CSC 335 – Classes, Interfaces, and their Modeling III

Dr. Jonathan Misurda
jmisurda@cs.arizona.edu

When do we use inheritance?

- If the class you are writing needs properties of another class X, ask yourself the following question:

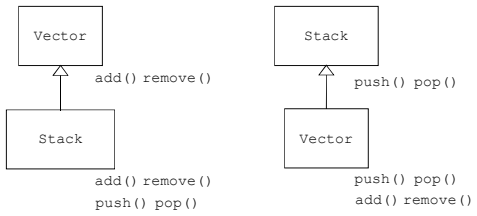
"Does my class need to be an X? Or does it just need to have an X?"

- If your class needs to have an ArrayList to keep elements, it should HaveA ArrayList—create an instance variable for it.
- If your class is a better version of an ArrayList, it IsA ArrayList—be sure to use the extends keyword.

IsA vs. HasA

- Point to remember: if you choose to extend another class (decide that your class IsA version of the extended class), it must do all of the things its parent does
- An example of IsA vs. HasA gone wrong is Java's Stack class
  - Stack extends Vector (a class very similar to ArrayList)
  - Which means that in addition to push() and pop(), Stack inherited the methods add(), remove(), and many other methods it should not have (which allows someone to, for example, add an element in the stack not at the top)
  - Instead, Vector should extend Stack?

Vector versus Stack



IsA vs. HasA Exercise

Should it be IsA or HasA?

- Given a Radio,
  - Write a Car
- Given a Computer,
  - Write a Laptop
- Given a Player,
  - Write a Team
- Given a File,
  - Write a Folder
- Given an AbstractSet,
  - Write a SortedSet
- Given an ArrayList,
  - Write a Queue
- Given a Doctor,
  - Write a Lawyer
- Given a Student,
  - Write a SectionLeader

Java's Inheritance Model

- Single Inheritance
  - Every class has one and only one parent class
  - The top of the inheritance tree is Object
- Single inheritance avoids ambiguity—if multiple parents had the same method implemented in different ways, which behavior would be correct in the child?
- Classes with different parents can still be grouped together conceptually using Java's interfaces (for example, Collections have forced common behavior like add(), remove(), iterator(), …)

Overriding Methods

- **Overriding** is the practice of writing a method in a subclass with the same partial signature as a method in the superclass it extends from.
- Since Object has a toString() method, every time we write one, we are actually overriding.

```
public class Person {
  public String toString() {
    return “Name: “ + getName() + “ Age: ” + getAge();
  }
}
```

java.lang.Object

protected Object	**clone**() Creates and returns a copy of this object.
boolean	**equals**(Object obj) Indicates whether some other object is "equal to" this one.
protected void	**finalize**() Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
Class<?>	**getClass**() Returns the runtime class of this Object.
int	**hashCode**() Returns a hash code value for the object.
void	**notify**() Wakes up a single thread that is waiting on this object's monitor.
void	**notifyAll**() Wakes up all threads that are waiting on this object's monitor.
String	**toString**() Returns a string representation of the object.
void	**wait**() Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.
void	**wait**(long timeout) Causes the current thread to wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.
void	**wait**(long timeout, int nanos) Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object, or some other thread interrupts the current thread, or a certain amount of real time has elapsed.

Using super

- Many times when overriding methods or constructors, you will want to reuse code from the same method in the superclass you extended. This can be done by using the keyword super.

```
public class Student extends Person {
   ...

   // Returns person summary plus their gpa
   public String toString() {
    return super.toString() + “ gpa: ”+ getGPA();
   }
   ...
}
```

@Override

```
public class Student extends Person {
        ...

        // Returns person summary plus their gpa
        @Override
        public String toString() {
                return super.toString() + “ gpa: ”+ getGPA();
        }
        ...
}
```

- @Override is not required but helps to prevent errors
  - If a method marked with @Override fails to correctly override a method in one of its superclasses, the compiler generates an error.

Annotations

- An **Annotation** is a form of *metadata*,
  - Metadata is data that describes data
  - Provides information about a program that is not part of the program itself.
- Annotations have no direct effect on the operation of the code they annotate.
- Annotations have a number of uses, among them:
  - **Information for the compiler** — Annotations can be used by the compiler to detect errors or suppress warnings.
  - **Compile-time and deployment-time processing** — Software tools can process annotation information to generate code, XML files, and so forth.
  - **Runtime processing** — Some annotations are available to be examined at runtime.

Java Interfaces

- A Java **interface** is just a list of methods without code
- Each class implementing an interface has a “CanDo” relationship with that interface: that class must provide an implementation of each of the interface’s methods.

