UML Class Diagrams

**UML:** language for specifying and visualizing OOP software systems

**UML class diagram:**
- specifies class name, instance variables, methods, ...

---

Classes with Stuff in Common

- Lots of duplication across the two classes
- More importantly, many clients should be able to work with both: don’t want to duplicate their code.
- How do we eliminate the duplication?

---

Abstracting Common Stuff

**Inheritance hierarchy:**
- subclasses (UG, PG) inherit from superclass (Student)
- arrow with open head indicates generalization in UML class diagram
Subclasses and superclasses

- Subclass (e.g. UG) inherits the members of superclass (e.g. Student)
- Subclass is a specialization of superclass — superclass is generalization of subclass

*[details to be further specified...]*

X IS-A Y?

The IS-A test

- Is ClassX a subclass of ClassY?
- Test: can we say that ClassX IS-A (‘is a kind of’) ClassY?
- Does an instance of ClassX have all the properties (and maybe more) that an instance of ClassY has?

**IS-A Candidates**

1. Kitchen subclass-of Room
2. Room subclass-of House
3. Violinist subclass-of Musician
4. Sink subclass-of Kitchen
5. Musician subclass-of Person
6. Lady Gaga subclass-of Singer
7. Student subclass-of Musician

Inheritance

- Subclass inherits all the public and protected members (instance variables and methods) of the superclass.
  - protected members and methods are not accessible from outside the inheritance structure but available within, i.e. to all subclasses
- In Java: subclass extends superclass.
- A subclass can add new members of its own
- By default, methods that are inherited from superclass have same implementation in subclass
- except if...
  - subclass overrides the inherited methods.

Doctor Example, 1

- `treatPatient()`: void
- `makeIncisions()`: void
- `giveAdvice()`: void

For handy guide to UML, see http://www.loufranco.com/blog/assets/cheatsheet.pdf
Doctor Example, 2

```java
public class Doctor {
    public void treatPatient() {
        // perform a checkup
    }
}
```

Doctor Example, 3

```java
public class FamilyDoctor extends Doctor {
    public void giveAdvice() {
        // tells you to wrap up warmly
    }
}
```

NB We put this class into a new file FamilyDoctor.java

Doctor Example, 4

```java
public class Surgeon extends Doctor {
    public void treatPatient() {
        // perform surgery
        // overrides inherited method
        // Can call Doctor's version:
        super.treatPatient();
    }
    public void makeIncisions() {
        // use a scalpel
        // a new method
    }
}
```

NB We put this class into a new file Surgeon.java

Method Overriding

- Method $m$ in subclass $B$ overrides method $m'$ in superclass $A$ if $m$ has exactly the same signature (i.e. name and parameters) as $m'$. (Return type? Later...)
- Normally, $m$ replaces the implementation of $m'$.

Doctor

```java
Doctor d = new Doctor();
d.treatPatient(); // Use implementation in Doctor class
```

Surgeon

```java
Surgeon s = new Surgeon();
s.treatPatient(); // Use implementation in Surgeon class
```
**Type Hierarchy View in Eclipse**

- **Hierarchy Tree Pane**
- **Members Pane**

**Doctor Subclasses**

Right-click on a class name and select **Open Type Hierarchy**

**FamilyDoctor Members**

Inherited and non-inherited members

**The Design Process**

1. Look for objects that have common attributes and behaviours.
2. Design a class that represents the common state and behaviour.
3. Decide if a subclass needs method implementations that are specific to that particular subclass type.
4. Carry out further abstraction by looking for groups of subclasses that might have common behaviours.
Encapsulation and Inheritance

**Student**

```java
public class Student {
    private final String firstName;
    private final String lastName;
    private final String matric;

    public Student(String fn, String ln, String m) { ... }

    public String getFirstName() { ... }
    public String getLastName() { ... }
    public String getMatric() { ... }
}
```

**UG**

```java
public class UG extends Student {
    private String tutGroup = "";

    public void setTutGroup(String s) {
        tutGroup = s;
    }

    public String getTutGroup() {
        return tutGroup;
    }

    public String toString() {
        return "UG [firstName=" + firstName + ",
                 lastName=" + lastName +
                 ", matric=" + matric +
                 ", tutGroup=" + tutGroup + "]";
    }
}
```

Won't work!

```java
public class UG extends Student {
    private String tutGroup = "";

    public String toString() {
        return "UG [firstName=" + getFirstName() + ", 
                 lastName=" + getLastName() +
                 ", matric=" + getMatric() +
                 ", tutGroup=" + getTutGroup() + "]";
    }
}
```
Encapsulation and Inheritance

▶ **private** instance variables (fields) cannot be directly accessed by subclass.
▶ Can only be accessed via setter and getter methods (which are inherited from superclass).

