
    }
  }
```

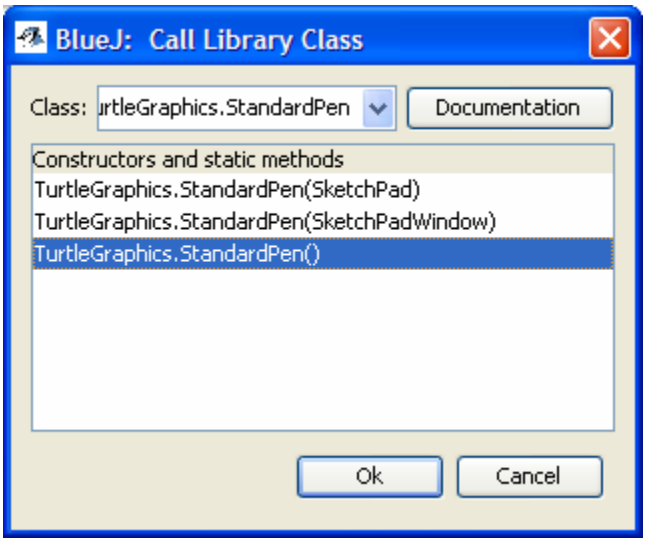
```
  private void drawIn(Color color)
  {
    // Lift the pen, move it to the top of the circle,
    // and lower it again
    TurtlePen pen = new TurtlePen();
    pen.setColor(color);
    pen.setWidth(strokeWidth);
    pen.up();
    pen.move(this.center.x(), this.center.y() );
    pen.setDirection(90);
    pen.move(radius);
    pen.turn(90); pen.move(0.8);
    pen.down();

    // Draw the circle
    // The circle is drawn by creating
    // 120 short line segments and turning the
    // pen slightly ( 3 degrees ) after
    // each segment is drawn
    for (int i = 1; i <= 120; i++)
    {
       pen.move(2.0 * Math.PI * radius / 120.0 );
       pen.turn(3);
    }
  }
```

- Open the v5_furtherPolish project in BlueJ. Create a Rectangle object. Notice that you now don't have to create or specify a TurtlePen object. This is an improvement because now all that you specify is pertinent to the task that you are performing. The information you now provide is necessary for describing the type of Rectangle that you want to create. The TurtlePen was essentially an **implementation detail** that the client really doesn't need to know.

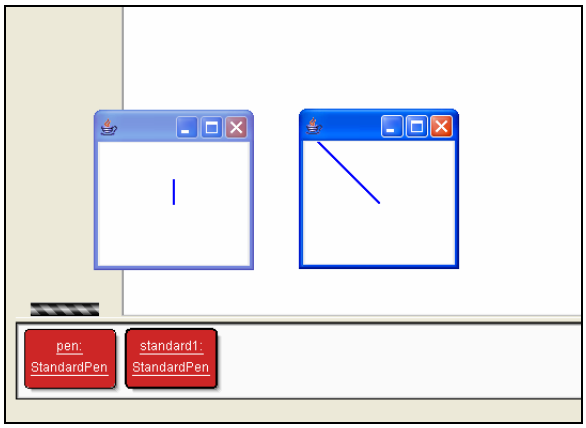
TurtlePen: Façade and Singleton patterns

We will now take a closer look at the implementation of the TurtlePen class. The TurtlePen class uses another class called StandardPen which can be found in Lambert and Osborne's TurtleGraphics library from *Beginning Java* (Addison Wesley 2002). TurtlePen was created in order to use the StandardPen class in a simplified way for use in these lessons. Perform the steps below to see a nuance of the StandardPen class.

- In BlueJ click on the Tools, Use Library Class menu item.
- When the Call Library Class dialog appears, type in TurtleGraphics**.**StandardPen and then hit the tab key. Select the line that says TurtleGraphics.StandardPen() and then click Ok.



- You now have a pen object on the object bench. Right click on the object instance and then click on *Inherited from AbstractPen*. The list of methods that appears should look familiar to those in TurtlePen for the most part. Call a few methods and then observe the results in the drawing window.
- Click on Tools, Use Library Class again and create another pen object in the same manner as above. Perform some pen movements on the second pen. Do you see the lines you drew from the first pen ? What is happening ?



What has happened is that the creation of the second pen results in the creation of a second, unrelated drawing window. Certainly there are times when we would want to create a second drawing window but likely more often we would want to draw multiple items in the *same* window. Why this is an issue for us is that there is a potential in our system for more than one pen object to be created, since we now have pen objects being created as needed by the shape classes. In prior lessons whether or not there were multiple pen objects created depended on the user. In any case, there is potential confusion and potential for undesired results due to the fact that making multiple pen objects results in multiple drawing windows. "Who creates the pen, and when, and how often ?" become potentially confusing questions.

A way to solve this problem is to architect a solution so that it can be ensured that only one pen object is ever created, and to have the pen created implicitly.

The logic of TurtlePen does just that. Part of TurtlePen's implementation is that of a **Singleton Pattern**. A Singleton Pattern is a coding technique used to ensure that only one instance of a particular class gets created. The logic of TurtlePen ensures that only one underlying StandardPen is ever created, thus ensuring that all drawing occurs in the same window.

- Confirm the above statement. Create two TurtlePen objects on the object bench. Perform drawing operations on each. All of the drawing should appear in the same drawing window.

- Open the code for TurtlePen and study its implementation.

The Singleton pattern often uses a static field, which is a field that is stored only once per class, rather than once per instance. The TurtlePen code ensures that the underlying StandardPen object is created once and only once.

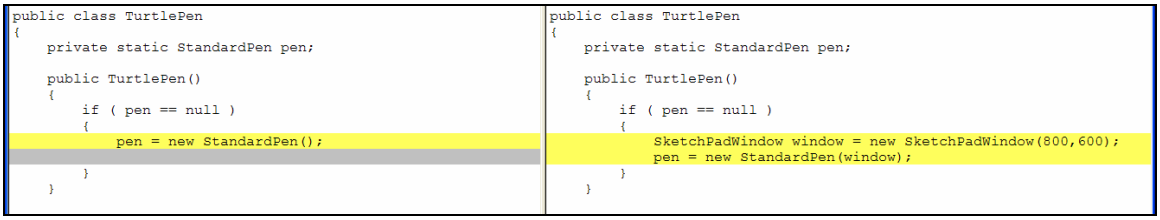
Making a larger drawing area

An annoyance that we have endured so far is the small initial size of the drawing window. We shall remedy that now.

If a StandardPen object is created with no parameters then a small drawing window appears to display the pen movements.

A StandardPen object can be created and given a SketchPadWindow object to draw in. SketchPadWindow is another class found in the TurtleGraphics library. It can be constructed with a specific size.

- View the file compare below and make the change to TurtlePen so that your version of the code appears as on the right (the code should currently appear as on the left). Be sure to compile the project.



```
public class TurtlePen
{
    private static StandardPen pen;

    public TurtlePen()
    {
        if ( pen == null )
        {
            pen = new StandardPen();

        }
    }
```

```
public class TurtlePen
{
    private static StandardPen pen;

    public TurtlePen()
    {
        if ( pen == null )
        {
            SketchPadWindow window = new SketchPadWindow(800,600);
            pen = new StandardPen(window);
        }
    }
```

The code to add is shown here in a larger font:

```
SketchPadWindow window = new SketchPadWindow(800,600);
pen = new StandardPen(window);
```

- Create a shape object and draw it. Notice that the drawing window comes up larger without needing manual resizing. With the above change drawing windows will from now on come up with a larger initial size.

Design Patterns

The Singleton Pattern discussed above is a specific example of a **design pattern**. Design patterns are proven solutions to design problems that tend to arise fairly frequently in development of systems.

Another design pattern that is present in TurtlePen is that of a **façade**. A façade pattern is used when we want to put a front-end to another class or system and use a subset of its functionality. TurtlePen offers a subset of the interface of StandardPen. TurtlePen allows us to use a StandardPen object in a specific

way. TurtlePen insulates clients from nuances of the StandardPen. Clients of TurtlePen do not need to be aware of the multiple-window nuance, or the small-window nuance or SketchPadWindow objects. We will soon hide another nuance of the StandardPen – we will provide an alternative to having the origin existing in the center of the window. Notice that we are introducing many layers in our system. TurtlePen exists to hide some complexities of StandardPen. Our first consideration of this lesson was to hide the existence of TurtlePen from users of our system (those clients only interested in drawing shapes). TurtlePen is still needed by our shape classes such as Circle. The fewer dependencies that we have in our system, and the more generic the interfaces between classes, the easier it is to make changes and insert different implementations.

Additional Responsibilities of Point and TurtlePen

We have examined in detail the responsibilities and behaviors of the shape classes and also those of the Point class.

Have we identified all areas of responsibility for Point ? Are there any potential problems that could arise or do we have any behavior in the system that is undesirable related to the location of shapes ?

We analyzed the shape classes for possibilities of invalid requests by the user. Is there a similar possibility that a Point object could be put into an invalid state ? Are we preventing unreasonable values for the x and y location values ? Do we allow unreasonably large numbers or negative numbers ? The Point class should have the intelligence to disallow unreasonable requests.

We must first analyze, consider and decide what constitutes valid locations in the drawing window.

Recall that for TurtlePen, locations are specified with the understanding that (0,0) is the center of the drawing window. An argument could be made that (0,0) as the center is not a very intuitive choice. Computer graphics applications often assume that (0,0) is the upper left corner and that horizontal values increase to the right and vertical values increase as you go down. In mathematics, the origin is the lower left, unless there is a need to graph negative values. A problem with the origin as center approach is that 75% of the screen will involve locations with at least one dimension expressed as a negative number (see the figure below). Negative numbers are not intuitive to most people, and it would be nice if our user did not need to think in terms of negative numbers when determining the locations of items on a drawing.

The drawing window

negative x, positive y values	positive x, positive y values **(0,0)**
negative x, negative y values	positive x, negative y values

The notion of location exists in the following places in our system:

- The Point class is used by users of the system to define the locations of the shapes that they want to draw.
- The shape classes instruct a TurtlePen object to move to the point that represents the location of the shape. The pen movements that follow are all relative movements from this shape location point.
- The TurtlePen object forwards pen movement requests on to its contained StandardPen object.
- The StandardPen class interprets its movements according to (0,0) as the center. StandardPen is not owned by us, so we do not have the liberty to change the code. Ultimately we have to call StandardPen methods that interpret locations with the origin as center.

What would be desirable is if we can maintain two coordinate systems:

- a logical, intuitive system where (0,0) is the lower left corner and all coordinate values are non-negative. This would be the view used by users of our system.
- a physical view consistent with StandardPen's interpretations of location.

What we must consider is what knowledge each class should have concerning location.

Our users deal with the Point and shape classes, so it would make sense for these classes to interpret location in terms of the logical view of the origin as the lower left corner.

TurtlePen is the class that interfaces with StandardPen. TurtlePen exists to hide nuances of StandardPen from other classes. TurtlePen could be responsible for mapping logical coordinate values to physical coordinate values before handing off requests to StandardPen.

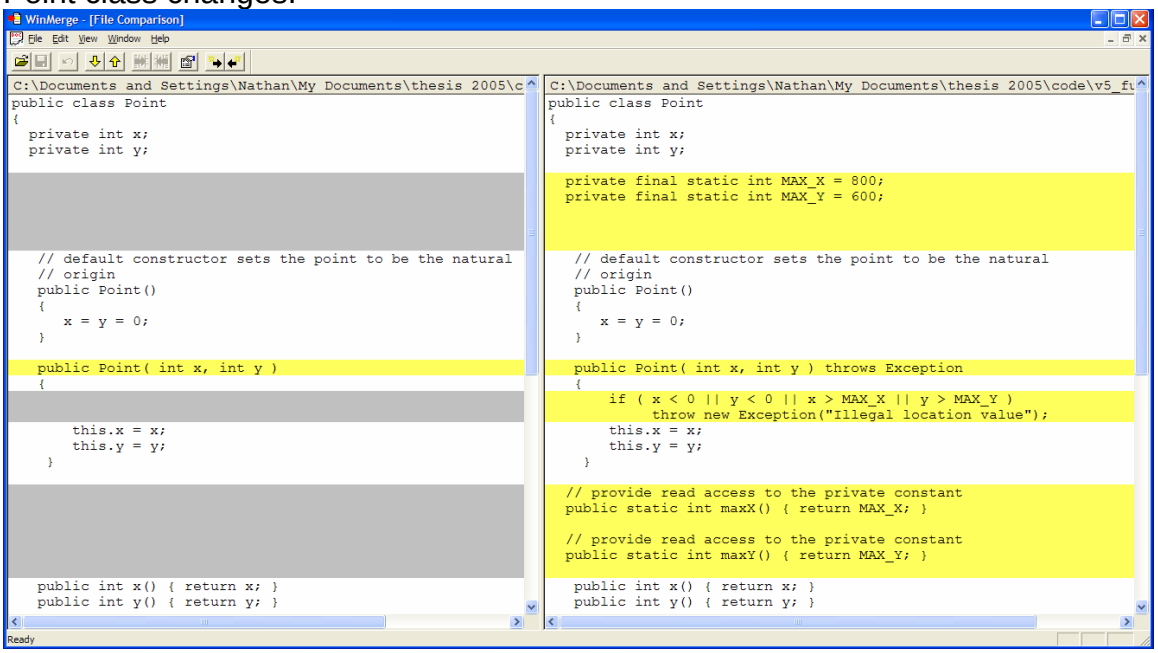
In order to map values from the lower left corner to the center we need to know the overall size of the drawing window. Earlier in the lesson we used hard-coded values of 800 x 600 but it would be a better design to use symbolic constants of some sort. Since the Point class represents the notion of location, it would seem logical for the Point class to "know" what the maximum x and y values are that a user could specify.

The version of Point in the v5_furtherPolish project has already been modified with the changes shown below. We have added two constants to the Point class, a MAX_X and a MAX_Y . We have added logic to ensure that user-requested values are not negative and not greater than these maximum values. The static methods maxX() and maxY() can be used to query the maximum values. A static method is one that can be called without creating an instance of a class. To call these static methods of Point use

