Working with Classes

Atul Prakash

Class

- Corresponds to real-world entities, such as student, course, dog, animal, shape, etc., which are represented inside a program
- E.g., a class of students
- Objects or class instances: individual instances that belong to the class

Filing Cabinet Analogy

- Think of a class as a drawer in a filing cabinet
- Think of objects as folders within the drawer. Each folder has information about the object
- Folders can be added/removed over time

Example

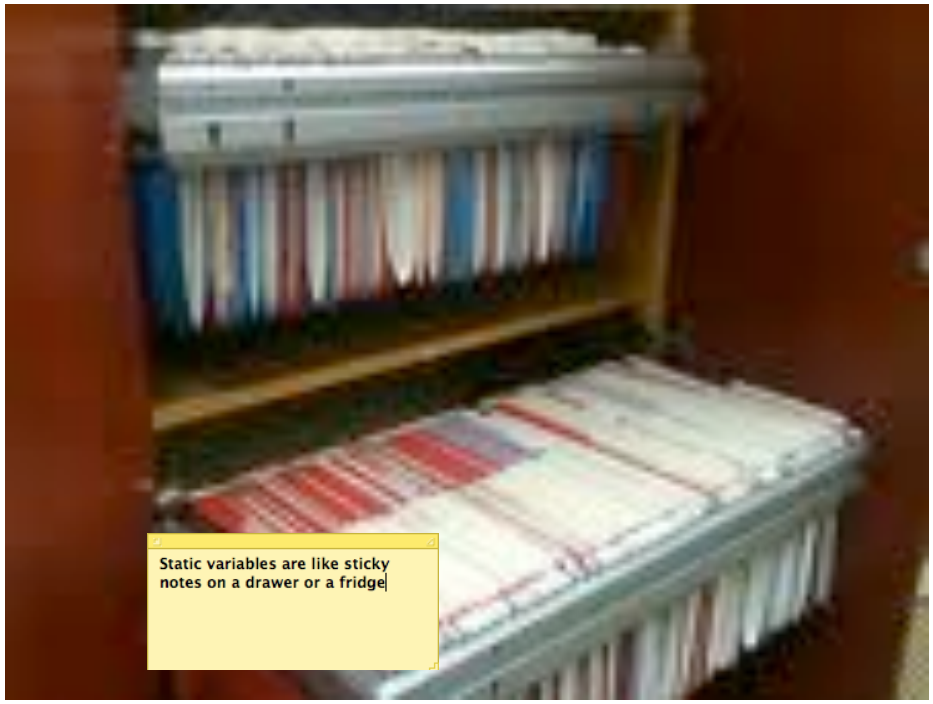
```
public class Bicycle {
    public int gear;
    public int maxspeed;
}

/* Create two bicycles */

.... this code should be in main or another function ...

Bicycle b1 = new Bicycle();
Bicycle b2 = new Bicycle();
b1.gear = 10;
b1.maxspeed = 40;
b2.gear = 1;
b2.maxspeed = 20;
```

The above code defines a drawer for containing information about bicycles. We created two bicycles with different gears and maximum speeds, respectively.

Static variables

- In a filing cabinet, there is often information that is specific to a drawer, not a folder. In our example:
  - Total number of bicycles created
  - Maximum possible speed of a bicycle
- Static variables are drawer-specific, rather than folder-specific.

Static variables - sticky notes on a drawer



Representation in Java

```
public class Bicycle {
    public int gear;
    public int maxspeed;
    public static int numberOfBicycles  = 0;

}
```

```
Bicycle b1 = new Bicycle();
Bicycle.numberOfBicycles++;
Bicycle b2 = new Bicycle();
Bicycle.numberOfBicycles++;
b1.gear = 10;
b1.maxspeed = 40;
b2.gear = 1;
b2.maxspeed = 20;
```

Which variables are object-specific, which ones are global to the whole class?

