Java Objects and Classes

- *Object*: a structured collection of *fields* (aka *instance variables*) and *methods*

- *Class*: a template for creating objects

- The class of an object specifies
  - the types and initial values of its local state (fields)
  - the set of operations that can be performed on the object (methods)
  - one or more *constructors*: code that is executed when the object is created (optional)

- Every Java object is an *instance* of some class

- Can (optionally) implement an *interface* that specifies it in terms of its operations

Objects in Java

```
public class Counter {          class name

    private int r;              instance variable

    public Counter () {         constructor
      r = 0;
    }

    public int inc () {         methods
      r = r + 1;
      return r;
    }

    public int dec () {
      r = r - 1;
      return r;
    }
}
```

object creation and use

```
public class Main {

  public static void
    main (String[] args) {      constructor
                                invocation
      Counter c = new Counter();

      System.out.println( c.inc() );

  }                             method call
}
```

Creating Objects

- *Declare* a variable to hold the `Counter` object
  - Type of the object is the *name* of the class that creates it
- *Invoke* the constructor for `Counter` to create a `Counter` instance with keyword "new" and store it in the variable

```
Counter c;
c = new Counter();
```

- … or declare and initialize together (preferred)

```
Counter c = new Counter();
```

Constructors with Parameters

```
public class Counter {

  private int r;

  public Counter (int r0) {
    r = r0;
  }

  public int inc () {
    r = r + 1;
    return r;
  }

  public int dec () {
    r = r - 1;
    return r;
  }
}
```

Constructor methods can take parameters

Constructor must have the same name as the class

object creation and use

```
public class Main {

 public static void
   main (String[] args) {

    Counter c = new Counter(3);

    System.out.println( c.inc() );

  }
}
```

constructor invocation

Creating objects

- Every Java variable is mutable

```
Counter c;
c = new Counter(2);
c = new Counter(4);
```

- A Java variable of *reference* type contains the special value "null" before it is initialized

```
Counter c;
if (c == null) {
   System.out.println ("null pointer");
}
```

Single = for assignment
Double == for equality testing

Using objects

- At any time, a Java variable of reference type can contain either the special value "null" or a pointer in the heap
  - i.e., a Java variable of reference type "T" is like an OCaml variable of type "T option ref"
  - The dereferencing of the pointer and the check for "null" are implicitly performed every time a variable is used

```
let f (co : counter option ref) : int =
  begin match !co with
  | None ->
     failwith "NullPointerException"
  | Some c -> c.inc()
  end
```

```
class Foo {
  public int f (Counter c) {
     return c.inc();
  }
}
```

- If null value is used as an object (i.e. with a method call) then a NullPointerException occurs

Encapsulating local state

```
public class Counter {

  private int r;

  public Counter () {        constructor and
    r = 0;                   methods can
  }                          refer to r

  public int inc () {
    r = r + 1;
    return r;
  }

  public int dec () {
    r = r - 1;
    return r;
  }
}
```

```
public class Main {           other parts of the
                              program can only access
  public static void          public members
    main (String[] args) {

    Counter c = new Counter();

    System.out.println( c.inc() );

                              method call
  }
}
```

Encapsulating local state

- Visibility modifiers make the state local by controlling access

- Basically:
  - public : accessible from anywhere in the program
  - private : only accessible inside the class

- Design pattern, first cut
  - Make all fields private
  - Make constructors and methods public

(There are a couple of other protection levels — protected and "package protected". The details are not important at this point.)

Interfaces

- Give a type for an object based on what it *does*, not on how it was constructed

- Describes a contract that objects must satisfy

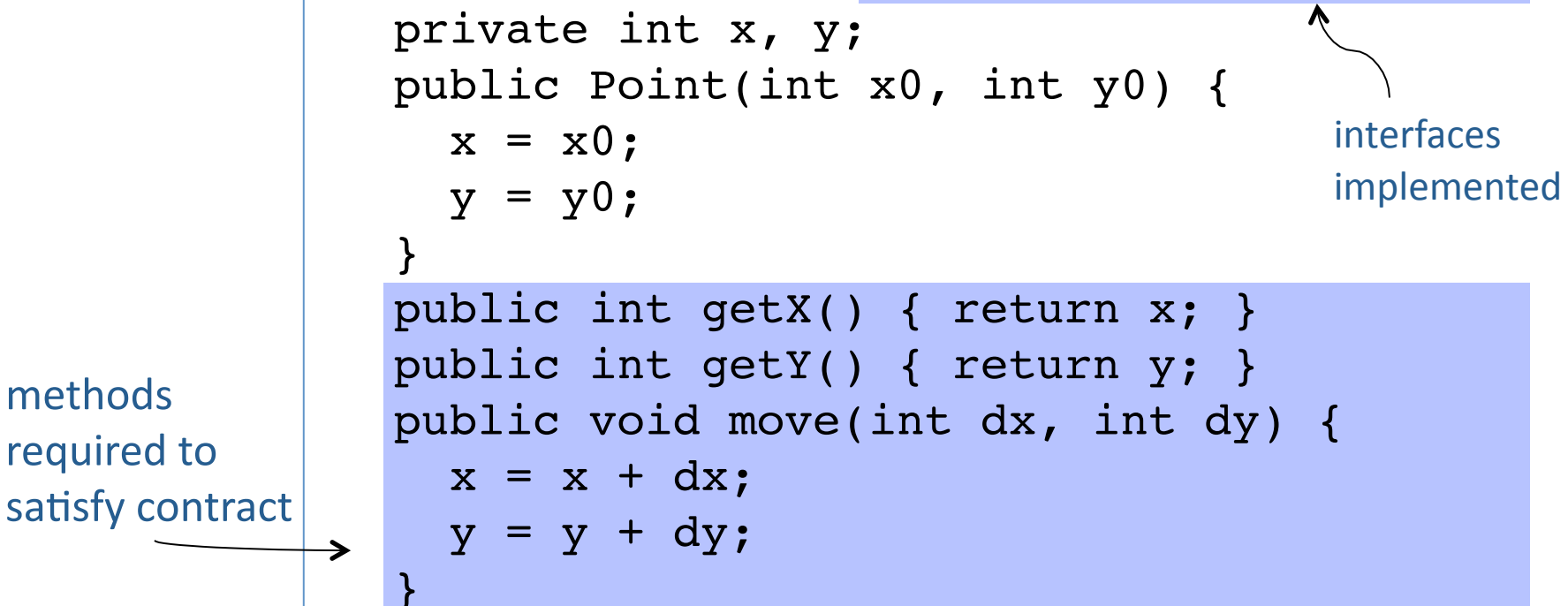- Example: Interface for objects that have a position and can be moved

