Recitation 2

Syed Raza (raza@mit.edu)

Fall 2010
Reminders

Lab 0.2
- Read the assignment and the reading materials before coming to lab in 32-123 (1-4pm).

Problem Set 1
- Due on Wednesday (Sept 22).

Project 1 Partners
- Project 1 will be released Wednesday (Sept 22), and is a group project (3 students per group).
- Please use the Class Wiki on the course website to sign up your team before lecture on Monday (Sept 20).
- If you do not register your team before lecture on Monday (Sept 20), we will randomly assign partners for you.
- You can only team up with someone once during the term, so pick your partners wisely!
Today

Recap: Lectures 2 & 3

- Review of Java concepts, including: Classes, Inheritance, Interfaces and Exceptions.
- Reflection on Weather and Page classes from Lecture.
Classes

Difference between a Class and an Instance

- A Class is a template (or a “blueprint”) that defines a new type in Java.
  - e.g. A `Person` Class may be defined to store the names & addresses of everyone in 6.005.

- An Instance is an object created using a Class definition.
  - e.g. A `Person` object that stores John’s name and address is an instance. Another `Person` object (or instance) may store Ben’s name and address.
Classes

Difference between a Class and an Instance

```java
public class Person {
    private String name;
    private String address;

    public Person(String name, String address) {
        this.name = name;
        this.address = address;
    }
}

Person ben = new Person("Ben", "EC");
Person john = new Person("John", "Baker");
```

Two instances of Person.

Class Definition for Person.
public class Person {
    private String name;
    private String address;

    public Person(String name, String address) {
        this.name = name;
        this.address = address;
    }

    public String sayHello() {
        return "I am " + name + ", and I live in " + address + ".";
    }
}
Classes

Fields, Constructors and Methods

```java
public class Person {
    private String name;
    private String address;

    public Person(String name, String address) {
        this.name = name;
        this.address = address;
    }

    public String sayHello() {
        return "I am " + name + ", and I live in " + address + ".";
    }
}
```

Constructor of the Person Class. Stores instance fields, just like the `__init__` method in Python.
public class Person {
    private String name;
    private String address;

    public Person(String name, String address) {
        this.name = name;
        this.address = address;
    }

    public String sayHello() {
        return "I am " + name + ", and I live in " + address + ".";
    }
}

An instance method for the Person class. Usage: ben.sayHello();
To disambiguate arguments and fields, use the “this” keyword.
Classes

Inheritance

- We defined a *Person* Class to store the names & addresses of everyone in 6.005.
- People in 6.005 can be further subdivided into Students, TAs and the Professor.
- To model these three groups, we could write three different classes for them from scratch.
- Could we do something better?
Classes

Inheritance

- We defined a Person Class to store the names & addresses of everyone in 6.005.
- People in 6.005 can be further subdivided into Students, TAs and the Professor.
- To model these three groups, we could write three different classes for them from scratch.
- Could we do something better? Yes! We can reuse code in the Person class (via inheritance).
Classes

Inheritance

```java
public class Student extends Person {
    public Student(String name, String address) {
        super(name, address);
    }
}
```

This is *all* the code needed to create the `Student` class. We get the name & address fields for free, along with the `sayHello()` method from the `Person` class.
This says that the Student class is a subclass of the Person class.

That is, the Student class inherits from the Person class.
Classes

Inheritance

```java
public class Student extends Person {
    public Student(String name, String address) {
        super(name, address);
    }
}
```

This calls the constructor of the Person class.

While “this” refers to the object itself, “super” refers to the enclosing instance of the parent class.
Classes

Inheritance

```java
public class Student extends Person {
    public Student(String name, String address) {
        super(name, address);
    }
}

Student ben = new Student("Ben", "EC");
System.out.println(ben.sayHello());
```

I am Ben, and I live in EC.
Classes

Inheritance

```java
public class Student extends Person {
    public Student(String name, String address) {
        super(name, address);
    }
}
```

Student ben = new Student("Ben", "EC");
System.out.println(ben.sayHello());

What if we want the student to say something more, such as:

I am Ben, and I live in EC.
I am taking 6.005 this term.

How would you implement this behavior, without changing the Person class?
Classes

Overriding Methods

```java
public class Student extends Person {
    public Student(String name, String address) {
        super(name, address);
    }

    @Override
    public String sayHello() {
        String hello = super.sayHello();
        return hello + "\nI am taking 6.005 this term."
    }
}

Student ben = new Student("Ben", "EC");
System.out.println(ben.sayHello());
```

I am Ben, and I live in EC.
I am taking 6.005 this term.
Classes

Overriding Methods

```java
public class Student extends Person {
    public Student(String name, String address) {
        super(name, address);
    }

    @Override
    public String sayHello() {
        String hello = super.sayHello();
        return hello + "\nI am taking 6.005 this term."
    }
}
```

This is a Java annotation, that tells the compiler that you wish to override a method from a parent (or grandparent etc) class.

