LESSON STUDY IN COMPUTER SCIENCE: INHERITANCE

PART 1: OVERVIEW

**Lesson Title:** Discovering Inheritance through a Popular Video Game in CS1

**Authors:** Terry Mason, Diane Christie, Bruce Johnston, and Radi Teleb at the University of Wisconsin-Stout

**Contact:** Terry Mason, mason@uwstout.edu

**Discipline or Field:** Computer Science

**Date:** 1/31/2008

**Course Name:** Computer Science 1

**Course Description:** Computer Science 1 is the introductory course for computer science. It introduces problem solving and algorithm development using Java as the high-level programming language. The course uses an objects first approach to programming. The majority of students are freshman in the Applied Math and Computer Science program. The lesson targets these students for developing a deeper understanding of inheritance.

**Executive Summary:** The lesson topic is inheritance in Computer Science 1 (CS1) courses. Inheritance is a powerful tool which is generally not fully understood by beginning students [16, 2, 4, 3] in computer science. They may understand the mechanics of making inheritance work, but don’t always comprehend the utility and power of it [3]. A deeper understanding of the topic is a learning goal that all teachers strive for in their students. This topic has a broad application as the introduction to programming is a course that is taught by many instructors in colleges and high schools throughout the world.

**Learning Goals** – The goal of this lesson is to illustrate the power and utility of inheritance as a tool in computer science with the graphics and engagement experienced by students playing video games. The lesson is designed using a familiar Mario game implemented in Java. The students were engaged in the project by first playing the game to identify the sprite objects. This sets up a class discussion on how these objects are organized into an inheritance hierarchy through shared characteristics and functionality. The students complete the project by using inheritance to complete the functionality of the game.

**Lesson Evaluations** – The results of surveys and quizzes compare the results of one section of students that completed the older inheritance laboratory with two sections of students that completed the new video game based laboratory. Student engagement in the new laboratory ranked close to exciting versus a ranking of marginally interesting to interesting for the older lesson. Student surveys show that students believe that the new lesson was exciting and it increased their understanding of inheritance hierarchies, the power of inheritance, and the usefulness of the lab. Student grades on a quiz administered four days after the laboratories show that student scored slightly higher after completing the new lesson compared to students completing the older lesson.
Observations and Exit Interviews – The lessons were observed by members of the lesson study team. Students showed a high level of engagement in the game and identifying the objects for missing functionality. They expressed a sense of accomplishment in extending the functionality of the game. In addition they showed a sense of accomplishment. Two different groups shouted “Yes!” when their new code provided the expected functionality of the game. In addition, students were engaged enough in the lesson to spend extra time to further investigate the code.
PART 2: THE LESSON

1 HOW THE LESSON IS INTENDED TO WORK

Students in the first course in computer programming tend to have difficulties absorbing core programming concepts, while they are learning the details of a new computer language[10]. Specifically, inheritance is a powerful tool which is generally not fully understood by beginning students [16, 2, 4, 3]. They may understand the mechanics of making inheritance work, but don’t always comprehend the utility and power of it [3]. A deeper understanding of the topic is a learning goal that all teachers strive for in their students. This topic has a broad application as the introduction to programming is a course that is taught by many instructors in colleges and high schools throughout the world.

In addition to the challenges of teaching inheritance, instructors must find a way to meet the expectations of a graphical computing experience expected by the Nintendo generation [17, 5]. This provides the challenge of finding an interactive lesson to impart the power and elegance of inheritance in a way expected by students. Engaging students through games has proven valuable in the past few years [10, 9, 6, 14, 18, 8, 15]. By basing the project on a familiar game, students required little or no background information to identify the objects and discover [1] the inheritance hierarchy.

This work illustrates how a single laboratory lesson engages students in the exercise while demonstrating the value of inheritance in object oriented programming. The laboratory lesson was designed to provide the following features:

- Engaging and familiar audio/visual learning environment with easily identifiable objects
- Guided student discovery of an object hierarchy through directed game play
- Immediate visual feedback on the results of code changes through its graphical interface.
- Active involvement in creating a sub-class to complete functionality of the game, thus gaining experience with code reuse
- Meaningful demonstration of the power of inheritance
- Direct example for discussing educational copyright laws

The Lesson Study focuses on a single two hour laboratory session in the normal structure of CS1. Students use pair programming [12, 11, 13] to complete projects during these weekly laboratories to apply topics covered in the older weeks lectures. The current programming language used to teach CS1 is Java.

Students will be creating a new class derived from an existing class. In this lesson, we will be using a video game to demonstrate the concepts and value of inheritance. Students will be able to play a version of the very popular video game, Mario. It will not work completely as expected. Students will have to revise and extend the code to allow the game to work correctly.
2 The Lesson Plan

2.1 Preparation of the Lesson
Preparation for this lesson was started by ensuring that the software that we were using adheres to all applicable educational fair use copyright laws. The initial code called Infinite Mario is public domain. This code is cleaned up and simplified for use by the students. The sprites package of code was refactored to reflect proper object-oriented design. The program was structured to play the game without all of the functionality completed. This permits the student the opportunity to complete the code through inheritance and the power of code reuse. In order to meet the requirement of Educational Fair Use for the images, the code is limited to just one single scene from the Mario game with a limited number of core characters.

