Designing Objects

Today, we look at how to create and apply custom object types in Java. We will also see a useful data structure called a HashMap.

Content Learning Targets

After completing this activity, you should be able to say:

- Explain the purpose of constructor, accessor, and mutator methods.
- Implement the equals and toString methods for a given class design.
- Design a new class (UML diagram) based on a general description.
- Explain what a HashMap represents.
- List and apply essential HashMap operations.
- Describe the benefits and limitations of HashMaps.

Process Skill Goals

During the activity, you should make progress toward:

- Identifying attributes and data types that model a real-world object. (Problem Solving)

Facilitation Notes

Model 1 begins with questions about constructors, getters, and setters. Report out as soon as the majority of teams have finished. You may find it helpful to project the source code for *Color.java* while students work through the questions. Reinforce the concept of immutable objects, and point out that the `String` class is designed this way.

The questions at the beginning of Model 2 require students to understand the source code of `equals` and `toString`. If this is the first time they have seen language features like `Object`, `instanceof`, and `String.format`, you might want to give a 5-minute lecture (but avoid giving away the answers).

When reporting out Model 3, have presenters write their designs on the board. Compare the trade-offs of their different designs. For example, to store ID numbers some teams may use strings, others may use ints/longs/arrays of ints.

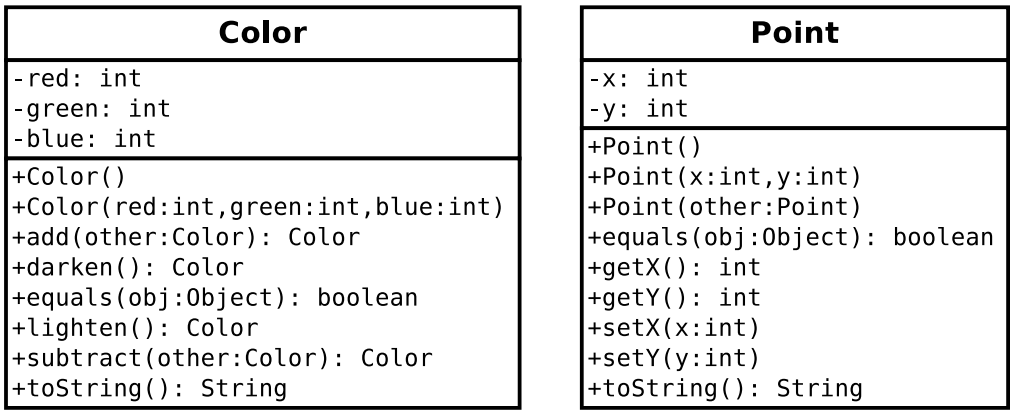
Model 4 introduces HashMaps in the context of the Animal Shelter example.

Key questions: #7, #11, #18, #27

Source files: *Color.java*, *Point.java*, *AnimalShelterMain.java*

Model 1 Common Methods

Classes are often used to represent abstract data types, such as `Color` or `Point`:



As shown in the UML diagrams, classes generally include the following kinds of methods (in addition to others):

- **constructor** methods that initialize new objects
- **accessor** methods (getters) that return attributes
- **mutator** methods (setters) that modify attributes

Questions (15 min) Start time:

1. Identify the constructors for the `Color` class. What is the difference between them?

There are two constructors: one that takes no parameters (the default constructor), and one that takes three integers for the RGB values.

2. What kind of constructor does the `Point` class have that the `Color` class does not?

The `Point` class also has a copy constructor: one that "copies" the values of another object.

3. Identify an accessor method in the `Point` class.

a) What is the name of the method? `getX` or `getY`

b) Which instance variable does it get? `this.x` or `this.y`

c) What arguments does the method take? none

d) What does the method return? The value of `x` or `y`

4. Identify a mutator method in the `Point` class.

a) What is the name of the method? `setX` or `setY`

b) Which instance variable does it set? `this.x` or `this.y`

c) What arguments does the method take? The value of `x` or `y`

d) What does the method return? nothing

5. How would you define accessor methods for each attribute of the `Color` class? Write your answer using UML syntax.

```
+getRed(): int
+getGreen(): int
+getBlue(): int
```

6. How would you define mutator methods for each attribute of the `Color` class? Write your answer using UML syntax.

```
+setRed(red:int)
+setGreen(green:int)
+setBlue(blue:int)
```

7. The `Color` class does not provide any accessors or mutators. Instead, it provides methods that return new `Color` objects. Why do you think the class was designed this way?

Other than the constructor, there are no methods that change the `red`, `green`, and `blue` values. This design makes the class immutable, which means that objects can be reused. The `String` class is also designed this way.

Model 2 Object Methods

In addition to providing constructors, getters, and setters, classes often provide `equals` and `toString` methods. These methods make it easier to work with objects of the class.

As a team, review the provided *Color.java* and *Point.java* files. Run each program to see how it works. Then answer the following questions using the source code (don't just guess).

Questions (15 min) Start time:

8. Based on the output of *Color.java*, what is the value of each expression below?

```
Color black = new Color();
Color other = new Color(0, 0, 0);
Color gold = new Color(255, 215, 0);
```

a) `black == other` false

b) `black == gold` false

c) `black.toString()` "#000000"

d) `black.equals(other)` true

e) `black.equals(gold)` false

f) `gold.toString()` "#ffd700"

9. What is the purpose of the `toString` method?

It returns a `String` representation of the `Color` (in HTML/CSS format). The `toString` method makes it easier to examine and debug objects.

