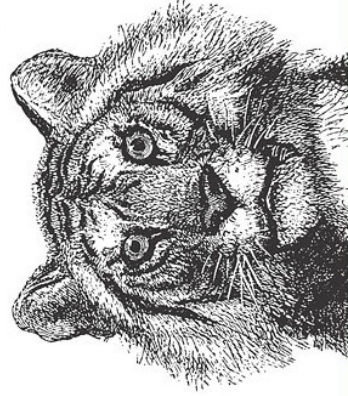


CLASS DESIGN

Today

- Today we will talk about the basics of defining classes in Java.
- In particular we will look at:
 - Class definition
 - Access to data fields
 - Constructors
 - Finalizers
 - Methods
 - Class fields and methods
 - Subclasses



- Again, lots of this material is drawn from *Java in a Nutshell*, David Flanagan, O'Reilly ...

A first class

- Here is the definition of a simple class that we can use to hold *x* and *y* coordinates.
- As we will see, this can be useful in writing programs that display graphic elements.

```
public class Point{
    public int x, y;
}
```
- This defines a kind of datatype called `Point` which holds two *fields* called *x* and *y* which are both `ints`.

- Once we have defined the class, we can use it.
- The following code creates an *instance* of the class:


```
Point p;
p = new Point();
```
- The first line creates a reference to a `Point`.
- The second creates a `Point` *object* and makes `p` refer to it.
 - Note the syntax: `Point` is followed by a pair of parentheses.
- We can combine the two lines into:


```
Point p = new Point();
```

- Since `x` and `y` are `public`, we can handle them as follows.
- Setting their value:


```
p.x = 3;
```
- Reading their value:


```
System.out.println(p.x);
```
- Changing their value:


```
p.y = p.x + 3;
```
- However, having `public` fields is generally considered bad programming practice.

A second class

- Here is a better `Point` class:


```
public class Point{
    private int x, y;
}
```
- However, making the fields `private` means you can't access them directly.
 - That is what `private` means.
- The solution is to write `public` functions to read and write `private` fields.
 - A simple API

- For example, we can add these functions to the definition of `Point`.


```
public void setX(int x){
    this.x = x;
}

public int getX(){
    return x;
}
```
- For the full definition of `Point` see `Point.java` on the course website.

- Note the use of `this` in `setX` to disambiguate between `x`, the parameter, and `this.x` the field.
- `this` is a reference that refers to the object where the `this` occurs.
- We don't need to use `this` in `getX` because there is no ambiguity.

Constructors

- When we create a new object, we make a call like:
`Point p = new Point();`
- The reason for the parentheses above is that creating the `Point` involves calling a function.
– Constructor
- Java will create a constructor for every class we define.
- Often we want to make our own.

- Here is a constructor for `Point`:

```
public Point () {
    this.x = 0;
    this.y = 0;
}
```
- The point of constructors is to combine:
 - Object creation
 - Object initialisation.

- We are allowed to write multiple constructors provided that they have different numbers and/or types of parameters.
 - Java uses the combination of parameters to distinguish between constructors.
- Here is another:

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

- We can invoke one constructor from another, for example:

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

- The second line does what we would expect — it uses the parameter to set the value of the `x` field.
- The first line uses `this ()` to invoke the constructor with no parameters.

- We could also have written this:

```
public Point (int x) {  
    this (5, 5);  
    this.x = x;  
}
```

where the first line would have invoked the constructor with two parameters.

- If you are going to use `this` to invoke another constructor in this way, it has to be the *first* line in the constructor it is called in.
- It even has to go before any local variable definitions.

Finalizers

- In C++ we often need to define destructors to release memory when an object is deleted.
- In Java we don't have to worry about this
 - Garbage collection.
- However, sometimes there are things we need to do when an object is no longer being used.
 - Close a file or a socket.
- We define a function `finalize()` to do whatever is required.
- `finalize()` is always declared `public` or `protected` (We will get too protected shortly).

Composition

- *Composition* is using classes as fields within another class.
- For example, here is a class that uses `Point`:

```
public class Circle{
    private Point location;
    private double radius
}
```
- Since the fields are private, we have to write API methods for them.
- The ones to access `location` can use the `Point` methods:

```
public void setLocationX(int x){
    location.setX(x);
}
```

and so on.

- Let's also write a constructor for `Circle`:

```
public Circle(){
    location.setX(1);
    location.setY(1);
    radius = 1.0;
}
```

- Note that even in the constructor we have to use the API functions for `location` since its `x` and `y` values are private to it (the `Point`).

Class fields and methods

- When dealing with circles it is helpful to have the value of `pi` easily accessible.
- So let's add this:

```
public static final double PI = 3.1415927
```

to `Circle`
- We use `final` because we are defining a constant.
- More important here is the fact that we use the keyword `static`.
- In this context it defines a *class field*.

- That is a field that is associated with the class, not with a specific instance/object.
 - Just one copy of it.
- Since it is associated with the class, it is available to every object in the class.

- In addition, since (in this case) it is `public`, it is available to objects outside the class.
 - A private class field would *only* be available to objects of that class.
- For those other objects to access it, they must use its full name: `Circle.PI`.
- Public class fields are rather like global variables.
 - But the need to use the full name ameliorates the name clash problem.

- We can also have class methods, again declared using the `static` modifier:


```
public static double radsToDegs(double radians){
    return radians * 180/PI;
}
```
- Note that class methods can use class fields, but *not* the regular fields (the non-class fields).

Subclasses

- Here is a sub-class of `Circle`

```
public class PCircle extends Circle{
    private Point origin;

    public PCircle() {
        super();
        this.origin.setX(0);
        this.origin.setY(0);
    }

    public PCircle(int x, int y) {
        super();
        this.origin.setX(0);
        this.origin.setY(0);
        this.location.setX(x);
        this.location.setY(y);
    }
}
```

- The use of the `extends` keyword defines `PCircle` to be a *subclass* of `Circle`.
- Equally, `Circle` is a superclass of `PCircle`.
- Can create a subclass of any existing class:
 - Java library class
 - User defined

- As a sub-class, `PCircle` *inherits* all of the methods and fields of `Circle`.
 - Hence the references to `location`
- It also contains new fields that are declared in `PClass`.
- Note the use of `super ()` in the constructors.
- This is a reference to the constructor for `Circle`.
- Like `this ()`, it has to be the first thing in the constructor
 - Before local variable declarations.

Summary

- This lecture introduced the basics of classes and objects:
 - Defining and instantiating a class.
 - Constructors and finalizers
 - Subclasses
- We will cover more advanced topics on classes next week.