In order to evaluate the engagement and effectiveness of the lesson, the new lesson was completed by two of the three Computer Science 1 sections in the Fall of 2007. All three sections were taught by the same professor with many years of experience teaching this course. The results of surveys and quizzes administered to one section of students that completed the older inheritance laboratory were compared with two sections of students that completed the new video game based laboratory. A survey shown in Appendix C was administered as the last step of the lesson. In addition, a quiz on inheritance shown in Appendix D was completed by all three sections on the Friday following the lesson. The older lesson was based on the well known bank accounts hierarchy and is widely used and well established in computer science as a successful lesson.

2.2 Administering the Lesson
Teachers planning to complete this lesson will need a way to provide the students the starter code. In addition, the teacher should be prepared to lead a meaningful discussion about the structure of the inheritance hierarchy and the value of code reuse in BigMario and FireMario. The rest of the lesson is completed through student interaction with the game.

2.3 Getting the game files (10 minutes)
Students should work in pairs, so there should be one computer for every two students (three if there is an odd number). Have the students download the zip file and unzip it on the computer. The Mario code is in the zipped file. The program is started by running the FrameLauncher class in com/mojang/mario package. Open FrameLauncher.java in the Java IDE and compile the code. The student may now play the game by running this file.

2.4 Identifying objects in a program (15 minutes)
Students should take turns with their partner playing Mario. As they play, they should be discussing what objects are in the game. The student not playing at the time should be writing a list of all objects in the game.

2.5 Making an inheritance hierarchy (20 minutes)
Stop game play. Gather the group together with their object lists. On the board, brainstorm a combined list of objects in the game. Solicit student input as to how to separate the objects into two distinct categories (objects that move and objects that do
not move). Focus on the objects that move. How are they related? Solicit student input to draw an inheritance hierarchy on the board and group the objects into three categories of Avatar (controlled by user), Movers (move around the scene), and Animation (move in a single location in the scene).

During the full class discussion, the common elements of the sprites such as x and y coordinates, and image are identified to develop a hierarchy. There are three different types of Sprites in the game. Mario is the main character controlled by the user. This is known as an Avatar type of Sprite. Elements that appear in the game and move about the scene are grouped into movers, while the elements that are fixed to their location in the scene are identified as Animation. The discovery of objects and grouping them with common elements efficiently helps to establish the idea of sharing common functionality in a super-class that is used by the sub-classes. The Enemy class has a variable type that displays a Koopa or Gumba. This will be object oriented on the next set of code. The students are focused on Mario and did not ask about this structure. It will be fully object oriented in future lessons.

2.6 Discovering the missing derived class (15 minutes)
Students resume taking turns playing Mario in pairs. As they are playing, they should be looking for anything that doesn’t work as expected from the classic game. The student not playing at the time should write down a list of anything that doesn’t work as expected.

2.7 Thinking inheritance (20 minutes)
Stop the game play. Gather the group together with their list of program problems. A well known (to our students) functionality in the performance of the game is missing from the original code. The students are challenged to identify the missing functionality and develop a strategy to add this functionality to the game. Brainstorm ways to fix this problem by soliciting ideas from students. What are the characteristics of Mario that would
be helpful in developing a BigMario object. Students are led to the concept that BigMario is just an extended version of Mario with additional characteristics and functionality.

Discussion of the missing functionality should lead the class to discuss the missing elements in the code. The students identified the need for a Big Mario sprite. Brainstorming led to the discovery that the existing Mario class had most of the characteristics of Big Mario. We could make BigMario a sub-class of Mario and reuse the existing Mario code. Their discovery of this solution gives them ownership in their code and the project.

2.8 Writing a derived class (30 minutes)

Give the students a UML diagram of the Sprite, Avatar, Mario and BigMario classes and explanations of relevant methods. Discuss how a Mario object currently works and how a BigMario object should work. Students should work in pairs to write the BigMario class as specified. This code should be placed one level down in the com/mojang/mario/sprites directory (same directory as Mario class).

![UML diagram of Sprite, Avatar, Mario, and BigMario classes]

Figure 2. Inheritance Hierarch with BigMario and FireMario Classes

2.8.1 Extending the Hierarchy

Students are provided UML diagrams shown in Appendix B for the lesson illustrating the hierarchy from the Sprites class down through Mario to the new BigMario class. UML diagrams are used through the semester and familiar to the students. By extending the existing Mario class and using the code already developed in the Sprites class student are able to significantly impact the codes functionality with minimal amounts of coding. This allows students to experience the value of code reuse in addition to learning the mechanics and syntax of inheritance.
2.8.2 Testing the Code
Students were quite motivated to “complete” the game. Exclamations of “Yes!” could be heard as Mario grows upon finding a Mushroom during testing of the new BigMario class. This provided a contrast to other projects as students normally seem to find little motivation to test their own code.

