Revised: 12/17/2019

Object Oriented Programming

Aggregation and Enumerators

Michael C. Hackett

Computer Science Department

Community College of Philadelphia

Lecture Topics

- Aggregation
- Inner Classes
- Array of Objects
- Enumerators

Colors/Fonts

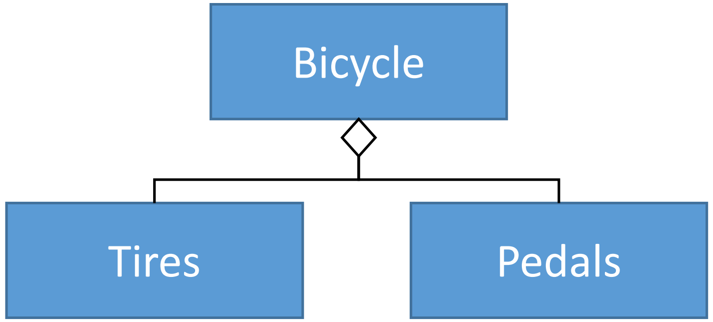
• Local Variable Names	–	Brown
• Primitive data types	–	Fuchsia
• Literals	–	Blue
• Keywords	–	Orange
• Object names	–	Green
• Operators/Punctuation	–	Black
• Field Names	–	Lt Blue
• Method Names	–	Purple
• Parameter Names	–	Gold
• Comments	–	Gray
• Package Names	–	Pink

Source Code	–	**Consolas**
Output	–	`Courier New`

Aggregation

- Real-life objects are often comprised of several other objects.
    - For example, a bicycle is made up of tires, a chain, pedals, handlebars, etc.
        - Together, these smaller, simpler objects are used to create a larger, more complex object.

- A software object can be designed in a similar way, where we have the more complex objects *aggregating* more specific objects into it.



Aggregation: The “has a” Relationship

- In Object Oriented Programming, aggregation is used to create a “has a” relationship among classes and objects.
  - A bicycle “has” tires.
  - A car “has a” steering wheel.
  - A classroom “has a” whiteboard.

- The aggregated objects have attributes and behaviors.
  - The aggregating object incorporates these objects in its own design/functionality.

Aggregation in Object Oriented Design

- There is no special syntax or keywords for object aggregation.
- Aggregation is achieved by using other objects as the fields of the aggregating class.
- For example, a Bicycle class could have two fields, frontTire and backTire.
    - Both of those fields could be Tire objects.

Aggregation in Object Oriented Design

- The below example shows a class for a Tire object.

```
public class Tire {

    private int pressure;
    private int radius;

    public Tire(int pressureIn, int radiusIn) {
        pressure = pressureIn;
        radius = radiusIn;
    }

    ------------//------------ (Accessor and mutator methods)

}
```

Aggregation in Object Oriented Design

- The below example shows a Bicycle class aggregating Tire objects.

```
public class Bicycle {

    ————//———— (Other Fields)
    private Tire frontTire;
    private Tire backTire;

    ————//———— (Other Constructors)

    Bicycle(int gearIn, String colorIn) {
        gear = setGear(gearIn);
        speed = 0;
        color = colorIn;
        frontTire= new Tire(40, 15);
        backTire= new Tire(42, 15);
    }

    ————//———— (Various methods)

}
```

Aggregation in Object Oriented Design

- We can include accessor methods in the Bicycle class to retrieve pressure data from the two Tire fields.

- We could also add accessor methods to retrieve the radius from each Tire field.

```
public class Bicycle {

    // (Other Fields)
    private Tire frontTire;
    private Tire backTire;

    // (Constructors and other methods)
    public int getFrontPressure() {
        return frontTire.getPressure();
    }

    public int getBackPressure() {
        return backTire.getPressure();
    }

}
```

Aggregation in Object Oriented Design

- We can include mutator methods in the Bicycle class to change the pressure data in the two Tire fields.

