SKILLSTORM

<https://skillstorm.instructure.com/>

# OOP

Foundation for Java Language. Object is a representation of something in real-life.

## Encapsulation

Take everything about a thing and put it in one place

## Inheritance

A way of reusing classes. Not repeating code

## Abstraction

Create a class that generalize common behavior. Clocks having hour and minute

## Polymorphism

Many forms. Change behavior under different circumstances

### Classes

Blueprints used to define objects; The definition of the state and behavior of an object

* Real world objects
* Application components (code)

Includes:

* Variables (state)
* Methods (behavior)
* Constructors (initialize the state); no return type; used to initialize the state of variables. i.e., speed = 0 for Vehicle class.
* Parameters – value passed through method
* class Vehicle {// class body
* int speed; // declared variable
* void accelerate() {//method body
* }
* Vehicle(){ //Constructor
* speed = 0;
* }
* }
* public class HelloWorld {
* public static void main(String[] args) {
* System.out.println("Hello World");
* }
* }

# Classes and Objects

### Data Types

* Primitive: numeric, text, true/false
* Non-primitive: Any class wanting to be used; create your own
* class Vehicle {// class body
* //Primitive
* int speed = 2000000; // declared variable
* byte wheels = 4;
* short weight = 2000;
* char fuel = 'D';
* boolean running = false;
* long serial = 123456789123456789L;
* float fuelRemaining = 100.1234567890123f;
* String make = "Toyota";
* //Can use own class as data type
* Vehicle toy;
* void accelerate() {//method body
* int count = 0;
* }
* }

### Arrays

* Stores multiple values in a single reference of the same data type
* Sequential block of memory
* Must provide dimensions to declare
* Always an object – always allocated on the heap (like all objects)
* public class Arrays {
* public static void main(String[] args) {
* int[] array = new int[6];
* array[0] = 155;
* array[5] = 77;
* String[] words = {"abc", "def", "jkl"};
* for (int i = 0; i < words.length; i++) {
* System.out.println(words[i] + "");
* }
* }
* }

## Method Declaration and Invocation

* Declaration: create a method: ***Public void go (int x, String y) {}***
* Invocation: call a method: ***object.go(10,”A”);*** ***Public void go (int x) {} object.go(100);***

-Method overloading

Static methods are called without an object. E.g. ***Math.random();***

public class Method {

    Method example = new Method();

    int returned = example.go(); //non-static must use object

    int value = Method.go(5); //static uses class

    public int go(){

        return 11;

    }

    public static int go(int x){

        return x;

    }

}

## Constructors

* Used to initialize an object’s state
* No argument constructor: ex:
* ***Pizza pizza = new Pizza();*** = blank pizza ***Pizza pizza = new Pizza(“Pepperoni”);*** = pepperoni pizza

- constructor overloading

\*Objects are instances of classes.\*

public class Constructors {

    public static void main(String[] args) {

        Pizza pizza = new Pizza();

        pizza.slices = 8;

        Pizza pie = new Pizza(6);

        pie.topping = "pepperoni";

    }

}

class Pizza {

    int slices;

    String topping;

    Pizza() {} // must define no argument constructor to create instance of pizza

    Pizza(int pieces){

        slices = pieces;

    }

    public void eat(){ //each overloaded constructor will have access to this method

        slices -= 1;

    }

}

# Java Virtual Machine

## Compilation and Bytecode

* Java Development Kit (JDK): compiles code
* Java Virtual Machine (JVM): executes code at runtime…reads .class files; and interpret into machine code
* The output of the Java compiler is bytecode, which leads to the security and portability of the Java code. It is a highly developed set of instructions that are designed to be executed by the Java runtime system known as Java Virtual Machine (JVM).

## Memory Management

#### Automatic memory management

* Stack: Used to store Method Calls, Variables declared in methods, Primitive local variables, and Reference Variables cleaned up as method completes
* Heap: Objects, Instance variables, JVM cleans up objects not in use

Both automatically created.