2.9 Making it work (5 minutes)
Students should uncomment the LevelScene class code (located in com/mojang/mario/), compile BigMario and LevelScene, and run FrameLauncher to access the game with the code supporting the extended functionality.

2.10 Making it work (FireMario) (25 minutes)
Implement FireMario and then the students should uncomment the LevelScene class code (located in com/mojang/mario/), compile FireMario and LevelScene, and run FrameLauncher to play the game with the extended functionality. See Appendix B for details about the code and Appendix A for the solutions.

3 STUDENT LEARNING OBJECTIVES
Five objectives were identified prior to the development of the lesson. The sixth objective of discussing the ethics of using copyrighted material developed with the decision to use the familiar Mario game. Below is a discussion of each objective with observable or measurable results.

3.1 Ability to define a class hierarchy
Using a familiar game allowed the students to quickly identify the objects (characters and elements) in the game as shown in Figure 1. The entire class stops the lab to discuss the hierarchical structure of the sprites objects and together defines the inheritance hierarchy.

3.2 Derive a class from an existing class
Incomplete functionality in the original code should lead the students to identify the need to support the Mario sprite in becoming big when colliding with a mushroom sprite. Students suggest the addition of a BigMario sprite to complete the game. Additional discussion led the students to realize that BigMario is a Mario object with more data and/or operations. They realized that they could add the desired functionality by creating a sub-class of the existing Mario class.

3.3 Create a derived-class in which methods are overridden and fields are hidden
All pair programmers in this course were successful in creating a BigMario and a FireMario sprite by extending the existing Mario class. Some required additional guidance from the teacher or student lab assistant, but all ultimately completed the functionality of the game.

3.4 Recognize inheritance concepts
In two exit interviews the students stated that they now understood the “tree” or “tree thing”. Expressing their understanding of the hierarchy in their own words gives encouragement that the concept was internalized and not just repeated.
3.5 Understand the value of inheritance
The power of code reuse in inheritance appeared to make an impact on the students, as many students expressed surprise at the ability to change the game through only 30 lines of code. A change to a video game like this was expected to require significant amounts of coding. They appear to understand the value of code reuse provided through inheritance.

3.6 Explore the Ethics of Creating a Game with Copyrighted Material
The lesson provides a concrete example of the decisions to use copyrighted materials based on educational fair use by carefully restricting how much of the original material is used. During the lecture periods prior to the lesson, the ethics of using a Mario game must be discussed. We established that this is only acceptable for a single lesson with only a single scene from a game. Why did we use the Mario game instead of making our own? The Mario game allowed students to be completely familiar with the characters and able to quickly identify the objects. This removed the common overhead of explaining objects in code. These objects are well known to the “Nintendo generation”.
PART 3: THE STUDY

1 APPROACH
In order to quantify the difference between the new lesson and the older lesson we attempted to create a control group. The same professor taught three sections of Computer Science 1 during the fall semester. One of the sections completed the well established inheritance lesson based on bank accounts. The two other sections completed the newly created Mario lesson. The three sections were presented the same course materials except for this lesson during the semester. In fact, we did not decide which section would be the control section until the week before the laboratory. We wanted to make sure that the sections were performing similarly on the preceding exams and laboratories. We treated all three laboratories the same. The students completing the older lesson were filmed and given the surveys in the same manner that the new lesson was observed. Aside from signing the waiver to gather data, the students did not know that anything was different and they did not know that different sections would complete different laboratories.

2 FINDINGS
Two video cameras captured student responses during the lesson. Exit interviews were completed with two separate pairs of programmers. Important observations are available for review on the web [7]. In addition, the students completed a survey shown in Appendix D with results showing an increase in engagement in the new lesson compared to the established lesson. A quiz was administered four days after the laboratory session on inheritance. The students completing the new lab performed slightly better on the quiz than those that completed the older lesson. The details of the findings are explained in the next few sections.

2.1 Exit Interviews
Two teams of pair programmers were interviewed on video [7] after the completion of the laboratory project. Each team consisted of one male and one female.

2.1.1 Team1
They stated that it was the most fun that they have had in a lab through the opportunity to interact with a graphics environment. The team was more engaged in the programming as it was “not boring”. The female team member suggested that this lesson really helped her understand the “tree” and how to “pass in and override” to make the code “shorter”.

2.1.2 Team2
This team expressed the same enthusiasm about the engagement of the project. The female felt “confident and understood what we were doing today” after expressing frustrations in understanding earlier course material. In response to what she learned she claimed to “understand the tree.” In a telling remark she expressed that the ability to “see what is actually going on is what I really enjoyed.”

Her partner explained that he was familiar the objects and the game scenario as he had the background of playing video games. He would like to dig into the program some more and believed it would be fulfilling to complete a series of projects based on elements of this game.
2.2 Student Engagement Survey Results
At the end of each laboratory throughout the semester, students complete a set of questions regarding the lab before receiving credit for the lab. The students receive a single participation grade for completing the laboratory and set of questions. The survey shown in the appendix with the following questions was administered at the conclusion of the lesson in the place of the usual questions. The older lesson was based on the well known bank accounts hierarchy and is widely used and well established in computer science as a successful lesson.

