Inf1B
Inheritance B

Perdita Stevens
adapting earlier versions by Ewan Klein, Volker Seeker, et al.

School of Informatics
Abstracting Common Stuff

Inheritance hierarchy:
Subclass (UG, PG) inherit from superclass (Student) inherits from superclass (Object)
Flat vs. Nested Hierarchies
Flat Animal Hierarchy

- Animal
  - sleep()
  - makeNoise()
  - roam()

- Lion
  - makeNoise()

- Cat
  - makeNoise()

- Wolf
  - makeNoise()

- Dog
  - makeNoise()
Our base class: Animal

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.");
    }
}
```
1. Lion subclass-of Animal
2. Override the makeNoise() method.

Lion

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

Animals Example, 2
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!");
    }
}
```
1. Wolf subclass-of Animal
2. Override the makeNoise() method.

```java
public class Wolf extends Animal {
    public void makeNoise() {
        System.out.println("Howling: Ouoooooo!");
    }
}
```
1. Dog subclass of Animal
2. Override the makeNoise() method.

```java
public class Dog extends Animal {
    public void makeNoise() {
        System.out.println("Barking: Woof Woof!".LayoutInflater);
    }
}
```
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
    }
}

Output

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

Lion
=====
Roaring: Rrrrrrr!
Roamin’ on the plain.
Sleeping: Zzzzz
Nested Animal Hierarchy

- 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.
Nested Animal Hierarchy

- Animal
  - sleep()
  - makeNoise()
  - roam()

- Feline
  - roam()

- Canine
  - roam()

- Lion
  - makeNoise()

- Cat
  - makeNoise()

- Wolf
  - makeNoise()

- Dog
  - makeNoise()
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.");
    }
}
```
The new class Feline

Feline

```java
public class Feline extends Animal {
    public void roam() {
        // Override roam()
        System.out.println("Roaming: I’m roaming alone.");
    }
}
```
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.");
    }
}
```
1. Lion subclass-of Feline
2. Override the makeNoise() method.

Lion

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

▶ Similarly for Cat.
1. Wolf subclass of Canine
2. Override the `makeNoise()` method.

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

▶ Similarly for Dog.
Which method gets called?

1. Wolf wolfie = new Wolf();
2. wolfie.makeNoise();
3. wolfie.roam();
4. wolfie.sleep();
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 Canine
        wolfie.sleep(); // from Animal

        System.out.println("\nLion\n=====");
        Lion leo = new Lion();
        leo.makeNoise(); // from Lion
        leo.roam(); // from Feline
        leo.sleep(); // from Animal
    }
}
<table>
<thead>
<tr>
<th>Animal</th>
<th>Howling</th>
<th>Roaming</th>
<th>Sleeping</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wolf</td>
<td>Ouooooo!</td>
<td>I’m with my pack.</td>
<td>Zzzzzz</td>
</tr>
<tr>
<td>Lion</td>
<td>Rrrrrr!</td>
<td>I’m roaming alone.</td>
<td>Zzzzzz</td>
</tr>
</tbody>
</table>
Polymorphism
Typing and Polymorphism

- **polymorphism** (＝ ‘many shapes’): the same piece of code can be assigned multiple types.
- A class defines a type, roughly 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 $T$. 
private static void goToBed(Animal tiredAnimal) {
    tiredAnimal.sleep();
}

public static void main(String[] args) {
    Animal myAnimal = new Animal();
    goToBed(myAnimal);
}
Polymorphism in Java

“the same piece of code can be assigned multiple types”

Especially, an object passed into a method could come from any one of many classes.

In Java, those classes will be related by inheritance (“OO polymorphism”).

E.g. any method that was written to expect an object of class Animal will still work if it is given an object of class Wolf.

(Work? Compile, anyway. The “is-a” check between classes helps to make sure you get sensible results...)
private static void goToBed(Animal tiredAnimal) {
    tiredAnimal.sleep();
}

public static void main(String[] args) {
    Animal myAnimal = new Wolf();
    goToBed(myAnimal);
}

The subclass can do at least everything the superclass can do.
(maybe a bit different though)

Formal Notation: Wolf <: Animal (Wolf is a subtype of Animal)
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();
            goToBed(a);
        }
    }
}
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.