EX: If JVM calls main first and main calls a run method, if the run method calls pizza object, the pizza is stored on the heap. The reference variable is stored on the stack. Call the eat method to effect pizza object. A slice is now gone. Pizza is still on the heap until the run method completes. The pizza no longer on heap because nothing points to it

# Encapsulation

* Wrapping related state/behavior in a single unit…
* Protect class members from undesirable access (private variables with public getters/setters)

## Packages

* Folders that bundle related classes
* Typically, reverse domain name i.e., ***com.example.project***
* Formed in hierarchical
* First line must be package statement
* package com.example.hello; //must be first line of code  
  import com.example.hi.Key;  
  public class Door {  
   Key key; // must import key because it's in a diff package  
   Frame frame; // No need for import since Frame is in same package as Door  
   String string; //String is in java.lang so will never have to import  
  }

## Access Modifiers

* restrict visibility to class members
* can be applied to classes, methods, variables, and constructors
* public: access anywhere in application
* protected: access within package and subclass
* private: access only in class…getters and setters to access variables
* **final: used to prevent inheritance on class level, to prevent overriding on method level, or create constant variable**

## Variable Scopes

* effects Visibility of variable in class
* effects Lifespan of variable

Four scopes:

* Class variables (static)
* Declared with static modifier
* Declared in class body
* No instance needed
* EX: ***static Planet planet;…***inside person class; each person has the same planet

***Person.planet = “Earth”;*** … each subsequent person will have a planet named Earth.

* Instance variables
* Each new object has its own
* Declared in class body
* EX: ***int age;***…each person has an age
* Local variables
* Declared in method body
* Declared as method parameters
* EX: ***public void code(Computer pc) { IDE ide = new IDE(); }***…Computer called pc and IDE object created in method are local variables
* Block-scope variables
* Declared within a block of code
* If statements, loops, try-catch…only accessible in the block
* EX: ***public void jump (boolean b) { If (b) { int height = 2; } }…***height variable only avail in jump method

# Inheritance

* Passing down state and behavior from one class to another…extends
* Super-class/ sub-class
* Parent-child
* Code reusability
* Extensibility to change existing classes

## Extends

package com.example;  
  
public class Farmer {  
 public static void main(String[] args) {  
 Pickle pickle = new Pickle();  
 pickle.grow(); // pickle extends cucumber so has access to grow method  
 }  
}  
  
class Cucumber {  
 double calories;  
  
 public void grow(){}  
}  
  
class Pickle extends Cucumber {  
 public Pickle() {  
 calories = calories \* 0.70; // inherits the properties of Cucumber  
 }  
}

## Variable Shadowing

* Child variable shadows the Parent variable
* EX: ***Parent: int x = 10;*** ***Child: int x = 5;*** super.x = 10 & this.x = 5 when in the child class.
* ***Public void setX (int x) { this.x = x; }***
* package com.example;  
    
  public class Shadow {  
   public static void main(String[] args) {  
   Child child = new Child();  
   int returned = child.getX();  
   System.*out*.println(returned);  
   }  
  }  
    
  class Parent {  
   int x = 10;  
  }  
    
  class Child extends Parent {  
   int x = 5;  
    
   public void setX(int x) {  
   this.x = x;  
   }  
   public int getX() {  
   return super.x; // returns value of parent  
   }  
  }

## Constructor Chaining

* Every class inherits from java.lang.Object
* Without defining a superclass, extends Object is implied
* The first line of a constructor must be a call to super or this **super();**
* Calling a constructor causes a chain of constructors to fire
* Object -> Employee -> Software professional -> Java Developer
* Omits redundant coding
* package com.example;  
    
  public class Chaining {  
   public static void main(String[] args) {  
   JavaDeveloper dev = new JavaDeveloper("Java"); // calls constructor with variable  
   }  
  }  
  class Employee {  
   String name = "Ryan Hunter";  
   public Employee () {  
   System.*out*.println("Employee");  
   }  
  }  
  class JavaDeveloper extends Employee {  
   String language;  
   public JavaDeveloper(){  
   super(); // default...happens without code...prints "Employee"  
   System.*out*.println("JavaDeveloper");  
   }  
   public JavaDeveloper(String language) {  
   this(); // prints "JavaDeveloper"  
   this.language = language; //stores the value of the param into the inst var  
   System.*out*.println(language); //prints "Java"  
   }  
  }