2.2.1 Survey Results

<table>
<thead>
<tr>
<th>Number of Students</th>
<th>Lesson</th>
<th>Question Number</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>23</td>
<td>Older Bank Account Lesson</td>
<td>2.74</td>
<td>2.70</td>
<td>2.70</td>
<td>2.48</td>
<td>2.96</td>
</tr>
<tr>
<td>46</td>
<td>Mario Lesson</td>
<td>3.28</td>
<td>3.33</td>
<td>2.87</td>
<td>3.78</td>
<td>3.5</td>
</tr>
<tr>
<td>Increase in new lesson</td>
<td></td>
<td>19.8%</td>
<td>23.4%</td>
<td>6.5%</td>
<td>52.6%</td>
<td>18.4%</td>
</tr>
</tbody>
</table>

Table 1. Survey Results comparing the new Mario Lesson the older bank account lesson.

2.2.2 Survey Questions- Results and Discussion

1. How much did today’s lab exercise increase your understanding of the inheritance hierarchy in an object oriented program? (1-No help, 2-Somewhat helpful, 3-Helpful, and 4-Extremely helpful)
   New lesson moved above the helpful range with an average of 3.28 compared to 2.74 for the older approach. The opportunity to visually observe the sprites objects and creating the hierarchy as a group targeted this improvement.

2. How much did today’s lab exercise help you understand the power of inheritance in Java (i.e., based classes and derived classes)? (1-No help, 2-Somewhat helpful, 3-Helpful, and 4-Extremely helpful)
   The new lesson again moved above the helpful range with an average of 3.33 compared to 2.70 for the older approach. The opportunity to visually observe the sprites objects and creating the hierarchy as a group targeted this improvement. In addition, the students had the opportunity to add functionality to a complex set of code with minimal effort through code reuse in inheritance.

3. How well did today’s lab exercise help you to understand the process of overloading and overriding base class methods? (1-No help, 2-Somewhat helpful, 3-Helpful, and 4-Extremely helpful)
   The new lesson showed only comparable results to the older lesson. The new lesson could use more emphasis on overloading and overriding methods. The older lesson is strong in this area.

4. How interesting was the application used for today’s lab exercise? (1-Boring, 2-Marginally interesting, 3-Interesting, and 4-Exciting)
   This is the area where the new lesson was targeting an increase. We believed that a familiar video game would increase the interest in the subject. The new lesson scored close to the top level of exciting with a
52% increase in student excitement results. In fact the older lesson showed its lowest score in this area, while the new lesson showed its highest score. The opportunity to play a familiar game for learning a computer science subject proved exciting to the students. In addition, the opportunity for students to complete the functionality in a familiar game was observed to be motivating and reinforced through exit interviews.

5. How would you rate this lab exercise overall? 1-Not useful, 2-Somewhat useful, 3-Useful, and 4-Extremely useful)
While the well written and established older lesson was seen as useful for the students, the new lesson was seen as extremely useful by half of the students.

2.3 Quiz Results
The overall scores for the lesson show that the new lesson is as successful as the well established bank accounts approach for teaching inheritance. The older lesson was based on the well known bank accounts hierarchy and is widely used and well established in computer science as a successful lesson. The new approach showed a slight increase of 3% over the accounts lesson.

<table>
<thead>
<tr>
<th>Students</th>
<th>Lesson</th>
<th>Quiz Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>Older Bank Account Lesson</td>
<td>7.476</td>
</tr>
<tr>
<td>46</td>
<td>New Mario Lesson</td>
<td>7.698</td>
</tr>
</tbody>
</table>

Table 2. Quiz Results for students that completed the two lessons.

2.4 Additional Observations
The entire lesson is observed by all members of the Lesson Study team. Many of these significant observations captured on videotape are available online[7].

2.4.1 High Level of Engagement
Observations of the students showed them clearly engaged in the game and identifying the objects for missing functionality. They expressed a sense of accomplishment in extending the functionality of the game.

2.4.2 Sense of Accomplishment
Two different groups shouted “Yes!” when their new code provided the expected functionality of the game. In our 60 years of experience in teaching labs on inheritance, none of the observing professors remembers this happening. The graphical demonstration of their results was valued by the students.

2.4.3 Exploring beyond the Assigned Project
From the back of the room it was observed that students would “sneak” some time to complete other tasks during the class. Students were observed investigating how the images could look like Mario is running on the screen. Other students were noted to explore code to try to further understand how the program worked. It is encouraging that students showed enough engagement to further investigate the program.
2.5 DISCUSSION
Using a familiar video game allowed student to easily identify the object in a large amount of code. The familiar game engaged the students quickly into the project by completing the tasks in about half the expected time for the project. Students were motivated to add functionality that they expected to be in the game allowing the illustration of the value of code reuse in inheritance. The students working on the project expressed fulfillment and increased understanding through the visual feedback provided by a graphical environment. The survey and quiz provide quantitative results showing that students met the objectives of better understanding inheritance. It is hoped that the increased usefulness and interest in the project expressed by the students, results in a deeper appreciation and understanding of the value of inheritance in object oriented programming. Student engagement in the new laboratory ranked close to exciting versus a ranking of marginally interesting to interesting for the older lesson. In addition student surveys show that students believe that the lesson increased their understanding of inheritance hierarchies, the power of inheritance, and the usefulness of the lab compared the students completing the older lab.

