English language & third generation programming language pedagogical practice analysis

Vincent Falbo

English Language & Third Generation Programming Language Pedagogical Practice Analysis

By

Vincent Falbo

Thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Information Technology

Rochester Institute of Technology

B. Thomas Golisano College
of
Computing and Information Sciences

February 25, 2008

Thesis Reproduction Permission Form

Rochester Institute of Technology

B. Thomas Golisano College
of
Computing and Information Sciences

Master of Science in Information Technology

English Language & Third Generation Programming Language Pedagogical Practice Analysis

Table of Contents

ABSTRACT

In an effort to provide better computer programming instruction to more students at an introductory level, pedagogical methods could be improved using a paradigm of instruction based on the same strategies used to teach students spoken languages. Although many different methodologies of instruction have been explored in the past, this document identifies relationships between spoken languages and computer languages that encourage the exploration of the best practices of teaching English Language Arts so that they could be applied to computer programming instruction.

Those with backgrounds in mathematics and science initially completed programming tasks. Much literature about the problem solving aspects of programming is available; however, the researcher of this document found it difficult to obtain literature about the opportunities for growth provided by the humanities. This research is an attempt to foster the programming skills of students based on language skills. Given the similarities between spoken languages and object-oriented programming languages that have emerged, there is much encouragement that there may be possibilities for a new instructional paradigm.

Following is an analysis of how computer languages are taught and how English is taught, as well as a description of fundamental learning theories. Example demonstrations in a high-level programming language and example problem sets that could be used in an introductory programming course are included that use the best practices employed in English classes, the best practices of computer programming instruction, and the generally accepted learning principles defined by educational theorists.

INTRODUCTION

RECOMMENDATION AND PLAN

Different designs for instructing computer programming students in the literature have been investigated and implemented; however, they have not used the experience gained from English teachers to foster the development of this course of study that is concerned with the acquisition of language. Third generation programming languages are clearly languages, yet they are in their infancy compared to the English language. Since programming is still a relatively new discipline, it makes sense to explore what has been successful in other disciplines that have been taught for many years.

The programming language constructs and the constructs of the languages we speak and write are similar even though they were developed in isolation from one another. The goal of this work is to use one language used for human communication, English, and explore how we can use the theories and pedagogical practices of the language to teach students how to be better programmers. Drawing parallels between language classrooms and introductory computer science classrooms does not require much imagination (Applin, 2001).

Articulating why this method of instruction may be possible requires many steps. First, a comparison of the English language with the third generation language used in the provided demonstrations and problem sets, Java, that shows the similarities and translations that are possible. Given that these translations are possible, it follows that instructional techniques should transfer.

Second, an exploration of the factors that impact all learning – motivation, aptitude, environment – are outlined to show that this method of instructing programming students is in accord with educational psychology.   Using the conclusions of many twentieth century theorists to guide the development of these instructional materials shows that the theoretical underpinnings that lie at the foundation of the pedagogical strategies adopted by preschool teachers are applicable to adult education and every level in between. Regardless of the course of study and demographics of the learner, these fundamental issues must be addressed appropriately to meet the needs of the learner.

Next, an exploration of the generally accepted instructional techniques and the materials used to teach students how to read and write English is outlined. Although there will always be different philosophies, there is a benefit to extracting what has been working in English classes based on the fundamental principles that were established many years ago. The instructional techniques used to teach students how to read and write are the basis for the included demonstrations and problem sets since programming is essentially communicating via written communication with a machine.

Following the accepted instructional techniques used in English classrooms is information about the best practices used when teaching programming. After outlining these challenges, a list of work done by others to overcome the difficulties faced by programming professionals attempting to teach others is summarized. The work already done by others will be used to propel this effort to instruct programming students with the strategies used to teach students English.

Finally, a description of how the accompanying problem sets resolve some of the challenges facing computer programming instructors is included before the actual demonstrations and problem sets. Each problem set focuses on specific object concepts, how programmers communicate with objects, and how objects communicate with each other. These ideas are developed while students focus on the grammar necessary to communicate with the computer. Much like people can go to another country and focus on learning a language to communicate native to that area, students should be able to focus on how to use a language to communicate with a machine.

PURPOSE

To recognize the role this framework is intended to play in instructing students, it is helpful to state what this is not intended to do. It should be noted that this instructional paradigm is not intended to replace all of the current techniques employed by programming instructors. This is not an attempt to discontinue what has already been successful in assisting students or what has already been successful in assisting with the development of robust software; instead, this simply enhances instruction by providing students with a stronger foundation before engaging in more rigorous problem solving. Given the complexity of software development it should be no surprise that any one perspective could not guide all of the necessary instruction. This is the case with software development so it is logical that the same should be expected of software development instruction.

Currently there are systems engineers using the Unified Modeling Language (UML) to design software systems. Many different diagrams are used to model the various aspects of the software systems. For example, class diagrams and sequence diagrams play a role in capturing the static elements of the system and the dynamic elements of the system respectively. It would not be reasonable to attempt substituting one modeling tool for the other because their intended purposes are very different. Hence, there are many different modeling tools that comprise UML and the same should be true for teaching how to construct software.

UML is often used to model object-oriented systems while flow charts are often used to design successful structured programming functions and programs. Using UML models does not replace the need for constructing flow charts and doing a top-down design; yet, they enhance what was already in use. The flow charts, top-down design, and stepwise refinement remain methods for algorithm development. Algorithm development requires the more mathematical elements of the system development. Writing pseudo-code while doing a top-down design fostered a student's ability to translate the algorithm into code. This is not an attempt to replace the methods used for designing algorithms.

This is also not an attempt to design a tool for doing direct English to Java translations. The details of that process could lead to the development of higher level programming languages, but remains conjecture at this point. It is not an attempt to focus on specific programming languages or spoken languages either, but languages must be chosen to communicate these ideas with concrete examples. The upcoming details about how English translates into Java could be done using other spoken languages and other third generation programming languages, and it is simply an example of the possible translations. These translations are not an attempt to give examples of how translations should be done when designing software; they support the rationale that teaching students with a focus on language development is a viable option given that the same elements of grammar exist in both domains.

The main goal is to teach students the programming language and allow them to develop enough comfort with the language that it improves their ability to solve problems. According to Applin (1999), computer programming is a difficult task that is extremely complex and it requires that two things are being taught simultaneously. Students must know how to break a large problem into an organized series of small tasks, and they must know how to codify instructions for those tasks in a given programming language. Students need to be able to write code fluidly before engaging in the more rigorous work of decomposing problems. All programming languages are simply a means to an end – creating functional software. By having the ability to write code well and test ideas, students will have the language skills necessary to express themselves well and to implement sophisticated ideas.

This goal could be accomplished via the use of carefully designed demonstrations and problem sets similar to those included at the end of this document. This method would allow students to focus on the language as it is necessary for them to consider their studies in computer programming. The attrition rate in programming courses is quite high. For example, two professors researched this topic in Why the High Attrition Rate for Computer Science Students: Some Thoughts and Observations, and stated that at Southeastern Louisiana University there are over four hundred declared Computer Science majors; yet, each semester only about fifteen or twenty students graduate in the field (Beaubouef & Mason, 2005). This may be attributed to many reasons, including the fact that there may be a lack of understanding of how to put all of the programming language constructs together to form a coded problem solution (Applin, 1999). By changing the arrangement of topics and the focus of instruction students could have a different experience.

EXPLORATORY STUDY

The theory that instructional techniques in the English class transfer to programming classes given the similarities of the languages is not tested in this report through a formal exploratory research study. A strong rationale for this study emerges and lessons are developed in this spirit that could be used as a model for such an exploratory study. A formal exploratory research study would require a realistic sample size and control groups that may only be available on college campuses where students seek technical degrees. It would also require more materials to be developed that would be enough for one semester of work. Research has been done by others that nods at the possibility of a study of this nature, and a high correlation would indicate an important link between this method of instruction and the resulting improved programming skills of students (Buckland, 1996). Such a correlation would give additional credibility to the argument that students need to write statements that include objects based on the grammar of the language.

The fact that a formal exploratory research study has not been conducted should not detract from the logical argument that this method of instruction would foster student development. As stated in *Research Perspectives on the Objects-Early Debate* (Lister et al., 2006), educational research should not necessarily be suspect because it is not quantitative. Those who believe that research is suspect often have been exposed to a tradition of quantitative research and natural sciences. Many computer scientists may be inclined to think that education research should use the same quantitative methods.

Education is about ideas and it behooves us to explore ideas. Programming may be as much about language acquisition and development as it is about mathematics and problem solving, and further research about this idea may lead to the progress of computer science and information technology disciplines. If there are unexplored relationships between programming languages and spoken languages, it could lead to a new paradigm for instruction.

LITERATURE REVIEW

Reviewing the literature has provided the researcher with information about general learning principles that have been established by educational theorists, information about methods of teaching computer programming, and information about methods of teaching English Language Arts. To develop an instructional paradigm solely based on the fact that grammatical translations exist between languages would be remiss; therefore, the following accepted educational theories and successful practices must be applied. This summary about various widely accepted ideas from educational theorists prompts the need to explore this instructional paradigm since it fosters what educational theorists have known for many years.

EDUCATIONAL THEORISTS & GENERAL LEARNING PRINCIPLES

Material has already been written about how students learn and the best way to instruct students at various ages. Volumes can still be developed; the goal now is not to debate these theories, but to describe the learning principles chosen to guide this work. The theoretical underpinnings of how people learn were identified by many theorists over the past century. Identified below are generally accepted principles that are applied throughout the development of the accompanying materials:

- Environment that includes appropriate furnishings and atmosphere for the learner
- Instructor observation and assistance
- Motivation due to purposeful work, immediate feedback, and positive feedback
- Appropriate level of instruction
- Making connections
- Developing meaning constructively
- Repetition and practice without penalty

These principles were clearly articulated by educators like Maria Montessori, Jean Piaget, Lev Vygotsky, and others with a passion for guiding the learning process of individuals. Maria Montessori provided so much insight into how to create a learning environment for students that we have reached a point where we take her influence for granted and no longer note the environments that we create for children (Mooney, 2000). At the forefront of her contributions, was the need to provide children with child-sized furnishings that made learning possible. This need was based in the belief that a learner needs to be able to access and explore physical elements in the environment. It follows that Maria Montessori believed that instructors must be able to observe their students in order to be able to help them (Mooney, 2000). She was the first woman allowed in medical school in Italy, and her medical training clearly influenced her belief that in order to teach you need to observe the learners if you were going to be able to help them. She fostered the belief that large blocks of unstructured time were required for children to explore and build meaning. This time to play was not considered a break from the learning process; instead, it was a fundamental building block of the learning process.

Piaget continued to foster the belief that children need blocks of time to play and that they would build meaning for themselves during this time (Mooney, 2000). Piaget's stages of cognitive development indicate that there is a linear process to learning and that we cannot expect children to learn when they are not allowed to progress through necessary stages. Without this understanding of the learner's thought process, attempts at instructing others would be futile.

Being knowledgeable of how to create an appropriate environment and how children acquire new information allows instructors to focus on their role in orchestrating the learning process for others. Perhaps Lev Vygotsky provides us with the most significant amount of information about our roles in fostering the thought process of others (Daniels, 1996). His work defines Zones of Proximal Development (ZPD) and how to reach individuals at different levels. Vygotsky defines an independent level, an instructional level, and a frustration level for all individuals (Daniels, 1996). The independent level is the level where a person can function without the assistance of others. The frustration level is the level where a person will no longer engage in a learning activity on his own because the chance for succeeding does not exist. The instructional level is the level where a person can succeed with the assistance of a peer or teacher. This level is the level at which all instruction should be designed and it generally involves only ten to twenty percent of new information or skills. Expecting others to perform when there is a higher percentage of unfamiliar material would lead one to the frustration level and ultimately quitting by the learner. Vygotsky's work clearly indicates the need for the instructor and peers to be near when trying to synthesize new information.

COMPUTER PROGRAMMING

The Evolution of Languages used in Undergraduate Programming Courses

Computer programming has gone through several changes that will be referred to here in three phases: pre-structured programming, structured programming, and post-structured programming. Structured programming techniques were designed to make it easier to create logically correct programs. Rules for writing small modules, procedures, were devised to make it possible to give computers precise steps. These modules contained sequence, selection, and repetition instead of GOTO statements. This structured approach views a system as a collection of computer programs (Satzinger & Orvik, 2001).

Prior to the 1970's programming was in its infancy and programmers were not yet using structured programming. Based on the work of Creak (2003), it would be fair to say that prior to structured programming the BASIC programming language provided labels, variables, and parameters, necessary to begin communicating with hardware, and the work of the programmer was still largely concerned with managing bits in memory. Machines had different processors, which meant that there were different compilers, and the work of the programmer required the execution of commands based on line numbers and commands like PRINT, LET, and GOTO. The structure of a program was mainly dictated by the order in which the instructions were carried out (Creak, 2003). Understanding bit-level manipulation and hardware is an important part of the history of programming that is worth mentioning, and it contributed to the current state, but it does not require extensive exploration for this work. There was a place for BASIC and not using a structured programming language, but that time has passed based on a new environment and a new time (Creak, 2003).

It was during the 1970's when the GOTO statement began to disappear from programs and the structured languages began a new era that fostered a top-down approach to problem solving (Mitchell, 2000). It was during the 1970's and 1980's that there appeared to be a common understanding about teaching programming with top-down design, and the structured languages taught said nothing of objects. During this structured programming era Pascal was used in most coursework in teaching structured programming; Pascal was easy to learn for beginners and the environment offered sufficient support to the students (Brilliant & Wiseman, 1996).

It was when the post-structured programming era arrived that what to teach started to become more complicated by other paradigms. The 1990's were in disarray and instructors realized that object oriented programming was not just a simple add-on to structured programming. The confusion was created by different languages being touted at the introductory level, C++ being instituted for the Advanced Placement (AP) exam in computer science, and new visual approaches to programming being promoted by the major players in the programming industry: Sun Microsystems and Microsoft (Mitchell, 2001). These changes created questions about what to teach that began many conversations, and it was clear that instructors were uneasy about letting go of low-level details (Mitchell, 2001).

It was also during the 1990's that instructors realized that objects were not going to remain isolated in upper electives (Mazaitis, 1993). Functional decomposition and structured analysis was changing to a new paradigm, and conversations about the paradigm shift to object-oriented programming were taking place. Exactly how this shift to object-oriented programming would take place was not clear, and not all instructors have made that shift in the same way. Many began to make the transition to object-oriented programming through C++ (Mazaitis, 1993). Since instructors were familiar with C it was a logical transition, but one that did not require the use of objects. Others were using languages like Scheme and Eiffel, but learned that students were overwhelmed by libraries or struggled to learn about the environment (Mazaitis, 1993). Challenges were present, but it was clear that objects offered more powerful and flexible encapsulations (Rasala, 2000).

Currently, Java is taught in many undergraduate programs. Java has replaced many languages that were previously used in introductory courses that taught structured programming (Clark, MacNish, & Royle, 1998). The reason why Java has emerged as a dominant teaching language is an interesting topic, but it will not be explored in great detail. Given its acceptance in many undergraduate courses, it will be used in the examples in this document.

What Has Been Learned

Throughout the past several decades many changes have taken place, but there are some underlying themes that are clearly present throughout these changes. Programming should be presented to students with an environment that is simplistic (Kolling, 2003). This ease of use means that they have the opportunity to work at higher levels of abstraction (Kolling, 1999). Looking at the history of programming also indicates that it should be possible to create programs with less effort, and it should be possible for non-specialists to use computers to solve problems (Bergin, 2007).

Beginning students are obviously not specialists, and they will also need to be motivated to learn the material that is being presented to them. Using an industry standard language that is being used to develop commercial software interests students, and it has been a decision making factor for faculty members (Brilliant & Wiseman 1996). It is important that students learn a language that can solve current problems in industry. Furthermore, faculty members have made decisions about the language and methods of teaching based on other factors as well: having textbooks available, being sure faculty are prepared to teach, and being realistic about the amount of material being covered in introductory level courses (Brilliant & Wiseman 1996).

The literature also contains information about some attempts at teaching object-oriented programming that did not work. In general they involve the need of the programmer to address low-level machine internals instead of remaining in the problem domain, and a specific example is the difficulty that many instructors experienced when using C++ to teach object-oriented programming (Kolling, 1999). C++ was criticized for its lack of type safety because of the risk for errors that it created and memory leaks that were caused by improper storage handling (Kolling, 1999). These are clearly not issues that make learning programming easy. The use of constructs for low-level manipulations and forcing programmers to think at an unnecessarily low level further complicate learning how to program. The object model supported by C++ is complex, and using this hybrid language was not easy to incorporate into a programming course for beginners.

These past several decades have taught instructors much about how to best introduce new students to programming, yet challenges remain and new solutions could be explored. An exploration of the current pedagogical practices involved with teaching object-oriented programming may provide further insight.

Pedagogical Practices

The changes that have taken place in undergraduate courses bring us to an investigation of how object-oriented programming is being taught. A review of the literature reveals many different teaching methods:

- Microworlds
- Specific environments
- Toolkits and libraries
- Case studies
- Robotics
- Outcomes based instruction
- Test Driven Development
- Writing

Each of these teaching methods has strengths and weaknesses, and each is briefly summarized below. Comments about the literature are included that support or dispute how successful these methods are in assisting beginning programming students.

A significant number of “microworld” environments have been designed to assist students (Powers, Ecott, Hirshfield, 2007). The microworld category of programming tools is the category that allows storytelling to be incorporated into programming. Powers, Ecott, and Hirshfield (2007) mention several different microworld environments – Alice, Karel the Robot, Jeroo, and Greenfoot – but focus on Alice since it has been the subject of efficacy studies. Alice allows for the creation of 3-D graphical objects that can be added and manipulated in a virtual world. Teaching with Alice allows students to drag and drop objects into the environment instead of focusing on various syntax rules, and students construct videos using the graphical objects.

Alice's interface delays learning syntax and forces the construction of programs that are syntactically correct. In the observation and analysis of Alice (Powers et al., 2007), there are several reasons why students hit a wall when transitioning to other languages and environments.

- Students were so engrossed in the display of 3-D objects that they did not focus on programming concepts
- Confidence developed with Alice did not transition to other environments
- Confidence was lost due to syntax related problems when students knew what they wanted to do, but couldn't implement it
- The storytelling aspect of Alice may have contributed to the wall students hit when transitioning to other languages
- The Alice object model is not thoroughly implemented or object oriented
- There is a blur between object and class concepts
- The metaphor created in Alice quickly broke down when investigating more advanced topics
- Misconceptions created while using Alice could be detrimental to the development of programming skills

- Without visual cues, students were confused about the organization of a program
- Pedagogical pitfalls to the approach

It is noteworthy, that Alice, like some other microworlds, is an environment. Exploring the various environments available for instruction is a vast topic that is not articulated at great length in this document; however, an appropriate instructional environment, BlueJ, has been chosen for this work for reasons stated later. Environments for instruction should be used for the goal of allowing students to focus on programming tasks instead of struggling with the environment (Kolling, Rosenberg, 2001).