# Abstraction

* Build partial or incomplete classes
* Create foundation for other classes
* Generalize group of shared functionalities into a parent
* Create classes without knowing how they will be implemented in the future
* Abstract classes or interfaces

## Abstract classes

* Partially built classes
* Method defined do not need a body
* May have 0 to many methods
* Class must be declared abstract
* Cannot be instantiated
* package com.example;  
    
  public class Abstraction {  
   public static void main(String[] args) {  
   Shape shape = new Shape(); // cannot instantiate because it is abstract  
   }  
  }  
  abstract class Shape {  
   abstract double area();  
  }  
    
  class Triangle extends Shape {  
   @Override  
   double area() {  
   return 0;  
   }  
  }  
  class Square extends Shape {  
   @Override  
   double area() {  
   return 10;  
   }  
  }

## Interfaces

* 100% abstract, can’t be instantiated
* Define a contract that classes follow
* Usually used to establish API behavior !!!!!
* Interface methods are public abstract
* Classes can implement many interfaces
* Abstract class (what it is), Interface (what it does)
* package com.example;  
    
  public class Interfaces {  
   public static void main(String[] args) {  
   }  
  }  
  interface Drawable {  
   public abstract void draw();  
  }  
  class Rectangle implements Drawable{  
   @Override  
   public void draw() {} //must implement to be Drawable or error occurs  
  }

# Polymorphism

* Many forms
* The ability to be used flexibly
* Covariance and Virtual Method Invocation
* How Java handles overridden methods
* Covariant assignment must pass the IS-A check

***Animal animal = new Dog();***

### Covariant Variable Assignment

package com.example;  
  
public class Store {  
 public static void main(String[] args) {  
 Furniture chair = new Chair(); // reference type is Furniture object type is Chair  
 }  
}  
class Furniture{ }  
class Chair extends Furniture{}  
class Table extends Furniture{}

### Method Overriding

* Method with same signature/name as one in a parent class
* Allows us to change behavior in a specialized class
* Overridden class take precedence over superclass (Virtual Method Invocation)
* Only non-static methods can be overridden
* package com.example;  
    
  public class Override {  
   public static void main(String[] args) {  
   Poppable obj = new Balloon();  
   obj.pop(); // calls the balloon pop method  
   }  
  }  
  class Poppable {  
   public void pop(){  
   System.*out*.println("Pop");  
   }  
  }  
  class Balloon extends Poppable{  
   public void pop() {  
   System.*out*.println("Pop!");  
   }  
  }

### Polymorphism with Arrays

package com.example;  
  
public class Kennel {  
 public static void main(String[] args) {  
 Animal[] pets = new Animal[3];  
 pets[0]= new Dog();  
 pets[1]= new Cat();  
 pets[2]= new Dog();  
 for (Animal a : pets) {  
 a.speak(); // Bark Meow Bark  
 }  
 }  
}

abstract class Animal {  
 abstract void speak();  
}  
class Dog extends Animal{  
 void speak() {  
 System.*out*.println("Bark");  
 }  
}  
class Cat extends Animal{  
 void speak() {  
 System.*out*.println("Meow");  
 }  
}

Programming in Java

# Control Flow Statements

### If and if-else Statements

* tells a program to execute a certain section of code only if a particular test evaluates to **true**.
* The **if-then-else** statement provides a secondary path of execution when an **if** clause evaluates to **false**.
* package com.example;  
    
  public class IfElse {  
   public static void main(String[] args) {  
   *branch*(true);  
   *branch*(false);  
   }  
   static void branch(boolean cond){  
   if(cond){  
   System.*out*.println("The condition is true");  
   } else{  
   System.*out*.println("The condition is false");  
   }  
   }  
  }