2.6 FUTURE WORK
This code provides a foundation for build in new games with different (non-copyrighted) materials. This would support the development of additional projects on this somewhat familiar code. A follow up lesson on polymorphism, interfaces, and abstract classes based on this code would further reinforce the value of object oriented code. In addition, a group of students are developing a generic side scrolling game engine. This would allow the development of a project for students to build their own object-oriented games based on their knowledge acquired through this lesson.

2.7 ACKNOWLEDGMENTS
We would like to acknowledge the University of Wisconsin – Stout Nakatani Center for funding this Lesson Study Training Grant.
REFERENCES


APPENDIX
A. **Lesson Plan** - Overview of the lesson for instructors including solutions.
B. **Inheritance Lab** - Student handout for the day of the laboratory.
C. **Engagement Survey** – Conducted at the conclusion of the laboratory in class.
D. **Chapter Quiz** – Quiz administered on the Friday following the Monday laboratory.
Appendix A

Lesson Plan

1. Overview
Students will be creating a new class derived from an existing class. In this lesson, we will be using a video game to demonstrate the concepts and value of inheritance. Students will be able to play a version of the very popular video game, Mario. It will not work completely as expected. Students will have to write the code to allow the game to work correctly.

2. Getting the game files (10 minutes)
Students should work in pairs, so there should be one computer for every two students (three if there is an odd number). Have the students download the zip file and unzip it on the computer. The Mario code is in the zipped file. The program is started through the FrameLauncher class in com/mojang/mario package (and directories). Open FrameLauncher.java in their Java IDE. Compile and run.

3. Identifying objects in a program (15 minutes)—Task 1
Students should take turns with their partner playing Mario. As they play, they should be discussing what objects are in the game. The student not playing at the time should be writing a list of all objects in the game.

4. Making an inheritance hierarchy (20 minutes)—Task 2
Stop game play. Gather the group together with their object lists. On the board, brainstorm a combined list of objects in the game. Solicit student input as to how to separate the objects into two distinct categories (objects that move and objects that do not move). Focus on the objects that move. How are they related? Solicit student input to draw an inheritance hierarchy on the board and group the objects into three categories of Avatar (controlled by user), Movers (move around the scene), and Animation (move in a single location in the scene).

5. Discovering the missing derived class (15 minutes)—Task 3
Students should resume taking turns playing Mario in pairs. As they are playing, they should be looking for anything that doesn’t work as expected from the classic game. The student not playing at the time should write down a list of anything that doesn’t work as expected.

6. Thinking inheritance (20 minutes)
Stop the game play. Gather the group together with their list of program problems. Focus on Mario not getting larger when he gets a mushroom. Brainstorm ways to fix this problem by soliciting ideas from students. What are the characteristics of Mario that would be helpful in developing a BigMario object.

7. Writing a derived class (30 minutes)—Task 4
Give the students a UML diagram of the Sprite, Avatar, Mario and BigMario classes and explanations of relevant methods. Discuss how a Mario object currently works and how a BigMario object should work. Students should work in pairs to write the BigMario class as specified. This code should be placed one level down in com/mojang/mario/sprites directory (same directory as Mario class).
<table>
<thead>
<tr>
<th><strong>Sprite (abstract)</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>+x:int</td>
</tr>
<tr>
<td>+y:int</td>
</tr>
<tr>
<td>+xFlipPic:Boolean</td>
</tr>
<tr>
<td>+sheet:Image[][]</td>
</tr>
<tr>
<td>+Sprite(world:LevelScene):</td>
</tr>
<tr>
<td>+move():void [abstract]</td>
</tr>
<tr>
<td>+render(og:Graphics, alpha:float): void</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Avatar (abstract)</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>+keys:boolean[]</td>
</tr>
<tr>
<td>-sideWaysSpeed:float</td>
</tr>
<tr>
<td>+Avatar(world:LevelScene):</td>
</tr>
<tr>
<td>+rightPressed():boolean</td>
</tr>
<tr>
<td>+leftPressed():boolean</td>
</tr>
<tr>
<td>+aPressed():boolean</td>
</tr>
<tr>
<td>+sPressed():boolean</td>
</tr>
<tr>
<td>+downPressed():boolean</td>
</tr>
<tr>
<td>+move():void</td>
</tr>
<tr>
<td>+move(xa:float, ya:float):Boolean</td>
</tr>
<tr>
<td>+win():void</td>
</tr>
<tr>
<td>+die():void</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Mario</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>+coins:int</td>
</tr>
<tr>
<td>+lives:int</td>
</tr>
<tr>
<td>+Mario(world: LevelScene):</td>
</tr>
<tr>
<td>+getCoin(): void</td>
</tr>
<tr>
<td>+selectPicture():void</td>
</tr>
<tr>
<td>+move():void</td>
</tr>
<tr>
<td>+rightAction():void</td>
</tr>
<tr>
<td>+leftAction():void</td>
</tr>
<tr>
<td>+sAction():void</td>
</tr>
<tr>
<td>+aAction():void</td>
</tr>
<tr>
<td>+stomp(enemy:Enemy):void</td>
</tr>
<tr>
<td>+stomp(shell:Shell):void</td>
</tr>
<tr>
<td>+kick(shell:Shell):void</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>BigMario</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>+BigMario(world: LevelScene):</td>
</tr>
<tr>
<td>+move(): void</td>
</tr>
<tr>
<td>+downAction(): void</td>
</tr>
</tbody>
</table>
### BigMario class

