Topic 24
Classes Part II

"Object-oriented programming as it emerged in Simula 67 allows software structure to be based on real-world structures, and gives programmers a powerful way to simplify the design and construction of complex programs. "

- David Gelernter

Based on slides for Building Java Programs by Reges/Stepp, found at http://faculty.washington.edu/stepp/book/

More on Classes

- Classes are programmer defined data types
- In Java the primitives (int, char, double, boolean) are the core data types already defined
- All other data types are classes, and are defined by programmers
  - even the classes in the Java Standard Library
  - look at source code
- Classes are another technique for *managing complexity*
  - along with sub programs (methods) and arrays (data structures)

Instance Methods and Encapsulation

Programming Paradigms

- Classes are a major part of a style of programming called Object Oriented Programming
- One technique for managing complexity and building correct programs
- Not the only one

Accessor methods

- **accessor**: A method that returns state of the object, or computes and returns values based on the object's state.
  - Unlike mutators, accessors do not modify the state of the object.
- example: Write a method named `distance` in the Point class that computes and returns the distance between two Points. (Hint: Use the Pythagorean Theorem.)
- example: Write a method named `distanceFromOrigin` that computes and returns the distance between the current Point and the origin at (0, 0).

Problem: printability

- By default, println'ing new types of objects prints what looks like gibberish:

```
Point p = new Point(10, 7);
System.out.println(p);  // Point@9e8c34
```

- We can instead print a more complex String that shows the object's state, but this is cumbersome.

```
System.out.println("(" + p.x + ", " + p.y + ")");
```

- We'd like to be able to simply print the object itself and have something meaningful appear.

```
// desired:
System.out.println(p);  // (10, 7)
```


Special method toString

- If you want your new objects to be easily printable, you can write a method named `toString` that tells Java how to convert your objects into Strings as needed.

- The `toString` method, general syntax:

```
public String toString() {
    <statement(s) that return an appropriate String> ;
}
```

  - Example:

```
// Returns a String representing this Point.
public String toString() {
    return "(" + this.x + ", " + this.y + ")";
}
```


How toString is used

- Now, in client code that uses your new type of objects, you may print them:
  - Example:

```
public class UsePoint2 {
    public static void main(String[] args) {
        Point p = new Point(3, 8);
        System.out.println("p is " + p.toString());
    }
}
```

OUTPUT:

```
p is (3, 8)
```

- Java allows you to omit the `.toString()` when printing an object. The shorter syntax is easier and clearer.

```
System.out.println("p is " + p);
```

Multiple constructors

- It is legal to have more than one constructor in a class.
  - The constructors must have different parameters.

```
public class Point {
    int x;
    int y;

    // Constructs a Point at the origin, (0, 0).
    public Point() {
        this.x = 0;
        this.y = 0;
    }

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    // ...
}
```


The this keyword

- To avoid redundant code, one constructor may call another using the `this` keyword.

```
public class Point {
    int x;
    int y;

    public Point() {
        this(0, 0); // calls the (x, y) constructor
    }

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    // ...
}
```


Encapsulation

- It is considered good style to protect your objects' data fields from being externally modified.
  - Fields can be declared *private* to indicate that no code outside their own class can change them.
  - Declaring a private field, general syntax:
    `private` ***<type> <name>*** `;`
  - Example:
    `private int x;`
- Once fields are private, they otherwise would not be accessible at all from outside. We usually provide accessor methods to see (but not modify) their values:

```
public int getX() {
    return this.x;
}
```


The equals method

- The == operator essentially does not work as one might expect on objects:
  - Example:

```
Point p1 = new Point(5, -3);
Point p2 = new Point(5, -3);
System.out.println(p1 == p2);  // false
```

- Instead, objects are usually compared with the equals method. But new types of objects don't have an equals method, so the result is also wrong:

```
System.out.println(p1.equals(p2));  // false
```

- We can write an equals method that will behave as we expect and return true for cases like the above.

Writing an equals method

- The equals method, general syntax:

```
public boolean equals(Object <name>) {
    <statement(s) that return a boolean> ;
}
```

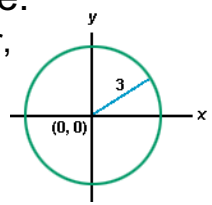
  - To be compatible with Java's expectations, the parameter to equals must be type `Object` (which means, 'any object can be passed as the parameter').
  - The value that is passed can be cast into your type.
  - Example:

```
// Returns whether the have the same x/y
public boolean equals(Object o) {
    Point p2 = (Point) o;
    return this.x == p2.x && this.y == p2.y;
}
```

  - This is our first version of equals. It turns out there is much more involved in writing a correct equals method

Object practice problem

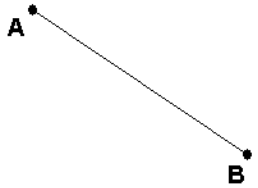
- Create a new type of objects named Circle.
  - A circle is represented by a point for its center, and its radius.
  - Make it possible to construct the unit circle, centered at (0, 0) with radius 1, by passing no parameters to the constructor.



