Inheritance (Outsource: 9-12 – 9-14)

Inheritance is a way to form new classes using classes that have already been defined. The new classes, known as derived classes, inherit attributes and behavior of the pre-existing classes, which are referred to as base classes. The purpose of inheritance is to allow the reuse of existing code with little or no modification.

Inheritance is sometimes referred to as generalization, because the **is-a** relationships represent a hierarchy between classes. For instance, an "animal" is a generalization of "cat", "dog", and "horse". One can consider animal to be an abstraction of cat, dog, and horse. Conversely, since dogs are animals (i.e., a dog is-a animal), dogs may naturally inherit all the properties common to all animals.

Each subclass inherits properties (in the form of variable declarations) from the superclass. Dogs, cats, and horses share some properties: they all have two eyes, two ears, four legs, etc. Also, each subclass inherits methods from the superclass. Dogs, cats, and horses share some behaviors: they all eat, they all sleep, etc.

However, subclasses are not limited to the state and behaviors provided to them by their superclass. Subclasses can add variables and methods to the ones they inherit from the superclass. Dogs hunt in packs, cats climb trees, horses are herbivores.

Subclasses can also override inherited methods and provide specialized implementations for those methods.

You are not limited to just one layer of inheritance. The inheritance tree, or class hierarchy, can be as deep as needed. Methods and variables are inherited down through the levels. In general, the farther down in the hierarchy a class appears, the more specialized its behavior.

Extending Existing Classes (Outsource: 11-30)

Inheritance is accomplished by using the keyword extends. The subclass extends the superclass. For example:

```
    public class Dog extends Animal
```

creates a subclass called Dog that inherits all the attributes of its superclass, Animal.

What You Can Do in a Subclass

A subclass inherits all of the *public* and *protected* members of the superclass, no matter what package the subclass is in. If the subclass is in the same package as its parent, it also inherits the *package-private* members of the parent. You can use the inherited members as is, replace them, hide them, or supplement them with new members:

- The inherited fields can be used directly, just like any other fields.

- You can declare a field in the subclass with the same name as the one in the superclass, thus *hiding* it (not recommended).
- You can declare new fields in the subclass that are not in the super class.
- The inherited methods can be used directly as they are.
- You can write a new *instance* method in the subclass that has the same signature as the one in the superclass, thus *overriding* it.
- You can write a new *static* method in the subclass that has the same signature as the one in the superclass, thus *hiding* it.
- You can declare new methods in the subclass that are not in the superclass.
- You can write a subclass constructor that invokes the constructor of the super class, either implicitly or by using the keyword super.

Private Members in a Superclass

A subclass does not inherit the private members of its parent class. However, if the superclass has public or protected methods for accessing its private fields, these can also be used by the subclass.

A nested class has access to all the private members of its enclosing class—both fields and methods. Therefore, a public or protected nested class inherited by a subclass has indirect access to all of the private members of the superclass.

Applications of inheritance

There are many different reasons to use the principals of inheritance. Different uses focus on different properties, such as the external behavior of objects, internal structure of the object, structure of the inheritance hierarchy, or software engineering properties of inheritance.

Specialization

One common reason to use inheritance is to create specialized classes out of existing classes. In specialization, the new class or object has data or methods that are not part of the inherited class. For example, a "Bank Account" class might have data for an "account number", "owner", and "balance". An "Interest Bearing Account" class might inherit "Bank Account" and then add data for "interest rate" and "interest accrued" along with behavior for calculating interest earned.

Overriding

A subclass may want to provide a new version of a method in the superclass. In fact, this is often the reason for creating a subclass.

When a subclass method matches in name and in the number and type of arguments to the method in the super-class (that is, the method signatures match), the subclass is said to override that method.

Extension

Another reason to use inheritance is to provide additional data or behavior features. This practice is referred to as extension. Extension is often used when incorporating the new features into the base class is either not possible or not appropriate.

Reusable Code

One of the fundamental reasons for using inheritance is to allow a new class to re-use code which already exists in another class. This practice is referred to as implementation inheritance.

Polymorphism (Outsource: 11-34)

The ability to treat derived class members just like their base class's members is an example of polymorphism.

Polymorphism is the ability of objects belonging to different types to respond to method calls of methods of the same name, each one according to an appropriate type-specific behavior. The programmer (and the program) does not have to know the exact type of the object in advance, so this behavior can be implemented at run time.

