Introduction to Objects: Semantics and Syntax

- Defining an object
- Creating an object
- Instance variables
- Instance methods
What is OOP?

- Object-oriented programming (constructing software using objects) and procedural-oriented programming are different programming styles
  - It's all about decomposing complexity
  - Procedural: use methods for modularization and abstraction
  - OOP: use objects for modularization and abstraction
- OOP approaches break down problems into program “things” (objects) that parallel real-world “things” in the problem domain itself.
  - We construct fields (instance variables) - things that the object “knows” or represents a property or thing that the object “has”. Object state!
  - We construct routines (methods) - things that the object can “do”.

<table>
<thead>
<tr>
<th>name</th>
<th>Point</th>
<th>Card</th>
<th>Boat</th>
</tr>
</thead>
<tbody>
<tr>
<td>fields</td>
<td>x</td>
<td>suit</td>
<td>name</td>
</tr>
<tr>
<td></td>
<td>y</td>
<td>rank</td>
<td>maxSpeed</td>
</tr>
<tr>
<td>methods()</td>
<td>display()</td>
<td>show()</td>
<td>go()</td>
</tr>
</tbody>
</table>

UML (Unified modeling language) diagrams

Wright State University, College of Engineering
Dr. T. Doom, Computer Science & Engineering
CS 241
Computer Programming II
Example: TETRIS

- What are the game’s objects?
  - piece, board
- Capabilities: What do those objects know how to do?
  - piece: be created, fall, rotate, stop at collision
  - board:
    - be created, remove rows, check for end of game
- Properties: What attributes and components do they have?
  - Piece: orientation, position, shape, color
  - board: size, rows
Classes and objects

- A class is used to describe the name, fields, and methods for an object.
- A class is *not* an object. It's more like a blueprint for *how* to make one and a set of instructions for *what* it does and *how*.
- Each object *instance* made from a class has the same fields (instance variables), but the *values* of those fields (state) may differ from object instance to object instance.
- Analogy: Rolodex
  - A class is like a set of instructions for creating a blank Rolodex card. It notes that you need to have an entry for name, phoneNumber, etc.
  - When you create an object (using the new operator) you create a blank Rolodex card with those space for that information.
  - You then *set* those values for that specific card/object instance.
  - Another card/object instance would have the same type of information, but not necessarily the same values.
Creating/using an object

- It generally takes two classes to develop an object:
  - One to create the object
  - One to test/use the object (this one has main()).

```java
class PointTest {
    public static void main(String[] args) {
        Point p1 = new Point();
        p1.x = 5;
        p1.display();
    } // end method main
} // end class PointTest

class Point {
    int x;
    int y;

    void display() {
        System.out.println("(" + x + "," + y + ")");
    } // end method display
} // end class Point
```

- **Instance variables** are declared inside a class, but not within a method.
  - These fields define state and last for as long as the object lives!
- **Local variables** are declared within a method and are not persistent.
class PointTest {
    public static void main (String[] args) {
        Point p1 = new Point();
        p1.x = 5;
        p1.display();
    } // end method main
} // end class PointTest

class Point {
    int x;
    int y;

    void display() {
        System.out.println("(" + x + "," + y + ")");
    } // end method display
} // end class Point

PointTest HAS-A Point
Point is contained by PointTest
Procedural Vs. OO programming

- Consider a program that uses a deck of cards.
  - Write a procedural program to implement a random draw of a card
  - Write the description of objects to do the same thing
- Now use your implementation to play blackjack
  - You now need to draw two cards
  - Or more (on a hit)
- What if there are six decks of cards in the shoe?
  - How does that change your procedural program?
  - How does that change your object-oriented program?

- OOP mantra: Flexibility and Extendibility.
  - Specs ALWAYS change
  - Code should easy to reuse in future projects
Objects and Reference variables

- In your prerequisite coverage, you saw:
- Primitive data types (int, boolean, double, etc.)
- Derived data types (String, Scanner, etc).
- There is actually no such thing as an “object variable”, all non-primitive data types are actually the same thing, a *reference variable*.
- Reference variables hold bits that represent a way to *access* an object
  - Where are the objects variables in memory?
  - Where do I go in memory to find the instructions to run an object’s methods?
- It doesn’t hold the object itself, just a “pointer”, or “address”, or “reference” to the object.