Constructors

- We want an invariant that the number of bicycles created is always equal to the numberOfBicycles.
- But, right now, there is no guarantee of that

```
public class Bicycle {
    public int gear;
    public int maxspeed;
    public static int numberOfBicycles  = 0;
}
// code that uses the class...
Bicycle b1 = new Bicycle();
Bicycle b2 = new Bicycle();
Bicycle.numberOfBicycles++;
```

Two bikes were created,
but the user forgot to bump up
the count

Constructors provide safety

```
public class Bicycle {
    public int gear;
    public int maxspeed;
    private static int numberOfBicycles = 0;

    public Bicycle(int g, int s) {
        this.gear = g;
        this.maxspeed = s;
        Bicycle.numberOfBicycles++;
    }
}
// Use
Bicycle b1 = new Bicycle(10,40);
Bicycle b2 = new Bicycle(1, 20);
// Correct count will be printed
System.out.println(Bicycle.numberOfBicycles);
```

- Constructors are public functions within the class with the same name as the class name.
- Parameters can be passed in to initialize the object

Details

```
public class Bicycle {
    public int gear;
    public int maxspeed;
    public static int numberOfBicycles = 0;

    public Bicycle(int g, int s) {
        this.gear = g;
        this.maxspeed = s;
        Bicycle.numberOfBicycles++;
    }
}
```

g and s are parameters

this.gear means this object's gear value, etc.

this is a keyword in the language

Variables in classes

```
public class Bicycle {
    public int gear;
    public int maxspeed;
    public static int numberOfBicycles = 0;

    public Bicycle(int g, int s) {
        gear = g;
        maxspeed = s;
        numberOfBicycles++;
    }
}
```

OK to omit "this" in most places. The system figures out that gear refers to this bicycle's gear, etc.

Innermost rule

```
public class Bicycle {
    public int gear;
    public int maxspeed;
    public static int numberOfBicycles = 0;

    public Bicycle(int gear, int s) {
        this.gear = gear;
        maxspeed = s;
        numberOfBicycles++;
    }
}
```

this.gear -> bicycle's gear variable
gear -> local parameter gear
maxspeed -> bicycle's maxspeed, since no local variable with the same name

- Here, this.gear is necessary. There are two variables called gear in the constructor. Without "this", the innermost definition of gear wins.

Class invariants

- We may want to guarantee certain properties for all objects in a class. For example, suppose we want all bikes to have the following constraints:
  - 0 <= gears
  - 0 <= maxspeed <= 150

Problem

- A user of the class can violate the constraints easily by accessing the class variables directly

```
public class Bicycle {
    public int gear;
    public int maxspeed;
...
}

/* Create two bicycles */

.... this code should be in main or another function ...

Bicycle b1 = new Bicycle(10, 40);
b1.gear = -100;           // bad value, but legal in Java
b1.maxspeed = 5000;
```

Private variables and public methods

- Solution: make variables private in the class
  - Provide safe public methods for users to read/write the variables

Using private variable and public methods

```
public class Bicycle {
    private int gear;
    private int maxspeed;
    public static int numberOfBicycles = 0;

    public Bicycle(int g, int s) {
        assert(g >= 0);
        assert(s >= 0 && s <= 150);
        gear = g;
        maxspeed = s;
        numberOfBicycles++;
       // invariant should be true at this point
    }

    public boolean setSpeed(int s) {
        if (s < 0 || s > 150) {
            return false;
        }
        maxspeed = s;
        return true;
    }

    public int getSpeed() {
        return maxspeed;
    }
```

```
    public boolean setGear(int g)
        if (g < 0) return false;
        gear = g;
        return true;
    }

    public int getGear() {
        return gear;
    }

} // end class
```

gear and *maxspeed* made private. Added setters and getters. (Eclipse can generate setters and getters.)

Parsing a Method

```
public boolean setSpeed(int s) {
    if (s < 0 || s > 150) {
        return false;
    }
    maxspeed = s;
    return true;
}
```

- setSpeed takes a parameter s of type integer as an argument
- It returns a boolean value as a result

Getters and Setters

- Methods to read private variables are called getters. Usually, named get*VariableName*().
- Methods to set private variables safely are called setters. Usually, named set*VariableName*().