- **Constructor**
  - The constructor uses the LevelScene object representing Mario’s world in the game. Use the world variable to initialize all of the super classes with a call to the superconstructor.
  - Initialize sheet to Art.mario, xPicO to 16, yPicO = 31, wPic to 32, hPic to 32, height to 24.

- **move method**
  - Override the move method in the super class.
    - Implement the ducking function of BigMario.
      - If downPressed call downMove method and then call afterMove to complete the ducking
      - Else set height to 24 and execute move in Mario

- **downAction method**
  - if onground is true
    - Set xPic to 14 for the BigMario ducking image and height to 12 as ducking BigMario is shorter.
  - otherwise return

### BigMario.java (solution)

```java
import com.mojang.mario.Art;
import com.mojang.mario.LevelScene;
public class BigMario extends Mario {
    public BigMario(LevelScene world) {
        super(world);
        sheet = Art.mario;
        xPicO = 16;
        yPicO = 31;
        wPic = 32;
        hPic = 32;
        height = 24;
    }

    public void move() {
        if (downPressed()) {
            downAction();
            afterMove();
        }
        else {
            height = 24;
            super.move();
        }
    }

    public void downAction() {
        if (onGround) {
            xPic = 14;
            height = 12;
        }
    }
}
```

8. **Making it work (5 minutes)**—Task 5
   Students should uncomment the LevelScene class code (located in com/mojang/mario/), compile BigMario and LevelScene, and run FrameLauncher to play the game with the extended functionality.

9. **Making it work even more (25 minutes)**—Task 6
   Implement FireMario and then the students should uncomment the LevelScene class code (located in com/mojang/mario/), compile FireMario and LevelScene, and run FrameLauncher to play the game with the extended functionality.

---

**LevelScene.java**

Uncomment `getHurt` and `powerUp` methods to add BigMario
<table>
<thead>
<tr>
<th>Mario</th>
</tr>
</thead>
<tbody>
<tr>
<td>+coins:int</td>
</tr>
<tr>
<td>+lives:int</td>
</tr>
<tr>
<td>+Mario(world: LevelScene):</td>
</tr>
<tr>
<td>+getCoin(): void</td>
</tr>
<tr>
<td>+selectPicture():void</td>
</tr>
<tr>
<td>+move():void</td>
</tr>
<tr>
<td>+rightAction():void</td>
</tr>
<tr>
<td>+leftAction():void</td>
</tr>
<tr>
<td>+sAction():void</td>
</tr>
<tr>
<td>+tAction():void</td>
</tr>
<tr>
<td>+stomp(enemy:Enemy):void</td>
</tr>
<tr>
<td>+stomp(shell:Shell):void</td>
</tr>
<tr>
<td>+kick(shell:Shell):void</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BigMario</th>
</tr>
</thead>
<tbody>
<tr>
<td>+BigMario(world: LevelScene):</td>
</tr>
<tr>
<td>+move(): void</td>
</tr>
<tr>
<td>+downAction(): void</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FireMario</th>
</tr>
</thead>
<tbody>
<tr>
<td>-oneShotPerKey: boolean</td>
</tr>
<tr>
<td>+FireMario(world: LevelScene):</td>
</tr>
<tr>
<td>+aAction(): void</td>
</tr>
</tbody>
</table>
FireMario class

- Instance variable oneShotPerKey is a Boolean that allows only one shot per move.
- Constructor
  - Use the LevelScene variable world to initialize the super classes with a call to the superconstructor.
  - Initialize sheet to Art.fireMario and set oneShotPerKey to true;
- Override the aAction method to shoot a fireball when it is pressed.
  - Use super to call the aAction method in the super classes.
  - Add the following code to shoot a fireball. Note that only two fireballs may be on the screen at one time and that a single click of a should only shoot one fireball.

```java
//Code to add to the aAction method after calling the parent classes aAction method
if (world.fireballsOnScreen<2&&oneShotPerKey)
{
    world.sound.play(Art.samples[Art.SAMPLE_MARIO_FIREBALL], this, 1, 1, 1);
    world.addSprite(new Fireball(world, x+facing*6, y-20, facing));
}
if(oneShotPerKey)
    oneShotPerKey = false;//one shot per aPressed()
else
    oneShotPerKey = true;
```