The Object Superclass

Object is the superclass of every class in Java!
▶ If a class doesn’t explicitly extend some superclass, then it **implicitly** extends Object.
▶ That is, we don’t need to add `extends Object`.

Doctor Example in Eclipse

Where does Doctor belong in the class hierarchy?

The class `Object`

defines the `treatPatient` method.

Doctor's Superclass

```java
public class Doctor {
    void treatPatient() {
        ...
    }
}
```
Some Methods of Object

Object defines methods that are available to every class. E.g.,

- `equals(Object o)` — test whether two objects are equal.
- `hashCode()` — numerical ID; equal objects must have equal hash codes.
- `toString()` — returns a textual representation of an object; automatically invoked by methods like `System.out.println()`.
- Since every class inherits `toString()` from `Object`, you have already been overriding this method!

Flat vs. Nested Hierarchies

Flat Animal Hierarchy

Animals Example, 1

Our base class: Animal

```java
public class Animal {
    public void sleep() {
        System.out.println("Sleeping: Zzzzz");
    }
    public void makeNoise() {
        System.out.println("Noises...");
    }
    public void roam() {
        System.out.println("Roamin' on the plain.");
    }
}
```
Animals Example, 2

1. Lion subclass-of Animal
2. Override the makeNoise() method.

Lion

```java
public class Lion extends Animal {
    public void makeNoise() {
        System.out.println("Roaring: Rrrrrr!");
    }
}
```

Animals Example, 3

1. Cat subclass-of Animal
2. Override the makeNoise() method.

Cat

```java
public class Cat extends Animal {
    public void makeNoise() {
        System.out.println("Miaowing: Miaooo!");
    }
}
```

Animals Example, 4

1. Wolf subclass-of Animal
2. Override the makeNoise() method.

Wolf

```java
public class Wolf extends Animal {
    public void makeNoise() {
        System.out.println("Howling: Ouoooo!");
    }
}
```

Animals Example, 5

1. Dog subclass-of Animal
2. Override the makeNoise() method.

Dog

```java
public class Dog extends Animal {
    public void makeNoise() {
        System.out.println("Barking: Woof Woof!");
    }
}
```
public class AnimalLauncher {
    public static void main(String[] args) {
        System.out.println("\nWolf\n=====");
        Wolf wolfie = new Wolf();
        wolfie.makeNoise(); // from Wolf
        wolfie.roam(); // from Animal
        wolfie.sleep(); // from Animal
        System.out.println("\nLion\n=====");
        Lion leo = new Lion();
        leo.makeNoise(); // from Lion
        leo.roam(); // from Animal
        leo.sleep(); // from Animal
    }
}

Lions and cats can be grouped together into Felines, with common roam() behaviours.
Dogs and wolves can be grouped together into Canines, with common roam() behaviours.

Output

Wolf
=====
Howling: Ouooooo!
Roamin’ on the plain.
Sleeping: Zzzzz

Lion
=====
Roaring: Rrrrrr!
Roamin’ on the plain.
Sleeping: Zzzzz

Nested Animal Hierarchy
Animals Example, 1

Same as before.