10. Based on the output of *Point.java*, what is the value of each expression below?

```
Point p1 = new Point();
Point p2 = new Point(0, 0);
Point p3 = new Point(3, 3);
```

a) `p1 == p2` false

b) `p1.toString()` "(0, 0)"

c) `p3.toString()` "(3, 3)"

d) `p1.equals(p2)` true

e) `p1.equals("(0, 0)")` false

f) `p3.equals("(3, 3)")` false

11. What is the purpose of the `equals` method?

It determines whether two objects have the same attribute values. The `equals` method is useful for testing with `assertEquals`.

12. Examine *Point.java* again. What is the purpose of the `if`-statement in the `equals` method?

Since `equals` can take any type of `Object`, you need to check if the argument is a `Color` or `Point` instance before using it as such.

13. How could you modify the `equals` method to cause both #10e and #10f to return `true`?

Change the last line to `return this.toString().equals(obj);`

You could instead add `if (obj instanceof String)`, but since the `String.equals` method takes an `Object`, there's no need to convert the `obj` parameter before calling `String.equals`.

Model 3 Shelter Animal

Classes often represent objects in the real world. In this section, you will design a new class that represents an `Animal` in a shelter like the one below. This will be the first in a series of activities in which we build up a record-keeping system for an animal shelter.

Photo by Anna Kumpan on Unsplash

Questions (15 min) Start time:

14. Identify four or more attributes (a.k.a. fields) that should be in the `Animal` class. For each attribute, indicate what data type would be most appropriate.

Answers may include `name:String`, `id:int/String`, `species:String`, `ageYears:int`, `dateArrived:Date`, `birthdate:Date`, `sex:String`, `size:String`, `weight:double`, `color:Color`, `isSpayedNeutered:boolean`, `isHouseTrained:boolean`, etc.

15. Using UML syntax, define two or more constructors for the `Animal` class.

```
+Animal(id:int)
+Animal(id:int, name:String, species:String)
```

16. Define two or more accessor methods for the `Animal` class. Include arguments and return values, using the same format as a UML diagram.

```
+getID(): int
+getName(): String
+getSpecies(): String
+getAgeYears(): int
+getWeight(): double
```

17. Define two or more mutator methods for the `Animal` class. Include arguments and return values, using the same format as a UML diagram.

```
+setWeight(weight:double): void
+setName(name:String): void
+setSpayedNeutered(status:boolean): void // could omit parameter
+setAgeYears(age:int): void
```

18. Describe how you would implement the `equals` method of the `Animal` class.

Two animals would be considered equal if they have the same ID number, assuming we have been careful to keep these IDs unique.

19. Describe how you would implement the `toString` method of the `Animal` class.

The `toString` would print the animal's ID, species, name, and other field values, each separated by a comma.

20. When constructing (or updating) an `Animal` object, which arguments would you need to validate? What are the valid ranges of values for each attribute?

The ID number should have a consistent length (number of digits), age in years should be nonnegative, dates need to have valid months and days, names should be at most some upper bound number of letters and not contain digits or other characters, weight should be positive, etc.

Model 4 HashMaps

Questions (25 min) Start time:

We will often find useful a ***map*** data structure, which associates ***keys*** with ***values***. A key is some type of unique identifier, and a value is an object containing the data attached to that identifier. Each key-value pair is called an ***entry*** in the map. In some languages, a map is called a dictionary; keys are like words in a dictionary, and the values are like definitions.

21. Suppose we have a `Student` class and we want to create a map that lets us look up a student by their (String) username. Which should be the key type, and which should be the value?

The keys should be Strings (the usernames), and the values should be the `Student` objects.

The built-in Java map class that we will use is `HashMap`. Behind the scenes, `HashMap` uses a technique called hashing to speed up its operations. You will learn more about hashing in CSSE 230, but for now, just know that HashMap lookups are fast. We declare/initialize HashMaps like we do for ArrayLists, but with two type parameters (key, then value), instead of one.

22. Complete the following declaration and initialization of the student-username map.

```
HashMap< String , Student > usernameToStudent = new HashMap<>();
```

23. Find the HashMap class docs in the Java API. Looking at its methods, how would we...

add the new entry `username` → `student`? `usernameToStudent.put(username, student);`

check if the map already has `username`? `usernameToStudent.containsKey(username)`

check if the map already has `student`? `usernameToStudent.containsValue(student)`

determine whether the map has no entries? `usernameToStudent.isEmpty()`

Alternative: use `size()` and compare to zero. But `isEmpty()` is preferred.

retrieve the Student for `username`, if it exists? `usernameToStudent.get(username);`

There is also `getOrDefault` if you don't want to return `null` on failed lookups.

You will encounter other useful HashMap methods in a programming assignment. We'll now see how HashMaps can accelerate object lookups based on identifiers.

24. Review *AnimalShelterMain.java*. How does this class currently store its `Animal` objects?

It uses an `ArrayList` of type `Animal` named `animals`.

25. Examine the methods `getAnimalbyName` & `updateAnimalWeight`. What flaw(s) do you see?

To find an Animal given its name or ID, these methods iterate through the entire ArrayList of animals. This could take a long time if the animal we're looking for is near the end.

26. Replace the `animals` field of type `ArrayList<Animal>` with two HashMap fields: `animalsByID` and `animalsByName`. What should the key and value types be for these HashMaps?

`animalsByID` maps IDs to Animals, so its declaration should be `HashMap<Integer, Animal>`. `animalsByName` maps names to Animals, so its declaration should be `HashMap<String, Animal>`.

27. Update the constructor, `addAnimal`, `getAnimalbyName`, and `updateAnimalWeight` methods to use the new fields. After all changes, run the program again. How has the output changed?

The HashMap is printed a bit differently from the old ArrayList due to their `toString` implementations. With the HashMap, when we add an animal with a repeated ID, it overwrites the old entry in `animalsByID` with that key.

28. What is one potential downside to using the second `HashMap`, `animalsByName`?

The shelter may get animals with matching names; the `animalsByName` map will only store the most recently added animal.