Another attempt to organize the pedagogical practices of programming instructors involves an outcomes-based approach. By definition, this approach focuses on getting a view of a course, its sequence, and its curriculum, and constructing assessments that guide instruction that help achieve the required skills with a product-oriented approach. With this approach goals and objectives are identified, performance criteria are established that determine observable student behaviors that show if the objective is reached, student learning activities are designed to help them achieve the performance criteria, evaluation methods are identified, a timeline is constructed for the implementation of the evaluation methods, and feedback is disseminated and used for upcoming lessons (Cooper, Cassel, Cunningham, Moskal, 2005).

The outcome-based approach may not be in use because there are few models to choose from, programming educators have little experience using them, and faculty are not trained in setting up objectives and measuring outcomes (Cooper et al., 2005). Although this could give a view of a course and curriculum in a different way, most computer science faculty have not been trained in assessment and computer science departments are not prepared to implement this strategy.

The use of toolkits and libraries is another approach used to assist beginning students. Toolkits consist of code designed to meet three specific criteria: necessity, economy, and pedagogy. They are necessary because they allow students to engage in computing activities that would otherwise not be possible, economical because they avoid repetitive programming tasks, and pedagogically sound because they allow for examples that show how concepts interact and play out on a large scale (Rasala, 2000). Instead of teaching principles of computing using features of a chosen language, students are taught in the framework of toolkits, and it maintains student interest and because of the pedagogical importance of using rich examples that are possible with toolkits (Rasala, 2000). According to Rasala (2000), this toolkit approach allows students to be shielded from ugly system details while thinking in terms of the problem being solved instead of the machine. There are other benefits of using toolkits to teach students: it allows for examples that are richer than those that could appear in a text book and it allows student programming to be more precise in a part of a program where new concepts are central (2000).

A similar approach to using toolkits created by Bruce, Danyluk, and Murtagh (2001), uses a library approach that provides many of the advantages of a microworld, while not being a microworld. Due to fear that the microworld approach leaves students feeling that they can't write complete programs, the objectdraw library was constructed in an effort to use graphics and to teach objects-first – the library reaps the benefits of the microworld approach while having the additional advantage of remaining a useful tool throughout the course (Bruce, Danyluk, Murtagh, 2001).

Educators have recognized for a long time that using robots as a teaching tool and motivator has potential (Fagin, Merkle, 2002). According to Fagin and Merkle (2002), quantitative studies that assess the effectiveness of robots was missing from the literature until their formal study showed that test scores were actually lower in programming sections that used robotics than those that did not, and using robots did not have any measurable effect on students choosing computer science or engineering as a potential field of study when they explored the merits of using robots. Based on this quantitative study that used a large sample with solid, reliable data, no further exploration of the use of robotics is necessary for the exploration of this topic.

Another pedagogical practice used in programming classes is writing to learn (WTL). WTL is different from the usual writing for a professional audience done with writing in the discipline (WID). The focus of WTL is to focus students on being intellectually active and use writing to think and develop understanding (Hoffman, Dansdill, Herscovici, 2006). According to Hoffman, Dansdill, and Herscovici (2006), this type of writing is useful for several reasons:

- Students are not writing to develop a professional skill, but because it is an effective means of student-centered teaching and learning
- It requires students to focus on main topics and reinforce concepts related to the exercise they are completing
- Thinking about a program and writing about it facilitates the actual writing of the program and discourages students from writing code haphazardly.

According to Anewalt (qtd. in Hoffman et al., 121), an attempt should be made to determine the effectiveness of writing activities with long-term retention of concepts, but this research has not been done to support observations.

Test-driven development inspired a test-first approach to instruction to assist students. This test-first approach is motivated by students' need for immediate feedback in forming hypotheses and testing them instead of tweaking code by trial and error (Edwards, 2003). Students writing test cases before adding new code characterize this approach, and it is then assessed based on correctness, test completeness, test validity, and code quality (Edwards, 2003). According to Edwards (2003), this approach can be combined with many of the recent advances in computer science pedagogy, and it can be used in classrooms that are structured differently also: closed laboratory sessions for hands-on learning, pair programming and increased peer-to-peer learning are also possible; yet a question remains about if it is possible for students to write test cases from the start of CS1.

Some educators also use a case study approach. Case studies are complete applications that are designed for the learner to get exposure to programming techniques that would otherwise be too difficult for beginning programmers. Nevison and Wells (2004) state that it is imperative to teach objects from the very beginning of the course and have used a case study to give students visual feedback and reward while working in a context where concepts are presented in a simple setting within the more complex environment. This final method has proven to be useful for several reasons that include complexity in a controlled situation, introduction of objects early, and providing a realistic context for introducing application concepts and design patterns.

Each of the aforementioned pedagogical practices could contribute to the motivation of a new instructional paradigm; some for their merits and success, while others for lessons learned that would lead programming instruction in another direction. Perhaps even a subtle new direction would lead to significant improvement for programming students, and the direction could incorporate all of the good aspects of these pedagogical practices. The proposed pedagogical practices enhance the actual construction of code while building on the effectiveness of current practices.

ENGLISH LANGUAGE ARTS

A review of the literature about pedagogical practices and English language arts reveals findings that are categorized into four sections for this research:

- Teaching Methods
- The Teacher's Role
- Student Needs
- Writing Instruction

Each of these items is briefly described below, and this analysis is used as a rationale for the upcoming section that explains how each of these important elements in the English Language Arts classroom could be applied in computer programming classrooms. Worthy of note is the last section that focuses on writing instruction. Since the aspect of English Language Arts instruction that mainly relates to learning programming languages is writing, listening and speaking do not need to be explored. There is a need for students to read the language; however, the main goal is to write the language well enough to solve problems. Students translate requirements using a third generation programming language to solve a problem.

Pedagogical Practices

Teachers use different instructional strategies, and the methods of instruction chosen to guide this work refer to the elements of effective instruction outlined in the work by Madeline Hunter. Hunter's model of instruction includes several elements of instruction that are highlighted here: choosing an objective, input, modeling, checking for understanding, guided practice, and independent practice. Each of these elements is briefly described and will be of particular importance in the upcoming sections of this document.

The objective defines what will be taught based on the skills that have already been acquired by the students, and teachers begin by identifying the objective for the class and providing some input to the learners based on that objective (Hunter, 1982). This input is sometimes purely verbal, but it is often modeled for the students. Models could be concrete, replications, symbolic, verbal, written, or spoken, and they have four important characteristics. Models highlight critical attributes, avoid controversial issues, are accurate and unambiguous, and eventually lead to the introduction of nonexamplars (Hunter, 1982). Once this modeling is complete, it is often followed by a method of checking for student understanding. The teacher could ask students questions or observe them performing a basic task based on the instruction so their level of understanding can be noted. Given a reasonable level of understanding, students then engage in practice exercises that are guided by the teacher before beginning independent practice. Initial practice is extremely important because knowing how a task should be completed and actually completing it represents a significant leap in the learning process. Practice must be monitored by examining group progress, individual progress, and written responses.

There are other models of instruction; however, this method of instruction is the basis for the remainder of this research, and the upcoming section about the teacher's role assumes that the teacher is operating in one of these phases of instruction.

The Teacher's Role

The role of the teacher in the language arts class is very important and much research points to the fact that the teacher is the number one indicator in how successful students will be. The profound impact that a single teacher could have on a student is more significant than school level factors (Marzano, 2003).

LightBown and Spada (1999) comment on some behaviors that are of particular interest to this study:

- Errors are frequently corrected. Accuracy tends to be given priority over meaningful interaction.
- Input is structurally graded, simplified, and sequenced by the teacher and the textbook. Linguistic items are presented and practiced in isolation, one item at a time, in a sequence from what is assumed to be ‘simple’ to that which is ‘complex’ (p.94).

Frequent error correction and sequencing of linguistic items to be practiced in isolation makes sense in any language-centric discipline. The insight provided by Marzano, LightBown, and Spada highlights the need to explore how teaching materials are organized, and how students need materials to be organized in a particular fashion if they are going to be able to succeed.

Student Needs

Students have two particular needs when learning language. Given that students are motivated and engaged, they could complete reading and writing tasks, but they need to be assigned tasks that are within their Zone of Proximal Development (ZPD), and they need to be provided text to read that has been designed in their ZPD. As previously defined, the three zones of proximal development include the independent level, the instructional level, and the frustration level. These two needs of students’ when learning language requires them to be in their instructional level, which includes 10% - 20% of new material.

Following is an example of some work that has been created for beginning language arts students that emphasizes the importance of two types of text designed for beginners: pattern text and controlled text. As students begin to recognize words and read, they are given these two different types of text. Pattern text consists of the same repetitious series of words with one or two words that change. For example, a simple first text would consist of a series of sentences describing preferences: I like dogs. I like cats. I like pizza. I like apples. I like grapes. I like hats. I like shoes. I like balls. I like dolls. These sentences are placed on individual pages with a supporting image that allows a beginner to use the initial consonant sound and the image to decode the word. As a beginning reader progresses, the text progresses as well so that additional words are added to each sentence.

Controlled text also consists of simple sentences and eases a beginner through a serious of transitions. The text limits the sounds that are used but does not use the exact same pattern as is done in pattern text. Below is a list of the sentences from four books included in a series, Bob Books, published by Scholastic Inc. The first book, Mat, introduces four sounds: m, short a, t, and s. The second book, Sam, introduces two new sounds: c and d. The third book, Dot, introduces four sounds: short o, h, g, and r. The fourth book introduces only one new sound: b.

Mat
Mat.
Mat sat.
Sam.
Sam sat.
Mat sat.
Sam sat.
Mat sat on Sam.
Sam sat on Mat.
Mat sat. Sam sat.
(Maslen, 1976, p.1)

Sam
Sam and Cat.
Mat and Cat.
Sam, Mat, and Cat.
Cat sat on Sam.
Mat sat on Sam.
Sad Sam. Sad Mat.
Sam sat. Mat sat.

O.K., Sam. O.K., Mat. O.K., Cat.
(Maslen, 1976, p.1)

<u>Dot</u>
Dot has a hat.
Dot has a cat.
The cat has a hat.
Dot has a dog. Dog has a hat.
Dog has a rag hat.
Sad dog.
Sad Dot. Sad cat.
Dog has on a rag hat.
(Maslen, 1976, p.1)

<u>Mac</u>
Mac had a bag.
The bag had a dog.
Mac had a bag and a dog.
Mag had a rag.
Mac can tag Mag.
Mac got the rag.
Mac sat on the rag.
(Maslen, 1976, p.1)

These four books allow a beginning reader to complete full books with very limited skills. This provides success and motivates the beginner to continue. Completing a full book would not be possible for a beginner if the text were not carefully designed.

Focusing on Writing Instruction

The Language Arts aspect that mainly relates to programming languages is writing, so further exploration of writing instruction is necessary to allow for its later application to programming instruction. This exploration consists of two parts: the structure of writing lessons and the structure of writing classrooms. Writing lessons fit in two categories consisting of expressional lessons and follow-up/correctional lessons. Writing classrooms also fit into two categories consisting of lab programs and learning centers. The lessons are explored first, followed by a description of the classroom structures that foster these lessons.

Writing Lessons

Expressional lessons and follow-up correctional lessons have different purposes working towards the same goal. These two types of lessons address what is necessary and whether or not it is done correctly. Expressional lessons are first and they introduce content (Petty & Jensen, 1975). These are the lessons that mainly focus on the modeling element of instruction and require the instructor to develop a model for students. Models are particularly important to the beginning writer because it is easier to repeat what has been viewed more easily than it is possible to create something from imagination. It is difficult for learners to rely on their memory and an effective model highlights critical attributes (Hunter 1982). Giving students a model of written expression gives both a mechanical model and a creative model. With time, students will be able to use the model to create their own models prior to beginning a writing exercise.

Once a model has been created for students to emulate, follow-up correctional lessons take place. These are the lessons that allow students to actively participate and practice, and they are important because regardless of whether the goal of the lesson is to address “the what” or “the how” about a particular aspect of language, the students learn better by doing. Students examine the model provided by the teacher, and they begin to complete practice exercises that allow them to hone their skills using the model as a guide.

For these follow-up correctional lessons to possibly maximize student potential, some key elements must be present. Real improvement stems from practical writing experience; therefore, students must understand the purpose of exercises if they are going to help them to progress (Petty & Jensen, 1975). Children need reasons to write and carefully constructed exercises allow them to learn new skills and improve. Furthermore, it is necessary for students to see the logical progression in the exercises to perform well.

Given that students understand the purpose of writing exercises, see a logical progression in those exercises, and are given reasons to write and gain practical experience, the editing process can take place. This is possible because once a student has a model to emulate it becomes necessary to do practice making the model and to make necessary modifications; this self-editing process is learned over time. If students learn to edit their own writing and use models to compare their work to the specific goals outlined by the teacher, this process is accelerated. Students must have models and time for this self-evaluation (Petty & Jensen, 1975).

To help foster this editing process it is necessary to provide many follow-up lessons that allow students to hone their skills editing. The expressional lesson and the students' progress dictate what is going to be necessary in the follow-up lessons and the students edit their work several times during the follow-up. Proof-reading and self-editing allows students to discover errors and empowers them to make positive changes.

Another aspect of the writing process that English teachers must carefully monitor during this editing process is the corrective practice. If students can identify their own errors and make modifications it is possible for them to benefit from well-planned exercises. Students should be taught how to identify when something is incorrect and make a modification that corrects the error. They need to understand the fundamentals of the writing process and the English language to do this effectively. If they do understand it well, proof-reading and self-editing allow students to discover errors and empowers them to make positive changes.

Writing Classrooms

The two categories of writing classrooms that make expressional writing lessons and follow-up correctional writing lessons possible are laboratory programs and learning centers. A laboratory program allows students to work individually while the instructor circulates to assist them with their assignments. Learning centers provide students with a similar type of lesson, but they have great flexibility. This concept of learning centers emerged from British primary schools and it allows students to choose the work they would like to complete. In this model, the learner becomes more active in the learning process (Petty & Jensen, 1975). It is important to note that laboratory programs and centers both require a significant amount of participation from the students. They must have the desire to engage in the work before them and be motivated to complete tasks.

This analysis of Language Arts instruction with the preceding sections about educational psychology learned from theorists and past practices in computer programming education, provide a strong foundation for developing a paradigm of instruction that addresses the needs of beginning programming students. This review of the literature nods at the possibility that a new instructional paradigm could be created, and this paradigm is further supported by the language analysis that follows because it shows that third generation programming languages have the same elements as the English language

COMPARE AND CONTRAST LANGUAGE CONSTRUCTS

RATIONALE

If it is possible to show that spoken languages and computer languages are composed of the same types of sentences and the same parts of speech while expressing ideas and making decisions in the same fashion, it is logical that they could be taught using the same instructional practices. Below is an exploration of the English language and the Java language that shows these mappings not only exist, but are also being used on a regular basis. It is humbling to believe that although computer languages were not specifically developed with the intention to resemble spoken languages that they have arrived at that point. Given that third generation programming languages have evolved to this point, it is time to bring this awareness to the forefront of students' consciousness so they could learn more easily.

TYPES OF SENTENCES

Sentences in the English language consist of the kinds of words that compose them, their functions, their patterns, and the ways those patterns can be used in conjunction with one another. Understanding basic grammar allows people to construct clear sentences that effectively relate ideas (Fowler & Aaron, 1989). The basic syntax of Java statements consist of the kinds of words that compose them, the functions of those words, the patterns on which statements are built, and the ways those patterns can be expanded and elaborated. Understanding basic syntax and statement structure can help create clear statements that effectively relate ideas to computers instead of other people.

There are three types of sentences in the English language – simple, compound, and complex – and we can map them to structures in programming languages. Simple sentences consist of a single main clause. They are simple as long as they contain only one complete main clause and no subordinate clause (Fowler & Aaron, 1989). Below are examples in English and java.

Examples: Bob walked.

bob.walked( )

Bob, buy apples at the store.

bob.buy(apples, store);

The statements written in Java convey the same meaning as the statements written in English, and they can do it with fewer words. The actual parameters in the parameter list correspond to the formal parameters of the method definition and it allows for this more simplistic grammar that eliminates prepositional phrases. Despite this slight difference, statements can be translated; if statements can be translated similar instructional techniques and language acquisition strategies could be used to help beginning students.

Compound sentences consist of two or more main clauses that may be joined in various ways (Fowler & Aaron, 1989). Similarly, a Java statement can become compound with the addition of another simple statement; however, it does not always need the same connecting methods since they would be superfluous. The computer uses other methods of punctuation (brackets, commas, parentheses, and semicolons) as delimiters; yet, the same meaning is conveyed that is conveyed in the English sentence by using different parts of speech and punctuation. Since the position of the actual parameters match the location of the formal parameters of the method definition, the one-to-one mapping allows for simpler statements.

Example: Bob bought apples; consequently, the store increased its revenue.

store.increase(bob.bought (apples));

Complex sentences contain one main clause and at least one subordinate clause (Fowler & Aaron, 1989). The same translations possible for compound sentences are also possible for complex sentences. As demonstrated below, the result of the main Java clause weather.good( ) impacts the result of the dependent clause people.celebrate( ).

Example: The weather was good, so people celebrated.

people.celebrate(weather.good( ))

PARTS OF SPEECH

Sentences could be translated into programming statements, so it follows that the parts of a sentence could be translated into the parts of a programming statement. Nouns are defined as a person, place, or thing and the objects and primitives in Java are also a person, place, or thing. It is clear whether a word in English is a noun because it can be referenced with a specific name: Bob, apple, carrot, love, joy. When we communicate with one another, we use a specific label that allows us to identify the entity we are describing. We can call these items by name and when we try to implement these objects in a program, they are also given a name by which they are referenced. The computer can then identify the items we are referencing, the same way that a human references them. Primitives and objects both behave the same way in this sense.

Objects require more descriptors than primitive data types. If we would like to identify a ball to another person when there is more than one, we will often describe it to the other individual with adjectives and adverbs: Mike, please pass me the big red ball. This is not as efficient as requesting the same ball from a computer system because the computer system has a name for the entity being described and the adjectives and adverbs are not included out of necessity in every sentence. ball1 = new Ball("big", "red") includes the necessary descriptors that would be used as adjectives or adverbs in an English sentence and they allow the same sentence to be stated more efficiently in Java: mike.pass(me, ball1);.

The adjectives and adverbs in the English sentence are what indicate the necessary parameters used in the Java constructor. The adjectives and adverbs in the parameter list have modified the object in the computers memory; therefore, it is not necessary to modify every noun throughout the program with the adjectives and adverbs to clarify which entity is being referenced. The reference to the noun relates it to the necessary adjectives and adverbs that are associated with it because they are stored in memory. For example, if a programmer instantiates a shirt object, the object has instance variables that represent the various attributes. The parameters in the constructor represent these attributes: Shirt(blue, small). Blue and small are adjectives that describe the shirt in the statement Shirt shirt1 = new Shirt(blue, small);

Examples

Dave, please dye that white shirt blue.

Dave.dyeShirt(white, blue);

This example could be extended to show that indirect and direct objects are also included in the parameter list in some cases.

Example:

Dave, please dye that white shirt blue.

Dave.dye(shirt, white, blue.);

Unlike English sentences, adjectives and adverbs could be in different locations in parameter lists with respect to the nouns and verbs that they modify. Adjectives precede nouns in English; yet, the meaning in programming comes from the actual parameters corresponding to the position of the formal parameters that is determined by the method definitions in other files in the system. For beginners, they could learn how to write programs using many features of the language without needing to focus on aspects outside of these simple sentences.

