Advanced Object-Oriented Programming

Objects, Classes, and Packages

Kulwadee Somboonviwat

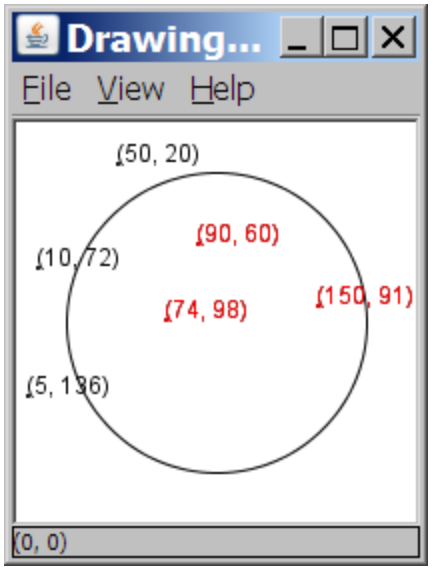
International College, KMITL

kskulwad@kmitl.ac.th

The Bomber Problem

- Given a file of cities' (x, y) coordinates, which begins with the number of cities:

```
6
50 20
90 60
10 72
74 98
5 136
150 91
```



- Write a program to draw the cities on a `DrawingPanel`, then drop a "bomb" that turns all cities red that are within a given radius:

```
Blast site x? 100
Blast site y? 100
Blast radius? 75
Kaboom!
```

A Bad Solution

```
Scanner input = new Scanner(new File("cities.txt"));
int cityCount = input.nextInt();
int[] xCoords = new int[cityCount];
int[] yCoords = new int[cityCount];

for (int i = 0; i < cityCount; i++) {
    xCoords[i] = input.nextInt();   // read each city
    yCoords[i] = input.nextInt();
}
...
```

- **parallel arrays**: 2+ arrays with related data at same indexes.
  - Considered poor style.

Analyzing our first solution

- The data in this problem is a set of points.
- It would be better stored as `Point` objects.
  - A `Point` would store a city's x/y data.
  - We could compare distances between `Point`s to see whether the bomb hit a given city.
  - Each `Point` would know how to draw itself.
  - The overall program would be shorter and cleaner.

Objects

- OOP = Programming using OBJECTS
- An **object** represents an entity in the real world that can be distinctly identified, *e.g. a student, a bank account*

- 3 keys characteristics of objects

  State – defined by data fields
  e.g. a circle object has a radius field to represent its state

  Behavior – defined by methods
  e.g. circle.getArea()

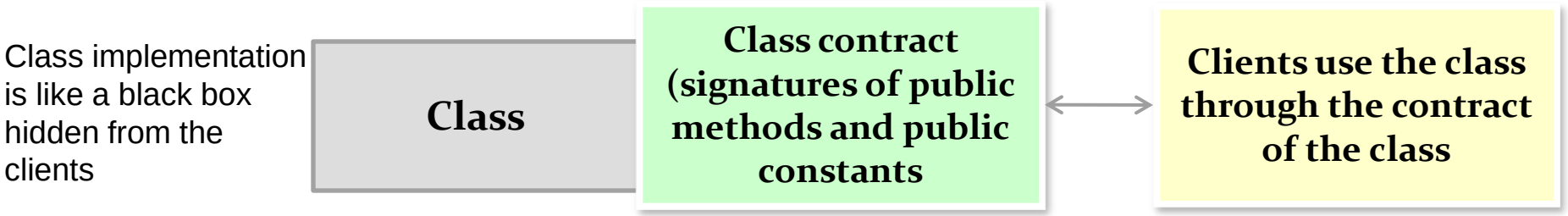
  Identity - each object are distinct even they may have the same state and behavior

Class

- A **class** is a blueprint (or a template) that defines what an object's data fields and methods will be.
  - A class provides **abstraction** of real-world objects
  - An object is an instance of a class
  - Creating an object or an instance is called ***instantiation***

Class Abstraction and Encapsulation

- **Class**
  - An **abstraction** that separates its implementation from its usage
  - An **encapsulation** that hides the details of implementation from the user

