SWEN-601
Software Construction

Objects II: Identity, Equality, & `static`

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**)	

Activity: A Car Class

1. Let's begin by creating a new class to represent a `Car`. Start by defining the ***state*** that all cars have, including:
   a. Make, e.g. `"Dodge"`
   b. Model, e.g. `"Dart"`
   c. Year, e.g. `2013`
   d. A Color, e.g. `"red"`
   e. Vehicle Identification Number (VIN), e.g. `"1234ABCD5"`.
2. Write an *initializing constructor* for your `Car` class.
3. Add accessors ("getters") for all of the fields.
4. For which fields might you add a mutator ("setter")?
5. Finally, *override* the default `toString()` method to return a more useful description of your `Car`.
6. Write a `main` method to test your `Car`.

Equality

Whether you are concerned with shallow equality, which simply asks “do I have two distinct individuals”...

...or “deep equality” which asks whether or not important, internal attributes are the same depends on the problem that you are trying to solve.

- Consider a pair of identical twins. If we were to compare one twin to the other, would we consider them to be equal?
  - The answer is that it depends on what we mean by *“equality.”*
- At the surface level, it’s easy to see that we have two individuals, each with a unique identity..
  - If we look, for example, at social security numbers or some other unique identifier, they won’t match.
  - This is called *shallow equality*, because we look at surface details and no *deeper.*
- But if we compare more deeply, we will find that they have the same hair color, eye color, blood type, even DNA.
  - This is called *“deep equality,”* because we look deeper to compare the internal details of each individual to the other.

Shallow Equality

- The == operator compares only the value stored in memory for each identifier.
  - For primitive types, this is the primitive value. Given the tables to the right:
    - `x == y` will be `true`.
      - The value in x's location in memory is `57`.
      - The value in y's is also `57`.
  - For references, this is the address of the ***object*** to which they refer.
    - `wiz1 == wiz2` will be `false`.
      - The value in `wiz1`'s location in memory is the address `0x200`.
      - The value in `wiz2`'s is `0x500`.
- This is "shallow equality" because it only looks at the value (stored in the variable's address) and no deeper.
  - This is fine for primitive values.
  - For objects, the state ***inside*** of the objects is not evaluated at all.

Variable Table

identifier	type	address
wiz1	Wizard	0x200
x	int	0x300
wiz2	Wizard	0x500
y	int	0x600

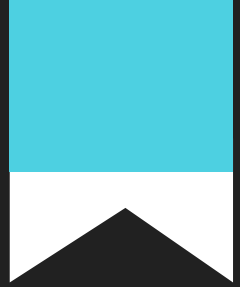
Memory

address	value
0x100	
0x200	0x100
0x300	57
0x400	
0x500	0x400
0x600	57

But what if the author of the class decides that two `Wizards` with the same `name` and `age` should be considered equal to each other?

```
Wizard (0x100)
name = "Harry"
age = 17
```

```
Wizard (0x400)
name = "Harry"
age = 17
```



Activity: Shallow Equality With Primitive Types

x == y?

Q: Did you find any results surprising?

1. Create a new class to experiment with shallow equality using `==`. You may name it whatever you'd like.
2. Add a `main` method. Demonstrate that using `==` to compare two primitive values evaluates to `true` if the values are the same.
    a. Compare variables to literals.
    b. Compare different variables to each other.
    c. Show that this is true even when the types are different, e.g. comparing a `float` with an `int` of the same value.

What's Going On?

```
public class EqualityTester {

  public static void main(String[] args) {

    int w = 5;
    float x = 5.0;
    int y = w;
    int z = 4;

    System.out.printn("w == w " + (w == w));
    System.out.printn("w == 5 " + (w == 5));
    System.out.printn("w == x " + (w == x));
    System.out.printn("w == y " + (w == y));
    System.out.printn("w == z " + (w == z));
  }
}
```

As before, if we build a **variable table** and a **memory table**, we can much more easily visualize what is happening...

Variable Table

identifier	type	address
w	int	0x100
x	int	0x200
y	float	0x300
z	int	0x400

Memory

address	value
0x100	5
0x200	5.0
0x300	5
0x400	4

Remember that a variable is just a ***name*** for a l***ocation in memory***...

The ***value*** for each variable is stored in the variable's location in memory.

The `==` operator compares these values in memory and evaluates to `true` if they are the same, and `false` otherwise.

Activity: String Equality I

1. Write a new class, `StringTester`. Add a `main` method.
2. Assign 3 variables to string literals. Two of the literals should include exactly the same characters.
   a. `String x = “Buttercup”;`
   b. `String y = “Buttercup”;`
   c. `String z = “Thunder”;`
3. Show the result of using the == operator to compare the variables to each other.

`“ABC” == “ABC”?`

Q: Did you find any results surprising?