Using the necessary adjectives and adverbs only once simplifies the remaining statements that follow in the program because both parties (i.e. the person and the computer, not two people) are already aware of the details of the noun being discussed and they could obtain them in another fashion. Even if the object changes state, the label attached to it allows it to be discussed in a simpler statement.

Every English sentence also contains a verb and the methods used in programming languages are verbs. Verbs create action in the English language and verbs create action in object-oriented programming languages.

Examples:	Sit Bob.	Stand Bob.	Bob, walk.
	bob.sit( )	bob.stand( )	bob.walk( )

Not only methods behave as verbs, but operators behave as verbs as well. Finding the sum of two integers is a simple process that requires the addition operator to act as the verb and the assignment operator to act as a verb. The simple values six and four take no action when simply stated as nouns. When they are used in conjunction with operators, actions take place that make modifications. i.e. $x = 6 + 4$ causes two actions to occur. First, the sum is calculated, then the value is assigned to a location. The methods associated with objects are more sophisticated verbs that could make other actions take place at the same time.

LOGICAL OPERATORS & CONDITIONAL STATEMENTS

Some of the other aspects of the English language that easily translate into Java are the logical connectors used for conjunctions and disjunctions. In English we use the words “and” and “or” to convey the meaning of the Boolean operators && and || in Java. These logical connectors coupled with conditional statements allow for the construction of more sophisticated sentences and more sophisticated programs.

Syllogism is more easily communicated using the pure mathematical representation of a truth table than describing it in English. Given that "p" is the antecedent and "q" is the consequent, and representing true and false with "T" and "F", the following truth table summarizes the possible outcomes for the combinations of any two entities, and shows that p implies q.

P	q	$p \rightarrow q$
T	T	T
T	F	F
F	T	T
F	F	T

This mathematical principle could be expressed as a formal logic proof as well:

Given: $p \rightarrow q$
p
Therefore: q

This mathematical representation is expressed in Java and in English using the following conditional statements.

```
if( p )
{
    q
}
```

If p is true, than q is true.

With these examples, at this point it is clear that every type of sentence that contains a conditional statement in the English language could be translated into Java.

Based on the author's research, every decision made by the computer with Java is based on this law of syllogism that humans use to reason with English. This is the most primitive decision making mechanism that could be expanded into more complex forms of the basic law, and it shows that the computer is simulating the human thought process. Focusing on this fundamental, this research shows that English could be translated into Java since the derivation of both English and Java are based on human thought and expression. This idea is clearly articulated by George Boole in his book The Laws of Thought (Boole, 1958), in his discussion about how the cause and effect thinking that we apply to the natural sciences can clearly be applied to the science of our intellectual prowess – logic. The relations among objects follow specific rules based on a cause, p, and an effect, q.

George Boole's work illustrates that the seemingly sophisticated thought processes of human beings are nothing more than a composition of this law of syllogism. Just as every mechanical device is composed of simple machines, every electronic device is composed of the same electronic components, and every geometric proof is composed of accepted undefined terms and postulates, every thought resulting in a conclusion is a composition of the law of syllogism. The control structures that we see in programming languages may be different in notation or in some other superficial way; however, they all express the law of syllogism or a composition of the law of syllogism in some way. Focusing on selection, this means that every computer program could be completed using only if statements. It would be cumbersome, but it is possible.

We use English to express actions that will occur in a sequence just as a computer follows Java in a specific sequence. A sequence is one of the seven control structures used by Java. In English, clauses are stated based on a condition, as is the case with an if statement in a Java program. People could take one action or a different action based on the outcome of a condition as is the case with an if/else statement in Java. The third and final way that statements are made to describe behavior is to select one of many different options based on a specific condition, as is the case with a switch statement.

Below are examples that demonstrate the simplest control structure, sequence, and the three control structures that compose the various types of selection statements in Java. As is the case when translating two spoken languages, there are other ways to translate statements. These translations simply support the argument that third generation programming languages have a grammar and syntax sufficiently similar to spoken languages to borrow pedagogical practices and learning principles.

Examples:

Tom purhases apples.
Bob purchases apples.

```
tom.purchaseApples( )
bob.purchaseApples( )
```

Tom purchases apples if Bob purchases apples.

```
if (bob.purchaseApples( ) )
        tom.purchaseApples( )
```

Tom will purchase apples if Bob purchases apples, or he will purchase oranges.

```
if (bob.purhaseApples( )
        tom.purchaseApples( )
else
        tom.purchaseOranges( )
```

Tom wakes up if it is 7:00 am, he eats at 11:00 am, he is home at 5:00 pm, or else he is working.

```
switch(time)
{
        case 7:     tom.wake( );
                    break;

        case 11:    tom.eat( );
                    break;

        case 5:     tom.go(home);
                    break;

        default:    tom.work( );
                    break;
}
```

REPETITION STATEMENTS

If repetition statements are added to this analysis, a simple investigation shows two things:

1. Only a while statement would be necessary to write every computer program.
2. A slight variation to selection statements would be to repeat a selection statement.

First, an exploration of why while statements are the only repetition statement necessary leads to an understanding that although Java has do/while statements and for statements, they are nothing more than variations of the while statement. The while statement is a pre-test looping mechanism, the do/while statement is a post-test looping mechanism, and the for statement is a fixed repetition looping mechanism, so the do/while statements and for statements could be translated into while statements as seen below.

```
Tom jogs for ten minutes

int minute = 1;
while(minute <= 10)
{
        tom.jog( );
        minute++;
}

for(int minute = 1; minute <= 10; minute++)
{
        tom.jog( );
}

int minute = 1;

do
{
        tom.jog( );
}while(minute <= 10);
```

Other languages may have different control structures for repetition, but they are simply variations on these pre-test, post-test, and fixed repetition control structures.

Exploration of the second issue shows that a repetition statement is nothing more than an if statement checking a single condition multiple times. As already seen, there are two variations to a while control structure in Java, for and do/while; yet, they only vary in the syntactical differences and timing of the conditional checking. Hence, it is only necessary to show that the while statement is an expression of multiple if statements.

Demonstrating that the while statement is a composition of if statements is even more important than the repetition control structure translations. It is the same if statement executed multiple times with the antecedent slightly changing. As demonstrated below, while loops could be translated into if statements.

```
int x = 1;
while(x <= 3)
{
        System.out.println(x + “ “ );
        x++;
}
```

The above statement is logically equivalent to the statements below.

```
int x = 1;
if(x <= 3)
{
        System.out.println(x + “ “ );
        x++;
}

if(x <= 3)
{
        System.out.println(x + “ “ );
        x++;
}

if(x <= 3)
{
        System.out.println(x + “ “ );
        x++;
}
```

```
if(x <= 3)
{
        System.out.println(x + " " );
        x++;
}
```

Given these translations, it is evident that all decision making performed by the computer is a simple implementation of the law of syllogism; therefore, every English sentence could be translated into Java even if it contains conditional statements. No matter how sophisticated one may like to believe his thought process may be, it is never more complex than this basic truth that was uncovered by logicians many years ago. It is unfortunate that George Boole was unable to have such a powerful computing tool when doing his work.

A PARIDIGM BASED ON BEST PRACTICES

SUMMARY & PLAN FOR JUSTIFICATION

The previous section of this document contained a language analysis that shows that there are many similarities between the English language and the Java language that could be mapped; encouraging a possible relationship between teaching strategies. It also contains a review of the literature that provides insight into the instruction provided in the English Language Arts classroom, an outline of various methods of computer programming instruction with commentary about the pros and cons of these pedagogical practices, and a description of many principles of educational psychology that have been described by various educational theorists.

The goal of this work is to state the possibility of exploring a new instructional paradigm that introduces programming to beginning programmers using the pedagogical practices of Language Arts teachers before introducing them to lower levels of abstraction and machine level details. This would allow students to have the literacy skills required to communicate with the machine so that they could explore more sophisticated computer concepts.

This section of the document provides a rationale for implementing this instructional paradigm based on the three aforementioned areas:

1. It suggests how this paradigm could be implemented by explicitly stating how best practices used in English Language Arts classrooms could be applied in a Java classroom.
2. It suggests how this paradigm could be implemented while still using the best practices in computer programming that were previously explored.
3. It articulates how this paradigm for instruction is based on the principals of educational psychology that are encouraged by many educational theorists.

Throughout this rationale references will be made to the attached demonstrations and problem sets that have been designed in the spirit of this instructional paradigm, which could be used as a model for future work in this area.

APPLYING BEST PRACTICES OF ENGLISH LANGUAGE ARTS

The English Language Arts instruction previously outlined focuses on two different types of language classrooms, laboratory programs and learning centers, and two different types of lessons, expressional and follow-up correctional, that foster the best practices of English Language Arts instruction. Using the stated types of lessons in the stated types of classrooms allows for many best practices to be implement:

- Errors are frequently corrected
- Accuracy is given priority over meaning
- Input is simplified and sequenced
- Concepts are presented and practiced in isolation
- Work progresses from simple to complex
- Students are given items to explore
- Traditional teaching methods that include elements of effective instruction are used
- Practice is crucial
- There is a logical progression
- Proof reading and self-editing fosters better writing
- Corrective practice is developed
- Pattern text and controlled text are employed to assist students

English Language Arts Classrooms

The attached demonstrations and problem sets have been used by the researcher in a laboratory setting. They could be used in learning centers as well, but given the amount of observation that must take place with beginning students, the laboratory setting seems most appropriate. Using a laboratory setting allows the instructor to use four of the Language Arts best practices: providing practice for linguistic items that are introduced and practiced in isolation, using controlled text, using pattern text, and instructing students with the elements of effective instruction.

The attached problem sets have modeled isolating concepts based on the types of statements that must be written. The first pattern established in the attached problems is how to construct an object. Constructing an object is very simple when that is the only focus. Students work on constructing tangible objects with a clearly defined state and behavior that makes the use of parameters meaningful:

```
FramedRect mrRectangle;
mrRectangle = new FramedRect(10, 30, 50, 100, canvas);
```

The students learn the grammar of the language and construct objects that allow them to check their progress immediately. The students learn that nouns are classified, and constructing a noun could require a statement that includes adjectives and adverbs.

The second set of problems allows the students to write simple sentences that use both subjects and predicates. By focusing on only accessors and modifiers in isolation, students transfer what they already know about objects having state, and they learn how to extract this information in a programming environment. The problem set begins with exercises about accessors and then adds modifiers. Students continue to use the constructors and add new skills to their developing set of skills that allow them to see how nouns and verbs create actions and changes.

It is important to note that these exercises focus on programming literacy by isolating types of statements, but at the same time students are learning other important skills. This is not an attempt to eliminate underlying computing concepts and the ancillary skills required to become a good programmer; instead, this approach should foster those skills. Allowing students to do these repetitious exercises allows for the development of the most fundamental programming skills while establishing a strong awareness of the issues addressed when developing classes. By the end of these three simple units, students already know about constructors, accessors, and modifiers; how to read an API; how to use a simple IDE; how to instantiate objects; how to access information about objects; how to change the state of an object; and how objects interact. All of this is done quickly and easily while building students' confidence.

These problems use both pattern text and controlled text with significant blocks of time for students to play. Isolating the topics based on language structure allows for this use of pattern text and controlled text so that students can learn the Java language and master it in small subsets. For example, the first problem set ends with students drawing a house using the shapes that they know how to construct. By using open-ended questions like this, students can use the very few language elements that they know to create programs that are only limited by their imagination. By allowing for this exploration in a setting where students could work/play collaboratively, they could make progress that motivates them to learn the next topic of study. The attached problem sets are an example of how to progress in a logical manner with much repetition that allows the student to become an expert of a subset of language structures before continuing.

This type of classroom setting requires the use of all of the elements of effective instruction with a focus on clearly stated objectives and practice. Since this laboratory setting allows students much time to work, the instructor has a significant responsibility to observe students and guide their thinking when necessary. In a properly constructed lab, the teacher should be able to monitor all of the students' work from a single vantage point and intervene when the desired outcome is not present. The teacher could model how to construct objects, access information from objects, or modify an object for students while explicitly stating the pattern that should have been identified in the document that they use as a model.

English Language Arts Lessons

The type of classroom structure chosen by an instructor and the classroom atmosphere that is created are very important to fostering student development, but instruction allows for the implementation of even best practices. English Language Arts lessons were previously categorized as expressional and follow-up/correctional and although each type of lesson could foster all best practices, they have strengths that foster some aspects of best practices better than others:

Expressional Lessons foster three best practices well

- There is a logical progression
- Input is simplified and sequenced
- Work progresses from simple to complex

Follow-up Correctional foster six best practices well

- Practice is crucial
- Errors are frequently corrected
- Students are given items to explore
- Accuracy is given priority over meaning

- Proof reading and self-editing fosters better writing
- Corrective practice is developed

Expressional Lessons

The demonstrations in Java classrooms are the fundamental component of the expressional lessons. This is the input that the instructor models for students. Preparation of demonstrations requires the programming instructor to identify a logical progression, and sequence and simplify the input so that work may progress from simple to complex. Providing work that progresses from simple to complex is important and Applin states in her work that the guidance of this sequence has been neglected.

> "There is guidance on the national level from the Association for Computing Machinery (ACM) and the Institute of Electrical and Electronic Engineers (IEEE) in the form of curriculum recommendations for both high school and for college. These guidelines are very broad giving topics to be covered throughout a four year program but not really suggesting when, where, or how to cover them." (Applin, 1999, p.5)

It is unfortunate that such research about alternative sequences of computer programming topics is limited, especially since nearly a decade has passed since Applin stated that empirical studies in the area of computer science education are lacking and very few have studied teaching methodology at the entry level. Applin's comment suggests that the research that is currently lacking in the discipline is what basis to use to progress from simplicity to complexity. Given the similarities with English, it is important to try teaching Java based on sequences used in English language arts:

- This proposed solution focuses on a logical progression for programming that like

English Language Arts places the acquisition phase, reading, before the production phase, writing, and provides exposure to demonstrations and APIs that foster this development.

- This proposed solution focus on the development of simple statements and language constructs that progress to more complex statements and language constructs.

This approach is possible because the acquisition phase is developed with the use of good demonstrations that allow students to explore the language and test conjectures about how it works. "Little use has been made of pre-written or partly-written software (Robertson & Lee, 1995, p.11)." By making use of pre-written examples, students can learn more about the structure of the language and become empowered to develop more code on their own.

This approach is possible because the progression of simple statements to complex statements is done by allowing students to learn fundamental programming language concepts by using objects without needing to explore the classes from which they are instantiated. The "objects first" approach that has been promoted by many educators has actually been a "classes first" approach. Allowing students to write sentences about objects without understanding the underlying issues associated with the development of a class, allows for the use of the methods of teaching English. Otherwise, a top-down approach, logic, and problem solving are the vehicles accessible to teach programming. Teaching classes before objects is like teaching your children that they cannot play with a ball until they first know how the ball is constructed. A young child can learn about the state and behavior of a ball simply by touching it, throwing it, bouncing it, rolling it, and kicking it. To describe how it rolls and why it bounces would lead to a discussion about physics and mathematics that would make young children never play with a ball. It makes sense that asking programming students to create classes before using an object leads to such discouragement and much of the attrition seen in computer science and information technology. Eliminating the need to develop classes by providing students with many developed classes, allows the focus of instruction to be on writing the statements and using the language constructs.

It is unfortunate that constraints made it impossible to do a formal exploratory research study based on the sequence of the attached materials, but anecdotal evidence encourages such work. The most encouraging anecdotal evidence is the fact that the instructor developing these materials spent much less time addressing programming syntax problems and debugging issues than discussing creative designs and problem solving issues.

Follow-up Correctional Lessons

Follow-up correctional lessons allow for the implementation of 6 more best practices: practice is crucial, errors are frequently corrected, students are given items to explore, accuracy is given priority over meaning, proof reading and self-editing fosters better writing, and corrective practice is developed. Follow-up correctional lessons are about providing students time to practice, and it is clear that these six best practices are related based on this premise. These lessons are constructed to exploit that relationship because they allow students to have the time to do corrective practice that is characterized by proof reading and self-editing that leads to frequent error correction. To make this possible the lessons require items for the students to explore and stress the importance of accuracy over meaning.

In this case the objects that students are given to explore and write about are shapes they are already familiar with, and writing about them allows students to develop accuracy even though the final product does not have the same meaning as an enterprise commerce web site or desktop application. The objects are familiar to all of the students and yet simple enough to manipulate and write about with simple statements. In preschool and kindergarten children learn to manipulate shapes and use colors very early, so finding a way to apply this to programming is necessary. Given that these objects are now available for students to explore, they are used in the accompanying examples.

The corrective practice in English Language Arts classes that is characterized by proof reading and self-editing that leads to frequent error correction is implemented in this paradigm by providing students with some problems containing enough complexity that it requires students to rewrite and recompile their code on a regular basis to determine if they obtain the desired outcome. The problems are designed so that students could write very few lines of code, compile, and check their progress with tangible output. They could then go on to another phase, compile, and check their progress with respect to the provided demonstration. Forcing more frequent correction of errors is helpful to a learner. This fosters good corrective practices, and this principle that is so important to language arts instruction is applicable to Java instruction. It is not only applicable to Java instruction, but also even easier to foster for Java instructors given the technology.

APPLYING BEST PRACTICES OF COMPUTER PROGRAMMING

The literature review revealed many important aspects about best practices in computer programming instruction and explored other instructional paradigms that use many of these best practices. This instructional paradigm was carefully designed so that all of the best practices are used while leveraging the tools and techniques developed by others, and so that it could be used in conjunction with other instructional paradigms that have impacted students in a positive manner. The best practices that were identified have been summarized into 6 different elements:

- An appropriate instructional environment must be used

- The environment used must be simplistic
- The ease of the environment should allow for initial work to be done at higher levels of abstraction
- Initial programs should be created with little effort by non-specialists
- The problems assigned should allow students to remain in the problem domain
- The use of a library allows for rich examples, and allows students to focus on a precise part of a program. The library used should allow students to feel like they can create a full program, and it should have the potential of being a useful tool throughout the course.

Following is a description about the tools and techniques that have been created by other instructors that allow for this particular paradigm of instruction so that best practices may be followed. This section finishes with statements about how this paradigm could be used in conjunction with other established paradigms, such as a case study approach, with the use of the tools and techniques described.

Appropriate Environment

This instructional paradigm is possible for several reasons. One important reason why this is possible is because of a simple IDE, BlueJ, which was created with beginning programmers in mind. It fosters the first three best practices of computer programming because it is an appropriate instructional environment, it is simplistic, and it allows for initial work to be done at higher levels of abstraction. In particular, it allows the focus to be on constructing clear statements as would be done in an English language Arts class. A further description of this IDE is helpful, but first a look at the challenges faced by beginning programmers and their instructors when limited to professional IDEs.