```
Point.maxX()
   or
Point.maxY()
```

- Open up TurtlePen and change the literal values of 800 and 600 to be calls to the maxX() and maxY() methods of Point. Compile the project and then create and draw a shape. Be sure that the screen still comes up as 800 x 600.

Point class changes.

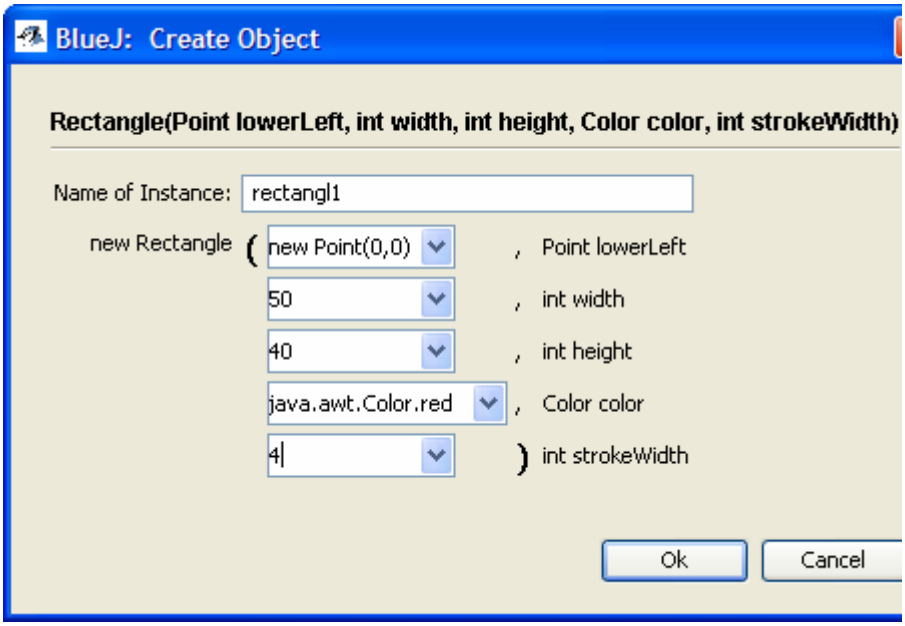


Now that we have a way to know the width and height of the drawing area we can make modifications to TurtlePen to map location values from the origin-as-left-corner perspective to the origin-as-center perspective.

- Open up TurtlePen and change the move(x,y) method to look as below:

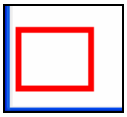
```
public void move(int x, int y)
{
    pen.move(x - Point.maxX() / 2 ,
             y - Point.maxY() / 2);
}
```

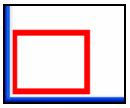
- To test this code, create a Rectangle with the characteristics shown below and then draw the rectangle.



Did the rectangle appear neatly in the lower left corner ? No, unfortunately it did not. Software is never simple, even in a seemingly simple set of lessons about drawing shapes.

The rectangle is clipped. Apparently if we give a window size of 800 x 600 window borders and title bars are not taken into account and we have less than 800 x 600 left to draw in. We must accept for the time being that our system may not be exactly perfect. Through experimentation we can come up with a SketchPadWindow size that will be large enough to include window borders, title bars and an 800 x 600 drawing area. This experimentation is easily done in BlueJ. We can use the moveTo method to nudge the rectangle along until we get it tucked just inside the corner of the window border.

 Moving to a point of (10,30) gets us closer.

 Nudging a little further to (7, 21) gets us close to the corner while still being able to make out the left and bottom edges of the rectangle.

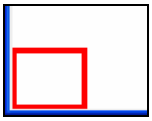
Through further experimentation and empirically looking at the results it appears that adding 14 pixels horizontally and 38 pixels vertically leaves us enough space for all four borders and the title bar. What we can test next is to see what happens if we create a drawing window that is 14 pixels wider and 38 pixels taller. But where do we make such an adjustment ? The Point class is concerned with the logical, intuitive, conceptually pure notion of location. It would be desirable to not pollute such a class with such an informal fudge factor. TurtlePen is the façade in front of the StandardPen and the SketchPadWindow classes. TurtlePen can be the class that for now knows about the adjustment needed to get the correct window size.

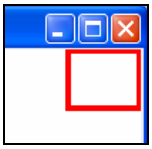
- Open up TurtlePen and find the line of code that creates the SketchPadWindow object. Adjust it so that it appears as below:

```
SketchPadWindow window = new SketchPadWindow(Point.maxX() + 14,
                                             Point.maxY() + 38);
```

Our rectangle is 50 x 40 so if we were to move it to a point just inside the upper right corner of the drawing area we would move it to the point (750, 560).

Testing a 50x40 rectangle at locations (0,0) and (750,560) gives us these results:

 at Point (0,0)

 at Point (750,560)

Although certainly not ideal or perfect, we appear to have a solution which gives us close to what we want.

You may be wondering why the StandardPen was used at all for these lessons. Now would be a good time to explain. StandardPen (and TurtlePen) were used because:

- A turtle-graphics type object is a very simple, well-defined object with observable behavior that is a good choice for being a first object to *use*.

- The pen object does give us a means of drawing. Java 2D classes are more complicated and have interfaces that would be distracting.

- Using the pen gives us an opportunity of defining one class (such as Circle) in terms of another class (TurtlePen). The concept of classes using other classes is so prevalent in object technology that it was advantageous to show this approach early.

- Finally, the author is not yet an expert on Java 2D classes and using the pen object was a way to reuse working behavior rather than learn a complex new library of classes. Leveraging existing classes is a valuable skill.

Assignment

Part One

Adjust LineSegment so that the user does not need to create a TurtlePen object when creating a line segment.

Part Two

Insert a copy of your Cat picture (from a previous lesson) into the v5_furtherPolish project. Adjust the object creation statements so that you are no longer passing a TurtlePen object. Also, adjust coordinate values if needed to account for the origin now being in the left corner. It is ok if your picture is now in a different spot in the window but make sure that the picture fits within the window. Make sure your program compiles and runs.

7.7. Lesson Document: Lesson 6 Inheritance

Files Needed

v6_inheritance project files.
v6_inheritance2 project files.
V6_inheritance3 project files.

Deliverables

Assignment hand-in. See assignment section at the end of the lesson.

Objectives

By the end of this lesson you will be able to:

- Place behavior common to a group of classes into a new base class
- Inherit methods and fields from a base class into a derived class
- Make a specialized version of a class
- Describe the three basic class relationships
- Distinguish between abstract and concrete methods

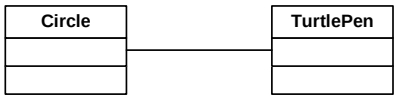
Introduction

In this lesson we will introduce some new classes and introduce some new coding techniques. We will discuss how to use inheritance to create categories of related classes and avoid code redundancy.

Class Relationships

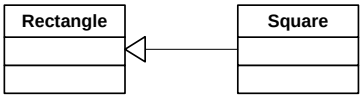
Object technology relies on fundamental relationships between classes. There are three fundamental relationships.

- One class may *use* another class. For example the Circle class uses the TurtlePen class to perform the drawing of the circle. This type of relationship is called **association.** In UML class diagrams this relationship is shown with a line between the two classes.



- Conceptually, one class may *contain* another class. We may think of an object of one class as being *made up of* objects of another class. For example, our House picture from an earlier lesson was made up of a triangle, a circle and rectangles. Having, or containing relationships (sometimes called *Has-a)* are more formally called either **composition** or **aggregation**. Composition relationships are those where the contained objects are part of the containing object, such as a handle being part of a broom, or the roof on a house. Aggregation relationships are where the contained objects are not a part of the containing object, but are more of a collection, such as buses in a bus garage. A diamond is used on the UML class diagram to denote a having relationship. The diamond is filled in for a composition relationship but not for an aggregation. Multiplicity symbols can be added to document how many of the contained item exist for each containing item.

- The third type of relationship is **inheritance**. Inheritance is used to express a situation where one class is a special case of another class. For example, a Square is a special kind of Rectangle. We can state that a Square *is a* Rectangle. Everything that a Rectangle is, a Square is also. A Rectangle has a location, a color, a height, a width and a thickness. So does a Square. A Square has an additional characteristic that the height and the width must be the same. A Square can be thought of as a *specialized* Rectangle. Everything that we can do to a Rectangle, such as drawing, erasing, moving, .etc can be done to a Square as well. A Square *inherits* all characteristics and behaviors from Rectangle. We add specializing characteristics or behavior to the Square class (such as ensuring the height and width are the same). A class such as Rectangle is called the **parent**, **base**, or **super** class (all synonyms). A class such as Square is called the **child**, **derived**, or **sub** class. Inheritance is denoted on UML diagrams by an open arrow pointing from child to parent.



Implementing Inheritance

Inheritance is specified in Java by using the **extends** keyword. A child class can be thought of as an *extension* of a parent class. When coding the class definition for the Square class we would do so as shown below:

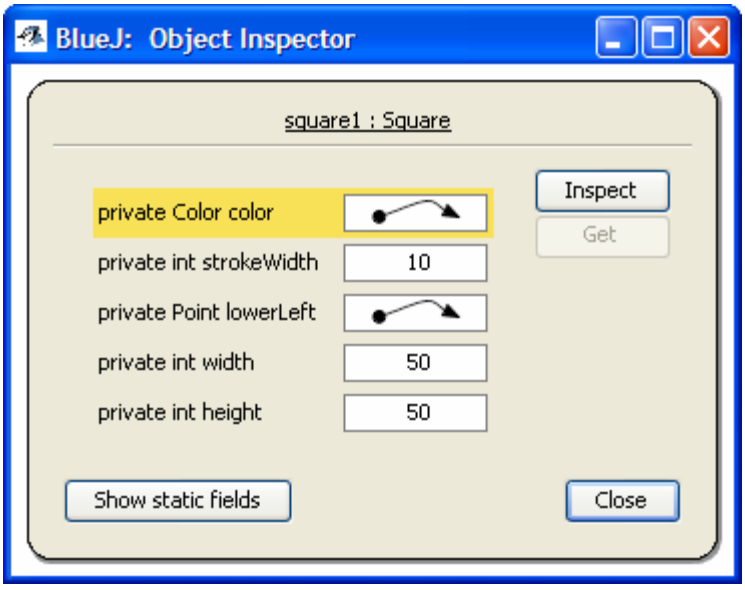
```
public class Square extends Rectangle
```

The extends keyword can be thought of as having the meaning "is a kind of". So when reading the class header line above we should read it as "Square is a kind of Rectangle".

The implementation effect of inheritance is that instances of Square automatically inherit all fields and methods defined in Rectangle. We can use BlueJ's object inspector to confirm this.

- Open the v6_inheritance project in BlueJ. Create a Square object. Draw it. Note that to find the draw() method you will need to click on the *Inherited from Rectangle* menu. All of the methods that are available to Rectangle objects are available to Square objects (after all, a Square *is a* Rectangle). These methods are defined and coded in the Rectangle class, not in the Square class. However, since Square inherits from Rectangle all methods in Rectangle apply to Square instances.

- Right click on the Square object and then click on *Inspect*. Although our client does not need to be aware of how our object is wired internally, as developers and students it is useful on occasion to examine object internals to see how classes are built. Notice that the Square instance has values for all of the fields defined in the Rectangle class. Just as Square instances inherit all methods from their parent, all fields are inherited as well. A height and a width are stored, even though when we created the Square we only specified a side length. (All rectangles have a height and a width. For Squares, the height and the width happen to be the same).



Initialization Under Inheritance

We will now look more closely at the coding mechanisms and runtime behavior that allow a Square object to act like a Rectangle. The code for the Square class is shown below.

```
public class Square extends Rectangle
{
  Square(Point lowerLeft,
         int sideLength,
         Color color,
         int strokeWidth) throws Exception
  {
      super(lowerLeft,sideLength,sideLength,color,strokeWidth);
  }

}
```

The Square class provides a constructor for clients to call. Clients must supply a Point, a side length, a color and a line thickness. By virtue of inheritance, Square inherits all fields from its parent Rectangle. We need a mechanism to initialize the fields received from Rectangle. The Rectangle constructor is a method whose purpose is to initialize these fields. As a child class we can (and must) call our parent's constructor to initialize the fields received from our parent. This is done by using the **super** keyword. Coding a super method call will result in the calling of the parent constructor (Rectangle in this case). For reference, Rectangle's constructor signature is shown below, along with Square's call to super(). Note how the values on the super call correspond to the parameters expected by Rectangle's constructor.

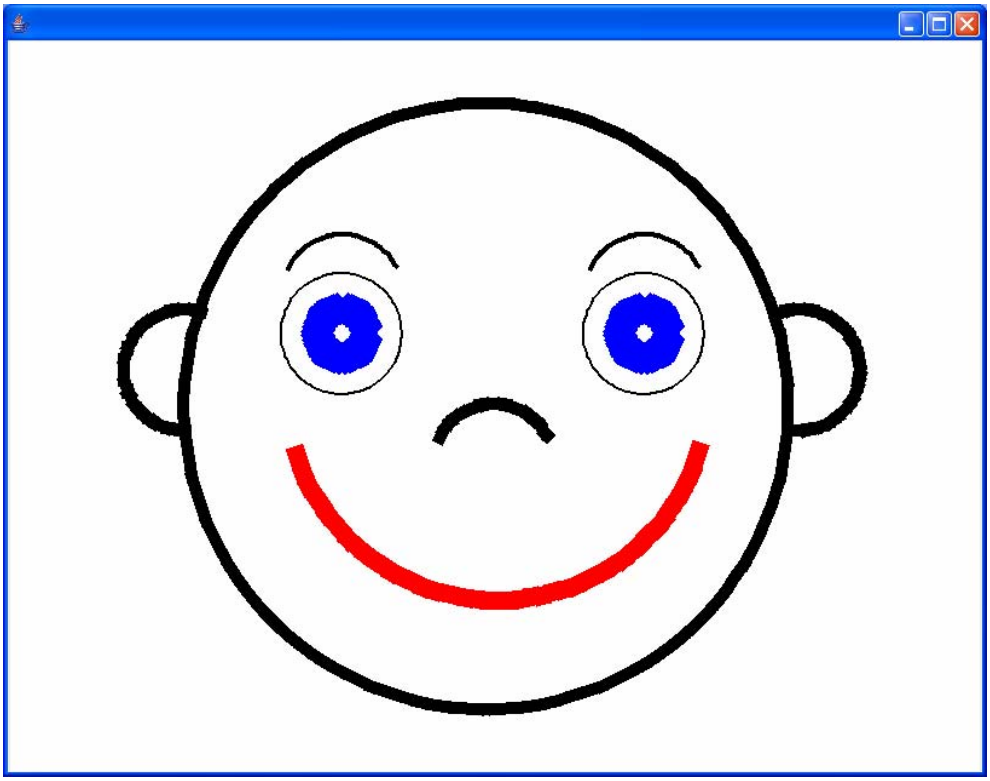
```
    // In Square
    super( lowerLeft,   sideLength,  sideLength,  color,  strokeWidth);