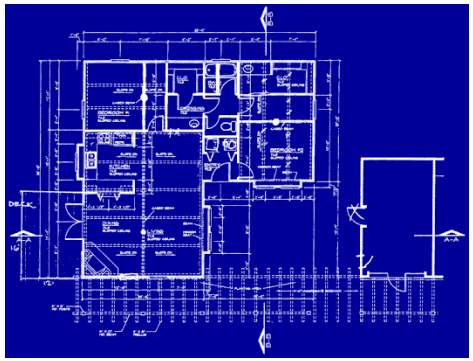


The Blueprint Analogy

iPod blueprint

state:
- current song
- volume
- battery life

behavior:
- power on/off
- change station/song
- change volume
- choose random song



creates

iPod #1

state:
- song = "1,000,000 Miles"
- volume = 17
- battery life = 2.5 hrs

behavior:
- power on/off
- change station/song
- change volume
- choose random song

iPod #2

state:
- song = "Letting You"
- volume = 9
- battery life = 3.41 hrs

behavior:
- power on/off
- change station/song
- change volume
- choose random song

iPod #3

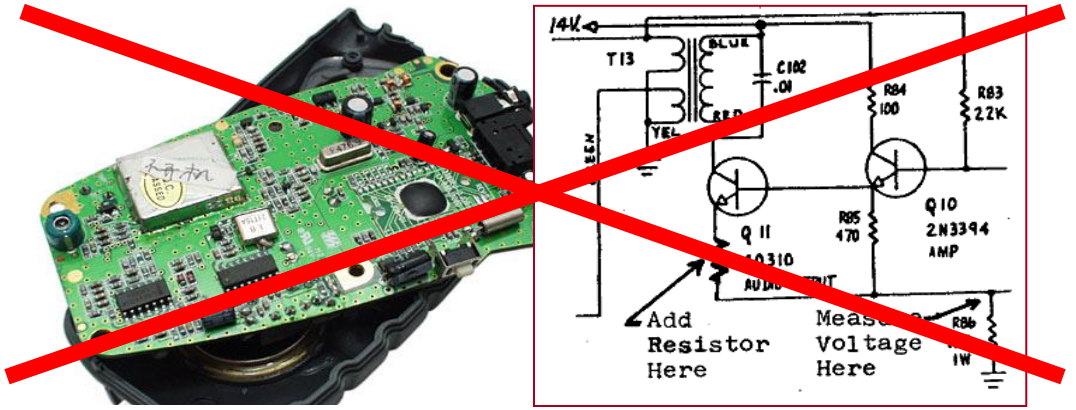
state:
- song = "Discipline"
- volume = 24
- battery life = 1.8 hrs

behavior:
- power on/off
- change station/song
- change volume
- choose random song

Abstraction

- **abstraction**: A distancing between ideas and details.
  - We can use objects without knowing how they work.

- abstraction in an iPod:
  - You understand its external behavior (buttons, screen).
  - You don't understand its inner details, and you don't need to.



Defining the class Point

- Let us implement a `Point` class as a way of learning about defining classes.
  - We will define a type of objects named `Point`.
  - Each `Point` object will contain x/y data called **fields**.
  - Each `Point` object will contain behavior called **methods**.
  - **Client programs** will use the `Point` objects.

Desired Point objects

```
Point p1 = new Point(5, -2);
Point p2 = new Point();          // origin, (0, 0)
```

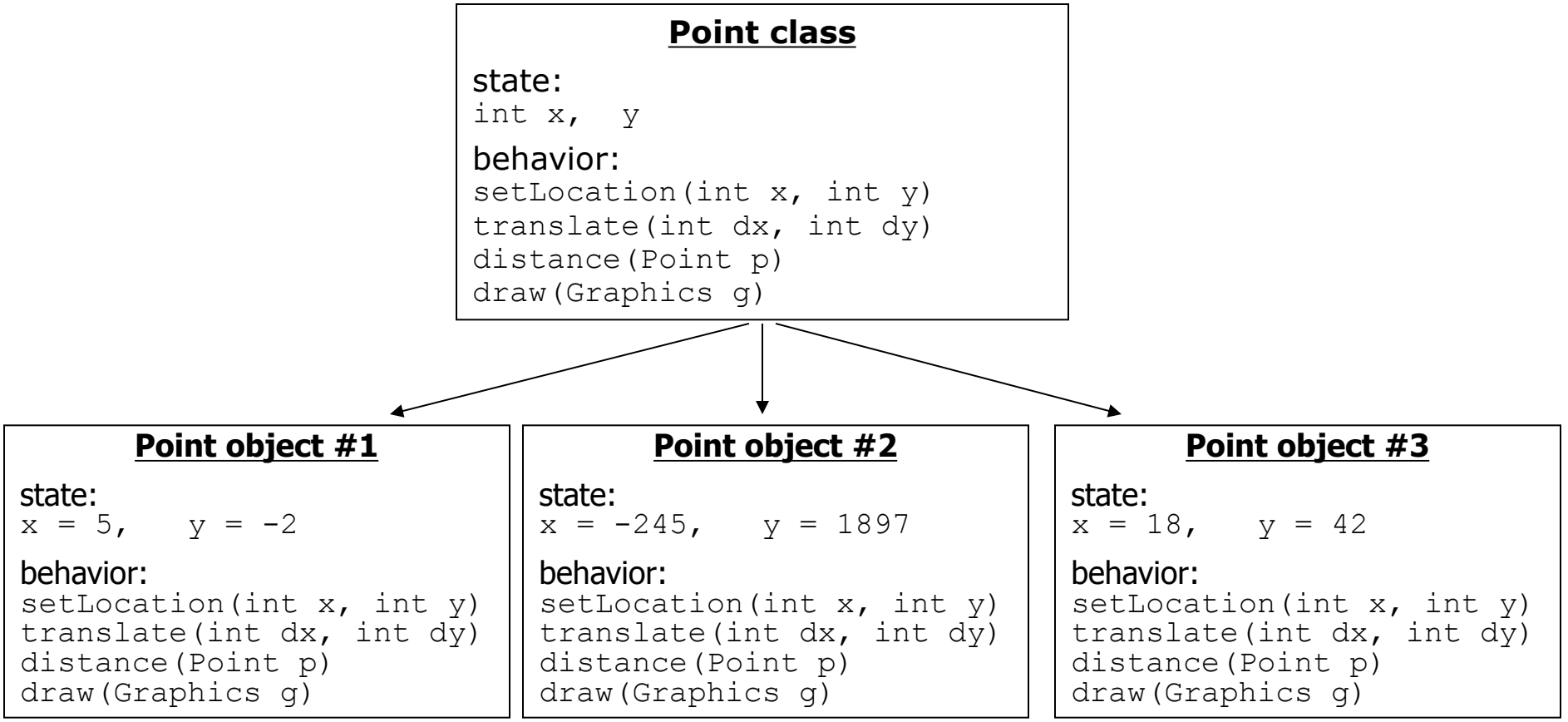
- Data in each `Point` object:

Field name	Description
`x`	the point's x-coordinate
`y`	the point's y-coordinate

- Methods in each `Point` object:

Method name	Description
`setLocation(`**`x, y`**`)`	sets the point's x and y to the given values
`translate(`**`dx, dy`**`)`	adjusts the point's x and y by the given amounts
`distance(`**`p`**`)`	how far away the point is from point *p*
`draw(`**`g`**`)`	displays the point on a drawing panel

The Point class blueprint



- The class (blueprint) will describe how to create objects.
- Each object will contain its own data and methods.

Defining a Class in Java

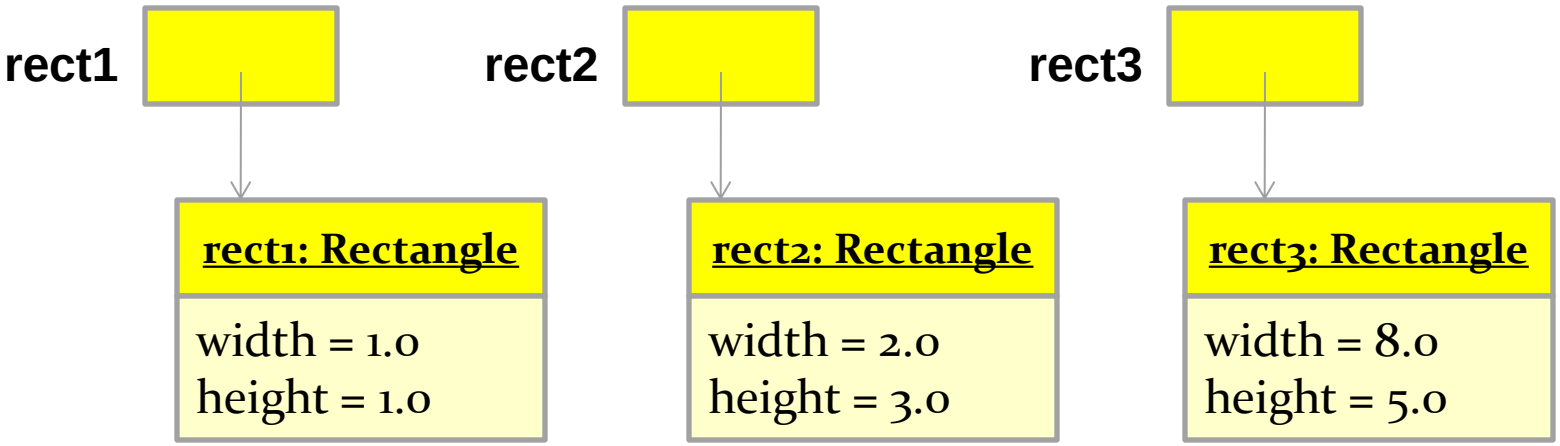
```
class ClassName{
   /* data fields */
   /* constructors */
   /* methods */
}
```

