User Defined Classes
Part 2
Class vs. Instance methods

- Compare the Math and String class methods that we have used:
  - Math.pow(2,3);
  - str.charAt(4);

- One is called using the class name (Math) and the other using an object identifier (str)

- How are they different?

- pow() is a \textit{class method} whereas charAt is an \textit{instance method}
Class Methods

- A **class method** is one which is defined with the **static** keyword.

- A class (**static**) method
  - can be called directly from any other static method of the class
  - can be called using the class name from an instance method or from another class (if it is public)
  - cannot directly call an instance method of the same class -- must be called on an object of that class
  - cannot access instance data members -- only through an object
Instance Methods

- An *instance method* is one which is defined without the static keyword.
- An object method
  - can be called directly from any other object method of the class
  - can be called on an object of that class from either a static method or a method of another class.
  - can call a class method using the name of the class.
Class and Instance Methods

```java
public class Test{
  int id;
  public void methodA() {
    id = 5;
    methodB();
  }
  public void methodB() {
    id = 7;
    methodT();
    Test.methodT();
  }
  public static void methodS() {
    Test t = new Test();
    methodT();
    t.methodA();
    methodA();
    id = 3;
  }
  public static void methodT() {
    ...
  }
}
```

- **Instance method can't call a class method without the class name.**
- **Class method can't call instance method without an object.**
- **Class method can't access an instance data member without an object.**
Calling Methods of Same Class

class Student {
  private String id;
  private String name;

  public void setName(String newName){
    . . .
  }
  private void setId(String newID){
    setName("ID changed");
    . . .
  }
  public String getId(){
    . . .
  }
}

No object specified.

setName() will use the same object as the one on which setId() was called.
Class Data Members

- As with methods, we can have both instance data members and class data members.
- Both are defined outside any method.
- Class data members are those with the static keyword.
- There is only one copy of each class data member -- shared by all objects of the class.
- Each object has its own copy of all instance data members.
Consider the Student class discussed earlier. Modify it so that student IDs are automatically generated as follows:
- Each id is of the form: PUIDnnnnn where nnnnn is a digit.
- Each successive student object created should get a unique id in sequence, starting with PUID1, then PUID2, PUID3, ...
Solution

- How will the constructor know what was the last ID generated for a student?
- We have to save this somewhere.
  - Where?
  - This can't be stored as a data member. Why?
- This information can be saved in a class data member.
  - There is only one copy of it for the entire class.
class Student {
  static int nextID = 1;

  private String name, id;
  private double gpa;

  public Student() {
    id = "PUID" + Student.nextID;
    Student.nextID++;
    name = "Unknown";
    gpa = 0.0;
  }

  ... 
}

Student student1, student2;
student1 = new Student();
student2 = new Student();

NOTE: Class, not object.
Class Data Members

Class data members can be
- directly accessed by class methods
- accessed by instance methods by using the class name
- can be constants (defined with static final)
- instance constants should usually be class constants (less space is used).
Main method

- Recall that main is a class method.
- Thus, it can't call any instance methods directly.
  - we must have an object to do this
- Any methods called from main should also be class methods.
- None of these class methods can access instance data members.
public class Roster{
  public static void main(String[] args) {
    Student[] studentList;
    Student student1;

    studentList = initializeRoster();
    student1 = findStudent("2334", studentList);
    if(student1 == null)
      System.out.println("Student with id 2334 not found in class");
    else
      student1.printNeatly();
  }

  public static Student findStudent(String id, Student[] sList){
    Student s;
    int i;
    for(i=0; i < sList.length;i++)
      if(id.equals(sList[i].getId()))
        return sList[i];
    return null;
  }
}
Alternative: Class Data Member

```java
public class RosterV2{
    static Student[] studentList;

    public static void main(String[] args) {
        Student student1;

        studentList = initializeRoster();
        student1 = findStudent("2334");
        if(student1 == null)
            System.out.println("Student with id 2334 not found in class");
        else
            student1.printNeatly();
    }

    public static Student findStudent(String id){
        int i;
        for(i=0; i < studentList.length;i++)
            if(id.equals(studentList[i].getId()))
                return studentList[i];
        return null;
    }
}
```