The different objects involved only need to present a compatible interface to the clients. That is, there must be public methods with the same name and the same argument lists in all the objects. In principle, the object types may be unrelated, but since they share a common interface, they are often implemented as subclasses of the same parent class. Though it is not required, it is understood that the different methods will also produce similar results.

In practical terms, polymorphism means that if class B inherits from class A, it doesn’t have to inherit everything about class A; it can do some of the things that class A does differently. This means that the same “verb” can result in different actions as appropriate for a specific class, so controlling code can issue the same command to a series of objects and get appropriately different results from each one.

Using the super Keyword (Outsource: 11-33)

The super keyword lets you refer to original methods in a base class from overridden methods in a subclass. Therefore if class B extends class A, and class B overrides some of the methods in class A, class B can use the original versions of the overridden methods, as defined in class A, by prefixing the method name with super.

For example, the following example shows a call to the eat method of the super class from the eat method of the subclass:

```
public class Animal
{
    public void eat() {
        // code not shown
    }
}

public class Dog extends Animal
{
    public void eat() {
        super.eat();            // method call to eat in the superclass
        // other code not shown
    }
}
```

Abstract Classes (Outsource: 9-15, 11-36 - 11-39)

An abstract class is one from which you cannot create any *concrete objects*, but from which you can inherit. You use the keyword abstract when you declare an abstract class.

Abstract classes are like regular classes because they have data and methods. The difference lies in the fact that you cannot create instances of abstract classes by using the new operator. You create abstract classes simply to provide a super class from which other objects may inherit. Usually, abstract classes contain abstract methods. An abstract method is a method with no method statements. When you create an abstract method, you provide the keyword abstract and the intended method type, name, and arguments, but you do not provide any statements within the method. When you create a subclass that inherits an abstract method from a parent, you must provide the actions, or implementation, for the inherited method. In other words, you are required to code a subclass method to override the empty superclass method that is inherited.

For example, suppose you want to create classes to represent different animals, such as Dog and Cat. You could create a generic abstract class named Animal so you can provide generic data fields, such as the animal's name. An Animal is generic, but all Animals can make a sound. The actual sound differs from Animal to Animal. If you create an abstract speak method in the abstract Animal class, then you require all future Animal subclasses to code a speak method. The following example shows an abstract Animal class containing a data field for the name, a constructor, a getName method, and an abstract speak method:

```
public abstract class Animal
{
    private String myName;

    public Animal(String name) {
        myName = name;
}

    public String getName( ) {
        return myName;
    }

    public abstract void speak( );
}
```

The Animal class is declared as abstract. You cannot place a statement such as

```
    Animal dog = new Animal(“Bowser”);
```

within a program because Animal cannot be instantiated.

The Animal class is meant to be extended (inherited from). The Animal parent class contains a constructor that requires a String containing the Animal’s name, so any class that extends the Animal class must contain a constructor that passes a String along to its *superclass* constructor.

For example, we could create a Dog class by extending the Animal class:

```
public class Dog extends Animal
{
    public Dog(String name) {
        super(name);
    }

    public void speak( ) {
        System.out.println(“Woof”);
    }
}
```

The speak method within the Dog class is required because the abstract, parent Animal class contains an abstract speak method. You can code any statements you like within the Dog speak method, but the method must exist. The Dog class is a **concrete class** and can be instantiated.

instanceof Operator

A binary operator that takes an object reference as its first operand and a class or interface as its second operand and produces a boolean result. The instanceof operator evaluates to true if and only if the runtime type of the object is assignment compatible with the class or interface. For example:

```
String s = new String("Hello World");
s instanceof String        // results in true
s instanceof Integer       // results in false
```

Since instanceof is a boolean operator the results can be used as a conditional statement, i.e.

```
if (s instanceof String)
{
    // do something
}
```

This holds true for inheritance. If ClassB extends ClassA then an object of ClassB is an instanceof ClassA.