Rectangle(Point lowerLeft, int width, int height, Color color, int strokeWidth)
```

When using super() we must pass along parameters as defined by the parent constructor. Rectangle's constructor expects a Point, a height, a width, a color and a line thickness. Since a square's height and width are the same we pass along the side length as both the height and the width. The call to the super class constructor must be the first line of code in the child class constructor. Child classes can have additional fields defined (there are none for Square in this example). The child field values are initialized in the child constructor underneath the call to the parent constructor. Child classes can also define additional methods in the class body of the child.

Another Example Of Inheritance

We can identify potential parent-child class relationships if we can identify situations where one class is a specific occurrence of something more general. Consider Circle, for example. A Circle is formed by drawing an arc 360 degrees a fixed distance around a center point. What if we wanted an arc that was not a full circle, such as a half circle ? How about other arc lengths ?

The face picture below is made up of several different kinds of arcs, having different lengths and starting at different angles.

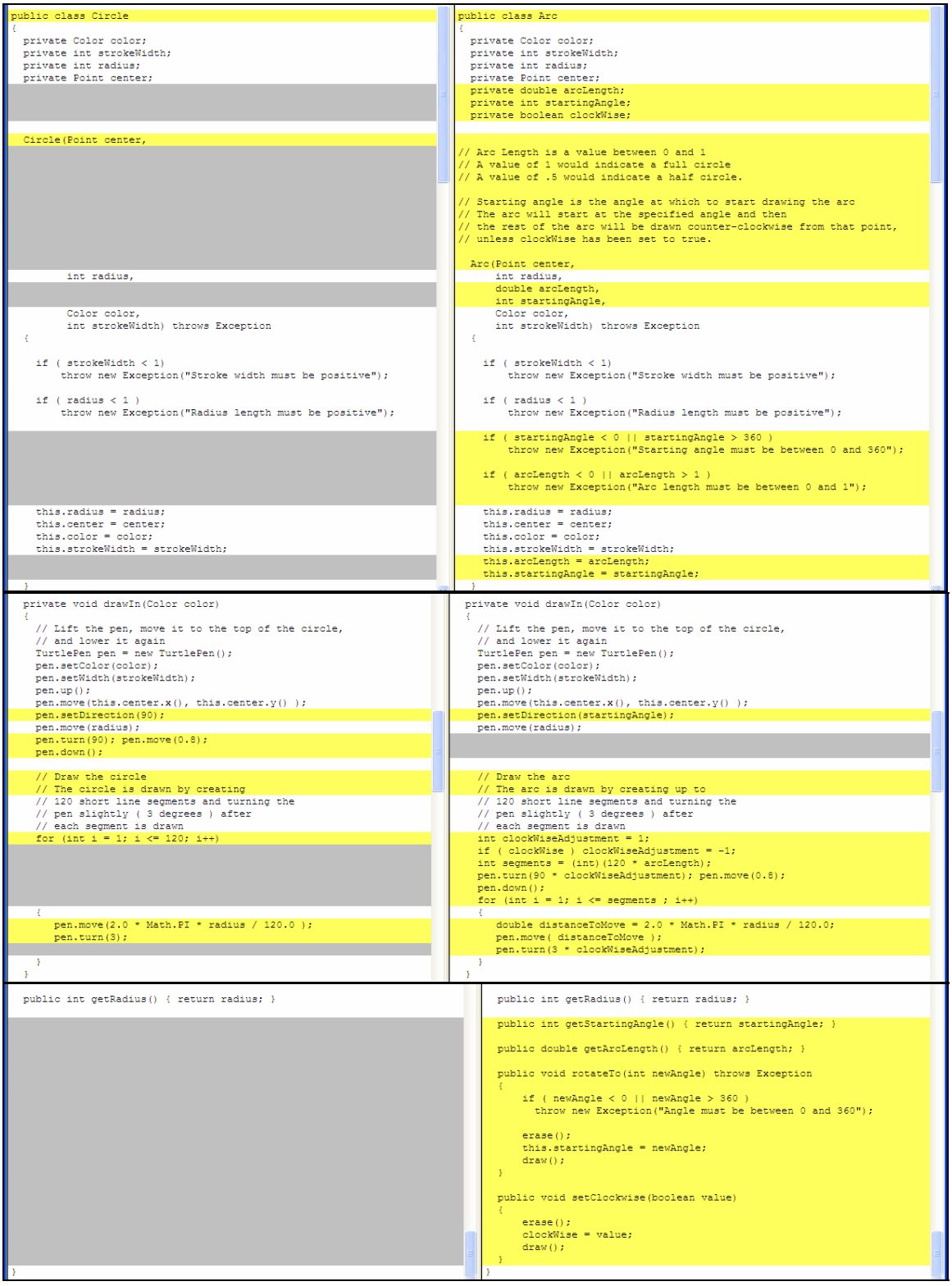


The Arc class in the v6_inheritance project is a modification of our old Circle class in order to make it more general. The Arc class allows for a custom arc length, a starting angle and an ability to draw the arc clockwise rather than counterclockwise (in case this would make it easier for clients to define their arcs).

The code compare below shows the old Circle class on the left and the new Arc class on the right. The differences are due to the arc length, the starting angle and the calculations needed to arc part way around a circle rather than full circle. Only sections of the files with differences are shown.

The arc length setting is a value between 0 and 1. A value of 1 would indicate a full circle and a value of 0.5 would indicate a half circle. The starting angle is the angle at which to start drawing the arc. The arc will start at the specified angle

and then the rest of the arc will be drawn counter-clockwise from that point, unless the clockWise setting has been set to true, in which case the arc will be drawn in a clockwise direction.



```
public class Circle
{
  private Color color;
  private int strokeWidth;
  private int radius;
  private Point center;

  Circle(Point center,

         int radius,

         Color color,
         int strokeWidth) throws Exception
  {

    if ( strokeWidth < 1)
        throw new Exception("Stroke width must be positive");

    if ( radius < 1 )
        throw new Exception("Radius length must be positive");

    this.radius = radius;
    this.center = center;
    this.color = color;
    this.strokeWidth = strokeWidth;

  }
```

```
public class Arc
{
  private Color color;
  private int strokeWidth;
  private int radius;
  private Point center;
  private double arcLength;
  private int startingAngle;
  private boolean clockWise;

// Arc Length is a value between 0 and 1
// A value of 1 would indicate a full circle
// A value of .5 would indicate a half circle.

// Starting angle is the angle at which to start drawing the arc
// The arc will start at the specified angle and then
// the rest of the arc will be drawn counter-clockwise from that point,
// unless clockWise has been set to true.

  Arc(Point center,
      int radius,
      double arcLength,
      int startingAngle,
      Color color,
      int strokeWidth) throws Exception
  {

    if ( strokeWidth < 1)
        throw new Exception("Stroke width must be positive");

    if ( radius < 1 )
        throw new Exception("Radius length must be positive");

    if ( startingAngle < 0 || startingAngle > 360 )
        throw new Exception("Starting angle must be between 0 and 360");

    if ( arcLength < 0 || arcLength > 1 )
        throw new Exception("Arc length must be between 0 and 1");

    this.radius = radius;
    this.center = center;
    this.color = color;
    this.strokeWidth = strokeWidth;
    this.arcLength = arcLength;
    this.startingAngle = startingAngle;
  }
```

```
  private void drawIn(Color color)
  {
    // Lift the pen, move it to the top of the circle,
    // and lower it again
    TurtlePen pen = new TurtlePen();
    pen.setColor(color);
    pen.setWidth(strokeWidth);
    pen.up();
    pen.move(this.center.x(), this.center.y() );
    pen.setDirection(90);
    pen.move(radius);
    pen.turn(90); pen.move(0.8);
    pen.down();

    // Draw the circle
    // The circle is drawn by creating
    // 120 short line segments and turning the
    // pen slightly ( 3 degrees ) after
    // each segment is drawn
    for (int i = 1; i <= 120; i++)

    {
       pen.move(2.0 * Math.PI * radius / 120.0 );
       pen.turn(3);

    }
  }
```

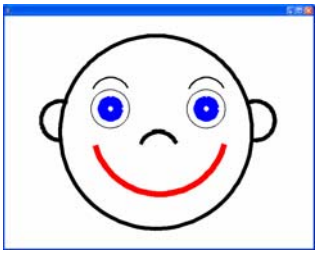
```
  private void drawIn(Color color)
  {
    // Lift the pen, move it to the top of the circle,
    // and lower it again
    TurtlePen pen = new TurtlePen();
    pen.setColor(color);
    pen.setWidth(strokeWidth);
    pen.up();
    pen.move(this.center.x(), this.center.y() );
    pen.setDirection(startingAngle);
    pen.move(radius);

    // Draw the arc
    // The arc is drawn by creating up to
    // 120 short line segments and turning the
    // pen slightly ( 3 degrees ) after
    // each segment is drawn
    int clockWiseAdjustment = 1;
    if ( clockWise ) clockWiseAdjustment = -1;
    int segments = (int)(120 * arcLength);
    pen.turn(90 * clockWiseAdjustment); pen.move(0.8);
    pen.down();
    for (int i = 1; i <= segments ; i++)
    {
       double distanceToMove = 2.0 * Math.PI * radius / 120.0;
       pen.move( distanceToMove );
       pen.turn(3 * clockWiseAdjustment);
    }
  }
```

```
  public int getRadius() { return radius; }

}
```

```
  public int getRadius() { return radius; }

  public int getStartingAngle() { return startingAngle; }

  public double getArcLength() { return arcLength; }

  public void rotateTo(int newAngle) throws Exception
  {
      if ( newAngle < 0 || newAngle > 360 )
        throw new Exception("Angle must be between 0 and 360");

      erase();
      this.startingAngle = newAngle;
      draw();
  }

  public void setClockwise(boolean value)
  {
      erase();
      clockWise = value;
      draw();
  }
}
```

The code for the face picture is shown below. Arcs are specified with their center point, radius length, arc length, starting angle, color and line thickness.

```
Point faceCenter = new Point(400,300);
Circle face = new Circle(faceCenter,250,Color.black,10);
Arc rightEar = new Arc(new Point(650,330),50,.55,270,Color.black,10);
Arc leftEar = new Arc(new Point(145,330),50,.55,270,Color.black,10);
leftEar.setClockwise(true);
Circle leftEye = new Circle(new Point(275,360),50,Color.black,2);
Circle rightEye = new Circle(new Point(525,360),50,Color.black,2);
Circle leftPupil = new Circle(new Point(275,360),20,Color.blue,20);
Circle rightPupil = new Circle(new Point(525,360),20,Color.blue,20);
Arc nose = new Arc(new Point(400,250),50,.35,30,Color.black,10);
Arc leftBrow = new Arc(new Point(275,390),50,.35,30,Color.black,4);
Arc rightBrow = new Arc(new Point(525,390),50,.35,30,Color.black,4);
Arc mouth = new Arc(new Point(400,320),175,.4,200,Color.red,15);

face.draw();
rightEar.draw();
leftEar.draw();
leftEye.draw();
rightEye.draw();
leftPupil.draw();
rightPupil.draw();
nose.draw();
leftBrow.draw();
rightBrow.draw();
mouth.draw();
```



Now that we have an Arc class defined it is easy to redefine Circle in terms of Arc. A Circle can be viewed as an Arc that goes all the way around (full circle). Since the arc is drawn full circle the starting angle and clockwise settings do not have an effect.

The code for the Circle class is shown below:

```
public class Circle extends Arc
{
  Circle(Point center,
         int radius,
         Color color,
         int strokeWidth) throws Exception
  {
    super(center,radius,1,90,color,strokeWidth);
  }
}
```

We have defined Circle as being a kind of Arc. A Circle constructor accepts the same information as before (a center point, radius length, color and line thickness). When a Circle is constructed the circle constructor calls the Arc constructor to initialize the fields contained within Arc. A value of 1 (indicating 100%) is passed as the arc length. A value of 90 is passed as the starting angle. For reference, Arc's constructor signature is shown below, along with the super call in Circle. Note how the parameter values on the super call match up to the parameters defined on the Arc constructor.

```
  // In Circle
  super(center,radius,1,90,color,strokeWidth);

  Arc(Point center,
      int radius,
      double arcLength,
      int startingAngle,
      Color color,
      int strokeWidth)
```

- Create a circle object. You shouldn't notice a difference when creating the object, since the same information is required by the user. Draw the circle. (You will need to click on *Inherited from Arc*). It should draw as before. Right click the object instance again on the object bench and view the method list.

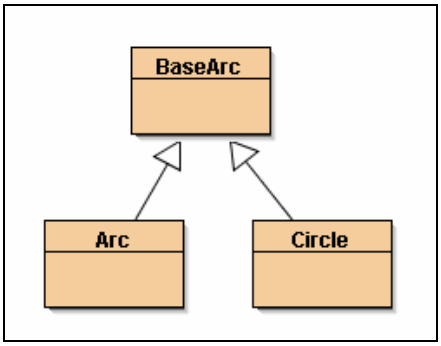
Circle has inherited all of Arc's methods, which is good in many regards, but in certain cases it is a problem. We have inherited methods that will likely be

confusing for our user. Starting angle, arc length and rotation make no sense for a circle.

We have succeeded in reusing code from Arc when implementing Circle but we have failed to maintain an intuitive interface for users of our Circle class.

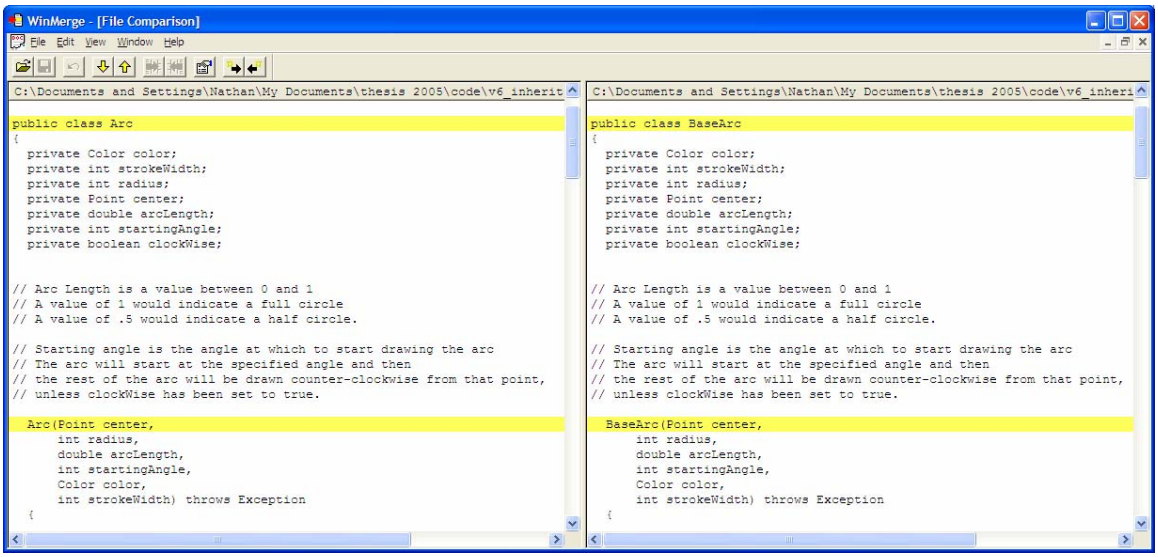
What we can do is to make a base class called BaseArc which will contain all code and methods common to both Circles and to Arcs. We can derive Circle from BaseArc and we can also derive Arc from BaseArc. Arc will have methods that are sensible for arcs but not for circles (like setting the starting angle) .

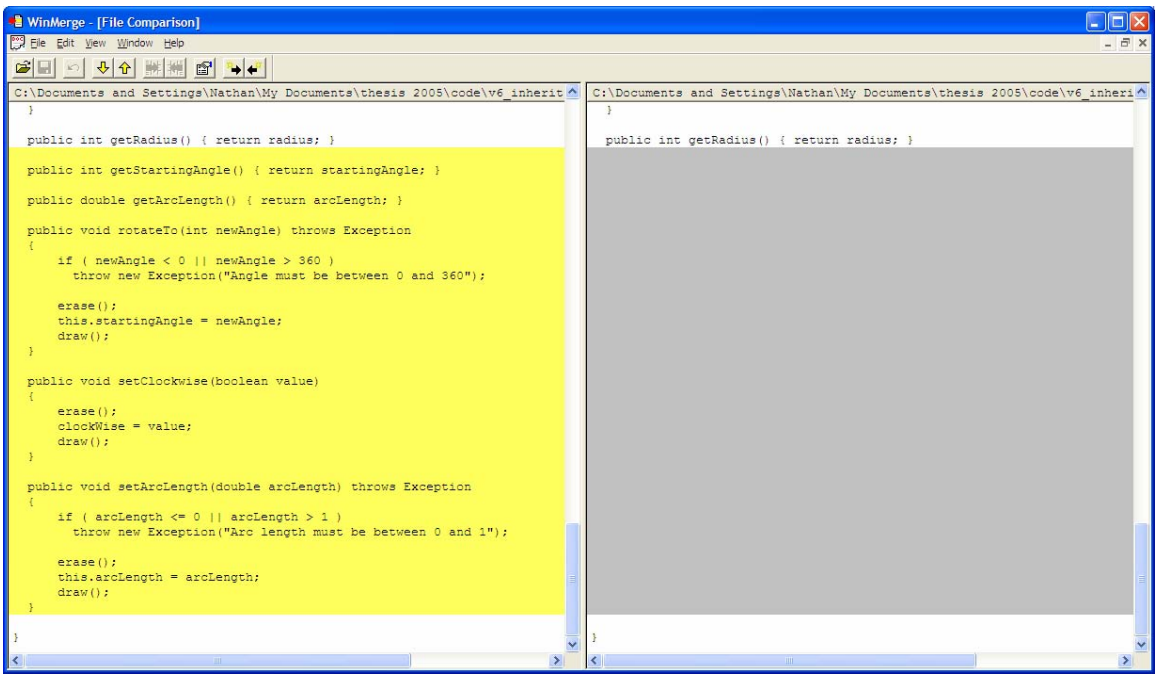
The inheritance diagram would like this:



- Open the v6_inheritance2 project. You should see (among other class symbols) the three class symbols for BaseArc, Arc and Circle as arranged in the diagram above.
- Open the Circle class. The only change to Circle is that it inherits from BaseArc rather than from Arc.
- Create and draw a Circle. It should work as before.

BaseArc is the same as the old Arc class from the v6_inheritance project, except that we have removed methods that would not make sense for full-circle arcs. See the code compare below.





The Arc class is defined to inherit from BaseArc and it contains methods which were removed from BaseArc.

- Open up the Arc class and compile it. What happened ? You should have gotten an error on this line:

