SWEN-601
Software Construction

Objects I: Encapsulation

Activity: Getting Started

If you are working on a different computer than the last time, you will need to clone your SWEN-601 Activity Repository.

1. If necessary, clone your Activity Repository (or pull your latest code) onto the computer that you plan to use today.
2. Make a new package: `activities.session05`

Next Two Weeks

WEEK 03	SUN	MON	TUES	WEDS	THURS	FRI	SAT
QUIZ			Quiz #4		Quiz #5		
LECTURE			Objects I: Encapsulation		Objects II: Identity, Equality, `static`		
HOMEWORK	Hwk 4 Due (**11:30pm**)		*Hwk 5 Assigned*		*Hwk 6 Assigned*	Hwk 5 Due (**11:30pm**)	

WEEK 04	SUN	MON	TUES	WEDS	THURS	FRI	SAT
QUIZ			Quiz #6				
LECTURE			Inheritance & Polymorphism		Practicum 1		
HOMEWORK	Hwk 6 Due (**11:30pm**)		*Hwk 7 Assigned*			Hwk 7 Due (**11:30pm**)	

Java State : Fields

It is often the case that a Java class is based on some real world category of things, like dogs.

Hey, that's a great idea! Let's make a Java class for dogs!

- In Java, variables may be declared outside of the methods in a class.
  - By stylistic convention, this kind of variable is always declared at the top of the class, just below the class declaration.
- The scope of these variables is the entire class, which means that they can be used inside of any of the non-`static` methods in the class (more on this later).
- A variable that belongs to a class is referred to as a ***field***.
  - Fields make up the state (data) of the class.

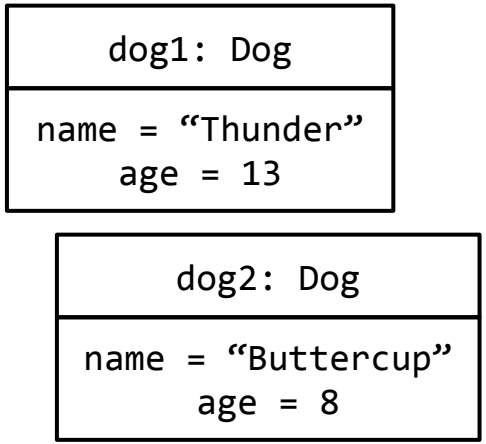
```
public class Wizard {
  String name;
  int age;
}
```

Activity: A Dog Class

1. Create a new class, Dog.
2. Add at least 3 fields to the class. You can add whatever you'd like, but remember, all dogs have similar state. Make sure to choose an appropriate type for each field.
    a. Name
    b. Age
    c. Breed
    d. Weight
    e. etc.

Objects

One useful way to differentiate one instance from another is to use an object diagram.



An object diagram shows the specific values that make up the state of each object

- A Java class is a blueprint or template that defines the state that belongs to a category of things.
- For example, all dogs have:
  - A name
  - A weight
  - An age
  - A hunger status
  - etc.
- But each individual Dog has different values for the state defined by the class Dog.
  - And changing the state for one dog doesn't affect the state of another dog.
- It is therefore necessary that we differentiate between the ***Class***, which defines the state of a Dog, and an ***Object***, which is an individual instance of the Dog class.

The Default Constructor

Type	Default Value
byte	0
short	0
int	0
long	0
float	0
double	0
char	0 ('\u0000')
boolean	false
reference	null

- So, once we have defined a class, how do we create new objects of the class?
- Java provides a default constructor for every class.
  - It takes no parameters.
  - It assigns the default value to each field in the new object.
  - It is called using the new keyword.
    - `Dog bc = new Dog();`

```
Wizard wiz1 = new Wizard();

Wizard wiz2 = new Wizard();
```

The Default Constructor

```
public class Wizard {
    String name;
    int age;

  Wizard() {
    name = null;
    age = 0;
  }

}
```

Java makes a ***default constructor*** "for free." It takes no parameters, is invisible (you won't see it in your class), and sets all fields to default values. You invoke the default constructor when you use `new` with the name of the class to create a new object of the class.

```
Wizard wiz = new Wizard();
```

Again, the default constructor is ***invisible***, but if you could see it, it would look like this.