### Switch Statement

* The **switch** statement, unlike the **if-then** and **if-then-else** statements, can have a number of possible execution paths.
* package com.example;  
    
  public class Switch {  
   public static void main(String[] args) {  
   *switchCase*("default");  
   }  
   static void switchCase(String val){  
   switch (val){  
   case "A":{  
   System.*out*.println("A shows");  
   break; //without, it goes to next case  
   }case "B":{  
   System.*out*.println("B shows");  
   break;  
   } default:{  
   System.*out*.println("Default shows");  
   break;  
   }  
   }  
   }  
  }

### For loop Statement

* provides coders with a compact way to iterate over a range of values

package com.example;  
  
public class ForLoop {  
 public static void main(String[] args) {  
 *loop*(10); //prints 123456789  
 }  
 static void loop(int max){  
 //counter variable; condition; increment  
 for (int i = 0; i < max; i++) {  
 System.*out*.println(i); //prints until 10<10 false  
 }  
 }  
}

### Enhanced for loop Statement

package com.example;  
  
public class Enhanced {  
 public static void main(String[] args) {  
 *Loop*(new int[]{1,2,3,4,5}); //prints 1,2,3,4,5 vertically  
 }  
 static void Loop(int[] array){  
 for (int i : array){ //for each int in array  
 System.*out*.println(i);  
 }  
 }  
}

### While loop Statement

* Loop that continues to run while condition is true…will run until false
* package com.example;  
    
  public class While {  
   public static void main(String[] args) {  
   *Loop*(); //call method  
   }  
   static void Loop(){ //create method  
   boolean running = *check*(); //running = boolean method  
   while (running){ //while loop creates conditional loop  
   System.*out*.println("running"); //prints "running"  
   running = *check*(); // boolean variable = method's boolean  
    
   }  
   }  
   static boolean check(){ //created the check method  
   if(Math.*random*() < 0.5){ //generates random num 0-1   
   return false; // if num generated is less than 0.5 loop stops  
   } else {  
   return true; // if num is greater than 0.5 loop runs and continues  
   }  
   }  
  }

### Do-while loop Statement

package com.example;  
  
public class DoWhile {  
 public static void main(String[] args) {  
 *loop*(false); //prints "Running" once  
 }  
 static void loop(boolean running){  
 do { //ensures loop happens at least once  
 System.*out*.println("Running");  
 } while (running);  
 }  
}

# Operators

## Assignment Operator

public class Assign {  
 public static void main(String[] args) {  
 boolean bool = false;  
 while (bool == true) { //bool is false so doesn't print  
 System.*out*.println("run");  
 }  
 }  
}

## Mathematical Operator

* (+, -, \*, /, % remainder)
* public class MathOps {  
   public static void main(String[] args) {  
   int add = 2+2;  
   int subtract = 2-2;  
   int divide = 2/2;  
   int multiply = 2\*2;  
   int remainder = 2%2;  
   }

## Relational Operator

* (== equal to, != not equal to, > greater than, >= greater than or equal to, < less than, <= less than or equal to)
* public class Relation {  
   public static void main(String[] args) {  
   boolean eq = 5 == 4;  
   boolean ne = 5 != 4;  
   boolean lt = 5 < 4;  
   boolean le = 5 <= 4;  
   boolean gt = 5 > 4;  
   boolean ge = 5 >= 4;  
   }  
  }

## Logical (Conditional) Operator

* (& AND, && short-circuit AND, | OR, || short-circuit OR)
* public class Logical {  
   public static void main(String[] args) {  
   boolean holiday = false;  
   boolean weekend = true;  
   boolean work = true;  
    
   if (holiday || weekend && !work){  
   System.*out*.println("Day off");  
   }  
   }  
  }

## Increment Operator

* (+ positive, - negates, -- decrement, ++increment, ! inverse)

# Exception Handling

## Checked and Unchecked Exceptions

* JDB finds and fixes bugs
* Exceptions “exceptional event” disrupt the flow of the program during runtime
* Every Exception has a type : IOException, ConnectException, ArithmeticException, NullPointerException
* When an exceptions occurs it has been thrown…all exceptions inherits from Throwable class
* public class Risky {  
   public static void main(String[] args) {  
   */\*\*Unchecked Exceptions\*\*/* //compiles w/ no (red) errors  
   int x = 5/0; //produces ArithmeticException at runtime  
    