FireMario Class (Solution)

```java
import com.mojang.mario.Art;
import com.mojang.mario.LevelScene;
public class FireMario extends BigMario {
    private boolean oneShotPerKey;
    public FireMario(LevelScene world) {
        super(world);
        sheet = Art.fireMario;
        oneShotPerKey = true;
    }
    public void aAction()
    {
        super.aAction();
        if (world.fireballsOnScreen<2&&oneShotPerKey)
            { 
                world.sound.play(Art.samples[Art.SAMPLE_MARIO_FIREBALL], this, 1, 1, 1);
                world.addSprite(new Fireball(world, x+facing*6, y-20, facing));
            }
        if(oneShotPerKey)
            oneShotPerKey = false;//shoot only once per key click
        else
            oneShotPerKey = true;
    }
}
```
Appendix B

Inheritance Lab

Today’s Objectives

- Be able to derive a class from an existing class
- Be able to define a class hierarchy in which methods are overridden
- Understand the value of inheritance

Introduction

In this lab, you will be creating a new class derived from an existing class. We will be using a video game to demonstrate the concepts and value of inheritance. You will be able to play Mario. It may not work completely as expected. You will have to write the code to allow the game to work correctly.

Download the zip file and unzip it on your computer. The Mario code is in the zipped file. The program is started through the FrameLauncher class in com/mojang/mario package (and directories). Open FrameLauncher.java in JGrasp. Compile and run.

Task 1 Finding the Objects

(15 minutes). Take turns with your partner playing Mario. Make a list of all objects in the game.
Task 2 Making an Inheritance Hierarchy

(15 minutes). Together with the class, brainstorm objects in the game. Draw the inheritance hierarchy below.

Task 3 Discovering the Missing Derived Class

(15 minutes). Write down anything that doesn’t work as expected from the classic game.

Task 4 Writing a Derived Class

BigMario class

- Constructor
  - The constructor uses the LevelScene object representing Mario’s world in the game. Use the world variable to initialize all of the super classes with a call to the superconstructor.
  - Initialize sheet to Art.mario, xPicO to 16, yPicO = 31, wPic to 32, hPic to 32, height to 24.
- move method
  - Override the move method in Mario to add the ducking function of BigMario.
    - If downPressed call the new downMove method and then call afterMove to complete the ducking move.
    - Else set height to 24 and execute move in superclass Mario
- downAction method
  - if onground is true
    - Set xPic to 14 to show the BigMario ducking image.
    - Set height to 12 as a ducking BigMario is shorter.
  - otherwise return
<table>
<thead>
<tr>
<th><strong>Sprite {abstract}</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>+x:int</td>
</tr>
<tr>
<td>+y:int</td>
</tr>
<tr>
<td>+xFlipPic:Boolean</td>
</tr>
<tr>
<td>+sheet:Image[][]</td>
</tr>
<tr>
<td>+Sprite(world:LevelScene):</td>
</tr>
<tr>
<td>+move():void {abstract}</td>
</tr>
<tr>
<td>+render(og:Graphics, alpha:float): void</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Avatar {abstract}</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>+keys:boolean[]</td>
</tr>
<tr>
<td>-sideWaysSpeed:float</td>
</tr>
<tr>
<td>+Avatar(world:LevelScene):</td>
</tr>
<tr>
<td>+rightPressed():boolean</td>
</tr>
<tr>
<td>+leftPressed():boolean</td>
</tr>
<tr>
<td>+aPressed():boolean</td>
</tr>
<tr>
<td>+sPressed():boolean</td>
</tr>
<tr>
<td>+downPressed():boolean</td>
</tr>
<tr>
<td>+move():void</td>
</tr>
<tr>
<td>+move(xa:float, ya:float):Boolean</td>
</tr>
<tr>
<td>+win():void</td>
</tr>
<tr>
<td>+die():void</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Mario</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>+coins:int</td>
</tr>
<tr>
<td>+lives:int</td>
</tr>
<tr>
<td>+Mario(world: LevelScene):</td>
</tr>
<tr>
<td>+getCoin(): void</td>
</tr>
<tr>
<td>+selectPicture():void</td>
</tr>
<tr>
<td>+move():void</td>
</tr>
<tr>
<td>+rightAction():void</td>
</tr>
<tr>
<td>+leftAction():void</td>
</tr>
<tr>
<td>+sAction():void</td>
</tr>
<tr>
<td>+aAction():void</td>
</tr>
<tr>
<td>+stomp(enemy:Enemy):void</td>
</tr>
<tr>
<td>+stomp(shell:Shell):void</td>
</tr>
<tr>
<td>+kick(shell:Shell):void</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>BigMario</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>+BigMario(world: LevelScene):</td>
</tr>
<tr>
<td>+move(): void</td>
</tr>
<tr>
<td>+downAction(): void</td>
</tr>
</tbody>
</table>
Task 5 Making BigMario Work

In the same folder as Mario (com/mojang/mario/sprites) is the Mushroom class with the commented out code to create the BigMario object. Uncomment the code so that the class you have written will be implemented. You will also find the other classes listed in the UML.