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

No fields, no constructors, no method bodies!

Implementing the interface

- A class that implements an interface provides appropriate definitions for the methods specified in the interface

- That class fulfills the contract implicit in the interface

```
public class Point implements Displaceable {
   private int x, y;
   public Point(int x0, int y0) {
      x = x0;
      y = y0;
   }
   public int getX() { return x; }
   public int getY() { return y; }
   public void move(int dx, int dy) {
      x = x + dx;
      y = y + dy;
   }
}
```

interfaces implemented

methods required to satisfy contract

Another implementation

```
public class Circle implements Displaceable {
   private Point center;
   private int radius;
   public Circle(Point initCenter, int initRadius) {
      center = initCenter;
      radius = initRadius;
   }
   public int getX() { return center.getX(); }
   public int getY() { return center.getY(); }
   public void move(int dx, int dy) {
      center.move(dx, dy);
   }
}
```

Objects with different local state can satisfy the same interface

Delegation: move the circle by moving the center

And another…

```
class ColorPoint implements Displaceable {
   Point p;
   Color c;
   ColorPoint (int x0, int y0, Color c0) {
      p = new Point(x0,y0);
      c = c0;
    }

   public void move(int dx, int dy) {
      p.move(dx, dy);
   }
   public getX() { return p.getX(); }

   public getY() { return p.getY(); }

   public Color getColor() { return c; }
}
```

Flexibility:
Classes may contain
more methods than
the interface

Yet another…

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

Interfaces as types

- Can declare variables of interface type

```
Displaceable d;
```

- Can assign any implementation to the variable

```
d = new Circle(new Point(1,2), 3);
```

- … but can only operate on the object according to the interface

```
d.move(-1,1);
…
… d.getX() …          ➔  0.0
… d.getY() …          ➔  3.0
```

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 ColorPoint(-1,1, red);
d0.move(-2,0);
d1.move(-2,0);
d2.move(-2,0);
…
… d0.getX() …        ➔  -1.0
… d1.getX() …        ➔  0.0
… d2.getX() …        ➔  -3.0
```

Abstraction

- The interface gives us a single name for all the possible kinds of shapes. This allows us to write code that manipulates arbitrary "displaceables", without caring whether it's dealing with points or circles.

```
class DoStuff {
  public void moveItALot (Displaceable s) {
    s.move(3,3);
    s.move(100,1000);
    s.move(1000,234651);
  }

  public void dostuff () {
    Displaceable s1 = new Point(5,5);
    Displaceable s2 = new Circle(new Point(0,0),100);
    moveItALot(s1);
    moveItALot(s2);
  }
}
```

Multiple interfaces

- An interface represents a point of view

  …but there are multiple points of view

- Example: Geometric objects
  - All can move (all are Displaceable)
  - Some have area

Area interface

- Contract for objects that that have an area
  - Circles do
  - Points and ColorPoints don't

```
public interface Area {
      public double getArea();
}
```

Circle implementation of Area

```
public class Circle implements Displaceable, Area {
   private Point center;
   private int radius;

      …
   public double getArea() {
     return Math.PI * radius * radius;
   }
}
```

Rectangle implementation of Area

```
public class Rectangle
         implements Displaceable, Area {
  private Point lowerLeft;
  private int width, height;
  …

  public double getArea() {
    return width * height;
  }
}
```

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")

Pragmatics: Java identifiers

- Variable, class and method names are identifiers
- Alphanumeric characters or _ starting with a letter or _
  - `size`
  - `myName`
  - `MILES_PER_GALLON`
  - `A1`
  - `the_end`
- Interpretation depends on context: variables and classes can have the same name

Identifier abuse

Class, instance variable, constructor, and method with the *same name*...

```
public class Turtle {

  private Turtle Turtle;

  public Turtle() { }


  public Turtle Turtle (Turtle Turtle) {

     return Turtle;

  }

}
```

Naming conventions

kind	part-of-speech	identifier
class	noun	`RacingCar`
variable	noun	`initialSpeed`
constant	noun	`MAXIMUM_SPEED`
method	verb	`shiftGear`

The OO Style

- Core ideas:
  - objects (state encapsulated with operations)
  - classes ("templates" for object creation)
  - dynamic dispatch ("receiver" of method call determines behavior)
  - subtyping (grouping object types by common functionality)
  - inheritance (creating new classes from existing ones)

- Good for:
  - GUIs!
    - and other complex software systems that include many different implementations of the same "interface" (set of operations) with different behaviors (cf. widgets)
  - Simulations
    - designs with an explicit correspondence between "objects" in the computer and things in the real world