Constructing Objects with the new operator

Syntax:

```
ClassName objectRefVar = new ClassName();
```

```
Rectangle rect1 = new Rectangle();
Rectangle rect2 = new Rectangle(2.0, 3.0);
Rectangle rect3 = new Rectangle(8.0, 5.0);
// rect1, rect2, rect3 are references to objects
// created by the new operator.
```



Sending Messages to Objects with the dot operator

```
Syntax:        // dot operators
               objectRefVar.dataField
               objectRefVar.methodName(arguments)
```

```
Rectangle rect1 = new Rectangle();
Rectangle rect2 = new Rectangle(2.0, 3.0);
Rectangle rect3 = new Rectangle(8.0, 5.0);

System.out.println("Rect2 width is " + rect2.width);
System.out.println("Rect2 height is " + rect2.height);
System.out.println("Rect2 area is " + rect2.getArea());
```

The Point Class and its clients

```
public class Point {
    int x;
    int y;
// Changes the location of this Point object.
    public void draw(Graphics g) {
        g.fillOval(x, y, 3, 3);
        g.drawString("(" + x + ", " + y + ")", x, y);
    }
    public void setLocation(int newX, int newY) {
        x = newX;
        y = newY;
    }

    public void translate(int dx, int dy) {
        setLocation(x + dx, y + dy);
    }
    public double distance(Point other) {
        int dx = x - other.x;
        int dy = y - other.y;
        return Math.sqrt(dx * dx + dy * dy);
    }
}
```

The Point Class and its clients

PointMain.java (client program)

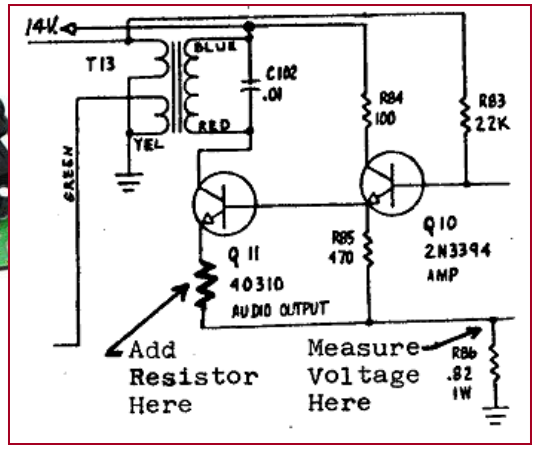
```
public class PointMain {
  main(String args) {
    Point p1 = new Point();
    p1.x = 7;
    p1.y = 2;

    Point p2 = new Point();
    p2.x = 4;
    p2.y = 3;
    ...
  }
}
```

Point.java (class of objects)

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

Encapsulation

- **encapsulation**: Hiding implementation details from clients.
  - Encapsulation forces *abstraction*.
    - separates external view (behavior) from internal view (state)
    - protects the integrity of an object's data



Private Fields / Methods

A field that cannot be accessed from outside the class

`private` type name;

- Examples:

```
private int id;
private String name;
```

- Client code won't compile if it accesses private fields:

```
PointMain.java:11: x has private access in Point
System.out.println(p1.x);
                   ^
```

Accessing Private fields with getter/setter methods

```
// A "read-only" access to the x field ("accessor")
public int getX() {
    return x;
}

// Allows clients to change the x field ("mutator")
public void setX(int newX) {
    x = newX;
}
```

– Client code will look more like this:

```
System.out.println(p1.getX());
p1.setX(14);
```

The new better version of Point class

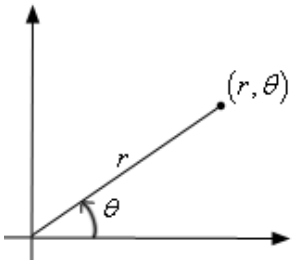
```
// A Point object represents an (x, y) location.
public class Point {
    private int x;
    private int y;

    public Point(int initialX, int initialY) {
        x = initialX;
        y = initialY;
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }

    public double distanceFromOrigin() {
        return Math.sqrt(x * x + y * y);
    }

    public void setLocation(int newX, int newY) {
        x = newX;
        y = newY;
    }

    public void translate(int dx, int dy) {
        setLocation(x + dx, y + dy);
    }
}
```

Benefits of Encapsulation

- Abstraction between object and clients

- Protects object from unwanted access
  - Example: Can't fraudulently increase an Account's balance.

- Can change the class implementation later
  - Example: Point could be rewritten in polar coordinates (r, θ) with the same methods.



- Can constrain objects' state (**invariants**)
  - Example: Only allow Accounts with non-negative balance.
  - Example: Only allow Dates with a month from 1-12.