**Steps of Object Creation**

- Step 1: Declare a reference variable
  ```java
  Point p1;
  ```
- Step 2: Create an object
  ```java
  new Point();
  ```
- Step 3: Store the new object’s reference in the variable (*link* the object)
  ```java
  p1 = new Point();
  ```
Strings are objects

- String is a derived data type (an object)
- String is such a useful data type that it is built into the language (java.lang)
  - String has its own context for +
    - Concatenate
  - String objects are built automatically (without the need to invoke new).
  - String objects have methods like length(), toLowerCase(), and charAt()

String message1 = “Hello” + “ ” + “World” + “
”;
String message2 = “Goodbye”;  
char letter = message2.charAt(0);

Primitive data types store values
int x = 10; int y = 5;

Objects use references/pointers
message1 → Hello World
message2 → Goodbye
message1 = message2;

message1 → Hello World
message2 → Goodbye
Arrays are objects

- A variable can be assigned to any object of the appropriate type.
  // Create an array referenced numbers.
  int[] numbers = new int[10];
  int[] digits = new int[500];
  ...
  numbers = digits;

- Both variables now reference the same object (NOT a copy)
- If an object has no references then the object is lost
  - it is “garbage” and will be “collected”
Life and death on the heap

- The default value of a newly declared reference variable is \textit{null}.
  - It doesn’t point at anything until you store a reference in it
- All new objects exist in an area of controlled memory called the \textit{heap}.
- As long as an object has an active reference then it is \textit{reachable}.
- An object that does not have an active reference is \textit{abandoned}
  - In Java, abandoned objects are eligible for garbage collection (automatic destruction)
  - In C++, objects must be explicitly destroyed by the program. Objects that are abandoned can never be destroyed and cause a \textit{memory leak}.

Point p1;
Point p2 = new Point();  \hspace{1cm} \text{At the end of this code segment:}
p1 = p2;
p2 = new Point();
p1 = null;

- How many active references exist?
- How many reachable objects?
- How many abandoned objects?

Consider drawing a diagram…
Passing objects as method arguments

- Object references can be passed to methods like any data type.

```java
int[] numbers = {5, 10, 15};
printList(numbers);
...

void printList(int[] list){
    for (int i = 0; i < list.length; i++)
        System.out.print(list[i] + " ");
}
// end method printList

void clearList(int[] list){
    for (int i = 0; i < list.length; i++)
        list[i] = 0;
}
// end method clearList
```

- What does this call by reference imply to changes made in clearList()?
Arrays of objects

- Array elements can contain *primitive data types* or *references* to objects.
  - We’ve seen this so far as arrays of String
    
    ```java
    String[] nameList = {"Sam", "Bobbie", "Pat", "Kim", "Teri"};
    ```

- An array of references can be made of any object type
  
  ```java
  Point[] pointList = {new Point(), new Point()};
  ```
/**
 * goFish: Searches a hand for all cards of a named rank
 * @param rank The rank of card to be searched. Must be passed as a String in the form “Ace”, “King”, …, “Two”
 * @param hand The hand of cards to be searched, found cards will be removed!
 * @return All cards contained in hand that have the specified rank (may be empty)
 */
ArrayList<Card> goFish(String rank, ArrayList<Card> hand) {
    ArrayList<Card> foundCards = new ArrayList<Card>();
    for (Card theCard : hand) {
        if (theCard.rank.equals(rank)) {
            foundCards.add(theCard);
            hand.remove(theCard);
        } // end if found card of correct rank
    } // end for each card in hand
    return foundCards;
} // end method goFish
Static: Class variables

Point p1 = new Point();
p1.x = 7;
p1.y = 8;
Point.numPoints++;
Point p2 = new Point();
p2.x = 3;
p2.y = 1;
p2.numPoints++;
System.out.println(p1.numPoints);

- Static variables:
  - a single memory location shared by all instances of the class
  - one value per class, instead of one value per instance
  - can be accessed w/o an instance

- p1.numPoints++ Vs. p1.x++ Vs. Point.x++ Vs. Point.numPoints++

```
class Point {
    static int numPoints = 0;
    int x;
    int y;
    ...
} // end class Point
```
Static methods

- Static methods can run without any instance of the class
  - Behavior is not dependent on the state of an object instance
- Thus they may be called without any instance variables actually exist
  - Therefore, static methods may NEVER use instance variables
  - Static methods can only use static variables
  - Static methods of a class cannot use regular (non-static) methods of a class!
- Style: Static methods and variables should be called using the class name rather than an object reference variable
Example: Calculating the midpoint of a line segment

- Problem: Develop a method that, given two points returns the midpoint of the line (round to the closest integer coordinates).