Teachers experience challenges when instructing beginning programming students how to program (Kolling, Quig, Patterson, & Rosenberg, 2003). These challenges are in many ways due to ancillary issues that must be addressed before teaching the students how to communicate with the machine so that it can be used to solve a problem. For example, to write their first program students must be familiar with general computer skills like opening software and creating files, they must be able to use text editors, and they will be using a compiler in some manner. Other shortcomings of traditional systems include environments that are not object-oriented, complex, and that focus on user interfaces (Kolling et.al, 2003). Beginning students may not need an integrated development environment (IDE) that includes many tools and features that are not only of no interest to beginning programmers, but also are not necessary to begin to learn the language – which is the intended goal.

The situation faced by many beginning programmers was well summarized by James Gosling (Barnes & Kolling, 2003) when he summarized his daughter's struggle in an introductory Java course that was using a commercial IDE. The complexity of learning was significantly compounded due to the introduction of a sophisticated tool that did not address the needs of beginning students. It was full of features helpful to the advanced programmer, but the features were unnecessary to his daughter and her classmates. Mr. Gosling himself stated that BlueJ is a perfect fit for a situation like this (2003).

When beginning students attempt to focus on learning a third generation programming language like Java, they are stymied by the quantity and complexity of the ancillary issues that must be addressed. As stated by the authors of BlueJ, Barnes & Kolling (2003),

> The minimal Java program to create and call an object typically includes:

■writing a class;

■writing a main method, including concepts such as a static methods, parameters, and arrays in the signature;

■a statement to create the object (‘new’);

■an assignment to a variable;

■the variable declaration, including variable type;

■a method call, using dot notation;

■possibly a parameter list.

As a result, textbooks typically either

■have to work their way through this forbidding list, and only reach objects somewhere around Chapter 4; or use a ‘Hello, world’-style program with a single static main method as the first example, thus not creating any objects at all.

This does not mean that there is no place for IDEs and sophisticated programming tools, only that they are not necessary for beginning programmers. Even though adults teaching students how to read might hope that they one day read Shakespeare, they do not introduce his work first. It follows that students learning how to program should not be exposed to the sophisticated tools and concepts of the discipline without being eased into it. Students learning to read begin with Dick and Jane type material so they build a strong foundation that later allows them to read classic literature. Using the de facto tools and techniques of the programming community will be within the reach of all students who are provided a means to acquire language in a systematic fashion. First, it is necessary to obtain a tool that allows instructors and students to focus on programming semantics and concepts without the medium being burdensome, and this is the reason why BlueJ has been chosen for this work. Since BlueJ assists with the move away from the problem space and into the solution space, it makes sense that beginning students would be able to focus more on human-computer interaction and communication without worrying about low-level details. BlueJ supports this attempt to teach programming grammar and composition first, with a keen focus on object-oriented concepts.

Objectdraw Library

The accompanying materials begin with the objectdraw library because it fosters three more best practices. The best practices fostered with this library include allowing for initial programs to be created with little effort by non-specialists, assigning problems that allow students to remain in the problem domain, using a library that allows for rich examples, and allowing students to focus on a precise part of a program. This library allows students to feel like they can create a full program, and it has the potential of being a useful tool throughout the course.

The instructional paradigm being proposed requires the use of very specific objects with distinct state and behavior so that beginning students can write about these objects. This library assists in presenting material at students' instructional level because the library provides students concrete objects to manipulate. Students can create complete programs with pictures. Students are familiar with pictures so using this library keeps them in the problem domain very well, and even though they are non-specialists with respect to the programming environment, all students are specialists with shapes. The graphics library allows instructors to focus on a small set of objects that can be used in the creation of pattern text and controlled text. Students will get instant feedback about the result of their programs and be able to respond to the results provided on the screen quickly and easily. Errors are visually displayed so students can easily understand the machines response to their instructions. The authors of objectdraw chose to use graphics because they are malleable objects that provide feedback rapidly. Young students understand the objects well and get instant feedback about their programs because they can see on the screen if the various shapes are behaving as anticipated. These objects are already engrained in their schema and they could manipulate their state using useful methods that go beyond the basic accessors and modifiers that most beginning students learn (Bruce, Danyluk, & Murtagh, 2006).

The use of a simplified programming environment with a library makes this programming paradigm possible. This could then segue into using a case study or other best practice. This does not mean that this instructional paradigm will not stand on its own, only that it extends other best practices.

BEST PRACTICES ADHERING TO THE PRINCIPLES OF EDUCATION PSYCHOLOGY

The portion of the literature review that described the principals of education psychology outlined 10 principles that are important, and they are all apparent in this work:

- Students must be provided an appropriate environment with child-sized furnishings
- Students are motivated when asked to do purposeful work, when given immediate feedback, and when given positive feedback
- Students must be instructed in their zone of proximal development (ZPD)
- Students learn with their instructors and peers near
- Students must be given the tools and information that allows them to make connections
- Students develop meaning constructively
- Students learn from repetition without penalty
- Students obtain the necessary assistance when the instructor observes the learner
- Students should be provided with large blocks of unstructured time to play

- Students learn in a linear process

Each of these principles is applied in this instructional paradigm, and a description of how this is done follows.

Students must be provided an appropriate environment with child-sized furnishings

This first principle is based on the work of Maria Montessori. Fortunately, David J. Barnes and Michael Kolling have already created that programming environment – BlueJ. Many instructors can provide the physical nurturing environment for learning, but the computer environment for instructing beginners was unavailable until the authors of BlueJ shared their product. BlueJ provides the furnishings for beginners without overwhelming them with unnecessary features and options that may lead to confusion and cause individuals to explore something that is unnecessary. Providing a different IDE would be like leaving sharp objects around for the exploration of a young child.

Students are Motivated When Asked to do Purposeful Work and are Given Immediate Positive Feedback

Students must be engaged in meaningful work that gives them immediate feedback and positive feedback. This is also present in this paradigm of instruction with the assistance of BlueJ and objectdraw. BlueJ allows for students to begin writing code with minimal preparation and objectdraw allows for immediate positive feedback. The graphics library assists with this paradigm of instruction because students can write a single statement and get feedback. Only one line of code must be learned by the students to be able to see if they properly constructed an object. The visual display provided by objectdraw makes this possible. So if the work is properly sequenced based on the types of statements written, students will continue to obtain immediate, positive feedback.

Students must be instructed in their zone of proximal development (ZPD) & Students learn with their instructors and peers near: Students are instructed within their zone of proximal development because of the important sequence of topics. The instructor must sequence the topics from simple to complex based on the types of statements students must write. By focusing on language aspects, it is possible to limit the new content that students see in each exercise, and this keeps them in their ZPD. Other aspects of Lev Vygotsky's work are implemented because this method of instruction allows students to work with their peers and their instructor close by to help them when necessary. By placing demonstrations and problem sets on a server or web site, students could work at their own pace and get assistance from the others around them. This process provides the instructor the necessary time that is necessary to help students in class.

Students must be given the tools and information that allows them to make connections
The software tools, demonstrations, and problem sets provided allow students to make connections because everybody is familiar with parts of speech, so basing programming instruction on this prior knowledge assures that the instructor is using examples that are in the students' schema. The objectdraw library assists by making it possible to draw on the prior knowledge of every student in the room, and the assigned work progresses in a fashion that allows students to continually apply the learning of the previous lesson.

Students develop meaning constructively

Providing students with a minimal set of objects and assigning a sequence of problems that requires a minimal set of statements allows students to explore the programming environment and make conclusions about the work they are doing. This stands in sharp contrast to the objectivist model where students listen to instruction. Students are given much time to explore with this methodology.

Students learn from repetition without penalty

The problem sets provided repeat similar statements multiple times so that students can practice the skills that they are learning. For example, using the same objects makes it possible for students to become familiar with the constructors, accessors, and modifiers of each object. Students have the ability to continually repeat each process by making slight variations to the statements that they must write. They could compile and debug and get feedback that does not penalize them if they are incorrect.

Students Obtain the Necessary Assistance When the Instructor Observes the Learner

The instructor can observe the actions of the learner in this environment for two reasons: graphical displays are easily created with the use of objectdraw which allows the instructor to observe from a distance, and in the researcher's experience less time is required to debug programs because students are comfortable with the syntax of the language. In a properly arranged computer lab, an instructor could simultaneously observe the actions of every student in the room. The beauty of using BlueJ and objectdraw in conjunction with one another is that an environment has been created that Montessori and Piaget could have only imagined. Every student in the room has his own environment to manipulate and change. Each student works at an appropriate level and a teacher good at diagnosing the needs of students will observe the changes. Focusing on the pedagogical practices of English Language Arts instruction with these tools makes it very apparent when students are not doing something correctly because the instructor is well aware of the anticipated output and can identify students who are not writing statements correctly.

Students Should be Provided with Large Blocks of Unstructured Time to Play

Including open-ended exercises similar to the included exercise about building a house invites exploration and discussion that allows for the play that was so important to Maria Montessori and Jean Piaget. Students have the time to explore the API and learn by trial and error when this time is provided. Given that students will be working with a very small subset of the language structures, they will be able to explore without being frustrated by syntax errors.

Students Learn in a Linear Process

Looking at the work of Montessori through the lens of Piaget, there is a need for students to perform tasks in a specific sequence. To fulfill this principle, the sequence chosen for this work is based on language structure and is guided with the development of carefully designed problem sets and demonstrations. Students should complete the assigned tasks in sequence. More specifically, the attached samples begin by constructing objects, than accessing information from them, and then modifying them. This requires very few types of statements to be written.

Given that these principles of education psychology are present in this instructional paradigm, best practices of education psychology, best practices of computer programming instruction, and best practices of English Language Arts instruction are all at work together in an effort to more gently ease programming students into the discipline. Given that this paradigm is theoretically solid and has been successful in the researcher's experience, it may be important to consider this method of instruction in more introductory level programming courses.

Considerations for Success

The above section provides a logical argument that leads to two reasonable questions:

- If there is validity to a computing program that is based on literacy strategies and language arts, what similar prior research has been done to support this paradigm of instruction?
- If this paradigm of instruction could be helpful to the development of beginning programmers, what challenges must be overcome to implement this paradigm of instruction?

Research that Supports this Paradigm of Instruction

Research about the correlation between computer programming instruction and English language arts instruction is difficult to find; yet, there is research that actually explicitly states that the lack of this research is disappointing: “Unfortunately programming has traditionally been taught with little reference to natural language pedagogy. In particular it has concentrated on the writing of code rather than the reading of existing code.” (qtd. in Robertson 11) Furthermore, this approach is supported by the work done by Robertson and Lee (1995); they investigated the relationships between natural language theories and programming languages and recognized the disappointing situation that exists in the fact that programming languages are developed with little attention to spoken languages.

Despite the lack of research and assertions made by other researchers that this work is lacking, the similar research that is available – particularly the work of Buckland and Applin – supports this paradigm of instruction. Anne Gates Applin used language acquisition studies as the theoretical basis for drawing parallels between language classrooms and computer science classrooms. Believing that reading well-written programs would lead to writing well-written programs, she was inspired to research whether template programs can be crafted so that they provide a glimpse of things to come and actually assist in student understanding when those concepts are covered in the classroom. Her results indicated that when students went on to their second level programming course, the students who were provided templates wrote better code. The code they wrote was more modular, used more effective parameter passing, and was better documented (Applin, 2001).

The work of Richard Buckland indicates that this paradigm of instruction would not only foster students by using the pedagogical practices and language similarities of the English classroom, but would also affect the atmosphere as a whole and making the learning process more enjoyable. Mr. Buckland shares his anecdotal experience in his article, "Can we improve teaching in Computer Science by looking at how English is taught?" He concludes that students enjoy the humanities much more than math and science courses because of the fundamental differences in the ways that they are taught to students. He concludes that this is due to several factors:

1. The students have greater control.
2. More informal discussions take place
3. Humanities are taught in a much more active way
4. Student's opinions are valued more highly.
5. The teacher acts as a facilitator instead of as an expert

Richard Buckland was able to create this atmosphere in his computer programming course by creating tutorials that are more student-centered and less teacher-centered. By creating these tutorials the class was more active, more enjoyable, and allowed students to obtain a deeper understanding of what they were studying. He had students work together and included activities that got people to speak during the first 10-15 minutes to set a pattern of behavior for the course (Buckland, 1996).

None of these stated outcomes that resulted from research are counter-intuitive; yet, no significant changes have been made to programming instruction. This leads to the importance of exploring what challenges are present that contribute to preventing the implementation of a new paradigm of instruction, and what must be done to overcome such challenges.

Required Needs to Implement this Paradigm of Instruction

Hypothesizing about the challenges present when implementing new instructional paradigms may lead to many different possibilities, but there are significant impediments that may need to be addressed. The difficulties of implementing a new instructional paradigm may be in large part due to the acceptance of the current ubiquitous pedagogical practices, the time constraints of the instructor, or due to the fact that most people teach the way they were taught.

Instructors could trace their own intellectual path for their students because it is natural, yet it could hinder the progress of their students. Culwin (qtd. in Lister 147) stated this at the turn of the century, but it remains difficult for instructors to change.

> The people who decide upon the nature, content and focus of undergraduate curriculum are, in general, of an age where their own professional and intellectual development mirrors the development of computing over the last twenty years or so. Hence their personal perception of how they learned ... [is generalized] ... to a conception that knowledge of the old paradigm is a necessary pre-requisite for learning the new paradigm…

The ubiquity of current teaching methods may also be difficult to change because of the amount of time necessary to engage in the development of a new instructional paradigm. Instructors have many responsibilities and obligations to fulfill that consume much time to maintain current levels of productivity; the sentiments of some computer science instructors were captured in Teaching Tips We Wish They'd Told Us Before We Started (Astrachan, Parlante, Garcia, & Reges, 2007, p.2).

> When we started teaching, our more seasoned colleagues were probably ready with pearls of wisdom to share with us. They no doubt pointed us to several of the excellent resources on teaching as a new faculty member. As an instructor, there were *so* many hats to wear: lecturer, teaching staff mentor, exam / project / lab author, grader and leader of office hours. It was a lot to take in, and even *with* all that counsel, it was probably still quite daunting!

Regardless of the difficulties involved with introducing a new instructional paradigm, the attached samples will provide an example for instruction that could be expanded in a way that improves computer programming instruction for future students. Java could be taught in a way that is similar to the way that English is taught, since programming languages have evolved to the point where this is now possible. This has not been emphasized in the past based on this research; and even though it is difficult to change something as ubiquitous as the instructional practices of our institutions, there are several reasons to make the attempt.

CONCLUSION

The preceding content identifies relationships between English and Java that encourage the possibility of teaching programming languages like natural languages. Similarities between the two languages prompt an analysis of the best practices of teaching English Language Arts and computer programming, with an explanation of how these best practices could be maintained in this paradigm of instruction. Ideas about how this method of instruction adheres to fundamental principles of learning identified by educational theorists are also provided.

Demonstrations and problem sets based on these ideas have been used in the classroom, and more students have been retained with this method instruction than others used by the researcher. This anecdote and the application of best practices encourage the need to further explore this method of instruction via a formal exploratory research study.

Demonstrations

```
/**
 * Template to be used for the development of other programs
 *
 * @author V. Falbo
 * @version 1.0
 */

import java.awt.*;
import java.applet.Applet;

import objectdraw.WindowController;
import objectdraw.Line;
import objectdraw.Location;
import objectdraw.*;

public class Template extends WindowController
{
        // declarations

        public void begin( )
        {
                // instantiation

        }

}
```

```
/*
        Constructs a single line.  Other objects are instantiated in a similar fashion.
*/

import java.awt.*;
import java.applet.Applet;

import objectdraw.WindowController;
import objectdraw.Line;
import objectdraw.Location;
import objectdraw.*;

public class Line1 extends FrameWindowController
{
        Line horizontalLine;

        public void begin()
        {
                horizontalLine = new Line(10, 10, 50, 50, canvas);
        }
}
```

```
/**
 * Rectangle Accessors
 * Uses basic accessors of a rectangle and displays the attributes in text boxes.
 * Displays the height and the width.
 *
 * @author V. Falbo
 * @version 1.0
 */

import objectdraw.WindowController;
import objectdraw.FramedRect;
import objectdraw.Location;
import objectdraw.Text;
import objectdraw.*;

public class RecAccess extends FrameWindowController
{
   // instance variables

   FramedRect BigRect;             // Rectangle to provide attributes for display
   Location RectLocation;          // Location where the rectangle will be placed

   Text WidthTextBox;              // Text Box for displaying the width of the Rectangle
   Location WidthTextLocation;     // Location where the width text box will be displayed

   Text HeightTextBox;             // Text Box for displaying the height of the Rectangle
   Location HeightTextLocation;    // Location where the text box will be displayed

   /**
    * Called by the browser or applet viewer to inform this Applet that it
    * has been loaded into the system. It is always called before the first
    * time that the start method is called.  For standard Java applets the
    * init() method would be called.  Since this applet uses objectdraw,
    * the begin() method is used instead and this method calls init().
    */

   public void begin()
   {
      double rectangleWidth;              // Stores the width of the rectangle
      double rectangleHeight;             // Stores the height of the rectangle

      RectLocation = new Location(100, 100);
      WidthTextLocation = new Location (380, 10);
      HeightTextLocation = new Location (380, 30);

      BigRect = new FramedRect(RectLocation, 200, 300, canvas);
```

```
        rectangleWidth = BigRect.getWidth();
        rectangleHeight = BigRect.getHeight();

        WidthTextBox = new Text(rectangleWidth, WidthTextLocation, canvas);
        HeightTextBox = new Text(rectangleHeight, HeightTextLocation, canvas);
    }
}
```

```
/**
 * RecMod1
 * Uses basic modifiers of to display a rectangle and its position.
 * The position of the rectangle and the display will change as the mouse
 * is clicked.
 *
 * @author V. Falbo
 * @version 1.0
 */

import objectdraw.WindowController;
import objectdraw.Location;
import objectdraw.FilledRect;
import objectdraw.Text;
import objectdraw.*;

public class RecMod1 extends FrameWindowController
{
   // instance variables

   Text LabelTextBox;           // Text Box for displaying the label of the text box
   Text PositionTextBox;       // Text Box for displaying the position of the Rectangle

   Location PositionTextBoxLocation;   // Position of the rectangle text box display
   Location LabelTextBoxLocation;       // Position of the label text box display

   FilledRect MovingRectangle;         // Rectangle that moves based on mouse events
   Location MovingRectangleLocation;   // The value of the current rectangle position

   /**
    * Constructs the objects that must be displayed
    */

   public void begin()
   {
      LabelTextBoxLocation = new Location (300, 50);
      PositionTextBoxLocation = new Location (320, 80);

      MovingRectangle = new FilledRect(50, 50, 10, 10, canvas);
      MovingRectangleLocation = MovingRectangle.getLocation();

      LabelTextBox = new Text("Current Rectangle Location:", LabelTextBoxLocation, canvas);
      PositionTextBox = new Text(MovingRectangleLocation, PositionTextBoxLocation,
                                           canvas);
   }
   /**
    * Moves the rectangle up the window and updates the display of its Location
    */

   public void onMouseClick(Location point)
   {
      MovingRectangle.move(10, 10);
      MovingRectangleLocation = MovingRectangle.getLocation();
      PositionTextBox.setText(MovingRectangleLocation);
   }
}
```

