Objects and Classes

- **Object**: a structured collection of fields (or instance variables)
- **Class**: a template for creating objects
- Every Java object is an *instance* of some class
- The class of an object specifies
  - the types and initial values of its fields
  - the set of operations that can be performed on the object (methods)
  - (optionally) a constructor method that is executed when the object is first created

Point Class

```java
public class Point {
    private double x, y;
    public Point () {
        x = 0;
        y = 0;
    }
    public double getX () { return x; }
    public double getY () { return y; }
    public void move (double dx, double dy) {
        x = x + dx;
        y = y + dy;
    }
}
```

(Single = means assignment in Java; “double” is the primitive type of double-precision floating-point numbers)

Creating Objects

- **Declare** a variable of appropriate type to hold the **Point** object
- **Invoke** the constructor for **Point** to create a **Point** instance and store it in the variable
  ```java
  Point c;
  c = new Point();
  ```

- ... or declare and initialize together:
  ```java
  Point c = new Point();
  ```
Objects with Parameters

Constructor methods can take parameters...

```java
public class Point {
    private double x, y;
    public Point (double initX, double initY) {
        x = initX;
        y = initY;
    }
    public double getX () { return x; }
    public double getY () { return y; }
    public void move (double dx, double dy) {
        x = x + dx;
        y = y + dy;
    }
}
```

```java
Point p = new Point(1.5, 3.5);
...Point p = new Point(1.5, 3.5);
...
```

Public and private

- Basically:
  - **public**: accessible from anywhere in the program
  - **private**: only accessible inside the class
- Protect object internals from outside interference

- (There are a couple of other protection levels — **protected** and “package protected”. The details are not too important at this point.)

Invoking Methods

```java
Point p = new Point(0,0);
...
p.move(-1,0);
...
System.out.println("(" + p.getX() + "," + p.getY() + ")");
```

Relationships between objects

- Many situations involve multiple interacting parts
  - ... in software and in the real world
The has a pattern

- A circle has a center
- A person has an address
- A song has a performer
- ...

An object with an instance variable that refers to another object

Building on Point

- A circle is defined by its center (a point) and its radius (a double)

```java
public class Circle {
    private Point center;
    private double radius;
    public Circle(Point initCenter, double initRadius) {
        center = initCenter;
        radius = initRadius;
    }
    public Point getCenter() { return center; }
    public double getRadius() { return radius; }
}
```

Circle has a Point

```java
Point p = new Point(1, 2);
Circle c = new Circle(p, .5);
```

Moving a Circle

Delegate the move to the circle's center:

```java
public class Circle {
    private Point center;
    private double radius;
    ...
    public void move(double dx, double dy) {
        center.move(dx, dy);
    }
}
```
Delegating Functionality

class ColorPoint {
    Point p;
    Color c;
    ColorPoint (double initX, double initY, Color initC) {
        p = new Point(initX, initY);
        c = initC;
    }
    public void move(double dx, double dy) {
        p.move(dx, dy);
    }
    public Color getColor() { return c; }
}

Abstract Data Types (ADTs)

- Describe objects by their *externally visible behaviors*
- Bank account
  - Deposit
  - Withdraw
  - Write check
- Game character
  - Location
  - Strength
  - Move
- No details about the implementation of these behaviors
- Provides a *contract* any implementation must fulfill

Interfaces

- Capture what is common to a set of similar classes
- Provide a contract that every member of that set must fulfill
- Example: Interface for objects that have a *position* and can be *moved*

```java
public interface Displaceable {
    public double getX();
    public double getY();
    public void move(double dx, double dy);
}
```

Implementing the interface

```
public class Point implements Displaceable {
    private double x, y;
    public Point(double initX, double initY) {
        x = initX;
        y = initY;
    }
    public double getX() { return x; }
    public double getY() { return y; }
    public void move(double dx, double dy) {
        x = x + dx;
        y = y + dy;
    }
}
```
public class Circle implements Displaceable {
    private Point center;
    private double radius;
    public Circle(Point center, double radius) {
        center = initCenter;
        radius = initRadius;
    }
    public double getX() { return center.getX(); }
    public double getY() { return center.getY(); }
    public void move(double dx, double dy) {
        center.move(dx, dy);
    }
}

public class ColorPoint implements Displaceable {
    private Point p;
    private Color c;
    ColorPoint(double initX, double initY, Color initC) {
        p = new Point(initX, initY);
        c = initC;
    }
    public void move(double dx, double dy) {
        p.move(dx, dy);
    }
    public Color getColor() { return c; }
}

public class Rectangle implements Displaceable {
    private Point lowerLeft;
    private double width, height;
    public Rectangle(Point initLowerLeft, double initWidth, double initHeight) {
        lowerLeft = initLowerLeft;
        width = initWidth;
        height = initHeight;
    }
    public double getX() { return lowerLeft.getX(); }
    public double getY() { return lowerLeft.getY(); }
    public void move(double dx, double dy) {
        lowerLeft.move(dx, dy);
    }
}
Using interface types

- Interface variables can refer (during execution) to objects of any class implementing the interface

Displaceable d0, d1, d2;
d0 = new Point(1, 2);
d1 = new Circle(new Point(2, 3), 1);
d2 = new Rectangle(new Point(-1, 1), 2, 1);
d0.move(-2, 1);
d1.move(-2, 1);
d2.move(-2, 1);
...
... d0.getX() ... ➔ -1.0
... d1.getX() ... ➔ 0.0
... d2.getX() ... ➔ -3.0

Recap

- **Object**: A collection of related fields (or instance variables)
- **Class**: A template for creating objects, specifying
  - types and initial values of fields
  - code for methods
  - optionally, a constructor method that is executed when the object is first created
- **Interface**: A “signature” for objects, describing a collection of methods that must be provided by classes that implement the interface
- **Object Type**: Either a class or an interface (meaning “this object was created from a class that implements this interface”)

Multiple interfaces

- An interface represents a point of view
- Can see objects from multiple points of view
- Example: Geometric objects
  - All are displaceable
  - Some have area

Area interface

- Contract for objects that that have an area
  - Circles and rectangles do
  - Points don’t

    public interface Area {
        public double getArea();
    }
public class Circle implements Displaceable, Area {
    private Point center;
    private double radius;
    ...
    public double getArea() {
        return Math.PI * radius * radius;
    }
}

public class Rectangle implements Displaceable, Area {
    private Point lowerLeft;
    private double width, height;
    ...
    public double getArea() {
        return width * height;
    }
}

Subtyping

• Some object types are subtypes of others
  – any object belonging to the subtype can be stored in a variable declared with the supertype
• A class that implements an interface is a subtype of the interface
  – variables of the interface type may refer to instances of the implementing class

Area a = new Circle(new Point(2,3), 1);
## Static vs. Dynamic Types

- The **dynamic type** of an object is the class that it was created from.
- The **static type** of a variable is an object type (class or interface) that describes what objects can be stored in that variable.
  - Similarly, the **static type** of an expression is an object type that describes what we know about the possible results of evaluating this expression.