- Recall: A method can only return one value
- Example: Test that given the point (3,3) and the point (5,5) that your method returns the point (4,4)

- Note: It is often a good idea to come up with the test code first.
  - This is one of the fundamental principles of extreme programming.
Introduction to Objects: Style

- accessors/mutators
- access modifiers
- encapsulation
- keyword: this
- standard methods in java
- constructors
Say we want to use our “Point” class to keep track of things on a graphics canvas of a specific standard size

What if someone using our CanvasPoint object does this:
- CanvasPoint point1 = new CanvasPoint();
- Point1.x = 1000;

We need a way to protect our instance values from misuse from an uninformed user (or an honest mistake)
- Unfortunately, if someone has access to our variables, they can do pretty much anything that they want to with them.

We need a way to FORCE the user to check the values
Using mutators (set methods)

- We can create new methods whose only job is to provide “checked” access to the variables.

```java
public void setX(int xValue) {
    if (xValue < 0) {
        xValue = 0;
    }
    if (xValue > 800) {
        xValue = 800;
    }
    x = xValue;
} // end method setX

public void setY(int yValue) {
    if (yValue < 0) {
        yValue = 0;
    }
    if (yValue > 600) {
        yValue = 600;
    }
    y = yValue;
} // end method setY
```

- These are called a mutator methods.
  - It changes the state of the object (instance variable value)
- What if later we decide to use a different sized Canvas?
- But can we force the user to use the mutator?
Hiding data

- **We can use access specifier** to change to is allowed to access an object’s variables and methods
  - **public**: any class can access our fields
  - **private**: only methods in the class can access its fields

```java
public class CanvasPoint {
    private int x;
    private int y;
    public void setX (int xValue) {
        // our code here...
    }
    public void setY (int yValue) {
        // our code here...
    }
} // end class CanvasPoint
```

- How can other classes look at (and use) the private values of x and y?
Using accessors (get methods)

- We create methods whose only job is to access private variables and return the result for use.

```java
public class CanvasPoint {
    private int x;
    private int y;
    public void setX (int xValue)...
    public void setY (int yValue)...

    public int getX () {
        return x;
    }
    public int getY () {
        return y;
    }
} // end class CanvasPoint
```

- These are called accessor methods.
Encapsulation

- This is the *most* important OO rule for style
  - Hide your data! Do *not* expose your variables
  - Mark all instance variables private
  - Mark access/mutators (getters/setters) public
- Always force the use of getters/setters.
  - *Never expose your data!*
  - Even if you do *no* checking right now, you may want to later
  - This means less code changing later!

Well encapsulated!

CanvasPoint

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>-x</td>
<td>-y</td>
</tr>
<tr>
<td>+display() : void</td>
<td></td>
</tr>
<tr>
<td>+setX(xValue : int) : void</td>
<td></td>
</tr>
<tr>
<td>+setY(yValue : int) : void</td>
<td></td>
</tr>
<tr>
<td>+getX() : int</td>
<td></td>
</tr>
<tr>
<td>+getY() : int</td>
<td></td>
</tr>
</tbody>
</table>

But what is with xValue and yValue stuff?
Keyword: *this*

- Stylistically, it makes sense to use the logical name for the method parameters that you use for the private instance variables.
  - How do we deal with the namespace collision?
- Note: Overloading variables names is a feature!

```java
public class CanvasPoint {
    private x;
    ...
    public void setX (int x) {
        // code to check valid x
        x = x; // semantic error!
    }
    ...
} // end class CanvasPoint
```

```
public class CanvasPoint {
    private x;
    ...
    public void setX (int x) {
        // code to check valid x
        this.x = x;
    }
    ...
} // end class CanvasPoint
```

- The keyword *this* refers to the current object.
Making your objects behave like library objects

- What happens when you do this?
  
