More on Classes

Savitch ch. 5-6, Lewis ch 5

Printing Objects

- By default, Java doesn’t know how to print objects:
  
  ```java
  Account acct = new Account(...);
  System.out.println("acct: " + acct);
  // result: acct: Account@9e8c34
  ```

- We give Account a print method that gives a more informative result:
  ```java
  acct.print();
  ```

- Java gives us a better mechanism using the toString method

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

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

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

Variable shadowing

- An instance method parameter can have the same name as one of the object’s instance variables:
  ```java
  public class Point {
      int x;
      int y;

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

- Instance variables `x` and `y` are shadowed by parameters with same names.
Avoiding shadowing w/ this

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

- Inside the setLocation method,
  - When this.x is seen, the instance variable x is used.
  - When x is seen, the parameter x is used.

Multiple constructors

- It is legal to have more than one constructor in a class.
  - The constructors must accept different parameters.

```java
public class Point {
    private int x;
    private int y;
    public Point() {
        x = 0;
        y = 0;
    }
    public Point(int x, int y) {
        this.x = y;
        this.y = y;
    }
    ...
}
```

Constructors and this

- One constructor can call another using this:

```java
public class Point {
    private int x;
    private int y;
    public Point() {
        this(0, 0); // calls the (x, y) constructor
    }
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    ...
}
```

Method overloading

- Can you write different methods that have the same name?
  - Yes!

```java
System.out.println("I can handle strings");
System.out.println(2 + 2);
System.out.println(3.14);
System.out.println(object);
Math.max(10, 15); // returns integer
Math.max(10.0, 15.0); // returns double
```

- Useful when you need to perform the same operation on different kinds of data.

```java
public int sum(int num1, int num2) {
    return num1 + num2;
}
public int sum(int num1, int num2, int num3) {
    return num1 + num2 + num3;
}
```

- A method’s name + number type and order of its parameters: method signature
- The compiler uses a method’s signature to bind a method invocation to the appropriate definition
The return value is not part of the signature

- You cannot overload on the basis of the return type (because it can be ignored)

Example:
```java
public int convert(int value) {
    return 2 * value;
}

public double convert(int value) {
    return 2.54 * value;
}
```

Example (cont)
```
public Pet()
public Pet(String name, int age, double weight)
public Pet(int age)
public Pet(double weight)
```

Suppose you have a horse that weights 750 pounds then you use:
```java
Pet myHorse = new Pet(750.0);
```
but what happens if you do:
```java
Pet myHorse = new Pet(750);  
```

Object Equality

- The == operator checks if the addresses of the two objects are equal
- If you want a different notion of equality define a .equals method.
- Do pet1.equals(pet2) instead of pet1==pet2

```
public boolean equals (Pet other) {
    return ((this.age == other.age) &&
           (Math.abs(this.weight - other.weight) < 1e-8) &&
           (this.name.equals(other.name)));
}
```

Example

- Consider the class Pet
```java
class Pet {
    private name;
    int age;
    double weight;
    ...
}
```

Object Equality

- Suppose we have two pet instances pet1 and pet2
- How does the statement pet1==pet2 behave?
Summary: Access Protection

Access protection has three main benefits:
- It allows you to enforce constraints on an object’s state.
- It provides a simpler client interface. Client programmers don’t need to know everything that’s in the class, only the public parts.
- It separates interface from implementation, allowing them to vary independently.

General guidelines

As a rule of thumb:
- Classes are public.
- Instance variables are private.
- Constructors are public.
- Getter and setter methods are public (unless…)
- Other methods must be decided on a case-by-case basis.