Data and method visibility

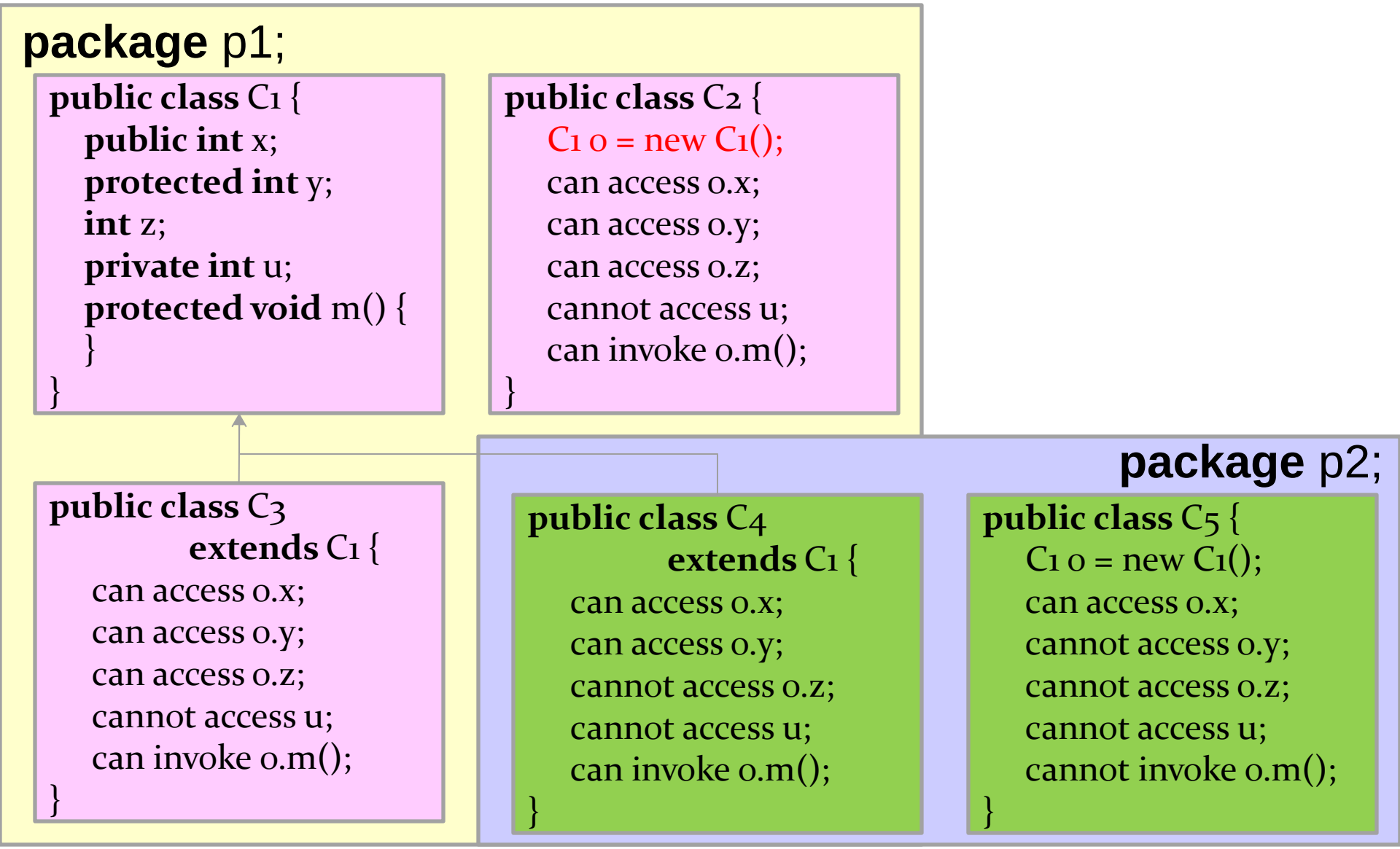
- Besides the **private** keyword, java also provides three other types of keywords for controlling data and method visibility:
  - public, protected, default(package)

Data and Methods Visibility (1/2)

Modifier on members in a class	Accessed from the same class	Accessed from the same package	Accessed from a subclass	Accessed from a different package
public	○	○	○	○
protected	○	○	○	×
(default) *	○	○	×	×
private	○	×	×	×

* **default** access has no modifier associated with it

Data and Methods Visibility (2/2)



Valid Application of Visibility Modifiers

Modifier	Class	Constructor	Method	Data	block
(default)*	O	O	O	O	O
public	O	O	O	O	x
protected	x	O	O	O	x
private	x	O	O	O	x

* **default** access has no modifier associated with it

this keyword

- **`this`** : Refers to the implicit parameter inside your class.
  (a variable that stores the object on which a method is called)

  – Refer to a field: `this.`**field**

  – Call a method: `this.`**method**`(`**parameters**`);`

  – One constructor can call another: `this(`**parameters**`);`

Variable shadowing

- **shadowing**: 2 variables with same name in same scope.
  - Normally illegal, except when one variable is a field.