Animal

```java
public class Animal {
    public void sleep() {
        System.out.println("Sleeping: Zzzzz");
    }
    public void makeNoise() {
        System.out.println("Noises...");
    }
    public void roam() {
        System.out.println("Roamin' on the plain.");
    }
}
```

Animals Example, 2

The new class Feline

Feline

```java
public class Feline extends Animal {
    public void roam() {
        // Override roam()
        System.out.println("Roaming: I'm roaming alone.");
    }
}
```

Animals Example, 3

The new class Canine

Canine

```java
public class Canine extends Animal {
    public void roam() {
        // Override roam()
        System.out.println("Roaming: I'm with my pack.");
    }
}
```

Animals Example, 4

1. Lion subclass-of Feline
2. Override the makeNoise() method.

Lion

```java
public class Lion extends Feline {
    public void makeNoise() {
        System.out.println("Roaring: Rrrrrr!");
    }
}
```

▶ Similarly for Cat.
Animals Example, 5

1. Wolf subclass-of Canine
2. Override the makeNoise() method.

Wolf

```java
public class Wolf extends Canine {
    public void makeNoise() {
        System.out.println("Howling: Ouooooo!");
    }
}
```

> Similarly for Dog.

Animals Example, 6

The Launcher

```java
public class AnimalLauncher {
    public static void main(String[] args) {
        System.out.println("\nWolf
=====");
        Wolf wolfie = new Wolf();
        wolfie.makeNoise(); // from Wolf
        wolfie.roam(); // from Canine
        wolfie.sleep(); // from Animal
        System.out.println("\nLion
=====");
        Lion leo = new Lion();
        leo.makeNoise(); // from Lion
        leo.roam(); // from Feline
        leo.sleep(); // from Animal
    }
}
```

Animals Example, 7

Output

Wolf
=====
Howling: Ouooooo!
Roaming: I’m with my pack.
Sleeping: Zzzzz

Lion
=====
Roaring: Rrrrrr!
Roaming: I’m roaming alone.
Sleeping: Zzzzz
Polymorphism

Declaring and Initializing a Reference Variable

Polymorphism

Typing and Polymorphism

▶ polymorphism ( = ‘many shapes’): the same piece of code can be assigned multiple types.
▶ A class defines a type, namely the signatures of its methods.
▶ S is a subtype of T, written S <: T, if a value of type S can be used in any context where a value of type T is expected.
▶ The relation <: is reflexive: T <: T
▶ The relation <: is transitive: if S <: T and T <: U, then S <: U.
▶ NB: We say T is a supertype of S if S is a subtype of T.
▶ Inclusion polymorphism: objects of different types S₁, S₂, . . . may be treated uniformly as instances of a common supertype T.

Declaring and Initializing a Reference Variable

Wolf wolfie  =  new Wolf();
create a Wolf object

Declaring and Initializing a Reference Variable

declare a reference variable

Wolf wolfie  =  new Wolf();
Declaring and Initializing a Reference Variable

Wolf wolfie = new Wolf();

Polymorphic ArrayList

public class AnimalLauncher2 {
    public static void main(String[] args) {
        Wolf wolfie = new Wolf();
        Lion leo = new Lion();
        Cat felix = new Cat();
        Dog rover = new Dog();
        ArrayList<Animal> animals = new ArrayList<Animal>();
        animals.add(wolfie);
        animals.add(leo);
        animals.add(felix);
        animals.add(rover);
        for (Animal a : animals) {
            a.makeNoise();
        }
    }
}

Polymorphic Arrays

ArrayList<Animal> is polymorphic.

- animals.add(wolfie)
  add an object of type Wolf. OK since Wolf <: Animal.
- for (Animal a : animals)
  for each object a of type T such that T <: Animal ...
  a.makeNoise()
  if a is of type T, use T's makeNoise() method.
**Overriding and Overloading**

Method Overriding, 1

If a class C **overrides** a method m of superclass D, then:

- Parameter lists must be same and return type must be compatible:
  1. signature of m in C must be same as signature of m in D; i.e. same name, same parameter list, and
  2. return type S of m in C must such that S <: T, where T is return type of m in D.

- m must be at least as accessible in C as m is in D

Method Overriding, 2

```java
method in Animal
public void makeNoise() {
    ...
}
```

Wrong: method in Wolf

```java
public void makeNoise(int volume) {
    ...
}
```

Wrong: method in Wolf

```java
private void makeNoise() {
    ...
}
```

Method Overloading, 1

Overloading: two methods with **same** name but **different** parameter lists.