Public versus Private

- private: visible only within the class
  - private String myname;
- public: visible outside the class and package
  - public String getName()

With gear and maxspeed as private:

- Following is possible:
  - Bicycle b = new Bicycle(10, 60);
  - b.gear = -2; // should be illegal
  - b.maxspeed = -10; // should be illegal

Methods

- Besides getters and setters, additional methods can exist to access or modify the state of an object. In the filing drawer analogy, think of a method as a standard recipe attached to each folder that explains how to access or modify the folder data
- Public methods: intended for outsider use
- Private methods: intended as helper functions to assist the public methods. Not accessible directly from outside the class

Testing Invariants

- How do we know that we implemented setters correctly, respecting the invariants?
- Test the invariants at the end of the constructor and at start and exit from each public method

Adding invariants for safer code

```
public boolean setSpeed(int s) {
    testInvariants();
    if (s < 0 || s > 150) {
        testInvariants();
        return false;
    }
    maxspeed = s;
    testInvariants();
    return true;
}

private void testInvariants() {
    assert (maxspeed >= 0);
    assert (maxspeed <= 150);
    assert (gear >= 0);

}
```

If you forget a check in setSpeed, hopefully, the invariant checks will catch the error

Constant variables

- Variables can be declared as "final" to tell Java that they will not change in value.
    - public static final double PI = 3.14159265358979323846;

Example

- double x = Math.PI;
- Declaration of PI within the Math class:
  - public static final double PI = 3.14159265358979323846;

Class methods

- You may need methods that are associated with the drawer, rather than a particular folder within the drawer
- For example, reading the number of bicycles, which is a static variable
  - Declare a method as "static" to indicate that it is not object-specific, but a class method

```
public class Bicycle {
    private int gear;
    private int maxspeed;
    private static int numberOfBicycles;

    public static int getNumberOfBicycles() {
        // int x = this.gear; // illegal
       return numberOfBicycles;
    }

    public Bicycle(int g, int s) {
        gear = g;
        maxspeed = s;
        numberOfBicycles++;
    }
    … rest same as before …
}
// usage:
int count = Bicycle.getNumberOfBycles();
```

Note the word static in getNumberOfBicycles()

Functions

- In Python and C++, you are used to simply using functions that do not belong to a particular class
- Java does not permit functions to be outside a class
- But, functions can emulated in Java by using static methods and placing them in a class like “Global” or in any class of your choice and making them public

Example

```
double root = Math.sqrt (17.0);
double angle = 1.5;
double height = Math.sin (angle);
```

sqrt

```
public static double sqrt(double a)
```

Returns the correctly rounded positive square root of a `double` value. Special cases:

- If the argument is NaN or less than zero, then the result is NaN.
- If the argument is positive infinity, then the result is positive infinity.
- If the argument is positive zero or negative zero, then the result is the same as the argument.

Otherwise, the result is the `double` value closest to the true mathematical square root of the argument value.

Parameters:
`a` - a value.

Returns:
the positive square root of `a`. If the argument is NaN or less than zero, the result is NaN.

sqrt and sin functions in Math class: They are class methods, not object methods. They can be used without creating Math objects.

Code for the Math class

http://www.docjar.com/html/api/java/lang/Math.java.html

Or google search for "Math java source"

main program in Java

- To run a Java program:
  java <Classname>
- Java executes <Classname>.main() method.
  - Must be declared static since it is class-specific, not object-specific
- When the method completes, the program ends.
- It is OK for multiple classes to have main() methods. E.g., SpaceshipGame and SpaceshipGameTest. Only one is executed.

javadoc

- Java provides an automatic HTML documentation generator from code.
- It generates HTML from the code + comments
- For it to work, the comments must follow a certain style
- Javadoc enclosed within /** and */

Example

Source code

```
124       /**
125        * Returns the trigonometric cosine of an angle. Special cases:
126        * <ul><li>If the argument is NaN or an infinity, then the
127        * result is NaN.</ul>
128        *
129        * <p>The computed result must be within 1 ulp of the exact result.
130        * Results must be semi-monotonic.
131        *
132        * @param   a   an angle, in radians.
133        * @return  the cosine of the argument.
134        */
135       public static double cos(double a) {
136           return StrictMath.cos(a); // default impl. delegates to StrictMath
137       }
138
```

HTML docs

cos

`public static double` **`cos`**`(double a)`
Returns the trigonometric cosine of an angle. Special cases:

- If the argument is NaN or an infinity, then the result is NaN.

The computed result must be within 1 ulp of the exact result. Results must be semi-monotonic.

Parameters:
`a` - an angle, in radians.
Returns:
the cosine of the argument.