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

    ...

    // this is legal
    public void setLocation(int x, int y) {
        ...
    }
```

  - In most of the class, `x` and `y` refer to the fields.
  - In `setLocation`, `x` and `y` refer to the method's parameters.

Fixing the Variable shadowing

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

    ...

    public void setLocation(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

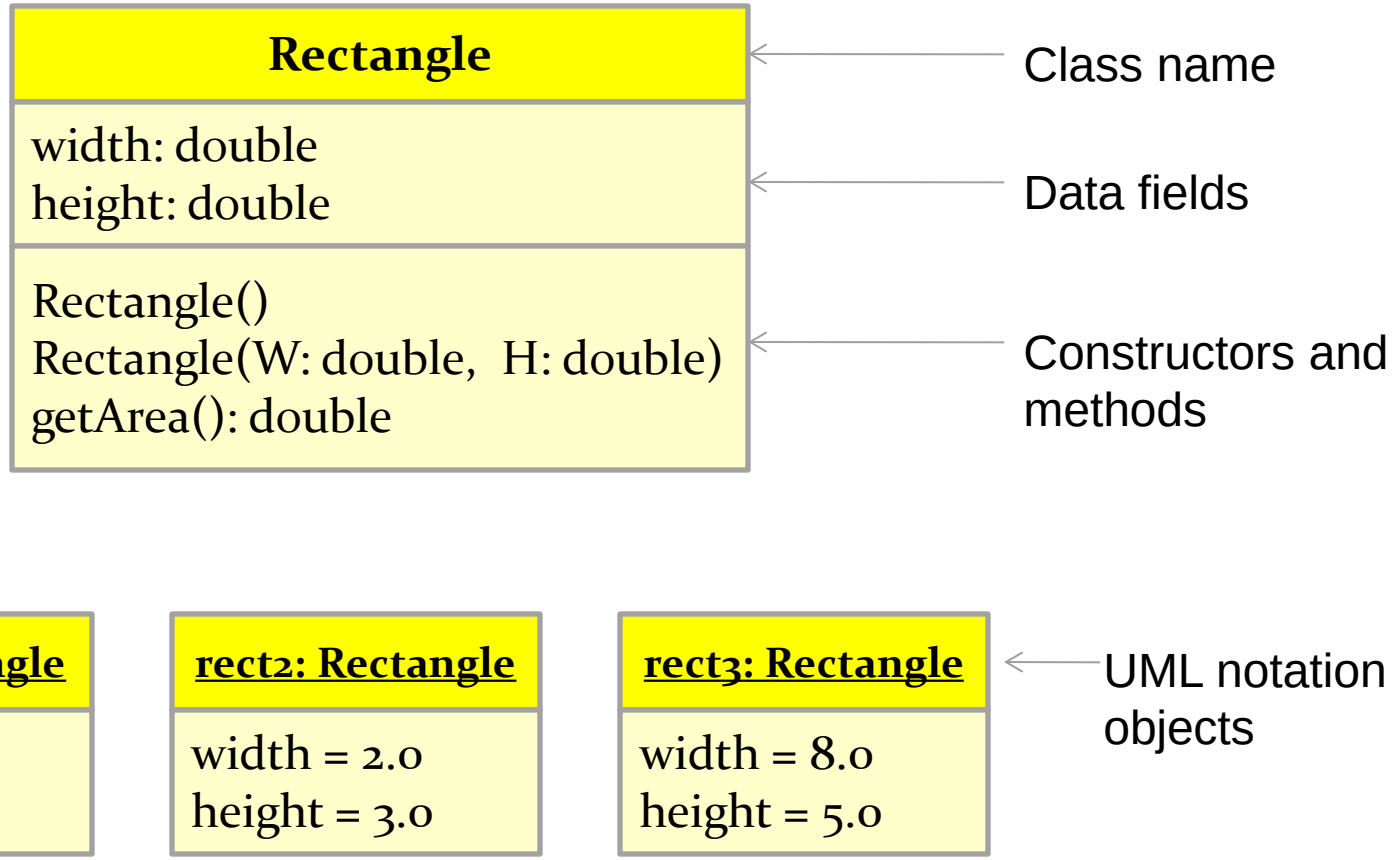
- Inside `setLocation`,
  - To refer to the data field `x`, say `this.x`
  - To refer to the parameter `x`, say `x`

Calling another constructor

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

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

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

    ...
}
```