```
public int getStartingAngle() { return startingAngle; }
```

The error message is " *startingAngle has private access in BaseArc* ". What is causing this error message ?

Recall that we have generally used a pattern of making fields private in our classes to enforce data-hiding and encapsulation. We don't want our data and internal implementation details to be accessible by the general public. We want to reserve the right to rewire our objects and as long as we continue to provide the same public interface clients should be none the wiser if we do change our private data.

When we have inheritance relationships we have a somewhat different situation. The Arc class inherits all fields from BaseArc, but just because Arc is a child of BaseArc doesn't mean that code in Arc can access private data inherited from BaseArc. In fact, private data is inaccessible to any outside class, even children. There are situations where it would be convenient to allow children to access field data of their parents, without opening the floodgate by allowing general public access. Java provides a level of access that sits between public and private. This level of access is called **protected**. Protected means that a field is private to the outside world but public to derived classes. If we make the startingAngle, clockWise and arcLength variables protected in BaseArc then these fields will be accessible in Arc but not accessible by other classes not inheriting from BaseArc.

- Open up the BaseArc class and change the access of the startingAngle, clockWise and arcLength variables from private to protected. Compile the project.

- Test the new Circle and Arc classes (and thereby implicitly testing BaseArc) by calling the draw() method of the Face class. The code for Face creates several Arc and Circle objects.

Some developers don't like to use protected access because it has a possibility of creating many dependencies between parent and child classes. However, if the classes are all under the control of one developer or department, there are situations where using protected data is a reasonable tradeoff. An alternative to using protected access is to keep the data private in the parent class and have children use public accessor and mutator methods to modify fields just like any other class would need to.

We still have one minor problem to address regarding BaseArc.

- Create an Arc object on the object bench with an arc length of less than one. Draw it. Now rotate it to a different starting angle. Change the arc length to a different percentage.

- Next create a BaseArc object on the object bench also with an arc length less than one. Draw it. Notice that there is no rotateTo() or setArcLength() methods. These methods are only defined in Arc.

Consider these questions:

- Would we create a BaseArc object if we intend to rotate the arc or change the arc length ?
- Would it ever make sense to create a BaseArc rather than an Arc object ?
- Is it a good interface for our clients if they have to wonder whether to create a BaseArc or an Arc object ?

The answer to all three of the above questions is *no*.

The problem we have here is that it never really ever made sense to create a BaseArc object. An Arc object would always be preferable because it offers a full interface of operations. BaseArc exists to factor out commonality between Arc and Circle. It was never intended that instances of BaseArc be created. It would be nice if we could communicate to our users that BaseArc is not a class to create instances from. Fortunately, Java has an **abstract** keyword that can be used to state that instances cannot be created from a class. We will see and discuss other abstract classes later in this lesson.

- Open up BaseArc and adjust the class header line so that it includes the abstract keyword (as shown below):

```
public abstract class BaseArc
```

Notice that BlueJ adds an <<abstract>> modifier to the BaseArc class symbol. (In UML, abstract class names are shown in *italics* on a class diagram).

- Try to create a BaseArc object. BlueJ does not allow you to. The Java compiler would issue an error if it encountered a **new** statement which requested the creation of an abstract class.

The Shape Class

We have, in fact encountered another abstract class many times already in these lessons. We have encountered the class not by looking at its code but by encountering the class *conversationally*. We have naturally used the word **shape** many times in our discussion of the classes Circle, Triangle, Rectangle and Line Segment (and most recently Arc) as a way to refer to them all

collectively, as a category of things. Just as it is natural to use an English word to specify a general word for similar things, we can use a Java abstract base class to specify a class that contains behavior and characteristics common to similar classes.

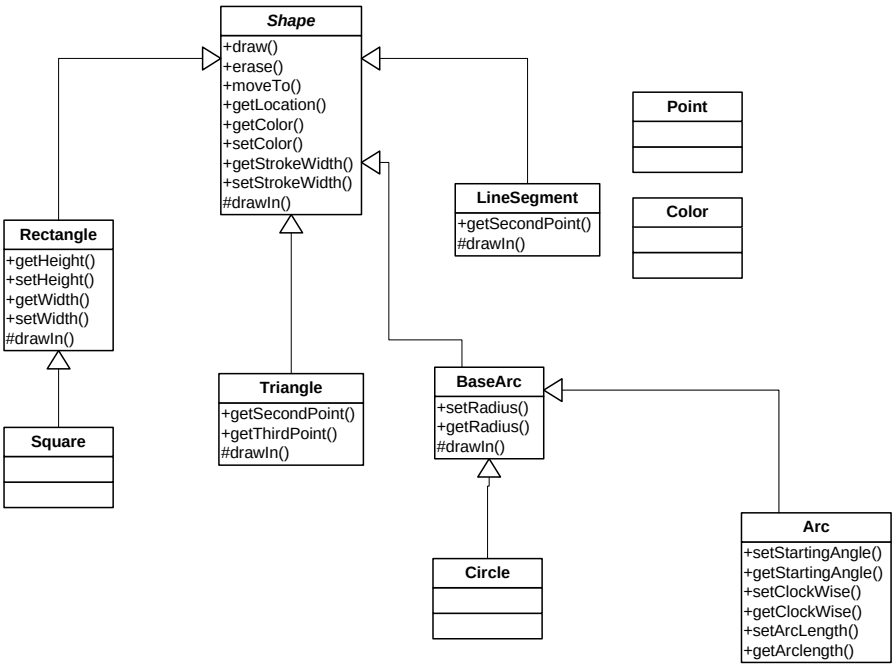
What can we do to all shapes ? We can:

- erase them
- draw them
- change their color
- move them
- change their line thickness

We can move these methods into a class called Shape and we can then have each specific shape class (such as Circle) derive from Shape. The Shape class will also have all fields that are common to every shape class. The derived classed will contain behavior and characteristics that are specific or unique to that shape type. Each shape subclass will know how to draw a shape of its own kind.

Shape Class Interface

Our class interfaces and relationships are shown in the diagram below.

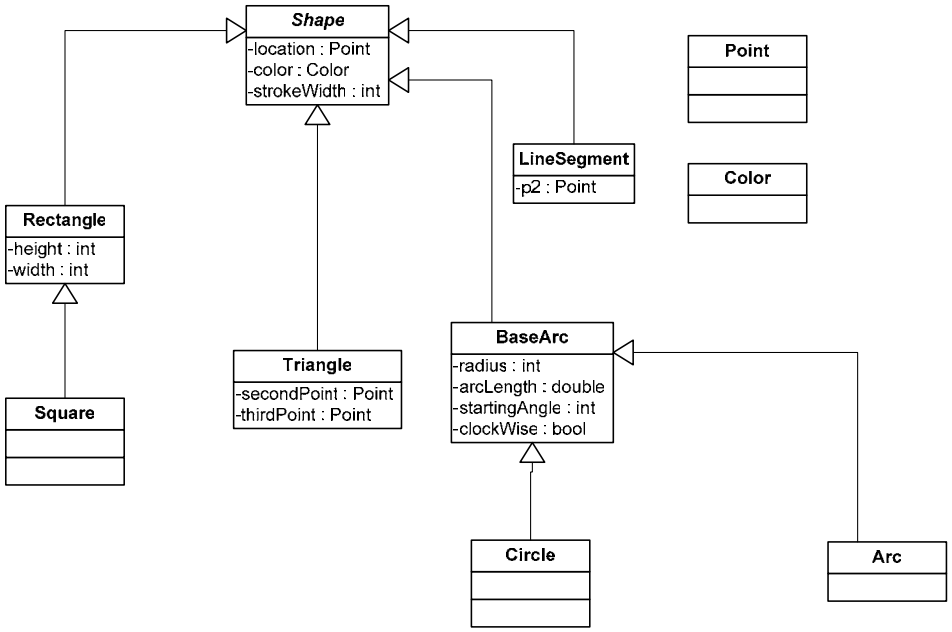


Some key observations regarding the diagram are as follows:

- Shape is an abstract class.
- Shape is the parent to Rectangle, Triangle, LineSegment and BaseArc.
- Rectangle, Triangle, LineSegment and BaseArc all inherit from Shape, so all methods found in Shape are available to instances of these subclasses.
- Each Shape subclass defines its own drawIn() method. Each drawIn() method contains code that draws the specific shape type.
- The Shape class also defines a drawIn() method, but this method does not contain any code. The Shape class is declaring that all instances of Shape (which will actually be an instance of some subclass such as Triangle) must know how to draw themselves. When considering the Shape class, which is a definition of shapes in general, we cannot possibly know the exact code to write because the concept of "Shape" is *too* general. Although we cannot put in detailed code describing how to draw we can assert the conceptual truth that shapes *can be drawn*. The drawIn() method in Shape is an example of an **abstract method**. All subclasses of Shape must either provide an implementation of drawIn() or they must themselves be declared as abstract.
- Square inherits from Rectangle. This means that Rectangle is simultaneously both a parent class and a child class. Square inherits methods both from its parent, as well as from any other ancestors higher in the hierarchy, such as Shape. A Square is therefore a kind of Rectangle and it also is a kind of Shape.
- Both Circle and Arc are children of BaseArc. Circles and Arcs are each also kinds of Shapes.
- Both Point and Color, although they do not have any parent classes shown, each have an implied parent class. Actually, every Java class has a parent class – the global Object class (the Object class itself is the only class that has no parent). The Object class defines operations that are legal to be performed on any Java object. Usually, subclasses will provide a customized version of these operations because the default behavior is not always desirable. We have seen two operations, both in Point, that are customizations of methods that were initially defined in Object. We provided customized implementations of the equals() and toString() methods which offer useful functionality for Point objects.

Shape Class Implementation

The diagram below shows the fields contained in each of the classes in the hierarchy.



Significant points regarding the diagram are discussed below.

- The Shape class defines fields that are common to all shape types. All shapes have a location, a color and a line thickness (stroke width). Each subclass implementation inherits these fields.

- Each subclass interprets its location point differently. For circles, the location is the center of the circle. For rectangles, the location is the lower left corner. For line segments and triangles, the location is the first point specified by the user. Notice that the field name is now generically named "location" rather than "center" or "lowerLeft" or "point one". When creating parent classes we must think and also code in generic terms. The variable name "location" is a generic term that can be applied to any shape.

- Each subclass adds fields as necessary to store information that is specific to that shape type. Rectangles have a width and a height. Arcs have a radius. Lines have one additional point and triangles have two additional points.

- Some classes such as Square, Arc and Circle do not define any additional fields. This is because the fields defined by the parent are sufficient. Subclasses at times may simply supply logic that puts constraints on the field values, such as a square having an equal width and height, or a circle having an arc length of 100%.

We will now examine the source code for Shape and also discuss the changes that were made to the shape subclasses that resulted in introducing inheritance.

- Open up the source code for the Shape class and find each of the code sections that are discussed below.

- The Shape class is declared to be abstract, since the concept of a shape is a very generic concept. We do not intend to ever create an instance of the Shape class. Any shape object will actually be an instance of a Shape subclass, such as Triangle. The class header line is shown below.

```
public abstract class Shape
```

- The purpose of the Shape class is to define all characteristics and behavior that are common to all shapes. In Shape we define fields which will hold values for the common shape characteristics. The fields are flagged as protected, which means that they are private to the outside world but are accessible to code in child classes.

```
protected Color color;
protected int strokeWidth;
protected Point location;
```

- The constructor for the Shape class accepts a location, color and stroke width. Clients of our shape classes will never call the constructor directly. Since Shape is abstract it is illegal to create a Shape instance. Clients will, however, create instances of Shape subclasses. The subclass constructors will call the Shape constructor in order to initialize fields that they have inherited from Shape.

- In the section above on class interfaces, we discussed the abstract method drawIn(). The drawIn() method is declared in the Shape class to assert that Shape subclass instances must have a defined drawIn() method. Shape subclasses are responsible for providing these implementations. Here, in the parent class we declare the method, flag it as abstract and leave the method body empty.

```
protected abstract void drawIn(Color color);
```

- The remainder of the Shape class contains implementations of methods that are common to all shapes. The code for methods such as draw(), erase() and setColor() can be written in a generic way that is applicable to every shape type.

- The moveTo() method is interesting because the implementation in the Shape class is sufficient for some but not all shapes. The code, shown below, works for shapes such as Circle and Rectangle that are defined by a single point. Shapes such as Triangle and LineSegment, however, require more work for shapes of their type to be moved. We cannot simply adjust the location of one point for these shapes. We must adjust all of the points that define the shape in order to implement the move operation. In situations like this where the base class implementation is not sufficient, derived classes can supply their own implementation which will **override** the base class implementation. Therefore, if a moveTo() operation is performed on a LineSegment, it will be the code defined in LineSegment that is executed. If a moveTo() operation is performed on a Rectangle, however, the code found in Shape will be executed, since Rectangle does not override the moveTo() method.

```
public void moveTo(Point newLocation)
{
  erase(); // erase ( at old location )
  location = newLocation; // change location
  draw();  // draw ( at new location )
}
```

We will now examine the Rectangle class, redefined to be a subclass of Shape. Similar changes were required for the other shape subclasses.

- Open up the Rectangle class source code and find each of the code sections relevant to the discussions below.

- The class header line (shown below) declares Rectangle to be a subclass of Shape. Since Rectangle does *not* have an abstract keyword specified the class is considered **concrete** and instances of the class are allowed to be created.

```
public class Rectangle extends Shape
```

- The Rectangle class no longer needs to define as many fields because it inherits several fields from Shape. The only fields that need to be defined are those that are specific to rectangles.