Special tags in Javadoc

- @param <tab> parameter <tab> description
- @return <tab> description of what is returned
- @author <tab> name of the author

The above show up properly formatted in HTML

Eclipse and Javadoc

- Eclipse can automatically insert Javadoc-style comment stub for you.
- Select a method name, go to Source -> Generate Element Comment
- To generate javadoc files, do Project -> generate Javadoc...
- Browse to the project's doc directory to see the generated HTML files

Command-line Javadoc

- Use the javadoc command
  - E.g., % javadoc <Java files>
- More details on Javadoc. Read: http://java.sun.com/j2se/1.5.0/docs/tooldocs/windows/javadoc.html

Object References

- Dog d;
  - d is a reference to a dog. It is not the actual dog.
- d = null;
  - null is a special object to help initialize references. Like 0.
- d = new Dog("Fido", "woof");
  - d now refers to a dog object

d → 

Dog class code

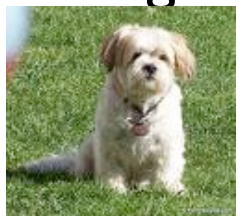
```
public class Dog {
    String name;
    String bark;

    public Dog(String name, String bark) {
        this.name = name;
        this.bark = bark;
    }

    public void bark() {
        System.out.println(bark);
    }
}
```

Constructor method

method on a dog

Dog d; // Does not create a dog. Just a reference to dog.
d = new Dog("Fido", "woof");
d.bark();

d →



Creating Objects

- Usually, done using the new operator.
  - Point p = new Point(10, 15);
  - Dog d = new Dog("fido", "woof");

d refers to an object of type Dog

create a new dog object whose name is fido and who woofs

Assigning Object References

- Dog d1, d2;
  - Creates two object references
- d1 = new Dog("Fido", "woof");

d1 ⟶ 

- d2 = d1
  - Only copies the reference, not the object

d1 ⟶
d2 ⟶ 

Aliasing of references

d1 ⟶
d2 ⟶

name: "Fido"
bark: "woof"

- d1.setBark("huff");

d1 ⟶
d2 ⟶

name: "Fido"
bark: "huff"

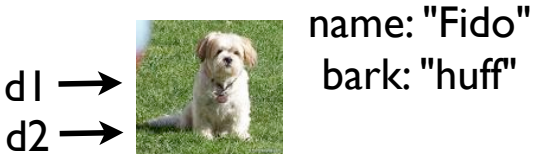
- What are d1.getBark() and d2.getBark()?

Java Variables

- All variables in Java are really pointers or references to objects.
  - *Exceptions:* variables of primitive types, such as int, boolean, float, double, long, short, etc.
- Array variables are always references