- studentList is a class data member.
- No need to pass studentList as a parameter.
- studentList accessible directly by a class method.
public class RosterV3{
    Student[] studentList;
    public static void main(String[] args) {
        Student student1;
        RosterV3 roster = new RosterV3();
        student1 = roster.findStudent("2334");
        if(student1 == null)
            System.out.println("Student with id 2334 not found in class");
        else
            student1.printNeatly();
    }
    public RosterV3(){
        int classSize, i;
        classSize = Integer.parseInt(JOptionPane.showInputDialog(null,
            "Enter number of students in class"));
        studentList = new Student[classSize];
        for(i=0; i<classSize;i++)
            studentList[i] = new Student();
    }
    public Student findStudent(String id){
        ...
        for(i=0; i < studentList.length;i++)
            if(id.equals(studentList[i].getId()))
                return studentList[i];
        return null;
    }
}

studentList is an instance data member.
Create a RosterV3 object.
Call a class method on object.
Constructor
Directly access instance data member.
Static Initializer

- Earlier, we initialized static variables upon declaration. This initialization takes place when the class is loaded.
  - Imported or used for the first time in a program.
- What if we want to do more?
  - E.g. set the initial value based upon user input?
- We can define a static initializer segment that gets executed only once when a class is loaded.
As with static methods, we cannot reference any non-static method or data member from the static initializer block.
Reserved Word **this**

- Each object has a special data member called **this** that references the object itself.
- This keyword is useful for three purposes:
  - Accessing data members of the object
  - Accessing instance methods
  - Calling one constructor from another
Using **this** to Refer to Data Members

```java
class Student {
    String name;

    public void setName(String newName) {
        this.name = newName;
    }
}
```

Usually implied and omitted.
Referencing Data Members

```java
class Student {
    String name;
    public void setName(String name) {
        name = name;
    }
}
```

Doesn't work as expected since the formal parameter masks the data member.

```java
class Student {
    String name;
    public void setName(String name) {
        this.name = name;
    }
}
```

Achieves the desired effect.
null is a special value. Its type is that of a reference to an object (of any class).

We can set an object identifier to this value to show that it does not point to any object.

Student student1=null;

A method that returns objects (of any class) can return a null value.

Note that you will get a run-time error if you access a data member of call a method of a null reference -- null pointer exception.
Overloading

- Java allows multiple methods to share the same name
  - as long as they have different method signatures (i.e., different types of arguments, in order)
  - this is called overloading

- Which version is executed when called?
  - the one that matches the signature of the call.

- **Method signature:**
  - the sequence of types taken as input
  - does not include return type
public class Test {
    void methodA() {
        ... 
    }
    void methodA(int a){
        ... 
    }
    void methodA(int a, int b){
        ... 
    }
    void methodA(int a, float b){
        ... 
    }
    void methodA(float b, String s){
        ... 
    }
}
Overloaded Methods

```java
public class Test {
  void methodA() {
    ...
  }

  void methodA(int a){
    ...
  }

  void methodA(int a, int b){
    ...
  }

  void methodA(int a, Student b){
    ...
  }

  void methodA(Student b, int a){
    ...
  }
}
```

```java
public class Test {
  void methodA() {
    ...
  }

  void methodA(int a, int b){
    ...
  }

  void methodA(int b, int a){
    ...
  }

  int methodA(int c, int d){
    ...
  }

  int methodA(){
    ...
  }
}
```
Overloaded Constructor

- The same rules apply for overloaded constructors
  - multiple constructors can be defined as long as each has a unique signature.

```java
public Student() {
    ...
}
public Student(String name) {
    ...
}
public Student(String name, double gpa) {
    ...
}
```
Calling one Constructor from Another