- We could also add mutator methods to change the radius in each Tire field.

```
public class Bicycle {

    // (Other Fields)
    private Tire frontTire;
    private Tire backTire;

    // (Constructors and other methods)
    public void setFrontPressure(int p) {
        frontTire.setPressure(p);
    }

    public void setBackPressure(int p) {
        backTire.setPressure(p);
    }

}
```

Aggregation in Object Oriented Design

```
public class BicycleTest {

    public static void main(String[] args) {
        Bicycle testBike = new Bicycle();

        System.out.println("Front Pressure: " + testBike.getFrontPressure());
        System.out.println("Back Pressure: " + testBike.getBackPressure());
        testBike.setFrontPressure(45);
        testBike.setBackPressure(46);
        System.out.println("Front Pressure: " + testBike.getFrontPressure());
        System.out.println("Back Pressure: " + testBike.getBackPressure());
    }

}
```

```
Front Pressure: 40
Back Pressure: 42
Front Pressure: 45
Back Pressure: 46
```

Aggregation in Object Oriented Design

- We can incorporate the aggregated Tire objects’ states into the Bicycle’s speedUp method:
  - If the pressure of either Tire is too low, it sets the speed to zero.

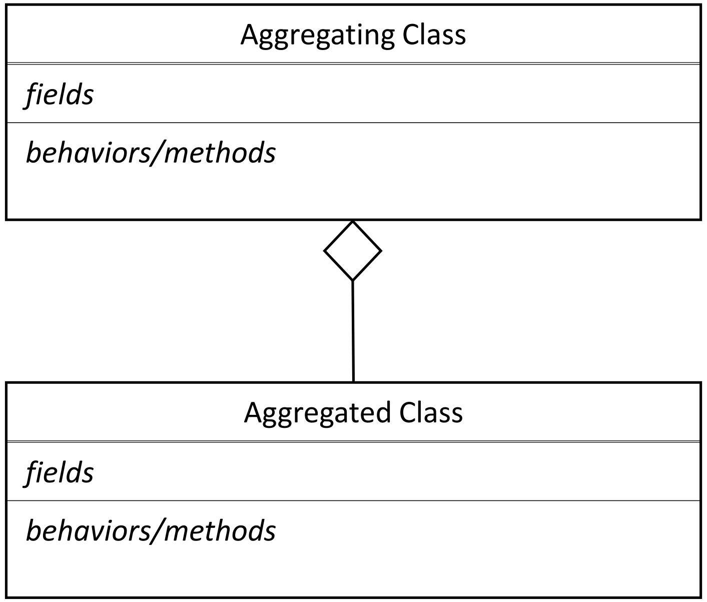
```
public void speedUp() {
    if(frontTire.getPressure() <= 10 || backTire.getPressure() <= 10) {
        speed = 0;
    }
    else {
        speed += 5;
    }
}
```

Aggregation in Object Oriented Design

```java
public class BicycleTest {

    public static void main(String[] args) {
        Bicycle testBike = new Bicycle();

        System.out.println("Front Pressure: " + testBike.getFrontPressure());
        System.out.println("Back Pressure: " + testBike.getBackPressure());
        testBike.speedUp();
        testBike.speedUp();
        System.out.println("Speed: " + testBike.getSpeed());
        testBike.setFrontPressure(5);
        testBike.speedUp();
        System.out.println("Speed: " + testBike.getSpeed());

    }

}
```

```
Front Pressure: 40
Back Pressure: 42
Speed: 10
Speed: 0
```

Aggregation in Class Diagrams



Inner Classes

- An ***inner class*** is a class defined within another class.

- How an inner class is used depends on the program's design.
  - Allows for better encapsulating of data within classes.
  - Can help eliminate redundant code in a class.
  - Can make the class's code easier to maintain.

- Typically, objects made from the inner class are only used in its outer class.
  - Inner classes should be closely related to the function of the outer class.

Inner Classes

- Inner classes are defined within the body of another class.
- Inner classes can contain constructors, fields and methods.
- There is no limit to the number of inner classes that can be defined within a class.

```
class OuterClass {

    class InnerClass1 {

    }

    class InnerClass2 {

    }

    class InnerClass3 {

    }

}
```

Inner Classes

- This example shows a Car class with a GasTank inner class.
- The GasTank class should have setter/getter methods.
  - For brevity, they won’t be included here.
- Inner classes are normally private.

```
public class Car {

    (Car Class Fields, Constructors, and Methods)

    private class GasTank {
        private final int CAPACITY;
        private int fuel;

        public GasTank(int c, int f) {
            CAPACITY = c;
            fuel = f;
        }
    }

}
```

Inner Classes

- The Car class would have a field that is the inner class's type.