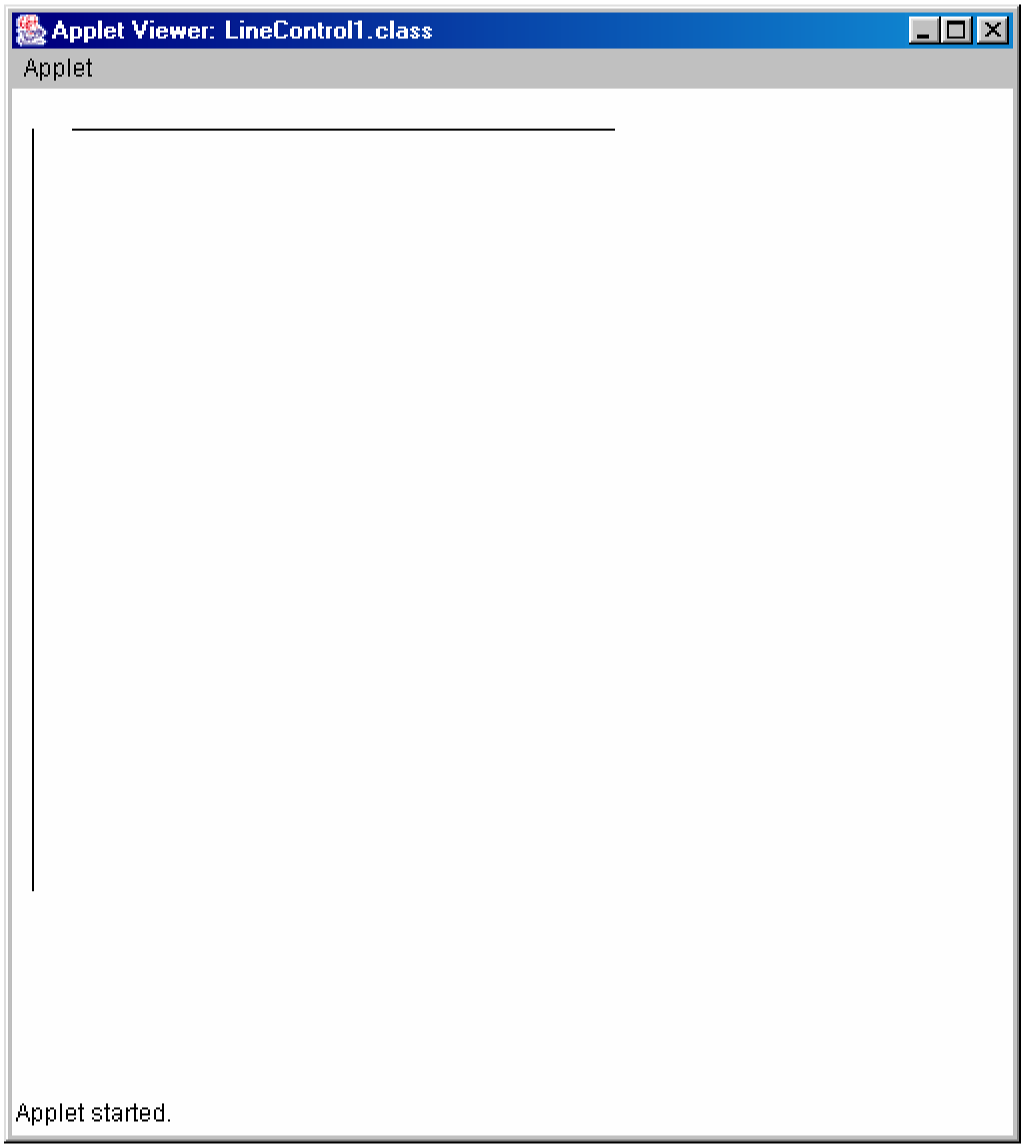
Problem Sets

Unit 1: Objects & Constructors

Exercises 1: Constructing Lines

1. Line Controller 1 – This line controller will be used to construct two lines. The first line will be a vertical line defined by ordered pairs (10, 20) and (10, 400). The second line will be a horizontal line defined ordered pairs (30, 20) and (300, 20).

   The applet size is 500 x 500



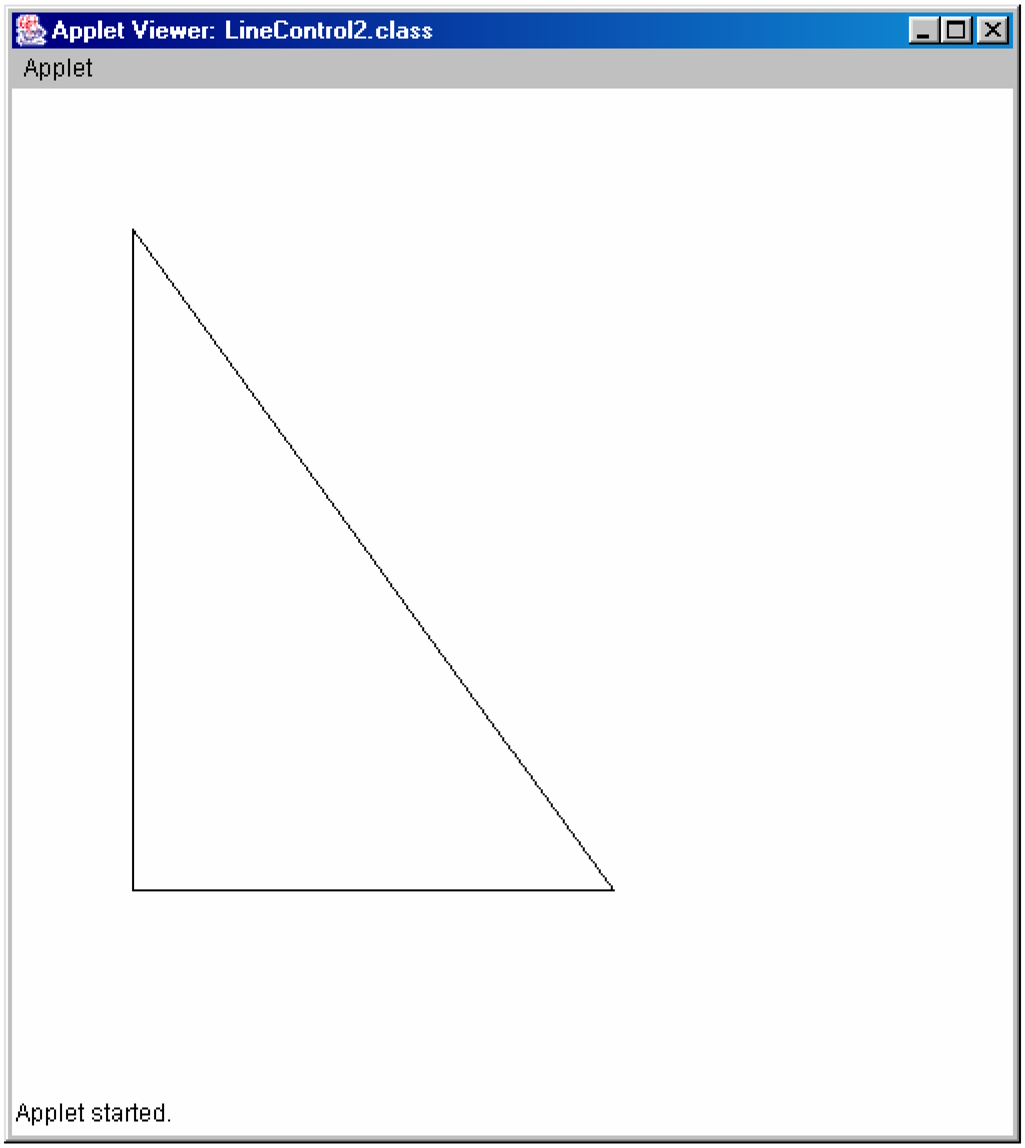
Unit 1: Objects & Constructors

Exercises 1: Constructing a Triangle with Lines

2. Line Controller 2 – This line controller will be used to construct a triangle. The endpoints of the line segments forming the three sides are below.

   Leg1: (60, 70) and (60, 400)
   Leg2: (60, 400) and (300, 400)
   Hypotenuse: (60, 70) and (300, 400)

   The applet size is 500 x 500

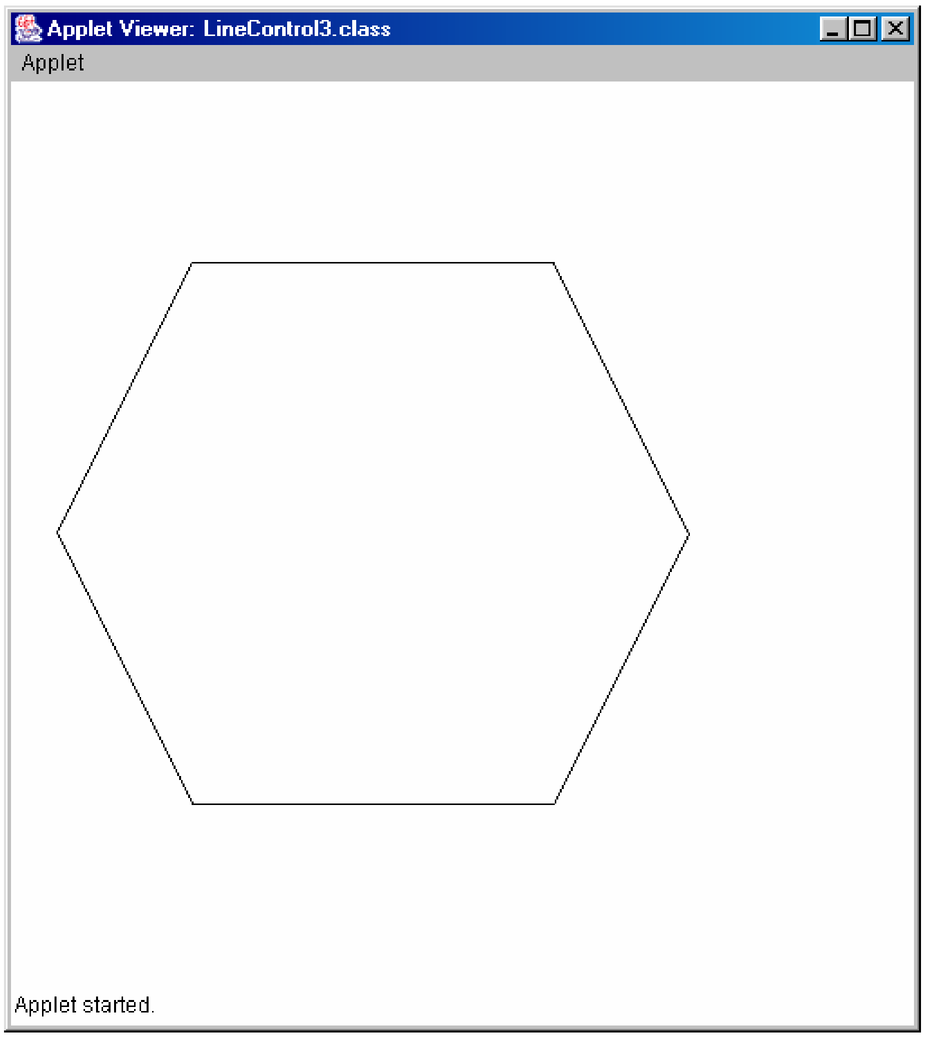


Unit 1: Objects & Constructors

Exercises 1: Constructing a Hexagon with Lines

3. Line Controller 3 – This line controller will be used to construct a hexagon. Location objects will determine the endpoints of the line segments. The coordinates of the Location objects that should be used are listed below.
   Location A: (100, 100)
   Location B: (25, 250)
   Location C: (100, 400)
   Location D: (300, 400)
   Location E: (375, 250)
   Location F: (300, 100)

   Use the locations above to construct the line objects that are required to construct the hexagon: SideAB, SideBC, SideCD, SideDE, SideEF, and SideFA.
   The applet size is 500 x 500

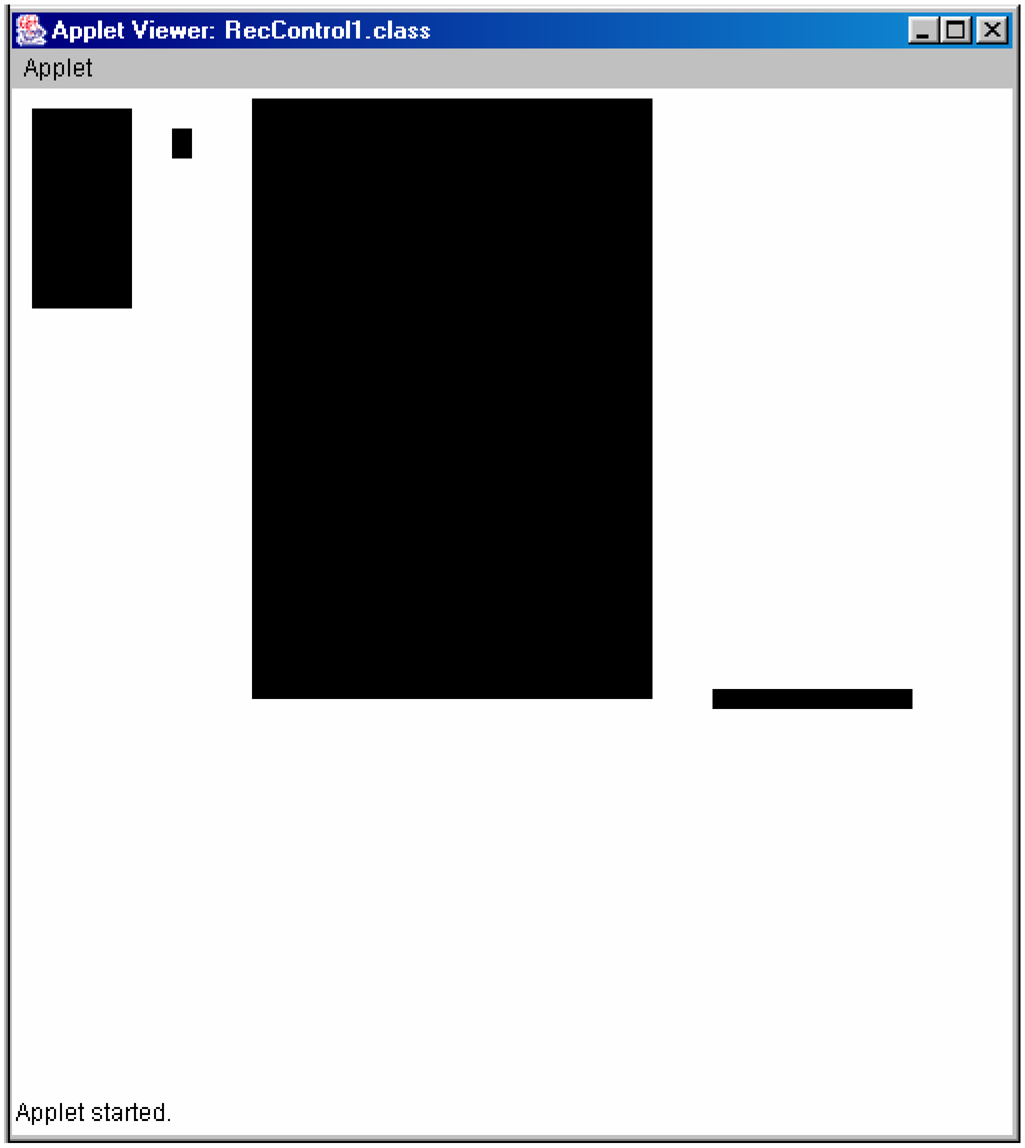


Unit 1: Objects & Constructors

Exercises 2: Constructing Rectangles

1. Rectangle Controller 1 – This rectangle controller will be used to construct four filled rectangles: SimpleRect, LittleRect, BigRect, and WideRect. Each rectangle will be constructed with the position and dimensions below.

   SimpleRect: position (10, 10), dimensions 50 x 100
   LittleRect: position (80, 20), dimensions 10 x 15
   BigRect: position (120, 5), dimensions 200 x 300
   WideRect: position (350, 300), dimensions 100 x 10



Unit 1: Objects & Constructors

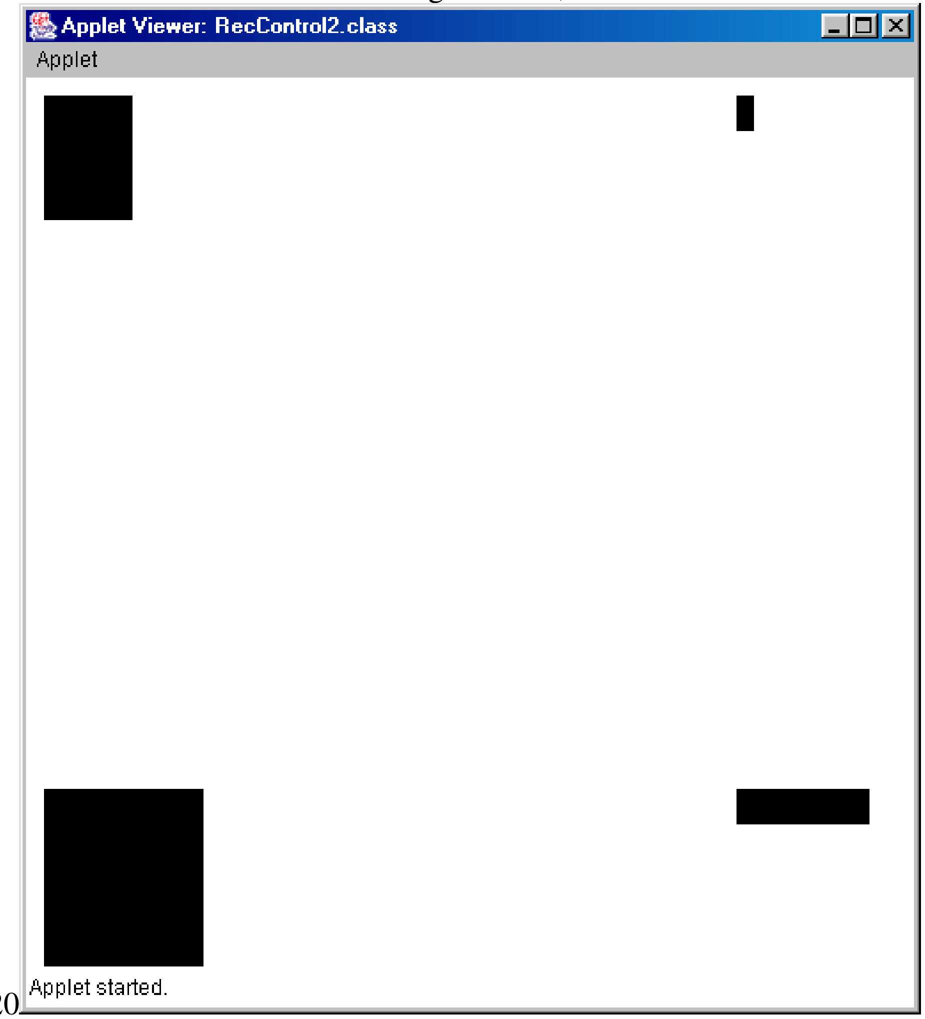
Exercises 2: Constructing Rectangles

1. Rectangle Controller 2 – This rectangle controller will be used to construct four filled rectangles in four different locations. Location objects will be used to determine where the rectangles are placed. The locations and their coordinates are listed below.