It is strongly recommended that you put the @Override annotation whenever you override some parent method. This can prevent mistakes from typos or wrong signatures.
We get the result of the sayHello() method from the Person class, and then append another string.
The method signature includes only the method name, and the types and number of arguments of a method.

Overriding Methods

```java
public class Student extends Person {
    public Student(String name, String address) {
        super(name, address);
    }

    @Override
    public String sayHello() {
        String hello = super.sayHello();
        return hello + "\nI am taking 6.005 this term.";
    }
}
```

Overriding is different from overloading. In overloading, two methods have the same name but different signatures.

Overloading is useful for reusing code:

```java
void printFile(String fileName, int n, bool color_print);
void printFile(String fileName);
```

The second method may simply call the first one with n=1, color_print=false (default values).
Classes

Overriding Methods

```java
public class Student extends Person {
    public Student(String name, String address) {
        super(name, address);
    }

    @Override
    public String sayHello() {
        String hello = super.sayHello();
        return hello + "\nI am taking 6.005 this term.";
    }
}
```

Overriding is different from overloading. In overriding, two methods have the same return type, signature and access
modifiers but are in different classes which have a parent-child (or
grandparent-child etc) relationship.
Why Override Methods?

```java
public class Student extends Person {
    public Student(String name, String address) {
        super(name, address);
    }

    @Override
    public String sayHello() {
        String hello = super.sayHello();
        return hello + "\nI am taking 6.005 this term."
    }
}
```

We could have just defined a new method instead of `sayHello()` e.g. `sayHelloStudent()`. Why is that not a good idea?
Why Override Methods?

With Overriding (good design in this example).

```java
public class Dinner {
    private ArrayList<Person> guests;

    public Dinner(ArrayList<Person> guests) {
        this.guests = guests;
    }

    public void printIntroductions() {
        for (Person person : guests) {
            System.out.println(person.sayHello());
        }
    }
}
```

The printIntroductions() method does not need to care about which subtype of Person each guest is (when we use overriding).

Java checks at compile-time that we call methods from Person (declared type), but at run-time the method will be called from the instance’s class.
Classes

Why Override Methods?

```java
public class Dinner {
    private ArrayList<Person> guests;

    public Dinner(ArrayList<Person> guests) {
        this.guests = guests;
    }

    public void printIntroductions() {
        for (Person person : guests) {
            if (person.getClass().equals(Student.class)) {
                Student s = (Student) person;
                System.out.println(s.sayHelloStudent());
            } else {
                System.out.println(person.sayHello());
            }
        }
    }
}
```

Without Overriding (bad design in this example).

Notice code is more complicated… Why don’t we use `instanceof`? Why do we need to make a `cast`?
Classes

Casts, Object Creation

Person person = new Student("Chuck Norris", " Everywhere");
Student student = (Student) person;

Does this modify person?
Is student a new object?

Person person = new Student("Chuck Norris", " Everywhere");
Person other = person;
other.setAddress("Stata");
System.out.println(person.getAddress());

What is the output?
Classes

Casts, Object Creation

```java
Person person = new Student("Chuck Norris", "Everywhere");
Student student = (Student) person;
```

These two lines of Code do not create a new object, or modify `person`. `student` is just the same object, but it is being treated as a Student type.

```java
Person person = new Student("Chuck Norris", "Everywhere");
Person other = person;
other.setAddress("Stata");
System.out.println(person.getAddress());
```

New objects are only created with `new`. Only primitives can be copied over without using `new`.

That is: `int x = 3; int y = x; y++; // x = 3;`
Classes

Assigning Student IDs

```java
public class Student extends Person {
    public Student(String name, String address) {
        super(name, address);
    }

    @Override
    public String sayHello() {
        String hello = super.sayHello();
        return hello + "\nI am taking 6.005 this term.";
    }
}
```

Say each student in 6.005 has a unique ID. How would you assign student IDs to students?
public class Student extends Person {

    private static int nextStudentID = 0;
    private int myStudentID;

    public Student(String name, String address) {
        super(name, address);
        this.myStudentID = Student.nextStudentID;
        Student.nextStudentID++;
    }

