Demonstration of Encapsulation

```
6
7       private ChoiceType handOne;
8       private ChoiceType handTwo;
9       String winner;
10      int playerOnescore = 0;
11      int playerTwoscore = 0;
12
13      public Game(){
14          handOne = null;
15          handTwo = setUpComputer();
16          winner = null;
17
18      }
19
20
21      //Setting up method for Computer's randomhand
22      private ChoiceType setUpComputer(){
23          int computerHand = new Random().nextInt(ChoiceType.values().length);
24          return ChoiceType.values()[computerHand];
25      }
26
27      //Getters and setters for handTwo and HandOne (namely for tests)
28      public ChoiceType getHandTwo() {
29          return handTwo;
30      }
31
32      public void setHandTwoForTest(ChoiceType choiceType){
33          this.handTwo = choiceType;
34      }
35
36      public ChoiceType getHandOne() {
37          return handOne;
38      }
```

Left: Java class contains two private variables that can only be accessed via public getter and setter method. This allows the class to have complete control over what's stored.

Demonstration of Inheritance

```java
public abstract class Bandit {

    String name;
    int hlthPts;
    ArrayList<Weapon> hand;

    public Bandit(String name, int health, ArrayList hand){
        this.name = name;
        this.hlthPts = health;
        this.hand = new ArrayList<Weapon>();
    }

    abstract public String speak();
    abstract public String attack();
    abstract public String takeDmg();
```

```java
public class BanditKing extends Bandit {

    public BanditKing(String name, int health, ArrayList hand) {
        super(name, health, hand);

    }

    @Override
    public String speak() {
        return "Argh!";
    }
```

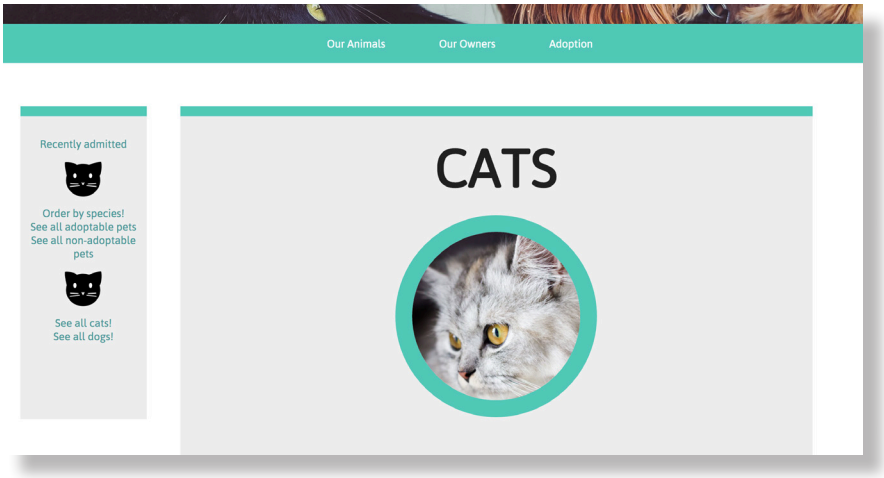
```java
    @Override
    public String attack(){
        Weapon weapon = this.hand.get(0);
        if (getNumberOfItems() == 0){
            return "Alas I am without a weapon, HALP!";
        }
        return "Don't come a step closer or I will hit you with my" + weapon.getName();
    }
```

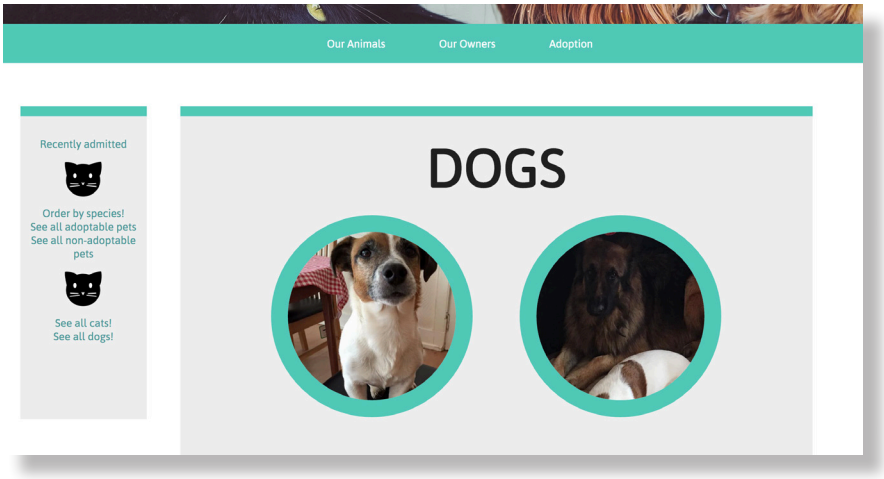
Left: Bandit is an abstract class that contains abstract methods. It allows a programmer to make a binding contract with whatever class extends it. An abstract method is where the method has been declared but there is no implementation or logic written.

