ICS 111
Overriding, Polymorphism

- overloading methods
- overriding methods
- object polymorphism
- abstract methods, abstract classes
- final methods, final classes
- example from textbook: bank accounts

Method Overloading

- We have already seen cases where methods with the same name are called on different parameters:
  - System.out.println() and System.out.println(String s)
  - constructors with different parameter types
- In each case, these are different methods that just happen to have the same name
- This is called **overloading**
  - for example, System.out.println is an overloaded method
  - constructors can be overloaded
- Overloading is fine under two conditions:
  - it doesn't cause confusion to callers of the methods
    - i.e. the methods should do "the same thing", even though on different parameters
  - it is done on purpose, rather than accidentally
    - accidental overloading can happen when overriding methods – discussed on the next slide

Method Overriding

- when a subclass extends a superclass, it inherits its public+protected methods
- sometimes in the subclass we want to modify what one of the superclass methods does:
  - suppose I create a MyAL class which extends ArrayList
  - I want to modify the single-parameter add method to add at the beginning of the array list, rather than at the end
- I can do this by re-declaring the same method in the subclass, with the same parameters, and re-implementing it
- the method in the subclass may use `super` to call the method in the superclass
  - example for MyAL:

```
public void add(Value v) {
  super.add(0, v);
}
```

- accidental overloading happens when we intend to override, but use a different set of parameter types

Overriding and overloading: remembering the difference

- Overriding is when a subclass re-implements the method of a superclass
  - the new method overrides (takes over from) the corresponding method in the superclass
- Overloading is when the same name refers to different methods
  - the name is overloaded because, instead of referring to a single method, it refers to several different methods
    - the same name has to “carry” multiple methods – it is overloaded

What method gets called?

- Suppose we have a method that takes as parameter an Object and prints it:

```
public static void printObject(Object x) {
  String s = x.toString();
  System.out.printf("%s\n", s);
}
```

- if I have a variable of type ArrayList<String> al = …
- and given that ArrayList overrides the toString() method of its superclass
- calling printObject(al) calls which toString method?
  - Object.toString(), or
  - ArrayList.toString()
- calling the method in ArrayList is more useful
- and this is what Java does:
  - method calls are determined **dynamically** by the actual underlying object, not by the type declaration

Polymorphism

- in Greek, “poly” refers to many, and “morph-” refers to form, shape, or type
- in computer science, polymorphism refers to a single variable possibly having values of different types
- we have seen polymorphism in the example on the previous slide: the parameter is Object, the actual value is of type ArrayList<String>
- within the method that has a parameter of type Object, we can only use methods of the Object class
  - but as we have seen, the method that is actually called is the most specific possible method, determined dynamically
- so:

  1. only the methods of the declared type can be used
  2. of these methods, the one from the actual object is the one that is used
- this is important for writing correct programs!
- fortunately, it is also rather intuitive

things to be careful about

- remember to use `super` when calling methods from the superclass
  - `this` and `super` help resolve name clashes
- use accessor and mutator methods to access the private variables in the superclass
- constructor calls to `this()` or `super()` must be the first statement in the body of a constructor
- `this` refers to the actual object, not the declared object type
  - `this.toString()` calls the toString() method of the subclass, not of the superclass, nor Object.toString()

abstract classes and methods

- sometimes a class is designed to be subclassed
- the designer of the superclass may want to require the subclass to provide a specific method
- this method is called `abstract` in the superclass
  - and does not have an implementation in the superclass

```
public abstract String concatenate(String s);
```

- any class with one or more abstract methods is an abstract class
  - and must be declared with the keyword abstract

```
public abstract class StringOperations { ….
```

- abstract classes have no constructors
- we cannot create an object of an abstract class
- but we can have variable and parameter types be abstract classes

```
public class Example extends StringOperations { … }
StringOperations s1 = new Example();
```

- summary: an abstract class forces implementers of subclasses to implement all the methods that are abstract in the superclass
- implementers of subclasses still inherit any non-abstract methods from the superclass

final classes and methods

- we have seen that variables declared with `final` are constants
- the `final` keyword is used in a similar sense in class declarations to mean that a class cannot be subclassed

```
public final class String { … }
```

- final can also be used in a method declaration, to mean that the method cannot be overridden:

```
public final void doNotOverrideThis(int x) { …
```

- abstract classes are common in the Java standard library, final classes are not as common

Worked-out example: Bank Account class

- from textbook Section 9.4, How-To 9.1
- design and implement a class hierarchy to represent different types of bank account
- at the root of the hierarchy is a BankAccount object that can represent any account
  - it keeps the balance in an instance variable
  - it has a getBalance() accessor method
  - it has mutator methods for deposits and withdrawals
  - it has a method to do end-of-month processing
    - which doesn’t do anything
    - but may be overridden by subclasses

Worked-out example: subclasses

- each subclass of BankAccount, e.g SavingsAccount and CheckingAccount, provides the deposit, withdrawal, getBalance, and monthEnd methods
  - only overriding whatever methods it needs to override
  - we could easily have an account type that does not override any methods
- the SavingsAccount overrides the monthEnd method to deposit interest into the bank account once a month:

  ```
  double interest;
  public void monthEnd() {
    super.deposit(interest * super.getBalance());
  }
  ```

  the book handles a few more cases, specifically computing the interest on the minimum rather than the final balance
- in this example, both uses of `super` are optional, since SavingsAccount does not override getBalance and deposit

Review: Objects and Classes

- classes define the type of object values
- the implementation of a class includes all the class variables (including the instance variables) and the class methods and their code
- classes are grouped hierarchically so that every class (except Object) extends another class
  - a value of a subclass type can always be used where a value of the superclass type is needed
  - but not the other way around, e.g. you cannot use an Object where a String is needed
- extending a class gives us all of that class’s methods
  - with the option of overriding some of those methods
  - and of course the option of declaring our own methods and variables

Summary

- we are starting to see that programming with objects is more than just getting our code to work: it is also about representing our data in clear and useful ways
- once we have created data representations useful for the task at hand, the actual code can be relatively simple
- coding includes the coding of methods inside a class, and the coding of methods that create and use objects