   TopLeftCorner: (10, 10)
   TopRightCorner: (400, 10)
   BottomLeftCorner: (10, 400)
   BottomRightCorner: (400,400)

   The four rectangles will be placed in the four locations with the given dimensions.

   SimpleRect: Location TopLeftCorner, dimensions 50 x 70
   LittleRect: Location TopRightCorner, dimensions 10 x 20
   BigRect: Location BottomLeftCorner, dimensions 90 x 100
   WideRect: Location BottomRightCorner, 75 x



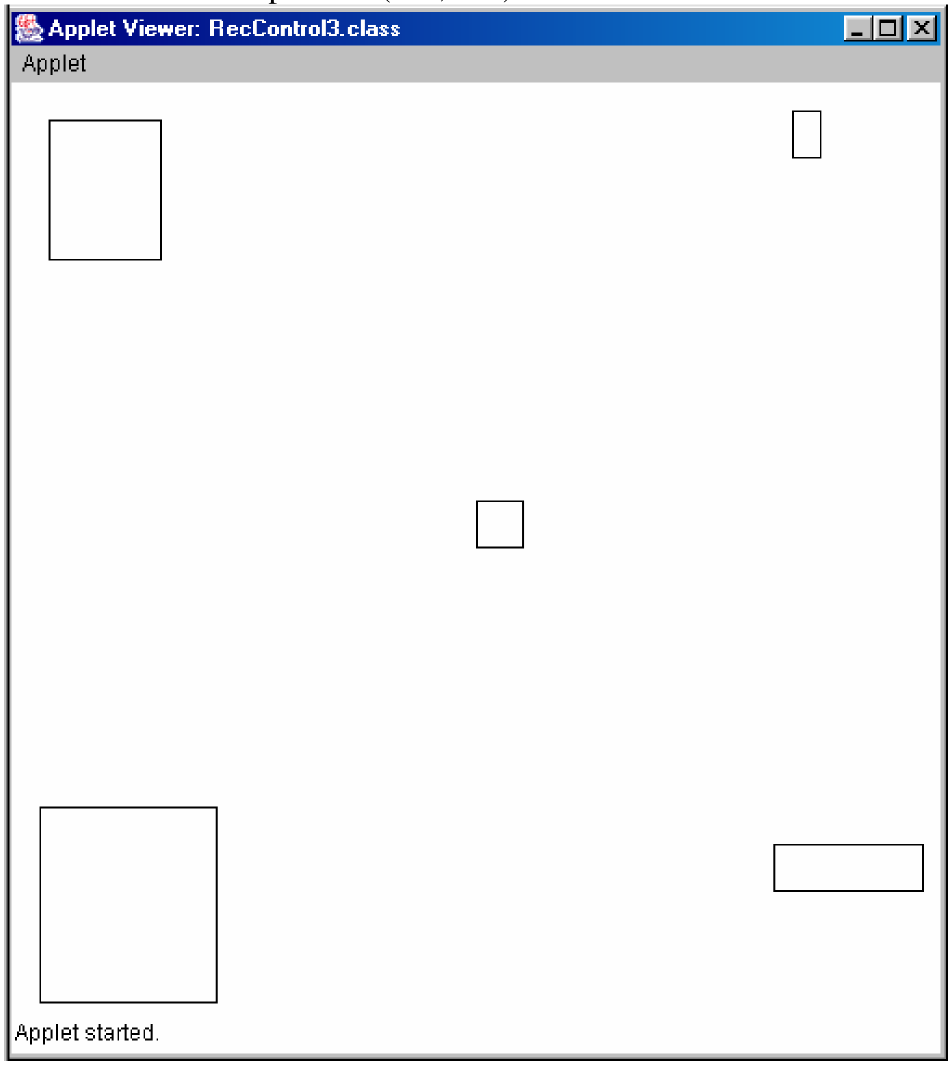
20

Unit 1: Objects & Constructors

Exercises 2: Constructing Rectangles

2. Rectangle Controller 3 – This rectangle controller will be used to construct five framed rectangles. Location objects will determine where four of the rectangles are placed while the middle rectangle will specify its position explicitly. The coordinates of the Location objects that should be used are listed below.
   TopLeftCorner: (20, 20)
   TopRightCorner: (420, 15)
   BottomLeftCorner: (15, 390)
   BottomRightCorner: (410,410)

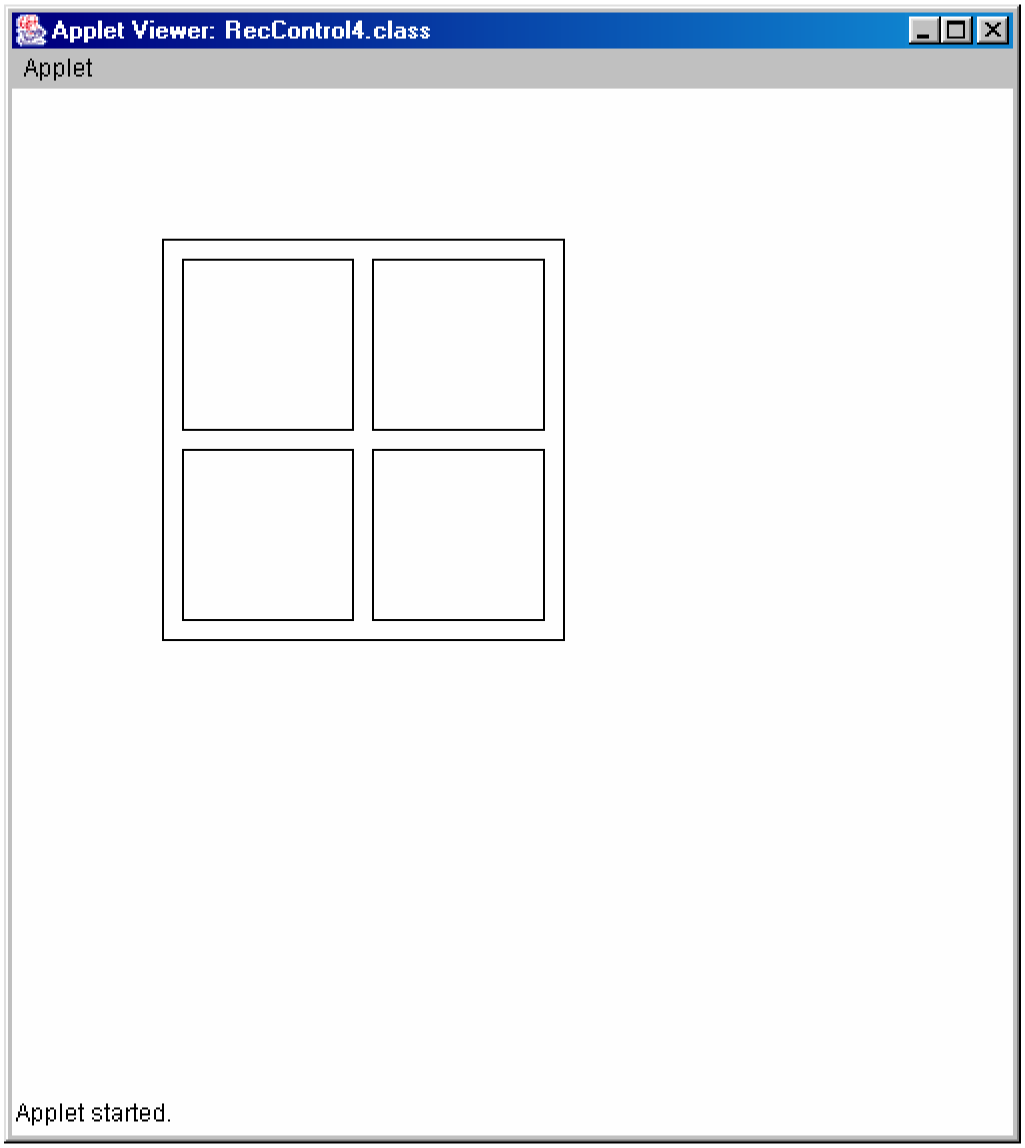
   SimpleRect: Location TopLeftCorner, dimensions 60 x 75
   LittleRect: Location TopRightCorner, dimensions 15 x 25
   BigRect: Location BottomLeftCorner, dimensions 95 x 105
   WideRect: Location BottomRightCorner, dimensions 80 x 25
   MiddleRect: position (250, 225) dimensions 25 x 25



Unit 1: Objects & Constructors

Exercises 2: Constructing Rectangles

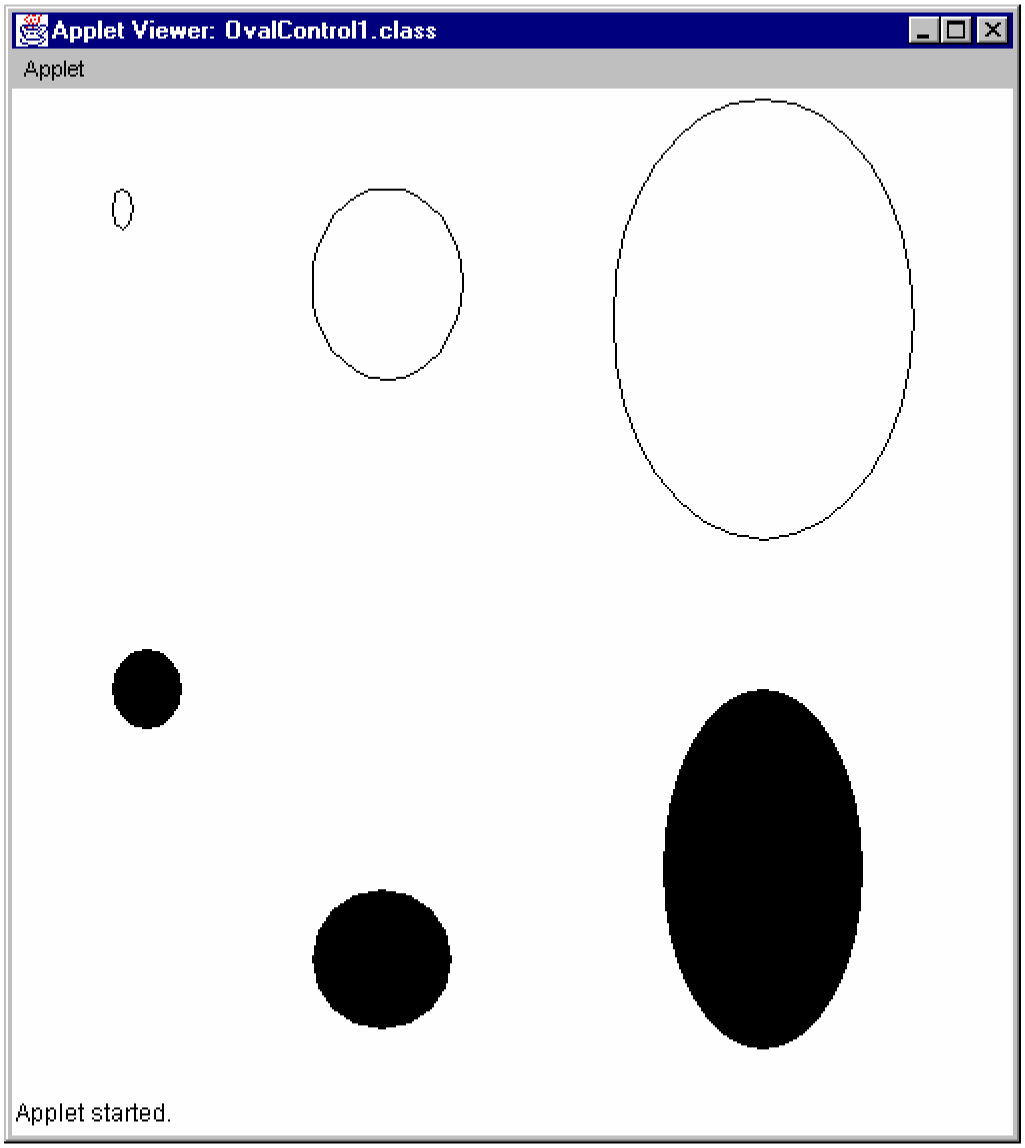
3. Rectangle Controller 4 – This rectangle controller will be used to construct five framed rectangles that together will form a window. Choose any constructors you like to create the picture below.



Unit 1: Objects & Constructors

Exercises 3: Constructing Ovals

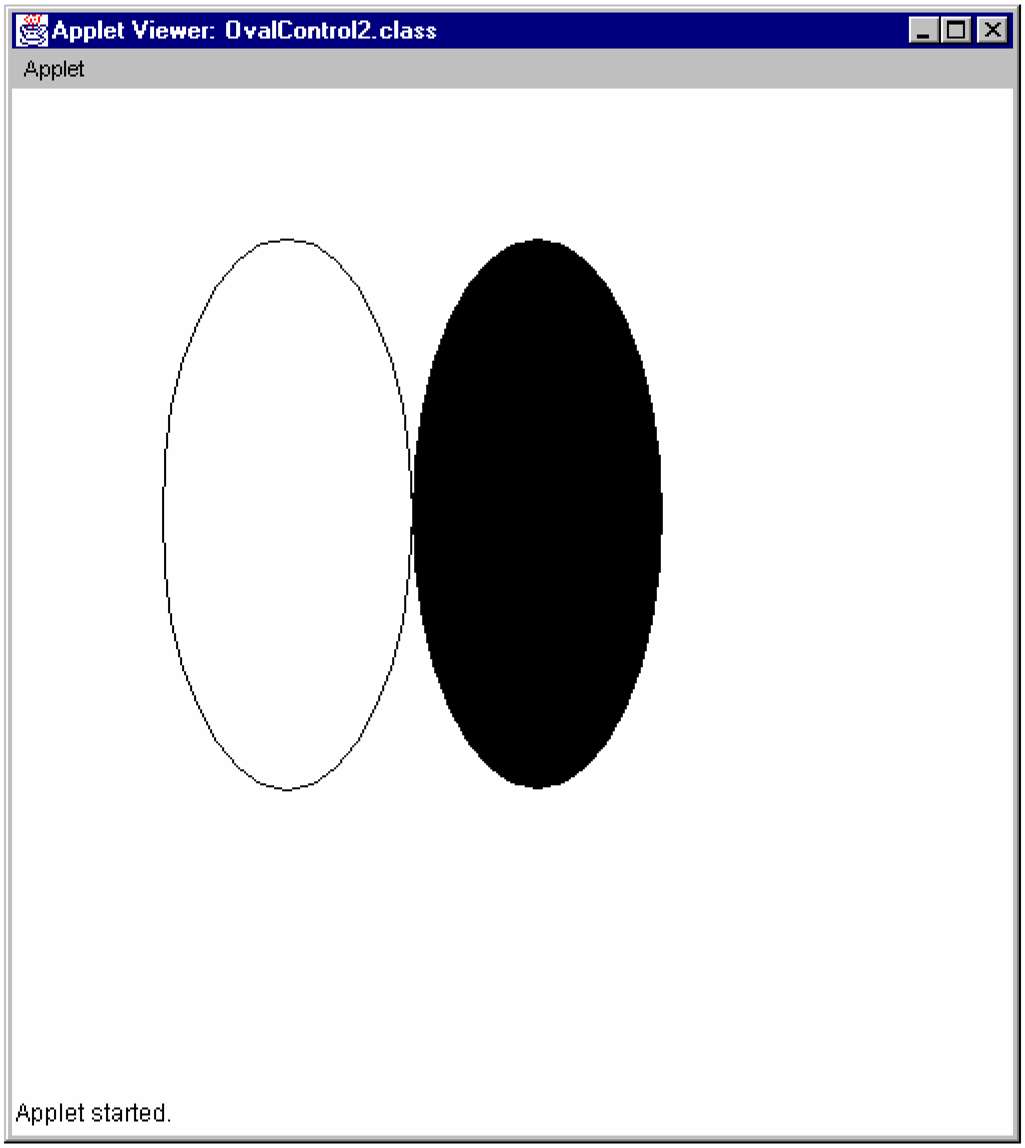
1. Oval Controller 1 – This oval controller will be used to construct six ovals that represent the use of each constructor in the object draw API so be sure to refer to the API. Output does not have to be exactly like that shown below; however, none of your ovals should intersect.



Unit 1: Objects & Constructors

Exercises 3: Constructing Ovals

2. Oval Controller 2 – This oval controller will be used to construct two ovals that are touching. Determine which constructors you would like to use but be certain one oval is filled and one is framed.



Unit 1: Objects & Constructors

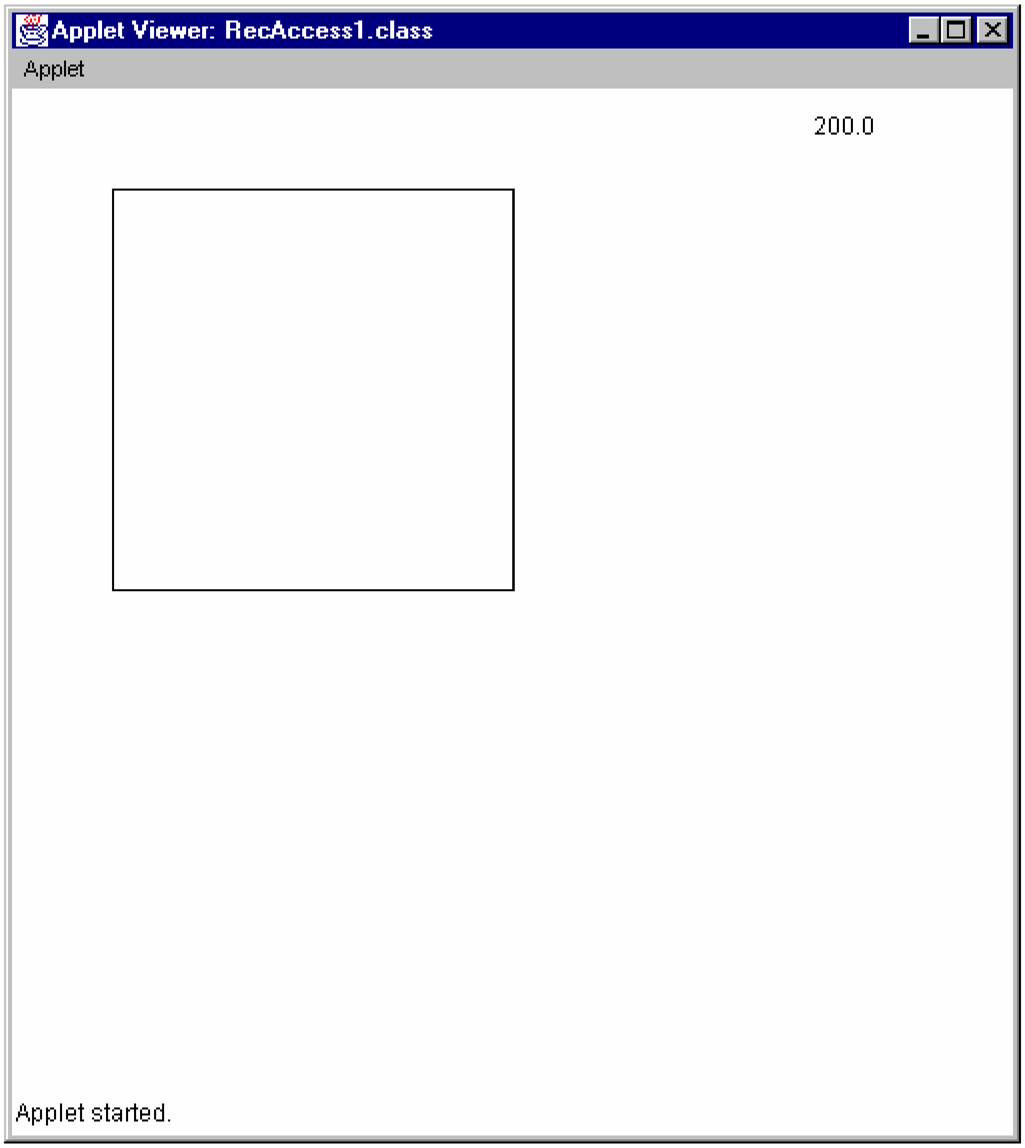
Exercises 4: Constructing a House

2. Home Builder – Create a new project (House) and use everything you have learned so far to construct a house. You can try to use new objects and methods if you would like to experiment; however, be sure to save your document before using unfamiliar code and do this without any assistance.