  - Circles should be able to tell whether a given point is contained inside them.
  - Circles should be able to draw themselves using a Graphics.
  - Circles should be able to be printed on the console, and should be able to be compared to other circles for equality.

Object practice problem

- Create a new type of objects named LineSegment.
  - A line segment is represented by two endpoints.



  - A line segment should be able to compute its slope (y2-y1)/(x2-x1).
  - A line segment should be able to tell whether a given point intersects it.
  - Line segments should be able to draw themselves using a Graphics.
  - Line segments should be able to be printed on the console, and should be able to be compared to other lines for equality.

Advanced Object Features

Default initialization

- If you do not initialize an object's data field in its constructor, or if there is no constructor, the data field is given a default 'empty' value.
  - Recall the initial version of the Point class:

```
public class Point {
    int x;
    int y;
}
```

  - Example (using the above class):

```
Point p1 = new Point();
System.out.println(p1.x);  // 0
```

- This is similar to the way that array elements are automatically initialized to 'empty' or zero values.

Null object data fields

- What about data fields that are of object types?
  - Recall the initial version of the Point class:

```
public class Circle {
    Point center;
    double radius;
}
```

  - Example (using the above class):

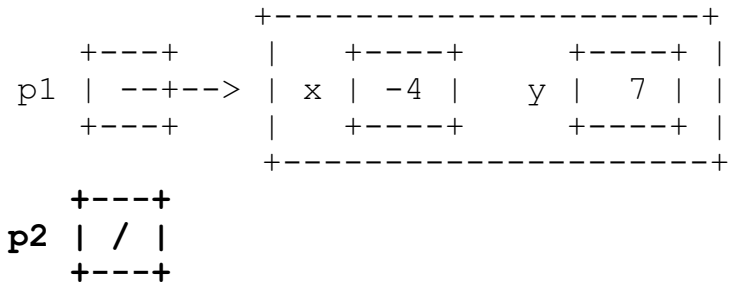
```
Circle circ = new Circle();
System.out.println(circ.center);  // null
```

- Java prints the bizarre output of 'null' to indicate that the circle's center data field does not refer to any Point object (because none was constructed and assigned to it).

The keyword null

- **null**: The absence of an object.
  - The Java keyword `null` may be stored into a reference variable (a variable intended to refer to an object) to indicate that that variable does not refer to any object.
  - Example:

```
Point p1 = new Point(-4, 7);
Point p2 = null;
```



NullPointerException

- If you try to call a method on a variable storing null, your program will crash with a NullPointerException.
  - Example:

```
Point p = null;
System.out.println("p is " + p);
System.out.println(p.getX());  // crash
```

  - Output:

```
p is null
Exception in thread "main"
java.lang.NullPointerException
        at UsePoint.main(UsePoint.java:9)
```

- To avoid such exceptions, you can test for null using `==` and `!=`.
  - Example:

```
if (p == null) {
    System.out.println("There is no object here!");
} else {
    System.out.println(p.getX());
}
```

Violated preconditions

- What if your precondition is not met?
  - Sometimes the author of the other code (the 'client' of your object) passes an invalid value to your method.
  - Example:

```
// in another class (not in Point.java)
Point pt = new Point(5, 17);
Scanner console = new Scanner(System.in);
System.out.print("Type the coordinates: ");
int x = console.nextInt();  // what if the user types
int y = console.nextInt();  // a negative number?
pt.setLocation(x, y);
```

  - How can we scold the client for misusing our class in this way?

Throwing exceptions

- **exception**: A Java object that represents an error.
  - When an important precondition of your method has been violated, you can choose to intentionally generate an exception in the program.
  - Example:

```
// Sets this Point's location to be the given (x, y).
// Precondition: x, y >= 0
// Postcondition: this.x = x, this.y = y
public void setLocation(int x, int y) {
    if (x < 0 || y < 0) {
        throw new IllegalArgumentException();
    }

    this.x = x;
    this.y = y;
}
```


Exception syntax

- Throwing an exception, general syntax:

```
throw new <exception type> ();
```

or,

```
throw new <exception type> ("<message>");
```

  - The ***<message>*** will be shown on the console when the program crashes.

- It is common to throw exceptions when a method or constructor has been called with invalid parameters and there is no graceful way to handle the problem.
  - Example:

```
public Circle(Point center, double radius) {
    if (center == null || radius < 0.0) {
        throw new IllegalArgumentException();
    }
}
```