- One can call a constructor of the same class from another using the keyword this.

```java
static int nextID;

public Student() {
    this("Unknown Name");
}

public Student(String name) {
    this(name, 0.0);
}

public Student(String name, double gpa) {
    this.id = "PUID" + Student.nextID++;
    this.name = name;
    this.gpa = gpa;
}
```
Overloading and Type Casting

- When calling a method, if not matching signature is found, the compiler will attempt a match with automatic type casting.

```java
byte b;
methodA(b);
```

- Can match any of these methods:

```java
public methodA (short val) { ... }
public methodA (int val) { ... }
public methodA (long val) { ... }
public methodA (float val) { ... }
public methodA (double val) { ... }
public methodA (Short val) { ... }
public methodA (Integer val) { ... }
public methodA (Long val) { ... }
public methodA (Float val) { ... }
public methodA (Double val) { ... }
```
Copy constructor

- A copy constructor can be very handy.
- It takes an object as input and creates another object (of the same class) and copies the values.
- Useful also for preventing surreptitious access to private objects.
- If a method returns a pointer to a private object, then the client can modify the private object!
- Avoid this by returning a copy object
class Corruptor {
   Jedi luke;
   Person p;
   public static void main(String[] args){
      luke = new Jedi(new Person("Unknown"));
      p = luke.getFather();
      p.setName("Darth Vader");
      p = luke.getFather();
      System.out.println(p.getName());
   }
}

class Jedi {
   private Person father;

   public void Jedi(Person f){
      father = f;
   }

   Person getFather(){
      return father;
   }
}

class Jedi {
   private Person father;

   public void Jedi(Person f){
      father = f;
   }

   Person getFather(){
    Person x;
    x = new Person(father);
      return x;
   }
}
Multiple classes per file

- So far, we only defined a single class per file.
- It is possible to define multiple classes in a single file.
- However, only one file can be public. This class must have the same name as the file (.java).
- Only public classes can be imported.
Files and Classes

class CS180Staff {
    . . .
}
class CS180Student {
    . . .
}

Two class files are created. Both are available to other classes in the same directory.

File: Test.java
javac Test.java

File: CS180Staff.class
File: CS180Student.class

0101110010101....
1110010111010....

© Sunil Prabhakar, Purdue University
Organizing Classes into a Package

For a class, A, to use another class, B,
- their byte code files must be located in the same directory, OR
- Class B has to be organized as a package and A must import class B

A package is a collection of java classes that can be imported.
- E.g., javax.swing.* contains numerous classes that we import.
Creating a package

To create the CS180 package with CS180Staff as an included class:

- Add package CS180; at the top of CS180Staff.java
- CS180Staff must be a **public** class
- Compile CS180Staff and place the .class file into the CS180 directory
- Add the path to CS180 to the CLASSPATH variable
Creating a package

```java
package CS180;
public class CS180Staff {
    public static void main (String[] arg){
        ...
    }
}
```

File: CS180Staff.java

```java
import CS180.*;
class MyClass {
    public static void main (String[] arg) {
        CS180Staff t;
        ...
    }
}
```

File: MyClass.java

javac CS180Staff.java

CS180 package directory must be in CLASSPATH variable when running MyClass.
Files and Classes

class CS180Staff {
    public static void main (String[] arg) {
        ...
    }
}
class CS180Student {
    public static void main (String[] arg) {
        ...
    }
}

Two class files are created. Both can be executed with java ...

javadoc Test.java

java CS180Staff

java CS180Student

© Sunil Prabhakar, Purdue University
Multiple Classes in a File

```
package CS180;
public class CS180Staff {
    public static void main (String[] arg) {
        ...
    }
}
class CS180Student {
    ...
}
```

The `main` method of `CS180Staff` must be called when running the program.

```
import CS180.*;
class MyClass {
    public static void main (String[] arg) {
        CS180Staff t;
        ...
    }
}
```

Only `CS180Staff` can be imported and used outside the package directory.

CS180Student can only be used within the CS180 directory.

CS180 package directory must be in CLASSPATH variable when running MyClass.