What is Going On?

```
String x = “Buttercup”;
String y = “Buttercup”;
String z = “Thunder”;

System.out.println(“x == x ” + (x == x));
System.out.println(“x == y ” + (x == y));
System.out.println(“x == z ” + (x == z));
```

Your program should have produced output that looks something like this….

```
$>java StringTester
x == x true
x == y true
x == z false
```

Variable Table

identifier	type	address
x	String	0x300
y	String	0x400
z	String	0x500

Memory

address	value
0x100	“Buttercup”
0x200	“Thunder”
0x300	0x100
0x400	0x100
0x500	0x200

The JVM optimizes the use of string literals to avoid wasting memory.

The ***first*** time it sees a new String literal, it stores it in memory.

Any other references to the same literal value use the address of the copy already stored in memory.

So two variables that refer to the same literal (e.g. x and y in this example), are references to the same address in memory.

The == operator compares the values in memory (the addresses) and is `true` if they are the same, `false` otherwise.

Activity: String Equality II

1. In your `StringTester` main, use a `Scanner` and prompt the user to enter two strings.
2. Print the result of comparing the two strings using the `==` operator.
3. Test your program to see what happens if you type the same `String` in twice.

“ABC” == “ABC”?

Q: Did you find any results surprising?

What is Going On?

```
System.out.print("Enter String a: ");
String a = scanner.nextLine();
System.out.print("Enter String b: ");
String b = scanner.nextLine();

System.out.println("a == b " + (a == b));
```

Your program should have produced output that looks something like this….

```
$>java StringTester
Enter String a: Buttercup
Enter String b: Thunder
a == b false

$>java StringTester
Enter String a: Buttercup
Enter String b: Buttercup
a == b false
```

Q: But why?!

Variable Table

identifier	type	address
a	String	0x300
b	String	0x400

Memory

address	value
0x100	"Buttercup"
0x200	"Buttercup"
0x300	0x100
0x400	0x200

The literal pool ***only*** works with ***literals***. Strings created dynamically aren't included.

So even though the user enters the ***same*** characters twice, two ***different*** strings are created and stored in memory.

So the two variables are referring to ***different*** addresses in memory.

And because the == operator compares the values stored in memory, it will evaluate to `false`.

Activity: Shallow Equality With Reference Types

1. In your `main` method, create two `Cars` with the same values (make, model, VIN, etc.).
   a. Use the `toString()` method that you wrote to show that the cars have identical values for all fields.
2. Show that using == to compare the two `Cars` will evaluate to false.
3. Create a third variable that refers to one of the same two cars. How does == behave with this new variable? Why?

Special Methods: `equals(Object)`

- Every class in Java has an equals(Object) method that can be used to compare one object to another.
- The default implementation works exactly the same as the == operator; it only returns `true` if an object is compared to ***itself*** (it compares identity, i.e. the object's address).
- The author of a class may choose to write their own implementation, typically by following this pattern:
  - Since any type of object may be passed as an argument to the `equals(Object)` method, you must first check the make sure that the object is an ***instance of*** the same class.
  - Java is a statically typed language: before you can access the state and behavior in the object, you must ***cast*** it into a variable of the correct type.
  - Compare each of the <u>important</u> fields in both instances to see if they match.
    - Note that if the field is a reference type (like a `String`), use the `equals` method (not ==).
- This kind of comparison is called *"deep equality."*

```
@Override
public boolean equals(Object o) {
  if(o instanceof Dog) {
    Dog dog = (Dog)o;

    return dog.age == this.age &&
        dog.name.equals(this.name);
  } else {
    return false;
  }
}
```

Remember, because `Strings` are references, we want to make sure to use `equals` and not == to compare the names of the `Dogs`!

Activity: Deep Equality

1. Add an `equals(Object)` method to your `Car` class.
2. It should return true if two *different* objects are compared that represent the same Car.
   a. What fields will you compare?
3. Update the `main` method in your test class to compare the two cars you created and show that they are equal according to your method.
4. Create a third car with different values for make, model, VIN, etc. and show that it is not equal to the first two.

Reference Type vs. Actual Type

- The `java.lang.Object` type is very special in Java.
- Every other reference type (e.g. `String`, `Dog`, `Wizard`, `Car`) can be assigned to a variable with a type of `Object`.

```
Wizard wiz1 = new Wizard("Harry", 17);
Wizard wiz2 = new Wizard("Harry", 17);
Object obj = wiz2;
```

- Remember that the type of the parameter to the equals method ***is*** `Object`.
  - This means that, when a reference is passed into the method, it is assigned to a variable (the parameter) of type Object.
- The reference type determines which state and behavior you can access!