- The outer class will incorporate the field in its functionality/design, just as it does any other field.

```
public class Car {

    private GasTank tank;
    private String make;
    private String model;
    private int year;
    private int speed;

                    (Constructors, Methods, Inner Class)
}
```

When Should I Use an Inner Class? Should It Be Public or Private?

- Inner classes are usually small (few fields and methods.)
    - If the inner class is large (lots of code), is revised often and/or is used by other classes, it’s best to make the inner class its own class.

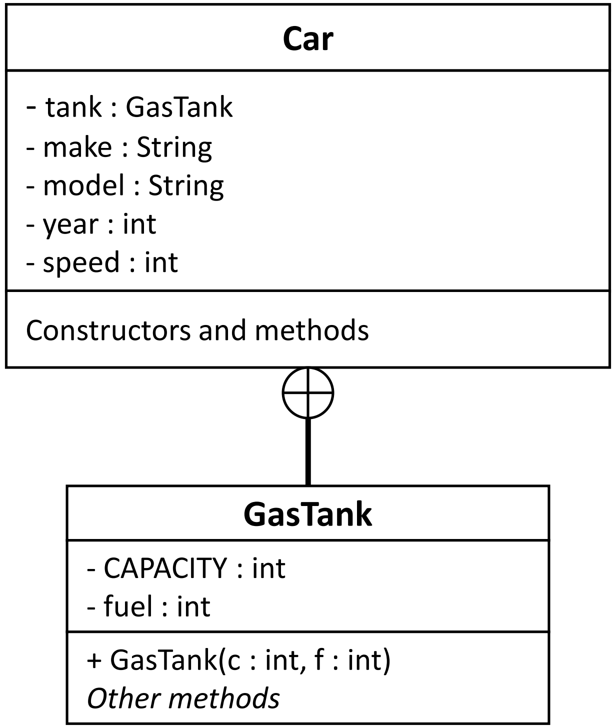
- Inner classes are normally private, so that only the outer class can use objects of that type.
    - Objects of a private inner class type can only be used in its outer class.

Inner Class Access

- Inner classes **can** access the outer class’s fields and methods, regardless of if the field/method is public or private.

- Outer classes **cannot** directly access private fields and methods of an inner class.

Inner Classes in Class Diagrams



Array of Objects

- Arrays can contain references to objects.

