Outline

- Object-Oriented (OO) Programming Review
- Initial Implementation
- Constructors
- Other Standard Behaviors
- Unified Modeling Language (UML)
- Testing

```java
public class Class1 {
    String name;
    int age;

    public Class1(String n) {
        name = n;
    }

    public Class1(String n, int a) {
        this(n);
        age = a;
    }
}
```
• Data are objects
• Objects is a specific instantiation of a class
• Class defines the data components
  • Think of this as a blueprint
  • You can make many objects from a single blueprint
  • Each object can have small variations
• In Python we used classes for thing like trees
  • But we can use them for more complex things
• Unlike Python we can **encapsulate** things in Java
  • Allows us to limit what others can see/use
  • Private items are hidden from others
Example: Simulation of cars driving on a race track

- Vehicle
  - Contain a number of simple data values
    - Speed
    - Fuel level
    - Color
    - Steering direction
    - Location
    - And possibly many more…
  - Can also contain states
    - Brake
    - Accelerate
    - Refuel
    - Etc..
  - We can control these states
    - So someone cannot set the speed directly
      - We can stop a change from 0 to 60 in zero seconds!)
    - Behaviors can validate the data before it is set
Initial Implementation

- Parts of a Class
  - Instance variables... also called fields
    - Different for each object, or instance, of the class
    - Note a type is given for each
    - Each is private... cannot be changed by others!
      - They have been encapsulated

```java
public class Vehicle {
    private String name;
    private double tankCapacity;
    private double fuelLeft;
    private int maxSpeed;

    private double speed;
    private double distanceTravelled;
}
```
Initial Implementation

- Parts of a Class (continue)
  - Behaviors… also called methods or functions
    - Also go in class scope
    - Behaviors work the same for all instances of this class
  - Example: `accelerate`

```java
public void accelerate(int factor) {
    speed += factor;
    if (speed > maxSpeed) {
        speed = maxSpeed;
    }
}
```

- Has access to class variables
- `public` so the outside world can use it
- `void` because it returns nothing
- Validates max speed has not been exceeded
Initial Implementation

- Parts of a Class (continue)
  - Behaviors… also called methods or functions
    - Need outside access to a class variable?
    - You must provide a behavior

```java
public boolean outOfGas() {
    return fuelLeft == 0;
}
```
Initial Implementation

- Parts of a Class (continue)
  - Constructors
    - Create new instances of the class
    - Invoked when using the `new` keyword
    - Example: `Vehicle v = new Vehicle();`
    - It is optional to write a constructor
      - Java will give you a default one
      - It will have no parameters
      - Will leave class variables in a default state
        - Null for objects
        - Zero for numeric types
  - In general you want to write a constructor
    - Except for very simple classes

```java
Person p1 = new Person();
```

- datatype of `p1`
- reference datatype
- create a new person object
- variable name
Initial Implementation

• Parts of a Class (continue)
  • Constructors (continue)
    • Must have the same name as the class
    • No return type…
    • Must be public

```java
public Vehicle(String nameVal, int mpgVal,
                int tankCapacityVal, int maxSpeedVal) {
    name = nameVal;
    mpg = mpgVal;
    tankCapacity = tankCapacityVal;
    maxSpeed = maxSpeedVal;

    fuelLeft = tankCapacity;
    speed = 0;
    distanceTravelled = 0;
}
```

• Create a new instance:

```java
Vehicle v = new Vehicle("Honda", 32, 12, 110);
```
Initial Implementation

- Parts of a Class (continue)
  - Constructors (continue)
    - Java allows multiple constructors

```java
public Vehicle() {
    name = "Default";
    mpg = 30;
    tankCapacity = 15;
    maxSpeed = 100;

    fuelLeft = tankCapacity;
    speed = 0;
    distanceTravelled = 0;
}
```

- It takes no arguments
- We set some default values with this constructor
- But this looks like a lot of effort duplicated...
Initial Implementation

- Parts of a Class (continue)
  - Constructors (continue)
    - Calling a constructor in a constructor…
    - Use this instead of the constructor name…

```java
public Vehicle() {
    this("Default", 30, 15, 100);
}
```

- Better yet we can avoid “magic numbers” here

```java
public static final String DEFAULT_NAME = "Default";
public static final int DEFAULT_MPG = 30;
public static final double DEFAULT_CAPACITY = 15;
public static final int DEFAULT_MAX_SPEED = 100;

public Vehicle() {
    this(DEFAULT_NAME, DEFAULT_MPG, DEFAULT_CAPACITY, DEFAULT_MAX_SPEED);
}
```

- `final` means it cannot be changed!
- `static` tells the compiler not to create a copy for each instance
  - Only one per class since it cannot be changed
Other Standard Behavior

- Just like Python, there are certain behaviors common in all classes
  - We will see later that all classes have these
    - Whether we write them or not...
- The `toString()` method
  - Produces a text representation for printing
  - Has a required form
    - Must be `public String toString() { ... }`
  - For our Vehicle class we could do this:

```java
public String toString() {
    return name + " is going " + speed + " miles per hour and has " +
    fuelLeft + " gallons of gas left";
}
```
Other Standard Behavior

- Another is equality
  - Java has two separate ideas of equality
  - Exact same object under two names ( == )
  - Given by Java
  - Or two distinct objects with same internal state ( equals() )
  - Written as `public boolean equals( Object o ) { ... }`
  - We must write this one!
  - Any object can be passed in
  - We need to decide if it is equivalent to our vehicle

```java
public boolean equals(Object o) {
    if (!(o instanceof Vehicle))
        return false;
    Vehicle ov = (Vehicle)(o);
    return name.equals(ov.name) && maxspeed == ov.maxspeed;
}
```

- Notice we do not have to use all of the class variables!
Unified Modelling Language (UML)

- UML
  - Graphical display of our program
  - Displays classes and their relationships
    - Does so in a standardized way
  - You will get used to seeing this!

```java
+ Vehicle
+DEFAULT_NAME : String = "Default"
+DEFAULT_MPG : int = 30
+DEFAULT_CAPACITY : double = 15
+DEFAULT_MAX_SPEED : int = 100
+REFILL_THRESHOLD : double = 1
-name : String
-mpg : int
-tankCapacity : double
-fuelLeft : double
-speed : int
-maxSpeed : int
-location : int
-random : Random

<<create>> +Vehicle()
<<create>> +Vehicle(name : String, mpg : int, tankCapacity : double, maxSpeed : int)
+travel(minutes : int, road : Road) : void
+getName() : String
+getLocation() : int
+accelerate(factor : int) : void
+decelerate(factor : int) : void
+toString() : String
+equals(other : Object) : boolean
```
Testing

- Our Vehicle class does not do much on its own
  - Only when we have another class to test it
    - The class **TestVehicle** contains example usage
      - Creates some vehicles
      - Runs some functions on them
      - This is the idea behind **unit-testing**
        - A special class to test our class and its behaviors
        - Very important in OO programming

- The code in **CannonballRun** is a more involved simulation

- **To the code!!!**
Questions?