    @Override
    public String sayHello() {
        String hello = super.sayHello();
        hello = hello + "\nI am taking 6.005 this term";
        return hello+"\nMy student number is "+myStudentID+.";";
    }
}
public class Student extends Person {

    private static int nextStudentID = 0;
    private int myStudentID;

    public Student(String name, String address) {
        super(name, address);
        this.myStudentID = Student.nextStudentID;
        Student.nextStudentID++;
    }

    @Override
    public String sayHello() {
        String hello = super.sayHello();
        hello = hello + "\nI am taking 6.005 this term";
        return hello + "\nMy student number is " + myStudentID + ".";
    }
}

Solution

A static field is shared by all instances, unlike the instance field.
public class Student extends Person {

    private static int nextStudentID = 0;
    private int myStudentID;

    public Student(String name, String address) {
        super(name, address);
        this.myStudentID = Student.nextStudentID;
        Student.nextStudentID++;
    }

    @Override
    public String sayHello() {
        String hello = super.sayHello();
        hello = hello + "\nI am taking 6.005 this term";
        hello = hello + "\nMy student number is " + myStudentID + ".";
        return hello;
    }
}

A static field (or method) must be invoked using the class name e.g. Student.nextStudentID
public class Student extends Person {

    private static int nextStudentID = 0;
    private int myStudentID;

    public Student(String name, String address) {
        super(name, address);
        this.myStudentID = Student.nextStudentID;
        Student.nextStudentID++;
    }

    @Override
    public String sayHello() {
        String hello = super.sayHello();
        hello = hello + "\nI am taking 6.005 this term"
        hello = hello + "\nThe student number is " + myStudentID + ".";
        return hello + "\nMy student number is " + myStudentID + ".";
    }
}

This updates the static field for all instances of Student.
The “Math” class is a container class that contains many static methods.

These methods are static because they do not access/change any fields in “Math”. In fact, these *stateless* methods are just convenience macros that can be called from anywhere, without instantiating anything.
The "Math" class also contains static constants that can be accessed from anywhere.

Because they are "final", they cannot be changed.
Classes in Practice

public class Integer {

    /* Static Field: Constant */
    public static final int MAX_VALUE = ... // 2^31-1

    /* Convenience Static Method */
    public static Integer decode (String string) {
        ...
    }

    /* Instance Method */
    public int intValue () {
        ...
    }

    /* Constructor */
    public Integer(int value) {
        ...
    }
}

Classes in practice have static fields, static methods, instance fields and methods. Try to figure out why we use static/instance methods/fields in this class.
Classes

Assigning Student **Immutable** IDs

```java
public class Student extends Person {

    private static int nextStudentID = 0;
    private int myStudentID;

    public Student(String name, String address) {
        super(name, address);
        this.myStudentID = Student.nextStudentID;
        Student.nextStudentID++;
    }

    @Override
    public String sayHello() {
        String hello = super.sayHello();
        hello = hello + "\nI am taking 6.005 this term";
        return hello+"\nMy student number is "+ myStudentID +".";
    }
}
```

How to make myStudentID fixed?
Classes

Assigning Student **Immutable** IDs

```java
public class Student extends Person {

  private static int nextStudentID = 0;
  private final int myStudentID;

  public Student(String name, String address) {
    super(name, address);
    this.myStudentID = Student.nextStudentID;
    Student.nextStudentID++;
  }

  @Override
  public String sayHello() {
    String hello = super.sayHello();
    hello = hello + "\nI am taking 6.005 this term";
    return hello+"\nMy student number is "+ myStudentID +".";
  }
}
```

A final instance field can only be assigned **once**, either where it is declared or inside the constructor. It cannot be written to anywhere else. A class is immutable if none of its fields can be changed.
Classes

Person Immutable Class

```java
public class Person {
    private final String name;
    private final String address;

    public Person(String name, String address) {
        this.name = name;
        this.address = address;
    }

    public String sayHello() {
        return "I am " + name + ", and I live in " + address + ".";
    }
}
```
### Classes

#### Person Immutable Class

```java
public class Person {
    private final String name;
    private final String address;

    public Person(String name, String address) {
        this.name = name;
        this.address = address;
    }

    public String sayHello() {
        return "I am " + name + ", and I live in " + address + ".";
    }
}
```

What if we want to allow people to drive cars? Let’s add a `driveToSchool(...)` method.
DriveToSchool Method

Person Immutable Class

```java
public class Person {
    private final String name;
    private final String address;

    public Person(String name, String address) {
        this.name = name;
        this.address = address;
    }

    public void driveToSchool(Car car) {
        car.enter();
        car.startIgnition();
        car.driveForward(5);
    /* And So On */
    }

    /* Other Methods */
}
```
Interfaces

public interface Car {
    void enter();
    void startIgnition();
    void driveForward(int miles);
    /* Other Methods */
}

We could have used a Car class, and subclassed in with specific types of cars. Here, we use a different approach.
Interfaces

```java
public interface Car {
    void enter();
    void startIgnition();
    void driveForward(int miles);
    /* Other Methods */
}
```

Note:
1. An interface cannot be instantiated.
2. An interface can only contain method declarations (no body) or constants.
3. All of an interface’s methods or constants are public (it is implicit).