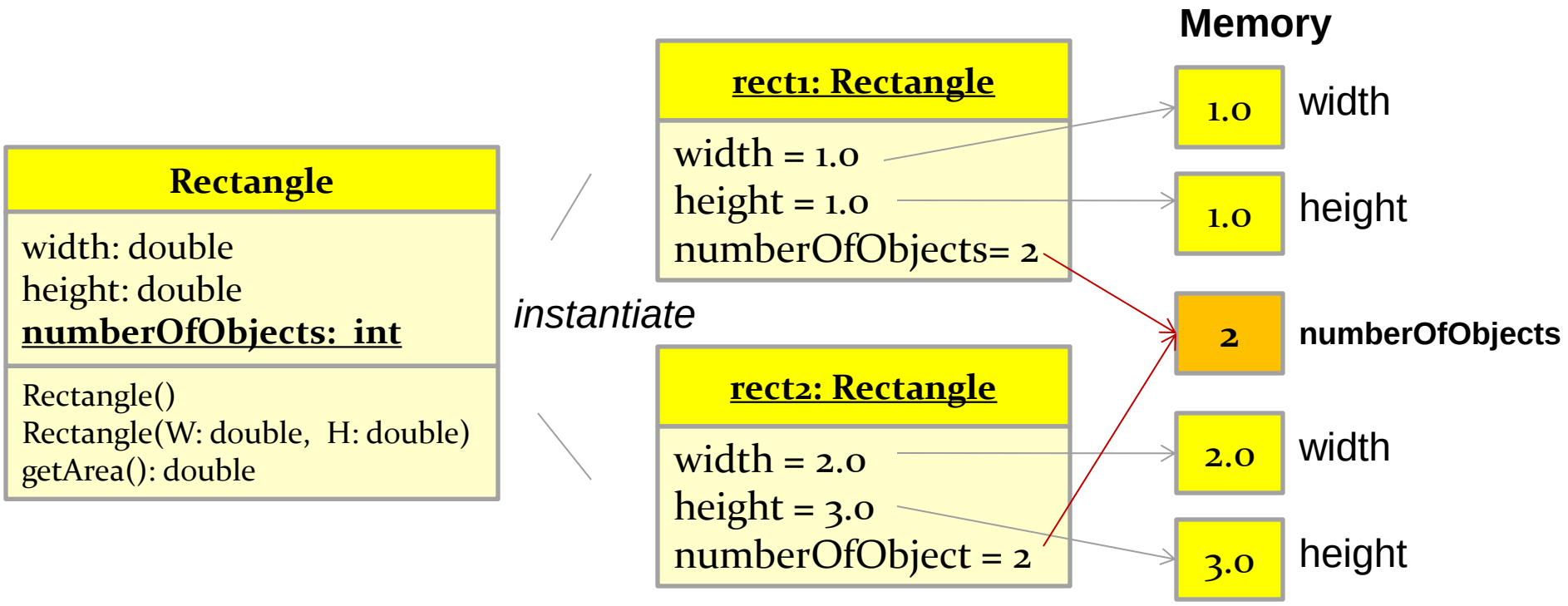
- Avoids redundancy between constructors
- Only a constructor (not a method) can call another constructor

UML Class Diagram



Instance vs. Class Fields (or Methods)

- An **instance field or method** belongs to an instance of a class.
- A **static field or method** is shared by all instances of the same class, and can be invoked without using an instance



Example: class Rectangle

```
class Rectangle {
   /* data fields */
   double width = 1.0;
   double height = 1.0;
   /* constructors */
   Rectangle() {          }
   Rectangle( double W, double H) {
       width = W; height = H;
   }
   /* methods */
   double getArea() {
       return width*height;
   }
}
```

Adding a static field and method to our class

```
class Rectangle {
   /* data fields */
   double width = 1.0;
   double height = 1.0;
   static int numberOfObjects;
   /* constructors */
   Rectangle() {  }
   Rectangle( double W, double H) {
       width = W; height = H;
   }
   /* methods */
   double getArea() { return width*height; }
   static int getNumberOfObjects() {
       return numberOfObjects;
   }
}
```

Using static fields and methods

syntax: **ClassName.staticDataField**

ClassName.staticMethodName()

```
public static void main(String[] args)
{
   Rectangle rect1 = new Rectangle();
   Rectangle rect2 = new Rectangle(2.0,3.0);
   System.out.println("There are " +
       Rectangle.numberOfObjects +
       " rectangles.");
   // or we can call the static method
   System.out.println("There are " +
       Rectangle.getNumberOfObjects() +
       " rectangles.");
}
```