Compile BigMario and Mushroom. Run FrameLauncher again. Play the game with your partner. The game should now work as expected.

Task 6 Extra Challenge: Making FireMario Work

Implement FireMario and then the students should uncomment the LevelScene class code (located in com/mojang/mario/), compile FireMario and LevelScene, and run FrameLauncher to play the game with the extended functionality.

FireMario class

- Instance variable oneShotPerKey is a Boolean that allows only one shot per move.
- Constructor
  - The constructor uses the LevelScene object representing the Avatar’s world in the game. Use the world variable to initialize the super classes with a call to the superconstructor.
  - Initialize sheet to Art.fireMario and set oneShotPerKey to true;
- Override the aAction method the extend the functionality of the superclasses to shoot a fireball when it is pressed.
  - Use super to call the aAction method in the super classes.
  - Add the following code to shoot a fireball. Note that only two fireballs may be on the screen at one time and that a single click of a should only shoot one fireball.

//Code to add to the aAction method after calling the parent classes aAction method

```java
if (world.fireballsOnScreen<2&&oneShotPerKey)
{
    world.sound.play(Art.samples[Art.SAMPLE_MARIO_FIREBALL], this, 1, 1, 1);
    world.addSprite(new Fireball(world, x+facing*6, y-20, facing));
}
if(oneShotPerKey)
    oneShotPerKey = false;//one shot per aPressed()
else
    oneShotPerKey = true;
```
<table>
<thead>
<tr>
<th>Mario</th>
</tr>
</thead>
<tbody>
<tr>
<td>+coins:int</td>
</tr>
<tr>
<td>+lives:int</td>
</tr>
<tr>
<td>+Mario(world: LevelScene):</td>
</tr>
<tr>
<td>+getCoin(): void</td>
</tr>
<tr>
<td>+selectPicture():void</td>
</tr>
<tr>
<td>+move():void</td>
</tr>
<tr>
<td>+rightAction():void</td>
</tr>
<tr>
<td>+leftAction():void</td>
</tr>
<tr>
<td>+sAction():void</td>
</tr>
<tr>
<td>+aAction():void</td>
</tr>
<tr>
<td>+stomp(enemy:Enemy):void</td>
</tr>
<tr>
<td>+stomp(shell:Shell):void</td>
</tr>
<tr>
<td>+kick(shell:Shell):void</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BigMario</th>
</tr>
</thead>
<tbody>
<tr>
<td>+BigMario(world: LevelScene):</td>
</tr>
<tr>
<td>+move(): void</td>
</tr>
<tr>
<td>+downAction(): void</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FireMario</th>
</tr>
</thead>
<tbody>
<tr>
<td>-oneShotPerKey: boolean</td>
</tr>
<tr>
<td>+FireMario(world: LevelScene):</td>
</tr>
<tr>
<td>+aAction(): void</td>
</tr>
</tbody>
</table>
Appendix C

CS1 Inheritance Lab
Exit Survey

1. How much did today's lab exercise increase your understanding of the inheritance hierarchy in an object oriented program?
   - No help
   - Somewhat helpful
   - Helpful
   - Extremely Helpful

2. How much did today's lab exercise help you understand the power of inheritance in Java (i.e., base classes and derived classes)?
   - No help
   - Somewhat helpful
   - Helpful
   - Extremely Helpful

3. How well did today's lab exercise help you understand the process of overloading and overriding base class methods?
   - No help
   - Somewhat helpful
   - Helpful
   - Extremely Helpful

4. How interesting was the application used for today's lab exercise?
   - Boring
   - Marginally interesting
   - Interesting
   - Exciting

5. How would you rate this lab exercise overall?
   - Not useful
   - Somewhat useful
   - Useful
   - Extremely Useful
Appendix D

Quiz 9  Chapter 9  Name____________________________

1. A program uses two classes: **Flower** and **Rose**.
   a. Which class is the base class and which is the derived class?

   b. Write the class heading for the derived class.

2. Can a derived class **directly** access
   a. The private members of the base class?

   b. The public members of the base class?

   c. The protected members of the base class?

3. Since constructors are not inherited, a derived class must have its own constructor. What is the first thing that should be done in this constructor?

4. What is the difference between **overriding** a base class method and **overloading** a base class method?

5. How could you access an overridden **toString** method from the derived class?

6. For the following questions, **check all items that apply**.
   a. Building a derived class using inheritance could include which of the following steps?
      ___ Adding new specialized data that is not available in the parent class.
      ___ Adding new methods that are not available in the parent class.
      ___ Removing data items from the parent class that are not appropriate for the derived class.
      ___ Overriding one or more methods from the parent class.

   b. Which of the following inheritance operations are logically correct?
      ___ public class Airplane extends AirplaneSeat
      ___ public class AisleSeat extends AirplaneSeat
      ___ public class WindowSeat extends AirplaneSeat
      ___ public class CoffeeTable extends Table
      ___ public class CableModem extends Modem