  ```java
  ... String s = "Hello Class";
  System.out.println(s);
  ...
  ...
  ... int s = 5;
  System.out.println(s);
  ...
  ...
  ... Scanner s = new Scanner();
  System.out.println(s);
  ...
  ...
  ... CanvasPoint s = new CanvasPoint();
  System.out.println(s);
  ...
  ...
  ```

- Why?
  Print methods call the `toString()` method on your object
  If you don’t have one, it just prints the `value` stored in the reference variable (more on this later)
  What other methods should all new objects define?

---

CanvasPoint

<table>
<thead>
<tr>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>-x</td>
</tr>
<tr>
<td>-y</td>
</tr>
<tr>
<td>+display() : void</td>
</tr>
<tr>
<td>+setX(x : int) : void</td>
</tr>
<tr>
<td>+setY(y : int) : void</td>
</tr>
<tr>
<td>+getX() : int</td>
</tr>
<tr>
<td>+getY() : int</td>
</tr>
</tbody>
</table>
public class CanvasPoint {

    ... public String toString() {
        return "(" + x + "," + y + ")";
    } // end method toString

    public boolean equals (CanvasPoint obj) {
        return ( (x == obj.getX()) &&
                (y == obj.getY()) );
    } // end method equals

    public CanvasPoint clone () {
        CanvasPoint cloneObject;
        cloneObject = new CanvasPoint();
        cloneObject.setX(x);
        cloneObject.setY(y);
        return cloneObject;
    } // end method clone

    ... } // end class CanvasPoint
Constructors

- Often there is code that we need/want to run *every* time we create a new object

```java
... CanvasPoint p1 = new CanvasPoint();
p1.setX(x);
p1.setY(y);
...
```

- *Constructor*: a method automatically called when an object is created

```java
public CanvasPoint (int x, int y) {
    this.setX(x);
    this.setY(y);
} // end constructor
```

```
<table>
<thead>
<tr>
<th>CanvasPoint</th>
</tr>
</thead>
<tbody>
<tr>
<td>-x</td>
</tr>
<tr>
<td>-y</td>
</tr>
<tr>
<td>+CanvasPoint( x: int, y : int )</td>
</tr>
<tr>
<td>+display() : void</td>
</tr>
<tr>
<td>+setX(x : int) : void</td>
</tr>
<tr>
<td>+setY(y : int) : void</td>
</tr>
<tr>
<td>+getX() : int</td>
</tr>
<tr>
<td>+getY() : int</td>
</tr>
<tr>
<td>+toString() : String</td>
</tr>
<tr>
<td>+equals(obj : CanvasPoint) : boolean</td>
</tr>
<tr>
<td>+clone() : CanvasPoint</td>
</tr>
</tbody>
</table>
```
Default/Multiple Constructors

public CanvasPoint () {} // no-arg
public CanvasPoint (int x, int y) {
    this.setX(x);
    this.setY(y);
} // end constructor

- Constructors have no return type and must have the same name as the class
- The default constructor is a no-arg constructor
  - The compiler adds this for you if you do not define your own constructor for the class
  - If you do define a constructor then the compiler does not add this for you.
- You can overload the constructor method
  - Multiple constructors
  - Re-add a “default” constructor if you’d like

<table>
<thead>
<tr>
<th>CanvasPoint</th>
</tr>
</thead>
<tbody>
<tr>
<td>-x</td>
</tr>
<tr>
<td>-y</td>
</tr>
<tr>
<td>+CanvasPoint()</td>
</tr>
<tr>
<td>+CanvasPoint( x: int, y : int )</td>
</tr>
<tr>
<td>+display() : void</td>
</tr>
<tr>
<td>+setX(x : int) : void</td>
</tr>
<tr>
<td>+setY(y : int) : void</td>
</tr>
<tr>
<td>+getX() : int</td>
</tr>
<tr>
<td>+getY() : int</td>
</tr>
<tr>
<td>+toString() : String</td>
</tr>
<tr>
<td>+equals(obj : CanvasPoint) : boolean</td>
</tr>
<tr>
<td>+clone() : CanvasPoint</td>
</tr>
</tbody>
</table>

Discussion topic:
Should programming languages provide destructors?