- `goToBed(a)`
  You get at least an Animal, so you can call every method on it an Animal has
Student Hierarchy

Subclass (UG, PG) inherit from superclass (Student) inherits from superclass (Object)
Casting Object Types

Does this work?

```java
class Student {
    public void tutor();
}

class PGStudent {
    // Constructor...
}

private static void giveTutorial(Student support) {
    support.tutor();
}

public static void main(String[] args) {
    Student support = new PGStudent();
giveTutorial(support);
}
```

Compiler Error! Student does not have a tutor() method
Casting Object Types

Does this work?

```java
private static void giveTutorial(Student support) {
    support.tutor();
}

public static void main(String[] args) {
    Student support = new PGStudent();
giveTutorial(support);
}
```

**Compiler Error!** Student does not have a tutor() method
Casting Object Types

Does this work?

```java
private static void giveTutorial(Student support) {
    PGStudent pgsupport = (PGStudent) support;
    pgsupport.tutor();
}

public static void main(String[] args) {
    Student support = new PGStudent();
    giveTutorial(support);
}
```
Casting Object Types

Does this work?

```java
private static void giveTutorial(Student support) {
    PGStudent pgsupport = (PGStudent) support;
    pgsupport.tutor();
}
```

```java
public static void main(String[] args) {
    Student support = new PGStudent();
    giveTutorial(support);
}
```

Yes, I do actually get a PGStudent as argument.

But what if not??
Casting Object Types

Casting Object Types Should be Protected

```java
private static void giveTutorial(Student support) {
    if (support instanceof PGStudent) {
        PGStudent pgsupport = (PGStudent) support;
        pgsupport.tutor();
    }
}
```

```java
public static void main(String[] args) {
    Student support = new UGStudent();
    giveTutorial(support);
}
```

This works and nothing will be printed. But this code smells really bad!
Overriding vs. Overloading
Method Overriding

If a class $C$ overrides a method $m$ of superclass $D$, ...

For Example

```java
public class Animal {
    public Animal findPlaymate() {
        ...
    }
}

public class Wolf extends Animal {
    ???
}
```
Method Overriding

If a class \( C \) overrides a method \( m \) of superclass \( D \), then:

- Parameter lists must be the same.

```java
public class Animal {
    public Animal findPlaymate() {
        ...
    }
}

public class Wolf extends Animal {
    public Animal findPlaymate(int number) { // This is not overriding
        ...
    }
}

public class Wolf extends Animal {
    public Animal findPlaymate() { // This is overriding
        ...
    }
}
```
Method Overriding

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

▶ Parameter lists must be the same.
▶ The return type must be the same or a subclass of the original.

```java
public class Animal {
    public Animal findPlaymate() {
        //
    }
}

public class Wolf extends Animal {
    public Student findPlaymate() { // This is not overriding
        //
    }
}

public class Wolf extends Animal {
    public Wolf findPlaymate() { // This is overriding
        //
    }
}
```
Method Overriding

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

▶ Parameter lists must be the same.
▶ The return type must be the same or a subclass of the original.
▶ The overridden method must be at least as accessible as the original.

```java
public class Animal {
    protected Animal findPlaymate() {
        // ...
    }
}

class Wolf extends Animal {
    private Animal findPlaymate() { // This is not overriding
        // ...
    }
}

class Wolf extends Animal {
    public Wolf findPlaymate() { // This is overriding
        // ...
    }
}
Method Overriding

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
Most versions I showed that did not override, do in fact compile.
Most versions I showed that did not override, do in fact compile.

But they **overload** the method rather than **override** it.
Method Overloading

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

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 = "Ouooooo!";
    return howl;
}
```

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

https://www.theodysseyonline.com/your-brain-is-muscle-exercise-it
public class Vehicle {
    public void drive() {
        System.out.println("drivedrive");
    }
}
public class Car extends Vehicle {
    public void drive() {
        System.out.println("rollroll");
    }
}
public class Bike extends Vehicle {
    public void drive() {
        System.out.println("pedalpedal");
    }
}
public class Main {
    public static void main(String[] args) {
        Vehicle c = new Car();
        c.drive();
        Vehicle b = new Bike();
        b.drive();
    }
}
What does it print?

```java
public class Vehicle {
    public void drive() {
        System.out.println("drivedrive");
    }
}
public class Car extends Vehicle {
    public void drive() {
        System.out.println("rollroll");
    }
}
public class Bike extends Vehicle {
    public void drive() {
        System.out.println("pedalpedal");
    }
}
public class Main {
    public static void main(String[] args) {
        Vehicle c = new Car();
        c.drive();
        Vehicle b = new Bike();
        b.drive();
    }
}
```