**Overloaded makeNoise**

```java
public void makeNoise() {
    ...
}
public void makeNoise(int volume) {
    ...
}
```

**Overloaded println**

```java
System.out.println(3); // int
System.out.println(3.0); // double
System.out.println((float) 3.0); // cast to float
System.out.println("3.0"); // String
```
Method Overloading, 2

1. Return types can be different.
2. You can’t just change the return type — gets treated as an invalid override.
3. Access levels can be varied up or down.

Incorrect override of makeNoise

```java
public String makeNoise() {
    String howl = "Ouoooo!";
    return howl;
}
```

Exception in thread "main" java.lang.Error:
Unresolved compilation problem:
The return type is incompatible with Animal.makeNoise()

Abstract Classes

Concrete vs. Abstract

Concrete
- Examples: Cat, Wolf
- Specific enough to be instantiated.

Abstract
- Examples: Animal, Feline
- Not intended to have instances.
- Only useful if extended.
- Any ‘instances’ will have to be instances of a subclass of the abstract class.

Animal Objects?

Creating new objects

Wolf wolfie = new Wolf();
Animal leo = new Lion();
Animal weird = new Animal();

- Animal class is meant to contain information that all animals have in common.
- But this is not enough to define any one specific animal.
The Abstract Animal, 1

Animal

```java
public abstract class Animal {
    public void sleep() {
        System.out.println("Sleeping: Zzzzz");
    }
    public void makeNoise() {
        System.out.println("Noises...");
    }
    public void roam() {
        System.out.println("Roamin’ on the plain.");
    }
}
```

Just put the keyword `abstract` before the class declaration.

The Abstract Animal, 2

Animal

```java
public abstract class Animal {
    public void sleep() {
        System.out.println("Sleeping: Zzzzz");
    }
    public abstract void makeNoise();
    public abstract void roam();
}
```

Now has abstract methods!

The Abstract Animal, 3

Animal

```java
public abstract class Animal {
    public void sleep() {
        System.out.println("Sleeping: Zzzzz");
    }
    public abstract void makeNoise();
    public abstract void roam();
}
```

Now has abstract methods!

- `roam()` and `makeNoise()` are abstract methods:
  - no body;
  - must be implemented in any concrete subclass (implemented ~ overridden);
  - don’t have to be implemented by an abstract subclass;
  - can only be declared in an abstract class;
- `sleep()` is not abstract, so can be straightforwardly inherited.
Abstract Classes in Animal Hierarchy

Using Abstract Classes

- Use an abstract class when you have several similar classes that:
  - have a lot in common — the implemented parts of the abstract class
  - have some differences — the abstract methods.

Interfaces

- We will not cover interfaces in detail in this course
- Used a lot in the Java API
- they can be seen as abstract classes where every method is abstract

Constructor Chaining
Constructor Chaining, 1

▶ All constructors in object’s inheritance tree run when a new instance is created.
▶ FamilyDoctor extends Doctor

Constructor Chaining, 2

▶ Make a new FamilyDoctor().
▶ Call the no-argument superclass constructor, i.e. Doctor().
▶ Call the no-argument superclass constructor, i.e. Object().
▶ Constructor of the immediate superclass invoked with super().
▶ If you don’t explicitly call this, the compiler will, but only for no-argument constructors.

Constructor Chaining, 3

Student

... public Student(String fn, String ln, String m) {
    firstName = fn;
    lastName = ln;
    matric = m;
}
...

UG extends Student

... private String tutGroup
...

public UG(String fn, String ln, String m, String tutGroup) {
    super(fn, ln, m); // call the superclass constructor
    this.tutGroup = tutGroup;
}
...

Summary

Inheriting from a superclass:
▶ the subclass gets all the public members (instance variables and methods) of the superclass;
  ▶ public class Foo extends Baz
▶ the subclass may add members, and also override methods.
▶ So subclass extends (adds to) the behaviour of its superclass.
▶ Inheritance corresponds roughly to taxonomic relations for everyday concepts.
▶ In Java, you can only inherit from one superclass.

Problems with using inheritance:
▶ Easy to get muddled with inheritance hierarchies.
▶ Subclass is tightly coupled with superclass.
▶ Changes in superclass can break subclass — fragile base class problem.
Reading

Java Tutorial
pp193-217, i.e. Chapter 6 *Interfaces and Inheritance*, from *Inheritance* to the end of the chapter.