```
private int width;
private int height;
```

- The Rectangle constructor supplies the same interface to the user as before. It is worth emphasizing that our reworking of our system to incorporate inheritance has not changed *anything* from our user's perspective. Inheritance has allowed us to reuse code and avoid redundancy. From our client's perspective nothing has changed, because we have not changed the external interface that our clients have been using.

- The Rectangle class constructor contains as a first line a call to the Shape class constructor (accomplished with the **super** keyword). Derived class constructors *must* call their parent's constructor as their first line of code. We must ensure that our objects are initialized properly. The remainder of the child constructor is concerned with initializing fields that are unique to the child class.

- Since Rectangle derives from Shape, and since Shape declares an abstract drawIn() method, Rectangle *must* provide an implementation for drawIn().

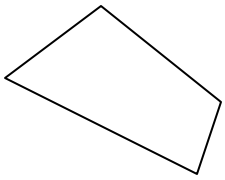
- Temporarily comment out the drawIn() method. (Use a multi-line comment. Put a "slash-star" (/*) before the method and "star-slash" (*/) after the method). Compile the Rectangle class. You should receive an error message explaining the necessity for Rectangle to define a drawIn() method. Remove the comment and make sure that the Rectangle class successfully compiles.

- The remaining code for Rectangle consists of methods that implement specific behavior for rectangles, such as setting or getting the height and width.

- Notice that many of the methods that used to be in Rectangle have been removed. Rectangle had contained many methods whose code was identical to equivalent methods in other shape classes. Several methods previously defined in Rectangle now reside in the Shape class. Rectangle (as well as the other shape subclasses) inherit these implementations. Redundancy is never good in any sort of programming. Placing common methods in one location (the base class) facilitates code reuse and avoids redundancy.

Assignment

Create a shape subclass that represents a quadrilateral. Quadrilaterals are four-sided shapes. Assume that these shapes are irregular (unlike Rectangle) and that the user needs to specify the four points of the shape. When drawing the shape draw the edges from point to point, in the order that the points were supplied by the user.



7.8. Lesson Document: Lesson 7 Polymorphism

Files Needed

Deliverables

Assignment hand-in. See assignment section at the end of the lesson.

Objectives

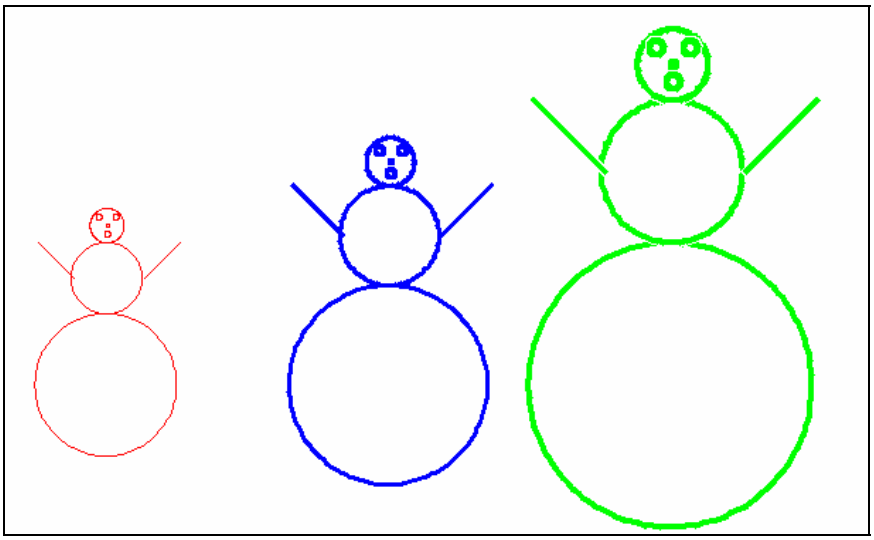
By the end of this lesson you will be able to:

- Appreciate the power of polymorphism
- Write source code that results in dynamic runtime behavior
- Process objects polymorphically
- Use a base class variable to hold subclass object instances
- Compare polymorphic and non-polymorphic code

Introduction: Picture Class as Concept

Thus far the pictures we have drawn with our class library have served mostly as test programs. The drawing of pictures has been a way to create several shape objects programmatically without having to create each shape object interactively in BlueJ. Our picture classes have not been reusable in any fashion. The pictures were drawn at a fixed location with fixed characteristics. Our task in this lesson will be to create Picture classes which allow our users to create more than one instance of a picture, with customizations of size, location and color. When implementing the Picture class we will rely heavily on the object oriented concept of polymorphism.

We have drawn simple pictures of a house and a face. The house picture was made up of a collection of triangles, circles, squares and rectangles. The face picture was made up of circles and arcs. Pictures should "know" their location and their size, so that we can move or scale the pictures. We should be able to create multiple instances of a picture with each instance having potentially different characteristics, such as the picture below of snowmen.



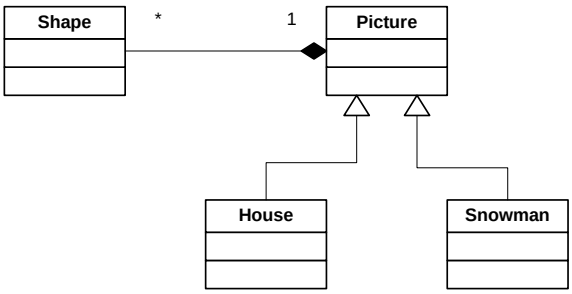
A snowman is made up of seven circles and two line segments. The shapes are sized and positioned relative to one another, which allows us to scale the picture and create snowmen of various sizes.

- Open up the v7_polymorphism project in BlueJ. Create and draw different instances of a House and Snowman, each with different characteristics.

A Picture can be defined as a collection of shapes.

Our Picture class will define pictures *in general*. The Picture class should define the methods and fields common to *all* pictures. The concept of "picture" is very abstract. A *particular* picture, such as a snowman, is *a kind of picture*, so the Snowman class, or any other tangible picture is related to the Picture class by means of inheritance.

The diagram below shows our class relationships.



Picture Class as Specification

What we can do to a picture is essentially what we can do to an individual shape. Pictures can be drawn, erased, moved and sized. We could change the color of all shapes in the picture to a new color or we could change the thickness of the lines.

We need a way to add shapes to a picture, to group them together as a collection.

The interface needed for Picture is shown below:

Method	Notes
Picture(Point location, int size)	Create a picture at a certain location in a certain size.
draw()	Draw the picture (draws each shape).
erase()	Erase the picture (erases each shape).
setStrokeWidth(int strokeWidth)	Change the stroke width of each shape in the picture.
setColor(Color color)	Change the color of each shape in the picture.
moveTo(Point newLocation)	Move the picture to a new location.
setSize(int newSize)	Resize the picture.
addShape(Shape shape)	Add a shape to the picture.

Picture Class as Implementation

There is a close connection between creating, sizing and moving a picture.

Upon creation we have to create a set of shape objects that are sized and positioned relative to one another. For example, the snowman is drawn by creating a circle for the base, then a circle half the base size for the body, and then a third circle half the body size for the head and so on. Sizing the picture will involve not just resizing shapes but also moving certain shapes. For example, if we make a snowman smaller, the head of the snowman will be a smaller circle but the head will also need to move down to stay connected to the body.

We could take two approaches to implementing a resize operation. One approach would be to cycle through the list of shapes and move and size each one. However, this code would be very redundant to the code in the constructor. When first creating the shape objects we need to determine where and how big they will be, based on the size and the location of the picture in general. We could put the shape creation code in a separate method and have the constructor, move and size operations call this code. We are essentially deciding between run-time performance and source code redundancy. The shapes will need to be erased and redrawn every time we move or size the picture, and this

is the main performance hit. Recreating the Java objects in memory would not slow down the performance much.

Each picture subclass then, will define a method to create, size and position the shape objects based on the size and location of the picture. This method will be an abstract method in the Picture class because the knowledge of the shape specifics will only be known by a Picture subclass that implements a particular picture. The abstract method will be called createShapes(). The access level is protected since our external clients will not be calling this method directly (rather we will call the method internally as needed).

```
protected abstract void createShapes();
```

The Picture subclass will call createShapes() after it has done any necessary subclass initialization (in the subclass constructor).

- Open the Snowman class source code and study it. Notice that Snowman derives from Picture. The bulk of the logic in Snowman exists in the createShapes() method. Notice how each shape is created in reference to another shape.

- Open up the House class source code. This class uses a slightly different coding style than that in the Snowman class, but essentially the same type of work is being done. The House class derives from Picture and supplies a customized createShapes() method which contains code to create the shapes that make up the picture of the house.

The setSize() and moveTo() methods can be defined generically in the Picture base class as shown below.

```
public void moveTo(Point newLocation)
{
    this.location = newLocation;
    erase();
    shapeList.clear();
    createShapes();
    draw();
}

public void setSize(int newSize)
{
    this.size = newSize;
    erase();
    shapeList.clear();
    createShapes();
    draw();
}
```

The fields of the Picture class are shown below.

```
// shapes that make up the picture
protected ArrayList<Shape> shapeList;

// location of the picture
protected Point location;

// relative picture size
protected int size;
```

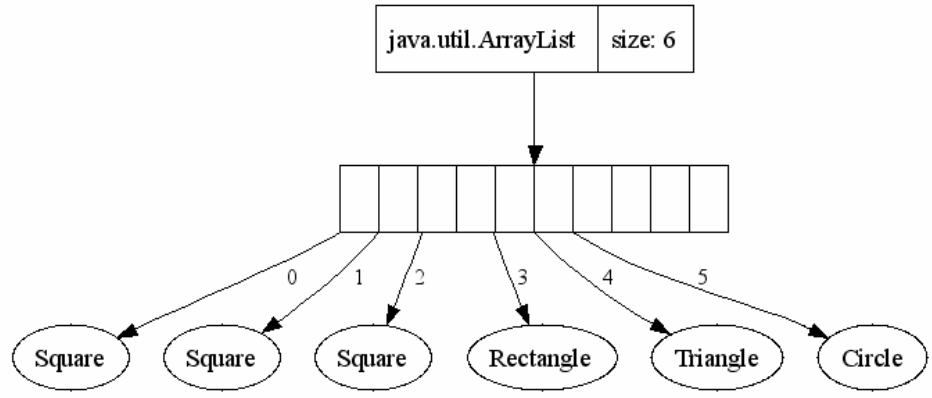
The location and size fields of the Picture class store the location and relative size of the picture. The shapeList field is of type ArrayList. ArrayList is a Java collection class defined in the java.util package. Collection classes allow us to gather objects together, add or remove objects from the collection and iterate over all items in the collection.

The most commonly used methods of the ArrayList class are shown in the table below:

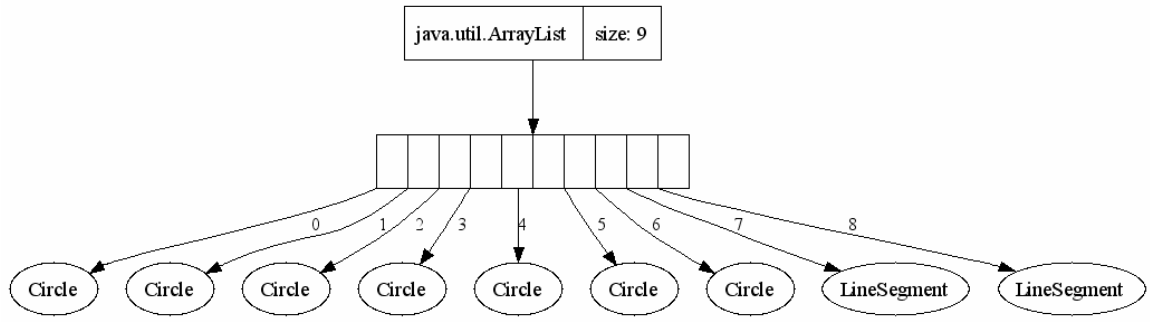
Method	Notes
add(Object o)	Add an object to the collection
clear()	Remove all items from the collection
contains(Object o)	Returns true if the collection contains the specified object
get(int index)	Returns a reference to the object at the specified position
remove(int index)	Removes the object at the specified position
size()	Returns the number of items in the collection

The moveTo() and setSize() methods each call a draw() and an erase() method. The draw and erase methods iterate over all shapes in the collection, drawing or erasing each shape, respectively.

It is important to note that the shape list contains objects of different types. We call such collections **heterogeneous collections**. For example, consider the figure below, which shows the shape list field contents for an instance of the House picture class.



The following picture shows the arrangements of objects for a Snowman instance.



The Java collection classes support heterogeneous collections because they are defined to work with Object class instances. Every Java object is the system inherits from the Object class, so essentially every Java object *is a kind of* Object. Defined in terms of Object, collection classes can hold objects of any type, since everything is an Object.

The code for the draw() method is below. The code for erase(), setColor() and setStrokeWidth() is very similar.

```
public void draw()
{
  Iterator i = shapeList.iterator();

  while (i.hasNext())
  {
      Shape shape = (Shape)i.next();
      shape.draw();
  }
}
```

The draw() method code contains a loop which iterates over every shape object in the picture and draws each shape. The code within the loop body needs a fair amount of explanation. The first line is:

```
Shape shape = (Shape)i.next();
```

The method called next() is used to retrieve the next item in the collection. As mentioned above, collection classes are defined to hold objects of type Object.

However, when retrieving items the object must be cast to a specific type in order to call methods on the object.

Note that we *don't* need to cast the object to the specific shape subclass type such as Circle. We cast to the base class type Shape instead.

A base class variable can hold an object reference of any class that is a subclass of the base type. A variable of type Shape can at one point hold a reference to an object of type Circle and then at a later point hold a reference to an object of type Square. This is legal because of the "is-a" notion of inheritance. A Circle *is a* Shape, so it is legal for a Shape variable to hold an object of type Circle.

Once we have the Shape object in a variable we call it's draw method:

```
shape.draw();
```

The draw() method is defined in the Shape class as:

```
public void draw()
{
    drawIn(this.color);
}
```

The drawIn() method is defined as abstract in Shape, meaning there is no method implementation defined.

So what code *is* run when drawIn() gets called ?

The answer is: *it depends*

The code that gets run depends on the *actual* type of the object contained in the Shape variable. Remember, the Shape variable can hold any object that is derived from Shape, such as Circle, Triangle, LineSegment, Arc, Rectangle or Square.

- If the Shape variable is holding a ***Rectangle*** instance, then ***Rectangle's*** version of drawIn() will get called.

- If the Shape variable is holding a ***Triangle*** instance, then ***Triangle's*** version of drawIn() will get called.

- and so on.

The single `shape.draw()` line of code will result in different and various run time behaviors over time, depending on the state of the system (a particular

Picture that is being drawn may have various arrangements of different kinds of shapes).

The term describing this dynamic behavior is **polymorphism**, which means "many forms".

Polymorphism has many advantages for system development. Parts of a system can be written without concern for all of the details of another component. The Picture class does not need to be concerned about all of the various kinds of shapes. From Picture's point of view, the items in its collection are all Shape objects. Picture can be written generically without concern or knowledge of the actual Shape subclasses it is operating on. The right code will get ran at run time. The term **dynamic binding** is used to describe the runtime behavior of dynamically figuring out the subclass type and executing the proper method. The burden is removed from the programmer for having to test to determine exact object type. We can add new shapes to the hierarchy and the Picture class code *will not only run unchanged, it will not need to be recompiled !*

The `shape.draw()` line of code is so short that its significance may be easily overlooked. In order to emphasize the elegance of polymorphism we will next look at a *non-polymorphic* version of the Picture class code.

- Open up the v7_polymorphism2 project in BlueJ.
- Open up the Picture class source code.

If we did not have the feature of polymorphism, when drawing shapes we would need to:

- Determine the exact object type of the object we retrieved from the collection.
- Cast the object to the type we had determined.
- Call the draw() method of the object.

The non-polymorphic version of the draw method is shown below:

```
  public void draw()
  {
    Iterator i = shapeList.iterator();
    Object shape;

    while (i.hasNext())
    {
        shape = i.next();
        if ( shape instanceof Circle )
        {
            Circle circle = (Circle)shape;
            circle.draw();
        } else
        if ( shape instanceof Arc )
        {
            Arc arc = (Arc)shape;
            arc.draw();
        } else
        if ( shape instanceof Rectangle )
        {
            Rectangle rectangle = (Rectangle)shape;
            rectangle.draw();
        } else
        if ( shape instanceof Triangle )
        {
            Triangle triangle = (Triangle)shape;
            triangle.draw();
        } else
        if ( shape instanceof Square )
        {
            Square square = (Square)shape;
            square.draw();
        } else
        if ( shape instanceof LineSegment )
        {
            LineSegment lineSegment = (LineSegment)shape;
            lineSegment.draw();
        }
    }
  }
```

The above code uses Java's **instanceof** operator to test to see what kind of instance the object retrieved from the collection is. Notice that we have to test for every known Shape subtype.

In older languages every method or function name had to be unique within the system, so having a method called draw() existing in multiple locations would be illegal. In such languages the code would look closer to what is shown below (lines modified from the above listing are in **bold**)

```
  public void draw()
  {
    Iterator i = shapeList.iterator();
    Object shape;

    while (i.hasNext())
    {
        shape = i.next();
        if ( shape instanceof Circle )
        {
            Circle circle = (Circle)shape;
            circle.drawCircle();
        } else
        if ( shape instanceof Arc )
        {
            Arc arc = (Arc)shape;
            arc.drawArc();
        } else
        if ( shape instanceof Rectangle )
        {
            Rectangle rectangle = (Rectangle)shape;
            rectangle.drawRectangle();
        } else
        if ( shape instanceof Triangle )
        {
            Triangle triangle = (Triangle)shape;
            triangle.drawTriangle();
        } else
        if ( shape instanceof Square )
        {
            Square square = (Square)shape;
            square.drawSquare();
        } else
        if ( shape instanceof LineSegment )
        {
            LineSegment lineSegment = (LineSegment)shape;
            lineSegment.drawLineSegment();
        }
    }
  }
```

Maintaining long series of if-tests like the one above was a near daily tasks for programmers of older languages. What is most significant is that **when a new subclass was added (such as adding a new Shape type), changes would be required in many parts of the system.**

- Open up the Picture class source code. View the setColor(), erase() and setStrokeWidth() methods.

The lengthy if statements in setColor(), erase() and setStrokeWidth() are nearly identical to those in the draw() method. If a new Shape type is added we would need to find every spot in our system which tested for shape type and *add a new case*, perhaps something like.