Variable Table

identifier	type	address
wiz1	Wizard	0x200
wiz2	Wizard	0x300
obj	Object	0x500

Memory

address	value
0x100	
0x200	0x100
0x300	0x400
0x400	
0x500	0x400

In this example, the ***actual*** type of the object that `obj` refers to is a `Wizard`. But the ***reference*** type is `Object`. So `Wizard` state/behavior is not accessible through `obj`.

`Wizard (0x100)`

`name = "Harry"`
`age = 17`

`Wizard (0x400)`

`name = "Harry"`
`age = 17`

Reference Type vs. Actual Type

The ***reference type*** is the type declared along with the variable.

```
Wizard wiz1 = new Wizard("Harry", 17);
Wizard wiz2 = new Wizard("Harry", 17);
Object obj =  new Wizard("Ron", 39);

// this works
System.out.println(wiz2.getName());

// this is a compiler error
System.out.println(obj.getName());
```

The ***actual type*** is specified when `new` is used to create the new object.

The ***reference type*** determines which state and behavior can be used, regardless of what the ***actual type*** is.

So because `Object` does not have a `getName()` method, you can't call it using `obj` (even though the actual type is a `Wizard`)!

static

`static` methods are called directly on a class, without needing to first create an instance of the class.

For example:

```
SomeClass.staticMethod()
```

Similarly, `static` fields belong to the class and can be used without an instance.

For example:

```
SomeClass.STATIC_FIELD
```

`static` fields are often used as global variables, and therefore should be used with care.

- Before this week, all of the methods that you wrote included the `static` keyword in the method declaration.
- This week, none of the methods we wrote were marked as `static` (except for `main`). So what's the difference?
- All of the methods that we wrote this week belong to an ***object*** of the class.
  - Each uses or changes the fields that belong to one specific instance of the class.
  - This means that we need to ***create*** an instance of the class (along with its fields) before we can call the methods.
    - We can't get the color of a specific `Car` if we don't have a `Car` in the first place!
- A `static` member belongs to the ***class*** itself (not an object).
  - There is ***one*** copy of the method or field.
  - It is important to note that each individual object ***does not*** get its own copy of a `static` field; changes made to a `static` field affect the one and only copy of that field.
- `static` methods exist in a `static` context, and can only access or use other methods or fields that are also `static`.
  - Trying to use non-`static` methods or fields will cause a syntax error.

Activity: `static` Fields & Methods

We'd like to keep a count of all of the Cars that get created.

1. Add a `static` field to your `Car` class called *CAR_COUNT* and set it to a value of `0`.
2. Add a `static` accessor that returns the current value.
3. Increment the value in your `Car` constructor.
4. In your `main` method, print the value of *CAR_COUNT* before and after you create each of your cars.

enum

- Sometimes it is appropriate for a variable to have one of a fixed set of specific values.
  - Days of the week
  - Moths of the year
  - etc.
- It's also often the case that representing these discrete values as integers and strings is possible but not desirable.
  - It is difficult to restrict the values to only those that are valid.
  - For example, the valid integers for days of the week may be `0`-`6`, but it would still be possible to assign an `int` variable a value of `23` or `-257`.
- Java provides a special type that allows the programmer to define a new type and restrict the possible values to a specified list: an `enum`.

```
public enum Day {
  MONDAY,
  TUESDAY,
  WEDNESDAY,
  THURSDAY,
  FRIDAY,
  SATURDAY
}
```

Once you have created an enum, it is a new type that you can use, i.e. as a variable. But you can only assign it one of the defined values.

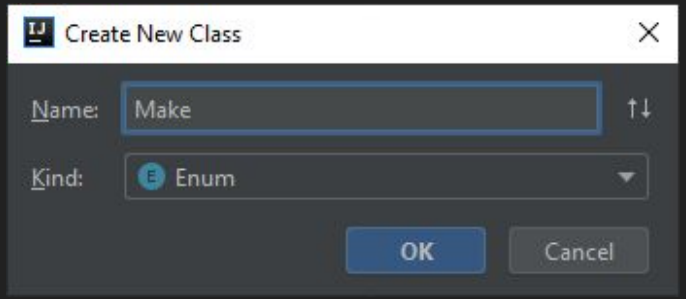
```
Day today;

today = Day.THURSDAY;
```

Activity: `enum` for Make

We'd like to restrict the Make of our vehicles to just a few car companies.

1. Create a new `enum`. Begin by selecting *New → Java Class* from the menu, then change the *"Kind:"* drop-down to select Enum.



2. Name your `enum` Make.
3. Add values for several different companies, e.g. Ford, Dodge, Chrysler, etc.

Activity: Use Your `enum`

We'd like to restrict the make of our vehicles to just a few car companies.

1. Modify your `Car` class so that the type for the `make` field is your `enum` instead of `String`.
2. This will cause lots of compiler problems in your code! Fix them by changing the `Strings` to valid values from your `enum`.