Summary: objects and classes

- A class is a template for objects.
  - declared by a **class** keyword and a class name
  - class declaration is populated with a combination of field, method, and constructor declarations
- An object is an instance of a class.
  - Use the **new operator** to create an object
  - Use the **dot operator** to access fields and methods
- A **field** is a variable that stores a value of an object's attribute
- A **method** is a named block of code with an optional list of arguments and a return value.
- **instance fields/methods** : associated with individual objects
- **static fields/methods** : shared by all objects of the same class

Packages

- Packages are used to organize classes
- All standard Java packages are inside the **java** and **javax** package hierarachies
- Uses packages to guarantee the uniqueness of class names
- To put a class into a package add

  package packageName;

  as the first non-comment and non-blank statement in the program

Using Public Class from other packages

- add the full package name in front of *every class name*

  java.util. Date aday = new **java.util.**Date();

- use the *import statement*

  import java.util.*;

  Date aday = new Date();

  Or

  import java.util.Date;

  Date aday = new Date();

The *java.util* package

http://docs.oracle.com/javase/6/docs/api/java/util/package-summary.html

Package java.util

Contains the collections framework, legacy collection classes, event model, date and time facilities, internationalization, and miscellaneous utility classes (a string tokenizer, a random-number generator, and a bit array).

See:

Description

Interface Summary	
Collection<E>	The root interface in the *collection hierarchy*.
Comparator<T>	A comparison function, which imposes a *total ordering* on some collection of objects.
Deque<E>	A linear collection that supports element insertion and removal at both ends.
Enumeration<E>	An object that implements the Enumeration interface generates a series of elements, one at a time.
EventListener	A tagging interface that all event listener interfaces must extend.
Formattable	The `Formattable` interface must be implemented by any class that needs to perform custom formatting using the `'s'` conversion specifier of `Formatter`.
Iterator<E>	An iterator over a collection.
List<E>	An ordered collection (also known as a *sequence*).
ListIterator<E>	An iterator for lists that allows the programmer to traverse the list in either direction, modify the list during iteration, and obtain the iterator's current position in the list.
Map<K,V>	An object that maps keys to values.
Map.Entry<K,V>	A map entry (key-value pair).
NavigableMap<K,V>	A `SortedMap` extended with navigation methods returning the closest matches for given search targets.
NavigableSet<E>	A `SortedSet` extended with navigation methods reporting closest matches for given search targets.
Observer	A class can implement the `Observer` interface when it wants to be informed of changes in observable objects.

The *java.util.Date* class source code

http://javasourcecode.org/html/open-source/jdk/jdk-6u23/java/util/Date.java.html

```
/*
 * %W% %E%
 *
 * Copyright (c) 2006, Oracle and/or its affiliates. All rights r
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license ter
 */

package java.util;

import java.text.DateFormat;
import java.io.IOException;
import java.io.ObjectOutputStream;
import java.io.ObjectInputStream;
import java.lang.ref.SoftReference;
import sun.util.calendar.BaseCalendar;
import sun.util.calendar.CalendarDate;
import sun.util.calendar.CalendarSystem;
import sun.util.calendar.CalendarUtils;
import sun.util.calendar.Era;
import sun.util.calendar.Gregorian;
import sun.util.calendar.ZoneInfo;

/**
 * The class <code>Date</code> represents a spec
 * in time, with millisecond precision.
```

```
public class Date
    implements java.io.Serializable, Cloneable, Comparable<Date>
{
    private static final BaseCalendar gcal =
                CalendarSystem.getGregorianCalendar();
    private static BaseCalendar jcal;

    private transient long fastTime;

    /*
     * If cdate is null, then fastTime indicates the time in millis.
     * If cdate.isNormalized() is true, then fastTime and cdate are in
     * synch. Otherwise, fastTime is ignored, and cdate indicates the
     * time.
     */
    private transient BaseCalendar.Date cdate;

    // Initialized just before the value is used. See parse().
    private static int defaultCenturyStart;

    /* use serialVersionUID from modified java.util.Date for
     * interoperability with JDK1.1. The Date was modified to write
     * and read only the UTC time.
     */
    private static final long serialVersionUID = 7523967970034938905L;

    /**
     * Allocates a <code>Date</code> object and initializes it so that
     * it represents the time at which it was allocated, measured to the
     * nearest millisecond.
     *
     * @see     java.lang.System#currentTimeMillis()
     */
    public Date() {
        this(System.currentTimeMillis());
    }
```

Java API doc. and Source code

- The best resources for learning the Java language are the API documentation and its source code.
- You can download them from Oracle's Java SE site and store on your computer for offline browsing
- Or, browse the online version at
  - http://docs.oracle.com/javase/7/docs/api/
  - http://javasourcecode.org/