Below: BanditKing extends or inherits the abstract class and must implement the abstract methods inside Bandit to work because by inheriting Bandit, it is accepting the contract mentioned above. In this case, it's overriding the abstract methods for both speak() and attack() and returning a string.

Demonstration of searching and sorting data

```ruby
70
71      def self.cats()
72        sql = "SELECT * FROM animals WHERE species = 'cat';"
73        result = SqlRunner.run(sql)
74        return result.map{|options| Animal.new(options)}
75      end
76
77      def self.dogs()
78        sql = "SELECT * FROM animals WHERE species = 'dog';"
79        result = SqlRunner.run(sql)
80        return result.map{|options| Animal.new(options)}
81      end
82
83      def self.order_adopt_status()
84        sql = "SELECT * FROM animals WHERE adoption_status = 'Ready';"
85        result = SqlRunner.run(sql)
86        return result.map{|options| Animal.new(options)}
87      end
```





Left: These functions search and sort data according to species or adoption_status.

Left, Second: Shows the result of the searching sorting functions. In the case of self. cats( ) function, it searches through the database of animals and sorts species tagged as 'cat' so they are the only animals displayed.

Left, Third:
Shows the results of the function self.dogs( ) acts like the cats function as it searches and sorts the animals by only dogs.

Demonstration of an Array, a function that uses it and results

```java
import java.util.ArrayList;

public class Library{

  private String name;
  private ArrayList<Book> stock;

  public Library(String name){
    this.name = name;
    this.stock = new ArrayList<Book>();
  }

  public String getName(){
    return this.name;
  }

  public int stockCount(){
    return stock.size();
  }

  public void addStock(Book book){
    if (this.isFull() == false){
       stock.add(book);
   }
  }
```

```java

  @Test
  public void checkStockIsEmpty(){
    assertEquals(0, mitchell.stockCount());
  }

  @Test
  public void addBookToStock(){
    mitchell.addStock(trainspotting);
    assertEquals(1,mitchell.stockCount());
  }

  @Test
  public void checkIfLibraryGoesOverCapacity(){
    mitchell.addStock(trainspotting);
    mitchell.addStock(crash);
    mitchell.addStock(hobbit);
    mitchell.addStock(fishnet);
    mitchell.addStock(treasureIsland);
    // mitchell.addStock(catInHat);
    assertEquals(true, mitchell.isFull());
  }
```

Left: Library class holds stock with an arraylist of book objects.

Left, Second: Shows a series of tests that uses a method to check the size of the arraylist, a second test that adds a single book to the stock arraylist.

Demonstration of an Hash, a function that uses it and results

```ruby
require('minitest/rg')

class Test_library < Minitest::Test

  def setup

    @bookone = {"title" => "Crash", "author" => "J.G.Ballard"}



  end

  def test_get_title

    assert_equal("Crash", @bookone["title"] )
  end
end
```

Left: A book object contains a hash map of various key and values. In this case it has a key of title and author that contains a value of ‘Crash’ and ‘J.G.Ballard’.

The below function uses the key “title” to get the value stored. Resulting in a passing test.

Demonstration of Polymorphism

```java
public abstract class Bandit {

    String name;
    int hlthPts;
    ArrayList<Weapon> hand;

    public Bandit(String name, int health, ArrayList hand){
        this.name = name;
        this.hlthPts = health;
        this.hand = new ArrayList<Weapon>();
    }

    public abstract String speak();
    public abstract String attack();
    public abstract String takeDmg();

    public String shout(){
        return "I AM A BANDIT!";
    }
```

```java
public class BanditKing extends Bandit {

    public BanditKing(String name, int health, ArrayList hand) {
        super(name, health, hand);

    }

    @Override
    public String shout(){
        return "I AM " + this.name + "and I am the Bandit King!";
    }
```

Left: Inheriting an abstract class is an example of dynamic polymorphism. In the Bandit class we have a method called shout(). This method can be called by any child class without any changes but below we can see that BanditKing class now overrides this methods and changes the behaviour of the superclass.