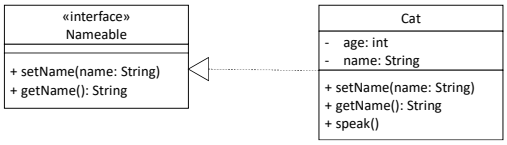
Declaring an interface

```
interface Nameable {
	public String getName();
	public void setName(String name);
}
```

- Use the keyword `interface` instead of `class`
- List of methods with no implementations
  - Java 8+ can provide `default` implementation

Implement an Interface

- Interface implementation is realized in UML as a dotted line with a hollow arrowhead that points from the class implementing the interface to the interface



Polymorphism and Casting

- Since a Student IsA Person, it should be able to do all the things that a Person does—and furthermore we should even be able to cast it to a Person

```
Student bob = new Student();
Person p = bob;
```

- Since this is an upcast (widening) the cast doesn't need to be explicit

Static vs Dynamic

- **Static** is a term that means "at compile time" or "before the program executes"
- **Dynamic** is a term that means "at run time" or "while the program is running"

- Java is a **statically typed** language that uses **dynamic dispatch**

Statically-Typed

- To determine if your code is legal, the compiler enforces the code to go through a check that types make sense:

```
String s = 3;		//Error
```

  - Left hand side of = is a String, right hand side is an int, that's not okay

```
Person p = bob;
p.getGPA();		//Error
```

  - Left hand side of . is a Person, Person doesn't have a method called `getGPA()`

Dynamic Dispatch

- What about our overridden methods?

```
Student bob = new Student("bob", 20, 3.1);
System.out.println(bob); //Student's toString()

Person bob = new Student("Bob", 20, 3.1);
System.out.println(bob); //Student's toString()
```

- Java uses the runtime type of the instance to determine the version of the method to call – **dynamic dispatch**

Checking IsA: Java's instanceof

- If we have a Person, and we want to know if it is a Student or not, we can perform a check using Java's `instanceof` keyword

```
public boolean isStudent(Person p) {
        return (p instanceof Student);
}
```

- Notice how the left side has a variable name, the right side has a class, and the entire expression is a boolean

Polymorphism and Casting, cont.

- So now what if we get a Person, and want to call methods which only Student has, such as getGPA()?
- We can perform a check, and then a cast:

```
public void printGPA(Person p) {
    if (p instanceof Student) {
            System.out.println(((Student)p).getGPA());
    }
    else {
            System.out.println("Sorry, no GPA.");
    }
}
```

- Casting from Person to Student is a downcast, and the compiler does not trust that we have done it correctly. Therefore we must explicitly cast with parentheses: (Student)

Casting and Polymorphism

- Since implementing classes have an IsA relationship with interfaces, we can cast an object of an implementing class to the interface type.
- The contract ensures the method will exist on all instances of the implementing class.

```
class Cat implements Nameable { … }

Nameable n = new Cat(); // Legal
n.getName(); // We can do anything that a Nameable promises
```

Abstract Classes

- **Abstract classes** are meant to be superclasses, or parent classes—that is, they are made to be extended (actually, must be extended to be used…)
- Abstract classes extend other classes, just like regular classes do, allowing you to improve upon or change features within the underlying superclass
- Abstract classes give you the best of two worlds
  - Code methods you want all subclasses to have.
    - Get common code written the way you want it!
  - Prototype methods you want all subclasses to implement.
    - Force others to write code that will change in varying implementations (similar to interfaces).

Syntax

- Methods can be declared to be abstract; these methods have no code body, and force extending classes to implement them

```
public abstract void myMethod();
```

- If you have any abstract methods in a class, the class must be declared to be abstract in the header

```
public abstract class SampleClass {
    //…
}
```

- Or you can declare the class abstract anyway…

Abstract Classes, cont.

- Abstract classes cannot be instantiated directly; that is to say, you cannot write the code:

```
new SampleClass();
```

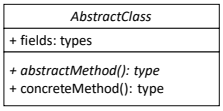
- and have it compile.
- This makes sense, since the abstract class doesn't have code for all of its methods.
- Classes can be declared to be abstract without having abstract methods, meaning the class cannot be instantiated.

Abstract Classes and Interfaces

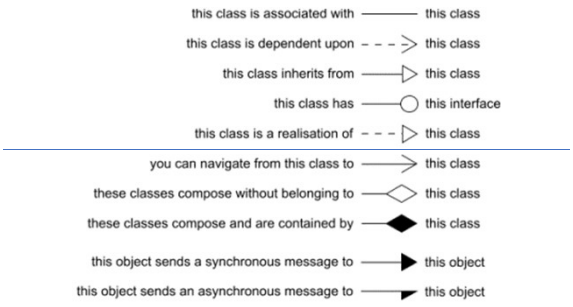
- The declaration for an abstract method is similar to the declaration for a method in an interface.
- Abstract classes that implement interfaces may choose to implement *all*, *some*, or *none* of the required methods
  - Those it does not implement are passed down with its abstract methods to extending subclasses.
- If you do not fully implement an interface, that you must declare the class to be `abstract`.

Abstract in UML

- Abstract classes or methods are indicated in UML diagrams by putting their names in *italics*
- Concrete methods within an abstract class are just shown in normal font
  - Concrete (real, tangible) is the antonym of abstract



Summary of UML Associations



Summary

- A class can have multiple methods and constructors with the same name; the partial signature is what is important.
- The IsA and HasA relationships tell us whether to use an instance variable or extend a class.
- Java uses single inheritance for extension, allows some methods to fall through and others to be overwritten and accesses parent methods using super.
- Java automatically upcasts; the user must explicitly downcast. IsA can be checked using instanceof.
- Interfaces ensure common methods to classes and give the impression of multiple inheritance.
- Abstract classes give you the best of both worlds.