Unit 2: Accessors & Modifiers

Exercises 1: Accessing Rectangle Attributes

1. Rectangle Accessor 1 – This rectangle accessor will be used to determine the width of a rectangle. Construct a text box to display the value.

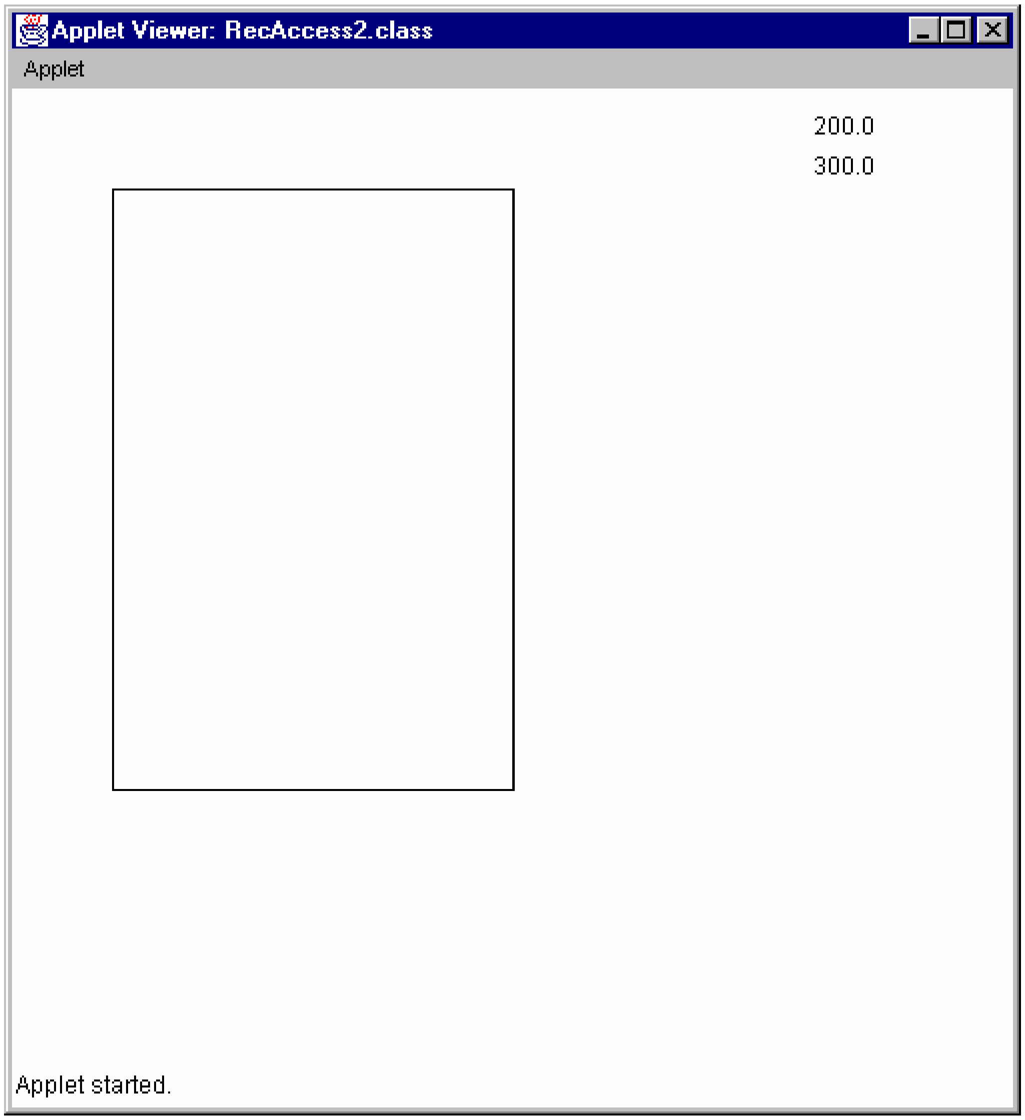


```
200.0
```

Unit 2: Accessors & Modifiers

Exercises 1: Accessing Rectangle Dimensions

2. Rectangle Accessor 2 – This rectangle accessor will be used to determine the height and width of a rectangle. Construct text boxes to display the values.

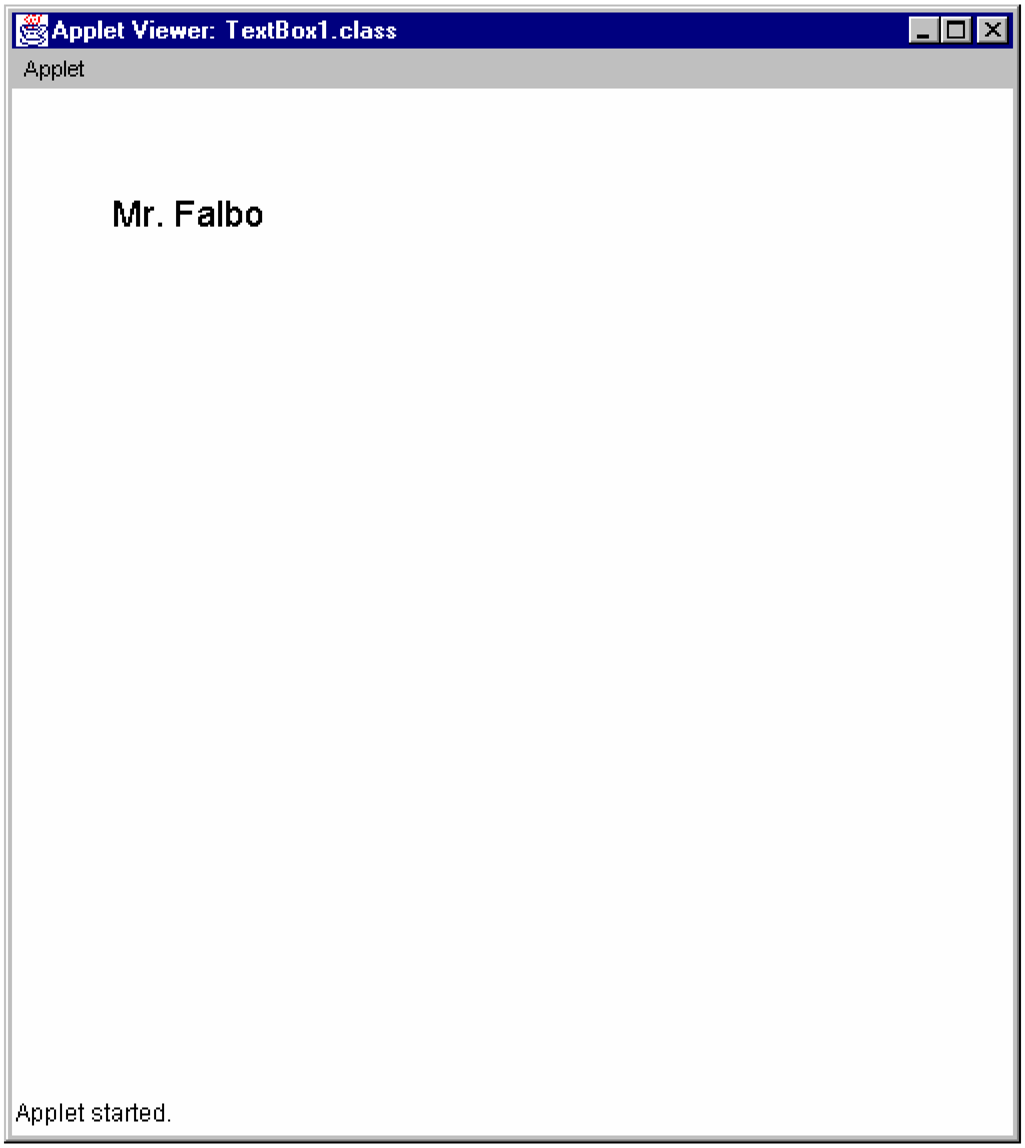


```
200.0
300.0
```

Unit 2: Accessors & Modifiers

Exercises 2: Accessing and Modifying Text Box Attributes

1. Text Box 1 – Construct a text box, and then use the font modifier to change the font to size 18. Print your name in this font.



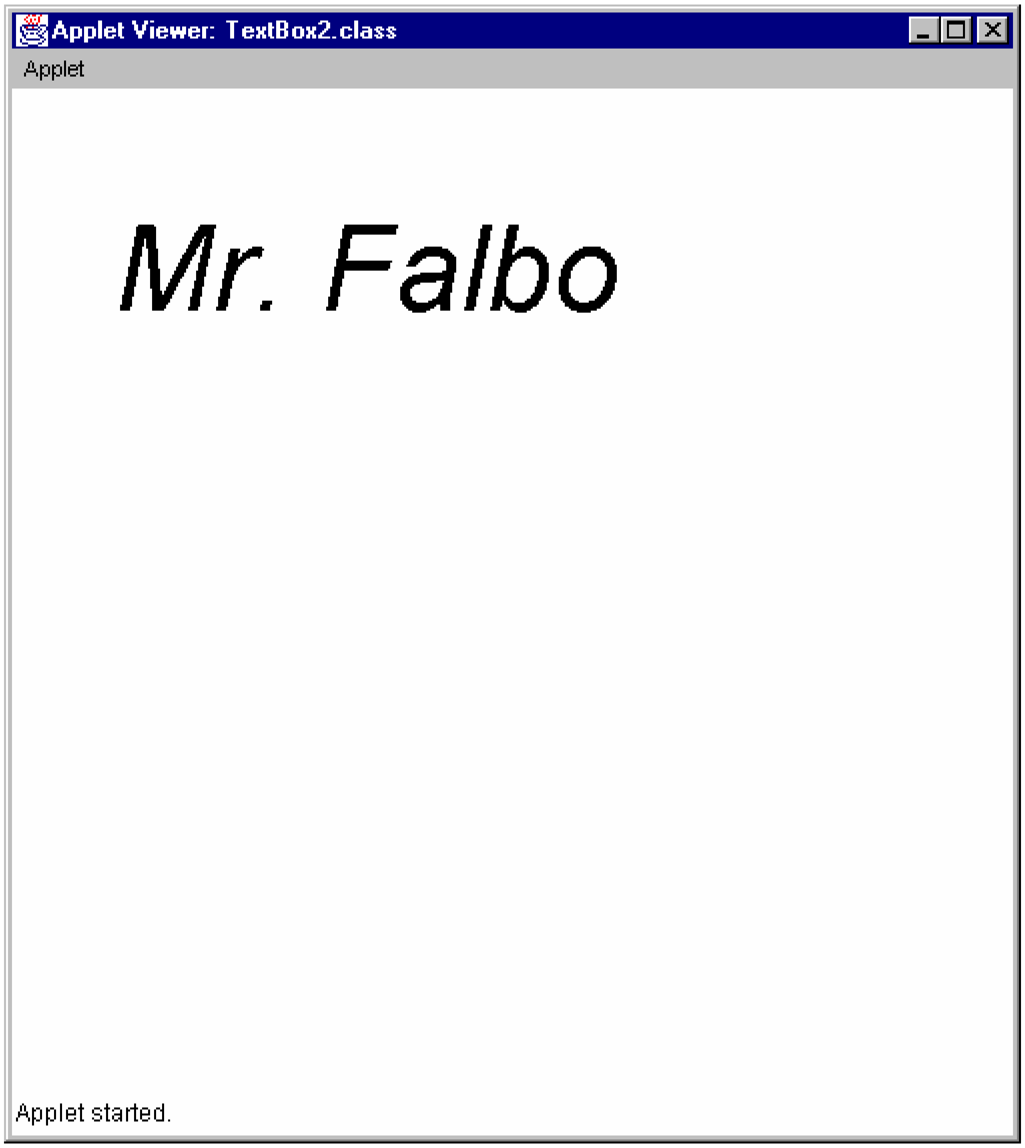
```
Mr. Falbo
```

```
Applet started.
```

Unit 2: Accessors & Modifiers

Exercises 2: Accessing and Modifying Text Box Attributes

2. Text Box 2 – Use the correct modifiers to print your name in a size 60 font that is bold and italic. Use the API to identify the correct modifiers.



Mr. Falbo

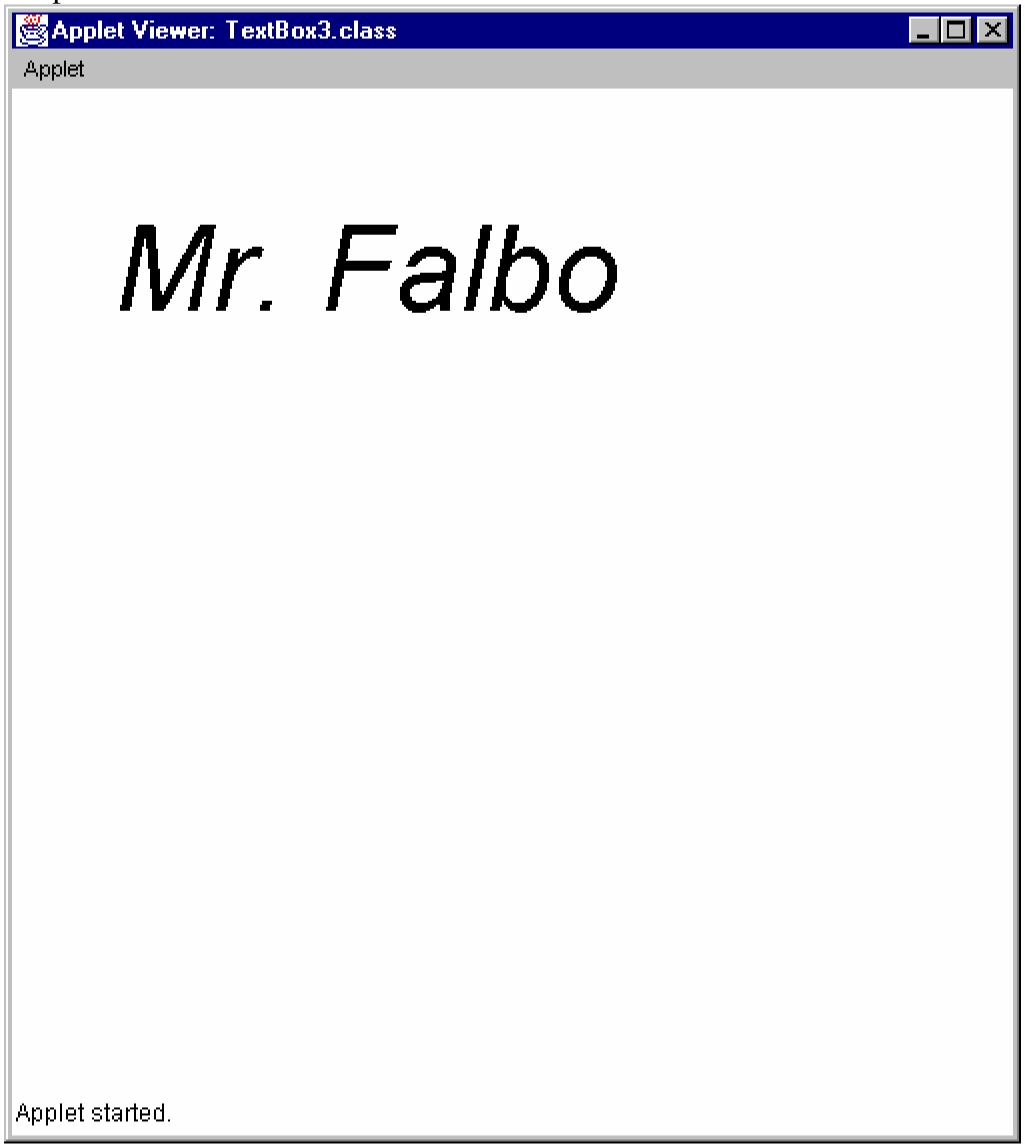
Applet started.

Unit 2: Accessors & Modifiers

Exercises 2: Accessing and Modifying Text Box Attributes

3. Text Box 3 – Display your name with a size 12 font and when the mouse is clicked change it so your name is printed in a size 60 font that is bold and italic.

Output after the mouse event has been handled.



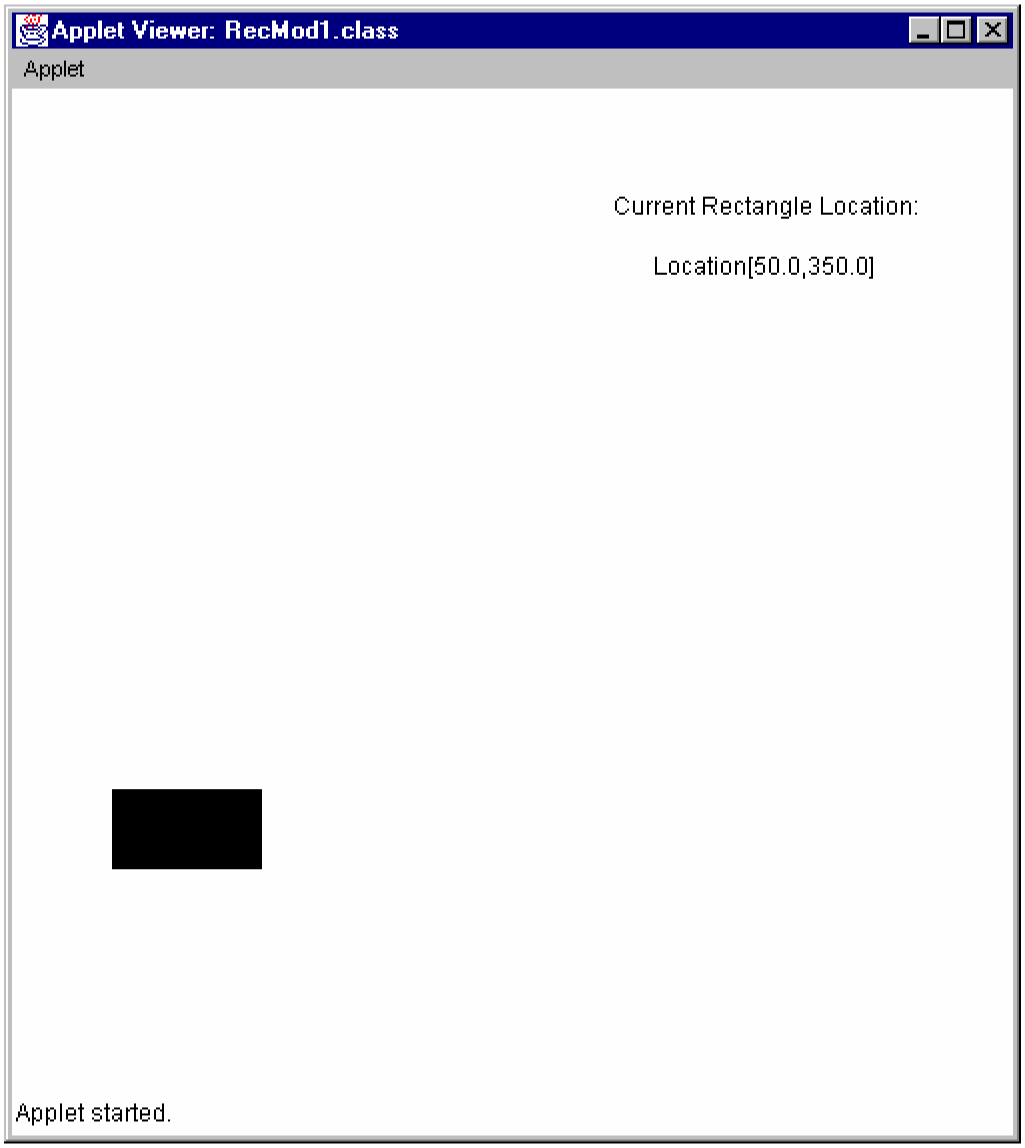
Mr. Falbo

Applet started.

