Designing Classes

The `String` class provides methods for working with text. The `Random` class provides methods for generating random numbers. In this activity, you'll learn how to make your own classes that represent everyday objects.

Manager:		Recorder:	
Presenter:		Reflector:	

Content Learning Objectives

After completing this activity, students should be able to:

- 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.

Process Skill Goals

During the activity, students 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 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).

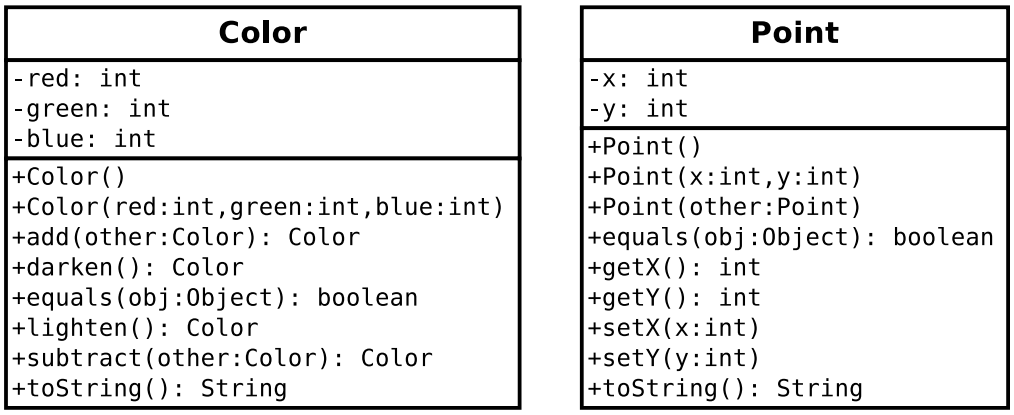
When reporting out Model 3, have presenters write their designs on the board. Compare the trade-offs of their different designs. For example, to store credit card numbers some teams may use strings, others may use arrays of integers, and some may use a `long` variable (`int` won't work because of the range).

Key questions: #7, #11, #18

Source files: *Color.java*, *Point.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 Credit Card

Classes often represent objects in the real world. In this section, you will design a new class that represents a `CreditCard` like the one below:

Questions (15 min) Start time:

14. Identify two or more attributes that would be necessary for the `CreditCard` class. For each attribute, indicate what data type would be most appropriate.

Answers may include `number:long`, `expire:Date`, `name:String`, `code:int`, etc.

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

```
+CreditCard()
+CreditCard(number:long, name:String)
```

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

```
+getNumber(): long
+getExpire(): Date
+getName(): String
+getCode(): int
```

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

```
+setNumber(number:long): void
+setExpire(expire:Date): void
+setName(name:String): void
+setCode(code:int): void
```

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

Two credit cards would be considered equal if they have the same account number, assuming there are no duplicates in the bank.

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

The `toString` would print the account number, expiration date, and cardholder's name, each separated by a comma.

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

The number should have 16 digits, dates need to have valid months and days, names should be at most 22 letters and not contain digits or other characters, code should be 3–4 digits, etc.