```
if ( shape instanceof Star )
{
    Star star = (Star)shape;
    star.drawStar();
}
```

While not extremely difficult coding to do, the fact that a human being has to frequently modify critical source code is a **recipe for disaster**.

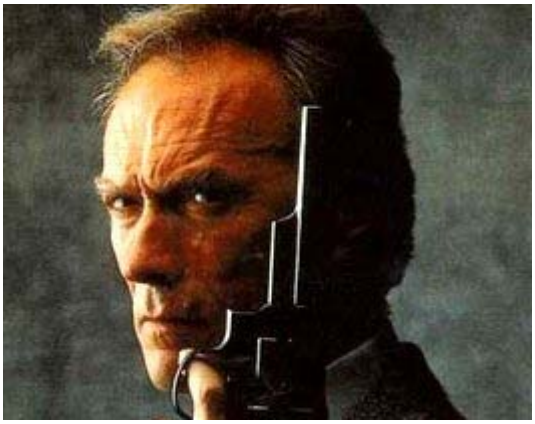
Some anecdotes might help to emphasize the importance of polymorphism.

Benjamin Franklin perhaps prophesied about legacy system development when writing his "For Want of a Nail":

For want of a nail, the shoe was lost,
For want of a shoe, the horse was lost.
For want of a horse, the rider was lost,
For want of the rider, the message was lost.
For want of the message, the battle was lost.
For want of the battle, the war was lost,
And all for the want of a horseshoe nail.
Benjamin Franklin

Not a nail perhaps, but easily a misplaced closing parenthesis or bracket or semi-colon could alter logic and have a ripple effect that could spell doom for a system under maintenance.

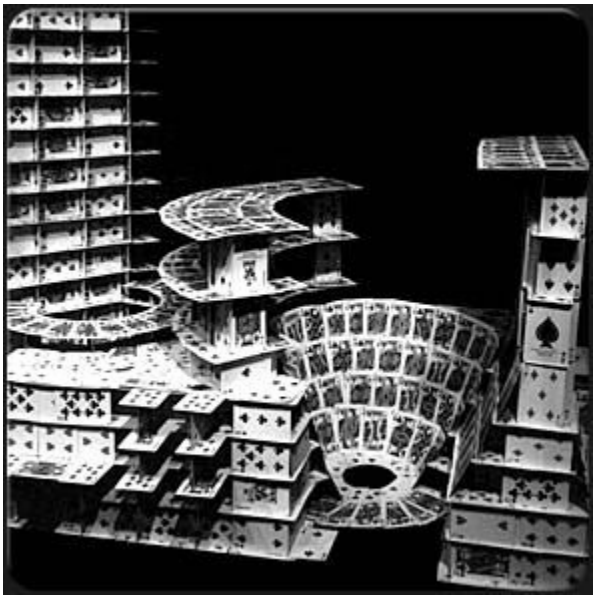
A programmer at IBM who got tired of having to modify critical system files every time a new device was supported by the AS/400 computer (which was often) hung this sign on his door alluding to Clint Eastwood's Dirty Harry "Go ahead, make my day" fame:



Go ahead, ask for one more device.

In our polymorphic solution the code for the Picture class *does not need to be touched* if a new Shape type is added. In object oriented systems the heart or the core of the system gets developed, tested and matures and remains relatively stable and in many instances does not need to be touched.

Systems have been known to "break" after programmers have gone in "only to add source code comments". Software systems can be very delicate and very difficult to test.



It would be desirable to not have to touch delicate systems.

Assignment

- Create a new Picture subclass and add it to the v7_polymorphism project (*not* the v7_polymorphism2 project).

- Create a picture of an office building that has multiple floors with windows on each floor.

- Your picture can be a simple one made up of mostly rectangles. However, use more than one shape type in your picture.

- When designing your picture determine the size of each of the shapes relative to the size of the picture as a whole. Select one point as a reference point for your picture and then determine the placement of other shapes based on the reference point and the size of the picture.

- Refer to the House and Snowman classes for examples of the suggested approach.

- Hand in your project files.

7.9. Lesson Document: Lesson 8 Composite Pattern – Pictures of Pictures

Files Needed

v8_compositePattern project files.

Deliverables

Assignment hand-in. See assignment section at the end of the lesson.

Objectives

By the end of this lesson you will be able to:

- Create pictures that are made up of other pictures
- Describe the Composite design pattern
- Use a Java interface to define a set of behaviors

Introduction

In the last lesson we discussed the notion of a Picture as a collection of shapes. With Picture defined in our system our users can create and work with named arrangements of shapes. The user can work with the Picture as a unit and draw or erase at the picture level. It is natural to take this line of thinking one step further. It would be desirable if our users could arrange multiple pictures into a new arrangement, give this arrangement a name and then work with this composite object as a unit. This "picture of pictures" would itself become a named unit, something that the user could draw, erase and perform other operations on. The user could create still more levels of grouping by combining composite pictures together to form other, more complicated pictures. The potential levels of grouping could have no limit. A "picture" , already an abstract term, is now getting even more abstract. What *is* a picture ? Perhaps we can think of a picture as something we can draw (and erase, .etc). A single Shape would fit the general definition of a picture, then. Although structurally different, shapes, groups of shapes formed into a picture, groups of pictures formed into a composite picture all exhibit the same set of behavior in that they can be drawn and erased. In this lesson we will provide features that allow users to arrange pictures of various arrangements and combine them together. We will accomplish this by designing in such a way that we can treat drawable things in a common manner.

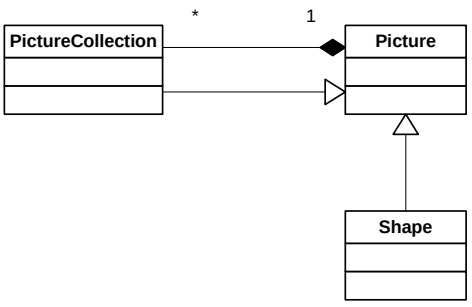
Java Interfaces

A Java interface is similar to an abstract base class, except that an interface can only contain abstract methods. An interface cannot contain any implemented methods. Rather than extending, as in a derived class, a class *implements* an interface by supplying implementations for the abstract methods defined in the interface. A class can only inherit from one base class, but a class may implement many interfaces. An interface is a way to define common behavior.

To handle pictures that are made up of pictures, we will modify our class design as follows:

- Rename the existing Picture class to be PictureCollection.
- Create a new Java interface class called Picture, declaring that Pictures are classes of things that can be drawn, erased, have their color changed or their line width changed.
- PictureCollection will now contain a collection of Pictures, rather than Shapes.
- Declare that PictureCollection implements the Picture interface, since we have methods in PictureCollection to draw, erase, setColor and setLineWidth. In other words, a collection of pictures is itself considered to be a picture.
- Declare that single shapes also implement the Picture interface. Shapes have every method required by the interface so we can treat single shapes as pictures.

The class relationships are shown in the diagram below.

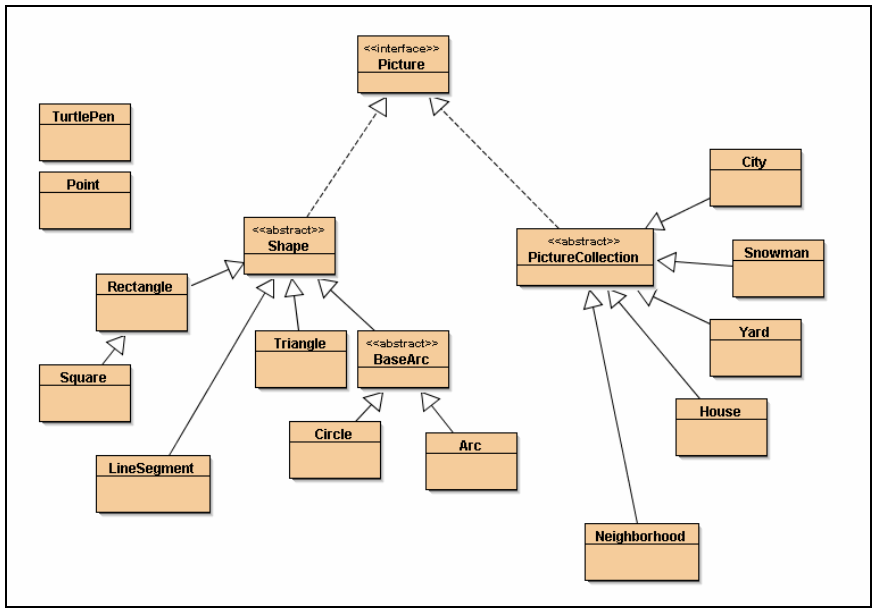


The above diagram shows that:

- a Shape is a Picture
- A PictureCollection is a Picture
- A PictureCollection contains Pictures

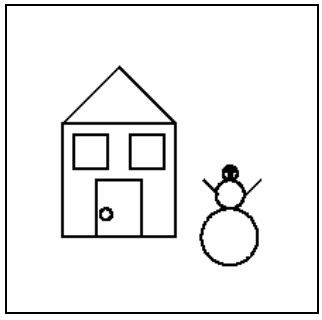
This relationship structure allows for a Picture to be anything from a single Shape, a collection of Shapes, a collection of Pictures or collections of collections of Pictures.

The relationships of all of our classes are shown below. Note that class diagrams in BlueJ don't show aggregation.



PictureCollection Examples

- Open up the v8_compositePattern project in BlueJ and study the source code for the pictures discussed below.

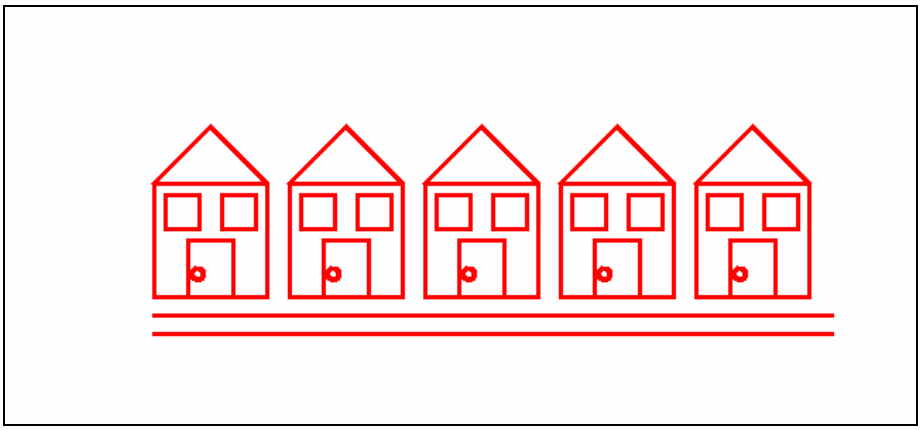


The Yard class, is defined as being made up of a House picture and a Snowman picture. The Yard class inherits from the PictureCollection class.

The createPictures() method (formerly called createShapes()) creates the sub pictures that make up the picture collection:

```
public void createPictures() throws Exception
{
    addPicture(new House(location,size,color,strokeWidth));
    addPicture(new Snowman(
        new Point((int)(location.x() + size*1.5), location.y()),
        size/4,color,strokeWidth));
}
```

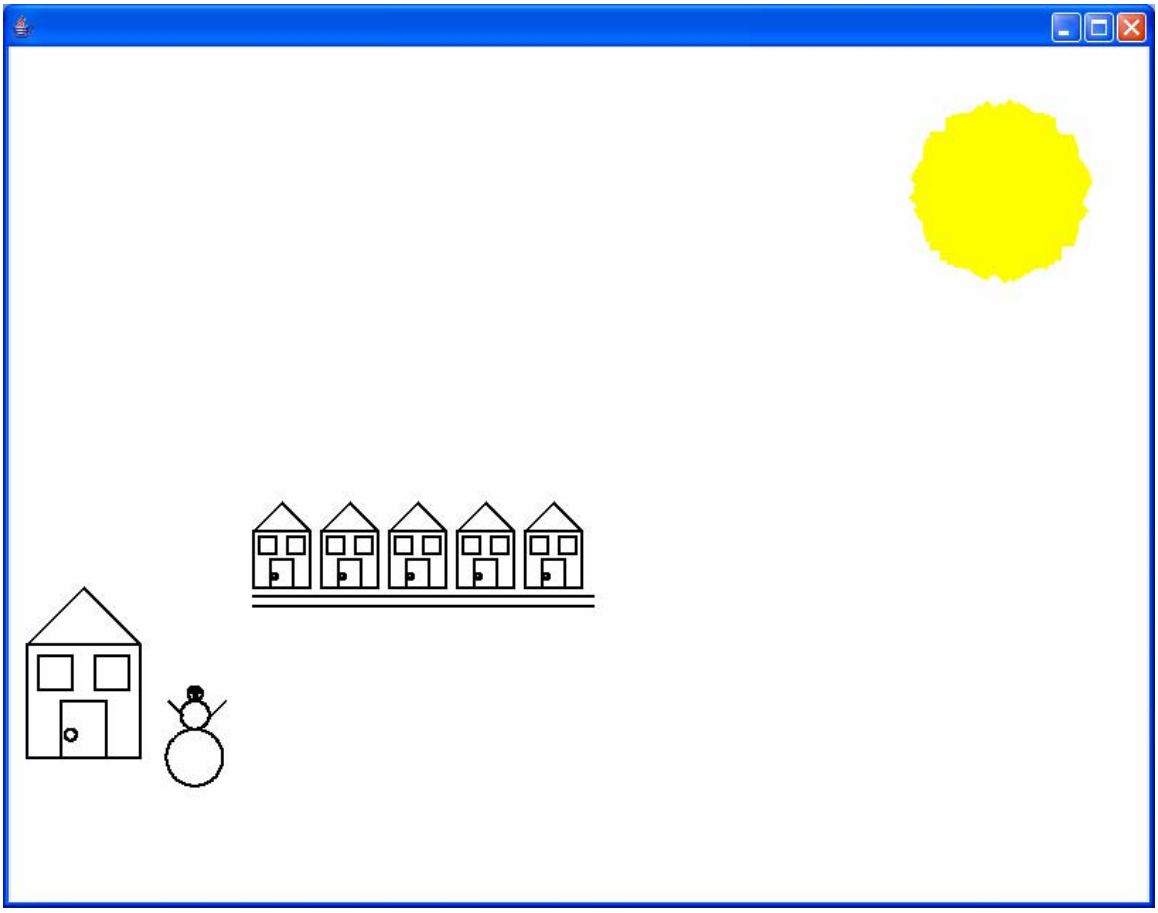
The Neighborhood picture collection class is defined as being made up of several House pictures and two line segments.



The pictures and shapes are created in the createPictures() method:

```
public void createPictures() throws Exception
{
    int offset = location.x();
    for ( int i = 0; i < 5; i++, offset += (int)size*1.2)
    {
        addPicture(new House(new Point(offset,location.y())
                             ,size,color,strokeWidth));
    }

    addPicture(new LineSegment(new Point(location.x(),
                                         location.y() – size/6),
                               new Point(offset,location.y() - size/6),
                               color,
                               strokeWidth));


    addPicture(new LineSegment(new Point(location.x(),
                                         location.y() - 2*size/6),
                               new Point(offset,location.y() - 2*size/6),
                               color,
                               strokeWidth));
}
```

The City class, which uses the most levels of nesting, is composed of a Yard object, a Neighborhood object and a Circle object for the sun (drawn with a very thick pen).



Picture Interface Implementation

The Picture interface code is shown below. An interface is usually briefly stated. What we are declaring here is that a Picture is something that can be drawn and erased and have its color and line width changed. A class which implements the Picture interface (and then *becomes* a type of Picture) is a class which supplies implementations for these methods declared in Picture.