   Object obj = null;  
   obj.toString(); //NullPointerException for  
  // calling a method on an object that hasn't been initialized  
   */\*\*Checked Exception\*\*/* //doesn't compile & shows error  
   FileInputStream file = new FileInputStream("file.txt");  
   }  
  }

## Try-Catch Block

public class TryCatch {  
 static FileInputStream *file*;  
  
 public static void main(String[] args) {  
 try {  
 *file* = new FileInputStream("file.txt");  
 } catch (FileNotFoundException e){  
 //recovery procedures  
 //create file then open  
 //log exception  
 }  
 }  
}

## Throws Declaration

public class Declaring {  
 public static void main(String[] args) {  
 *read*(); //handle somewhere else  
 }  
 //declared the exception, and can now compile  
 static void read() throws FileNotFoundException {  
 FileInputStream file = new FileInputStream("file.txt")  
 }  
}

## Throwing Exceptions

public class Parachute {  
 public static void main(String[] args) {  
 *pull*(); //Have to handle checked exception  
 }  
// use throws declaration so anyone who uses method has to handle the exception  
 static void pull() throws DeploymentException{ //uses throws  
 if (Math.*random*() < 0.001) {  
 throw new DeploymentException(); //uses throw inside method  
 }  
 }  
}  
class DeploymentException extends Exception{}

## Multiple Catch Blocks

* exception handler that can handle more than one type of exception. Indicated by its argument
* public class MultiCatch {  
   public static void main(String[] args) {  
   try {  
   *process*();  
   //Specialized ex first --> more generalized (e.g.Exception)  
   } catch (ExecutionException | DataFormatException x){  
   //same code  
   }  
   }  
   static void process() throws DataFormatException, ExecutionException{  
   //code  
   }  
  }

## Finally Block

* Always executes when there’s a **try** block. Even if an exception occurs
* Useful for closing resources like files, networks, db connections
* public class Finally {  
   public static void main(String[] args) {  
   FileInputStream f = null;  
   try {  
   f = new FileInputStream("demo.txt");  
   } catch (FileNotFoundException e) {  
   System.*out*.println("File not found");  
   } finally { // will always execute  
   if (f != null){//f could get NullPointerException  
   try{  
   f.close(); //throws IOException...surround w/ try/catch  
   } catch (IOException e){}  
   }  
   }  
   }  
  }

# Sorting

## Bubble Sort

* Algorithms are pieces of reusable functionality provided by Java
* From Collections class
* Basic sorting algorithm
* Creates sorted array from least to greatest
* Sorts data by comparing adjacent elements
* “Bubbles up” largest numbers 🡪 end of the array
* Big O Notation: algebraic equation that reps time complexity O(N2)
* public class Bubble {  
   public static void main(String[] args) {  
   int[] unsorted= {3, 9, 1, 6, 8, 10};  
   Arrays.*sort*(unsorted);  
   for (int i: unsorted){  
   System.*out*.print(i + " ");  
   }  
   }  
  }

Data Structures

# Object Comparison

* These two methods are used frequently in data structures.
* Every **Object** in Java includes an **equals** and a **hashCode** method. They must be overridden to work properly.
* Turns hashcode into String
* \*\*\*public String toString(){ //method turns params into Strings  
   return name + " " + age;  
  }\*\*\*

### Equals

* An equals method is used to compare equality of two Objects and returns a boolean.
* Java.lang.Object.equals method ***public Boolean equals(Object other)***

public class Equals {  
 public static void main(String[] args) {  
 Pizza one = new Pizza();  
 Pizza two = new Pizza();  
// one = two; // must override hashcode in order to return true for one.equals(two)  
 System.*out*.println(one.equals(two)); //prints true  
 System.*out*.println(one.topping + " " + two.topping); // prints pepperoni pepperoni  
 }  
}  
  
class Pizza{  
 int size = 0;   
 String topping = "pepperoni";  
  