Prints **rollroll** and **pedalpedal** because polymorphic references c and b contain instances of Car and Bike.
public class Addition{
    public int add(int a, int b){
        int sum = a+b;
        return sum;
    }
    public int add(int a, int b, int c){
        int sum = a+b+c;
        return sum;
    }
    public double add(double a, double b, double c){
        double sum = a+b+c;
        return sum;
    }
}
public class Main {
    public static void main (String[] args) {
        Addition op = new Addition();
        System.out.println(op.add(1,2));
        System.out.println(op.add(1,2,3));
        System.out.println(op.add(1.0,2.0,3.0));
    }
}
What does it print?

```java
public class Addition{
    public int add(int a, int b){
        int sum = a+b;
        return sum;
    }
    public int add(int a, int b, int c){
        int sum = a+b+c;
        return sum;
    }
    public double add(double a, double b, double c){
        double sum = a+b+c;
        return sum;
    }
}

public class Main {
    public static void main (String[] args) {
        Addition op = new Addition();
        System.out.println(op.add(1,2));
        System.out.println(op.add(1,2,3));
        System.out.println(op.add(1.0,2.0,3.0));
    }
}
```

Prints **3, 6 and 6.00000** because `add` is overloaded once by using more parameters and once by using different parameter types.
public class Birthday {
    public void greet(String name, int age){
        System.out.println("Happy\n" + age + ".\nbirthday,\n" + name + "!");
    }
    public void greet(int age, String name){
        System.out.println("All\nthe\nbest\nfor\nyour\n" + 
age + ".\nbirthday,\n" + name + "!");
    }
}

public class Main {
    public static void main (String[] args) {
        Birthday b = new Birthday();
        b.greet("Jack", 5);
        b.greet(7, "Jill");
    }
}
public class Birthday {
    public void greet(String name, int age){
        System.out.println("Happy \(\) + age + ". \(\) birthday, \(\) + name + \"!\")
    }
    public void greet(int age, String name){
        System.out.println("All \(\) the \(\) best \(\) for \(\) your \(\) +
                        age + ". \(\) birthday, \(\) + name + \"!\")
    }
}

public class Main {
    public static void main (String[] args) {
        Birthday b = new Birthday();
        b.greet("Jack", 5);
        b.greet(7, "Jill");
    }
}

Prints **Happy 5. birthday, Jack!** and **All the best for your 7. birthday, Jill!** because `greet` is overloaded by swapping parameter types around. **This smells.**
What does it print?

```java
public class Addition{
    public int add(int a, int b){
        int sum = a+b;
        return sum;
    }
    public double add(int a, int b){
        int sum = a+b;
        return sum;
    }
}

public class Main {
    public static void main (String[] args) {
        Addition ob = new Addition();
        System.out.println(ob.add(1,2));
        System.out.println(ob.add(1,2));
    }
}
```

Does not compile because changing only the return type when overloading is not enough.
What does it print?

```java
public class Addition {
    public int add(int a, int b) {
        int sum = a + b;
        return sum;
    }
    public double add(int a, int b) {
        int sum = a + b;
        return sum;
    }
}

public class Main {
    public static void main(String[] args) {
        Addition ob = new Addition();
        System.out.println(ob.add(1, 2));
        System.out.println(ob.add(1, 2));
    }
}
```

Does not compile because changing only the return type when overloading is not enough.
Inheritance structures can be long and nested.

But that’s not a good thing – OO beginners tend to use inheritance too much!

Polymorphism is when code can be given several types. E.g.
  - you can collect objects of various subtypes in a collection that is defined at the common supertype
  - you can use the same client code on objects of different subtypes of the type the code is written for

Overriding needs to follow three rules (parameter list, return type, access).

Otherwise it is likely overloading.

Overloading across a class hierarchy can be confusing: usually, best avoided.
Objects First
Chapter 10.7 SubTyping
Chapter 11 More About Inheritance