```
interface Picture
{
  public void draw();
  public void erase();
  public void setColor(Color color);
  public void setStrokeWidth(int strokeWidth) throws Exception;
}
```

PictureCollection and Shape both implement the Picture interface. No other changes were needed for the Shape class.

```
public abstract class PictureCollection implements Picture
public abstract class Shape implements Picture
```

PictureCollection Implementation

The PictureCollection class, as discussed above, implements the Picture interface. The PictureCollection class is mostly the same code as the Picture class from the previous lesson, except that what the class contains is Pictures, rather than Shapes.

```
protected ArrayList<Picture> pictureList;
```

Composite Design Pattern

The classes in this lesson include the **composite** design pattern – a common design pattern in which composite and non-composite objects are treated the same. A Line is treated the same as a House. A House is treated the same way as a Neighborhood. All are objects that can be drawn. The composite pattern illustrates the flexibility of the Java language. We can define concepts very generically and abstractly and if we have a common interface between a set of classes we can write code that treats all of the objects uniformly.

Assignment

Study the Yard, Neighborhood and City picture collection classes. Come up with a picture collection of your own and add the class to the v8_compositePattern project.

Hand in your project files.

7.10. Lesson Document: Lesson 9 Object Persistence With XML

Files Needed

v9_objectPersistence project files.

Deliverables

Assignment hand-in. See assignment section at the end of the lesson.

Objectives

By the end of this lesson you will be able to:

- work with the JDOM class library.
- store an object's state as XML data.

Introduction

With our system our users can create complex pictures. Thus far we have created pictures and shape objects either interactively through BlueJ or programmatically with a test program. It is important to remember that if we packaged our features into an application that the end user would likely work through a front-end graphical interface that allowed them to select shapes and pictures from a palette and to drag and move and directly manipulate the shape objects. The code we have written so far is the "model" layer, and a graphical interface layer would communicate with our model layer and create objects in response to user mouse and keyboard events. Our end user would likely want to save the pictures that they create. The theme of this lesson is to consider how we might save our object state data into a file so that the objects that make up the picture could be recreated again at a later time. **Object persistence** is a term used to describe saving object state to disk.

XML and JDOM

XML is a suitable choice for a data format to use to store our object data. XML provides a flexible, intuitive, hierarchical format. XML is becoming a very standard choice as a data file format and many tools exist to assist with the processing of XML files. Java developers can use an open source class library called JDOM (www.jdom.org) to create or process XML data in a Java program. JDOM provides classes such as Document, Element and Attribute which

represent common XML constructs. We will work with JDOM classes in order to add object persistence capability to our shape and picture classes.

The key JDOM class is the Element class. The Element class represents an XML element. An XML element consists of an element name and element content. For example, the elements below are named **circle, point**, and **radius**. The content of radius is 100. The content of the point element is stored in two attribute values named **x** and **y**. The circle element is a parent element and its content is the set of child elements point and radius.

```
<circle>
  <point x="10" y="100" />
  <radius> 100 </radius>
</circle>
```

Our task is to add behavior to each shape and picture object so that the object knows how to store its state data as XML elements, in a structure whose hierarchy matches the structure of the object.

Each object must know how to store its state, even minor objects like Point and Color. An object such as Point can store its state in a manner similar to the XML code shown above. The Java toXML() method shown below has the logic necessary to store a Point object's data in an XML Element object.

```
// In Point class
public void toXML(Element parentElement)
{
     Element p = new Element("point");

     p.setAttribute(new Attribute("x",String.valueOf(this.x())));
     p.setAttribute(new Attribute("y",String.valueOf(this.y())));

     parentElement.addContent(p);
}
```

All XML data exists within a top level element called a root element. The Point object should not have to know on its own where within a document it is stored. The toXML() method is passed a parent element object. Any XML data produced should be placed within this parent element.

The Point class toXML() method creates an element named "point" which will be a child element of the parentElement parameter. Attribute objects are created which represent the x and y attributes of the point element.

Shape and picture objects can use Point's toXML() method to store their location field. The Shape or Picture objects must also define how the rest of their state data is to be stored.

PictureCollection's toXML() method is shown below.

```
  public void toXML(Element parentElement)
  {
    Element p = new Element("picture");
    Element szElement = new Element("size");
    location.toXML(p);
    szElement.addContent(String.valueOf(size));
    p.addContent(szElement);

    Iterator i = pictureList.iterator();
    Picture picture;

    while (i.hasNext())
    {
        picture = (Picture)i.next();
        picture.toXML(p);
    }


    parentElement.addContent(p);

  }
```

We create an element named "picture" and we create a child element called "size" to store the picture's size as a child element. We store the picture's location as an XML element by calling the Point class toXML() method. We then iterate over every picture in the collection and recursively ask each picture to store itself as XML data. Finally, we add the outer picture element as a child element to the parent element we were given.

Let us examine how a Shape element is stored. A Shape subclass instance will have some of its data inherited from the Shape class and perhaps some data stored by the subclass (such as Rectangle). Both the Shape class and subclasses that define their own fields need intelligence to store their state as XML.

The Shape version of toXML() is shown below.

```
  public void toXML(Element parentElement)
  {
    location.toXML(parentElement);

    Element c = new Element("color");

    c.setAttribute(new
      Attribute("red",String.valueOf(color.getRed())));

    c.setAttribute(new
      Attribute("green",String.valueOf(color.getGreen())));

    c.setAttribute(new
      Attribute("blue",String.valueOf(color.getBlue())));

    parentElement.addContent(c); // add the color

    Element s = new Element("strokeWidth");
    s.addContent(String.valueOf(strokeWidth));
    parentElement.addContent(s);
  }
```

The location is stored first. Next, we store the shape color by storing three attributes which contain the relative percentages of red, green and blue represented by the color. Finally, we store the stroke width.

The Shape class version of toXML() is called by derived classes as part of their toXML() processing. Rectangle's toXML() code is shown below.

```
  public void toXML(Element parentElement)
  {
    Element r = new Element("rectangle");

    super.toXML(r);

    Element w = new Element("width");
    w.addContent(String.valueOf(width));

    Element h = new Element("height");
    h.addContent(String.valueOf(height));

    r.addContent(w); // add the width
    r.addContent(h); // add the height

    parentElement.addContent(r);
  }
```

Rectangle's toXML() code creates a child rectangle element and then calls the Shape version of toXML() which stores the fields that Rectangle inherits from Shape. Then we create elements which store the height and width of the rectangle and we add those elements as child elements to the element representing the rectangle.

The other shape classes are coded in a similar manner, with each class having logic to store their specific field data as XML.

The XML data below shows the result of calling the toXML() method on a Yard object. Recall that a Yard object contains a House and a Snowman, with both the house and the snowman consisting of several basic shapes. The lines of code highlighted in yellow correspond to the shapes making up the House. The lines of code highlighted in grey correspond to the snowman objects.

The XML code was produced by the TestYard Java program. The code for TestYard is shown below:

```
Yard y = new Yard(new Point(10,100),80,java.awt.Color.black,2);

Element root = new Element("root");
Document myDocument = new Document(root);

y.toXML(root);

XMLOutputter outputter = new XMLOutputter("  ", true);


try {
  outputter.output(myDocument, System.out);
}
catch (java.io.IOException e) {
       e.printStackTrace();
}
```

We first create a Yard object. Next, we create a JDOM Element object and specify it as the root element of a JDOM Document object. We call the toXML() method of the Yard object (toXML() is inherited from the PictureCollection class), giving the root element object as the parentElement parameter. Finally, we use a JDOM XMLOutputter object to print the XML data to the terminal screen.

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
  <picture>
    <point x="10" y="100" />
    <size>80</size>
    <picture>
      <point x="10" y="100" />
      <size>80</size>
      <rectangle>
        <point x="10" y="100" />
        <color red="0" green="0" blue="0" />
        <strokeWidth>2</strokeWidth>
        <width>80</width>
        <height>80</height>
      </rectangle>
      <rectangle>
        <point x="18" y="148" />
        <color red="0" green="0" blue="0" />
        <strokeWidth>2</strokeWidth>
        <width>24</width>
        <height>24</height>
      </rectangle>
      <rectangle>
        <point x="58" y="148" />
        <color red="0" green="0" blue="0" />
        <strokeWidth>2</strokeWidth>
        <width>24</width>
        <height>24</height>
      </rectangle>
      <rectangle>
        <point x="34" y="100" />
        <color red="0" green="0" blue="0" />
        <strokeWidth>2</strokeWidth>
        <width>32</width>
        <height>40</height>
      </rectangle>
      <triangle>
        <point x="10" y="180" />
        <color red="0" green="0" blue="0" />
        <strokeWidth>2</strokeWidth>
        <point x="50" y="220" />
        <point x="90" y="180" />
      </triangle>
      <arc>
        <point x="42" y="116" />
        <color red="0" green="0" blue="0" />
        <strokeWidth>2</strokeWidth>
        <radius>4</radius>
        <startingAngle>90</startingAngle>
        <arcLength>1.0</arcLength>
        <clockWise>false</clockWise>
      </arc>
    </picture>
```

```
<picture>
      <point x="130" y="100" />
      <size>20</size>
      <arc>
        <point x="130" y="100" />
        <color red="0" green="0" blue="0" />
        <strokeWidth>2</strokeWidth>
        <radius>20</radius>
        <startingAngle>90</startingAngle>
        <arcLength>1.0</arcLength>
        <clockWise>false</clockWise>
      </arc>
      <arc>
        <point x="130" y="130" />
        <color red="0" green="0" blue="0" />
        <strokeWidth>2</strokeWidth>
        <radius>10</radius>
        <startingAngle>90</startingAngle>
        <arcLength>1.0</arcLength>
        <clockWise>false</clockWise>
      </arc>
      <arc>
        <point x="130" y="145" />
        <color red="0" green="0" blue="0" />
        <strokeWidth>2</strokeWidth>
        <radius>5</radius>
        <startingAngle>90</startingAngle>
        <arcLength>1.0</arcLength>
        <clockWise>false</clockWise>
      </arc>
      <arc>
        <point x="128" y="147" />
        <color red="0" green="0" blue="0" />
        <strokeWidth>2</strokeWidth>
        <radius>1</radius>
        <startingAngle>90</startingAngle>
        <arcLength>1.0</arcLength>
        <clockWise>false</clockWise>
      </arc>
      <arc>
        <point x="132" y="147" />
        <color red="0" green="0" blue="0" />
        <strokeWidth>2</strokeWidth>
        <radius>1</radius>
        <startingAngle>90</startingAngle>
        <arcLength>1.0</arcLength>
        <clockWise>false</clockWise>
      </arc>
      <arc>
        <point x="130" y="145" />
        <color red="0" green="0" blue="0" />
        <strokeWidth>2</strokeWidth>
        <radius>1</radius>
        <startingAngle>90</startingAngle>
        <arcLength>1.0</arcLength>
        <clockWise>false</clockWise>
      </arc>
      <arc>
        <point x="130" y="143" />
        <color red="0" green="0" blue="0" />
        <strokeWidth>2</strokeWidth>
        <radius>1</radius>
        <startingAngle>90</startingAngle>
        <arcLength>1.0</arcLength>
        <clockWise>false</clockWise>
      </arc>
      <lineSegment>
        <point x="120" y="130" />
        <color red="0" green="0" blue="0" />
        <strokeWidth>2</strokeWidth>
        <point x="110" y="140" />
      </lineSegment>
      <lineSegment>
        <point x="140" y="130" />
        <color red="0" green="0" blue="0" />
        <strokeWidth>2</strokeWidth>
        <point x="150" y="140" />
      </lineSegment>
    </picture>
  </picture>
</root>
```

Assignment

Add a copy of your solution to the inheritance lesson to this project. (In the inheritance lesson you created a new shape subclass).

For this assignment, add a toXML() method which will store the shape data as XML.

Study and compare the other shape subclasses.

Be sure to add this line at the top of your class:

```
import org.jdom.*;
```

7.11. Lesson Document: Conclusion To The Lessons

Where To Go From Here

If further development is done on the lessons it will likely occur in these areas:

- Allow for scaling of shapes.
- Redesign the classes to use Java 2D drawing rather than the TurtlePen. The TurtlePen will remain in the current lessons but we could demonstrate design patterns by showing how we could minimize knowledge of the pen even further and then replacing the pen altogether.
- Implement filled in shapes
- Animation. The drawing and erasing with TurtlePen is extremely slow. With a faster drawing mechanism it would be easy to draw, erase and move in a repetitive fashion in order to show primitive cartoon animation.
- Sort shapes according to their relative size by using a custom Comparator object.
- “Chain” shapes together in a picture in a different way. Design a solution so that each picture element “knows” what it is “attached” to.
- Create a GUI which a user can use to create and manipulate a drawing with direct interaction.

7.12. Guidelines For Students

Software packaging, distribution, review, testing and tracking are very important concepts.

These activities are the main activities that occur at the end of the software development lifecycle (the planning, design and construction of your programs represent the beginning phases of the lifecycle).

The handing in of your assignments is excellent practice for **packaging and distributing** software.

The instructor will perform two main activities when critiquing your assignments:

3. **Peer Code Review.** Your program code will be browsed and reviewed and feedback will be given on style and coding practices. Peer code reviews are commonplace in industry.

4. **"Black-Box" Testing.** Your program will then be ran and tested for correctness, robustness and usability. "Black Box" means that the tester is not concerned with **how** you made your program (i.e. the tester does not care what is "in" the box), but rather that the program runs correctly and efficiently from an end-user's point of view.

The instructor will send feedback to you regarding each assignment.