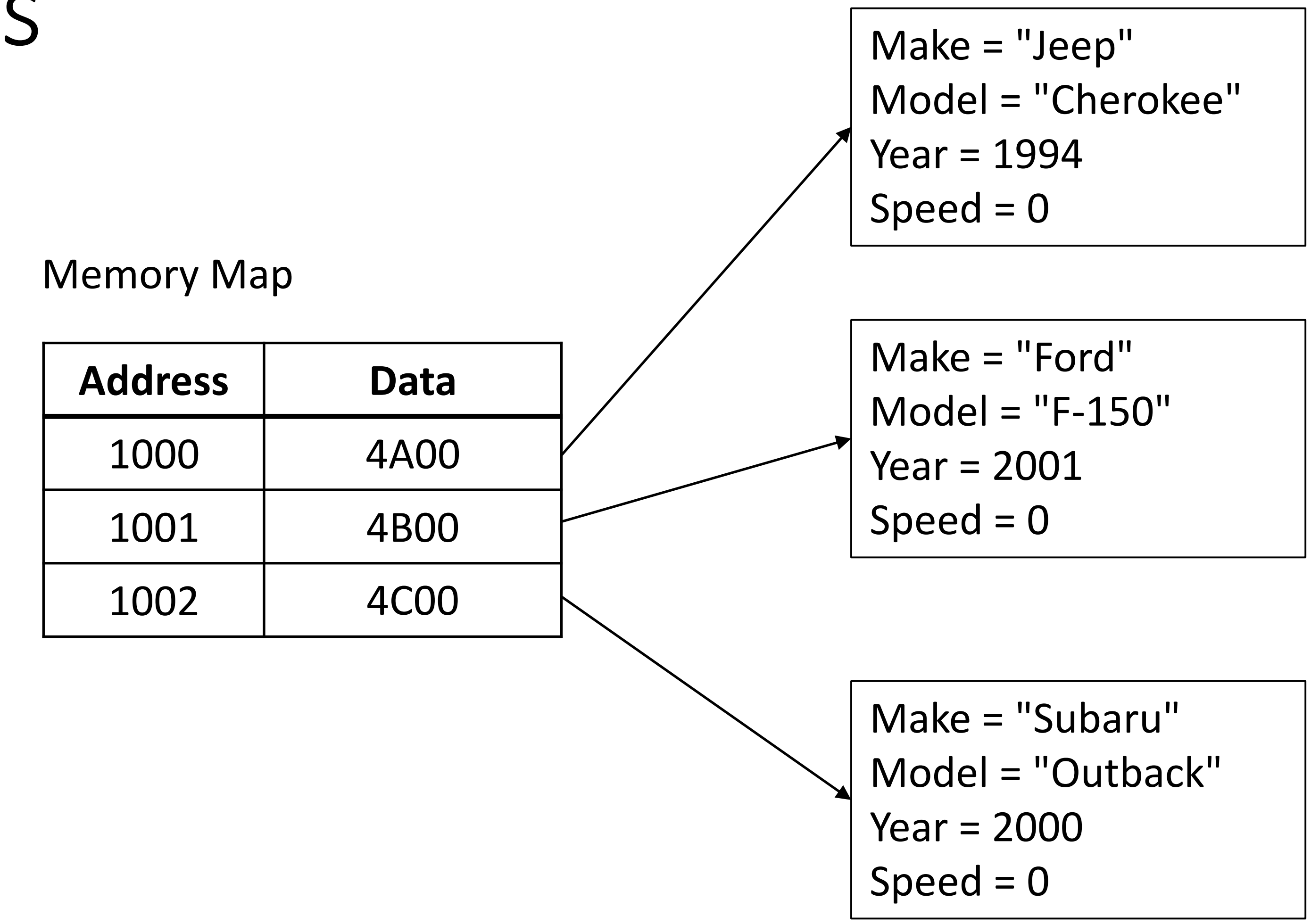
- The statements below create an array of three Car objects.

```
Car[] myCars = new Car[3];

myCars[0] = new Car("Jeep", "Cherokee", 1994);
myCars[1] = new Car("Ford", "F-150", 2001);
myCars[2] = new Car("Subaru", "Outback", 2000);
```

Array of Objects

Symbol Table

Symbol	Address
myCars	1000



Memory Map

Address	Data
1000	4A00
1001	4B00
1002	4C00

Make = "Jeep"
Model = "Cherokee"
Year = 1994
Speed = 0

Make = "Ford"
Model = "F-150"
Year = 2001
Speed = 0

Make = "Subaru"
Model = "Outback"
Year = 2000
Speed = 0

(The memory addresses shown are hypothetical/for illustration purposes.)

Array of Objects

```java
Car[] myCars = new Car[3];

myCars[0] = new Car("Jeep", "Cherokee", 1994);
myCars[1] = new Car("Ford", "F-150", 2001);
myCars[2] = new Car("Subaru", "Outback", 2000);

System.out.println(myCars[1].getMake());
System.out.println(myCars[2].getYear());
System.out.println(myCars[0].getModel());
```

```
Ford
2000
Cherokee
```

Arrays as Method Arguments

- An array can be passed to a method as an argument.

- Must match the array type specified as the parameter.

```
public int sum(int[] numbers)
```

Arrays as Method Arguments

```
public int sum(int[] numbers) {
    int sum = 0;
    for(int number : numbers) {
        sum += number;
    }
    return sum;
}
```

```
int[] threeNums = {4, 5, 6};
sum(threeNums);
```

Would return 15.

Variable Length Arguments

- ***Variable Length Arguments*** (or ***varargs***) allow a method to accept an undetermined number of parameters/arguments.

```
public int sum(int... numbers)
```

- The varargs must all be of the correct type.

- The varargs will be treated as an array inside the method.
  - Varargs *are* arrays, just not declared as such.

Variable Length Arguments

```
public int sum(int... numbers) {
    int sum = 0;
    for(int number : numbers) {
        sum += number;
    }
    return sum;
}
```

```
sum(4, 5, 6);
sum(2, 3);
sum(7, 8.5);
```

Valid. Would return 15.
Valid. Would return 5.
Not valid.