LAB 19 - ASSIGNMENT

Lab 19A - 70 points

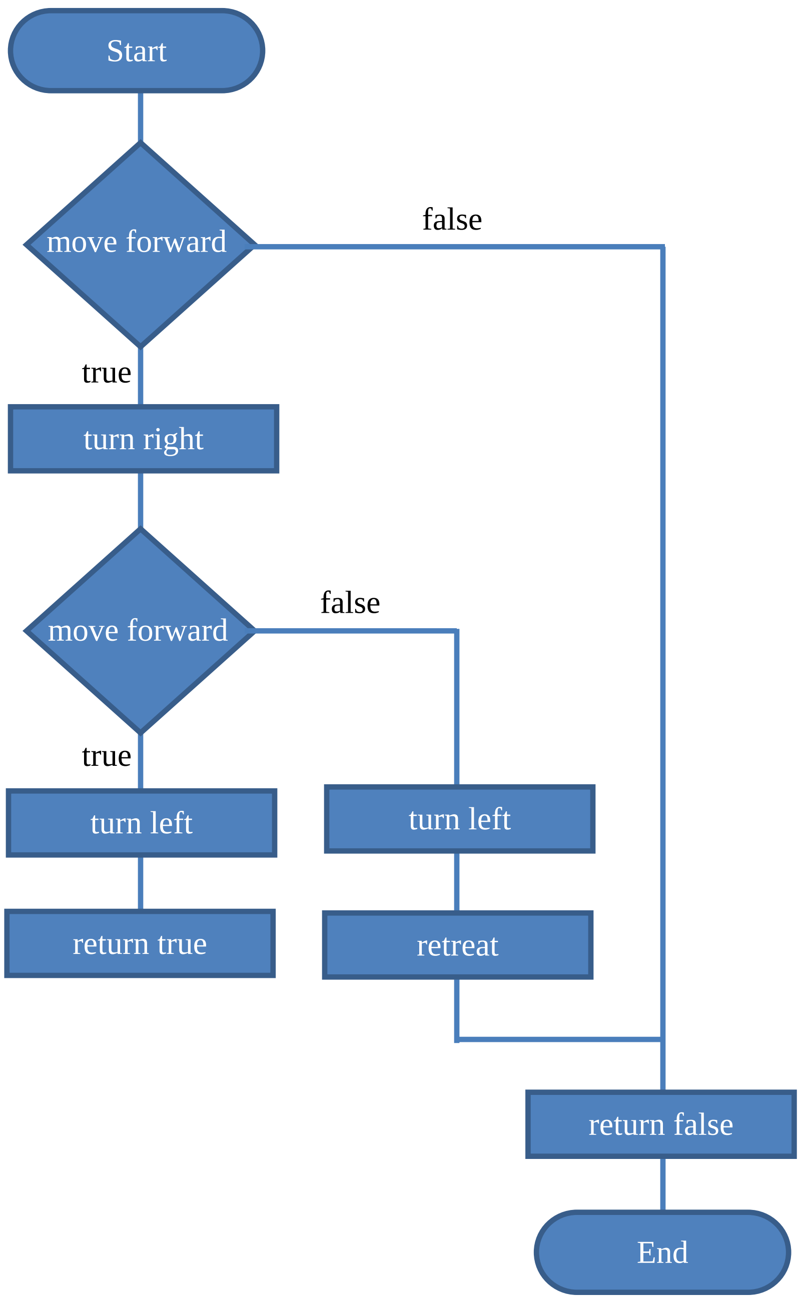
OBJECTIVE

Write a class that extends (inherits from) the FastRobot class. Call this class the ZigZagRobot. The ZigZagRobot will inherit all the methods of the FastRobot (in other words, he can already do everything the FastRobot can do) and in addition will be able to "Zig Zag Left" (move forward one and left one) and "Zig Zag Right" (move forward one and right one). USE THE EXISTING METHODS TO EVERY EXTENT POSSIBLE.

METHOD SUMMARY

- **main** – instantiate an object of your class.
- **constructor** – **Argument List:** a string holding the name of this robot. Pass the name on to the superconstructor.
- **act** – **Argument List:** A string holding the command to be executed. **Return:** true if the command received as an argument is a defined operation for the ZigZagRobot (or any of the Robots he inherits from), false otherwise. Make's the ZigZagRobot respond to any new actions received as an argument.
- **zigLeft** – **Return:** true if the robot was able to complete the move, false otherwise. Makes the ZigZagRobot move forward one space and left one space. The robot remains facing the same direction. If the robot is unable to complete both moves it returns to the original position.
- **zagRight** – **Return:** true if the robot was able to complete the move, false otherwise. Makes the ZigZagRobot move forward one space and right one space. The robot remains facing the same direction. If the robot is unable to complete both moves it returns to the original position.

Flow Chart: **public boolean zigZagRight()**



Lab 19B - 80 points

OBJECTIVE

Write a class that extends (inherits from) the ZigZagRobot class. Call this class the FastZigZagRobot. The FastZigZagRobot will inherit all the methods of the ZigZagRobot (in other words, he can already do everything the ZigZagRobot can do) and in addition will be able to "Zig Left Three" (move forward one and left one) and "Zag Right Three" (move forward one and right one). USE THE EXISTING METHODS TO EVERY EXTENT POSSIBLE.