You are able to submit revisions of assignments additional times as opportunities to correct items and get additional feedback.

The creation, submission, review, testing, correction and resubmission of your assignments results in the creation of several files and documents. Additionally, there can be many items of email correspondence for each assignment (submission, first feedback, resubmission, final grading, .etc). The management of this volume of information is what is termed **software and document tracking**. It is critical that these important files and documents remain organized. Deliverables in the software industry are nearly always files, so file management is extremely important for your career. Capturing critical information including human conversation is extremely beneficial to any project.

Your consistent compliance with the program hand in guidelines will result **in an efficient software review, testing and feedback process**. You should use the program hand in guidelines as a checklist each time you submit an assignment.

7.13. Java Style Standards Guidelines

"Style" refers to how your code looks, and includes such items as indentation, capitalization, .etc.

- All class names should be uppercase: `class Temperature`
- All variable names should be lower case: `double salary;`
- Class and variable names should follow a scheme of capitalizing sub words (avoid the use of the underscore (_) and the dash (-)).

  ```
  class TemperatureConversion
  double  monthlySalary
  ```

- Your Source Files should contain a comment at the top (as describe in the hand-in guidelines).
- All code should be consistently indented from it's surrounding block.
  - A block is any section of code delimited by brackets: { }

```
// Nathan Dodge
// Example of Indentation
// The class line is flush left ( not indented ).
class IndentationExample
{
  // method code is indented within the class
  public static void main(String args[])
  {
    // all statements within a method are indented
    int i =0;

    if ( i > 0 )
    {
       // all statements with an if ( or else ) are indented
       System.out.println("a positive number");
    }

    for(int i = 1; i <= 10 ; i++ )
    {
       // all statements within a loop body are indented
       System.out.println(i);

       if (i==10)
       {
           // use additional indentation for nested compound
           // statements ( i.e. if test within a loop )
           System.out.println("We are almost done looping!");
       }
    }
  }
}
```

7.14. Submitting Code Revisions

Configuration Directions (do once).

1. Be sure you have read over the **File Comparison Tools** discussion found at the end of this document.
2. Download, install and experiment with WinMerge and CSDiff.

Revision Hand In Directions (do for every revision)

1. Be sure to save the current version of your code under a unique file name.
2. Save your revisions under a new file name.
3. When you have coded and tested your corrections in the new file, use CSDiff to create a file compare. (You can, in addition, use WinMerge, if interested, to more easily "see" the differences "side-by-side").
4. Be sure that the "old" file is on the left and that the "new" version is on the right.
5. Save the CSDiff file compare as an html document.
6. Include the html document when you are submitting your updated code.
7. When sending your revision for grading, **be sure to reply to the last message received from the instructor.**
    - To be more specific, reply to the last ***meaningful*** message received regarding that program. (not "resubmit according to guidelines" or other miscellaneous messages)
    - When replying, ensure that the **body of the message is included,** so that we maintain a complete documentation trail in the message as we correspond.
    - Attach a zip file containing new versions of files (as well as the file comparison document) before sending.

File Comparison Tools

WinMerge

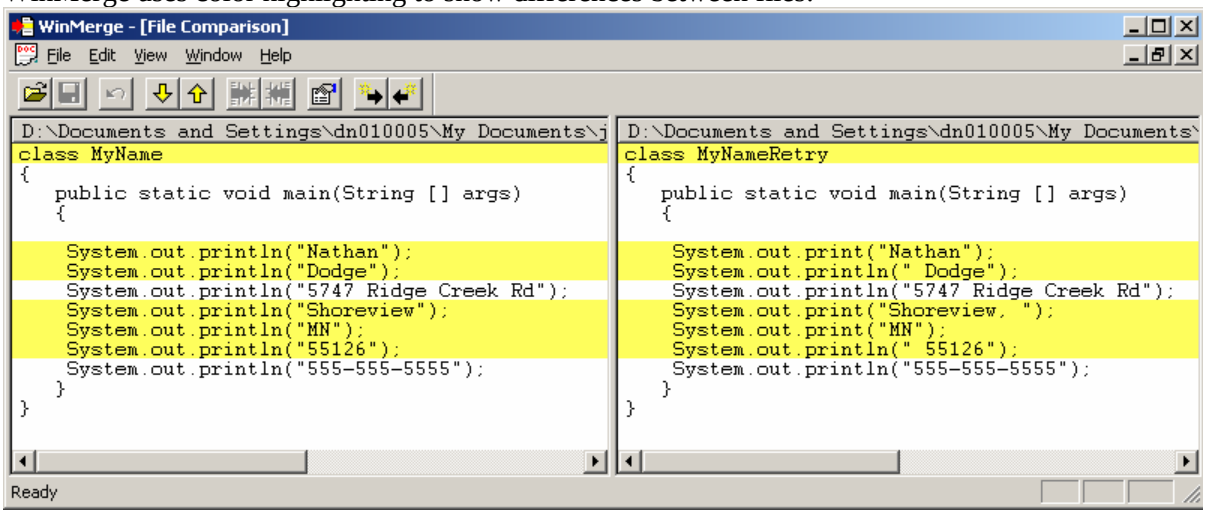
- Nice for visual side-by-side comparisons.
- http://winmerge.sourceforge.net/

CSDiff

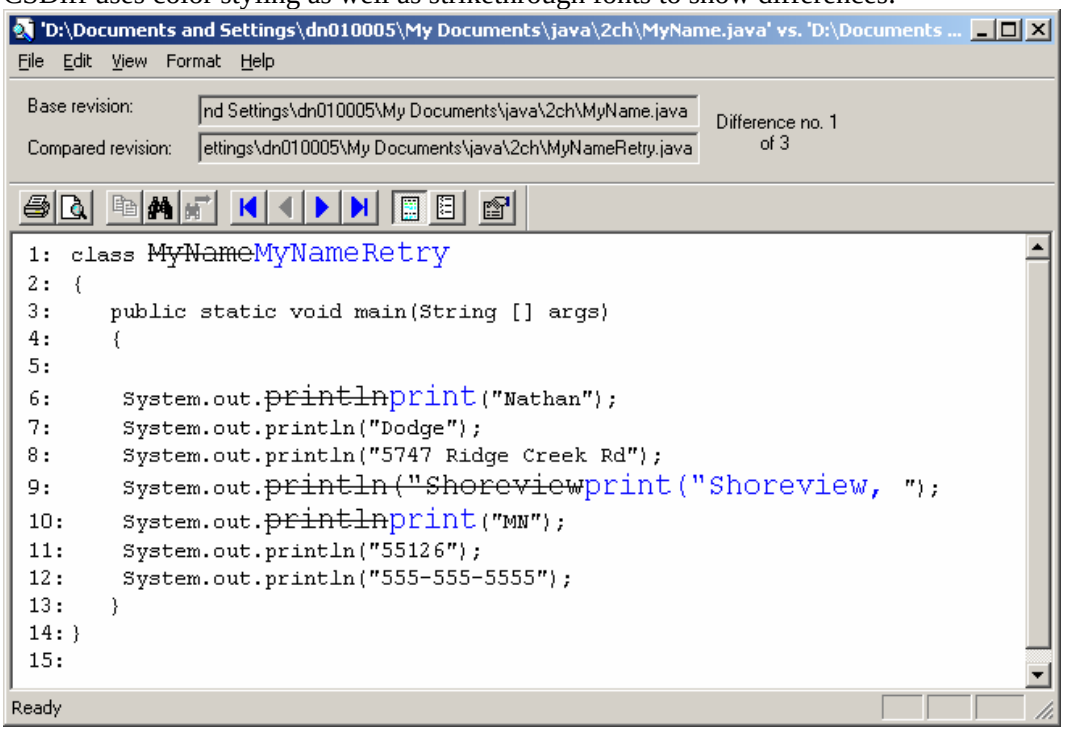
- Nice because you can print out a comparison or save a comparison as an HTML file.
- http://www.componentsoftware.com/products/csdiff/

Note: The examples below show comparisons of Java files, however any type of text files can be compared.

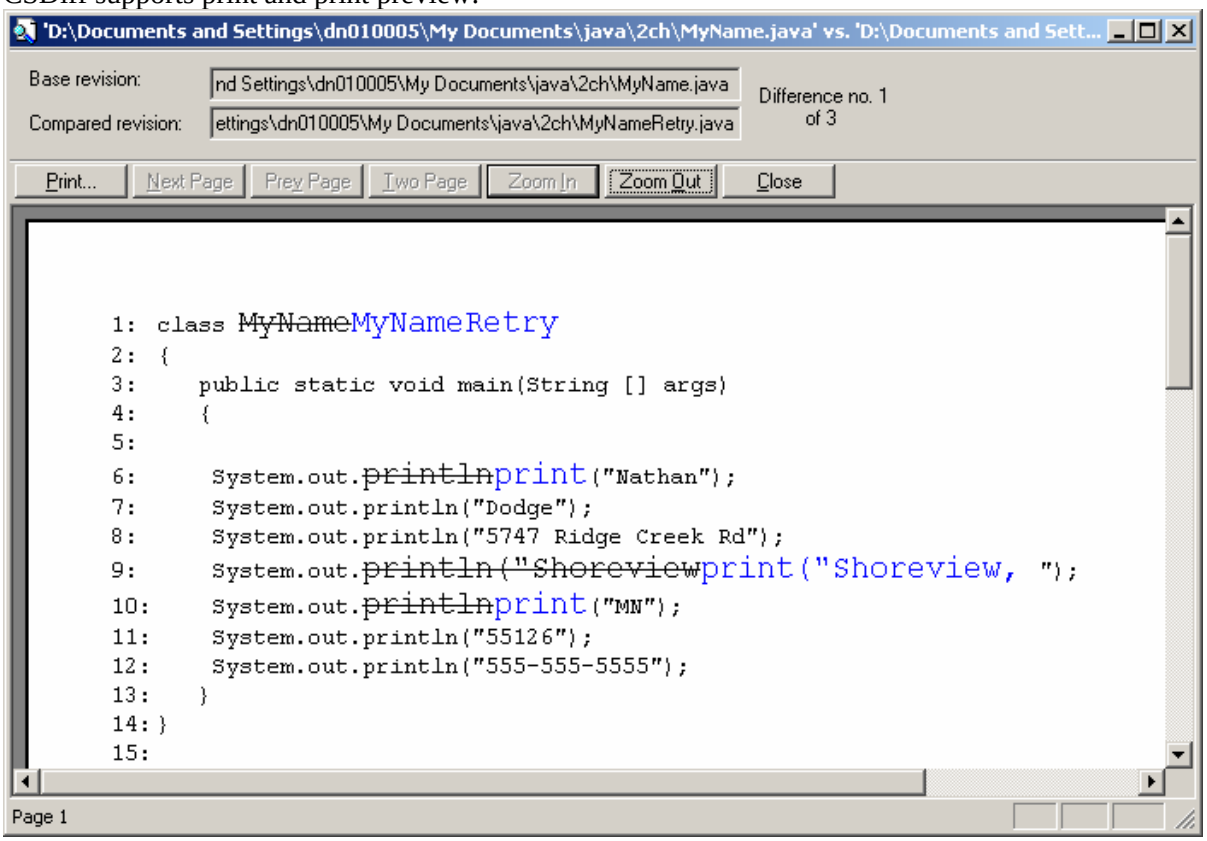
WinMerge uses color highlighting to show differences between files:



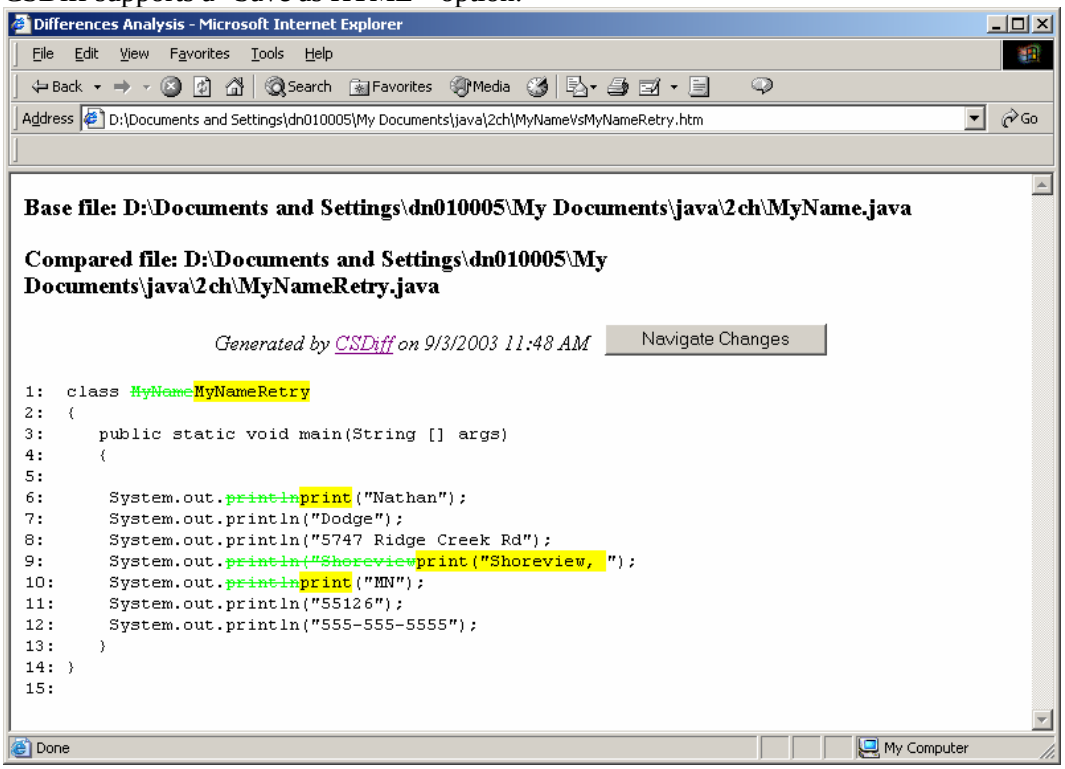
CSDiff uses color styling as well as ~~strikethrough~~ fonts to show differences:



CSDiff supports print and print preview:



CSDiff supports a "Save as HTML" option:



7.15. Permission to Use TurtleGraphics Library

Dodge, Nathan

From: Ken Lambert [LambertK@wlu.edu]
Sent: Wednesday, February 12, 2003 7:48 AM
To: nathan.dodge@neicoltech.org; MartinOsborne@netos.net
Subject: Re: permission to use TurtleGraphics in an educational setting

That all sound good, Nathan. Please send us a zip of your code when you are done.

Ken

>>> "Nathan Dodge" <nathan.dodge@neicoltech.org> 02/11/03 03:28PM >>>
I wish to use your TurtleGraphics package as part of a course I am developing on OOP.

The course will be taught at NEI College of Technology in Minneapolis, MN.

The course development is also serving as my thesis project for my master's degree from Regis University out of Denver.

My intended use is to use the TurtleGraphics code as a base for implementing a set of objects representing pictures and shapes (a picture will be a heterogeneous list of shapes, which will include lines, circles, rectangles, squares, triangles). The goal will be to develop a class hierarchy which demonstrates polymorphism, abstract classes, interfaces, collection classes and design patterns.

I see your note on your website:

"Authors who use this software in their textbooks must acknowledge its source and provide a link to this website. Such authors must write their
own verbiage to go along with their presentation of the software and may
not copy documentation from this website without our written permission"

If I intend to use your code as a base and create my own classes which use or extend your classes, do I need to do more than acknowledge the source and provide a website link ?

Thanks,

Nathan Dodge

NEI College of Technology