Polymorphism in Java
Homework

• HW6 due tomorrow
• For string comparison, use `equals` instead of `==`
  – For example
    ```java
    String str1 = “Hello”;
    String str2 = “World”;
    str1.equals(str2) is false
    ```
• Will post HW7 tonight
Signatures

• In any programming language, a **signature** is what distinguishes one function or method from another
• In C, every function has to have a different name
• In Java, two methods have to differ in their *names* or in the *number* or *types* of their parameters
  – *foo*(int *i*) and *foo*(int *i*, int *j*) are different
  – *foo*(int *i*) and *foo*(int *k*) are the same
  – *foo*(int *i*, double *d*) and *foo*(double *d*, int *i*) are different
• In C++, the signature also includes the *return type*
  – But not in Java!
Polymorphism

• Polymorphism means many (poly) shapes (morph)
• In Java, polymorphism refers to the fact that you can have multiple methods with the same name in the same class
• There are two kinds of polymorphism:
  1. Overloading
     – Two or more methods with same names but with different signatures
     – Eclipse example: Overloading.java
  2. Overriding
     – Replacing an inherited method with another having the same signature
Why overload a method?

• So you can use the same names for methods that do essentially the same thing
  – Example: `println(int), println(double), println(boolean), println(String)`, etc.
• So you can supply defaults for the parameters:
  ```java
  int increment(int amount) {
      return count + amount;
  }
  int increment() {
      return increment(1);
  }
  ```
• So you can supply additional information:
  ```java
  void printResults() {
      System.out.println("total = " + total + ", average = " + average);
  }
  void printResult(String message) {
      System.out.println(message + ": ");
      printResults();
  }
  ```
• You can also do “the same thing” with different data
  – `print` and `println` are heavily overloaded
  – Python allows natural overloading (think back to the midterm)
Legal assignments/method calls

• **Widening** is legal
  – double d = 5; // legal

• **Narrowing** is illegal (unless you **cast**)
  – int i = 3.5; // illegal
  – int i = (int) 3.5; // legal

• The following call to **myPrint** is legal due to widening

```java
class Test {
    public static void main(String args[]) {
        myPrint(5);
    }

    static void myPrint(double d) {
        System.out.println(d);
    }
}
```
Illegal method calls

• The following call to `myPrint` is illegal due to narrowing

```java
class Test {
    public static void main(String args[]) {
        myPrint(5.0);
    }

    static void myPrint(int i) {
        System.out.println(i);
    }
}
```

• In general, Java uses the most specific method, i.e., requiring the least number of casts internally
  – Eclipse example: Overloading.java
Multiple constructors

• You can “overload” constructors as well as methods:
  ```java
  Counter() {
    count = 0;
  }
  Counter(int start) {
    count = start;
  }
  ```

• Same rules for calling constructors apply as for methods
  – Calling constructors within constructors more complicated

• A common reason for overloading constructors is (as with methods) to provide default values for some parameters
An aside: Namespaces

- In Python, if you named a variable **list**, you could no longer use the **list()** method
- This sort of problem is very rare in Java
- Java figures out what kind of thing a name refers to, and puts it in one of seven different namespaces:
  - package names
  - type names
  - field names
  - method names
  - local variable names (including parameters)
  - Labels (we won’t talk about this in this class)
  - enums
- This is a separate issue from overloading or overriding
Overriding

• Occurs when a subclass redefines a method defined in a superclass
  – Won’t talk about inheritance in much detail today
  – Just enough to talk about overriding two important methods
• Note that a green arrow appears in the right margin in Eclipse to indicate that a method overrides another method
• Eclipse examples: Animal.java, Dog.java
The Object class

• By definition, every Java class extends the class **Object**
• Object has multiple methods that are called under the hood in Java
  – Some examples are **toString**, **equals**, **hashCode**, etc.
  – You usually want to override those methods
More about toString()

• It is almost always a good idea to override public String toString() to return something “meaningful” about the object
  – When debugging, it helps to be able to print objects
  – When you print objects with System.out.print or System.out.println, they automatically call the objects toString() method
  – When you concatenate an object with a string, the object’s toString() method is automatically called

• You can explicitly call an object’s toString() method
  – Since toString() is used for printing, it’s something you want to be able to change easily (without breaking your test methods)
  – It’s usually better to write a separate method, similar to toString(), to use in your JUnit tests

• Eclipse example: Dog.java
The equals method

• Checking for equality in Java depends on the type
  – Primitives can always be tested for equality with ==
  – For objects, == tests whether the two are the same object (wink, wink: it compares the pointers)
  – Two strings "abc" and "abc" may or may not be == !
• Objects can be tested with the method public boolean equals(Object o)
  – The argument must be of type Object, which isn’t what you want, so you must cast it to the correct type
  – Unless overridden, this method just uses ==
  – It is overridden in the class String
  – It is not overridden for arrays; == tests if its operands are the same array
• Morals:
  – Never use == to test equality of Strings or arrays or other objects!
  – Use equals for Strings, java. util.Arrays.equals(a1, a2) for arrays (assuming the objects in the arrays have overridden equals)
  – If you test your own objects for equality, override equals
• Eclipse example: Dog.java
Why override a method?

- You override methods when the functionality of the method in the superclass is not sufficient
  - E.g., default `toString` and `equals` don’t work for most objects
- You still have access to the overridden method by using the keyword `super`
  - More on this when we get to inheritance
- We won’t be doing a lot of overriding in this class, other than for methods defined in `Object`
- But inheritance is one of Java’s most used features, so it is good to understand overriding as soon as possible