METHOD SUMMARY

- **main** – instantiate an object of your class.
- **constructor** – **Argument List:** a string holding the name of this robot. Pass the name on to the superconstructor.
- **act** – **Argument List:** A string holding the command to be executed. **Return:** true if the command received as an argument is a defined operation for the FastZigZagRobot (or any of the Robots he inherits from), false otherwise. Make's the FastZigZagRobot respond to any new actions received as an argument.
- **zigLeftThree** – **Return:** the number of times the robot was able to successfully move. Makes the FastZigZagRobot attempt to move diagonally left three space. The robot remains facing the same direction.
- **zagRightThree** – **Return:** the number of times the robot was able to successfully move. Makes the FastZigZagRobot attempt to move diagonally right three space. The robot remains facing the same direction.

Lab 19C - 90 points

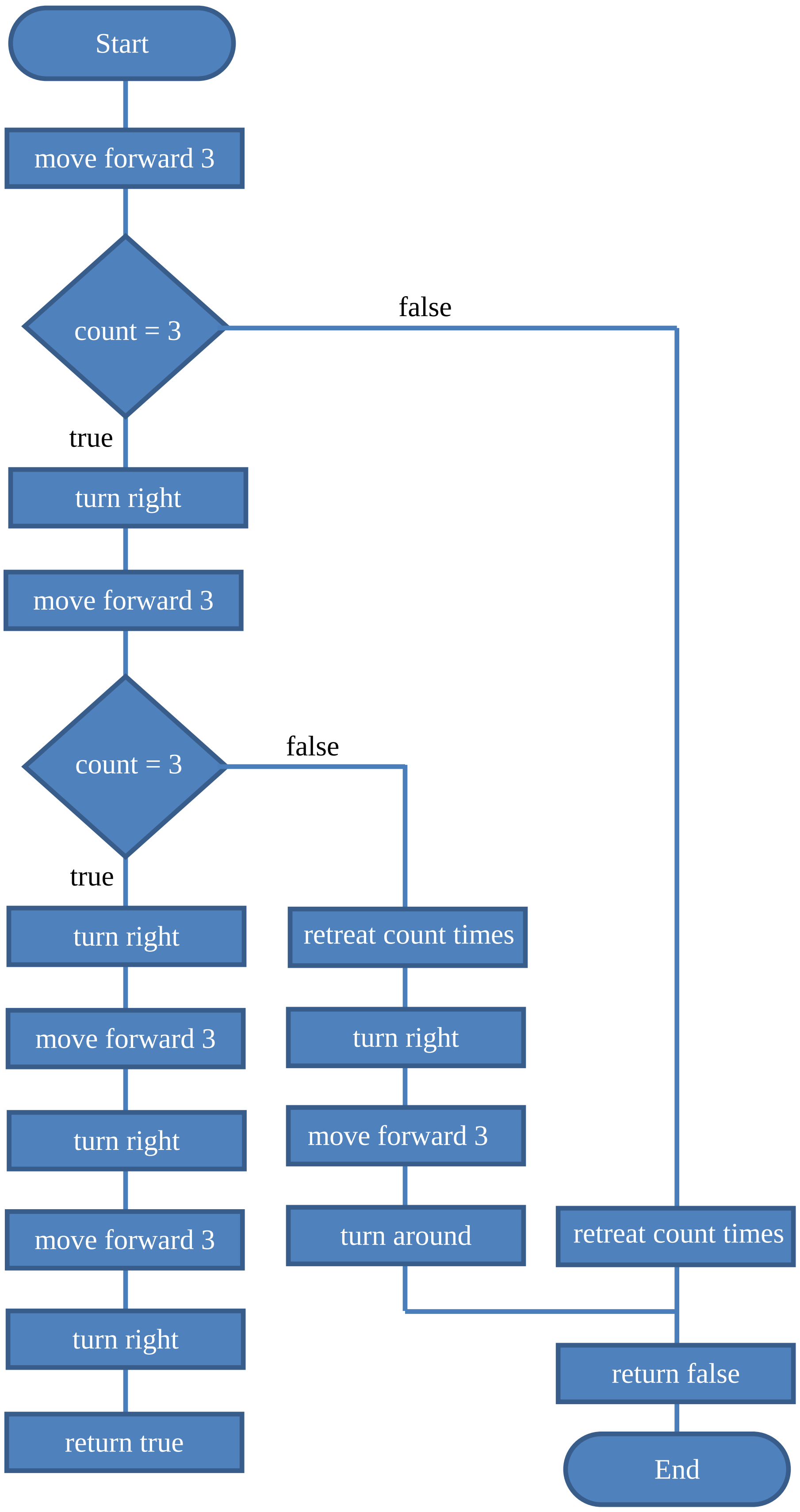
OBJECTIVE

Write a class that inherits from the FastZigZagRobot. Call this class the SquareRobot. The SquareRobot will inherit all the methods of the FastZigZagRobot (in other words, he can already do everything the FastZigZagRobot can do) and in addition will be able to "Left Square" (walk in a left square – three paces forward, turn left, three paces forward, turn left, three paces forward, turn left, three paces forward and turn left) and "Right Square". When a "Left Square" or "Right Square" is completed the robot will be facing the same direction as he was originally facing. If the SquareRobot is unable to complete ALL the moves, he will return to his original location facing his original direction.

METHOD SUMMARY

- **main** – instantiate an object of your class.
- **constructor** – **Argument List:** a string holding the name of this robot. Pass the name on to the superconstructor.
- **act** – **Argument List:** A string holding the command to be executed. **Return:** true if the command received as an argument is a defined operation for the SquareRobot, false otherwise. Make's the SquareRobot respond to any new actions received as an argument.
- **leftSquare** – **Return:** true if the robot was able to complete the move, false otherwise. Makes the SquareRobot move in a left square. A left square is defined as three spaces forward, a left turn, three spaces, a left turn, three spaces, a left turn, three spaces and a left turn. The robot should end up facing the same direction as it started. If the robot is unable to complete the entire movement it should return to the original position facing the original direction.
- **rightSquare** – **Return:** true if the robot was able to complete the move, false otherwise. Makes the SquareRobot move in a right square. A right square is defined as three spaces forward, a right turn, three spaces, a right turn, three spaces, a right turn, three spaces and a right turn. The robot should end up facing the same direction as it started. If the robot is unable to complete the entire movement it should return to the original position facing the original direction.