Example: Primitive Types vs. object types

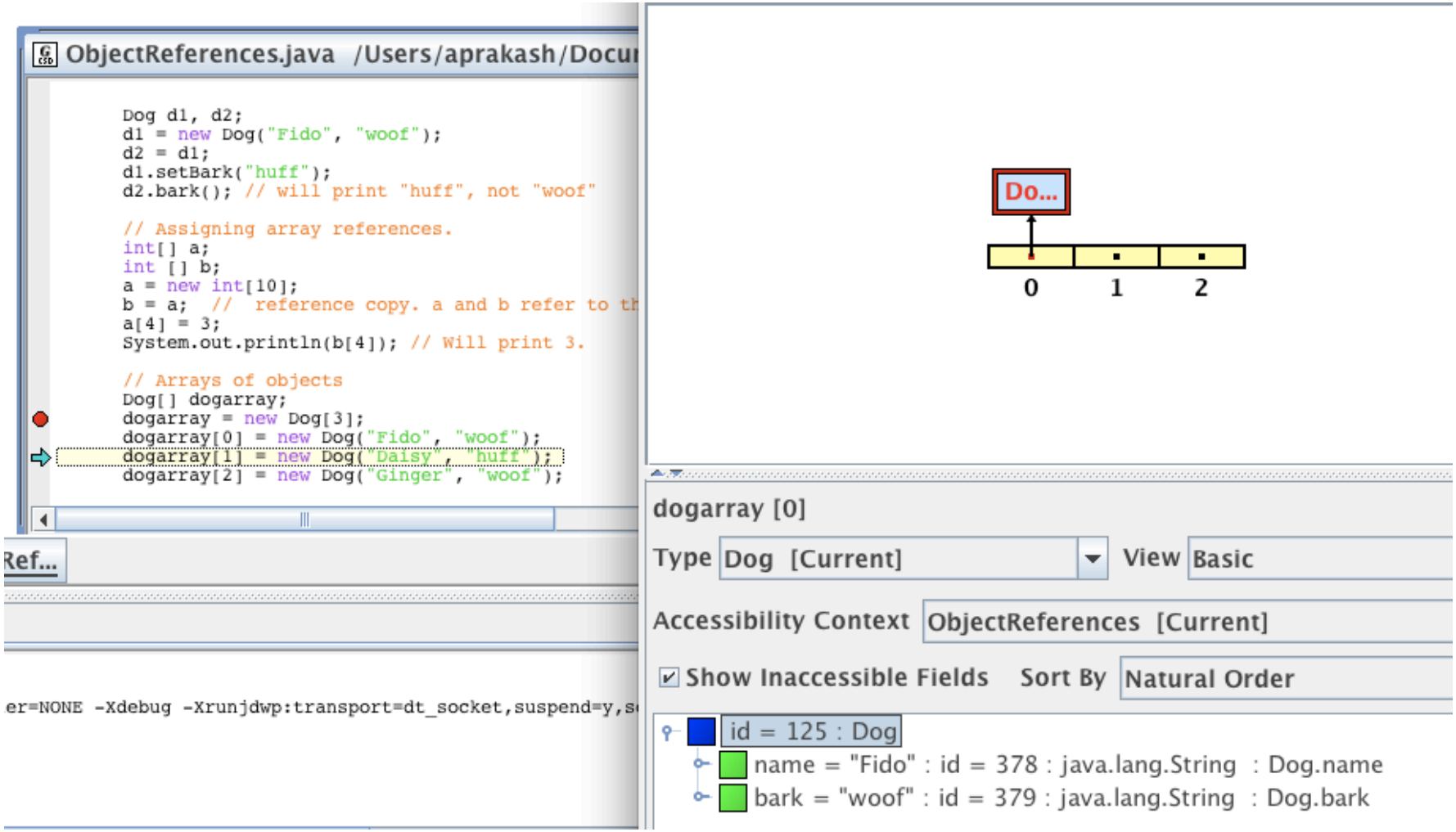
```
public static void main(String[] args) {
    int i, j; // Not references. Basic types. i = 0, j = 0
    i = 2;    // i = 2, j = 0
    j = i;    // i = 2, j = 2.
    i = 3;    // i = 3, j = 2
    System.out.println(j); // will print 2, not 3.

    Dog d1, d2; // References.
    d1 = new Dog("Fido", "woof");
    d2 = d1;
    d1.setBark("huff");
    d2.bark(); // will print "huff", not "woof"


}
```



*Java convention: Types starting with **small cap** (e.g., int) are **primitive**. Others should start with a capital letter (e.g., String, Dog) and are object types.*

Arrays of Objects

```
// Arrays of objects
Dog[] dogarray;// Create a reference to an array
dogarray = new Dog[3]; // Create 3 references to dogs

 // Create the dogs
dogarray[0] = new Dog("Fido", "woof");
dogarray[1] = new Dog("Daisy", "huff");
dogarray[2] = new Dog("Ginger", "woof");
```


ObjectReferences.java /Users/aprakash/Docu
Dog d1, d2;
d1 = new Dog("Fido", "woof");
d2 = d1;
d1.setBark("huff");
d2.bark(); // will print "huff", not "woof"
// Assigning array references.
int[] a;
int [] b;
a = new int[10];
b = a; // reference copy. a and b refer to th
a[4] = 3;
System.out.println(b[4]); // Will print 3.
// Arrays of objects
Dog[] dogarray;
dogarray = new Dog[3];
dogarray[0] = new Dog("Fido", "woof");
dogarray[1] = new Dog("Daisy", "huff");
dogarray[2] = new Dog("Ginger", "woof");
Ref...
er=NONE -Xdebug -Xrunjdwp:transport=dt_socket,suspend=y,s
Do...
0
1
2
dogarray [0]
Type Dog [Current]
View Basic
Accessibility Context ObjectReferences [Current]
Show Inaccessible Fields
Sort By Natural Order
id = 125 : Dog
name = "Fido" : id = 378 : java.lang.String : Dog.name
bark = "woof" : id = 379 : java.lang.String : Dog.bark