 @java.lang.Override //rc: generate: equals & hashcode  
 public int hashCode() {  
 return Objects.*hash*(size, topping);  
 }  
  
@java.lang.Override  
public boolean equals(Object o) {  
 if (this == o) return true;  
 if (o == null || getClass() != o.getClass()) return false;  
 Pizza pizza = (Pizza) o;  
 return size == pizza.size && topping.equals(pizza.topping);  
}

}

### Hashcode

* used to give consistent integer that represents an object…returns intfina ***public int hashCode()***
* two objects with same hashcode must be equal
* public class Equals {  
   public static void main(String[] args) {  
   Pizza one = new Pizza();  
   one.size = 14;  
   one.topping = "pepperoni";  
   Pizza two = new Pizza();  
   two.size = 14;  
   two.topping = "cheese";  
   //one = two; //makes hashcode equal  
   System.*out*.println(one.hashCode());  
   System.*out*.println(two.hashCode());  
   //hashcodes are diff  
   }  
  }  
  class Pizza{  
   int size = 0; //used twice because two pizzas were created  
   String topping = null; //used twice because two pizzas were created  
    
   @java.lang.Override  
   public boolean equals(Object o) {  
   if (this == o) return true;  
   if (o == null || getClass() != o.getClass()) return false;  
   Pizza pizza = (Pizza) o;  
   return size == pizza.size && topping.equals(pizza.topping);  
   }  
    
   @java.lang.Override  
   public int hashCode() {  
   return Objects.*hash*(size, topping);  
   }  
  }

# List

* ordered and unsorted data structures that store data in a linear fashion, looking at ArrayList and LinkedList from the Java Collections API.
* ordered **Collection** that may contain duplicate elements. There are two general-purpose **List** implementations: **ArrayList** and **LinkedList**….have IS-A relationship with List

### ArrayList

* ArrayList is a part of collection framework, **ordered** and implements the List interface using an array.
* It provides dynamic arrays and is helpful in programs where you need a resizable array.
* Array vs ArrayList:

**An array is a fixed-length data structure.** It serves as a container that holds the constant number of values of the same type.

**ArrayList is a variable-length data structure**. Add to list

public class ArrayA {  
 public static void main(String[] args) {  
 ArrayList<String> list = new ArrayList<>(); //default size is 16, grows 50% in size w/ each added element  
 list.add("abc");  
 list.add("def");  
 System.*out*.println(list.get(1)); //prints def  
 System.*out*.println(list); //prints [abc, def]  
 for(String i: list){  
 System.*out*.print(i); //prints abcdef  
 }  
 }  
}

### LinkedList

* LinkedList is **ordered** by index position, its elements are **doubly-linked** to one another.
* This allows for new methods for adding and removing from the beginning or end, making it an easy choice for implementing a stack or queue.
* Linear series of Nodes – wraps values
* “Linked” by a reference to the next node -- “doubly” – previous and next
* \*\*\*Allows to get by First or Last…
* public class ArrayB {  
   public static void main(String[] args) {  
   LinkedList<Pickles> list = new LinkedList<>();  
   list.add(new Pickles("Dill"));  
   list.add(new Pickles("Kosher"));  
   String f = list.get(1).flavor; //f = "Kosher"  
   list.contains(new Pickles("Dill")); //Saying Is there a Dill Pickle in this List?  
   }  
  }  
    
  class Pickles{  
   String flavor;  
   public Pickles(String flavor) {  
   this.flavor = flavor;  
   }  
  }

# Queue

* for processing in a first-in/first-out and first-in/last-out fashion using Queues and Stacks

### Queue

* a collection for holding elements prior to processing. FIFO
* **FIFO**: push to tail. Pull from head
* Offer (add), Peek, Poll(remove)
* public class Checkout {  
   public static void main(String[] args) {  
   Queue<Person> queue = new PriorityQueue<>(); //Priority puts queue in some type of order  
   queue.offer(new Person()); //adds person/pushes to tail  
   queue.peek();//check for what's at the head  
   System.*out*.println(queue.poll()); //pop the head of the queue  
   System.*out*.println(queue.size()); //prints 0  
   }  
  }  
  class Person{}