Flow Chart: **public boolean rightSquare()**



Lab 19D - 100 points

OBJECTIVE

Write a class that inherits from the SquareRobot. Call this class the CluelessRobot. This Robot does everything in reverse. Every time you tell him to "turn right" he will turn left. Every time you tell him to "turn left" he turns right. When you tell the ClulelessRobot to "move forward" he retreats, and so forth. As in the previous Robot programs he should respond that he has or has not been successful in carrying out an order. Since the CluelessRobot does not include any new actions the act method does not need to be overwritten. **The CluelessRobot will NOT have an `act` method.**

METHOD SUMMARY

- **main** – instantiate an object of your class.
- **constructor** – **Argument List:** a string holding the name of this robot. Pass the name on to the superconstructor.
- **moveForward** – **Return:** a boolean true if the robot was able to move, false otherwise. Makes the CluelessRobot back up one space.
- **turnLeft** – Makes the CluelessRobot turn 90 degrees to his right.
- **turnRight** – Makes the CluelessRobot turn 90 degrees to his left.
- **moveForwardThree** – **Return:** an integer indicating the number of moves actually made. Makes the CluelessRobot back up three spaces. After executing this command the CluelessRobot should be facing the same direction. If the CluelessRobot is unable to back up three spaces he should back up as far as he can.
- **retreat** – **Return:** a boolean true if the robot was able to move, false otherwise. Makes the CluelessRobot move forward one space.
- **zigLeft** – **Return:** true if the robot was able to complete the move, false otherwise. Makes the CluelessRobot move forward one space and right one space. The robot remains facing the same direction. If the robot is unable to complete both moves it returns to the original position.
- **zagRight** – **Return:** true if the robot was able to complete the move, false otherwise. Makes the CluelessRobot move forward one space and left one space. The robot remains facing the same direction. If the robot is unable to complete both moves it returns to the original position.
- **zigLeftThree** – **Return:** the number of times the robot was able to successfully move. Makes the FastZigZagRobot attempt to move diagonally RIGHT three space. The robot remains facing the same direction.
- **zagRightThree** – **Return:** the number of times the robot was able to successfully move. Makes the FastZigZagRobot attempt to move diagonally LEFT three space. The robot remains facing the same direction.
- **leftSquare** – **Return:** true if the move is successfully completed. Moves the CluelessRobot in a right square.
- **rightSquare** – **Return:** true if the move is successfully completed. Moves the CluelessRobot in a left square.