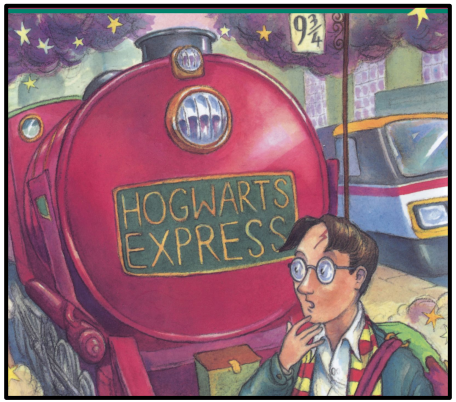
Activity: Making Dogs

1. Add a main method to your Dog class.
2. Create at least two dogs using the default constructor.
3. Print the value of each dog's fields.
    a. You can access the fields using dot notation.

```
Wizard wiz1 = new Wizard();
System.out.println(wiz1.name);
System.out.println(wiz1.age);
```

Initializing Constructors

An initializing constructor gives the caller the ability to specify a value for each and every field in the new object.



In the case of our `Wizard` class, that means providing an argument for `name` and `age`.

- In Java, the programmer may also write their own initializing constructor.
  - Such a constructor defines one parameter for each field in the class.
  - When the constructor is called, an argument is provided for each parameter.
  - The arguments are used to initialize the fields of the new object.

```
Wizard wiz1 = new Wizard("Harry", 17);

Wizard wiz2 = new Wizard("Albus", 88);
```

Initializing Constructors

```
public class Wizard {
    String name;
    int age;

  public Wizard(String n,
                int a) {
    name = n;
    age = a;
  }


}
```

Initializing constructors may be used to initialize fields with values other than the default values at the same time that the object is constructed. Arguments are passed in for the constructor parameters when "`new`" is used with the class to create a new object of the class.

```
Wizard h = new Wizard("Harry", 17);
Wizard a = new Wizard("Albus", 88);
```

Activity: Initializing Dogs

1. Add an initializing constructor to your Dog class.
   a. Remember to declare one parameter for each field.
   b. Use the parameters to initialize the fields.
2. Update your main method to call your constructor to create two different dogs.
3. Print the values of each dog's fields.

```
Wizard wiz1 = new Wizard("Harry", 17);
System.out.println(wiz1.name);
System.out.println(wiz1.age);
```

`this`

- So far, we have used constructor parameter names like “n” for `name` and “a” for age.
  - But stylistic convention dictates that the parameters be named the same as the fields that they are used to initialize (self documenting code).
- This poses a problem: how does Java tell the difference between the parameter and the field if they both have the same name?
  - By default, a variable name always refers to the variable that is closest in scope.
  - “Closest in scope” means the variable that has been declared most recently.
  - In a method, this would be a parameter or a local variable (not the field with the same name).
- However, Java provides a keyword that an object can use to refer to ***itself***: `this`.
  - Using `this.variableName` always refers to the ***field*** with the specified name, even if another variable is closer in scope.
  - This is called ***disambiguation*** (it eliminates ambiguity).

this

Use of the `this` keyword ***always*** insures that we are referring to the ***field*** with the specified name.

Using the identifier by itself refers to the variable that is closest in scope. This may be a ***parameter***, a ***local variable***, or the ***field***.

```
this.variableName = variableName;
```

Using the `this` keyword allows an object to refer to itself. It has several different uses, some of which we will discuss presently.

Activity: Using `this`

1. Update your initializing constructor so that the parameters are named the same as the fields that they are used to initialize.
   a. Use the `this` keyword to disambiguate.

```
Wizard(String name, int age) {
  this.name = name;
  this.age = age;
}
```

Parameterless Constructors

The programmer may choose to leave the body of a parameterless constructor empty.