### Stack

* A Deque is a double-ended-queue.
* **LIFO**…push to the tail & pop from tail
* Peek, Pop (remove), Push (add)
* public class Casino {  
   public static void main(String[] args) {  
   Stack<Card> stack = new Stack<>();  
   stack.push(new Card("A of Spades")); //adds card to stack  
   stack.push(new Card("K of Diamonds"));//adds card to stack  
   stack.push(new Card("Q of Hearts")); //adds card to stack  
   System.*out*.println(stack.pop().value); //prints (removes!) Q of Hearts   
   System.*out*.println(stack.peek().value); //prints K of Diamonds  
   }  
  }  
    
  class Card{  
   String value;  
   public Card(String value){  
   this.value = value;  
   }  
  }

# Map

* an object that maps unique keys to values. Ex: SSN maps to a U.S citizen
* Java contains three general-purpose **Map** implementations: **HashMap**, **TreeMap**, and **LinkedHashMap**.
* Underneath, HashMap stores “Entry” objects in an array…Hashcode helps calculate the index to store it

### HashMap

* A LinkedList of arrays.
* public class Country {  
   public static void main(String[] args) {  
   //High-performance way of accessing elements based on a key...Search  
   HashMap<String, Citizen> citizens = new HashMap<>();  
   citizens.put("5001", new Citizen("Brian"));  
   citizens.put("4524", new Citizen("Diane"));  
   System.*out*.println(citizens.get("5001").name); //prints Brian  
   citizens.put("5001", new Citizen("Ryan"));  
   System.*out*.println(citizens.get("5001").name); //Overwrites and prints Ryan  
   citizens.size(); //number of key-value  
   System.*out*.println(citizens.size()); //prints 2 because 5001 was overwritten  
   citizens.containsKey("5001"); //true  
   citizens.containsKey("Brian");//false because brian overwritten  
   citizens.entrySet(); //SSN, Citizen  
   citizens.keySet(); //L4 of SSN  
   citizens.values(); //citizens  
   }  
  }  
  class Citizen{  
   String name;  
   public Citizen(String name){  
   this.name = name;  
   }  
  }

# Set

* \*\*Can’t have duplicate elements and adds a stronger contract on the behavior of the **equals** and **hashCode** operations….Stores unique objects
* Java contains three general-purpose **Set** implementations: **HashSet**, **TreeSet**, and **LinkedHashSet**.
* Stores values inside of a HashMap
* \*\*\*Typically used when wanting to do operations on every element in the HashSet…i.e. for loops
* public class Party {  
   public static void main(String[] args) {  
   HashSet<Friend> friends = new HashSet<>();  
   friends.add(new Friend("Dan", 24));  
   friends.add(new Friend("Diana", 41));  
   // hashcode/equals memory location by default  
   friends.add(new Friend("Dan", 24));  
   System.*out*.println(friends);//prints [Dan 24, Diana 41]  
   }  
  }  
  class Friend{  
   String name;  
   int age;  
   Friend(String name, int age){  
   this.name = name;  
   this.age = age;  
   }  
    
   @java.lang.Override  
   public boolean equals(Object o) { //generate equals and hashcode  
   if (this == o) return true;  
   if (o == null || getClass() != o.getClass()) return false;  
   Friend friend = (Friend) o;  
   return age == friend.age && Objects.*equals*(name, friend.name);  
   }  
    
   @java.lang.Override  
   public int hashCode() { //generate equals and hashcode  
   return Objects.*hash*(name, age);  
   }  
    
   public String toString(){ //method turns params into Strings  
   return name + " " + age;  
   }  
    
  }

Relational Databases and SQL

* Data gives us useful information that provides clarity and knowledge
* What you do with the data is what provides the value

# Relational Databases

### Relational Model

# Normalization and Keys

# Querying and Filtering

# DML Statements

# Joins

# DDL Statements