We could have used a Car class, and subclassed in with specific types of cars. Here, we use a different approach.
public interface Car {
    void enter();
    void startIgnition();
    void driveForward(int miles);
    /* Other Methods */
}

public class BMW_3232 implements Car {
    @Override
    public void startIgnition() {
        /* Do Stuff */
    }
    @Override
    public void driveForward(int miles) {
        /* Do Stuff */
    }
    /* Other Methods */
}
Interfaces

Benefits of Interfaces

- Interfaces ‘decouple’ clients and implementation.
  - People learn how to drive without worrying about how every car in the world will work;
  - Manufacturers alter implementation features for cars but they have to provide a simple & consistent exterior interface for drivers.
  - In our code, a Person can drive a BMW_3232 or any other car. Implementation details of a Car are abstracted away.
- Interfaces support multiple inheritance, which is particularly useful when the Interface is a “functionality” rather than a type.
- See Effective Java (2nd edition), Item 18.
Interfaces

Using Interfaces Correctly

- Interfaces can also have their own hierarchy (e.g. interface x extends y).
- Interfaces specify “contracts” or “functionalities” that may not fit “is a special type of” relationships.
- For instance, a Student is a special type of Person (we therefore use Inheritance and subclassing).
- A Bed is SomethingToSitOn and SomethingToSleepOn. These are two different functionalities which are possibly disjoint (we therefore define them as interfaces).

For a Bed, you cannot use subclasses easily because each Java class can have only one parent class. Thus, if Bed must inherit from both SomethingToSitOn and SomethingToSleepOn, they also need to have a parent-child relationship… but that doesn’t make sense.
public class Person {
    /* Ignore other stuff*/
    private final boolean haveLicense;

    public Person(String name, String address) {
        Person(name, address, false);
    }

    public Person(String name, String address, boolean haveLicense) {
        this.name=name; this.address=address; this.haveLicense=haveLicense;
    }

    public void driveToSchool(Car car) {
        if(!haveLicense)
            throw new NoLicenseException();
        /* Do Stuff*/
    }
}
Exceptions

Checked & Unchecked Exceptions

- Checked exceptions **force** the programmer to add a `try... catch... finally` block or update the method declaration with a `throws ...` clause.

- As a rule of thumb, *checked exceptions* are semantic errors that you *want* the programmer to deal with (e.g. asking a person without a license to drive is a recoverable error and should be a checked exception).

- Unchecked exceptions do not need to be caught, and should be irrecoverable errors that usually result from programmer error (e.g. a NullPointerException).
Java Questions

Difference between == and equals()

- == checks whether two variables point to the same object.
- equals() is an Object method that checks for semantic equality.

- String s = “yo”;
- String t = “yo”;

  s==t returns false  
  s.equals(t) returns true.

- You will have to implement equals() later in the course. If equals() is not explicitly defined, it just calls the == by default. This is true for all classes you’ll write for now. Java’s classes (e.g. String) implement equals() methods that behave correctly.
Why use a different declared type from the actual type?

```java
List<Integer> list = new ArrayList<Integer>();
// code that calls methods on list
```

The code calls only methods on the List interface…
Using List interface type ensures:

First, we can only call methods of List interface (this is nice – we may not want to care about unique methods of ArrayList).

Second, we can switch from ArrayList<Integer> to any other implementation of List (e.g. LinkedList<Integer>) by changing only one line of code:

```java
List<Integer> list = new LinkedList<Integer>();
```

// code that calls methods on list object.

We may want to change the implementation for any reason e.g. performance.
Reflection on Lecture 3 Code

Problem with Page and Weather classes

- The cache was messy, because it could contain a Page object or a Weather object.

Solution:

- The Page class should have a private cache containing content of pages for URLs.
- The Weather class should have its own private cache of temperature/condition corresponding to URLs.
- Page class behaves as before, but only adds Page objects to its cache.
- Weather class behaves similarly and only adds Weather objects to its cache.

- The cache interface had no support for eviction.
- A cache should be finite so the constructor should have taken the size of the cache.