Unit 2: Accessors & Modifiers

Exercises 3: Rectangle Modifiers

1. Rectangle Modifier 1 – Place one rectangle in the window and display its current location. As the mouse is clicked in the window move the rectangle up 50 pixels and update the display of its location.



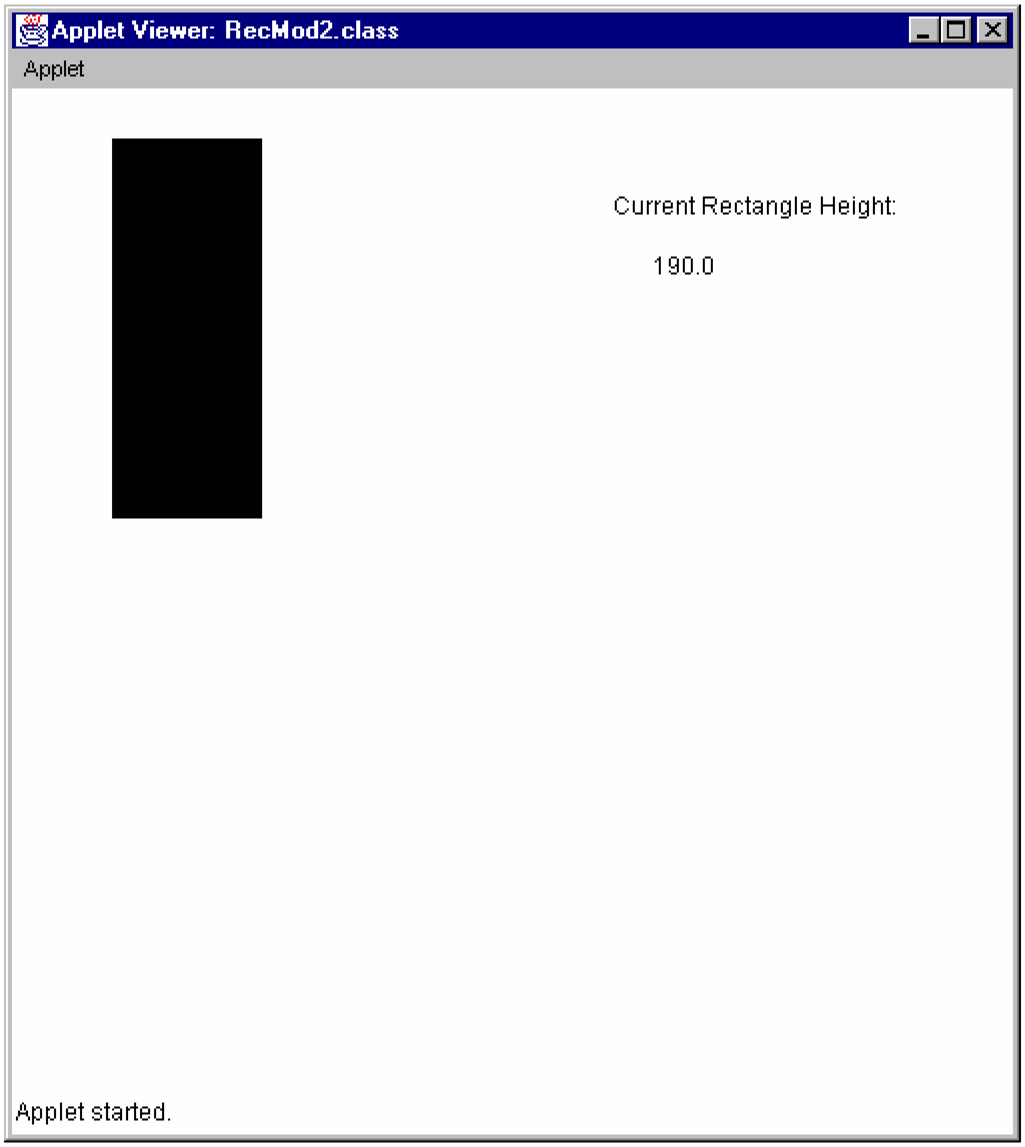
```
Current Rectangle Location:

Location[50.0,350.0]
```

Unit 2: Accessors & Modifiers

Exercises 3: Rectangle Modifiers

2. Rectangle Modifier 2 – Place one rectangle in the window and display its current height. As the mouse is clicked in the window increase the height of the rectangle by 50 pixels and update the display of its height.

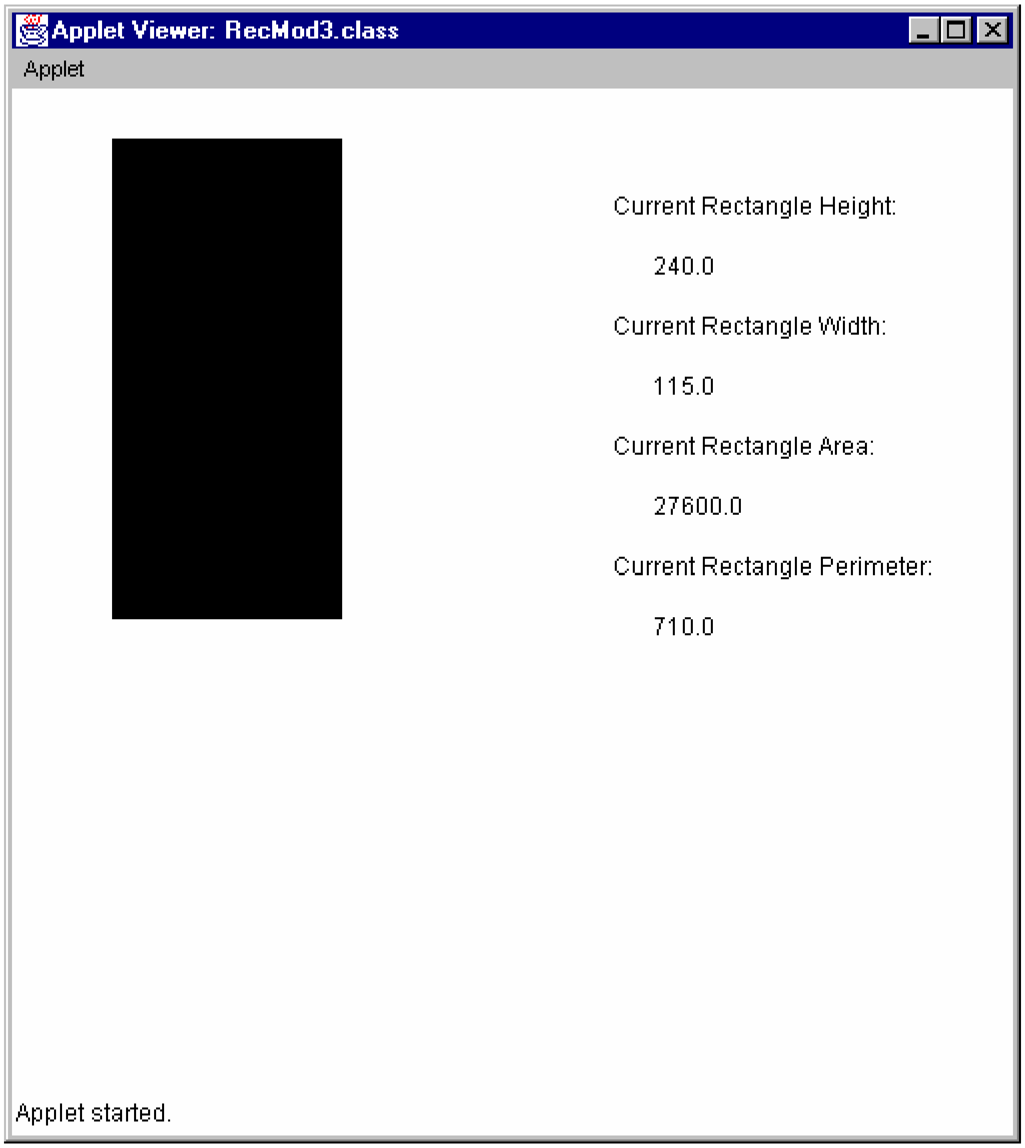


```
Current Rectangle Height:
190.0
```

Unit 2: Accessors & Modifiers

Exercises 3: Rectangle Modifiers

3. Rectangle Modifier 2 – Place one rectangle in the window and display its current height, width, area, and perimeter. As the mouse is clicked in the window increase the height of the rectangle by 50 pixels and the width of the rectangle by 10 while updating all of the values displayed.



```
Current Rectangle Height:
    240.0
Current Rectangle Width:
    115.0
Current Rectangle Area:
    27600.0
Current Rectangle Perimeter:
    710.0
```

Applet started.

Unit 2: Accessors & Modifiers

Exercises 4: Constructing a House

4. Home Builder – Use everything you have learned so far to construct a house. You can try to use new objects and methods if you would like to experiment; however, be sure to save your document before using unfamiliar code and do this without any assistance.

   This house must have features that demonstrate the use of accessors and modifiers. Use any accessors or modifiers you would like but be sure to use at least three. Make it possible to open the windows, open the doors, rain, snow, grow grass, or any other feature you would like.

Works-Cited

Applin, A. G. (2001). Second language acquisition and CS1. *SIGCSE '01: Proceedings of the Thirty-Second SIGCSE Technical Symposium on Computer Science Education,* Charlotte, North Carolina, United States. 174-178. Retrieved February 19, 2007, from ACM Digital Library.

Applin, Elizabeth Anne Gates. (1999). The application of language acquisition theory to programming concept instruction: Chunks versus programs from scratch. (Doctoral Dissertation, University of Southern Mississippi, 1999). (9960875)

Astrachan, O., Parlante, N., Garcia, D. D., & Reges, S. (2007). Teaching tips we wish they'd told us before we started. *SIGCSE '07: Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education,* Covington, Kentucky, USA. 2-3. Retrieved February 24, 2007 from ACM Digital library.

Barnes, D. J., & Kolling, M. (2003). *Objects first with java, A practical introduction using BlueJ* (Second Edition ed.). New York: Pearson Education.

Beaubouef, T., & Mason, J. (2005). Why the high attrition rate for computer science students: Some thoughts and observations. *SIGCSE Bulletin (Association for Computing Machinery, Special Interest Group on Computer Science Education), 37*(2), 103-106. Retrieved January 19, 2008, from ACM Digital Library.

Bergin, T. (2007). A history of the history of programming languages. *Communications of the ACM*. New York, NY, USA. 69—74. Retrieved March 17, 2007, from ACM Digital Library.

Boole, G. (1958). *The laws of thought*. New York, NY: Dover Publications, Inc.

Brilliant, S., & Wiseman, T. (1996). The first programming paradigm and language dilemma. *SIGCSE Bulletin (Association for Computing Machinery, Special Interest Group on Computer Science Education),* 28(1), 338-342. Retrieved February 13, 2007, from ACM Digital Library.

Bruce, K. B., Danyluk, A. P., & Murtagh, T. P. (2006). *Java an eventful approach*. Upper Saddle River, New Jersey: Pearson Education.

Bruce, K. B., Danyluk, A. P., & Murtagh, T. P. (2001). A library to support a graphics-based object-first approach to CS 1. *SIGCSE '01: Proceedings of the thirty-second SIGCSE technical symposium on Computer Science Education*, ACM Press, New York, NY, USA. 6-10. Retrieved February 13, 2007, from ACM Digital Library.

Buckland, R. (1996). Can we improve teaching in computer science by looking at how english is taught? *ACSE '97: Proceedings of the 2nd Australasian Conference on Computer Science Education,* The Univ. of Melbourne, Australia. 155-162. from Retrieved February 18, 2007, from ACM Digital Library.

Clark, D., MacNish, C., and Royle, G. (1998). Java as a teaching language\—opportunities, pitfalls and solutions. *ACSE '98: Proceedings of the 3rd Australasian conference on Computer science education*, ACM Press, New York, NY, USA. 173-179. Retrieved April 18, 2007, from ACM Digital Library.

Cooper, S., Cassel, L., Cunningham, S., Moskal, B. (2005). Outcomes-Based Computer Science Education. *SIGSCE 2005*, St. Louis, Missouri, USA 260-261.

Creak, A. (2003). How things were, programming lessons from days gone by. *ACM SIGPLAN Notices*, ACM Press, New York, NY, USA. 11-16. from Retrieved February 16, 2007, from ACM Digital Library.

Daniels, H. (1996). *An introduction to vygotsky*. London; New York: Routledge.

Edwards, S. (2003). Rethinking computer science education from a test-first perspective. *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ACM Press, New York, NY, USA. 148-155. Retrieved March 19, 2007, from ACM Digital Library.

Fagan, B. Merkle, L. (2002), Quantitative Analysis of the Effects of Robots onIntroductory Computer Science Education. *ACM Journal of Educational Resources in Computing*, 2(4), 1-18.

Fowler, H. R., & Aaron, J. E. (1989). *The little brown handbook* (Fourth Edition ed.). Boston: Scott, Foresman and Company.

Hoffman, M. Dansdill, T. Herscovici, D. (2006) Bridging Writing to Learn and Writing in the Discipline in Computer Science Education. *SIGCSE '06: Proceedings of the 37th SIGCSE technical symposium on Computer science education*, Houston, Texas, USA: ACM Press. 117 – 121.

Hunter, M. (1982). *Mastery Teaching*. El Segundo, California: TIP Publications.

Kolling, M. (1999). The problem of teaching object-oriented programming, Part 1: Languages. *Journal of Object-Oriented Programming,* 11(8), 8-15. Retrieved March 17, 2007, from ACM Digital Library.

Kolling, M. (1999). The problem of teaching object-oriented programming, Part 2: Environments. *Journal of Object-Oriented Programming,* 11(9), 6-12. Retrieved March 17, 2007, from ACM Digital Library.

Kolling, M. & Rosenburg, J. (2001). Guidelines for teaching object orientation with Java. *ITiCSE '01: Proceedings of the 6th annual conference on Innovation and technology in computer science education,* New York, NY, USA ACM Press. 33-36. Retrieved February 19, 2007, from ACM Digital Library.

Kolling, M., Quig, B., Patterson, A., & Rosenberg, J. (2003). The BlueJ system and its pedagogy. *Journal of Computer Science Education, Special Issue on Learning and Teaching Object Technology,* 13(4), 1-12. Retrieved February 19, 2007, from ACM Digital Library.

Lightbown, P. M., & Spada, N. (1999). *How languages are learned* (Second Edition ed.). New York: Oxford University Press.

Lister, R., Berglund, A., Clear, T., Bergin, J., Garvin-Doxas, K., Hanks, B., et al. (2006). Research perspectives on the objects-early debate. *ITiCSE-WGR '06: Working Group Reports on ITiCSE on Innovation and Technology in Computer Science Education,* Bologna, Italy. 146-165. from Retrieved February 20, 2007, from ACM Digital Library.

Marzano, R. J. (2003).*What Works in Schools, Translating Research Into Action.* Alexandria, Virginia: ASCD Publications.

Maslen, B. L. (1976). *Dot*. New York: Scholastic Inc.

Maslen, B. L. (1976). *Mac*. New York: Scholastic Inc.

Maslen, B. L. (1976). *Mat*. New York: Scholastic Inc.

Maslen, B. L. (1976). *Sam*. New York: Scholastic Inc.

Mazaitis, D. (1993). The Object-Oriented Paradigm in the Undergraduate Curriculum: A Survey of Implementations and Issues. *SIGCSE Bulletin (Association for Computing Machinery, Special Interest Group on Computer Science Education), 25*(3), 58-64. Retrieved February 21, 2007, from ACM Digital Library.

Mitchell, W. (2000). A paradigm shift to OOP has occurred…implementation to follow. *CCSC '00: Proceedings of the fourteenth annual consortium on Small Colleges Southeastern conference*, Virginia, United States. 94—105. from Retrieved February 21, 2007, from ACM Digital Library.

Mitchell, W. (2001). A paradigm shift to OOP has occurred\…implementation to follow. *JCSC*, 16(2), 94-105.

Mooney, C. (2000). *Theories of childhood : an introduction to Dewey, Montessori, Erikson, Piaget and Vygotsky*. St. Paul, MN: Redleaf Press.

Nevison, C. and Wells, B. (2004). Using a maze case study to teach: object-oriented programming and design patterns. *ACE '04: Proceedings of the sixth conference on Australasian computing education*, Dunedin, New Zealand. 207-215. Retrieved March 17, 2007, from ACM Digital Library.

Petty, W. T., & Jensen, J. M. (1975). *Developing children's language*. Boston: Allyn and Bacon.

Powers, K., Ecott, S., Hirshfield, L. (2007). *Through the looking glass: teaching CS0 with Alice*. SIGCSE '07: Proceedings of the 38th SIGCSE technical symposium on Computer science education. p. 213 – 217. New York, NY, USA: ACM Press. Retrieved March 19, 2007, from ACM Digital Library.

Rasala, R. (2000). *Toolkits in first Year Computer Science: A Pedagogical Imperative*. SIGCSE '00: Proceedings of the thirty-first SIGCSE technical symposium on Computer science education. p. 185-191. Austin, Texas, United States: ACM Press. Retrieved February 18, 2007, from ACM Digital Library.

Robertson, S. A., & Lee, M. P. (1995). The application of second natural language acquisition pedagogy to the teaching of programming languages\—a research agenda. *SIGCSE Bulletin (Association for Computing Machinery, Special Interest Group on Computer Science Education), 27*(4), 9-12. Retrieved February 18, 2007, from ACM Digital Library.

Satzinger, John. W., & Tore, V. Orvik (2001). *The Object-Oriented Approach, Concepts, System Development, and Modeling with UML.* Boston, MA: Course Technology.