In this case, the fields will be initialized to Java's default values (i.e. `0` for numbers, `false` for booleans, `null` for references).

In other words, a parameterless constructor with an empty body works exactly like the default constructor provided by Java.

- Once the programmer has defined their own constructor, Java no longer provides a default constructor.
  - Meaning that `Wizard w = new Wizard();` causes a syntax error.
- But what if the programmer would still like callers to be able to construct a new instance of the class without needing to specify any parameters?
  - They write a parameterless constructor.
  - Such a constructor takes no parameters and sets the fields in the class to default values chosen by the programmer.

Parameterless Constructors

```
public class Wizard {
    String name;
    int age;

    public Wizard() {
      name = “Ron”;
      age = 21;
    }

}
```

If you find it necessary, you can make your own ***parameterless constructor*** that takes no parameters and sets fields to whatever values you think are appropriate defaults.

```
Wizard wiz = new Wizard();
```

Activity: Parameterless Constructor

1. Add a parameterless constructor to your Dog class. Set the value of each field to whatever you think is appropriate.

```
Wizard() {
  name = “Hermione”;
  age = 39;
}
```

Chaining Constructors

Recall that the `this` keyword is used by an object to refer to itself.

When one constructor calls another in the same class, the `this` keyword is used rather than the function name.

```
Con(int x) {
  this.x = x;
}

Con() {
  this(5);
}
```

- A class may include any number of constructors provided that the signature of each is different from the others.
  - A parameterless constructor.
  - An initializing constructor.
  - A constructor that defines parameters for a subset of the fields in the class and uses default values for the others.
- Writing lots of constructors may result in lots of duplicate code unless you call one constructor from another.
  - This is referred to as ***constructor chaining***.
- For example, a parameterless constructor may call the initializing constructor.

Activity: Chaining Constructors

1. Add a constructor that takes at least one parameter and calls another constructor with default values for the remaining parameters.

```
Wizard(String name) {
  this(name, 17);
}
```

Every object gets its own copy of the fields and methods that are defined by its class.

Therefore, calling a method on one object of the class does not affect other objects of the same class.

Java Behavior : Methods

- We will use Java classes to ***encapsulate*** both state and behavior by writing functions inside of the class.
  - A function that is a member of a class is called a ***method***.
  - Methods in the class may use the fields in the class.
  - Because the methods are called on the object itself, there is no reason to pass a reference to the object into the method.

Activity: Dog Behavior

1. Add at least two methods to your dog class. Each method should use or modify the state of the dog.
2. Do ***not*** make the methods `static`.

```
public class Wizard {
  void birthday() {
    age = age + 1;
    System.out.println("Happy Birthday," +
      name + "! You turned " + age +
      " years old!");
  }
}
```

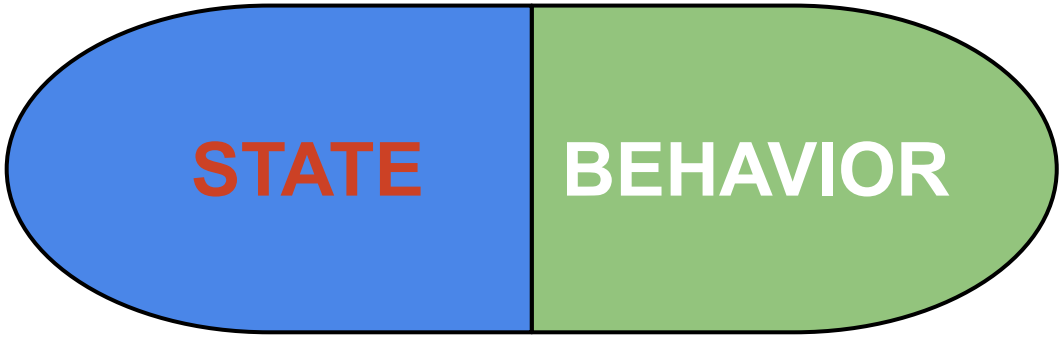
Encapsulation

It is usually the case that the fields in an object should not be modified willy-nilly by any part of the program at any time.