Variable Length Arguments

```
public int sum(int... numbers) {
    int sum = 0;
    for(int number : numbers) {
        sum += number;
    }
    return sum;
}
```

```
int[] myOriginalArray = {3, 5, 7, 9};

sum(myOriginalArray);
```

You can pass an array to a vararg. The sum method would return 24 in this example.

Variable Length Arguments

- No additional parameters can follow a vararg.
  - Would be OK if it were explicitly an array instead of a vararg.

```java
public int doMath(int... numbers, String operationType) {
```

INVALID

- Although, there can be any number of non-vararg parameters preceding it.

```java
public int doMath(String operationType, int... numbers) {
```

VALID

Variable Length Arguments

```
public int doMath(String operationType, int... numbers) {
    int answer = 0;
    if(operationType.equals("+")) {
        for(int number : numbers) {
            answer += number;
        }
    } else if(operationType.equals("*")) {
        answer = 1;
        for(int number : numbers) {
            answer *= number;
        }
    }
    return answer;
}
```

```
doMath("+", 4, 5, 6);
doMath("*", 7, 3);
```

Valid. Would return 15.

Valid. Would return 21.

Variable Length Arguments

```
public int doMath(String operationType, int... numbers) {
    int answer = 0;
    if(operationType.equals("+")) {
        for(int number : numbers) {
            answer += number;
        }
    } else if(operationType.equals("*")) {
        answer = 1;
        for(int number : numbers) {
            answer *= number;
        }
    }
    return answer;
}
```

```
int[] threeNums = {4, 5, 6};
doMath("+", threeNums);
```

Valid. Would return 15.

Returning an Array from a Method

- An array can be returned by a method.
  - Be sure the method’s return type is an array.

```
public int[] getNumbers() {
   int[] threeNums = {4, 5, 6};
   return threeNums;
}
```

Enumerators

- An ***enumerated data type*** consists of a set of predefined values.

- The fields in an enumerated data type are constant.
  - Those fields cannot be changed, new fields cannot be added and fields cannot be removed.

- Enumerators can only hold values that belong to the enumerated data type.

Why use Enumerators?

- The enumerated data type has a specific set of values.
  - No more, no less.

- Say you wanted a data type named Directions and the only values you wanted a variable of that data type to hold were north, south, east, and west.

Declaration

- An enumerator declaration begins with the keyword enum, followed by the name, followed by a comma-separated list of values in braces (similar to an array)

enum Name { one or more constants }

Declaration Example

- The code below shows the enumerator named Directions, as previously described.

```
enum Directions {NORTH, SOUTH, EAST, WEST}
```

No semicolon needed

- Conventions:
  - An enumerated data type normally begins with an uppercase letter, as it is an object.
  - The values are normally in all uppercase (the convention for constants).

Declaration

- Enumerators are **not** declared within any method or constructor.

```
public class Compass {

    public static void main(String[] args) {
        enum Directions {NORTH, SOUTH, EAST, WEST}
    }

}
```

Will not compile

```
public class Compass {

    enum Directions {NORTH, SOUTH, EAST, WEST}

    public static void main(String[] args) {

    }

}
```

Will Compile

Declaring a variable of an Enumerator

- The data type of the enumerator variable is the name of the enumerator.

```
public class Compass {

    enum Directions {NORTH, SOUTH, EAST, WEST}

    public static void main(String[] args) {
        Directions myDirection;
    }

}
```

Initializing a variable of an Enumerator

- The variable’s value can only be one of the valid enumerated constants.

```
public class Compass {

    enum Directions {NORTH, SOUTH, EAST, WEST}

    public static void main(String[] args) {
        Directions myDirection;
        myDirection = Directions.NORTH;
    }

}
```

Could be done in one line:

```
Directions myDirection = Directions.NORTH;
```

Enumerators

- Enumerators are specialized classes.
  - The enumerated data type *is* its own object.
  - Each of the constants are instances of that object.

```
public class Compass {

    enum Directions {NORTH, SOUTH, EAST, WEST}

    public static void main(String[] args) {
        Directions myDirection = Directions.NORTH;
    }

}
```

- Directions is a object, and Directions.NORTH, Directions.SOUTH, Directions.EAST, and Directions.SOUTH are all instances of the Directions object.

Enumerator Methods

- As previously stated, each of the constants are objects.
- They come with methods.
  - toString
  - ordinal
  - compareTo
  - equals

toString Method

- Returns the name of the constant as it was declared.

```
public class Compass {

    enum Directions {NORTH, SOUTH, EAST, WEST}

    public static void main(String[] args) {
        Directions myDirection = Directions.NORTH;
        String directionString = myDirection.toString();
    }

}
```

Returns “NORTH”

ordinal Method

- Returns the (int) position of the constant when was declared.
  - Similar to an array index.

```
public class Compass {

    enum Directions {NORTH, SOUTH, EAST, WEST}

    public static void main(String[] args) {
        Directions myDirection = Directions.NORTH;
        int positionNumber = myDirection.ordinal();
    }

}
```

0	NORTH
1	SOUTH
2	EAST
3	WEST

Returns 0

compareTo Method

- Returns:
  - A negative integer if the object’s ordinal is less than the other object’s ordinal.
  - Zero if the object’s ordinal is equal to the other object’s ordinal.
  - A positive integer if the object’s ordinal is greater than the other object’s ordinal.

- Only works for that enumerated data type.

compareTo Method

```
public class Compass {

    enum Directions {NORTH, SOUTH, EAST, WEST}

    public static void main(String[] args) {
        Directions myDirection = Directions.NORTH;
        if(myDirection.compareTo(Directions.EAST) < 0) {

        }
    }

}
```

- The compareTo method will return **-2**
    - myDirection (Directions.NORTH, ordinal 0) – Directions.EAST (ordinal 2)
        - 0 – 2 = -2

compareTo Method

```
public class Compass {

    enum Directions {NORTH, SOUTH, EAST, WEST}

    public static void main(String[] args) {
        Directions myDirection = Directions.NORTH;
        Directions otherDirection = Directions.EAST;

        if(myDirection.compareTo(otherDirection) < 0) {

        }
    }

}
```

- Same as last slide
- This just illustrates using a variable of the same enumerated data type.
- The compareTo method will return **-2**
  - myDirection (Directions.NORTH, ordinal 0) – otherDirection (Directions.EAST, ordinal 2)
    - 0 – 2 = -2

compareTo Method

```
public class Compass {

    enum Directions {NORTH, SOUTH, EAST, WEST}

    public static void main(String[] args) {
        Directions myDirection = Directions.WEST;
        Directions otherDirection = Directions.WEST;

        if(otherDirection.compareTo(myDirection) == 0) {

        }
    }

}
```

- The compareTo method will return **0**
  - myDirection (Directions.WEST, ordinal 3) – otherDirection (Directions.WEST, ordinal 3)
    - 3 – 3= 0

equals Method

- Returns a boolean.
  - True if they are equal (have the same ordinal)
  - False if they are not equal (do not have the same ordinal)

- Only works for that enumerated data type.

equals Method

```
public class Compass {

    enum Directions {NORTH, SOUTH, EAST, WEST}

    public static void main(String[] args) {
        Directions myDirection = Directions.WEST;

        if(myDirection.equals(Directions.EAST)) {

        }
    }

}
```

- myDirection (Directions.WEST, ordinal 3) is not equal to Directions.EAST (ordinal 2)

equals Method

```
public class Compass {

    enum Directions {NORTH, SOUTH, EAST, WEST}

    public static void main(String[] args) {
        Directions myDirection = Directions.WEST;
        Directions otherDirection = Directions.EAST;

        if(myDirection.equals(otherDirection)) {

        }
    }

}
```

- myDirection (Directions.WEST, ordinal 3) is not equal to otherDirection (Directions.EAST, ordinal 2)
- This just illustrates using a variable of the same enumerated data type.

Switching on an Enumerator

```
public class Compass {

    enum Directions {NORTH, SOUTH, EAST, WEST}

    public static void main(String[] args) {
        Directions myDirection = Directions.WEST;

        switch(myDirection) {
            case NORTH: System.out.println("I'm going north!");
                        break;
            case SOUTH: System.out.println("I'm going south!");
                        break;
            case EAST:  System.out.println("I'm going east!");
                        break;
            case WEST:  System.out.println("I'm going west!");
                        break;
        }
    }

}
```