For this reason, all state and behavior in an object generally starts as `private` (visible only to the object itself).

Access can always be changed if and when it is necessary to do so.

Making good choices about data privacy is important and will be a component of your style grade from now on.



- Encapsulation refers to the fact that an object acts like a container, holding its ***state*** and ***behavior*** together in one place.
  - Every object has its own copy of the ***non-static*** state and behavior defined by its class.
- But there is another aspect to encapsulation as well: an object can protect access to its fields and methods using access modifiers.
  - This is referred to as ***data privacy***, and it is an important part of object oriented programming.

Access Modifiers

- An access modifier is a keyword that is placed before a class, method, or field declaration and determines its visibility.
  - e.g. `private int age;`
  - ***Visibility*** refers to which other parts of the program can access or modify the class, method, or field.

If no access modifier is specified, the default access level is ***package-private***.

Visibility	`public`	`protected`	*package private*	`private`
Objects of the Same Class	✓	✓	✓	✓
Other Classes in the Same Package	✓	✓	✓	
Child Classes*	✓	✓		
The Entire Program	✓			

Activity: Data Privacy

1. Add access modifiers to every field and method in your class.
   a. Make fields `private`.
   b. Make methods `public`.

The programmer may choose to provide an accessor but not a mutator or vice versa.

In this way, they have finely grained control over access to the fields in the class.

Accessors & Mutators

- Stylistic convention (and therefore your grade) requires that fields are always `private`. This means that they cannot be directly accessed from outside of the class.
- But what if the programmer wants other parts of the program to be able to see or modify some of the fields in the class? They write special methods to provide access to those fields.
  - An accessor is a method that returns the current value of the field.
    - These methods are also called "*getters*."
  - A mutator is a method that uses a parameter to change the value of a field. These methods are also called "*setters*."

Accessors

- Defined for each instance variable that the programmer *wants* to make visible outside of the object.
  - Allows another part of the program to ask the object "*what is the current value of this field?*"
- Named with the "`get`" prefix and the variable name, e.g. `getName`.

By providing only an accessor, the programmer can make a field **read only**.

Other parts of the program can see the value of the field, but cannot change it.

```
public class Wizard {
    private String name;
    private int age;

    public String getName() {
      return name;
    }

}
```

Mutators

- Defined for each instance variable that the programmer *wants* to make modifiable from outside of the object.
  - Allows another part of the program to tell the object to "*change this variable to a name value.*"
- Named with the "set" prefix and the variable name, e.g. setAge.

By providing only a mutator, the programmer can make a field **write only**.

Other parts of the program can change the value of the field, but cannot see the current value.

While less commonly used than read only access, write only access can be useful in some cases.

```
public class Wizard {
    private String namey
    private int age;

    public void setAge(int age) {
      this.age = age;
    }

}
```

Activity: Accessors and Mutators

1. Write at least one accessor and mutator for the fields in your Dog.
    a. Think about whether or not it makes sense for another part of the program to see or change the value of the field.

Special Methods: `toString`

The `toString` method is automatically invoked in a number of circumstances.

For example, when concatenating an object onto a string with the + operator, the `toString` method is called on the object.

Similarly, when using `System.out.println` to print an object, the object's `toString` method is called.

Because of this, it is generally a good idea to write a good `toString` method for each your classes.

- In Java, the responsibility for translating a value into a string (i.e. for printing) belongs to the object itself using its `toString` method.
- Every class in Java has a default `toString()` method.
  - The string that it returns is not very useful, e.g. `Wizard@1540e19d`.
- It is up to the developer to ***override*** the method and provide a more useful implementation.

```
@Override
public String toString() {
  return "Wizard[name=" + name +
         ", age=" + age + "]";
}
```

Activity: `toString`

1. Write a toString method for your Dog class. Use whatever format you think is good, but include the value of all of the fields.
2. Update your `main` method to print your Dog instances rather than the individual fields.
    a. Remember, the `toString` method will be called automatically on your Dog instances when you use `System.out.println`.